TF-Net: Deploying Sub-Byte Deep Neural Networks on Microcontrollers

JIECAO YU, University of Michigan
ANDREW LUKEFAHR, Indiana University Bloomington
REETUPARNA DAS and SCOTT MAHLKE, University of Michigan

Deep Neural Networks (DNNs) have become an essential component of various applications. While today's DNNs are mainly restricted to cloud services, network connectivity, energy, and data privacy problems make it important to support efficient DNN computation on low-cost, low-power processors like microcontrollers. However, due to the constrained computation resources, it is challenging to execute large DNN models on microcontrollers. Using sub-byte low-precision input activations and weights is a typical method to reduce DNN computation. But on byte-addressable microcontrollers, the sub-byte computation is not well supported. The sub-byte inputs and weights need to be unpacked from bitstreams before computation, which incurs significant computation and energy overhead.

In this paper, we propose the TF-Net pipeline to efficiently deploy sub-byte DNNs on microcontrollers. While TF-Net allows for a range of weight and input precision, we find Ternary weights and Four-bit inputs provide the optimal balance between model accuracy, computation performance, and energy efficiency. TF-Net first includes a training framework for sub-byte low-precision DNN models. Two algorithms are then introduced to accelerate the trained models. The first, direct buffer convolution, amortizes unpacking overhead by caching unpacked inputs. The second, packed sub-byte multiply-accumulate, utilizes a single multiplication instruction to perform multiple sub-byte multiply-accumulate computations. To further accelerate DNN computation, we propose two instructions, Multiply-Shift-Accumulate and Unpack, to extend the existing microcontroller instruction set. On the tested networks, TF-Net can help improve the computation performance and energy efficiency by 1.83× and 2.28× on average, respectively.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Embedded systems**; *Architectures*;

Additional Key Words and Phrases: Deep neural networks, microcontrollers, sub-byte computation

ACM Reference format:
Jiecao Yu, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2019. TF-Net: Deploying Sub-Byte Deep Neural Networks on Microcontrollers. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 45 (October 2019), 21 pages.
https://doi.org/10.1145/3358189

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2019.
Authors' addresses: J. Yu, R. Das, and S. Mahlke, University of Michigan; emails: {jiecaoyu, reetudas, mahlke}@umich.edu; A. Lukefahr, Indiana University Bloomington; email: lukefahr@indiana.edu.

1539-9087/2019/10-ART45 $15.00
https://doi.org/10.1145/3358189

1 INTRODUCTION

Deep Neural Networks (DNNs) are utilized in various fields including object detection [28], natural language processing [8] and semantic segmentation [12]. Large DNN models provide high prediction accuracy but with high computation and energy cost. These costs make it difficult to deploy DNN models on extremely low-power microcontrollers due to their limited computation and storage resources. For applications like health care [14] and keyword spotting [37], user data is usually collected at the microcontroller-based sensor nodes and then transmitted to the data center for processing using DNNs. However, this cloud-based solution suffers from unstable data connections, high data-transmission energy, and data-privacy issues. In this case, it is of great importance to support efficient DNN computation locally on resource-restricted microcontrollers.

Reducing the precision of input activations and weights is a common method to lower the required computation of DNN models. DNN models generally have a large internal redundancy. Not all the parameters and computation are necessary for achieving the same accuracy level. Using low-precision inputs and weights can simplify computation operations by eliminating unnecessary precision and also reduce memory access, both of which *should* improve energy efficiency.

Vanhoucke et al. [32] show that using 8-bit integers for inputs and weights can still maintain the original accuracy. To compress weights, Courbariaux et al. [6] and Zhu et al. [39] propose to use binary and ternary weights, respectively, with a small level of accuracy loss. Then, Courbariaux et al. [7] and Rastegari et al. [27] use binary values for both weights and inputs to further compress DNN models. This input binarization can help further reduce the required computation but will cause high accuracy loss. As an example, for the Network-in-Network (NIN) [19] model on CIFAR-10 [16] dataset, the model accuracy only decreases by 0.71% after using ternary weights, but further binarizing the inputs will lead to a large accuracy drop of 5.09%. Therefore, using a higher input precision is necessary to maintain prediction accuracy. Through our evaluation, we find that using 4-bit inputs can achieve a similar level of accuracy as using inputs of floating-point numbers. For the NIN model, increasing the input precision back to 4-bit can reduce the accuracy loss to only 0.89%. With a limited accuracy loss, using 4-bit inputs and ternary weights for DNN models can dramatically reduce the required computation and memory access.

However, counter-intuitively, *using sub-byte values for inputs and weights hurts DNN computation performance and energy efficiency on microcontrollers*. Here "sub-byte" refers to precision ≤8-bit. The main reason is the incompatibility between the sub-byte format and the byte-addressable memory hierarchy. Sub-byte values must first be unpacked from the bitstream (the sub-byte format) before computation. This unpacking operation adds computation overhead and also increases memory access, yielding decreased energy efficiency. As an example, for the NIN model, using 4-bit inputs and ternary weights will lead to a 3.80× increase in execution time compared with a 16-bit baseline implementation. The energy consumption is also increased by 3.16×. The only exception is using binary values for both weights and inputs. The multiply-accumulate (MAC) computation between binary values can be replaced with logic operations, and unpacking is not required. However, forcing weights and inputs both to be binary will lead to a significant drop in model accuracy as previously mentioned.

To address this challenge and fully utilize the precision reduction in DNN models, this paper proposes *TF-Net*. TF-Net is a new pipeline for deploying sub-byte DNN models with **T**enary weights and **F**our-bit inputs on microcontrollers. We choose this precision configuration since it achieves a balance between DNN accuracy and the performance improvement from precision reduction.

The TF-Net pipeline includes two stages: training and deploying. In the training stage, we provide a training framework based on XNOR-Net [27] to train DNN models with sub-byte inputs and weights. For the deploying stage, we introduce two computation algorithms, *direct buffer*

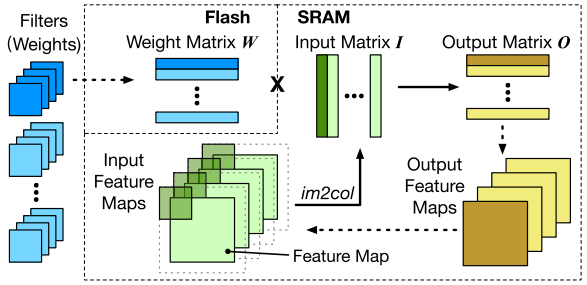

Fig. 1. Conventional computation algorithm of convolutional layers on microcontrollers.

convolution and packed sub-byte multiply-accumulate (packed MAC). First, direct buffer convolution keeps the unpacked input values in a buffer space. The buffered input values are then reused for all weight filters to generate the corresponding output pixels, which amortizes the input unpacking overhead. Second, the packed MAC fuses the MAC operations for four pairs of inputs and weights into *one* conventional 32-bit multiplication instruction. It dramatically increases the computation parallelism and also reduces memory access. To further accelerate DNN computation, we propose two instructions, Multiply-Shift-Accumulate (MSA) and Unpack (UPK), as an extension to Arm v7M instruction set architecture (ISA). These ISA extensions help improve computation performance and energy efficiency while minimizing the hardware overhead.

This paper makes the following contributions:

- We demonstrate that a straightforward implementation of sub-byte DNN models on microcontrollers leads to worse computation performance and energy efficiency.
- We analyze how model accuracy changes with input/weight precision and find that using 4-bit inputs and ternary weights can provide a balance between model accuracy and precision reduction.
- TF-Net, a pipeline for efficiently deploying sub-byte DNN models on microcontrollers, is proposed. We evaluate TF-Net with three network benchmarks. On average, TF-Net can improve the computation performance and energy efficiency by 1.40× and 1.87× without the ISA extension, respectively. Adding the new instructions further increases the improvements to 1.83× and 2.28×, respectively.

2 BACKGROUND AND MOTIVATION

Deep Neural Network (DNN) models achieve high prediction accuracy with a high computation cost. However, a significant part of the computation is not necessary to maintain the same level of accuracy. Using sub-byte weights and inputs is a typical method to remove the redundant computation.

