Multi-Partner Project: A Deep Learning Platform Targeting Embedded Hardware for Edge-AI Applications (NEUROKIT2E)

Rajendra Bishnoi[1] Mohammad Amin Yaldagard[1] Said Hamdioui[1] Kanishkan Vadivel[2]
Manolis Sifalakis[2] Nicolas Daniel Rodriguez[3] Pedro Julian[4] Lothar Ratschbacher[3]
Maen Mallah[5] Yogesh Ramesh Patil[5] Rashid Ali[5] Fabian Chersi[6]
[1] Delft University of Technology, The Netherlands. Email: {R.K.Bishnoi, M.A.Yaldagard, S.Hamdioui}@tudelft.nl
[2] IMEC, The Netherlands. Email: {kanishkan.vadivel, manolis.sifalakis}@imec.nl
[3] Silicon Austria Labs, Austria. Email: {nicolas-daniel.rodriguez, lothar.ratschbacher}@silicon-austria.com
[4] Universidad Nacional del Sur IIIE-DIEC, Argentina. Email: pjulian@uns.edu.ar
[5] Fraunhofer IIS, Germany. Email: {maen.mallah, yogesh.ramesh.patil, rashid.ali}@iis.fraunhofer.de
[6] CEA-LIST, Paris, France. Email: fabian.chersi@cea.fr

***Abstract*—The goal of the NEUROKIT2E project is to create an open-source Deep Learning framework for edge and embedded AI built around an established European value chain. This framework, called AIDGE, supports a wide range of application areas that operate independently and serve a global user community. It provides easy and fast full-stack solutions from Neural Network design and optimization to AI application development all the way down to hardware implementations while enabling code generation for application-specific targets. This platform provides flexibility for academic users in the AI domain to explore and innovate while allowing them the possibility to prototype systems, ensuring their work aligns well with industrial needs. This paper presents the results and achievements of the first part of this three-year project, along with its roadmap and expected outcomes.**

I. Introduction

In today's era of Artificial Intelligence (AI), we can say that AI is everywhere. It is not only transforming industries but also enhancing daily tasks and shaping the future of technology. As new, larger, and more advanced AI models emerge and bigger datasets become available, the demand for new hardware and more computing power is also increasing. In addition to larger and more capable processing architectures (e.g. GPUs) that accelerate DNNs' training and execution, this is leading to the development of optimized hardware that is capable of reducing the power consumption during inference by employing clever computation optimization and data transfer techniques. In this context, embedded systems are specialized computers that are integrated into larger systems, offering control and automation capabilities, and when combined with AI, they enable smarter automation, predictive maintenance, and real-time decision-making [1], [2]. AI-embedded edge systems hold significant importance as they facilitate rapid, low-latency data processing directly at the network's edge, which is essential for real-time applications such as self-driving cars and intelligent manufacturing [3]. Additionally, by performing computations locally, it strengthens data privacy and security by minimizing the transmission of sensitive information to external servers [4], [5]. The NEUROKIT2E project develops a set of tools covering the whole chain, starting from designing AI applications and model development to implementing them in hardware, allowing for the automatic generation of standalone code for a specific constrained hardware platform.

In the NEUROKIT2E project, we address the challenges associated with each abstraction layer of the project development cycle. For instance, a key challenge is to understand the application requirements and align them with the constraints and capabilities of the execution platform. Next, for Deep Neural Network (DNN) models, one of the main challenges is to find the right balance between high accuracy and maintaining computational efficiency. This is because performance improvements usually demand more resources that can consume high energy as well as longer training and inference times. Moreover, the challenge associated with neural network mapping on hardware is to perform an effective conversion of the complex structure of DNN models into hardware configurations that improve performance while using less energy and resources. The main challenges in hardware modeling are to effectively simulate the behavior of hardware while ensuring the model reflects its key design metrics such as latency, energy consumption, area, leakage, etc. For such hardware implementations, a challenge lies in translating high-level designs into physical hardware while providing generic building blocks, methodologies, and tools that can be reused across different architecture classes, such as Neuromorphic, Tensor-based, and programmable systems. In this project, we address the aforementioned challenges and develop an open-source framework for embedded edge AI systems that can support various applications.

