DTMM: Deploying TinyML Models on Extremely Weak IoT Devices with Pruning

Lixiang Han, Zhen Xiao, Zhenjiang Li
Department of Computer Science, City University of Hong Kong, China

***Abstract*—DTMM is a library designed for efficient deployment and execution of machine learning models on weak IoT devices such as microcontroller units (MCUs). The motivation for designing DTMM comes from the emerging field of tiny machine learning (TinyML), which explores extending the reach of machine learning to many low-end IoT devices to achieve ubiquitous intelligence. Due to the weak capability of embedded devices, it is necessary to compress models by pruning enough weights before deploying. Although pruning has been studied extensively on many computing platforms, two key issues with pruning methods are exacerbated on MCUs: models need to be deeply compressed without significantly compromising accuracy, and they should perform efficiently after pruning. Current solutions only achieve one of these objectives, but not both. In this paper, we find that pruned models have great potential for efficient deployment and execution on MCUs. Therefore, we propose DTMM with pruning unit selection, pre-execution pruning optimizations, runtime acceleration, and post-execution low-cost storage to fill the gap for efficient deployment and execution of pruned models. It can be integrated into commercial ML frameworks for practical deployment, and a prototype system has been developed. Extensive experiments on various models show promising gains compared to state-of-the-art methods.**

I. Introduction

Current Internet of Things (IoT) systems make wide use of resource-constrained devices such as microcontroller units (MCUs) for data acquisition in various sensor network applications in industry [10], healthcare [13], [41], agriculture [23], and smart city [40], because MCUs have ultra-low power consumption, in milliwatts or microwatts [4], [20], which are suitable for sustainable and widespread deployment. Recent studies find that running machine learning (ML) models on such end devices can further provide useful device-side data processing to avoid frequent data transmissions (thereby reducing energy consumption), preserve data privacy and enable new application scenarios [12]. Therefore, an emerging field of tiny machine learning (TinyML) [3], [9], [48] has appeared recently, focusing on adopting ML models on embedded IoT devices to expand their reach for ubiquitous intelligence.

However, the size and computation cost of ML models are usually not small, *e.g.*, on the order of MB [16], [42], which contradicts to the extremely limited resource on MCUs. For example, a typical MCU (such as NUCLEO-F446RE) has only 512 KB of static random-access memory (SRAM) for temporary runtime data storage, 1 MB of on-chip embedded flash (eFlash) memory for program storage, and 180 MHz CPU frequency for task processing. Computation offloading is recently used to facilitate the deployment of ML models on MCUs to bypass their resource constraints [19]. However, offloading relies on additional infrastructure, such as edge for support, which is not conducive to large-scale deployment. To achieve fully autonomous operation and local inference, we revisit model compression and find that weight pruning [21], [28] is still a viable but underexploited solution.

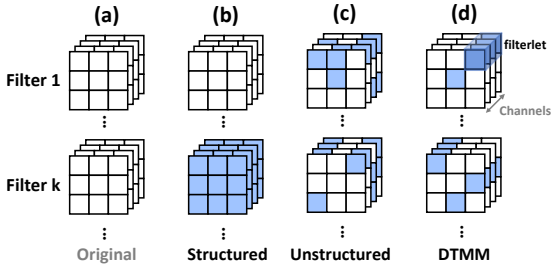

Fig. 1. Illustration of weight pruning to fit TinyML models on MCUs. Blue squares represent the weights to be pruned. (a) One convolution layer contains multiple filters. (b) Structured pruning removes all the weights from the selected filter(s). (c) Unstructured pruning can remove arbitrary weights. (d) DTMM removes all the weights from the selected filterlets.

Commercial ML frameworks, such as Tensorflow Lite Micro [9] and CMSIS-NN [27], already supports *structured* pruning [28], [44], [49], which treats each filter (or other structures (§VI)) as an atomic unit to remove all weights of selected filters to reduce the amount of model weights (Figure 1(b)). Structured pruning is easy to implement, but the pruned filters may still contain many useful weights, leading to coarse-grained pruning. The model accuracy drop significantly when high compression ratios are required on the MCU.

To improve pruning granularity, another solution called *unstructured pruning* [15] is proposed to prune arbitrary weights (Figure 1(c)). It achieves high accuracy by identifying and removing less important weights, but introduces two unique challenges on MCUs. First, due to the removal of arbitrary weights, each unpruned weight needs an index to record its existence, which incurs significant additional storage costs and lacks an efficient storage mechanism to fit the MCU's tight memory. Second, more importantly, the format of the remaining weights, indicated by their index values, is incompatible with commercial ML frameworks. It requires non-trivial overhead to convert these weights prior to inference [32], which can significantly slow down model inference with low-end MCU processors (see §III-B for details).

To overcome these problems, we are inspired by recent attempts [34], [49] to group weights to form new pruning units with a granularity between entire filters and individual weights (§VI). We choose to group an entire line of weights at the same position across all channels in the filter as an atomic pruning

TABLE I
SPECIFICATION COMPARISON BETWEEN MCU AND OTHER PLATFORMS. "RAS" AND "N" ARE SHORT FOR RASPBERRY AND NUCLEO.

Device	Freq.	RAM	Storage	Power
Jetson NX	1.1–1.9 GHz	8 GB	16 GB	20 W
Pixel 5	1.8–2.4 GHz	8 GB	128 GB	1.2 W
RAS Pi 3B	1.2 GHz	1 GB	Micro SD	1.3 W
N-F446RE	180 MHz	128 KB	512 KB	0.1 W

unit for the MCU and name it *filterlet*. Figure 1(d) shows that two and three filterlets (blue ones) are to be removed from Filter 1 and k, respectively. Removing weights in the granularity of filterlets allows more flexible and finer-grained results than pruning filters, which thus can achieve higher compression ratio yet with little compromise of accuracy.

However, if we only change the pruning unit to filterlets, similar limitations to unstructured pruning still exist. The main reason we use this new unit is because we observe that weights in each filterlet are stored contiguously on the MCU (§III). This local memory continuity can be used to design a new data structure to efficiently store pruned models and a novel operator to significantly speed up model inference, based on which we can also devise a specialized scheduler for deriving the optimal pruning strategy given resource constraints before actual pruning, resulting in a systematic and practical solution to deploy TinyML models on MCUs.

In this paper, we enable all these designs in DTMM, a full-stack library that fills in the important but missing pieces of post-execution low-cost storage, runtime inference acceleration and pre-execution pruning optimization to efficiently deploy TinyML models on MCUs through pruning. Pruning using filterlets in DTMM is essentially a hybrid of the two types of pruning described above to leverage the advantages of both, but pruning is not the only step in deploying a model, and DTMM further avoids the respective shortcomings, making the pruned model finally usable and running on the real device.

