DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs

Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Member, IEEE, Davide Rossi, Francesco Conti, Member, IEEE

Abstract—The deployment of Deep Neural Networks (DNNs) on end-nodes at the extreme edge of the Internet-of-Things is a critical enabler to support pervasive Deep Learning-enhanced applications. Low-Cost MCU-based end-nodes have limited on-chip memory and often replace caches with scratchpads, to reduce area overheads and increase energy efficiency – requiring explicit DMA-based memory transfers between different levels of the memory hierarchy. Mapping modern DNNs on these systems requires aggressive topology-dependent tiling and double-buffering. In this work, we propose DORY (Deployment Oriented to memoRY) – an automatic tool to deploy DNNs on low-cost MCUs with typically less than 1MB of on-chip SRAM memory. DORY abstracts tiling as a Constraint Programming (CP) problem: it maximizes L1 memory utilization under the topological constraints imposed by each DNN layer. Then, it generates ANSI C code to orchestrate off- and on-chip transfers and computation phases. Furthermore, to maximize speed, DORY augments the CP formulation with heuristics promoting performance-effective tile sizes. As a case study for DORY, we target GreenWaves Technologies GAP8, one of the most advanced parallel ultra-low power MCU-class devices on the market. On this device, DORY achieves up to 2.5× better MAC/cycle than the GreenWaves proprietary software solution and 18.1× better than the state-of-the-art result on an STM32-F746 MCU on single layers. Using our tool, GAP 8 can perform end-to-end inference of a 1.0-MobileNet-128 network consuming just 63 pJ/MAC on average @ 4.3 fps – 15.4× better than an STM32-F746. We release all our developments – the DORY framework, the optimized backend kernels, and the related heuristics – as open-source software.

Index Terms—Deep Neural Networks, IoT, edge computing, DNN acceleration

1 INTRODUCTION

The Internet of Things (IoT) envisions billions of wireless-connected end-nodes [1], which can sense, process and transmit data for a wide range of applications such as surveillance [2], health monitoring [3], agriculture [4], robotics [5], and others. However, major challenges are linked to this new computation paradigm, including reliability, security, capacity, together with the production of high-bandwidth data. In this scenario, edge-based Deep Learning (DL) is an attractive approach thanks to its capability to extract high-level features from raw sensor data, reducing off-node transmissions, and improving security by doing most processing in-place.

Modern Deep Neural Network (DNN) inference tasks run on cloud servers, personal computers, or smartphones. Even in the most constrained scenario of mobile devices, their execution can count on GB of memory and significant processing power available, under a power envelope of a few watts. Conversely, deploying DNNs on a microcontroller-based IoT end-node has to deliver similar performance while dealing with i) strict constraints in terms of memory (a few MB off-chip, and typically 1MB on-chip at most), ii) limited computational capabilities, and iii) battery constraints and a peak power envelope of 100-200 mW. The deployment of DL-based algorithms on the IoT demands aggressive hardware, software, and algorithmic co-optimization to exploit the scarce resources on these systems to the maximum degree [6]. In particular, the scarce availability of memory constitutes a real Deep Learning Memory Wall [7]: a fundamental limitation to the maximum performance of an embedded DNN compute system.

Recently introduced algorithmic improvements such as quantized DNN inference [8] aim at matching a DNN’s full-precision accuracy while using exclusively 8-bit (or smaller) integer data, to reduce memory occupation and execution complexity. On the hardware side, accelerators [9], [10], [11] and ISA extensions [12] that exploit quantization have been introduced to speed up the computation, lessen the impact of memory constraints and minimize energy consumption. In essence, 8-bit networks are now supported by most of the frameworks, such as TensorFlow and PyTorch. Recently proposed architectural paradigms aim at maximizing DNN performance and efficiency on IoT end-nodes while safeguarding the flexibility of typical Microcontroller Unit (MCUs), so that common control-oriented MCU tasks can be mixed with DNNs and non-DL-based data processing tasks. These architectures often couple a conventional MCU with an accelerator [13], [14]. Parallel Ultra-Low-Power computing (PULP), for example, is an architectural paradigm based on flexible software-oriented acceleration for DNNs and other data processing tasks in multi-core end-nodes. The core idea of PULP is to couple an I/O-dedicated core with a multi-core cluster of processors optimized for data-parallel processing, sharing a high-bandwidth multi-core L1 [15].

Accelerated IoT end-nodes employ multi-level hierarchies of on- and off-chip memories. In some cases, they do away entirely with energy-expensive coherent data caches, exploiting manually managed scratchpad memories instead to maximize area and energy efficiency. For example, PULP architectures complement a small (~128kB) L1 with a
bigger-but-slower (∼1 GB/s) on-chip L2 memory, and by an off-chip L3 low-power IoT DRAM \cite{15} that provides high capacity, but at a slower speed (∼100 MB/s) and with relatively high energy penalty (> 50 pJ/B). These composite memory hierarchies are becoming necessary even in low-power systems to cope with the memory footprint of DNN inference, without paying the cost of huge on-chip caches. However, to “unlock” such a system’s theoretical performance often requires carefully managed data movement by means of cache locking or explicit DMA transfers. To reduce the related development overhead, software caches \cite{17} and data tiling strategies \cite{18} have been proposed; however, most DL-based applications can improve upon general-purpose solutions by exploiting the regular structure of DNNs, with ad-hoc memory management flows to minimize inference time. \cite{19}, exploit data reuse, and optimize scheduling. \cite{20} Conversely, automatic solutions for end-to-end deployment of real-world DNNs on MCUs so far rely either on slow and inefficient interpretation (e.g., TF-Lite Micro \cite{21}), or on proprietary code generation frameworks (e.g., ST XCUBE-AI \cite{22}, GW \cite{23} and AutoTiler \cite{23}).

In this paper, we introduce a novel lightweight framework called DORY, Development Oriented to memory, which aims at the deployment of end-to-end DNNs on memory-starved end-nodes, and particularly tuned to the class of end-nodes based on the PULP paradigm. As a case study, we target GWT GAP-8 \cite{24} – one of the most advanced low-power edge nodes available in the market, embodying the PULP architectural paradigm with DSP-enhanced RISC-V cores. We introduce several novel contributions:

1. A memory tiling methodology based on Constraint Programming (CP) optimization tool to match on- and off-chip memory hierarchy constraints with DNN requirements such as the relationships between input, weight, and output tensor dimensions.
2. A set of heuristics to maximize the performance of the CP solution on PULP platforms using the dedicated back-end library PULP-NN \cite{14}, to maximize throughput and energy efficiency in the RISC-V based GAP-8 target.
3. Generation of the adaptive programming logic to orchestrate off- and on-chip memory transfers and computation phases. This process combines the two-level memory transfers to almost completely hide the memory hierarchy.

We evaluate the performance and energy efficiency of the deployed networks produced by DORY on GWT GAP-8, considering both single layers and end-to-end networks. DORY achieves up to 18.1× better MAC/cycle than the state-of-the-art result on a conventional cache-based microcontroller, the STM32-F746 MCU, in single layer execution. Using DORY, end-to-end deployment of 8-bit quantized networks such as 1.0-MobileNet-v1-128, 1.0-MobileNet-v2-128 (64.4% and 65.2% Top1 accuracy, respectively) achieve 8.00 and 5.27 MACs/cycle respectively, with a 15.4× improvement compared to the same networks running on the STM32-F746 using the state-of-the-art ST X-CUBE-AI. Furthermore, on a layer by layer basis, DORY can achieve up to 2.5× better throughput than the proprietary GWT autotiler, on the same GAP-8 platform, and up to 27% better performance on full network execution. Our results show that image recognition on an extreme edge-node can run in as little as 11.9 mJ/classification @ 4.3 fps.

