A Quantization Framework for Neural Network Adaption at the Edge

Mengyuan Li, Xiaobo Sharon Hu
Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA, 46556
E-mails: {mli22, shu}@nd.edu

***Abstract*—Edge devices employing a neural network (NN) inference engine running a pre-trained model often perform poorly or simply fail at unseen situations. Meta learning, consisting of meta training, NN adaptation and inference, has been shown to be quite effective in quickly learning and responding to a new environment. The adaption phase, including both forward and backward computation, should be performed on edge devices to maximize the benefit in the few-shot learning application. However, deploying high-precision, full-blown training accelerators at the edge can be rather costly for most Internet of Things applications. This paper reveals some unique observations in the adaptation phase and introduces a quantization framework, AIQ, based on these observations to support adaption at the edge with inference-level bit widths. AIQ includes two key ideas, i.e., gated weight buffering and dynamic error scaling, to reduce memory and computational needs with minimal sacrifice in accuracy. Major modules of AIQ are synthesized and evaluated. Experimental results show that AIQ saves 41% and 70% weight memory for two widely used datasets while incurring minimum hardware overhead and negligible accuracy loss.**

I. Introduction

To offer edge intelligence, a widely-used paradigm is to pre-train a neural network (NN), then deploy it on an edge device with a NN inference engine. However, such a setup often performs poorly or simply fails at unseen situations due to the inflexibility in the pre-trained NN. Meta learning, also known as learn to learn, tackles this challenge by quick learn and respond to a new environment [1], [2].

Meta learning, consisting of meta training, NN adaptation and inference, gathers knowledge from various tasks to meta-train a NN model prior. The model prior can then be rapidly adapted to the ever-changing environment with only a few training samples. For edge intelligence, the adaption and inference phase should be implemented on edge devices to quickly respond to environmental changes. There are three common types of meta-learning approaches: (1) black-box amortized, (2) non-parametric, and (3) optimization based, where optimization-based meta learning is more desirable in terms of handling varying and large sample sizes and out-of-distribution tasks [2]. With optimization-based meta learning, after a model is pre-trained, the adaption phase employs the typical *gradient-based learning* on a few training samples to adapts to a new task.

In this paper, we study hardware acceleration of NN adaption at the edge for efficient realization of the optimization-based meta learning. The basic operations in NN adaptation are similar to training in standard supervised learning, and works in three stages: 1) feed forward (FF), 2) feed backward (FB), 3) weight gradient computation (WG). A straightforward way to accelerate NN adaption is to deploy at the edge specialized DNN training hardware such as [3], [4]. However, such hardware typically require 16 or more bits to represent NN parameters (weights, gradients, etc.), thus incur high area, energy and latency overhead. Furthermore, since edge devices are mostly used for inference and only need to perform adaption when environment changes occur, the hardware resources dedicated to training have rather low utilization.

An alternative way is extending inference accelerators by adding hardware for FB and WG to support NN adaption. Inference engines deployed at the edge mostly adopt low-precision (e.g., 4- or fewer-bit) representations to keep hardware cost low. This is possible thanks to the advancement in quantization methods for inference [5], [6]. If NN adaption could also work with such low precision representations, it would not significantly increase hardware cost. However, supporting FB and WG generally requires higher bit precision because quantizing gradients and errors to the same bitwidth as inference weights typically leads to considerable accuracy loss.

We aim to develop techniques for designing NN adaptation accelerators with inference-level bitwidths so that they can be deployed at the edge with minimal overhead. Specifically, we introduce a quantization framework, AIQ (for Adaptation at Inference Quantization), to support back propagation with low-bitwidth integers. To our best knowledge, there is no existing work studying the quantization problem for the adaption phase.

The major contributions of our work are summarized below. (1) We make several observations on the properties of weight gradients and errors during the adaption phase. These properties, different from those in traditional training-from-scratch, are exploited in our quantization framework. (2) We devise a gated weight buffering technique to dynamically identify sensitive gradients and preserve them in high-precision during adaptation. (3) We introduce a lightweight and effective method to adjust the error stepsize dynamically. The method reduces the quantization error with little hardware cost. (4) We implement several modules to support the proposed gated weight buffering technique in the state-of-art SIMD-like learning accelerator architecture with minimal overhead.