2.1 Sub-Byte DNNs

Existing DNN models contain two main types of layers: convolutional (CONV) layers and fully-connected (FC) layers. Layers are integrated into an end-to-end fashion. In FC layers, each input activation is connected to all neurons. For the inference computation, input values are stored as a 1-D vector and a matrix-vector multiplication is performed to generate the outputs which are also stored as a 1-D vector. For CONV layers, each consists of a stack of 2-D matrices named feature maps. Figure 1 shows the detailed computation in CONV layers. The input feature maps, which is a 3-D tensor, are rearranged into the input matrix $\boldsymbol{I}$ using the image-to-column (im2col) function. Then a matrix-matrix multiplication is performed between the weight matrix $\boldsymbol{W}$ and the input

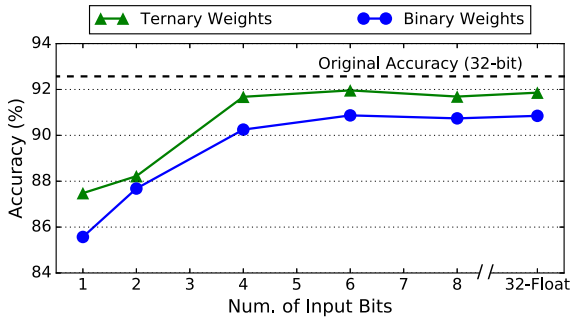

Fig. 2. Prediction accuracy for NIN with different input/weight precision.

matrix $\boldsymbol{I}$ to generate the output matrix $\boldsymbol{O}$ as $\boldsymbol{O} = f(\boldsymbol{W} \cdot \boldsymbol{I})$. The output matrix is considered as the input feature maps for the next layer. f is the element-wise activation function, e.g., ReLU.

Modern microcontrollers have two types of memory storage: SRAM and flash. Through the DNN computation, the flash memory is considered as read-only since writing to flash has a long latency and a high energy cost. As shown in Figure 1, the weight matrix, which is fixed in DNN computation, can be stored in the flash memory. But the input and output matrices are generated at run-time and, therefore, need to be stored in the SRAM. The SRAM space allocated for the input feature maps can be rewritten by the output matrix since the input feature maps have been unrolled into the input matrix.

Using sub-byte weights and inputs is a typical method to remove the internal redundancy of DNN models. As an example, XNOR-Net [27] proposes to use binary values for both weights and inputs. It binarizes the weights as

$$\boldsymbol{W}'_{i,:} = \alpha_i \cdot \text{sign}(\boldsymbol{W}_{i,:}) \qquad \alpha_i = \text{mean}(|\boldsymbol{W}_{i,:}|) \tag{1}$$

where $\boldsymbol{W}_{i,:}$ is the i-th row of the weight matrix which corresponds to the i-th weight filter. α_i is the mean absolute value of $\boldsymbol{W}_{i,:}$. sign() is the sign function and, therefore, sign($\boldsymbol{W}_{i,:}$) contains only binary values. Input values are binarized with a similar strategy

$$\boldsymbol{I}'_{:,j} = \beta_j \cdot \text{sign}(\boldsymbol{I}_{:,j}) \qquad \beta_j = \text{mean}(|\boldsymbol{I}_{:,j}|) \tag{2}$$

where $\boldsymbol{I}_{:,j}$ is the j-th column of the input matrix which corresponds to the j-th convolution window. In this case, we can calculate the output matrix as

$$\boldsymbol{O}_{i,j} \approx \boldsymbol{W}'_{i,:} \cdot \boldsymbol{I}'_{:,j} = \alpha_i \beta_j \cdot [\text{sign}(\boldsymbol{W}_{i,:}) \cdot \text{sign}(\boldsymbol{I}_{:,j})] \tag{3}$$

Here the MAC operations for sign($\boldsymbol{W}_{i,:}$) $\cdot$ sign($\boldsymbol{I}_{:,j}$) can be replaced by XNOR and popcount to accelerate the computation. For example, using a single bit 0/1 to represent binary values $-1/+1$, the computation $-1\times-1 = +1$ can be converted to 0 XNOR 0 = 1. Then the popcount operation counts the number of bits "1", performing the accumulation.

However, using binary values for both inputs and weights hurts the prediction accuracy much. We build a training framework based on XNOR-Net to support ternary/binary weights and 1- to 8-bit inputs. Figure 2 shows the validation accuracy against input precision for the Network-in-Network (NIN) model [19] on CIFAR-10 [16] dataset. Using binary weights reduces the model accuracy by 1.72% and further binarizing the inputs leads to a large accuracy drop of 7.0%. Therefore, using higher-precision inputs and weights is necessary to maintain accuracy. For example, as shown in Figure 2, using 4-bit inputs with ternary weights can limit the accuracy loss to only 0.89%.

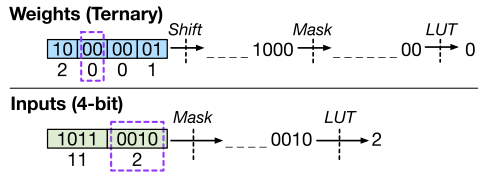

Fig. 3. Examples of unpacking sub-byte inputs/weights.

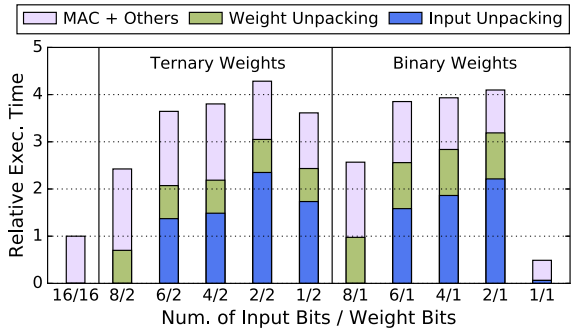

Fig. 4. Relative execution time breakdown of NIN with different input/weight precision. Unpacking input/weight values from the sub-byte format will lead to an execution time increase.

2.2 Unpacking Overhead

Using sub-byte inputs and weights reduces the precision required for the MAC operations and also the memory footprint through the DNN computation. It should, therefore, help reduce total energy consumption. However, it is still challenging to deploy sub-byte DNN models on microcontrollers.

The main challenge is the incompatibility between the sub-byte format and the byte-addressable memory hierarchy. The sub-byte inputs and weights need to be stored in the sub-byte format as bitstreams, and cannot be directly accessed since the memory space is not bit-addressable. Instead, they have to be unpacked from the bitstreams before being used for computation. Here *unpacking* means extracting sub-byte values from the bitstream into a format, e.g., 8-bit/16-bit unsigned integers, suitable for general-purpose instructions. Figure 3 gives two unpacking examples. For ternary weights, we use unsigned integers 0/1/2 as possible weight values. Each byte can store 4 weights. If we need to use the second weight "0" for computation, we need to perform the unpacking operation to load the weight value. We shift the loaded byte by 4 bits, mask out the last 2 bits, and then get the weight value using a lookup table (LUT). For 4-bit inputs, using unsigned integers for possible input values, we still need to extract the input value "2" by the mask and LUT operations. Here the LUT operation is not necessary for unpacking a single weight or input but it benefits when we need to unpack multiple values for SIMD computation. For example, we can unpack the first two weights, "2" and "0", simultaneously with LUT after the shift operation.

These unpacking operations incur a significant overhead into the sub-byte DNN computation on microcontrollers. Figure 4 shows the execution time breakdown of NIN on our test board with different input/weight precision. Here we use 16-bit inputs and weights for the baseline implementation. Arm v7M ISA provides SIMD instructions for efficient 16-bit MAC computation. 16-bit computation can, therefore, achieve better performance than using 8-bit values.

As shown in the figure, the computation overhead for weight and input unpacking dramatically increase the execution time. The only exception is using binary values for both weights and inputs. With binary weights and inputs, MAC operations can be replaced by XNOR and popcount, and

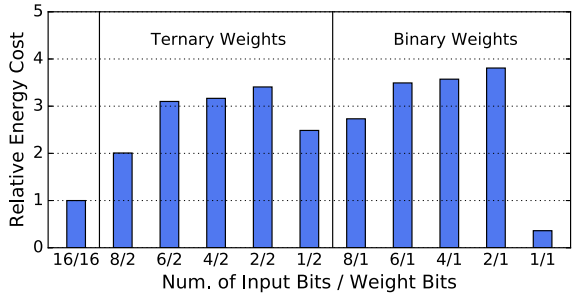

Fig. 5. Relative energy cost for one inference of NIN with different input/weight precision.

