PULP-TrainLib: Enabling On-Device Training for RISC-V Multi-core MCUs Through Performance-Driven Autotuning

PULP-TrainLib: Enabling On-Device Training for RISC-V Multi-Core MCUs through Performance-Driven Autotuning

Davide Nadalini[1,2][0000−0002−8003−7633], Manuele Rusci[2][0000−0001−7458−4019], Giuseppe Tagliavini[3][0000−0002−9221−4633], Leonardo Ravaglia[2], Luca Benini[2,4][0000−0001−8068−3806], and Francesco Conti[2][0000−0002−7924−933X]

[1] DAUIN, Politecnico di Torino, 10129 Torino, Italy, davide.nadalini@polito.it
[2] DEI, University of Bologna, 40136 Bologna, Italy, {d.nadalini,manuele.rusci,luca.benini,f.conti}@unibo.it
[3] DISI, University of Bologna, 40134 Bologna, Italy, giuseppe.tagliavini@unibo.it
[4] IIS, ETH Zurich, 8092 Zurich, Switzerland, lbenini@iis.ee.ethz.ch

Abstract. An open challenge in making Internet-of-Things sensor nodes "smart" and self-adaptive is to enable on-chip Deep Neural Network (DNN) training on Ultra-Low-Power (ULP) microcontroller units (MCUs). To this aim, we present a framework, based on *PULP-TrainLib*, to deploy DNN training tasks on RISC-V-based Parallel-ULP (PULP) MCUs. *PULP-TrainLib* is a library of parallel software DNN primitives enabling the execution of forward and backward steps on PULP MCUs. To optimize *PULP-TrainLib*'s kernels, we propose a strategy to automatically select and configure (autotune) the fastest among a set of tiling options and optimized floating-point matrix multiplication kernels, according to the tensor shapes of every DNN layer. Results on an 8-core RISC-V MCU show that our auto-tuned primitives improve MAC/clk by up to 2.4× compared to "one-size-fits-all" matrix multiplication, achieving up to 4.39 MAC/clk - 36.6× better than a commercial STM32L4 MCU executing the same DNN layer training workload. Furthermore, our strategy proves to be 30.7× faster than AIfES, a state-of-the-art training library for MCUs, while training a complete TinyML model.

Keywords: Deep Learning · On-Device Training · Multi-Core MCUs · Autotuning

1 Introduction

Deep Neural Network (DNN) inference on *ultra-low-power* devices has been the target of an extensive body of research and development in recent years [28, 19, 9, 29]. This disruptive paradigm has provided a critical mass for strong innovation in the architecture, programming models, and development stacks available for MicroController (MCU) devices, targeting edge sensors and smart devices.

In this context, the emerging open challenge concerns how to enable *on-device learning* and model adaptation directly on this class of devices. The present

design flow of DNN models for deployment on edge devices is extremely static. DNN training is performed with a *train-once-deploy-everywhere* approach, using GPU-based servers featuring massive computation and memory resources. Only after training, the DNN model is deployed on edge platforms for inference-only execution. This static *train-once-deploy-everywhere* approach prevents adapting DNN inference models to changing conditions in the deployment environment, which may impact the statistical distribution of data and lead to significant in-field accuracy degradation and unreliable predictions [1].

Recently proposed methodologies based on Transfer Learning [21], Continual Learning [22], or Incremental Learning [16] aim at softening the risks of the *train-once-deploy-everywhere* design flow by taking a trained DNN model and enhancing or adapting it to data collected in-the-field. All of these methods perform DNN training tasks starting from the trained model and make use of backpropagation and stochastic gradient descent (SGD) algorithms. The deployment of these DNN training algorithms on MCU platforms results in a three-fold challenge. First, each layer of the DNN has to be computed both in forward and in backward propagation, differently from inference tasks, which require only the forward computation. Second, SGD demands floating-point operations, significantly more expensive than heavily quantized ones (8 bits or less) largely used for edge inference. Third, SGD often needs many iterations (100 or more) to achieve convergence instead of a single pass for inference - and contrarily to server training, it typically has to work on a single element, with little chance to employ batching to boost the training kernels' parallelism.

This paper addresses these challenges by introducing a framework for the efficient deployment of DNN training tasks on energy-efficient MCU devices. In particular, we rely on recent architectural advancements, which make multi-iteration backprop-based training more feasible on MCU-class devices [26, 2]. Among these, Parallel Ultra-Low-Power (PULP) devices [14] are multi-core MCUs exploiting architectural parallelism and near-threshold execution to enable high-performance and low-power processing at the edge. Our work introduces *PULP-TrainLib*, the first DNN training software library for RISC-V-based multi-core PULP devices. Our library consists of latency-optimized and tunable forward and backward training primitives of DNN layers, with multiple alternative configurations available for each training primitive tuned to better suit a given tensor shape in the forward and backward passes. Furthermore, to empower our framework for on-device DNN-training and optimize performance, we couple *PULP-Trainlib* with *AutoTuner*, an automated tool to select the fastest configuration and tile partitioning for every DNN layer.

In detail, our work includes the following novel contributions:

- *PULP-TrainLib*, the first open-source training library for RISC-V-based multi-core MCUs, including a set of performance-tunable DNN layer primitives to enable DNN training on ultra-low-power devices;
- *AutoTuner*, an automated tool exploiting an HW-in-the-loop strategy to opportunistically select the layer primitive that leads to the lowest latency, given the DNN structure and the memory constraints of the target device;

– A detailed analysis of *PULP-TrainLib* and the AutoTuner targeting an 8-Core RISC-V-based PULP platform, and a comparison with state-of-the-art results achievable on commercial STM32 MCUs.

