Bit-serial Weight Pools: Compression and Arbitrary Precision Execution of Neural Networks on Resource Constrained Processors

Shurui Li [1] **Puneet Gupta** [1]

Abstract

Applications of neural networks on edge systems have proliferated in recent years but the ever-increasing model size makes neural networks not able to deploy on resource-constrained microcontrollers efficiently. We propose bit-serial weight pools, an end-to-end framework that includes network compression and acceleration of arbitrary sub-byte precision. The framework can achieve up to $8\times$ compression compared to 8-bit networks by sharing a pool of weights across the entire network. We further propose a bit-serial lookup based software implementation that allows runtime-bitwidth trade-off and is able to achieve more than $2.8\times$ speedup and $7.5\times$ storage compression compared to 8-bit networks, with less than 1% accuracy drop.

1 Introduction

The ever-increasing size of neural network models and rapid proliferation of machine learning in resource-constrained edge devices have catalyzed research into a variety of model compression techniques, as well as software and hardware acceleration of deep learning on edge devices.

General-purpose microcontrollers have been a platform of choice for edge devices due to their low power, low cost and programmability. However, this comes at the cost of limited memory: these processors usually do not have any DRAM and often have less than 2MB total memory (SRAM + Flash); and small available compute power: these processors usually have small datapaths and simple pipelines running at modest clock rates. This makes the execution of complex machine learning models on this ubiquitous class of processors very challenging. A variety of model compression techniques have, therefore, garnered attention in the embedded machine learning community (Berthelier et al., 2021).

Weight sharing (Nowlan & Hinton, 1992) as a model compression technique shares a set of weight vectors across the entire neural network, so that only the indices of the shared weight vectors need to be stored, instead of actual weight values. For convolutional neural networks (CNNs), weight sharing methods can achieve compression ratios between 4-16x, compared to 8-bit baselines. Since weight sharing does not modify the structure nor the precision of the network, it can be combined with other compression techniques like pruning and quantization to further improve compression ratio and runtime. Furthermore, recent works (Choi et al., 2018; Banner et al., 2018) have shown that sub-byte quantization of weights and/or activations can achieve inference accuracy comparable to full-precision networks.

Though weight sharing and sub-byte quantization are both promising for storage and runtime improvement, neither has native support in microcontroller-class general purpose processors commonly deployed in edge devices. As a result, these compression techniques can often hurt performance rather than improve it. For instance, processing a neural network with sub-byte precision naively can lead to worse runtime due to bit unpacking overhead (Hu et al., 2018). Hence, there is a need for optimized software implementations of weight-shared neural networks, as well as methods that can support and accelerate sub-byte precision neural networks on microcontrollers.

In this work, we present a framework for efficiently deploying large neural networks on small microcontrollers. The proposed framework contains two parts. The first part is neural network compression, where a pool of weight vectors (e.g., a 1×8 8-tuple of weights) along channel dimension are shared across the entire network. We refer to networks using our weight sharing method as weight pool networks in the rest of this paper. The second part of the framework is the software implementation of weight pool networks on microcontrollers, where we utilize bit-serial lookup tables to support and accelerate weight pool networks with 8-bit

[1]Department of Electrical and Computer Engineering, University of California, Los Angeles, California, USA. Correspondence to: Shurui Li <shuruili@ucla.edu>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

or lower activation bitwidth. The main contributions can be summarized as follows.

- We show that z-dimension weight pools, as small as 512 total parameters can realize popular networks such as ResNet and MobileNet with negligible accuracy loss.
- We develop a bit-serial lookup based method for efficient arbitrary-precision execution of weight pool networks on general purpose microcontrollers. This delivers 2.38X speedup (compared to well-optimized ARM CMSIS-NN library (Lai et al., 2018)) at 8-bit precision and even greater speedup at lower bitwidth on popular neural networks.
- We explore the design-space of weight pool networks experimentally to develop an optimized software implementation of weight pool networks targeted for small, memory-starved microcontrollers.
- We show that weight pool arbitrary precision networks can be 2.8X faster and 6.51X more compact than CMSIS on ResNet-10, with less than 1 percent drop of accuracy on CIFAR-10, and better compression and speedup can be achieved on larger networks.

The next section outlines the motivation behind the bit-serial weight pool approach.

2 Addressing Compression and Quantization Challenges for General Purpose Processors

Compression with weight pools. Our weight pool networks essentially store vectors of weights along the channel dimension as one entry. The 3D filters used in CNNs would then be composed of these vectors. For instance, a $3 \times 3 \times 32$ filter would use $3 \times 3 \times 4 (= 36) 1 \times 8$ weight vectors selected from the available pool of weight vectors. There is no limitation on the reuse of vectors. Weight pool networks would reduce the parameter storage from the total number of parameters in the network to the total size of the weight pool. If done correctly, this can reduce parameter storage requirements of neural networks by orders of magnitude with minimal accuracy drop. Furthermore, the parameter storage here becomes independent of network size.

However, naively implementing weight pool networks would likely worsen inference latency because of additional memory reads (some form of index storage lookup followed by the actual weight lookup) with no reduction in total number of operations. One could try reducing the number of operations by directly storing the results of the (partial) dot product on the weight pools. For a pool vector size of 8, it would replace 8 multiply-accumulate operations with one memory lookup. Unfortunately, for 8-bit activations, this would require a lookup table size of 2^{8^8} entries for just one pool vector which is impractical.

Arbitrary precision computation using bit-serial arithmetic. Like conventional neural networks, the activation bitwidth of weight pool networks can be reduced to sub-byte regions while still achieving decent accuracy on many tasks. The sub-byte activation bitwidth provides an opportunity to improve the runtime and overall energy efficiency.

Sub-byte precision is not well supported in most microcontrollers (or most processors in general). Naively implementing networks with sub-byte activation bitwidth is not useful as it would worsen runtime because of the bit unpacking overhead with no actual compute reduction (since underlying hardware still executes higher precision arithmetic).

To support and accelerate neural networks with sub-byte activation bitwidth, bit-serial multiplication seems to be a suitable candidate since it processes a multiplication serially by looping through all the bits of one operand. The runtime of bit-serial multiplication is proportional to the bitwidth of the bit-unrolled operand. There are many bit-serial multiplication based hardware neural network accelerators (Li et al., 2021; Judd et al., 2016; Sharma et al., 2018), but there is no support of bit-serial multiplication in microcontrollers due to the lack of bit-serial multipliers.

Bit-serial-lookup-based weight pool networks. We address the challenges outlined above by doing bit-serial execution *but* saving computation by lookup of partial dot product results on pool vectors. Since activations are processed one bit at a time (most significant to least significant bit), the dot product lookups only need to be on 1-bit operands. Therefore, the lookup table for activation bitwidth of 8 bits is just 2^8 entries. This would replace 8 multiply-accumulate operations with 8 memory reads and accumulations. Later we show how despite this, substantial runtime reduction can be achieved by careful implementation optimizations leveraging the value reuse properties of weight pools. Furthermore, reducing activation bitwidth now just amounts to truncating the temporal bit-serial execution earlier which gives proportionate further runtime improvement.

