TU Delft

Delft University of Technology

Document Version
Final published version

Licence
CC BY

Citation (APA)
den Blanken, D. M. J., & Frenkel, C. (2026). Chameleon: A Multiplier-Free Temporal Convolutional Network Accelerator for End-to-End Few-Shot and Continual Learning from Sequential Data. *IEEE Journal of Solid-State Circuits*, *61*(7), 3814 - 3829. https://doi.org/10.1109/JSSC.2025.3645640

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

Chameleon: A Multiplier-Free Temporal Convolutional Network Accelerator for End-to-End Few-Shot and Continual Learning from Sequential Data

Douwe den Blanken, *Graduate Student Member, IEEE*, and Charlotte Frenkel, *Member, IEEE*

***Abstract*—On-device learning at the edge enables low-latency, private personalization with improved long-term robustness and reduced maintenance costs. Yet, achieving scalable, low-power (LP) end-to-end on-chip learning, especially from real-world sequential data with a limited number of examples, is an open challenge. Indeed, accelerators supporting error backpropagation optimize for learning performance at the expense of inference efficiency, while simplified learning algorithms often fail to reach acceptable accuracy targets. In this work, we present Chameleon, leveraging three key contributions to solve these challenges: 1) a unified learning and inference architecture supports few-shot learning (FSL), continual learning (CL), and inference at only 0.5% area overhead to the inference logic; 2) long temporal dependencies are efficiently captured with temporal convolutional networks (TCNs), enabling the first demonstration of end-to-end on-chip FSL and CL on sequential data and inference on 16-kHz raw audio; and 3) a dual-mode, multiplier-free compute array allows either matching the power consumption of the state-of-the-art (SotA) inference-only keyword spotting (KWS) accelerators or enabling 4.3× higher peak GOPS. Fabricated in 40-nm CMOS, Chameleon sets new accuracy records on Omniglot for end-to-end on-chip FSL (96.8%, 5-way 1-shot and 98.8%, 5-way 5-shot) and CL (82.2% final accuracy for learning 250 classes with 10 shots), while maintaining an inference accuracy of 93.3% on the 12-class Google Speech Commands dataset at an extreme-edge power budget of 3.1 μW.**

***Index Terms*—Continual learning (CL), digital accelerator, few-shot learning (FSL), keyword spotting (KWS), multiplier-free, sequential data, system-on-chip (SoC), temporal convolutional network (TCN).**

I. Introduction

THE rise of edge computing has driven demand for deploying deep learning models on resource-constrained devices [1], forming the backbone of the Internet-of-Things (IoT) ecosystem [2]. However, most edge artificial intelligence (AI) devices focus on inference [2], relying on pretrained models that cannot be adapted post-deployment. This lack of adaptability limits robustness to emerging features and data distribution shifts [2], making smart sensors unreliable over time [3]. Fig. 1(a) shows that while cloud-hosted retraining

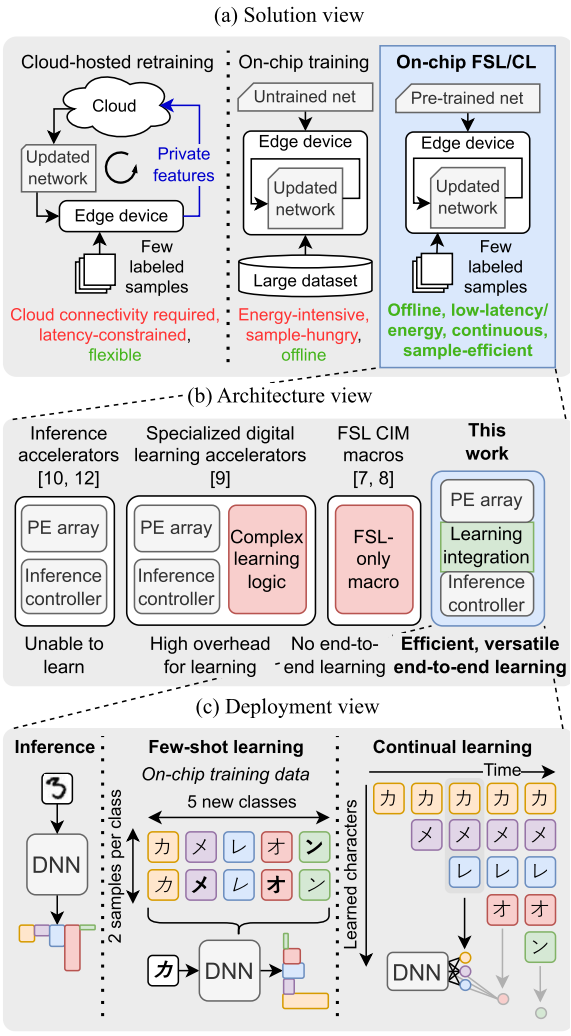


Fig. 1. (a) Overview of three approaches to endow edge AI devices with the ability to adapt, where on-device FSL and CL enable offline, low-latency, and sample-efficient learning. (b) Comparison of an inference and three learning architectures, including ours; we minimize the footprint of learning by integrating it into the inference pipeline rather than treating it separately. (c) Our architecture supports end-to-end on-chip inference, FSL, and CL at a low overhead.

Received 12 June 2025; revised 4 November 2025; accepted 1 December 2025. Date of publication 14 January 2026; date of current version 1 July 2026. This article was approved by Associate Editor Priyanka Raina. This work was supported in part by Dutch Research Council (NWO) through the Project AdaptEdge in NWO Talent Programme–Veni under Grant 20267. *(Corresponding author: Douwe den Blanken.)*

The authors are with the Microelectronics Department (EEMCS Faculty), Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: d.m.j.denblanken@tudelft.nl; c.frenkel@tudelft.nl).

Digital Object Identifier 10.1109/JSSC.2025.3645640

© 2026 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

offers a workaround where new labeled data are uploaded from an edge device to a central server for model updates, this approach is cloud-connectivity-dependent and does not fit latency-constrained scenarios, while leading to privacy concerns [4]. On the other hand, training a model from scratch on-device incurs high energy penalties and still requires the availability of a large labeled dataset, which is not realistic at the edge. Therefore, enabling efficient on-device learning at the edge requires the ability to continually learn from limited data.

However, over a device's lifetime, far fewer tasks are learned than inferences performed. Hence, it is key for an edge AI device to support learning without degrading inference performance and efficiency [Fig. 1(b)]. Yet, current efforts to extend inference accelerators with few-shot learning (FSL) capabilities incur significant area, energy, and accuracy penalties. For example, using quantized gradient descent (GD) to implement on-chip FSL [1], [5], [6] requires maintaining all intermediate activations in memory, transposing weight matrices, and using costly data types, such as floating-point (FP) representations [1]. Alternatively, gradient-free FSL approaches using compute-in-memory (CIM) macros [7], [8] have been proposed to accelerate distance computation between input and stored embedding vectors for FSL. However, they fail to provide *end-to-end* learning support as they rely on off-chip FP32 deep neural networks (DNNs) to compute high-quality embeddings. To the best of our knowledge, only one prior work computes the DNN embeddings on chip for gradient-free FSL [9]. This design adds dedicated learning hardware alongside inference modules, yielding a 2× increase in peak power, a 25% area overhead, and requires external memory to store all model parameters for learning, undermining its suitability for extreme edge scenarios.

Another challenge is that the state-of-the-art (SotA) on-chip FSL architectures lack the ability to handle temporal information in two key ways. First, they cannot process *sequential data*. All existing FSL accelerators target image classification tasks [1], [5], [6], [7], [8], [9], whereas real-world applications often involve sequential data, such as keyword spotting (KWS), where audio is typically sampled at frequencies ranging from 16 to 48 kHz. However, due to the limited input context length of current end-to-end KWS accelerators (60 [10], [11], [12] to 101 [13]), feature extraction (FE), such as mel-frequency cepstral coefficients (MFCCs), is typically performed for two-order-of-magnitude reductions in input sequence length at the expense of a 50%–75% area overhead [10], [14]. Hence, capturing temporal dependencies that span multiple orders of magnitude is not only key for complex temporal tasks, it also allows streamlining system design. Second, current on-chip FSL designs cannot perform *sequential learning of tasks with data distributions that shift over time*. This continual learning (CL) scenario is typical of edge applications, where users add features, such as new keywords, over time. While high-quality embeddings produced by off-chip embedders have recently been shown to also support CL over up to 100 classes [15], end-to-end on-chip CL is still an open challenge.

Overall, due to limited on-chip memory and strict power, area, and energy budgets, no extreme-edge solution has been proposed that effectively balances efficiency and versatility for end-to-end learning on sequential data. In this work, we present Chameleon, the first work to demonstrate scalable on-chip FSL and CL on sequential data, in a fully end-to-end fashion, and without degrading the inference performance or efficiency [Fig. 1(c)]. We achieve this through three key contributions.

1) To minimize the footprint of on-device learning, we build on prototypical networks (PNs) [16] to reframe FSL using distance computation as a forward pass through an equivalent fully connected (FC) layer, thereby leading to a *unified learning and inference architecture*. By integrating learning within the inference process, we enable gradient-free, end-to-end FSL and CL at only 0.5% area overhead and 0.02% added latency while setting new FSL accuracy records (96.8%, 5-way 1-shot) and CL accuracy (82.2% final, 89.0% average for learning 250 classes with ten shots) on the Omniglot [17] dataset.
2) To enable accurate FSL and CL with temporal information, a DNN that captures long-range relationships to produce high-quality embeddings is crucial. Hence, we propose to use temporal convolutional networks (TCNs), whose parameter efficiency scales better than recurrent neural networks [18], as efficient on-chip PN embedders for sequential data at the edge. Compared to SotA TCN accelerators [11], [13], [19], we expand the receptive field by 160× while cutting activation memory by 4×.
3) To enable inference at an extreme-edge power budget while facilitating a high throughput for learning, we introduce a dual-mode multiplier-free processing element (PE) array employing only bit shifters. Combined with system-level power gating, we achieve 3.1-μW real-time KWS or 4.3× higher peak GOPS than SotA KWS accelerators.

The remainder of this article is structured as follows. Section II covers DNN training for FSL and silicon deployments for various FSL methods, while the proposed architecture of the Chameleon system-on-chip (SoC) is introduced in Section III. Section IV then presents the measurement results for FSL, CL, and inference tasks, with concluding remarks in Section V. To promote reproducibility, reuse, and improvement, all code for this article is open-source, including the accelerator source code, the training framework, and the test/simulation setup.[1]

II. Background

To give more insight into FSL algorithms, Section II-A presents the *meta-learning* framework used to train DNNs for FSL [20], also known as *learning to learn*. Section II-B then examines various meta-learning methods and their suitability for on-chip deployment.

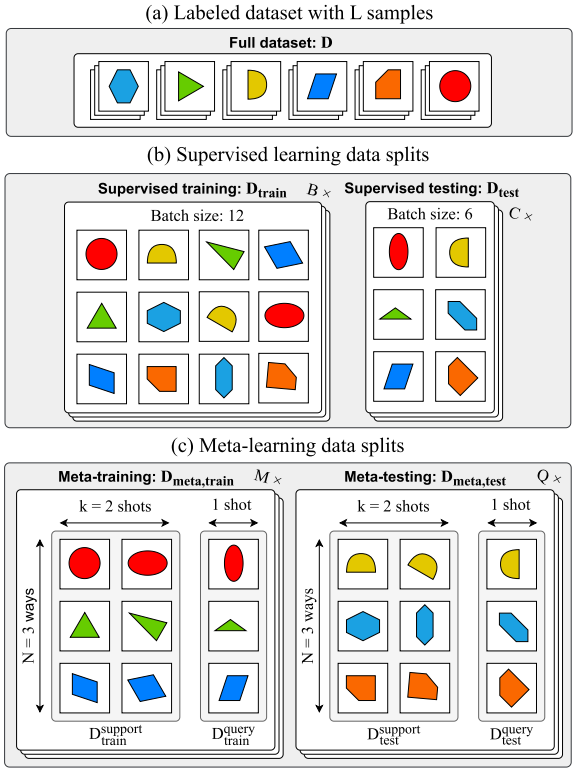


Fig. 2. (a) Standard dataset comprising L labeled examples across multiple classes. (b) Supervised training and testing splits, illustrated for batch sizes of 12 and 6, respectively. (c) Meta-training and meta-testing splits for an example three-way two-shot (D^{support}) task with one-shot query data (D^{query}). Notice that the classes do not overlap between $D_{\text{meta,train}}$ and $D_{\text{meta,test}}$.