We develop a prototype of DTMM, integrate it into TensorFlow Lite Micro [9] and evaluate its performance on Arm Corstone-300 of the Cortex-M55 processor [2] using various models, including VGG-11 [42], ResNet-12 [16] and YOLO [39] on three datasets: CIFAR-10 [25], Visual Wake Words (VWW) [8], and Face Detection Dataset Benchmark (FDDB) [22]. We compare DTMM with state-of-the-art structured and unstructured methods, CHIP [44] and PatDNN [37]. Overall, DTMM outperforms the structured method CHIP by reducing model size and inference latency up to 42.8% and 27.7%, respectively, without compromising accuracy. Compared to the unstructured method PatDNN, DTMM achieves up to 33.7% and 74.6% reduction on model size and inference latency, respectively, without compromising accuracy. In summary, this paper makes the following contributions:

- We propose a novel library called DTMM for deploying TinyML models on weak IoT devices to achieve high compression ratio, high accuracy and small overhead.
- We choose a suitable pruning unit and design a set of new techniques for model storage, operator acceleration and optimization to realize the design of DTMM, which is compatible to commercial ML frameworks.
- We develop a prototype of DTMM and extensively evaluate its performance using different ML models and rich datasets, showing remarkable performance gains.

II. BACKGROUND

A. Computation Capacity of MCU

MCU [43] is a small computer on a compact integrated circuit chip with CPU, memories, I/O peripherals, etc.

1) Hardware footprint. MCU is designed with low-end CPU (with 32–216 MHz operating frequencies, consuming only milliwatts or microwatts [4]) and extremely small memories, such as SRAM (for temporary runtime data) in the range of 8–320 KB [43] and eFlash (for program and data storage) in the range of 64–1024 KB. The hardware footprint in the table shows that MCUs are typically small, low-cost and low-power. These advantages make MCUs economical and sustainable for widespread deployment in IoT applications [40].

2) Comparison with other platforms. In addition to weak devices, there are other popular computing platforms in IoT systems such as edge AI boards (*e.g.*, Jetson NX) [1], mobile devices (*e.g.*, Pixel 5) [24], [47], and Raspberry Pi (*e.g.*, Pi 3B) [38]. Unlike the low cost of a few dollars for an MCU, these advanced platforms may cost hundreds of dollars to bring more computing resources, including a powerful CPU with GHz operating frequency and at least 1 GB memory. Table I summarizes the hardware specification differences between the MCU (*e.g.*, NUCLEO-F446RE) and other computing platforms. In addition, these platforms are equipped with multi-core CPUs and GPUs. They can divide computing tasks into multiple threads and assign them to different cores to execute in parallel [46], which are not supported on MCUs.

B. Use of TinyML in IoT Applications

In smart cities [50], MCUs are widely used in various sensors for data acquisition [51]. By running TinyML models locally, they can analyze sensory data and avoid frequent data transmissions (the main energy consumer [45]), which saves energy and helps extend the battery life. In tiny autonomous machines, ML models can also bring intelligence to small autonomous machines by performing more complex tasks, such as path planning and navigation in nano-drones [10]. In addition to saving energy, on-device processing can also preserve data privacy, such as health data on wristbands [13] and user speech samples in keyword spotting [52].

C. Model Deployment Process on MCU

The process of deploying ML models with pruning on the MCU follows four main steps:

Step 1): The pruner estimates the importance of each potential pruning unit (*e.g.*, filterlets), which can be quantified by various metrics [15], [31]. The pruner then decides which weight units should be pruned based on the memory budget and performance requirements. This step in DTMM is handled by our pruning strategy scheduler in §III-C.

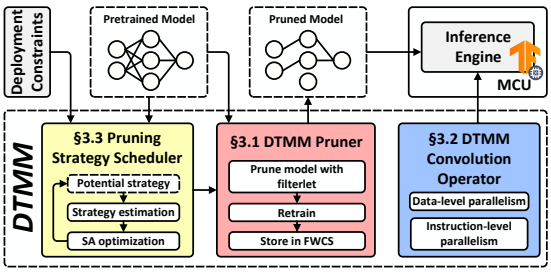

Fig. 2. Overview of the DTMM design.

Step 2): Given a pruning strategy, *i.e.*, which weight units to prune, the pruner only resets them to zero in this step, because the model needs to be retrained (or fine-tuned). The retraining process only updates the unpruned weights.

Step 3): After retraining, the pruner applies the pruning strategy to remove weights and uses an appropriate data structure to store the pruned model with reduced model size. Quantization is also often used to further shrink the model due to tight memory and flash space on MCUs. This step in DTMM is covered by our storage structure design in §III-A.

Step 4): The pruned model can then be deployed on the MCU and operated by the ML framework for execution. Since commercial ML frameworks currently only support structured pruning, suitable runtime support is also required for deployment. A naive solution is to zero-pad the pruned weights to restore the structure of all filters, but this leads to unacceptable computational and storage overhead. Hence, we propose a novel operator design in §III-B that can run DTMM-pruned models directly to significantly speed up model inference.

III. System Design

The architecture of the DTMM design is illustrated in Figure 2 and consists of three main components:

- **DTMM pruner (§III-A)**: it prunes weights at a granularity of filterlet and stores the remaining discrete weights with a compact data structure, called FWCS, that is efficient for both storage and inference.
- **DTMM convolution operator (§III-B)**: it enables unpruned weights stored in FWCS to run in existing ML frameworks without reconstruction overhead, and accelerates the inference via single instruction, multiply data (SIMD) and instruction-level parallelism techniques.
- **Pruning strategy scheduler (§III-C)**: it provides a pruning strategy for the pruner by formulating the search of the optimal strategy as an optimization problem to minimize the latency with accuracy and memory constraints.

A. DTMM Pruner

We first introduce the design of DTMM pruner, starting with a discussion of the problems of existing structured and unstructured pruning methods used on IoT devices.

1) Problems in existing pruning methods: In a convolution layer, the weights are comprised of a set of *filters*. Each filter contains C *kernels*, and one kernel has $H \times W$ weights. Different from other platforms, in commercial ML frameworks for MCUs, weights at the same position across all the channels are stored *contiguously* in physical storage following a *channel-major* order [27], as shown in Figure 3.

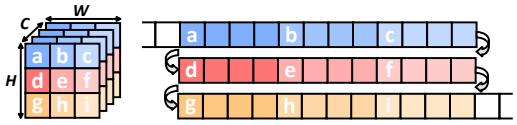

Fig. 3. Weights in filters are stored contiguously in physical storage following a channel-major order. We plot the physical storage in three lines.

Structured pruning. When applying a pruning method, in addition to storing the values of the unpruned weights, their positions in the convolution layer should also be recorded to index the corresponding values from input feature maps during inference. The structured pruning methods [28], [44] remove weights at the granularity of filters, thus directly reducing model size. Meanwhile, it requires no additional index overhead since all computations are removed for each pruned filter. However, when the model needs to be deeply pruned for MCUs, coarse-grained pruning granularity (*i.e.*, the entire filter) can affect the model accuracy.

Unstructured pruning. These methods [15], [37] preserve model accuracy by carefully choosing arbitrary weights to remove, but require extra storage to index the unpruned weights. Figure 4 shows a typical compressed sparse row (CSR) structure used in unstructured pruning [32]. Weights from different filters within the same convolution layer are stored consecutively. The values of the unpruned weights are stored in an array `Arr`, and their positions are represented using another two arrays `cPtr` and `fIdx`. [1] Unstructured pruning introduces at least one index to record each unpruned weight, making the model size after pruning much larger than expected. More importantly, the format of unpruned weights, indicated by their index values, is not compatible with commercial ML frameworks. Recovering them to an executable format before inference requires non-trivial overhead, which can significantly slow down model inference on MCUs [32].