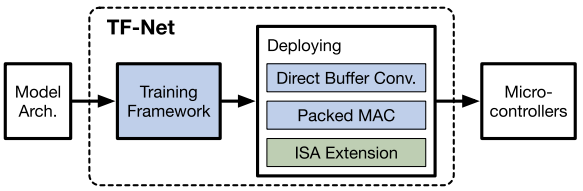

Fig. 6. Overview of the TF-Net pipeline. Only the ISA extension requires extra hardware support.

unpacking is not required through the computation. However, this performance benefit comes with an unacceptable accuracy loss. In the figure, weight unpacking has lower overhead than input unpacking since we reuse the unpacked weights for multiple input values.

Besides the computation overhead, the unpacking operations also increase energy consumption. First, the computation slowdown leads to higher static energy consumption. Second, the LUT operations for unpacking requires more SRAM access which has a high energy cost. Figure 5 shows the relative energy cost of NIN with different input and weight precision. Except weights and inputs both being binary, using sub-byte DNN models increases the energy cost for all configurations by up to 3.8×.

Therefore, due to the unpacking overhead, using sub-byte weights and inputs cannot benefit the DNN computation although it reduces the computation and memory access in theory.

3 TF-NET

To address the computation and energy overhead from the unpacking operations, we propose TF-Net, a pipeline for deploying DNN models with **T**ernary weights and **F**our-bit input activations on microcontrollers.

Figure 6 shows an overview of the TF-Net pipeline. It consists of two phases: training and deploying. Given the model architecture, the training framework first trains the sub-byte DNN models. For deploying, we introduce the direct buffer convolution and the packed MAC to reduce the computation latency. In the direct buffer convolution, input values in one convolution window are unpacked and cached to be reused for generating all the corresponding outputs. For the packed MAC, we use the conventional unsigned 32-bit integer multiplication instruction to perform the MAC operations between 2-bit weights and 4-bit inputs. Also, we propose two new instructions, Multiply-Shift-Accumulate and Unpack, for Arm v7M ISA to further accelerate the computation and reduce the energy consumption.

3.1 Training Framework

In the first phase of TF-Net, we need to train the sub-byte networks with the architecture specified by users. Although there are many low-precision training frameworks and strategies [11, 27, 38,

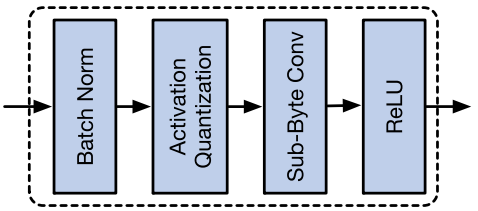

Fig. 7. Building block of TF-Net.

39] proposed, only a few support activation precision between 1- to 8-bit. DoReFa-Net [38] does support activation precision below 8-bit but it cannot achieve the same accuracy with XNOR-Net [27] when weights and inputs are both binary. HWGQ [3] and LQ-Net [36] focus on training networks with binary weights and 2-bit input activations, but they are using non-uniform units for input values. Using inputs with non-uniform units requires processing different low-precision values separately. It will lead to a large computation overhead on microcontrollers. Therefore, in this paper, we build a new training framework by extending XNOR-Net to support 1- to 8-bit input activations and ternary/binary weights.

Basic network block. We use a similar network building block, shown in Figure 7, as in XNOR-Net. The batch normalization (BN) layer [15] is placed before the activation quantization and the sub-byte CONV layer. To achieve higher accuracy, we use the ReLU function in all network blocks to ensure enough non-linearity in the DNN model.

Activation quantization. For binary inputs, the input matrix $\boldsymbol{I}$ are binarized as $\boldsymbol{I}' = \text{sign}(\boldsymbol{I})$. Compared with Equation (2), β is removed and it will not affect the prediction accuracy [27].

For k-bit ($1 < k \leq 8$) inputs, $\boldsymbol{I}$ is quantized as

$$\boldsymbol{I}' = s \cdot \boldsymbol{H} + m \qquad \boldsymbol{H} = \text{round}\left[\text{clamp}\left(\frac{\boldsymbol{I} - m}{s}\right)\right] \tag{4}$$

where s and m are the unit step and the lower bound, respectively, used for input quantization. Both of them are floating-point numbers. $\boldsymbol{H}$ is the quantized matrix with only k-bit unsigned integers. Therefore, the clamp() function needs to clamp the transformed input matrix ($\frac{\boldsymbol{I}-m}{s}$) to $[0, (2^k - 1)]$. s and m need to be chosen to minimize the quantization error $||\boldsymbol{I}' - \boldsymbol{I}||_2$. They can be calculated based on the distribution of the input values. Assume the outputs from the $(l-1)$-th building block is $\text{ReLU}(\boldsymbol{O}^{l-1})$. $\boldsymbol{O}^{l-1}$ represents the output tensor from the corresponding CONV layer. The i-th output feature map $\boldsymbol{O}_i^{l-1}$ is equivalent to the i-th row of the output matrix. We can assume the output tensor $\boldsymbol{O}^{l-1}$ follows a normal distribution of $N(\mu, \sigma^2)$. For a typical DNN model, around 40% of the CONV layer outputs are below zero and will be clamped to zero with the ReLU function. Therefore, we can set $\mu = 0.25 \cdot \sigma$ since values sampled from the distribution $N(0.25\sigma, \sigma^2)$ have a 40% chance to be under zero. Here we use $\boldsymbol{F}$ to represent the input feature maps, which is a 3-D tensor, before the activation quantization. In the l-th block, the i-th input feature map $\boldsymbol{F}_i^l$ is

$$\boldsymbol{F}_i^l = \frac{\text{ReLU}(\boldsymbol{O}_i^{l-1}) - \rho_i}{\phi_i} \cdot \gamma_i + \tau_i \tag{5}$$

after the BN layer. ρ_i and ϕ_i are the mean value and standard deviation of $\text{ReLU}(\boldsymbol{O}_i^{l-1})$, respectively. γ_i and τ_i are two learnable parameters in the BN layer. Notice that the input matrix $\boldsymbol{I}$ is generated by rearranging the input feature maps $\boldsymbol{F}$ with the im2col function. Therefore, the quantization of $\boldsymbol{I}$ in Equation (4) is equivalent to the quantization of $\boldsymbol{F}$. If we only have one feature map, since a

large number of the entries in $\text{ReLU}(\boldsymbol{O}_i^{l-1})$ is 0, we simply set the lower bound m_i as

$$m_i = \frac{0-\rho_i}{\phi_i} \cdot \gamma_i + \tau_i = -\frac{\rho_i}{\phi_i} \cdot \gamma_i + \tau_i \tag{6}$$

It avoids quantization error for the $\boldsymbol{F}_i^l$ entries corresponding to the 0 values in $\text{ReLU}(\boldsymbol{O}_i^{l-1})$. In this case, we have

$$\boldsymbol{F}_i^l = \frac{\text{ReLU}(\boldsymbol{O}_i^{l-1})}{\phi_i} \cdot \gamma_i + m_i \tag{7}$$

Compared with Equation (4), we still need to find s_i for quantization. Since ϕ_i is the standard deviation of $\text{ReLU}(\boldsymbol{O}_i^{l-1})$, $[\text{ReLU}(\boldsymbol{O}_i^{l-1})/\phi_i]$ follows a distribution irrelevant to σ or ϕ_i. Therefore, for a certain input precision, the s_i value minimizing the quantization error $||\boldsymbol{F}_i'^l - \boldsymbol{F}_i^l||_2$ should be proportional to γ_i. Assuming $s_i = \lambda \cdot \gamma_i$, to determine the constant λ, we do not need to profile the DNN activations. Instead, we generate a large sample of $\boldsymbol{O}_i^{l-1} \sim N(0.25\sigma, \sigma^2)$ and then do a parameter sweep to find the λ value which minimizes the quantization error. Since λ is irrelevant to σ, we can fix σ to 1.0 when searching the λ value. As an example, for 4-bit inputs ($k = 4$), we have $\lambda = 4.61/(2^4 - 1)$.

