Hardware-efficient tractable probabilistic inference for TinyML Neurosymbolic AI applications

Jelin Leslin*, Martin Trapp†, Martin Andraud*‡
*Aalto University, Department of Electronics and Nanoengineering {firstname.lastname}@aalto.fi
†Aalto University, Department of Computer Science {firstname.lastname}@aalto.fi
‡UC Louvain, ICTEAM, Belgium {firstname.lastname}@uclouvain.be

2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS) | 979-8-3315-2037-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/COINS65080.2025.11125733

***Abstract*—Neurosymbolic AI (NSAI) has recently emerged to mitigate limitations associated with deep learning (DL) models, e.g. quantifying their uncertainty or reason with explicit rules. Hence, TinyML hardware will need to support these symbolic models to bring NSAI to embedded scenarios. Yet, although symbolic models are typically compact, their sparsity and computation resolution contrasts with low-resolution and dense neuro models, which is a challenge on resource-constrained TinyML hardware severely limiting the size of symbolic models that can be computed. In this work, we remove this bottleneck leveraging a tight hardware/software integration to present a complete framework to compute NSAI with TinyML hardware. We focus on symbolic models realized with tractable probabilistic circuits (PCs), a popular subclass of probabilistic models for hardware integration. This framework: (1) trains a specific class of hardware-efficient *deterministic* PCs, chosen for the symbolic task; (2) *compresses* this PC until it can be computed on TinyML hardware with minimal accuracy degradation, using our n^{th}-root compression technique, and (3) *deploys* the complete NSAI model on TinyML hardware. Compared to a 64b precision baseline necessary for the PC without compression, our workflow leads to significant hardware reduction on FPGA (up to 82.3% in FF, 52.6% in LUTs, and 18.0% in Flash usage) and an average inference speedup of 4.67× on ESP32 microcontroller.**

I. Introduction

Deep learning and deep neural networks (DNNs) have become a standard for embedded AI applications, combining their accuracy and hardware computation properties. Yet, while DNNs are powerful for perception tasks, they often lack explicit structure for representing domain knowledge and can struggle with calibrated uncertainty estimation and providing robust guarantees [1], [2]. Having such guarantees is crucial for embedded applications, for instance regarding explainability or safety-critical aspects. To enable them, one possible solution is Neuro-Symbolic AI (NSAI). NSAI aims to integrate the strengths of data-driven learning of DNNs ("neuro") with the interpretability and reasoning capabilities of symbolic methods [3]. Among symbolic approaches, structured probabilistic models (PMs) are capable of representing dependencies and performing sound inference with formal guarantees [4], [5]. Such combination between DNNs and PMs have been recently proposed for outlier detection [6], uncertainty estimation [7] and related applications [4], [5]. Hence, the benefits of NSAI applications are coming to embedded and TinyML applications, which require hardware implementations that can efficiently execute this symbolic reasoning. In that regard, while DNNs have seen large and successful integration into TinyML [8], symbolic reasoning remains challenging. This is mostly due to the model nature of symbolic tasks: a much higher computation resolution (64-bit floating-point or higher for some PMs) compared to DNNs; and a typically sparse structure (e.g. tree-shaped), dominated by element-wise operations, instead of dense vector-matrix multiplications for DNNs. This presents a core challenge: *How to reliably execute these potentially complex symbolic models on low-precision, resource-constrained TinyML hardware?*