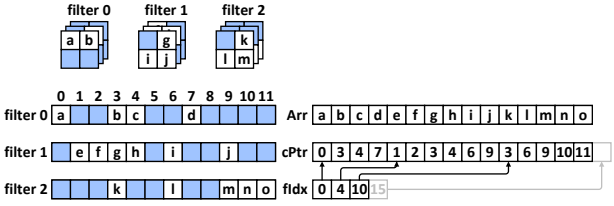

Fig. 4. Illustration of how unpruned weights from three filters are managed with CSR structure in unstructured pruning.

Therefore, to prune the ML model with high accuracy, less storage and small execution overhead for MCUs, we next introduce the weight pruning with filterlet and the new data structure design to efficiently store the unpruned weights.

2) Weight pruning and storage in DTMM: **Filterlet unit.** For each filter, we group all the weights at the same position across all channels to form a *filterlet*. Figure 5 shows how to utilize filterlet for pruning. In this example, each filter

[1] `cPtr` records the index of each unpruned weight in their corresponding filters. For `fIdx`, its length represents the amount of remaining filters. Each element in `fIdx` is a pointer pointing to the element in `cPtr`, which stores the position of the first unpruned element for the current filter.

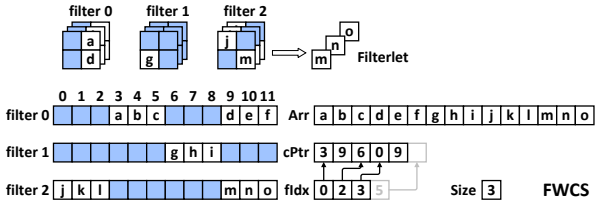

Fig. 5. Illustration of how weights are pruned with filterlet and how remaining weights are stored using FWCS.

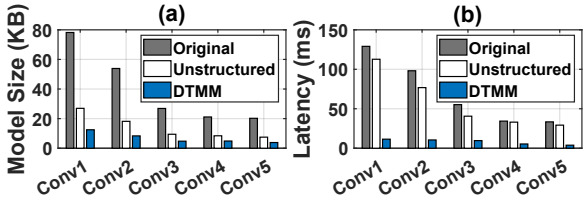

Fig. 6. (a) Storage usage and (b) inference latency of convolution layers with 90% weights pruned by different methods.

contains three channels, so a filterlet contains three weights, *e.g.*, weights $\langle m, n, o\rangle$ represent a filterlet of "filter 2". Since weights are stored in a channel-major order on MCUs (Figure 3), all weights in each unpruned filterlet are located in contiguous space, which enables an efficient operator design for inference (§III-B). Compared to structured pruning that regards the whole filter as an atomic unit to prune, filterlet provides finer granularity to preserve important weights in each filter for better accuracy (§III-C).

Compressed weight storage. Since each filterlet is part of the filter, the weights from unpruned filterlet also require extra storage to record their positions in the filter. To this end, we employ the key observation that all weights in a filterlet are located in contiguous space to propose a compact data structure, called filterlet weight compressed storage (FWCS). Similar to the unstructured pruning, FWCS stores the values of the remaining weights in an array `Arr`. The position information of each weight is encoded in other three arrays:

1) `size`: it stores the filterlet size that is measured by the number of weights. For example, `size` is 3 in Figure 5.

2) `cPtr`: it stores the index of the first weight of a filterlet in the corresponding filter. Each element of `cPtr` corresponds to one filterlet. For example, in Figure 5, the second element of `cPtr` is 9, meaning that the index of the first weight d in the second filterlet ($\langle d, e, f\rangle$) is 9 in its filter.

3) `fIdx`: each of its element represents one filter in the current convolution layer. The value is a pointer to the index in `cPtr`, which stores the position of the first remaining filterlet for the current filter. For example, in Figure 5, the second element of `fIdx` represents the second filter ("filter 1"), and its value 2 means that the first filterlet $\langle g, h, i\rangle$ of the second filter starts from column `cPtr[2]` (= 6).

The effectiveness of FWCS. Unlike unstructured pruning, we can store the position of each filterlet instead of individual weights, leading to a substantial reduction in model size. With the number of channels C, removing the weight with filterlet reduces the amounts of indexes by C times. Hence, in Figure 4 and 5, pruning with filterlet results in less storage overhead with the same number of pruned weights.

In Figure 6, we conduct a preliminary experiment to compare the model size after pruning between unstructured pruning and filterlet. We prune 90% weights for five convolution layers with different configurations. Figure 6(a) shows that FWCS can reduce the model size by 49.6% on average compared to unstructured pruning.

In addition to reducing model size, pruning with filterlet also enables efficient model inference. Direct execution of our pruned model in ML frameworks is slow (§III-B), as FWCS lacks specialized runtime optimizations. To this end, we further design an efficient convolution operator to significantly speed up model inference. Next, we introduce this operator.

B. DTMM Convolution Operator

1) Convolution operation at nutshell: A convolution layer performs a convolution operation on its filters and input feature maps. Each filter slides across the input feature maps, and the dot product between the filter and input feature maps is computed to generate the output feature map. Specifically, each value of the output feature map $o_{x,y,n}$ is computed by:

$$o_{x,y,n} = \sum_{h=0}^{H-1}\sum_{w=0}^{W-1}\sum_{c=0}^{C-1}(k^n_{h,w,c} \times f_{x+h,y+w,c}), \quad (1)$$

where $k^n_{h,w,c}$ and $f_{x+h,y+w,c}$ are the values of the n-th filter and input feature maps, respectively.

A naive way to compute Eq. (1) is to use nested loops, which is very slow. Therefore, ML frameworks for MCUs typically utilize the single instruction, multiple data (SIMD) technology to accelerate computation [27]. With SIMD, multiple operations in difference lanes (operands) are performed in parallel within an instruction achieving higher throughput.

The SIMD multiply-accumulate (MAC) operation is the main building block to implement dot product between the filter and input feature maps in a convolution layer. It computes and adds the dot product of two numbers to an accumulator ($a = a + \mathbf{b} \cdot \mathbf{c}$). SIMD then exploits the data-level parallelism of the hardware to increase the speed of computation based on the maximum lane number of the operands.

2) DTMM convolution with SIMD: SIMD requires the weights in a filter to be contiguous in physical memory to efficiently load the weights and corresponding values in input feature maps for computation. However, due to the fine-grained weight removal, the unpruned weights of DTMM pruning (as well as unstructured pruning) become discrete, preventing a direct adoption of SIMD for speedup.

1) Opportunity. Unlike individual weight removal in unstructured pruning, the weights of a filterlet are contiguous in the physical memory since they come from the same kernel position across all channels. Unpruned weights in a convolution layer can be discrete, but the local memory continuity of weights in filterlet brings further opportunities to exploit SIMD for inference speedup.

2) When SIMD meets filterlet pruning. Figure 7 illustrates how it works. In this example, a filterlet has eight weights, and the lane number of the SIMD instruction is four. Since filterlet

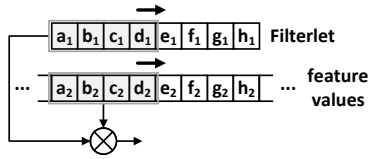

Fig. 7. Weights in filterlet can be operated by SIMD.

weights are contiguous in physical memory, SIMD multiply-accumulate (MAC) can compute the dot product for four pairs of weights and feature map values in one instruction, instead of four. The speedup gain increases as the number of SIMD lanes increases, which is 2–16 in practice.