Our approach demonstrates that training techniques such as Incremental and Continual Learning are achievable with realistic latencies ($< 1ms$ to train a tinyMLPerf's AutoEncoder[5] on a new input) on a device consuming an MCU-class active power budget (<100 mW), without sacrificing full programmability. The forward and backward primitives of *PULP-TrainLib* show an almost linear parallel speed-up on a multi-core RISC-V platform. Benchmarking over individual PointWise, DepthWise Convolution, and Linear layers, the AutoTuner optimizes the throughput by up to, respectively, 18.3×, 8×, and 11× compared to a naïve implementation. This descends from the selection of the fastest matrix multiplication kernel for each set of tensor shapes, with a top performance of 4.39 Multiply-Accumulate operations per clock cycle (MAC/clk) to execute a single training step. Our optimized solution outperforms the execution on an STM32L476RG commercial MCU by 36.6×. Additionally, we benchmark our methodology on two TinyMLPerf benchmarks, a DS-CNN for Keyword Spotting (KWS) and a Deep Autoencoder for Anomaly Detection. Results on an 8-core PULP Platform show that the latency of a complete DNN training step is reduced to a minimum of 0.39 million clock cycles on the Deep Autoencoder (i.e., < 1 ms @400 MHz), applying the AutoTuner, 30.7× faster than AIfES, the current state-of-the-art training library for MCUs. To foster future research on extreme-edge DNN training, we release PULP-Trainlib as open-source software[6].

2 Related Work

While edge DNN inference has been widely explored, on-device training is still an open problem due to the strict constraints of embedded platforms. To accelerate backprop-based training, Federated Learning [11, 18, 17] distributes the training process between several agents. TinyTL [5] proposes a lightweight (transfer) learning technique by limiting back-propagation to biases only, which also prevents the storage of the intermediate tensors required for weight gradient back-propagation. Compared with these works, we focus on accelerating the computational primitives of SGD, which can be adopted in the future to bring these approaches to low-power MCUs.

A limited set of works have already addressed the problem of model adaptation on MCUs. In TinyOL [25], the authors propose to use an Arduino Nano equipped with a 64MHz ARM Cortex-M4 for Transfer Learning, using a single trainable layer on top of a frozen and quantized inference model. Targeting the same platform, TinyFedTL [12] extends the concepts of TinyOL with a Federated approach, aiming at retraining the last layer in a distributed way. Ravaglia et al. [24] target for the first time Continual Learning [22] on MCUs.

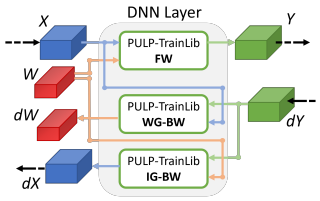
[5] TinyML Perf Models: https://github.com/mlcommons/tiny/tree/master/benchmark
[6] PULP-TrainLib: https://github.com/pulp-platform/pulp-trainlib

The proposed HW/SW platform learns the model coefficients of multiple DNN layers using backpropagation. Our work generalizes this approach into a training framework that can be used beyond Continual Learning use-cases only, e.g. in an Incremental or Transfer Learning setting. Moreover, we improve the performances of the key computational steps by up to 2.3× thanks to novel optimized SW kernels and the Autotuning functionality. Close to our work, AIfES[7] is the only other publicly-released training software framework for MCUs. More in detail, such a framework currently enables the training of DNN models composed of exclusively Fully-Connected (aka Dense or Linear) layers. The software primitives are optimized on ARM Cortex-M MCUs using kernels of the CMSIS-DSP library, in particular floating-point matrix multiplications. In contrast, PULP-TrainLib includes a wider set of DNN layers (like 2D and Depthwise Separable Convolutions) and features a higher optimization level by applying autotuned loop unrolling and parallelization, requiring 30.7× fewer clock cycles than AIfES to execute a complete DNN model.

Our work combines PULP-TrainLib's software with an autotuning tool, searching for the software configuration (MM kernel and tiling) that optimizes a cost function, i.e. minimal latency. Autotuning is a widespread approach in high-performance, parallel computing (readers may refer to [3, 20]). Representative examples are FFTW [10] and ATLAS [30], which aim at optimizing FFT and Basic Linear Algebra Subroutines (BLAS), respectively, using a code-generator autotuner. They achieve a speed-up of up to 55% and 20%, respectively, compared to non-optimized algorithms. Halide [23] integrates an autotuner on top of a stencil pipeline, also leveraging stochastic search to speed up the execution to up 5×. Tensor Comprehensions [27] extends this concept for DNN inference, providing a Just-In-Time compiler that exploits operator fusion, delegated memory management, and an autotuner to find the fastest implementation among a vast search space. To bypass the complexity of hardware-in-the-loop autotuning, the TVM [6] compiler for DNN inference replaces an autotuner with a learnable cost model to predict the fastest solution starting from a trial-based approach. The same authors present an alternative method [7] that learns domain-specific cost models by exploring the solution space of the target task.

To the best of our knowledge, our work is the first one to target autotuned hardware-in-the-loop optimization of DNN training on an MCU platform. We adopt a trial-based approach like FFTW and ATLAS [10, 30]. Compared with Ragan-Kelley et al. [23], which adopts stochastic search, we prefer an exhaustive hardware-in-the-loop approach, capable of finding the best optimization for the problem by direct performance profiling. MCU-oriented autotuning tools, like uTVM[8] and the work of Chen et al. [8], are currently limited to inference – and currently do not achieve a speed comparable to hand-tuned libraries, like CMSIS-NN [13] and TinyEngine [15].

[7] AIfES for Arduino: https://www.arduino.cc/reference/en/libraries/aifes-for-arduino/
[8] Apache μTVM: https://tvm.apache.org/2020/06/04/tinyml-how-tvm-is-taming-tiny




```
function train_linear(X, W, Y, label){
   pulp_linear_fp32_fw_cl(X, W, Y);
   dY = loss(Y, label);
   pulp_linear_fp32_bw_input_grads_cl(dY, W, dX);
   pulp_linear_fp32_bw_param_grads_cl(dY, X, dW);
   W = update (W, dW);
}
```

Fig. 1: SW primitives of *PULP-TrainLib* library (left) and corresponding pseudo-code showing how to train an individual linear layer (right). On the left, elements in blue show the path of the input tensor's data and gradient, while the red and green ones represent the weight and output, respectively.