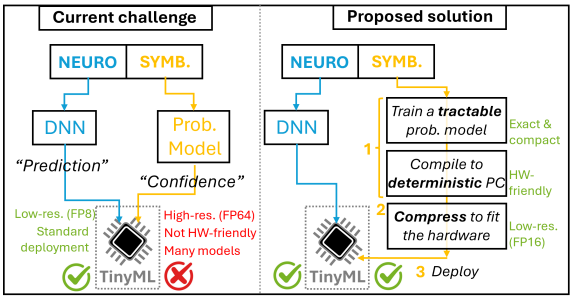

Fig. 1. Overview of the proposed framework

In this work, we propose a complete framework for TinyML NSAI (Fig.1) solving this challenge, achieved in three steps:
(1) Finding a hardware-suitable PM for TinyML. Here, we start with Bayesian Networks (BNs), which offer a powerful way to encode domain knowledge and causal relationships [9]. As exact inference with BNs is computationally expensive, we compile them into "hardware-friendly" Probabilistic Circuits (PCs) [10]. PCs are computational graphs offering a *tractable* representation (i.e. exact and computationally-manageable) [11] for multiple probabilistic queries, as demonstrated in recent NSAI applications [6], [7].
(2) Compressing the PC for TinyML constraints. As most TinyML hardware have a limited **computation resolution** using 16 or 32-bit precision, only small PCs are computed without underflow [12]. Hence, our key novelty lies in leveraging the structural and numerical properties of deterministic PCs (det-PC), such as the pre-selected PC in step (1). It allows us to perform a n^{th}-root transformation of the model parameters. This effectively compresses the model's dynamic range to fit within the constraints of TinyML hardware and preserves the model's exact inference capabilities.
(3) Hardware NSAI deployment. After compression, we

979-8-3315-2037-3/25/$31.00 ©2025 European Union

Authorized licensed use limited to: Univ Catholique de Louvain/UCL. Downloaded on January 22,2026 at 09:16:29 UTC from IEEE Xplore. Restrictions apply.

can accelerate both neural and symbolic parts on TinyML hardware. We demonstrate our approach on FPGA and ESP32 microcontroller platforms with various benchmarks. Our key results show significant reductions in memory footprint (up to 82.3% in FF, 52.6% in LUTs, and 18.0% in Flash usage) and average inference speedups of 4.67× for PC models compared to their baseline implementation. Our experiments on existing NSAI tasks confirm the feasibility of deploying PCs as robust reasoning modules on embedded hardware. The techniques demonstrated could also potentially improve performance and reduce memory consumption in existing embedded AI systems relying on discrete Bayesian Network inference, such as those described in [12]–[16].

This work is structured as follows: Section II introduces BNs, PCs, and ACs. Section III discusses the TinyML context, the challenges of hardware implementation, and the key ideas leveraging PC properties. Section IV presents our hardware deployment studies and results.

II. Deterministic PCs as Symbolic Models for TinyML Applications

As a starting point, we consider a NSAI model to be run on a TinyML hardware, such as an ESP32 microcontroller. The NSAI consists of two distinct computing tasks: the "neuro" part, typically a DNN, and the "symbolic" part, which can take the form of various models. Here, we focus on symbolic tasks where we require probability estimation, useful for instance in outlier detection [6], rule-based integration [17], or certifiable robust reasoning [7].

For these types of tasks, the chosen PM has to offer several characteristics: **I** offering *tractable* inference, which means exact and computationally manageable probabilistic queries; **II** having a known potential for efficient hardware implementation, and **III** being computed using the constraints of TinyML hardware, in particular a limited computation resolution. In this section, we will detail our strategy to find a suitable model for these three constraints.

A. Tractable Probabilistic Inference

As a baseline, we start with Bayesian Networks (BNs), as illustrated in figure 2(a). BNs are popular PMs, providing a graphical framework for representing probabilistic relationships among a set of variables $X = \{X_1, ..., X_n\}$ [9]. They consist of a Directed Acyclic Graph (DAG) encoding conditional independencies and Conditional Probability Tables (CPTs) quantifying the strength of these relationships. The joint probability distribution is compactly factored as:

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|\pi_i) \tag{1}$$

where π_i denotes the set of parent variables of X_i. While BNs excel at modeling, exact probabilistic inference (e.g., computing marginal probabilities $P(X_i)$ or conditional probabilities $P(X_i|\mathbf{e})$ given evidence $\mathbf{e}$) is not tractable [9]. Typical exact inference methods, such as Junction Trees [18] are NP-hard, while more computationally efficient methods are approximate, hence give less theoretical guarantees. To satisfy (**I**), BNs have been compiled into computational graphs [19], [20]. Such representation is often referred to as an Arithmetic Circuit (AC) [10], [21], part of the family of *tractable PMs*, called Probabilistic Circuits (PCs).

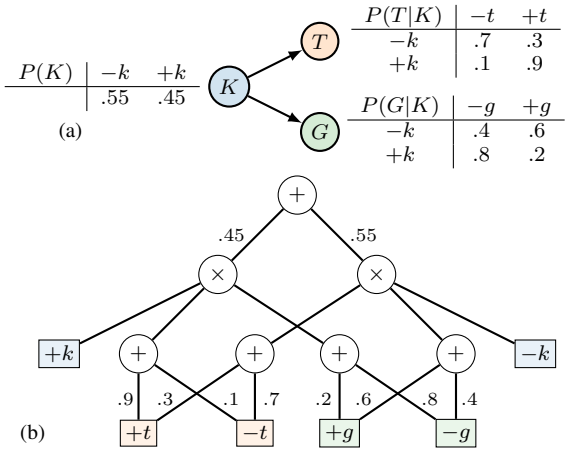

Fig. 2. (a) A Bayesian network encoding a distribution with three variables and (b) An equivalent probabilistic circuit.