Our framework achieve little accuracy drop with 4-bit quantization in the widely-used few-shot learning scenarios. Simulation results show that AIQ saves 41% and 70% weight memory for two widely used datasets while incurring only 3% hardware overhead and negligible accuracy loss.

978-3-9819263-5-4/DATE21/©2021 EDAA

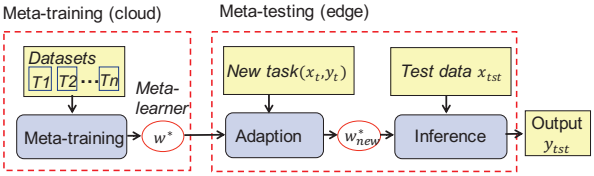

Fig. 1. General computation flow of meta learning.

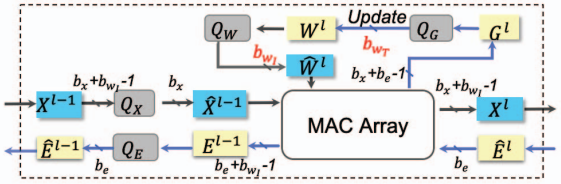

Fig. 2. Representative quantization flow of a convolution/FC layer in quantized training. Black arrow: FF pass. Blue arrow: FB pass.

II. Background

Below, we first briefly discuss meta learning with an emphasis on the adaptation phase, and then review quantization approaches for traditional training.

A. Adaption in optimization-based meta learning

Meta learning aims to derive learning models that can learn from a few training examples and thus rapidly adapt to new environments. Usually, meta learning consists of two phases: meta training and meta testing. The meta testing phase is further divided into adaption and inference. In this work, we consider the simple and powerful optimization-based meta learning. Fig. 1 depicts a typical flow of meta-learning algorithms.

As shown in Fig. 1, in the meta-training phase, the meta-learner is trained with different tasks. For the optimization-based method, the meta-learner, also serving as the model prior, is optimized to have the fast adaption ability. The meta-trained NN model can then be deployed on the edge device for fast adaption and inference. On the edge device, with a few samples from a new task, the network can learn from the new task by a few update steps. Learning in the adaptation phase is achieved by gradient descent based optimization similar to standard supervised learning but only a few samples are used.

B. Typical quantization flow in supervised learning

Quantization, i.e., employing low-bitwidth, fixed-point (FXP) representations for NN parameters, is an approach that is widely used for compressing DNN models in order to reduce the latency and energy. Fig. 2 depicts a representative quantized computation flow of a single convolution or fully-connected (FC) layer for on-chip training. The training process includes FF and FB propagation, indicated by the black and blue arrows, respectively. Weight W, activation X, error E, gradient G are quantized and denoted by $\hat{*}$. The superscripts for each parameter indicate the corresponding layer index. Here we separate error and gradient following the definition in [7], where error E is the gradient of activation X, and gradient G refers to the gradient of weight W. The blue/yellow boxes represent the data in the FF/FB propagation; the grey boxes (labeled with Q_*) represent the respective quantizer functions. The multiply-accumulate (MAC) array box is for conducting matrix multiplication. The labels along the data paths, b_*, indicate the bitwidths for the corresponding data. Note that b_{w_T}, is decided by the required precision of gradients while b_{w_I}, is the quantization bit of weights used in FF and FB computations. Thus, there are two W^l representations in the figure.

For supervised learning where training is done from scratch, quantized training has been studied though not as extensively as for quantized inference. DoReFa-Net [8] uses 1-bit weights, 2-bit activation, 6-bit gradients and float-point (FLP) errors for training. However, the weights are still accumulated with FLP. WAGE [7] adopts 2-bit weights, 8-bit activation, 8-bit gradients and 8-bit errors for quantized training, where b_{w_T} is 8 bits and b_{w_I} is 2 bits. But the method only works well for small datasets and have over 5% accuracy drop for CIFAR10 datasets. Recently, researchers have shown that higher precision (such as 16-bit) gradients are required for better performance [9]. Thus, in general, b_{w_T} and b_e in the FB stage are always greater than b_x and b_{w_I} in the FF stage. If one were to develop a specialized hardware for both training and inference, though the FF stage of the training hardware can be used for inference, the longer bitwidths required by training are wasteful in the inference phase. Such longer bitwidths lead to higher memory usage (for weights storage) and higher computation resource (e.g., for matrix multiplication).

