IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 22, 2021, accepted December 14, 2021, date of publication December 20, 2021,
date of current version December 27, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3136888

Arithmetic Coding-Based 5-Bit Weight Encoding and Hardware Decoder for CNN Inference in Edge Devices

JONG HUN LEE[1], (Student Member, IEEE), JOONHO KONG[1,2], (Member, IEEE), AND ARSLAN MUNIR[3], (Senior Member, IEEE)
[1]School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, South Korea
[2]School of Electronics Engineering, Kyungpook National University, Daegu 41566, South Korea
[3]Department of Computer Science, Kansas State University, Manhattan, KS 66506, USA

Corresponding author: Joonho Kong (joonho.kong@knu.ac.kr)

This research was supported by Kyungpook National University Research Fund, 2020.

ABSTRACT Convolutional neural networks (CNNs) have gained a huge attention for real-world artificial intelligence (AI) applications such as image classification and object detection. On the other hand, for better accuracy, the size of the CNNs' parameters (weights) has been increasing, which in turn makes it difficult to enable on-device CNN inferences in resource-constrained edge devices. Though weight pruning and 5-bit quantization methods have shown promising results, it is still challenging to deploy large CNN models in edge devices. In this paper, we propose an encoding and hardware-based decoding technique which can be applied to 5-bit quantized weight data for on-device CNN inferences in resource-constrained edge devices. Given 5-bit quantized weight data, we employ arithmetic coding with range scaling for lossless weight compression, which is performed offline. When executing on-device inferences with underlying CNN accelerators, our hardware decoder enables a fast in-situ weight decompression with small latency overhead. According to our evaluation results with five widely used CNN models, our arithmetic coding-based encoding method applied to 5-bit quantized weights shows a better compression ratio by 9.6× while also reducing the memory data transfer energy consumption by 89.2%, on average, as compared to the case of uncompressed 32-bit floating-point weights. When applying our technique to pruned weights, we obtain better compression ratios by 57.5×–112.2× while reducing energy consumption by 98.3%–99.1% as compared to the case of 32-bit floating-point weights. In addition, by pipelining the weight decoding and transfer with the CNN execution, the latency overhead of our weight decoding with 16 decoding unit (DU) hardware is only 0.16%–5.48% and 0.16%–0.91% for non-pruned and pruned weights, respectively. Moreover, our proposed technique with 4-DU decoder hardware reduces system-level energy consumption by 1.1%–9.3%.

INDEX TERMS Convolutional neural networks, arithmetic coding, weight compression, edge devices, 5-bit quantization.

I. INTRODUCTION

Recently, convolutional neural networks (CNNs) have been widely deployed in many artificial intelligence (AI) applications. Due to the advancements of the processing units (e.g., central processing units (CPUs), graphics processing unit (GPUs), neural processing units (NPUs), etc.), CNNs are lately being executed on-device. Furthermore, CNNs are recently being employed even in embedded resource-constrained Internet-of-Things (IoT) devices for fast and efficient inferences for vision-based AI applications. However, the huge weight (parameter) size of the CNNs is one of the major hurdles for deploying the CNNs in resource-constrained edge devices. For example, ResNet-152 [1] and VGG-16 [2], which are very widely used CNN models, have weight sizes of 235MB and 540MB, respectively (in 32-bit floating-point precision). In addition, for improving the accuracy of the CNN models, CNN models and parameter sizes are expected to be further increased. Though the large weight size causes several important challenges for

The associate editor coordinating the review of this manuscript and approving it for publication was Seok-Bum Ko.

CNN deployment in resource-constrained devices, one of the most serious problems is limited memory (and/or storage size) of these devices. The large weight data cannot be often fully loaded into the small memory and storage of the resource-constrained devices. In addition, the large weight size inevitably causes latency, power, and energy overhead when transferring the weight data between the storage/memory and the processing units, such as CPUs, GPUs, NPUs, or accelerators, which is not desirable for resource-constrained edge devices.

In order to resolve these problems, several well-known techniques such as weight pruning [3] and quantization [4] have been introduced. The weight pruning increases the sparsity of the weight data by replacing the near-zero weight elements with zero-valued elements. The quantization reduces weight elements' size with a negligible loss of the CNN accuracy. For example, while the conventional precision of the elements in CNNs is single-precision floating-point (32-bit), 8-bit integer and 16-bit fixed-point precisions are also widely used for cost-efficient CNN inferences, which can reduce the weight size by 4× and 2×, respectively. Even further, as a more aggressive solution, several works have proposed to use 5-bit weight elements for deploying the CNN models in resource-constrained systems [5], [6]. Though, these works have shown successful results on reducing the weight data size, we could further reduce the weight size by applying the data encoding schemes such as Huffman coding or arithmetic coding. By only storing the encoded (hence, the reduced size of) weight data in device's memory and/or storage, we could enable more cost-efficient deployment of the CNN models in resource-constrained devices.

In this paper, we introduce an arithmetic coding-based 5-bit quantized weight compression technique for on-device CNN inferences in resource-constrained edge devices. Once the weight elements are quantized to a 5-bit format (quantization can be done by other methods such as [6]), we leverage arithmetic coding for weight compression, which has been generally employed for entropy-based data compression (e.g., for image compression in [7] and [8]). In addition, we also employ range scaling for lossless compression, meaning that there is no accuracy loss in CNN inferences as compared to the case of using uncompressed 5-bit weight element. Compared to Huffman coding-based compression, which is commonly used, our arithmetic coding-based technique leads to better compression ratio, resulting in less memory and storage requirements for weights. In addition, when applying our technique to the pruned weights, one can obtain much higher compression ratio as compared to Huffman coding-based weight compression. For an in-situ weight decompression for edge devices which contain a CNN accelerator or NPU, we propose a hardware decoder which can decompress the compressed weight with a small latency overhead.