NEUROKIT2E aims to develop **AIDGE**, an open-source AI code generation framework optimized for both common (CPU, GPU) and embedded targets. Here, applications span Electric Vehicles, Digital Society (Earth Observation and Biometrics), Mobility (Rail Transport), Health & Well-being, and Smart Buildings. The development workflow for our project,

978-3-9826741-0-0/DATE25/© 2025 EDAA

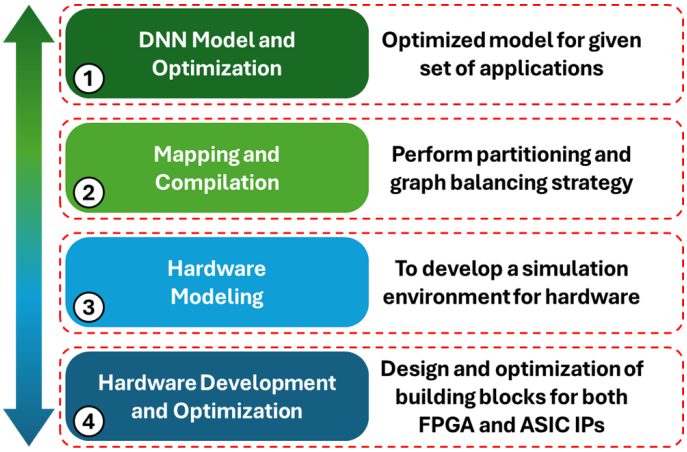


Fig. 1: Overview of the development flow for the NEUROKIT2E project.

illustrated in Fig. 1, includes DNN model development and optimization, mapping and compilation, hardware modeling, and hardware development and optimization. The four main objectives of our project are listed below:

- Develop a Unified and Versatile Software Development Kit (SDK) and Hardware Modeling Framework.
- Develop Innovative Hardware-Aware Machine Learning Optimization Techniques.
- Address Industrial Challenges for Usability of AI.
- Enable Full-Scale HLS IP Generation.

In this paper, we present the full-stack solutions for AI-embedded edge systems that we are developing as part of the NEUROKIT2E project. First, we provide an overview of the AIDGE framework in Section II. In Section III, we present the details of DNN modeling and optimizations, followed by descriptions of neural network mapping to hardware configurations in Section IV. We address hardware modeling in Section V, describing key aspects of the hardware simulation environment. In Section VI, we cover hardware generation and optimization, where we describe the hardware design implementations. Section VII concludes the paper.

II. AIDGE Framework

We introduce AIDGE, an end-to-end framework for training, optimizing, and compiling deep neural networks (DNNs) tailored for low-power edge devices. Designed for efficient compilation, flexibility, and portability, AIDGE combines graph analysis with structured programming methods. The platform encompasses database construction, data pre-processing, network design, optimization, quantization, testing, and hardware export functionalities (see Fig. 2). AIDGE facilitates DNN design by allowing quick prototyping of multiple network topologies and automatic performance comparison.

AIDGE is built on a modular principle, featuring a "Core Module" that can be extended with plugins to enhance functionality. Developed in C++ (14) with Python bindings (> 3.7), the Core Module enables:

- Creation and modification of a computational graph representing a DNN.
- Graph querying/matching for specific operator sequences.
- Instantiation of operators and data structures like Tensors.
- Creation of schedulers for graph execution.
- Access to optimization functionalities, such as operator fusion.

AIDGE distinguishes between description and implementation, using abstract operator definitions that adapt to specific hardware and libraries. The framework's "Backend" defines the hardware target and implementation library.

AIDGE aims to produce interpretable and auditable outputs by generating source code and resource files packaged as the Software Toolkit. The export strategy in the AIDGE framework includes two main phases: Export Mapping and Export Implementation. In the Export Mapping phase, optimization techniques are applied to adapt the computational graph for the target hardware. The Export Implementation phase then generates source code based on this adapted graph. Additionally, an optional memory file may indicate static resource allocation for the network. Once all files are exported, the project can be compiled using "make" if targeting C++.

III. Modeling and Optimization

A. HATAI: HW-Aware Training And Inference Tool

The **HATAI** tool [6] is originally designed to support mixed-signal accelerators with analog compute cores where the computations are non-ideal due to process and mismatch variation in analog processing. In addition to Quantization-Aware Training [7] support, the tool offers Fault-Aware Training (**FAT**) [8] where the faults/errors are modeled and introduced during training to train a robust NN against said faults/errors. The **HATAI** tool supports injecting any fault/error into any parameter (weight, bias, etc.) and activation in any layer of the NN. The fault injection can be used during training (to produce a robust NN against said faults) as well as inference (to test for robustness and measure accuracy without the need to deploy). The faults are parametrized, and new error types can be implemented and introduced for further customization. The tool is expanded to more generic support and will be released within the NEUROKIT2E open-source framework. For more details, please refer to Mateu et. al. [6].

B. Hardware-Efficient AI Modeling

Hardware-aware AI modeling encompasses techniques to create models that are compatible with or optimized for specific hardware. This includes quantization, both post-training and during training, enabling model weights and parameters to align with the data types supported by various accelerators. Additionally, the term hardware-efficient AI modeling covers optimizations that exploit specific hardware features, where a methodology is proposed for accelerator architects to relay critical information to algorithm designers, allowing for tailored implementations that balance latency, power consumption, and energy efficiency.