A. Meta-Learning Framework

In supervised learning, given a standard labeled dataset $D = \{(x_1, y_1), \ldots, (x_L, y_L)\}$ with L examples x_i and labels y_i [Fig. 2(a)], the training D_{train} and testing D_{test} splits share classes but not examples [Fig. 2(b)]. However, in meta-learning, entire classes are assigned to training or testing splits [Fig. 2(c)], ensuring each split has a distinct set of classes.

For meta-training, a set of M tasks [21], referred to as $D_{\text{meta,train}}$, are used to train the DNN for FSL, defined as

$$D_{\text{meta,train}} = \left\{\left(D_{\text{train}}^{\text{support}}, D_{\text{train}}^{\text{query}}\right)^{(i)}\right\}_{i=1}^{M}, D_{\text{meta,train}} \subset D. \quad (1)$$

The support set $D_{\text{train}}^{\text{support}}$ provides labeled data for the model to learn the task, while the query set $D_{\text{train}}^{\text{query}}$ measures performance on the task. Both sets consist of N ways (unique classes), with k shots (examples) per class.

After meta-training, the learned *meta-knowledge* is used to learn unseen tasks from a set of Q tasks [21] provided by the target dataset $D_{\text{meta,test}}$ during meta-testing

$$D_{\text{meta,test}} = \left\{\left(D_{\text{test}}^{\text{support}}, D_{\text{test}}^{\text{query}}\right)^{(i)}\right\}_{i=1}^{Q}, D_{\text{meta,test}} \subset D. \quad (2)$$

Like before, the support set $D_{\text{test}}^{\text{support}}$ provides data to learn the new task, while $D_{\text{test}}^{\text{query}}$ is used to measure test performance on the task, with both sets potentially differing in ways and shots from meta-training.

[1]https://github.com/cogsys-tudelft/chameleon

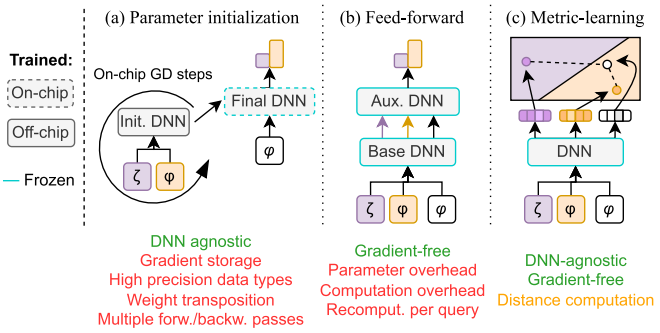


Fig. 3. Qualitative comparison of the suitability of the three meta-learning method types for on-chip deployment.

B. Evaluating Meta-Learning for On-Chip Deployment

While a plethora of meta-learning methods exist that exploit this task-based training regime, they can be categorized into three groups: *parameter initialization*, *feed-forward*, and *metric-learning* methods [21], which we introduce with a focus on their suitability for on-chip deployment.

Parameter Initialization—These methods aim to find optimal initial DNN parameters, enabling quick fine-tuning on unseen tasks with few examples [21]. Model-agnostic meta-learning (MAML) [20] is a widely used approach in this category [21] and employs nested GD. During meta-training, the inner GD loop is carried out for each new task, while the outer loop optimizes the initial parameters, such that the inner loop requires only a few steps of GD to achieve high accuracy. Using the initial parameters resulting from meta-training, the deployment of MAML requires only the inner loop to be executed on-chip [Fig. 3(a)]. Key advantages are that it introduces no extra learnable parameters [20] and imposes no architectural constraints on the DNN. However, a major drawback for on-chip deployment is that GD must be executed on-chip. Although the proposals of initialization-based methods for edge deployment exist [1], [5], [6], they require: 1) buffering all intermediate activations required for GD operation, instead of discarding them as soon as the next-layer values are computed during inference; 2) support for on-chip weight transposition operations; and 3) the use of more expensive data types than quantized inference would typically require, such as (block) FP representations [1], [6]. Additionally, the proposed quantized FSL setup in [1] requires 300 forward and backward passes to be performed for on-chip GD, significantly increasing the compute requirements for learning. Furthermore, to maintain high accuracy, Murthy et al. [1] resort to the Adam optimizer [22], leading to a triplication of the required on-chip weight storage.

Feed-Forward Models—These methods aim to learn a direct mapping from support examples D^{support} to parameters needed for predictions on query examples D^{query} [21]. Unlike initialization-based methods, feed-forward models eliminate the need for gradient updates during deployment. However, they can introduce significant parameter overheads stemming from auxiliary DNNs or require handcrafted neural network (NN) architectures that may not align with DNNs typically

supported for inference [Fig. 3(b)]. A well-known example is the simple neural attentive meta-learner (SNAIL) [23], in which the model predicts, using $N \cdot k$ support example-label pairs and a query example, the class of the query in one forward pass. SNAIL uses the same base DNN as MAML [20] but requires a second, auxiliary module that has ~29× more parameters compared to the base DNN. Another method in this category is memory-augmented NNs (MANNs) [24], which use an external memory to store vector representations of previous tasks for rapid adaptation to new ones. FSL-HDnn [9] implements an on-chip MANN through hyperdimensional computing (HDC). In HDC, base DNN embeddings are projected into a high-dimensional space via multiplication with a random $\{-1, 1\}$-valued matrix [25] (typically >1 M parameters). Classification is then performed by finding the stored class encoding with the closest Hamming distance to the encoded input. In FSL-HDnn, a dedicated module performs encoding, storage, and classification but increases on-chip power consumption by 120% relative to the power for the base DNN's forward pass.

Metric Learning—These methods aim to learn a model that generates vector representations for each support and query sample [Fig. 3(c)], which can then be used to enable the classification of unseen examples without any gradient updates, similar to feed-forward methods. A popular approach in this category is PNs [16]: a feature vector (i.e., *embedding*) is computed for each sample in the support set D^{support}, after which these embeddings are averaged classwise to obtain a set of *prototypes*. Classification is performed by comparing the query sample embedding to the prototypes of all support classes using a *distance function*, where the L2 distance was identified as the most accurate for PNs in [16]. The query is finally assigned to the class with the closest prototype. A key advantage of this method is that, like MAML, it is model-agnostic, requiring no additional network modules and parameters. However, unlike MAML, it is also gradient-free, with only a single extra step—distance computation—outside the standard NN inference pipeline. Kim et al. [7] and Li et al. [8] implement a CIM-based PN variant that stores support examples individually instead of averaging them. Only the embedding storage and distance computation/classification circuits are implemented in silicon, with embedding generation relying on off-chip FP DNN inference. Both designs use L1 distance, causing a 7% accuracy drop in 32-way and 5% in 5-way 1-shot tasks for [7], [8], respectively, compared to an $L2$-based PN in FP32.

III. Chameleon SoC Architecture

This section describes the Chameleon accelerator SoC, as shown in Fig. 4, which highlights the key building blocks that implement our three contributions. The unified architecture for FSL, CL and inference is detailed in Section III-A, while Section III-B covers the mechanism for processing long input sequences using TCNs. Finally, Section III-C discusses the dual-mode multiplier-free PE array.

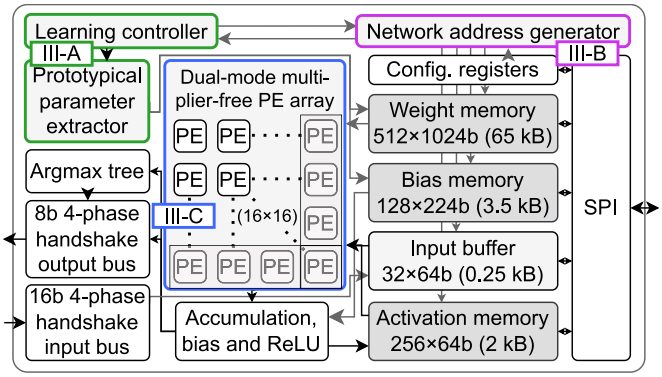


Fig. 4. Architecture of the Chameleon SoC, building on a typical DNN inference accelerator design. The learning controller and prototypical parameter extractor endow this architecture with on-chip learning capabilities, while the network address generator enables long-context learning and inference using TCNs. The dual-mode multiplier-free PE array allows for switching between high-throughput and low-leakage operation.

A. Unified Learning and Inference Architecture

Chameleon's architecture builds on the typical design of a DNN inference accelerator, featuring separate memory banks for activations (4 bits), weights (4 bits), and biases (14 bits). The activation and weight memories are connected to a 16×16 PE array tailored for efficient TCN execution.

To endow this baseline inference accelerator with learning capabilities, we propose to leverage a reformulation of PNs (Fig. 5) as an equivalent FC layer. We begin by defining each prototype as

$$\boldsymbol{P}_j = \frac{\boldsymbol{s}^j}{k}, \quad \boldsymbol{s}^j = \sum_{l=1}^{k} \boldsymbol{e}_{l,j} \tag{3}$$

where $\boldsymbol{e}_{l,j}$ is the lth V-dimensional support embedding for the jth way and k indicates the number of shots. Building on the insight of [16] by using the L2 distance function and squaring it to factor out its square root term, the PN distance computation between a prototype $\boldsymbol{P}_j$ and a query embedding $\boldsymbol{x}$ can be reformulated as

$$\begin{aligned} \left\| \frac{\boldsymbol{s}^j}{k} - \mathbf{x} \right\|^2 = D_j^2 &= \sum_{i=1}^{V} \left(\frac{s_i^j}{k} - x_i \right)^2 \\ &= \sum_{i=1}^{V} \left(\frac{s_i^{j2}}{k^2} + x_i^2 - \frac{2 s_i^j x_i}{k} \right). \end{aligned} \tag{4}$$

As the term $\sum_{i=1}^{V} x_i^2$ is a constant offset that only depends on $\boldsymbol{x}$ and does not influence the *relative* distances D_j between $\boldsymbol{x}$ and the prototypes $\boldsymbol{P}_j$, this yields

$$D_j^2 \propto \frac{1}{k^2} \sum_{i=1}^{V} s_i^{j2} - \frac{2}{k} \sum_{i=1}^{V} s_i^j x_i. \tag{5}$$

Then, after rescaling by $(k/2)$, we equivalently get

$$D_j^2 \propto b_j + \boldsymbol{W}_j \cdot \boldsymbol{x}, \quad \text{with } b_j = \frac{1}{2k} \sum_{i=1}^{V} s_i^{j2}, \quad \boldsymbol{W}_j = \boldsymbol{s}^j. \tag{6}$$

Equation (6) thus corresponds to an FC layer whose output neuron j has a bias b_j and a weight vector W_j, which can both

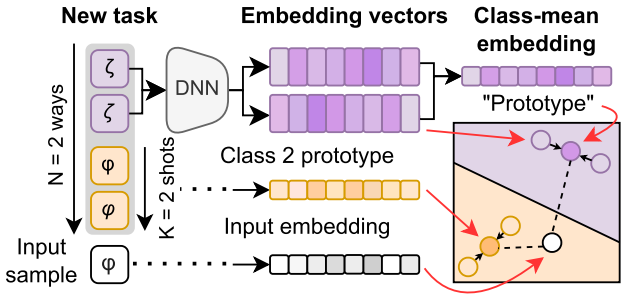


Fig. 5. Learning a new task using prototypes as performed in Chameleon: N ways with k shots are embedded using a fixed DNN to create $N{\cdot}k$ embedding vectors: averaging these classwise results in N prototypes. To classify an input sample, it is embedded and compared via L2 distance to the N prototypes, assigning it to the class with the minimum distance.