To foster further research on real-world deeply embedded DNN applications, we release the DORY framework, the optimized backend kernels, and the related heuristics discussed in this paper as open-source. \cite{32}

2. Related Work

DNN algorithm minimization

From the algorithmic viewpoint, the first task in DL deployment is making sure that the DNNs are “minimally redundant”, in the sense that they do not perform any additional operation unless it leads to a better quality-of-results. In this direction, a current research trend is to adapt DNN architectures to deployment in constrained platforms by shrinking the DNN topologies themselves, either directly \cite{25, 26} or using neural architecture search \cite{27, 28}. Orthogonally, techniques for post-training quantization \cite{29} and quantization-aware fine-tuning \cite{30} can be used to reduce the cost of single operations in terms of energy and of single parameters in terms of memory – trying to minimize the price in terms of quality-of-results.

Optimized software & ISA for DNN computation

Given a size-optimized and precision-tuned DNN, we need to address the deployment challenge, i.e., achieve maximal utilization of the computing units, while minimizing the performance and energy penalties associated with data transfers across the memory hierarchy. Application-specific hardware architectures are very useful in accelerating specific layers and, in some cases, entire networks \cite{9, 10, 11} – but their lack of flexibility can be a liability in a field such as DL, where every year researchers introduce tens of new topologies and different ways to combine the DNN basic blocks. To provide higher flexibility, in many cases, DNN primitives are implemented in highly optimized software instead of full-hardware blocks. Over the years, several software libraries of DNN kernels have been proposed \cite{14, 29, 31, 32} to maximize the efficiency of DNN execution with DSP-oriented single-instruction multiple-data (SIMD) ISA capabilities \cite{33}. These libraries leverage either the Height-Width-Channel (HWC) or Channel-Height-Width (CHW) data layout, to minimize operations and memory footprint. CHW optimizes data reuse in the spatial dimensions. Therefore, it is faster on convolutions with larger filters and lower channel connectivity; HWC naturally favors channel-wise data reuse, often requiring the construction of a flattened data structure (‘im2col’ buffer) to exploit spatial data reuse partially \cite{31}. Further, there is an increasing trend towards more targeted ISA specialization (e.g., ARM Helium \cite{34}, xPULPN \cite{12}) to support and accelerate the pervasive convolutional layers with low-bitwidth linear algebra instructions.

Memory hierarchy management

The most critical challenge in DNN deployment is memory hierarchy management, because modern DNNs generate high amounts of data traffic between different levels of the memory hierarchy, which may constitute a significant bottleneck. It is therefore essential to optimize the flow of execution and data transfer, exploiting data reuse and tiling to the fullest. A valuable example of a holistic methodology for optimizing the data flow is presented in dMazeRunner \cite{35}, which targets DNN inference on course-grained programmable accelerators. It explores a vast design space of execution methods to optimize loop nets and data transfers on dataflow accelerators. Academic researchers and industries have significantly investigated this aspect, by

1. GreenWaves Technologies.

2. https://github.com/pulp-platform/dory
including in their edge-node solutions either specialized caches (e.g., NXP [36]), or explicitly managed scratchpad memories (e.g., GWT [24]).

Major frameworks for DNNs have so far focused on cloud-scale training and inference, and only recently started to put attention on inference at the edge. Tensorflow Lite (TFLite) from Tensorflow creators is an open-source framework for the deployment of DNN models on top of mobile-class devices. A Tensorflow DNN model is converted into a compressed flat buffer and interpreted on-device. Similarly, Larg Compute Engine (LCE) [27] is a framework targeting the deployment of heavily quantized neural networks on edge mobile devices. LCE leverages optimization techniques such as tiling, vectorization, and multi-threading parallelization to speed-up the execution of inference tasks. On a Pixel1 phone and a Raspberry Pi4, LCE shows the inference of a QuickNet binary network in 20-120 milliseconds in a power envelope well above 1 watt. Both TFLite and LCE are not suitable for IoT MCUs because they require the target device to be capable of booting a full-fledged operating system, and they are not tuned to target low-cost low-power MCUs with less than 1MB of on-chip SRAM and just a few MB of off-chip memory.

**DNN-oriented microcontrollers and related tools**

Recently, the first generation of low-power neural-network oriented microcontrollers has been introduced, coupling optimized software and ISA extensions for DNN computing with “traditional” control and I/O-bound activities. A growing trend to enable optimal execution of both kinds of tasks is that of parallel and heterogeneous processing; for example, ST Microelectronics [35] and NXP [36] have recently introduced new-generation dual-core microcontrollers with an ARM M0 processor dedicated to I/O and an ARM M4 processor with single-cycle multiply-and-accumulate and SIMD capabilities. These platforms show an increased complexity in terms of memory hierarchy with respect to conventional flat-memory MCUs, with an L1 memory optimized for speed and an L2 optimized for capacity. At the same time, there is a trend towards explicit management of memory hierarchy, with hand-tunable data caches featuring locking for hand-crafted data management. To manage this complexity, these MCUs include dedicated infrastructure for data marshaling such as general-purpose DMA controllers to speed-up memory transfers and reduce the memory access bottleneck.

The **Parallel Ultra-Low-Power** paradigm [15] magnifies these industry-wide architectural trends: it introduces a multi-core accelerator instead of a single core to accelerate data parallel compute-bound code; and it removes data caches, replacing them with small on-chip scratchpad memories. The GreenWaves Technologies GAP-8 [24] system-on-chip that we use as a case study in this work was introduced in 2018 as a commercial embodiment of this paradigm: it features one I/O core and an 8-core SIMD-optimized DSP cluster using an extension of the RISC-V ISA.

Programming these DNN-oriented MCUs is typically more complicated with respect to conventional MCUs; new tools such as TFLite-Micro [21] and uTensor [39] have been developed to offer a modularity and diagnostics deployment framework. Both are non-vendor-locked framework supporting platforms based on ARM Cortex-M and RISC-V. Their runtimes take up only 16 kB on a Cortex-M3, and they come with implementations for floating-point as well as 8-bit integer layers. As shown in Section 5, however, TFLite-Micro is not fast enough to deploy modern DNN models of reasonable size. Similarly, FANN [40] is a library generating code for low-power MCUs based on ARM and RISC-V.

However, this library focuses exclusively on Multi-Layer Perceptrons (MLP) – a tiny subset of machine learning models.

To the best of our knowledge, the two most powerful DNN deployment tools available in the state-of-the-art have been proposed by the industry as proprietary, vendor-locked solutions for their own MCUs. X-CUBE-AI [22] from STMicroelectronics is an automatic NN library generator optimized on computation and memory. It converts a pre-trained DNN model from most used DNN tools available like Keras, and Tensorflow, into a compiled C-like library ready to be executed on top of the ARM Cortex-M cores embedded in STM32 series MCUs. It supports storage of weights on external memory to enable the deployment of real-world DNNs. On the other hand, GWT designed a tool called AutoTiler [23] to target their GAP-8 SoC, a RISC-V based multi-core ultra-low-power microcontroller. The AutoTiler has a wider scope than other strictly AI-focused tools, also targeting the deployment of traditional vision and signal processing algorithms. One of its primary functions, however, is to take a pre-trained DNN and generate code for memory tiling and efficient transfers of weight and activation data between all memory levels (on- and off-chip). The GWT AutoTiler tool is proprietary, but its background basic kernels are available as open-source as part of the GAP-8 SDK [3].

