This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

Optimising Convolutions for Deep Learning Inference on ARM Cortex-M Processors

Antonio Maciá-Lillo, Sergio Barrachina, Germán Fabregat, Manuel F. Dolz

***Abstract*—We perform a series of optimisations on the convolution operator within the ARM CMSIS-NN library to improve the performance of deep learning tasks on Arduino development boards equipped with ARM Cortex-M4 and M7 microcontrollers. To this end, we develop custom microkernels that efficiently handle the internal computations required by the convolution operator via the lowering approach and the direct method, and we design two techniques to avoid register spilling. We also take advantage of all the RAM on the Arduino boards by reusing it as a scratchpad for the convolution filters. The integration of these techniques into CMSIS-NN, when invoked by TensorFlow Lite for microcontrollers for quantised versions of VGG, SqueezeNet, ResNet, and MobileNet-like convolutional neural networks enhances the overall inference speed by a factor ranging from 1.13× to 1.50×.**

***Index Terms*—Deep learning, Convolution operator, Edge computing, Microcontrollers, High performance, ARM Cortex-M, CMSIS-NN.**

I. Introduction

DEEP neural networks (DNNs) have demonstrated impressive achievements in a variety of machine learning tasks such as image classification, speech recognition, and object detection [1]–[3]. However, even the mild computational requirements of DNN inference pose severe constraints on the deployment of these technologies on the embedded devices that are common in Internet-of-Things (IoT) scenarios, due to their limited autonomy (battery life), computational power and memory capacity, as well as security and latency issues [4]. To address these constraints, multiple efforts have emerged to enable inference on the edge, including the design of light-weight DNN models, such as MobileNet [5] or SqueezeNet [6]; the application of advanced quantisation techniques; and the development of specialised inference frameworks for embedded devices, including TensorFlow Lite for Microcontrollers (Google), Embedded Learning Library (Microsoft), PyTorch Mobile (Facebook), microTVM (Apache), and Cube AI19 (STM).

For microcontroller units (MCUs) that integrate ARM Cortex-M processors [7], the aforementioned inference frameworks heavily rely on ARM's Common MCUs Software Interface Standard for Neural Networks (CMSIS-NN) library to deliver high performance at a low energy cost [8]. CMSIS-NN comprises a set of optimised functions for distinct types of neural network operator layers, such as fully-connected, convolution, pooling, or non-linear functions. In general, these operators are implemented using optimised algorithms and data structures, specifically tailored to run efficiently on Cortex-M processors. For the case of the convolution operator, CMSIS-NN v2.0.2 implements two algorithms: 1) the lowering variant, in which the input image is reordered and expanded column-wise using the IM2COL transform [9] to cast the convolution into a general matrix-matrix multiplication (GEMM) [10]; and 2) the direct algorithm, which consists of 7 nested loops that iterate over the dimensions of the output activations and filters [11]. The latter algorithm is only leveraged for the ARM Cortex-M0/-M3 processor series while the former is utilised in devices supporting the M-Profile Vector Extension (MVE) or Digital Signal Processing (DSP) instructions, as is the case of the ARM Cortex-M4 and M7 processors [12], [13].

In this work we tackle the acceleration of the convolution operators for DNN inference on ARM Cortex-M4 and M7 processors, introducing a number of optimisations that further improve the performance of both the lowering and direct convolution algorithms already available in CMSIS-NN v2.0.2.

In more detail, we make the following specific contributions:

- We reformulate the microkernel for the lowering convolution algorithm in CMSIS-NN into a new 2×3 vectorised variant that improves performance and data reuse.
- We design and incorporate a new vectorised micro-kernel for the direct convolution algorithm in CMSIS-NN which operates with blocks of 2×2 elements. This microkernel carefully uses well-known computer architecture techniques, such as loop unrolling and vector instructions to improve data reuse and performance.
- We propose two novel techniques that mitigate register spilling in the aforementioned microkernels by handling matrix operands leading (static) dimensions using: 1) several microkernels for different leading dimensions that use constant literals; and 2) run-in-RAM microkernels that can be modified at run time to tailor the current leading dimensions.
- We present an effective strategy that migrates the convolution filters from flash memory to RAM. Since accesses to data in flash are typically delayed between two and three cycles in these types of low-memory ARM Cortex-M-based MCUs, placing the filters in RAM generally reduces access delays and improves overall performance.

S. Barrachina, G. Fabregat, and M. F. Dolz are with the Department Computer Science and Engineering, Universitat Jaume I, Spain.
E-mail: {barrachi,fabregat,dolzm}@uji.es
Antonio Maciá-Lillo is with the Department of Technology Information and Computation, Universitat d'Alacant, Spain.
E-mail: a.macia@ua.es

0000–0000/00$00.00 © 2023 IEEE

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

- We perform a complete experimental evaluation of the gains obtained by both convolution variant optimisations, along with the use of constant literals/self-modifying code techniques and the use of RAM as scratch memory. For this purpose, we target VGG-, SqueezeNet-, ResNet-, and MobileNet-like CNNs, on the ARM Cortex-M4 and M7 processors embedded in the Arduino Nano 33 BLE Sense and Portenta M7 Lite MCUs. The overall results of the optimised implementation reveal inference speedups ranging from 1.13× to 1.50×.

The rest of the paper is structured as follows. Section II reviews some related work in the area and describes some of the relevant techniques to optimise convolutions on MCUs. In Section III, we briefly review the ARM Cortex-M4 and Cortex-M7 microarchitectures, their memory organisation, and the proposed strategy to utilise the RAM as a scratchpad memory. Section IV presents our techniques to accelerate both the lowering and direct convolution algorithms along with the optimisation strategies proposed in this work. In Section V, we evaluate the global inference time performance improvements of different DNNs and convolution layer levels achieved by the new microkernels, employing techniques to mitigate register spilling. Finally, Section VI concludes the paper with a few remarks and a brief discussion on potential future work.

II. Background and related work

This section provides the necessary background to understand the main techniques used in this work for the optimisation of convolutional operators and their integration into IoT applications for deep learning on ARM Cortex-M processors. It also provides an overview of the existing literature on framework development, the use of optimised DL models, and the optimisation of convolutional operators for MCUs, highlighting their contributions to the field and delineating how this work differs from existing approaches.

A. Background

Improving the performance of DNN operators for MCUs requires the application of key strategies within strong resource constraints. Specifically, in the context of convolution, optimisation is framed as a GEMM operation when using the IM2COL transform [10] or the direct algorithm [11]. Different from other approaches in the literature, our work is characterised by building on the CMSIS-NN framework [8] to refine the convolution operation through microkernel optimisation. This focus on the innermost computational component of GEMM proves crucial in achieving higher performance levels.

A key consideration in optimising the GEMM microkernel is to improve the ratio of floating-point or integer operations to the number of bytes (or operands) loaded from RAM into registers. On regular CPUs, matching GEMM microkernel operand sizes ($C = A \cdot B$) to cache levels (e.g. L1, L2 and L3) improves overall performance [14]. However, MCU-oriented processors such as the ARM Cortex-M series typically lack caches, so the optimisation strategy shifts to maximising processor register utilisation by increasing microkernel operand sizes. This approach effectively maximises the ratio of operations per transferred byte but is only feasible up to a certain threshold before requiring more registers than the ones available in the processor architecture. Beyond this critical point, the occurrence of *register spilling* forces the eviction of microkernel variables from registers into RAM and the subsequent reloading into registers during execution, as discussed in [15]. This process heavily degrades performance due to compiler-inserted store/load instructions to/from RAM.

To mitigate this effect, certain low-level and compilation strategies can effectively reduce the number of registers used by the microkernel to fit the processor register set. One approach is to convert certain static microkernel parameters, initially defined as variables in the code, to literals and then replicate the microkernel code for different literal values. In extreme cases, however, this method may be impractical due to limited flash space. Alternatively, special instructions and pragmas can be used at compile time to place the code in RAM, allowing it to *execute from RAM* after dynamically altering the literals encoded in the instructions to the desired values [16]. Although implementing this technique requires identifying the instructions (and addresses) where the literals are located, it allows the microkernel to accommodate slightly larger operand sizes by repurposing the registers previously allocated to constant values, thereby improving performance without incurring the penalties associated with register spilling.

Another consideration is that convolution filters are stored in flash by default, as they are static during the inference stage. However, accessing them from flash is much slower than accessing them from RAM. Given this, another optimisation that we have taken into account in this work is to copy the filters from flash to RAM, provided that there is enough space, in order to improve the overall convolution performance.

B. Related work

a) Deployment and optimisation frameworks for MCUs: Nguyen et al. [17] assemble a state-of-the-art family using the open-source NNoM deployment framework. They perform an experimental characterisation of convolution operator implementations and observe a linear relationship between theoretical multiply-accumulate operations (MACs) and energy consumption, highlighting the benefits of using computationally efficient primitives such as shift convolution. In the study, they highlight the impact of SIMD instructions and data reuse in reducing latency and power consumption.

Similarly, Deutel et al. [18] present a framework for exploring different DNN pruning, quantisation and deployment strategies specifically designed for low-power ARM Cortex-M-based systems. Through this exploration, trade-offs between accuracy, memory consumption, execution time and power consumption are analysed. Experimental results using the CMSIS-NN library show that DNN compression can reduce the number of parameters without significant loss of precision.

In contrast to existing approaches focused on analysing and optimising different operators through pruning and quantisation, our work is targeted at reducing the inference time of the convolution operator within the ARM Cortex-M processors.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

We achieve this by implementing a number of innovative techniques to improve the performance of the convolution algorithm available in the CMSIS-NN library.

b) Optimising DL inference on MCUs: Related to this category, Grzymkowski et al. [19] performed a performance analysis of CNNs deployed on an NXP i.MX RT1050 development board equipped with an ARM Cortex-M7 core. They investigated the impact of various factors, including core frequency, memory access and DSP instruction usage, using TensorFlow Lite and CMSIS-NN as the inference engines. They also highlight the importance of using DSP instructions, considering memory access latency, and optimising core frequency and cache configurations to achieve an optimal system.

In a similar work, Cerutti et al. [20] developed an IoT outdoor sound event detection DL application on an STM32L476RG platform equipped with an ARM Cortex-M4 processor. They trained a VGG-like CNN via knowledge distillation for extreme compression using the UrbanSound8K dataset. The evaluation, taking into account performance, power consumption and accuracy, reveals the importance of using an efficient SIMD convolution implementation for 8-bit quantisation, which is enabled by CMSIS-NN.

Faraone et al. [21] adapted a convolutional recurrent NN designed for cardiac arrhythmia detection to be deployed on the nRF52 system-on-chip (SoC) equipped with an ARM Cortex-M4 processor. The work focused on the inference process and the trade-offs between model complexity and performance degradation, using the CMSIS-NN library.

Also, Sadiq et al. [22] propose a novel approach to TinyML that challenges conventional methods, demonstrating that the use of efficient models with low inference latency, unbound by internal memory constraints, can outperform traditional approaches. By leveraging external memory and using the TinyOps inference framework, their method achieves up to 6.7% higher accuracy and 1.4× faster latency for TinyML ImageNet classification compared to state-of-the-art internal memory approaches.

In addition, Liberis and Lane [23] address the challenge of deploying CNNs on MCUs by modifying the execution order of layer operators, independent of other compression methods. For that, they propose a novel approach to reduce memory usage through a tool for reordering operators in TensorFlow Lite models and demonstrate its effectiveness by significantly decreasing the memory footprint of a CNN, enabling deployment on an NUCLEO-F767ZI prototyping board with 512KB SRAM.