We summarize our contributions as follows:

- We introduce a lossless arithmetic coding-based 5-bit quantized weight compression technique;
- We propose a hardware-based decoder for in-situ decompression of the compressed weights in the NPU or CNN accelerator, and also implement our hardware-based decoder in field-programmable gate array (FPGA) as a proof-of-concept;
- Our proposed technique for 5-bit quantized weights reduces the weight size by 9.6× (by up to 112.2× in the case of pruned weights) as compared to the case of using the uncompressed 32-bit floating-point (FP32) weights;
- Our proposed technique for 5-bit quantized weights also reduces memory energy consumption by 89.2% (by up to 99.1% for pruned weights) as compared to the case of using the uncompressed FP32 weight;
- When combining our compression technique and hardware decoder (16 decoding units) with various state-of-the-art CNN accelerators [9]–[11], our technique incurs a small latency overhead by 0.16%–5.48% (0.16%–0.91% for pruned weights) as compared to the case without our proposed technique and hardware decoder.
- When combining our proposed technique with various state-of-the-art CNN accelerators [9], [10], our proposed technique with 4 decoding unit (DU) decoder hardware reduces system-level energy consumption by 1.1%–9.3% as compared to the case without using our proposed technique.

II. RELATED WORK

There have been many works for weight size reduction or compression. In [3], Han *et al.* have proposed a network pruning technique that removes weak (i.e., a weight value is less than a certain threshold) synapse connections. To improve the accuracy, it re-trains a model with the pruned weights. After pruning, the number of parameters is reduced by 9× and 13× in AlexNet and VGG-16, respectively with a negligible accuracy loss. In [4], Han *et al.* have also proposed a weight compression technique with Huffman coding. It presents a quantization method which reduces a size of the weight element with a small accuracy loss. They observed that the accuracy of deep neural networks (DNNs) does not hugely decrease until 4-bit precision (2.0% and 2.6% accuracy drop in top-1 and top-5 accuracies, respectively). When combining the quantization with weight pruning and data compression by Huffman coding, storage reduction of 35×– 49× has been reported. Similarly, in [12], Choi *et al.* have proposed a weight size reduction technique that exploits quantization and Huffman coding. They proposed an entropy-constrained quantization method which minimizes accuracy losses under a given compression ratio. In addition, with Huffman coding, it achieves over 40× compression ratio with less than 1% accuracy losses.

In [13], Ko *et al.* have proposed a JPEG-based weight compression technique. In order to minimize accuracy losses from the JPEG encoding, it adaptively controls a quality factor according to error sensitivity. It achieves a 42× compression ratio for multilayer perceptron (MLP)-based network with

an accuracy loss of 1%. In [14], Ge *et al.* have proposed a framework that reduces weight data size by using approximation, quantization, pruning, and coding. For the coding method, the framework encodes only non-zero weights with their positional information. It is reported that the framework proposed in [14] shows a compression ratio of 21.9× and 22.4× for AlexNet and VGG-16, respectively. In [15], Reagan *et al.* have proposed a lossy weight compression technique that exploits Bloomier filter and arithmetic coding. Due to the probabilistic nature of Bloomier filter, it also retrains weights based on lossy compressed weights. It shows a compression ratio of 496× in the first fully connected layer of LeNet5. However, it also shows an accuracy loss of 4.4% in VGG-16 top-1 accuracy. In [16], Choi *et al.* have proposed a universal DNN weight compression framework with lattice quantization and lossless coding such as bzip2. By leveraging the dithering which adds a randomness to the sources, the framework proposed in [16] can be universally employed without the knowledge on source distribution. In [17], Young *et al.* have proposed a transformation-based quantization method. By applying the transformation before the quantization, it achieves a significantly low bit rate around 1–2 bits per weight element. As explained in [17], the quantization method is orthogonal to our compression technique, meaning that the transform quantization and our technique can be deployed synergistically. Although our technique is geared towards 5-bit quantized weight compression, our arithmetic coding-based compression/decompression can be extended to N-bit weight elements.

The weight size reduction techniques introduced so far have mostly been focusing on 8-bit, 16-bit, or 32-bit precision weights, meaning that the weight size reduction for 5-bit weights has been largely overlooked. On the contrary, our work is based on 5-bit quantized weights which is more suitable for resource-constrained edge devices. Moreover, for fast in-situ decompression of the weights, we also propose a novel hardware decoder, which has not been introduced yet.

III. BACKGROUND

A. 5-BIT WEIGHT QUANTIZATION FOR CNNs

Recently, for deploying the CNNs in tightly resource-constrained systems, several researches have been focusing on reducing the weight element size even less than 8-bit integer. One of the representative approaches is 5-bit quantization, which represents each weight element with either 5-bit log-based or 5-bit linear-based one. Between the log- and linear-based quantization, the log-based representation is more preferred. This is because (1) log-based approach can achieve a better accuracy as compared to the linear-based approach [5], and (2) log-based representation is more hardware-friendly as it can replace multiplier with shifter [18]. It has been reported that the log-based representation shows very small accuracy drops (e.g., 1.7% and 0.5% top-5 accuracy drop in AlexNet and VGG-16, respectively).

