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

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-H743 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-H743. 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 1 MB 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 instruction set architecture (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-banked L1 memory [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 (< 128 kB) L1 with a bigger-but-slower (∼1 GB/s) on-chip L2 memory, and by an off-chip L3 low-power IoT DRAM [16] that provides high capacity, but at a slower speed (∼100 MB/s) and with

- *A. Burrello, A. Garofalo, N. Bruschi, D. Rossi and F. Conti are with the Department of Electrical, Electronic and Information Engineering, University of Bologna, 40136 Bologna, Italy.*
  G. Tagliavini is with the Department of Computer Science and Engineering, University of Bologna, 40136 Bologna, Italy.
  E-mail: alessio.burrello@unibo.it, angelo.garofalo@unibo.it, nazareno.bruschi@unibo.it, giuseppe.tagliavini@unibo.it, davide.rossi@unibo.it, f.conti@unibo.it
- *This work was supported in part by the EU Horizon 2020 Research and Innovation projects OPRECOMP (Open trans-PREcision COMPuting, g.a. no. 732631) and WiPLASH (Wireless Plasticity for Heterogeneous Massive Computer Architectures, g.a. no. 863337) and by the ECSEL Horizon 2020 project AI4DI (Artificial Intelligence for Digital Industry, g.a. no. 826060).*

Manuscript received July, 2020.

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 [17] and data tiling strategies [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 [5], [19], exploit data reuse, and optimize scheduling [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 [21]), or on proprietary code generation frameworks (e.g., ST XCUBE-AI [22], GWT[1] AutoTiler [2]).

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 our main case study, we target GWT GAP-8 [23] – 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 tool for multi-level memory tiling aiming at the deployment of realistically sized DNNs on memory-starved MCUs. Relying on Constraint Programming (CP) optimization, our tool matches on- and off-chip memory hierarchy constraints with DNN geometrical 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 backend library PULP-NN [14], to maximize throughput and energy efficiency in the RISC-V based GAP-8 target.
3. A code generator using tiling solutions to produce ANSI C code for the target platform, with all data L3-L2-L1 orchestration implemented as fully pipelined, triple-buffered DMA transfers and integrated calls to the computational backend (PULP-NN).
4. Tailored optimizations for common architectural features of modern DNNs: i) for residual connections, a bidirectional stack avoiding memory fragmentation; ii) for depthwise layers, a new optimized backend not present in PULP-NN.

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$\times$ better MAC/cycle than the state-of-the-art result on a conventional cache-based microcontroller, the STM32-H743 MCU, in single layer execution. Using DORY, end-to-end deployment of 8-bit quantized networks such as *0.5-MobileNet-v1-192*, achieve up to 8.00 MACs/cycle, with a 13.2$\times$ improvement compared to the same networks running on the STM32-H743 using the state-of-the-art ST X-CUBE-AI. Furthermore, on a layer by layer basis, DORY can achieve up to 2.5$\times$ 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 put our results into context, we compare the efficacy of DORY on GAP-8 with that obtainable in a state-of-the-art single-core ARM M7 core, the off-the-shelf ST32-H743 MCU with 16 kB of L1 data cache (D$) and 128 kB of L1 scratchpad memory. For a set of 44 DNN layers of different size, we compare i) single-core execution on GAP-8 with DORY-based memory management, ii) M7 execution with active D$, iii) M7 execution with DORY-managed DMA transfers on the scratchpad memory. Our results show that on the M7, DORY automatic memory management is up to 9% faster than the 16 kB hardware cache, and never slower. We also show that single-core execution on GAP-8 is, on average, 2.5$\times$ faster than on the M7 in cycles/cycles thanks to the full exploitation of the DSP-enhanced RISC-V cores.

To foster research on real-world deeply embedded DNN applications, we release the DORY framework, the optimized backend kernels, and the PULP heuristics discussed in this paper as open-source [3].

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 [30], [31] or using neural architecture search [32], [33]. Orthogonally, system designers can adopt techniques for post-training quantization [34] and quantization-aware fine-tuning [35] 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 particular layers and, in some cases, entire networks [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 [14], [34], [36], [37] to maximize the efficiency of DNN execution with DSP-oriented single-instruction multiple-data (SIMD) ISA capabilities [38]. 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 [36]. Further, there is an increasing trend towards more targeted ISA specialization (e.g., ARM Helium [4],

1. GreenWaves Technologies.
2. https://greenwaves-technologies.com/manuals/BUILD/AUTOTILER/html/index.html
3. https://github.com/pulp-platform/dory
4. https://www.arm.com/why-arm/technologies/helium

TABLE 1
Data flow scheduling and tiling in literature for different computing scales, super computing, ASIC accelerators, and tiny MCUs.

Work	Networks	Optimizations	Output	Open-Source	Precision
Supercomputers					
DMIAYN [24]	Transformers	1) Operator Fusing, 2) Data Layout Exploration	Transformer Primitives	Yes	fp32
DNN Accelerators					
dMazeRunner [25]	CNN, Nested Loops	1) Loop Ordering, 2) Loop Tiling, 3) Memory Movements	Temporal/Spatial Schedule, Loop Tiling	Yes	Flexible
MAESTRO [26]	CNN	1) Mapping & Data Reuse, 2) PEs Design	PEs array, Temporal/Spatial Schedule	Yes	Flexible
Interstellar [27]	CNN, LSTM, MLP	1) Loop Ordering, 2) Loop Tiling, 3) PEs+Mem. Design	PEs + Mem. Array, 7-Loops Ordering and Tiling	Yes	16 bits
Timeloop [28]	CNN	1) Loop Ordering 2) Loop Tiling	Model Scheduling, Latency/Energy Estimation	No	Flexible
Mobile & MCUs					
LCE [29]	BNN	1) Loop Tiling, 2) Vectorization, 3) Parallelization	C++ Runtime Interpreter, C++ Descriptor	Yes	1bit
TFLite Micro [21]	CNN, MLP	1) Hand-configurable Mem., 2) Optimized Backends	C++ Runtime Interpreter, C++ Descriptor	Yes	int8-fp32
Cube-AI [22]	CNN, MLP	1) Mem. Access Opt.	C Optimized Executable	No	int8-fp32
GWT AutoTiler	CNN, MLP	1) Loop Tiling, 2) Mem. Access Opt.	C Optimized Executable	Partially	int8-int16
DORY	CNN, MLP	1) Loop Tiling, 2) Mem. Access Opt. 3) Mem. Fragmentation	C Optimized Executable	Yes	int8

xPULPNN [12]) to support and accelerate the pervasive convolutional layers with low-bitwidth linear algebra instructions.

Memory hierarchy management

One of the most critical challenges in DNN deployment is memory hierarchy management: modern DNNs generate high amounts of weight and activation traffic between different levels of the memory hierarchy, which may constitute a significant bottleneck. In Table 1, we report different methods for data flow scheduling and generation that cover three broad classes of devices, namely high-performance computing systems [24], DNN accelerators [25], [26], [27], [28], and embedded systems [29], [39]. For what concerns high-performance computing systems, [24] propose new transformer primitives to exploit data reuse and limit data movement by fusing pointwise operators. On the other hand, [25], [26], [27], [28] discuss DNN optimization on AI-specialized accelerators based on systolic arrays of processing elements (PEs), with a focus on loop tiling and/or reordering to i) efficiently move the data to fastest memory regions and ii) correctly schedule layers in space and time to maximize PE utilization. The output of these tools can be either an accelerator model to run a given DNN [26], [27] or the spatial scheduling to maximize PE array utilization on a target accelerator [25], [28].

