Enabling Efficient Deep Learning on MCU With Transient Redundancy Elimination

Jiesong Liu, Feng Zhang, *Member, IEEE*, Jiawei Guan, Hsin-Hsuan Sung, Xiaoguang Guo, Saiqin Long, Xiaoyong Du, *Member, IEEE*, and Xipeng Shen, *Senior Member, IEEE*

***Abstract*—Deploying deep neural networks (DNNs) with satisfactory performance in resource-constrained environments is challenging. This is especially true of microcontrollers due to their tight space and computational capabilities. However, there is a growing demand for DNNs on microcontrollers, as executing large DNNs on microcontrollers is critical to reducing energy consumption, increasing performance efficiency, and eliminating privacy concerns. This paper presents a novel and systematic data redundancy elimination method to implement efficient DNNs on microcontrollers through innovations in computation and space optimization. By making the optimization itself a trainable component in the target neural networks, this method maximizes performance benefits while keeping the DNN accuracy stable. Experiments are performed on two microcontroller boards with three popular DNNs, namely CifarNet, ZfNet and SqueezeNet. Experiments show that this solution eliminates more than 96% of computations in DNNs and makes them fit well on microcontrollers, yielding 3.4-5× speedup with little loss of accuracy.**

***Index Terms*—MCU, deep learning, compiler optimization, edge AI, heterogeneous systems, approximate computing.**

I. Introduction

FROM servers to the edge, and now to microcontroller-based devices, deep neural networks (DNNs) are in high demand. Microcontrollers dominate the computing engine market for small, low-cost or energy-efficient devices [1], [2]. Microcontrollers exist everywhere, from household appliances to cars, consumer electronics, wearables and so on [3]. It is estimated that 250 billion microcontrollers are already in use [4]. Microcontrollers are very resource-constrained. For example, STM32 F469I contains only 324KB SRAM and 2048KB flash on-chip memory. As a result, they have been widely regarded viable only for simple applications (e.g., keyword spotting [5]), rather than complex DNN models.

Received 24 August 2023; revised 29 May 2024; accepted 23 July 2024. Date of publication 23 August 2024; date of current version 8 November 2024. Recommended for acceptance by R. Wang. *(Corresponding author: Feng Zhang.)*

Jiesong Liu is with the North Carolina State University, NC 27695 USA, and also with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, Beijing 100872, China (e-mail: jliu93@ncsu.edu).

Feng Zhang, Jiawei Guan, Xiaoguang Guo, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, Beijing 100872, China (e-mail: liujiesong@ruc.edu.cn; fengzhang@ruc.edu.cn; guanjw@ruc.edu.cn; xiaoguangguo@ruc.edu.cn; duyong@ruc.edu.cn).

Saiqin Long is with the College of Information Science and Technology, Jinan University, Guangzhou 510632, China (e-mail: saiqinlong@jnu.edu.cn).

Hsin-Hsuan Sung and Xipeng Shen are with the Computer Science Department, North Carolina State University, NC 27695 USA (e-mail: hsung2@ncsu.edu; xshen5@ncsu.edu).

Digital Object Identifier 10.1109/TC.2024.3449102

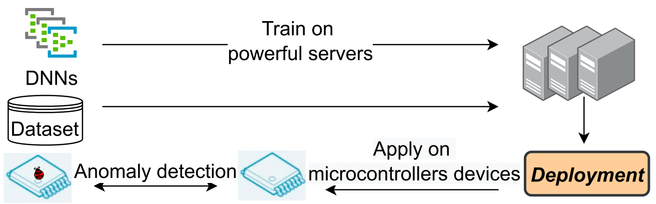


Fig. 1. Use case of anomaly detection.

However, the demands for DNNs on microcontrollers [6], [7] keep growing, for three reasons. First, making DNNs runnable on microcontrollers can expand the range of AI-powered applications on more devices. Second, executing large DNNs locally on microcontrollers is essential for reducing energy consumption while increasing performance efficiency. By eliminating the need for streaming from edge to the cloud, application latency can also be reduced and avoiding network congestion issues [8]. Finally, such designs eliminate many privacy concerns, as all user data is processed locally [9].

The efficiency of DNNs is important and is a central topic in the community [10]. We show a use case of anomaly detection in Fig. 1, which applies microcontrollers on a servo motor to retrieve vibration data [11]. With the help of microcontrollers, anomalies can be detected and reported in real time. In this particular case, the detection accuracy of 0.67 and a latency of over 1 second are improved to 0.82 and 317ms when DNNs are moved from cloud to the device end [12]. In the same vein, many other tasks can benefit from on-device AI, ranging from exoskeletons [13], to voice activation [14], object detection [15], and so forth.

There are some prior efforts trying to enable DNNs on microcontrollers, by creating light-weight neural networks [16] and other designs [17]. However, the available DNNs that can be executed on microcontrollers are still very limited, and those that can run are often subject to long latency [18].

Challenges to efficient DNN on microcontrollers lie in how to mitigate the tensions between performance and limited resources, shown in three aspects: (i) how to minimize the long execution time of DNN inferences under limited computing resources; (ii) how to host complicated DNNs in the limited memory including its inputs, outputs, weights, activation maps, and intermediate results; (iii) how to establish an optimal setting for minimizing the accuracy loss when optimizing the DNNs.

In this work, we propose an approach to eliminate the transient redundancy in DNNs to substantially improve the state of the arts on microcontrollers. The key idea of transient redundancy elimination is to identify similar elements in the input data of a convolution layer and avoid repeating similar computations on the fly. This approach makes the reuse mechanism a trainable component of a DNN. Because input data changes across inferences, the redundancy that our approach exploits is transient, hence the name.

While the theoretical TREC (Transient Redundancy Elimination-based Convolution) inspired this work, we provide a deep examination of the challenges in making Transient Redundancy Elimination effective for space-constraint devices, and details the optimizations that enable this approach to run efficiently on Microcontrollers. Specifically, we have made progress in three ways. First, as the new operator introduces additional space overhead, for space efficiency, we design a kernel reuse technique to reduce the space for storing parameters in the newly-built network. Second, we embed a two-step stack for storing clustering ID in the redundancy elimination process and use a reversed index to help locate the entries in the stack. Third, with the systematic design for the new network, we manage to integrate our techniques into back-propagation and keep marginal accuracy loss compared to the conventional networks. Our initial work has been presented in the study [19], which proposes to use transient redundancy elimination on convolutional layers. Compared to the previous study [19], we provide a more general framework for transient redundancy elimination for the DNN family on microcontrollers.

We evaluate our solution by applying it to three popular DNN networks, namely CifarNet [20], ZfNet [21], and SqueezeNet [22], on two microcontroller models. Our experiments show that, by implementing our solution on microcontrollers, we are able to avoid over 96% computations on convolution layers, and achieve an average of 3.4-5$\times$ reduction in the overall network latency with no or marginal accuracy loss.

Several previous studies have exploited similarities in inputs for DNNs. They either rely on special hardware [23] or use random hashing vectors that cause unstable inference accuracy [24]. None of them target microcontrollers or deal with the stringent space limitation. To our best knowledge, the solution from this work is the first that welds similarity-based computation reuse into DNN in a space-efficient manner for microcontrollers.

The main contributions of our work are as follows:

- We reveal the challenges of incorporating transient redundancy elimination for reducing computations in DNNs running on microcontrollers.
- We introduce a set of optimizations to mitigate the space overhead incurred by transient redundancy elimination.

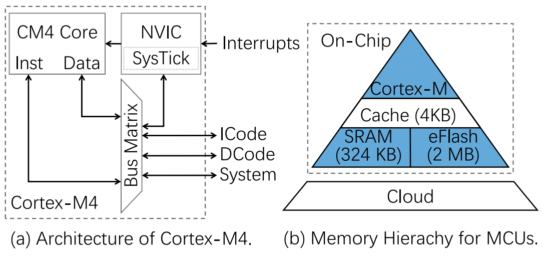


Fig. 2. An illustration of the architecture and memory hierarchy for microcontrollers. (a) Architecture of Cortex-M4. (b) Memory hierarchy for MCUs.

- We empirically evaluate the effectiveness of the new solution on two models of microcontrollers, confirming the substantial benefits of the new solution in enabling efficient DNNs on microcontrollers.

II. Background

In this section, we introduce the deployment of neural networks on microcontrollers, the work related with input reuse and work related with model compression.

Microcontrollers. A microcontroller unit (MCU) is an energy-efficient processor that is ubiquitous in our lives. We show an example of MCU architecture and its memory hierarchy (STM32F469I in our case) in Fig. 2. Fig. 2(a) shows that the Cortex-M4 core architecture consists of a 32-bit processor (CM4) and a small number of critical peripherals. The CM4 core is a Harvard-architecture, which means that it utilizes distinct interfaces to fetch instructions (Inst) and data (Data). This helps ensure that the CPU does not run out of memory, as it enables simultaneous access to the data and instruction memories. The special feature that distinguishes Cortex M4 from CM3 [25] is that CM4 includes, for the processor, single-instruction multiple-data (SIMD) extensions that are effective in achieving faster arithmetic computing performance in the CPU context.

Fig. 2(b) shows its on-chip memory hierarchy consisting of very limited memory space. We can see that microcontrollers are typically comprised of a central processing unit (CPU), cached memory for frequently accessed data, static random-access memory (SRAM), and an on-chip flash memory for storage.

Microcontrollers are especially energy efficient with low-power (0.166W for F469I board) and cost effective (around $10) compared to common processors like CPU and GPU [26]. Given this, microcontrollers provide very limited computing resources and a limited volume of storage (around 1 cm^3) that developers need to take care of. In addition to physical microcontrollers, cloud functions, which can be seen as virtual resource-constrained devices in the cloud or the edge, are also widely used for cost-efficient computation and data processing tasks [27]. Optimizations that improve the space efficiency on microcontrollers may apply to cloud-function-based applications.