B. ENTROPY-BASED CODING

The entropy-based coding is a very widely used data compression scheme. It adopts a variable length encoding based on the occurring probability of a certain symbol in a datastream. Based on the probability, it assigns different code lengths for encoding of the datastream. There are two widely used entropy-based coding schemes: Huffman coding and arithmetic coding. When encoding the data with Huffman coding, one generates Huffman tree (a type of binary tree) which represents how data is encoded for each symbol based on the occurring probabilities of each symbol. When encoding the data, one performs a tree traversal from the root node until the symbol is found from the tree. On the other hand, arithmetic coding encodes the data by mapping a stream of the symbols into the real number space [0, 1). As introduced in Section II, several weight compression approaches based on Huffman coding have been introduced [4], [12] because of its simplicity whereas little attention is paid to arithmetic coding. However, in general, arithmetic coding is known to result in a better compression ratio as compared to Huffman coding [19]. Hence, in our work, we employ arithmetic coding in order to compress the 5-bit quantized weight data to aim at better compression ratio than Huffman coding.

IV. WEIGHT COMPRESSION WITH HARDWARE-BASED DECODING

A. ARCHITECTURE AND DESIGN OVERVIEW

For a real-world deployment of our weight compression technique, we introduce a system overview and execution flow to support fast and cost-efficient weight encoding/decoding. The overall architecture and execution flow of our proposed technique are illustrated in Figure 1. First, the weight quantization and encoding (upper part of Figure 1) are performed offline. In the cloud (or datacenter), the trained 32-bit FP weight elements[1] can be quantized to 5-bit format (and can also be pruned), and compressed by using our arithmetic coding-based compression technique, and sent to resource-constrained edge devices for CNN inference. The compressed weight data is stored in the memory or storage of the edge devices and will be accessed when running the CNN inference. We assume there is a CNN accelerator (or NPU) in the edge device because recent edge devices are widely adopting CNN accelerators (e.g., edge tensor processing unit in Google Coral platform [20]). Please note that the 5-bit quantization can be performed with any already proposed technique (e.g., [4], [21], and [22]), and thus the proposal of a 5-bit quantization method is outside the scope of this paper. A clear description of our contributions and execution flow is depicted in Figure 2.

In case of CNN inference in the baseline system (i.e., without our technique), the weight data will be directly loaded into CNN accelerator's private local memory (PLM). In this case,

[1] Instead of FP32, our compression technique can also be applied to fixed-point (e.g., 16-bit) weight values as long as they can be quantized to 5-bit weight format.

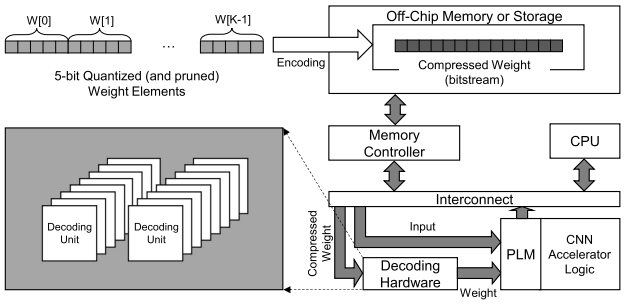


FIGURE 1. Architecture and execution flow of our proposed technique.

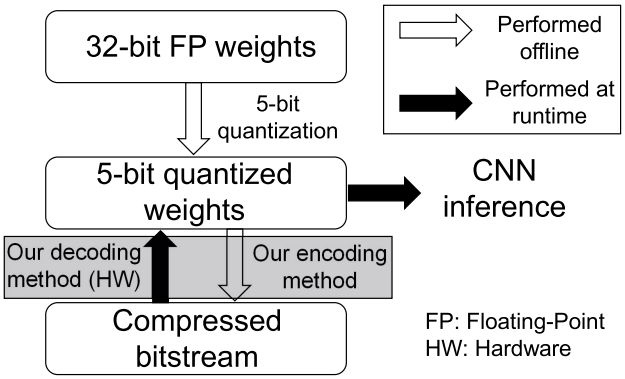


FIGURE 2. A quantization, encoding, and decoding flow for CNN inference in our proposed technique. The gray-shaded part denotes the main contributions of our proposed technique.

non-compressed 5-bit weight data is fully or partially loaded into CNN accelerator's PLM. However, with our proposed technique, before we send the weight data to the CNN accelerator, we need a fast in-situ decompression (i.e., decoding) of the compressed weight data, which necessitates a hardware decoder. The hardware decoder receives the bitstream of the compressed weight and generates the original 5-bit quantized weight data. In the hardware decoder, there are multiple decoding units (DUs) to expedite the runtime decoding process. Please note that our decoding hardware does not have any dependency to the CNN accelerator that can perform convolution operations with 5-bit quantized weight.

B. ARITHMETIC CODING-BASED WEIGHT ENCODING AND DECODING WITH RANGE SCALING

1) ALGORITHM OVERVIEW

In this subsection, we describe an overview of our algorithm. Figure 3 summarizes the comparison between the theoretical arithmetic coding-based encoding (a), binary arithmetic coding-based encoding without range scaling (b), and our binary arithmetic coding-based encoding with range scaling (c) and decoding (d). As shown in Figure 3 (a), arithmetic coding encodes the original data into a certain real number between 0 and 1 ([0, 1)). Theoretically, the arithmetic coding can compress any raw data (i.e., a sequence of the symbols) to one real number since we can have an infinite number of real numbers between 0 and 1. In general, however, when we encode certain data to a bitstream, the encoded data should be mapped to a finite binary number space, which can cause underflow. In this case, the encoded data may be lossy because different binary data can be mapped to the same encoded binary due to the limited space for data encoding. As shown in Figure 3 (b), if we have a long sequence of the weight elements and we do not have a sufficiently large binary mapping space, the underflow can occur. This is because the feasible mapping space will become smaller and smaller as more weight elements are encoded.