In general, DTMM convolution works as follows (Algorithm 1). The operator slides over the input feature map during convolution. For each step of sliding, a patch of the input feature map is prefetched, from which the operator extracts the values corresponding to the weights in filterlet through the index information stored in FWCS. The dot product between them is then performed using SIMD MAC operations.

Algorithm 1: DTMM Convolution Operator

Input: Input feature maps: I; Filters in FWCS format: W; Height, width and channel of output maps: H_{out}, W_{out}, C_{out}; Height and width of filters: H_{ker} and W_{ker}; Convolution stride: $stride$; Number of SIMD lanes: l;

Output: Output feature maps O;

```
1  for h = 0 to H_out − 1 do
2      for w = 0 to W_out − 1 do
3          buf ← Prefetch(I, h × stride, w × stride, H_ker, W_ker) ;
           // Fetch data necessary for O_{h,w,c} computation
4          i ← 0 ;
5          for c = 0 to C_out − 1 do
6              for j = W.fIdx_c to W.fIdx_{c+1} − 1 do
7                  for k = W.cPtr_j to W.cPtr_j + W.size − 1 step l do
8                      O_{h,w,c} ← buf_{k:k+l} · W.arr_{i:i+l} + O_{h,w,c} ;
9                      i ← i + l;
```

```
1:     wlstp.8          lr, %[cnt], 1f
2: loopStart:
3:     vldrb.8          q0, [%[arr]], #16// Load weights
4:     vldrb.8          q1, [%[in]], #16 // Load feature values
5:     vmladava.s8      %[out], q0, q1   // MAC operation
6:     letp             lr, loopStart
```

Fig. 8. Main code block for our convolution operator with SIMD instructions on Armv8.1-M architecture.

We further transplant Algorithm 1 into the ML framework, and Figure 8 shows the code block of our operator's main operation. In particular, SIMD load `vldrb.8` loads operand data into registers, *e.g.*, `q0` and `q1`, and the SIMD MAC `vmladava.s8` computes dot product between the operand vectors. If the registers cannot hold a complete filterlet, the computation is performed in iterations, and `wlstp.8` is used to record the number of iterations in `lr`. At the end of each iteration in line-6, `lr` is decremented by the number of lanes. If `lr` is non-zero, computation is repeated starting from line-2.

The speedup of our operator is mainly due to data-level parallelism, benefiting from the local weight continuity in a filterlet to take advantage of SIMD. In addition to this data-level speedup, we also observe further acceleration opportunities through instruction-level parallelism in the next subsection.

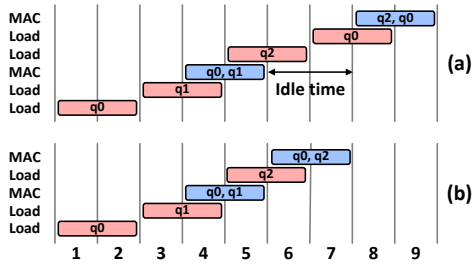

Fig. 9. (a) Inefficient and (b) improved workflow of operations.

3) Instruction-level acceleration: When our convolution operator works, we observe that in Figure 8 there are two types of instructions executed by two different hardware components on the MCU, including

- memory unit: it executes SIMD loads, and
- arithmetic logic unit (ALU): it executes SIMD MACs.

Since the memory unit and ALU are independent hardware components, there is an opportunity for these two types of instructions to be executed in parallel to further speed up computation, while we encounter the following problem.

1) Problem. Figure 9(a) shows a snapshot of the memory unit (Load) and ALU (MAC) over time as we apply our operator to perform convolution. With recent vector processing technology [33], after partially loading data in the first CPU cycle, the ALU can immediately use it for computation, resulting in the overlap of Load and MAC, *e.g.*, in cycle "4".

We can see that in Figure 9(a), the ALU is idling during CPU cycles "6" and "7" as it waits for the memory unit to finish loading the two new operands into registers.[2] We find that the ALU is periodically idle as the memory unit needs to be loaded with two new values for each MAC computation.

2) Proposed solution. To avoid consecutive memory loads, we propose to maintain the value of a register, *e.g.*, `q0`, and reorder the subsequent computations that require the value in `q0`. In this way, we only need to load one new value by the memory unit in the next period of time, so that the memory unit and the ALU can work alternatively without the ALU's idling waiting, as shown in Figure 9(b), where two MACs complete in seven CPU cycles vs. nine in Figure 9(a).[3]

In DTMM, we observe that the above idea applies to convolutions for the following reason. The default order of computation is to fix a position in the input feature map and iterate the dot product for all filters. However, the values in the feature map change every time because each unpruned filterlet is in a different position in its filter. Thus, each MAC requires two memory loads, similar to Figure 9 (a).

However, if we adjust the order of computation by fixing the filterlet weights in a register (*e.g.*, `q0`), we can reuse their index information to alternately load values (at the same

[2]We introduce a new register `q2` in Figure 9(a) to improve execution efficiency. Executing our code in Figure 8 requires only two registers, but the next two new values are loaded in cycle "6" (to `q0`) and "8" (to `q1`), as the ALU is still using them in cycle "5", causing the ALU to wait three cycles before the next calculation.

[3]We use two registers `q1` and `q2` that alternately load new values to maximize the efficiency of the ALU.

```
 1:     wlstp.8          lr, %[cnt], 1f
 2: loopStart:
 3:     vldrb.8          q0, [%[arr]], #16// Load weights
 4:     vldrb.8          q1, [%[in1]], #16// Load feature values
 5:     vmladava.s8      %[out], q0, q1   // MAC operation
 6:     vldrb.8          q2, [%[in2]], #16// Load feature values
 7:     vmladava.s8      %[out], q0, q2   // MAC operation
 8:     vldrb.8          q1, [%[in3]], #16
 9:     vmladava.s8      %[out], q0, q1
10:     vldrb.8          q2, [%[in4]], #16
11:     vmladava.s8      %[out], q0, q2
12:     letp             lr, loopStart
```

Fig. 10. Improved convolution operator design using SIMD and instruction-level acceleration on Armv8.1-M.

position but from different patches of input feature maps) into `q1` or `q2` and perform MAC (for `q0` and `q1`/`q2`). Thus, only one memory load is required per MAC before all computations requiring the current filterlet weights are finished, similar to Figure 9(b). Note that the reordering only changes the computation sequence, while the result is unchanged.

We incorporate this instruction-level acceleration to upgrade our convolution operator design in Figure 10. Benefits come from two aspects. First, it avoids unnecessary idle states to improve CPU utilization. Second, it also avoids the redundant load of the same unpruned filterlet weights as they are used multiply times in the convolution. As shown in Figure 6(b), this design can reduce the inference latency by an average of 84.61% compared to unstructured pruning.

3) Summary. The key innovation of our acceleration design is not entirely in the use of SIMD and parallelism. The main challenge is that the chances of acceleration on MCUs without rich hardware features are very small. Therefore, we perform an in-depth analysis of convolutions to find a valuable opportunity to reshape their computational flow to fit the model structure and leverage (almost) the only advanced features (SIMD and parallelism) on MCUs to achieve acceleration.