Work on input reuse. Several studies have explored similarities in inputs for DNN accelerations. *Deep reuse* [24], [28],

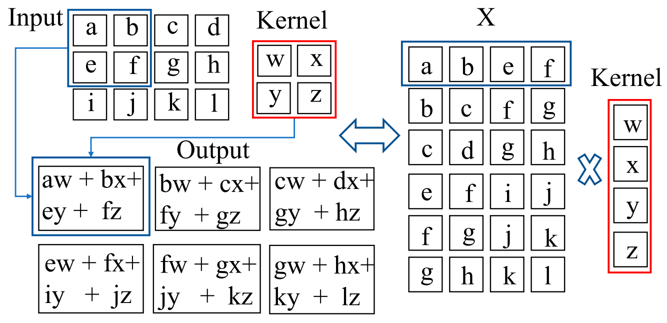


Fig. 3. An illustration of convolution in *deep reuse*.

[29], [30] is a pure software approach. It inspired this current work. *Deep reuse* deals with convolution as a General Matrix Multiplication (GEMM) in the form illustrated on the right side of Fig. 3. After transformation, each row in X is a neuron vector. *Deep reuse* finds and clusters similar neuron vectors as a group. It thus avoids repeating similar computations by reusing the computation results of the cluster centroid as the results of the rest of the neuron vectors in the same group. The work uses some random locality-sensitive hashing vectors for the online clustering. It does not consider space efficiency; it actually increases the space usage substantially due to the extra space needed to store hashing vectors and cluster indices. The use of random hashing vectors also cause large fluctuations in the inference accuracy (detailed in Section VIII).

Relations with model compression approaches. Many studies on DNN compression [31] have exploited the redundancy among DNN parameters, which is orthogonal to input-level reuse. Examples include quantization [32], squeezing filter size [33], conducting feature compression to activation map [34], and so on. These techniques can significantly reduce the model size (i.e., weights and biases), the amount of computations, and data required to stream from edge to the cloud. For resource-stringent microcontrollers, input reuse and model compression have to be applied at the same time as shown later (TREC design in Sections IV and V and model compression in Section VII) in this paper.

Locality Sensitive Hashing (LSH). LSH is an online clustering method used in multiple solutions (including TREC) for enabling computation reuse in DNN optimizations. As shown in Equation 1, for a parameter vector $\mathbf{v}$, the input vector $\mathbf{x}$ is transformed into 1 or 0 follow the hash function $h_{\mathbf{v}}$:

$$h_{\mathbf{v}}\left(\mathbf{x}\right) = \begin{cases} 1, & if \quad \mathbf{v}\cdot\mathbf{x} > 0 \\ 0, & if \quad \mathbf{v}\cdot\mathbf{x} \leq 0 \end{cases} \tag{1}$$

If H hash functions are used, for a given input vector, LSH maps it to a H-bit vector. Nearby input vectors often produce the same bit vector after hashing, and the number of hash functions, H, adjusts clustering roughness.

Each neuron vector then can be labeled an ID number according to the corresponding H-bit vector. And neuron vectors with the same ID can form a cluster and reuse the computing results from the centroid vector in place of the individual results for each vector.

III. Overview

Running large DNN inference on microcontrollers is especially challenging because of the strict resource constraints both in terms of computing capabilities and memory footprint. Removing computation redundancy and achieving space efficiency is thus essential for enabling efficient DNN deployment on microcontrollers.

Redundancy in DNNs can be categorized, based on where the redundancy originates, into *lasting redundancy* and *transient redundancy*. Lasting redundancy originates from the model parameters. As parameters are unchanged when applying inference, this kind of redundancy can be eliminated by methods such as pruning and quantization. Transient redundancy, however, exists in the form of similar tiles inside an input data or activation map.

There are many techniques that can be adopted for the elimination of lasting redundancy when running neural networks. For example, CMSIS-NN [2] applies quantization to the kernel weights and consequently avoids floating point computations for DNN networks, thus improving inference performance. However, ways of reducing transient redundancy on microcontrollers remain insufficiently explored.

To address this issue, we devise a principled way to efficiently detect and remove transient redundancy for DNNs on microcontrollers. It is based on Transient Redundancy Elimination-based Convolution (TREC), an idea of integrating reuse into a DNN as new kinds of DNN operators. This architecture has multiple benefits. First, by eliminating transient redundancy, TREC minimizes the computation volume, thus bringing the maximum computation-elimination benefits and improving the performance of the entire network. Second, TREC is compatible with both training and inference tasks, allowing for a simple plug-and-play replacement of convolutional layers in standard convolutional neural networks (CNNs) for training and inferencing. This approach guarantees highly robust model accuracy. Third, applying TREC in the model is orthogonal to methods targeting removal of lasting redundancy, which can bring further benefits when combined together. Section IV gives more details of the TREC architecture.

Adding the new operators into the DNN inference stage reduces the amount of required computations significantly, but also introduces space overhead. To achieve space efficiency, we introduce *two-step stack substitution* for the DNN network to minimize the space occupancy for the clustering containers. Furthermore, for extreme cases where efficient space usage is of paramount importance (e.g., microcontrollers), we propose the *kernel reuse* technique to further remove the overhead while still eliminating the transient redundancy when performing DNN inference. Take convolutional layers for example. Fig. 4 outlines the overall process of space-efficient TREC. Aside from convolutional neural networks, we also explore transient redundancy elimination for recurrent neural networks. We explain space efficient TREC in more details first, followed by transient redundancy elimination for RNNs in Section VI.

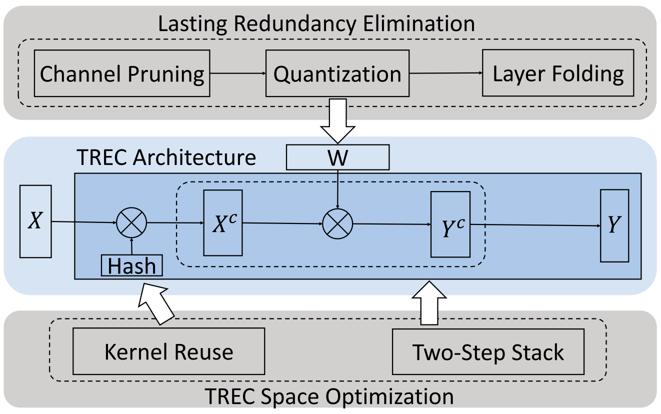


Fig. 4. Overview of space-efficient TREC.

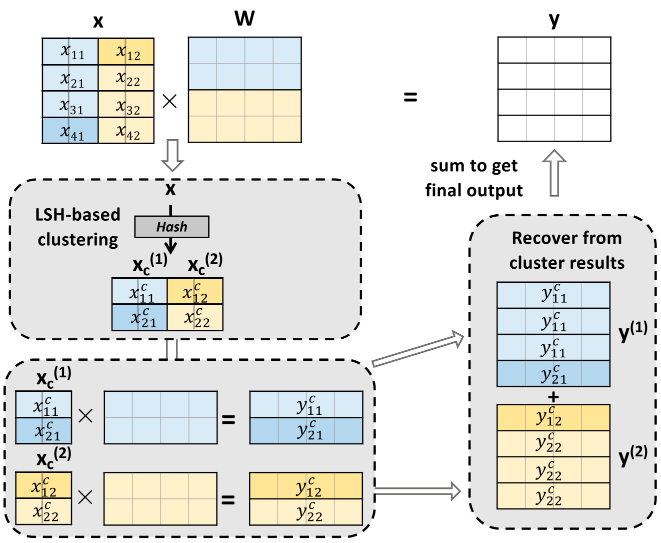


Fig. 5. An illustration of TREC.

IV. TREC Architecture

To help understand the description on Space Efficient TREC, we first give a review of the basic architecture of TREC [35], its training and properties.

The main aim of TREC is grouping similar neuron vectors into clusters and then redirecting clustering results back to the DNN. To achieve this objective, it fuses the detection and avoidance of transient redundancy into DNN, making them part of its inherent architecture.

As illustrated in Fig. 5, the original GEMM convolution is $\mathbf{y} = \mathbf{x} \cdot \mathbf{W}$. The TREC operator can be decomposed into four steps. First, the input matrix $\mathbf{x}$ passes through a clustering component. At a high level, TREC uses Locality-Sensitive Hashing (LSH) to do the clustering. In this example, $\mathbf{x} \in \mathbb{R}^{4\times 4}$ is first sliced into two 4×2 sub-matrices, each of which has its own hash functions. And the 4 row vectors in each sub-matrix are grouped into two clusters. In the second step, for each $\mathbf{x}$, $\mathbf{x} \cdot \mathbf{W}$ is performed using the representative neuron vectors, allowing the matrix size for computing to be significantly reduced. In the third step, for each sub-matrix, the vectors in the same cluster use the computation results from the corresponding centroid neuron vector to recover the full-sized output matrix. At last, the final result is obtained by adding the results computed separately for each sub-matrix.

The main computation savings come from the clustering component. This is because in GEMM-based convolution, we should compute four results for each neuron vector. By using TREC, we use the result of the centroid computation in place of the per-row results for each of the row vector. Therefore, the number of vector-matrix computation reduces from four to two.

Benefits and the key conditions. By grouping neuron vectors into clusters, the size of the input matrix is significantly reduced, allowing for low computational complexity for the subsequent matrix multiplication. Consider that an input matrix X for GEMM (after im2col) is of $N \times K$ dimension and a weight matrix W is $K \times M$ size. The clustering step can be abstracted as applying a hash function matrix $Hash$ to X. Let $K \times H$ be the dimension of the hash matrix. We can assume that the neuron vectors can be grouped into N_c clusters. For each input image or activation map, the benefits for removing the *transient redundancy* can thus be measured by a *redundancy ratio*, i.e., $r_t = 1 - \frac{N_c}{N}$ as the reduction of input size. N is the total number of neuron vectors, and the number of centroid vectors is equal to the number of clusters, namely N_c. The number of required computations is thus reduced to N_c. We can conclude that r_t indicates the fraction of $transient\, redundancy$ within input images or activation maps.

Using this measure, the total Floating-point Operations (Ops) for a conventional GEMM-based convolution is $N \cdot K \cdot M$, while the Ops for TREC will be $\left(\frac{H}{M} + 1 - r_t\right) \cdot N \cdot K \cdot M$, since TREC also involves a $Hash$ matrix multiplication. Therefore, in order for TREC to expedite DNN inference (i.e., $\left(\frac{H}{M} + 1 - r_t\right) \cdot N \cdot K \cdot M < N \cdot K \cdot M$), the following *key condition* must hold: $\frac{H}{M} < r_t$. As shown in the experiments, the average transient redundancy elimination benefits r_t exceeds 96%.