In our proposed technique, we also map the weight elements into the unsigned integer-based binary mapping space. Since our compression (i.e., encoding) technique aims to generate losslessly encoded data, we also employ a range scaling method that can adaptively scale the range of the binary mapping space. In the case of encoding (Figure 3 (c)), depending on the mapped sub-range for a certain element, we adaptively scale this sub-range according to the range scaling condition (we explain the details on the scaling condition in the next subsection). In this case, we record the scaling information as well as the information on the mapped and scaled sub-range to the compressed bitstream. In the case of decoding (Figure 3 (d)), by referring to a sliding window (gray shaded area in Figure 3 (d)) within a bitstream, we find which sub-range the unsigned integer (the number Z converted from the binary in the sliding window) belongs to. By referring to the found sub-range, we decode a weight element to which the sub-range corresponds. By shifting the sliding window,[2] we decode the next weight element in the similar way we describe above.

2) WEIGHT ENCODING ALGORITHM

In this subsection, we explain how we compress (i.e., encode) the 5-bit quantized weight data in details. Figure 4 shows a pseudocode of our arithmetic coding-based weight encoding with range scaling. For input, 5-bit quantized weight data (W), the occurrence probabilities for each weight value ($PROB$), and the number of weight elements (K) (i.e., the number of parameters) are required. To obtain $PROB$, we can calculate the probability based on how many times each weight appears in the weight data. The output is an encoded weight bitstream (BS). Prior to encoding, we need to set variables: N means the number of bits for mapping the weight data to an unsigned binary with arithmetic coding. RS contains range scaling information, which is initially set to 0. The MAX, $HALF$, and QTR are calculated according to the N. For initialization, low and $high$ are first set to 0 and MAX, respectively. We also need to collect the cumulative probabilities for each weight value[3] ($F[X] = PROB[x < X]$ where $0 \leq X \leq 32$).

[2]To decode one weight element, the sliding window can be shifted by more than 1-bit depending on the range scaling condition.

[3]Though the actual weights are interpreted by 5-bit log-based or linear-based values, here we represent 5-bit binary as an unsigned integer (0∼31) for an ease of explanation.

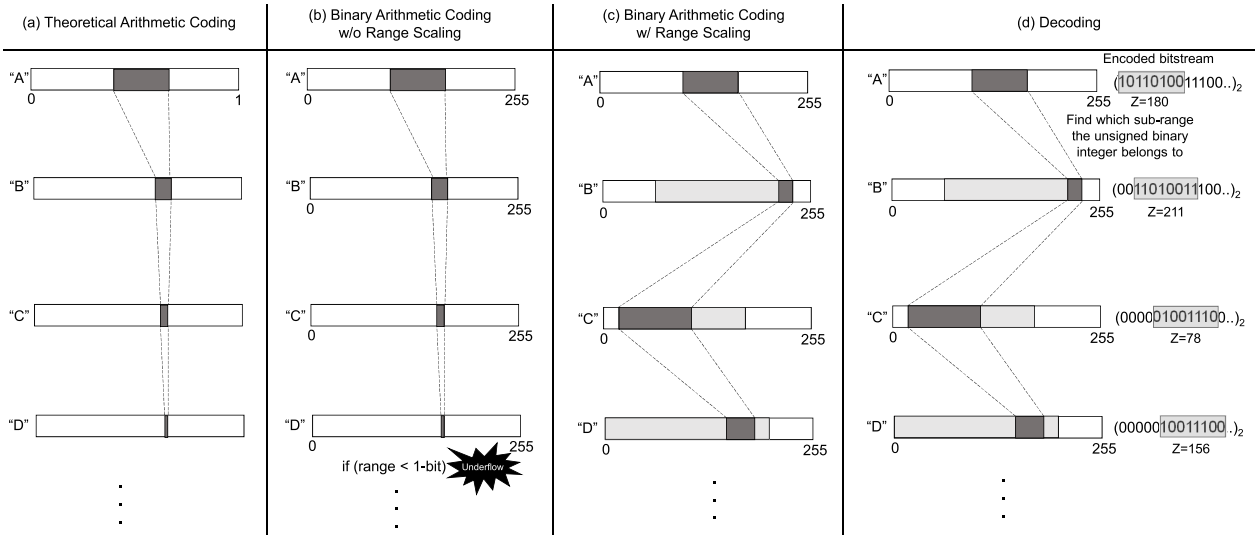


FIGURE 3. An overview of theoretical arithmetic coding (a), binary arithmetic coding without range scaling (b), and our encoding (c) and decoding algorithm (d). For illustrative purpose, we assume that the unsigned integer binary is 8-bit size (thus, mapping space is 0–255). In this example, we encode or decode a sequence of symbols (weight elements) "A", "B", "C", and "D". In (d), the numbers specified for the bitstream and Z values are merely exemplary values.