III. Overview of AIQ Framework

The goal of AIQ is to achieve high adaption accuracy with inference-level bitwidths such that a same specialized hardware can be used for both adaption and inference without too much resource wastage. Below, we discuss several key observations that help us to achieve our goal. We then present an overview of our proposed quantization flow.

A. Observations

1) Gradients related observations: Intuitively, the quantization settings that work for FXP training-from-scratch should also work well for FXP adaption. We validate this hypothesis by applying the typical training quantization flow shown in Fig. 2 to the adaption phase. That is, b_{w_T} bits are used for weights and gradients accumulation, and b_{w_I} bits for the FF and FB stage. Experimental results[1] reveal the following observation.

Observation 1: If gradients as well as weights are represented with b_{w_T} bits, but matrix multiplications (for both FF and FB propagation) use b_{w_I}-bit weights in the adaption phase, the accuracy loss compared to the FLP representation is negligible. Reducing gradients and weights to b_{w_I} bits, however, leads to significant accuracy loss.

Observation 1 confirms that adaption requires b_{w_T} bits for weights and gradient accumulation, which leads to no reduction in model size as well as the memory space and energy.

To investigate whether it is possible to compress weight memory, we conducted experiments to study the distribution of the gradient accumulation during adaptation. Specifically, we apply the same dynamic range setting for the gradient and weight quantizer. But we split the signed FXP gradient and

[1]Details on experimental setups and datasets are given in Sec. VI.

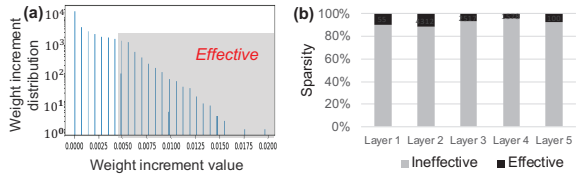

Fig. 3. Weight increment data in the adaptation phase for one layer[1]. (a) Distribution of the values of weight increments (LBG part). Shaded ones are the effective weight increments. The Y-axis is plotted in log-scale. (b) The sparsity of weight increments for different layers.

weight representations into two parts: the HBG part (containing the most significant b_{w_I} bits of the gradients/weights, i.e., $[b_{w_T}-1, b_{w_T}-b_{w_I}]$) and the LBG part (the next $b_{w_T}-b_{w_I}$ bits, i.e., bits $[b_{w_T}-b_{w_I}-1, 0]$ and one sign bit). We are particularly interested in the behavior of the LBG part of weights and gradients since the HBG part (i.e., the inference bitwidth) is essential for inference and we only need to reduce the memory needed to store the LBG part. It is easy to see that the LBG part of weights does not impact the computation immediately because only the HBG part of weights are used for computation (see Fig. 2). However, the LBG part of weights is still important as it reflects the weight increments (i.e., accumulation of the LBG part of gradients) across multiple iterations.

Fig. 3(a) depicts the distribution for the weight increments *at the end* of the adaption phase for the Omniglot 20way-5shot experiment[1]. Here $b_{w_I}=4$, $b_{w_T}=8$, and log scale is used for y-axis. We refer to the weight increments whose quantized absolute value are larger than $2^{b_{w_T}-b_{w_I}-1}$ as effective weight increments (EWIs), and the rest as ineffective weight increments (IWIs) as they do not impact the HBG part of the final weights. The EWIs are shaded in grey in Fig. 3. It is easy to see that the number of EWIs is much smaller than that of IWIs.

We further quantitatively compare EWIs and IWIs. We define sparsity of weight increments as the ratio of IWIs over all weights. Fig. 3(b) illustrates the sparsity for each layer of the network in the Omniglot 20way-5shot experiment[1]. We can see that the sparsity of every layer is very large, which means that only a small percentage of weight increments impact the final updated weights. This is summarized in the observation below.

Observation 2: The LBG parts of most weight increments in the adaptation phases are small and only a small percentage are sufficiently large to impact the final weight updates.