Comparison with *deep reuse*. Prior studies like *deep reuse* have treated *transient redundancy* in an ad-hoc manner. *Deep reuse* takes place as extra operations outside DNN, using LSH with random hashing vectors for online data clustering. Such ad-hoc treatment causes severe uncertainty about the impact of the clustering errors on the DNN performance. Experimental results show that *deep reuse* causes significant (e.g., 5%) fluctuations on DNN accuracy.

TREC overcomes the limitations of *deep reuse*. The newly introduced DNN operator TREC incorporates transient redundancy detection and avoidance directly into DNN's inner architecture, while *deep reuse* functions outside the DNN. As a result, TREC achieves a consistent inference performance and accuracy.

Training TREC. So far we have discussed how TREC can benefit from reusing the centroid results. We have also shown how TREC performs efficient convolutions in the following stages: 1) clustering: grouping similar neuron vectors through hash functions, 2) computing and reusing results from centroid

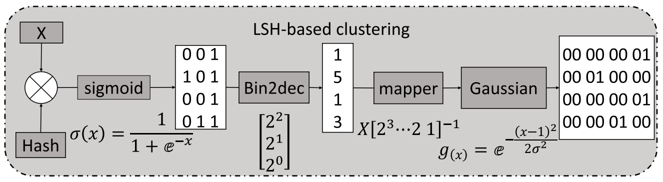


Fig. 6. Training design for TREC clustering component.

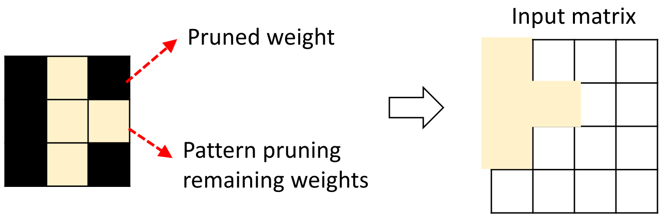


Fig. 7. Illustration of a sparse pattern kernel.

vectors in each cluster in place of the results of each vector in the cluster, and 3) recovering back to final results. Now, we explain how to jointly learn the hash functions and kernel weight parameters, $Hash$ and W. Direct learning of TREC leads to problems: 1) discrete mapping problem, where we need to map values to 0 or 1 according to their signs, and 2) combinatorial optimization problem, where we have to compute the centroids for each cluster. To get around this, we reformulate the clustering stage.

We propose the following workflow for clustering in Fig. 6 to make back-propagation work with the clustering in TREC. First, to solve the discrete mapping problem, after the input matrix $\mathbf{x}$ is multiplied by the *Hash* matrix, the projected matrix is subjected to an element-wise sigmoid functioning as a binary classifier. In line with the proposed design, each neuron vector is thus transformed into a H-bit vector. In the next step, the bit vector is converted into a cluster ID, akin to converting a numerical value from its binary representation to the decimal counterpart. We then apply the mapper matrix to the ID vector and, after applying a Gaussian function, we obtain a bitmap of the neuron vector where each column indicates whether this vector belongs to a corresponding cluster. This bitmap helps to transform the computation of centroids from a combinatorial problem to a matrix multiplication.

After obtaining centroids, we can multiply the centroids with the weights W and perform the rest of the computation step of TREC as in Fig. 4.

Properties of TREC.

- **Accuracy.** Reusing results from the centroid vector for the vectors in the whole cluster is driven by the similarities among the vectors in the cluster. The learning process in TREC helps minimize the approximation errors. If we train the neural network with *random* LSH vectors, i.e., without learning the *Hash* matrix in the back propagation, accuracy could drop sharply. The *learned Hash* matrix in TREC makes a big difference in improving the accuracy of the network, as our experiments in Section VIII will show. The lesson is that the *Hash* matrix is learned and updated in a way that it resonates with the clustering process. In a sense, it chooses the optimal LSH vectors that are able to detect the similarities between vectors, and it helps group the similar vectors into the same cluster.
- **Robustness.** Traditional GEMM-based CNNs are robust in that they possess stability for small input perturbation. In order to assess the robustness of a neural network, we apply the *Lipschitz constant* L to analyze TREC. L gives the relations between the input perturbation ϵ and the output variations δ, namely $\delta \leq L\epsilon$. The *Lipschitz constant* for TREC is rigorously proved to be bounded by the GEMM-based Convolution, namely, $L(TREC) \leq L(Conv)$. Therefore, when the input image suffers from small perturbation, according to the properties of *Lipschitz continuous* of TREC, the output variation, denoted as δ_{TREC}, is bounded by the L_∞-norm of the weight matrix W, i.e., $\delta_{TREC} \leq L(TREC)\,\epsilon \leq L(Conv)\,\epsilon \leq \|W\|_\infty \epsilon$. This shows that TREC is robust to input perturbation.
- **Convergence.** TREC-equipped DNNs converge under reasonable assumptions: (a) The objective function $F(W, H)$, in which W and H correspond to the weight matrix and *Hash* matrix, is continuously differentiable, and (b) $F(W, H)$'s partial derivatives ∇F are *Lipschitz continuous* where their first and second moments meet certain limits. Given the above assumptions, we are able to obtain the fact that the expected change in F between two iterations are bounded. Therefore, for stochastic gradient descent optimization strategy, the partial derivatives of the objective function F converge to zero after many iterations. This is true to both fixed and diminishing stepsizes and thus proves that we have found the locally optimal solution [35].
- **Compatible with Sparse Matrices**. TREC is applicable in the presence of sparse weights matrix or sparse input matrix, making it possible to be used together with other DNN optimizations (e.g., pruning). We next provide details for sparse input matrices.

Converting Sparse Convolution to Dense Convolution. TREC is compatible with sparse input matrices through *sparse convolution*. Although this property is not directly used in this work, it could be an important property to exploit when a user would like to combine TREC with DNN sparse pruning or other optimizations. We explain this property in this part to offer a complete understanding of TREC.

One typical case where sparse matrix shows up is when weight pruning is used. We will draw on PatDNN [36] as our example. PatDNN prunes a DNN kernel via patterns. The left graph in Fig. 7 shows a 4-element pattern left after a 3x3 kernel is pruned. In PatDNN, the sparse weight matrix is encoded in a Filter-Kernel-Weight (FKW) format [36]. When a convolution kernel slides through an image or activation map, only the pixels within the pattern's frame contribute to the output. In the example in Fig. 7, only four elements in the input are used

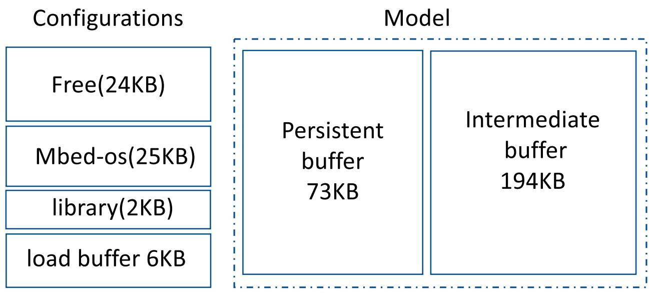


Fig. 8. Breakdown of SRAM memory for an original DNN without TREC on STM32F746ZG.

TABLE I
Space Overheads Introduced by TREC for the Three Networks and Their Ratio to the SRAM of Four Boards

	Network	CifarNet		ZfNet		SqueezeNet	
		Hash	Cluster	Hash	Cluster	Hash	Cluster
	Space overhead	20 KB		40 KB		56 KB	
	Breakdown	60.6%	39.4%	65.8%	34.2%	91.1%	8.9%
Space ratio	F205 (64KB)	31.3%		62.5%		87.5%	
	F303 (80KB)	25.0%		50.0%		70.0%	
	F446 (128KB)	15.6%		31.3%		43.8%	
	F746 (320KB)	6.3%		12.5%		17.5%	

in the dot-product with the kernel in each step. TREC can be applied by regarding the four elements used in each step as a vector for clustering. If two of such vectors are similar and fall into the same cluster, TREC avoids the repeated dot-product calculations.

We next look at how TREC can be applied if the input matrix is sparse. We discuss it in two scenarios. In the first scenario, the original algorithm simply treats the sparse input in the same way as it treats a dense input. Clearly, TREC can be applied as we have already described, and because there are many zero vectors, TREC would cluster them together and give significant speedups. In the second scenario, the original algorithm already tries to do calculations only on the non-zero parts of the input. *Sparse convolution reorganization* is one way to do that [37]. It groups non-zero elements together to form a dense matrix. This matrix is then multiplied with reorganized kernel matrices. In this way, it avoids the inefficiency of frequent irregular accesses. TREC can be applied to the matrix after the reorganization.

V. Achieving Space Efficiency

This section provides the techniques we propose that make TREC space-efficient, and explain how to make TREC-based DNN efficient on Microcontrollers.

A. Space Pressure For DNNs on Microcontrollers

In this section, we analyze the space pressure for DNNs on microcontrollers. Unlike desktop systems, microcontrollers have a rather flat memory system, as they are equipped with on-chip main memory only. To fit large DNNs into such a memory system, given the small size of microcontroller devices, we carefully analyze the space usage for the original network and the additional space required for TREC.

Fig. 8 provides a typical memory breakdown for neural networks on the STM32F746ZG SRAM. The runtime overhead for the mbed-os is fairly small, requiring just 25 KB on SRAM. Loading buffers are needed for image input and are allocated. As CifarNet is equipped with only 2 convolutional layers, the full weights and biases are stored and buffered on the persistent buffers, where pointers to the intermediate buffers are also stored. Activation maps are also allocated on SRAM as well.

Additional space for TREC. Adding the TREC operator to the DNN can minimize the required computations for the whole network, allowing for efficient inference performance. However, it brings additional space overhead. Specifically, an extra *Hash* matrix and containers for holding cluster information are needed. Since there are at most 2^H different clusters, the additional space overhead for clustering computing is $O(L \times 2^H)$, space required to be allocated on the *free* section in Fig. 8. Consider TREC operators in N layers, the total space overhead for the *Hash* matrix, however, is $\Sigma_{n=1}^{N} C_{i-1} \times k_i^2 \times H$, where k_i and C_i denotes the kernel size and the channel size of layer i, respectively.