B. Efficient Inference with PCs

Figure 2(b) illustrates a PC, computing the same joint probability distribution as the BN, $P(T, G, K)$. It takes the form of a computational graph, that is easily translatable into computational steps (potentially satisfying **II**). A PC is composed of input nodes (representing variable assignments or binary indicator variables), sum ($\oplus$) nodes, and product ($\otimes$) nodes. Intuitively, product nodes represent independence assumptions between random variables (RVs) and sum nodes to represent mixtures, i.e., replacing the independence assumptions of product nodes with conditional independence assumptions. In addition, each node in a PC is associated with a scope, assigned through a scope function, which determines the set of RVs/input dimensions a node defines a probability distribution over. Evaluating the PC for a given evidence is tractable, i.e. computes the probability of that evidence in time linear in the size of the circuit [9].

The tractability and efficiency of PCs often depend on their structural properties [10]:

- **Decomposability:** Product nodes combine inputs representing disjoint sets of variables.
- **Smoothness:** Sum nodes have children that scope over the same set of variables.
- **Determinism:** For any input, at most one child of a sum node evaluates to non-zero.

These structural properties are crucial as they determine the efficiency of answering various probabilistic queries, such as computing marginal probabilities of variables (MAR), finding the most likely state of a subset of hidden variables given evidence (e.g., diagnosing a disease from symptoms) (Maximum A Posteriori-MAP), or determining the overall most

Authorized licensed use limited to: Univ Catholique de Louvain/UCL. Downloaded on January 22,2026 at 09:16:29 UTC from IEEE Xplore. Restrictions apply.

probable assignment to all hidden variables explaining the evidence (e.g., reconstructing a full image from a partial one) (Most Probable Explanation-MPE). While MAR is generally tractable for smooth and decomposable PCs, determinism enables efficient MAP and MPE inference. MAP, seeking the most likely hidden state $\mathbf{h}'^*$ given evidence $\mathbf{e}$ and a subset of hidden variables H', is defined as:

$$\mathbf{h}'^* = \arg \max_{\mathbf{h}' \in \text{dom}(H')} P(\mathbf{h}' \mid \mathbf{e}) \propto \sum_{\mathbf{h}'' \in \text{dom}(H \setminus H')} P(\mathbf{h}', \mathbf{h}'', \mathbf{e}). \tag{2}$$

While potentially tractable via dynamic programming in general PCs [11], MAP inference becomes particularly efficient and often analytically solvable in linear time within deterministic PCs due to their unique path structure. MPE, aiming to find the most likely assignment $\mathbf{h}^*$ to all hidden variables H:

$$\mathbf{h}^* = \arg \max_{\mathbf{h} \in \text{dom}(H)} P(\mathbf{h}, \mathbf{e}), \tag{3}$$

is generally NP-hard but reduces to a linear-time upward (max-sum):

$$\text{LogMaxVal}(s, \mathbf{e}) = \max_i \{\log w_i + \text{LogMaxVal}(C_i, \mathbf{e})\} \tag{4}$$

and downward pass for reconstruction in deterministic PCs. This unique active path, guaranteed by determinism, underpins the tractability of these complex inference tasks.

III. Compressing Det-PCs for TinyML constraints

Deploying the PCs described in Section II satisfies the constraints (I) and (II). Yet, the TinyML context requires addressing the inherent constraints of embedded hardware ((III)): limited memory, computational power, and, crucially, finite-precision arithmetic.

A. Computing PCs on hardware

In that regard, the apparent computationally-friendly nature of PCs led to specialized hardware for their acceleration [12], [15], [22]. Yet, accelerating PCs on hardware faces two main challenges: their sparsity, resulting in very low parallelization opportunities; and their resolution, as they multiply, add and propagate probabilities. Focusing on the computation resolution, existing accelerators favour linear computing formats with a large resolution. In this case, standard floating-point representations (like float32 or float16) have limited dynamic range and precision, leading to numerical underflow [12]. While this represents a loss of precision, the more critical issue is that underflow often results in an intermediate value becoming exactly zero. Due to the prevalence of multiplications in PCs, a zero value can propagate through the computation, potentially collapsing the final output to zero irrespective of other inputs and thus preventing meaningful inference. This can be addressed by developing custom formats [22], or computing in the logarithm domain. In a comparative study, [15] found that Posit and Logarithmic Number System (LNS) implementations achieved throughput similar to optimized custom floating-point designs. However, Posit incurred higher logic and resource usage, while LNS showed mixed efficiency, often demanding more memory and control logic. These specialized formats, although promising, require dedicated hardware support, which is typically absent in commercial TinyML platforms (like ARM Cortex-M or ESP32). Instead, these platforms rely on standard hardware for integer and float32 operations, which means that they are ill-suited for PC computation.