2) Error related observations: Similar to training-from-scratch, the distribution of errors in adaptation changes from one epoch to another, and the final loss impacts the magnitude of the FB errors of each layer. Furthermore, due to the chain rule, errors are proportional to weights. Usually, as the model gets closer to the optimal one, the loss decreases and the magnitude of errors becomes smaller and smaller. However, a unique feature of the adaption phase is that the weights do not change significantly when adapting to a single new task. We confirmed this hypothesis through detailed experimental studies (omitted due to page limit). This hypothesis leads to the following observation about the error distribution behavior.

Observation 3: In the adaptation phase, the relative ratio of the error dynamic ranges among layers keeps stable because of small weight changes.

B. AIQ *framework*

Based on the observations introduced above, we have developed a hardware-friendly quantization framework, AIQ, specifically for the adaption phase. To reduce hardware cost, AIQ employs the uniform quantizer, defined in Eq. (1), which are applied to weights, activation, errors, and gradients.

$$\begin{aligned}\hat{Z} &= \mathcal{Q}_\Theta(Z) = \mathcal{C}(\lfloor Z/s \rceil, q_{min}, q_{max}) * s \\ &= \begin{cases} \lfloor Z/s \rceil \cdot s & q_{min} \le Z \le q_{max} \\ q_{min} & \text{if } Z < q_{min}; \\ q_{max} & \text{if } Z > q_{max}; \end{cases} \end{aligned} \quad (1)$$

Here $\hat{Z}$ is the quantized counterpart of any data Z, $\mathcal{Q}_\Theta(x)$ is the quantizer based on quantization parameter setting $\Theta = \{s, q_{min}, q_{max}, b\}$, s is the quantization stepsize; q_{min} and q_{max} are the minimum and maximum value represented by the quantized representation, respectively; b is the number of bits, and $\mathcal{C}$ is the clipping function. $[q_{min}, q_{max}]$ is referred to as dynamic range. Note that q_{min} and q_{max}, can be derived once s and b are given. Specifically, for weights, errors and gradients, $q_{min} = -s \cdot 2^{b-1}$, and $q_{max} = s \cdot (2^{b-1}-1)$. For activation, $q_{min}=0$, and $q_{max} = s \cdot (2^b - 1)$.

Fig. 4 depicts the overall flow of AIQ. After the meta-training phase, the model prior is determined and the best quantization parameter settings (Θ's, indicated by the purple boxes) for weights, activation, gradients are chosen and fixed during both adaption and inference phase. Specifically, all the weights are converted from FLP to FXP offline using the quantizer with setting $\Theta_{b_{w_I}}$ and stored in on-chip memory. Similar to generally adopted practices, we apply the MSE method to choose the optimal dynamic range for weights. For activation, we collect statistics on the training datasets and decide the dynamic range based on the statistics. It can seen that except the two dashed red boxes, the rest of the flow is basically the same as that shown in Fig. 2. Thus we focus on the new components.

The two additions in Fig. 4 are for handling gradients/weights and errors. Fig. 4(a) illustrates the gradient/weight management flow including the proposed gradient gate and weight buffer unit. According to Observation 1, AIQ splits the representation of gradients and weights into two parts: the HBG part and the LBG part. A weight memory of b_{w_I}-bit wide is employed to store the low-precision weights. However, as accumulating weights in high precision (i.e., with b_{w_T} bits) is essential for achieving high adaption accuracy. A gated

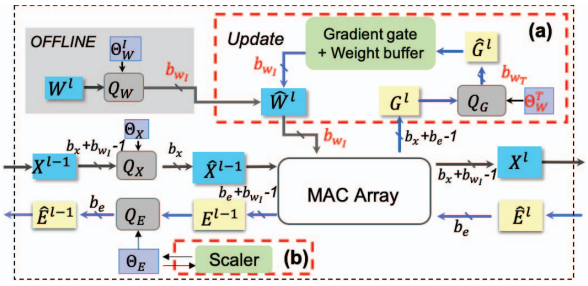

Fig. 4. The computation flow of AIQ for a convolution/FC layer. (a) The dynamic gradient management flow including the proposed gradient gate and weight buffer. (b) The dynamic error scaling flow including the error scaler.