Empirical analysis for space overhead. Space resource is very limited on microcontrollers. Specifically, SRAM is important as primary storage because direct computations are performed on it. Flash memory stores read-only data and uses SRAM to load data when doing computation. However, applying TREC can bring additional burden to SRAM spaces. Table I shows the SRAM size of four typical STM32 microcontrollers and the space ratio of TREC overheads to these boards. For each DNN network, *Hash* denotes the additional space required for the LSH matrix, while *Cluster* relates to the indexing and storing vectors for each cluster. For example, TREC requires an additional 16KB to conduct clustering operations and 41KB for hash tables for SqueezeNet.

Space limitations. Original networks have large model sizes and need *lasting redundancy* elimination to fit in the microcontroller board. For example, the sizes of ZfNet and SqueezeNet are 5MB, which are reduced to below MB after pruning. TREC affects the footprint of the pruned model. In fact, Table I shows that TREC space overheads are substantial compared to the total SRAM capacities on microcontrollers. Therefore, the space overhead introduced by TREC is non-negligible and hinders running DNN applications on microcontrollers, considering facets such as larger batch sizes.

B. TREC Space Optimization

Although only trivial KB-level space occupancy overhead is introduced, which is negligible in the context of cloud computing, this capacity requirement poses challenges in the materialization of TREC on microcontrollers. Achieving space-efficient TREC is hence crucial.

Conventional optimization. Conventional implementations for space savings include DNN channel pruning, quantization,

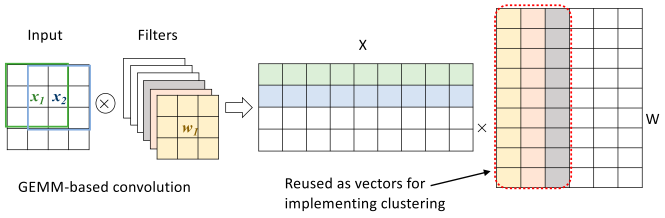


Fig. 9. A closer look at the weight matrix.

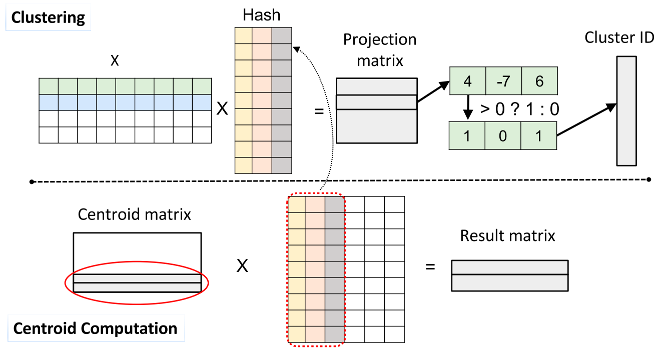


Fig. 10. Illustration of kernel reuse.

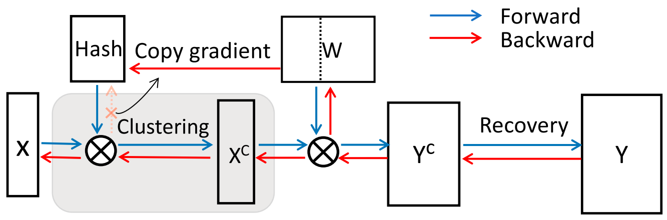


Fig. 11. Illustration of *kernel reuse* training.

and layer folding. For these techniques, the goal is to prune model parameters and reduce the required static space for storing the model structure and parameters. Therefore, these *offline* pruning methods rely on delicate detection and examination of unnecessary parameters in the network, aiming at removing *lasting* redundancy (i.e., redundancy existing in DNN parameters). These techniques are discussed as implementation details in Section VII-A. Another category of potential space savings lies in reducing the *online* dynamic execution space occupancy of TREC, namely space savings for the clustering component. Emphases are placed on the *Hash* matrix and the choice of data structure facilitating storing and indexing vectors.

TREC optimizations. In the following, we propose two techniques for achieving space-efficient TREC. The first idea is to implement hashing in a space efficient way by re-using parts of a matrix that is already part of the DNN computation as the collection of vectors required to implement clustering. Vectors making up this matrix thus serve "double duty" as hashing vectors and matrix elements. The technique avoids huge memory overheads and achieves high speedups on real DNN computations.

For the second design, we embed a two-step stack for storing clustering ID in TREC and use a reversed index to help locate the entries in the stack. We will go through these two techniques in further depth.

C. Kernel Reuse

As discussed in Section V-A, vanilla TREC increases the space usage due to extra space needed to store hashing vectors. We hereby propose *kernel reuse* as a technique to address the space issues for the *Hash* matrix. The idea is to implement hashing by reusing parts of the weight matrix as the collection of vectors required to conduct clustering.

A closer look at the weight matrix. For a GEMM convolution, each filter slides through the input matrix as illustrated in Fig. 9. There are six filters in total, and each filter is composed of one 3×3 kernel. We focus on the weight matrix W after *im2col*, where each filter is expanded to a column vector. Since there are six filters and nine elements in each filter, the weight matrix has a 9×6 size. Now, we consider the clustering step. Suppose we set $H = 3$, namely, the hashing comprises of three vectors for clustering. Immediately, we find it possible to take advantage of the first three columns of the weight matrix W and reuse them as the *Hash* matrix for clustering the rows of the X matrix. Specifically, an output matrix $Y^{n\times c}$ is obtained after $X^{n\times m}$ multiplies $W^{m\times c}$, thus the partial results from the first three columns, namely $Y^{n\times 3}$, serve double duty as matrix outputs and hashing outputs. This method avoids huge memory overheads and achieves time savings.

Detailed design. A detailed workflow of the proposed technique is illustrated in Fig. 10. First, we reuse parts of the weight matrix as *Hash* matrix. Next, we multiply X with *Hash* and obtain a projection matrix. Based on this result, we project each row vector to a cluster. Specifically, for each row, i.e., a neuron vector $\mathbf{x}$, we apply $h_{\mathbf{v}_i}(\mathbf{x})$ for all $\mathbf{v}_i, 0 \leq i < H$. In the end, the cluster ID of a neuron vector is encoded as a H bit integer $clusterId(\mathbf{x}) = (h_{\mathbf{v}_0}, h_{\mathbf{v}_1}, ...h_{\mathbf{v}_{H-1}})$. Vectors with the same cluster ID are grouped in the same cluster. Third, we compute a centroid vector for each cluster and compile them into a centroid matrix. Finally, we multiply the centroid matrix with the weight matrix W and obtain the result for each cluster.

Solving the dilemma for *kernel reuse* training. Training TREC with *kernel reuse* is one key challenge. *Kernel reuse* seeks ways to embed LSH vectors in the weight matrix. However, TREC requires separately updating *Hash* and the weight matrix W in each training iteration, which leads to inherent discrepancies between *Hash* and W. To get around this, we embed the LSH vectors in the weight matrix by binding *Hash* to the columns in W and sharing the gradients in each iteration. As illustrated in Fig. 11, in the back propagation, instead of updating LSH vectors and the weight matrix independently, the gradients for *Hash* are copied directly from the gradients of the corresponding positions in the weight matrix. The *Hash* matrix is thus embedded in the weight matrix after each iteration.

Maintaining properties of TREC. Another factor to note for building *kernel reuse* based TREC is to understand the impact of *kernel reuse* on the correctness, robustness, and convergentability of TREC.

Correctness. As stated above, *kernel reuse* introduces errors in clustering because gradients are directly copied from those of the weight matrix. This error is then passed on to the next forward propagation and got corrected in the back propagation. In this way, TREC is still able to exploit the similarities among the vectors in the cluster. Through training, *kernel reuse* minimizes errors while saving space.

Robustness and Convergence. Kernel reuse does not change the *Lipschitz constant* for TREC, and thus it keeps the robustness of TREC. The convergence of *kernel reuse* can be proved following the same steps as those of TREC by viewing the objective function F as a function of W, namely, $F(W)$.

Space savings. *Kernel Reuse* removes the space overhead from *Hash* matrix and greatly saves space. This space saving does not sacrifice the expressive functionality of LSH to cluster vectors. This is because LSH vectors are kept to be mutually independent even by reusing parts of the weight matrix. Since filters are trained independently in the weight matrix, columns of the weight matrix are mutually independent. This guarantees that LSH vectors are independent from each other.

D. Two-Step Stack Substitution

General design. Another part of space overhead of TREC comes from the data structures for indexing vector entries. The space constraints on microcontrollers require efficiently storing and accessing neuron vectors in different clusters while minimizing the space occupancy. Therefore, with the help of a reversed ID table and a stack, a representative vector is stored in each cluster.

Detailed design. Suppose now we have each vector's $cluster$ ID. The goal is to do clustering by grouping the vectors with the same ID together and finally get a centroid matrix containing the centroid vector for each cluster. However, storing vectors into groups according to cluster IDs and computing the centroids for each cluster is not space efficient. This is because pre-allocating fixed-size arrays for each cluster poses too much burden on the memory. To make it more space efficient, we form the centroid matrix in a streamlined way.

At any time, the space overhead is only a representative centroid vector for each cluster. These centroid vectors are stored in a stack with an index table for entry access. After finishing processing, these centroid vectors form a centroid matrix, and we can multiply it with the weight matrix. We then use the result of the centroid computation in place of the per-vector results for each vector in the cluster. A workflow sample is shown in Fig. 12.