3 Bit-Serial Weight Pool Methodology

Figure 1 shows the high-level flow of the proposed framework, which is split into two parts. The left block shows the compression part, where the input is a pretrained CNN. The corresponding weight pool and weight indices (original weights are converted to indices of the weight pool) are generated and the pretrained CNN is hence compressed. Analysis of minimum activation bitwidth of the compressed CNN is carried out afterward. Finally, the dot product lookup table is generated from the weight pool, and loaded into microcontrollers' flash memory along with weight indices and precision information. The compression part is entirely executed on the host side and the generated weight pool CNN is sent to the microcontroller.

The second part is CNN inference acceleration, which is executed on the microcontroller. At this stage, the original CNN has already been compressed and transformed into weight pool CNN, and the activation bitwidth has been determined. The framework uses a bit-serial lookup table based algorithm to accelerate the inference of weight pool CNNs, and is able to further improve runtime by reducing the activation bitwidth. The rest of this section describes

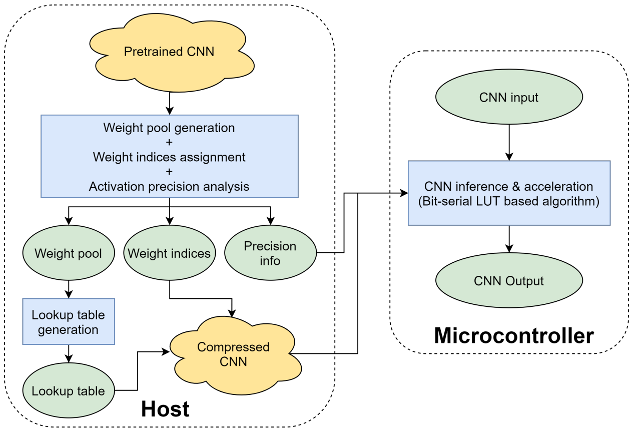
Pretrained CNN
Weight pool generation + Weight indices assignment + Activation precision analysis
Weight pool
Weight indices
Precision info
Lookup table generation
Lookup table
Compressed CNN
Host
CNN input
CNN inference & acceleration (Bit-serial LUT based algorithm)
CNN Output
Microcontroller

Figure 1. High level flow of the proposed framework. Pretrained weights are clustered into weight vectors pool, any fine tuning and activation bitwidth selection are done offline. At inference time, the processor only stores the weight pool dot product results and indices to weight vectors used in the network.

each of these steps in detail.

Weight pool networks achieve compression by sharing a fixed pool of weight vectors among all the layers of a network, so that the network only needs to store indices of the weight pool, plus the weight pool itself. In this work we use a weight sharing pipeline similar to (Son et al., 2018) to generate weight pool CNNs, but instead of clustering 2D convolutional kernels, we apply the clustering algorithm along the z-dimension of a 3D filter (clustering across the filter channels) as shown in Figure 3. Figure 2 shows the proposed training pipeline. The pretrained weights are grouped into 1×8 weight vectors along the channel dimension and clustered using K-means clustering (with a cosine distance metric to avoid scaling dependence). After the clustered weight pool is generated, the original CNN's weights are converted to the indices of the weight vectors in the weight pool. The network is retrained to fine-tune the weight in-

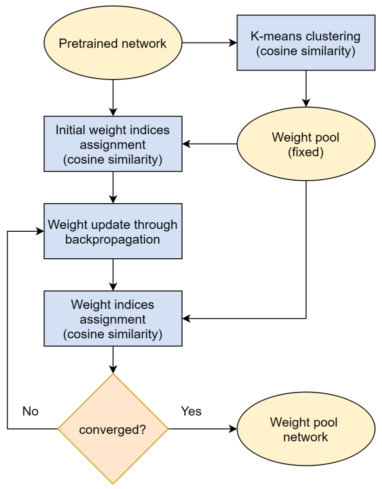
Pretrained network
K-means clustering (cosine similarity)
Initial weight indices assignment (cosine similarity)
Weight pool (fixed)
Weight update through backpropagation
Weight indices assignment (cosine similarity)
No
converged?
Yes
Weight pool network

Figure 2. Overall flow of generating a weight pool network from a pretrained network.

dices assignment (with a fixed weight pool) and fully connected layer's weights. The backward pass updates the network weights and the forward pass reassigns indices to the nearest weight pool vector. Weight pool network may be further fine-tuned, if needed, for reduced activation bitwidth.

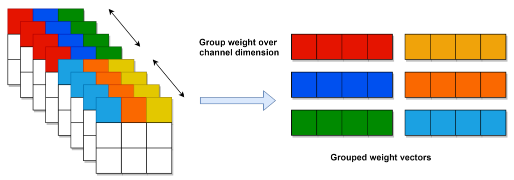

Figure 3. Visualization of the z-dimension weight grouping. This example shows a $8 \times 3 \times 3$ filter with weight vector size of 4. The weights are grouped in the channel dimension and same color represent weights in a single group. After the z-dimension grouping, 18 $4 \times 1 \times 1$ weight vectors (6 are shown in the figure) are generated for the given filter.

To show the effectiveness of the z-dimension weight pool and determine the optimal pool size, we benchmark the 3×3 kernel weight pool (xy-dimension weight pool) with and without scaling coefficient, as well as the proposed z-dimension weight pool using ResNet-14 (modified ResNet-18 (He et al., 2016) with last block truncated) on the CIFAR-10 dataset. For each setup three weight pool size are tested. The result is shown in Figure 4. For all three weight pool sizes, the z-dimension weight pool performs slightly better than the xy-dimension weight pool with coefficients and significantly better than the xy-dimension weight pool without scaling coefficients. Regarding the pool size, 64 is enough for this network and 32 also achieves a decent result.

The reason for the better accuracy of the z-dimension weight pool is more weights are grouped together in the

Group size	4	8	16
Accuracy (%)	91.22	91.13	87.96

Table 1. Accuracy of z-dimension weight pool with different group size. The network is ResNet-14 and dataset is CIFAR-10. Original network accuracy is 92.26%.

xy-dimension weight pool than the z-dimension (9 vs 8). Considering a 3x3 convolution layer with weight shape (8,8,3,3), the total number of possible unique weight vectors for 64 weight pool size is 64^{72} for z-dimension and 64^{64} for xy-dimension. Another reason might be if a certain 2D kernel (a channel of an entire filter) has high importance, the z-dimension weight pool can closely reconstruct this kernel by sacrificing other channels, while for xy-dimension it can only be directly chosen from the weight pool.

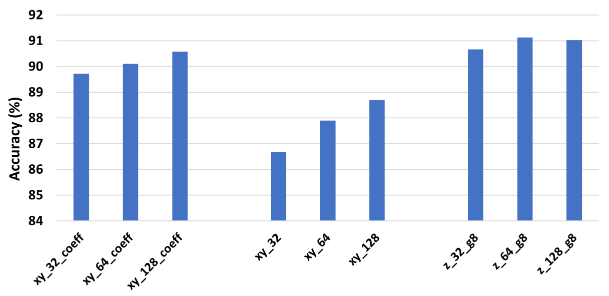

Figure 4. Accuracy of weight pool ResNet-14 with different setups, on the CIFAR-10 dataset. For a weight pool with 3×3 kernels, its setups are denoted by xy_n_(coeff), where n means the weight pool size (how many weight vectors in the weight pool) and coeff means the version with scaling coefficients. For the z-dimension weight pool, the setups are denoted by z_n_g8, where n is the weight pool size and g8 means the weight vector size (group size) is 8. The original accuracy is 92.26%.