3 On-Device Training on PULP

3.1 The PULP platform

PULP (parallel ultra-low power) is a computational platform for energy-efficient and scalable edge computing based on RISC-V cores [26]. In this work, we consider a PULP SoC instance that includes an MCU for control-related tasks and a multi-core cluster for parallel computation. The MCU features a single RISC-V core, a set of IO peripherals, and a 2 MB SRAM memory (L2) accessible by the cluster side through a DMA engine. The cluster includes 8 RISC-V cores sharing a 64 kB L1 memory, accessible in a single cycle. Each core implements a basic set of standard RISC-V extensions (RV32IMFC); additionally, a custom extension (Xpulp) provides DSP-like features to reduce overhead in highly uniform workloads, including post-increment load/store operations and 2-level nested hardware loops. Finally, each CPU is granted access to a private Floating Point Unit (FPU), capable of performing complex, single-cycle DSP instructions, like Fused Multiply-Add (FMA).

3.2 PULP-TrainLib Library

PULP-TrainLib is the first software library for DNN training on MCU-class RISC-V multi-core platforms. More in detail, we provide an optimized software design for the forward and backward propagation passes of multiple DNN layers, targeting the execution on the PULP Platform. Our library adopts floating-point data formats (fp32 and lower) for all the computations of the training steps.

Figure 1 (left) graphically represents the computational flow of the Forward and Backward training steps of a single DNN layer trained with the backpropagation algorithm. The *Forward* (FW) step receives the input tensor X and produces the output tensor Y based on the trainable weights W. This output propagates through the model's layers to compute the loss score with respect to the data label. Unlike inference, the intermediate tensors computed during the FW pass must be kept in memory for the *Backward* (BW) step. For every layer, the weight gradient dW is computed by the *Weight Gradient Backward* (WG-BW)

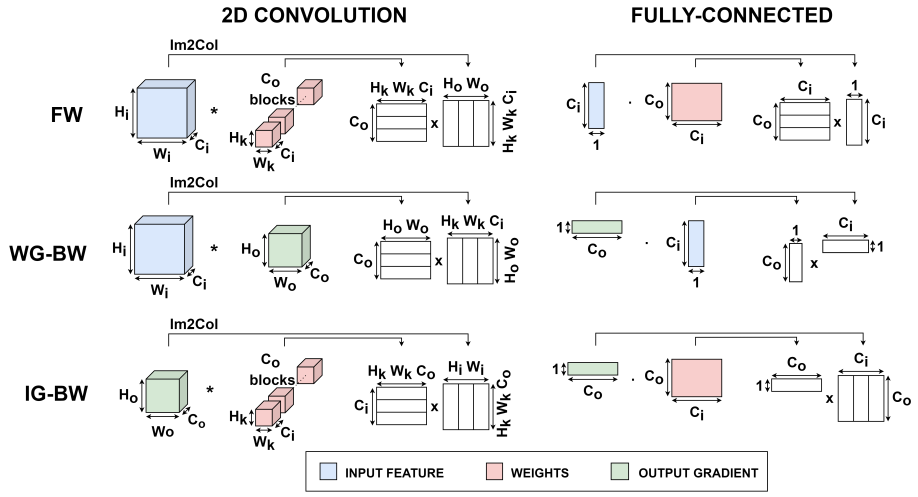


Fig. 2: DNN training matrix operations of FW and BW steps of a 2D Convolution (left) and a Fully-Connected layer (right). The tensor (and respective gradient) shapes involved in 2D Convolution have size $C_i \cdot H_i \cdot W_i$ for the input, $C_o \cdot C_i \cdot H_k \cdot W_k$ for the weights, $C_o \cdot H_o \cdot W_o$ for the outputs. In the case of a Fully-Connected, instead, their shape is C_i, $C_o \cdot C_i$, C_o, respectively. The Im2Col operator transforms the input feature map (FW, WG-BW) or the output gradient (IG-BW) to matrix form to exploit fast matrix computations.

function based on the stored activation feature map X and the gradient tensor dY, which is the derivative of the loss function with respect to the activation tensor. Such gradient signal then propagates to the previous layer by applying the *Input Gradient Backward* (IG-BW) step. This latter returns dX based on the transposed weight tensor W'. *PULP-TrainLib* provides a set of SW kernels that implement these training steps. For instance, Figure 1 (right) illustrates the pseudo-code for training the coefficients of an individual fully-connected layer using *PULP-TrainLib*'s primitives.

Figure 2 shows the implementation details of the training steps in the case of a 2D Convolution and a Fully-Connected Layer. To leverage computationally efficient linear algebra kernels, the convolution operations are reshaped as Matrix Multiplications (MMs). To this aim, the data layout of the input X (or the output gradient dY) of a Conv2D layer is reshaped at runtime from a CHW tensor shape to a matrix form through an *im2col* transformation. This routine copies the input data within the receptive field of the filter into column contiguous arrays, stored into the L1 memory, to feed the MM kernel.

3.3 Accelerating SW Training Primitives

The workload of PULP-TrainLib primitives is dominated by MMs. The pseudo-assembly on a RISC-V core of a MM naïve kernel is reported in Figure 3 (a).