B. Scaling PCs with Uniform Weight Multiplication

From the previous observation, two solutions can be broadly envisaged: using more computation resolution, or compressing the model to fit e.g. 32-bit computation. Using higher resolution (for instance float64) is possible but this often incurs substantial performance penalties on resource-constrained devices lacking native hardware support, and increases memory usage (as we will show in our experiments, Section IV). On the other hand, compressing PCs is a challenging task, as typical compression techniques used for DNNs do not work on PCs. For instance, pruning the smallest parameters may impact the overall accuracy (i.e. generating zero at the output). Simplifying the PC structure for approximate inference methods are also undesirable, as they sacrifice the core benefit of PCs: guaranteed exactness. Therefore, an ideal compression technique should be able to scale (down) the probability distribution encoded in the PC, while keeping the same relationships between parameters and the same graph structure. In this section, we prove that such compression is possible by *scaling all parameters of the PC* by a constant c, either scaling up if $c > 1$, or scaling down if $c < 1$. This lead to our proposed $n^{th} - root$ compression technique for deterministic PCs.

a) "scaling up" with exponential scaling: Consider a BN over n variables $X = \{X_1, ..., X_n\}$, where π_i denotes the set of parent variables of X_i. The joint distribution is expressed by eq.1 (see section II). Let $\theta_{x_i|\pi_i}$ represent the parameter $P(X_i = x_i|\pi_i)$. After compilation, the corresponding PC encodes a network polynomial related to this distribution. Let the set of all parameters (weights) used within the compiled PC be denoted as $\{\theta\}$. The PC computes a polynomial function $F(\{\theta\})$. When each parameter θ in the circuit is uniformly scaled by a constant $c > 0$ to get a new set of parameters $\{\theta' = c\theta\}$, the polynomial computed by the modified circuit, $F'(\{\theta'\})$, scales with a factor c^n.

□

Proof of Exponential Scaling:

The PC polynomial represents the (potentially unnormalized) probability, often structured as a sum over terms, where each term corresponds to a complete variable instantiation $x = (x_1, ..., x_n)$ compatible with the evidence. Due to the BN factorization structure inherited during compilation, each such term $T_x(\{\theta\})$ is effectively a product involving parameters related to the n variables. When every parameter θ is scaled to $\theta' = c\theta$, each term T_x becomes $T_x(\{\theta'\}) = c^n T_x(\{\theta\})$. This assumes that precisely n parameters (or parameters

Authorized licensed use limited to: Univ Catholique de Louvain/UCL. Downloaded on January 22,2026 at 09:16:29 UTC from IEEE Xplore. Restrictions apply.

whose scaling collectively contributes a factor of c^n) contribute multiplicatively to each full instantiation's value in the polynomial computed by the PC, reflecting the structure derived from the original n-variable BN. The PC computes the polynomial $F(\{\theta\}) = \sum_x T_x(\{\theta\})$. Hence, the scaled polynomial is $F'(\{\theta'\}) = \sum_x T_x(\{\theta'\}) = \sum_x c^n T_x(\{\theta\})$. Factoring out c^n (which is constant for all terms x), we get:

$$F'(\{\theta'\}) = c^n \sum_x T_x(\{\theta\}) = c^n F(\{\theta\}) \quad (5)$$

Crucially, the PC's determinism ensures that for any given input evidence, the circuit structure correctly aggregates these terms (or the unique active path computes the relevant term) without duplication or omission, thus calculating exactly $F(\{\theta\})$ or $F'(\{\theta'\})$. Therefore, the overall output scales precisely by c^n. For instance, scaling every weight in a det-PC by 10 for $n = 10$ variables multiplies the output by 10^{10}. □

b) "scaling down" with $n^{th} - root$ scaling: Conversely, if all weights are scaled by $c < 1$ (e.g., halving each weight with $c = 0.5$), the output shrinks by a factor of c^n, i.e., exponentially in the number of variables. As an illustration, consider square-root scaling, where each parameter is scaled by $c = \frac{1}{\sqrt{k}}$; the output then scales as:

$$F'(\{\theta'\}) = \left(\frac{1}{\sqrt{k}}\right)^n F(\{\theta\}) = k^{-n/2} F(\{\theta\}) \quad (6)$$