Each neuron vector is projected to a cluster ID Id. When it is the first vector with cluster ID $= Id$, we add this vector in the buffer stack and store a pointer to it in the *reverse ID table*. In fact, the x-entry in the reverse ID table stores the pointers, if existed, to the buffer stack for the cluster with $Id = x$. When another vector $\mathbf{c}$ with the same cluster Id arrives, we can directly find the representative centroid vector $\mathbf{v}_{old}$ in the stack and add the weight of this vector to it. Formally, if $count_x$ denotes the number of vectors in cluster x, we have the

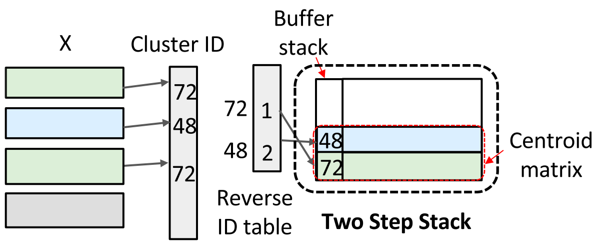


Fig. 12. Stack for the representative vectors.

new centroid vector

$$\mathbf{v}_{new} = \frac{\mathbf{v}_{old} \times count_x + \mathbf{c}}{count_x + 1}, count_x = count_x + 1 \quad (2)$$

There are at most 2^H (Id ranges from 0 to $2^H - 1$) clusters, thus at most 2^H representative vectors in the stack. Finally, the centroid vectors form a centroid matrix.

For each representative vector, we multiply it with the weight matrix to obtain a result vector, so that the vectors in the same cluster can reuse the result to speedup the computation. Note that the representative vectors stored in the stack actually form a matrix. Therefore, we only have to multiply the centroid matrix by the weight matrix, and the result vectors can be retrieved as the corresponding row in the result matrix.

VI. Transient Redundancy Elimination for RNNs

We further explore transient redundancy elimination on recurrent neural networks (RNNs). Different from convolutional neural networks, RNNs make use of recurrency in a network to incorporate temporal information for better prediction. In this section, we investigate how transient redundancy is tackled in RNNs.

A. Motivation

Recurrent Neural Networks (RNNs) have emerged as a crucial tool in the field of artificial intelligence and machine learning, playing a pivotal role in various domains [7], [38]. Their unique architecture enables them to efficiently process sequential and temporal data, making them invaluable in tasks such as natural language processing, speech recognition, machine translation, and time series analysis.

Transient redundancy in RNNs is an important issue that has not received extensive exploration in the research community. Just as in CNNs, transient redundancy in RNNs also arises from input data. Specifically, strong similarities may exist in input data from different time steps.

However, the intrinsic dependency across stages caused by the recurrent architecture poses a significant challenge for transient redundancy elimination in RNNs. How to target redundancies in RNNs requires a new perspective while taking factors such as memory limitations into consideration. This is because we cannot group all the input data together and perform clustering in the first place since the data processing at time t also depends on the results of time $t - 1$. Therefore, we design new approaches to detect and remove transient redundancy in RNNs.

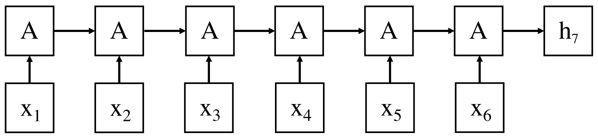


Fig. 13. Illustration of GRU.

Specifically, we start from Gated Recurrent Units (GRU) and design three strategies to make transient redundancy elimination effective on RNNs.

B. Transient Redundancy Elimination for RNNs

Gated Recurrent Units (GRU) are among the most common recurrent architectures. In the following illustration Fig. 13, for each step, an input vector is fed into the network, multiplied by a matrix A, and results in a hidden state h_i. This hidden state h_i together with input x_i are put in layer i for processing.

Inheriting the idea from TREC, we target the computations in the form of GEMM, namely $\mathbf{y} = \mathbf{x} \cdot \mathbf{W}$. To exert the potential for saving computations, there should be more neuron vectors in the matrix $\mathbf{x}$. This is because, with the fixed number of centroid vectors N_c, more neuron vectors result in a larger N, yielding a larger redundancy ratio r_ts since $r_t = 1 - \frac{N_c}{N}$ In the following content, we introduce three designs to increase the number of neuron vectors in $\mathbf{x}$, namely, the number of rows of $\mathbf{x}$.

Batch inference. Conducting batch inference with GRU proves a good way to increase the number of neuron vectors, namely N, in the GEMM. Specifically, if we originally have $N_{original} = n$ neuron vectors in $\mathbf{x}$, by doing batch inference, the new number of neuron vectors in $\mathbf{x}$ is multiplied by a factor of $batch\_size$, namely, $N_{new} = n \times batch\_size$.

Enlarge the number of output channels. The number of output channels is another factor influencing the effect of transient redundancy elimination. A larger output channel is able to make better predictions because it stores more hidden information. But this also means more redundancies exist in the hidden states. Therefore, by eliminating more redundancies, we are able to make better predictions while saving more computations and reducing the latency on microcontrollers.

Batch matrix multiplications across all steps. The sequential dependency in GRU causes the computation of each step to be halted until the previous step has fully executed, imposing restrictions on further computation savings. If we can batch all the matrix multiplications across all steps and aggregate them into one matrix multiplication, the number of neuron vectors in $\mathbf{x}$ will increase and we are able to save more computations. Simple Recurrent Units (SRU) are thus created with light recurrence that makes batch matrix multiplications possible while preserving the sequence modeling capacity. This makes it an ideal choice for running RNNs on microcontrollers.

Specifically, a single layer in SRU is described by the following:

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{v}_f \odot \mathbf{c}_{t-1} + \mathbf{b}_f) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + (1 - \mathbf{f}_t) \odot (\mathbf{W}\mathbf{x}_t) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{v}_r \odot \mathbf{c}_{t-1} + \mathbf{b}_r) \\
\mathbf{h}_t &= \mathbf{r}_t \odot \mathbf{c}_t + (1 - \mathbf{r}_t) \odot \mathbf{x}_t
\end{aligned} \tag{3}$$

Here, $\odot$ represents element-wise multiplication. During training, the parameter matrices $\mathbf{W}_f, \mathbf{W}, \mathbf{W}_r$ and the parameter vectors $\mathbf{v}_f, \mathbf{v}_r, \mathbf{b}_f$, and $\mathbf{b}_r$ are learned. The SRU architecture is comprised of a lightweight recurrent component that iteratively calculates the hidden states $\mathbf{c}_t$ by sequentially processing the input vector $\mathbf{x}_t$ at each step t.

By considering an input sequence $X = x_1, \cdots, x_L$ where each $x_t \in R^d$ represents a d-dimensional vector, SRU condenses the three matrix multiplications across all time steps into a single multiplication operation. This consolidation brings about a notable enhancement in computation intensity, resulting in a larger N of neuron vectors, thus more space for transient redundancy elimination. More precisely, the batched multiplication can be viewed as a linear projection of the input tensor $X \in R^{L \times d}$:

$$\mathbf{U}^{\mathrm{T}} = \begin{pmatrix} \mathbf{W} \\ \mathbf{W}_f \\ \mathbf{W}_r \end{pmatrix} \mathbf{X}^{\mathrm{T}} \tag{4}$$

VII. Implementation

A. Lasting Redundancy Elimination

To load a large DNN model onto Microcontrollers, the first steps are to eliminate lasting redundancy through *offline* methods so as to reduce the size of the model. These include effective pruning to reduce the number of parameters in a model when performing inference (detailed in Section VII-A), quantization of transforming 32-bit floating point data into 8-bit integer (detailed in Section VII-A), and the use of batch norm layer folding into the convolution layer that proves efficiency and effectiveness (detailed in Section VII-A).

Parameterized model Pruning. The first and foremost method of reducing the size of a network is model pruning. An effective pruning method should first reduce the number of parameters in the network significantly and, at the same time, easy to maintain the architecture structure of the model. In this work, we identified torch-pruning as the pruning method. Torch-pruning is a widely used pruning method [39] and is special in that, instead of trying to mask certain entries in a weight matrix, it removes the whole channels from the neural network for acceleration.

Network quantization. Floating point weights and activations are used to train neural networks. Previous research [40] has shown that fixed-point weights are enough for executing neural networks with minimum accuracy loss. This reduces weights from 32 bit to 8 bit, thus reducing the model size, and is good for microcontroller devices deployment. Furthermore, fixed-point integer operations in common microcontrollers are substantially faster than non-fixed point operation. This makes another justification for executing quantized models on microcontrollers.

Batch norm layer folding. When we apply quantization, problems arise for $batchnorm$ layer, since the original floating point means that μ and standard deviation σ are required. To solve this problem, we fold batch norm into the convolution when performing inference for the batch norm layers in DNNs.

In batch norm folding, we replace the convolution followed by a batch normalization with just one convolution with different weights. This would remove the precision loss if we quantize μ and σ from 32 bit floating point to 8 bit integer. Furthermore, it reduces the number of operations to be performed at inference time, thereby speeding up the entire network.

Specifically, the convolution operator followed by a batch normalization can be expressed, for an input $\mathbf{x}$, as:

$$\begin{aligned} \mathbf{z} &= W * \mathbf{x} + \mathbf{b} \\ y_i &= \gamma \cdot \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \end{aligned} \tag{5}$$

When we rearrange the value of W and $\mathbf{b}$ such that:

$$\begin{aligned} W_{\text{folded}} &= \gamma \cdot \frac{W}{\sqrt{\sigma^2 + \epsilon}} \\ b_{\text{folded}} &= \gamma \cdot \frac{b - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \end{aligned} \tag{6}$$

We manage to remove the whole batch norm layer and obtain the same result with only one convolution.

B. Implementing TREC

TREC design is at the core of our work in enabling efficient DNNs on microcontrollers. We need to pay special attention to the hardware architecture of the microcontrollers, and how the constraint space limits affect these above. We next explain each of these aspects.

Use of SIMD kernels. The energy-efficient 32-bit RISC cores in the ARM Cortex-M processors are optimized for microcontroller applications. Our work focuses on enabling large DNNs on CM platforms with specialized 16-bit MAC instructions like SMLAD. These instructions are crucial for DNN execution.

In the Cortex-M processor, each 32-bit word can hold four integers. Since we use 16-bit MAC operators, we convert quantized 8-bit data to 16-bit, extending and reordering four 8-bit integers within a word.

Matrix multiplication is vital in our network. We optimize matrix multiplication performance, particularly using the MAC instruction __SMLAD, for projection and reuse computation. Our solution outperforms the original Cortex-M library kernel. It eliminates the need for matrix transposition, improving space locality, and benefits from SIMD kernels for more efficient computation. Experimental results demonstrate notable improvements, such as a 1.67× speedup in projection during reuse compared to the original Cortex-M library matrix multiplication.