Although previous works focused on performance-energy-accuracy trade-off analyses, SIMD factorisation of convolution operators, compression methods for traditional models, or even the use of efficient models with reduced memory footprint, our approach is significantly different in that we propose a number of innovative techniques to improve performance, maximise data reuse and reduce latency for the operands of the convolution operator.

c) Optimised convolutional operators: Lai et al. [24] present CMSIS-NN, one of the first libraries for Cortex-M processors, which provides optimised kernels that maximise DNN performance while minimising memory requirements. The authors of this paper also present techniques to evaluate a NN architecture search within the memory/compute constraints of typical MCUs on a keyword spotting application.

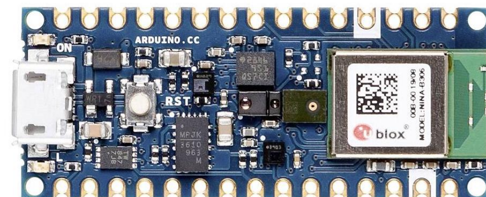
Similarly, Cho and Brand [25] proposed Memory-Efficient Convolution (MEC) to optimise IM2ROW operations by reusing data from the input feature map in the height direction. This technique reduces the IM2ROW buffer size and minimises the data copied during IM2ROW, showing improvements over traditional IM2ROW-based algorithms. Despite this, MEC has a significant memory overhead and involves some data copying during IM2ROW, although less than the size of the input tensor.

Following this trend, Wang et al. [26] presented an innovative IM2ROW-based convolution method that exploits data reuse from adjacent convolution windows to reduce memory consumption and data copying. They use a data type extension technique from `q7_t` to `q15_t` in the algorithm, eliminating data reordering instructions. Experimental evaluations show a speedup of 1.42× over CMSIS-NN across several convolutional layers.

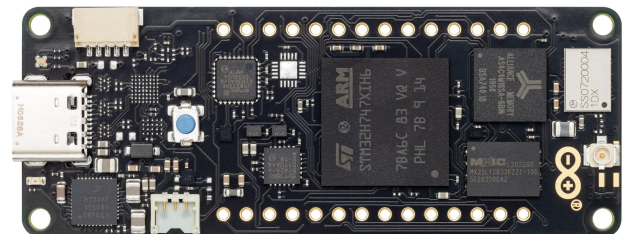
While these approaches follow the trend discussed in this work, they differ in that they focus on introducing new versions of the IM2ROW algorithm coupled with vectorised kernels to achieve significant speedup. In contrast, our techniques are specifically designed to mitigate the negative impact of register spilling as the convolution microkernel operand sizes increase.

III. ARM CORTEX-M-BASED MICROCONTROLLERS

In this section, we review in detail the target ARM Cortex-M processors architectures, with the memory organisation of the Arduino Nano 33 BLE Sense, (NANO) and the Arduino Portenta H7 Lite (PORT) MCUs targeted in this work (see Figure 1). We also present some basic performance microbenchmarks focused on timing memory accesses.



(a) Arduino Nano 33 BLE Sense.



(b) Arduino Portenta H7 Lite.

Fig. 1: Arduino MCUs used in this work *(Source: Arduino store website)*.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

A. Arduino Nano 33 BLE Sense

The NANO is a compact MCU board based on the Nordic Semiconductor nRF52840 chip with the CPU and memory organisation described next [27].

1) CPU: The NANO comprises an ARM Cortex-M4 processor equipped with a 32-bit ARMv7E-M core running at 64 MHz that implements the Thumb-2 ARM Instruction Set Architecture (ISA) and embeds 16 general-purpose registers, and a new set of Digital Signal Processor (DSP) and Single Instruction, Multiple Data (SIMD) instructions [12]. The core can compute one 32-bit or two 16-bit multiply-and-accumulate operations in a single cycle with saturated results. Additionally, the processor has a single-precision floating-point unit (FPU) and a comprehensive suite of data transfer instructions, enabling efficient packing and unpacking of values for the vector units.

The 3-stage pipeline CPU effectively manages data hazards associated with memory accesses that involve registers used for effective address calculation and data loading. By carefully scheduling consecutive memory access instructions, the delays caused by the dependency hazards can be minimised.

2) Memory organisation: The memory of the NANO utilises flash memory for code and constant data, and SRAM for variables, with flash in general being slower than the SRAM. The nRF52840 MCU in NANO features 1 MiB of flash and 256 KiB of SRAM [28]. The NMVC flash controller incorporates an instruction-only cache. Therefore, instruction accesses in flash can be cached by the memory controller. The same does not hold for data flash accesses, leading to a delay of two or three cycles for those, depending on the access pattern. Although the code is typically retrieved directly from the flash memory, it is also possible to instruct the boot loader to allocate a code portion in the SRAM. This enables the designated code segment to be modified at run time, providing flexibility for introducing changes in the code while the program is running.

B. Arduino Portenta H7 Lite

The PORT is a powerful microcontroller board based on the STMicroelectronics STM32H747XI chip [29]. It is designed for demanding industrial applications that require high computational power and low latency communication [30]. The CPU and memory organisation of this device are described next.

1) CPU: The PORT features a dual-core ARM processor with Cortex-M7 and Cortex-M4 cores. The Cortex-M7 shares the same ISA as in the Cortex-M4, including the DSP and SIMD extensions, but delivers higher performance due to its higher operating frequency (480 MHz) and the 2-way superscalar pipeline with 6 stages and branch prediction [13]. In addition, the CPU implements a 4-way associative data cache and a 2-way associative instruction cache with 16 KiB each. The Cortex-M4 core shares the same characteristics as in the NANO, except for a higher operating frequency: 240 MHz.

2) Memory organisation: Given the complexity of the system, with two cores and their respective buses, the memory organisation of PORT is also quite complex. The STM32H747XI chip incorporates high-speed embedded memories with a dual-bank 2-MiB SRAM and 1-MiB flash memory, distributed across two TCM (Tightly Coupled Memory) buses: the AXI (Advanced eXtensible Bus) for the Cortex-M7, and the AHB (Advanced High-performance Bus) for the Cortex-M4 core. Access to flash is highly penalised in the latter as the whole flash memory is connected to AXI, and access from the Cortex-M4 is performed through bridges across the AHB bus. In general, the AXI bus design demonstrates superior throughput [31], [32]. Despite both AXI and AHB being clocked at 200 MHz, the AXI bus boasts a 64-bit width, whereas the AHB is limited to 32 bits.

C. Benchmarking memory accesses

To gain some initial insight into the memory access latency of both MCUs, we developed and executed a series of synthetic microbenchmarks on the Cortex-M4 core in NANO and the Cortex-M7/M4 cores in PORT. The first test performs contiguous and non-contiguous memory accesses to RAM/flash, while the second realises a matrix-matrix multiplication with the operands placed in flash/RAM. To prevent unexpected compiler optimisations, the microbenchmarks were implemented using assembly code inlined with the `__asm__` instruction within the Sketch Arduino program. To read data from RAM/flash, we used the ARM instruction `ldr rx, [ry, #disp]` to load 32-bit words from RAM/flash addresses (`[ry, #disp]`) to a given register (`rx`).

TABLE I: Comparison of access times (in ns) for the NANO Cortex-M4, and PORT Cortex-M7/M4 cores.

		NANO-M4	PORT-M4	PORT-M7
Contiguous accesses	RAM	16.2	5.1	1.3
	Flash	49	22	1.3
Non-contiguous accesses	RAM	16.2	5.1	1.3
	Flash	65	21	1.3

Table I details the access times (in ns) measured with our benchmark for reading contiguous and non-contiguous memory RAM and flash addresses on the three target processors. From the results, the first observation is that the memory access latency for the Cortex-M4 cores is much higher when the data is on flash compared with the data on RAM, while for the Cortex-M7 there is no difference between reading data in RAM or flash. This is due to the effect of the data cache of the Cortex-M7 and its buffers to handle memory accesses. The second observation for the Cortex-M4 cores is that accessing contiguous flash addresses is slightly slower than when these are non-contiguous. We relate this slowdown to the organisation in banks of the flash memory and the contention in the memory access buses.

TABLE II: Comparison of access times (in ms) for the NANO Cortex-M4, and PORT Cortex-M4 and M7 cores.

	NANO-M4	PORT-M4	PORT-M7
Basic	418	204	50
Copy column to RAM	355	122	39

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

Table II shows the execution time (in ms) of the matrix-matrix multiplication $C = A \cdot B$, using matrices of size 160×160, with the operands A and B stored in flash in row- and column-major order, respectively. In order to increase performance, here we unrolled the innermost loop of the multiplication by a factor of four, as four 8-bit integer values are loaded (via `ldr`) in each memory read. In this case, the first row of the table (Basic) reports the execution time of the operations when both operands are read from flash, while the second row (Copy column to RAM) refers to a version of the algorithm that copies each column of B to RAM before multiplying it by a row of A. From the results, it can be observed that the relative gain is higher for the PORT M4 core while accessing data in flash is highly penalised.

With these figures in mind, we can conclude that an optimisation for speeding up DNN inference is to copy, whenever possible, the static operands (i.e., weights and biases) from flash to RAM before performing the corresponding layer operations, and maintain them there during the computations.

IV. THE CONVOLUTION ALGORITHM

The convolution operator

$$O = \mathsf{Conv}(F, I), \tag{1}$$

receives a 4D filter tensor F, and a 4D input tensor I, to produce a 4D output tensor O, where:

- F comprises c_o filters of dimension $h_f \times w_f \times c_i$ each, where $h_f \times w_f$ correspond to the filters height $\times$ width.
- I consists of b input images of size $h_i \times w_i \times c_i$ each, with $h_i \times w_i$ denoting the images height $\times$ width, and c_i stands for the number of input channels.
- O is composed of b outputs of size $h_o \times w_o \times c_o$ each, where $h_o \times w_o$ represent the outputs height $\times$ width, and c_o is the number of output channels.

With these operands, the basic algorithm convolves a subtensor of the inputs, of the same dimension as the filter, to render a single scalar value (entry) for each one of the c_o outputs. The filter is then repeatedly applied to the whole input, in a sliding window manner, to produce the complete entries of this single output [33]. For simplicity, hereafter we will consider that the filter is applied with unit vertical/horizontal strides; and the output is not padded so that $h_o = h_i - h_f + 1, w_o = w_i - w_f + 1$.

In the following subsections, we briefly review the lowering approach and the direct algorithm to compute the convolution operator as well as the adaptations carried out in the CMSIS-NN implementation for 8-bit integer quantisation in order to accelerate the execution of these algorithms.

A. The lowering approach

A high-performance implementation of the convolution algorithm can be obtained by lowering the operation into a large matrix-matrix multiplication (GEMM) of the form $C = A \cdot B$. For this purpose, the approach proceeds as follows:

- Assuming that the input/output tensors are stored using the NHWC (Batch size, Height, Width, Channels) layout and the filters in the KRSC (Output Channels, Filter height, Filter width, Input Channels) layout, the IM2COL transforms the 4D input tensor I yielding an augmented 2D matrix B of size $k \times n = (h_f w_f c_i) \times (b h_o w_o)$; see Figure 2 [9]. Depending on the convolution parameters, the size of this augmented matrix is considerable, especially concerning the memory capacity of microcontrollers.
- Compute the output of the convolution directly from the GEMM $C = A \cdot B$, where $C \equiv O$ is the output tensor, viewed as an $m \times n = c_o \times (b h_o w_o)$ matrix; and $A \equiv F$ is the filter tensor, viewed as a $m \times k = c_o \times (h_f w_f c_i)$ matrix. This lowering approach performs the same arithmetic operations as the direct convolution and, therefore, has the same numerical properties.