Table 1 shows the accuracy results of different group size (weight vector size) for z-dimension weight pool on ResNet-14. Clearly, a group size of 8 achieves a good balance between compression ratio and network accuracy. We choose 8 as the default group size so the weight pool contains multiple 1×8 weight vectors. Compared to clustering 3×3 kernels, clustering along z-dimension has a few advantages:

- It achieves the same or better network performance (accuracy) without the additional scaling coefficient as used in (Son et al., 2018), which improves the compression ratio from $4.5\times$ (clustering 3×3 kernels) to $8\times$ over an 8-bit network.

- It is more flexible. It can fit networks with arbitrary kernel sizes including 1×1 kernels, and can apply to fully connected layers as well.

The main rationale behind our choice of using the z-dimension weight pool is not its accuracy but its flexibility. It can work on all filter sizes including 1×1 filters, while the xy-dimension weight pool only works on 3×3 filters. The accuracy for the xy-dimension weight pool is severely impacted for 5x5 filters due to the reduction in representability (>10% accuracy drop on CIFAR-10).

Grouping weights along z-dimension for layers with depth less than 8 (e.g., typical input layers in image CNNs) incurs underutilization. In most, if not all popular CNNs, such reduced depth layers account for a small fraction of storage and compute. Therefore, we choose to keep such layers (usually just the first layer) uncompressed for better inference accuracy. Not compressing the first layer has minimal impact on compression ratio and runtime for most CNNs since the first layer usually just have three input channels. Another alternative can be grouping all the channels together and zero pad the vector size to 8.

Although the main focus of this work is compressing and accelerating CNNs, we apply the weight pool compression on one dense network to demonstrate the generalization capability of weight pool compression. We evaluate a 3-layer dense network (784-256-128-10) using the FashionMNIST dataset. The original accuracy is 88.65% and after weight pool compression (64 vectors) the accuracy is 88.01% ($< 1\%$ reduction). This is a promising result for adopting the weight pool compression to other types of networks.

3.1 Lookup Table Based Bit-serial Computation

As introduced in section 2, lookup tables can be used to accelerate convolutions by looking up the vector dot product results directly from memory, instead of computing them. Lookup table offers a trade-off between space complexity and time complexity, and can improve runtime when the memory is large enough and fast enough. However, for dot product operations, the size of lookup table can be huge. Consider the dot product between two 8-element vectors with 8-bit precision, the total number of entries required for the lookup table is $2^{8^{2^8}} = 3.40 \times 10^{38}$. Clearly, such lookup table implementation is not feasible unless the lookup table size can be massively shrunk.

The huge lookup table size is partly caused by both inputs having no restriction on their values, leading to 65536 total input combinations for a simple two-input multiplication. However, this is not the case for weigh-pool networks. Unlike normal neural networks where inputs and weights can be any possible values, weights are fixed for weight pool networks, meaning a single 8-bit multiplication only requires 256 lookup table entries. The lookup table size for the aforementioned 8-element dot product operation with weight fixed is 1.84×10^{19} entries, which is significantly smaller than 3.40×10^{38}, but still impractical.

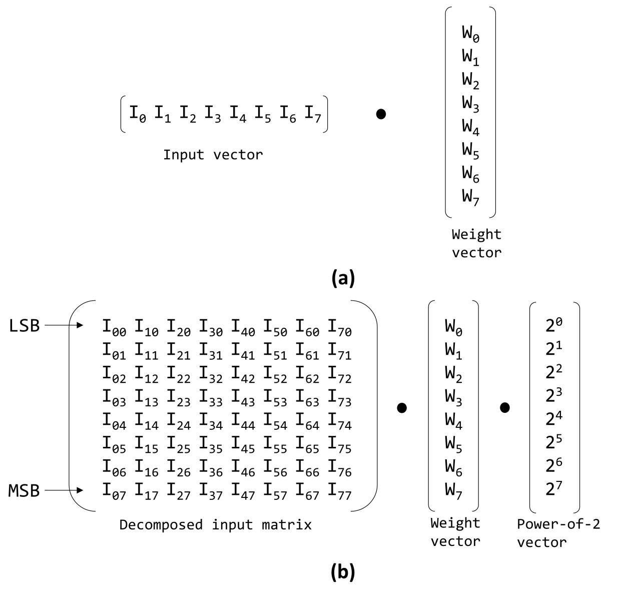

Figure 5. Visualization of the bit decomposition step. (a): The original 8-element dot product between input and weight vectors. (b): The original dot product is transformed into matrix-vector multiplication followed by dot product after bit decomposition. I_{mn} means the n^{th} bit (starting from LSB) of the m^{th} element. The original input vector is decomposed into an 8×8 matrix with each element representing a single bit. Each column represents all the bits of an input value while each row represents a unique bit position of all input values. The weight vector is kept the same and is multiplied with all the bit positions of input. The result of the matrix-vector multiplication should be the dot product of input and weight vector at every input bit position. The result is then multiplied with the power-of-two vector which represents bit weights to generate the final dot product result.

To further reduce the lookup table size and support bit-serial multiplication, a key step in our proposed method is *bit decomposition*. For an N-element dot product between input and weight vector (both M bits), the dot product between input (activation) vector and weight vector can be calculated as:

$$\vec{a} \cdot \vec{w} = \sum_{i=0}^{N-1} a_i \times w_i \tag{1}$$

Where a_i and w_i are the i-th elements of vectors $\vec{a}$ and $\vec{w}$ respectively and N is the width of the dot product. The input element a_i can be decomposed as:

$$a_i = \sum_{j=0}^{M-1} 2^j \times a_i[j] \tag{2}$$

Where $a_i[j]$ is the j-th bit (from LSB) of activation a_i, and M is the bitwidth of the activation. Hence each input element is decomposed into M binary values each representing a single bit, and the input vector is hence decomposed into an $M \times N$ matrix where each row represents a bit position. Each time one row (one bit position) of the input matrix is multiplied with the weight vector by looking up the correct dot product result, and then the result is multiplied with the corresponding bit weight. This step is repeated M times until all the bits are processed and all the results are accumulated to calculate the final result. Doing so, the dot product is effectively calculated in a bit-serial way, and it takes M iterations to compute the original dot product. Figure 5 visualizes the decomposition process using the 8-element 8-bit dot product example.