Pseudocode for Encoding (Compression)

INPUT : W – 5-bit quantized weight elements
PROB – Probability of each weight value occurrence in W
K – # of weight elements
OUTPUT : BS – Encoded weight bitstream

N = # of bits for mapping the weight data to an unsigned integer number
RS = 0
MAX = 2^N-1, HALF = *round*(MAX/2), QTR = *round*(MAX/4)
low = 0, high = MAX
F[X] = PROB[x < X] (X=0 to 32, F[0]=0, F[1]=PROB[0], ... F[32]=1)

1 **for** i=0 to K–1
2 range = high – low
3 high = low + *floor*(range * F[W[i]+1])
4 low = low + *floor*(range * F[W[i]])
5 **while** high < HALF **or** low >= HALF
6 **if** low >= HALF
7 write 1_2 and RS-bits of 0_2s to BS, RS=0, low=low–HALF, high=high–HALF
8 **else**
9 write 0_2 and RS-bits of 1_2s to BS, RS=0 **endif**
10 low=low << 1, high=high << 1 **endwhile**
11 **while** low >= QTR **and** high < 3*QTR
12 RS++, low = (low–QTR) << 1, high = (high–QTR) << 1 **endwhile**
13 **endfor**
14 RS++
15 **if** low <= QTR
16 write 0_2 and RS-bits of 1_2s to BS
17 **else**
18 write 1_2 and RS-bits of 0_2s to BS **endif**

FIGURE 4. Pseudocode of our weight encoding technique.

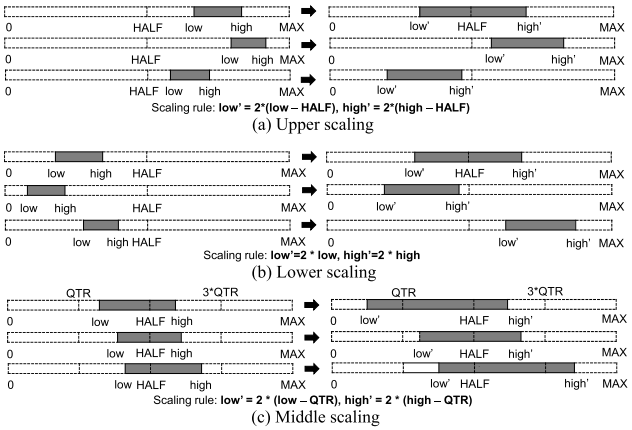


FIGURE 5. Three possible cases and scaling rules for range scaling.

The encoding procedure is performed by per weight element basis. For each weight element (from 0 to $K-1$), we set the *high* and *low* values to a range for mapping a certain weight element to an unsigned integer binary number space (lines 2-4 in Figure 4). We call this range ([*low*, *high*]) as *sub-range*. When mapping the elements in the sub-range with range scaling, there can be three cases of the range scaling: (a) upper scaling, (b) lower scaling, and (c) middle scaling. These three cases of the range scaling are also shown in Figure 5. Lines 5-7 and 10 in Figure 4 correspond to the case (a) in Figure 5: upper scaling. In this case, we write 1_2 followed by RS-bits of 0_2s in the bitstream (for example, if $RS = 3$, we write $(1000)_2$ to BS), reset RS to 0, and update *low* and *high* values by following the upper scaling rule for further range scaling. The reason why we write 1_2 in the upper scaling case is that the sub-range is in the upper half of the binary mapping space, meaning that the most-significant bit (MSB) of the mapped unsigned integer binary value will be 1_2. RS-bits of 0_2s incorporate the information on how many middle scaling has been done before. Performing more middle scalings implies that the original sub-range (i.e., before performing the upper and middle scalings) was closer to the center point of the range (Figure 6 (a)). This is why we write more number of 0_2s as we perform more middle scalings in the previous loop iteration. In case (b) in Figure 5

(lower scaling: lines 5 and 8-10 in Figure 4), we write 0_2 (MSB of the mapped unsigned integer binary value will be 0_2) followed by RS-bits of 1_2s in the bitstream, reset RS to 0, and update low and $high$ values by following the lower scaling rule for further range scaling. As depicted in Figure 6 (b), the reason why we write 0_2 and RS-bits of 1_2s can be explained in a similar way of the upper scaling case. In the case (c) in Figure 5 (middle scaling: lines 11-12 in Figure 4), we do not encode the element data and only scale the sub-range by following the middle scaling rule as long as the condition of the middle scaling case (line 11 in Figure 4) is met while we also keep incrementing RS to track how many times of middle scaling has been done. There can be a case where the sub-range does not belong to any of the three scaling cases. In this case, we do not scale the sub-range and we move onto the next iteration (i.e., next iterations of the **for** loop or terminate the **for** loop in the case of the last loop iteration). This procedure (lines 1-13 in Figure 4) is repeatedly performed until the entire W elements are encoded to BS. After that, lines 14-18 in Figure 4 perform a bitstream writing that corresponds to the last part of the weight data which has not been encoded yet. In this part, we do not have a middle scaling case and only record the bits by following either upper or lower scaling rule. Although our technique maps weight elements to an N-bit unsigned integer binary number space with arithmetic coding, the encoded bitstream also contains range scaling information (RS) along with iteratively appended binary bits depending on the scaling condition (i.e., lower and upper scaling: Lines 6-9 in Figure 4). Thus, the encoded bitstream is typically much longer than N-bits.