Methodology for placing weights and intermediate activations. For microcontrollers running DNNs, SRAM is used to allocate activation buffers, while Flash memory is used to allocate model weights and biases, as well as the network definition. Alternatively, weights can be stored in SRAM. However, we find that storing weights in Flash only results in a slight increase in end-to-end latency, roughly 1%. This can bring huge benefits because storing weights in Flash saves the limited space usage for activations, which can be placed in SRAM. Storing intermediate activations in Flash, on the other hand, renders a major drop-off in performance (over 20% increase in latency for a layer to store activation maps on the Flash in the current CifarNet setting through an explorative experiment). This is because of the need for block writes, rather than byte writes in Flash. Specifically, convolution layers write the output data to the Flash memory where the next layer reads and processes data. For efficient processing, studies are needed on modifications of kernel implementations, design, and testing of performance models.

VIII. Evaluation

A. Experimental Setup

Methodology. To demonstrate the efficacy of TREC in enabling large DNN inference on microcontrollers, we first apply our approach to the entire DNN and measure end-to-end inference performance. Second, we apply TREC and measure the single-layer acceleration and accuracy for individual layers (specifically the densely-computed layers like convolutional layers). Third, we focus on the space savings of our method. Specifically, how much space is saved from the original DNN models with *offline* methods including pruning, quantization, and layer folding. Since we have shown the TREC space overhead for the three networks in Table I, we examine the amount of space savings from the two ideas, i.e., *kernel reuse* and *two-step stack*. Fourth, we analyze the performance impact of the number of layers when applying *kernel reuse*. Fifth, we explore TREC performance on sparse inputs. For all the measurements, the SRAM and Flash occupancy is determined using the Mbed compiler (Mbed OS) and the microcontroller latency is measured using the Mbed Timer API.

Platform. DNN inference is performed on the STM32F469I and STM32F746ZG microcontroller featuring SIMD extensions. The STM32F469I platform comes with 324KB SRAM and 2048KB Flash as memory storage while the STM32F746ZG board has 320KB SRAM and 1024KB Flash. We also deploy CMSIS-NN kernels for optimized performance on the Cortex-M4 and Cortex-M7 CPU equipped on the STM32 board [2]. In addition, DNN training is performed on a server equipped by a 20-core 3.60GHz Intel Core i7-12700K CPU with 128GB RAM and an NVIDIA GeForce RTX A6000 GPU with 48 GB memory. We use PyTorch 1.10.1 (open-source software with a BSD license) as our frame for training.

Workloads. We evaluate our approach with the most popular compact DNNs that fit microcontrollers, namely CifarNet [41], ZfNet [21], and SqueezeNet, with and without complex bypass [42]. In all cases, we use the public dataset CIFAR-10 for training and evaluation. All networks are optimized by SGD. The learning rate starts from 0.001 and decreases by 0.1 at 15-epochs intervals. The batch size, weight decay, and momentum are set to 256, 0.9, and 10^{-4}, respectively, and the maximum number of training iterations is set to 100.

B. Performance

We compare performance of TREC-equipped DNNs to conventional Convolutional DNNs, *deep reuse* featured DNNs, and

TABLE II
LATENCY (MS) COMPARISON OF CMSIS-NN CONVOLUTION, LCNN, LCNN WITH TREC, AND TREC FOR THE CIFARNET

	Conv1 ($H, W = 32,\ K = 75$)	Conv2 ($H, W = 14,\ K = 1600$)	Conv1 + 2
CMSIS Convolution	96.64	57.4	154.69
LCNN	103.51	59.1	162.69
LCNN + TREC	93	49.46	143.18
TREC	53.4	37.76	91.35

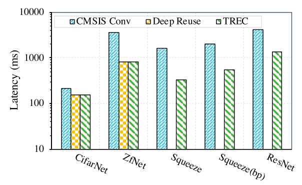
TABLE III
END-TO-END ACCURACY COMPARISON (%)

Conv Methods	CifarNet	ZfNet	SqueezeNet	SqueezeNet (Bypass)	ResNet (ImageNet-64x64)
CMSIS Conv.	78.2	80.1	83.5	85.6	48.0
Deep reuse	73.2 ∼ 76.1	72.5 ∼ 76.6	79.8 ∼ 81.9	80.5 ∼ 83.1	37∼44.5
TREC	76.5	78.9	83	85.3	47.1

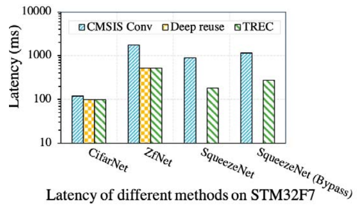
LCNNs. Note that after channel pruning, the baseline accuracy of ZfNet is slightly lower than its standard accuracy of 83%, but the model size is 7.6% of the original model, which can well meet the memory constraint of the microcontrollers. Read-only data can be placed in flash memory for microcontroller storage to hold the entire network.

The conventional convolution implementation is from the CMSIS-NN library [2] that achieves the state-of-the-art performance for convolutional neural networks on edge devices. For the datasets, inputs are first converted from floating point representation to 8-bit integers. This implementation is also from the CMSIS-NN library. We compare TREC with LCNN. LCNN stores only a dictionary (which can be regarded as a subset of weight vectors) for reconstructing weights so as to save space and computations. This strategy is actually complementary to TREC. In particular, when we perform convolution in LCNN, the input tensor undergoes a matrix multiplication with dictionary vectors, where our LSH approach can fit in and speed up this matrix multiplication. Through experiments, as shown in Table II, we find that LCNN experiences slowdown for CifarNet. This is because LCNN requires multiple memory accesses to the dictionary for reconstructing the original weight matrix, which are not suitable for microcontroller applications.

For comparison, we also include the results from *deep reuse*, as this state-of-the-art method avoids transient redundancy in DNNs. Given the inherent randomness of *deep reuse*, we present its results in 150-run intervals. Note that *deep reuse* requires storing LSH vectors and index structures on microcontroller memory, which exceeds the memory capacity (denoted as "—" in Table III for SqueezeNet and SqueezeNet (Bypass)). For TREC-equipped DNNs, all the space-saving techniques are applied to the model. In Table III and Fig. 14, we have the following findings. First, compared to the conventional convolution (by CMSIS-NN library), TREC achieves 3.61× speedup on average. The same amount of speedups are observed on both microcontrollers. STM32F746ZG's total end-to-end inference time is less than half of that measured on STM32F469NI. Unlike the Cortex-M4, the Cortex-M7 is capable of dual issuing load and ALU instructions, which enhances its instructions per cycle (IPC). When this capability is paired with a 20% increase



(a) Latency on STM32F4 board.



(b) Latency on STM32F7 board.

Fig. 14. End-to-end performance comparison on STM32F4 and STM32F7 boards.

in clock speed, it results in the STM32F7 being roughly twice as fast as the STM32F4. Note that TREC cache hit rate could be lower than that of the original convolution, but since it saves over 90% of computations, TREC still reduces overall execution time significantly.

Second, TREC experiences only around 0.7% accuracy loss. *Deep reuse* [24] is subject to its inherent randomness, with accuracy loss reaching up to 11%. TREC is hence better in terms of stability and accuracy. Since both TREC and *deep reuse* exploit *transient redundancy* elimination, they have comparable inference time. The optimizations in TREC make it however more space efficient. It enables large DNNs (i.e., SqueezeNet and ResNet) to run on microcontrollers, while *deep reuse* does not.

Third, when looking into the effects TREC bring to different networks, we find the most speedups in SqueezeNet. This is because there are more computation-intensive convolution layers in SqueezeNet. The more kernel channels are in a convolution layer, the more redundant computations can be detected and eliminated. We have a more detailed analysis in the following section.

C. Single Layer Speedup

We run experiments on each single convolutional layer with a different range of clustering configuration and collect the redundancy ratio. For the purpose of study, we find the configurations that can reduce the maximum amount of computations without threatening inference accuracy. Latencies are collected from the STM32F469NI board.

We explore the single-layer speedups and accuracy losses attained by substituting TREC for the traditional convolution. The following observations have been made. First, TREC finds and removes approximately 96.22% of the *transient redundancy* indicated by r_t. This is where the speedup comes from.

Second, with an average speedup of 4.28×, TREC significantly improves the performance of each convolution layer. Specifically, TREC speeds up SqueezeNet expand layers by up to 18.66×, showing that computation-intensive layers can particularly benefit from TREC. However, due to the additional cost from clustering, the speedup is smaller than what is suggested by the ratio of *transient redundancy* that TREC reduces.

Third, there is a limited and mixed impact of TREC on accuracy. It may result in a 0.8% accuracy decrease when applied to certain layers, but it may also result in a modest accuracy improvement when applied to some other layers. Overall,

TABLE IV
SPACE SAVINGS BY DIFFERENT TECHNIQUES

	Technique	CifarNet	ZfNet	SqueezeNet
Static Space	Baseline Size	420 KB	4 MB	5 MB
	Channel Pruning	45.1%	92.4%	90.8%
	CP + Q	86.8%	97.9%	97.4%
	CP + Q + LF	86.8%	98.9%	98.2%
Dynamic Space	Baseline Size	20 KB	40KB	56 KB
	Kernel Reuse	41.2%	68 .2%	71.9%
	KR + TSS	60.2%	73.4%	77.3%

CP: Channel Pruning, Q: Quantization, LF: Layer Foldings,
KR: Kernel Reuse, TSS: Two-Step Stack

TREC keeps the accuracy much better than *deep reuse* does (around 3%-7% drop). The single-layer results show that TREC can operate at maximum efficiency, achieving notable speedups with the least amount of accuracy loss.

D. Space Saving Analysis

There are static space and dynamic space required for storing models. Specifically, static space refers to the space for weights and biases (from the original model). Dynamic space contains TREC space, namely, *Hash* matrix and index structures. The baseline size for original static and dynamic space occupancy is detailed in Table IV.

For static space, we apply existing techniques to remove *lasting redundancy*. For dynamic space, we use *kernel reuse* and *two-step stack* to eliminate *transient redundancy*. Table IV summarizes how much space is saved for the original model size and TREC size, respectively. For static space, since there is no Batch norm layer in CifarNet Overall, the original model size witnesses an average of over 90% reduction as a result of these *lasting redundancy elimination* based methods. For dynamic space, we report the space reduction for *kernel reuse* with configurations where there is virtually no accuracy loss ($< 0.1\%$), and provide a detailed analysis in Section VIII-E. The dynamic space reduction exceeds 70% when we combine *kernel reuse* and *two-step stack*.