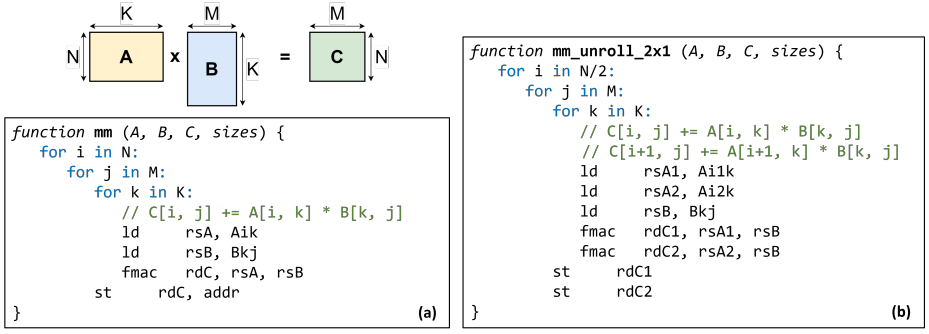


Fig. 3: Pseudo-assembly of the inner loops of (a) a naive MM and (b) of an optimized *2×1* unrolled MM. The matrix dimensions are highlighted on the top-left of the figure.

This baseline implementation makes use of three nested loops, which iterate over the matrix dimensions. The inner loop features 3 instructions: the 2 loads of the MM's operands and the floating-point FMA. Thus, in this case, the total number of instructions is $N \times M \times K \times 3$, where $N \times M$ is the size of the output matrix $C = A \cdot B$ and K is the size of the shared dimension.

The total number of instructions can be reduced by means of *loop unrolling*. This technique leads to a faster implementation by computing multiple outputs within the inner loop, hence exploiting data reuse. As a convention, we define the *unrolling factor* as $U \times V$, where U and V are, respectively, the number of rows and columns of the output matrix concurrently computed within the inner loop of the MM. In particular, Figure 3 (b) represents a *2×1* loop unrolling, which features a 33% higher FMA/Load ratio with respect to the naïve baseline, therefore reducing the overall number of instructions.

Typically, the instruction count decreases with increasing unrolling factor $U \times V$. However, because of the limited size of the register file of the CPU, a spilling effect is observed with a *4×4* or larger unrolling. When this happens, the compiler generates code to spill accumulation registers that do not fit within the CPU's register file to the stack, resulting in a very significant reduction of efficiency and increased latency. Additionally, a slowdown effect can be observed if a dimension of the input matrices is not an integer multiple of the unrolling factor $U \times V$. The computation of the remaining elements that do not fit the MM's inner loop, which we call *leftover*, is handled by different sub-routines that employ less aggressive unrolling and incur in larger latency. In this case, a smaller unrolling factor can result in a faster solution. If the input tensors are smaller than the $U \times V$ unrolling size, the MM kernel turns into all-leftover computation, which greatly slows down the execution.

To address this performance issue, we provide multiple optimized MM functions (listed in Table 1), which can be used within the training primitives of the *PULP-TrainLib* to speed up the execution. In addition to unrolling, every MM

Table 1: PULP-TrainLib's Optimized MM Kernels

MM Type	Unrolling Factor	Parallelism	Description
mm	–	N or M	Naive MM
mm_uJ	$J = 2$	N or M	Unroll J inner products in K
mm_unroll_1xV	$U = 2, 4, 8$	N or M	Unroll U columns of C
mm_unroll_Ux1	$V = 2, 4, 8$	N or M	Unroll V rows of C
mm_unroll_UxV	$U, V = 2, 4$	N or M	Unroll U,V rows and columns of C

kernel exploits data parallelism to efficiently run on the multi-core PULP cluster. Two parallelization strategies are proposed: each core can compute different output rows (`mm_unroll_UxV`) by splitting the workload over the N dimension, or different output columns (`mm_M_unroll_UxV`), by chunking the M size. Simple unrolling over the K dimension is also provided (`mm_uJ`) to cover a set of corner cases in which the reduced sizes of N or M prevent the use of unrolling on both sizes (e.g., $N = 8$, $M = 1$, $K >> N$, with parallelization on N).

3.4 AutoTuner

As part of *PULP-TrainLib*, we propose an autotuning flow to optimally configure the DNN training primitives (e.g., which MM to use) based on *(i)* the DNN layer type and training step (e.g., PointWise FW) and *(ii)* the PULP platform settings (e.g., the number of cluster's cores and the L1 memory size). The AutoTuner is run offline, before the deployment to the MCU and the actual training - the best (fastest) software setting is identified, then deployed on the target device.

Because of the limited memory footprint, the DNN layer's input and output tensors may not entirely fit the on-chip memories, and in particular the L1 memory of the cluster. Therefore, a *tiling* process is required to partition the DNN layer's tensors into smaller sub-tensors, also referred to as *Tensor Tiles*. The tiles are transferred from L2 and L1 and processed in sequence. For each tensor tile requiring an *im2col*, this transformation is performed in L1, minimizing the impact on memory. Because the Tensor operand shapes highly impact the MM optimization, the AutoTuner aims at finding concurrently the optimal Tile shapes and the MM kernel, indicated as $TileShape^*$ and MM^*, that lead to the lowest latency.

Algorithm 1 details the procedure adopted by the AutoTuner to achieve this objective. To avoid complex and typically imprecise high-level modeling of the HW/SW platform, our AutoTuner exploits HW-in-the-loop to search for the optimal solution. In particular, we use GVSoC [4], a behavioural event-based MCU simulator capable of modeling any PULP-based device with less than 5% error on estimated cycles. For each training step, the tool measures the latency on the target HW of a pool of configurations identified by the tuple (MM, $TileShape$) and selects the one leading to the lowest latency.