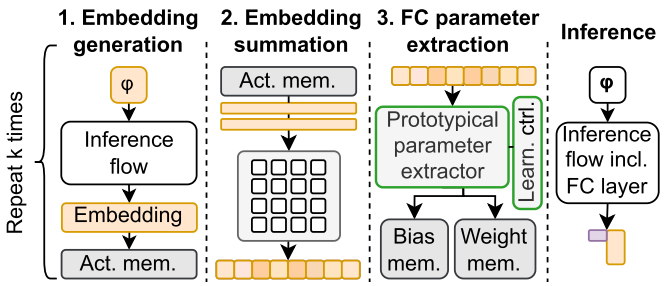


Fig. 6. FSL in Chameleon is performed in three steps. 1. For all k shots of the new class, inference is performed to compute the embeddings, which are saved in the activation memory. 2. The embeddings from the activation memory are loaded and fed into the PE array to compute their sum. 3. This sum is converted to equivalent FC bias and weights that are stored in their respective memories. It is then possible to perform inference using the new FC layer to classify an input sample.

be calculated from s^j, the sum of the k support embeddings as per (3). Although the absolute distances D_j^2 differ from the ones in (4), the relative distance ordering between classes is fully preserved.

By reformulating PNs as an equivalent FC layer, it becomes possible to equip any inference architecture with learning capabilities with minimal overhead. To support learning, only the ability to extract the equivalent FC weight and bias parameters from a prototype is required, since the resulting FC layer can then be accelerated by the existing inference datapath. Importantly, this setup is invariant to the type of off-chip meta-training performed pre-deployment: if the final model produces high-quality embeddings, it can be deployed for on-chip FSL. Moreover, unlike initialization-based and feed-forward meta-learning methods (Section II), PNs naturally extend to CL as this simply requires the storage of additional class prototypes over time. In Chameleon, PN weight and bias extraction is handled by two dedicated modules: the *learning controller* and *prototypical parameter extractor* (Fig. 4, highlighted in green). The former tracks states such as the current way or shot and manages the latter, which extracts the weights and biases for new classes during learning. The learning process is carried out in hardware as per Fig. 6, which illustrates the three steps that Chameleon performs only once when learning each new class (way). First, all shots are embedded, after which their embeddings are summed. Then, these sums are converted to FC weights and biases using the parameter extractor and saved in their respective memories: inference can now be performed for the newly learned class using these parameters.

In terms of area, this process maximizes the reuse of the inference datapath, logic, and memories: as a result, it only induces a 0.5% area overhead relative to the total core area. The learning controller consists of only a few counters and a state machine, while the parameter extractor area scales linearly with the PE array size, independent of the number of shots, ways, or embedding dimensions. These values only affect the on-chip memory requirements, where Chameleon's small 71-kB SRAM is already sufficient to accommodate up to 250 classes.

In terms of latency, the prototypical parameter extraction only requires $(k+2)\lceil V/16\rceil + 1$ additional cycles to learn a new class on top of the embedding computation,[2] scaling linearly with both the number of shots and the embedding size. For example, learning ten new classes with 5 shots results in 0.0075% extra latency compared to the time required to embed the shots. Similarly, inference with a new FC layer takes only $\lceil V/16\rceil \cdot \lceil N/16\rceil$ clock cycles, adding just 0.005% latency in the 10-way scenario.

Overall, our prototypical learning strategy enables a unified architecture for learning and inference, by reusing the existing inference memories and thus eliminating the need for FSL-specific storage. It extends naturally to CL by storing new prototypes over time and can be applied to the existing inference accelerator designs, with learning reduced to minimal control logic for converting prototypes into weights and biases. Unlike designs focused solely on learning, our approach enables a lightweight integration of learning into the inference process.

B. Long-Context Learning and Inference Using TCNs

Real-world natural data are often sequential, including biological signals (e.g., heart rate, blood pressure), environmental data (e.g., wind speed, humidity), and perceptual data (e.g., video, audio). However, the development of algorithms for FSL and CL has historically focused on benchmarks with static data, such as images [21]. To the best of our knowledge, beyond a first proof of concept in [26] for olfactory data, no chip design has demonstrated competitive performance for *sequential* FSL and CL on well-established benchmarks.

As our learning strategy with PNs relies on learning high-quality embeddings, we propose to extend this approach to sequential data by adopting a scalable embedder that can capture long-range dependencies across full sequences. Typical DNN models for learning with temporal data range from recurrent NNs (RNNs) to transformers [27]. On the one hand, RNNs have the advantage of scaling with $O(1)$ memory requirements in sequence length, at the expense of stability issues for long-range dependencies. They typically fail at ≥1k sequence lengths [18] that are necessary, e.g., for raw-audio processing. On the other hand, transformers

[2]The division by 16 accounts for the 16 × 16 PE array.

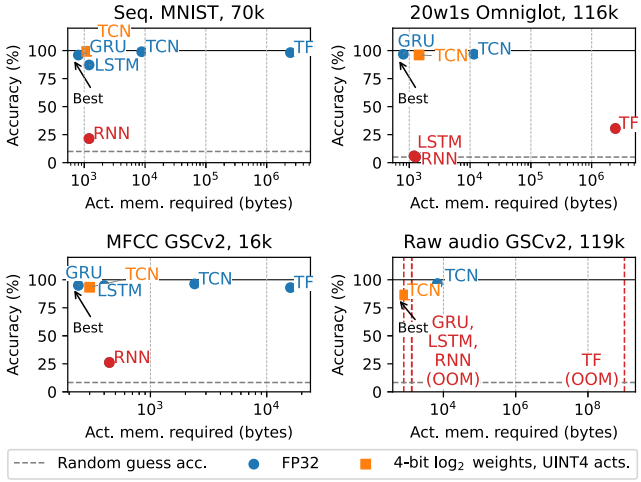


Fig. 7. Accuracy versus required activation memory for RNNs, LSTMs, GRUs, transformers (TFs), and TCNs at iso-parameter counts on benchmarks used for architecture selection in this work. Plain RNNs struggle with all tasks, while LSTMs and TFs fail on sequential Omniglot. GRUs closely trail TCNs in all benchmarks, but training both RNNs and TFs results in OOM errors for raw-audio KWS. Only TCNs succeed in all benchmarks in FP32, an assessment that is preserved when quantized to 4-bit $\log_2$ weights and 4-bit unsigned activations. FP32 accuracies on sequential MNIST have been taken from [29].

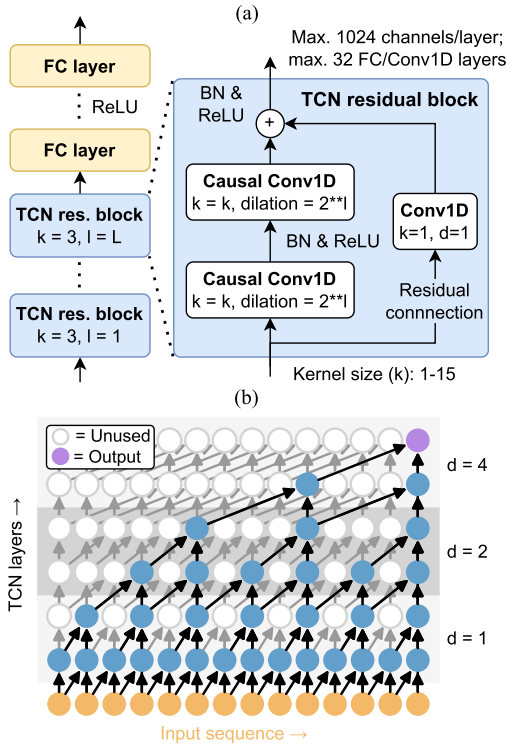


Fig. 8. (a) DNN structure supported by Chameleon. Each TCN residual block contains two causal 1-D convolutions, each followed by BN and ReLU activation. If input and output channels match, the Conv1D residual can be replaced with an identity. Both convolutional layers in a residual block share the same dilation factor d, doubling in successive blocks, starting at $d = 1$. (b) Computational graph of a six-layer TCN (three stacked residual blocks), where white circles indicate zero-valued activations introduced by dilation.

[27] achieve SotA performance when trained on large datasets ($>10^6$ examples), surpassing recurrent and convolutional architectures. Yet, on smaller datasets, RNNs like LSTMs [28] typically outperform transformers [29]. Furthermore, transformer encoders, required for embedding generation, also incur $O(n^2)$ memory and time complexity, making their end-to-end deployment at a μW-level power budget an open challenge. In contrast, convolutional architectures like TCNs [18] offer $O(\log_2 n)$ memory complexity, while iso-parameter comparisons show that they outperform both RNNs and transformers on tasks with $<10^6$ examples [29]. In Fig. 7, we compare plain (Elman) RNNs [30], GRUs [31], LSTMs [28], transformers (TFs) [27], and TCNs on relevant benchmarks for our embedder choice across sequence lengths 63, 784, and 16 000. These benchmarks are sequential MNIST [32], [33] classification, 20-way 1-shot learning on Omniglot [17] and KWS on both MFCC features and directly on the raw data. It can be seen that in FP32, TCNs are the only architecture that succeed in all benchmarks without leading to out-of-memory (OOM) errors. They also remain competitive in both learning and inference when fully quantized, indicating the high quality of the quantized embeddings, even for long sequences.

Hence, we propose to use TCNs [18] as efficient, scalable embedders for inference and FSL on sequential data at the edge. As shown in Fig. 8(a), TCNs employ stacked causal 1-D convolutions with residual connections to ease training in deep networks [34] while preserving the expressiveness enabled by this depth. They are able to capture long-range dependencies via dilation, which doubles at each residual block, resulting in a receptive field (R) that grows exponentially with network depth

$$R = 1 + \sum_{l=1}^{L/2} 2^l \cdot (k-1) \tag{7}$$

where k is the kernel size and L is the number of layers, which should be set to ensure that R matches at least the input sequence length. In classification scenarios, dilation also makes deeper layers exponentially sparser, as illustrated by the white circles in Fig. 8(b), resulting in a streaming memory complexity of $O(\log_2 n)$ when skipping these computations. However, leveraging this favorable memory scaling is nontrivial due to the computational graph structure induced by dilation and the residual connections in TCNs. We solve this by introducing two key techniques.

Our first technique targets the efficient exploitation of dilation-induced sparsity in the TCN's computational graph. In TCN-CUTIE [19], 1-D convolutional kernels are mapped to equivalent 2-D kernels: however, this introduces 80% zero multiplications for $k = 2$, as dilation is emulated via zero-padding, with sequence lengths limited to 24 timesteps for TCN processing. UltraTrail [13] adopts a native 1-D convolutional kernel mapping to perform TCN inference instead but lacks dilation support, thereby restricting its applicability to tasks requiring small receptive fields (demonstrated up to 101 timesteps). Furthermore, its weight-stationary dataflow necessitates full sequence preloading, which is incompatible with streaming inference of long sequences. Giraldo et al. [11] introduce a FIFO-based partial output stationary dataflow that supports dilation and reuses activations across timesteps. However, since the model outputs one

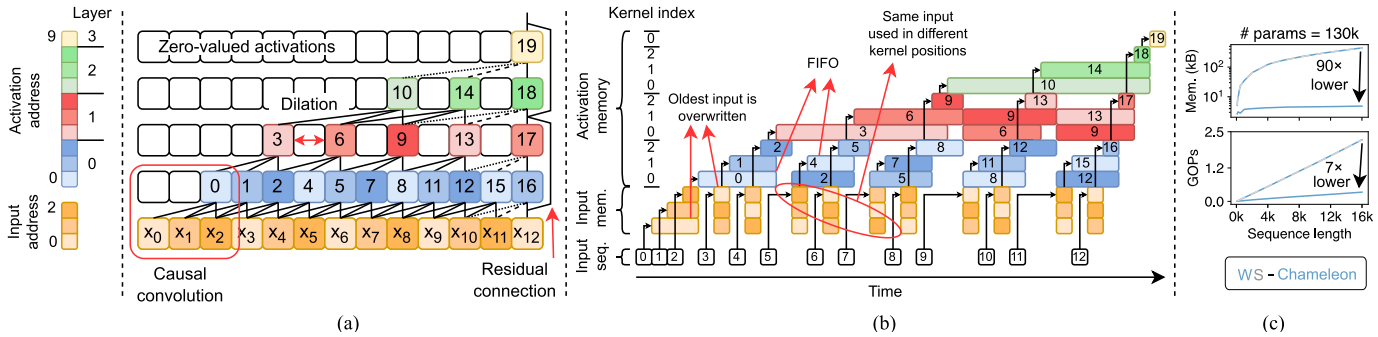


Fig. 9. (a) Four-layer TCN with 13 inputs. Indices show greedy processing order; colors indicate per-layer memory locations. Causal convolutions and residuals are annotated, while the processing and storage of zero activations resulting from dilation (white) is skipped by Chameleon. (b) FIFO-style activation memory allocation over time in Chameleon. Each layer overwrites its oldest activation during execution. Indices show processing order; bar lengths indicate activation lifetimes. (c) Memory and compute comparison between WS TCN inference and Chameleon's strategy, using 130k parameters and producing identical outputs.

