# Generalizing Reuse Patterns for Efficient DNN on Microcontrollers

Jiesong Liu
North Carolina State University
Raleigh, North Carolina, USA
jliu93@ncsu.edu

Bin Ren
William & Mary
Williamsburg, Virginia, USA
bren@wm.edu

Xipeng Shen
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

## Abstract

Deep Neural Networks (DNNs) face challenges in deployment on resource-constrained devices due to their high computational demands. Leveraging redundancy in input data and activation maps for computation reuse is an effective way to accelerate DNN inference, especially for microcontrollers where the computing power is very limited. This work points out an important limitation in current reuse-based DNN optimizations, the narrow definition of reuse patterns in data. It proposes the concept of *generalized reuse* and uncovers the relations between generalized reuse patterns and row/column reorder of a matrix view of the input or activation map of a DNN. It revolutionizes the conventional view of explorable reuse patterns, drastically expanding the reuse space. It further develops two novel *analytical models* for analyzing the impacts of reuse patterns on the accuracy and latency of DNNs, enabling efficient selection of appropriate reuse patterns. Experiments show that *generalized reuse* consistently brings significant benefits, regardless of the differences among DNNs or microcontroller hardware. It delivers 1.03-2.2× speedups or 1-8% accuracy improvement over conventional reuse.

***CCS Concepts:*** • **Computer systems organization** → **Approximate computing**; **Real-time systems**; • **Software and its engineering** → **Compilers**; • **Computing methodologies** → **Neural networks**.

***Keywords:*** real-time machine learning, compiler optimization

**ACM Reference Format:**
Jiesong Liu, Bin Ren, and Xipeng Shen. 2025. Generalizing Reuse Patterns for Efficient DNN on Microcontrollers. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716257

ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3716257

## 1 Introduction

Recent years have witnessed an increasing interest in running computer vision DNNs on microcontrollers (MCU), such as smart cameras and devices used in smart manufacturing. DNNs are computationally intensive. Efficient DNNs inferences on MCU are especially challenging, due to the limited space and computing power on those devices [14, 35, 36, 57]. Removing redundancies is an important way to speed up DNN inference [12, 15, 20, 26, 56, 70, 71].

Extensive studies have been conducted to exploit the redundancy among DNN parameters. Examples include model compression techniques [6, 19, 22, 47] such as quantization [60], filter size reduction [12], feature compression [50].

Those techniques, however, are oblivious to the redundancy in the input data. Redundancy is ubiquitous in real-world data. As illustrated in Figure 1, multiple tiles in a channel of an image may be similar to one another. Reuse [17, 37, 41, 42, 55] is a promising way to leverage the redundancy for speedups. It exploits similar tiles (which form so-called *neuron vectors*[42] in a matrix view of the input) within an image or activation map (i.e., the outputs of a neural network layer). By detecting data redundancies of the input data and activation maps through online clustering, reuse methods eliminate the data redundancy by reusing the computation results for similar tiles. Reuse-based DNN optimizations are general and beneficial. Prior work [17] shows that they can eliminate over 90% of computations for a convolutional layer while suffering little accuracy loss. (Although reuse can also apply to fully connected layers and recurrent neural networks, the discussion in this paper concentrates on convolution for its pivotal role in computer vision applications.)

All the previous explorations on reuse, however, have been based on a single, most straightforward pattern. Here, a reuse pattern refers to the way used to define a *neuron vector* in the input image or activation map. *Neuron vector* is the unit used by the clustering operation in a reuse-based DNN inference to identify similar parts for computation reuse. In existing reuse-based DNN optimizations, a neuron vector just comprises the values in a flattened tile of pixels in a channel of an input image or activation map [17, 37, 41, 42].

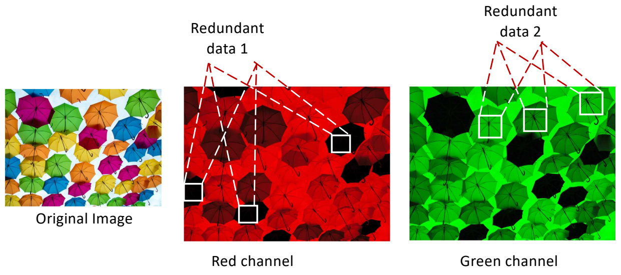

**Figure 1.** Illustration of similar tiles in an image in each of its channels (only two channels are shown).

In this work, we point out that the conventional reuse is actually a special case of reuse, and that the narrow definitions of reuse patterns have critically restrained the full exertion of the power of reuse-based DNN optimizations.

To fully unlock the potential, as the first-fold contribution of this work, we propose the concept of *generalized reuse*. Compared to previous reuse, generalized reuse has several innovations at the conceptual level:

- First, it provides two novel insights on reuse patterns and how they relate to the row/column reorder of a matrix view of the input or activation map.
- Second, based on the two insights, it establishes the concept of *reuse space* as a systematic way to characterize various reuse patterns.
- Third, it generalizes the reuse unit from 1-D *neuron vector* to 2-D *neuron block*, and introduces a new reuse direction (*horizontal reuse*), which together help significantly expand the varieties of reuse patterns.
- Fourth, it characterizes the connections between reuse patterns and matrix reorder, from which, it derives an easy way to generate various reuse patterns.

These innovations revolutionize the conventional view of explorable reuse patterns, drastically expanding reuse options and hence elevating the potential of reuse-based DNN optimizations to a new level.

The new potential brings a new challenge: As the explorable reuse patterns dramatically expand, how to efficiently choose the appropriate reuse pattern to use for a given problem? For a given input, different reuse patterns could discover different amounts of similar tiles and hence let reuse save different amounts of computations and cause different accuracy losses. The best reuse pattern can differ for different inputs. Conventional reuse-based DNN [17, 37, 41, 42] empirically examines each reuse pattern by retraining the DNN optimized with the reuse pattern and then checking the accuracy and latency. DNN training takes time. The enormous reuse space in *generalized reuse* makes this method impractical to use. As the second-fold contribution of this work, we create two novel analytical models for analyzing the impacts of the generalized reuse patterns, respectively on the accuracy and the latency of DNN inferences. The accuracy model leverages the Squared Freobenius norm to quantify the upper bound of the accuracy loss due to the use of a reuse pattern. The latency model approximates the time savings by a reuse pattern through the analysis of the saved computations and the overhead. The models allow users to quickly focus on a small set of promising reuse patterns, who may then use the full empirical measurement to check the effectiveness of each of the reuse patterns in that set. By reducing the number of reuse patterns needed for a full empirical check, the models eliminate the major barrier to realizing the potential of the generalized reuse.

We evaluate generalized reuse on four popular DNN networks, namely CifarNet [29], ZfNet [61], and two variants of SqueezeNet [23] (with and without bypass). Experiments on two different MCUs show that generalized reuse consistently brings significant benefits, regardless of the differences among DNNs or MCU hardware. It helps DNNs avoid over 96% computations on convolution layers. Compared to conventional reuse, the *generalized reuse* brings 1.03-2.2× speedups with similar accuracy, or 1-8% accuracy increase with similar latency.

The main contributions of this work are as follows:

- It points out an important limitation on reuse patterns in prior reuse-based DNN optimizations.
- It proposes the concept of *generalized reuse* that revolutionizes the conventional view of explorable reuse, and establishes a reuse space that characterizes a much broader range of reuse patterns.
- It uncovers the relations between reuse patterns and row/column reorder of a matrix view of the input or activation map of a DNN, based on which, it provides an easy way to generate various reuse patterns.
- It develops two novel analytical models for analyzing the impacts of reuse patterns on the accuracy and the latency of DNN inference, enabling efficient selection of appropriate reuse patterns.
- It empirically evaluates the effectiveness of the new solution on two models of MCUs, confirming the substantial benefits of the new solution in enabling efficient DNN inference.

## 2 Background

This section provides background on MCUs and locality sensitive hashing (LSH).

***MCU.*** MCU is an energy-efficient processor used everywhere, from household appliances [44], to cars [49], consumer electronics [54], wearables [53] and so on [3, 4, 25, 28, 51, 59]. An estimated 250 billion microcontrollers are currently in use [52]. Figure 2 presents an example of an MCU architecture and its memory hierarchy, specifically the STM32F469I. As shown in Figure 2 (a), the Cortex-M4 core architecture features a 32-bit processor (CM4) along with a minimal set of essential peripherals. The CM4 core follows a

Harvard architecture, meaning it employs separate interfaces for fetching instructions (Inst) and data (Data). This design enables simultaneous access to instruction and data memory, preventing CPU stalls due to memory access bottlenecks. A key distinction between the Cortex-M4 and its predecessor, the CM3 [11], is the inclusion of single-instruction multiple-data (SIMD) extensions, which significantly enhance arithmetic computing performance. From the CM4's perspective, all components appear as memory, distinguishing only between instruction fetches and data accesses.

Figure 2 (b) illustrates the on-chip memory hierarchy, which is notably constrained in terms of available space. Microcontrollers generally consist of a central processing unit (CPU), cached memory for frequently accessed data, static random-access memory (SRAM), and on-chip flash memory for storage.

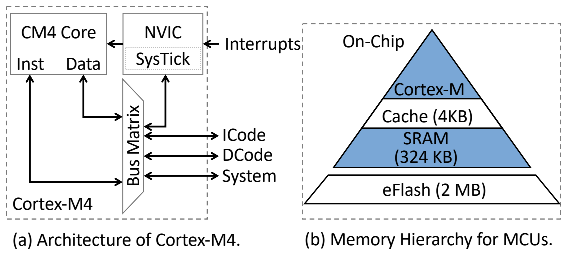

**Figure 2.** An illustration of the architecture and memory hierarchy for microcontrollers.

Microcontrollers are highly energy-efficient, consuming minimal power (0.166W for the F469I board), and are more cost-effective than conventional processors such as CPUs and GPUs [63]. Microcontrollers provide very limited computing resources and a limited volume of storage (324 KB SRAM and 2 MB on-chip flash memory) that developers need to take care of. Optimizations to DNN efficiency is hence essential to the deployment of DNN on microcontrollers [2, 13, 18, 30, 39, 44, 69].

***Locality Sensitive Hashing (LSH).*** LSH is an online clustering technique widely used in various solutions, including our work, to facilitate computation reuse in DNN optimizations [45]. As defined in Equation 1, a parameter vector $\mathbf{v}$ is used to transform an input vector $\mathbf{x}$ into either 1 or 0 via a hash function $h_{\mathbf{v}}$:

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

When $H$ hash functions are applied, each input vector is mapped to an $H$-bit binary vector. Similar input vectors are likely to yield identical hash outputs, naturally forming clusters, while the parameter $H$ determines the granularity of clustering.