In Section 5, we perform several quantitative comparisons with the best results obtained with ST Microelectronics, GWT AutoTiler, and our own DORY.

### 3 Background

#### 3.1 Quantized Neural Networks

Post-training quantization [29] or quantization-aware training [30] produce as output a Quantized Neural Network (QNN). In a QNN, all tensors \(x\) (e.g., weights \(w\), inputs \(x\), or outputs \(y\)) are real-valued but \(i\) are discretized (they can assume only a countable set of values), and ii) are defined only in a specific range \([\alpha_i, \beta_i]\). In deployment, we can map these discretized tensors into purely integer ones, which we call integer images of the discretized tensor. In this work, we start from QNNs already discretized using linear, uniform per-layer quantization: the \(N\)-bit integer image \(\hat{t}\) of tensor \(t\) is connected to its real-valued quantized counterpart through a bijective mapping:

\[
t = \epsilon_t + \hat{t},
\]

where \(\epsilon_t = (\beta_t - \alpha_t)/(2^N - 1)\). We call \(\epsilon_t\) the quantum because it is the smallest amount that we can represent in the quantized tensor.

Each QNN layer is composed of a sequence of three operators: Linear, Batch-Normalization (optionally) and Quantization/Activation. Without loss of generality, we consider that \(\alpha_x = \alpha_y = 0\) for all the inputs of Linear and the outputs of Quantization/Activation operator. Using Eq. 1 all operators are mapped in the integer domain:

**LIN**:

\[
\varphi = \sum_n w_{m,n} x_n \iff \hat{\varphi} = \sum_n \hat{w}_{m,n} \cdot \hat{x}_n
\]

**BN**:

\[
\varphi = \kappa \cdot \varphi + \lambda \iff \hat{\varphi} = \hat{\kappa} \cdot \hat{\varphi} + \hat{\lambda}
\]

Note that the dot product operation in Eq. 2 results in a shrinking of the quantum used to represent \(\varphi\), which will be

3. [https://github.com/GreenWavesTechnologies/gap-sdk](https://github.com/GreenWavesTechnologies/gap-sdk)
4. If the original activation is a ReLU, then the QNN automatically satisfies this condition; otherwise, it can be transformed to satisfy it.
5. In inference, the statistical and learned parameters of BN can be combined: \(\kappa = \gamma/\sigma\) and \(\lambda = \beta - \mu_\gamma/\sigma\).
efficient digital signal processing, with hardware loops, post-modified access LD/ST, and SIMD instructions down to 8-bit vector operands.

The cores of the cluster share a first level of memory, a 64 kB multi-banked L1 memory Tightly-Coupled Data Memory (TCDM), accessible from the cluster’s cores through a high-bandwidth, single-cycle-latency logarithmic interconnect, featuring a 2x banking factor and a word-level interleaving scheme to reduce the probability of contention [32]. In order to manage data transfers between the L1 TCDM memory and a second-level 512 kB of memory (managed as a scratchpad as well) available in the SoC domain, the cluster DMA [44] can manage data transfers between L1 and L2 with a bandwidth up to 2 GBytes/s and a latency of 80ns at the maximum frequency. On the other hand, to interface the L2 memory with the external world, and in particular with the Cypress Semiconductor’s HyperRAM/HyperFlash module [16] available on the GAPuino board, GAP-8 can use an autonomous I/O subsystem called I/O DMA [45]. Through the HyperBus interface, the external L3 HyperRAM and/or HyperFlash memory can be connected to the system, enabling a further 64 MB of storage for read-only data on Flash and 8-16 MB for volatile data on DRAM, with a bandwidth up to 200 Mbytes/s.

### 3.3 QNN Execution Model on GAP-8

Computational backends are by construction tied to a specific target platform as they need to fully exploit the architecture’s strength. As optimized QNN backend for our GAP-8 case study, we relied on the open-source PULP-NN [14] library. PULP-NN is based on the HWC data layout. An efficient QNN layer is implemented in the backend library as a combination of three phases, summarily shown in Figure 2: first, the lm2Col step copies the pixels needed to produce a single output pixel (i.e., the receptive field) from their 3-D input non-sequential in memory arrangement into a 1-D vector using load/store operations. Note that this step is not performed for 1×1 convolutions, since all the necessary input pixels (1×1×Cw) are already sequential in memory, given the HWC data layout. Then, the linear part of the kernel, the Matrix Multiplication (MatMul), convolves the current 1-D vector with the weight parameters of the layer, exploiting the RI5CY SIMD instructions to implement the integer part of Eq. 2. To improve performance, the innermost loop of the MatMul accumulates the partial results of the convolution over registers, eliminating the store instructions inside the loop and reusing the 1-D input vector elements along with 4 different sets of filters. This enables the computation of 2 adjacent output pixels in parallel, thus maximizing reuse and reducing the cost of loads. In this way, the innermost loop consists of just 6 load (ld) instructions and 8 SIMD MAC instructions (sdotp), for a total of 32 MACs per loop iteration. In this work, we extended the PULP-NN [14] library to support also Batch-Normalization and Quantization/Activation as defined in Eqs. 5 and 6, respectively, which together compose the Norm/Qnt phase. Note that the PULP-NN library assumes that all the activations and weights are stored in the L1
memory. Readers may refer to [14] for detailed information about this library.

3.4 QNN Tensor Tiling

In the context of QNN deployment, a tiling strategy consists of a regular software-managed fragmentation of the data tensors mentioned in Section 3.1 to i) fit within the available memory, and ii) transparently move data between levels, using double buffering and DMA of the next tile in parallel with computation on the current tile. In this work, we target a hardware architecture with three levels of memory hierarchy: a virtually unlimited-size off-chip L3; an on-chip L2 memory balancing size (e.g., 256 kB to a few MB) and bandwidth; and an on-chip L1 with virtually unlimited bandwidth to the compute units, but of limited size (typically <128 kB).

If we consider a convolutional layer in a DNN, in general, inputs, outputs, and weights should all be tiled to satisfy memory constraints at all levels Li (see Table 1 for the notation adopted throughout the paper). The main challenge of tiling is to maximize the size of all tiles while i) fitting within the size constraints imposed by the size of layer Li, and ii) guaranteeing that all relationships between the tensors are respected both on the tiles in Li and on the full tensors in L(i + 1).

4 DORY: DEPLOYMENT ORIENTED TO MEMORY

DORY targets a compute node with three levels (L3, L2, and L1) in the memory hierarchy, as described in Section 3. It supports L3-L2 and L2-L1 tiling of both weights and activations. Storage of weights in L3 (> 512 KB) is essential for the deployment of most non-trivial networks such as [25], [26]. On the other hand, activations’ tiling is typically necessary only for networks working on high-resolution images with big spatial dimensions, which are rare in the edge computing domain. The operation of DORY is organized in three steps, performed offline before network deployment. First, the ONNX decoder receives as input a QNN graph using the Open Neural Network Exchange (ONNX format). Then, the layer analyzer optimizes and generates code to run the tiling loop, orchestrate layer-wise data movement and call a set of backend APIs to execute each layer of the network, individually. Finally, the network parser merges information from the whole network to infer memory buffer sizes in each hierarchical level and orchestrate the end-to-end network execution. It uses this information to generate an ANSI C file that embodies the whole DNN execution and can be compiled for the target platform.

