A TinyML Platform for On-Device Continual Learning With Quantized Latent Replays

Leonardo Ravaglia, Manuele Rusci, Davide Nadalini, Alessandro Capotondi, *Member, IEEE*, Francesco Conti, *Member, IEEE*, and Luca Benini, *Fellow, IEEE*

***Abstract*— In the last few years, research and development on Deep Learning models & techniques for ultra-low-power devices – in a word, *TinyML* – has mainly focused on a *train-then-deploy* assumption, with static models that cannot be adapted to newly collected data without cloud-based data collection and fine-tuning. Latent Replay-based Continual Learning (CL) techniques (Pellegrini *et al.*, 2020) enable online, serverless adaptation in principle, but so far they have still been too computation- and memory-hungry for ultra-low-power TinyML devices, which are typically based on microcontrollers. In this work, we introduce a HW/SW platform for end-to-end CL based on a 10-core *FP32*-enabled parallel ultra-low-power (PULP) processor. We rethink the baseline Latent Replay CL algorithm, leveraging quantization of the frozen stage of the model and Latent Replays (LRs) to reduce their memory cost with minimal impact on accuracy. In particular, 8-bit compression of the LR memory proves to be almost lossless (-0.26% with 3000LR) compared to the full-precision baseline implementation, but requires 4× less memory, while 7-bit can also be used with an additional minimal accuracy degradation (up to 5%). We also introduce optimized primitives for forward and backward propagation on the PULP processor, together with data tiling strategies to fully exploit its memory hierarchy, while maximizing efficiency. Our results show that by combining these techniques, continual learning can be achieved in practice using less than 64MB of memory – an amount compatible with embedding in TinyML devices. On an advanced 22nm prototype of our platform, called *VEGA*, the proposed solution performs on average 65× faster than a low-power STM32 L4 microcontroller, being 37× more energy efficient – enough for a lifetime of 535h when learning a new mini-batch of data once every minute.**

Manuscript received May 15, 2021; revised September 12, 2021; accepted October 4, 2021. Date of publication October 19, 2021; date of current version December 13, 2021. This work was supported in part by the ECSEL Horizon 2020 Project Artificial intelligence for Digital Industry (AI4DI) under Grant 826060, in part by the EU Horizon 2020 Project BonsAPPs under Grant 101015848, and in part by CINECA through the ISCRA Initiative under Project NAS4NPC. This article was recommended by Guest Editor I. Partin-Vaisband. *(Corresponding author: Leonardo Ravaglia.)*

Leonardo Ravaglia, Manuele Rusci, Davide Nadalini, and Francesco Conti are with the Department of Electrical, Electronic and Information Engineering (DEI), University of Bologna, 40136 Bologna, Italy (e-mail: leonardo.ravaglia2@unibo.it; manuele.rusci@unibo.it; d.nadalini@unibo.it; f.conti@unibo.it).

Alessandro Capotondi is with the Department of Physics, Informatics and Mathematics, University of Modena and Reggio Emilia, 41125 Modena, Italy (e-mail: alessandro.capotondi@unibo.it).

Luca Benini is with the Department of Electrical, Electronic and Information Engineering (DEI), University of Bologna, 40136 Bologna, Italy, and also with the Integrated Systems Laboratory (IIS), ETH Zürich, 8092 Zürich, Switzerland (e-mail: lbenini@iis.ee.ethz.ch).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JETCAS.2021.3121554.

Digital Object Identifier 10.1109/JETCAS.2021.3121554

***Index Terms*— TinyML, continual learning, deep neural networks, parallel ultra-low-power, microcontrollers.**

I. Introduction

THE internet-of-Things ecosystem is made possible by miniaturized and smart end-node devices, which can sense the surrounding environment and take decisions based on the information inferred from sensor data. Because of their tiny form factor and the requirement for low cost and battery-operated nature, these smart networked devices are severely constrained in terms of memory capacity and maximum performance and use small Microcontroller Units (MCUs) as their main on-board computing device [2]. At the same time, there is an ever-growing interest in deploying more accurate and sophisticated data analytics pipelines, such as Deep Learning (DL) inference models, directly on IoT end-nodes. These competing needs have given rise in the last few years to a specific branch of machine learning (ML) and DL research called *TinyML* [3] – focused on shrinking and compressing top-accurate DL models with respect to the target device characteristics.

The primary limitation of the current generation of TinyML hardware and software is that it is mostly focused on *inference*. The inference task can be strongly optimized by quantizing [4] or pruning [5] the trained model. Many vendors of AI-oriented system-on-chips (SoCs) provide deployment frameworks to automatically translate DL inference graphs into human-readable or machine code [6]. This *train-then-deploy* design process rigidly separates the learning phase from the runtime inference, resulting in a *static* intelligence model design flow, incapable of adapting to phenomena such as *data distribution shift*: a shift in the statistical properties of real incoming data vs. the training set that often impacts applications, causing the smart sensors platform to be unreliable when deployed in the field [7].

Even if the algorithms themselves are technically capable to learn and adapt to new incoming data, the update process can only be handled from a centralized service, running on the cloud or host servers [8]. In this regard, the original training dataset would have to be enriched with the newly collected dataset, and the model would have to be retrained from scratch on the enlarged dataset, adapting to the new data without forgetting the original information [8]. Such an adaptive mechanism belongs to the *rehearsal* category and requires the storage of the full training set, often amounting to gigabytes of data. Additionally, large amounts of data

have to be collected in a centralized fashion by network communication, resulting in potential security and privacy concerns, as well as issues of radio power consumption and network reliability in non-urban areas.

We argue that a robust and privacy-aware solution to these challenges is enabling future smart IoT end-nodes to Life-long Learning, also known as *Continual Learning* [9](CL): the capability to autonomously adapt to the ever-changing surrounding environment by learning continually (only) from incoming data without forgetting the original knowledge – a phenomenon known as *catastrophic forgetting* [10]. Despite many approaches exists to learn from data [11], recently the focus has moved to improve the recognition accuracy of DL models because of their superior capabilities, accounting on new data belonging to known classes (*domain-incremental CL*) or a new classes (*class-incremental CL*) [12], [13]. The CL techniques recently proposed are grouped in three categories: architectural, regularization and memory (or rehearsal) strategies. The architectural approaches specialize a subset of parameters for every (new and old) task but require the task-ID information at inference time, indicating the nature of current task in a multi-head network, and therefore they are not suitable for class or domain incremental continual learning. Concerning these latter scenarios, memory-based approaches, which preserve samples from previous tasks for replaying, perform better than regularization techniques, which simply address catastrophic forgetting by imposing constraints on the network parameter update at low memory cost [13]–[15]. This finding was confirmed during the recent CL competition at CVPR2020 [16], where the best entry leveraged on rehearsal based strategies.

The main drawback of memory-based CL approaches concerns the high memory overhead for the storage of previous samples: the memory requirement can potentially grows over time preventing the applicability of these methods at the tiny scale, e.g. [17]. To address this problem, Pellegrini *et al.* [1] have recently introduced Continual Learning based on *Latent Replays* (LRs). The idea behind this is to combine a few *old* data points taken from the original training set, but encoded into a low-dimensional latent space to reduce the memory cost, with the new data for the incremental learning tasks. Hence, the previous knowledge is retained by means of Latent Replays samples, i.e. the intermediate feature maps of the DL model inference, selected so that they require less space with respect to the input data (up to 48× smaller compared to raw images [1]). This strategy also leads to reduced computational cost: the Latent intermediate layer splits the network in a *frozen* stage at the front and an *adaptive* stage at the back, and only layers in the latter need to be updated. So far, LR-based Continual Learning has been successfully prototyped on high-performance embedded devices such as smartphones, including a Snapdragon-845 CPU running Android OS in the power envelope of a few Watts.[1] On the contrary, in this work, we focus on IoT applications and TinyML devices, with 100× tighter power constraints and 1000× smaller memories available.

In our preliminary work [18], we proposed the early design concept of a HW/SW platform for Continual Learning based on the Parallel Ultra Low Power (PULP) paradigm [19], and assessed the computational and memory costs to deploy Latent Replay-based CL algorithms.

In this paper, we complete and extend that effort by introducing several novel contributions from the software stack, system integration and algorithm viewpoint. To the best of our knowledge, we present the first TinyML processing platform and framework capable of on-device CL, together with the design flow required to sustain learning tasks within a few tens of mW of power envelope (> 10× lower than state-of-the-art solutions). The proposed platform is based on *VEGA*, a recently introduced end-node System-on-Chip prototype fabricated in 22nm technology [20]. Unlike traditional low-power and flexible MCUs design, VEGA exploits explicit data parallelism, by featuring a multi-core SW programmable RISC-V cluster with shared Floating Point Units (FPUs), DSP-oriented ISA and optimized memory management to enable the learning paradigm on low-end IoT devices. Additionally, to gain minimum-cost on-device retention of Latent Replays and better enable deployment on an ultra-low-power platform, we extend the LR algorithm proposed by Pellegrini et al. [1] to work with a fully quantized *frozen* front-end and compress Latent Replays using quantization down to 7 bits, with a small accuracy drop (almost lossless for 8-bit) when compared to the single-precision floating-point datatype (*FP32*) on the Core50 CL classification benchmark.

In summary, the contributions of this work are:

1) We extend the LR algorithm to work with an 8-bit quantized and frozen front-end without impact on the CL process and to support LR compression with quantization, reducing up to 4.5× the memory needed for rehearsing. We call this extension *Quantized Latent Replay-based Continual Learning* or *QLR-CL*.
2) We propose a set of CL primitives including forward and backward propagation of common layers such as convolution, depthwise convolution, and fully connected layers, fine-tuned for optimized execution on VEGA, a TinyML platform for Deep Learning based on PULP [19], fabricated in 22nm technology. We also introduce a tiling scheme to manage data movement for the CL primitives.
3) We compare the performance of our CL primitives on VEGA with that on other devices that could in the future target on-chip at-edge learning, such as a state-of-the-art low-power STM32L4 microcontroller.