Each neuron vector is assigned a unique ID based on its corresponding $H$-bit vector. Neuron vectors sharing the same ID are grouped into clusters, allowing them to reuse the computed result of the centroid vector instead of performing redundant individual computations. When LSH is used in reuse-based DNN, the appropriate hash vectors can be learned in the DNN training process [17, 37].

## 3 Generalized Reuse

Real images contain data redundancies. As shown in Figure 1, several parts in an image are similar in each of its channels. These redundancies expose reuse opportunities for DNN acceleration. Originated from Lin and others [42], *deep reuse* is a main technique for exploiting the redundancies to reduce DNN inference computations [17, 37, 41, 42, 55]. This section first reviews *deep reuse* and then introduces the concept of *generalized reuse*.

### 3.1 Review of Deep Reuse

Figure 3 illustrates *deep reuse* in convolution. As the top of Figure 3 shows, the convolution of a kernel on an input tile is reformed into the multiplication of two vectors; the values in an input tile are put into a *neuron vector* as part of an input matrix (which is also called *im2col matrix*), and the values in a kernel is put into a *kernel vector* as part of a weight matrix. The two matrices are represented as $\mathbf{X}$ and $\mathbf{W}$ matrices on top of the *deep reuse* workflow in Figure 3. Note that a neuron vector can be part of a row in the input matrix. In Figure 3, each row in $\mathbf{X}$ is evenly split into two *neuron vectors*. The matrix $\mathbf{X}$ is viewed (called *sliced*) as two submatrices (blue and yellow in Figure 3).

The *deep reuse* process comprises four steps. First, each sub-matrix $\mathbf{X}_i, i = 1, 2, \cdots, \frac{m}{l}$ ($m$ is the row length and $l$ is the neuron vector length), goes through a clustering step. At a high level, it uses LSH (Sec 2) to do clustering. In either sub-matrix in Figure 3, the four neuron vectors are grouped into two clusters (denoted by two different colors). These centroid vectors in the clusters constitute the centroid matrices $\mathbf{X}_i^c$. Second, since $\mathbf{X}_i^c$ attend to different parts of $\mathbf{W}$, each centroid matrix multiplies the corresponding weights to compute the centroid results, $\mathbf{Y}_i^c$. Third, to compute $\mathbf{Y}_i$ from $\mathbf{Y}_i^c$, it recovers the results of the rest vectors $\mathbf{x}_{*,i}$ in $\mathbf{X}_i$ by duplicating the centroid result of the particular cluster that $\mathbf{x}_{*,i}$ belongs to. Finally, the reuse process produces the final output $\mathbf{Y}$ by adding up $\mathbf{Y}_i$.

We make two notes. (i) Our illustration has been using input images, but *deep reuse* can apply to the outputs of each neural network layer as well, that is, *activation maps*. (An activation map can have more than three channels.) (ii) After the initial proposal of *deep reuse*, studies have extended it in various ways. The most recent progress is TREC, which makes the LSH hash vectors (used in the online clustering in deep reuse) be automatically learned as part of the DNN training process [17, 37]. Compared to the use of random

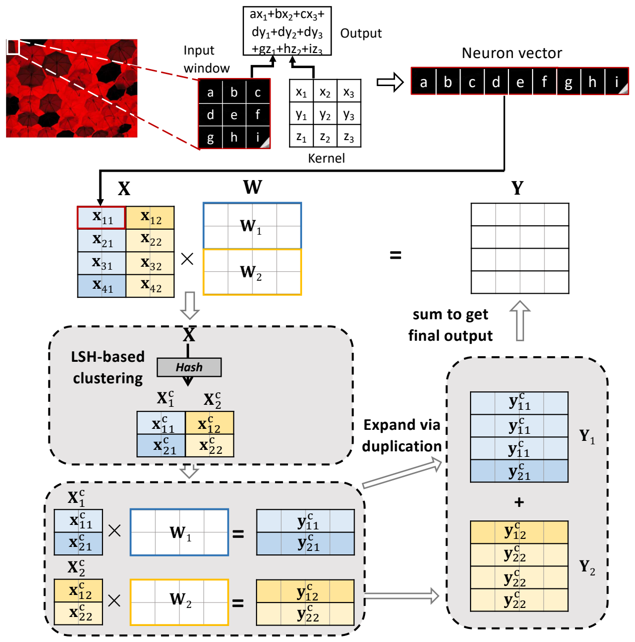

**Figure 3.** An illustration of deep reuse on *im2col* matrix **X**. Each row is divided into several neuron vectors such that each neuron vector attends to certain parts of the weight matrix **W**. The technique clusters neuron vectors in each submatrix. The resulting matrices are then duplicated to recover the full results for each submatrix; they are finally summed up into the final output matrix.

hash vectors in the original deep reuse, TREC achieves significantly better speed and accuracy[1]. In the rest of this paper, unless noted otherwise, *deep reuse* refers to the TREC version by default. It is worth noting that the reuse technique is more useful for convolutional layers than fully connected layers: The fully connected layers, especially those close to the output layer, are typically more sensitive to reuse in terms of the impact to accuracy. A recent study shows that reuse can be applied to recurrent neural networks [38] as well.

## 3.2 Key Insights and Reuse Space of Generalized Reuse

*Deep reuse* reduces computations in DNN inference through reuse, but its definition of the reuse unit is narrow—the neuron vector can only be consecutive elements in a channel, and the reuse direction is only among the vectors within a vertical panel in an input matrix[2]. As a result, the reuse patterns it can exploit are very limited.

[1]Random hashing reuse causes huge fluctuations in the model accuracy, e.g., 0.73 to 0.76 for CifarNet. This significantly defects the model deployment.
[2]Prior work [42] also explored reuse across input matrices but a neuron vector is still defined as one tile in a channel of a single image.

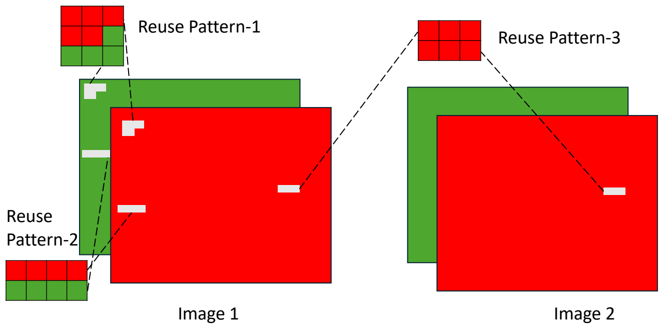

**Figure 4.** Reuse patterns.

Besides the reuse patterns that *deep reuse* can exploit, there are countless other reuse patterns. Figure 4 illustrates several examples. Pattern-1 forms a neuron vector by taking 5 elements in the red channel and 4 elements in the green channel of an image; pattern-2 uses 4 elements in each channel; pattern-3 uses 3 elements from the red channel of two images. Here, patterns 1, 2, and 3 each consist of two tiles; the two tiles reside in either different channels (as in patterns 1 and 2) or different images (as in pattern 3). In each of these patterns, the two tiles together form a single reuse unit. By defining reuse unit in this way, each pattern offers a possible way to enable efficient reuse by capturing the similarities across reuse units. Pattern 3, specifically, picks two tiles in two images together as one reuse unit, which is used to compare to other reuse units formed by other tiles in these two images. Note that none of those reuse patterns are covered by existing *deep reuse*. Moreover, the reuse unit has always been a *1-D* vector in the im2col matrix. Can it be a *2-D* block?

In this work, we propose *generalized reuse* to lifts the limits and to offer a way to effectively explore the full reuse pattern space. The proposal is based on two key insights:

- **Insight-1:** A reuse pattern is determined by three factors, the definition of reuse unit, the reuse direction, and the granularity of a neuron vector.
- **Insight-2:** The various definitions of neuron vectors in the *image view* (e.g., Figure 4) of an input or activation map can be systematically materialized through row or column reordering on its *matrix view* (namely, the matrix after the im2col expansion), with coordinated adjustment to the *memory view* (the data layout in the memory). [3] Therefore, reordering on the matrix view, plus choices in reuse directions and granularities, produces all possible reuse patterns.

[3]Simply put, the image view represents data across multiple channels (e.g., RGB) in its original size. This data can be transformed into a single matrix using the im2col operation. The memory view shows how this matrix is organized and stored in memory.

The first insight is straightforward to understand. The definition of reuse unit and its granularity determine the tiles used in similarity checking for reuse, and the reuse direction (detailed in Section 3.4) determines what other tiles to check against (e.g., tiles on the same row or column). So together they give a 3-D space, as shown in Figure 5. Every point in the space is a combination of the three factors and defines a reuse pattern. We call the space *reuse space*.

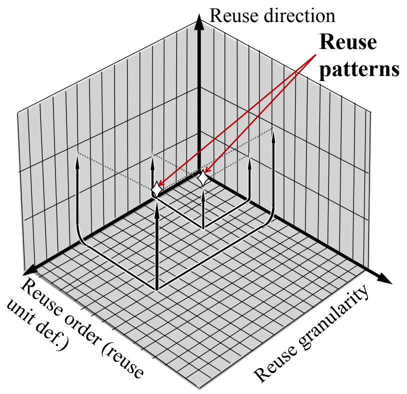

**Figure 5.** Illustration of generalized reuse space. Any reuse pattern is essentially a combination of choices in the three dimensions: *reuse order*, *reuse direction*, and *reuse granularity*.

The second insight entails that one of the dimensions of the *reuse space*, reuse unit definition, can be replaced with row/column order in the matrix view. It also suggests a systematic way to generate all the possible reuse patterns.

We next explain the three views involved in the second insight and the connections between the order and reuse patterns.

## 3.3 Generalized Reuse: Three Views and Reordering

Understanding the three views mentioned in Insight-2 and their relations is the key to understanding how reordering on the matrix view (i.e., *im2col view*) can generate various neuron vectors in the reuse patterns.