This decay behavior is again exact in det-PCs and can be leveraged to study the impact of precision or quantization in hardware without needing Monte Carlo or statistical sampling.

c) Linearity in Log-Domain and Deterministic Aggregation: the multiplicative behavior under weight scaling maps to linearity in log-space. Taking logs of eqn 5 yields:

$$\ln F'(\{\theta'\}) = \ln(c^n F(\{\theta\})) = n \ln c + \ln F(\{\theta\}) \quad (7)$$

This property is critical in log-domain hardware implementations (e.g., log-sum-exp circuits or custom log-scale multipliers), allowing designers to trade off arithmetic depth for accuracy [11]. Moreover, the deterministic structure ensures that each term in the polynomial is disjoint and aggregated exactly once, eliminating concerns about redundant computations or overlapping subpaths.

C. From static scaling to n^{th}-root transformation

Based on the previous analysis, we can devise multiple techniques to avoid computing tiny probability values. A first idea can be **static pre-scaling**, where we multiply weights by a factor $S > 1$ (especially in upper layers prone to underflow) *before* deploying the model online. This leverages the same predictable scaling principle as (Eq. 5) and can be applied offline. However, scaling all weights only shifts the distribution, and does not compress it. Hence, this methodology does not reduce the computational range. Another method is **dynamic rescaling**, where we dynamically scale the weights while the inference runs on the device if a given computation leads to underflow. upon underflow detection, the intermediate values below the underflow threshold are multiplied by a scaling factor (e.g. $SF = 10$), before moving on to the next layer. We can keep track of the current scaling factor on hardware by using a counter k. Leveraging the predictable scaling (Eq. 5), the true result is recovered by calculating P_{scaled}/SF^k. Although this technique scales only the necessary nodes, it requires restarting the inference when encountering underflow. On our test cases, this occurred $k \approx 28$ times on average, often in upper circuit layers (as multiplicative paths lead to progressively smaller values deeper in the circuit, making upper levels prone to exhibiting the final underflow). As a result, the inference cost is drastically increased. To obtain a lower computation resolution and effectively compressing the distribution, we propose the **n^{th}-Root Transformation**. Here, we replace weights θ with $\theta^{1/n}$. The integer n is chosen minimally such that $(P_{\text{min}})^{1/n} \geq \epsilon_{uf}$, using the circuit's minimum possible non-zero output P_{min} [12] and the underflow threshold ϵ_{uf} of the given hardware. This compresses values towards 1, preventing underflow. The deterministic structure analyzed in Section III, which guarantees a single active computational path for any evidence and allows the n^{th} root operation to distribute correctly over the effective multiplications along that path ($(a \cdot b)^{1/n} = a^{1/n} \cdot b^{1/n}$), ensures this transformation propagates correctly. This enables exact recovery post-inference via $(F'(\mathbf{x}))^n$.

IV. Hardware NSAI deployment studies for TinyML

Our proposed workflow, outlined in Fig.3, is evaluated on two hardware platforms: (1) **FPGA:** the hardware accelerator logic for the PC evaluation are generated using the AMD Vitis HLS tool [23] from C++ descriptions and synthesized for a target clock frequency of 150 MHz on a Xilinx Virtex UltraScale+ device (`xcvu9p-flga2104-2-i`). We provide inputs and collect results via the onboard System-on-Chip (SoC) processor. For a focused analysis on the benefits of lower precision enabled by the n^{th}-root transformation, we excluded the use of DSP blocks on the FPGA during synthesis, forcing computations to rely on logic resources (LUTs and Flip-Flops - FFs). We compare our results with a baseline double-precision (float64) implementation.

(2) **ESP32:** we use a standard Atom-m5 module, featuring a dual-core 32-bit Tensilica Xtensa LX6 microprocessor running at 240 MHz [24]. This platform is representative of low-power controllers used in TinyML and IoT applications. While the LX6 core includes a hardware Floating-Point Unit (FPU) supporting single-precision (float32) arithmetic, double-precision (float64) operations are emulated in software by the compiler, at the expense of execution time. We compared a baseline implementation using float64 with an implementation using native float32 precision enabled by the n^{th}-root transformation.