Our results show that the Quantized Latent Replay based Continual Learning lead to a minimal accuracy loss on the Core50 dataset compared to the *FP32* baseline, when compressing the Latent Replay memory by 4× by means of 8-bit quantization. Compression to 7 bit can also be exploited but at the cost of a slightly lower accuracy, up to 5% wrt the baseline when retraining one of the intermediate layer. When testing the QLR-CL pipeline on the proposed VEGA platform, our CL primitives demonstrated to run up to 65× faster with respect to the MCUs for TinyML that can be found currently on the market. Compared against edge devices with a power envelope of 4W our solution is about 6× more energy-efficient, enough to operate 317h with a typical battery for embedded devices. The rest of the paper is organized as follows: Section II discusses related work in CL, inference and learning at the

[1]https://hothardware.com/reviews/qualcomm-snapdragon-845-performance-benchmarks

edge, and hardware architectures targeted at edge learning. Section III introduces the proposed methodology for Quantized Continual Learning. Section IV describes the HW/SW architecture of the proposed TinyML. Section V evaluates and discusses experimental results. Section VI concludes the paper.

II. Related Work

In this section, we first review the recent memory-efficient Continual Learning approaches before discussing the main solutions and methods for the TinyML ecosystem, including the first attempts for on-device learning on embedded systems.

A. Memory-Efficient Continual Learning

Differently from *Transfer Learning* [21], [22], which by design does not retain the knowledge of the primitive learned task when learning a new one, *Continual Learning* (CL) has recently emerged as a new technique to tackle the acquisition of new/extended capabilities without losing the original ones – a phenomenon known as *catastrophic forgetting* [12], [13]. One of the main causes of this phenomenon is that the newly acquired set breaks one of the main assumptions underlying supervised learning – i.e., that training data are statistically independent and identically distributed (IID). Instead, CL deals with training data that is organized in non-IID *learning events*. Maltoni *et al.* in [26] sort the main CL techniques into three groups: *rehearsal*, which includes a periodic replay of the past information; *architectural*, relying on a specialized architecture, layers, and activation functions to mitigate forgetting; and *regularization*-based, where the loss term is extended to encourage retaining memory of pre-learned tasks.

Among these groups, *rehearsal* CL strategies have emerged as the most effective to deal with catastrophic forgetting, at the cost of an additional replay memory [1], [27], [28]. In the recent CL challenge at CVPR2020 on the Core50 image dataset, $\sim$90% of the competitors used rehearsal strategies [16]. The best entry of the more challenging New Instances and Classes track (the same scenario considered in our work) [17], which is evaluated in terms of test accuracy but also memory and computation requirements, scores 91% by replaying image data. Unfortunately, this strategy results untractable for an IoT platform because of the expanding replay memory (up to 78k images) and the usage of a large DenseNet-161 model. Conversely, the Latent Replay-based approach [1] relies on a fixed, and relatively small, amount of compressed *latent* activations as replay data; it scores 71% if retraining only the last layer, which presents a peak of 52$\times$ lower (compressed) data points than the winning solution. Additionally, the *Jodelet* entry – also employing LR-based CL – achieves 83% thanks to 3$\times$ more replays and a more accurate pre-trained model (ResNet50) [16]. In our work, we focus on [1] because of the tunable accuracy-memory setting. Nevertheless, our proposed platform and compression methodology can be applied to any replay-based CL approach.

Also related to our work, ExStream [29] clusters in a streaming fashion the training samples before pushing them into the replay buffer while [30] uses discrete autoencoders to compress the input data for rehearsing. In contrast, we propose low-bitwidth quantization to compress the Latent Replay memory by $>$4$\times$ and, at the same time, reduce the inference latency and the memory requirement of the inference task of the *frozen stage* if compared to a full-precision *FP32* implementation.

B. Deep Learning at the Extreme Edge

Two main trends can be identified for TinyML platforms targeting the extreme edge. On the one hand, Deep Learning applications are dominated by linear algebra which is an ideal target for application-specific HW acceleration [31], [32]. Most efforts in this direction employ a variety of inference-only acceleration techniques such as pruning [33] and byte and sub-byte integer quantization [4]; the use of large arrays of simple MAC units [34] or even mixed-signal techniques such as in-memory computing [35].

On the other hand, there are also many reasons for the alternative approach: running TinyML applications as software on top of commercial off-the-shelf (COTS) extreme-edge platforms, such as MCUs. Extreme-edge TinyML devices need to be very cheap; they have to be flexible due both to economy of scale and to their need for integration within larger applications, composed of both neural and non-neural tasks [36]. For these reasons, there is a strong push towards squeezing the maximal performance out of platforms based on COTS ARM Cortex-M class microcontrollers and DSPs, such as STMicroelectronics STM32 microcontrollers,[2] or on multi-core parallel ultra-low-power (PULP) end-nodes, like GreenWaves Technologies GAP-8.[3] To cope with the severe constraints in terms of memory and maximum compute throughput of these platforms, a large number of deployment tools have been recently proposed. Examples of this trend include non-vendor-locked tools such as Google TFLite Micro [6], ARM CMSIS-NN [37], Apache TVM [38], as well as frameworks that only support specific families of devices, such as STMicroelectronics X-CUBE-AI,[4] GreenWaves Technologies NNTOOL,[5] and DORY [39]. Internally, these tools employ hardware-independent techniques, such as post-training compression & quantization [40]–[42], as well as hardware-dependent ones such as data tiling [43] and loop unrolling to boost data reuse exploitation [37], coupled with automated generation of optimized backend code [44].

As previously discussed, all of these efforts are mostly targeted at extreme edge inference, with little hardware and/or software dedicated to training. Most of the techniques used to boost inference efficiency are not as effective for learning. For example, the vast majority of training is done in full precision floating-point (*FP32*) or, with some restrictions, using half-precision floats (*FP16*) [45] – whereas inference is commonly pushed to *INT8* or even below [4], [40]. IBM has recently proposed a specialized 8-bit format for training called *HFP8* [46], but its effectiveness is still under investigation.

Hardware-accelerated on-device learning has so far been limited to high-performance embedded platforms (e.g., NVIDIA TensorCores on Tegra Xavier[6] and mobile platforms such as Qualcomm Snapdragon 845 [1]) or very narrow in scope. For example, Shin *et al.* [47] claim to implement

[2]https://www.st.com/content/st_com/en/ecosystems/stm32-ann.html
[3]https://greenwaves-technologies.com/gap8_gap9/
[4]https://www.st.com/en/embedded-software/x-cube-ai.html
[5]https://greenwaves-technologies.com/sdk-manuals/nn_quick_start_guide
[6]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx

TABLE I
ON-DEVICE LEARNING METHODS ON TINY EMBEDDED SYSTEMS

Method	Learning Approach	Problem	Proc. Device	Tiny Device	On-Device Learning	Compute Cost	Memory Cost	Continual Learning
Transfer Learning [21]	Retraining last layer's weights	Image Classification	Coral Edge TPU		✓	LOW	LOW	
TinyTL [22]	Retraining Biases	Image Classification	EPYC AMD 7302		✓	MEDIUM	LOW / MEDIUM	
TinyOL [23]	Add layer for transfer-learning based on streaming data	Anomaly Detection	Arduino Nano 33 BLE	✓	✓	LOW	LOW	
TinyML Minicar [8]	CNN backprop. from scracth on increasing dataset	Linear Camera Class. 7 actions	GAP8	✓		-	-	✓
TML [24]	kNN Classifier	Audio/Image Class. 2 classes	STM32F7	✓	✓	LOW	HIGH (unbounded)	✓
PULP-HD [25]	Hyperdimensional Computing	EMG 10 gestures Classification	Mr. Wolf	✓	✓	MEDIUM	LOW	✓
LR-CL [1]	CNN backprop. w/ LRs	Image Class. 50 classes	Qualcomm Snapdragon		✓	HIGH	HIGH / MEDIUM	✓
QLR-CL [This Work]	CNN backprop. w/ Quantized LRs	Image Class. 50 classes	VEGA	✓	✓	HIGH	MEDIUM	✓

an online adaptable architecture, but this is done using a simple LUT to selectively activate parameters, and does not support more powerful mechanisms based on gradient descent. A few recently proposed hardware accelerators for low-power training platforms [48]–[51] enable partial gradient back-propagation by using selective and compressed weight updates, but they do not address the large memory footprint required by training. Finally, several online-learning devices using bio-inspired algorithms such as Spiking Neural Networks [52] and High-Dimensional Computing [25] have been proposed [53]–[55]. Most of these approaches, however, have only been demonstrated on simple MNIST-like tasks.

In this work, we propose the first, to the best of our knowledge, MCU-class hardware-software system capable of continual learning based on gradient back-propagation with a LR approach. We achieve these results by leveraging on few key ideas in the state-of-the-art: *INT8* inference, *FP32 continual* learning, and exploitation of linear algebra kernels, back-propagation, and aggressive parallelization by deploying them on a multi-core FPU-enhanced PULP cluster.

C. On-Device Learning on Low-End Platforms

Table I lists the main edge solutions featuring on-device learning capabilities. Every approach is evaluated by considering the memory and computational costs for the continual learning task and the suitability for deployment on highly resource-constrained (tiny) devices.

A first group of works deals with on-device transfer learning. The Coral Edge TPU, which presents a power budget of several Watts, features SW support for on-device fine-tuning of the parameters of the last fully-connected layer [21]. TinyTL [22] demonstrated on a high-end CPU that the transfer learning task results more effective (+32% on the target Image Classification task) by retraining the bias terms and adding lite residual modules. TinyOL [23] brought the transfer learning task on a tiny devices, i.e. an Arduino Nano platform featuring a 64MHz ARM Cortex-M4, by adding a trainable layer on top of a frozen inference model. Because only the coefficients of the last layer are updated during the online training process, no backpropagation of error gradients applies. Compared to these works, we address a continual learning scenario and therefore we provide a more capable and optimized HW/SW solution to match the memory and computational requirements of the adopted CL method.

Differently from the above works, Prado *et al.* [8] proposed a Continual Learning framework for self-driving mini-cars. The embedded PULP-based MCU engine streams new data to a remote server, where the inference model is retrained from scratch on the enhanced dataset to improve the accuracy over time. This *fully-rehearsal* methodology cannot be migrated to low-end devices because of the unconstrained increase of the memory footprint. In contrast, Disabato *et al.* [24] presented an online adaptive scheme based on a kNN classifier placed on top of a frozen feature extraction CNN model. The final stage is updated by incrementally adding the labeled samples to the knowledge memory of the kNN classifier. This approach has been evaluated on a tiny STM32F76ZI device but unfortunately has proven the effectiveness only on limited 2-classes problems and presents an unbounded memory requirement, which scales linearly with the number of training samples. PULP-HD [25] showed few-shot continual learning capabilities on an ultra-low power prototype using Hyperdimensional Computing. During the training phase the new data are mapped into a limited hyperdimensional space by making use of a complex encoding procedure; at inference time the incoming samples are compared to the computed class prototypes. The method has been demonstrated on a 10 gesture classification scenario based on EMG data but lacks of experimental evidences to be effective on complex image classification problems. In contrast to the these works, we demonstrate superior learning capabilities for a TinyML platform by *i)* running backpropagation on-device to update intermediate layers, and *ii)* supporting a memory-efficient Latent Replay-based strategy to address catastrophic forgetting on a more complex Continual Learning scenario. An initial CNN-based prototype of a Continual Learning system was presented in in [1] using Latent Replays. The authors demonstrated the on-device learning capabilities using a Qualcomm Snapdragon processor, which features a power envelope 100× higher than our target and therefore it results not suitable for battery-operated tiny devices. In contrast to them, we also extend the LR algorithm by leveraging on quantization to compress the LR memory requirements.