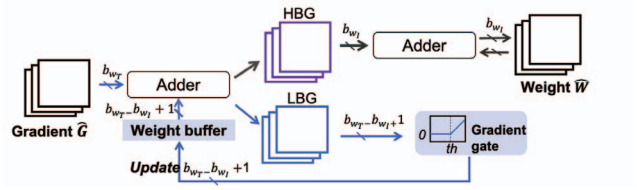

Fig. 5. The proposed dynamic gradient management method.

weight buffering (GWB) technique is proposed based on Observation 2, which identifies and maintains the LBG part of a small number of weight increments at run-time. Furthermore, since the same dynamic range is shared by the weight and gradient quantizer in AIQ, the b_{w_I}-bit weights can be obtained by rounding the b_{w_T}-bit weights directly without needing an additional quantizer as in Fig. 2. Fig. 4(b) shows the dynamic error scaler which adjusts the stepsize for the error quantizer to capture the decreasing trend of error magnitude based on Observation 3. Next, we present the details on these techniques.

IV. Detailed approaches in AIQ

In this section, we describe the design details of gated weight buffering and error scaling units.

A. Gated weight buffering

The goal of GWB is to identify and preserve the most important high-precision gradients at each update and accumulate the LBG part into the weight increments. GWB is built on the idea of split representation for gradients and weights. The HBG part of weights are denoted as $\hat{W}$ and stored in the weight memory, while the LBG part of a *subset* of the weights are stored in a dedicated weight buffer.

Fig. 5 depicts the detailed gradient management and weight update process with the shaded gradient gating and weight buffering units. The method works as follow. After a new b_{w_T}-bit gradient is computed, it is added with the *corresponding entry* of W_{LBG} in the weight buffer. The result (still referred to as gradient for simplicity) is then splitted into two parts: the HBG and the LBG part similar to weights. The HBG part including the HBG part of the newly computed gradient and the carry-out of the addition with the weight buffer entries is added with the corresponding entry in the weight memory (storing the low-precision weights). The result is the updated weight and will be stored back to the weight memory. The LBG part, i.e., the updated weight increments, is gated by a pre-determined threshold and the result is stored back into the weight buffer.

The computation flow of GWB, which is applied in each weight update, is summarized in Algorithm 1 where $Round(Z, i)$ is the rounding function that rounds Z to i bits.

Regarding the gradient gating threshold, the threshold value is determined offline and remains unchanged. Our experimental results show that the magnitudes of the gradients for each layer can be quite different. Thus, we use layer-specific thresholds from the range [0, $2^{b_{w_T}-b_{w_I}-1}$]. To ensure minimal loss in the final accuracy and reduce memory cost, we have found that having threshold values that preserve around 3% of the gradients is a good rule of thumb. We will demonstrates the effect of threshold choices in the experimental part.

Algorithm 1: Gated weight buffering at iteration t

Input: From previous iteration $t-1$, b_{w_I}-bit weights W^{t-1} and (b_{w_T}-b_{w_I} +1)-bit weight buffer entries WB^{t-1}; b_{w_T}-bit gradients G^t and threshold θ

Output: W^t, WB^t

```
G_n = G^t + WB^{t-1}; // Update gradients;
G_hb = Round(G_n, b_wI); // Split gradients;
G_lb = G_n - G_hb;
for g in G_lb do
    if abs(g) > θ then
        WB^t.add(g); // Store g to weight buffer;
    end
end
W^t = W^{t-1} - G_hb;
return W^t, WB^t
```

Initial values in the weight buffer can greatly impact the accuracy since there is only a limited number of weight updates in the adaption phase. We introduce a 3-step initialization procedure for the weight buffer: (1) use the uniform quantizer to quantize the FLP initial weights (obtained from offline meta-training) to b_{w_T} bits; (2) split the b_{w_T}-bit weights to the HBG part and LBG part; (3) apply threshold-based gating to the LBG part of the weights to sparsify them. The resulting non-zero entries are stored as the initial values of the weight buffer.

B. Dynamic error scaling

AIQ applys a dynamic scaler (Fig. 6(b)) to adjust the stepsize of the FB error quantizer for each layer in order to reduce quantization errors. As discussed in Sec. III-A1, the errors dynamic range can change significantly from one iteration to another, thus a fixed stepsize is not acceptable. Usually, to minimize quantization errors, the dynamic range of FB errors is measured for each batch and the most appropriate stepsize is determined accordingly. This method reduces quantization errors but incurs high hardware cost. Observation 3 states that the relative ratio of the error dynamic ranges among layers keeps stable and is highly related to the loss at the output layer. Based on this, we propose a lightweight and effective method to dynamically adjust the error stepsize for each layer.