Optimizations typically involve constraints-based methods, either enforced during training to minimize task-related loss while meeting hardware constraints or applied post-training for

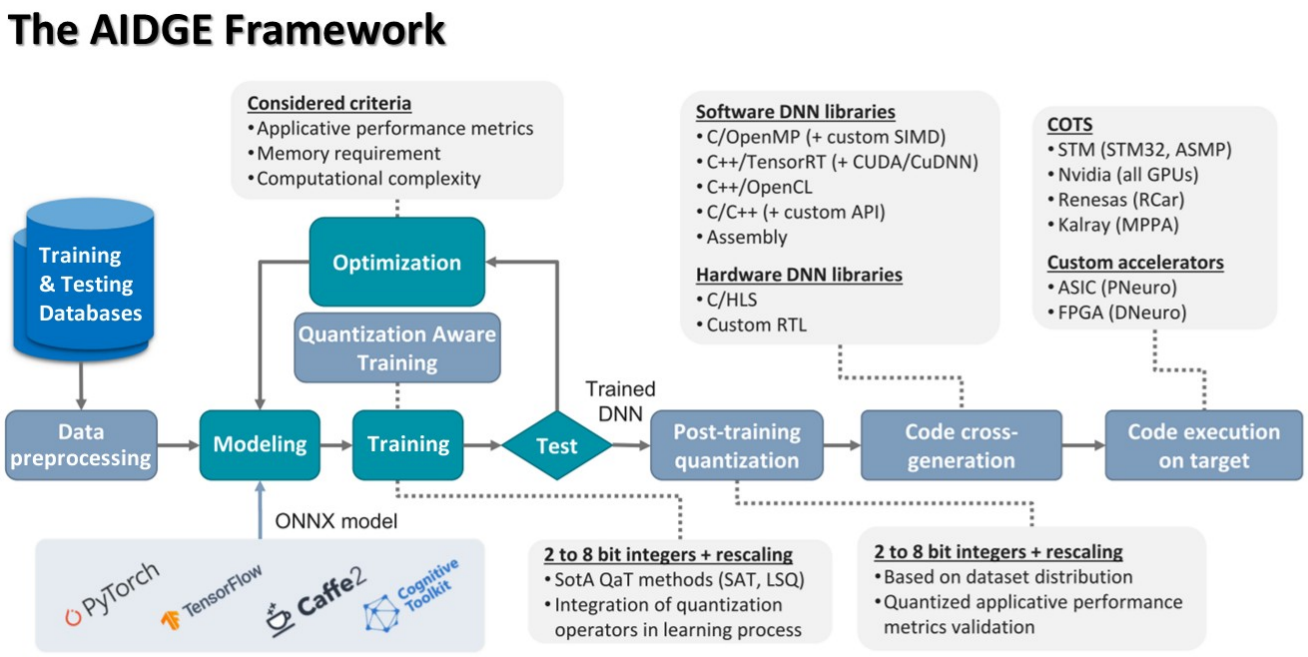


Fig. 2: Schematic representation of the AIDGE Framework with its main components and functionalities.

model fine-tuning. This approach, often termed hardware-aware training, contrasts with the complementary bottom-up methods, which are more amenable to compiler and mapper tools.

Common hardware-efficient modeling techniques in mainstream machine learning include:

- **Weight/activation quantization**, which reduces model weights and activations from floating-point to lower precision formats, speeding up inference by minimizing memory transactions and eliminating the need for floating-point units.
- **Structured sparsification**, a form of pruning that generates sparse weight matrices during training, allowing MAC operations to be skipped during vectorized processing, thereby decreasing inference latency.
- **Model distillation**, which involves training a large *teacher model* to guide the training of smaller *student models* that retain performance while being more memory-efficient.
- **Model pruning**, which iteratively removes synapses with small weights to reduce model size without sacrificing performance, enabling more efficient vectorized processing.
- **Neural architecture search**, an expensive meta-learning method that automates the search for optimal neural network architectures tailored to hardware efficiency constraints.

Moreover, event-based/dataflow acceleration in neuromorphic processors has spurred additional model-driven hardware optimizations for low-latency and energy-efficient processing:

- **Spatio-temporal activation sparsification** encourages sparsity in model activations across spatial and temporal dimensions, beneficial for event-based processors that exploit unstructured sparsity.
- **Activation grouping** delays and batches activation processing in event-based accelerators, optimizing memory I/O by aligning transactions for weight fetching and state updates.
- **Depth-first convolution prefetching-caching** optimizes memory access during convolutional processing, combining input-stationarity with prefetching to lower latency and memory I/O.
- **Asynchronous scheduling optimizations** leverage asynchronous processing, allowing some neurons to fire before all activations in a layer are processed, thus potentially concluding inference early and reducing latency.
- **Compact model re-parameterizations** involve nature-inspired configurations that maintain effective capacity while minimizing memory interactions, such as multi-compartment neurons, dendritic structures and synaptic delays.