Listing 1. DORY L2-L1 loop nest implementing the double buffering scheme as represented in right part of Figure 3. At each most internal loop iteration, two asynchronous Cluster DMA calls are made to copy the weights and input activation of the next tile into L1 memory, the basic kernel is executed on the current tile, and one other cluster DMA transfer is executed to copy the output back on the L2 memory.

4.1 ONNX Decoder

The first operation performed by DORY is decoding the input ONNX graph representing an already quantized DNN, and reorganizing it in a set of layers. In DORY, a layer corresponds to a canonical sequence of operations performed by distinct ONNX graph nodes. Each layer includes i) a Linear/add/pooling operation, ii) an optional Batch-Normalization operation, iii) a Quantization/Activation operation, and iv) a SW-cache generator.

4.2 Layer Analyzer

In the first optimization phase, DORY layers are considered separately from each other, using only weight dimension information from the previous layer. The layer analyzer includes three submodules: a platform-agnostic tiling solver, a set of heuristics optimizing execution over a target-specific backend; and a SW-cache generator.

4.2.1 DORY Tiling Solver

In the following discussion, we use the terminology defined in Section 3 and denote a buffer residing in Li memory as Li,b, where i is the name of the tensor. The Solver relies on a 2-step engine, which solves the L3-L2 tiling constrained problem, and then the L2-L1 one.

The Solver verifies whether the layer memory occupation fits the L2 memory or needs to be stored in L3:

\[ L_{2w,next} + L_{2w,curr} + L_x + L_{2y} \leq L_2. \]  (5)
If this condition is not satisfied, a four stage L3 tiling is executed. At each stage, different buffers are tiled to try to fit the above constraint. If it is satisfied with the produced tiles, the L3-L2 Tiling Solver is stopped, and the dimensions of tiles are saved. Otherwise, the next stage begins. The starting point is L3-tile x, w, y = OFF, OFF, OFF:

- **Stage 1.** L3-tile x = ON. If the output of the previous layer has been tiled from L3, the input activation tiling is enabled; the tiling is performed on the \( h_x \) dimension, maximizing the L2 buffer’s memory occupation. The tiler effectively splits the layer in a series of identical ones that work on a different stripe of the input image;

- **Stage 2.** L3-tile w = ON. Weight tiling is enabled on the \( C_y \) dimension, dividing the layer in a set of smaller layers that work on different channels of the output image with \( C'_y < C_y \), each separately tiled between L2 and L1;

- **Stage 3.** L3-tile w = OFF, L3-tile y = ON. The weight tiling is disabled while the output one is enabled: the same input tiling approach is used, with identical reasoning on dimension \( h_y \);

- **Stage 4.** L3-tile w = ON, L3-tile out = ON. The L3 tiling is enabled on all buffers \((x, y, weights)\).

After the L3 tiling step, the DORY solver processes the layer to find a suitable L2-L1 tiling scheme, which requires more effort due to the typically small sizes of L1 memories. DORY abstracts this as a Constraint Programming (CP) problem, and exploits the CP solver from the open-source OR-Tools developed by Google AI to meet hardware and geometrical constraint (e.g., \( C'_y \) for output and weights must be the same), while maximizing an objective function. The base objective function of the solver is to maximize L1 memory utilization:

\[
\max(L_{1x} + L_{1y} + L_{1w})
\]

manipulating the tile dimensions, e.g., \( C'_y, C'_w \). The hardware constraint is related to the max L1 buffer dimensions:

\[
L_{1x} + L_{1y} + L_{1w} + L_{1backend} < \frac{L_1}{2}.
\]

with \( L_{1backend} \), the overhead of the backend kernel, e.g., the im2col memory occupation of PULP-NN backend [14] if present (i.e., for non-1 x 1 convolutions) or intermediate full-precision accumulators for CHW based convolutions.

6. https://developers.google.com/optimization/

Topological and geometrical constraints are due to the relationships between each tensor’s characteristic dimensions and other parameters of a layer; for example,

\[
h'_y = \left(h'_x - (K_h - 1) + 2 \cdot p\right)
\]

embodies the relationship between the height dimension in the output and the input tiles, with \( p \) representing padding.

### 4.2.2 Target-specific Heuristics & Constraints

To maximize performance, the objective function of Eq. (6) can be augmented with a series of heuristics targeting a specific backend. The heuristics are combined with the objective function of Eq. (6) by means of a set of tweakable parameters:

\[
\max\left(\alpha(L_{1x} + L_{1y} + L_{1w}) + \sum_i \beta_i H_i\right) .
\]

Here, we list four heuristics related to PULP-NN, the backend library exploited by DORY in our GAP-8 case study.

- **HIDE.IM2COL**: the PULP-NN im2col buffer is reused for each output pixel; therefore, maximizing the number of output channels optimizes the reuse of input pixels, reducing the overhead to create the im2col:

\[
H_{im2c} = C'_y
\]

- **PAR_BALANCE**: PULP-NN divides workload among cores following primarily the \( h \) dimension (i.e., a chunk of rows per core). Therefore, making this a multiple the number of cores \( 8 \) maximizes balance:

\[
H_{par} = (h'_y - 1) \mod 8
\]

- **MATMUL.W** and **MATMUL.CH**: the innermost loop of PULP-NN is a 4x2 matrix multiplication on 4 output channels and 2 pixels in \( w \) direction. Maximizing adherence of a tile to this scheme optimizes performance:

\[
H_{mmw} = (w'_y - 1) \mod 2 \ ; \ H_{mmch} = (C'_y - 1) \mod 4
\]

Section 6.1 discusses the effectiveness of the PULP-NN heuristics in delivering a good quality-of-results. Additionally, Section 6.1 describes the impact of applying these heuristics both to the main tiling problem and to the sizing of the layer borders tile.

7. The PAR_BALANCE constraint is changed to \( H_{par} = (b'_y \times w'_y - 1) \mod 16 \) for “pathological” output activations with \( h'_y < 8 \).
We also impose an additional constraint to always perform a full computation along the channel direction:

\[ C_x^t = C_x \].

This is because PULP-NN does not allow for intermediate storage of partially computed outputs. While this restricts the solution space significantly, we observe that most of the purged solutions would anyway be of low quality. In essence, the intermediate storage would have to happen at 32-bit precision, forcing the usage of much smaller tiles.

### 4.2.3 DORY SW-cache Generator

The SW-cache Generator is charged of automatically generating C code orchestrating the execution of a whole layer given the tiling solution found by the Tiling Solver. It instantiates asynchronous data transfers and calls to the backend kernels, without any manual effort. DORY uses a *triple-buffering* approach for the communication between L3-L2 and L2-L1 memories: specifically, double-buffering is applied simultaneously between L3-L2 and L2-L1 (Figure 3), and all data transfers are pipelined and asynchronous. With this approach, we can almost completely hide the memory transfer overhead, as discussed in Section 4. While the code generator is necessarily not platform-agnostic, the approach we follow can be easily generalized to any computing node with a three-level memory hierarchy.