Authorized licensed use limited to: Univ Catholique de Louvain/UCL. Downloaded on January 22,2026 at 09:16:29 UTC from IEEE Xplore. Restrictions apply.

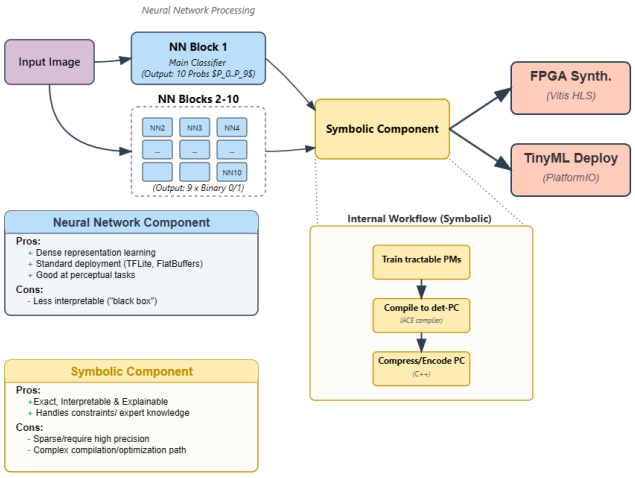

Fig. 3. NeSy deployment workflow

The numerical optimisations achieved via the n^{th}-root transformation, along with the resulting FPGA resource utilization and ESP32 performance, are summarized in Table I. The values of n chosen for each model and platform reflect the different P_{min} values and the target precision's underflow threshold (ϵ_{uf}).

A. Results on PC-only benchmarks

Experiments use the DNA (180 variables) and BNetFlix (100 variables) datasets commonly used in the PC community [25]. These benchmarks are chosen for their complexity suitable for testing numerical limits on target hardware (FPGA or embedded processor). On the FPGA, our n^{th}-root transformed weights allow us to use half-precision (float16) instead of double precision as used otherwise. This results in dramatic resource savings: FF usage was reduced by over 72% and LUT usage by 40-52%. This reduction is critical for fitting larger or multiple models onto resource-constrained FPGAs. On the ESP32, the benefit manifests primarily as speed. The transformation allows us to use native float32, instead of emulated double precision to compute the PC. This yields to inference speedups of 4.32× for BNetFlix and 4.71× for DNA. Additionally, the code footprint (Flash usage) is reduced by 13-18%, which is also valuable in memory-limited embedded systems. These results clearly demonstrate the effectiveness of leveraging the PC's properties to enable efficient low-precision hardware execution.

B. Results in the case of a NSAI system

Many recent Neuro-Symbolic systems combine neural networks for perception with PCs or similar formalisms for reasoning [4], [5], [7]. This often involves computationally heavy NN modules feeding into a PC reasoning component. As a second experiment, we chose a representative NSAI task "colep", containing both a DNN and a symbolic part using a PC [7]. Here, the DNN task (i.e. image recognition) is a main DNN coupled with "knowledge models" implemented with smaller DNNs. The knowledge models can recognize specific image features to get a degree of confidence in the DNN prediction, allowing for uncertainty quantification. This degree of confidence is estimated by the symbolic part, taking all DNN outputs and computing a likelihood value.

Regarding this task, we performed a preliminary evaluation of resource usage for such a hybrid system structure on the ESP32, targeting a CIFAR-10 classification task. In the neuro part, we consider only one main DNN classifier (represented by a pre-optimized ResNet model from MLPerf Tiny benchmarks [8]) producing class probabilities ($\hat{\boldsymbol{\pi}}$). On the symbolic part, we use a det-PC which takes the DNN's 10 probabilities ($\hat{\boldsymbol{\pi}}$) and potentially 9 additional knowledge attributes ($\mathbf{k}$) as input , for a total of 19 variables. For hardware implementation we focus on the main DNN and the PC only, while we assume that the additional attributes ($\mathbf{k}$) are not computed in the device itself, (for instance, 10 devices can be used to share this information, monitoring the input from different angles).

The NN component consists of 9 2D convolutional layers followed by a fully connected layer, totaling 195,616 parameters. The model is quantized to 8-bit weights and 32-bit activations, resulting in a compact TFLite model occupying only 96KB of memory and compiled to utilize integer hardware. We compile and deploy the PC component onto the ESP32 using the n^{th}-Root Transformation discussed in Section III, allowing it to run efficiently on the ESP32's FPU hardware. Despite having only 431 parameters, the PC demands a double higher precision to produce non-zero likelihoods.