To accelerate the sub-byte computation, we need to share the same s and m across all input feature maps. In this case, we calculate the shared s and m values as

$$s = \sqrt{\frac{\sum {s_i}^2}{N}} = \lambda \sqrt{\frac{\sum {\gamma_i}^2}{N}} \qquad m = \sqrt[3]{\frac{\sum {m_i}^3}{N}} \tag{8}$$

where N is the number of feature maps. Here we use the power of three for calculating m since m_i may have negative values.

Weight quantization. For 1-bit binary weights, they are quantized as in XNOR-Net: $\boldsymbol{W}'_{i,:} = \text{mean}(|\boldsymbol{W}_{i,:}|) \cdot \text{sign}(\boldsymbol{W}_{i,:})$ where $\boldsymbol{W}'$ and $\boldsymbol{W}$ are the quantized and original weight matrices, respectively. For ternary weights, we use the same quantization strategy as [18]

$$\boldsymbol{W}'_{i,j} = \eta_i \cdot \boldsymbol{B}_{i,j} - \eta_i \tag{9}$$

$$\boldsymbol{B}_{i,j} = \text{trisect}(\boldsymbol{W}_{i,j}) = \begin{cases} 0, & \boldsymbol{W}_{i,j} < -\Delta_i \\ 1, & -\Delta_i \le \boldsymbol{W}_{i,j} \le \Delta_i \\ 2, & \Delta_i < \boldsymbol{W}_{i,j} \end{cases} \tag{10}$$

$\boldsymbol{B}$ contains only unsigned integers 0/1/2. Δ_i is the threshold used for dividing weights into three groups. It equals to 0.7 of the mean absolute value of the weights $\boldsymbol{W}_{i,:}$. η_i is the mean absolute value of the weights with absolute values higher than Δ_i.

Backpropagation gradients. The derivative of the rounding functions is zero almost everywhere, making it incompatible with backpropagation. For binary and ternary weights, the rounding function refers to the sign() function and the trisect() function (Equation (10)), respectively. The trisect() function can be represented with the sign() function as

$$\text{trisect}(\boldsymbol{W}_{i,j}) = 1 + \frac{\text{sign}(\boldsymbol{W}_{i,j} - \Delta_i) + \text{sign}(\boldsymbol{W}_{i,j} + \Delta_i)}{2} \tag{11}$$

For activations, the round[clamp()] function in Equation (4) is considered as the rounding function.

We use $r' = \text{q}(r)$ to represent a rounding function. Assuming *loss* is the loss value, we can get the gradient of the rounded value r', $\frac{\partial loss}{\partial r'}$, with conventional backpropagation computation. However, we need to compute the gradient of r for the training process. Based on the straight-through estimator used in [7], we calculate the gradient of r by

$$\frac{\partial loss}{\partial r} = \frac{\partial loss}{\partial r'} \cdot \frac{\partial r'}{\partial r} = \frac{\partial loss}{\partial r'} \cdot 1_{ql \le r \le qh} \tag{12}$$

$$1_{ql \le r \le qh} = \begin{cases} 1, & ql \le r \le qh \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

where ql and qh are two constants choosen based on the target rounding function. For the sign() function, we use $ql = -1$ and $qh = +1$. For the round[clamp()] function, we set $ql = 0$ and $qh = (2^k - 1)$, corresponding to the lower and higher bounds of the clamp() function in Equation (4).

For the CONV layers with ternary weights and input precision higher than 1-bit, using the proposed training framework, the computation we need to perform for each output pixel becomes

$$\boldsymbol{O}_{i,j} \approx \boldsymbol{W}'_{i,:} \cdot \boldsymbol{I}'_{:,j} = \eta_i \cdot s \cdot (\boldsymbol{B}_{i,:} \cdot \boldsymbol{H}_{:,j}) - \eta_i \cdot s \sum \boldsymbol{H}_{:,j} + m \sum (\eta_i \cdot \boldsymbol{B}_{i,:} - \eta_i) \tag{14}$$

Since weight values are fixed through the computation, the last item $m \sum (\eta_i \cdot \boldsymbol{B}_{i,:} - \eta_i)$ can be calculated off-line. The second item $\eta_i \cdot s \sum \boldsymbol{H}_{:,j}$ needs to be generated through the computation but $s \sum \boldsymbol{H}_{:,j}$ can be shared for all weight filters. The corresponding computation overhead is therefore amortized and becomes negligible. In this case, the main execution is consumed by $(\boldsymbol{B}_{i,:} \cdot \boldsymbol{H}_{:,j})$, which our acceleration will focus on.

In TF-Net, we choose to use ternary (2-bit) weights and 4-bit inputs for two reasons. First, this configuration provides a balance between the numerical precision and model accuracy. It reduces the DNN computation but still maintains a similar level of accuracy as the original model. For NIN, compared with the 16-bit baseline implementation, using ternary weights and 4-bit inputs can reduce the weight and input precision by 8× and 4×, respectively. But at the same time, the model accuracy only decreases by 0.89%. Second, the computation between 4-bit and 2-bit values can be effectively accelerated with the packed MAC algorithm and the ISA extension. Besides, when the memory space is more constrained, TF-Net can still be extended to support lower input precision or binary weights with a cost of higher accuracy loss.

3.2 Direct Buffer Convolution

To deploy the DNN models with 4-bit inputs and ternary weights on microcontrollers, we first propose a new convolution algorithm, direct buffer convolution. The unpacked inputs are cached in a buffer space and can then be reused for all the weight filters.

The direct buffer convolution is designed based on the implicit generalized matrix-matrix product (GEMM) [5, 22, 25]. The implicit GEMM is a typical algorithm to reduce the memory footprint of the convolution computation. Instead of generating the input matrix, the input values are kept in the input feature maps. In the computation, the implicit GEMM either generates and processes one small part of the input matrix, e.g., one column, at a time or fetch the necessary input matrix values directly from the input feature maps. However, directly applying the implicit GEMM still suffers from the input unpacking overhead.

Figure 8 shows the details of the direct buffer convolution. As the first step, instead of keeping the inputs in the original format, we fetch the input values in the current sliding window and unpack them into 16-bit integers. The unpacked input values are stored in the window buffer. Then the window buffer is multiplied with all the weight filters to generate the output pixels on the corresponding locations. By reusing the unpacked input values, each input only needs to be unpacked once through the computation. In this case, the overhead for input unpacking can be amortized. Here we unpack inputs into 16-bit integers to utilize the SIMD support.

We test the performance of NIN with the direct buffer convolution and Figure 9 shows the result. Here we use the number of required clock cycles as the metric of performance. It is equivalent to using the execution time since the relative performance results are evaluated under a fixed clock frequency. The direct buffer convolution can help improve performance across all input precision compared with the original unpacking algorithm. However, since the direct buffer convolution

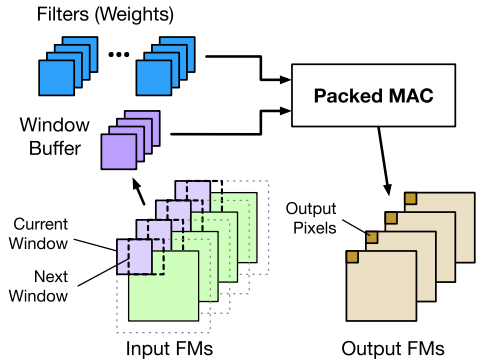

Fig. 8. Direct buffer convolution with packed MAC.

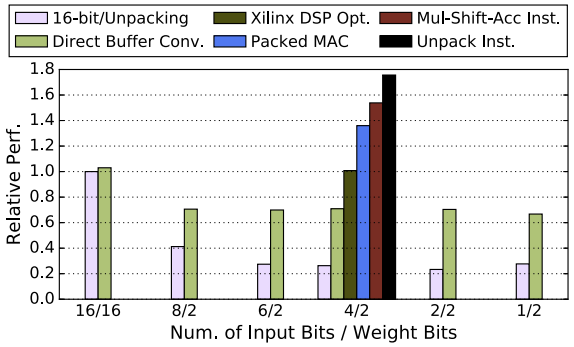

Fig. 9. Relative computation performance of NIN with different convolution algorithms. The baseline uses 16-bit integers for both inputs and weights. The conventional sub-byte convolution algorithm is annotated as *Unpacking*.

only reduces the unpacking overhead for inputs but not weights, we still cannot achieve the baseline performance.