IV. Mapping Tool for Multi-Core Heterogeneous Accelerators

The mapping tool solves the task of distributing atomic operations of Deep Neural Networks (DNNs), ensuring efficient on-chip data-flow. Data movements contribute the most to energy consumption in neural network computations [9]. Analog crossbar-based In-Memory Computing (IMC) architectures offer better energy and latency than digital counterparts [10], [11].

Despite the usage benefits of analog crossbar arrays, existing tools like [9], [12]–[14] are mainly intended for digital accelerators or fail to harness some of the analog IMC core properties. Further, the design space of analog cores is large and complex to explore, as factors like IMC core utilization,

Algorithm 1: Multi-core mapping algorithm

```
for layer ∈ DNN layers do
    for core ∈ accelerator cores do
        Compute core mapping space considering IMC constraints
        Calculate resource utilization for core mapping
        Strategy-based search to explore intra-core mapping space
    end for
    Explore the mapping space
    Evaluate inter-core mapping space using cost model
    Update mapped space
end for
```

ADC utilization, and memory hierarchy need to be considered during the workload mapping stage. Although the CiMLoop tool [15] considers analog IMC core properties, it lacks exploration of multi-core architectures, which is crucial as the mapping space grows exponentially with an increasing number of cores. Our multi-core heterogeneous mapping tool distributes workload efficiently per user-specified Key Performance Indicators (KPIs) e.g., throughput, energy, and latency. Additionally, the tool uses operations like Digital to Analog Conversion (DAC), Multiply And Accumulate (MAC), Analog to Digital Conversion (ADC), etc., and their respective costs in the cost model, which makes it agnostic to the memory element and its semiconductor technology.

In Algorithm 1, for every layer in the neural network workload, the mapping space is generated for the cores that support operators defined in the layer. For intra-core mappings, IMC resource utilization is considered. If parts of neural networks are mapped on the IMC core, the mapping algorithm first attempts to realize complete weight stationarity and then optimizes the search strategy for partial weight reconfiguration. Lastly, the mapped space is updated, which can be reconsidered while mapping the next layers in the neural network.

The runtime of the algorithm is dominated by mapping search. Therefore, we have inter-core and intra-core cost models. The inter-core cost model is responsible for making a statistical decision of what valid set of cores are to be explored for unrolling filters in the X and Y dimensions of the crossbar array. The intra-core cost model filters out unwanted mappings to reduce the search time. The described algorithm was validated with a full-fledged analog IMC core-based accelerator [16].

To cater to the rapid design advancements of AI accelerators, in NEUROKIT2E, we aim to extend our tools to support not only analog and digital accelerator cores but also the spiking neural network cores.

V. Hardware Modeling

A. Computation-in-Memory (CIM) Accelerator Modeling

We are also developing models for a Computation-in-memory (CIM) accelerator using emerging resistive random access memory (RRAM). CIM can act as a more efficient alternative to von-Neumann architecture for implementing vector-matrix multiplication (VMM) in neural network hardware [17]–[22]. The fundamental building block of CIM architecture is depicted in Fig. 3. In this process, Digital-to-Analog Converters (DACs) translate bit values at each timestep into corresponding voltages, generating a current within each crossbar column. These currents are captured by sample and hold circuits (S&H) and then converted into digital outputs by Analog-to-Digital Converter (ADC) [23]–[29]. To manage the distribution of weights across the crossbar columns, a shift-and-add operation is applied to the ADC outputs for each column. An additional shift-and-add operation is also performed to combine these partial outputs from multiple timesteps, resulting in the final high-precision digital output [30].

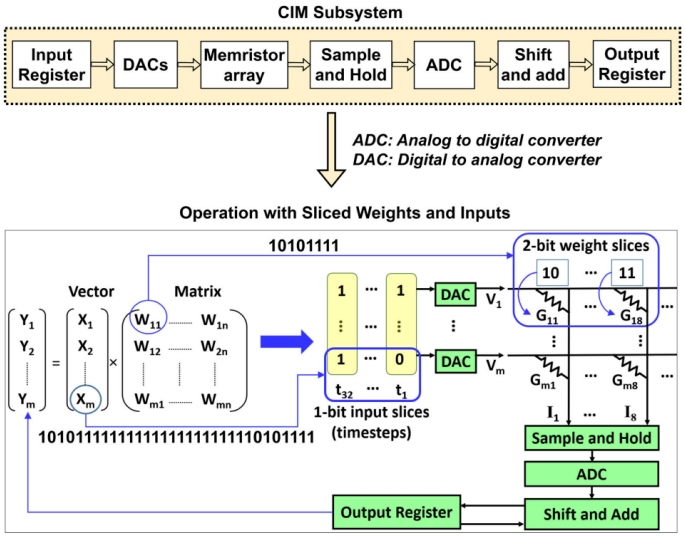


Fig. 3: CIM hardware architecture [31].