MCU data flow scheduling tools show similarities to frameworks such as DMazeRunner, as both target the optimization of a dataflow schedule given an externally known architecture. However, the MCU scenario also imposes some additional unique challenges, such as the fact that DNN execution has to be adapted to a general-purpose architecture and the small amount of memory that MCU platforms include. Further, the kernel instructions are heavily influenced by the limited size of the register file, which causes additional load-store operations and thus demand for an optimal loop sizing to avoid register spilling overhead. Academic researchers and industries have significantly investigated this aspect by including in their edge-node solutions either specialized caches (e.g., NXP [5]) or explicitly managed scratchpad memories (e.g., GWT [23]).

DNN-oriented microcontrollers and related tools

Recently, the first generation of low-power neural-network oriented MCUs has been introduced, coupling optimized software and ISA extensions for DNN computing with "traditional" control and I/O-bound activities. To enable optimal execution of both kinds of tasks, these MCUs exploit parallel and heterogeneous processing; for example, ST Microelectronics[6] and NXP 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 compared 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.

New platforms magnify these industry-wide architectural trends, introducing multi-core and AI-specific accelerators and removing data caches, replacing them with small on-chip scratchpad memories. For instance, the Kendrite K210 [7] is a RISC-V dual-core 64 bits system-on-chip with a neural network processor (KPU) on which the cores can offload the computation. It also includes dedicated memory banks for the NN accelerator and a DMA unit to explicitly manage the transfers. The SONY Spresense board [8] features

5. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc4300-cortex-m4-m0
6. https://www.st.com/en/microcontrollers-microprocessors/stm32h7-series.html
7. https://canaan.io/product/kendryteai
8. https://developer.sony.com/develop/spresense/

a 6-cores M4 accelerator with a maximum clock speed of 156 MHz, 1.5 MB of SRAM and 8 MB of Flash. The GreenWaves Technologies GAP-8 [23] system-on-chip, which we target as a case study in this work, was introduced in 2018 as a commercial embodiment of the *Parallel Ultra-Low-Power* paradigm [15]: it features one I/O core and an 8-core SIMD-optimized DSP cluster accelerator using an extension of the RISC-V ISA. Programming these DNN-oriented MCUs is typically more complicated with respect to conventional MCUs. Maximizing the exploitation of computational resources is challenging, and scratchpads require manually managed data orchestration and tiling.

New tools such as TFLite Micro [21] and the Larq Computing Engine (LCE) [29] offer a model-agnostic deployment framework and overcome these problems. Both are non-vendor-locked tools supporting ARM Cortex-M and RISC-V cores. Their library memory footprints require only 16 kB on a Cortex-M3; however, by default they rely on graph interpretation at runtime, limiting achievable performance. To offset this limitation, TFLite Micro allows plugging in optimized kernels and declaring vectors in different memory regions. However, it does not include any tiling mechanism to execute layers that do not fit on-chip memory.

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 DNN tools such as Tensorflow into a precompiled library for the ARM Cortex-M cores embedded in STM32 series MCUs. X-CUBE-AI relies on relatively large on-chip L1 caches (up to 16 kB) to deliver performance on STM32 MCUs, and it does not tackle software-based memory management. On the other hand, GWT designed a tool called AutoTiler, to target the GAP-8 RISC-V based multi-core ultra-low-power microcontroller. One of its primary functions 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 directly tackles the data-movement and tile sizing challenge to optimize memory access, reaching state-of-the-art performance on the execution of many networks. The tool is proprietary, but its backend basic kernels are available as open-source as part of the GAP-8 SDK[9].

DORY is the first open-source framework to directly tackle the MCU memory hierarchy management challenge, with a comprehensive exploration of data tiling, optimized loop ordering for different layers (i.e., pointwise and depth-wise), and a solution for the data fragmentation problem that is critical to deploy residual layers at the edge. In Section 5, we perform several quantitative comparisons with the best results obtained with STM X-CUBE-AI, GWT AutoTiler, and our own DORY. DORY consistently outperforms all the competitors on all the proposed benchmarks.

3 Background

3.1 Quantized Neural Networks

Post-training quantization [34] or quantization-aware training [35] produce as output a Quantized Neural Network (QNN). In the context of this work, we consider QNNs produced with *linear uniform per-layer quantization*, where all tensors $\mathbf{t}$ (e.g., weights $\mathbf{w}$, inputs $\mathbf{x}$, or outputs $\mathbf{y}$) defined in a range$[\alpha_{\mathbf{t}}, \beta_{\mathbf{t}})$ can be mapped to N-bit integer tensors $\widehat{\mathbf{t}}$ through a bijective mapping:

$$\mathbf{t} = \alpha_{\mathbf{t}} + \varepsilon_{\mathbf{t}} \cdot \widehat{\mathbf{t}}\ , \tag{1}$$

where $\varepsilon_{\mathbf{t}} = (\beta_{\mathbf{t}} - \alpha_{\mathbf{t}})/(2^N - 1)$. We call $\varepsilon_{\mathbf{t}}$ 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 $\alpha_{\mathbf{x}} = \alpha_{\mathbf{y}} = 0$ for all the inputs of Linear and the outputs of Quantization/Activation operators[10], but not for weights. Using Eq. 1, all operators are mapped in the integer domain:

$$\text{LIN}: \quad \varphi = \sum_n \mathbf{w}_{m,n}\mathbf{x}_n \iff \widehat{\varphi} = \sum_n \widehat{\mathbf{w}_{m,n}} \cdot \widehat{\mathbf{x}_n} \tag{2}$$

$$\text{BN}[11]: \quad \varphi' = \kappa \cdot \varphi + \lambda \iff \widehat{\varphi}' = \widehat{\kappa} \cdot \widehat{\varphi} + \widehat{\lambda}\ . \tag{3}$$

The dot product operation in Eq. 2 results in a shrinking of the quantum used to represent $\widehat{\varphi}$, which will be $\varepsilon_{\varphi} = \varepsilon_{\mathbf{w}}\varepsilon_{\mathbf{x}}$. Hence, we need to represent the integer output of the Linear operator ($\widehat{\varphi}$) with higher precision (e.g., 32 bits) with respect to its inputs, before re-quantizing it at the end of the accumulation. A similar consideration applies to Batch-Normalization and its output $\widehat{\varphi}'$.

The final Quantization/Activation operator *i)* provides a non-linear activation essential for the QNN to work at all, and *ii)* collapses the accumulator into a smaller bitwidth:

$$\text{QNT/ACT}: \quad \widehat{\mathbf{y}} = m \cdot \widehat{\varphi}' \gg d\ ;\ m = \left\lfloor \frac{\varepsilon_{\varphi'} \cdot 2^d}{\varepsilon_{\mathbf{y}}} \right\rfloor\ . \tag{4}$$

d is an integer chosen during the quantization process in such a way that $\varepsilon_{\varphi}/\varepsilon_{\mathbf{y}}$ can be represented with sufficient accuracy inside m. A method similar to Eq. 4 is also used when multiple branches of the network, each with its own ε, reconverge in a single tensor (typically using summation). In that case, the branches are "matched" to the same quantum using a variant of Eq. 4.