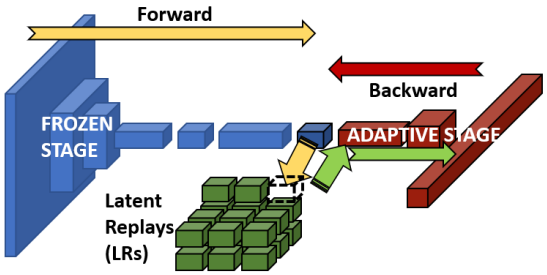

Fig. 1. Continual learning with latent replays. The frozen stage is the light-blue part (first half) of the network. After the first forward of the inputs (yellow arrow), activations (namely LRs) are stored apart. After having stored them, they will be used later mixed with the new images coming through the frozen stage and used to retrain the adaptive portion of the network.

III. Methods

In this section, we analyze the memory requirements of the Latent Replay-based Continual Learning method and present *QLR-CL*, our strategy to reduce the memory footprint of the LR vectors based on a quantization process.

A. Background: Continual Learning With Latent Replays

In general, supervised learning aims at fitting an unknown function by using a set of known examples – the training dataset. In the case of Deep Neural Networks, the training procedure returns the values of the network parameters, such as weights and biases, that minimize a loss function. Among the used optimization strategies, the mini-batch Stochastic Gradient Descent (SGD), which is an iterative method applied over multiple learning step (i.e. the epochs), is widely adopted. In particular, The SGD algorithm computes the gradient of the parameters based on the loss function by back-propagating the error value through the network. This error function compares the model prediction, i.e. the output of the forward pass, with the expected outcome (the data label). Parameter gradients obtained after the backward pass are weighted over a mini-batch of data before updating the model coefficients.

As introduced at the beginning of this work, the Latent Replay CL method [1] is a viable solution to gain TinyML adaptive systems with on-device learning capabilities based on the availability of new labeled data. In Fig. 1 we illustrate the CL process with Latent Replays. The new data are injected into the model to obtain the latent embeddings, which are the feature maps of a specific intermediate layer. We indicate such a layer with the index l, where $l \in [0, L)$, assuming the targeted model to be composed by L stacked layers. At runtime, the new latent vectors are combined with the precomputed N_{LR} Latent Replays vectors to execute the learning algorithm on the last $L - l - 1$ layers. More specifically, the coefficient parameters of the *adaptive stage* are updated by using a mini-batch gradient descend algorithm. Every mini-batch includes both new data (in the latent embedding form) and LR vectors. The typical ratio of new data over the full mini-batch is 1/6 [1]. The coefficient gradients are computed through forward and backward passes over the adaptive (learned) layers. Multiple iterations, i.e. the epochs, of the learning algorithms take place within the training procedure.

B. Memory Requirements

We model the Latent Replay-based Continual Learning task as operating on a set of new data coming from a sensor (e.g., a camera), which is interfaced with an embedded digital processing engine, namely the *TinyML Platform*, and its memory subsystem. Given the limited memory capacity of IoT end-nodes, the quantification of the learning algorithm's memory requirements is essential. We distinguish between two different memory requirements: additional memory necessary for CL, e.g., the LR memory, and that required to save intermediate tensors during forward-prop to be used for back-prop – a requirement common to all algorithms based on gradient descent, not specific to CL.

Concerning the LR memory, the system has to save a set of N_{LR} LRs, each one of the size of the feature map computed at the l-th layer of the network. In our scenario, LR vectors are represented employing floating-point (*FP32*) datatype and typically determine the majority of the memory requirement [18]. Since LRs are part of the static *long-term memory* of the CL system, for their storage, we use non-volatile memory, e.g., external Flash.

On the other hand, forward- and back-prop of the network model require to allocate the space for N_P network parameters statically. In addition, forward-prop requires dynamically allocated buffers to store the activation feature maps for all layers. Up to the l-th layer, these buffers are temporary and can be released after their usage. Conversely, the system must keep in memory the feature maps after l to compute the gradients during back-prop. They can only be released after the corresponding layer has been back-propagated. Lastly, the system must also keep in memory the coefficients' gradients, demanding a second array of N_P elements. To keep accuracy on the learning process, every tensor, i.e. coefficients, gradients, and activations, employ a *FP32* format in our baseline scenario. Different from LRs, these tensors are kept into volatile memories, except the frozen weights, which are stored in a non-volatile memory.

C. Quantized Latent Replay-Based Continual Learning

Quantization techniques have been extensively used to reduce the data size of model parameters, and activation feature maps for the inference task, i.e. the forward pass. An effective quantization strategy reduces the data bitwidth from 32-bit (*FP32*) to low bit-precision, 8-bit or less (Q bits, in general) while paying an almost negligible accuracy loss.

In this paper, we introduce the Quantized Latent Replay-based Continual Learning method (QLR-CL) relying on low-bitwidth quantization to speed up the execution of the network up to the l-th layer and at the same time reduce the memory requirement of the LR vectors from the baseline *FP32* arrays. To do so, we split the deep model into two sub-networks, namely the *frozen stage* and the *adaptive stage*. The *frozen stage* includes the lower layers of the network, up to the Latent Replay layer l. The coefficients of this sub-network, including batch normalization statistics, are frozen during the incremental learning process. On the contrary, the parameters of the *adaptive stage* are updated based on the new data samples.

In QLR-CL, the Latent Replay vectors are generated by feeding the *frozen stage* sub-network with a random subset of training samples from the CL dataset, which we denote as X^{train}. The *frozen stage* is initialized using pre-trained weights from a related problem – in the case of Core50, we use a

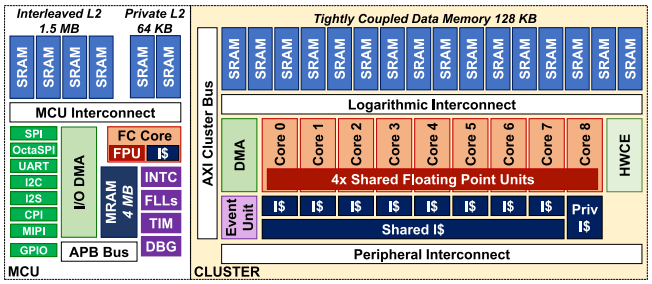

Fig. 2. Architecture outline of the proposed PULP-based system-on-chip for continual learning.

network pre-trained on the ImageNet-1k dataset. Post-Training Quantization of the *frozen stage* is based on training samples X^{train}. We apply a standard Post-Training Quantization process that works by *i)* determining the dynamic range of coefficient and activation tensors, *ii)* dividing the range into equal steps, using a uniform affine quantization scheme [56]. While the statistics of the parameters can be drawn without relying on data, the dynamic range of the activation features maps is estimated using X^{train} as a calibration set. If we denote the dynamic range of the weights at the i-th layer of the network as $[w_{i,min}, w_{i,max}]$, we can define the $w_{i,quant}$ *INT-Q* representation of parameters as

$$w_{i,quant} = \left\lfloor \frac{w_i}{S_{w,i}} \right\rceil, \quad S_{w,i} = \frac{w_{i,max} - w_{i,min}}{2^Q - 1} \tag{1}$$

where Q is the number of bits, w_i is the full-precision output of the *frozen stage*. The representation of activations is similar, but we further restrict (1) for activations a_i by considering the effect of ReLU's: a_i are always positive and $a_{i,quant}$ can be represented using an unsigned *UINT-Q* format:

$$a_{i,quant} = \left\lfloor \frac{a_i}{S_{a,i}} \right\rceil, \quad S_{a,i} = \frac{a_{i,max}}{2^Q - 1} \tag{2}$$

where $a_{i,max}$ is obtained through calibration on X^{train}.

Quantized Latent Replays (QLRs) $a_{l,replay}$ are represented similarly to other quantized activations, setting the layer i to the LR l. Their value is initialized during the initial setup of the QLR-CL process using the latent quantized activations $a_{l,quant}$ over the X^{train} set.

During the QLR-CL process, the *adaptive stage* is fed by dequantized vectors obtained as $S_{a,l} \cdot a_{l,replay}$, along with the dequantized latent representation of the new data sample $S_{a,l} \cdot a_{l,quant}$. Hence, the single *FP32* parameter $S_{a,l}$ is also stored in memory as part of the *frozen stage*. In our experiments, we set the bitwidth Q of all activations and coefficients to 8-bit, while the output of the *frozen stage* is compressed to 8-bit or less, as further explored in Section V.

IV. Hardware/Software Platform

In this section, we describe the hardware architecture of the proposed platform for TinyML learning and the related software stack.

A. Hardware Architecture

The CL platform we propose is inspired and extends on our previous work [18]. We build it upon an advanced PULP-based SoC, called *VEGA*, which combines parallel programming for high-performance with ultra-low-power features. An advanced prototype of this platform has been taped out in GlobalFoundries 22nm technology [20]. The system architecture, which is outlined in Fig. 2, is based on an I/O-rich MCU platform coupled with a multi-core cluster of RISC-V ISA digital signal processing cores which are used to accelerate data-parallel machine learning & linear algebra code. The MCU side features a single RISC-V core, namely the Fabric-Controller (FC), and a large set of peripherals. Besides the FC core, the MCU-side of the platform includes a large L2 SRAM, organized in an FC-private section of 64kB and a larger interleaved section of 1.5MB. The interleaved L2 is shared between the FC core and an autonomous I/O DMA controller, connected to a broad set of peripherals such as OctaSPI/HyperBus to access an external Flash or DRAM of up to 64MB, as well as camera interfaces (CPI, MIPI) and standard MCU interfaces (SPI, UART, I2C, I2S, and GPIO). The I/O DMA controller is connected to an on-chip magnetoresistive RAM (MRAM) of 4MB, which resides in its power and clock domain and can be accessed through the I/O DMA to move data to/from the L2 SRAM.