There are three views for a given input or activation map, *image view*, *im2col view*, and *memory view*; changes in the reuse patterns in the *image view* prompt corresponding changes in the other two views. The *image view* of an input or activation map of $C$ channels consists of $C$ 2-D views, with each holding the values in one channel, as illustrated in Figure 6 (a). The *im2col view* is a 2-D matrix that encompasses all the values in all the channels in the *image view*. The default mapping between them is shown in Figure 6 (b): One row in *im2col view* consists of the values of all the channels in a tile, laid out channel by channel. The im2col views of multiple images (called a *batch*) could be stacked in one im2col matrix. The *memory view* is 1-D, showing how the matrix elements in the *im2col view* are stored in memory, which is sometimes also called *memory layout*. What Figure 6 (c) illustrates is

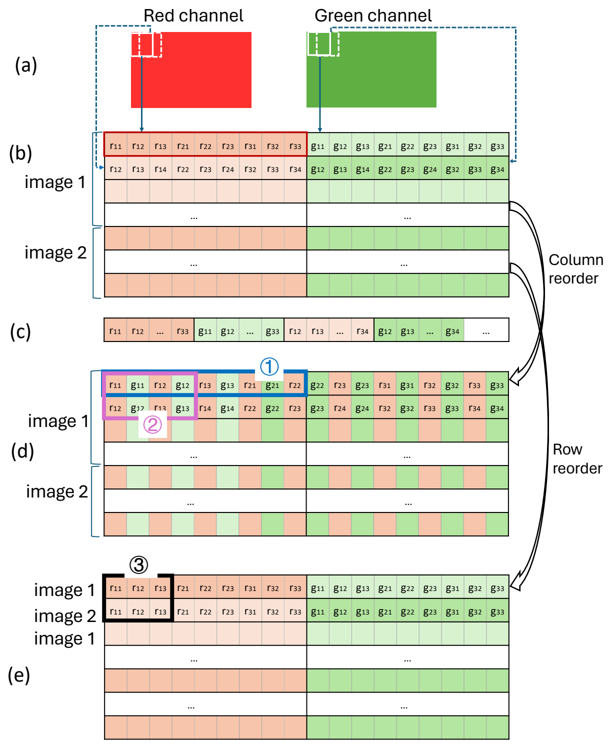

**Figure 6.** Example of three views and their mappings. (a) image view (showing only two channels); (b) im2col view (default); (c) memory view derived from part b (row major); (d) im2col view after a column reorder; (e) im2col view after a row reorder.

row-major, where the elements in the Figure 6 (b) im2col matrix are laid out row by row in memory; this layout is often used in CPU memory; column-major is often used in GPU memory. There are other layouts when matrix tiling is used.

Before discussing reordering, we note that *generalized reuse* expands the definition of a reuse unit from 1-D to 2-D. In *deep reuse*, the reuse unit is a neuron vector which must be a segment in a row in the im2col matrix. In *generalized reuse*, the reuse unit is a *neuron block* which is a 2-D block of elements in an im2col matrix. The old definition becomes a special case with one dimension of the block being 1.

Now we can see how reordering on the matrix view (i.e., *im2col view*) generates various neuron blocks. The key observations are that (i) the mapping from the *image view* to the *im2col view* affects the content of a neuron block, and (ii) row or column reorder in the im2col matrix changes the content of a neuron block, and hence the area it maps to in the *image view* and hence the reuse pattern. For instance, Figure 6 (d) is the result of column reorder of the im2col matrix in Figure 6 (b).

Using numpy API, this column reordering can be achieved as follows:

```
# b is an im2col matrix as illustrated in Figure 6(b)
b = np.resize(b, batch_size * output_height * output_width,
    num_channel * kernel_height * kernel_width)
b = np.reshape(b, batch_size * output_height * output_width,
    num_channel, kernel_height * kernel_width)
b = np.moveaxis(b, 1, -1)
```

The neuron block ① previously corresponding to a 3x3 tile in the red channel now corresponds to the irregular pattern (pattern-1) in Figure 4. Pattern-2 in Figure 4 corresponds to neuron block ② in Figure 6 (d). Pattern-3 in Figure 4 comes from the result of a row reorder as shown in Figure 6 (e) and its neuron block ③. The reordering can be implemented as follows in numpy API:

```
# b is an im2col matrix as illustrated in Figure 6(b)
b = np.resize(b, batch_size * output_height * output_width,
    num_channel * kernel_height * kernel_width)
b = np.reshape(b, batch_size, output_height * output_width,
    num_channel * kernel_height * kernel_width)
b = np.moveaxis(b, 1, 0)
```

These examples illustrate only several of countless neuron block definitions that can be produced through reordering. Some most intuitive reorders include the permutations of the channel, kernel height, and kernel width, with or without tiling. But theoretically speaking, any row or column reorder can be used.

It is worth noting that the order in the im2col matrix is orthogonal to memory layout in the sense that an im2col matrix can be laid out in many ways in memory. But for better data locality in clustering, it is often beneficial to keep elements in a neuron block consecutive in memory.

### 3.4 Reuse Directions

In addition to the systematic ways to define and explore reuse patterns, *generalized reuse* features the inclusion of a new reuse direction. In *deep reuse*, the reuse direction is always vertical in the im2col matrix: As Figure 3 illustrates, a neuron vector is compared against those neuron vectors in the same vertical panel to find similar ones in the clustering step. We call that *vertical reuse direction.*

*Generalized reuse* introduces a novel *horizontal reuse direction.* It compares a neuron vector against those in the same horizontal panel, and uses distributive property of tensor linear algebra to save computations.

We explain it using Figure 7. The main idea of horizontal reuse is that if $\mathbf{a}$ and $\mathbf{b}$ are similar, then we use

$$\mathbf{c} \times (\mathbf{w}_j + \mathbf{w}_k)$$

to approximate $\mathbf{a} \times \mathbf{w}_j + \mathbf{b} \times \mathbf{w}_k$, where the centroid vector $\mathbf{c} = (\mathbf{a} + \mathbf{b})/2$.

The *horizontal reuse* has four steps. First, after slicing $\mathbf{X} \in \mathbb{R}^{n\times m}$ into $\frac{n}{l}$ vertically concatenated sub-matrices, each sub-matrix $\mathbf{X}_i$ goes through a clustering step. In each sub-matrix in Figure 7, the four sub-neuron vectors are grouped into two clusters (denoted by two different colors), resulting in two centroids each. These centroid vectors in the clusters comprise the centroid matrices $\mathbf{X}_i^c$. Second, for each $\mathbf{X}_i$, we sum reduce the weight matrix $\mathbf{W}$ according to the clustering results of $\mathbf{x}_{i,j}$ and obtain the centroid weight matrix $\mathbf{W}_i^c$. We then compute $\mathbf{Y}_i = \mathbf{X}_i^c \times \mathbf{W}_i^c$. Finally, by concatenating $\mathbf{Y}_i$ we get the final output $\mathbf{Y}$.

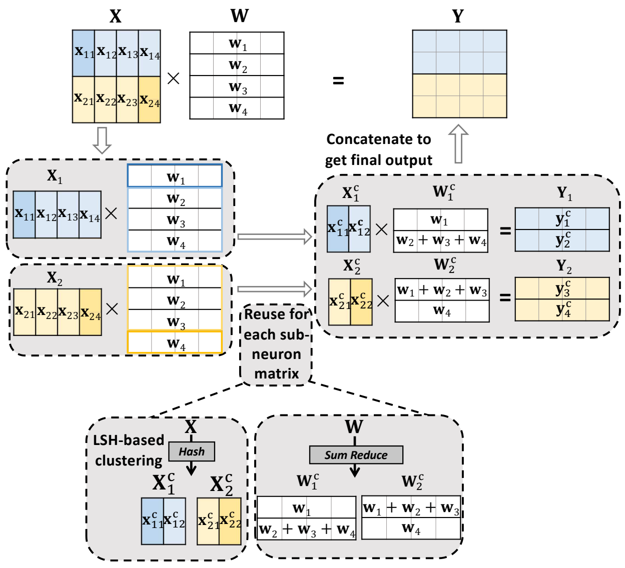

**Figure 7.** An illustration of *horizontal reuse* on *im2col* matrix $\mathbf{X}$. One column consists of two neuron vectors. Colors in matrix $\mathbf{X}$ indicate the similarities among the neuron vectors. Vectors $\mathbf{x}_{12}, \mathbf{x}_{13}, \mathbf{x}_{14}$, for instance, are similar and hence fall into one cluster with the centroid calculated as $\mathbf{x}_{12}^c$. Their outer product with $\mathbf{W}$ is replaced with the outer product of $\mathbf{x}_{12}^c$ and $\mathbf{w}_2 + \mathbf{w}_3 + \mathbf{w}_4$.

The introduction of *horizontal reuse direction* opens up new possible ways of reuse. Along with the *vertical reuse direction*, it enriches reuse patterns in *generalized reuse.*

**Relations between reuse unit definition and reuse direction.** The reuse unit definition and the reuse direction have a sequential relationship, where the reuse unit must be defined first, followed by determining the reuse direction these reuse units are clustered so we can reuse the centroid results. In the matrix view, a reuse unit definition identifies what a neuron block in the matrix consists of. By reordering the columns and rows in the matrix, a reuse unit comprises elements in different positions in the original matrix. Reuse direction determines in what direction those reuse units are compared and clustered (a reuse unit is compared against those reuse units in the same vertical panel as in Figure 3, or against those in the same horizontal panel as in Figure 7).

### 3.5 Reuse Granularity

Reuse granularity refers to the size of a neuron vector. In Figure 6, we have already shown the use of different granularities, which, along with reuse order, causes different reuse patterns.

One point worth mentioning is after expanding neuron vector to 2-D neuron block, a new phenomenon appears: As reuse granularity changes, the importance of an element in a neuron vector for reuse may also change, and the importance of different elements may differ. In the neuron vector ② in Figure 6 (d), for instance, elements $r_{12}$ and $g_{12}$ both appear twice; they hence weight more than other elements in the similarity calculation with other neuron vectors in clustering.

Overall, the granularity options are largely expanded in the 2-D case, which further enriches the reuse patterns.

### 3.6 Problem Formulation of Generalized Reuse

Before introducing the solution for generalized reuse, we first formally define the generalized reuse pattern optimization problem (GENERALIZED-REUSE) as follows.

*Given:* A dataset consisting of a training part $TRAIN$ and a test part $TEST$; a DNN model $M$; reuse pattern space $S$.

*Objective:* Using only $TRAIN$ dataset, find the reuse patterns in $S$ that lead to the Pareto optimal (w.r.t. the accuracy and latency of $M$ on $TEST$).