classification per input in the sequence and as each output is connected to a different set of TCN graph nodes, this strategy does not avoid computing the unused nodes shown in Fig. 8(b). Expanding on this FIFO-based approach, we introduce a *greedy dilation-aware execution* (Fig. 9) scheme. This scheme is implemented by a specialized network address generator that processes TCNs greedily through layerwise FIFO activation storage while skipping the network's redundant, dilation-induced activations. Fig. 9(a) presents the greedy computational graph of a four-layer TCN as executed in Chameleon together with the per-layer activation memory allocation. Considering the indicated computation order, it can be seen that an activation in the next layer of the TCN is only computed when the previous layer has produced all the required inputs. When additional inputs are required, control reverts to greedily processing earlier layers, one at a time, until this requirement is met. Due to this greedy strategy, both data and structural hazards are inherently avoided. Furthermore, this scheme efficiently leverages our FIFO-style activation memory, as illustrated in Fig. 9(b). The address generator ensures that the output of a new timestep always overwrites the oldest, unused entry in that layer's FIFO buffer. The same mechanism applies to the input memory, where outdated inputs are automatically replaced. Since the computational graph of the deployed TCN is known in advance, operations are scheduled following a predefined layerwise schedule to prevent FIFO underflows. This greedy processing is then combined with dilation-aware operation, which skips computations for zero-valued activations introduced by dilation that do not contribute to the final output. This combination of techniques enables the memory required for TCN inference to scale linearly with depth. However, since the TCN's receptive field doubles with each additional layer, the total activation storage increases only with $\log_2$ of the receptive field size. These scaling properties directly translate into practical gains: we demonstrate in Fig. 9(c) that Chameleon achieves 90× memory and 7× compute reduction over weight stationary (WS) nondilation-optimized TCN inference at a sequence length of 16k (raw audio) with a fixed network size of 130k parameters (Chameleon's maximum supported size).

Our second technique enables efficient handling of residual connections. As sequence lengths grow, larger dilation factors

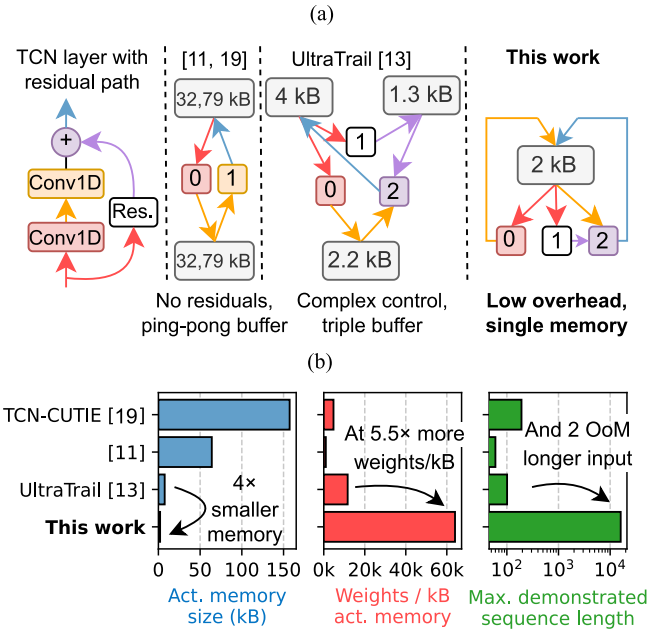


Fig. 10. (a) Comparison of residual operation steps and activation memory sizes across TCN accelerators. Giraldo et al. [11] and Scherer et al. [19] use ping-pong buffers but lack residual layer support; Bernardo et al. [13] require a triple buffer setup for residual layers; Chameleon uses only a single dual-port memory, reducing memory and control complexity. (b) Impact of computation strategy on activation memory size, maximum number of weights/KB of activation memory, and input sequence length between TCN accelerators.

stemming from deeper networks are needed to expand the receptive field (Fig. 8), making residual connections essential for effective optimization of these deep networks [34]. While TCN-CUTIE [19] supports up to ten convolutional layers, it lacks support for residual connections, severely limiting scalability. UltraTrail [13] does support residuals but requires three separate buffers and complex memory access control logic. Giraldo et al. [11] omit residuals but still employ a ping-pong buffer setup. To eliminate the need for multi-buffer schemes to manage residual layers without sacrificing FIFO-style inference, our *residual-aware register file processing* (Fig. 10(a), right) uses a two-port register file that enables simultaneous reads and writes. When the next layer needs the previous layer's output in the same cycle the latter computed,

the network address generator inserts a one-cycle delay to avoid read/write conflicts. This negligible one-cycle latency overhead would only happen for a tiny 16×16 FC layer immediately followed by another FC layer and does not occur for practical network sizes, such as the ones used in this article. Finally, in order to support asynchronous streaming input alongside ongoing processing, Chameleon stores the incoming DNN inputs in a dedicated 0.25-kB memory.

By combining the two techniques described above and as shown in Fig. 10(b), Chameleon reduces activation memory by $76\times$, $28\times$, and $4\times$ compared to prior TCN accelerators [11], [13], [19], while enabling deployment of DNNs with up to $5.5\times$ more weights per kB of activation memory. This enables, for the first time, raw audio KWS on 16k-step inputs—two orders of magnitude longer than in previous TCN accelerators.

C. High-Throughput or Low-Leakage Operation Using a Dual-Mode, Multiplier-Free PE Array

Since learning in Chameleon is integrated within the inference pipeline, we propose two architectural enhancements, so as to: 1) reduce the PE array and memory footprints with inference-optimized quantization, allowing for multiplier-free operation with 4-bit signed $\log_2$ weights; and 2) endow the PE array with the flexibility to optimally support different inference and learning scenarios.

Our first technique focuses on *inference-optimized quantization*. For instance, proposals for deploying parameter initialization meta-learning methods on-chip require GD, which entails FP precision requirements. Murthy et al. [1] use block FP (8-bit shared exponent and 4-bit mantissa) for weights, activations, gradients, and errors. Compared to regular integer operation, additional logic for exponent management, as well as normalization and alignment is required. Wang et al. [6] demonstrate the feasibility of using signed 16-bit uniform quantization of the weights and gradients for various initialization-based meta-learning methods but reduce memory requirements only by $2\times$ compared to learning with FP32. FSL-HDnn [9] uses brain floating point (Bfloat)-16 to generate embeddings for HDC during FSL, also saving $2\times$ memory compared to a vanilla FP32 implementation. A deployment proposal for MAML [5] demonstrates the use of 4-bit signed uniform weights but still requires 8- or 16-bit gradients depending on task difficulty, hence increasing memory requirements by $2-4\times$ compared to inference-only quantization.

As Chameleon operates in a purely forward manner due to its gradient-free learning, it can rely on a simple quantization scheme that is optimized for inference. Hence, we propose to use $\log_2$ weights [35]: this allows replacing the multiplier in each PE with a bit shifter, thereby enabling multiplier-free inference and embedding computation [Fig. 11(a)]. We show the FSL performance for various quantization scenarios in Table I, using the Omniglot dataset [17], where the activation quantization impacts accuracy much more than weight quantization. Yet, significant savings can be achieved: the 4-bit $\log_2$ ($\log_2$-4) quantization scheme limits the accuracy penalty compared to the FP32 baseline to only 1.9%–2.3% in the ten- and 5-shot cases and outperforms INT8 and INT4

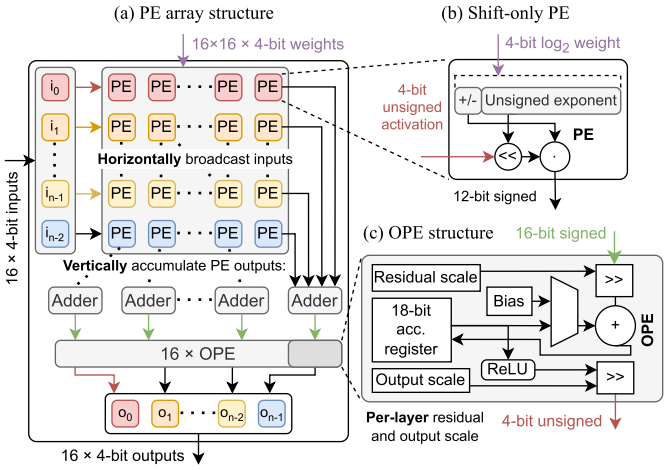


Fig. 11. (a) Diagram of the multiplier-free PE array, operating under an output-stationary dataflow. (b) Individual PE, which uses a bit shift in place of a multiplication. (c) OPE module, which performs input/output rescaling as well as bias addition and ReLU activation.

TABLE I

QUANTIZATION ACCURACY ASSESSMENT ON 20-WAY OMNIGLOT TASKS USING 1, 5, AND 10 SHOTS WITH A 116K-PARAMETER TCN. FOR ALL QUANTIZED EXPERIMENTS, WE APPLY THE FC LAYER CONVERSION OF THE PROTOTYPES PER (6). THE ± SHOWS 95% CONFIDENCE INTERVALS OVER 100 TASKS, WHILE THE BOLD FONT INDICATES THE BEST ACCURACY WHEN FULLY QUANTIZED

Quantization format			20-way accuracy		
Weight	Act.	Proto.	1-shot	5-shot	10-shot
FP32	FP32	FP32	94.5 ± 0.5%	98.4 ± 0.2%	98.7 ± 0.2%
INT8	FP32	FP32	96.1 ± 0.4%	98.8 ± 0.3%	98.9 ± 0.3%
INT8	UINT4	FP32	88.9 ± 0.8%	97.0 ± 0.3%	97.8 ± 0.3%
INT8	UINT4	UINT7	86.4 ± 0.9%	94.8 ± 0.4%	96.2 ± 0.3%
INT4	UINT4	UINT7	**89.2 ± 0.8%**	95.8 ± 0.4%	**96.9 ± 0.3%**
INT4	UINT4	UINT3	86.8 ± 0.8%	91.9 ± 0.6%	93.0 ± 0.6%
$\log_2$-4	UINT4	UINT3	89.0 ± 0.8%	94.2 ± 0.5%	95.1 ± 0.4%
$\log_2$-4	UINT4	$\log_2$-4	**89.1 ± 1.3%**	**96.1 ± 0.5%**	**96.8 ± 0.3%**

weight quantization. The one-shot case is more sensitive to quantization due to reduced prototype precision: while this is illustrated by an increased accuracy degradation against FP32 at 5.4%, performance still compares favorably to SotA chip implementations, as we will show in Section IV-B.

Replacing the 4-bit multiplier inside each PE with a shifter not only improves accuracy but also reduces PE area by 42%. The $\log_2$ PEs [Fig. 11(b)] first left-shift the horizontally broadcast input by the weight's exponent and then apply a sign correction. These outputs are summed vertically and accumulated in 18-bit signed uniform registers within output PEs (OPEs), detailed in Fig. 11(c). The OPE performs input rescaling (for residuals), bias addition, rectified linear unit (ReLU) activation, and output rescaling before its output is written back to the activation memory. Finally, the PE array adopts an output-stationary dataflow during inference to efficiently support our greedy TCN processing scheme. Each cycle, it receives 16 4-bit unsigned uniform ReLU-based activations, either from the input memory (first layer) or activation memory (subsequent layers), along with 16×16 $\log_2$-quantized weights from the weight memory.

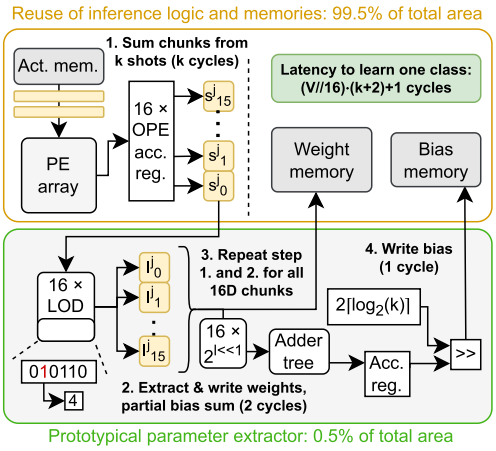


Fig. 12. Detailed circuit-level view of the multiplier-free prototypical parameter extractor, which includes an LOD for the $\lfloor \log_2 \rfloor$ operation. Additionally, the extensive reuse of the inference hardware for learning is illustrated. Per extraction operation, the latency in clock cycles is annotated.