3.2 Lookup Table Bitwidth and Weight Pool Storage

By decomposing the input vector, the lookup table only needs to store the results of the dot product between N 1-bit input elements and N fixed weight elements. The required lookup table size is thus reduced to 2^N entries, which is 256 entries for the 8-element dot product example. Assuming 64 fixed weight vectors are needed for a weight pool network (we will show later 64 is enough for most cases), and the results are stored in 8-bit precision, the total lookup table storage for the entire network is just 16 kB. Since the lookup table needs to be stored in memory, this storage overhead should be considered when calculating the overall compression ratio of weight pool networks. Besides the activation/weight vector length N, We also denote the lookup table bitwidth by B_l and the size of weight pool by S, the formula for lookup table storage in bits is:

$$Storage_{LUT} = 2^N \times S \times B_l \tag{3}$$

For a network with W total weight parameters and weight bitwidth of B_w, the total network storage in bits is $W \times B_w$. Assuming all the weights of the network are compressed by the weight pool method, the maximum compression ratio that can be achieved is:

$$CR = \frac{W \times B_w}{(\frac{W}{N} \times log_2 S + 2^N \times S \times B_l)} \tag{4}$$

, where the term $\frac{W}{N} \times log_2 S$ is the weight index storage. $log_2 S$ is the minimum bitwidth required for the weight index, but in actual implementation it may be more efficient to use 8 or 16 bits.

Interestingly, the weight bitwidth of weight pool networks can be arbitrary since the weights are not explicitly stored. The entire weight pool is converted to a lookup table and the dot product results are stored instead of weights. In this case, the lookup table bitwidth matters, as it determines how much memory space is required for storing the lookup table, as well as the inference accuracy of the network. Storing the lookup table at low bitwidth essentially reduces bitwidth (precision and/or range) of dot-product partial sums and may compromise the inference accuracy. We experimentally show that 8-bit lookup table precision is good enough for most cases. The full results are shown in 5.3.

3.3 Activation Bitwidth and Weight Pool Network Runtime

In terms of theoretical runtime performance, for the 8-element 8-bit dot product example, the proposed method requires 8 iterations to loop over bit positions and each iteration contains two memory loads (input and result), one shift and one accumulates operation. The weight indices are the same for all the bits and hence can be shared. Normal convolution also requires 8 iterations to loop over individual vector elements and each iteration requires two memory loads (activation and weight), one multiplication and one accumulation. This analysis shows that our proposed method has an almost identical theoretical runtime compared to the 8-bit baseline without considering overheads and optimizations. This is a promising result since the proposed method can have better runtime than the baseline by simply reducing the activation bitwidth below 8 bits. We will show that with various reuse and optimizations, our proposed method has better runtime even at 8-bit activation bitwidth compared to the 8-bit baseline using ARM's CMSIS library (Lai et al., 2018).

4 Weight Pool Implementation: Overheads and Optimizations

There are many runtime overheads associated with software bit-serial processing and weight sharing. Here we discuss these overheads and the corresponding optimizations to overcome them.

4.1 Bit Unpacking Overheads and Optimized Dataflow

For software sub-byte precision computation, bit unpacking causes significant runtime overhead since processors typically are byte-addressable. For our bit-serial lookup method, the bit decomposition step needs to unpack each element of the input vector into individual bits, and the same bit position of different input elements (rows of the decomposed input matrix in Figure 5) should be grouped together for lookup table computation. Doing this in software requires iterating over all the input elements and for each input element there is an inner loop to extract all the bits. For the 8-element, 8-bit dot product example, 64 iterations are required for a single dot product, while only 8 iterations are required for the actual computation. Implementing bit unpacking for every dot product can significantly slow down the runtime, making it roughly 9$\times$ slower than baseline hence negating any potential speedup by reducing the activation bitwidth.

To address the bit unpacking overhead, we utilize input reuse in our dataflow so that the bit unpacking step (activation vector decomposition) can be shared. For CNNs, the same input can be reused for all the filters of a layer, so that the bit unpacking overhead per result lookup is reduced by a factor equal to the number of total filters in a layer. To implement input reuse and share the bit unpacking overhead, we order the loops such that the filter lookup is inside the loops over input channels and filter x, y dimensions. The activation vector decomposition (bit unpacking) is implemented right before the filter loop, so that the decomposed activation matrix can be reused. Algorithm 1 shows the overall flow including the modified loop order. The bit-unpacking step happens at line 7 of Algorithm 1. For a convolution layer with N filters, the time spent on bit unpacking is reduced by a factor of N and is significantly less than the time spent on result lookup for most layers.

Algorithm 1 The simplified algorithm flow of the bit-serial lookup table implementation. Number of input channel group is number of total input channels divided by weight vector size.

```
 1: for loop over batch do
 2:   for loop over output x-dimension do
 3:     for loop over output y-dimension do
 4:       for loop over kernel x-dimension do
 5:         for loop over kernel y-dimension do
 6:           for loop over input channel groups do
 7:             Activation vector decomposition (bit
                unpacking
 8:             Lookup table caching (flash to ram)
 9:             if Precomputation then
10:               for loop over weight pool vectors do
11:                 for loop over activation bits do
12:                   Results lookup
13:                   Shift and accumulate
14:                 Store results in RAM
15:               for loop over filters do
16:                 Precomputed results lookup
17:             else
18:               for loop over filters do
19:                 for loop over activation bits do
20:                   Result lookup
21:                   Shift and accumulate
```

4.2 Memory Latency and Lookup Table Caching

In a typical microcontroller, flash memory is used as the main storage and SRAM is used for holding variables during computation. Flash memory has more storage space than SRAM but operates slower. However, due to SRAM's limited size (typically 16-128 kB), it can only be used to hold activations and some temporary variables. The network weights are normally stored in flash memory (size ranges from 128 kB - 2 MB), and during the computation the weights are loaded from the slower flash memory. For weight pool networks, the lookup table size is typically 8-32 kB, which is similar to the SRAM size of some small microcontrollers. For such really tiny, low-cost processors, the lookup table cannot fit in SRAM and need to be stored in flash, hence the result lookup latency will be higher and hurt runtime.

To improve the result lookup latency, we cache the active part of the lookup table in SRAM. Before explaining what is the active part of a lookup table, we first discuss how data can be arranged inside a lookup table. The lookup table of the proposed method contains the dot product results between all weight vectors and all possible input (activation) bit vectors. There are two ways to order the lookup table contents when storing them in memory, one is weight oriented order and the other is input oriented order. Visualization of the two lookup table orders are shown in the appendix. Assume the total number of weight vectors in the weight pool is S and the activation bitwidth is M. For weight oriented order, the lookup table can be split into S smaller concatenated lookup tables, each containing the results of all possible inputs related to a single weight vector. For input oriented order, the lookup table consists of 2^M smaller lookup tables and each of them contains the results of one input with all weight vectors. Input oriented order is more compatible with input reuse dataflow since a few blocks (results corresponding to the bit-vectors generated by the input matrix decomposition) of the lookup table is repeatedly accessed in the filter loop, with other blocks of the lookup table staying idle. We utilize this property and cache the active blocks of the lookup table from flash to SRAM during computation. We use input oriented lookup table in our implementation to reduce the flash access overhead and improve runtime.