Thanks to the mapping of Eq. 1, it is possible to execute the entire network using only integer data. In this work, we target networks using 8-bit quantization for both $\widehat{\mathbf{w}}$ (signed), and $\widehat{\mathbf{x}}$ / $\widehat{\mathbf{y}}$ (unsigned); $\widehat{\varphi}$, $\widehat{\varphi}'$, and the $\widehat{\kappa}$, $\widehat{\lambda}$, m, d parameters use 32-bit integers (signed). We relied on the open-source NEMO library [40] to generate QNN topologies in the format described in this Section. Note that using different quantization techniques such as non-linear 8 bits quantization or clustering [41] for network compression and execution would be possible with DORY replacing the software backend employed.

3.2 Parallel Ultra-Low-Power computing paradigm

Research and industry are dedicating increasing attention to edge-nodes with specialized co-processors (accelerators) and hierarchical memories, designed to exploit the pervasive data-regularity in emerging data-analytics tasks (e.g., deep-learning). Parallel Ultra-Low Power computing is an architectural paradigm leveraging near-threshold computing to achieve high energy efficiency, coupled with parallelism to improve the performance degradation at low-voltage [42]. The PULP paradigm builds upon the trends explained in Section 2: ISA optimizations for DSP and DNN

9. https://github.com/GreenWaves-Technologies/gap_sdk

10. If the original activation is a ReLU, then the QNN automatically satisfies this condition; otherwise, it can be transformed to satisfy it.

11. In inference, the statistical and learned parameters of BN can be combined: $\kappa = \gamma/\sigma$ and $\lambda = \beta - \mu\gamma/\sigma$.

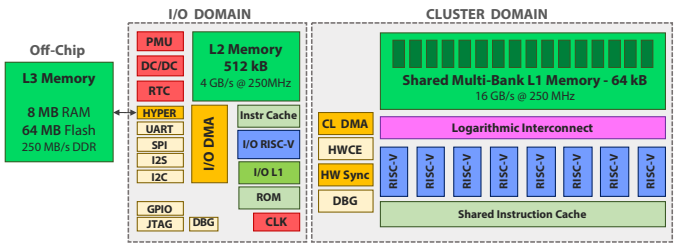


Fig. 1. GWT GAP-8 MCU block diagram.

TABLE 2
Symbols used throughout this work.

Input $\mathbf{x}$ dims (height/width/chan)	h_x / w_x / C_x
Output $\mathbf{y}$ dims (height/width/chan)	h_y / w_y / C_y
Weight $\mathbf{w}$ dims (out c/height/width/in c)	C_y / K_h / K_w / C_x
Buffer for tensor q at i-th level of mem. hier.	Li_q
Tiled dimension d_q of a tensor $\mathbf{q}$	d_q^t

computing; heterogeneous parallel acceleration, with architecturally different compute units dedicated to unrelated tasks; and explicitly managed memory management. PULP systems are centered around a state-of-the-art single-core microcontroller (*I/O domain*) with a standard set of peripherals. The I/O core offloads parallel tasks to a software-programmable parallel accelerator composed of N additional cores, standing in its own voltage and frequency domain (*cluster domain*).

GWT GAP-8 [23] (depicted in Figure 1) is a commercial PULP system with 9 extended RISC-V cores (one I/O + an eight-core cluster), which we chose as the reference platform in this work since it represents one of the most advanced embodiments of the DNN-dedicated MCU trends.

The GAP-8 'cluster' is composed by eight 4-stage in-order single-issue pipeline RI5CY [38] cores, implementing the RISC-V RV32IMCXpulpV2 Instruction Set Architecture (ISA). XpulpV2 is a domain-specific extension meant for 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 2× banking factor and a word-level interleaving scheme to reduce the probability of contention [43]. 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 GB/s and a latency of 80 ns 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/HyperFash 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 MB/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 Im2Col 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 ($1 \times 1 \times C_x$) 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. 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 2 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 [30], [31]. 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).

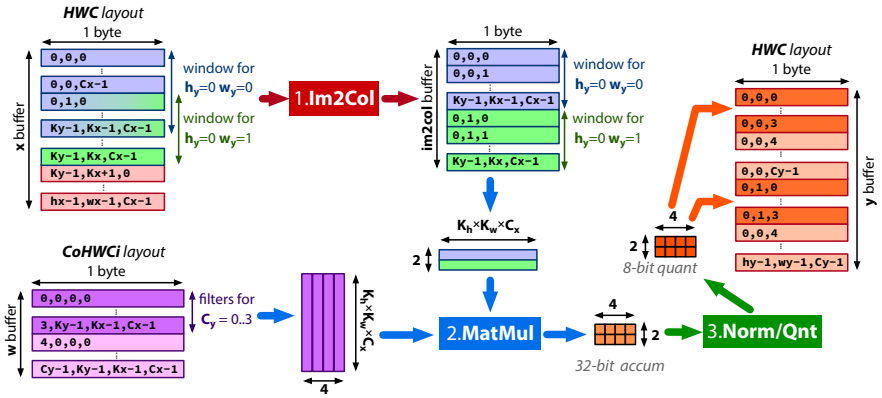


Fig. 2. PULP-NN [14] execution model divided in Im2Col, MatMul, Norm/Qnt phases (see Table 2 for the buffer naming scheme).

```
LTO: for (o = 0; o < C_y^t; o++)
  LTH: for (h = 0; h < h_y^t; h  ++)
    LTW: for (w  = 0; w < w_y^t; w ++)
      LTI: for (i = 0; i < C_x^t; i ++)
        dma_wait(L1_{x,load}); swap(L1_{x,load}, L1_{x,exec})
        dma_async(L1_{x,load} <- L2_x[i, w, h])
        dma_wait(L1_{w,load}); swap(L1_{w,load}, L1_{w,exec})
        dma_async(L1_{w,load} <- L2_w[i, o])
        if (o  + h + w  + i > 0)
            DNN_kernel (L1_{x,exec}, L1_{w,exec}, L1_{y,exec})
# from 3° iteration: fully operating pipeline
      if (o  + h  + w  + i > 1)
        dma_wait(L1_{y,load})
        dma_async(L1_{y,load}  -> L2_y[ o, w, h])
        swap(L1_{y,load} , L1_{y,exec})
```

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.

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.

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. Each DORY layer uses quantized inputs, outputs, and weight, while the representation of any temporary data is 32-bit signed integer.

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 & constraints* optimizing execution over a target-specific backend and limiting the tiling search space; 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 $Li_{\mathbf{t}}$, where $\mathbf{t}$ is the name of the tensor. The Solver relies on a 2-step engine, which solves the L3-L2 tiling constrained problem first, and the L2-L1 one afterwards. With L3-L2 tiling, we enable storing activations and weights in the L3 off-chip memory instead of the on-chip L2. With respect to tools that do not support L3 tiling for activations, such as Tensorflow Lite Micro, this feature enables to support significantly larger layers. The Solver verifies whether the layer memory occupation fits the L2 memory input constraint or needs to be stored in L3:

$$L2_{w,next} + L2_{w,curr} + L2_x + L2_y \overset{?}{<} L2\ . \qquad (5)$$

We search an L3 tiling solution using a five-stage cascaded procedure. At each stage, we try to tile a different selection of buffers to fit the constraint of Eq. 5. Whenever possible, the tiler tries to avoid L3-L2 tiling of output activations, which always requires a double number of transfers (from L2 to L3 when produced, and from L3 to L2 when consumed by another layer). Instead, the tiler tries to keep output activations in L2 as much as possible. If a stage satisfies Eq. 5, the L3-L2 Tiling Solver is stopped and the dimensions of tiles are saved. Otherwise, the next stage is tried.

stage 0. `L3-tile x, w, y = OFF, OFF, OFF`. If Eq. 5 is directly satisfied, we proceed without L3-L2 tiling.

stage 1. `L3-tile x = ON`. This solution is selected when the output of the previous layer was tiled in L3, and therefore input tiling cannot be avoided. Tiling is performed along the h_x dimension of the input, to avoid 2D transfers at the L3-L2 interface. The tiler 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 $C'_y < C_y$. This solution can only be selected when the output of the previous layer is already in L2.

stage 3. `L3-tile w , y = OFF, ON`. Weight tiling is disabled while output tiling is enabled: the approach is similar to input tiling, but requires doubling the DMA transfers for the tiled tensor across the full network execution.