The error scaler works as follows. An initial stepsize for the error quantizer at each layer is decided offline by collecting statistics on training datasets, which is optimal for the first update. For subsequent updates during adaption, the scaler compares the loss with a pre-determined threshold value and reduces the stepsize for each layer by half if the loss is below the pre-determined threshold. This operation can be viewed as guiding the stepsize decay as the adaptation progresses, and the threshold value settings will be discussed in Sec.VI. Adjusting the stepsize to half can be simply implemented by a shift operation in hardware and experiment results have shown that the final accuracy can be improved with the setting.

V. Hardware implementation for AIQ

In this section, we present a hardware design to support the GWB strategy based on a typical FXP CMOS accelerator

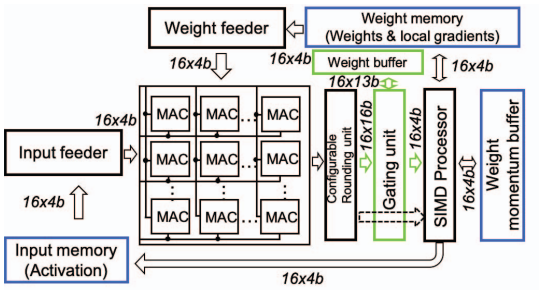

Fig. 6. Overview of an architecture for implementing AIQ. The architecture is based on that in [3]. The green boxes are the new units (GWB components including gating unit and weight buffer). Dotted line indicates the original flow without the new units.

architecture for training. Specifically, we chose the state-of-art SIMD-like learning accelerator proposed in [3] as the baseline. We focus on the support for the GWB technique. The hardware implementation of the dynamic error scaling unit is quite straightforward and we omit its discuss due to page limit.

The AIQ architecture is shown in Fig. 6. The components from the base architecture in [3] are shown in black/blue boxes while the newly added units are highlighted in green. The base architecture supports momentum-based optimization, and contains input memory, weight memory, weight momentum memory, 16X16 MAC array, a rounding unit, a SIMD processor, and a controller. The new units include the gradient gating unit and weight buffer. Note that the base architecture is built with FXP 16-bit precision. Our AIQ design reduces the bitwidths of the original 16-bit memory to 4 bits and the precision of the MAC from 16x16-bit to 4x4-bit. Minor modifications are also made to the rounding unit.

We briefly discuss the dataflow of the weight gradient computation (WG) phase here and omit the other processes due to page limit. First, the gradients at layer l, G^l are computed using activation X^{l-1} and errors E^l. The accelerator fetches X^{l-1} from the input memory, and computes E^l from the previous layer; then feeds them into the MAC array. The output of the MAC array, G^l, are fed to the *rounding unit* to get the b_{w_T}-bit gradient which is then sent to the newly added gating unit.

The detailed design of GWB is shown in Fig. 7, which is the major additional hardware in implementing AIQ. We use a compressed-sparse format for the weight buffer in order to keep the memory usage at minimal. An indexing scheme is needed to enable fast access to support SIMD operations. Here we adopt the bitmap format to encode the sparse data. The weight buffer implemented in SRAMs includes an extra index bank to store the bitmap information. The memory controller controls the data store and fetch based on the index information.

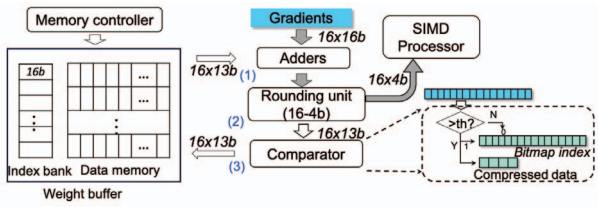

Fig. 7. Details of the GWB components.

The width of the index bank is set to 16 bits and the width of the buffer is set to $(b_{w_T} - b_{w_I} + 1) * 16$ bits (for 16-way parallelism). The size of the weight buffer is a tradeoff between the hardware cost and the accuracy related to the gradient-gate threshold. We will evaluate the buffer size choices in Sec. VI. The unit also contains a 16-way b_{w_T}-bit adder, a rounding unit, and a comparator for the gating function. As shown in Fig. 7, the GWB unit operates in three steps. (1) Gradient updating by the 16-way b_{w_T}-bit adders. (2) Rounding and splitting by the rounding unit. (3) Gated buffering by a $(b_{w_T} - b_{w_I})$-bit comparator. As the data fetch and store of the weight buffer are not on the critical path of gradient computation, the latency of the additional modules mainly comes from the gradient update step, which is negligible.