The multi-core cluster features nine processing elements (PE) that share data on a 128kB multi-banked L1 tightly coupled data memory (TCDM) through a 1-cycle latency logarithmic interconnect. All cores are identical, using an in-order 4-stage architecture implementing the RISC-V *RV32IMCFXpulpv2* ISA. The cluster includes a set of four highly flexible FPUs shared between all nine cores, capable of *FP32* and *FP16* computation [57]. Eight cores are meant to execute primarily data-parallel code, and therefore they use a hierarchical Instruction cache (I$) with a small private part (512B) plus 4kB of shared I$ [58]. The ninth core is meant to be used as a cluster controller for control-heavy data tiling & marshaling operations; it has a private I$ of 1kB. The cluster also features a multi-channel DMA engine that autonomously handles data transfers between the shared L1 and the external memories through a 64-bit AXI4 cluster bus. The DMA can transfer up to 8B/cycle between L2 and L1 TCDM in both directions simultaneously and perform 2D strided access on the L2 side by generating multiple AXI4 bursts. The cluster can be switched on and off at runtime by the FC core employing clock-gating; it also resides on a separate power domain than the MCU, making it possible to completely turn it off and to tune its Vdd using an embedded DC-DC regulator.

B. Software Stack

To execute the CL algorithm, the workload is largely dominated by the execution of convolutional layers, such as pointwise, and depthwise, or fully connected layers (∼98% of operations in MobileNet-V1). Consequently, the main load on computations is due to variants of matrix multiplications during the forward and backward steps, which can be efficiently parallelized on the 8 compute PEs of the cluster, leaving one core out to manage tiling and program data transfers. Thus, to enable the learning paradigm on the PULP platform, we propose a SW stack composed of parallel layer-wise primitives that realize the *forward step* and the back-propagation. The latter concerns either the computation of the activation gradients (*backward error step*) and coefficient gradients (*backward gradient step*). Fig. 3 depicts the dataflow of the *forward*

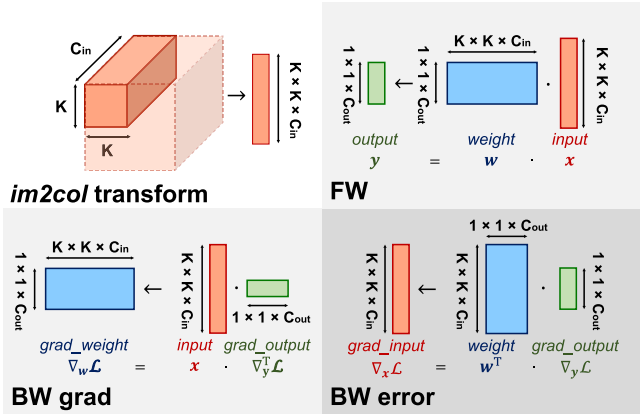

Fig. 3. Clockwise from top-left: *im2col* transform, forward and backward propagation for error and gradient calculation for a $K \times K$ conv layer.

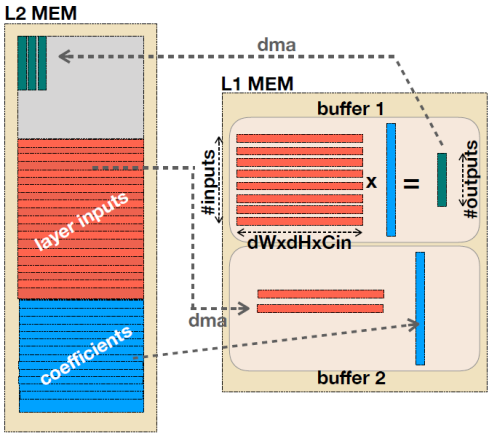

Fig. 4. Tiling scheme between L2 and L1 memories exploiting double-buffering. Two equal buffers are filled with the matrix multiplication terms that fit into half the size of L1. The second buffer is filled with the next terms of the convolution that have to be matrix-multiplied.

and *backward* for commonly used convolutional kernels such as pointwise (PW), depthwise (DW), and linear (L) layers. To reshape all convolution operations into matrix multiplications, the *im2col* transformation is applied to the activation tensors to reshape them into2D matrix operands [37]. The *FP32* matrix multiplication kernel is parallelized over the eight cores of the cluster according to a data-parallelism strategy, making use of *fmadd.s* (floating multiply-add) instructions made available by the shared FPU engines.

The cores must operate on data from arrays located in the low-latency L1 TCDM to maximize throughput and computational efficiency (i.e., IPC). However, the operands of a layer function may not entirely fit into the lower memory level because of the limited space (128kB). For instance, the tensors of the PW layer #22 of the used MobileNet-V1 occupy 1.25MB. Hence, the operands have to be sliced into reduced-size blocks that can fit into the available L1 memory and convolutional functions are applied on L1 tensor slices to increase the computational efficiency.

This approach is generally referred to as *tiling* [39], which is schematized in Fig. 4. By locating layer-wise data on the larger L2 memory (1.5MB), the DMA firstly copies individual slices of operand data, also referred to as *tiles*, intoL1 buffers, to be later fetched by the cores. Since the cluster DMA engine is capable of 2D-strided access on the L2 side, this operation can also be designed to perform *im2col*, without any manual data marshaling overhead on L1.

To increase the computation efficiency, we implement a software flow that interleaves DMA transfers between L2 and L1 and calls to parallel primitives, e.g. *forward*, *backward error*, or *backward gradient steps*, which operate on individual tiles of data. Hence, every layer is expected to load and process all the tiles of any operand tensor. To reduce the overhead due to the data copy, the DMA transfers take place in the background of the multi-core computation: the copy of the next tile is launched before invoking the computation on loaded tiles. On the other side, this optimization requires doubling the L1 memory requirement: while one L1 buffer is used for computation, an equally-sized buffer is used by the data movement task. From a different viewpoint, the maximum tile size must not exceed half of the available memory. At runtime, layer-wise *tiled* kernels are invoked sequentially to run the learning algorithm with respect to the input data. To this aim, LRs are loaded from external embedded memory, if not fitting the internal memory, and copied to the on-chip L2 memory thanks to the I/O DMA.

V. Experimental Results

In this section, we provide the experimental evidence about our proposed TinyML platform for on-device Continual Learning. First, we evaluate the impact of quantization of the *frozen stage* and the LR vectors upon the overall accuracy, and we analyze the memory-accuracy trade-off.

Secondly, we study the efficiency of the proposed SW architecture with respect to multiple HW configurations, namely #cores, L1 size and DMA bandwidth, introducing the tiling requirements and evaluating the latency for each kernel of computation. Then, we measure performance on an advanced PULP prototype, VEGA, fabricated in GlobalFoundries 22nm technology with 4 FPUs shared among all cores. We analyze the latency results for individual layers forward and backward and estimate the overall energy consumption to perform a CL task on our platform. Finally, we compare the efficiency of our TinyML platform to other devices used for on-device learning.

A. Experimental Setup

We benchmark the compression technique for the Latent Replay memory on the image-classification Core50 dataset, which includes 120k 128×128 RGB images of 50 objects for the training and about 40k images for the testing. On the Core50 dataset, the CL setting is regulated by the NICv2-391 protocol [59]. According to this protocol, 3000 images belonging to ten classes are made available during the initial phase to fine-tune the targeted deep model on the Core50 problem. Afterward, the remaining 40 classes are introduced at training time in 390 learning events. Each event, as described more in detail in Section III-A, comprises iterations over mini-batches of 128 samples each: 21 coming from actual images, all from the same class and typically not independent (e.g., coming from a video), and 107 latent replays. After each learning event, the accuracy is measured on the test set, which includes samples from the complete set of classes.

Following [1], we use a MobileNet-V1 model with an input resolution of 128×128 and width multiplier 1, pre-trained

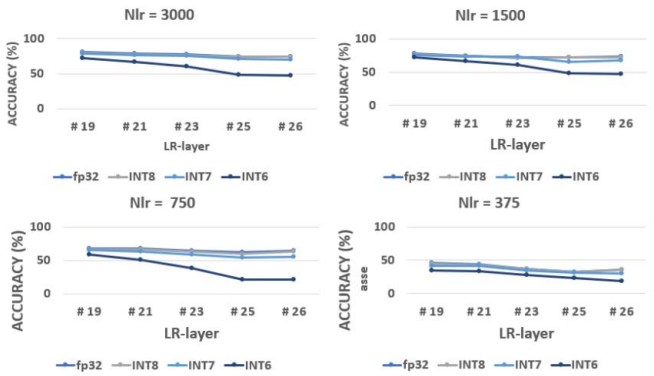

Fig. 5. Accuracy plots for $N_{LR} = \{375, 750, 1500, 3000\}$ and different levels of quantization. From these plots it is visible that below UINT-7 accuracy degrades rapidly.

on ImageNet; we start from their public released code[7] and use PyTorch 1.5. In our experiments, we replace BatchReNormalization with BatchNormalization layers and we freeze the statistics of the *frozen stage* after fine-tuning.

B. QLR-CL Memory Usage and Accuracy

To evaluate the proposed *QLR-CL* setting, we quantize the *frozen* stage of the model using the PyTorch-based NEMO library [60] after fine-tuning the MobileNet-V1 model with the initially available 3000 images. We set the activation and parameters bitwidth of the *frozen* stage to $Q = 8$ bit while we vary the bitwidth Q_{LR} of the latent replay layer. The quantized *frozen* stage is used to generate a set of N_{LR} Latent Replays, as sampled from the initial images.

The plots in Fig. 5 show the test accuracy on the Core50 that is achieved at the end of the NICv2-391 training protocol for a varying $N_{LR} = \{375, 750, 1500, 3000\}$ while sweeping the LR layer l. Depending on the selected layer type, the size of the LR vector varies as reported in Table III.

Each subplot of Fig. 5 compares the baseline *FP32* version with our 8-bit fully-quantized solutions with a varying $Q_{LR} = \{8, 7, 6\}$, denoted in the figures, respectively, as UINT-8, UINT-7 and UINT-6. For a $Q_{LR} < 6$, we observe the Continual Learning process to not converge on the Core50 dataset.