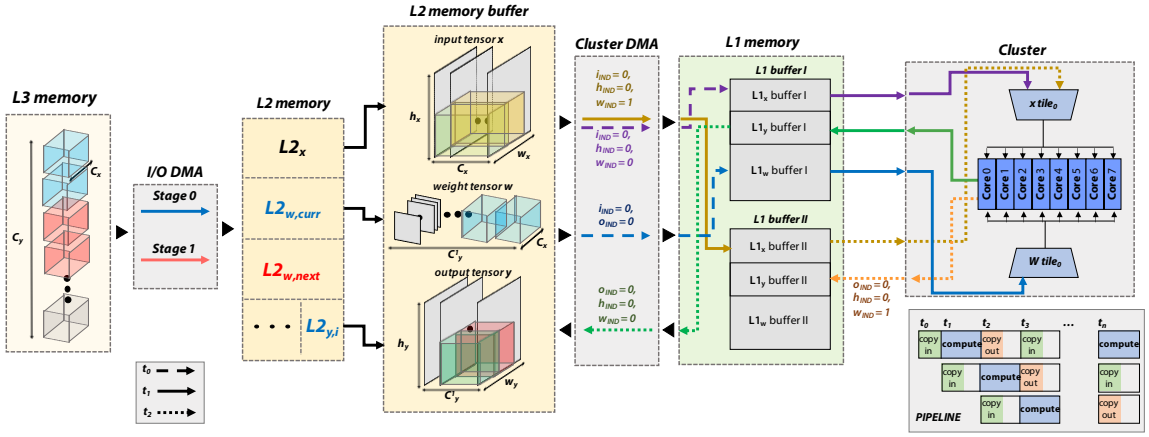


Fig. 3. DORY L3-L2-L1 layer routine example. On the left, the I/O DMA copies weights tile in case only C_y is L3-tiled. Two different buffers are used for $L2_w$. Then, the Cluster DMA manages L2-L1 communication using double-buffering, while the cores compute a kernel on the current tile stored in one of the L1 buffers.

stage 4. `L3-tile w, y = ON, ON`. The L3 tiling is enabled on both buffers, y, $weights$. This solution is selected when no other solution can fit L2.

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. Compared to high-end computation engines, with much larger memories, a suboptimal sizing of the tensors for the L1 small MCUs memory can be even more detrimental in terms of performance, as exposed in Section 6.1. DORY abstracts this as a Constraint Programming (CP) problem, and exploits the CP solver from the open-source OR-Tools developed by Google AI [12] to meet hardware and geometrical constraint (e.g., C_y^t 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:

$$\max(L1_x + L1_y + L1_w) \ , \tag{6}$$

manipulating the tile dimensions (e.g., C_x^t and C_y^t). The hardware constraint is related to the max L1 buffer dimensions:

$$L1_x + L1_y + L1_w + L1_{backend} < \frac{L1}{2} \ .$$

with $L1_{backend}$, the overhead of the backend kernel, such as the im2col memory occupation of PULP-NN backend [14] or any other support buffer (e.g., the intermediate full-precision accumulators for CHW based convolutions). Topological and geometrical constraints are due to the relationships between each tensor's characteristic dimensions and other parameters of a layer; for example,

$$h_y^t = \left(h_x^t - (K_h - 1) + 2 \cdot p\right)$$

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:

$$\max\left(\alpha(L1_x + L1_y + L1_w) + \sum_i \beta_i \mathcal{H}_i\right) \ . \tag{7}$$

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

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:

$$\mathcal{H}_{i2c} = C_y^t$$

- `PAR_BALANCE`[13]: 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:

$$\mathcal{H}_{par} = (h_y^t - 1) \bmod 8$$

- `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:

$$\mathcal{H}_{mm\_w} = (w_y^t - 1) \bmod 2 \ ; \ \mathcal{H}_{mm\_ch} = (C_y^t - 1) \bmod 4$$

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.

We impose an additional constraint to always perform a full computation along the channel direction:

$$C_x^t = C_x$$

We choose not to tile the C_x dimension to avoid the memory overhead of long-term storage (and therefore, transfer to $L2$ and $L3$) of 32-bit partially accumulated values produced by the backend. For the same reason, we do not tile the spatial dimension of filters, i.e., K_h and K_w. While these constraints restrict the solution space, we observe that the purged solutions are sub-optimal.

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

13. The `PAR_BALANCE` constraint is changed to $\mathcal{H}_{par} = (h_y^t \times w_y^t - 1) \bmod 16$ for "patological" output activations with $h_y < 8$.

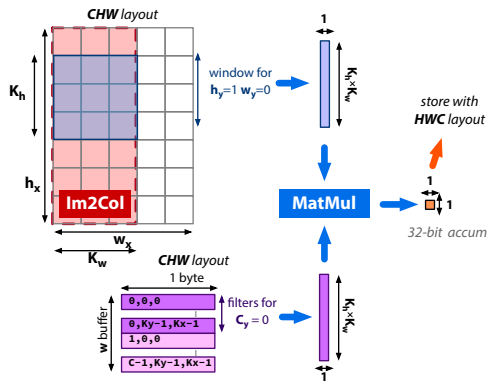


Fig. 4. Modified execution model for depthwise convolutions: the Im2Col buffer is built using a single channel out of CHW-layout activations; outputs are quantized and stored back using the PULP-NN model shown in Figure 2 (see Table 2 for the buffer naming scheme).

```
udma_async(L2_{w,load} <- L3_w[I_0])
udma_wait(L2_{w,load});
LTL: for (i = 0; i < nlayers; i ++)
# number of CNN layers
  udma_wait(L2_{w,load}); swap(L2_{w,load}, L2_{w,exec})
  if  (layer{i+1} fit L2 && is Conv)
      udma_async(L2_{w,load} <- L3_w[I_i])
  Layer{i} (L2_x, [L2_{x2}], [L3_w[I_i]], [L2_{w,exec}], L2_y)
  # [] optional arguments
      swap(L2_y, L2_x)
  if  (layer{i} has residual) # bypass management
      store  (L2_y->L2_{x2})
  if  (layer{i} is Sum)
      delete  (L2_x2)
      Stack_dealloc(L2_y) # stack control
      Stack_alloc(L2_x[I_{i+1}])
```

Listing 2: DORY network execution loop.