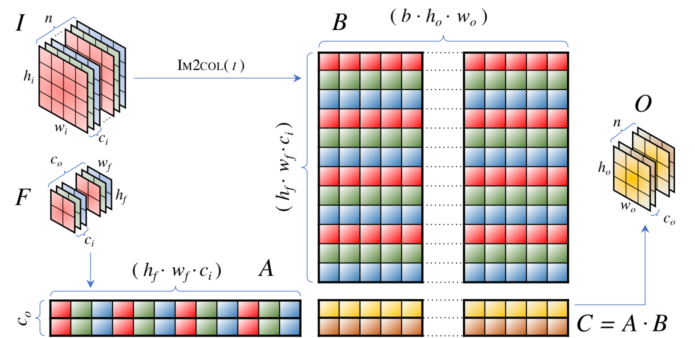


Fig. 2: Convolution operator via the IM2COL transform.

1) Lowering in CMSIS-NN: Due to the limited SRAM capacity of microcontrollers –in the order of KiB– the implementation of the lowering approach present in CMSIS-NN computes the IM2COL transform blockwise using loop unrolling techniques and allocating a small buffer $\hat{B}$, of size $(h_f w_f c_i) \times 2$, to accommodate two columns of the matrix B resulting from the IM2COL. The algorithm then performs a matrix multiplication between the filter tensor A and the two-column buffer $\hat{B}$. This proceeds by pairs of rows of A, using a tuned 2×2 microkernel (see Figure 3b), so that each iteration updates a 2×2 block of C. The algorithm in Figure 3a represents the lowering approach with blocks of γ columns, with this parameter set to 2 for CMSIS-NN.

This 2×2 microkernel uses `_SMLAD` 16-bit MAC vector instructions to multiply two pairs of signed 16-bit integers packed into two 32-bit registers and accumulate the result into a 32-bit register. Since the `_SMLAD` instruction operates with 16-bit integers, A and $\hat{B}$ are previously converted from the 8-bit `q7_t` CMSIS data type to the 16-bit `q15_t` using the `arm_q7_to_q15` utility function. In its final stage, the CMSIS-NN microkernel includes some operations to dequantise the result matrix C.

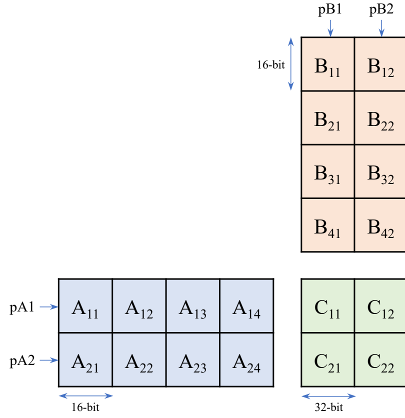
2) Microkernel optimal size: In an ideal scenario, the execution of a matrix multiplication involving two matrices, $A_{m \times k}$ and $B_{k \times n}$, would require reading each element of A and B only once, totalling $mk + kn$ memory reads. However, practical constraints arise due to the sequential execution of instructions in the default matrix multiplication algorithm. In the worst-case scenario, with large matrices and limited or no cache availability, obtaining each element of the resulting

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

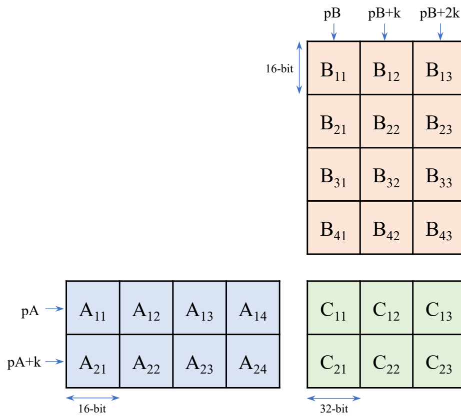
This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

L1: **for** $h = 0, \ldots, b-1$
L2: **for** $i = 0, \ldots, h_o - 1$
L3: **for** $j = 0, \ldots, w_o - 1$
$c = j + iw_o + hh_ow_o$
L4: **for** $l = 0, \ldots, h_f - 1$
L5: **for** $m = 0, \ldots, w_f - 1$
L6: **for** $n = 0, \ldots, c_i - 1$
$r = n + mc_i + lw_fc_i$
$\hat{B}[r][c \bmod \gamma] = I[h][i+l][j+m][n]$
// GEMM is realised every γ columns
if $c \bmod \gamma \; = \; \gamma - 1$
for $k = 0, \ldots, c_o - 1$, step γ
// $\gamma \times \gamma$ GEMM micro-kernel
$C[j:j+\gamma][c-\gamma:c] \mathrel{+}= A[j:j+\gamma] \cdot \hat{B}$

(a) Algorithm for the lowering convolution by blocks of γ columns.



(b) 2×2 microkernel used by CMSIS-NN.



(c) 2×3 microkernel proposed in our optimisation.

Fig. 3: The inner loop of the microkernels for matrix multiplication.

$C_{m \times n}$ matrix requires $2k$ memory reads (one row of A and one column of B), resulting in a total of $2mnk$ reads.

In contrast, a blocked matrix multiplication algorithm partitions matrices into blocks and computes each block of the resulting matrix C as the sum of the products of corresponding blocks from A and B, i.e., $C^{ij}_{m_c \times n_c} = \sum_h A^{ih}_{m_c \times k_c} \times B^{hj}_{k_c \times n_c}$. This approach allows the selection of block sizes such that the block matrices being multiplied fit into the cache, ensuring that each element is read only once from memory. This result in $m_ck_c + k_cn_c$ memory reads per blocked matrix multiplication. Consequently, each block of C requires $\frac{k}{k_c}(m_ck_c + k_cn_c) = k(m_c + n_c)$ reads, leading to a total of $mnk\frac{m_c+n_c}{m_cnc}$ reads. If

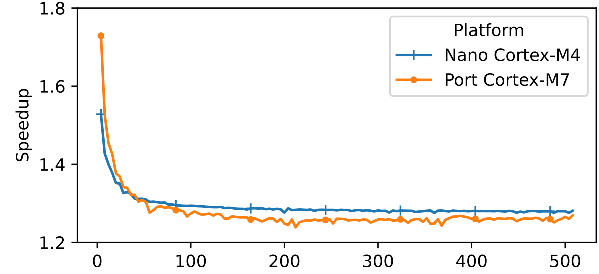


Fig. 4: Speedup of the 2×3 microkernel over the original 2×2 microkernel when multiplying matrices $A_{m \times k}$ with $B_{k \times n}$, where $m = 64$, $n = 96$, and k ranges from 4 to 512 on both NANO Cortex-M4 and the PORT Cortex-M7 processors.

m_c and n_c are greater than one, this is fewer than the $2mnk$ reads required for the worst-case scenario.

Even without a cache, a blocked matrix multiplication algorithm can still be advantageous by selecting sufficiently small m_c and n_c values to fit all the required data for a block matrix multiplication into registers. CMSIS-NN provides a 2×2 microkernel blocked matrix algorithm, resulting in a total of mnk reads —half of the reads in the worst-case scenario. To improve the data reuse, we have implemented a 2×3 microkernel, which leads to a total of $\frac{5}{6}mnk$ memory reads per matrix multiplication, representing a speedup of 1.2 over the 2×2 version.

It is important to recognise that using a larger microkernel can also lead to the following benefits: firstly, it diminishes the number of blocked matrices that need to be combined, thereby lowering the frequency of microkernel function calls; secondly, it enhances the locality of the output data; and thirdly, for MCUs, it conserves execution cycles as it allows for more sequential reads. These factors are likely to contribute to a modest enhancement in the previously mentioned acceleration.

To evaluate this, we performed a small-scale experiment involving matrix multiplication, specifically multiplying matrices $A_{m \times k}$ with $B_{k \times n}$, where $m = 64$, $n = 96$, and k ranges from 4 to 512. We compared the performance of the 2×3 microkernel against the original 2×2 microkernel on both the NANO Cortex-M4 and the PORT Cortex-M7 processors. This result, as depicted in Figure 4, shows that for sufficiently large values of k, the speedup achieved by the 2×3 microkernel stabilises at around 1.27 times that of the original microkernel.

3) Optimising the microkernel: While a slightly larger microkernel, of size 2×3, could improve data reuse, our initial attempts to implement it, mirroring the strategy used for the 2×2 variant, encountered an issue: register spilling. To solve this issue, we adopted the following approach:

- We designed a 2×3 microkernel that uses only two 32-bit registers to contain the initial addresses of A and $\hat{B}$, instead of five registers to individually reference the 2 rows of A and the 3 columns of $\hat{B}$. The individual elements of A and B are then accessed using simple arithmetic and the corresponding leading dimensions (see Figure 3c).
- To save an additional register, we propose two different approaches to handle the leading (static) dimension k of

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

the input matrices (columns of A or rows of $\hat{B}$):

1) Use of a constant literal of k in the code. For this, we moved the microkernel code into a macro that is expanded at compile time by the C preprocessor to generate a set of microkernels for particular values of k. For example, in a convolution layer with filters of size 3×3, the values of k are integers multiples of 9, where the multiplier is the number of input channels.
2) Use of a modifiable run-in-RAM version of the microkernel, where the value k can be adjusted at run time before calling it. For this approach, we annotated the microkernel function using `__attribute__((long_call, section(".data"), noinline))`, to force the compiler to place the function in RAM, i.e., in the `.data` section of the ELF binary. Next, we developed an auxiliary function to walk over the microkernel in order to encounter and annotate those instructions that depend on the value of k. With that information, a second auxiliary function modifies the microkernel code to adjust the corresponding k value each time the microkernel has to be called with a different k value.

All in all, as a trade-off for higher performance, the new 2×3 microkernel increases the buffer for storing $\hat{B}$ to a size of $(h_f w_f c_i) \times 3$. Also, handling the static k dimension requires re-declaring the microkernel for different values of k, which also involves creating a function pointer hash table to call the appropriate microkernel depending on the values of the parameters h_f, w_f and c_i of the convolution layer. The second approach to handling a static k value requires only a single copy of the microkernel function, which reduces the size of the executable, but results in higher instruction fetch costs because there is no instruction cache for in-RAM code.

4) Improving memory accesses: To further optimise the convolution operation, we made the corresponding modifications to the Arduino TensorFlow Lite for Microcontrollers (`tflite-micro`) suite to incorporate the techniques insights presented in Section III, which reported a superior efficiency of RAM accesses compared to flash. Specifically, we implemented a mechanism to copy the filter weights and biases from flash to RAM whenever enough space is available. As a result, the convolution operation can access a copy of the filters stored in RAM more efficiently. This copy operation is performed only once, before any inference, effectively compensating for any overhead incurred by the copying process during inference with subsequent samples.

B. The direct convolution algorithm

The direct convolution algorithm is organised as 7 nested loops that iterate over the operator dimensions $(b, c_o, w_o, h_o, h_f, w_f, c_i)$ performing the corresponding MAC operations between the input and the filter in order to compute the output tensor. This results in a certain memory access pattern, depending on the ordering of the loops and the layout of the tensors in memory. It is worth noting as well that the 7 loops in the algorithm are independent of each other. As a consequence, the algorithm loops can be reorganised in any order while still producing the correct result.