Listing 1 provides DORY’s scheduling scheme of L2-L1 layer execution, through LTO, LTW, LTH, and LTI loops on output channels, height, width, input channels tiles, respectively. Loop iteration limits are statically resolved by the DORY tiling Solver. Moreover, DORY autonomously controls the complete execution of the layer, by managing padding, stride, and overlap for every single tile (e.g., padding > 0 for border tiles whereas padding = 0 for internal ones, when the input padding parameter is > 0). Using statically resolved parameters, we maximize the usage of immediates, reducing load/store operations inside the inner loops of the layer tiling.

The layer-wise loop nest detailed in Listing 1 and Fig. 3 is executed in three concurrent pipeline stages: i) a new computation starts and fill the output buffer that was not used in the previous cycle; ii) the results of the last cycle are stored back in L2; iii) a new set of inputs is loaded in L1. At each pipeline cycle, we swap the load and the execution buffer (swap operation of Listing 1) to enable double buffering.

### 4.3 DORY Hybrid Model

In the HWC data layout, used by CMSIS-NN [31] and PULP-NN [14], pixels referring to channels are contiguous, while spatially adjacent ones are stored with stride > 1. This layout enables constructing very optimized convolutional layers out of a single optimized matrix-multiplication kernel, by exploiting the reuse of activations over input channels [14, 31] – contrary to the CHW layout, which requires separately handcrafted and optimized kernels for each kernel size/stride configuration. The main limit of this approach hits a specific category of convolutional layers, namely, depth-wise convolutions. These do not accumulate over multiple channels; instead, they project each input channel into a single output channel disjointly from other channels. Therefore, they do not show any possibility to exploit channel data reuse.

On the one hand, depth-wise convolutions are unavoidable in modern networks for the edge, to decouple the channel-mixing and spatial filtering actions of the convolutional layer [26]; on the other hand, they are typically only responsible for 10% or less of the overall operations [25, 26], meaning that directly optimizing for them may be suboptimal. This scenario suggests a hybrid approach: using the HWC layout for general convolutional layers (and pointwise 1x1 layers), but switching to a hybrid CHW/HWC layout in depth-wise layers.

Following this idea, we define new optimizations for existing layers and a new depth-wise convolution that consumes and produces activations in HWC layout from L2/L3 memory, but reorders them in CHW layout on L1 to maximize the data reuse and, therefore, computational efficiency. Specifically, multiple strided Cluster DMA transfers are used to marshal data from L2 converting it directly from the HWC to CHW layout. An Im2Col buffer is constructed simply as a contiguous vertical stripe of width \( K_w \); the innermost loop proceeds along the vertical stripe by computing a single output pixel per iteration. The output pixels are then quantized and stored in an output buffer using the HWC layout, which can be directly transferred to L2. Figure 3 shows the execution model adopted for depthwise convolutions. With this strategy, input data reuse – the only kind available in depth-wise convolutions – can be exploited along the vertical dimension, thanks to the fact that spatially adjacent pixels are contiguous in memory. For parallel execution, multiple cores operate simultaneously on different channels; due to the channel independence, this choice minimizes memory contention, and optimizes performance while still keeping a degree of flexibility: the same kernel can be used to compute depth-wise layers of various filter shapes and strides.

### 4.4 Network Parser

After layer-wise tiling has been completed by the Layer Analyzer, DORY uses the information extracted from all the layers to build a network graph, considering every single layer as a call able function. Fig. 3 and Listing 2 showcase the structure of a whole graph as created by our framework. At each step, three main tasks are concatenated: i) we transfer from L3 the weights of the following layer; ii) a new layer is executed pointing to the correct buffers inside the memory stack; iii) input and output buffer offsets are updated.

8. This phase is executed for layer \( i \) only if layer \( i+1 \) is a convolution or a linear one and if it fits the dedicated space in the L2 memory. On the contrary, only the space for the \( L_2w \) is allocated if the layer needs the L3-L2 tiling and no space at all is allocated if the layer \( i+1 \) is a pooling or an add.
Similarly to single layers, the network-wise code is generated automatically without programmer intervention. DORY produces a single function that can be called inside a custom application by passing two externally allocated memory buffers (for L1 and L2) and their maximum size as parameters.

4.4.1 Buffer allocation stack & Residual connections

To allocate layer-wise input and output buffers in the L2 memory, we extend the two-stack strategy proposed by Palossi et al. [5], employing a strategy based on a single bidirectional stack designed to avoid memory fragmentation and enable the execution of a sequence of differently sized layers. Buffers are allocated/deallocated from the buffer allocation stack, which is constituted by two concurrent Last-In-First-Out stacks growing in opposite directions. At the end of each layer’s weight buffer allocation, we reverse the end of the stack for the next memory allocations. By construction, the bidirectional stack is at worst as big as two concurrent stacks growing in the same direction. For example, in a simple case without residual connections the dimension of our bidirectional stack is

\[
D_{stack} = \max_i (L_{x,i} + 2L_{w,i} + L_{w,i+1} + L_{x,i+1}),
\]

which is always less or equal than the size of two concurrent stacks \(D_{stack,1}, D_{stack,2}\) due to the triangle inequality.

Before executing the \(i\)-th layer, the allocator manages the weight buffer \(L_{2,wi}\) and output buffer \(L_{2,yi}\); notice that \(L_{2,yi}\) is already allocated as the \(L_{2,yj}\) of a previously executed \(j\)-th layer (or the input of the network). To manage residual connections, each \(L_{2,yi}\) buffer has a lifetime counter associated. To allocate a buffer in the stack for the \(i\)-th layer:

1. one of the two corners of the stack is selected depending on a begin_end flag that is switched at each new weight allocation;
2. the allocator deallocates the last \(L_{2,wi-2}\) buffer on the corner;
3. the allocator checks if \(L_{2,wi-2}\) has its lifetime counter set to 0; if so, it is deallocated;
4. \(L_{2,yi}, L_{2,wi}\) are allocated in order in the selected corner (with \(L_{2,wi}\) nearest to the pointer);
5. the lifetime counter of \(L_{2,yi}\) is set to the lifetime of the activation buffer, i.e., the number of layers to be executed before its deallocation;
6. all lifetime counters are decreased by 1.

The buffer allocation stack is naturally suited to execute a network with different branches (i.e., residual connections). DORY always prioritizes the branch with the highest number of nodes. The overall size of the stack is computed offline statically, taking into account all residual connections: its dimension depends on the maximum sum of memory of two subsequent layers plus all the residuals from the previous layers.

5 RESULTS

In this section, we evaluate DORY in terms of quality-of-results (performance and energy efficiency) on both single layers and full networks, using GWT GAP-8 as a target platform for our exploration and our extended PULP-NN library as a backend. We also compare our results with those obtained on a STM32-F746 MCU using STM X-CUBE-AI and on the same GAP-8 platform using the proprietary AutoTiler tool. The results on single layers refer to a full 8-bit QNN layer as defined in Section 3.1 with Linear, Batch-Normalization, and Quantization/Activation sub-layers. We set \(\alpha\) to 0.5, \(\beta_{i\text{BIDE}_{\text{IM2COL}}}\) to \(10^2\), and other \(\beta_i\) to \(10^0\) in the objective function.