In our implementation, the dataflow is configured to boost input reuse, and lookup table accesses can also benefit from this dataflow by caching the lookup table in SRAM. In our input reuse dataflow, after activation decomposition the activation vector is multiplied with corresponding weight vectors for all filters. In this case, only a portion of the lookup table related to the generated activation vectors will be used inside the filter loop. Still considering 8-bit activation bitwidth and weight pool size of 64. After the bit decomposition step, 8 activation bit vectors are generated. For the input oriented lookup table, only 8 blocks of the original lookup table each with 64 entries (weight pool size) that corresponds to the activation bit vectors will be actively used in the filter loop. The total size of the active lookup table is just 512 bytes, which is small enough to fit into most microcontroller SRAMs.

Hence, as shown in line 8 of Algorithm 1, before entering the filter loop, we load the active portion of the lookup table from flash and cache them in SRAM. Figure 7 visualizes the lookup table caching process. The overhead of this lookup table caching step is again compensated by sharing it across all the filters. Doing so in the innermost loop of the lookup table results will be loaded from SRAM instead of flash, therefore the overall runtime can be improved.

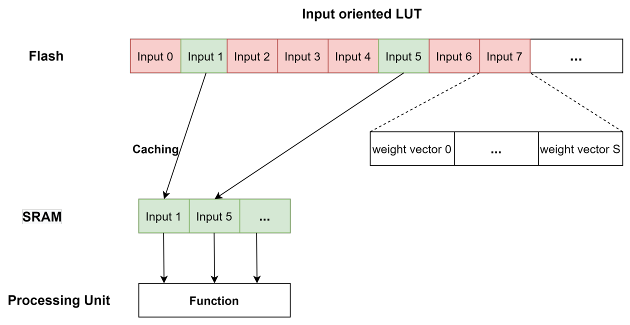

Figure 6. Visualization of lookup table caching. Green blocks represent active lookup table regions corresponding to the input vectors that are shared across filters. Red blocks represent the inactive lookup table regions. Active regions are cached into SRAM before the filter loop and the function only accesses lookup table results from SRAM.

To validate the analysis, we benchmark the lookup table caching optimization against the implementation without lookup table caching (everything else is the same) on individual layers with a different number of filters. The results are shown in Figure 7 (orange bars). The lookup table caching version outperforms baseline for all 4 layer configurations, and the speedup scales with the number of filters in the layer (due to better reuse). While lookup table caching only marginally improves runtime for layers with 32 filters, it achieves more than $1.4\times$ speedup for layers with 192 filters.

4.3 Weight Pool Computation Reuse Through Precomputation

The main property of weight pool networks is that a small pool of weight vectors is shared across the entire network. We have shown that using a pool of 32 or 64 8-element weight vectors is enough for maintaining the accuracy, and such pool sizes are often smaller than the number of filters of a large convolution layer, which can be more than 256. The relatively small pool size offers computation reuse opportunities on large convolution layers to further improve the runtime of weight pool networks.

A property of CNNs is that the same input vector can be reused for all the filters of a convolution layer. For weight pool networks, weights are selected from a group of weight vectors and the total number of distinct weight vectors is the pool size (32 or 64). If a convolution layer has more filters than the pool size, an input vector will inevitably multiply with some weight vectors multiple times when looping over filters. In other words, for a weight pool network, the maximum number of unique dot products that need to be computed for a given input vector is the weight pool size, regardless of the actual number of filters in that layer. To avoid unnecessary computation for large convolution layers, precomputation can be used to only compute the necessary dot products between inputs and weights and store them in another lookup table, hence repeated (bit-serial) computation will be replaced with result lookups. Another way to avoid repeated computation is memoization, where the dot product results are dynamically memoized during computation (inside the filter loop). We compare and evaluate the two methods (analysis is in appendix) and precomputation performs better. The simplified flow of precomputation is shown in lines 9-16 of Algorithm 1.

Precomputation should only be used for large convolution layers as its benefits rely on a large number of filters (it improves runtime when the number of filters of a layer is larger than the weight pool size). For a given layer, precomputation is used only when the number of filters is *larger* than the pool size. To demonstrate the effectiveness of precomputation, we combine precomputation with lookup table caching and evaluate the speedup against baseline implementation, using the same benchmark in section 4.2. The results in figure 7 show that for layers that have more filters than the weight pool size, precomputation can further improve the runtime of the lookup table caching version. For a layer with 192 filters, precomputation + lookup table caching achieves $2.45\times$ speedup against baseline implementation and is $1.7\times$ faster than just using lookup table caching. However, for layers with number of filters that are smaller or equal to the weight pool size, precomputation hurts runtime. This result supports our analysis that precomputation should not be used for those layers.

Run-time accuracy trade-off Precomputation not only accelerates wide convolution layers, it also offers another way to make trade-offs between runtime and accuracy, besides adjusting the activation precision. For a relatively wide network that contains layers wider than 32 filters, the runtime can be improved by reducing the weight pool size.

Name	Model	SRAM (kB)	Flash (kB)	Core	Freq. (MHz)
MC-large	F207ZG	128	1024	CM3	120
MC-small	F103RB	20	128	CM3	72

Table 2. STM Nucleo family microcontrollers used for benchmarking. Both use ARM Cortex M3 for the core.

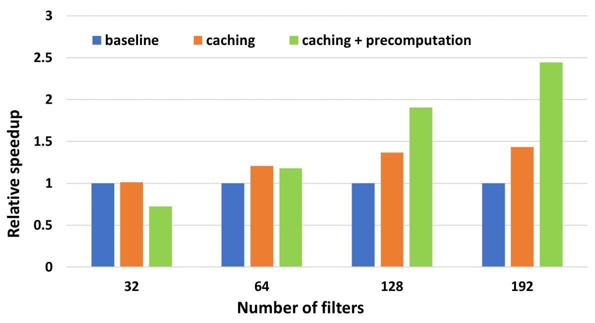

Figure 7. Relative speedup of just lookup table caching (orange) and precomputation + lookup table caching (green) against baseline implementation. Four 3×3 convolution layers with different number of filters are tested. The number of channel is set to be same as number of filters and the input size is 16×16. Weight pool size is 64.

Although we observed that a weight pool size of 64 works reasonably well in most cases and we set 64 as the default size, 32 is also good enough for many cases. The runtime can be improved with a tiny drop in accuracy by reducing the weight pool size in such cases.

5 Evaluation

5.1 Experimental Setup

We evaluate the accuracy and runtime of the z-dimension weight pool method on five different networks: TinyConv (Lai et al., 2018), MobileNet-v2 (Sandler et al., 2018), ResNet-10 (ResNet-18 with last two blocks truncated), ResNet-14 (ResNet-18 with last block truncated) and ResNet-s (scaled-down version of ResNet-18 used in (Banbury et al., 2021)). We use 2 datasets, CIFAR-10 and Quickdraw-100 (100 classes), and form 5 network-dataset combinations. All ResNets are tested on CIFAR-10 while MobileNet-v2 and TinyConv are tested on Quickdraw-100. The network structures are adjusted slightly to fit CIFAR-10 and Quickdraw-100. For the weight pool version of MobileNet-v2, only the 1×1 point-wise convolution layers are compressed using the weight pool. Depth-wise convolution layers are kept uncompressed since they do not fit our proposed implementation. Theoretically the depth-wise layers can be compressed using the xy-dimension weight pool, but it is not necessary - those layers account for a very small portion of storage (2.93%) and runtime.