B. Non-linear Time-Coding (NTC) modeling

A Non-linear Time-Coding (NTC) paradigm accelerator - also considered within NEUROKIT2E - encodes inputs into PWM-like signals by comparing them with a digital ramp, producing 1-bit signals where the number of clock cycles reflects the input value (see Fig. 4). Time-encoded inputs generate an address to select coefficients from memory, enabling accumu-

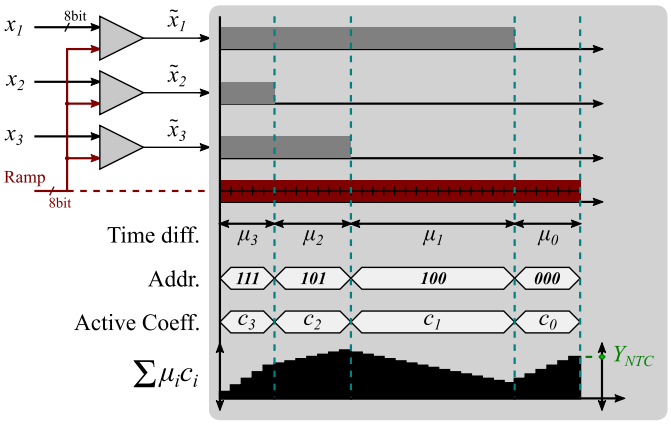
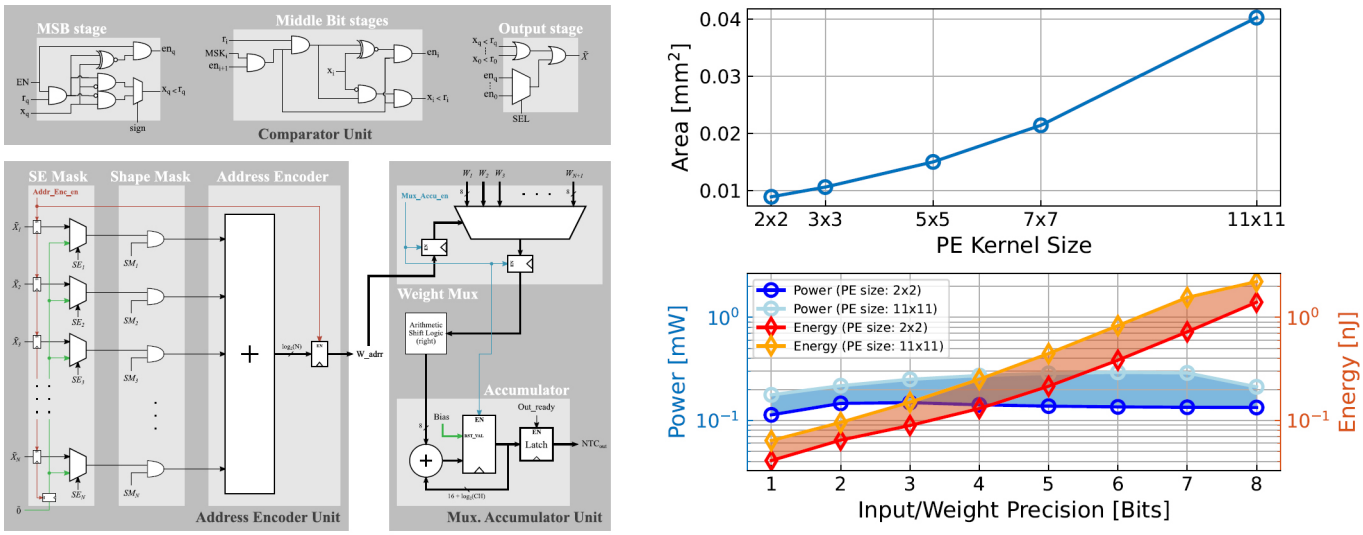


Fig. 4: Non-linear Time-Coding (NTC) processing paradigm.



(a) (b)

Fig. 5: (a) NTC's processing element (PE) architecture, (b) Area, power, and energy consumption of NTC's PE models (modeled).

lation. This method yields a piecewise linear function using only comparisons and additions, resulting in more compact implementations than conventional CNN processing units. The ramp duration, determined by input precision, makes NTC particularly efficient for processing numerous low-precision inputs. However, increasing inputs (N) exponentially raises the required number of coefficients (scaling with 2^N), complicating the efficient implementation of standard CNN layer due to higher area and power consumption. Thus, this project employs a symmetric variant of NTC, inspired by [32], reducing weights to $N+1$ while maintaining the advantages of the generic NTC approach with manageable complexity.