1) The direct convolution in CMSIS-NN: The direct convolution algorithm in CMSIS-NN v2.0.2 was only used for the ARM Cortex-M0/-M3 processor series, as these do not support DSP nor Cortex-M vector instructions. In such cases, the naive convolution algorithm iterates through the dimensions $b \to c_o \to w_o \to h_o \to h_f \to w_f \to c_i$ from the outermost to the innermost loop to multiply the corresponding elements of I and F and accumulate the result into O (see Figure 5a). On the one hand, this implementation on the ARM Cortex-M4 core is much less efficient than the original CMSIS-NN lowering convolution algorithm, as it is not vectorised and does not make use of an optimised microkernel. On the other hand, in contrast with the lowering approach, the direct algorithm does not require any additional memory workspace.

2) Optimising the direct convolution algorithm: Taking into account the insights gained with the implementation of the lowering algorithm, the optimised version of the direct algorithm presents the following characteristics:

- We reorder the loops traversing O and F from outermost to innermost as: $b \to h_o \to w_o \to c_o \to h_f \to w_f \to c_i$, with the dimension c_o traversed in the fourth loop, favouring, therefore, the NHWC data layout.
- The previous reordering permits the implementation of a microkernel that calculates a subtensor of O of size $\gamma \times \delta$ in two innermost dimensions, i.e., w_o and c_o. For that, both w_o and c_o loops are unrolled with factors γ and δ, respectively. Figure 5b depicts the 2×2 microkernel. These unrollings also enable some degree of data reuse. Specifically, the unrolling of w_o reuses the filter data, while the unrolling of c_o reuses the input panel where the filter is applied in the image. This reduces the number of memory access operations, as explained in Section IV-A2. Although large unrolling ratios γ and δ in principle provide higher efficiency, there exists an upper bound that depends on the number of hardware registers. Considering the 16 registers in the Cortex-M4 and Cortex-M7 cores, we determined that a microkernel of size 2×2 does not generate register spilling provided we employ the same techniques as in the lowering microkernel, i.e., using k_1 and k_2 as constant literals in the code. This can be achieved either by multiple declarations of the microkernel for different values of k_1 and k_2 via C macros, or by implementing a run-in-RAM microkernel, where these values are adjusted at run time, before its execution.
- Similar to the microkernel for the lowering algorithm, the microkernel for the direct convolution uses `_SMLAD` instructions to multiply two pairs of signed 16-bit integers belonging to F and I, packed into two 32-bit registers, and accumulate the result into a 32-bit register that will be part of the 2×2 panel computed by the microkernel.

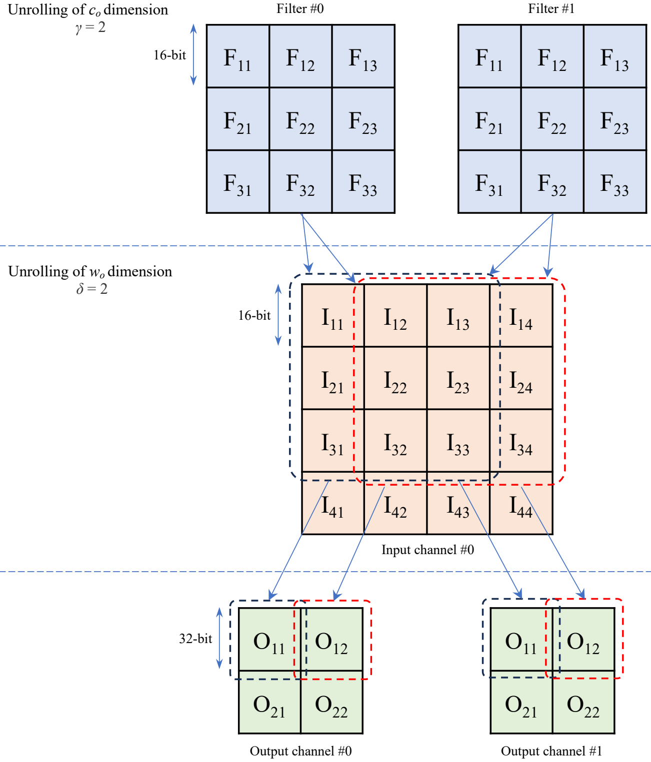
3) Improving memory access latency: As in the lowering convolution, the filters accessed by the new microkernel for direct convolution can be copied to RAM to speed up their access. This technique is implemented within the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335


```
L1: for h = 0, ..., b − 1
L2:   for i = 0, ..., h_o − 1
L3:     for j = 0, ..., w_o − 1
L4:       for k = 0, ..., c_o − 1
L5:         for l = 0, ..., h_f − 1
L6:           for m = 0, ..., w_f − 1
L7:             for n = 0, ..., c_i − 1
                  O[h][i][j][k] =
                    I[h][i + l][j + m][n] · F[k][l][m][n]
```

(a) Algorithm for the direct convolution.



(b) 2×2 microkernel applied for the direct convolution. The microkernel produces four output values, two on different columns (w_o dimension) and two on different output channels (c_o dimension).

Fig. 5: Algorithm and microkernel used for the direct convolution.

`tflite-micro` framework, on top of the CMSIS-NN library.

In general, the optimisation of the direct convolution algorithm builds on the insights gained from the lowering algorithm. With the reordered loops and the implementation of a microkernel, it is possible to achieve significant improvements, with the additional advantage that no significant workspace is required. The 2×2 microkernel combined with any of the aforementioned techniques to treat k_1 and k_2 as constants and avoid register spilling, as well as the mechanism for copying filters to RAM, improve the overall performance of this operation, which by default is not available in CMSIS-NN with DSP-enabled Cortex-M processors.

V. Experimental Results

In this section, we carry out an experimental evaluation of the optimisations proposed in this work for the lowering and direct convolution algorithms using two different CNNs in the Nano and Port MCUs. Specifically, we evaluate the performance benefits of the new algorithm microkernels combined with the techniques to avoid register spilling and to accelerate memory accesses by preloading the filters to RAM.

A. Hardware setup

As detailed in Section III, the evaluation of the algorithms is conducted on the ARM Cortex-M4 processor, operating at 64 MHz, specifically on the Nordic Semiconductor nRF52840 chip used in the Nano MCU. The Nano board features 256 KiB of SRAM and 1 MiB of flash memory. Furthermore, we utilised both the ARM Cortex-M4 and Cortex-M7 processors, operating at 240 MHz and 480 MHz respectively, on the STMicroelectronics STM32H747XI board chip, as part of the Port MCU. The microcontroller provides 1 MiB of SRAM and 2 MiB of flash memory.

B. DL framework and libraries

We utilised the EloquentTinyML v2.4.0, an Arduino library that facilitates the deployment of TensorFlow Lite (TFLite) models on Arduino boards using the Arduino integrated development environment (IDE). This package incorporates the Arduino TensorFlow Lite for Microcontrollers library v2.4.0-alpha internally linked with the CMSIS-NN library v2.0.2. The latter offers optimised functions specifically designed for microcontrollers, enabling efficient execution of DNNs on ARM Cortex-M platforms.

To automate the compilation and uploading of the binaries to the MCUs, we utilised the `arduino-cli` tool v0.33.0. For the compilation, we set the optimisation flag `-O3` for high performance. Additionally, for the Cortex-M4 processor from Nano, we also tested with the flag `-Os` for binary size reduction.

C. Testbed

To evaluate our optimisations, we trained four custom DNNs trimmed to fit the flash requirements of the MCUs: 1) VGG-like [34] (see Table A1); 2) SqueezeNet-like [6] (see Table A2); 3) ResNet-v1-like [35] (see Table A3)); and 4) MobileNet-v1-like [5] (see Table A4). In our experiments DNNs 1)–3) were trained on the CIFAR-10 dataset, while 4) was trained on the Visual Wake Words dataset [36]. The parameter reduction technique applied to these models consisted of removing blocks of layers (e.g. Conv–ReLU–BN for VGG or Conv–BN–ReLU for MobileNet) from the DNN models, preserving the layer patterns of the original DNN architecture until they fit into the flash memory of the target MCUs. It is important to note that convolutional layers using filters of size 1×1 in the SqueezeNet-like model are natively supported by CMSIS-NN, which provides a highly efficient fallback implementation for this case. We leave the optimisation of this convolution variant as part of future work.

The training of the aforementioned models was conducted on a separate server platform equipped with an NVIDIA A100 GPU using the TensorFlow library v2.4.0. After training, these models were converted to TFLite format using the `DEFAULT` optimisation flag and the `TFLITE_BUILTINS_INT8` flag

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

for full `int8` quantisation. As an example, Table A5 shows the transformations of the trimmed VGG-like model after exporting to TFLite using full 8-bit integer quantisation. Note that the BN layers have been transformed into pairs of pointwise multiplication (Mul) followed by addition (Add) layers.

It is important to note that for full 8-bit integer quantisation, the ranges (min, max) of all floating-point tensors within the DNNs, such as the model input, intermediate layer activation outputs, and model output, should be calibrated. To achieve this calibration, we used a representative dataset obtained from a subset of CIFAR-10 consisting of 500 randomly selected images, ensuring an equitable distribution of image classes (or labels) across this dataset. Additionally, to reduce potential measurement variations, all experiments reported next show the average results obtained from running the DNNs for 50 inferences. This approach ensures an accurate performance assessment of the optimised convolution algorithms on the target MCUs.

D. Cumulative results on the evaluated CNNs

In this subsection, we present the cumulative execution time of the different convolution variants for the inference of the evaluated CNNs on the ARM Cortex-M processors of both NANO and PORT. The experiments were conducted using the `-O3` compilation flag.

Figure 6 shows the cumulative execution time of the VGG-like (left column) and the SqueezeNet-like (right column) CNN layers during a sample inference using the different variants of the direct and lowering convolution algorithms running on the NANO Cortex-M4 and the PORT Cortex-M4 and Cortex-M7 processors. Figure 7 shows the same information as Figure 6 but for the ResNet-v1-like (left column) and MobileNet-v1-like (right column) CNN layers.

The results clearly demonstrate that the optimised 2×3 microkernel with k-constant via different functions and filters preloaded in RAM achieves the shortest execution time in all scenarios. They also show that on the PORT Cortex-M7 processor, the effect of preloading the filters in RAM is hardly noticeable, given that in this case the accesses to flash and RAM exhibit nearly the same latency (as previously reported in Table I).

Additionally, it can be observed that the optimised direct convolution variants achieve a similar or better performance than the original CMSIS-NN lowering method on the VGG, ResNet-v1 and MobileNet-v1-like CNNs. However, on the SqueezeNet-like model, the proposed direct convolution methods, although being far more efficient than the CMSIS-NN direct convolution method, result in a slightly worse performance than the original CMSIS-NN lowering method (especially for those cases which do not benefit from preloading the filters into RAM).

It should be noted that the ratio of execution times between the CMSIS-NN lowering convolution and direct algorithms varies widely depending on the target processor, so it is not fair to assess the performance benefits based solely on the separation of the lines in the plots of Figures 6 and 7. For example, the speedup for the different configurations and algorithms in the case of VGG and ResNet on the PORT Cortex-M4 processor appears to be less significant compared to the results using other CNNs and processors. Nevertheless, this is due to the default CMSIS-NN direct convolution algorithm producing suboptimal results in these scenarios, which are almost of a different order of magnitude concerning the other methods, causing the lines for the rest of the algorithms to appear much more concentrated.

For a more comprehensive comparison of speedups across the different CNNs, Table III shows the inference speedup achieved by the best configuration over the default CMSIS-NN lowering algorithm for each combination of CNN and processor. It is important to note that, while some of the reported speedups may appear modest, these methods are exclusively targeted at reducing the execution time of the CNN convolutional layers in the already highly optimised codes present in CMSIS-NN v2.0.2.