The direct buffer convolution can also help reduce the SRAM cost of DNN computation. As shown in Figure 1, for the original computation, we need to unroll the input feature maps into the input matrix. Each pixel is used in multiple convolution windows and, therefore, is copied for multiple times in the input matrix. In this case, the input matrix has a much larger size than the input feature maps. With the direct buffer convolution, we avoid generating the input matrix but only need to allocate SRAM space for the input feature maps, the window buffer and the output feature maps. Figure 10 shows the sizes of the allocated SRAM for the NIN model with different input precision. The allocated SRAM space is dramatically reduced for all input precision, e.g., from 396KB to 176KB for 4-bit inputs.

3.3 Packed Sub-Byte MAC

As shown in Figure 9, with the direct buffer convolution, the performance with sub-byte inputs and weights is still worse than the 16-bit baseline due to the computation overhead for unpacking ternary weight values. Therefore, to accelerate the computation, we propose the packed sub-byte multiply-accumulate (packed MAC). For 4-bit inputs and ternary weights, the packed MAC uses existing unsigned 32-bit integer multiplier to perform four multiplication and three accumulation operations in a single instruction.

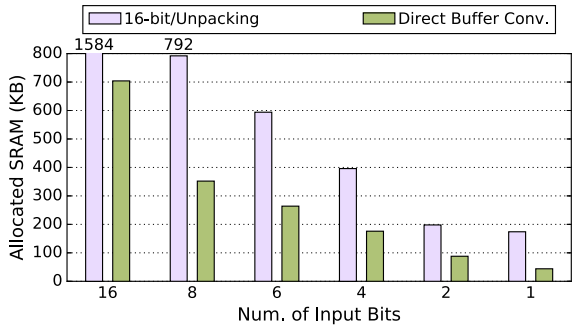

Fig. 10. Allocated SRAM sizes for NIN with different input precision.

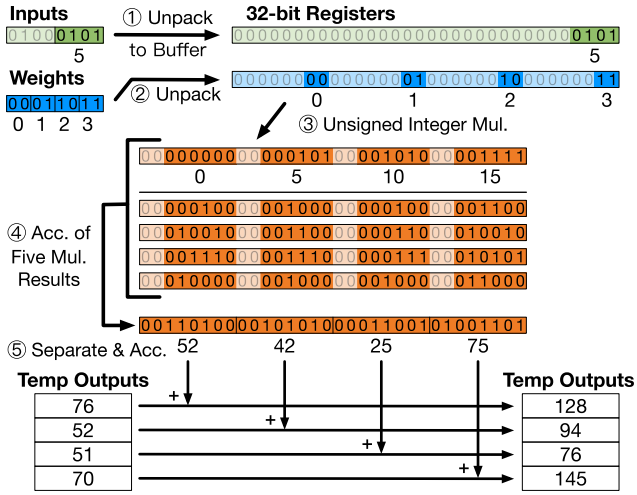

Fig. 11. Example of sub-byte MAC based on the Xilinx INT8 optimization for DSP slices.

The strategy of packing low-precision computation into higher-precision instructions is a typical method to improve computation parallelism. However, applying conventional computation packing methods to sub-byte networks suffers from the overflow problem. For example, Xilinx INT8 optimization for DSP slices [34, 35] proposes to pack two 8-bit MAC operations into one DSP slice. We adjust this algorithm for sub-byte computation, and Figure 11 shows the computation steps. Here we perform MAC between one input convolution window and four weight filters to generate four outputs at the same time. The original algorithm [34, 35] uses signed values for weights and inputs, which requires expensive "if" statements in the computation. Here we use unsigned values instead. For the first and second steps, one input and the corresponding weights from four weight filters are unpacked into two 32-bit registers. Then these two registers are multiplied to generate one temporary result. For the fourth step, five multiplication results are accumulated. Every 8 bits is used to perform the MAC operations for one weight filter. In the last step, the accumulation result is separated through shift and mask operations, and then get accumulated onto the 16-bit temporary outputs. Temporary outputs are stored as 16-bit integers to perform two accumulate operations within one 32-bit register. For the accumulation result in step 4, step 5 has to be performed for every *five* inputs to avoid the overflow from lower 8-bit blocks into the higher 8-bit blocks. It incurs much computation overhead and requires extra registers to

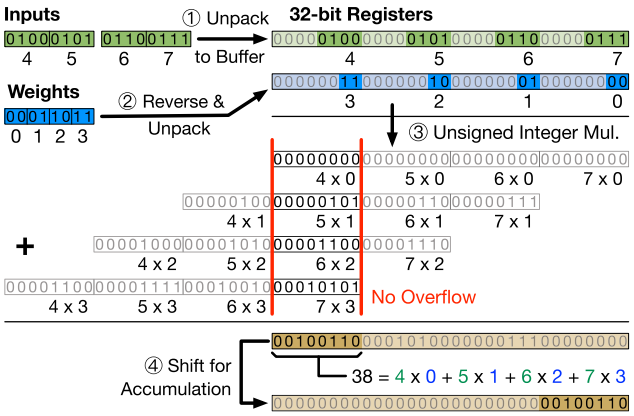

Fig. 12. Example of the packed MAC. The MAC result for the four pairs of inputs and weights is stored in the first 8 bits of the output register after the unsigned integer multiplication.

hold the temporary outputs. We test this algorithm on NIN, and the result is included in Figure 9. The relative performance is increased to 1.01×, achieving the 16-bit baseline performance.

In this paper, we introduce the packed MAC algorithm to perform the sub-byte computation between 2-bit weights and 4-bit inputs in a more efficient way. Figure 12 shows how the packed MAC works. Assume we need to perform the MAC operation between four inputs, $\{4, 5, 6, 7\}$, and four weights, $\{0, 1, 2, 3\}$. For the first step, as in the direct buffer convolution, the input values are unpacked and stored in the window buffer. The 4-bit input values are unpacked into 8-bit unsigned integers instead of 16-bit, and each 32-bit register can include four input values. Then the weight values are unpacked and stored in another 32-bit register but in a reversed order. This operation can be implemented with LUT. After unpacking the inputs and weights, we perform the unsigned integer multiplication between the registers. Since the weight values are reversed, the multiplication results between each pair of input and weight, e.g., 5×1, will all be accumulated into the first 8 bits of the output register. The largest possible MAC result between four pairs of 4-bit inputs and 2-bit weights is only $[(2^4-1) \times (2^2-1)] \times 4 = 180 = (10110100)_2$ which costs 8 bits. Therefore, the accumulation in the first 8 bits and the other 24 bits will not overflow into higher bits. In this case, the MAC results between the four inputs and weights, 38, is in the first 8 bits of the output register. As the last step, we only need to right shift the output register by 24 bits and accumulate it onto the temporary output value.

The packed MAC can increase the computation parallelism and, therefore, the performance. We test the performance of NIN and Figure 9 gives the result. Compared with the original unpacking algorithm, the packed MAC accelerates the computation from two sides. First, it reduces the weight unpacking overhead. In the original computation, to utilize the 16-bit SIMD support, we can only unpack two weights at the same time since each 32-bit register can only include two 16-bit unpacked weights. In the packed MAC, four weights can be unpacked at the same time. Second, the input loading instructions are also reduced. With the packed MAC, the unpacked inputs in the window buffer are stored as 8-bit integers instead of 16-bit. It means the required input loading instructions for the MAC computation is reduced by half.

Using sub-byte inputs and weights can now achieve much better performance than the 16-bit baseline. On the one hand, the packed MAC increases the computation parallelism. On the other hand, using sub-byte inputs and weights reduces the memory access to both SRAM and flash. Figure 9 shows that the packed MAC can improve the relative performance by 1.36× compared with the 16-bit baseline.

Table 1. Synthesis Results for Added and Modified Modules

Inst.	Modules	Energy/Op	Delay	Area Overhead
MSA	Mul+Acc	9.23 pJ	1.64 ns	1493.9 μm^2 (1.26%*)
	Mul+Shift+Acc	Δ +0.11 pJ	Δ +0.04 ns	
UPK	Unpack	0.02 pJ	0.16 ns	163.4 μm^2 (0.14%)

*Percentages are relative to the Arm Cortex-M4 CPU area (0.119mm^2) excluding FPU, debug modules and SRAM.