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 5. 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 [36] 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], [36] – 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 [31]; on the other hand, they are typically only responsible for 10% or less of the overall operations [30], [31], meaning that directly optimizing for them may be suboptimal. This scenario suggests a hybrid approach: using the HWC layout for general convolutional layers (and point-wise 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 4 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 callable function. Listing 2 showcases the execution loop of the DNN execution as created by our framework. At each step, three main tasks are concatenated : *i*) we transfer from L3 the weights of the following layer. [14] *ii*) a new layer is executed pointing to the correct buffers inside the memory stack; *iii*) input and output buffer offsets are updated.

Similarly to single layers, the network-wise code is generated automatically without programmer intervention. DORY produces a single function that can be called inside

14. 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 $L2_w$ 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.

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

$$D_{stack} = \max_i(L2_{x,i} + L2_{w,i} + L2_{w,i+1} + L2_{x,i+1}) \ ,$$

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 $L2_{w,i}$ and output buffer $L2_{y,i}$; notice that $L2_{x,i}$ is already allocated as the $L2_{y,j}$ of a previously executed j-th layer (or the input of the network). To manage residual connections, each $L2_{y,i}$ 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 $L2_{w,i-2}$ buffer on the corner;
3. the allocator checks if $L2_{y,i-2}$ has its `lifetime` counter set to 0; if so, it is deallocated;
4. $L2_{y,i}$, $L2_{w,i}$ are allocated in order in the selected corner (with $L2_{w,i}$ nearest to the pointer);
5. the `lifetime` counter of $L2_{y,i}$ 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-H743 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 α to 0.5, $\beta_{\texttt{HIDE\_IM2COL}}$ to 10^2, and other β_i to 10^6 in the objective function.

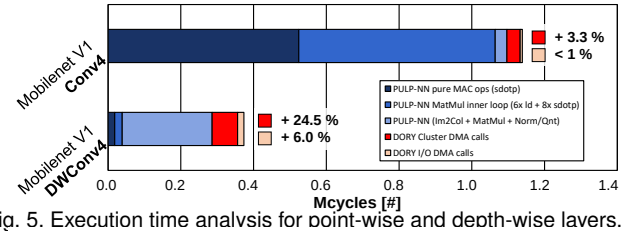


Fig. 5. Execution time analysis for point-wise and depth-wise layers.

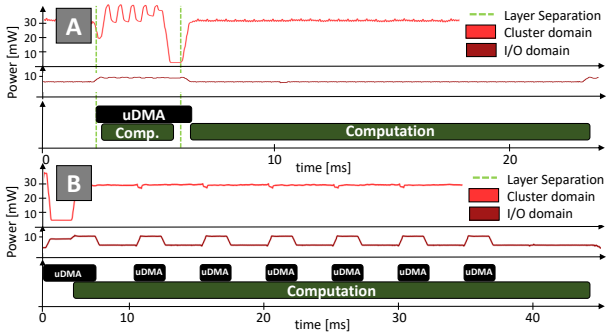


Fig. 6. In Part.A, the power traces of a point-wise Convolution following a depth-wise one. The I/O DMA causes the COREs to go in IDLE, waiting for the memory transfer end. In Part.B, an L3-tiled layer is executed and perfectly buffered to hide the memory hierarchy to the computing engine. fr = 100 MHz and V_{DD} = 1V have been used on the GAP8 MCU.

5.1 Single layer performance & SoA comparison

Fig. 5 analyzes all the execution time for two layers of MobileNet-v1 [30], the first representative of point-wise convolutional layers, the second of depth-wise ones. We observe several effects. 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., C_y^t not multiple of vector size 4). In the case of depth-wise layers, this latter class of loops dominates the backend execution time.

For what concerns the overhead introduced by DORY-generated tiling, we observe that the Cluster DMA does not impair the point-wise convolutional layers, since they are compute-bound and efficiently pipelined: further, the processing overhead of calling the Cluster DMA many times is parallelized over the 8 cores in the cluster, reducing the cost of realizing complicated tiling schemes. On the other hand, depth-wise layers are small, and both the Cluster DMA and I/O DMA overheads are exacerbated. Therefore, the load of the internal tiles and the asynchronous I/O DMA load of the following layer's weights are often impacting performance. Fig. 6 corroborates this conclusion, showing power valleys in the cluster computation while waiting for new tile transfers (smaller ones) and for the weights of next layers to be transferred from the external memory. Instead, Part.B of Fig. 6 shows the execution of a point-wise L3-tiled layer: in this case, the computation perfectly overlaps with the memory transfers, completely hiding the memory transfer overhead.

In Table 3, we compare our approach with three state-of-the-art frameworks for DNN deployment on MCU: TFLite Micro, STM X-CUBE-AI, and GWT AutoTiler. We focus on convolutional and depth-wise convolutional layers, which constitute the vast majority of computation in modern DNN models. Metrics are computed as the average between the layers in the MobileNet-v1 network. Results obtained on TFLite Micro and STM X-CUBE-AI refer to an STM32H743 microcontroller, based on an ARM Cortex-M7; those for GWT AutoTiler and DORY to a GWT GAP-8, described in Section 3.2. All results refer to 8-bit quantized networks, even if STM32 also supports 32-bit floating point; accuracy

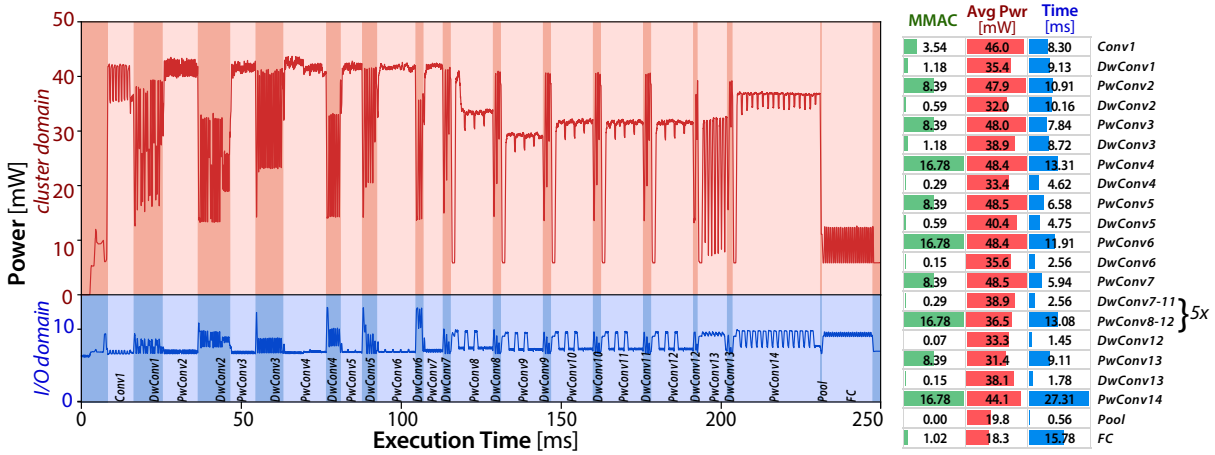


Fig. 7. In the left part, the 1.0-MobileNet-128 power profile when running on GAP-8 @ $f_{\text{cluster}} = f_{\text{io}} = 100\,\text{MHz}$ and $V_{DD} = 1\,\text{V}$. On the right, number of MAC operations, average power, and time for each layer of the network. Power was sampled at 64 KHz and then filtered with a moving average of $300\,\mu\text{s}$.