From the obtained results, we can observe the UINT-8 compressed solution featuring a small accuracy drop with respect to the full-precision *FP32* baseline. When increasing the number of latent replays N_{LR} to 3000, the UINT-8 quantized version results almost lossless (-0.26%), if LR $= 19$. On the contrary, if the LR layer is moved towards the last layer (LR $= 27$), the accuracy drop increases up to 3.4%. The same effect is observed when reducing N_{LR} to 1500, 750 or 375. In particular, when $N_{LR} = 1500$, the UINT-8 quantatized version presents an accuracy drop from 1.2% (LR $= 19$) to 2.9% (LR $= 27$). On the other hand, lowering the bit precision to UINT-7, the accuracy reduces on average of up to 5.2%, if compared to the *FP32* baseline. Bringing this further down to UINT-6 largely degrades the accuracy by more than 10%.

To deeply investigate the impact of the quantization process on the overall accuracy, we perform an ablation study to distinguish the individual effects of *i)* the quantization of the front-end and *ii)* the quantization of the LRs. In case of $N_{LR} = 1500$, Table II compares the accuracy on the Core50 dataset for different LR layers, if applying quantization to both the LR memory and the frozen stage or only to the LR memory. The accuracy statistics are averaged over 5 experiments; we report in the table the mean and the std deviation of the obtained results. In particular, we see that quantizing the LRs has a larger effect on the accuracy than quantizing the frozen graph. By quantizing only the LR memory to UINT-8, the accuracy drops by up to 1.2-2.6% (higher in case of larger adaptive stages) with respect to the FP32 baseline. On the contrary, the UINT-8 quantized frozen graph brings only an additional 0.5-1% of accuracy drop. With UINT-7 LRs, the accuracy drop is mainly due to the LR quantization: when compressing also the frozen stage to 8-bit the accuracy drop is up to -1%, which is small compared to the total 4-7% of accuracy degradation.

To facilitate the interpretation of the results, Fig. 6 reports the test accuracy for multiple quantization settings compared to the size (in MB) of the Latent Replay Memory. In red, we highlight a Pareto frontier of non-dominated points, to have a range of options to maximize accuracy and minimize the memory footprint. Among the best solutions, we detect two clusters of points on the frontier. The first cluster (**A**), corresponding to the low-memory side of the frontier, is constituted by experiments that use $l = 27$ with 1500 or 3000 LRs and UINT-7 or UINT-8 representation. On the other hand, if we aim at the highest accuracy possible for our QLR-CL classification algorithm, we can follow the Pareto frontier to the right towards higher accuracies at steeper memory cost, reaching cluster **B**. All points in cluster **B** features $l = 23$

TABLE II

ACCURACY ON CORE50 DATASET WITH MULTIPLE QUANTIZATION SETTINGS **A**+**B**, WHERE **A** DENOTES THE QUANTIZATION OF THE FROZEN STAGE (FP32 OR UINT-8) AND **B** INDICATES THE QUANTIZATION SCHEME OF THE LR VECTORS (FP32, UINT-8, UINT-7). THE BASELINE IS FP32

LR layer	FP32 baseline	FP32 + UINT-8	UINT-8 + UINT-8	FP32 + UINT-7	UINT-8 + UINT-7
27	**72.7** ± 0.34	**70.1** ± 0.54	**69.2** ± 0.48	**68.0** ± 0.63	**67.8** ± 1.14
25	**73.3** ± 0.58	**70.9** ± 0.65	**70.2** ± 0.67	**66.2** ± 0.75	**66.1** ± 0.94
23	**75.0** ± 0.83	**73.2** ± 0.46	**73.4** ± 0.66	**71.1** ± 0.63	**69.9** ± 1.25
21	**76.5** ± 0.63	**74.9** ± 0.51	**73.9** ± 1.67	**72.7** ± 0.74	**72.6** ± 1.30
19	**77.7** ± 0.73	**76.5** ± 0.48	**76.0** ± 0.80	**74.0** ± 0.57	**75.2** ± 1.10

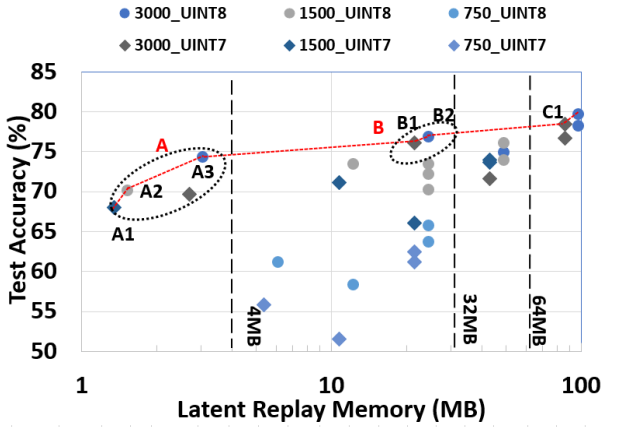

Fig. 6. Accuracy achieved by considering $N_{LR} = \{750, 1500, 3000\}$ and different precision, highlighting the Pareto frontier.

[7]Available at https://github.com/vlomonaco/ar1-pytorch/. While Pellegrini *et al.* [1] report lower accuracies in their paper, our *FP32* baseline results are aligned with their released code.

TABLE III
SIZE OF TILES FOR THE MOBILENET-V1 LAYERS

LR Layer l	Layer Type	LR Dim. ($H \times W \times C$)	LR Size (#elements)
19	DW	8×8×512	32k
20	PW	8×8×512	32k
21	DW	8×8×512	32k
22	PW	8×8×512	32k
23	DW	4×4×512	8k
24	PW	4×4×1024	16k
25	DW	4×4×1024	16k
26	PW	4×4×1024	16k
27	Linear	1×1×1024	1k

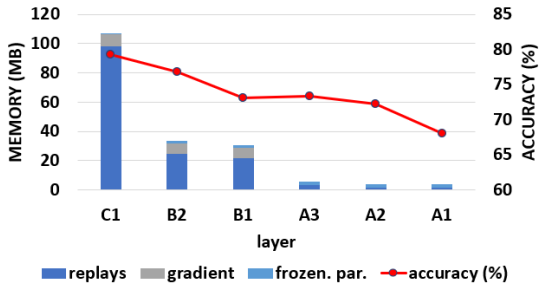

Fig. 7. Memory requirements for the points highlighted in Fig. 6. Each layer belongs to the Pareto frontier and accounts for all the memory components. Going deeper into the network, LRs (gray) dominate memory consumption. The other components are the parameters of the frozen stage, the gradient and the activations needed during the training.

as Latent Replay layer, which is a bottleneck layer of the network and allows to store more compact tensors as LR (refer to Table III). Adopting LR layers within **B** leads accuracy to an average of 76%, gaining ∼5% on average with respect to the layers within cluster **A**. A single point **C1** is shown further to the right, but still below 128MB.

For a deeper analysis of the Pareto frontier, in Fig. 7, we detail the memory requirements when analyzing the points into the two clusters **A** and **B**, as well as **C1**. We make two observations: first, in all **A** points, it would be possible to fit entirely within the on-chip memory available on VEGA, exploiting the 4MB of non-volatile MRAM. This would allow avoiding any external memory access, increasing the energy efficiency of the algorithm by a factor of up to ∼3× [20]. Moreover, considering that the maximization of accuracy is often the primary objective in CL, we observe that accumulating features at $l = 19$ with 1500 UINT-8 LRs (point **C1**) enables accuracy to grow above 77%, almost 10% more than the compact solutions in **A** (Fig. 7). This analysis allows us to also speculate over possible future architectural explorations to design optimized bottleneck layers that could facilitate better memory accuracy trade-off for QLR-CL.

C. Hardware/Software Efficiency

To assess the performance of the proposed solution, we study the efficiency of the CL Software primitives on the target platform and the sensitivity to some of the HW architectural parameters, namely the #cores, the L1 memory size and the cluster DMA Bandwidth.

1) Single-Tile Performance on L1 TCDM: Based on the tiling strategy described in Section IV-B, we run experiments concerning the CL primitives of the software stack that operates on individual tiles of data placed in the L1 memory. Figure 8 shows the latency performance, expressed as *MAC/cyc*, i.e. the ratio between Multiply-Accumulate operations (*MAC*) and elapsed clock cycles (*cyc*), for each of the main *FP32* computation kernels in case of single-core (*1-CORE*) or multi-core (*2-4-8-CORES*) execution. We highlight that a higher value of MAC/cyc denotes a more efficient processing scheme, leading to lower latency for a given computation workload, i.e. fixed MAC. More specifically, in this plot, we evaluate the forward (*FW*), backward error (*BW ERR*), and backward gradient (*BW GRAD*) for each of the considered layer for a varying size of the L1 TCDM memory, i.e. 128, 256 or 512kB. The shapes of the tiles for PointWise (*PW*), DepthWise (*DW*), and Linear (*Lin*) layers used for the experiments are reported in the tables on the left of the figure. Such dimensions are defined to fit three different sizes of the TCDM, considering buffers of size 64kB, 128kB and 256kB.

Focusing firstly on the *PW* layers (histograms at the top of the figure), we observe a peak performance in the 8-cores *FW* step, achieving up to 1.91 MAC/cyc for a L1 memory size of 512kB. We observe also a performance improvement of up to 11% by increasing the L1 size from 128kB to 512kB, which is motivated by the higher computational density of the kernel: if L1 = 512kB the inner loop features 4× iterations than a scenario with 128kB of L1 size. Moreover, the parallel speedup scales almost linearly with respect to the number of cores and archives 7.2× in case of 8 cores. With respect to the theoretical maximum of 8×, the parallel implementation presents some overheads mainly due to increased L1 TCDM contentions and cluster's cache misses.

If we look at *DW* convolutions, their performance is lower with respect to the others. The main reason is that it requires a software-based *im2col* data layout transformation, which increase the amount of data marshaling operations and adds an extra L1 buffer, thus reducing the size of matrices in the matrix multiplication, leading to increased overheads. Specifically, we measure the workload of the *im2col* to achieve up to 70% of the FW kernel's latency. As mentioned in Section IV, the primitives we introduce also support performing the im2col directly when moving the data tile from L2 via DMA transfer – in that case, this source of performance loss is not present, and the MAC/cyc necessary for depthwise convolutions increases up to 1 MAC/cycles for depthwise forward-prop, depending also on the L1 size selected. The remaining overhead with respect to pointwise convolutions is justified by the fact that depthwise convolutions can only exploit filter reuse (of size 3×3, for example, in MobileNet-V1 DW layers) and no input channel data-reuse, resulting in much shorter inner loops and more visible effect of overheads. This latter effect cannot be counteracted by efficient DMA usage; on the other hand, since depthwise convolutions account for less than 1.5% of the computation, their impact on the overall latency is limited, as we further explore in the following section.