The packed MAC can be used for all models with the input precision + the weight precision $\leq$6-bit. As shown in Figure 12, the packed MAC can work properly if there is no overflow in the highest 8 bits and the other 24 bits. The MAC result from four pairs of unsigned 4-bit and 2-bit values will not exceed $180 = (10110100)_2$ which costs 8 bits. But if we use 5-bit inputs and 2-bit weights, then the largest possible MAC result becomes $[(2^5-1) \times (2^2-1)] \times 4 = 372 = (101110100)_2$ which requires 9 bits. In this case, an overflow may happen in the highest 8 bits, and the packed MAC cannot work correctly.

For microcontrollers without SIMD support, the packed MAC can still be applied because it only needs the basic multiplication and shift instructions. The performance benefit will be higher since the baseline performance becomes worse.

3.4 Mul-Shift-Acc and Unpack Instructions

With the packed MAC, using sub-byte DNN models can now achieve better performance than the 16-bit baseline. In TF-Net, we also propose to further accelerate the sub-byte computation with specific hardware support. We choose to extend Arm v7M ISA instead of including extra accelerators. This is because adding accelerators in microcontrollers has a high design and manufacturing cost and, meanwhile, the DNN-specific accelerators cannot be utilized for other applications.

New instructions are proposed based on three rules. First, the direct buffer convolution with the packed MAC can be efficiently accelerated with the new instructions. Second, the new instructions should have the lowest hardware overhead. Third, it is better if the added instructions can be utilized for different input/weight precision. Based on these rules, we introduce two instructions for Arm v7M ISA: Multiply-Shift-Accumulate (MSA) and Unpack (UPK).

For the last step of the packed MAC, as shown in Figure 12, we need to shift the multiplication result and then do the accumulation. This multiply, shift and accumulate operation requires two instructions. On the other hand, the existing ISA already provides the multiply-accumulate instruction for 32-bit integers which can be executed in one cycle [21]. Therefore, the first proposed instruction, Multiply-Shift-Accumulate (MSA), reuses the existing multiply-accumulate instruction but also enables shifting of the intermediate multiplication result. The MSA instruction has the same format as the multiply-accumulate instruction: **MSA Rd, Rm, Rs, Rn**. It performs the computation $Rd = (Rm \times Rs) \gg 24 + Rn$. Rd is the output register and, Rm, Rs, Rn are the input registers. By reusing the existing multiplier+adder for the multiply-accumulate instruction, the hardware overhead is minimized. To measure the hardware overhead, we synthesize the original multiplier+adder module and also the modified module with shifting enabled. The result is shown in Table 1. The instruction decoder also needs to be extended, but we assume negligible area overhead. Compared with the original module, adding the shift operation will only increase the energy consumption by 0.11pJ per operation and the delay by 0.04ns. The area overhead is only 1.26% of the basic Arm Cortex-M4 CPU area (0.119mm^2) excluding the FPU and SRAM.

As the same with the original unpacking algorithm, the packed MAC still needs to unpack the weights before the computation (step 2 in Figure 12). We implement this weight reversing and unpacking operation by LUT, which increases the SRAM access and consumes much more energy.

Table 2. DNN Benchmarks

Networks	# CONV layer	# FC layer	Accuracy (32-bit FP)
NIN	9	0	92.57%
VGG-8	6	3	94.04%
ResNet-20	19	1	91.66%

Therefore, we propose Unpack (UPK) as the second instruction to do weight unpacking efficiently. It has the following format: **UPK Rd, Rm, #conf.** Rd is the output register, and Rm is the input register. The last 8 bits of Rm can include four ternary weights or eight binary weights. #conf is the immediate value configuring the unpacking behavior. For #conf==0, we do the unpacking for ternary weights as in Figure 12. Besides, we extend the UPK instruction to support binary weights. For #conf==1/2, the lower/higher four bits, which represent four binary weights, are reversed and unpacked to four weights stored in a 32-bit register. We synthesize the unpack module for the UPK instruction, and the result is shown in Table 1. The area overhead is only 0.14%.

We test the performance of NIN with the proposed two instructions, and Figure 9 give the results. The MSA instruction can help improve the relative performance from 1.36× to 1.54×, and the UPK instruction further boosts the performance to 1.76×. Here the first rule for choosing new instructions is satisfied. Considering the second rule, the proposed instructions only add area overhead of 1.40%. We reuse the existing multiplier+adder module to minimize the hardware overhead.

For the third rule, the proposed direct buffer convolution and packed MAC support all models with the input precision + the weight precision ≤6-bit. Since the MSA and UPK instructions are designed to accelerate the packed MAC, they both can be used for the same range of input/weight precision. However, weight precision higher than 2-bit is not supported here for two reasons. First, using 2-bit ternary weights can already limit the accuracy loss to a small level and using higher weight precision cannot benefit the model accuracy much. Second, using higher weight precision will dramatically increase the required storage size and also memory access. For these reasons, the UPK instruction only supports unpacking 1-bit and 2-bit weights.

4 EVALUATION METHODOLOGY

4.1 Hardware and DNN Benchmarks

Arm Cortex-M series are typical microcontrollers used for various applications. The test board used for our evaluation is STM32 Nucleo-F411RE [20]. It contains an Arm Cortex-M4 microcontroller. It also includes 128KB SRAM and 512KB flash memory. The microcontroller has a clock frequency of 100MHz.

In our experiment, we test three DNN models for the CIFAR-10 [16] dataset: Network-in-Network (NIN) [19], VGG-8 [6] and ResNet-20 [13]. The model structure and accuracy are listed in Table 2. Here we do not test networks for different datasets. Instead, networks with different layer sizes and structures are tested. NIN includes CONV layers with large (5×5) and small (1×1) kernel sizes. VGG-8 has three FC layers. ResNet-20 uses skip connections for the residual blocks. With our training framework, the skip connections are not quantized to sub-byte values. Both the input activations and weights use 16-bit integer values. The batch size of the DNN computation is fixed to 1. PyTorch [26] is used for building the training framework.

4.2 Performance, Area, Energy Measurement

For performance measurement, we directly run DNN computation on the test board and profile the execution cycles. The computation libraries are implemented and optimized considering the

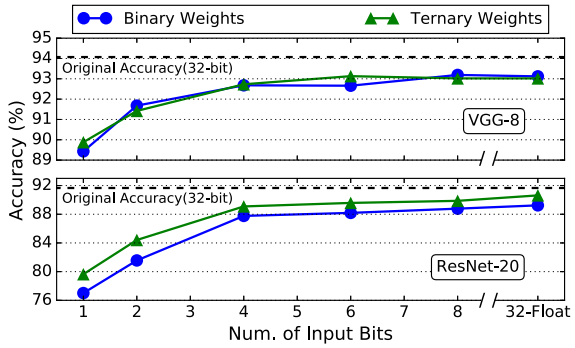

Fig. 13. Accuracy of sub-byte VGG-8 and ResNet-20. The result for NIN is shown in Figure 2.

SIMD support and data reuse. To measure the performance with the ISA extension, we use existing instructions to mimic the new instructions, holding the execution cycle and performing necessary register access. The MSA instruction can be replaced by the multiply-accumulate instruction since they share the same instruction format. For the UPK instruction, we use the shift and reverse subtract instruction to mimic it. This instruction requires one input register and also takes one cycle for execution. By using these two existing instructions to replace the proposed instructions, we can still run the DNN computation directly on the microcontroller to get the performance results. This strategy can maintain data dependencies and pipeline interlocks, which ensures accurate performance measurement. For the DNN computation requiring more SRAM than the test board has, we profile part of the computation to estimate the entire execution time.

We synthesize the added and modified modules with Synopsys Design Compiler (DC) under the IBM 45nm SC12 Standard Cell library to measure their area and energy costs. All the synthesis results are scaled up to 90nm which is the process feature size for the Arm Cortex-M4 CPU.

For energy measurement, we use the STM32 Power Shield board [30] to measure the energy cost of the entire test board including the SRAM and flash memory. To estimate the energy consumption with the proposed ISA extension, as the same for the performance measurement, the proposed instructions are replaced by the existing instructions. We directly run the DNN computation and profile the energy cost. Then we use the synthesis result to adjust the profiling result, which gives an accurate energy estimation.