5.1 Single layer performance & SoA comparison

In this section, we analyze the impact of the DORY optimizer on the execution of an entire layer, including both the unavoidable processing overhead to perform I/O DMA and Cluster DMA calls, and the data transfer overhead from imperfect pipelining. Fig. 6 analyzes all the execution time for two layers of MobileNet-v1 [25], the first representative of point-wise convolutional layers, the second of depth-wise ones. Other layers showed similar patterns and were not included in the plot to preserve space. We observe several effects. First, in both cases, the vast majority of time is spent in the backend PULP-NN kernel. For the point-wise layer, roughly all the time is spent in the innermost loop of MatMul (most of which is pure MAC operations); the rest is due to building the Im2Col buffer, Norm/Qnt and MatMul loops that cover the SIMD leftover cases (e.g.,
In this Section, we focus on the performance of DORY in deployment of two popular image detection networks end-to-end: MobileNet-v1 \cite{25} and MobileNet-v2 \cite{26}. These networks are used as benchmarks for many edge-oriented works \cite{29}. They include many of the topological characteristics of modern networks: convolution, depth-wise convolution, pooling, fully-connected layers, residual connections. We chose these networks as they are popular representatives of the modern image classification nets. Here, we focus on the specific configurations with 1.0 \textit{width multiplier} and 128x128 input frames (1.0-MobileNet-128 and 1.0-MobileNet-v2-128, respectively). Both networks were run on GWT GAP-8 verifying all intermediate results as well as the final result of end-to-end runs against a PyTorch-based bit-accurate golden model for QNNs \cite{31}, to confirm the correct functionality of the DORY framework and the PULP-NN backend.

### 5.2.1 End-to-end MobileNet-v1 and -v2 & SoA comparison

Table 4 highlights characteristics of MobileNet-v1 and -v2, along with results of DORY-based execution. While MobileNet-v1 has 1.98× the number of MobileNet-v2 MACs, it is only 22.6% slower. The second network has a higher prevalence of depth-wise convolutions (6.7% vs. 3.06%), which reduces the number of parameters and total MACs, but also leads to generally lower reuse and, thus, efficiency. On the other hand, the different topology of the network, with the insertion of residual connections, does not cause any slow down in execution.

Table 3 showcases a full comparison in terms of energy efficiency (GMAC/s/W), throughput (GMAC/s), latency, and energy per frame. For what concerns X-CUBE-AI, we partially extrapolated data from Capotondi et al. \cite{29} with the help of the authors. Different variations of the MobileNet-v1 have been compared, with the same topology but a different number of channels or input dimensions, to fit the on-chip memory of the STM32H7 for activations. As can be noticed from the Table, DORY on MobileNet-v1...
achieves up to $15.38 \times$ higher throughput in MAC/cycles than the execution on an STM32H7, using the best framework (X-CUBE-AI) currently available. On different operating points, we have up to $8.3 \times$ throughput (2.08 vs. 0.25 GMAC/s) and 14.8 $\times$ better energy efficiency, demonstrating the suitability for a wide range of applications on the edge of the network. Comparing both GAP8 and the STM32H7 using single core execution, we see that DORY is up to $3.2 \times$ (1.67 vs 0.52) faster. This result strongly suggests that the accurate offloading of neural-network computation to a co-processor with specialized instruction and optimized memory access can lead to important performance enhancement, implying that dedicated neural accelerators such as the cloud ones (e.g., TPUs) can also be the future in edge-nodes. We stress here that since the technique that DORY uses is not dependent on the specific target, this result could be extended to different platforms (e.g., NXP and STM32 dual-core M0/M4), using tiling and correct offloading to improve the cache friendliness of DNN primitives and exploit optimized ISAs.

On the same platform, the GAP-8 processor, our solution on the right, number of MAC operations, average power, and time for each layer of the network. Power was sampled at 64 KHz and then filtered with a moving average of 300 $\mu$s.

**Fig. 8.** In the left part, the 1.0-Mobilenet-128 power profile when running on GAP-8 @ $f_{\text{cluster}} = f_{\text{clk}} = 100$ MHz and $V_{DD} = 1.1$ V. On the right, parameters fitting 2MB ROM + 512 KB R/W RAM [29].

| TABLE 4 |
| --- |
| Analysis in terms of complexity, memory footprint, and latency of MobileNet-v1 and MobileNet-v2. |

| MobileNet-v1 (width mult 1.0, input 128x128) | Parameters | Cycles |
| --- | --- | --- |
| Conv 1$x$1 | 96.19% | 49.61% |
| DwConv 3$x$3 | 3.03% | 23.15% |
| FC | 0.18% | 36.20% |
| Add | - | 19.0M |
| Total | 186.4M | 23.3M |

| MobileNet-v2 (width mult 1.0, input 128x128) | Parameters | Cycles |
| --- | --- | --- |
| Conv 1$x$1 | 91.94% | 67.26% |
| DwConv 3$x$3 | 6.07% | 28.34% |
| FC | 1.30% | 36.20% |
| Add | - | 8.37% |
| Total | 100.1M | 19.0M |

5.2.2 *In-depth analysis of MobileNet-v1 execution*

Fig. depicts the power profile of the end-to-end execution of a MobileNet-v1 on GAP-8, with both the cluster and the fabric controller running at 100 MHz. The power consumption of the cluster domain (including 8 RISCy cores, the L1 and the Cluster DMA) and of the I/O domain (including 1
6 ABLATION STUDY

This section presents a detailed ablation study to investigate the separate impact of each of our contributions against state-of-the-art baselines. We first analyze the impact of the heuristics and their application to border tiles. We then show how DNN tensor tiling with DORY can effectively enable the execution of realistic DNN workloads even in memory-starved MCUs, with a comprehensive exploration of memory configurations for PULP-based systems.

6.1 Single tile performance

We analyze the effects that the heuristics proposed in Section 4.2.2 have on the quality-of-results of the tiling solution. Moreover, we show the effect of applying these techniques to the border tile, increasing the performance in different configurations. In particular, the size of the tile influences the execution efficiency of the backend layer. As such, a sub-optimal tiling choice can significantly reduce performance.

Fig. 9. Example of the effect of heuristic optimizations on convolutional layer performance. In this case, the "optimal" tile has output tensor $24 \times 4 \times 32$ (HWC) and weight tensor $32 \times 3 \times 3 \times 32$ (CoHWCi). Different optimizations are showed by varying $w_y$, $h_y$, and $C_y$ and violating the heuristics of Section 4.2.2.

Fig. 10. MobileNet-v1 execution with 512KB L2 memory and 22KB-44KB L1. The tiling configurations with Border Heuristic enabled (blue) and disabled (red) are compared.

RISCY core, the L2, and the I/O DMA) is shown separately in two separate subplots. In the cluster domain, power is dominated by the cores when the computation is in the active phase. Small valleys within a layer are given by (short) waits for the end of a memory transfer where the cores are all idle, or by Cluster DMA calls where a single core is active. In the I/O domain, we can notice the I/O DMA consumption spikes: at the beginning of each layer, the weights of the following one are transferred from L3 to L2.

6.2 DORY Hybrid Optimization

Here, we discuss the improvement of the new DORY kernel library over PULP-NN kernels [14] (HWC layout) and Greenwaves’ ones [23] (CHW layout). In Fig. 11, we show comparison on different layers, representative of the normal convolutions, and depth-wise ones. On classical convolutions, our approach is 2.5 × faster compared to the CHW layout. As discussed in Section 4.3, the DORY library includes an optimized depth-wise layer, reducing the penalty of using the HWC layout in its execution. Using an HWC layout on depth-wise layers can cause up to 3.7 × slow down if compared to the CHW one, strongly penalizing the performance for these layers. We reduce this loss by a factor of 2: our kernel is 1.5 × /2.0 × faster than the HWC one, reaching 0.54 × the performance of the Greenwaves’ one.