TABLE III: Inference speedup of the best algorithm with respect to the default CMSIS-NN lowering algorithm with 2×2 microkernel for each combination of CNN and processor.

	NANO-M4	PORT-M4	PORT-M7
VGG	1.28	1.13	1.33
SqueezeNet	1.50	1.26	1.36
ResNet-v1	1.13	1.19	1.13
MobileNet-v1	1.27	1.15	1.18

E. Convolutional layers results

In this subsection, we present the effect of our improvements on the convolutional layers of the aforementioned VGG-like CNN on both NANO and PORT MCUs. We only provide results for this CNN as similar results were observed for the other CNNs.

Figures 8 to 11 show the speedups of the lowering (top row) and direct (bottom row) convolution algorithms for the convolutional layers of the VGG-like model for the different MCUs and compilation options. The plots in the left column show the speedups achieved when the convolutional filters are read from flash, while the plots in the right column show the results when they are preloaded into RAM. It is important to note that the original CMSIS-NN lowering convolution was used as the baseline for calculating all the speedups.

In these figures, the speedups achieved solely from the optimised algorithms are visible in the left column plots of Figures 8 through 11. Likewise, the speedup attributed solely to preloading filters into RAM is depicted in the first bar of each layer within the plots on the right column. Lastly, the remaining bars in the right column plots display the combined speedup achieved through the combination of the proposed algorithms and the technique that preloads the filters into RAM.

1) Results on the NANO *MCU:* Figure 8 shows the speedups achieved on the ARM Cortex-M4 processor in NANO, compiled with the optimisations delivered by the optimise for size flag (`-Os`). It should be noted that this is the default optimisation used by the Arduino IDE and CLI.

For the lowering algorithms while reading filters from flash memory (top-left plot), the improved 2×3 microkernel

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

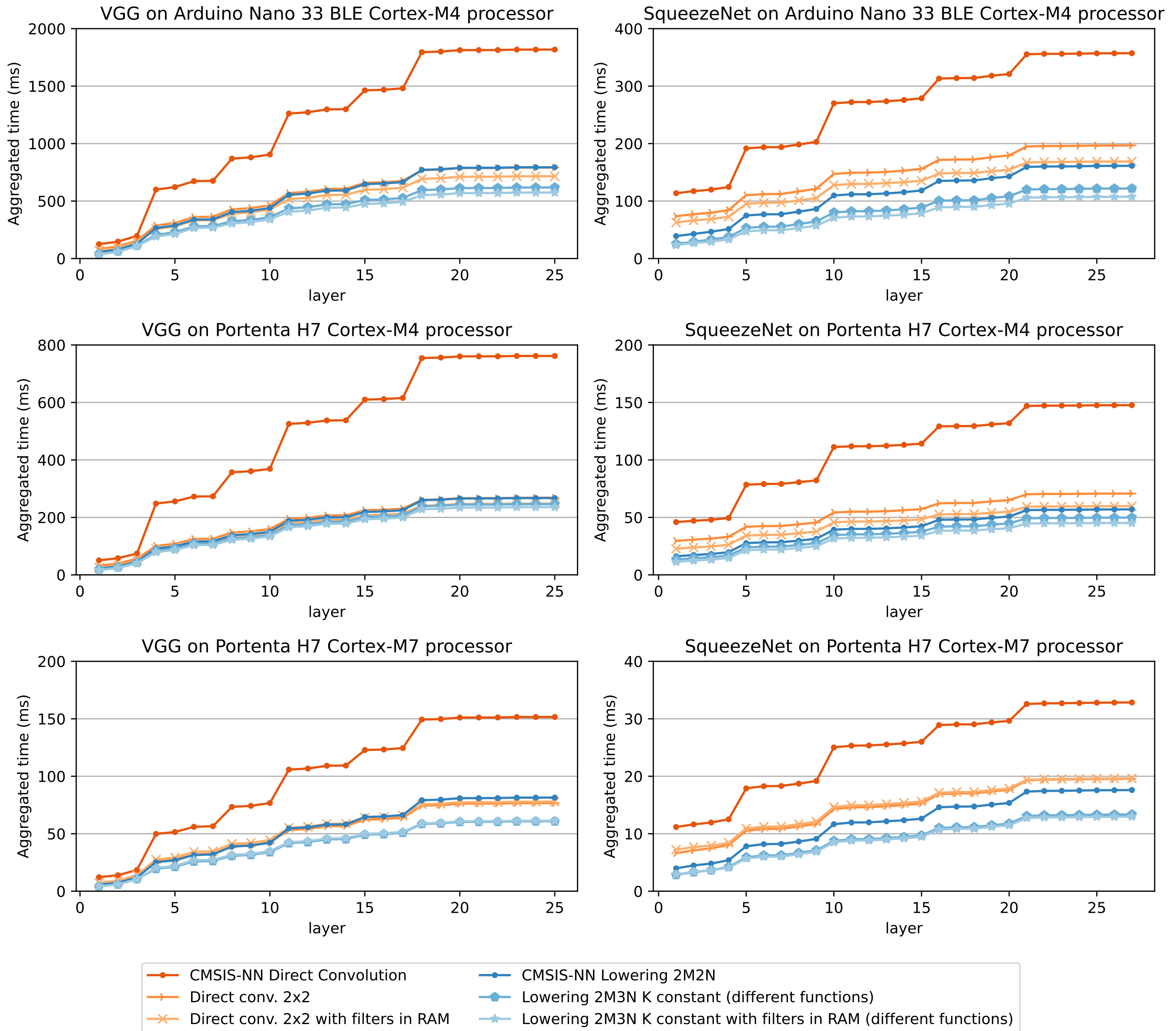


Fig. 6: Cumulative execution time for the VGG-like and SqueezeNet-like CNN layers with batch size 1 using the direct and lowering convolution algorithms on the NANO Cortex-M4, and PORT Cortex-M4/-M7 processors.

using a k-variable shows an average performance improvement of around 1.25×. Employing the k-constant with different function versions leads to a higher speedup of nearly 1.45×, primarily by reducing the number of registers utilised by the microkernel. However, if the k-constant is implemented via a run-in-RAM function, the speedup diminishes as this MCU lacks an instruction cache for RAM reads, impacting its performance. Despite its slightly lower performance, this technique offers benefits in terms of reduced binary sizes and improved portability by eliminating the need to replicate hardcoded functions with different k values.

Regarding the lowering algorithms with filters preloaded to RAM (top-right plot), we observe that all the variants improve their performance. The approach employing the k-variable achieves a speedup of 1.45×, while the versions using k-constant with different functions and k-constant via a run-in-RAM function generally achieve speedups of around 1.60× and 1.49×, respectively. Overall, the act of copying the filters to RAM leads to an approximately 10% increase in performance.

As for the direct convolution algorithms with filters read from flash memory (bottom-left plot), it can be observed that there is an initial notable performance gap between the default CMSIS-NN direct and lowering convolution algorithms. However, except for the first layer, the version using our 2×2 microkernel shows significant progress, catching up and even outperforming the standard CMSIS-NN lowering convolution algorithm by 1.3×. In this case, the speedups achieved by both k-variable and k-constant variants show comparable performance improvements.

Finally, in the case of the direct convolution algorithms with filters preloaded to RAM (bottom-right plot), both versions of our direct convolution algorithm exhibit speedups ranging from 1.25× to 1.50×. This results in an approximate 12%

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

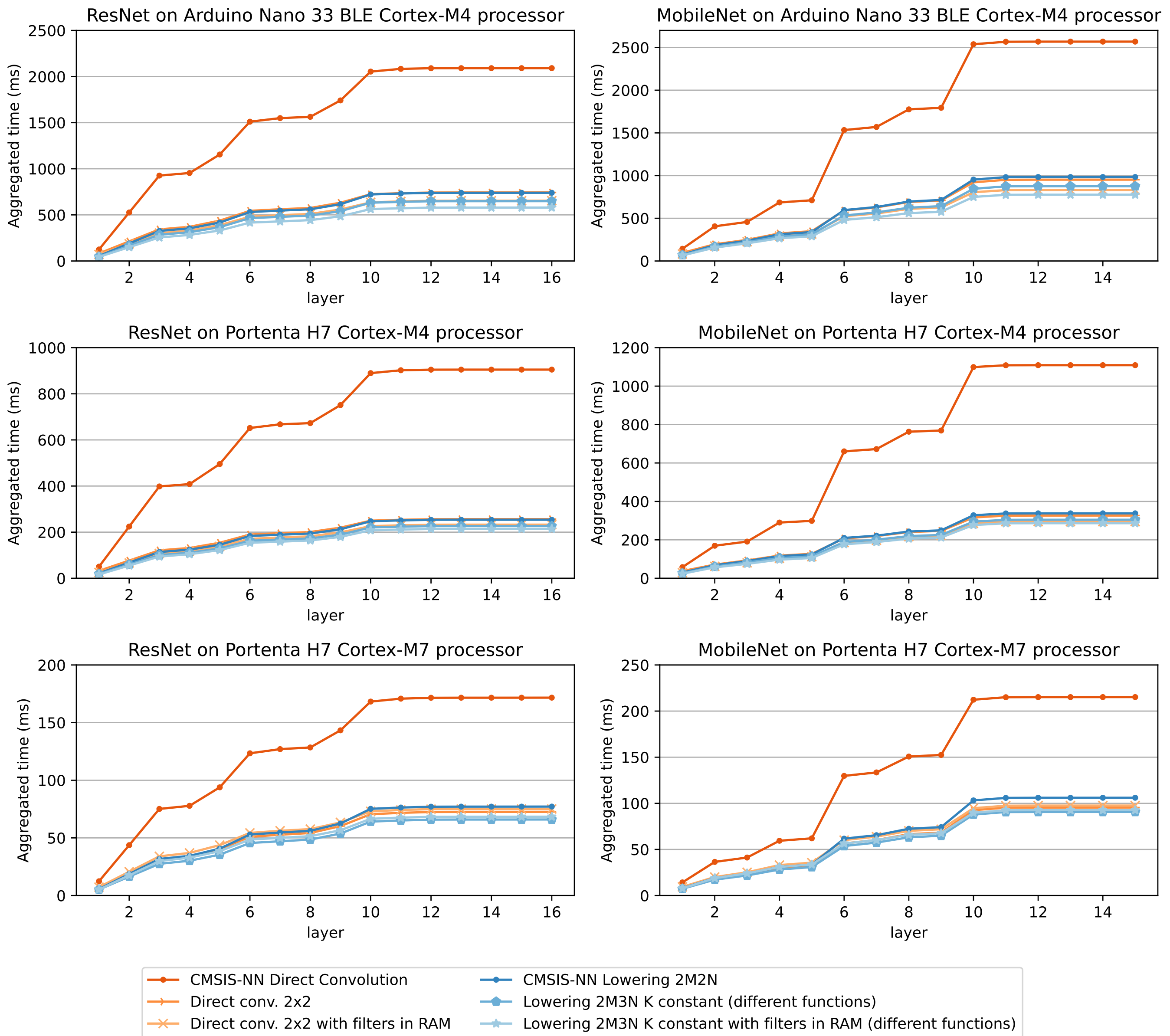


Fig. 7: Cumulative execution time for the ResNet-v1-like and MobileNet-v1-like CNN layers with batch size 1 using the direct and lowering convolution algorithms on the NANO Cortex-M4, and PORT Cortex-M4/-M7 processors.

improvement in processing time for this particular CNN configuration.