Moving our analysis towards the different performance between forward- and backward-prop layers (particularly BW grad), we observe that this effect is again due to different data re-use between the matrix multiplication kernels. The reduction in re-use in the backward-prop is due to the tiling strategy adopted (see Fig. 3) has a *grad_output* vector which is shorter than the input in the forward matrix multiplication.

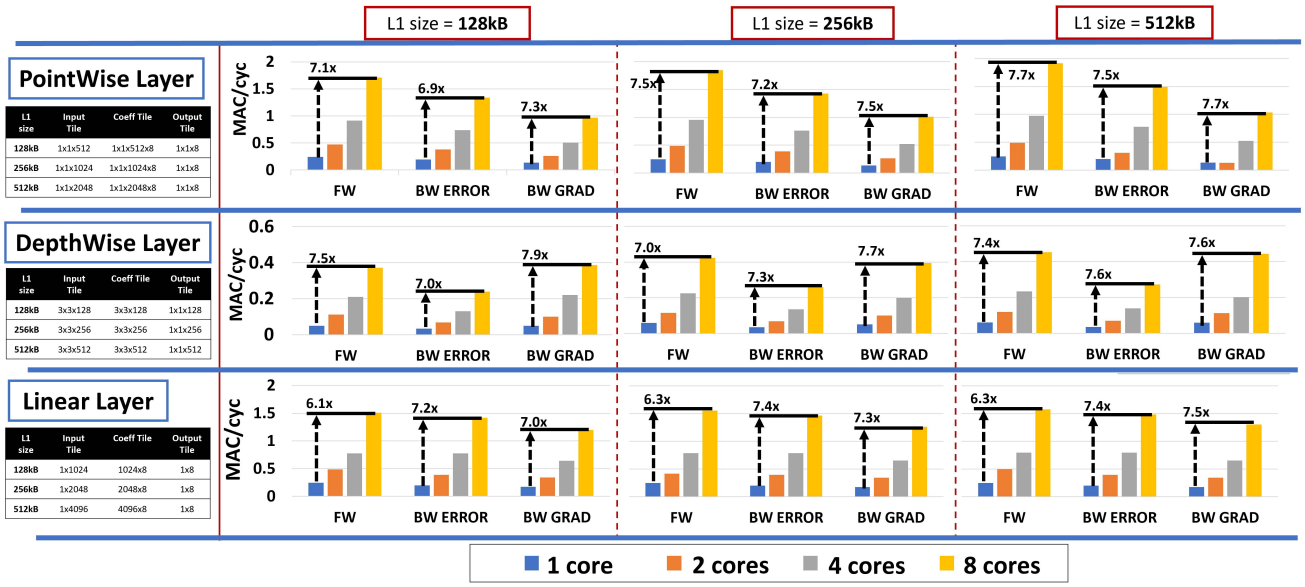

Fig. 8. Efficiency, expressed in MAC/cyc, of the proposed CL primitives for forward and backward pass: pointwise, depthwise, and linear layers. The analysis concerns a varying number of cores (1, 2, 4 or 8) and L1 memory size (128, 256 or 512 kB), which impacts on the dimension of the layer's tensor tiles as reported in the tables on the left.

Specifically, the input to the matrix multiplication has size 8×1x1 in backward, while the input shape in forward changes accordingly with the L1 memory: 512 × 1x1 for 128kB L1, 1024 × 1x1 for 256kB L1 and 2048 for 512kB L1. In this scenario, the inner loop of the matrix multiplication of a forward computation is 64×, 128× or 256× larger with respect to the backward kernels' cases. This fact motivates the lower MAC/cyc of the *BW ERR* step (22%) and *BW GRAD* step (−46%) if compared to the FW kernel.

2) L2-L1 DMA Bandwidth Effects on Performance: Next we analyze the impact of L2-L1 DMA Bandwidth variations, due to the Cluster DMA, on the overall performance of the learning task. In particular, we monitor the latency and the MAC/cyc for multiple values of L2-L1 bandwidth ranging from 8 to 128 bits per clock cycle (bit/cyc) and different configurations of #cores and L1 size. We remark that a higher value of MAC/cyc indicates a better performing HW configuration. Our analysis assumes a single half-duplex DMA channel, hence the bandwidth value accounts for either read or write transfers.

Fig. 9 reports the average MAC/cyc when running the forward and backward steps with respect to the L2-L1 cluster's DMA bandwidth. As a benchmark, we consider the adaptive stage of the MobileNetV1 model when the LR layer is set to the 19th layer. Hence, we adopt our tiling strategy and double-buffering scheme to realize the training. When increasing the L1 size, the tensor tiles feature a larger size, therefore demanding a higher transfer time to copy data between the L1 memory (used for computation) and L2 memory (used for storage). Thanks to the adopted double-buffering technique, such transfer time can be hidden by the computation time because the DMA works in the background of CPU operation (*compute-bound*). On the contrary, if the transfer time results dominating, the computation becomes *DMA transfer-bound*, with lower benefits from the multi-core acceleration.

In case of single core execution, the measured MAC/cyc does not vary with respect to the L1 size (128kB, 256kB or

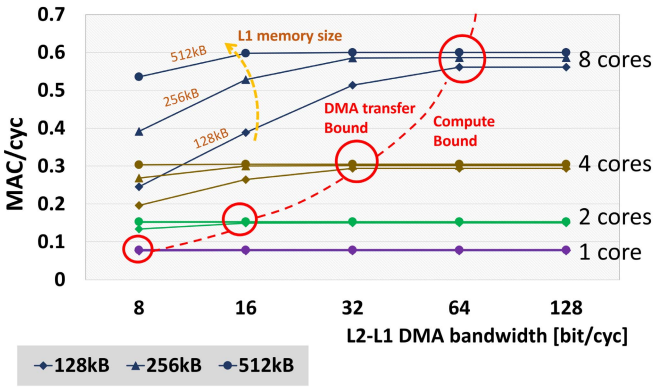

Fig. 9. SW efficiency, expressed as average MAC/cyc, when running forward and backward steps with respect to a varying L1-L2 bandwidth. Every line corresponds to a configuration of #cores (1, 2, 4 or 8 cores) and L1 memory size (128, 256 or 512kB).

512kB) as can be seen from the plot. In this scenario, the CPU time results as the dominant contribution with respect to the transfer time: the execution is compute-bound and a higher L2-L1 bandwidth does not impact the overall performance. Differently, in a multi-core execution (2, 4 or 8 cores), the average MAC/cyc increases and therefore the ratio between transfer time and the computation time decreases: from the plot we can observe higher performance if the DMA bandwidth is increased. If featuring a L1 size of 128kB, the sweet spots between DMA and compute bound are observed when the L2-L1 DMA bandwidth is 16 (2 cores), 32 (4 cores) and 64 (8 cores) bit/cyc, respectively, as highlighted by the red circles in the plot. These configurations denote the sweet spots to tune the DMA requirements with respect to the chosen L1 memory size and #cores.

If focusing more on the impact of the L1 memory size to the multi-core performance, we observe up to 2× efficiency gain with 8 cores with a larger L1 memory, increasing from

0.25 MAC/cyc for a 128kB L1 memory to 0.4MAC/cyc at L1=256kB and to 0.53MAC/cyc for 512kB of L1. At 64 bit/cyc of L2-L1 DMA bandwidth, the execution, which is dominated by the computation, reaches 0.52MAC/cyc, 2.12× faster than the low-bandwidth configuration.

From this analysis we can conclude that the best design point for the learning task on a low-end multi-core architecture can be pinpointed leveraging the L2-L1 DMA Bandwidth and the L1 memory size tuning: when using 8 cores, 128kB of L1 memory, which is typically the main expensive resource for the system, can lead already to the highest performance as long as the DMA features a bandwidth of 64 bit/cyc. On the contrary, if the DMA's bandwidth is as low as 8 bit/cyc, a 512 kB L1 memory is needed to gain maximum performance. The target chip VEGA includes a L1 memory of 128 kB; the DMA follows a full-duplex scheme and can provide up to 64 bit/cyc for read transactions and 64 bit/cyc for write transactions. Therefore the VEGA HW architecture can fully exploit the presented SW architecture and optimization schemes to reach the optimal utilization and performance for the learning task.

D. Latency Evaluation on VEGA SoC

We run experiments on the VEGA SoC to assess the on-device learning performance, in terms of latency and energy consumption, of the proposed QLR-CL framework. Specifically, we report the computation time, i.e. the latency, at the running frequency of 375MHz and the power consumption by measuring the current absorbed by the chip when powered at 1.8V. To measure the full layer latency, we profile *forward* and *backward* tiled kernels, which include DMA transfers of data, initially stored in L2, and calls to low-level kernel primitives, introduced above. On average, we observe a 7% of tiling overhead with respect to the single-tile execution on L1. This is not surprising, due to the large bandwidth availability between L1 and L2 and the presence of compute-bound matrix multiplication operations.

Based on the implemented tiled functions, we report the layer-wise performance in Table IV for any of the layers of the MobileNet-V1 model. We consider as complete time for the execution of a layer the cumulated time for *frozen stage* and *adaptive stage*. The latency of the *frozen stage* is obtained using DORY [39] to deploy the network, as this operation is performed as pure 8-bit quantized inference. We compute the full latency of the *adaptive stage* as the time needed to execute the forward and backward phases of each layer. Since we have multiple configurations, latencies for retraining start growing from the last layer (#27) up to layer #20, where retraining comprises a total of eight layers.

First of all, we note that *frozen stage* latencies are utterly dominated by the *adaptive stage*. Apart from the faster inference backend, which can rely on 8-bit SIMD vectorization, this is because only 21 images per mini-batch pass through the *frozen stage*, while the *adaptive stage* has to be executed on 128 latent inputs (107 LRs and the 21 dequantized outputs from the *frozen stage*), and it has to run for multiple epochs (by default, 4) in order to work.

When $l = 27$, the *adaptive stage* is very fast thanks to its very small number of parameters (it produces just the 50 output classes). This is the only case in which the frozen stage is non-negligible ($\sim$ 1/6 of the overall time). Progressing upward in the table, the frozen stage becomes negligible. The cumulative impact of forward and backward passes through all the other layers we take into account (l from #20 to #26) is in the range between 0.3h and 1.5h. In particular, $l = 23$ corresponds to $\sim$14 min per learning event; this LR layer corresponds to high accuracy (>75% in Core50, see Fig. 6), which means that in this time the proposed system is capable of acquiring a very significant new capability (e.g., a new task/object to classify) while retaining previous knowledge to a high degree.