E. Detailed Analysis

Kernel reuse for reaching accuracy stability. *Kernel reuse* embeds the *Hash* matrix in the weight matrix W. Fig. 15 shows how *kernel reuse* influences SqueezeNet in terms of both accuracy and space occupancy on the STM32F7 board. The x axis denotes the number of layers we apply *kernel reuse* for SqueezeNet. We first look at the accuracy. For comparison, we also include the accuracy of *deep reuse* at 150 runs. *Deep reuse* has random LSH vectors, and its accuracy varies between $80\% \sim 83\%$. Accordingly, the two green lines show the accuracy range that *deep reuse* produces. In contrast, *kernel reuse* produces stable accuracy results up to 85.6%. Although there is accuracy loss as x increases, the network accuracy drops slowly and remains high ($> 84\%$) for $x < 16$.

Second, *kernel reuse* brings huge space benefits. For example, when *kernel reuse* is in five layers, namely, $x = 5$, the

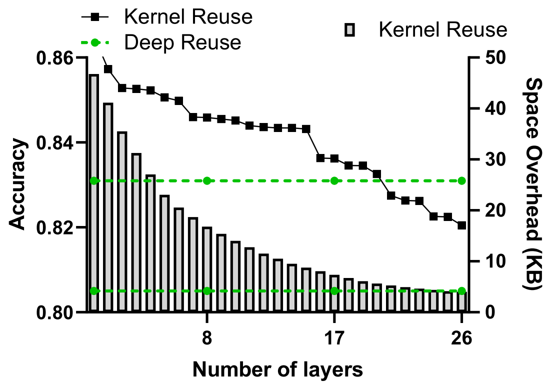


Fig. 15. Influence of accuracy and space overhead from the number of layers applying kernel reuse.

space overhead is cut by over 50% with accuracy loss $\delta < 0.5\%$. That is to say, for large DNNs, applying *kernel reuse* to a small number of layers can dramatically reduce the total space overhead with marginal accuracy loss.

F. Comparison With Other State-of-the-Art Networks Targeting MCU

Researchers have made many efforts to develop networks targeting resource-constrained devices such as MCU. The MCUNet work series (MCUNetV1, MCUNetV2) [43], [44] first uses evolutionary search of a pre-trained once-for-all (OFA) supernet and then enables the network model on MCU. MicroNets [7] feature a novel Differentiable NAS (DNAS) method to search models with low memory usage and low op count, where op count is treated as a viable proxy to latency. Accuracy in MicroNets is 95.3% for keyword spotting (KWS) tasks using convolutions and thus is superior to the original KWS on MCU (90.4%). Unified DNAS for Compressible TinyML Models (UDC) [45] targets model compression on MCU and Neural Processing Units (NPU). It uses an updated DNAS method to explore a large search space to generate compressible networks for NPU. Compared to MCUNet (version 4 on ImageNet-64x64), our method with ResNet-34 can achieve higher accuracy (47.1% compared to 46.07%) with comparable execution time (around 1 second for both). We did not find the open-sourced code for the latter two networks and thus could not compare it with our method.

Characterizing the applicability of reuse. TREC can eliminate data redundancies in DNNs, but this does not mean that it shows benefits for each case equally. There are certain scenarios where our method is more useful. (1) Generally a larger kernel size renders more reuse opportunities in an image, and it thus exposes more speedup potentials for our method. (2) The reuse technique is currently more useful for convolutional layers than fully connected layers, especially those close to the output. The performance of fully connected layers is more sensitive to reuse so accuracy can drop quickly. We provide these guidelines to help the community best exploit reuse opportunities on MCU.

TABLE V
PERFORMANCE FOR SPARSE INPUT MATRIX

ConvLayer	Kernel Size	Latency (ms) SparseConv	Latency (ms) TREC	Speedup
Conv1	75	21.98	6.51	3.38x
Conv2	1600	42.96	8.89	4.83x

TABLE VI
TRANSIENT REDUNDANCY ELIMINATION RESULTS FOR SRU

	SRU original	SRU with Reuse			
		H			
		5	8	9	10
Latency (ms)	83.23	19.92	26.19	47.21	61.16
Speedup	—	4.18×	3.18×	1.76×	1.36×
Accuracy	0.5163	0.4738	0.5046	0.5050	0.5055

IX. RESULTS AND DISCUSSIONS FOR TRANSIENT REDUNDANCY ELIMINATION

Aside from a comprehensive evaluation above of the space efficient TREC, transient redundancy is also applicable in many other scenarios. We give results and discussions in this section to offer a complete understanding of the idea behind TREC, namely transient redundancy elimination.

A. Sparse Convolution

We first investigate how transient redundancy elimination is applicable with sparse convolution and provide some experimental results on Cifarnet. Table V shows the performance for SparseConv and TREC. Input sparsity is set to 0.1 and batch size is 10. The speedup are 3.38× and 4.83× for the two convolution layers, respectively.

B. Transient Redundancy Elimination for RNNs

We experiment with how transient redundancy elimination help with the performance of SRU. We use Cifar-10 as the dataset for image classification. The time step is set to 96 and the batch size is 10. Accordingly, the input channel is 32. As for the model hyperparameters, the hidden size is set to 256 and the number of layers is 2. For transient redundancy elimination-specific configuration, L is set to 32.

The results are shown in Table VI. First, the reuse technique TREC is able to significantly speed up the latency of running RNNs on microcontrollers by detecting and removing the transient redundancy. Specifically, when H is set to 5, TREC-based SRU can reach a speedup of 4.18x compared to the original SRU.

Second, as H increases, namely, the number of centroids vectors increases, accuracy increases with minor latency slowdown. The sweet point in our case is $H = 8$ where we have satisfactory accuracy results compared to the original SRU and the latency speedup is also high. After that, the speedups of TREC-base SRU compared to original SRU drops quickly but there is only marginal accuracy increase.

TABLE VII
LATENCY OF GRU WITH DIFFERENT CONFIGURATIONS

Batch_Size	Output Channel	GRU (ms)	GRU With Reuse (ms)
1	32	0.03	1.07
1	64	0.06	1.24
25	32	0.52	1.25
25	64	1.05	1.47
100	32	2.12	1.63
100	64	4.13	2.04
1024	32	21.98	6.51
1024	64	42.96	8.89

We also explore the effect of batch size and the number of output channels on the running latency of GRU and obtained Table VII.

Through the table, we can see that, first, using the reuse technique to try to eliminate the transient redundancy does not pay much until the batch size reaches 100. As the batch size increases, the reuse technique results in more speedups since more redundancies are eliminated. Also, a larger number of output channels make transient redundancy elimination more effective since we remove more redundancy in the data.

Discussions on batch size. As shown in Table VI, for GRU, we can achieve up to 2× speedup for batch size 100, and 4.83× for batch size 1024. A batch size greater than one can be beneficial when high throughput and efficiency are required. Some autonomous systems may collect massive amounts of sensor data; processing these data in larger batches during off-peak times is feasible and allows for more efficient model updates and system improvement. Of course, the memory footprint required by a larger batch size increases because of the larger input size and activation maps. Batch size of 100 will increase the input size and the corresponding activation maps from 3K and 1.6K to 300K and 160K for CifarNet on Cifar-10. These data can still be stored on the on-chip eFlash, if not the SRAM.

C. Resource Constrained Architecture Exploration

Now that we have a complete understanding of transient redundancy elimination, we want to know what kind of models are suitable for this approach, specifically on microcontrollers. We then explore the properties of the resource-constrained model as follows. An ideal model running on microcontrollers would have high accuracy, small model size (number of model parameters) and footprint memory, and a low number of computations (FLOPs). To have a deeper understanding of how large a model can be executed on microcontrollers with acceptable accuracy loss, we next explore network architectures. Specifically, we are interested in the following: 1) what are the types of models and architectures that are suitable for running on microcontrollers, and 2) to what extent can we squeeze the model to make it executable on microcontrollers with marginal accuracy loss?

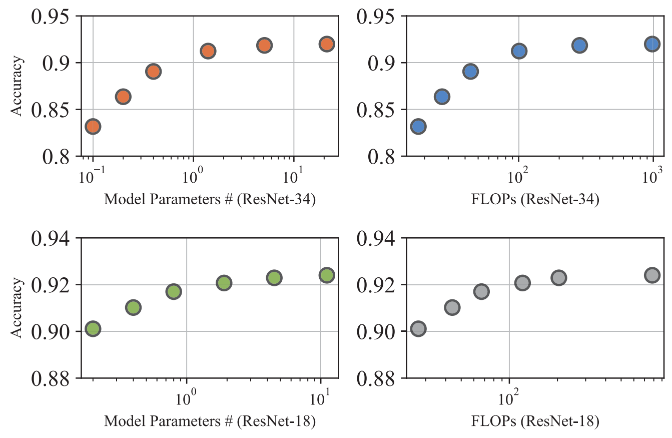


Fig. 16. Accuracy (%) vs. memory (M) and FLOPs of different NN models demonstrating the scalability of CNN models down to $<$ 8KB memory footprint and $<$ 500K operations.

As shown in the previous Evaluation section, models such as CifarNet with small sizes can make the best of their modeling capability without much model pruning. As for larger models, we show in Fig. 16 our exploration of ResNet-18 and ResNet-34 with regard to the influence of memory footprint and FLOPs on accuracy. We again use Cifar-10 as the dataset. We have some observations from the results. First, the original ResNet-18 and RestNet-34 have similar accuracy on the Cifar-10 dataset, which is around 92%. This shows that ResNet-18 is expressive enough in terms of its modeling capacity as ResNet-34 is larger in model size but with little accuracy increase.

Second, as the model size or the number of operations decreases, the model accuracy keeps stable in the first place and then drops sharply below 1 M (for model size) or 100 FLOPs (for operations). This means a model around 1M is appropriate for microcontrollers as it minimizes the number of operations while keeping marginal accuracy loss.