On the Mobilenet-v1-1.0 with resolution 128x128, updating the depth-wise and point-wise kernel from the HWC ones, we gain 1.79 MAC/cycles on the network’s overall execution. At a frequency of 100 MHz on both cluster and I/O domains, we improved the 3.0 FPS of HWC layout, reaching 4.3 FPS thanks to the optimized DORY kernel library.
6.3 DVFS optimization strategy

Since the I/O DMA and the cluster are in two different clock domains, the ratio of the two frequencies can significantly impact the bandwidth of both the L3-L2 and L2-L1 transfers and the performance and energy efficiency. In Fig. [12] we show the relationships between average power, execution time, and throughput in MAC/cycles, which are strictly related to the two frequencies. Energy efficiency is also shown in sub-plot A as a set of iso-energetic curves. A first significant effect that can be observed in these plots – particularly sub-plot B – is that increasing the fabric controller frequency strongly improves performance. In fact, increasing the fabric controller frequency directly causes the L3-L2 memory transfers to be faster, minimizing the fraction of time in which the system is memory bound. On the other hand, increasing frequencies also raises proportionally average dynamic power, as visible in sub-plot A. However, the memory-boundedness increase is more detrimental to the overall energy efficiency, as can be observed for the case of the fabric controller running at 50 MHz. It is also interesting to observe that, using voltage and frequency scaling, it is possible to scale the execution of MobileNet from a minimum latency of 93.9 ns at 24.6 mJ per frame to minimum energy of 12.5 mJ at 244 ms per frame.

6.4 Memory Hierarchy Sizing Exploration

We also investigate the impact of the memory dimensions on the network execution. To explore configurations with high dimensions of the memory, we used an FPGA-based emulator, realized with a Xilinx Zynq Ultrascale+ zcu102; the FPGA can host different instantiations of the PULP architecture template with different memories of various size.

Since DORY solves a series of tiling constrained problems, these constraints are relaxed/hardened if we increase/reduce, the internal MCU memories’ size. Fig. [13] depicts MAC/cycles and FPS while sweeping L1 between [22 KB, 400 KB] and L2 in [256 KB, 384 KB, 512 KB, 4 MB], highlighting different working corners in the tiling problem. L1 memory limits have been chosen since it 22 KB are needed to construct the smaller tile available and store the corresponding im2col buffer, and ii) over 400 KB no performance improvements are yet observed. L2 limits are related to chip design: while 256 KB is the lowest memory used as on-chip memory on a PULP platform [42], we foresee that 4 MB is the maximum memory that will be available in the near-future in low-cost IoT devices.

A first performance gap can be observed between the L2 = 256 KB and L2 = 512 KB configurations: with different L1 dimensions, using half of the memory causes up to 3.2 FPS loss @ 260 MHz. Using only half of the L2, 9 out of 29 layers demand the tiling of their activations from the external memory slowing down the execution of the first half of the network, since they can not fit the tightened constraint.

We can also observe a relatively constant decrease in performance when reducing L1 memory from 70 KB down to 22 KB with some abrupt performance loss. Two different phenomena can be observed: as the first point, the decrease of the L1 memory implies the creation of smaller tiles and hence more iterations, which cause more overhead from the DORY infrastructure. A second and more severe degradation can be observed when the tiling heuristics of the network layers could not be maximized anymore. As an example, from case A of Fig. [13] reducing L1 memory from 30 KB to 28 KB, we observe that the heuristics of 13 layers simultaneously get worse, with a corresponding degradation of 20% in the performance. Conversely, from 70 KB to 400 KB, there is a minimal gain given that all the tiling heuristics are already satisfied.

Fig. [13] clearly demonstrates that a well-designed tiling scheme is extremely effective at hiding the cost of both external and L2 memory access; the DORY layer solver can always create such a scheme for the L2-L1 communication while having sufficient L1 memory (i.e. > 30 KB). Further, activation L3-tiling tends to impair performance more heavily than weight L3-tiling. Nevertheless, the latter is much more necessary for practical-size networks. The red dashed lines in Fig. [13] depicts a MobileNet-v1 with input resolution 128x128 that only necessitates for L3 weight tiling in the last 8 point-wise convolutions, while eliminating external memory activations transfers.

Overall, thanks to DORY’s optimal exploitation of memory bandwidth and locality enhancements due to backend and tiling, we see that a 80KB L1 and 384 KB L2 memory configuration is sufficient to lead a MAC/cycle degradation of just 8% (from 10.57 to 9.74 MAC/cycles) compared to the largest memory configuration for the targeted network (4 MB L2 and 400 KB L1, which eliminates external memory transfer and L2-L1 tiling) – this results in a 91%/80% total L2/L1 memory size reduction in case this network is used to drive memory sizing. More in general, DORY can help sizing the memory hierarchy for a targeted family of networks, minimizing the performance loss using smaller memories.

7 Conclusion

It is now well understood that memory is a key limiting factor to the deployment of real-world DNNs on extreme edge computing platforms, constituting a “Deep Learning Memory Wall”; advanced microcontrollers dedicated to DNNs, which typically feature sophisticated on-chip memory hierarchies to enable high-throughput computation, suffer from this problem the most. In this work, we introduced a novel framework for DNN deployment, DORY (Deployment Oriented to memoRY), which unburdens the programmer from the manual optimizations of different neural networks on computing end-nodes. DORY obtains
near-optimal tiling of DNNs by combining constraint programming with a set of target-aware heuristics that enable the exploitation of the target architecture’s potential performance, even under very strict memory constraints. As a case study, we targeted an advanced DNN-oriented MCU, GWT GAP-8, showing that it can achieve 14.7× higher energy efficiency and 15.4× higher performance if compared to the industry-standard STMicroelectronics STM32F746—up to 26.6% end-to-end inference improvement compared to the proprietary tool from GWT. These results clearly show that part of the Deep Learning Memory Wall, i.e., the limited amount of on-chip memory, can be overcome using optimal multi-level tiling to drive a software-based caching scheme: we show that real-world networks designed for smartphone inference can be executed at real-time frame rates on a MCU with less than 1 MB of on-chip memory. All our developments are released as open-source at https://github.com/pulp-platform/dory. Our future work will focus on “flattening” the Deep Learning Memory Wall even further, adding support for stronger quantization techniques, hardware-accelerated primitives, and emerging memory technologies to support more high-accuracy networks directly on sub 10 mW extreme edge platforms.

ACKNOWLEDGEMENT

The authors thank Daniele and Margot Palossi for their help in setting up the RocketLogger to obtain GAP8 power traces.

REFERENCES

[1] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Baraghni, and A. P. Sheth, “Machine learning for internet of things data analysis: a survey,” Digital Communications and Networks, vol. 4, no. 3, pp. 161 – 175, 2018.

[2] N. H. Motlagh, M. Bagas, and T. Taleb, “UAV-based IoT platform: A crowd surveillance use case,” IEEE Communications Magazine, vol. 55, no. 2, pp. 126–134, 2017.