TABLE IV
CUMULATIVE LATENCY VALUES PER LEARNING EVENT FOR VEGA, STM32, AND SNAPDRAGON845

LR Layer l	**VEGA** @ 375 MHz **Adaptive [s]**	**Frozen [s]**	**Cumul. En. [J]**	**STM32L4** @ 80 MHz **Total [s]**	**Cumul. En. [J]**	**Snapdragon Total [s]**
20	$2.49 \cdot 10^3$	0.87	154	$1.65 \cdot 10^5$	5688	n.a.
21	$1.73 \cdot 10^3$	0.94	107	$1.15 \cdot 10^5$	3981	n.a.
22	$1.64 \cdot 10^3$	0.95	101	$1.08 \cdot 10^5$	3728	n.a.
23	$8.77 \cdot 10^2$	1.03	54.3	$5.86 \cdot 10^4$	2020	n.a.
24	$7.81 \cdot 10^2$	1.03	48.4	$5.12 \cdot 10^4$	1769	n.a.
25	$4.01 \cdot 10^2$	1.09	24.9	$2.65 \cdot 10^4$	915	n.a.
26	$3.81 \cdot 10^2$	1.10	23.5	$2.49 \cdot 10^4$	859	n.a.
27	2.07	1.25	0.13	$1.39 \cdot 10^2$	4.80	0.50

Having the basic mini-batch measurements, we can estimate any scenario, by considering that to train with 1500 LR and $l = 27$, we will need 300 new images, thus we need 14 mini-batches (300/21), which leads to 3.30 seconds to learn a new set of images, with an accuracy of 69.2%. If we push back the LR layer l, this leads to an increase of accuracy 76.5%, at the expense of much larger latency, up to 42 minutes for layer #20 (see Table IV).

E. Energy Evaluation on CL Use-Cases and Comparison With Other Solutions

To understand the performance of our system and its real-world applicability, we study two use-cases: a single mini-batch of the Core50 training we used, and the simplified scenario presented by Pellegrini *et al.* [1] in their demonstration video. We compare our results with another MCU targeting ultra-power consumption: a NUCLEO-64 board based on the STM32L476RG MCU, on which we ran a direct port of the same code we use on the PULP-based platforms. It has two on-chip SRAMs with 1-cycle access time and an overall capacity of 96kB. Performance results, in terms of latency, are reported in Table IV, where we take into account the cumulative latency values both for VEGA and STM32 implementations, along with the cumulative energy consumption. Cumulative latency is computed by adding from the linear layer of the network the latencies of the preceding layers.

On average, execution on VEGA's 8-cores on performs **65**× faster with respect to the STM32 solution thanks to three main factors. Firstly, the clock frequency of VEGA is 4.7× higher than the max clock frequency of the STM32L4 (375MHz vs. 80MHz), also thanks to the superior technology node. Secondly, VEGA presents a parallel speed-up of up to 7.2×. Lastly, thanks to the more optimized ISA and the core microarchitecture, VEGA performs less operations while executing the same learning task. For example, the inner loop of the matrix

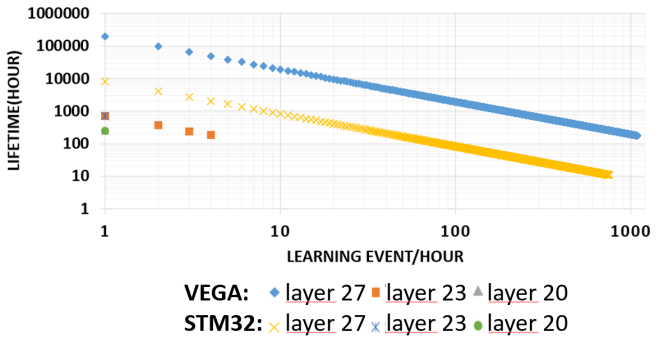

Fig. 10. Battery lifetime of the VEGA SoC and the STM32L4 devices when handling multiple learning events per hour.

multiplication on VEGA requires only 4 instructions while the STM32L4 takes 9 instructions, resulting 2.25× faster, mainly thanks to the HW loop extension and the fmadd.s instruction.

The latency speed up, leads to an energy gain of around **37**×, because the average power consumption of VEGA is **2**× higher than the STM32L4 at full load.

Notice that the latency measurement of the STM32L4 does not account for possible overheads due to the tiling data between the small on-chip SRAM banks and off-chip memory. Even then, our results show that fine-tuning from any layer above the last one results in too large a latency to be realistic on STM32L4 – in the order of a day per learning event with $l = 23$. On the contrary, CL on VEGA can be completed in $\sim$ 14 minutes if selecting $l = 23$ or as fast as 3.3 seconds if retraining only the last layer.

Given the reported energy consumption, we estimated the battery lifetime of our device when adapting the inference model by means of multiple learning events per hour; we assumed no extra energy consumption for the remaining time. In particular, Fig. 10 shows the battery lifetime (in hours) depending on the selected Latent Replay layer and the adaptation rate, expressed as the amount of learning events per hour. We considered a 3300 mAh battery as the only energy source for the device. By retraining only the last layer (LR = 27), an intelligent node featuring our device can perform more than 1080 continual learning events per hour, leading to a lifetime of about 175h. On the contrary, if retraining larger portions of the network, the training time increases and the maximum rate of the learning events reduces to less than 10/hour, with a lifetime in the range 200-1000h. In comparison, on a STM32L4, if retraining the coefficients of the last layer, the maximum learning rate per hour is limited to 750, with a lifetime of about 10h. This latter can be increased up to 10000h but retraining only once in one hour. At the same learning event rate, the battery lifetime of VEGA is 20x higher.

Lastly, we compare with the use-case presented by Pellegrini *et al.* [1], where they developed a mobile phone application that performs CL with LRs on a OnePlus6 with Snapdragon845. For this scenario, they consider only 500 LRs before the linear layer, these will be shuffled with 100 new images. Then, by construction the mini-batch is composed of 100 LRs and 20 new images, thus, for each of the 8 training epochs, the network will process 5 times over the 20 new images and the 100 LRs. This scenario leads them to obtain an average latency of 502 ms for a single learning event. On the other hand, considering our measurements on VEGA we obtain a forward latency of 1.25s and a training time of 2.07s for a whole learning event.

Considering the power envelope of a Snapdragon845 of about 4W, and the average power consumption of VEGA of 62mW, this implies that our solution is 9.7× more efficient in terms of energy. We additionally assess the energy consumption and the duration of a battery in the mobile application scenario, provided the energy measurements on VEGA, when using a 3300mAh battery. Thus, if we consider performing learning over a mini-batch of images once every minute in the ultra-fast scenario (just retraining the linear layer) and to perform an inference each second, we obtain an energy consumption of 0.25J per minute. This leads the accuracy of the model to achieve an average of 69.2%, with an overall lifetime of about 108 days.

VI. Conclusion

In this work, we presented what, to the best of our knowledge, is the first HW/SW platform for TinyML Continual Learning – together with the novel Quantized Latent Replay-based Continual Learning (QLR-CL) methodology. More specifically, we propose to use low-bitwidth quantization to reduce the high memory requirements of a Continual Learning strategy based on Latent Replay rehearsing. We show a small accuracy drop as small as 0.26% if using 8-bit quantized LR memory if compared to floating-point vectors and an average degradation of 5% if lowering the bit precision to 7-bit, depending on the LR layer selected. Our results demonstrate that sophisticated adaptive behavior based on CL is within reach for next-generation TinyML devices, such as PULP devices; we show the capability to learn a new Core50 class with accuracy up to 77%, using less than 64MB of memory – a typical constraint to fit Flash memories. We show that our QLR-CL library based on VEGA achieves up to ∼65× better performance than a conventional STM32 microcontroller.

These results constitute an initial step towards moving the TinyML from a strict *train-then-deploy* approach to a more flexible and adaptive scenario, where low power devices are capable to learn and adapt to changing tasks and conditions directly in the field.

Despite this work focused on a single CL method, we remark that, thanks to the flexibility of the proposed platform, other adaptation methods or models can be also supported, especially if relying on the back-propagation algorithm and CNN primitives, such as convolution operations.

Acknowledgment

The authors acknowledge CINECA for the availability of high-performance computing resources. The author would like to thank Vincenzo Lomonaco and Lorenzo Pellegrini for the insightful discussions.

References

[1] 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.

[2] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 kb RAM for the Internet of Things," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1935–1944.

[3] C. R. Banbury *et al.*, "Benchmarking tinyML systems: Challenges and direction," 2020, *arXiv:2003.04821*. [Online]. Available: http://arxiv.org/abs/2003.04821

[4] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: Parameterized clipping activation for quantized neural networks," 2018, *arXiv:1805.06085*. [Online]. Available: http://arxiv.org/abs/1805.06085

[5] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" in *Proc. Mach. Learn. Syst.*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 129–146.

[6] R. David *et al.*, "TensorFlow lite micro: Embedded machine learning on TinyML systems," 2020, *arXiv:2010.08678*. [Online]. Available: http://arxiv.org/abs/2010.08678

[7] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in AI safety," 2016, *arXiv:1606.06565*. [Online]. Available: http://arxiv.org/abs/1606.06565

[8] M. de Prado, M. Rusci, A. Capotondi, R. Donze, L. Benini, and N. Pazos, "Robustifying the deployment of TinyML models for autonomous mini-vehicles," *Sensors*, vol. 21, no. 4, p. 1339, Feb. 2021.

[9] M. Song *et al.*, "*In-situ* AI: Towards autonomous and incremental deep learning for IoT systems," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture (HPCA)*, Feb. 2018, pp. 92–103.

[10] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," in *Proc. Nat. Acad. Sci. USA*, vol. 114, no. 13, pp. 3521–3526, 2017.

[11] S. Dhar, J. Guo, J. J. Liu, S. Tripathi, U. Kurup, and M. Shah, "A survey of on-device machine learning: An algorithms and learning theory perspective," *ACM Trans. Internet Things*, vol. 2, no. 3, pp. 1–49, 2021.

[12] M. Delange *et al.*, "A continual learning survey: Defying forgetting in classification tasks," *IEEE Trans. Pattern Anal. Mach. Intell.*, early access, Feb. 5, 2021, doi: 10.1109/TPAMI.2021.3057446.