Algorithm 1: AutoTuner for PULP-TrainLib

Input: PULP L1 Memory Size $L1MemSize$ and $\#cores$; DNN Training Layer; Tensors' Shapes: $TensorShape$; Table 1
Output: PULP-TrainLib config: MM^*, $TileShape^*$
Parameter: K_T: max tiling solutions to evaluate on HW

```
1  Function Tiler(L1MemSize, T):
2      TensorList = ∅
3      for each Tile of T do
4          if Mem (Tile) < L1MemSize then
5              TensorList += (Tile, Mem (Tile))
6      TensorList = sort (TensorList);                    // Decreasing Mem
7      return Top-K_T(TensorList);                        // K_T Entries
8
9  AutoTuner:
10     if Mem (TensorShape) < L1MemSize then
11         TensorList = TensorShape                       // Single Entry
12     else
13         TensorList = Tiler (L1MemSize, TensorShape)
14     for each TensorShape in TensorList do
15         for each MM in Table 1 do
16             Lat = RunHW (DNN Train Layer, MM, TensorShape, #cores)
17             PerfLog += { Lat, TensorShape, MM }
18     return MM*, TileShape* = Top(PerfLog);             // Lowest Latency
```

The AutoTuner conducts a grid search to find the optimal tuple (MM^*, $TileShape^*$). Based on the layer type and the L1 memory size, a *Tiler* function computes all the possible tiling solutions and discards those exceeding the size constraint. The K_T feasible $TileShape$ featuring the largest memory occupations are selected and coupled with the MM kernels of Table 1 to form the pool of configurations to be tested on HW. For example, in the case of $K_T = 5$, used in all our experiments, the autotuning of the PointWise FW kernel demands $5 \times 24 = 120$ simulations on GVSoC (one for each of the 24 optimized MM kernels), which can run in parallel. Our AutoTuner setup exploits an Intel Xeon-equipped node running up to 64 simulations in parallel and takes less than 15 minutes to return the optimal configuration for all three training steps of the mentioned kernel. We remark that this process needs to be performed only once before the deployment on the target. On the contrary, if the Tensor shape fits the L1 memory entirely, the Tiler module can be bypassed, and the AutoTuner directly measures on HW the fastest MM kernel.

Thanks to the AutoTuner tool, we can effectively deploy full DNN model training on the PULP platform using *PULP-TrainLib*'s primitives. After the autotuning, every forward and backward function of the pipeline invokes specific

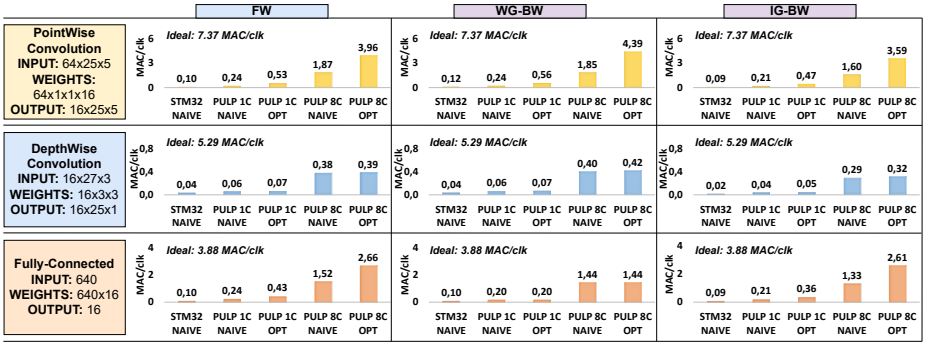


Fig. 4: Execution of the DNN training steps for a PointWise Convolution Layer (upper), a DepthWise Convolution Layer (middle), and a Fully-Connected Layer (lower), using the primitives from *PULP-TrainLib* library. Experiments are performed on an STM32L476RG and on PULP GVSoC.

MM kernels to achieve a computation faster than a "one-size-fits-all" setting, in which all the primitives are using the same MM function.

4 Experimental Results

In this section, we evaluate the effectiveness of the training primitives of *PULP-TrainLib* and the autotuning flow. First, we benchmark the training kernels of single layers on both the PULP Platform and a commercial STM32 MCU. Second, we analyze the outcome of our AutoTuner given multiple DNN layer steps and shapes. Third, we demonstrate how our methodology enhances on-device adaptation, by benchmarking the training procedure on full TinyML DNN models. Last, we compare the performances of *PULP-TrainLib* with AIfES, a state-of-the-art SW library targeting on-device training on embedded devices.

4.1 Latency Optimization on PULP-TrainLib

We evaluate the proposed *PULP-TrainLib* library by running experiments on the GVSoC simulator. Figure 4 reports the latency, expressed as the ratio between MAC operations and clock cycles (*MAC/clk*), of the forward and backward passes of multiple DNN layers fitting into the L1 memory: PointWise Convolution, DepthWise Convolution, and Fully-Connected. We compare the baseline configuration, which uses a naive MM, with the functions tuned by the AutoTuner in the case of both 1 and 8 cores execution. The figure shows the best kernel configuration returned by the AutoTuner (indicated as *OPT*), i.e., the one with the highest MAC/clk. Additionally, we report the ideal latency, obtained by accounting for only the load, store, and MAC operations that would occur in case of unbounded memory and no stalls. For comparison purposes, we also profile the PULP-TrainLib primitives on an off-the-shelf STM32L476RG MCU, featuring a

single-core ARM Cortex-M4 CPU with FPU support. All the experiments use the highest optimization level of the compiler (`-O3`).

The baseline PointWise and Fully-Connected kernels, when executed on a single core, achieve 0.24 MAC/clk during the forward step because they feature the same computational kernel (as shown in Fig. 2b). Also, the corresponding backward passes with naive MM show a similar performance level. On the contrary, the training steps of the DepthWise layers are $> 4\times$ slower than PointWise, mainly because of the overhead introduced by the *im2col* transformation and due to their inherently lower parallelism and possibility for data reuse.