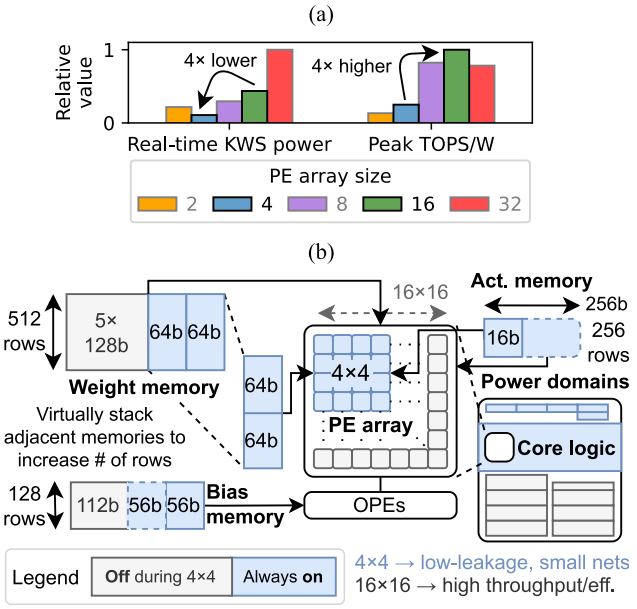


Fig. 13. (a) Comparison of simulated real-time KWS power and peak TOPS/W estimates for different PE array sizes. (b) Data layout for activation, bias, and weight memories to support both 4×4 and 16×16 PE array usage, allocating LSBs of weight memories to the top-left 4×4 section. In 4×4 mode, gray memories are powered off, while the remaining weight memories are virtually stacked to double the row count.

With inference being fully multiplication-free, we aim to remove the remaining multiplications during learning by also adapting (6) to a $\log_2$ formulation. This allows for efficient PN parameter extraction with improved FSL accuracy: Table I shows that $\log_2$-4 prototype quantization unilaterally outperforms UINT, while ensuring that the few-shot-learned FC layer runs on the existing bit-shift PEs, avoiding a dedicated 16×16 multiplier array. To this end, each support embedding sum ($\boldsymbol{s}^j$) is first $\log_2$-quantized, which removes the square in the bias term of (6). As such, the remaining multiplications in the datapath can be replaced by bit shifts

$$\boldsymbol{l}^j = \lfloor \log_2 \boldsymbol{s}^j \rfloor, \quad b_j = \frac{1}{2k} \sum_{i=1}^{V} 2^{l_i^j \ll 1}, \quad \boldsymbol{W}_j = 2^{\boldsymbol{l}^j}. \tag{8}$$

All operations in the above equation, consisting only of bit shifts and additions, are implemented on-chip as part of the prototypical parameter extractor: Fig. 12 shows its detailed circuit diagram. Note that the division by 2k in (8) is approximated by a right shift with $2\lceil \log_2(k) \rceil$ bits, while $\lfloor \log_2 \boldsymbol{s}^j \rfloor$ is computed by extracting the most significant bit's position using a leading-one detector (LOD).

Our second technique allows *efficiently supporting different deployment scenarios*, from leakage-dominated real-time KWS inference to throughput- and dynamic-power-constrained FSL operation with large DNN embedders. For instance, Vocell [10] has a 36% leakage contribution (excluding analog FE) during real-time KWS, similar to the 30% for UltraTrail [13] and ~33% for Giraldo et al. [11] with limited throughputs of 0.13, 3.8, and 0.26 GOPS, respectively. In contrast, TinyVers [12] achieves a $4-135\times$ higher throughput of 17.6 GOPS. However, this comes at the cost of an order-of-magnitude increase in real-time KWS power consumption. Hence, to support low-leakage operation without degrading the design's throughput, we introduce a dual-mode PE array whose size is reconfigurable, which enables Chameleon to efficiently support *both* μW-power real-time inference and high-throughput tasks with large DNN embedders.

To determine the optimal PE array sizes for the two modes, we simulate real-time KWS power and peak TOPS/W performance during FSL in Fig. 13(a), assuming SRAMs dominate total system power. The analysis identifies the array sizes of 4 and 16 as optimal. To support efficient operation for both sizes, we propose to use a single 16×16 PE array while we employ a special memory layout: instead of storing DNN weights purely in row- or column-major order, the top-left 4×4 weights are stored row-major in the first two banks (LSBs), followed by the remaining 240 weights [Fig. 13(b)]. This approach has two key advantages. First, by writing the network parameters for the 4×4 mode only to the LSB memories, referred to as *always-on* banks, the uninitialized MSB memories can be power-gated through a separate power rail that is physically powered off, thereby cutting leakage without data loss. The 240 unused PEs can also be clock-gated to reduce dynamic power. Second, the design of the network address generator can remain mode-independent: in both modes, the addresses are generated in the same way; the only difference is that in the 4×4 mode, a smaller weight matrix is loaded per cycle. Hence, as each address now contains fewer parameters, we virtually stack the LSB weight banks in 4×4 mode to enlarge the address space, supporting 16k weights over 1024 addresses, instead of 512 in 16×16 mode.

Comparing Chameleon to other KWS accelerators in Fig. 14, we observe that its dual-mode operation enables a $4.3\times$ higher peak throughput in 16×16 mode, a 16-fold increase from Chameleon's 4×4 mode. Meanwhile, in 4×4 mode, Chameleon achieves a $2\times$ power reduction for real-time end-to-end KWS inference while maintaining a classification accuracy on par with SotA inference-only accelerators.

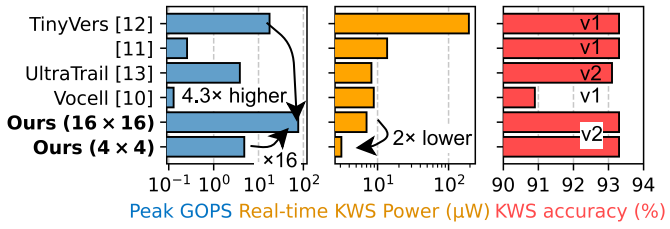


Fig. 14. Peak GOPS, real-time KWS power, and accuracy comparison on 12-class Google Speech Commands (GSC, v1 and v2) between KWS accelerators.

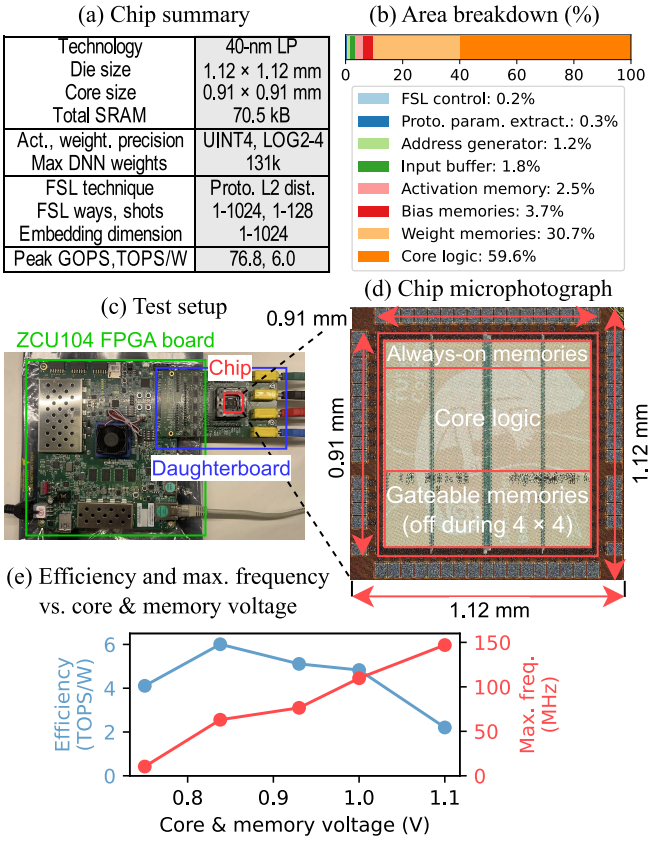


Fig. 15. (a) Metrics summary of the Chameleon SoC. (b) Area breakdown of the SoC per module. (c) Test setup, including a Zynq UltraScale MPSoC ZCU104 FPGA that connects to a daughterboard hosting the chip. (d) Annotated die photograph of the Chameleon SoC. The bottom section of the SoC, containing the gateable memories, can be powered off to enable inference with small networks using the 4×4 mode of the PE array. (e) Efficiency and maximum frequency characterization of Chameleon.

Varying the PE array size only impacts the throughput-power tradeoff, without impacting accuracy.

IV. Measurements

Chameleon was taped out in TSMC 40-nm low-power (LP) CMOS technology (Fig. 15). It occupies 1.25 mm^2 with a core area of 0.83 mm^2, including the power rings. The SoC has two separate power domains, corresponding to: 1) the core logic (using a mix of SVT and HVT standard cells) and always-on memories; and 2) the gateable MSB memories.

We first outline the software and measurement setup (Section IV-A), then present the results for FSL and CL (Section IV-B), followed by real-time KWS results (Section IV-C), and finally compare Chameleon to the SotA (Section IV-D).

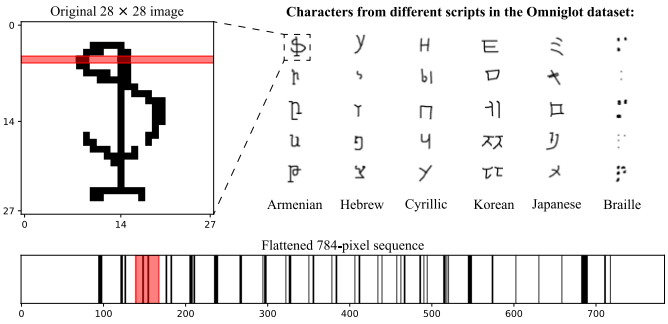


Fig. 16. Sample characters from different alphabets, taken from the Omniglot dataset, including a demonstration of flattening an image from the dataset to enable *sequential Omniglot* on 1-D sequences of pixel data.

A. Software and Measurement Setup

Our TCN models are based on the original architecture in [18], where we introduce several modifications incorporating techniques from modern deep convolutional residual networks (ResNets) [34] for normalization [36], [37] and initialization [37], [38].

To deploy TCNs on Chameleon, we perform quantization-aware training (QAT) using the Brevitas framework [39], after FP32 training. QAT starts from the FP32 checkpoint with the lowest validation loss, with all parameters quantized and batch normalization (BN) layers folded into the weights of the previous layer, as per [40], [41]. Within Brevitas, we implemented a custom variable-bit quantizer for signed $\log_2$ weights and a custom requantizer to simulate overflow of the 4-bit unsigned uniform activations in Chameleon. We apply asymmetric and symmetric quantization for activations and weights, respectively, both with per-tensor scaling [42]. During quantized PN training, the TCN embeddings are using 4-bit unsigned uniform quantization, like activations (see Section III-C). The resulting prototypes are quantized using 4-bit signed $\log_2$ quantization, as they are converted to prototypical parameters as per (8).

Our measurement setup is shown in Fig. 15(c). A daughter board hosting the Chameleon chip connects to external power supplies and is interfaced with a Zynq UltraScale MPSoC ZCU104 evaluation board, which provides test stimuli and records the results. The complete deployment and test setups are part of the open-source repository of Chameleon. Note that all metrics that we report in the remainder of this section are obtained at room temperature.

B. FSL and CL

To demonstrate end-to-end on-chip learning on Chameleon using our unified learning and inference architecture, we benchmark Chameleon for FSL and CL scenarios.