Third, ResNet-18 shows better scalability compared to ResNet-34. Specifically, with the same model size, models pruning from ResNet-18 demonstrate higher accuracy than that from ResNet-34. For example, the 0.4M model pruned from ResNet-18 has an accuracy of 91.02% while the model with the same size pruned from ResNet-34 scores an accuracy of 89.06%. This is because ResNet-34 suffers from overfitting as it has too many parameters for an image classification task running on MCU such as Cifar-10.

X. Conclusion

In this work, we demonstrate that large DNNs can be deployed on resource-constrained microcontrollers and executed efficiently by the introduction of transient redundancy elimination. It takes advantages of both computation reduction and space savings that make the best ability of the resource utilization of microcontrollers. As a result, the approach maximizes the performance benefits and obtain stable accuracy results. In the evaluation, our solution achieves 3.5×-5× speedups with virtually no accuracy loss.

References

[1] D. E. Bolanakis, “A survey of research in microcontroller education,” *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 14, no. 2, pp. 50–57, May 2019.
[2] L. Lai et al., “CMSIS-NN: Efficient neural network kernels for arm cortex-m CPUs,” 2018, *arXiv:1801.06601*.
[3] N. P. Novani et al., “Electrical household appliances control using voice command based on microcontroller,” in *Proc. Int. Conf. Inf. Technol. Syst. Innov. (ICITSI)*, 2020, pp. 288–293.
[4] S. Soro, “TinyML for ubiquitous edge AI,” 2021, *arXiv:2102.01255*.
[5] Y. Zhang et al., “Hello edge: Keyword spotting on microcontrollers,” 2017, *arXiv:1711.07128*.
[6] I. Fedorov et al., “Sparse: Sparse architecture search for CNNs on resource-constrained microcontrollers,” in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, pp. 4977–4989, 2019.
[7] C. Banbury et al., “MicroNets: Neural network architectures for deploying tinyML applications on commodity microcontrollers,” in *Proc. Mach. Learn. Syst. 3 (MLSys)*, 2021, pp. 517–532.
[8] J. Kaivo-oja, “Weak signals analysis, knowledge management theory and systemic socio-cultural transitions,” *Futures*, 2012.
[9] D. Kim et al., “Willingness to provide personal information: Perspective of privacy calculus in IoT services,” *Comput. Human Behav.*, vol. 92, pp. 273–281, 2019.
[10] X. Xu et al., “Scaling for edge inference of deep neural networks,” *Nature Electron.*, vol. 1, no. 4, pp. 216–222, 2018.
[11] C.-B. Tzeng, “Vibration detection and analysis of wind turbine based on a wireless embedded microcontroller system,” in *Proc. IEEE Int. Conf. Appl. Syst. Invention (ICASI)*, 2018, pp. 133–136.
[12] M. V. Ngo et al., “Adaptive anomaly detection for IoT data in hierarchical edge computing,” 2020, *arXiv:2001.03314*.
[13] S. Jacob et al., “Artificial muscle intelligence system with deep learning for post-stroke assistance and rehabilitation,” *IEEE Access*, vol. 7, pp. 133463–133473, 2019.
[14] A. Kolesau et al., “Voice activation systems for embedded devices: Systematic literature review,” *Informatica*, 2020.
[15] P. Sharma et al., “Intelligent object detection and avoidance system,” in *Proc. Int. Conf. Transforming IDEAS (Inter-Disciplinary Exchanges, Anal., Search) Viable Solutions*, 2019, pp. 342–351.
[16] Y. Yu et al., “FASSNet: Fast apnea syndrome screening neural network based on single-lead electrocardiogram for wearable devices,” *Physiol. Meas.*, vol. 42, no. 8, 2021, Art. no. 085005.
[17] R. David et al., “TensorFlow lite micro: Embedded machine learning for tinyML systems,” in *Proc. Mach. Learn. Syst. 3 (MLSys)*, 2021, pp. 800–811.
[18] A. Massa et al., “DNNs as applied to electromagnetics, antennas, and propagation—A review,” *IEEE Antennas Wireless Propag. Lett.*, vol. 18, no. 11, pp. 2225–2229, Nov. 2019.
[19] J. Liu et al., “Space efficient TREC for enabling deep learning on microcontrollers,” in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2023, pp. 644–659.
[20] A. Krizhevsky et al., “Learning multiple layers of features from tiny images,” 2009.
[21] M. D. Zeiler et al., “Visualizing and understanding convolutional networks,” in *Proc. Eur. Conf. Comput. Vis.*, New York, NY, USA: Springer-Verlag, 2014, pp. 818–833.
[22] F. N. Iandola et al., “SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5 mb model size,” 2016, *arXiv:1602.07360*.
[23] M. Riera et al., “Computation reuse in DNNs by exploiting input similarity,” in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2018, pp. 57–68.
[24] L. Ning et al., “Deep reuse: Streamline CNN inference on the fly via coarse-grained computation reuse,” in *Proc. ACM Int. Conf. Supercomput. (ICS)*, 2019, pp. 438–448.
[25] A. Company, “Cortex®-m4 technical reference manual,” The University of Texas at Austin. Accessed: Dec. 22, 2009. [Online]. Available: https://users.ece.utexas.edu/valvano/EE345L/Labs/Fall2011/CortexM4_TRM_r0p1.pdf
[26] F. Zhang et al., “Understanding co-running behaviors on integrated CPU/GPU architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 905–918, Mar. 2017.
[27] B. Haoqiong et al., “Using cloud functions as accelerator for elastic data analytics,” in *Proc. SIGMOD Companion Int. Conf. Manage. Data*, 2023, pp. 1–27.

[28] R. Wu et al., "Exploring deep reuse in winograd CNN inference," in *Proc. 26th ACM SIGPLAN Symp. Princ. Practice Parallel Program. (PPoPP)*, 2021, pp. 483–484.
[29] F. Zhang et al., "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.
[30] R. Wu et al., "Drew: Efficient winograd CNN inference with deep reuse," in *Proc. ACM Web Conf.*, 2022, pp. 1807–1816.
[31] S. Han et al., "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*.
[32] J. Yuan et al., "Power-efficient interrupt-driven algorithms for fall detection and classification of activities of daily living," *IEEE Sensors J.*, vol. 15, no. 3, pp. 1377–1387, Mar. 2015.
[33] A. E. Eshratifar et al., "BottleNet: A deep learning architecture for intelligent mobile cloud computing services," in *Proc. Int. Symp. Low Power Electron. Des. (ISLPED)*, 2019, pp. 1–6.
[34] J. Shao et al., "BottleNet++: An end-to-end approach for feature compression in device-edge co-inference systems," in *Proc. IEEE Int. Conf. Commun. Workshops (ICC Workshops)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1–6.
[35] J. Guan et al., "TREC: Transient redundancy elimination-based convolution," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2022, pp. 26578–26589.
[36] W. Niu et al., "PatDNN: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, 2020, pp. 907–922.
[37] X. Li et al., "Three-dimensional backbone network for 3d object detection in traffic scenes," 2019, *arXiv:1901.08373*.
[38] C. Yang et al., "A neural network approach to jointly modeling social networks and mobile trajectories," *ACM Trans. Inf. Syst.*, 2017.
[39] P. Bajcsy et al., "Baseline pruning-based approach to Trojan detection in neural networks," 2021, *arXiv:2101.12016*.
[40] L. Lai et al., "Deep convolutional neural network inference with floating-point weights and fixed-point activations," 2017, *arXiv:1703.03073*.
[41] "CifarNet," Places365. Accessed: Apr. 8, 2009. [Online]. Available: http://places.csail.mit.edu/deepscene/small-projects/TRN-pytorch-pose/model_zoo/models/slim/nets/cifarnet.py
[42] F. N. Iandola et al., "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and $<$ 0.5 MB model size," 2016, *arXiv:1602.07360.*
[43] J. Lin et al., "MCUNet: Tiny deep learning on IoT devices," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 11711–11722.
[44] J. Lin et al., "MCUNetv2: Memory-efficient patch-based inference for tiny deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 2346–2358.
[45] I. Fedorov et al., "UDC: Unified DNAs for compressible tinyML models for neural processing units," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 18456–18471.

Jiesong Liu is working toward the Ph.D. degree in computer science with the North Carolina State University. His research interests include optimizations for machine learning systems, machine learning-enhanced modeling, and simulation for heterogeneous computing.

Feng Zhang (Member, IEEE) received the bachelor's degree from Xidian University, in 2012, and the Ph.D. degree in computer science from Tsinghua University, in 2017. He is a Professor with the DEKE Lab and School of Information, Renmin University of China. His research interests include database systems, and parallel and distributed systems.

Jiawei Guan is working toward the Ph.D. degree with the School of Information, Renmin University of China. Her research interests include high performance computing, compression, and machine learning, with a focus on improving the performance of systems or algorithms.

Hsin-Hsuan Sung is working toward the Ph.D. degree in computer science with the North Carolina State University. His research interests include system software, autonomous driving systems (ADS), and compilers. In his recent research, he worked on multi-DNNs' co-run-conscious DAG scheduling policy on Autoware ADS. In the project, he proposed and implemented the scheduling algorithms in C++ and had hardware co-design to mitigate DNN's co-run effect on heterogeneous SoC.

Xiaoguang Guo is a Research Assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2020. His major research interests include database systems and distributed systems.

Saiqin Long received the Ph.D. degree in computer applications technology from the South China University of Technology, Guangzhou, China, in 2014. She is currently a Professor with the College of Information Science and Technology, Jinan University, China. Her research interests include cloud computing, edge computing, parallel and distributed systems, and Internet of things.

Xiaoyong Du (Member, IEEE) received the B.S. degree from Hangzhou University, Zhejiang, China, in 1983, the M.E. degree from Renmin University of China, Beijing, China, in 1988, and the Ph.D. degree from Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a Professor with the School of Information, Renmin University of China. His research interests include databases and intelligent information retrieval.

Xipeng Shen (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Rochester, in 2006. He is a Professor in computer science with the North Carolina State University. He is a recipient of the DOE Early Career Award, the NSF CAREER Award, the Google Faculty Research Award, and the IBM CAS Faculty Fellow Award. He is an ACM Distinguished Member and an ACM Distinguished Speaker. His research interest include programming systems and machine learning.