**Theorem 3.1.** *GENERALIZED-REUSE is NP-hard.*

## 4 Reuse Pattern Selection

For a given input, different reuse patterns could discover different amounts of similar tiles and hence let reuse save a different amount of computations. *Generalized reuse* dramatically expands the flexibility in reuse patterns, leading to more choices in reuse patterns and hence potentially better reuse and higher speed and/or accuracy for a given input. To realize the potential, however, it requires an effective way to determine the best reuse pattern. The issue is challenging especially for *generalized reuse* due to the much larger reuse space it enables. There are in addition some parameters in the reuse process (e.g., the number of hashing vectors in LSH) that also needs to be selected, which further exacerbate the selection difficulty.

Our strategy to address the problem is as follows:

(i) Make the reuse pattern selection for a dataset rather than each image. Ideally, the reuse pattern selection shall be done for every input, but it could introduce too much runtime overhead. In practice, an MCU device often works in a certain environment and the images it deals with share some commonalities. So using a set of historical images to offline find out a reuse pattern generally working well for those images could allow future images to use the pattern on the fly. This strategy offers a practical tradeoff.

(ii) Use an analytical-empirical combination to speed up the selection process. Even for offline reuse pattern selection, efficiency is critical. The reason is that the reuse space is huge but empirically checking the effectiveness of a reuse pattern is slow. To check the effectiveness of a reuse pattern empirically would require a training process of the DNN because the appropriate LSH hashing vectors need to be learned with the DNN training [17]. The conventional reuse-based DNNs simply rely on empirical checking to examine every candidate reuse pattern. It is impractical for general reuse for the much expanded reuse space. A combination of analytical models and empirical checking may drastically reduce the number of reuse patterns needed to go through the empirical checking.

Our solution is based on two novel analytical models we create, one for getting the bound of the accuracy loss due to the use of a reuse pattern, the other for approximating the impact of the reuse pattern on the inference latency of the DNN. These analytical models use some parameters that are determined through empirical measurements on the dataset; but those empirical measurements are lightweight, fast, and can happen on fast servers rather than slow MCUs. By using the results from the analytical models, the user can focus on a small set of promising reuse patterns, and then use the full empirical measurement to check the effectiveness of each of the reuse patterns in that set.

We next explain these two models first, and then describe the combined approach and the entire selection process.

### 4.1 Analytic Model of the Impact on Accuracy

Reuse is an approximation approach for general matrix multiplication (GEMM) computation. Let $\mathbf{Y}$ be the accurate result of $\mathbf{X}\times\mathbf{W}$ and $\hat{\mathbf{Y}}$ be the approximated result produced by reuse. An intuitive thought is that if the difference between $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ is small enough, then the impact of reuse on accuracy can be negligible.

Squared Frobenius norm ($\|\cdot\|_F^2$) is such a metric to quantify this error difference between $\mathbf{Y}$ and $\hat{\mathbf{Y}}$. $\|\mathbf{Y}-\hat{\mathbf{Y}}\|_F^2$ is defined as the squared sum of every element in the matrix $\mathbf{Y}-\hat{\mathbf{Y}}$. We next compute $\|\mathbf{Y}-\hat{\mathbf{Y}}\|_F^2$ by considering the L2-norm of each column separately (suppose we use vertical reuse here and a neuron vector is one row of $\mathbf{X}$).

For a column vector $\mathbf{y} = \mathbf{X}\cdot\mathbf{w}$ in the output matrix $\mathbf{Y}$, let $\hat{\mathbf{y}}$ be the reuse approximation of $\mathbf{y}$. Now consider each cluster group $\mathbf{X}^{(i)}$ after clustering the neuron vectors in $\mathbf{X}$. The difference in a position between $\mathbf{y}$ and $\hat{\mathbf{y}}$ comes from the practice of using the centroid vector in place of the original neuron vector. If we write out this quantity, we can rigorously prove that, for each cluster group, $\|\mathbf{y}-\hat{\mathbf{y}}\|^2$ is $m_i$ times $\mathbf{w}^\top\Sigma^{(i)}\mathbf{w}$ where $\Sigma^{(i)}$ is the covariance matrix of $\mathbf{X}^{(i)}$ and $m_i$ is the size of the cluster group. This value then proves to be upper bounded by the product of (i) the largest eigenvalue of the covariance matrix of $\mathbf{X}^{(i)}$, that is, $\lambda_{\max}^{(i)}$, (ii) $m_i$, and (iii)

$\|\mathbf{w}\|$. Summing up the upper bound for each cluster group, in turn, gives us the approximation error for the whole column.

For a more general case, a neuron vector can be part of the row of $\mathbf{X}$. In this case, $\mathbf{X}$ is sliced into $[\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_K]$ and $\mathbf{W}$ is sliced into $\begin{bmatrix}\mathbf{W}_1\\ \mathbf{W}_2\\ \vdots\\ \mathbf{W}_K\end{bmatrix}$. Consider each sub-matrix individually and we have the final approximation error $\|\mathbf{Y}-\hat{\mathbf{Y}}\|_F^2$ bounded by the squared Frobenius norm of the weight matrix ($\|\mathbf{W}_k\|_F^2$), largest eigenvalue of the covariance matrix for each cluster group, and the size of each cluster group:

$$\|\mathbf{Y}-\hat{\mathbf{Y}}\|_F^2 \le \sum_{k=1}^{K} \|\mathbf{W}_k\|_F^2 \sum_{i_k}^{N_{c_k}} \lambda_{\max}^{(i_k)} m_{i_k}.$$

The upper bound can then be used as an indicator of how well the reuse pattern can preserve the accuracy of the DNN. Note that the parameters $m_i$ in the upper bound come from a lightweight empirical measurement, where, a lightweight *deep reuse* is applied on the dataset. The lightweight *deep reuse* uses random hashing vectors rather than learned hashing vectors (which require a slow training process) for LSH clustering, and can run on servers, which help ensure the high speed of the lightweight checking. (For a more formal detailed description of the analytical model, see the supplementary materials.)

## 4.2 Analysis of Impact on Latency

*Generalized reuse* affects the inference latency in several aspects: (i) the time needed for im2col to create the corresponding data layout from the input image; (ii) the time to do clustering; (iii) the time to do the GEMM with the centroids; (iv) the time to reconstruct the output. The dominant influence of reuse pattern is on the third item because different reuse patterns lead to different number of centroids and hence the GEMM time. We hence focus our discussion on this aspect.

The impact is related with a concept called *redundancy ratio.* By grouping neuron vectors into clusters, the size of the input matrix is significantly reduced, allowing for lower computational complexity for the subsequent matrix multiplication. Suppose an input matrix $\mathbf{X}$ for GEMM (after im2col) is $N$ by $D_{\text{in}}$ and the weight matrix $\mathbf{W}$ is $D_{\text{in}}$ by $D_{\text{out}}$. The clustering overhead is an additional matrix multiplication $\mathbf{X}_i \cdot \mathbf{Hash}$ for each sub-matrix $\mathbf{X}_i$, where the hashing matrix is $L$ by $H$. Here, $H$ is the number of hashing vectors, and $L$ is the length of each hashing vector. Let $n_c$ denote the total number of centroids after clustering. The number of required computations is then reduced to $n_c$. For each input image or activation map, the benefits for removing the data redundancy can thus be measured by a *redundancy radio*, i.e., $r_t = 1 - \frac{n_c}{n}$ as the reduction of input matrix size. Here, $n = N \times K$ is the total number of neuron vectors where $K$ is the number of submatrices. In Figure 3 in Section 3.1, for example, the input matrix $\mathbf{X}$ is sliced into two sub-matrices, and there are in total four centroid vectors so $K$ is 2 and $n_c$ is 4 in this case. We conclude that $r_t$ indicates the fraction of redundancy within input images or activation maps. It can be used as the indicator of how effective a reuse pattern is in reducing the inference latency.

Similar to the case of the accuracy model, the measurement of $r_t$ is also through the same lightweight *deep reuse* running on servers.

It is worth noting that using this measure, one can analytically tell whether a reuse can indeed save computations. With the measure, the total floating-point operations (FLOPs) for a conventional GEMM-based convolution is $N \cdot D_{\text{in}} \cdot D_{\text{out}}$, whereas the total number of FLOPs after reuse is $\left(\frac{H}{D_{\text{out}}} + r_c\right) \cdot N \cdot D_{\text{in}} \cdot D_{\text{out}}$. We define $r_c = 1 - r_t$ for convenience and simplicity. The $\frac{H}{D_{\text{out}}}$ overhead comes from the additional hashing matrix multiplication. Therefore, in order for the reuse to expedite DNN inference (i.e., $\left(\frac{H}{D_{\text{out}}} + r_c\right) \cdot N \cdot D_{\text{in}} \cdot D_{\text{out}} < N \cdot D_{\text{in}} \cdot D_{\text{out}}$), the following *key condition* must hold: $\frac{H}{D_{\text{out}}} < r_t$. (As shown in the next section, the generalized reuse eliminates $r_t = 96\%$ computations on average). Note that generalized reuse typically removes more redundancies and thus brings greater speedups than TREC with the same level of approximation error as shown in Section 5.

## 4.3 Workflow for Reuse Pattern Selection

Figure 8 shows the full workflow of our analytical-empirical combined approach for reuse pattern selection. Based on the reuse space generation as described in Section 3, it generates a set of candidate reuse patterns according to a predefined *scope of reuse patterns.* The scope defines the set of reorders, reuse directions, and reuse granularities to consider. Our implemented framework has a default scope file that includes the most common options; that file is reconfigurable by users.

After that, the workflow uses the analytic models to get the error bounds and computation savings of the candidate reuse patterns. Based on those results, the *pattern selection* component identifies the pareto optimals and puts them into the set of promising patterns. The workflow then conducts the full check (including model training and LSH hash vector learning) on those promising patterns and finally outputs the best reuse patterns (i.e., the actual pareto optimals for the dataset).