5 EVALUATION RESULTS

5.1 Sub-Byte DNN Accuracy

We first test how DNN accuracy change with input/weight precision. The test results of VGG-8 and ResNet-20 are shown in Figure 13 and the result of NIN is shown in Figure 2. All the networks are trained from scratch.

For both NIN and ResNet-20, using ternary weights can achieve much higher accuracy than binary weights. For VGG-8, using ternary or binary weights achieves similar accuracy. VGG-8 has a large model size and using binary weights is enough to ensure the non-linearity. In TF-Net, we choose to use ternary weights for higher prediction accuracy.

For all three models, reducing the input and weight precision to 1-bit hurts the prediction accuracy. For example, ResNet-20 suffers from an accuracy loss of 14.65%. Increasing the input precision can dramatically reduce the accuracy loss. With respect to the accuracy loss caused by using 1-bit binary inputs and ternary weights, using 4-bit inputs can reduce the relative accuracy loss by 76% on average, while further increasing the input precision to 8-bit only saves another 5% accuracy

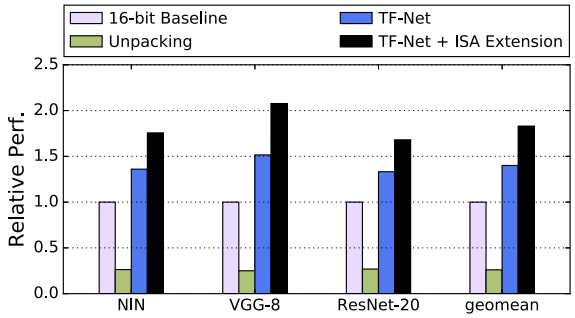

Fig. 14. Relative performance of the tested networks with ternary weights and 4-bit inputs.

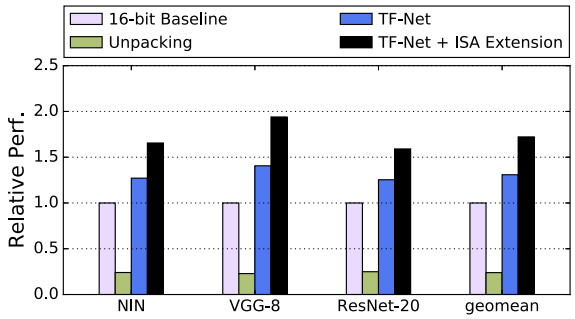

Fig. 15. Relative performance of the tested networks with ternary weights and 4-bit inputs. The tested microcontroller is running at a lower frequency (24MHz).

loss. Therefore, for TF-Net, we choose to use 4-bit inputs, which provides an appropriate balance between input precision and model accuracy. In the following experiments for performance and energy efficiency, we will focus on the configuration of 4-bit inputs and ternary weights.

5.2 Performance

We measure the computation performance of the tested DNN models and Figure 14 shows the relative results with 4-bit inputs and ternary weights. We still use the number of clock cycles as the metric for measuring computation performance. For each model, the baseline is the corresponding performance using 16-bit integers, which fully utilizes the SIMD support.

With the original unpacking algorithm, the relative performance decreases to only 0.26× on average. By deploying TF-Net, the relative performance can now be increased to 1.40× without the ISA extension (annotated as *TF-Net*). Adding the proposed ISA extension can further boost the relative performance to 1.83× (annotated as *TF-Net + ISA Extension*).

As shown in Figure 14, TF-Net can achieve a higher speedup on VGG-8 than NIN and ResNet-20. This is because VGG-8 has a much larger layer size than NIN and ResNet-20. Relatively more computation is for MAC operations which can be accelerated with TF-Net. In comparison, NIN and ResNet-20 cost relatively more computation for other operations, e.g., quantization. Therefore, VGG-8 benefits more from TF-Net.

To further demonstrate the effectiveness of TF-Net, we also perform the performance evaluation with a lower clock frequency. The microcontroller frequency is reduced from 100MHz to 24MHz. The number of wait states for flash fetching is decreased from 3 to 0, which means each flash fetching takes only 1 CPU cycle. Figure 15 shows the results. The baseline is still the corresponding

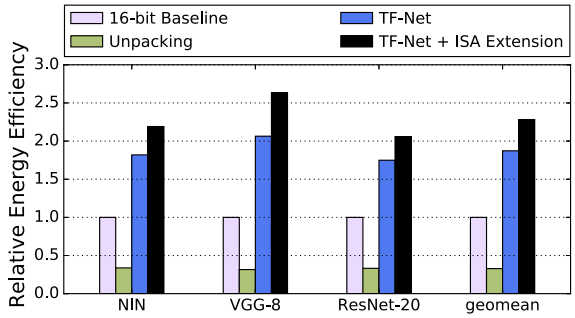

Fig. 16. Relative energy efficiency of the tested networks with ternary weights and 4-bit inputs.

performance with 16-bit integers but under the lower frequency. With a lower clock frequency, the average performance improvement from TF-Net and the ISA extension is slightly reduced to 1.72×. TF-Net helps reduce the flash access and, therefore, the performance benefit from TF-Net decreases when the flash fetching takes fewer cycles.

5.3 Energy Efficiency

After performance evaluation, we test the energy efficiency, and the results are shown in Figure 16. Here we define the energy efficiency as the number of DNN inferences can be completed per energy unit (#inferences/J).

It is shown in Figure 16 that the original unpacking algorithm reduces the relative energy efficiency to only 0.33×. With TF-Net, the relative energy efficiency can be improved to 1.87× and 2.28× on average with and without the ISA extension, respectively.

Comparing Figure 14 and Figure 16 shows that the improvement in energy efficiency is higher than in performance. This is because the performance improvement given by TF-Net comes with a large reduction in memory access to both SRAM and flash. For example, with the proposed Unpack (UPK) instruction, the weight unpacking does not require the LUT operation, which reduces the access to SRAM. In this case, TF-Net reduces the execution time and, at the same time, reduces the dynamic power consumption. Therefore, a higher improvement in energy efficiency is achieved than in computation performance.

As discussed in Section 3.4, the TF-Net pipeline can support all models with the weight precision ≤2-bit, and the input precision + the weight precision ≤6-bit. For the models supported by TF-Net with the ISA extension, the main part of the DNN computation, $(\boldsymbol{B}_{i,:} \cdot \boldsymbol{H}_{:,j})$ in Equation (14), takes similar instructions. Therefore, we expect TF-Net to achieve the same level of improvement in performance and energy efficiency for those models.

With the TF-Net pipeline, the execution time and energy consumption of each NIN inference are reduced to 2.8s and 145mJ, respectively. As a comparison, using a stable connection with a AT86RF212 transceiver [2], the transmission of the image data takes 0.61s latency and 34.5mJ energy. Offloading the DNN computation to the data center has a lower execution time and energy consumption. However, a stable data connection usually cannot be guaranteed. Also, for the application systems using ad hoc networks, the input data needs to go through multiple intermediate nodes before reaching the data center. In this case, the data transmission requires a much longer latency and higher energy consumption.

We also evaluate the energy efficiency with a lower clock frequency (24MHz), which is shown in Figure 17. When the clock frequency becomes lower, the relative energy efficiency can be improved to 1.97× and 2.41× with and without the ISA extension, respectively. This improvement is even better than the high-frequency scenario. The reason is that using a lower clock frequency

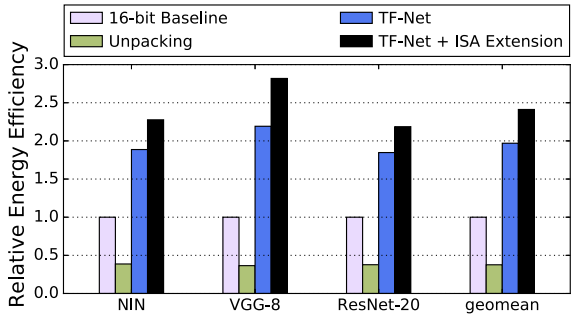

Fig. 17. Relative energy efficiency of the tested networks with ternary weights and 4-bit inputs. The tested microcontroller is running at a lower frequency (24MHz).