VI. Evaluation

A. Experimental setup

To validate the accuracy of AIQ, we implemented it in the PyTorch framework. The architecture of the task-learner follows the general practice of optimization-based few-shot learning, which stacks four blocks, each consisting of a 33 convolution, max-pooling, batch-normalization and ReLU activation. The model prior was trained using MAML [2] with the double-precision FLP representation on GPU. Two widely used few-shot learning datasets, Omniglot [10] and MiniImageNet, were considered in the experiments. The experimental protocol involved fast learning of N-way classification with 1 or 5 shots. Inference accuracy after adaption on new tasks are measured.

To study hardware cost, we developed RTL-level Verilog models of the proposed GWB components and synthesized them with Cadence Encounter for a CMOS 45nm library. The proposed components are based on a 16-way design to support the SIMD operations and enable more parallelism for the accelerator. The energy consumption consumed by on-chip memory were calculated using CACTI [11].

B. Accuracy study

Here, we set b_x and b_e (in Fig. 4) to 4 bits for all layers in the CNN as default unless otherwise specified. AIQ also needs to decide offline the b_{w_T} value for each dataset as shown in Fig. 4. Because MiniImageNet is a harder task than Omniglot, experiments show that b_{w_T} for MiniImagenet should be no less than 16 bits while 8-bit b_{w_T} is enough for Omniglot. Table I summarizes accuracy results collected for both Omniglot and MiniImageNet. The FLP column shows the accuracy results with every parameter represented with FLP; FXP 8 (Omniglot) / FXP 16 (MiniImageNet) and FXP 4 column are the cases that weights and gradients are represented with 8/16-bit and 4-bit, respectively; The last (AIQ) column shows the accuracy results obtained by AIQ using 4-bit weights. Two thresholds (0.13, 0.07) are set for the dynamic error scaler as the decay points. The data clearly show that the accuracy achieved by AIQ is similar to the FXP 8/16-bit one and notably better than FXP 4, especially for the more challenging MiniImageNet dataset.

The choice of the threshold value for gating gradients impacts the accuracy. Generally, a higher threshold leads to more gradients being dropped and less memory needed, but lower accuracy. Fig. 8 shows accuracy vs. the threshold value

TABLE I
ACCURACY DATA FOR OMNIGLOT AND MINIIMAGENET

Omniglot	FLP	FXP 8	FXP 4	AIQ
5way-1shot	0.986	0.972	0.868	0.97
5way-5shot	0.9872	0.992	0.9872	0.992
20way-5shot	0.976	0.955	0.897	0.955
MiniImageNet	FLP	FXP 16	FXP 4	AIQ
5way-1shot	0.434	0.439	0.297	0.442
5way-5shot	0.639	0.602	0.369	0.595

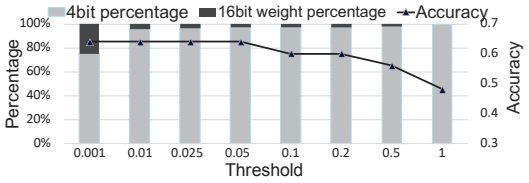

Fig. 8. Analysis of threshold impacts on accuracy.

for the MiniImageNet 5way-5shot case, where 13-bit weight increments are buffered. The threshold was swept from 0.001 to 1 (x2048). As seen from the plot, a threshold value of 0.05 (x2048) with 3% high-precision weight increments being kept, is a good choice for trading off accuracy with memory usage.

C. Memory and hardware overhead study

The goal of AIQ is to reduce the weight memory by using inference bitwidths in adaption. We examine quantitatively the memory saving. Fig. 9(a) depicts the weight-related memory needs for Omniglot and MiniImageNet. In the baseline, weights are represented in b_{w_T}-bit FXP (8/16-bit for the two datasets as mentioned in the accuracy study), while in AIQ the total weight memory includes the 4-bit weight memory and the (b_{w_T}-b_{w_I} +1)-bit weight buffers (as well as the corresponding index bank). The number of entries in the buffer is decided by the threshold with 3% of the total weight count. As can be seen, 41% and 70% memory space can be saved by AIQ for Omniglot and MiniImageNet, respectively. By using CACTI, we estimated that similar energy savings are also achieved.