## 5 Evaluation

We evaluate generalized reuse on four DNN networks that were used in the state-of-the-art (SOTA) work on reuse-based DNN optimizations [37] for direct comparisons. The evaluation is conducted in four folds. We first show the end-to-end

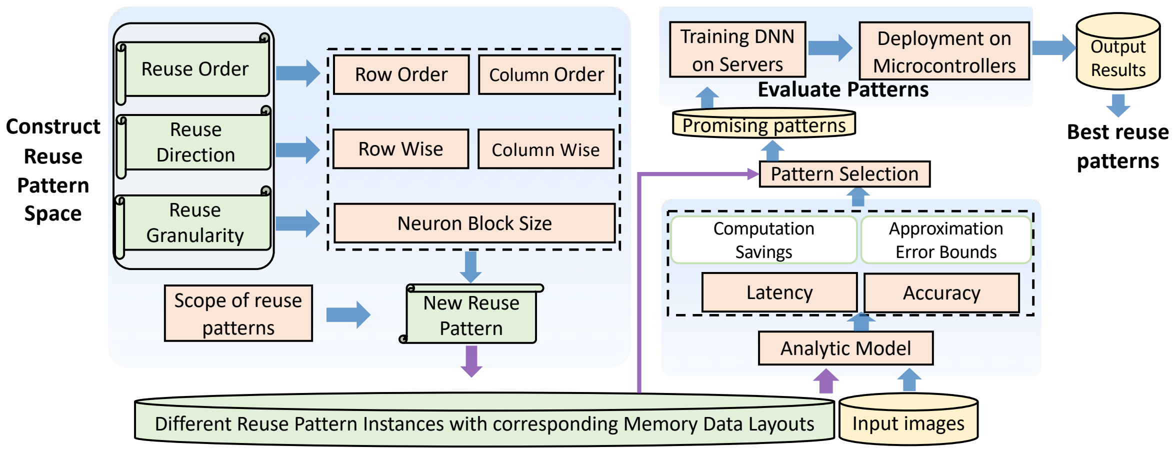

**Figure 8.** The workflow of analytical-empirical combined approach to reuse pattern selection.

performance benefits using two metrics, namely, accuracy and latency. We then provide an ablation study on the effect of our method. Specifically, we show the single-layer speedups and accuracy increase compared to the SOTA. We then analyze the performance impact of different reuse patterns and give insights based on these results. We also validate the efficacy of the analytic model to select the optimal reuse pattern for the given dataset.

### 5.1 Experimental Setup

**Hardware.** To check whether general reuse may consistently give benefits, we use two models of MCU, STM32F469I and STM32F767ZI, in our experiments. Both have SIMD extensions. The STM32F469I platform has a Cortex-M4 CPU, 324KB SRAM, and 2048KB Flash as memory storage. The STM32F767ZI board is more capable. It is equipped with a Cortex-M7 CPU, 512KB SRAM, and 2048KB Flash; it has a 20% faster clock than STM32F469I and can dual issue load and ALU instructions. The native DNN libraries `CMSIS-NN` [33] are installed on both. DNN training is performed on a server equipped with a 20-core 3.60GHz Intel Core i7-12700K CPU with 128GB RAM and an NVIDIA GeForce 4090 GPU with 24 GB memory. PyTorch 2.2.0 is used for training.

**DNN Models and Workloads.** Our experiments use four DNNs that were used in the SOTA [37] for direct comparisons, namely CifarNet [1], ZfNet [61], and two variants of SqueezeNet (with and without complex bypass [24]). These DNNs are popular on MCUs for their compactness and good accuracy. As in prior work, the public dataset CIFAR-10 [29] is used for training and evaluation. (Dataset ImageNet would run out of MCU memory.) All networks are optimized by SGD. The learning rate starts from 0.001 and decreases by 0.1 at 15-epoch intervals. The training batch size, weight decay, and momentum are set to 10, $10^{-4}$, and 0.95, respectively, and the maximum number of training iterations is set to 40.

**Comparison Counterparts.** We compare our method with the existing state-of-the-art reuse technique "TREC" [17] (denoted as `SOTA` in the rest part of this section). This reuse only considers the conventional reuse patterns (e.g., neuron vector defined in a single channel, vertical reuse direction, and 1-D vector-shape reuse granularity). In both SOTA and our method, the DNNs kernels are implemented with the native library CMSIS-NN [33] and SIMD instructions (e.g., 16-bit Multiply-and-Accumulate operations on ARM Cortex-M). The same common practices are followed in the implementation in both SOTA and our case: The DNN models are first transformed to fit MCUs, which includes the typical model pruning and quantization operations. Fixed-point weights, which reduce weight size from 32 bits to 8 bits, allow neural networks to run with minimal accuracy loss and faster operations on MCUs. This fixed-point format is especially useful for Cortex-M CPUs without floating-point units. Typical optimizations (e.g., fusion of the convolution and batch normalization) are applied in both SOTA and our case. (The code will be released to the public after the paper is accepted.) All performance results in our method include the im2col reordering costs. Pattern selection is conducted on the training set for each layer while the final evaluation is performed on the testing set. Finding an optimal pattern for each layer separately and combining them, however, can be sub-optimal as this is a global optimization problem; the full search space is the Cartesian product of the pattern spaces for each layer, which motivates the use of the analytic model (detailed in the supplementary material) in the pattern selection to quickly estimate the error bounds and computation savings.

### 5.2 End-to-End Performance

Figures 9 and 10 report the end-to-end performance. They show the results of four models using the reuse pattern chosen by our generalized reuse scheme compared to the original ones in SOTA. For each comparison, we show the models with a spectrum of accuracy and latency. The benefits of general reuse are clear.

First, it consistently reduces latency by 1.03–2.2× while maintaining the same level of accuracy. For instance, for "SqueezeNet (vanilla)" on STM32F4, when SOTA gets its highest accuracy (0.84), the latency is 1945ms, while at the same level of accuracy, our optimized model has a latency of 953ms, a 2.04× speedup. CifarNet and ZfNet have fewer convolution layers; the speedups are relatively modest, 1.03-1.21× at the same accuracy level. Overall, with no loss of accuracy, our method can achieve 1.03-2.04× speedups compared to SOTA across the two boards. Allowing for a minimum accuracy drop (0.005), our method can achieve an average of 1.2×, 1.5×, 2.0×, and 1.8× speedups for CifarNet, ZfNet, SqueezeNet (vanilla w/o bypass), and SqueezeNet (w/ bypass), respectively. When looking into individual pairs, there are even more speedups. More specifically, for CifarNet, ZfNet, SqueezeNet (vanilla w/o bypass), and SqueezeNet (w/ bypass), with a similar accuracy (such as 76.8%, 73%, 84%, and 84.2%), our method outperforms the SOTA by 1.1×, 1.3×, 2.2×, and 1.7× in terms of latency, respectively.

Second, with a similar latency (such as 200 ms, 1300 ms, 800 ms, and 850 ms), our method improves the accuracy by 1.8%, 8%, 3%, and 1%, respectively compared to the SOTA.

Third, note that STM32F7's total end-to-end inference time is less than half of that measured on STM32F4. 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 in clock speed, it results in the STM32F7 being roughly twice as fast as the STM32F4. The benefits of general reuse are significant on both MCUs, even more significant on the more powerful model Cortex-M7.

Fourth, when looking into the effects our methods bring to different networks, we find the most speedups in SqueezeNet. This is because there are more computation-intensive convolution layers in SqueezeNet, rendering a larger search space for reuse patterns. We have a more detailed analysis in the following section.

## 5.3 Ablation Study

We conduct an ablation study on the effect of general reuse. In detail, we investigate three aspects: (1) Performance impact of reuse patterns on single layers, (2) Analysis of the performance impact of different reuse patterns, and (3) Effectiveness of the analytic model.

**5.3.1 Detailed Single-Layer Performance.** Table 1 provides the single-layer performance benefits. All data are collected on the STM32F469I board. We make the following observations. First, choosing the optimal reuse pattern increases the model accuracy and, at the same time, brings speedups compared to the conventional reuse. For instance,

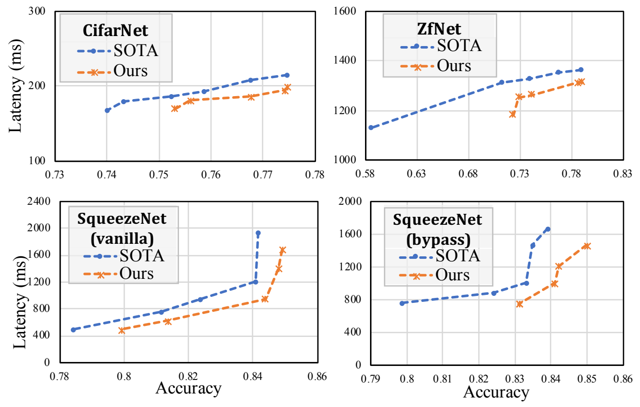

**Figure 9.** End-to-end results for different networks on STM32F4.

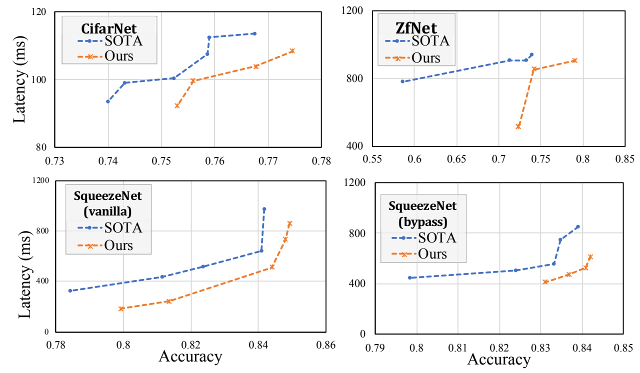

**Figure 10.** End-to-end results for different networks on STM32F7.

SqueezeNet achieves a 0.63% accuracy increase with over 2× speedups on average.

Second, when looking into the numbers, we see different speedups on different convolution layers. For SqueezeNet, for instance, `Fire2` 's activation map is larger than that of `Fire6`. That entails a larger room for better reuse patterns to exert their effects for `Fire2`, thus its greater speedups than that of `Fire6`.

Other findings include that a larger $L$ value typically leads to a greater speedup. This is because, if $L$ is larger, there are more variants of general reuse patterns to explore, and thus are more likely to include one with high accuracy and low latency. We also observe that vertical reuse gives the best results more often (over 80% time) than horizontal reuse.

**5.3.2 Analysis of Different Reuse Patterns.** In Figure 11 and Figure 12, we compare the details of the influence of different reuse patterns on different layers. Figure 11 shows the implication of reuse order on the model performance for CifarNet. The channel-last (i.e., the pattern similar to that in Figure 6 (b)) reuse pattern is better on Conv1, meaning more reuse opportunities are present within a channel on Conv1. For Conv2, however, channel-first reuse (i.e., the pattern

**Table 1.** Single-layer performance benefits. *Conf.* means configuration, $L$ is sub-matrix size, $H$ is the number of hash functions, and $D$ is the reuse direction (M-1 for vertical reuse and M-2 for horizontal reuse). $K$ is the kernel size and $M$ is the number of kernel channels. $r_t$ represents the redundancy ratio, and $\Delta R(\%)$ is the accuracy regret savings by choosing an optimal reuse pattern compared to the conventional reuse. Here, accuracy regret is defined as the difference between the accuracy we achieve with reuse and the original accuracy without reuse. ΔAcc is the difference between our method's accuracy and that of the SOTA.