C. Pruning Strategy Scheduler

A pruning strategy defines which pruning units should be removed from each layer. By making this strategy, one typical objective is to minimize the execution latency of the pruned model while ensuring that the accuracy is not compromised too much and model size becomes small enough.

1) Constraints of pruning strategy: Before we provide an overall formulation of the pruning strategy, we need to first derive the following essential performance metrics to be used.

1) Accuracy. To avoid the impracticality of frequent evaluating the accuracy of pruned models during solving the pruning strategy problem, we employ a method inspired by previous works [15], [31] on estimating the importance of existing pruning units. We capture the change of loss after pruning individual units, and require the overall loss change for the pruned model to be less than a given $\Delta\mathcal{L}_{max}$, *e.g.*, 0.001, to bypass deriving the accuracy of pruned models directly. Taylor expansion [36] is used to measure the change of $\Delta\mathcal{L}$:

$$\Delta\mathcal{L} \quad = \quad |\mathcal{L}(\mathcal{D}) - \mathcal{L}(\mathcal{D} \mid \mathbf{w} = \mathbf{0})| \approx \left| \frac{\partial \mathcal{L}(\mathcal{D})}{\partial \mathbf{w}} \mathbf{w} \right|, \quad (2)$$

where $\mathcal{D}$ is the training dataset and $\mathcal{L}(\mathcal{D} \mid \mathbf{w} = \mathbf{0})$ is the value of loss function when a set of weights $\mathbf{w}$ are pruned (set to zero). The gradient of the loss function to the weights can be obtained through the backpropagation algorithm.

Now we compute $\Delta\mathcal{L}$ for each filterlet according to Eq. (2) and use the error threshold $\Delta\mathcal{L}_{max}$ to obtain the filterlets that can be removed without exceeding $\Delta\mathcal{L}_{max}$. Then, we have the first constraint to ensure the model's performance after pruning:

$$\boldsymbol{C_{acc}}: \quad \Delta\mathcal{L}(s) \leq \Delta\mathcal{L}_{max}, \quad (3)$$

where s is the decision variable $\langle \alpha_1, \cdots, \alpha_i, \cdots, \alpha_L \rangle$ and each $\alpha_i \in [0, 1]$ specifies the percentage of filterlets to be removed from the i-th convolution layer.

2) Model size and memory consumption. Model size for flash memory includes the following two parts in DTMM:

- Amount of unpruned weights: $A_w = \sum_{i=1}^{L} \alpha_i \times |\mathbf{K}_i|$, where $|\mathbf{K}_i|$ is the filter size in the i-th convolution layer and L is the amount of convolution layers.
- Index overhead: $A_{idx} = m_0 \times \sum_{i=1}^{L} (1 - \alpha_i) \times N_i \times H_i \times W_i$, where m_0 is the bit width of each index (*e.g.*, 16 bits), and the size of `rPtr` and `size` in FWCS are omitted because they are very small.

With m-bit quantization, the model size is computed by:

$$\mathrm{Size}(s) \quad = \quad m \times A_w + A_{idx}. \quad (4)$$

The memory consumption of SRAM to execute ML models is dominated by intermediate feature maps during inference. For DTMM pruned models, the output feature map size for the i-th convolution layer is: $M_i = \alpha_i \times N_i \times FH_i \times FW_i \times m$, where FH_i and FW_i are the height and width of the output feature map for the i-th convolution layer. During inference, models are executed layer by layer, and the intermediate feature maps are released after execution. Therefore, the memory usage is determined by the maximum memory consumption between any two adjacent convolution layers:

$$\mathrm{Ram}(s) \quad = \quad max_i\{M_{i-1} + M_i\}, \quad i \in [1, L], \quad (5)$$

where M_0 is the size of the input data to the model.[4] Then, we have the following two memory constraints:

$$\boldsymbol{C_{mem}}: \quad \mathrm{Size}(s) \leq \mathrm{Mem}_f;\ \mathrm{Ram}(s) \leq \mathrm{Mem}_r, \quad (6)$$

where Mem_f and Mem_r are the constraints of flash memory and SRAM, respectively.

2) Pruning objective: Within the above constraints, we aim to find the strategy that leads to the smallest execution latency.

1) Execution latency per layer. We start with the execution latency T_i of the single convolution operation in convolution layer i, which consists of three components:

i) Feature fetching latency T_i^{ft}: the time used to fetch a patch of input feature values, and the volume is the same as a filter size. So, $T_i^{ft} = H_i \times W_i \times C_i \times t_{mem}$, where t_{mem} is the time to fetch one value, H_i and W_i are the height and width of the kernel, and C_i is the channel number in layer i.

ii) Computation latency T_i^{cm}: the time used to first index the corresponding input feature values based on FWCS, and then compute dot product via SIMD MAC for each unpruned filterlet. It can be calculated by $T_i^{cm} = N_i \times H_i \times D_i \times (1 -$

[4]For ML models with shortcuts, *e.g.*, ResNet, the memory usage can be estimated from the topology of their computation graph [29].

$\alpha_i) \times (t_{idx} + \lceil \frac{C_i}{l} \rceil \times t_{com})$, where l is the maximum lane number supported by SIMD MAC, N_i is the number of filters in convolution layer i, α_i is the percentage of filterlets to be pruned from this layer, t_{idx} is the time to index a feature value, and t_{com} is the time to perform a SIMD MAC.

iii) Post-procesing latency T_i^{ps}: the post-processing time, $T_i^{ps} = N_i \times t_{post}$, where t_{post} is the time of bias addition and quantization for each output value after the computation.

Therefore, we have $T_i = T_i^{ft} + T_i^{cm} + T_i^{ps}$, and the execution latency of the convolution layer i is $\hat{T}_i = T_i \times FH_i \times FW_i$, where FH_i and FW_i are the height and width of the output feature map of layer i. Therefore, we propose to perform a regression on the parameters used in the calculations of T_i^{ft}, T_i^{cm} and T_i^{ps} as follows:

$$f(N_i, H_i, W_i, C_i, \alpha_i) = \hat{T}_i. \tag{7}$$

So far, the parameters t_{mem}, t_{idx}, t_{com} and t_{post} of execution latency model $f(\cdot)$ are undetermined. We can then use the measured latency of different configurations of convolution layers and the percentage of filterlets to remove to train $f(\cdot)$ as a regression model. Based on our experiment, we find that utilizing 10 training samples can yield a low prediction error of 0.03, as measured by the mean squared error.

3) Pruning strategy. Finally, we formulate the search of the pruning strategy as an optimization problem to minimize the overall latency $\mathrm{Time}(s) = \sum_i \hat{T}_i$ over all the convolution layers. Given that convolution layers are the most time-consuming components, other layers (*e.g.*, pooling) have relatively small impact on the overall model latency, which is not considered in $\mathrm{Time(s)}$. Therefore, we have:

$$\min_{s} \quad \mathrm{Time}(s), \tag{8}$$

subjected to $\boldsymbol{C_{acc}}$ in Eq. (3) and $\boldsymbol{C_{mem}}$ in Eq. (6). After decision variable s is solved (*e.g.*, by simulated annealing (SA) solver), we obtain each α_i ($\in [0, 1]$), which specifies the percentage of filterlets to be removed from the i-th layer.