3) WEIGHT DECODING ALGORITHM

An overall structure of the decoding (i.e., decompression) procedure is very similar to that of the encoding procedure. The decoding procedure exactly follows the range calculation and scaling while we perform the weight element mapping with a part of the bitstream, which is an inverse operation of the encoding procedure. Figure 7 presents a pseudocode for the decoding. We need encoded weight bitstream (BS), the occurrence probabilities for each weight value ($PROB$), and the number of original weight elements (K) as inputs while the output of the decoding procedure is original 5-bit weight elements (W). We omit the explanation for variable initialization because the variable setting is same as the encoding except for $Z(idx)$ which corresponds to the N-bits starting from the bit index idx in the BS where N is the number of bits used for mapping the encoded data to an unsigned binary number space, which is same as in the encoding. For example, if $N = 8$, the $Z(idx)$ will be the 8-bits starting from the bit index 'idx', which can also be represented by an unsigned integer number ranging from 0 to 255 ($= 2^8 - 1$). Please note that $Z(idx)$ shifts from the starting point of the encoded bitstream in a sliding window manner as we explained in Section IV-B1.

The decoding procedure is performed until the encoded bitstream (BS) is fully decoded into original 5-bit weight

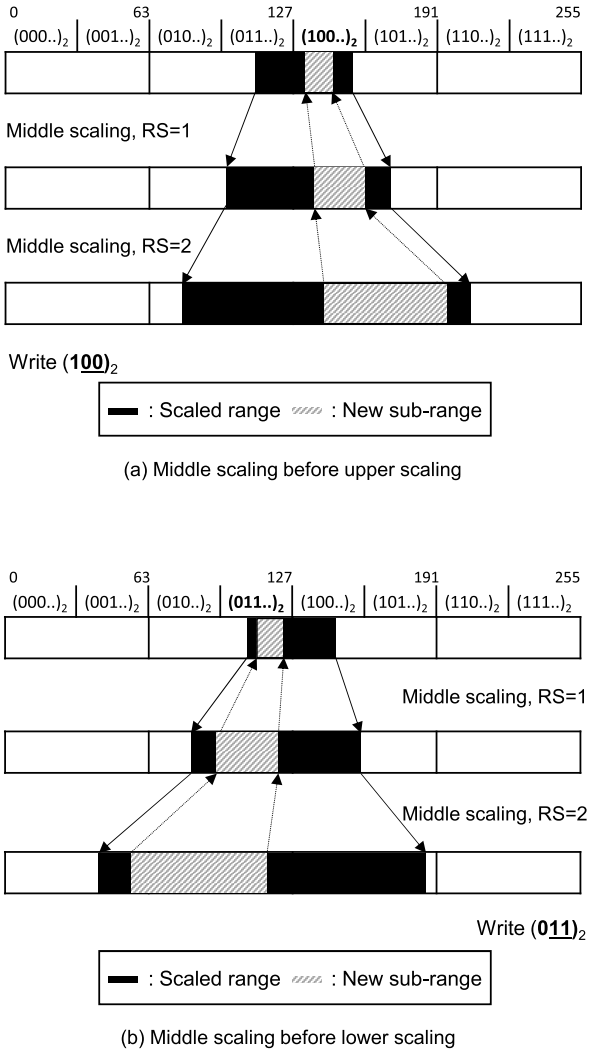


FIGURE 6. Range scalings in the case of middle scaling.

elements (W). We decode the bitstream by a unit of N-bits ($Z(idx)$). Here, the bit index of the bitstream begins with 0 ($idx = 0$). Thus, the initial N-bit window will be $Z(0)$. Before we perform the iterations, we need to append the $N - 1$ bits of 0_2s at the end of BS in order to enable the decoding of the last $N - 1$ bits in the BS (line 1 in Figure 7). As in the encoding procedure, decoding is also performed by per weight element basis (from 0 to $K - 1$: line 2 in Figure 7). In lines 3-8, we calculate the sub-range ([low, $high$]) for each weight value (i.e., from 0 to 31) and find which weight value's sub-range contains $Z(idx)$. If the sub-range of weight value j contains $Z(idx)$, we write j to the decoded weight $W[i]$. Similar to the case of the encoding, we then consider the three different range scaling cases: upper, lower, and middle scaling. In the cases of upper and lower scaling (lines 10-13 in Figure 7), we also scale the sub-range ([low, $high$]) by following the corresponding scaling rule as in Figure 5 (a) and (b), respectively, while also updating $Z(idx)$. Please note that the $Z(idx)$ can be regarded as a pointer

Pseudocode for Decoding (Decompression)

```
   INPUT :  BS – Encoded weight bitstream
            PROB – Probability of each weight value occurrence in W
            K – # of original weight element
   OUTPUT : W – Original 5-bit quantized weight elements

   N = # of bits for mapping the weight data to an unsigned integer number
   MAX = 2^N-1, HALF = round(MAX/2), QTR = round(MAX/4)
   low = 0, high = MAX, idx=0
   F[X] = PROB[x < X] (X=0 to 32, F[0]=0, F[1]=PROB[0], ... F[32]=1)
   Z(idx) = N-bits starting from bit index 'idx' in BS

1  Append N-1 '0's at the end of BS
2  for i=0 to K–1
3     range=high–low
4     for j=0 to 31
5        high = low + floor(range * F[j+1])
6        low = low + floor(range * F[j])
7        if low <= Z(idx) and Z(idx) < high
8           store weight value j to W[i], break    endif
9     endfor
10    while high < HALF or low >= HALF
11       if low >= HALF
12          low = low – HALF, high = high – HALF, Z(idx) = Z(idx) – HALF   endif
13       low = low << 1, high = high << 1, idx++   endwhile
14    while low >= QTR and high < 3 * QTR
15       low = (low – QTR) << 1, high = (high – QTR) << 1
16       Z(idx) = Z(idx) – QTR, idx++   endwhile
17 endfor
```

FIGURE 7. Pseudocode of our weight decoding technique.