TABLE 3
Average performance and efficiency on 8-bits MobileNet-V1 layers obtained with DORY and other SoA MCU-deployment frameworks.

		Performance (speed-up) MAC/cycle	GMAC/s	Efficiency GMAC/s/W
TFLite$_a$ Micro	DwConv	0.064 (0.2×)	0.03 (0.2×)	0.13 (0.2×)
	PwConv	0.056 (0.1×)	0.027 (0.1×)	0.11 (0.1×)
STM^a CUBE-AI	DwConv	0.39 (1×)	0.19 (1×)	0.8 (1×)
	PwConv	0.71 (1×)	0.34 (1×)	1.46 (1×)
GWT^b AutoTiler	DwConv	2.16 (5.5×)	0.22 (1.2×)	4.24 (5.3×)
	PwConv	7.87 (11.1×)	0.79 (2.3×)	15.4 (10.6×)
GWT^c AutoTiler	DwConv	2.16 (5.5×)	0.56 (3.0×)	2.16 (2.7×)
	PwConv	7.87 (11.1×)	2.05 (6.0×)	7.87 (5.4×)
DORY^b	DwConv	1.14 (2.9×)	0.11 (0.6×)	2.24 (2.8×)
	PwConv	12.86 (18.1×)	1.29 (3.8×)	25.2 (17.3×)
DORY^c	DwConv	1.14 (2.9×)	0.30 (1.6×)	1.14 (1.4×)
	PwConv	12.86 (18.1×)	3.34 (9.8×)	12.86 (8.8×)

[a] Collected on the STM32H743 @ 480MHz.
[b] Collected on the GWT GAP8 @ (100MHz, 1V).
[c] Collected on the GWT GAP8 @ (260MHz, 1.15V).

is equivalent to that of a non-quantized network.

TFLite Micro has the main advantage of being available on many different ARM and RISC-V MCUs; on the other hand, its performance is severely limited by the fact that it uses very general APIs without deep optimizations. X-CUBE-AI outperforms it by 6.1× to 12.7× on the same platform, thanks to its much more efficient backend. Nonetheless, layers generated by DORY for the GAP-8 platform outperform both TFLite Micro and X-CUBE-AI by a margin of 2.9× to 229.6× in terms of MAC/cycle. This significant advantage is due to the architectural benefits of GAP-8 (multi-core acceleration, DSP-enhanced instructions) that DORY can exploit fully through PULP-NN, as showcased in the previous section. In Section 6.1, we decouple DORY performance enhancement and architectural benefits to underline the benefits of our framework, deploying layers with DORY both on the STM32H7 and on GAP8 forced to run with a single-core.

When compared to GWT AutoTiler, which targets the same platform, DORY is 1.6× faster in point-wise convolutions, while it pays a performance toll in depth-wise convolutions, where it is 1.9× slower. These differences amount mainly to the different strategies followed by the tools in their respective backends and will be deeply discussed in Section 6.1. As explained in Section 3.2, the number of output channels strongly influences performance because re-using input data for more output channels offsets the cost of the Im2Col operation. For depth-wise convolutions, each input channel is linked to a single output channels: as a consequence, this source of data re-use is not available.

5.2 End-to-end network performance

In this Section, we focus on the performance of DORY in deployment full-networks that are already used as benchmarks for many edge-oriented works [34]. All the 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 [40], 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 showcases a full comparison in terms of energy efficiency (GMAC/s/W), throughput (GMAC/s), latency, and energy per frame. Different variations of the MobileNet-v1 have been compared, with the same topology but a different number of channels or input dimensions. For state-of-the-art, we show the biggest networks that fit the on-chip/off-chip memory of the STM32H7 and GAP8, respectively (compatible with the ones deployed with DORY). We will release similar benchmarks on all the Mobilenets on our public repository. As can be noticed from the Table, DORY on MobileNet-v1 achieves up to 13.19× higher throughput in MAC/cycles than the execution on an STM32H7 (on 0.5-M.V1-192), using the best framework (X-CUBE-AI) currently available. On different operating points, we have up to 7.1× throughput (1.78 vs. 0.25 GMAC/s) and 12.6× better energy efficiency, given the different frequencies and power consumption of the two platforms.

Compared with GWT-proprietary and partially closed-source AutoTiler run on the same GAP-8 platform, our results show that DORY performs on average 20.5% better. Moreover, we note that as a default execution model, GWT AutoTiler folds the BatchNormalizations inside the convolution transformation, by saving operations, but potentially leading to more severe accuracy loss. Contrarily to the Autotiler, DORY by default keeps the BN explicit, causing 3 extra LOADS and 1 additional MAC for each output pixel. When using a 1:1 identical network to the one used by GWT AutoTiler (including folding), the performance gain is further increased to 26.6%. As previously discussed, the advantage lies in *1)* the more efficient backend (PULP-

TABLE 4
End-to-end execution of image recognition MobileNet-v1 and MobileNet-v2 on GAP8 and STM32H7 MCUs.

Configuration	Params	Work MAC	Cycles	Perf MAC/cyc	Eff GMAC/s/W	Perf GMAC/s	Lat lat. [ms]	Energy E [mJ]	Eff GMAC/s/W	Perf GMAC/s	Lat lat. [ms]	Energy E [mJ]
DORY @ GAP8												
					Low energy 1V @ 100 MHz				Low latency 1.15V @ 260 MHz			
1.0-M.V1-128	4.2 M	186.4 M	23.3 M	8.00	15.68	0.80	233.11	11.89	7.93	2.08	89.66	23.51
0.5-M.V1-192	1.3 M	110.0 M	16.0 M	6.86	13.46	0.69	160.2	8.17	6.82	1.78	61.62	16.16
0.25-M.V1-128	0.5 M	13.5 M	2.8 M	4.74	9.30	0.47	28.50	1.45	4.69	1.23	10.95	2.87
1.0-M.V2-128	3.47 M	100.1 M	19.0 M	5.27	10.33	0.53	190.03	9.69	5.22	1.37	73.09	19.16
GWT AutoTiler @ GAP8												
					Low energy 1V @ 100 MHz				Low latency 1.15V @ 260 MHz			
1.0-M.V1-128	4.2 M	186.4 M	28.1 M	6.64	13.02	0.66	280.80	14.32	6.58	1.73	108.00	28.32
1.0-M.V2-128	3.47 M	100.1 M	19.7 M	5.07	9.95	0.51	197.38	10.07	5.03	1.32	75.92	19.91
X-CUBE-AI @ STM32H7, solutions fitting 2MB ROM + 512 kB R/W RAM @ 480 MHz [34]												
0.25-M.V1-128	0.5 M	13.5 M	26.0 M	0.52	1.07	0.25	51.14	12.67	n.a.	n.a.	n.a.	n.a.
0.5-M.V1-192	1.37 M	109.5 M	212.3 M	0.52	1.06	0.25	442.27	103.49	n.a.	n.a.	n.a.	n.a.