Benchmarking Setup—For both FSL and CL, we use the Omniglot dataset [17], which comprises 1623 classes of

TABLE II

FSL TEST ACCURACY COMPARISON BETWEEN FSL ACCELERATORS THAT HAVE REPORTED RESULTS IN SILICON ON THE OMNIGLOT DATASET. THE ± SHOWS 95% CONFIDENCE INTERVALS OVER 100 TASKS

	5-way		20-way		32-way
	1-shot	5-shot	1-shot	5-shot	1-shot
Kim *et al.** [7]	93.4%	98.3%	-	-	-
SAPIENS* [8]	-	-	-	-	72%
FSL-HDnn [9]	79.0%	-	-	79.5%	-
This work	**96.8 ± 1.6%**	**98.8 ± 0.5%**	**89.1 ± 1.3%**	**96.1 ± 0.5%**	**83.3 ± 1.2%**

* Uses an off-chip FP32 embedder.

handwritten characters from 50 alphabets (see Fig. 16, right, for selected examples). Every class has 20 examples, each of which is drawn by a different person. To align with other works on FSL, we use the Vinyals train-val-test split [43]. Following [16] and [24], we augment all splits by generating new classes via 90°, 180°, and 270° rotations and resize all images to 28×28 for consistency. To adapt the images for processing with TCNs, we flatten each image pixelwise (see Fig. 16, bottom), effectively creating a *sequential Omniglot* representation. While this dataset lacks the temporal complexity of real-world sequential data, it has the benefit of offering a reliable point of comparison with previous work, which we will later complement with the use of the Google Speech Commands Dataset in Section IV-C.

FSL Results—Using a 116k-parameter TCN with 14 layers, we report in Table II the classification accuracies on the Omniglot test set across 5, 20-way 1, 5-shot as well as 32-way one-shot scenarios found in the SotA. It can be seen that Chameleon outperforms the existing works [7], [8], [9] by up to 16 accuracy points, setting new records across all scenarios, even though both FSL CIM designs [7], [8] use FP32 off-chip embedders. Chameleon consumes 11.6 mW for end-to-end FSL on Omniglot at 100 MHz and 1.0 V. At 100 kHz and 0.625 V, it consumes 12.9 μW. The latencies for learning one shot of a new class are 0.59 ms and 0.54 s, respectively, yielding an energy per shot of 6.84 and 6.97 μJ, including the embedding phase. Since prototypical parameter extraction only takes a few clock cycles, <0.02% of the embedding computation time on Omniglot, learning, and inference have effectively identical total energy, power, and latency. This contrasts sharply with other on-chip training methods like CHIMERA [44], which employs low-rank training with 8-bit signed parameters and requires 10^3–10^4 RRAM update steps for learning. Similarly, Park et al. [45] use FP8 for full on-chip SGD-based training and hence could host MAML-like setups. However, their lowest reported power is already 13% higher than Chameleon's peak end-to-end power, demonstrating its incompatibility with extreme-edge operation. Alternatively, instead of learning new classes, Cioflan et al. [46] use embeddings to adapt to the speech characteristics of end users for KWS. A microcontroller updates an embedding layer using few-shot spoken keywords, inducing a 35% FLOPS overhead compared to embedding computation alone.

CL Results—We evaluate CL on Chameleon using the same TCN model as for FSL. CL is performed by learning one new class at a time, up to 250 classes, using 1, 2, 5, or

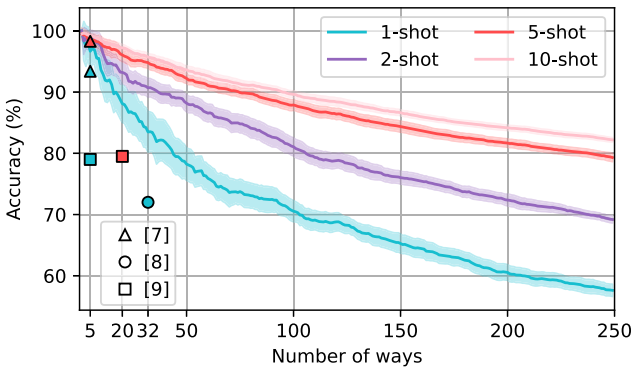


Fig. 17. Average end-to-end CL classification accuracies on the Omniglot dataset using Chameleon for 2–250 ways with 1, 2, 5, and 10 shots, compared to other FSL chips (points outlined in black). The shaded regions indicate 95% confidence intervals over 20 tasks. For <5 ways, shot count has little effect on accuracy; with more ways, additional shots help, with diminishing returns beyond five shots.

TABLE III

ACCURACY (ACC) AND BWT METRICS FOR CONTINUALLY LEARNING 250 CLASSES USING ONE, TWO, FIVE, OR TEN SHOTS. THE METRICS ARE AVERAGES OVER 20 TASKS. THE ± INDICATES 95% CONFIDENCE INTERVALS OVER THESE TASKS

	1-shot	2-shot	5-shot	10-shot
ACC	57.6 ± 0.8%	68.9 ± 0.6%	79.1 ± 0.5%	82.2 ± 0.5%
BWT	-0.127	-0.104	-0.076	-0.067

10 shots. As Chameleon is the only silicon-proven work to perform end-to-end fully on-chip CL, Fig. 17 compares its CL performance (continuous lines) to the FSL performance of prior works [7], [8], [9] (single data points). In Table III, we also report the average accuracy (ACC, equivalent to final accuracy in Fig. 17) and backward transfer (BWT) metrics [47] for continually learning 250 classes using Chameleon. We omit the forward transfer (FWT) metric, as it is not informative because the PNs implemented in Chameleon do not support zero-shot learning [47]. The slightly negative BWT values indicate that learning a new task decreases the performance on a preceding task, indicating forgetting, which is consistent with the curves in Fig. 17. Notably, in the similar few-shot class-incremental learning (FSCIL) [48] scenario, where a DNN learns a set of classes before deployment and a new set of classes online, there is currently also no scalable end-to-end fully on-chip design. For example, Karunaratne et al. [15] propose a CIM macro for MANNs targeting FSCIL: their design supports embedding aggregation (summing) and uses a $\sim 4\times$ larger embedder than [7], enabling 100-way FSCIL with five shots per class. However, embeddings are still generated off-chip with an FP32 embedder and binarized before on-chip use, similar to [7] and [8]. Alternatively, Wibowo et al. [49] demonstrate FSCIL with the embeddings for PNs computed on-chip, using a microcontroller. However, they require an external L3 memory to store the 8-bit ($2\times$ larger than Chameleon) DNN weights and use cosine similarity as the distance function for PNs, requiring an expensive division operation. More recently, Song et al. [50]

proposed CLO-HDnn, building on the FSL-HDnn accelerator [9], to target CL using HDC. To reduce computations, the authors introduce a bypass mechanism to only activate their embedder network for difficult tasks. However, similar to FSL-HDnn [9], the design does not support fully end-to-end on-chip operation: only 0.14% of the required DNN weights can be stored on-chip at any time. Furthermore, CLO-HDnn only supports up to 128 classes, compared to 250 for Chameleon, even though the design's total on-chip memory and total silicon area are ~3× and ~20× larger than Chameleon, respectively. Beyond embedding-based learning, Huo et al. [26] demonstrate a first proof of concept for fully on-chip end-to-end FSCIL. The design uses a spiking NN with spike timing-dependent plasticity and lateral inhibition: it supports classification of nine types of gas using olfactory data and overcomes data disturbances caused by sensor drift. However, since it supports networks of only up to 4k 4-bit parameters, its suitability for more complex tasks, such as Omniglot, is unclear.

C. Real-Time KWS

To characterize both: 1) Chameleon's inference efficiency using its dual-mode multiplier-free PE array; and 2) its long-context modeling capabilities using the greedy dilation-aware execution of TCNs, we conduct a series of KWS experiments.

Benchmarking Setup—We use the Google Speech Commands V2 (GSCv2) dataset [51], which comprises 105 829 utterances from 2618 speakers across 35 spoken-word classes, with varying sample counts per class. Each sample is recorded for 1 s at 16 kHz. Ten words serve as command-like keywords: *Yes*, *No*, *Up*, *Down*, *Left*, *Right*, *On*, *Off*, *Stop*, and *Go*. Along with an *unknown command* class (grouping the remaining words) and a *silence* class (1-s audio clips from background noise files), they form the standard 12-way classification setup. We follow the same validation and testing splits as [51] and [52]. To augment the training data, we apply up to 100 ms forward or backward shifts (as in [52] and [53]), followed by noise addition with a probability of 0.15. Following FP training, we perform QAT, during which all augmentations are disabled. For PE array characterization, the audio is transformed into a 28-D MFCC feature map [54] before it is fed to the DNN. To align with other KWS digital accelerators [10], [11], [12], we use a window size of 32 ms with a 16 ms shift for MFCC-extraction, yielding a sequence of 63 timesteps. For long-context modeling, the raw audio data are used.

Dual-Mode PE Array Characterization—To minimize the static-power footprint for real-time MFCC-based KWS, we use a TCN with 16.5k parameters and eight layers, which is small enough to fit in the always-on memories of Chameleon. Running this TCN in the 4×4 PE array mode results in a real-time power consumption of 3.1 μW at a clock frequency of 23.3 kHz with a core voltage of 0.73 V, achieving a classification accuracy of 93.3% on the 12-class test set of GSCv2. Executing the same network in the complete 16×16 PE array requires a clock frequency of 3.67 kHz to yield a real-time power of 7.4 μW, with both the MSB memories and the core at 0.73 V. Fig. 18 displays the contributions of the core leakage, MSB memories leakage and dynamic power for both real-time KWS scenarios. It can be seen that executing MFCC-based KWS with only the always-on memories leads to a power reduction of 44% compared to the 16×16 baseline. Note also that the dynamic power in the 4×4 mode is higher compared to the 16×16 mode: since the throughput in the latter is 4× higher, the clock frequency can be downscaled accordingly to achieve the same latency.

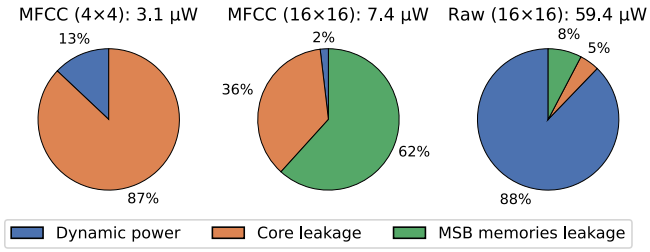


Fig. 18. Comparison of power contributions (at 0.73 V) during real-time KWS using MFCC feature vectors in both 4×4 and 16×16 mode, as well as raw audio in 16×16 mode.

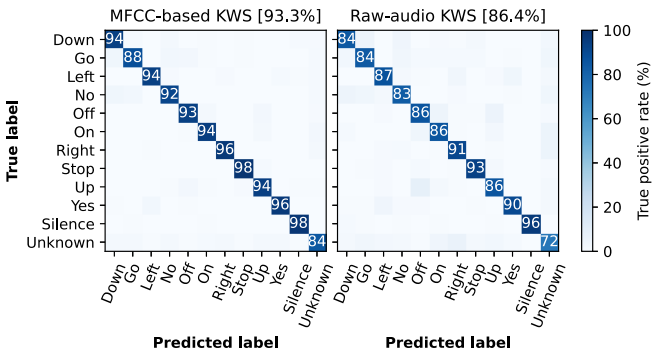


Fig. 19. Confusion matrices for MFCC-based KWS and raw audio KWS on the 12-class GSCv2 test set, including true positive rates for all keywords.