When running on 8 cores, the parallel speedup of the baseline version is greater than 7.5 for all the training steps of a PointWise or a Fully-Connected Layer, thanks to the highly parallelizable nature of the MM kernel. Differently, the top speed-up for the training steps of the DepthWise Convolution is limited to 7 because of the *im2col* overhead. If compared with the STM32L476RG implementation, the execution of the baseline kernels on a single-core of PULP results to be up to 2.4 faster because of the low-latency FMA instructions, which take only 1 clock cycle to execute, $2\times$ lower than the ARM-Cortex M4 implementation. Furthermore, the execution on PULP is accelerated thanks to PULP's Hardware Loops, which reduce the branching overhead of the inner loop.

By leveraging the optimized matrix multiplication kernel, the OPT solution speeds up the execution by up to $2.4\times$ with respect to the baseline version, for both a single core and 8 parallel cores. In particular, the AutoTuner selects the *2×4* unrolled MM kernel for the training steps of the PointWise Layer in Fig. 4, mainly because the input matrices are significantly larger than the unrolling factor of the MM's inner loop. When combining the 8-core execution and the autotuned MM, the latency gain on PULP vs the STM32L476RG implementation grows up to $36.6\times$.

In the case of a Fully-Connected Layer, instead, the Autotuner selects the `mm_unroll_1x4`, `mm` or `mm_M_unroll_1x8` for, respectively, the FW, the WG-BW, and IG-BW steps, with a performance of 2.66, 2.61, and 1.44 MAC/clk on 8 cores. Since a Fully-Connected Layer mainly relies on matrix-vector and vector-vector multiplications (as shown in Fig. 2), the most effective unrolling schemes feature a dimension of size 1 on the vector's sides.

Overall, if compared with the ideal scenario, the autotuned solution significantly increases FPU utilization, computed as the ratio between real and ideal MAC/clk performance, from 0.25 to 0.54 in the case of PointWise FW execution.

4.2 Effect of tensor shapes on AutoTuning

To better explore the output of the AutoTuner with respect to the tensor shapes, Figure 5 shows the latency on 8 cores of two training primitives (PointWise FW and FC IG-BW) applied on multiple tensor *Shapes*. The tested shapes are detailed in the Figure and fit the L1 memory. Every plot reports the latency of the Top-5 fastest kernel configurations normalized and compared to the baseline performance.

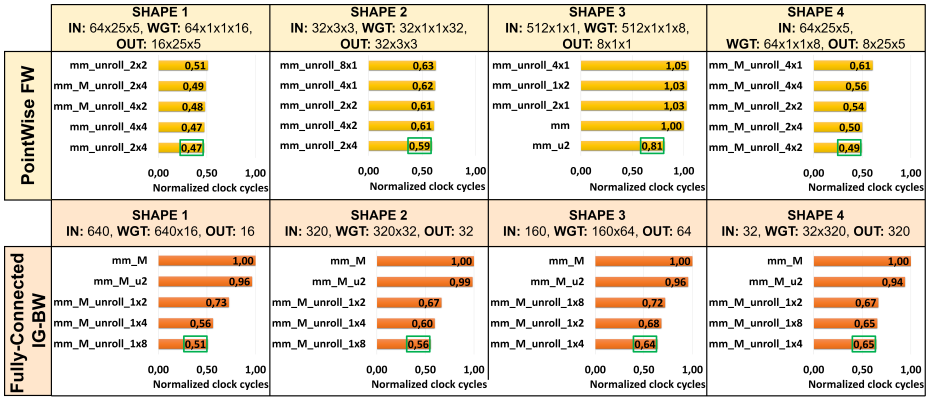


Fig. 5: Evaluation of the AutoTuner on a Pointwise Convolution Layer performing a Forward step (upper) and a Fully-Connected Layer performing an Input Gradient backpropagation step (lower) using multiple tensor Shapes (1-4). Every plot shows the latency measurements normalized with respect to the baseline, when using different optimized MM kernels.

In PointWise convolution, the top AutoTuner solution outperforms the baseline by 1.6-2.2×. The fastest solutions feature 2×4 or 4×2 unrolled MM kernels because the sizes of the involved matrices exceed both the number of cores and the maximum unrolling factors. An exception is represented by the *Shape 3* case: the presence of a tight input matrix, with shape 512×1×1, leads to the computation of a high number of consecutive elements in the inner loop but a single iteration for each outer loop. For this reason, the only MM capable of accelerating the execution is `mm_u2`, which unrolls two successive products in K, reducing the data dependencies from adjacent FMA and LD which are present in the inner loop of the naïve implementation. On the other hand, the matrix-vector or vector-vector products featured by the Fully-Connected primitives result faster in the case of 1×V unrolling. As can be seen by comparing Shapes 1, 2 and 3, 4, Fully-Connected Layers with tighter input sizes privilege 1×4 unrolling, while wider ones benefit from 1×8 unrolling.

4.3 TinyML Benchmark Training

To assess the effectiveness of our approach on real DNN models, we deploy on PULP a forward and a backward pass of a Deep Autoencoder for Anomaly Detection and a DS-CNN for Keyword Spotting. Differently from previous experiments, the AutoTuner concurrently searches for both the fastest MM kernels and the layer-wise tile shapes. Figure 6 shows the latencies of a forward and a backward pass obtained by applying the AutoTuner over the two considered benchmarks (OPT bars). These solutions are compared with other implementations featuring the same tiling configuration but using fixed MM kernels, i.e., a "one-size-fits-all" approach.

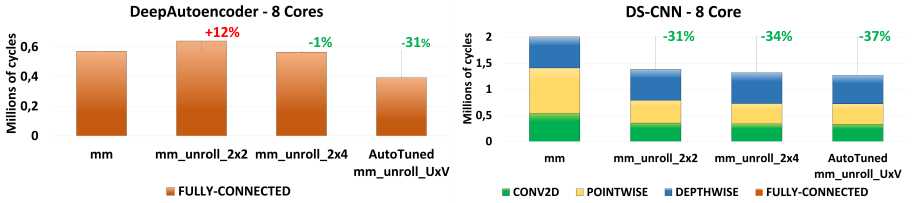


Fig. 6: Training time of two TinyML benchmarks - a Deep AutoEncoder (left) and a DS-CNN (right) - running on 8 cores and using 64 kB of L1 memory. The plot compares the autotuned vs "one-size-fits-all" solutions, agnostic to the DNN's structure.