The proposed NTC hardware implementation, as illustrated in Fig. 5(a), is based on the *SymSim* core from [33]. Its comparators leverage the digital ramp's counting behavior, activating only necessary circuit parts to minimize switching activity and power consumption. The *Address Encoder* takes time-encoded inputs and generates weight addresses each clock cycle. The design includes a *Mux. Accumulator* unit with registers for clock gating, arithmetic shift logic for varying weight precisions, and a buffered accumulator to read previous results while computing new ones. Buffered registers also facilitate pipelined writing and processing.

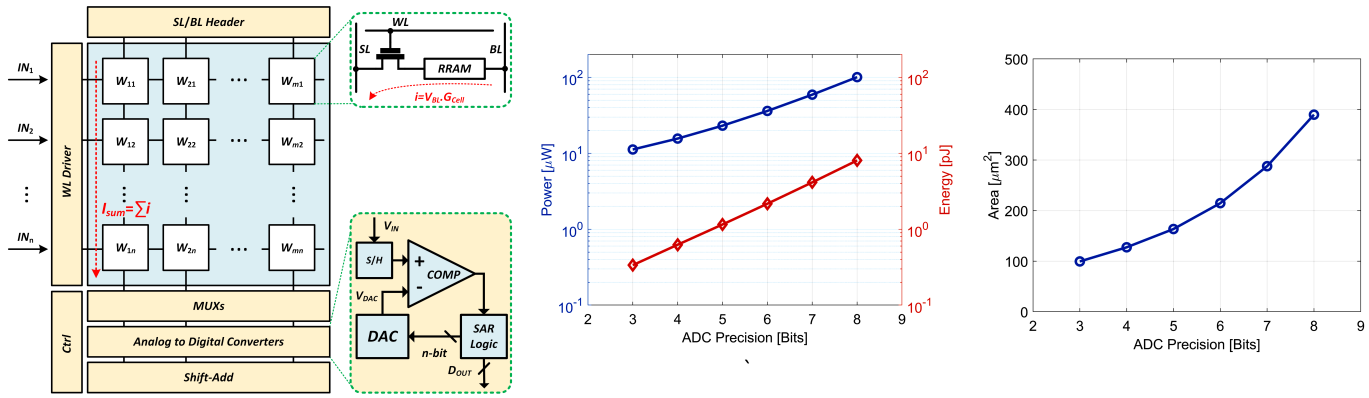
Fig. 5(b) shows the area, power, and energy results for the NTC's PE models, designed with one comparator per input, one *Address Encoder*, and eight *Mux. Accumulators*, utilizing standard CMOS 65nm technology. Area computations are based on a reference cell expressed as AND/OR gates (area approximately $1.8\mu m^2$), ignoring inverters. Power consumption is modeled by counting basic gates, including leakage and dynamic power (affected by a 25 MHz clock). Dynamic power accounts for switching activity during NTC computations, considering toggling in comparators and changes in the *Address Encoder* and *Mux. Accumulator*. Processing time is given by $T_{PE} = 2^q + T_{latency}$, where "$q$" is the input precision and $T_{latency}$ accounts for pipeline register delays. NTC's PE energy is modeled as the power consumption per PE size (based on maximum kernel size) multiplied by the processing time for the input precision used.

VI. Hardware Generation and Optimization

A. RRAM-based CIM Architecture

Fig. 6(a) shows an RRAM crossbar array configured for vector-matrix multiplications (VMM) using computation-in-memory (CIM). Here, deep neural network (DNN) weights are stored as resistance states in 1-transistor-1-resistor (1T1R) bitcells. Inputs applied to each row multiply with stored weights via current flow through the bitcells, and the results are summed along each column to perform multiply-and-accumulate (MAC) operations, enhancing computational parallelism and energy efficiency [34]. However, achieving optimal area and power efficiency in CIM hardware is limited by the need for analog-to-digital converters (ADC) [35], [36]. Flash-ADC and successive-approximation ADC (SAR-ADC) are widely used with voltage-mode or current-mode comparators [37].

Flash-ADC area increases exponentially with comparators, making a 6-bit Flash-ADC about 6.4 times larger than a 6-bit SAR-ADC [38]. However, Flash-ADCs offer lower latency and higher throughput, as their latency is constant, while SAR-ADC latency scales linearly with resolution. Flash-ADCs are often more energy-efficient at lower resolutions (3 bits or less), whereas SAR-ADCs are preferred at higher resolutions (4 bits or more), making them suitable for DNNs [37]. Fig. 6(a) shows a voltage-mode SAR-ADC and its estimated energy consumption (Fig. 6(b)) and area (Fig. 6(c)) in a 40-nm CMOS technology. The SAR-ADC performs one-bit comparisons per clock cycle through binary search, using SAR logic with multistage shift registers to adjust references bit-by-bit from MSB to LSB. A capacitive DAC generates the analog reference voltage via charge redistribution [38].