Long-Context Modeling Results—In most KWS accelerator designs, the footprint of the MFCC preprocessing exceeds that of the actual DNN inference logic. For example, in the work by Shan et al. [55], the MFCC extraction accounts for 60% of the core area, while in [14], the feature extractor is ~4× larger than the on-chip DNN accelerator. Similarly, the MFCC extraction module in [10] uses the same amount of area as the on-chip DNN accelerator (incurring a 2.4× area overhead when also considering the analog FE). Even fully digital designs like [56] allocate 40% of the core area to MFCC logic. To the best of our knowledge, there is currently only one design operating directly on the raw audio sequence without any specific preprocessing logic: the fully analog accelerator in [57] performs inference on 8-kHz audio by applying a trained filter bank to 4-ms input windows. However, several key DNN processing steps are performed off-chip, preventing end-to-end operation. In contrast, by enabling long-context inference, our approach eliminates the need for any data-specific pre-processing block, significantly increasing the number of possible deployment scenarios. Using a 24-layer, 118k-parameter model, Chameleon achieves 86.4%

TABLE IV

COMPARISON WITH BOTH KWS ACCELERATORS THAT HAVE REPORTED RESULTS ON GSC IN SILICON AND WITH FSL ACCELERATORS THAT HAVE REPORTED RESULTS ON OMNIGLOT IN SILICON. THE ± SHOWS 95% CONFIDENCE INTERVALS OVER 100 TASKS

	KWS accelerators				CIM accelerators for FSL		End-to-end FSL accelerators	
	Vocell [10] JSSC'18	TinyVers [12] JSSC'23	Tan *et al.* [57] JSSC'25	Park *et al.* [58] VLSI'24	Kim *et al.* [7] TCAS-II'22	SAPIENS [8] VLSI'21	FSL-HDnn [9] ESSERC'24	**This work**
Technology	65-nm	22-nm	28-nm	65-nm	40-nm LP	40-nm	40-nm	40-nm LP
Implementation	Digital	Digital	Analog	Mixed	Mixed signal	Analog	Digital	Digital
Core area (mm^2)	2.56	6.25	0.121	1.32	0.2	0.0367	11.3	0.83
On-chip memory	67 kB	132 kB	16 kB	5 kB	0.8 kB	8 kB	349 kB	71 kB
Supply voltage	0.6-1.2 V	0.4-0.9 V	0.35/0.9 V	0.4-1.2V V	0.5-1.1 V	0.85-1.1 V	0.9-1.2 V	0.6-1.1 V
Max. clock frequency	8 MHz	150 MHz	1 MHz	200 kHz	80 MHz	200 MHz	250 MHz	150 MHz
Network type	LSTM+FC	TCN	FS-CNN	TENet	Distance computation only, off-chip FP32 embedder		CNN	TCN+FCs
Max. # of on-chip weights	32k	400k	32.8k	7.4k			2.1k	133k
Max. demonstrated input length	62	60	8,000	46			28	**16,000**
Weight precision	4-bit LUT	8-bit	4-bit	4-bit $\log_2$			Bfloat-16	4-bit $\log_2$
Activation precision	8-bit	8-bit	3-bit unsigned	4-bit unsigned			16-bit[5]	4-bit unsigned
Inference support	✓	✓	✓	✓	✗ (off-chip embedder)		✓	✓
End-to-end inference	✓	✓	✗	✗	✗		✗	✓
Full on-chip weight storage	✓	✓	✓	✓	✗		✗	✓
FSL support	✗	✗	✗	✗	✓		✓	✓
End-to-end FSL	✗	✗	✗	✗	✗		✗	✓
CL support	✗	✗	✗	✗	✗		✗	✓
FSL method / distance metric	Do not support FSL				MANNs, L1	MANNs, L1	HDC, Hamming	PNs, L2[2]
FSL embedding dimension					64	32	16-1024	**1-1024**
# of FSL classes					1-25	32	2-128	**1-256**
# of FSL shots					1-5	1	1, 5[5]	**1-128**
Omniglot FSL/CL	Do not support FSL						(@ 100 MHz)	(@ 100 kHz) (@ 100 MHz)
Model size					7.46 MB	447 kB	5.5 MB[5]	**59 kB**
End-to-end power					N/A	N/A	27 mW	**12.9 µW** **11.6 mW**
End-to-end latency					N/A	N/A	53 ms	0.54s **0.59 ms**
Omniglot FSL accuracy	Do not support FSL							
5-way (1, 5 shots)					93.4%, 98.3%	-	79.0%, -	**96.8 ± 1.6%, 98.8 ± 0.5%**
20-way (1, 5-shots)					-	-	- , 79.5%	**89.1 ± 1.3%, 96.1 ± 0.5%**
32-way, 1-shot					-	72%	-	**83.3 ± 1.2%**
Omniglot 250-way CL acc.	Do not support continual learning							
1-shot (final, avg.)								**57.6 ± 1.0%, 70.3%**
2-shot (final, avg.)								**69.2 ± 0.6%, 80.2%**
5-shot (final, avg.)								**79.3 ± 0.7%, 87.1%**
10-shot (final, avg.)								**82.2 ± 0.4%, 89.0%**
GSC 12-class KWS	(@ 250 kHz)	(@ 5 MHz)	(@ 1 MHz)	(@ 200 kHz)	Do not support inference on temporal data			(@ 23.3 kHz) (@ 532 kHz)
Pre-processing	MFCC	MFCC	None	MFCC				MFCC None
Accuracy @ dataset version	90.87% (v1)	93.3% (v1)	91.8% (v2)	92.7%				**93.3%** (v2) 86.4% (v2)
Latency	16 ms	11 ms	2 ms	10.7 ms				16 ms 63 µs
Real-time power	10.6 µW[1]	193 µW	**1.73 µW**	4.68 µW[6]				3.1 µW 59.4 µW
Model size	16 kB	23 kB	11 kB	**3.7 kB**				8.5 kB 60 kB
Peak GOPS	0.13	17.6	N/C	0.0144	N/A[3]	N/A[4]	**154**	76.8
Peak TOPS/W	0.45[5]	17	N/C	3.1			5.7	**6.0**

[1] Excludes on-chip MFCC computation power. [2] Interpolated metrics. [3] Reported metrics are for distance computation only and amount to 4.85[5] GOPS and 27.7 TOPS/W. [4] Reported metrics are for distance computation only and amount to 0.4 GOPS[5] and 0.118 TOPS/W. [5] Estimated metrics. [6] Excludes analog front-end power.

test accuracy on GSCv2 (see Fig. 19) from 16-kHz raw audio with 59.4-μW real-time power at a clock frequency of 532 kHz and a core/MSB memory voltage of 0.73 V (Fig. 18, right). Although accuracy drops by 7 points, reflecting the challenge of capturing long-range dependencies, it remains competitive, outperforming [14] with 1.6 mm^2 analog MFCC at 86.0% 12-class accuracy.

D. Comparison With the SotA

Chameleon is the first SoC supporting both inference and end-to-end on-chip few-shot and continual learning: Table IV compares our design with both KWS and FSL accelerators that have silicon results on GSC and Omniglot datasets, respectively. This comparison highlights the performance of Chameleon across FSL and CL on-device learning, as well as KWS inference scenarios.

FSL Performance—Chameleon is the only end-to-end FSL chip that supports sequential data, while offering fully flexible support for embedding dimensions, number of ways, and shots. Chameleon is also the first work that not only enables multiplier-free inference but also completely multiplier-free FSL, without requiring any modifications of the inference datapath. Furthermore, across all FSL evaluation scenarios, Chameleon sets a new record, despite using an embedding DNN that is 7.6–126× smaller than other FSL chips. Compared with CIM accelerators for FSL distance computation, Kim et al. [7] implement MANNs (Section II-B) with an off-chip FP32 embedder of 7.46 MB, resulting in 3.4% and 0.5% lower accuracy than Chameleon for the five-way one- and 5-shot cases, respectively. SAPIENS [8] employs an RRAM-based backend to perform FSL, also using MANNs. On 32-way one-shot Omniglot, SAPIENS achieves 72% accuracy using an FP32-based four-layer CNN of 447 KB, which is 11.3% lower than Chameleon using an 8× larger network. To the best of our knowledge, the only other FSL accelerator with an on-chip embedder reporting metrics on Omniglot is FSL-HDnn [9], which uses HDC for FSL. It employs a Bfloat-16-based 5.5-MB ResNet-18 model for generating embeddings, achieving 79% accuracy for five-way one-shot and 79.5% for 20-way 5-shot tasks on Omniglot. These accuracies are significantly lower than Chameleon, which uses a network ~100× smaller that fits fully on-chip. Indeed, even though FSL-HDnn has a core area 12× larger than Chameleon, only 0.07% of its DNN weights can be stored on-chip at any time. Furthermore, while FSL-HDnn uses distance-based

classification, similar to Chameleon, its HDC encoding adds 120% power and 26% area overhead. Chameleon fully sidesteps this costly HDC encoding by computing distances directly on the embeddings. Overall, Chameleon uses 2.5× lower power than FSL-HDnn at the same clock frequency, while reducing the end-to-end latency by 90×, yielding an energy per shot 210× lower.

CL Performance—Chameleon is the first design to offer scalable end-to-end CL fully on-chip, establishing the first baseline for 250-way CL. Its unified learning and inference architecture, which enables a minimum CL power consumption of 12.9 μW, allows limiting the memory overhead of CL to scale with only 26 bytes per way on Omniglot [see (8)]. This is a negligible 0.04% of the total DNN footprint. Hence, by reusing its DNN weight memory to store FC parameters for new classes, Chameleon flexibly repurposes on-chip memory to scale beyond prior class limits: even with 90% of memory allocated to the deployed TCN, it can still accommodate 250 learned Omniglot classes. In contrast, FSL CIMs like Kim et al. [7] and SAPIENS [8] are limited to 25 and 32 classes, respectively, due to their fixed array sizes. FSL-HDnn [9] also uses a fixed separate memory to store its high-dimensional encodings, which is 80% larger than all on-chip memory in Chameleon. Yet, only up to 20-way learning is demonstrated, scaling with 8.2 KB/way on Omniglot.

Real-Time KWS Performance—Chameleon matches SotA 93.3% accuracy on GSC using MFCCs, with the lowest reported real-time end-to-end power (3.1 μW) and the highest peak throughput (76.8 GOPS) among end-to-end KWS inference accelerators. Compared to Vocell [10], Chameleon offers 2.5% higher accuracy with 2× lower power and 600× higher peak GOPS. Versus TinyVers [12], which uses a TCN on a RISC-V SoC with a reconfigurable accelerator, Chameleon matches accuracy with a 3× smaller model and 64× lower power. Tan et al. [57] avoid MFCC preprocessing to infer on raw audio instead: using a fully analog design, they improve raw-audio performance by 5.4 accuracy points at 1.8× lower power, with an estimated 25× higher peak efficiency compared to Chameleon. However, it does not perform end-to-end on-chip inference as several key steps of inference computation (e.g., ReLU activation, BN, stride adjustments, max pooling, activation clipping) are carried out off-chip. The design by Park et al. [58], on the other hand, does perform end-to-end KWS using the smallest total SRAM footprint across the compared designs of only 5 kB. However, the design's real-time power consumption is about 50% higher at a 0.6%-point reduction in accuracy compared to Chameleon, while also relying on an embedded MFCC feature extractor for inference, indicating the inability for the neural network to operate directly on the raw data.

V. Conclusion

In this work, we presented Chameleon, the first scalable SoC to enable end-to-end on-chip FSL and few-shot CL on sequential data, without compromising inference accuracy or efficiency. We achieve this through three key contributions.

First, by integrating FSL and CL into the inference process, we propose a unified learning and inference architecture built on PNs [16]. The learning integration incurs only <0.02% latency and 0.5% core area overhead. Second, our greedy dilation-aware TCN execution enables the first 16-kHz raw audio KWS on GSCv2, achieving a 160× larger receptive field than prior end-to-end KWS accelerators using only 2-kB activation memory—4× less than SotA TCN accelerators, while avoiding a 60%-400% area increase for audio-specific preprocessing blocks. Third, by using $\log_2$ quantization, we enable a multiplier-free PE array. Combined with its dual-mode operation through system-level power gating, this yields a peak throughput in 16 × 16 mode that is 4.3× higher than previous KWS accelerators or 2× lower real-time power in 4 × 4 mode.

Chameleon establishes new accuracy records for end-to-end on-chip FSL on the Omniglot dataset (96.8% ± 1.6% five-way one-shot and 96.1% ± 0.5% 20-way 5-shot) at 2.5× lower power than previous on-chip embedder FSL work [9]. It is the only end-to-end FSL accelerator with all weights on-chip and enables FSL starting at 12.9 μW. Considering CL, although several designs exist, none offer a fully end-to-end SoC solution with comparable accuracy, footprint, and scalability. Hence, Chameleon establishes the first baseline for end-to-end on-chip CL on Omniglot, demonstrating 250-way few-shot CL (82.2% final accuracy for ten shots). Beyond this learning performance, Chameleon maintains a SotA accuracy of 93.3% for 12-class GSCv2 compared to inference-only KWS accelerators.

Ultimately, by enabling: 1) a unified learning and inference architecture; 2) a flexible PE-array architecture that allows multiplier-free learning with low-resolution quantization levels that are typical of inference-only accelerators; and 3) an efficient TCN embedder that allows scaling this approach to long-range temporal dependencies up to raw-audio processing, Chameleon paves the way for efficient lifelong learning on extreme-edge devices.

Acknowledgment

The authors would like to thank Prof. Kofi Makinwa, Nicolas Chauvaux, Dr. Martin Lefebvre, and Dr. Adrian Kneip for their feedback and Dr. Filipe Cardoso for dicing and Nuriel Rozsa for die photographs.