[13] Z. Mai, R. Li, J. Jeong, D. Quispe, H. Kim, and S. Sanner, "Online continual learning in image classification: An empirical survey," 2021, *arXiv:2101.10423*. [Online]. Available: http://arxiv.org/abs/2101.10423

[14] G. M. van de Ven and A. S. Tolias, "Three scenarios for continual learning," 2019, *arXiv:1904.07734*. [Online]. Available: http://arxiv.org/abs/1904.07734

[15] A. Chaudhry *et al.*, "On tiny episodic memories in continual learning," 2019, *arXiv:1902.10486*. [Online]. Available: http://arxiv.org/abs/1902.10486

[16] V. Lomonaco *et al.*, "CVPR 2020 continual learning in computer vision competition: Approaches, results, current challenges and future directions," 2020, *arXiv:2009.09929*. [Online]. Available: http://arxiv.org/abs/2009.09929

[17] Z. Mai, H. Kim, J. Jeong, and S. Sanner, "Batch-level experience replay with review for continual learning," 2020, *arXiv:2007.05683*. [Online]. Available: http://arxiv.org/abs/2007.05683

[18] L. Ravaglia *et al.*, "Memory-latency-accuracy trade-offs for continual learning on a RISC-V extreme-edge node," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2020, pp. 1–6.

[19] D. Rossi *et al.*, "PULP: A parallel ultra low power platform for next generation IoT applications," in *Proc. IEEE Hot Chips Symp. (HCS)*, Aug. 2015, pp. 1–39.

[20] D. Rossi *et al.*, "A 1.3 TOPS/W 32 GOPS fully integrated 10-core SoC for IoT end-nodes with 1.7 μW cognitive wake-up from MRAM-based state-retentive sleep mode," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, vol. 64, Feb. 2021, pp. 60–62.

[21] S. Cass, "Taking AI to the edge: Google's TPU now comes in a maker-friendly package," *IEEE Spectr.*, vol. 56, no. 5, pp. 16–17, May 2019.

[22] H. Cai, C. Gan, L. Zhu, and S. Han, "TinyTL: Reduce memory, not parameters for efficient on-device learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 1–13.

[23] H. Ren, D. Anicic, and T. Runkler, "TinyOL: TinyML with online-learning on microcontrollers," 2021, *arXiv:2103.08295*. [Online]. Available: http://arxiv.org/abs/2103.08295

[24] S. Disabato and M. Roveri, "Incremental on-device tiny machine learning," in *Proc. 2nd Int. Workshop Challenges Artif. Intell. Mach. Learn. Internet Things*, Nov. 2020, pp. 7–13.

[25] S. Benatti, F. Montagna, V. Kartsch, A. Rahimi, D. Rossi, and L. Benini, "Online learning and classification of EMG-based gestures on a parallel ultra-low power platform using hyperdimensional computing," *IEEE Trans. Biomed. Circuits Syst.*, vol. 13, no. 3, pp. 516–528, Jun. 2019.

[26] D. Maltoni and V. Lomonaco, "Continuous learning in single-incremental-task scenarios," *Neural Netw.*, vol. 116, pp. 56–73, Aug. 2019.

[27] F. M. Castro, M. J. Marín-Jiménez, N. Guil, C. Schmid, and K. Alahari, "End-to-end incremental learning," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 233–248.

[28] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, "ICaRL: Incremental classifier and representation learning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 2001–2010.

[29] T. L. Hayes, N. D. Cahill, and C. Kanan, "Memory efficient experience replay for streaming learning," in *Proc. Int. Conf. Robot. Autom. (ICRA)*, May 2019, pp. 9769–9776.

[30] L. Caccia, E. Belilovsky, M. Caccia, and J. Pineau, "Online learned continual compression with adaptive quantization modules," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 1240–1250.

[31] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 246–247.

[32] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," Mar. 2017, *arXiv:1703.09039*. [Online]. Available: http://arxiv.org/abs/1703.09039

[33] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," Feb. 2016, *arXiv:1510.00149*. [Online]. Available: http://arxiv.org/abs/1510.00149

[34] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[35] M. Le Gallo *et al.*, "Mixed-precision in-memory computing," *Nature Electron.*, vol. 1, no. 4, pp. 246–253, Apr. 2018.

[36] M. Zemlyanikin, A. Smorkalov, T. Khanova, A. Petrovicheva, and G. Serebryakov, "512 KiB RAM is enough! live camera face recognition DNN on MCU," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. Workshop*, Oct. 2019, pp. 1–8.

[37] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm cortex-M CPUs," 2018, *arXiv:1801.06601*. [Online]. Available: http://arxiv.org/abs/1801.06601

[38] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*. Carlsbad, CA, USA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[39] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti, "DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs," *IEEE Trans. Comput.*, vol. 70, no. 8, pp. 1253–1268, Aug. 2021, doi: 10.1109/TC.2021.3066883.

[40] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 5, pp. 871–875, May 2020.

[41] J. Cheng, J. Wu, C. Leng, Y. Wang, and Q. Hu, "Quantized CNN: A unified approach to accelerate and compress convolutional networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 10, pp. 4730–4743, Oct. 2018.

[42] S. Ghamari *et al.*, "Quantization-guided training for compact tinyML models," 2021, *arXiv:2103.06231*. [Online]. Available: http://arxiv.org/abs/2103.06231

[43] L. Cecconi, S. Smets, L. Benini, and M. Verhelst, "Optimal tiling strategy for memory bandwidth reduction for CNNs," in *Advanced Concepts for Intelligent Vision Systems*, J. Blanc-Talon, R. Penne, W. Philips, D. Popescu, and P. Scheunders, Eds. Cham, Switzerland: Springer, 2017, pp. 89–100, doi: 10.1007/978-3-319-70353-4_8.

[44] T. Moreau *et al.*, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, Sep. 2019.

[45] D. Kalamkar *et al.*, "A study of BFLOAT16 for deep learning training," 2019, *arXiv:1905.12322*. [Online]. Available: http://arxiv.org/abs/1905.12322

[46] X. Sun *et al.*, "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 4900–4909.

[47] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 240–241.
[48] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 269–284, Feb. 2014.
[49] J. Shin, S. Choi, Y. Choi, and L.-S. Kim, "A pragmatic approach to on-device incremental learning system with selective weight updates," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.
[50] D. Han *et al.*, "HNPU: An adaptive DNN training processor utilizing stochastic dynamic fixed-point and active bit-precision searching," *IEEE J. Solid-State Circuits*, vol. 56, no. 9, pp. 2858–2869, Sep. 2021.
[51] S. Kang *et al.*, "GANPU: A 135 TFLOPS/W multi-DNN training processor for GANs with speculative dual-sparsity exploitation," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2020, pp. 140–142.
[52] J. L. Lobo, J. D. Ser, A. Bifet, and N. Kasabov, "Spiking neural networks and online learning: An overview and perspectives," *Neural Netw.*, vol. 121, pp. 88–100, Jan. 2020.
[53] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018.
[54] J. Pei *et al.*, "Towards artificial general intelligence with hybrid Tianjic chip architecture," *Nature*, vol. 572, no. 7767, pp. 106–111, Aug. 2019.
[55] G. Karunaratne *et al.*, "Robust high-dimensional memory-augmented neural networks," *Nature Commun.*, vol. 12, no. 1, p. 2468, Apr. 2021.
[56] 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.
[57] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini, "A transprecision floating-point architecture for energy-efficient embedded computing," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.
[58] I. Loi, A. Capotondi, D. Rossi, A. Marongiu, and L. Benini, "The quest for energy-efficient i$ design in ultra-low-power clustered many-cores," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 2, pp. 99–112, Apr./Jun. 2017.
[59] V. Lomonaco, D. Maltoni, and L. Pellegrini, "Rehearsal-free continual learning over small non-I.I.D. batches," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2020, pp. 989–998.
[60] F. Conti, "Technical report: NEMO DNN quantization for deployment model," 2020, *arXiv:2004.05930*. [Online]. Available: http://arxiv.org/abs/2004.05930

Leonardo Ravaglia received the M.Sc. degree in automation engineering from the University of Bologna in 2019, where he is currently pursuing the Ph.D. degree in data science and computation. His research interests include DNN algorithms for continual learning, parallel computing on ultra-low power devices, and quantized neural networks.

Manuele Rusci received the Ph.D. degree in electronic engineering from the University of Bologna in 2018. He is currently a Post-Doctoral Researcher with the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi," University of Bologna, and closely collaborates with GreenWaves Technologies. His main research interests include low-power embedded systems and AI-powered smart sensors.

Davide Nadalini received the M.Sc. degree in electronic engineering from the University of Bologna in 2021. He is currently a Research Fellow at the University of Bologna. His main research topics are hardware–software co-design and optimization of embedded artificial intelligence. His research interests include parallel computing, quantized neural networks, and low-level optimization.

Alessandro Capotondi (Member, IEEE) received the Ph.D. degree in electrical, electronic, and information engineering from the University of Bologna in 2016. He is a Post-Doctoral Researcher with the Università di Modena e Reggio Emilia (IT). His main research interests focus on heterogeneous architectures, parallel programming models, and TinyML.

Francesco Conti (Member, IEEE) received the Ph.D. degree in electronic engineering from the University of Bologna, Italy, in 2016. From 2016 to 2020, he was a Post-Doctoral Researcher with the Integrated Systems Laboratory, Digital Systems Group, ETH Zürich, and held a research grant with the DEI Department, University of Bologna. He is currently an Assistant Professor with the DEI Department, University of Bologna. His research is focused on the development of deep learning-based intelligence on top of ultra-low power and ultra-energy efficient programmable systems-on-chip. His research work has resulted more than 60 publications in international conferences and journals and has been awarded several times, including the 2020 IEEE TCAS-I Darlington Best Paper Award.

Luca Benini (Fellow, IEEE) received the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 1997. He was the Chief Architect of the Platform 2012/STHORM Project with STMicroelectronics, Grenoble, France, from 2009 to 2013. He held visiting/consulting positions with the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland; Stanford University; and IMEC, Leuven, Belgium. He is currently a Full Professor with the University of Bologna, Bologna, Italy. He is also the Chair of Digital Circuits and Systems with ETH Zürich, Zürich, Switzerland. He has authored over 700 papers in peer-reviewed international journals and conferences, four books, and several book chapters. His current research interests include energy-efficient system design and multicore system-on-chip design. He is a member of Academia Europaea.