All the accuracy results are evaluated using the PyTorch framework. For network training and retraining, SGD is used as the optimizer with learning rate scheduling, and batch size set to 128. For runtime results, we use two microcontrollers as shown in Table 2. We use ARM Compiler version 6 and runtime is measured using the built-in cycle counter. The frequency is set to maximum frequency for both boards.

5.2 Compression Ratio

Network	Total param	CR	LUT overhead
TinyConv	81600	2.32	29.8%
ResNet-s	170928	4.43	29.7%
ResNet-10	665280	6.51	13.8%
ResNet-14	2729664	7.55	4.3%
MobileNet-v2	2249792	6.22	4.5%

Table 3. Total number of parameters (uncompressed), overall compression ratio (CR) and lookup table overhead of the selected networks. The lookup table overhead is the proportion of lookup table storage to the total network storage after compression.

Table 3 shows the total number of parameters and the overall compression ratio of the networks with weight pool size of 64. The lookup table overhead is also shown and is compression limiting only for small networks such as TinyConv. The compression ratio improves as the network size increases, and is close to the theoretical maximum ($8\times$) for ResNet-14 (and even larger networks). Smaller networks further suffer in compression since the first convolution layer and fully connected layers are not compressed, whose effect is not well amortized. [1]

5.3 Accuracy Evaluation

5.3.1 *Weight Pool Size*

We first study the impact of weight pool size alone on accuracy without any quantization effects. Table 4 shows the accuracy of the z-dimension weight pool compression with three weight pool sizes without any activation quantization compared to an uncompressed floating-point baseline. A weight pool size of 64 ensures little accuracy drop for most networks and is our default for all experiments unless otherwise mentioned. ResNet-s, being already compressed, is tougher to compress without accuracy loss. The results demonstrate the effectiveness of the z-dimension weight pool compression, even for already small CNNs like TinyConv and ResNet-s.

5.3.2 *Lookup Table Bitwidth*

For the proposed bit-serial lookup table implementation, the dot product results between decomposed activation bit-vectors and weight vectors are stored in the lookup table, and the bitwidth of the lookup table may affect inference accuracy.

[1]Compressing fully connected layer with weight pools improves the compression ratio for Resnet-s (TinyConv) to 4.5(3.1) but at the cost of 0.7%(2.8%) additional accuracy drop. These compression ratios improve further to 5.7 (4.2) if weight pool size of 32 is used albeit, again at 0.5%-1% additional accuracy drop. In this work we do not compress them as they do not improve compression for most networks but affect accuracy.

Network	Original	32	64	128
		CIFAR-10		
ResNet-s	85.3	82.0	83.0	84.0
ResNet-10	91.0	89.3	89.8	90.1
ResNet-14	92.3	90.7	91.1	91.0
		Quickdraw-100		
TinyConv	82.2	81.7	82.2	82.3
MobileNet-v2	86.5	86.7	86.8	86.9

Table 4. Accuracy (%) of the z-dimension weight pool with different weight pool sizes on selected network-dataset combinations. Original means original network accuracy and 32/64/128 are the weight pool size.

To evaluate the impact of lookup table bitwidth on network accuracy, we simulate the proposed bit-serial lookup implementation using PyTorch. Results in table 5 show that a *lookup table bitwidth of 8* loses no accuracy and is the default for our experiments unless otherwise mentioned. Furthermore, since most processors are byte-addressable, using a bitwidth smaller than 8 would incur performance overheads albeit delivering a better storage compression for small networks.

Network	Lookup table bitwidth			
	No-LUT	16	8	4
		CIFAR-10		
ResNet-s	83.0	83.0	82.9	82.3
ResNet-10	89.6	89.9	89.9	89.4
ResNet-14	91.1	91.1	91.1	90.4
		Quickdraw-100		
TinyConv	82.2	82.2	82.1	81.6
MobileNet-v2	86.8	86.6	86.6	85.5

Table 5. Inference accuracy (%) of bit-serial lookup table implementation. No-LUT column shows accuracy that not using lookup table implementation. The activation bitwidth is 8 bit.

5.3.3 *Activation Bitwidth*

Although activation bitwidth does not affect the storage of a weight pool network, it affects the runtime when the weight pool network is implemented using the proposed bit-serial lookup table approach. We use an iterative search algorithm to determine the optimal range when quantizing activations. The weight pool size is 64 and the lookup table bitwidth is 8 for all cases. Table 6 shows that for 8-bit activation bitwidth, almost all networks achieve floating point accuracy (i.e.,"64" column in Table 4). At 5-bit activation bitwidth, most networks still maintain less than 1% accuracy drop except for MobileNet-v2 which is quantization-unfriendly (Sheng et al., 2018; Yun & Wong, 2021). Moreover, for lower bitwidths, the accuracy drop can be compensated by retraining the network with activation quantization. After retraining, activation bitwidth can go down to 3-4 bit within 1% accuracy drop for all networks except for MobileNet-v2, which requires 5 bits.

5.4 Runtime Evaluation

5.4.1 *Impact of Activation Bitwidth*

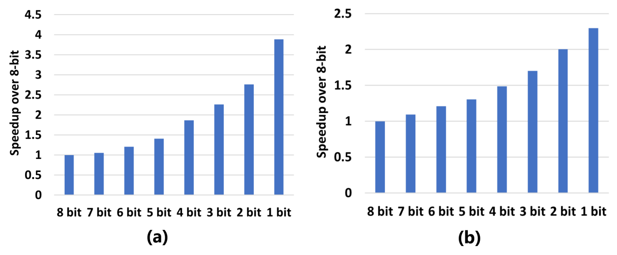

Figure 8. Speedup against 8-bit bit-serial lookup implementation for different activation bitwidths. (a): results without precomputation. (b): results with precomputation. The input size is 16 × 16 and number of channels and filters are both 128. Weight pool size is 64.

One of the main contributions of the proposed framework is the support of accelerating runtime by reducing activation bitwidth. We evaluate the runtime improvement from an 8-bit baseline on a layer with 128 channels and filters and pool size of 64 in Figure 8, using MC-large. Without precomputation, the speedup scales linearly according to activation bitwidth, and is almost 4× for 1-bit activation (less than the 8× theoretical speedup because of the fixed bit unpacking overhead). For the precomputation case, as the activation bitwidth reduces, the runtime of the bit-serial loop during precomputation reduces, but the runtime for precomputed results lookup does not change and starts to dominate the runtime. However, precomputation already accelerates the runtime significantly so the overall speedup is still better for large layers.