References

[1] N. S. Murthy, P. Vrancx, N. Laubeuf, P. Debacker, F. Catthoor, and M. Verhelst, "Learn to learn on chip: Hardware-aware meta-learning for quantized few-shot learning at the edge," in *Proc. IEEE/ACM 7th Symp. Edge Comput. (SEC)*, Dec. 2022, pp. 14–25.

[2] L. Ravaglia, M. Rusci, D. Nadalini, A. Capotondi, F. Conti, and L. Benini, "A TinyML platform for on-device continual learning with quantized latent replays," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 11, no. 4, pp. 789–802, Dec. 2021.

[3] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in AI safety," 2016, *arXiv:1606.06565*.

[4] L. Pellegrini, G. Graffieti, V. Lomonaco, and D. Maltoni, "Latent replay for real-time continual learning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2020, pp. 10203–10209.

[5] M. Li and X. S. Hu, "A quantization framework for neural network adaption at the edge," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 402–407.

[6] M. Wang, R. Xue, J. Lin, and Z. Wang, "Exploring quantization in few-shot learning," in *Proc. 18th IEEE Int. New Circuits Syst. Conf. (NEWCAS)*, Jun. 2020, pp. 279–282.

[7] S. Kim, W. Lee, S. Kim, S. Park, and D. Jeon, "An in-memory computing SRAM macro for memory-augmented neural network," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 3, pp. 1687–1691, Mar. 2022.

[8] H. Li et al., "One-shot learning with memory-augmented neural networks using a 64-kbit, 118 GOPS/W RRAM-based non-volatile associative memory," in *Proc. Symp. VLSI Technol.*, Jun. 2021, pp. 1–2.

[9] H. Yang et al., "FSL-HDnn: A 5.7 TOPS/W end-to-end few-shot learning classifier accelerator with feature extraction and hyperdimensional computing," in *Proc. IEEE Eur. Solid-State Electron. Res. Conf. (ESSERC)*, Sep. 2024, pp. 33–36.

[10] J. S. P. Giraldo, S. Lauwereins, K. Badami, and M. Verhelst, "Vocell: A 65-nm speech-triggered wake-up SoC for 10-μW keyword spotting and speaker verification," *IEEE J. Solid-State Circuits*, vol. 55, no. 4, pp. 868–878, Apr. 2020.

[11] J. S. P. Giraldo, V. Jain, and M. Verhelst, "Efficient execution of temporal convolutional networks for embedded keyword spotting," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 12, pp. 2220–2228, Dec. 2021.

[12] V. Jain, S. Giraldo, J. D. Roose, L. Mei, B. Boons, and M. Verhelst, "TinyVers: A tiny versatile system-on-chip with state-retentive eMRAM for ML inference at the extreme edge," *IEEE J. Solid-State Circuits*, vol. 58, no. 8, pp. 2360–2371, Aug. 2023.

[13] P. P. Bernardo, C. Gerum, A. Frischknecht, K. Lübeck, and O. Bringmann, "UltraTrail: A configurable ultralow-power TC-ResNet AI accelerator for efficient keyword spotting," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 4240–4251, Nov. 2020.

[14] K. Kim et al., "A 23-μW keyword spotting IC with ring-oscillator-based time-domain feature extraction," *IEEE J. Solid-State Circuits*, vol. 57, no. 11, pp. 3298–3311, Nov. 2022.

[15] G. Karunaratne et al., "In-memory realization of in-situ few-shot continual learning with a dynamically evolving explicit memory," in *Proc. IEEE 48th Eur. Solid State Circuits Conf. (ESSCIRC)*, Sep. 2022, pp. 105–108.

[16] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–7. [Online]. Available: https://proceedings.neurips.cc/paperfiles/paper/2017/file/cb8da6767461f2812ae4290eac7cbc42-Paper.pdf

[17] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, Dec. 2015. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aab3050

[18] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," 2018, *arXiv:1803.01271*.

[19] M. Scherer, A. D. Mauro, T. Fischer, G. Rutishauser, and L. Benini, "TCN-CUTIE: A 1,036-TOp/s/W, 2.72-μJ/inference, 12.2-mW all-digital ternary accelerator in 22-nm FDX technology," *IEEE Micro*, vol. 43, no. 1, pp. 42–48, Jan. 2023.

[20] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proc. 34th Int. Conf. Mach. Learn.*, vol. 70, Aug. 2017, pp. 1126–1135.

[21] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, "Meta-learning in neural networks: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 9, pp. 5149–5169, Sep. 2022.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[23] N. Mishra, M. Rohaninejad, X. Chen, and P. Abbeel, "A simple neural attentive meta-learner," 2017, *arXiv:1707.03141*.

[24] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap, "Meta-learning with memory-augmented neural networks," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1842–1850. [Online]. Available: https://proceedings.mlr.press/v48/santoro16.html

[25] J. Morris, R. Fernando, Y. Hao, M. Imani, B. Aksanli, and T. Rosing, "Locality-based encoder and model quantization for efficient hyperdimensional computing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 4, pp. 897–907, Apr. 2022.

[26] D. Huo et al., "ANP-G: A 28-nm 1.04-pJ/SOP sub-mm^2 asynchronous hybrid neural network olfactory processor enabling few-shot class-incremental on-chip learning," *IEEE J. Solid-State Circuits*, vol. 60, no. 7, pp. 2660–2670, Jul. 2025.

[27] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 5998–6008.

[28] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[29] Z. Wang, Y. Ma, Z. Liu, and J. Tang, "R-transformer: Recurrent neural network enhanced transformer," 2019, *arXiv:1907.05572*.

[30] J. Elman, "Finding structure in time," *Cognit. Sci.*, vol. 14, no. 2, pp. 179–211, Jun. 1990.

[31] K. Cho, B. Van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," 2014, *arXiv:1409.1259*.

[32] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[33] Q. V. Le, N. Jaitly, and G. E. Hinton, "A simple way to initialize recurrent networks of rectified linear units," 2015, *arXiv:1504.00941*.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015, *arXiv:1512.03385*.

[35] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016, *arXiv:1603.01025*.

[36] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015, *arXiv:1502.03167*.

[37] P. Goyal, "Accurate, large minibatch SGD: Training ImageNet in 1 hour," 2017, *arXiv:1706.02677*.

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1026–1034.

[39] A. Pappalardo, "Xilinx/brevitas," Xilinx, Tech. Rep., 2023, doi: 10.5281/zenodo.3333552. [Online]. Available: https://github.com/Xilinx/brevitas/

[40] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018, *arXiv:1806.08342*.

[41] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

[42] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," 2021, *arXiv:2106.08295*.

[43] O. Vinyals, C. Blundell, T. Lillicrap, K. Kavukcuoglu, and D. Wierstra, "Matching networks for one shot learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 3637–3645. [Online]. Available: https://proceedings.neurips.cc/paperfiles/paper/2016/file/90e1357833654983612fb05e3ec9148c-Paper.pdf

[44] K. Prabhu et al., "CHIMERA: A 0.92-TOPS, 2.2-TOPS/W edge AI accelerator with 2-MByte on-chip foundry resistive RAM for efficient training and inference," *IEEE J. Solid-State Circuits*, vol. 57, no. 4, pp. 1013–1026, Apr. 2022.

[45] J. Park, S. Lee, and D. Jeon, "9.3 A 40nm 4.81TFLOPS/W 8b floating-point training processor for non-sparse neural networks using shared exponent bias and 24-way fused multiply-add tree," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, vol. 64, Feb. 2021, pp. 1–3.

[46] C. Cioflan, L. Cavigelli, and L. Benini, "Boosting keyword spotting through on-device learnable user speech characteristics," 2024, *arXiv:2403.07802*.

[47] D. Lopez-Paz and M. Ranzato, "Gradient episodic memory for continual learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 6467–6476.

[48] X. Tao, X. Hong, X. Chang, S. Dong, X. Wei, and Y. Gong, "Few-shot class-incremental learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 12180–12189.

[49] Y. E. Wibowo et al., "12 mJ per class on-device online few-shot class-incremental learning," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Benin, Mar. 2024, pp. 1–6.

[50] C. E. Song et al., "Clo-HDnn: A 4.66 TFLOPS/W and 3.78 TOPS/W continual on-device learning accelerator with energy-efficient hyperdimensional computing via progressive search," in *Proc. Symp. VLSI Technol. Circuits (VLSI Technol. Circuits)*, Jun. 2025, pp. 1–3.

[51] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018, *arXiv:1804.03209*.

[52] O. Rybakov, N. Kononenko, N. Subrahmanya, M. Visontai, and S. Laurenzo, "Streaming keyword spotting on mobile devices," 2020, *arXiv:2005.06720*.

[53] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," 2017, *arXiv:1711.07128*.

[54] S. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-28, no. 4, pp. 357–366, Aug. 1980.

[55] W. Shan et al., "A 510-nW wake-up keyword-spotting chip using serial-FFT-based MFCC and binarized depthwise separable CNN in 28-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 56, no. 1, pp. 151–164, Jan. 2021.

[56] B. Liu et al., "A 22nm, 10.8 μW/15.1 μW dual computing modes high power-performance-area efficiency domained background noise aware keyword-spotting processor," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 12, pp. 4733–4746, Dec. 2020.

[57] F. Tan, W.-H. Yu, J. Lin, K.-F. Un, R. P. Martins, and P.-I. Mak, "A 1.8% FAR, 2 ms decision latency, 1.73 nJ/decision keywords-spotting (KWS) chip incorporating transfer-computing speaker verification, hybrid-IF-domain computing and scalable 5T-SRAM," *IEEE J. Solid-State Circuits*, vol. 60, no. 3, pp. 1103–1112, Mar. 2025.

[58] S. Park et al., "A 5.6μW 10-keyword end-to-end keyword spotting system using passive-averaging SAR ADC and sign-exponent-only layer fusion with 92.7% accuracy," in *Proc. IEEE Symp. VLSI Technol. Circuits (VLSI Technol. Circuits)*, Jun. 2024, pp. 1–2.



Douwe den Blanken (Graduate Student Member, IEEE) received the M.Sc. degree (Hons.) in embedded systems from Delft University of Technology (TU Delft), Delft, The Netherlands, in 2023, where he is currently pursuing the Ph.D. degree, under the supervision of Prof. C. Frenkel.

His current research interests include efficient learning algorithms and their implementation in silicon, as well as the quantization and acceleration of modern DNNs.



Charlotte Frenkel (Member, IEEE) received the M.Sc. degree (summa cum laude) in electromechanical engineering and the Ph.D. degree in engineering science from Université catholique de Louvain (UCLouvain), Louvain-la-Neuve, Belgium, in 2015 and 2020, respectively.

In February 2020, she joined the Institute of Neuroinformatics, UZH, and ETH Zürich, Zürich, Switzerland, as a Post-Doctoral Researcher. She is an Assistant Professor at Delft University of Technology, Delft, The Netherlands, since July 2022, and holds a Visiting Faculty Researcher position with Google since October 2024. Her research aims at bridging the bottom–up (bio-inspired) and top–down (engineering-driven) design approaches toward neuromorphic intelligence, with a focus on hardware-algorithm co-design for (Neuro)AI, digital hardware accelerators, and brain-inspired on-device learning.

Dr. Frenkel received the Best Paper Award at the IEEE International Symposium on Circuits and Systems (ISCAS) 2020 conference in the *Neural Networks* track, and her Ph.D. thesis was awarded the FNRS-FWO/Nokia Bell Scientific Award 2021 and the FNRS-FWO/IBM Innovation Award 2021. In 2023, she was awarded prestigious Veni and AiNed Fellowship grants from Dutch Research Council (NWO). She presented several invited talks, including keynotes at the tinyML EMEA technical forum 2021 and at the Neuro-Inspired Computational Elements (NICE) neuromorphic conference 2021. She serves or has served as the Program Co-Chair for NICE 2023–2024 and of the tinyML Research Symposium 2024, as a Co-Lead of the NeuroBench initiative for benchmarks in neuromorphic computing since 2022, as a TPC member of IEEE ESSERC for 2022–2024, and as an Associate Editor for IEEE TRANSACTIONS ON BIOMEDICAL CIRCUITS AND SYSTEMS since 2022.