**(a)** Single-layer performance of CifarNet.

| ConvLayer | $K$ | $M$ | Conf. |  |  | $r_t$ | Speedup (vs. CMSIS-NN) | Speedup (vs. Reuse) | $\Delta R$ (%) |
|---|---|---|---|---|---|---|---|---|---|
|  |  |  | $L$ | $H$ | $D$ |  |  |  |  |
| Conv1 | 75 | 64 | 15 | 4 | M-2 | 0.691 | 1.31× | 0.81× | 55.7 |
|  |  |  | 15 | 6 | M-1 | 0.948 | 1.61× | 1.00× | 52.4 |
|  |  |  | 20 | 3 | M-2 | 0.976 | 1.96× | 1.22× | 46.9 |
| Conv2 | 1600 | 64 | 20 | 3 | M-1 | 0.886 | 1.43× | 1.02× | 40.5 |
|  |  |  | 32 | 3 | M-1 | 0.862 | 1.59× | 1.14× | 37.3 |
|  |  |  | 20 | 1 | M-1 | 0.960 | 2.03× | 1.45× | -14.3 |
| Avg. |  |  |  |  |  | 0.887 | 1.66× | 1.11× | 36.4 |

**(b)** Single-layer performance of ZfNet.

| ConvLayer | $K$ | $M$ | Conf. |  |  | $r_t$ | Speedup (vs. Reuse) | ΔAcc (vs. Reuse) |
|---|---|---|---|---|---|---|---|---|
|  |  |  | $L$ | $H$ | $D$ |  |  |  |
| Conv1 | 147 | 96 | 21 | 10 | M-1 | 0.999 | 4.38× | 0.0358 |
| Conv2 | 2400 | 256 | 300 | 5 | M-1 | 0.997 | 1.30× | 0.0252 |
| Avg. |  |  |  |  |  | 0.998 | 2.84× | 0.0305 |

**(c)** Single-layer performance of SqueezeNet.

| ConvLayer | $K$ | $M$ | Conf. |  |  | $r_t$ | Speedup (vs. Reuse) | ΔAcc (vs. Reuse) |
|---|---|---|---|---|---|---|---|---|
|  |  |  | $L$ | $H$ | $D$ |  |  |  |
| Fire2.expand_3x3.conv | 144 | 64 | 24 | 2 | M-1 | 0.995 | 2.77× | 0.0139 |
|  |  |  | 24 | 2 | M-1 | 0.998 | 3.38× | 0.013 |
|  |  |  | 32 | 1 | M-1 | 0.998 | 4.16× | 0.0058 |
| Fire3.expand_3x3.conv | 144 | 64 | 20 | 5 | M-1 | 0.976 | 1.02× | 0.0094 |
|  |  |  | 24 | 5 | M-2 | 0.996 | 3.24× | 0.0063 |
|  |  |  | 24 | 5 | M-1 | 0.999 | 5.34× | 0.0025 |
| Fire4.expand_3x3.conv | 32 | 128 | 144 | 3 | M-2 | 0.993 | 2.04× | 0.0054 |
|  |  |  | 144 | 1 | M-2 | 0.999 | 2.65× | 0.0045 |
|  |  |  | 144 | 5 | M-1 | 0.999 | 2.92× | 0.0021 |
| Fire5.expand_3x3.conv | 288 | 128 | 32 | 2 | M-1 | 0.998 | 1.05× | 0.0039 |
|  |  |  | 40 | 2 | M-1 | 0.998 | 1.15× | 0.0015 |
|  |  |  | 50 | 2 | M-1 | 0.999 | 1.31× | 0.0012 |
| Fire6.expand_3x3.conv | 48 | 192 | 25 | 4 | M-1 | 0.990 | 1.31× | 0.0067 |
|  |  |  | 25 | 3 | M-1 | 0.995 | 1.42× | 0.006 |
|  |  |  | 25 | 2 | M-1 | 0.998 | 1.74× | 0.0065 |
| Fire7.expand_3x3.conv | 432 | 192 | 25 | 3 | M-1 | 0.976 | 1.33× | 0.011 |
|  |  |  | 25 | 2 | M-1 | 0.996 | 1.44× | 0.0108 |
|  |  |  | 25 | 1 | M-1 | 0.998 | 1.76× | 0.0103 |
| Fire8.expand_3x3.conv | 64 | 256 | 24 | 5 | M-1 | 0.995 | 2.18x | 0.0046 |
|  |  |  | 32 | 5 | M-1 | 0.996 | 2.64x | 0.0044 |
|  |  |  | 144 | 5 | M-2 | 0.998 | 2.68x | 0.0035 |
| Avg. |  |  |  |  |  | 0.995 | 2.06× | 0.0063 |

similar to that in Figure 6 (d)) can exploit more reuse opportunities and thus have better accuracy and latency results. This makes sense because the accuracy of reuse depends on the approximation error (the extent to which the centroid neuron matrix is similar to the original neuron matrix). For the original image, a channel expresses the image in R (or G, B) value; for a pixel, different channels represent different features of that position so it is better to explore reuse opportunities within a channel. After one convolution, the activation map becomes a more general representation of the original image, and the preferred reuse pattern is to view a position with all its channels.

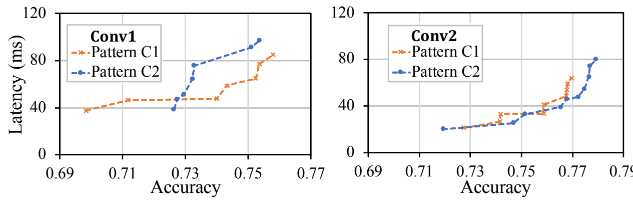

**Figure 11.** Experimental results of changing the reuse order (C1: channel last and C2: channel first) on CifarNet Conv1 and Conv2. The channel-last reuse pattern is better on Conv1 while the channel-first reuse pattern is better on Conv2.

Likewise, different layers have their own preferences for reuse direction in CifarNet (see Figure 12). Vertical reuse patterns show consistently better accuracy and latency on Conv2 than horizontal reuse patterns do. For Conv1, however, horizontal reuse patterns sometimes perform better.

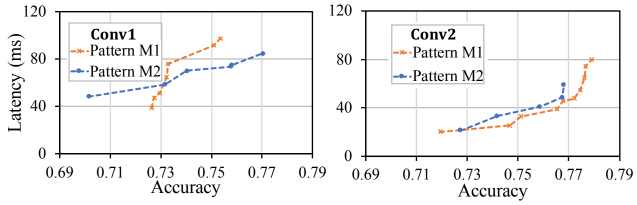

**Figure 12.** Experimental results of changing the reuse direction (M1: vertical reuse and M2: horizontal reuse) on CifarNet Conv1 and Conv2. Vertical reuse pattern is better on Conv2 while horizontal reuse pattern sometimes is better on Conv1.

Figure 13 provides results of five reuse patterns on CifarNet Conv1, illustrating how choosing different patterns influences the performance of a convolution layer. These results indicate that applying optimal reuse patterns can optimize both accuracy and latency. Users may select the desirable reuse patterns that best fit the requirements based on the Pareto optimal for different accuracy and latency requirements.

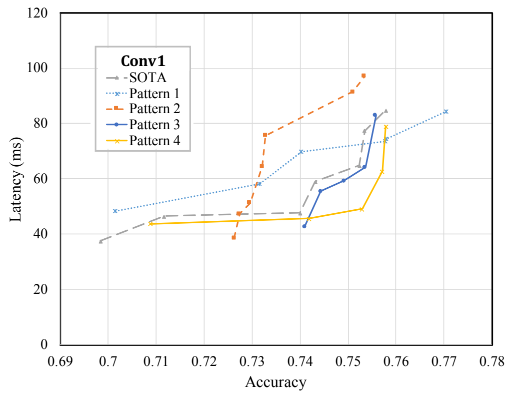

**Figure 13.** Experimental results of optimal *reuse pattern* on CifarNet Conv1.

**5.3.3 Effectiveness of the Analytic Model.** As shown in Section 3.6, solving the reuse pattern problem is NP-hard. Finding the theoretical optimal pattern in the vast reuse space remains an open problem. We hence resort to an empirical search space that contains 25 candidate reuse patterns, and enumerate each to obtain the upper bound to examine the room left if our reuse pattern selection method is used.

Figure 14 demonstrates the efficacy of our analytic models for choosing the best reuse patterns. Figure 14 compares the accuracy of our approach and those of two other strategies, as well as the empirical upper bound. Given a set of patterns, top-$k$ accuracy refers to the highest accuracy from the $k$ patterns we choose by using either our approach or the other two baseline methods. These two methods include random search, and one that uses redundancy ratio as heuristic indication of the potential quality of a reuse pattern and hence the possible degrees of accuracy loss if that reuse pattern is used. The analytic model needs much fewer trials (a smaller $k$) than the random strategy or heuristic method to obtain the best accuracy.

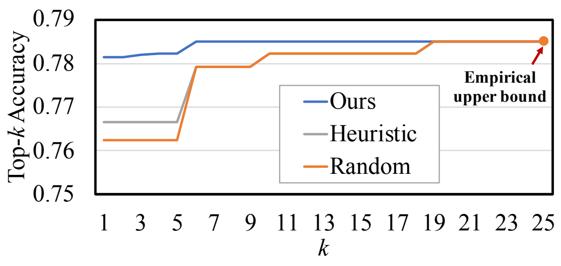

**Figure 14.** Experimental results of choosing optimal *reuse pattern* by the analytic model and random strategy; tested on CifarNet Conv2. For a given set of 25 reuse patterns, the figure shows the best accuracy we can achieve if we choose $k$ patterns using our analytic model (blue), based on heuristics (grey), or randomly (orange).