We use this trained PC, which models the distribution of consistent inputs seen during offline training ($P(\mathbf{x} = [\hat{\boldsymbol{\pi}}, \mathbf{k}_{\text{true}}])$), as a consistency checker. During this evaluation phase, we feed the PC a test input vector $\mathbf{x}' = [\hat{\boldsymbol{\pi}}', \mathbf{k}'_{\text{true}}]$, composed of the DNN's output probabilities for a given test image ($\hat{\boldsymbol{\pi}}'$) and the corresponding externally provided oracle ground truth attributes ($\mathbf{k}'_{\text{true}}$). If the DNN's output $\hat{\boldsymbol{\pi}}'$ is unusual or conflicting given these true attributes $\mathbf{k}'_{\text{true}}$ (representing a combination the PC hasn't commonly seen), the PC is expected to assign this inconsistent vector $\mathbf{x}'$ a lower log-likelihood, $\mathcal{L}(\mathbf{x}')$. This LLH score thereby serves as a signal indicating how well the DNN's current prediction aligns with the provided ground truth knowledge [7]. The hybrid system had a combined latency of 624 ms per inference, of which only 3 ms is for symbolic component, the combined flash memory usage was 42.95%.

V. Conclusion

In this work, we proposed a method to enable the acceleration of Neurosymbolic AI tasks on TinyML hardware, when the symbolic tasks use tractable probabilistic models. We leveraged the deterministic properties to compress the symbolic model and enable the use of efficient low-precision arithmetic. Our hardware deployment studies on FPGA and ESP32 empirically proved the effectiveness of this approach, yielding significant reductions in resource usage and substantial inference speedups. This validates PCs as practical and

Authorized licensed use limited to: Univ Catholique de Louvain/UCL. Downloaded on January 22,2026 at 09:16:29 UTC from IEEE Xplore. Restrictions apply.

TABLE I
HARDWARE PERFORMANCE SUMMARY FOR PC MODELS WITH n^{th}-ROOT TRANSFORMATION

Metric	BNetFlix_PC	DNA_PC	Cifar10_PC
Model and Numerical Properties			
Minimum Output Value (P_{min})	1.33×10^{-53}	4.6×10^{-271}	1.72×10^{-59}
Weight Root Index n (FPGA, target float16 $\epsilon_{uf} \approx 6.10 \times 10^{-5}$)	13	65	14
Weight Root Index n (ESP32, target float32 $\epsilon_{uf} \approx 1.18 \times 10^{-38}$)	2	9	2
FPGA Resource Usage (pipelined custom block)			
FF (double precision - baseline)	193,112	237,919	211,005
LUTs (double precision - baseline)	119,800	113,972	118,802
FF (half precision - with rooting)	51,807	66,128	37,272
LUTs (half precision - with rooting)	57,837	68,488	56,353
FF Reduction (half vs double)	**73.2%**	**72.2%**	**82.3%**
LUT Reduction (half vs double)	**51.7%**	**39.9%**	**52.6%**
ESP32 Performance			
Flash Usage (float64 - baseline)	26.5%	28.3%	27.95%
Flash Usage (float32 - with rooting)	22.9%	23.2%	23.1%
Flash Saving (float32 vs float64)	**13.6%**	**18.0%**	**17.4%**
Inference Time (float64 - baseline)	10,680 μs	13,893 μs	13,845 μs
Inference Time (float32 - with rooting)	2,470 μs	2,952 μs	2,778 μs
Speedup (float32 vs float64)	**4.32**×	**4.71**×	**4.98**×

robust reasoning modules for embedded Neuro-Symbolic AI systems.

VI. ACKNOWLEDGEMENTS

This work was supported in part by the HORIZON-EIC PATHFINDER challenge SUSTAIN (grant 101071179) and the Academy of Finland's project WHISTLE (grant 332218). Martin Trapp acknowledges funding from the Research Council of Finland (grant number 347279).

REFERENCES

[1] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1321–1330.

[2] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," in *Advances in Neural Information Processing Systems*, 2017.

[3] A. d. Garcez and L. C. Lamb, "Neurosymbolic ai: The 3 rd wave," *Artificial Intelligence Review*, vol. 56, no. 11, pp. 12 387–12 406, 2023.