5.4.2 *Full-network Benchmark*

To evaluate the overall runtime performance of the proposed method, we evaluate the full-network runtime performance on both microcontrollers with weight pool sizes of 32 and 64, and compare with ARM CMSIS implementation whenever possible. Only convolution layers are benchmarked since we do not apply weight pool on the fully connected layers. The results are shown in Table 7. For the minimum activation bitwidth case, the results for the 32-vector weight pool are for reference only, since the minimum bitwidth is determined from the results of the 64-vector weight pool. For all setups, the proposed implementation achieves better runtime than CMSIS and the speedup is better for larger networks. With less than 1% accuracy drop, the "right bitwidth" weight pools can achieve over 2.8× speedup over CMSIS for medium-sized CNNs like ResNet-10 and around 2× speedup for smaller CNNs like ResNet-s and TinyConv. There are several factors that make the speedup smaller for small CNNs, including lack of precomputation opportunity, more bit unpacking overhead and the relatively larger impact of not accelerating the first layer. Larger CNNs (ResNet-14, MobileNet-v2) do not fit into the microcontroller memory without the weight pool compression and hence a runtime comparison is not possible. Overall, the proposed method improves CMSIS runtime on CNNs regardless of network structure and activation bitwidth, and the speedup is larger for large networks.

5.5 Comparison with Binarized Networks

The theoretical compression ratio of a weight pool network is similar to the compression ratio of binarized networks but with much better accuracy. (Romaszkan et al., 2020) evaluates the implementation of binarized networks on microcontrollers and reports $2-4\times$ speedup compared CMSIS 8-bit implementations. For comparison, we trained the binarized version of TinyConv and the accuracy for CIFAR-10 is barely 66.9% as opposed to 81.2% with weight pools. Our method achieves 14.3% higher accuracy with just 1.24 × runtime overhead.

6 Related Work

6.1 Neural Network Weight Sharing

The concept of weight sharing in neural networks can be dated back to 1992 (Nowlan & Hinton, 1992), as an approach to simplify neural networks. Recently, weight sharing has been applied to convolution neural networks, by clustering and sharing 2D convolution kernels (Son et al., 2018; Wu et al., 2018) for all layers of the network. With tiny or almost no drop in accuracy, weight sharing can significantly compress the parameters of the neural network, which leads to 4.5× to 36× reduction in CNN's storage requirement, depending on the exact sharing method and baseline precision.

6.2 Lookup Table Based Vector Multiplication Acceleration

Lookup table is a widely used method to improve runtime by replacing computation with memory lookup. There are many works (Deng et al., 2019; Sutradhar et al., 2020; Ferreira et al., 2021) try to accelerate deep neural networks with lookup tables by memorizing vector multiplication results. However, due to the huge lookup table size (GB+) required for memorizing all possible results of a vector-vector multiplication, all of them are DRAM based in-memory accelerators, hence they are not software solutions.

Network	Activation bitwidth 8	7	6	5	4	3	Min. bitwidth < 1% a.d
CIFAR-10							
ResNet-s	82.9	83.0	83.1	82.9	82.5	80.4(80.4)	4
ResNet-10	89.9	89.9	89.8	89.6	88.9(89.2)	84.5(87.8)	4
ResNet-14	91.1	91.1	91.0	90.8	90.6(91.0)	88.5(90.2)	3
Quickdraw-100							
TinyConv	82.1	81.8	81.2	79.3(82.0)	69.2(81.2)	36.0(77.4)	4
MobileNet-v2	86.6	86.5	86.0	83.6 (85.9)	77.9(84.0)	36.4(73.0)	5

Table 6. Inference accuracy (%) of weight pool networks with different activation bitwidths. Results in brackets are accuracy after retraining. The last column shows the minimum activation bitwidth with less than 1% accuracy drop. The lookup table bitwidth is set to 8 bit.

Network	CM.	64-8	32-8	64-m	32-m
MC-large					
TinyConv	1.06	0.83	0.75	0.60	0.57
ResNet-s	0.60	0.49	0.43	0.31	0.28
ResNet-10	5.28	3.00	2.22	1.87	1.61
ResNet-14	/	3.46	2.59	1.92	1.73
MobileNet-v2	/	3.60	3.12	3.07	2.78
MC-small					
TinyConv	1.95	1.49	1.33	0.99	0.89
ResNet-s	1.24	1.07	0.89	0.63	0.55

Table 7. Full-network inference latency (in seconds) with different setups for both microcontrollers. CM. stands for CMSIS implementation, -8 means 8-bit activation precision while -m means minimum activation precision that has less than 1% accuracy drop that determined in Table 6. 32 and 64 are the weight pool size. / means the network cannot fit into flash memory.

6.3 Software Based Convolution Acceleration for Sub-byte Precision

There are a few software-focused works that develop algorithms to deploy sub-byte neural networks on CPUs. (Yu et al., 2019) utilizes a single multiplication instruction to implement multiple sub-byte multiplications through bit-packing, and is able to show performance improvement for four-bit input and ternary weight network over 16-bit baselines. (Cowan et al., 2018) and (Cowan et al., 2020) share the same main concept and propose a software method and corresponding optimizations for CPUs to compute sub-byte precision more efficiently by utilizing the popcount instruction. However, as their method has a time complexity proportional to the total number of weight bits times the total number of activation bits, moderate speedup over 8-bit baseline can only be demonstrated on very low activation and weight bitwidth (2-3 bits). (Umuroglu & Jahre, 2017) is another work that targeting extremely low precision CNN acceleration, with a similar idea that utilizes the popcount instruction. Current software methods for accelerating sub-byte neural networks have limited use cases due to their strict requirements on activation and weight bitwidth. For many applications, quantizing both activation and weight to 2-3 bits can severely impact the learning capability of neural networks. Besides, some versions require advanced instructions that are not available for low-power microcontrollers like ARM Cortex M0 and M3. We do not directly compare against these works as the target applications and platforms are not the same and they do not offer arbitrary sub-byte precision acceleration.

7 Conclusion

We have proposed the first framework for efficiently deploying weight pool networks on resource-constrained processors, with compression, training and execution methodologies. The proposed weight pool networks with bit-serial lookup table implementation support and accelerate arbitrary sub-byte precision execution, and can achieve up to $2.8\times$ speedup and up to $7.5\times$ compression compared to 8-bit networks, with less than 1% drop in accuracy. The proposed framework is more efficient on large networks, both in terms of compression and speedup, therefore is suitable for deploying large neural networks on small microcontrollers. We are able to fit and accelerate relatively large CNNs like MobileNet-v2 on a microcontroller with 1MB Flash memory, which otherwise will not fit in the processor memory.

Acknowledgements

We thank the ONR for the funding support of this project.

References

Banbury, C., Reddi, V. J., Torelli, P., Holleman, J., Jeffries, N., Kiraly, C., Montino, P., Kanter, D., Ahmed, S., Pau, D., et al. Mlperf tiny benchmark. *arXiv preprint arXiv:2106.07597*, 2021.

Banner, R., Nahshan, Y., Hoffer, E., and Soudry, D. Post-training 4-bit quantization of convolution networks for rapid-deployment. *arXiv preprint arXiv:1810.05723*, 2018.