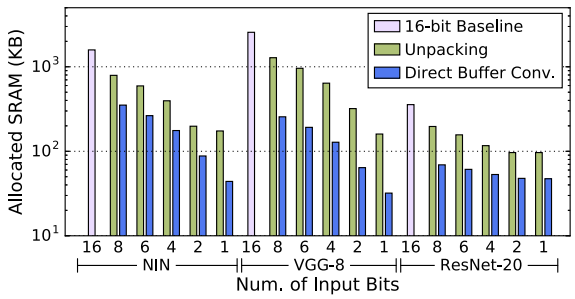

Fig. 18. SRAM allocated for the computation of the tested networks with various input precision.

can reduce the dynamic energy consumption of the processor but not flash access. Flash access now costs relative more energy in DNN computation. Considering that TF-Net helps dramatically reduce flash access, a higher improvement in energy efficiency can be achieved with a lower clock frequency.

5.4 SRAM Requirement

For DNN computation on microcontrollers, the input activations are generated at run-time and, therefore, need to be stored into the SRAM storage. However, the SRAM space on typical microcontrollers is restricted. For example, our test board contains only 128KB SRAM. Adding more SRAM will lead to higher manufacturing cost and leakage power.

Using low-precision input values is a typical way to reduce the SRAM required for DNN computation. We calculate the SRAM space allocated for different models with different input precision, and the results are shown in Figure 18. The y-axis is in log scale. The SRAM space allocated for DNN computation decreases much with lower input precision. Also, the direct buffer convolution further decreases the allocated SRAM size by avoiding generating the input matrix and use the window buffer to cache the unpacked input values. For ResNet-20 model, the allocated SRAM size does not decrease much for lower input precision. This is because the skip connections still keep a copy of the activation values with 16-bit integers to maintain the prediction accuracy.

6 RELATED WORK

DNN hardware acceleration. To accelerate the DNN computation, different hardware designs have been proposed, including ASIC accelerators [4, 23, 31, 40], in-memory processing

techniques [9, 33] and FPGA designs [10, 29]. For example, Eyeriss [4] introduces the row-stationary dataflow to explore the local reuse of weight filters and feature map pixels, minimizing data movement energy consumption. However, adding a separate accelerator will lead to large design and manufacturing costs. For instance, under 90nm, one processing element of Eyeriss costs $0.102mm^2$ area which is 86% of the Arm Cortex-M4 CPU. In this paper, we choose to extend the existing Arm v7M ISA instead of adding an extra accelerator. It can help accelerate the computation with minimized circuit overhead.

Bit-level computation optimization. For various ASIC accelerator and FPGA designs for DNN computation, bit-level computation optimization is adopted to improve the computation performance and energy efficiency [1, 17, 23]. For example, Pragmatic [1] proposes to use serial-parallel shift-and-add multiplication and skip the zero bits of the serial input to eliminate inefficient computation. The outlier-aware accelerator [23] proposes to perform dense and low-precision computation for the majority of the weights and activations while using sparse computation to handle the high-precision weights (outliers).

However, deploying sub-byte DNN models on general-purpose hardware is difficult due to the incompatibility between the sub-byte format and the byte-addressable memory hierarchy. Existing work solves this problem by two methods. First, Courbariaux et al. [6] and Rastegari et al. [27] use only binary weights but high-precision inputs. It can replace MAC operations with add/subtract operations but wastes the potential benefit from using sub-byte input values. Second, Courbariaux et al. [7] and Rastegari et al. [27] use binary values for both weights and inputs. It can use XNOR and popcount to replace MAC operations. However, forcing both inputs and weights to be binary values will hurt the accuracy a lot. In this paper, TF-Net provides a new pipeline for deploying sub-byte DNN models with 4-bit inputs and ternary weights onto microcontrollers, which achieves a balance between model accuracy and performance improvement.

Low-precision DNN. Using low-precision inputs and weights is a typical way to remove the internal redundancy of DNN models. Courbariaux et al. [6] and Zhu et al. [39] use binary and ternary weights, respectively. Courbariaux et al. [7] and Rastergari et al. [27] further lower down the input precision to only 1-bit. For residual structure, Park et al. [24] introduce the precision highway, using high-precision skip connections to achieve a higher model accuracy. In this paper, we build our training framework based on XNOR-Net [27] to support the training of sub-byte DNN models with 1- to 8-bit inputs and ternary/binary weights. We also use high-precision (16-bit) skip connections in the quantized ResNet-20 model for better accuracy. Besides, new algorithms for training low-precision DNN models can be combined into the TF-Net pipeline.

7 CONCLUSION

For DNN computation, using low-precision inputs and weights is a typical method to improve performance and energy efficiency. However, due to the incompatibility between the sub-byte format and the byte-addressable memory hierarchy, the traditional unpacking algorithm for sub-byte computation will actually hurt performance. This paper introduces TF-Net, a pipeline for deploying sub-byte DNN models on microcontrollers. It provides a sub-byte DNN training framework and two computation algorithms, direct buffer convolution and packed MAC, to accelerate the sub-byte computation. Two new instructions are proposed to further improve computation performance. With 4-bit inputs and ternary weights, TF-Net can improve the computation performance and energy efficiency by 1.83× and 2.28×, respectively, on average for the tested DNN models.

ACKNOWLEDGMENTS

This work is supported in part by Arm Ltd and by the National Science Foundation under grant XPS-1628991. The authors would like to thank fellow members of the CCCP research group, and the anonymous reviewers for their time, suggestions, and valuable feedback.

REFERENCES

[1] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 382–394.

[2] Atmel. 2010. Atmel AT86RF212 transceiver. http://ww1.microchip.com/downloads/en/DeviceDoc/doc8168.pdf.

[3] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5918–5926.

[4] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 367–379.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[7] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or- 1. *arXiv preprint arXiv:1602.02830* (2016).

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 383–396.

[10] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 1–14.

[11] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

[12] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*. 2961–2969.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[14] iFixit. 2013. Fitbit Flex Teardown. https://www.ifixit.com/Teardown/Fitbit+Flex+Teardown/16050.

[15] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[16] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).

[17] Lattice. 2013. Lattice. http://www.latticesemi.com/Products/FPGAandCPLD/iCE40.

[18] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[19] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400* (2013).

[20] Arm MBED. 2017. STM32 NUCLEO-F411RE development board. https://os.mbed.com/platforms/ST-Nucleo-F411RE/.

[21] Mark McDermott. 2008. The ARM Instruction Set Architecture. http://users.ece.utexas.edu/valvano/EE345M/Arm_EE382N_4.pdf.

[22] NVIDIA. 2019. cuDNN Installation Guide :: Deep Learning SDK Documentation. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html.

[23] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 688–698.

[24] Eunhyeok Park, Dongyoung Kim, Sungjoo Yoo, and Peter Vajda. 2018. Precision highway for ultra low-precision quantization. *arXiv preprint arXiv:1812.09818* (2018).

[25] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409* (2016).

[26] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[27] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[28] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[29] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 535–547.

[30] STMicroelectronics. 2018. STM32 Power Shield Datasheet. http://www.st.com/content/ccc/resource/technical/document/data_brief/group1/1d/46/2a/b9/60/98/47/13/DM00417848/files/DM00417848.pdf/jcr:content/translations/en.DM00417848.pdf.

[31] Fengbin Tu, Weiwei Wu, Shouyi Yin, Leibo Liu, and Shaojun Wei. 2018. RANA: Towards efficient neural acceleration with refresh-optimized embedded DRAM. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 340–352.

[32] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. Citeseer.

[33] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das. 2019. Bit prudent in-cache acceleration of deep convolutional neural networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 81–93. DOI: https://doi.org/10.1109/HPCA.2019.00029

[34] Xilinx. 2017. Deep Learning with INT8 Optimization on Xilinx Devices. https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf.

[35] Xilinx. 2017. Xilinx 8-Bit Dot-Product Acceleration. https://pdfs.semanticscholar.org/3ac6/4259d37ad76c640333bf8cfccd36bb9bc4f0.pdf.

[36] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 365–382.

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

[38] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

[39] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. 2016. Trained ternary quantization. *arXiv preprint arXiv:1612.01064* (2016).

[40] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018. Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. *arXiv preprint arXiv:1803.11232* (2018).

Received April 2019; revised June 2019; accepted July 2019