[4] S. Sidheekh, P. Tenali, S. Mathur, E. Blasch, and S. Natarajan, "On the robustness and reliability of late multi-modal fusion using probabilistic circuits," in *International Conference on Information Fusion (FUSION)*. IEEE, 2024, pp. 1–8.

[5] V. Manginas, N. Manginas, E. Stevinson, S. Varghese, N. Katzouris, G. Paliouras, and A. Lomuscio, "A scalable approach to probabilistic neuro-symbolic verification," *arXiv preprint arXiv:2502.03274*, 2025.

[6] R. Peharz, S. Lang, A. Vergari, K. Stelzner, A. Molina, M. Trapp, G. Van den Broeck, K. Kersting, and Z. Ghahramani, "Einsum networks: Fast and scalable learning of tractable probabilistic circuits," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7563–7574.

[7] M. Kang, N. M. Gürel, L. Li, and B. Li, "Colep: Certifiably robust learning-reasoning conformal prediction via probabilistic circuits," *arXiv preprint arXiv:2403.11348*, 2024.

[8] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau *et al.*, "Mlperf tiny benchmark," *arXiv preprint arXiv:2106.07597*, 2021.

[9] A. Darwiche, *Modeling and reasoning with Bayesian networks*. Cambridge University Press, 2009.

[10] Y. Choi, A. Vergari, and G. Van den Broeck, "Probabilistic circuits: A unifying framework for tractable probabilistic models," *UCLA. URL: http://starai. cs. ucla. edu/papers/ProbCirc20. pdf*, p. 6, 2020.

[11] A. Choi and A. Darwiche, "On relaxing determinism in arithmetic circuits," in *International Conference on Machine Learning*. PMLR, 2017, pp. 825–833.

[12] N. Shah, L. I. G. Olascoaga, W. Meert, and M. Verhelst, "Problp: A framework for low-precision probabilistic inference," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.

[13] S. Zermani, C. Dezan, H. Chenini, J. P. Diguet, and R. Euler, "FPGA implementation of bayesian network inference for an embedded diagnosis," in *Conference on Prognostics and Health Management (PHM)*. IEEE, 2015, pp. 1–10.

[14] J. Schumann, K. Y. Rozier, T. Reinbacher, O. J. Mengshoel, T. Mbaya, and C. Ippolito, "Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems," *International Journal of Prognostics and Health Management*, vol. 6, no. 1, 2015.

[15] L. Sommer, L. Weber, M. Kumm, and A. Koch, "Comparison of arithmetic number formats for inference in sum-product networks on fpgas," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 75–83.

[16] M. Andraud, L. Galindez, Y. Lu, Y. Makris, and M. Verhelst, "On the use of bayesian networks for resource-efficient self-calibration of analog/rf ics," in *International Test Conference (ITC)*. IEEE, 2018, pp. 1–10.

[17] J. Maene, V. Derkinderen, and P. Z. Dos Martires, "Klay: Accelerating arithmetic circuits for neurosymbolic ai," in *International Conference on Learning Representations (ICLR)*, 2025.

[18] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 50, no. 2, pp. 157–194, 1988.

[19] M. Chavira and A. Darwiche, "Compiling bayesian networks using variable elimination," in *Conference on Uncertainty in Artificial Intelligence (UAI)*. Morgan Kaufmann Publishers Inc., 2000, pp. 87–94.

[20] M. Chavira, A. Darwiche, and M. Jaeger, "Compiling relational bayesian networks for exact inference," *International Journal of Approximate Reasoning*, vol. 42, no. 1-2, pp. 4–20, 2006.

[21] A. Darwiche, "A differential approach to probabilistic inference," *Journal of the ACM (JACM)*, vol. 50, no. 3, pp. 280–305, 2003.

[22] N. Shah, L. I. G. Olascoaga, S. Zhao, W. Meert, and M. Verhelst, "Dpu: Dag processing unit for irregular graphs with precision-scalable posit arithmetic in 28 nm," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 8, pp. 2586–2596, 2021.

[23] AMD, *AMD Vitis High-Level Synthesis (HLS) Tool*, 2024, accessed: 2024-03-31. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

[24] E. Systems, *ESP32 Series Datasheet*, January 2023, https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[25] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch, "Automatic mapping of the sum-product network inference problem to fpga-based accelerators," in *International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 350–357.

Authorized licensed use limited to: Univ Catholique de Louvain/UCL. Downloaded on January 22,2026 at 09:16:29 UTC from IEEE Xplore. Restrictions apply.