[3] M. Zanghi, S. Benatti, A. Burrello, V. Kartsch, F. Conti, and L. Benini, “Robust Real-Time Embedded EMG Recognition Framework Using Temporal Convolutional Networks on a Multicore IoT Processor,” IEEE Transactions on Biomedical Circuits and Systems, 2019.

[4] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow, and M. N. Hindia, “An overview of Internet of Things (IoT) and data analytics in agriculture: Benefits and challenges,” IEEE Internet of Things Journal, vol. 5, no. 5, pp. 3758–3773, 2018.

[5] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, “A 64mW DNN-based Visual Navigation Engine for Autonomous Nano-Drones,” IEEE Internet of Things Journal, 2019.

[6] F. Conti, R. Schilling, P. D. Schiavone, A. Pullini, D. Rossi, F. K. Gurkanm, M. Muehlerberghuber, M. Gautsch, I. Loi, G. Haugou et al., “An IoT Endpoint System-on-Chip for Secure and Energy Efficient Near-Sensor Analytics,” IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 64, no. 9, pp. 2481–2494, 2017.

[7] F. Conti, M. Rusci, and L. Benini, “The Memory Challenge in Ultra-Low Power Deep Learning,” in NANO-CHIPS 2030. Springer, 2020, pp. 323–349.

[8] H. Gao, W. Tao, D. Wen, T.-W. Chen, K. Osa, and M. Kato, “Ioqnet: Integrated fixed-point quantization networks for embedded vision,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.

[9] Y. Chen, J. Emer, and V. Sze, “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks,” in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 367–379.

[10] F. Conti and L. Benini, “A Ultra-Low-Energy Convolution Engine for Fast Brain-Inspired Vision in Multicore Clusters,” in 2015 Design, Automation Test in Europe Conference Exhibition (DATE), 2015, pp. 683–688.

[11] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, “Hardware for machine learning: Challenges and opportunities,” in 2017 IEEE Custom Integrated Circuits Conference (CICC), IEEE, 2017, pp. 1–8.

[12] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, “XpulpNN: Accelerating Quantized Neural Network on RISC-V Processors Through ISA Extensions,” in To appear at Design, Automation and Test in Europe Conference (DATE) 2020, IEEE, 2020.

[13] G. Danesi, N. Chawla, L. Benini, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyaawasi, H. Singh, and N. Aggarwal, “A 2.9TOPS/W deep convolutional neural network Soc in FD-SOI 28nm for intelligent embedded systems,” in 2017 IEEE International Solid-State Circuits Conference (ISSCC), 2017.

[14] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, “PULPSNN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors,” Philosophical Transactions of the Royal Society A, vol. 378, no. 2164, p. 20190155, 2020.

[15] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, “PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision,” Journal of Signal Processing Systems, vol. 84, no. 3, pp. 339–354, 2016.

[16] Cypress. (2019) Cypress DRAM. [Online]. Available: https://www.cypress.com/products/hyperram-memory.

[17] S. Saito, Y. Endo, T. Lepley, and O. Maler, “Optimizing two-dimensional DMA transfers for scratchpad based MBench platforms,” Microprocessors and Microsystems, vol. 37, no. 8, 2013.

[18] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, “Adrenaline: An openvnx environment to optimize embedded vision applications on many-core accelerators,” in 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip. IEEE, 2015, pp. 289–296.

[19] L. Cecconi, S. Smets, L. Benini, and M. Verhelst, “Optimal tiling strategy for memory bandwidth reduction for cnns,” in International Conference on Advanced Concepts for Intelligent Vision Systems. Springer, 2017, pp. 89–100.

[20] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, “Memory-centric accelerator design for convolutional neural networks,” in 2015 IEEE 31st International Conference on Computer Design (ICCD), IEEE, 2013, pp. 13–19.

[21] M. Abadi, A. Agarwal, P. B. et al., “Tensorflow lite for microcontrollers,” 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers.

[22] ST Microelectronics. (2017) X-CUBE-AI. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html.

[23] Greenwaves Technologies. (2018) X-CUBE-ROCK. [Online]. Available: https://www.greenwaves-technologies.com/manuels/BUILD/AUTOTILER/html/index.html.

[24] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, “GAP-8: A RISC-V SoC for AI at the Edge of the IoT,” in 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE, 2018, pp. 1–4.
Alessio Burrello received his B.Sc and M.Sc degree in Electronic Engineering and Embedded Systems from the Politecnico of Turin, Italy, in 2016 and 2018. He is currently working toward his Ph.D. degree at the Department of Electrical, Electronic and Information Technologies Engineering (DEI) of the University of Bologna, Bologna, Italy. His research interests include Quantized Neural Networks, hardware oriented deep learning, and code optimization for multicore systems.

Angelo Garofalo received the B.Sc and M.Sc. degree in electronic engineering from the University of Bologna, Bologna, Italy, in 2016 and 2018 respectively. He is currently working toward his Ph.D. degree at DEI, University of Bologna, Bologna, Italy. His main research topic is Hardware-Software design of ultra-low power multiprocessor systems on chip. His research interests include Quantized Neural Networks, Hardware efficient Machine Learning, precision computing, and energy-efficient fully-programmable embedded architectures.

Nazareno Bruschi received the M.Sc degree in Electronic Engineering at the University of Bologna, Bologna, Italy, in 2020. Since then, he is a Ph.D. student in the Department of Electrical, Electronic and Information Technologies Engineering (DEI) of the University of Bologna. His research interests cover hardware and software optimization for low power and high efficiency embedded systems, parallel programming for multicore architectures and virtual prototyping.

Giuseppe Tagliavini received the Ph.D. degree in electronic engineering from the University of Bologna, Bologna, Italy, in 2017. He is currently an Assistant Professor with the Department of Computer Science and Engineering (DISI) at the University of Bologna. He has co-authored over 30 papers in international conferences and journals. His research interests include parallel programming models for embedded systems, runtime optimization for multicore and many-core accelerators, and design of software stacks for emerging computing architectures.

Davide Rossi received the Ph.D. degree in electronic engineering from the University of Bologna, Bologna, Italy, in 2012 where he currently holds an assistant professor position. His research interests focus on energy efficient digital architectures in the domain of heterogeneous and reconfigurable multi and many-core systems on a chip. This includes architectures, design implementation strategies, and runtime support to address performance, energy efficiency, and reliability issues of both high end embedded platforms and ultra-low-power computing platforms targeting the IoT domain. In these fields he has published more than 100 papers in international peer-reviewed conferences and journals. He is recipient of Donald O. Pederson Best Paper Award 2018 - 2020 IEEE Transactions on Circuits and Systems Darlington Best Paper Award, 2020 IEEE Transactions on Very Large Scale Integration Systems Prize Paper Award.

Francesco Conti received the Ph.D. degree in electronic engineering from the University of Bologna, Bologna, Italy, in 2016. He is currently an Assistant Professor in the DEI Department of the University of Bologna. From 2016 to 2020, he held a research grant in the DEI department of University of Bologna and a position as post-doctoral researcher at the Integrated Systems Laboratory of ETH Zurich in the Digital Systems group. His research focuses on the development of advanced deep learning based intelligence on top of ultra-low power, ultra-energy efficient programmable Systems-on-Chip – from both the hardware and software perspective. His research work has resulted in more than 40 publications in international conferences and journals and has been awarded several times, including the 2020 IEEE TCAS-I Darlington Best Paper Award.