**5.3.4 Time Breakdown of the Exploration Process.** Table 2 shows the time breakdown of the exploration process of our analytical-empirical approach and its comparison to the standard full-fledged exploration. Consider 100 potential reuse patterns on SqueezeNet. Our approach first uses lightweight profiling to get parameters such as $m_i$ for each reuse pattern. We then use the analytic model to prune the space so the number of remaining patterns is reduced to 20. From that, we train those models on the server and, finally, run the models on the MCU device to measure the latency. The time breakdown of these four steps is detailed in Table 2. Compared to the standard exploration, that is, training models using all the potential patterns and measuring their latency, we manage to save 80% of the exploration time.

**Table 2.** Breakdown of the exploration process.

| | Our Method | Standard |
|---|---|---|
| Profiling | 700 s | – |
| Prune | <1 s | – |
| Training | 20×37 min | 100×37 min |
| Measuring on MCU | 6 min | 30 min |
| Total exploration time | ~12h | >60h |

**5.3.5 Performance Breakdown.** Table 3 shows the performance breakdowns for our method. All latencies are collected on the F4 board. From the table, we see that, after reuse reduces over 90% of computation, GEMM only constitutes a small portion of the total time (e.g., 20%), with much of the time spent on memory access operations such as im2col.

**Table 3.** Performance breakdown of reuse (unit: ms). Transformation includes im2col and layout transformation.

| Network | ConvLayer | Latency | Breakdown | | | |
|---|---|---|---|---|---|---|
| | | | Transformation | Clustering | GEMM | Recovering |
| Cifarnet | Conv1 | 50.07 | 15.82 | 17.3 | 3.8 | 13.15 |
| | Conv2 | 41.03 | 12.53 | 5 | 8 | 15.5 |
| SqueezeNet | Fire2.expand_3x3.conv | 45.57 | 29.33 | 9 | 2.4 | 4.84 |
| | Fire3.expand_3x3.conv | 54.4 | 29.08 | 17.92 | 2.4 | 5 |
| | Fire4.expand_3x3.conv | 52.8 | 28.26 | 2.72 | 11.9 | 9.92 |
| | Fire5.expand_3x3.conv | 74.7 | 65.34 | 3.6 | 5.76 | 40.5 |
| | Fire6.expand_3x3.conv | 25 | 8.8 | 2.4 | 13.8 | 2.76 |
| | Fire7.expand_3x3.conv | 49.5 | 25.6 | 11.5 | 12.4 | 19.32 |
| | Fire8.expand_3x3.conv | 42.4 | 21.14 | 10.26 | 11 | 15.12 |

**5.3.6 Out-of-distribution Data.** It is well known that out-of-distribution (OOD) data are challenging for DNN models to perform accurately [58]. Results in Table 4 show that, for the original CifarNet model trained on the cifar-10 dataset, the accuracy drops from over 70% to around 10% when the model is tested on the SVHN [40] dataset, an OOD dataset from the cifar-10 training set. We observe that, after the DNN model gets optimized with our generalized reuse, the

**Table 4.** OOD data performance for original CNN (cifarNet) and CNN with reuse. Models are trained on the in-distribution (ID) training data. Accuracy (ID) denotes accuracy for the in-distribution testset and accuracy (OOD) shows accuracy for the out-of-distribution dataset.

| Model | Dataset | | Accuracy | | Detection rate |
|---|---|---|---|---|---|
| | ID | OOD | ID | OOD | |
| Traditional CNN | cifar10 | svhn | 0.77 | 0.105 | 0.363 |
| CNN with reuse | | | 0.7434 | 0.0902 | 0.674 |

model's sensitivity to OOD stays at a similar level (only a slight drop from 10.5% to 9.02%).

In Machine Learning, there are various proposed methods to detect OOD samples. We experiment one of the common methods which uses the maximum softmax probability for detection: If the maximum probability from the output of the softmax of the DNN model on an input sample is below a certain threshold (0.7 in our experiment), the system reports OOD. The method shows a 36.3% detection rate on the original CifarNet on the OOD dataset svhn, but a significantly improved detection rate (67.4%) if the model is optimized with our reuse. This resonates with observations in machine learning: Approximate methods (reuse is one of them) encourage the model to focus on essential patterns in the input data, rather than get overfit to minor details [5], which in turn helps the model learn more generalizable features, and hence become more alert to OOD data as they typically do not have those features.

**5.3.7 Additional Experiments for Larger Model.** We show the results of ResNet-18 in Figure 15. We use the down-sampled ImageNet with 64×64 resolution since the original large images cause ResNet to run out of MCU memory. The learning rate is 0.01, while the training batch size and momentum are 100 and 0.8. All results are collected on the F4 board. Our method identifies the optimal reuse pattern that minimizes approximation error and removes more redundancy than SOTA, resulting in a 1.63× speedup and improved accuracy in individual layers (with the exception of Conv3-2). Our method can also reduce over 20% of end-to-end latency.

**5.3.8 Alternative Quantization Method.** Beside fixed-point quanitzation, we also evaluate our reuse method on the INT8 linear quantization of CifarNet, for both the weights and activations. Figure 16 illustrates the end-to-end performance on the F4 board. Our generalized reuse consistently outperforms the SOTA and improves accuracy by finding the reuse pattern that exploits data redundancy while keeping the approximation error minimal.

**5.3.9 Relation with Other Optimization Techniques.** Generalized reuse is orthogonal to optimization techniques such as pruning, quantization, and hyperparameter tuning.

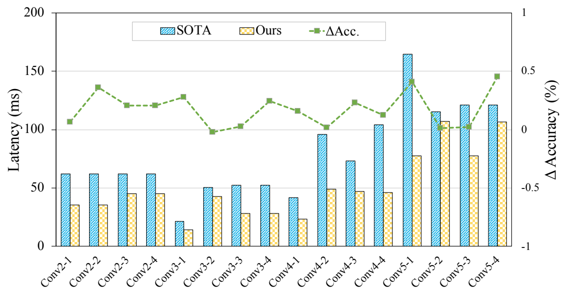

**Figure 15.** Experimental results of latency and accuracy performance for ResNet-18 on ImageNet-64x64. ΔAccuracy denotes the extra accuracy our method has over SOTA.

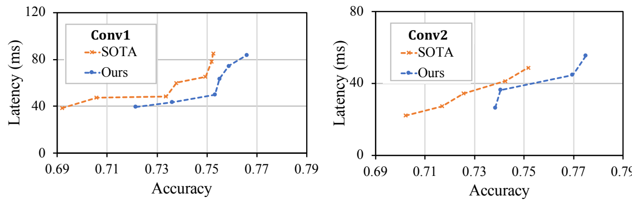

**Figure 16.** Experimental results of INT8 linear quantization.

Users can first apply these methods to transform their models and then apply reuse to that model to gain additional benefits. We provide an example where we apply channel pruning, fixed-point 8-bit quantization, and hyperparameter tuning for learning rate and momentum to the original model. For CifarNet, Table 5 demonstrates how the reuse technique serves as an efficient optimization strategy that preserves accuracy while providing further improvements when combined with these techniques.

**Table 5.** Experimental results of different tradeoff tools. CP refers to channel pruning, Q refers to quantization, and HPO refers to hyperparameter optimization.

| Technique | Accuracy | Latency (ms) | FLOPS |
|---|---|---|---|
| CP + Q + HPO | 0.78 | 217 | 15M |
| CP + Q + HPO + reuse | 0.76 | 187 | 6M |

## 6 Related Work

Computation reuse is an effective approach to saving computations and speeding up executions. Chen and others provide a comprehensive survey on exploiting data redundancy for optimization of Deep Learning [7]. A number of previous studies have explored the similarities in input to accelerate DNNs, through approaches in both hardware [8, 21, 31, 32] and software [10, 27, 34]. Hardware efforts [48] usually design special hardware to exploit input reuse. *Deep reuse* [43, 55, 67], a pure software approach, inspired this work. Different from the straight-forward reuse pattern in the Deep

Reuse, this work systematically characterizes the reuse pattern space and unlocks the potential benefits of reuse for DNN optimization.

DNN compression [19] that exploits the redundancy among DNN parameters, is orthogonal to input-level reuse. Some of the DNN compression techniques include quantization [60], squeezing filter size [12], conducting feature compression to activation map [50], more advanced neural architecture search and space optimization [2, 35, 36], and so on. These methods can substantially decrease the size of the model (i.e., weights and biases), the computational load, and the volume of data that needs to be transferred from the edge to the cloud. In resource-constrained microcontrollers, our design demonstrates that input reuse and model compression are implemented concurrently.

Another orthogonal approach to exploiting redundancy in data for speedups is direct processing on compressed data, that is, enabling data processing on compressed data without decompression. Because compression already eliminates many repetitions in content, the approach can automatically avoid repeated processing of the same content. This approach was initially proposed by Zhang and others for text analytics [64, 65, 67]. It was later extended to more general text processing tasks [66, 68], server and embedded GPUs [46, 62], graphs analytics [9] and deep learning [16].

## 7 Conclusion

This paper points out an important limitation on reuse patterns in prior reuse-based DNN optimizations, the narrow definitions of reuse patterns. It proposes the concept of *generalized reuse* that revolutionizes the conventional view of explorable reuse, and establishes a reuse space that characterizes a much broader range of reuse patterns. It uncovers the relations between reuse patterns and row/column reorder of a matrix view of the input or activation map of a DNN, based on which, it provides an easy way to generate various reuse patterns. It develops two novel analytical models for analyzing the impacts of reuse patterns on the accuracy and the latency of DNN inference, enabling efficient selection of appropriate reuse patterns. It empirically evaluates the effectiveness of the new solution on two models of MCUs, confirming the substantial benefits of the new solution in enabling efficient DNN inference across DNNs and MCU hardware.

## 8 Ackowledgements

We thank Dr. Trevor Erik Carlson for shepherding the final version of this paper and the anonymous reviewers for the constructive comments. This work was supported in part by the National Science Foundation (NSF) under the awards of CCF-2047516 (CAREER) and CCF2146873. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. NIH-1R01HD108473-01 and USDA P24-001771. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NIH or USDA.