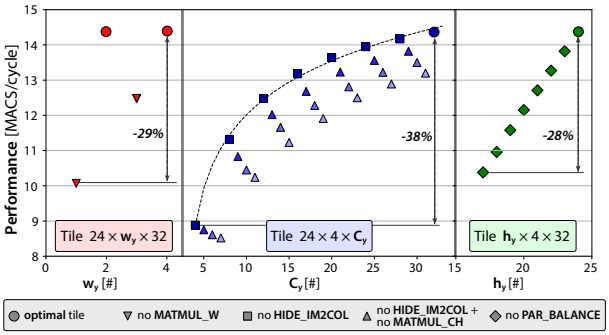


Fig. 8. Example of the effect of heuristic optimizations on convolutional layer performance. In this case, the "optimal" tile has output tensor 24×4×32 (HWC) and weight tensor 32×3×3×32 (CoHWCi). Different optimizations are showed by varying w_y, h_y, and C_y and violating the heuristics of Section 4.2.2.

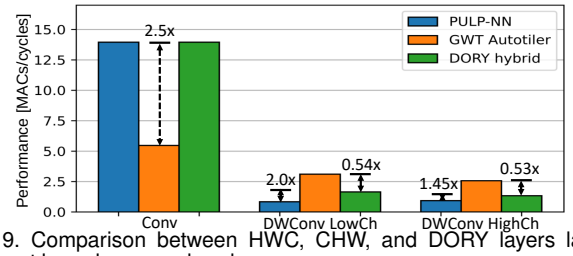


Fig. 9. Comparison between HWC, CHW, and DORY layers layout. Different kernels are explored.

NN) and *2)* the heuristics, which guarantee that the tiling solution is optimized for the PULP-NN execution model.

5.2.2 In-depth analysis of MobileNet-v1 execution

Fig. 7 depicts the power profile of the end-to-end execution of a MobileNet-v1 (1.0 width multiplier, 128 × 128 resolution) on GAP-8, with both the cluster and the fabric controller running at 100 MHz. The power consumption of the cluster domain (including 8 RI5CY cores, the L1 and the Cluster DMA) and of the I/O domain (including 1 RI5CY 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 Ablation Study

This section presents a detailed ablation study of each of our contributions against state-of-the-art baselines. We separately analyze the impact of: *i)* the proposed heuristics; *ii)* the hybrid optimization for depthwise layers; *iii)* voltage and frequency scaling on GAP-8; *iv)* the size of L1 and L2 memories; *v)* the specific GAP-8 architecture compared to standard MCUs.

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 in the execution of a single inner tile. Figure 8 exemplifies this phenomenon starting from an "optimal" tile of output tensor 24 × 4 × 32 (HWC) with a 32 × 3 × 3 × 32 filter (channel out - height - width - channel in, or CoHWCi). Violating `MATMUL_W/CH` leads to a maximum performance loss of 29%, violation of `HIDE_IM2COL` to a 38% loss, and violation of `PAR_BALANCE` to a 28% loss in this example layer. Note that the performance loss is cumulative since each heuristic is written to improve the performance of a different section of the PULP-NN kernel.

To further underline this effect, if we set all the β_i coefficients into the objective function of Eq. 7 to 0 and only focus on the maximization of the tile sizes, DORY chooses a tiling scheme that achieves only 2.78 MAC/cycles, 80.6% lower than the 14.37 MAC/cycles achieved with the β_i values previously reported. In fact, in contrast with a superficial intuition, border tiles can constitute up to 50% of the workload: for a layer of dimension 32×64×64×32 (CoHWCi), the DORY tiler generates a main 32×56×2×32 tile and a border 32×8×2×32 tile with a 28 kB L1 memory constraint; both tiles are executed 32 times.

6.2 Hybrid optimization for Depthwise layers

Here, we discuss the improvement of the new DORY kernel library over PULP-NN kernels [14] (HWC layout) and Greenwaves' ones (CHW layout). In Fig. 9, 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

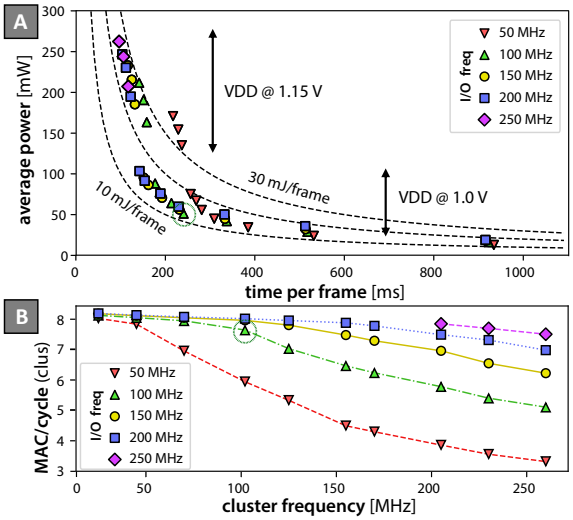


Fig. 10. Power, latency and MAC/cycles performance exploration with swiping frequencies. The 1.0-MobileNet-128 is used as a benchmark. CL frequency varies in [25 MHz, 260 MHz], I/O one in [50 MHz ,250 MHz]. A green dashed circle highlights the (100 MHz, 100 MHz) configuration that has been used throughout the paper.

of 2: our kernel is $1.5\times/2.0\times$ faster than the HWC one, reaching $0.54\times$ 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 Voltage and frequency scaling

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. 10, 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 $50\,\mathrm{MHz}$. 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 $93.9\,\mathrm{ms}$ at $24.6\,\mathrm{mJ}$ per frame to minimum energy of 12.5 mJ at 244 ms per frame.

6.4 Memory hierarchy sizing

We also investigate the impact of memory dimensions on the network execution time. 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.

Fig. 11 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 *i)* 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. The tool statically computes the minimum dimension of each memory level for the target DNN, raising an error if not compliant with input limits. Then, it maximizes the occupation of the Li buffers given as input constraints.

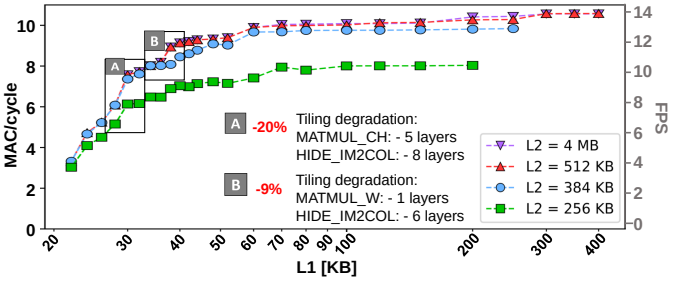


Fig. 11. MAC/cycles and FPS are explored with different configuration of L1-L2 memories using a 1.0-MobileNet-v1 with resolution 128x128. L2 varies from 256 kB (19/29 layers tiled from L3) to 4 MB (No L3 tiling), whereas L1 varies from 22 kB to 400 kB.

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 @ 260MHz. 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: *i)* reducing L1 memory requires smaller tiles and hence more iterations, increasing overhead; *ii)* reducing L1 memory too much can make the heuristics impossible to meet; for example, in case **A** of Fig. 11, a reduction 30 kB to 28 kB causes this effect on 13 layers simultaneously, dropping performance by 20%. Conversely, from 70 kB to 400 kB of L1 the gain is minimal, because all the tiling heuristics are already satisfied.

Overall, thanks to DORY's optimized exploitation of memory bandwidth and locality enhancements due to backend and tiling, we see that a 80 kB 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.