(a) (b) (c)

Fig. 6: (a) A generic RRAM-based CIM architecture with SAR-ADCs, (b) SAR-ADC power and energy consumption, (c) SAR-ADC area occupation.

B. NTC Implementation and Optimizations

Because of the modular partition of the NTC's PE implementation, as shown in Fig. 5(a), this design allows hardware reusage when performing computations in parallel. For example, in order to process spatial filters with the NTC approach, additional PEs can be synthesized in parallel with shared comparators for those inputs that overlap when sliding the kernel across the input image or tensor. In cases when the same structuring element (SE) and kernel shape mask can be used to compute different features in one NTC layer, several *Mux. Accumulator* units can be connected to the same *Address Encoder* block, producing many features in parallel with minimal hardware replication. Using this approach, the weight multiplexers in the *Mux. Accumulator* units can also be separated into a single one-hot encoder that becomes part of the *Address Encoder* block, with much smaller AND masks and OR trees for the weight selection, considerably reducing the *Mux. Accumulator* hardware footprint, especially when parallelizing the output computations. With the help of these optimization strategies, it is possible to produce a more efficient NTC architecture in terms of both area and power consumption.

Implementing these optimizations into the NTC's PE design (RTL code), a first synthesis with ARM standard cells (65nm technology) and following simulations where performed. Since the most used kernel size for spatial filters in CNNs is 3×3, the synthesized PE for the NTC algorithm has the same number of inputs, and 8 *Mux. Accumulators* for parallel outputs (same as models in Section V). The synthesis process (with Synopsis Design Compiler® tools) produced a PE with an area of approximately 0.0091mm^2, very close to what is predicted by Fig. 5(b), that presents 0.0106mm^2 with a 3×3 sized PE. This small difference between the modeled area and the synthesis results is caused by the tool optimization process, which replaces certain cells (or groups of them) with logic equivalents from the library that are more compact and with better power efficiency. The logic optimization process, combined with reduced activity in the weight accumulator block, also results in a post-synthesis design with lower power and energy consumption compared to the NTC model predictions in Fig. 5(b). This indicates that the models from Section V are valuable as conservative early-stage approximations for the NTC's hardware area and power behavior.

VII. Conclusions

This paper presents the concepts, details, and current progress of the initial phase of our NEUROKIT2E project. We have introduced full-stack solutions aimed at advancing AI-embedded edge systems as well as developing an open-source AI code generation framework. Our contributions include the development of a hardware-aware training and inference tool and mapping tool for multi-core heterogeneous accelerators. Additionally, we presented hardware modeling for a computation-in-memory (CIM) accelerator as well as a non-linear time-coding (NTC) scheme. Furthermore, we discussed hardware development for an RRAM-based CIM architecture, including the implementation and optimization of the NTC. With these efforts, we have made significant progress toward developing energy-efficient and scalable AI solutions at the edge targeting embedded applications.

Acknowledgment

This work was supported by the EU HORIZON-JU-RIA grant "NEUROKIT2E" project (grant agreement No. 101112268).

References

[1] V. Mazzia *et al.*, "Real-time apple detection system using embedded systems with hardware accelerators: An edge ai application," *Ieee Access*, vol. 8, pp. 9102–9114, 2020.

[2] V. Yazdanpanah *et al.*, "Reasoning about responsibility in autonomous systems: challenges and opportunities," *AI & SOCIETY*, vol. 38, no. 4, pp. 1453–1464, 2023.

[3] C. Badue *et al.*, "Self-driving cars: A survey," *Expert systems with applications*, vol. 165, p. 113816, 2021.

[4] H. Hua *et al.*, "Edge computing with artificial intelligence: A machine learning perspective," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[5] L. Huijbregts *et al.*, "Esam: Energy-efficient snn architecture using 3nm finfet multiport sram-based cim with online learning," in *Design Automation Conference (DAC)*, 2024, pp. 1–6.

[6] L. Mateu *et al.*, "Tools and methodologies for edge-ai mixed-signal inference accelerators," in *Embedded Artificial Intelligence*. River Publishers, 2023, pp. 25–34.

[7] A. Pappalardo, "Xilinx/brevitas," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.3333552

[8] G. Gambardella *et al.*, "Accelerated radiation test on quantized neural networks trained with fault aware training," in *Aerospace Conference (AERO)*. IEEE, 2022, pp. 1–7.

[9] A. Parashar *et al.*, "Timeloop: A systematic approach to dnn accelerator evaluation," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[10] N. R. Shanbhag *et al.*, "Benchmarking in-memory computing architectures," *IEEE Open Journal of the Solid-State Circuits Society*, vol. 2, pp. 288–300, 2022.