Berthelier, A., Chateau, T., Duffner, S., Garcia, C., and Blanc, C. Deep model compression and architecture optimization for embedded systems: A survey. *Journal of Signal Processing Systems*, 93(8):863–878, 2021.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V., and Gopalakrishnan, K. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

Cowan, M., Moreau, T., Chen, T., and Ceze, L. Automating generation of low precision deep learning operators. *arXiv preprint arXiv:1810.11066*, 2018.

Cowan, M., Moreau, T., Chen, T., Bornholt, J., and Ceze, L. *Automatic Generation of High-Performance Quantized Machine Learning Kernels*, pp. 305–316. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450370479. URL https://doi.org/10.1145/3368826.3377912.

Deng, Q., Zhang, Y., Zhang, M., and Yang, J. Lacc: Exploiting lookup table-based fast and accurate vector multiplication in dram-based cnn accelerator. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

Ferreira, J. D., Falcao, G., Gómez-Luna, J., Alser, M., Orosa, L., Sadrosadati, M., Kim, J. S., Oliveira, G. F., Shahroodi, T., Nori, A., et al. pluto: In-dram lookup tables to enable massively parallel general-purpose computation. *arXiv preprint arXiv:2104.07699*, 2021.

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

Hu, Y., Zhai, J., Li, D., Gong, Y., Zhu, Y., Liu, W., Su, L., and Jin, J. Bitflow: Exploiting vector parallelism for binary neural networks on cpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 244–253, 2018. doi: 10.1109/IPDPS.2018.00034.

Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., and Moshovos, A. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12. IEEE, 2016.

Lai, L., Suda, N., and Chandra, V. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

Li, S., Romaszkan, W., Graening, A., and Gupta, P. Swis–shared weight bit sparsity for efficient neural network acceleration. *arXiv preprint arXiv:2103.01308*, 2021.

Nowlan, S. J. and Hinton, G. E. Simplifying neural networks by soft weight-sharing. *Neural computation*, 4(4):473–493, 1992.

Romaszkan, W., Li, T., and Gupta, P. 3pxnet: Pruned-permuted-packed xnor networks for edge machine learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(1):1–23, 2020.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Sharma, H., Park, J., Suda, N., Lai, L., Chau, B., Chandra, V., and Esmaeilzadeh, H. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775. IEEE, 2018.

Sheng, T., Feng, C., Zhuo, S., Zhang, X., Shen, L., and Aleksic, M. A quantization-friendly separable convolution for mobilenets. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pp. 14–18, 2018. doi: 10.1109/EMC2.2018.00011.

Son, S., Nah, S., and Lee, K. M. Clustering convolutional kernels to compress deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 216–232, 2018.

Sutradhar, P. R., Connolly, M., Bavikadi, S., Dinakarrao, S. M. P., Indovina, M. A., and Ganguly, A. ppim: A programmable processor-in-memory architecture with precision-scaling for deep learning. *IEEE Computer Architecture Letters*, 19(2):118–121, 2020.

Umuroglu, Y. and Jahre, M. Streamlined deployment for quantized neural networks. *arXiv preprint arXiv:1709.04060*, 2017.

Wu, J., Wang, Y., Wu, Z., Wang, Z., Veeraraghavan, A., and Lin, Y. Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions. In *International Conference on Machine Learning*, pp. 5363–5372. PMLR, 2018.

Yu, J., Lukefahr, A., Das, R., and Mahlke, S. Tf-net: Deploying sub-byte deep neural networks on microcontrollers. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–21, 2019.

Yun, S. and Wong, A. Do all mobilenets quantize poorly? gaining insights into the effect of quantization on depthwise separable convolutional networks through the eyes of multi-scale distributional dynamics. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pp. 2447–2456, June 2021.

A LOOKUP TABLE ORDER

There are two orders that partial dot product results of the weight pool can be stored in the lookup table, which is input oriented order and weight oriented order, as shown in Figure 9. For weight oriented order, the lookup table can be split into S smaller concatenated lookup tables, each containing the results of all possible inputs related to a single weight vector. For input oriented order, the lookup table consists of 2^M smaller lookup tables and each of them contains the results of one input with all weight vectors.

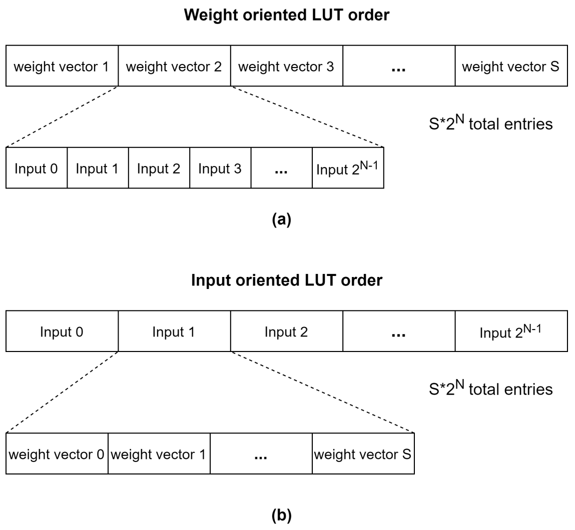

Figure 9. Two LUT data orders. (a) shows the weight oriented LUT order and (b) shows the input oriented LUT order.

B PRECOMPUTATION VS MEMOIZATION

There are two ways to avoid repeated dot-product computations, one is precomputation and the other is memoization. For precomputation the dot products are computed between the input vector and all weight vectors in the weight vector pool, before the filter loop starts. Inside the filter loop, the dot product results are directly looked up from the precomputed results, using the corresponding weight vector as the input to the lookup table. Dot products are not precomputed for the memoization method, instead, the dot product is computed normally inside the filter loop and the result is memoized if it has not been previously computed. If the weight vector has been already computed, it will be retrieved from the saved results and skip the computation. If many of the weight vectors in the weight vector pool are not being

layer number	# filters	# unique weight vectors
3	64	38.5
7	128	54.4
11	256	61.9

Table 8. Profiling results of three layers from ResNet-14 with a 64-vector weight pool. The three columns show the layer number, number of filters in that layer and the average number of unique weight vectors that an input vector need to multiply with.

used for a given convolution layer, memoization may perform better than precomputation since fewer dot products are computed. However if most of the weight vectors in the weight vector pool are being used, precomputation should be a better choice due to the additional computation and branching in memoization caused by its checking logic.

To determine the optimal method, we profiled the weight vector assignment using ResNet-14 (ResNet-18 without the last block) with weight pool compression. Table 8 shows the profiling results of three layers with different number of filters. The weight pool size is 64. The third column shows the average number of unique weight vectors that an input vector needs to multiply with, and the maximum value should be the weight pool size. The results suggest that for layers with more than 64 filters, an input vector needs to multiply with most of the weight vectors in the weight pool, hence precomputation should work better than memoization. Even for a layer with 64 filters, an input vector on average needs to multiply more than half of the weight vectors in the weight pool. The relatively high weight pool utilization ratio means the overhead caused by the memoization can overshadow the benefit it brings. We profiled the runtime of memoization assuming only 32 (out of 64) weight vectors are used for the entire network, which is less than the actual number. Yet the runtime on a ResNet-14 is $1.02\times$ slower than not using memoization.