6.5 Single core performance on different architectures

In this section, we explore the impact of architectural and microarchitectural choices on DNN deployment using DORY. We do so by directly comparing the single-core performance obtained on GAP-8 with that achievable on a commercial STM32H743ZI2 MCU in several configurations. This MCU features an ARM M7 core with 16 kB of D-Cache and a large two-banked 0-wait-states scratchpad of 128 kB called DTCM, allowing us to separately investigate the impact of software vs. hardware caching and that of the

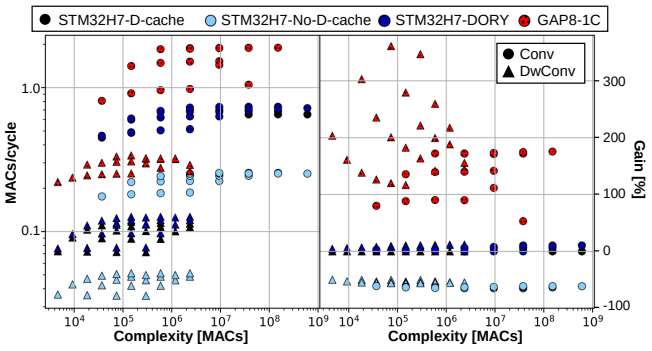


Fig. 12. On the left, absolute MAC/cycle of DORY framework on both STM32H7 and single-core GAP8, compared with default CUBE-AI/TensorFlow Lite for Micro layer backend, CMSIS-NN. On the right, relative gains compared to the fastest CMSIS-NN implementation.

different microarchitectures.

In our experiment, we tested 44 different configurations of layers (both depthwise and convolutional) spanning six orders of magnitudes of complexity. We explored four sets of solutions: for GAP-8, we used DORY and run on a single core in the cluster; for the STM32H7, we used CMSIS-NN[15] with and without D-Cache enabled. Finally, in the third STM32H7 configuration we ran using the DTCM scratchpad by combining DORY (for memory management) with CMSIS-NN. This was possible thanks to the modular architecture of DORY and required only changing the computational backend and adapting the code generator to use the correct DMA hardware abstraction layer calls.

The results are shown in Fig. 12. First of all, as expected performance drops dramatically deactivating the D-Cache on the STM32: we observe a degradation of 58.5 $\pm$ 5.5 % with respect to the baseline over all the benchmark layers. More interestingly, our results also show that the software caching mechanism realized by DORY on the DTCM can achieve the same performance as the D-Cache on average, with a slight speedup in some cases: on average, 9.1$\pm$ 2.1 % for depthwise layers and 3.9 $\pm$ 3.8 % for normal convolutions.

On the other hand, single-core execution on GAP-8 shows on average a speedup of 2.5$\pm$0.9$\times$ with respect to the STM32H7 baseline in terms of cycle/cycle. Since multi-core execution is disabled in this test, the speed up achieved in GAP8 with respect to the STM32H7 is referred mainly to the more specialized architecture, and in particular to the DSP extensions extensively exploited by the PULP-NN backend.

7 Conclusion

In this work, we introduced a novel framework for DNN deployment, DORY, which unburdens the programmer from the manual optimizations of neural networks on end-nodes. As a case study, we targeted a DNN-oriented MCU, GWT GAP-8, showing that it achieves 12.6$\times$ higher energy efficiency and 7.1$\times$ higher performance compared to the industry-standard STM32H743, and up to 26.6% end-to-end inference improvement compared to the proprietary tool from GWT. Our developments are released as open-source at `https://github.com/pulp-platform/dory`. Future work will focus on adding support for stronger quantization, hardware-accelerated primitives, and emerging memory technologies to support more high-accuracy networks directly on sub 10 mW extreme edge platforms.

15. We used CMSIS-NN instead of CUBE-AI due to its open-source nature; note that according to the ST forums, the fixed-point backend of CUBE-AI is "based on the low-level ARM CMSIS-NN functions".

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. Barnaghi, 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. Bagaa, and T. Taleb, "UAV-based IoT platform: A crowd surveillance use case," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 128–134, 2017.

[3] M. Zanghieri, 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. Gürkaynak, M. Muehlberghuber, M. Gautschi, 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, "Ifq-net: Integrated fixed-point quantization networks for embedded vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 607–615.

[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 Networkson RISC-V Processors Through ISA Extensions," in *To appear at Design, Automation and Test in Europe Conference (DATE) 2020*. IEEE, 2020.

[13] G. Desoli, N. Chawla, T. Boesch, S. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, 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, "PULP-NN: 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. Saidi, P. Tendulkar, T. Lepley, and O. Maler, "Optimizing two-dimensional DMA transfers for scratchpad Based MPSoCs platforms," *Microprocessors and Microsystems*, vol. 37, no. 8, 2013.

[18] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "Adrenaline: An openvx 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 *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 13–19.

[21] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev *et al.*, "Tensorflow lite micro: Embedded machine learning on tinyml systems," *arXiv preprint arXiv:2010.08678*, 2020.

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

[23] 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.

[24] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study of transformer networks," *arXiv preprint arXiv:2007.00072*, 2020.

[25] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "Dmazerunner: Executing perfectly nested loops on dataflow accelerators," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–27, 2019.

[26] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[27] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina *et al.*, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 369–383.

[28] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[29] L. Geiger and P. Team, "Larq: An Open-Source Library for Training Binarized Neural Networks," *Journal of Open Source Software*, vol. 5, no. 45, p. 1746, Jan. 2020.

[30] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," 2017.

[31] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018.

[32] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[33] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny Deep Learning on IoT Devices," 2020.

[34] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "CMix-NN: Mixed Low-Precision CNN Library for Memory-Constrained Edge Devices," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[35] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized Clipping Activation for Quantized Neural Networks," *arXiv preprint arXiv:1805.06085*, 2018.

[36] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[37] M. Rusci, A. Capotondi, F. Conti, and L. Benini, "Work-in-progress: Quantized nns as the definitive solution for inference on low-power arm mcus?" in *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2018, pp. 1–2.

[38] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

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

[40] F. Conti, "Technical Report: NEMO DNN Quantization for Deployment Model," 2020.

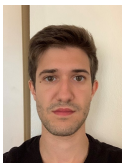
[41] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

[42] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, 2019.

[43] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters," in *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–6.

[44] D. Rossi, I. Loi, G. Haugou, and L. Benini, "Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014.

[45] A. Pullini, D. Rossi, G. Haugou, and L. Benini, "μDMA: An autonomous I/O subsystem for IoT end-nodes," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 2017, pp. 1–8.



Alessio Burrello received his B.Sc and M.Sc degree in Electronic Engineering at 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, Italy. His research interests include parallel programming models for embedded systems, machine and deep learning, hardware oriented deep learning, and code optimization for multi-core systems.

Angelo Garofalo received the B.Sc and M.Sc. degree in electronic engineering from the University of Bologna, , Italy, in 2016 and 2018 respectively. He is currently working toward his Ph.D. degree at University of Bologna. 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, and embedded architectures.

Nazareno Bruschi received the M.Sc degree in Electronic Engineering at the University of 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, Italy, in 2017. He is currently an Assistant Professor 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, and run-time optimization for multicore and many-core accelerators, and design of software stacks for emerging computing architectures.

Davide Rossi , received the PhD from the University of 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, runtime support to address performance, and energy efficiency of ultra-low-power computing platforms. In these fields he has published more than 100 papers in international 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, 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 postdoctoral researcher at the Integrated Systems Laboratory of ETH Zurich in the Digital Systems group. His research focuses on the development of deep learning based intelligence on top of ultra-low power, ultra-energy efficient programmable Systems-on-Chip. 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.