## References

[1] CifarNet. http://places.csail.mit.edu/deepscene/small-projects/TRN-pytorch-pose/model_zoo/models/slim/nets/cifarnet.py, 2020.
[2] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3:517–532, 2021.
[3] Jesús Benito-Picazo, Enrique Domínguez, Esteban J Palomo, Ezequiel López-Rubio, and Juan Miguel Ortiz-de Lazcano-Lobato. Deep learning-based anomalous object detection system powered by microcontroller for ptz cameras. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2018.
[4] Neel Bhave, Aniket Dhagavkar, Kalpesh Dhande, Monis Bana, and Jyoti Joshi. Smart signal–adaptive traffic signal control using reinforcement learning and object detection. In *2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 624–628. IEEE, 2019.
[5] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.
[6] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. Mongoose: A learnable lsh framework for efficient neural network training. In *International Conference on Learning Representations*, 2021.
[7] Jou-An Chen, Wei Niu, Bin Ren, Yanzhi Wang, and Xipeng Shen. Survey: Exploiting data redundancy for optimization of deep learning. *ACM Comput. Surv.*, 55(10), February 2023.
[8] Kun-Chih Jimmy Chen, Yueh-Chi Yang, and Yi-Sheng Liao. An arbitrary kernel-size applicable noc-based dnn processor design with hybrid data reuse. In *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 657–660. IEEE, 2021.
[9] Zheng Chen, Feng Zhang, Jiawei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. Compressgraph: Efficient parallel graph analytics with rule-based compression. *Proceedings of the ACM on Management of Data*, 1(1):1–31, 2023.
[10] Nihat Mert Cicek, Xipeng Shen, and Ozcan Ozturk. Energy efficient boosting of gemm accelerators for dnn via reuse. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(5):1–26, 2022.
[11] Arm Company. Cortex®-m4 technical reference manual, 2010.
[12] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. Bottlenet: A deep learning architecture for intelligent mobile cloud computing services. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2019.
[13] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems*, 32, 2019.
[14] Igor Fedorov, Ramon Matas, Hokchhay Tann, Chuteng Zhou, Matthew Mattina, and Paul Whatmough. Udc: Unified dnas for compressible tinyml models for neural processing units. *Advances in Neural Information Processing Systems*, 35:18456–18471, 2022.
[15] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. Detect to track and track to detect. In *Proceedings of the IEEE international conference on computer vision*, pages 3038–3046, 2017.

[16] Hui Guan, Umang Chaudhary, Yuanchao Xu, Lin Ning, Lijun Zhang, and Xipeng Shen. Recurrent neural networks meet context-free grammar: Two birds with one stone. In *Proceedings of the 2021 IEEE International Conference on Data Mining (ICDM)*, pages 78–87. IEEE, 2021.

[17] Jiawei Guan, Feng Zhang, Jiesong Liu, Hsin-Hsuan Sung, Ruofan Wu, Xiaoyong Du, and Xipeng Shen. Trec: Transient redundancy elimination-based convolution. In *Neural Information Processing Systems 35 (Neurips 2022)*, 2022.

[18] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. Protonn: Compressed and accurate knn for resource-scarce devices. In *International Conference on Machine Learning*, pages 1331–1340. PMLR, 2017.

[19] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[20] Wei Han, Pooya Khorrami, Tom Le Paine, Prajit Ramachandran, Mohammad Babaeizadeh, Honghui Shi, Jianan Li, Shuicheng Yan, and Thomas S Huang. Seq-NMS for Video Object Detection. *arXiv preprint arXiv:1602.08465*, 2016.

[21] Edward Hanson, Shiyu Li, Hai'Helen' Li, and Yiran Chen. Cascading structured pruning: enabling high data reuse for sparse dnn accelerators. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 522–535, 2022.

[22] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network (2015). *arXiv preprint arXiv:1503.02531*, 2, 2015.

[23] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[24] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

[25] Sunil Jacob, Varun G Menon, Fadi Al-Turjman, PG Vinoj, and Leonardo Mostarda. Artificial muscle intelligence system with deep learning for post-stroke assistance and rehabilitation. *Ieee Access*, 7:133463–133473, 2019.

[26] Kai Kang, Hongsheng Li, Junjie Yan, Xingyu Zeng, Bin Yang, Tong Xiao, Cong Zhang, Zhe Wang, Ruohui Wang, and Xiaogang Wang. T-CNN: Tubelets with Convolutional Neural Networks for Object Detection from Videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 28(10):2896–2907, 2017.

[27] Jungwoo Kim, Seonjin Na, Sanghyeon Lee, Sunho Lee, and Jaehyuk Huh. Improving data reuse in npu on-chip memory with interleaved gradient order for dnn training. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 438–451, 2023.

[28] Aliaksei Kolesau and Dmitrij Šešok. Voice activation systems for embedded devices: Systematic literature review. *Informatica*, 31(1):65–88, 2020.

[29] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[30] Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 kb ram for the internet of things. In *International Conference on Machine Learning*, pages 1935–1944. PMLR, 2017.

[31] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768, 2019.

[32] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[33] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[34] Yuanchun Li, Ziqi Zhang, Bingyan Liu, Ziyue Yang, and Yunxin Liu. Modeldiff: Testing-based dnn similarity comparison for model reuse detection. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 139–151, 2021.

[35] Ji Lin, Wei-Ming Chen, Yujun Lin, Han Cai, Chuang Gan, and Song Han. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. *arXiv preprint arXiv:2110.15352*, 2021.

[36] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020.

[37] Jiesong Liu, Feng Zhang, Jiawei Guan, Hsin-Hsuan Sung, Xiaoguang Guo, Xiaoyong Du, and Xipeng Shen. Space-efficient trec for enabling deep learning on microcontrollers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 644–659, 2023.

[38] Jiesong Liu, Feng Zhang, Jiawei Guan, Hsin-Hsuan Sung, Xiaoguang Guo, Saiqin Long, Xiaoyong Du, and Xipeng Shen. Enabling efficient deep learning on mcu with transient redundancy elimination. *IEEE Transactions on Computers*, 2024.

[39] Simon Mittermaier, Ludwig Kürzinger, Bernd Waschneck, and Gerhard Rigoll. Small-footprint keyword spotting on raw audio data with sinc-convolutions. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7454–7458. IEEE, 2020.

[40] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[41] Lin Ning, Hui Guan, and Xipeng Shen. Adaptive Deep Reuse: Accelerating CNN training on the fly. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1538–1549. IEEE, 2019.

[42] Lin Ning and Xipeng Shen. Deep Reuse: streamline CNN inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing*, pages 438–448, 2019.

[43] Lin Ning and Xipeng Shen. Deep reuse: streamline cnn inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing*, pages 438–448, 2019.

[44] Nefy Puteri Novani, Mohammad Hafiz Hersyah, and Ryon Hamdanu. Electrical household appliances control using voice command based on microcontroller. In *2020 International Conference on Information Technology Systems and Innovation (ICITSI)*, pages 288–293. IEEE, 2020.

[45] Zaifeng Pan, Feng Zhang, Hourun Li, Chenyang Zhang, Xiaoyong Du, and Dong Deng. G-slide: A gpu-based sub-linear deep learning engine via lsh sparsification. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):3015–3027, 2022.

[46] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. Exploring data analytics without decompression on embedded gpu systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(12):2950–2964, 2021.

[47] Zheng Qin, Zhaoning Zhang, Xiaotao Chen, Changjian Wang, and Yuxing Peng. Fd-mobilenet: Improved mobilenet with a fast downsampling strategy. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 1363–1367. IEEE, 2018.

[48] Marc Riera, Jose-Maria Arnau, and Antonio Gonzalez. Computation reuse in dnns by exploiting input similarity. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 57–68, 2018.

[49] Falk Salewski and Stefan Kowalewski. Hardware/software design considerations for automotive embedded systems. *IEEE Transactions*

*on Industrial Informatics*, 4(3):156–163, 2008.

[50] Jiawei Shao and Jun Zhang. Bottlenet++: An end-to-end approach for feature compression in device-edge co-inference systems. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2020.

[51] Prerna Sharma and Deepali Kamthania. Intelligent object detection and avoidance system. In *International Conference on Transforming IDEAS (Inter-Disciplinary Exchanges, Analysis, and Search) into Viable Solutions*, pages 342–351, 2019.

[52] Stanislava Soro. Tinyml for ubiquitous edge ai. *arXiv preprint arXiv:2102.01255*, 2021.

[53] Srinivasa R Sridhara. Ultra-low power microcontrollers for portable, wearable, and implantable medical electronics. In *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, pages 556–560. IEEE, 2011.

[54] Hidetoshi Teraoka, Fumiharu Nakahara, and Kenichi Kurosawa. Incremental update method for vehicle microcontrollers. In *2017 IEEE 6th Global Conference on Consumer Electronics (GCCE)*, pages 1–2. IEEE, 2017.

[55] Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. Drew: Efficient winograd cnn inference with deep reuse. In *Proceedings of the ACM Web Conference 2022*, pages 1807–1816, 2022.

[56] Fanyi Xiao and Yong Jae Lee. Video object detection with an aligned spatial-temporal memory. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 485–501, 2018.

[57] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1(4):216–222, 2018.

[58] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu. Generalized out-of-distribution detection: A survey. *International Journal of Computer Vision*, pages 1–28, 2024.

[59] JZ Yi, YK Tan, ZR Ang, and SK Panda. Microcontroller based voice-activated powered wheelchair control. In *Proceedings of the 1st international convention on Rehabilitation engineering & assistive technology: in conjunction with 1st Tan Tock Seng Hospital Neurorehabilitation Meeting*, pages 67–72, 2007.

[60] Jian Yuan, Kok Kiong Tan, Tong Heng Lee, and Gerald Choon Huat Koh. Power-efficient interrupt-driven algorithms for fall detection and classification of activities of daily living. *IEEE Sensors Journal*, 15(3):1377–1387, 2014.

[61] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[62] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. G-tadoc: Enabling efficient gpu-based text analytics without decompression. In *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE)*, pages 1616–1627. IEEE, 2021.

[63] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):905–918, 2016.

[64] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. In *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*, pages 1296–1309. VLDB Endowment, 2018.

[65] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. Zwift: A programming framework for high performance text analytics on compressed data. In *Proceedings of the 32nd ACM International Conference on Supercomputing (ICS)*, pages 347–358. ACM, 2018.

[66] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. Enabling efficient random access to hierarchically compressed data. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE)*, pages 841–852. IEEE, 2020.

[67] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. POCLib: a high-performance framework for enabling near orthogonal processing on compression. *IEEE Transactions on Parallel and Distributed Systems*, 33(2):459–475, 2022.

[68] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. TADOC: Text analytics directly on compression. *The VLDB Journal*, 30(2):163–188, 2020.

[69] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*, 2017.

[70] Xizhou Zhu, Yujie Wang, Jifeng Dai, Lu Yuan, and Yichen Wei. Flow-guided feature aggregation for video object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 408–417, 2017.

[71] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. Deep feature flow for video recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2349–2358, 2017.