We next study the reduction of total on-chip memory with AIQ. The total memory includes input/weight/weight momentum memories. Input memory stores all intermediate activation data in a mini-batch. As discussed in [3], a grouping method divides a mini-batch into groups to reduce the input memory needs. Here, we set the group size for Omniglot and MiniImageNet to 10 and 5, respectively. For both the baseline and AIQ, the size of input memory is set the same with 4-bit activation. The weight momentum memory has the same size as the weight memory. As shown in Fig. 9, AIQ can achieve 33% and 47% saving in terms of total memory for the two datasets.

We further investigate the area and power overhead of AIQ as discussed in Sec. VI-A. By using Cadence Encounter, we obtained gate count and power consumption of the GWB components as well as two MAC arrays with 4-bit and 8-bit precision. We then estimate the 16x16 MAC array gate count and power based on the coreresponding single MAC values. Note that the actual power of the 16x16 MAC array should be higher than the value reported here. Table II summarizes theses data, which show that the gate count of GWB components is

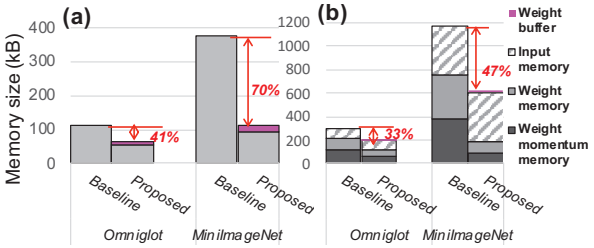

Fig. 9. (a) Memory usage for weights. (b) Total memory usage.

only 3.3% of the 16x16 4-bit MAC array and the power of GWB is 2.9% of the 4-bit MAC array. According to the data reported in [3], the MAC array only takes about 11% of the total chip area. Hence the overhead added by the GWB components is negligible with respect to the total chip area.

TABLE II
RESOURCE USAGE OF MAC ARRAYS AND GWB COMPONENTS

	MAC array (8-bit)	MAC array (4-bit)	GWB
Gate count (NAND2)	71538	25984	845
Power (mW)	17.72	6.52	0.19

VII. CONCLUSIONS

This paper introduces a quantization framework, AIQ, to support efficient adaption with low-bitwidth FXP representations. AIQ exploits unique data characteristics in the NN adaption phase and saves 41% and 70% weight memory for two widely used datasets with minimal hardware overhead and accuracy drop. The framework is efficient in performing on-chip adaption and inference at the edge for meta learning. The framework also has the potential to support on-site continual and incremental learning capabilities, which is left for future study.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation (NSF) under grant CCF-1640081.

REFERENCES

[1] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, "Meta-learning in neural networks: A survey," *arXiv preprint arXiv:2004.05439*, 2020.
[2] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *ICML*, 2017.
[3] S. Yin and J.-S. Seo, "A 2.6 tops/w 16-bit fixed-point convolutional neural network learning processor in 65-nm cmos," *ISSCL*, pp. 13–16, 2019.
[4] C. Li and et.al, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," *Nat. Commun.*, vol. 9, no. 1, pp. 1–8, 2018.
[5] R. Banner and et.al., "Post training 4-bit quantization of convolutional networks for rapid-deployment," in *NeurIPS*, pp. 7950–7958, 2019.
[6] Y. Choukroun and et.al., "Low-bit quantization of neural networks for efficient inference," in *ICCVW*, pp. 3009–3018, IEEE, 2019.
[7] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," *arXiv preprint arXiv:1802.04680*, 2018.
[8] S. Zhou and et.al, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.
[9] Y. Yang and et.al., "Training high-performance and large-scale deep neural networks with full 8-bit integers," *Neural Networks*, vol. 125, pp. 70–82, 2020.
[10] B. Lake, R. Salakhutdinov, J. Gross, and J. Tenenbaum, "One shot learning of simple visual concepts," in *Proceedings of the annual meeting of the cognitive science society*, vol. 33, 2011.
[11] R. Balasubramonian and et.al., "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *TACO*, vol. 14, no. 2, pp. 1–25, 2017.