Figure 9 illustrates the results for the same experiments as in Figure 8, except that the code has been compiled turning on all the optimisations delivered by the -O3 flag (which prioritises execution time over code size). Recall that in this case, the baseline CMSIS-NN lowering convolution algorithm compiled with -O3 is used for computing all the speedups in the plots.

In the results obtained for our lowering convolution algorithm using the 2×3 microkernel (top-left plot), we notice that employing the k-variable has a negative impact on performance. We attribute this decline to the fact that the optimisation flag applied to the baseline slightly improves performance in general, making the effects of register spilling more pronounced in this particular configuration. On the other hand, the other two variants, which involve using k-constant (either by declaring different functions or by using the run-in-RAM function), achieve speedups ranging from $1.08\times$ to $1.16\times$, with a slight performance loss for the run-in-RAM variant.

When filters are preloaded into RAM (top-right plot), there is a general improvement in performance for all variants of the algorithm, around 10%. This allows the version using k-constant with different functions to achieve a speedup of above $1.24\times$ for all CNN layers. It is worth noting that the performance obtained through our improvements increases as the CNN layers progress.

Regarding the direct convolution algorithm (bottom-left plot), except for the first layer, we observe that the speedups reach $1.14\times$, with both algorithm versions delivering very similar results.

Finally, The direct convolution experiments with the filters

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

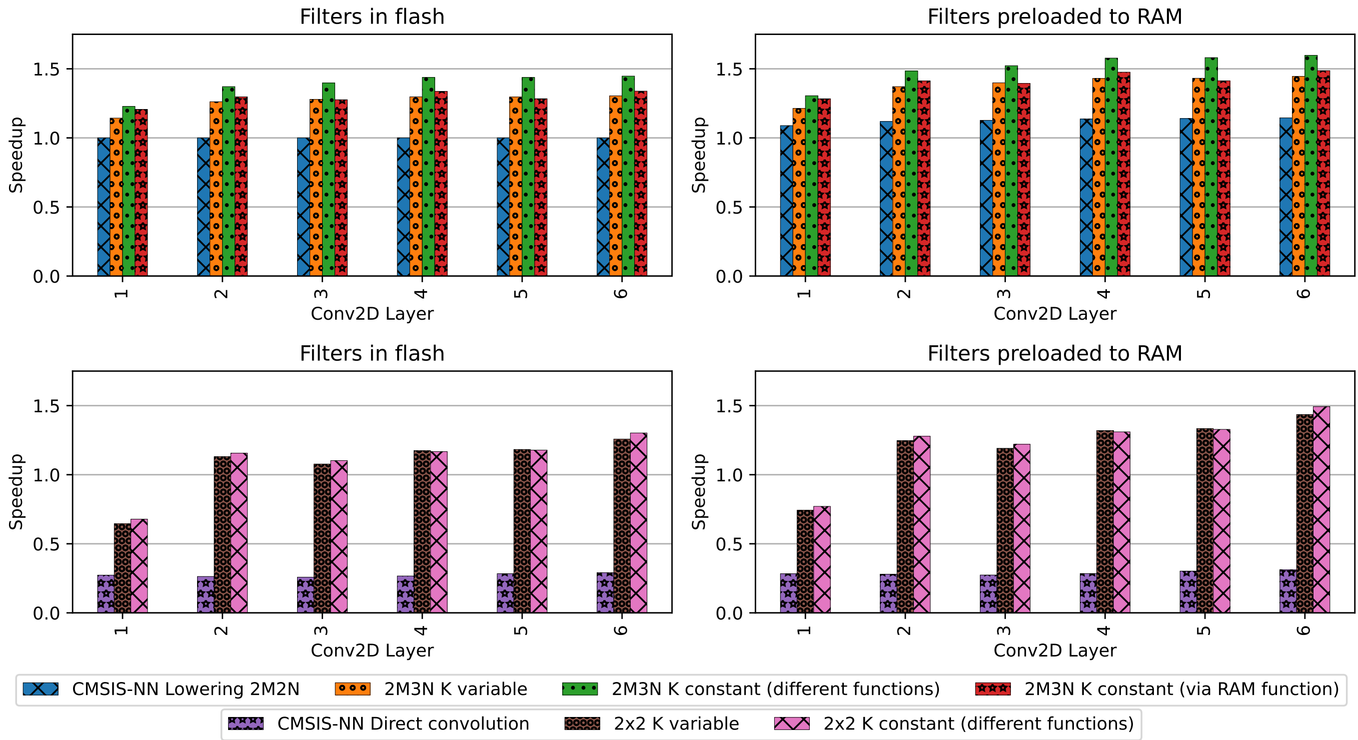


Fig. 8: Speedup of the lowering (top) and direct (bottom) convolution algorithms for the convolutional layers of the VGG-like CNN with respect to the baseline CMSIS-NN lowering convolution (with 2×2 microkernel) on the ARM Cortex-M4 processor from NANO and compiled with `-Os`. The plots in the left and right columns show the speedups achieved when the convolutional filters are read from flash or preloaded into RAM, respectively.

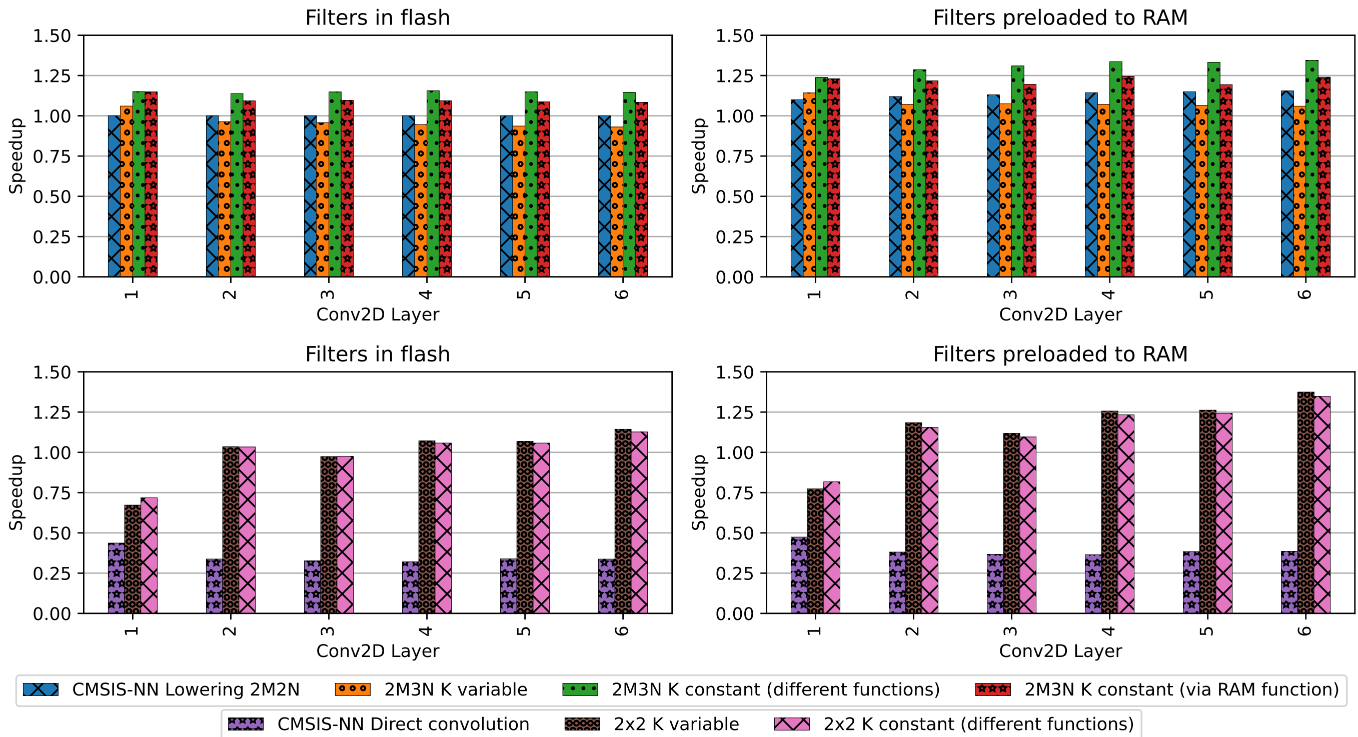


Fig. 9: Speedup of the lowering (top) and direct (bottom) convolution algorithms for the convolutional layers of the VGG-like CNN with respect to the baseline CMSIS-NN lowering convolution (with 2×2 microkernel) on the ARM Cortex-M4 processor from NANO and compiled with `-O3`. The plots in the left and right columns show the speedups achieved when the convolutional filters are read from flash or preloaded into RAM, respectively.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

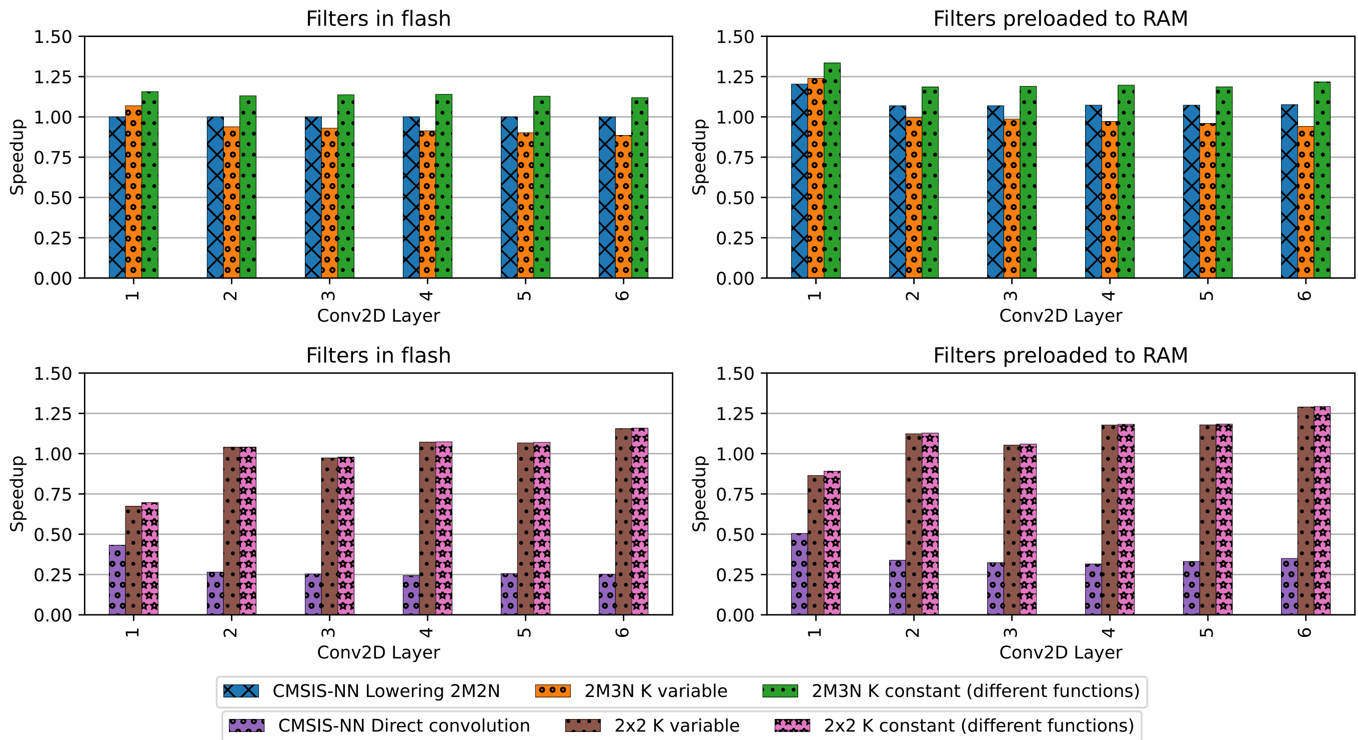


Fig. 10: Speedup of the lowering (top) and direct (bottom) convolution algorithms for the convolutional layers of the VGG-like CNN with respect to the baseline CMSIS-NN lowering convolution (with 2×2 microkernel) on the ARM Cortex-M4 processor from PORT and compiled with `-O3`. The plots in the left and right columns show the speedups achieved when the convolutional filters are read from flash or preloaded into RAM, respectively.