i	Starting range	W[i]	Sub-range	Scaling case	BS	Updated RS
0	[0, 255]	0	[0, 102]	Lower scaling	0_2	0
1	[0, 204]	1	[81, 163]	Middle scaling	0_2	1
2	[34, 198]	0	[34, 99]	Lower scaling	$(001)_2$	0
3	[68, 198]	1	[120, 172]	Middle scaling	$(001)_2$	1
4	[112, 216]	2	[195, 216]	Upper scaling	$(00110)_2$	0
4	[134, 176]	-	-	Upper scaling	$(001101)_2$	0
4	[12, 96]	-	-	Lower scaling	$(0011010)_2$	0
out of the loop	[24, 192]	-	-	-	$(001101001)_2$	1

FIGURE 8. An example for encoding by using our proposed technique.

that indicates the N-bits in the BS with a starting index of idx. Thus, updating the $Z(idx)$ also means updating N-bits in the BS starting at the bit-index idx, also affecting the following $Z(idx + 1)$ value which will be referenced next (i.e., referenced by shifting the sliding window). In the middle scaling case (lines 14-16 in Figure 7), we scale the sub-range as in Figure 5 (c) while also updating $Z(idx)$. As in the encoding procedure, for three scaling cases, the scaling procedure is repeatedly performed (in a *while* loop) as long as the scaling condition is satisfied. Once K weight elements are all decoded, the **for** loop (lines 2-17 in Figure 7) terminates and the decoding process is completed.

4) ENCODING AND DECODING EXAMPLES

Figure 8 shows an illustrative example of our arithmetic coding-based weight encoding. For brevity, we limit the types of weight values (i.e., possible weight element values) as "0", "1", and "2" (originally, possible weight element values are 0∼31: 32 different weight values) and assume that we encode the sequence of the weight elements {0, 1, 0, 1, 2} (weight elements flattened to a form of a 1d-vector[4]). From the sequence, we can recognize that the $F[x]$ values for a range [0, 3] ({$F[0], F[1], F[2], F[3]$}) will be {0.0, 0.4, 0.8, 1.0}. In this example, we assume that $N = 8$, meaning that MAX, $HALF$, and QTR are 255, 128, and 64, respectively.

At first, we encode the first weight element "0". By calculating the sub-range ([low, $high$]) of the weight element "0", a new sub-range will be [0, 102] (=[0+⌊255*0.0⌋, 0+⌊255*0.4⌋]) which corresponds to the case of lower scaling ($high < HALF$). In this case, we write a bit 0_2 in the output bitstream (RS is currently 0, thus we do not write 1_2). We then scale the lower bound and upper bound of the sub-range by 2× by following the lower scaling rule. A new scaled range will be [0, 204] which does not account for any of the three scaling cases. Thus, we move to the next iteration for encoding the second weight element "1". The next sub-range will be [81, 163] (=[0+⌊204*0.4⌋, 0+⌊204*0.8⌋]), which corresponds to the case of the middle scaling. In this case, we increment RS by 1 and scale the range by following the middle scaling rule, which makes the current range [34, 198] (=[(81-64)*2, (163-64)*2]). This range does not correspond to the middle scaling case, which means that we need to encode the following element. When we encode the third element "0", a new sub-range will be [34+⌊164*0.0⌋, 34+⌊164*0.4⌋] = [34, 99], which is the lower scaling case. Thus, we write $(01)_2$ (because of $RS = 1$) to the bitstream and reset RS to 0 while we also scale the lower and upper bounds of the sub-range by 2×, resulting in the range of [68, 198] which does not correspond to any of the three cases. For the fourth weight element "1", a new sub-range will be [68+⌊130*0.4⌋, 68+⌊130*0.8⌋] = [120, 172], which corresponds to the middle scaling case. Thus, we increment the RS by 1 and scale the sub-range by following the middle scaling rule ([(120-64)*2, (172-64)*2] = [112, 216]). The scaled range [112, 216] does not correspond to any of the three cases, we perform the next iteration. For the next weight element "2", a new sub-range will be [112+⌊104*0.8⌋, 112+⌊104*1.0⌋] = [195, 216], which corresponds to the upper scaling case. Since the current RS value is 1, we write $(10)_2$ to the bitstream and reset RS to 0. After the upper scaling, we obtain [(195-128)*2, (216-128)*2] = [134, 176] as a new scaled range, which is still upper scaling case. Thus, we write 1_2 to the bitstream and scale the range by following the upper scaling rule, which results in [(134-128)*2, (176-128)*2] = [12, 96]. It accounts for the lower scaling case, resulting in writing 0_2 to the BS. By following the lower scaling rule, we obtain a new scaled range of [12*2, 96*2] = [24, 192], which does not correspond to any of the three cases, terminating the main loop (lines 1-13 in Figure 4). Since we have already encoded all the weight elements, we need to process the last part of the encoding (lines 14-18 in Figure 4). After we increment RS by 1, we write $(01)_2$ to the BS because the lower bound of the current range (low) is 24 which is less than QTR (64). Finally, we obtain the encoded bitstream of $(001101001)_2$.