IV. Implementation

1) Hardware. We implement DTMM with the Cortex-M55 processor [2], the latest AI-capable processor with vector processing (*i.e.*, Helium SIMD extension [33]) for MCUs. The Cortex-M55 has eight 128-bit vector registers to store operands for SIMD instructions. Each vector register can be divided into lanes of width 8, 16 or 32 bits. For example, if each weight of a model is 8 bits (*i.e.*, 8-bit quantization), a SIMD instruction can process 16 (=128/8) weights simultaneously. Each Helium vector instruction requires two CPU cycles. After the data is partially loaded into a register in the first cycle, the ALU can immediately use it for computations.

2) Software. The software implementation of DTMM consists of two parts: offline toolchain and on-device runtime. We develop the offline toolchain using Python 3.6 and Tensorflow 2.6, and the on-device runtime using C and Helium SIMD extension in TensorFlow Lite Micro (TFLM) [9].

Offline toolchain. The offline toolchain includes the scheduler that determines the pruning strategy, the pruner that performs filterlet-based pruning, and other modules such as quantization and retraining. To make our storage structure FWCS used by the models after DTMM pruning compatible with TFLM, we further update TFLM's `schema.fbs` file, a format library (including metadata and runtime tensors), and add FWCS as a new format member.

On-device runtime. We implement an on-device runtime based on TFLM. Our convolution operator is developed using C and inline Helium assembly. All these updates to TFLM do not affect its execution on other ML models. In particular, our convolution operator is launched only if a layer is pruned by DTMM; Otherwise, the traditional one is used.

TABLE II
Particulars of each unpruned model with 8-bit quantization.

Dataset	Model	Accuracy (%)	Model Size (KB)	Memory (KB)
CIFAR-10 (Image Classification)	VGG-11	90.19	1921	64
	ResNet-12	90.15	1248	96
VWW (Visual Wake Words)	VGG-11	86.77	1919	256
	ResNet-12	85.23	1246	384
FDDB (Object Detection)	YOLO	60.29	2071	392

3) Models and datasets. We evaluate DTMM with three ML models widely adopted in the previous studies on MCUs, including VGG-11 [42], ResNet-12 [16] and YOLO [39], in three popular ML tasks: image classification, visual wake words, and objection detection. For the classification tasks of image classification and visual wake words, we use VGG-11 and ResNet-12. For the object detection task, we use YOLO.

We train these ML models on the following datasets, which were used in the previous TinyML studies [26], [30]:

- **1) CIFAR-10** [25]: for image classification with 60 K of 32×32 RGB images of 10 different classes.
- **2) Visual Wake Words (VWW)** [8]: for identifying the existence of person in an image. Images are resized to $64 \times 64 \times 3$ for more efficient execution on MCUs.
- **3) Face Detection Dataset Benchmark (FDDB)** [22]: for detecting the location and size of all faces in the image. Images are resized to $112 \times 112 \times 3$ for MCUs.

Table II summarizes the accuracy, model size, and runtime memory for each unpruned model with 8-bit quantization.

4) Training and pruning. We use the above three datasets to train a corresponding version of each model and then perform pruning, 8-bit quantization, and fine-turning on each version of the model before execution. For CIFAR-10, we follow the official training and testing division (5:1). We split the other two datasets (VWW and FDDB) into 70% and 30% for training/fine-tuning and testing, respectively.

V. Evaluation

A. Overall performance

Methods. We compare the following pruning methods:

1) CHIP [44]: a state-of-the-art structured pruning that detects and removes unimportant filters using the channel independence of the feature maps generated by each filter.

2) PatDNN [37]: a state-of-the-art unstructured pruning method. It includes an inference engine specifically for mobile

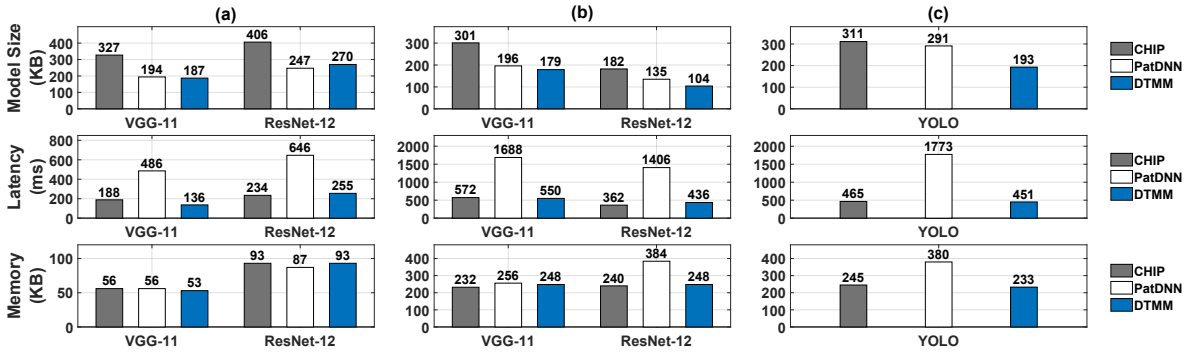

Fig. 11. Overall performance of (a) VGG-11 and ResNet-12 models on CIFAR-10, (b) VGG-11 and ResNet-12 models on VWW, and (c) YOLO model on FDDB. The latency is measured when the processor operates at 120 MHz.

devices. Since this engine is not MCU-compatible due to its reliance on the high-level parallel computing framework OpenCL, it is not included in the evaluation, and we implement a sparse convolution operator in TFLM for its inference.

3) DTMM: the method proposed in this paper.

Performance comparison. In this experiment, we compare the model size, execution latency, and runtime memory consumption of each model by maintaining model accuracy during pruning, *i.e.*, with accuracy loss less than 0.5%. For the object detection task, accuracy is measured using the average precision (AP). The resource constraints for flash and SRAM are 512 KB and 256 KB, respectively.

1) Model size. Figure 11 first shows the size of each model pruned by different methods. The unstructure-based PatDNN obtains a smaller model size than the structure-based method CHIP. DTMM can further reduce the model size due to its efficient storage design, outperforming CHIP and PatDNN by 39.53% and 11.92% on average, respectively.

2) Latency. The second row in Figure 11 shows the latency of each model after pruning. Although PatDNN can achieve smaller mode sizes, its pruned models execute much slower than CHIP due to the complexity of handling discrete weights by the CSR structure during inference. Unlike PatDNN, DTMM also achieves small latency. Overall, the latency performance of DTMM outperforms CHIP and PatDNN by an average of 1.09% and 68.70%, respectively.

3) Runtime memory. The runtime memory should fit within the SRAM of the device, and the runtime memory limit is set to 256KB in the evaluation. Both DTMM and CHIP can satisfy this constraint, but PatDNN may violate it in some cases due to the high indexing overhead.

Accuracy as model size decreases. We prune more weights for each method and examine the resulting accuracy change of each pruned model. To this end, we relax the accuracy loss requirement in our pruning strategy scheduler to prune each model for a smaller size. We also prune each model to the same smaller size for other two methods. Due to the page limit, Figure 12 (a–c) only plots the results of VGG-11 and ResNet-12 on CIFAR-10 and YOLO on FDDB. We can see that DTMM achieved the highest accuracy of pruned models than other two pruning methods (with similar model sizes) in most cases. The results are similar on other datasets.