[11] A. Shafiee *et al.*, "Isaac: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, p. 14–26, Jun. 2016.

[12] Y. N. Wu *et al.*, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[13] L. Mei *et al.*, "Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021.

[14] A. Symons *et al.*, "Towards heterogeneous multi-core accelerators exploiting fine-grained scheduling of layer-fused deep neural networks," 2022. [Online]. Available: https://arxiv.org/abs/2212.10612

[15] T. Andrulis *et al.*, "Cimloop: A flexible, accurate, and fast compute-in-memory modeling tool," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, May 2024, p. 10–23.

[16] B. Kundu *et al.*, "Processing crossbar semiconductor device," 2023, unpublished patent.

[17] S. Diware *et al.*, "Dynamic detection and mitigation of read-disturb for accurate memristor-based neural networks," in *IEEE International Conference on AI Circuits and Systems (AICAS)*, 2024, pp. 393–397.

[18] S. Diware *et al.*, "Mapping-aware biased training for accurate memristor-based neural networks," in *International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2023, pp. 1–5.

[19] R. Bishnoi *et al.*, "Energy-efficient computation-in-memory architecture using emerging technologies," in *International Conference on Microelectronics (ICM)*, 2023, pp. 325–334.

[20] S. Diware *et al.*, "Severity-based hierarchical ecg classification using neural networks," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 17, no. 1, pp. 77–91, 2023.

[21] S. Diware *et al.*, "Accurate and energy-efficient bit-slicing for rram-based neural networks," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 7, no. 1, pp. 164–177, 2022.

[22] A. El Arrassi *et al.*, "Afsram-cim: Adder free sram-based digital computation-in-memory for bnn," in *International Conference on Very Large Scale Integration (VLSI-SoC)*, 2024, pp. 1–6.

[23] A. Singh *et al.*, "A 115.1 TOPS/W, 12.1 TOPS/mm^2 computation-in-memory using ring-oscillator based ADC for edge AI," in *International Conference on Artificial Intelligence Circuits and Systems*, 2023, pp. 1–5.

[24] A. Singh *et al.*, "SRIF: Scalable and reliable integrate and fire circuit ADC for memristor-based CIM architectures," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 5, pp. 1917–1930, 2021.

[25] A. Singh *et al.*, "Accelerating rram testing with a low-cost computation-in-memory based dft," in *International Test Conference (ITC)*, 2022, pp. 400–409.

[26] M. Fieback *et al.*, "Defects, fault modeling, and test development framework for rrams," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 3, pp. 1–26, 2022.

[27] M. Mayahinia *et al.*, "A voltage-controlled, oscillation-based adc design for computation-in-memory architectures using emerging rerams," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 2, pp. 1–25, 2022.

[28] C. Bengel *et al.*, "Reliability aspects of binary vector-matrix-multiplications using reram devices," *Neuromorphic computing and engineering*, vol. 2, no. 3, p. 034001, 2022.

[29] A. Singh *et al.*, "Referencing-in-array scheme for RRAM-based CIM architecture," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 1413–1418.

[30] S. Diware *et al.*, "Hardware-aware quantization for accurate memristor-based neural networks," in *International Conference on Computer Aided Design (ICCAD)*, 2024.

[31] S. Diware *et al.*, "Reliable and energy-efficient diabetic retinopathy screening using memristor-based neural networks," *IEEE Access*, vol. 12, pp. 47 469–47 482, 2024.

[32] N. Rodriguez *et al.*, "Exploration of deep neural networks with symmetric simplicial layers for on-satellite earth observation processing," in *Argentine Conference on Electronics (CAE)*, 2022, pp. 31–36.

[33] N. Rodriguez *et al.*, "RISC-V based SoC platform for neural network acceleration," in *Argentine Conference on Electronics (CAE)*, 2024, pp. 142–147.

[34] M. A. Yaldagard *et al.*, "Read-disturb detection methodology for rram-based computation-in-memory architecture," in *International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2023, pp. 1–5.

[35] A. Singh *et al.*, "Low-power memristor-based computing for edge-ai applications," in *International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[36] H. Jiang *et al.*, "Enna: An efficient neural network accelerator design based on adc-free compute-in-memory subarrays," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 1, pp. 353–363, 2023.

[37] S. Yu *et al.*, "Compute-in-memory chips for deep learning: Recent trends and prospects," *IEEE Circuits and Systems Magazine*, vol. 21, no. 3, pp. 31–56, 2021.

[38] H. Jiang *et al.*, "Analog-to-digital converter design exploration for compute-in-memory accelerators," *IEEE Design & Test*, vol. 39, no. 2, pp. 48–55, 2022.