Observing Figure 6, the usage of a "one-size-fits-all" approach is particularly weak in the case of a Deep Autoencoder, which is composed of a sequence of Fully-Connected layers. In this case, the usage of 2×2 or 2×4 unrolling implies the presence of all-leftover computation, leading to a slow-down of 12% in the case of a fixed 2×2 optimization, and a very limited speed up of 1% in the case of a 2×4, when executing on 8 cores. Instead, the usage of autotuned kernels, which are mainly represented by 1×2, 1×4, and 1×8 unrolling, with both parallelism in N and M depending on the training step, introduces a high performance gain of 31% on 8 cores.

Concerning the DS-CNN, the large tensor sizes encourage the use of wide unrolling factors. However, 2×2 or 2×4 unrolling factors with parallelism in N are 6% and 3% slower than the autotuned execution. The use of 2×4 and 4×2 for Convolutions and of 1×4 and 1×8 for the last Fully-Connected classifier, each with both parallelism in N and M, produces a significant and uniform speed up.

Lastly, the introduction of autotuning on full DNN training steps of TinyML benchmarks enables real-time network updates, with a top performance of 0.4 million cycles for a Deep Autoencoder and 1.3 million cycles for a DS-CNN.

On a PULP-based device like Vega [26], running at 450 MHz, these performances result in 0.9 ms and 3 ms to execute a single training step. These latencies prove to be 15× and 108× faster, respectively, than the results available over the same models on a RISC-V MCU on MLCommons [9], which on the contrary performs a single inference step only, i.e no backpropagation.

4.4 Comparison with the State of the Art

As a comparison with the current state-of-the-art, we profile the same-task performances of *PULP-TrainLib* and AIfES[10], Fraunhofer IMS's on-device learning software library. For this purpose, we configure AIfES to use the MM kernels provided by the optimized ARM CMSIS library. Since the current version of AIfES does not provide Convolutional Layers, we profile its performances over

[9] TinyMLPerf results: https://mlcommons.org/en/inference-tiny-05/
[10] AIfES for Arduino: https://www.arduino.cc/reference/en/libraries/aifes-for-arduino/

the Deep Autoencoder of Figure 6, which is composed of Fully-Connected layers. The execution of AIfES kernels is measured, in clock cycles, on an ARM Cortex-M4-based STM32F407VGT6 MCU.

Figure 7 presents the results of a training step of the Deep Autoencoder with a batch size of 1 (i.e., the model forwards and backwards a single image at a time). With this setup, a complete model training with AIfES takes about 12 million clock cycles to execute (71.9 ms on an STM32F4 at 168 MHz), 3.3× higher than a non-optimized single-core training with *PULP-TrainLib*.

Compared to PULP-TrainLib, we notice a high slowdown in the execution of the IG-BW step. This is due to to the vector-vector product on which this step relies, which induces an all-leftover computation on the CMSIS-NN unrolled kernels, requiring 19 instructions to execute each MM inner-loop iteration (2 ld, 1 fma, 2 add/sub, 1 branch, and 13 others to monitor the state of the computation and iterate over the outer loops). For this reason, this step suffers a 5.9× slowdown with respect to our baseline, single-core implementation, which is built to handle vector-vector and vector-matrix products with less overhead.

Furthermore, the vector-matrix products of the FW and IG-BW steps require roughly 2× the clock cycles of our single-core execution with AIfES. Even with unrolled CMSIS MM kernels, DNN kernels on the STM32F407 require 19 instructions to execute (8 ld, 4 fma, 6 add/sub, 1 branch). On the PULP RISC-V cores, the branching overhead can be avoided employing Hardware Loops.

Overall, we observe that the application of the AutoTuner leads to a 4.5× performance increase with respect to AIfES. On 8 parallel cores, we achieve an average parallel speedup of over 6.5× over each tile and layer of the DNN model, with respect to both our non-optimized and optimized single-core execution. Therefore, we verify that our AutoTuner is able to effectively achieve the same degree of optimization on both a single and a multicore training setup, for each tile and layer of the DNN model. This results in a 30.7× higher performance with respect to AIfES, on 8 parallel RISC-V cores.

5 Conclusion

In this paper, we presented *PULP-TrainLib*, the first software library for DNN on-device training on multi-core RISC-V-based MicroController Units (MCUs). Included into *PULP-TrainLib*, we introduced an *AutoTuner*, capable of jointly selecting the fastest matrix multiplication kernel optimization and tile size for a given DNN layer configuration. By applying the *AutoTuner* on the training primitives of *PULP-TrainLib*, we are able to run complete training steps of TinyML DNN models in almost real-time.

6 Acknowledgements

This work was supported in part by the EU Horizon 2020 Research and Innovation Project WiPLASH under Grant 863337 and in part by the ECSEL Horizon 2020 Project AI4DI under Grant 826060.

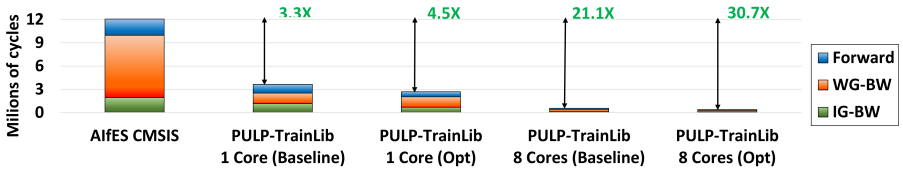


Fig. 7: Comparison of PULP-TrainLib with Fraunhofer IMS's AIfES training library for MCUs, using CMSIS as computational backend on a complete Deep Autoencoder model. AIfES' layer tiles are executed on an STM32F407VGT6 MCU.