Analyze pruned models. We first analyze the percentage of weights pruned on each layer. Figure 12(d) shows the result of each layer of VGG-11 on CIFAR-10, *e.g.*, 56.9% of the weights are pruned from layer "L1". With DTMM, weights can be pruned selectively from each layer. Overall, 37.5–99.0% of the weights from these layers are pruned.

We then analyze the components of each model pruned by DTMM. As shown in Figure 13, 77.2–86.3% of the model size after DTMM pruning is the weights. Indexing (and storage) overhead and other overhead (*e.g.*, quantization parameters and metadata) are only 1.1–3.2% and 12.6–13.9%, respectively, which shows the effectiveness of our FWCS structure design. In contrast, for similar model sizes, the useful weights of each model pruned by PatDNN only count as 27.9–28.4%, whereas its indexing and storage overhead is large, which explains why the model pruned by PatDNN loses more accuracy.

B. Micro-benchmarks

Ablation Study. We first conduct an ablation study to examine the efficacy of two designs in our convolution operator. To this end, we develop two intermediate versions of DTMM:

- **"DTMM-w/o-ins"**: it removes instruction-level acceleration from our convolution operator.
- **"DTMM-w/o-ins-vec"**: it completely disables our convolution operator and uses scalar instructions instead.

Figure 14 (a) shows that without instruction-level acceleration, "DTMM-w/o-ins" increases model execution latency by 20.8–55.3%. Also, our original convolution operator design plays a more important role, causing 231.8–635.0% increased latency if it is disabled. This experiment shows the effectiveness of our proposed techniques in speeding up the execution of DTMM pruned models.

Lane number of SIMD instructions. This is an important factor affecting the efficiency of our convolution operator design, as it affects how many dot products in convolution can be performed in parallel. In Figure 14 (b), we measure the latency of executing five DTMM pruned layers of different sizes by decreasing the number of lanes from 16 to 4. The results show that latency increases by an average of 16.2% and 48.9% when the number of lanes is reduced to 8 and 4, respectively. In Figure 14 (b), we use convolution layers instead of the whole

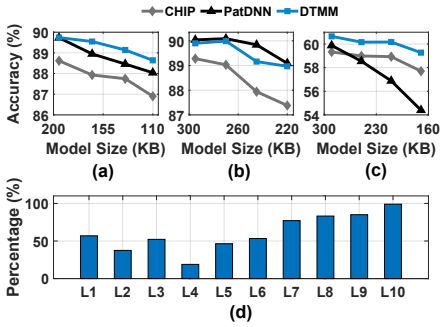

Fig. 12. Accuracy when (a) VGG-11, (b) ResNet-12, and (c) YOLO are pruned to smaller sizes. (d) Percentages of VGG-11 weights pruned.

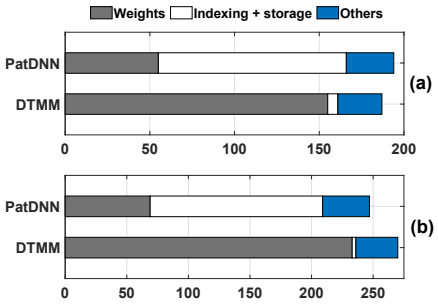

Fig. 13. Model size breakdowns for (a) VGG-11 and (b) ResNet-12 when they are pruned by PatDNN and DTMM on the CIFAR-10 dataset.

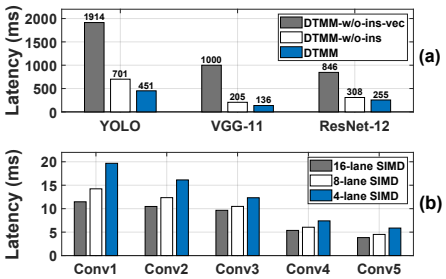

Fig. 14. (a) Ablation study with three DTMM versions. (b) Inference latency of convolution layers when the number of SIMD lanes changes.

model, since the whole model may contain unpruned layer(s) that are not executed by our operator.

Driven by the success of artificial intelligence, MCU processor manufactures begin to introduce SIMD with more lanes in their products. For example, the latest Cortex-M55 supports 16-lane SIMD MAC, while the second latest high-performance model Cortex-M7 released in 2014 only supports 2-lane SIMD MAC. For advanced processors with more lanes, DTMM can obtain higher speedup gains in the future.

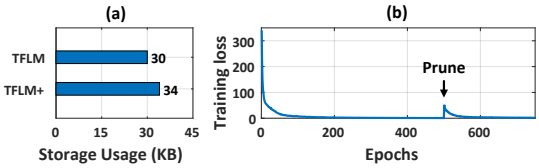

Fig. 15. Overhead of (a) storage and (b) training time.

C. System Overhead

Storage overhead. When we include only necessary operators for the ML models used in the experiments, the size of the original Tensorflow Lite Micro (TFLM) is 30 KB. In DTMM, we further integrate our new operator, and its storage overhead is small, *e.g.*, the size of TFLM with DTMM (TFLM+) only increases by 4 KB, as shown in Figure 15(a).

Training time. Finally, we study the training behavior of the model before and after pruning. In Figure 15(b), we train the original YOLO for 500 epochs. We then use DTMM to prune and fine-tune the pruned model, which takes about half of the initial training time to converge again.

VI. Related Works

ML models on MCUs. Deploying ML models on MCUs leads to an emerging field TinyML [3], [9], [48] for useful applications, such as autonomous nano drone [10] and smart health bracelet [13]. To support efficient model inference on MCUs, CMSIS-NN [27], microTVM [7], TinyEngine [30] and TensorFlow Lite Micro [9] are proposed. Based on these commercial frameworks, recent works improve the performance of ML models on MCUs by recording operator execution to reduce memory consumption [29], intermittent model execution [14], model architecture search [11], etc. DTMM is a systematic solution to enable efficient local inference on MCUs.

Model compression. There are existing studies to compress ML models on MCUs including knowledge distillation [18], low-rank factorization [5], and quantization [15]. They are orthogonal to DTMM and can be used simultaneously. For example, 8-bit quantization is currently used in DTMM (§IV).

Another popular model compression technique on MCUs is pruning [15], [44], which can be divided into structured and unstructured methods. Structured pruning removes model weights according to a given structure [6], [17], [28], [49], which might significantly reduce the model accuracy when the compression ratio is high on the MCU. Unstructured pruning [15], [37] can retain the accuracy. However, models after pruning perform slowly due to the large overhead [32]. Some research [34] proposes to group weights to form new pruning units, which can have different forms by applying different constraints [37], [49]. Unlike existing works above that focus on the structural design of pruning units, DTMM can actually run ML models with discrete weights after pruning on MCUs.

Other methods. In addition to model compression, there have been works proposed to deploy ML models on MCUs from other perspectives [26], [35], where computation offloading is a typical example [19]. However, offloading relies on extra infrastructure for support, such as edge or cloud, which is not conductive to large-scale deployment. In DTMM, we overcome the resource constraints of weak IoT devices to achieve fully autonomous operation and local inference.

VII. Conclusion

This paper presents DTMM, a specialized library for deploying TinyML models on low-end IoT devices like MCUs with pruning. We choose a suitable pruning unit and propose a dedicated storage structure to achieve high compression ratios while maintaining model accuracy. We also design a new operator, compatible with commercial ML frameworks, to efficiently execute DTMM pruned models, co-designed with a scheduler to derive the optimal pruning strategy. Evaluations show remarkable gains compared to state-of-the-art methods.