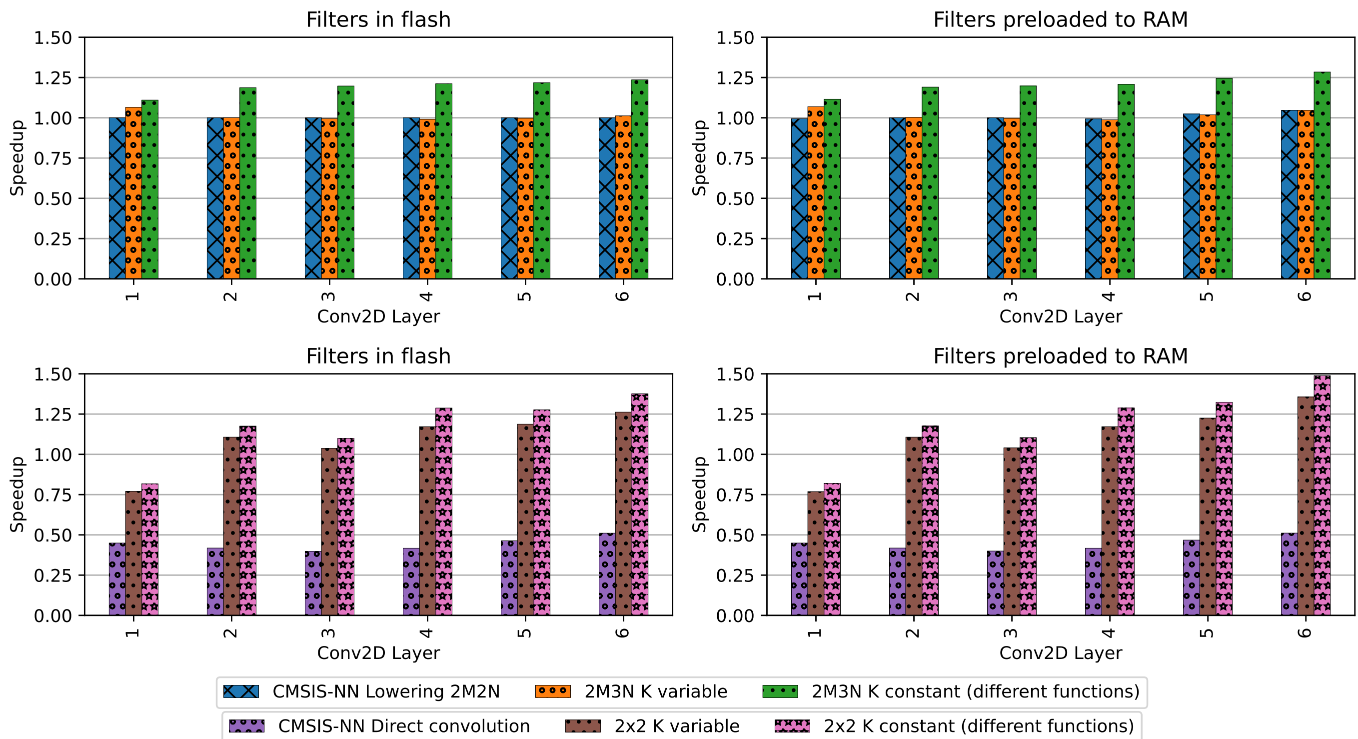


Fig. 11: Speedup of the lowering (top) and direct (bottom) convolution algorithms for the convolutional layers of the VGG-like CNN with respect to the baseline CMSIS-NN lowering convolution (with 2×2 microkernel) on the ARM Cortex-M7 processor from PORT and compiled with `-O3`. The plots in the left and right columns show the speedups achieved when the convolutional filters are read from flash or preloaded into RAM, respectively.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

preloaded to RAM (bottom-right plot) follow a very similar trend as in the previous case, except that they are boosted by approximately 10%. In general, these versions achieve speedups ranging between 1.15× and 1.37× compared to the original CMSIS-NN algorithm.

2) Results on the PORT *MCU:* Figures 10 and 11 show the speedups achieved on the ARM Cortex-M4 and Cortex-M7 processors in PORT, respectively, compiled with the `-O3` optimisation flag. It is important to note that the convolution algorithms utilising run-in-RAM functions were not available due to the PORT MCU not being able to use RAM for code storage.

Results on the ARM Cortex-M4 processor: When employing the lowering algorithms with filters read from flash memory (top-left plot in Figure 10), utilising k-variable results in an average performance loss of 0.94×. As mentioned previously, this is due to the use of compiler optimisations, which amplify the losses due to register spilling. On the other hand, using k-constant via different functions, this drawback is avoided, resulting in speedups ranging from 1.12× to 1.16×.

Regarding the lowering algorithms with filters preloaded into RAM (top-right plot), we observe speedups up to 1.22× for the k-constant via different functions case, while the k-variable algorithm does not show any improvement, except for the first layer.

As for the direct convolution algorithms with filters read from flash memory (bottom-left plot), both versions of the 2×2 microkernel behave very similarly, regardless of whether k is variable or constant via different functions. Unlike the k-variable lowering algorithm, the k-variable direct algorithm does not incur register spilling. These algorithms achieve speedups of up to 1.16× compared to the baseline CMSIS-NN lowering convolution.

Finally, for the direct convolution algorithms with the filters preloaded into RAM, the speedups range from 1.05× to 1.29× for all the convolution layers except the first one.

Results on the ARM Cortex-M7 processor: Figure 11 repeats the experiments in Figure 10, except that in this case, they are conducted on the ARM Cortex-M7 processor in PORT.

Regarding the lowering convolution algorithm with the filters read from flash (top-left plot), we observe that the variant using k-variable with the 2×3 microkernel reports the same performance as the baseline CMSIS-NN with the 2×2 microkernel. Although a larger microkernel should provide a better performance, its improvement is cancelled by the effects of register spilling. On the other hand, the k-constant via different functions variant reports speedups ranging between 1.11× and 1.24×. If, in turn, the filters are preloaded into RAM (top-right plot), we obtain a slight performance boost compared to the previous scenario for the k-constant variant. In this case, the speedup reaches 1.28× for the best scenario.

The results for the direct convolution algorithms with filters read from flash (bottom-left plot) are notable for all the convolutional layers, except for the first one. The speedups for all layers except the first range from 1.04× to 1.38×, leading to better results than the equivalent lowering algorithms.

Finally, for the direct convolution algorithms with the filters preloaded into RAM (bottom-right plot), we again obtain visible performance boosts for almost all layers with speedups ranging from 1.04× to 1.49× (for all the convolutional layers except the first one). Again, it can be observed that, for this processor, on some of the convolutional layers, the direct convolution variants outperform the lowering equivalents.

From these results, we can draw the general conclusion that, depending on the convolution layer parameters, a different lowering or direct convolution algorithm could be selected to obtain the best overall performance.

VI. CONCLUDING REMARKS

In this paper, we have presented several optimisations for the convolution operator on the ARM Cortex-M4 and Cortex-M7 processors found in the Arduino Nano 33 BLE Sense and Arduino Portenta H7 Lite MCUs. These optimisations involve the design of new highly-tuned and vectorised microkernels and the use of RAM as a scratchpad for storing the convolution filters.

The experimental results using our optimised convolution algorithms show substantial improvements in the inference time of the four evaluated CNN models on the three different Cortex-M processors available in the selected MCUs. Specifically, for VGG, we attained speedups in the range of 1.13× to 1.33×; for SqueezeNet, 1.26× to 1.50×; for ResNet, 1.13× to 1.19×; and for MobileNet, 1.15× to 1.27×. After a detailed analysis of the convolution algorithms at the layer level, we found that the lowering algorithm using k-constant via different functions and filters preloaded into RAM achieves the shortest execution times, resulting in speedups when compiled optimised for speed and the VGG-like CNN of up to 1.28× on the NANO Cortex-M4 processor, up to 1.13× on the PORT Cortex-M4 processor, and up to 1.33× on the PORT Cortex-M7 processor. Similarly, the direct convolution algorithm with k-constant via different functions and filters preloaded into RAM achieves speedups of up to 1.35× on the NANO Cortex-M4, up to 1.29× on the PORT Cortex-M4 processors, and up to 1.49× on the PORT Cortex-M7 processors, but only for those convolutional layers of the VGG-like model that exhibit a large number of filters on small input tensors.

Also, we observed that for the PORT Cortex-M7 processor, the convolution variants reading filters from flash or preloading filters into RAM show almost identical results due to the similar latencies for accessing flash and RAM on this MCU.

As part of future work, we plan to investigate different operating frequencies and Cortex-M8 processors with vector instructions (MVE). Additionally, we aim to dynamically select the best-performing algorithm based on the convolution layer parameters. Finally, we aim to explore the use of microTVM to automatically generate alternative microkernels for convolutions and GEMM operations involved in DNN layers.

ACKNOWLEDGMENTS

This research was funded by project TED2021-129334B-I00 supported by MCIN/AEI/10.13039/501100011033 and by the "European Union NextGenerationEU/PRTR". Manuel

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

F. Dolz was also supported by the Plan Gen–T grant CIDEXG/2022/13 of the *Generalitat Valenciana*. Antonio Maciá-Lillo is a PRE2021-099284 fellow supported by MCIN/AEI/10.13039/501100011033.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Comm. ACM*, vol. 60, no. 6, pp. 84–90, 2012.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

[3] L. Trinh Van, T. Dao Thi Le, T. Le Xuan, and E. Castelli, "Emotional speech recognition using deep neural networks," *Sensors*, vol. 22, no. 4, p. 1414, 2022.

[4] S. Branco, A. G. Ferreira, and J. Cabral, "Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: A survey," *Electronics*, vol. 8, no. 11, p. 1289, 2019.

[5] 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," *arXiv preprint arXiv:1704.04861*, 2017.

[6] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and less than 0.5 MB model size," *arXiv:1602.07360*, 2016.

[7] Arm Limited, "ARM Cortex-M processor series," 2021, https://developer.arm.com/ip-products/processors/cortex-m.

[8] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for Arm Cortex-M CPUs," 2018. [Online]. Available: https://arxiv.org/abs/1801.06601

[9] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *International Workshop on Frontiers in Handwriting Recognition*, 2006.

[10] S. Barrachina, M. F. Dolz, P. San Juan, and E. S. Quintana-Ortí, "Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors," *J. Parallel Distrib. Comput.*, vol. 167, no. C, p. 240–254, sep 2022.

[11] S. Barrachina, A. Castelló, M. F. Dolz, T. M. Low, H. Martínez, E. S. Quintana-Ortí, U. Sridhar, and A. E. Tomás, "Reformulating the direct convolution for high-performance deep learning inference on ARM processors," *Journal of Systems Architecture*, vol. 135, p. 102806, 2023.