References

1. Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., Mané, D.: Concrete problems in ai safety (2016)
2. Ankit, A., Hajj, I.E., Chalamalasetti, S.R., Agarwal, S., Marinella, M., Foltin, M., Strachan, J.P., Milojicic, D., Hwu, W.M., Roy, K.: Panther: A programmable architecture for neural network training harnessing energy-efficient reram. IEEE Transactions on Computers **69**(8), 1128–1142 (2020)
3. Ashouri, A.H., Killian, W., Cavazos, J., Palermo, G., Silvano, C.: A survey on compiler autotuning using machine learning. ACM Comput. Surv. **51**(5) (sep 2018)
4. Bruschi, N., Haugou, G., Tagliavini, G., Conti, F., Benini, L., Rossi, D.: Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In: 2021 IEEE 39th Intern. Conference on Computer Design (ICCD). pp. 409–416 (2021)
5. Cai, H., Gan, C., Zhu, L., Han, S.: Tinytl: Reduce activations, not trainable parameters for efficient on-device learning (2021)
6. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: Tvm: An automated end-to-end optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 578–594. USENIX Association, Carlsbad, CA (Oct 2018)
7. Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., Krishnamurthy, A.: Learning to optimize tensor programs (2019)
8. Chen, Y.R., Liao, H.H., Chang, C.H., Lin, C.C., Lee, C.L., Chang, Y.M., Yang, C.C., Lee, J.K.: Experiments and optimizations for tvm on risc-v architectures with p extension. In: 2020 Intern. Symposium on VLSI Design, Automation and Test (VLSI-DAT). pp. 1–4 (2020)
9. David, R., Duke, J., Jain, A., Reddi, V.J., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., Regev, S., Rhodes, R., Wang, T., Warden, P.: Tensorflow lite micro: Embedded machine learning on tinyml systems (2021)
10. Frigo, M., Johnson, S.: Fftw: an adaptive software architecture for the fft. In: Proc. of the 1998 IEEE Intern. Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181). vol. 3, pp. 1381–1384 vol.3 (1998)
11. Konečný, J., McMahan, H.B., Yu, F.X., Richtárik, P., Suresh, A.T., Bacon, D.: Federated learning: Strategies for improving communication efficiency (2017)
12. Kopparapu, K., Lin, E.: Tinyfedtl: Federated transfer learning on tiny devices (2021)

13. Lai, L., Suda, N., Chandra, V.: Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus (2018)
14. Lee, S., Nirjon, S.: Neuro.zero: A zero-energy neural network accelerator for embedded sensing and inference systems. In: Proceedings of the 17th Conference on Embedded Networked Sensor Systems. p. 138–152. SenSys '19, Association for Computing Machinery, New York, NY, USA (2019)
15. Lin, J., Chen, W.M., Lin, Y., Cohn, J., Gan, C., Han, S.: Mcunet: Tiny deep learning on iot devices (2020)
16. Losing, V., Hammer, B., Wersing, H.: Incremental on-line learning: A review and comparison of state of the art algorithms. Neurocomputing **275**, 1261–1274 (2018)
17. McMahan, B., Moore, E., Ramage, D., Hampson, S., Arcas, B.A.y.: Communication-Efficient Learning of Deep Networks from Decentralized Data. In: Singh, A., Zhu, J. (eds.) Proc. of the 20th Intern. Conference on Artificial Intelligence and Statistics. Proc. of Machine Learning Research, vol. 54, pp. 1273–1282. PMLR (20–22 Apr 2017)
18. Mills, J., Hu, J., Min, G.: Communication-efficient federated learning for wireless edge intelligence in iot. IEEE IoT Journal **7**(7), 5986–5994 (2020)
19. Murshed, M.G.S., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G., Hussain, F.: Machine learning at the network edge: A survey. ACM Computing Surveys **54**(8), 1–37 (Nov 2022)
20. Mustafa, D.: A survey of performance tuning techniques and tools for parallel applications. IEEE Access **10**, 15036–15055 (2022)
21. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Trans.S on Knowledge and Data Engineering **22**(10), 1345–1359 (2010)
22. Pellegrini, L., Graffieti, G., Lomonaco, V., Maltoni, D.: Latent replay for real-time continual learning. In: 2020 IEEE/RSJ Intern. Conference on Intelligent Robots and Systems (IROS). pp. 10203–10209 (2020)
23. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not. **48**(6), 519–530 (jun 2013)
24. Ravaglia, L., Rusci, M., Nadalini, D., Capotondi, A., Conti, F., Benini, L., Benini, L.: A tinyml platform for on-device continual learning with quantized latent replays. IEEE Journal on Emerging and Selected Topics in Circuits and Systems (2021)
25. Ren, H., Anicic, D., Runkler, T.: Tinyol: Tinyml with online-learning on microcontrollers (2021)
26. Rossi, D., Conti, F., Eggimann, M., Di Mauro, A., Tagliavini, G., Mach, S., Guermandi, M., Pullini, A., Loi, I., Chen, J., Flamand, E., Benini, L.: Vega: A ten-core soc for iot endnodes with dnn acceleration and cognitive wake-up from mram-based state-retentive sleep mode. IEEE Journal of Solid-State Circuits (2021)
27. Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions (2018)
28. Wang, F., Zhang, M., Wang, X., Ma, X., Liu, J.: Deep learning for edge computing applications: A state-of-the-art survey. IEEE Access **8**, 58322–58336 (2020)
29. Wang, X., Magno, M., Cavigelli, L., Benini, L.: Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things. IEEE IoT Journal **7**(5), 4403–4417 (2020)
30. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software. In: SC '98: Proc. of the 1998 ACM/IEEE Conference on Supercomputing. pp. 38–38 (1998)