Acknowledgement

This work is supported by the GRF grant (CityU 11202623) from Hong Kong RGC. Corresponding author is Zhenjiang Li.

REFERENCES

[1] Advanced ai embedded systems — nvidia jetson. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/.

[2] ARM Ltd. Arm® Corstone™ SSE-300 example subsystem technical reference manual. https://developer.arm.com/documentation/101773/latest, 2021.

[3] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. In *Proc. of MLSys*, 2021.

[4] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[5] S. Bhattacharya and N. D. Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proc. of ACM SenSys*, 2016.

[6] S. Chen and Q. Zhao. Shallowing deep networks: Layer-wise pruning based on feature representations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[7] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proc. of USENIX OSDI*, 2018.

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

[9] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden. Tensorflow lite micro: Embedded machine learning on tinyml systems. In *Proc. of MLSys*, 2021.

[10] B. P. Duisterhof, S. Krishnan, J. J. Cruz, C. R. Banbury, W. Fu, A. Faust, G. C. de Croon, and V. J. Reddi. Learning to seek: Autonomous source seeking with deep reinforcement learning onboard a nano drone microcontroller. *arXiv preprint arXiv:1909.11236*, 2019.

[11] I. Fedorov, R. P. Adams, M. Mattina, and P. Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. In *Proc. of ACM NeurIPS*, 2019.

[12] F. Fraternali, B. Balaji, D. Sengupta, D. Hong, and R. K. Gupta. Ember: energy management of batteryless event detection sensors with deep reinforcement learning. In *Proc. of ACM SenSys*, 2020.

[13] B. Fyntanidou, M. Zouka, A. Apostolopoulou, P. D. Bamidis, A. Billis, K. Mitsopoulos, P. Angelidis, and A. Fourlis. Iot-based smart triage of covid-19 suspicious cases in the emergency department. In *Proc. of IEEE GLOBECOM Workshop*, 2020.

[14] G. Gobieski, B. Lucia, and N. Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proc. of ACM ASPLOS*, 2019.

[15] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *Proc. of ICLR*, 2016.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*, 2016.

[17] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *Proc. of IEEE ICCV*, 2017.

[18] G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[19] K. Huang and W. Gao. Real-time neural network inference on extremely weak devices: agile offloading with explainable ai. In *Proc. of ACM MobiCom*, 2022.

[20] Q. Huang, Y. Mei, W. Wang, and Q. Zhang. Battery-free sensing platform for wearable devices: The synergy between two feet. In *Proc. of IEEE INFOCOM*, 2016.

[21] Y. Huang, X. Qiao, J. Tang, P. Ren, L. Liu, C. Pu, and J. Chen. Deepadapter: A collaborative deep learning framework for the mobile web using context-aware network pruning. In *Proc. of IEEE INFOCOM*, 2020.

[22] V. Jain and E. Learned-Miller. Fddb: A benchmark for face detection in unconstrained settings. Technical Report UM-CS-2010-009, University of Massachusetts, Amherst, 2010.

[23] P. P. Jayaraman, A. Yavari, D. Georgakopoulos, A. Morshed, and A. Zaslavsky. Internet of things platform for smart farming: Experiences and lessons learnt. *Sensors*, 2016.

[24] F. Jia, D. Zhang, T. Cao, S. Jiang, Y. Liu, J. Ren, and Y. Zhang. CoDL: efficient cpu-gpu co-execution for deep learning inference on mobile devices. In *Proc. of ACM MobiSys*, 2022.

[25] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. *Technical Report*, 2009.

[26] Y. D. Kwon, J. Chauhan, and C. Mascolo. YONO: Modeling multiple heterogeneous neural networks on microcontrollers. In *Proc. of IEEE/ACM IPSN*, 2022.

[27] L. Lai, N. Suda, and V. Chandra. CMSIS-NN: Efficient neural network kernels for Arm Cortex-M CPUs. *arXiv preprint arXiv:1801.06601*, 2018.

[28] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. In *Proc. of ICLR*, 2017.

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

[30] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, et al. Mcunet: Tiny deep learning on iot devices. In *Proc. of ACM NeurIPS*, 2020.

[31] T. Lin, S. U. Stich, L. Barba, D. Dmitriev, and M. Jaggi. Dynamic model pruning with feedback. In *Proc. of ICLR*, 2020.

[32] X. Ma, S. Lin, S. Ye, Z. He, L. Zhang, G. Yuan, S. H. Tan, Z. Li, D. Fan, X. Qian, et al. Non-structured dnn weight pruning–is it beneficial in any platform? *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[33] J. Marsh. Arm® Helium™ technology M-Profile Vector Extension (MVE) for Arm® Cortex®-M processors: Reference book, 2020.

[34] F. Meng, H. Cheng, K. Li, H. Luo, X. Guo, G. Lu, and X. Sun. Pruning filter in filter. In *Proc. of ACM NeurIPS*, 2020.

[35] H. Miao and F. X. Lin. Enabling large neural networks on tiny microcontrollers with swapping. *arXiv preprint arXiv:2101.08744*, 2021.

[36] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. In *Proc. of ICLR*, 2017.

[37] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proc. of the ACM ASPLOS*, 2020.

[38] R. Pi. Raspberry pi 3 model b. https://www.raspberrypi.org, 2015.

[39] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proc. of IEEE CVPR*, 2016.

[40] A. N. Roshan, B. Gokulapriyan, C. Siddarth, and P. Kokil. Adaptive traffic control with tinyml. In *Proc. of IEEE WiSPNET*, 2021.

[41] G. E. Santagati and T. Melodia. An implantable low-power ultrasonic platform for the internet of medical things. In *Proc. of IEEE INFOCOM*, 2017.

[42] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[43] STMicroelectronics. Stm32 high performance mcus. https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html, 2019.

[44] Y. Sui, M. Yin, Y. Xie, H. Phan, S. Aliari Zonouz, and B. Yuan. Chip: Channel independence-based pruning for compact neural networks. In *Proc. of NeurIPS*, 2021.

[45] A. Varshney, W. Yan, and P. Dutta. Judo: addressing the energy asymmetry of wireless embedded systems through tunnel diode based wireless transmitters. In *Proc. of ACM Mobisys*, 2022.

[46] M. Wang, S. Ding, T. Cao, Y. Liu, and F. Xu. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *Proc. of ACM MobiCom*, 2021.

[47] Y. Wang, J. Shen, and Y. Zheng. Push the limit of acoustic gesture recognition. In *Proc. of IEEE INFOCOM*, 2020.

[48] P. Warden and D. Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

[49] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *Proc. of ACM NIPS*, 2016.

[50] H. Xu, P. Zhou, R. Tan, M. Li, and G. Shen. Limu-bert: Unleashing the potential of unlabeled data for imu sensing applications. In *Proc. of ACM SenSys*, 2021.

[51] X. Zhang, A. Andreyev, C. Zumpf, M. C. Negri, S. Guha, and M. Ghosh. Thoreau: A subterranean wireless sensing network for agriculture and the environment. In *Proc. of IEEE INFOCOM Workshops*, 2017.

[52] Y. Zhang, N. Suda, L. Lai, and V. Chandra. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*, 2017.