[12] ARM, "ARM Cortex-M4 Processor Technical Reference Manual Revision r0p1," ARM Holdings, plc., accessed on May 21, 2023. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4

[13] ARM,' 'ARM Cortex-M7 Processor Technical Reference Manual Revision r1p2," ARM Holdings, plc., accessed on May 21, 2023. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m7

[14] A. Castelló, E. S. Quintana-Ortí, and F. D. Igual, "Anatomy of the BLIS family of algorithms for matrix multiplication," in *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 2022, pp. 92–99.

[15] G. Alaejos, A. Castelló, H. Martínez, P. Alonso-Jordá, F. D. Igual, and E. S. Quintana-Ortí, "Micro-kernels for portable and efficient matrix multiplication in deep learning," *The Journal of Supercomputing*, vol. 79, no. 7, pp. 8124–8147, 2023.

[16] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 31–46.

[17] B. Nguyen, P.-A. Moellic, and S. Blayac, "Evaluation of convolution primitives for embedded neural networks on 32-bit microcontrollers," 2023.

[18] M. Deutel, P. Woller, C. Mutschler, and J. Teich, "Deployment of energy-efficient deep learning models on Cortex-M based microcontrollers using deep compression," 2022.

[19] L. Grzymkowski and T. P. Stefański, "Performance analysis of convolutional neural networks on embedded systems," in *2020 27th International Conference on Mixed Design of Integrated Circuits and Systems (MIXDES)*, 2020, pp. 266–271.

[20] G. Cerutti, R. Prasad, A. Brutti, and E. Farella, "Compact recurrent neural networks for acoustic event detection on low-energy low-complexity platforms," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 4, pp. 654–664, 2020.

[21] A. Faraone and R. Delgado-Gonzalo, "Convolutional-recurrent neural networks on low-power wearable platforms for cardiac arrhythmia detection," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, aug 2020. [Online]. Available: https://doi.org/10.1109%2Faicas48895.2020.9073950

[22] S. Sadiq, J. Hare, S. Craske, P. Maji, and G. Merrett, "Enabling ImageNet-scale deep learning on MCUs for accurate and efficient inference," *IEEE Internet of Things Journal*, pp. 1–1, 2023.

[23] E. Liberis and N. D. Lane, "Neural networks on microcontrollers: saving memory at inference via operator reordering," *arXiv preprint arXiv:1910.05110*, 2019.

[24] L. Lai and N. Suda, "Enabling deep learning at the lot edge," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–6.

[25] M. Cho and D. Brand, "MEC: memory-efficient convolution for deep neural network," in *International Conference on Machine Learning*. PMLR, 2017, pp. 815–824.

[26] P. Wang, X. Wang, R. Luo, D. Wang, M. Luo, S. Qiao, and Y. Zhou, "An efficient im2row-based fast convolution algorithm for ARM Cortex-M MCUs," *IEEE Access*, vol. 9, pp. 124 384–124 395, 2021.

[27] "Arduino Nano 33 BLE Sense," https://store.arduino.cc/arduino-nano-33-ble-sense, accessed on May 21, 2023.

[28] Nordic Semiconductor, "nRF52840 product specification," Datasheet, accessed on May 21, 2023. [Online]. Available: https://www.nordicsemi.com/Products/nRF52840

[29] STMicroelectronics, "STM32H747XI datasheet," Datasheet, accessed on May 21, 2023. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32h747xi.html

[30] "Arduino Portenta H7 Lite," https://www.arduino.cc/pro/hardware/product/portenta-h7-lite, accessed on May 21, 2023.

[31] M. Chhillar, G. Yadav, and N. K. Shukla, "Comparing the two different generations of amba based protocols: AHB vs AXI," *International Journal of Science and Research (IJSR)*, vol. 3, no. 5, pp. 488–490, May 2014. [Online]. Available: https://www.ijsr.net/archive/v3i5/MDIwMTMxODA2.pdf

[32] D. Shankar and M. Design, "Comparing AMBA AHB to AXI bus using system modeling," *Design & Reuse*, 2023. [Online]. Available: https://www.design-reuse.com/articles/article24123.html

[33] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[36] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes, "Visual wake words dataset," *arXiv preprint arXiv:1906.05721*, 2019.

APPENDIX A
MODELS' SPECIFICATIONS

In this appendix, we provide detailed layer specifications of the DNNs used in this work: 1) VGG-like [34] (see Table A1); 2) SqueezeNet-like [6] (see Table A2); 3) ResNet-v1-like [35] (see Table A3)); and 4) MobileNet-v1-like [5] (see Table A4). The layer specifications provided in the tables encapsulate critical information, such as the layer ID, type of layer and activation functions, number of neurons and filter size, strides, and the output dimensions of each layer. In Table A5 we also display the transformations of the SqueezeNet-like DNN after being exported to TensorFlow Lite using full 8-bit integer quantisation.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Internet of Things Journal. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2024.3395335

TABLE A1: Specification of the custom VGG-like model used for the experimentation.

Id.	Layer type	Output shape
0	Input	$32 \times 32 \times 3$
1–6	Conv $(3 \times 3 \times 16)$ – ReLU – BN $\} \times 2$	$32 \times 32 \times 16$
7	MaxPool (2×2) (stride 2×2)	$16 \times 16 \times 16$
8–13	Conv $(3 \times 3 \times 32)$ – ReLU – BN $\} \times 2$	$16 \times 16 \times 32$
14	MaxPool (2×2) (stride 2×2)	$8 \times 8 \times 32$
15–20	Conv $(3 \times 3 \times 64)$ – ReLU – BN $\} \times 2$	$8 \times 8 \times 64$
21	MaxPool (2×2) (stride 2×2)	$4 \times 4 \times 64$
22–23	FC – ReLU	64
24–25	FC – SoftMax	10

TABLE A2: Specification of the custom SqueezeNet-like model used for the experimentation.

Id.	Layer type	Output shape
0	Input	$32 \times 32 \times 3$
1–2	Conv $(3 \times 3 \times 16)$ (stride 2×2) – ReLU	$15 \times 15 \times 64$
3	MaxPool (3×3) (stride 2×2)	$7 \times 7 \times 64$
4–17	Conv_11 $(1 \times 1 \times 16)$ – ReLU Conv_12 $(1 \times 1 \times 64)$ – ReLU Conv_13 $(3 \times 3 \times 64)$ – ReLU Concat(Conv_12, Conv_13) $\} \times 2$	$7 \times 7 \times 16$ $7 \times 7 \times 64$ $7 \times 7 \times 64$ $7 \times 7 \times 128$
18	MaxPool (3×3) (stride 2×2)	$3 \times 3 \times 64$
19–32	Conv_21 $(1 \times 1 \times 32)$ – ReLU Conv_22 $(1 \times 1 \times 128)$ – ReLU Conv_23 $(3 \times 3 \times 128)$ – ReLU Concat(Conv_22, Conv_23) $\} \times 2$	$3 \times 3 \times 32$ $3 \times 3 \times 128$ $3 \times 3 \times 128$ $3 \times 3 \times 256$
33	MaxPool (3×3) (stride 2×2)	$1 \times 1 \times 256$
34–35	Conv $(1 \times 1 \times 10)$ – ReLU	$1 \times 1 \times 10$
36–37	GlobalAvgPool – SoftMax	10

TABLE A3: Specification of the custom ResNet-v1-like model used for the experimentation.

Id.	Layer type	Output shape
0	Input	$32 \times 32 \times 3$
1–3	Conv_1 $(3 \times 3 \times 16)$ – BN – ReLU	$32 \times 32 \times 16$
4–8	Conv $(3 \times 3 \times 16)$ – BN – ReLU Conv_2 $(3 \times 3 \times 16)$ – BN	$32 \times 32 \times 16$
8–9	Add(Conv_1, Conv_2) – ReLU	$32 \times 32 \times 16$
10–14	Conv $(3 \times 3 \times 32)$ (str. 2×2) – BN – ReLU Conv_3 $(3 \times 3 \times 32)$ – BN	$16 \times 16 \times 32$
15	Conv_4 $(3 \times 3 \times 16)$ (str. 2×2)	$16 \times 16 \times 32$
16–17	Add(Conv_3, Conv_4) – ReLU	$16 \times 16 \times 32$
18–22	Conv $(3 \times 3 \times 32)$ (str. 2×2) – BN – ReLU Conv_5 $(3 \times 3 \times 32)$ – BN	$8 \times 8 \times 64$
23	Conv_6 $(3 \times 3 \times 16)$ (str. 2×2)	$8 \times 8 \times 64$
24–25	Add(Conv_5, Conv_6) – ReLU	$8 \times 8 \times 64$
26–30	Conv $(3 \times 3 \times 32)$ (str. 2×2) – BN – ReLU Conv_7 $(3 \times 3 \times 32)$ – BN	$4 \times 4 \times 128$
31	Conv_8 $(3 \times 3 \times 16)$ (str. 2×2)	$4 \times 4 \times 128$
32–33	Add(Conv_7, Conv_8) – ReLU	$4 \times 4 \times 128$
34	AvgPool (4×4)	$1 \times 1 \times 128$
35	FC	10

TABLE A4: Specification of the custom MobileNet-v1-like model used for the experimentation.

Id.	Layer type	Output shape
0	Input	$96 \times 96 \times 3$
1–3	Conv $(3 \times 3 \times 8)$ (str. 2×2) –BN – ReLU	$48 \times 48 \times 8$
4–6	Conv $(3 \times 3 \times 8)$ –BN – ReLU	$48 \times 48 \times 8$
7–9	Conv $(1 \times 1 \times 16)$ –BN – ReLU	$48 \times 48 \times 16$
10–12	Conv $(3 \times 3 \times 16)$ (str. 2×2) –BN – ReLU	$24 \times 24 \times 16$
13–15	Conv $(1 \times 1 \times 32)$ –BN – ReLU	$24 \times 24 \times 32$
16–18	Conv $(3 \times 3 \times 32)$ –BN – ReLU	$24 \times 24 \times 32$
17–21	Conv $(1 \times 1 \times 32)$ –BN – ReLU	$24 \times 24 \times 32$
22–24	Conv $(3 \times 3 \times 32)$ (str. 2×2) –BN – ReLU	$12 \times 12 \times 32$
25–27	Conv $(1 \times 1 \times 64)$ –BN – ReLU	$12 \times 12 \times 64$
28–30	Conv $(3 \times 3 \times 64)$ –BN – ReLU	$12 \times 12 \times 64$
31–33	Conv $(1 \times 1 \times 64)$ –BN – ReLU	$12 \times 12 \times 64$
34	AvgPool (12×12)	$1 \times 1 \times 64$
35	FC	2

TABLE A5: VGG-like model generated by TFLite once quantised to 8-bit integer.

Id.	Layer type	Output shape
0	Input	$32 \times 32 \times 3$
1–8	Conv $(3 \times 3 \times 16)$ – ReLU Mul – Add $\} \times 2$	$32 \times 32 \times 16$
9	MaxPool (2×2) (stride 2×2)	$16 \times 16 \times 16$
10–18	Conv $(3 \times 3 \times 32)$ – ReLU Mul – Add $\} \times 2$	$16 \times 16 \times 32$
19	MaxPool (2×2) (stride 2×2)	$8 \times 8 \times 32$
20–28	Conv $(3 \times 3 \times 64)$ – ReLU Mul – Add $\} \times 2$	$8 \times 8 \times 64$
29	MaxPool (2×2) (stride 2×2)	$4 \times 4 \times 64$
30–31	FC – ReLU	64
32–33	FC – SoftMax	10

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/