Figure 9 demonstrates an example of the weight decoding. Please note that the N, MAX, $HALF$, and QTR values are

[4]For a sequence of weight elements, we flatten the weight elements starting from the first convolutional (CONV) layer to the last CONV layer of the CNNs.

i	Starting range	Sub-range			Z(idx) – gray-shaded part	Output W	Scaling case	Updated idx
		j = 0	j = 1	j = 2				
0	[0, 255]	[0, 102]	-	-	$(00110100\,10000000)_2$ 52	{0}	Lower scaling	1
1	[0, 204]	[0, 81]	[81, 163]	-	$(01101001\,10000000)_2$ 105	{0,1}	Middle scaling	2
2	[34, 198]	[34, 99]	-	-	$(00101001\,00000000)_2$ 82	{0,1,0}	Lower scaling	3
3	[68, 198]	[68, 120]	[120, 172]	-	$(00010100\,10000000)_2$ 164	{0,1,0,1}	Middle scaling	4
4	[112, 216]	[112, 153]	[153, 195]	[195, 216]	$(00001100\,10000000)_2$ 200	{0,1,0,1,2}	-	-

FIGURE 9. An example for decoding by using our proposed technique.

equal to the encoding example. Before starting the first iteration, we append 7-bits of 0_2s at the end of *BS*. In this example, we will perform five iterations because K, the number of weight elements encoded in the *BS*, is 5. We first start with the 8-bit part $Z(0)$, $(00110100)_2$ (=52) in the first iteration. When we calculate the sub-ranges for each weight element, $Z(0)$ is within the sub-range of "0" ([0+⌊255*0.0⌋, 0+⌊255*0.4⌋] = [0, 102]). Thus, we write element "0" to the weight data $W[0]$. Since the sub-range [0, 102] corresponds to the lower scaling case, we scale the lower and upper bounds of the sub-range by 2× (thus, a new scaled range will be [0, 204]) while increasing the *idx* by 1. Since the scaled range [0, 204] does not correspond to any of the three scaling cases, we move to the next iteration. In the second iteration, $Z(1)$ is equal to $(01101001)_2$ (=105), meaning that it corresponds to the sub-range of the element "1" (([0+⌊204*0.4⌋, 0+⌊204*0.8⌋] = [81, 163])) which will be written to the $W[1]$. A new sub-range will be [81, 163] (sub-range of $j = 1$), which corresponds to the middle scaling case. After scaling the range ([(81-64)*2, (163-64)*2] = [34, 198]) and updating $Z(1)$ (105-64(QTR) = 41 ($(00101001)_2$)), the new scaled range does not correspond to any of the three cases. Thus, we perform the next iteration with $Z(2)$ ($(01010010)_2$ = 82). Similarly, after calculating the new sub-range for each weight element, $Z(2)$ is within the sub-range ([34+⌊164*0.0⌋, 34+⌊164*0.4⌋] = [34, 99]) of the element "0", which will be written to the $W[2]$, and we need to scale the sub-range with the lower scaling rule, resulting in the new scaled range [68, 198] (=[34*2, 99*2]). In the next iteration, we have a starting range of [68, 198] while $Z(3)$ is $(10100100)_2$ (=164), which exists within the sub-range of the element "1" (164 is within the sub-range [68+⌊130*0.4⌋, 68+⌊130*0.8⌋] = [120, 172]). Thus, we write element "1" to the $W[3]$, and perform the middle scaling with the sub-range [120, 172] while updating $Z(3)$ to 100 (=164-64(QTR)). After that, we have a new scaled range [112, 216] (=[(120-64)*2, (172-64)*2]), which does not correspond to the middle scaling case. Thus, we move to the next iteration to seek for a sub-range of $Z(4)$, $(11001000)_2$ (=200). After calculating the sub-ranges for each weight element, $Z(4)$ is within the sub-range of "2" ([112+⌊104*0.8⌋, 112+⌊104*1.0⌋] = [195, 216]) which will be stored to the $W[4]$. Since we have already decoded five weight elements, we finish the decoding procedure with the decoded weight output $W = \{0, 1, 0, 1, 2\}$.

C. DECODING HARDWARE

Figure 10 shows a block diagram of our decoder hardware with a single decoding unit (DU). There are two buffers, bitstream buffer (for *BS*) and decoded weight buffer (for W), and one table (probability table for *PROB*). The bitstream buffer contains the encoded weight bitstream which is delivered from the main memory (or storage). The probability table maintains $F[x]$ for each weight value ($F[0]$ and $F[32]$ are always 0 and 1, respectively, meaning that they do not need to be stored in the table). The decoded weight buffer will store the decoded weight elements. From the bitstream buffer, we send the $Z(idx)$ (where *idx* is the starting bit index of the N-bits) to the range scaling unit which performs the range scaling according to the three cases (upper, lower, and middle scaling). After that, the scaled ranges from the range scaling unit and probability values ($F[0] \sim F[32]$ in Figure 7) are sent to the range calculation unit, which performs calculations of a new sub-range for each weight value. In our design, we perform 32 sub-range calculation (corresponds to the lines 4-6 in Figure 7) in parallel, improving the performance of our decoder. In the comparator, the $Z(idx)$ value and each sub-range are also compared in parallel in order to figure out which weight value should be written to the output in the current iteration. Once the comparison is done, we store the corresponding element to the decoded weight buffer (lines 7-8 in Figure 7). This process is iteratively performed according to the orchestration of the control logic.

For proof-of-concept of our decoding hardware,[5] we have implemented our hardware in a field programmable gate array (FPGA) board (Xilinx ZCU106). We have used Xilinx Vivado design suite to implement our decoder design. We have synthesized our design for 150MHz with 16 DUs (the maximum number of DUs which our FPGA chip can accommodate), which results in 16× higher throughput than the 1-DU decoder implementation. To utilize 16 DUs, we divide the weight elements into 16 chunks as evenly as possible and encode each chunk into a separate bitstream. When decoding the bitstreams, we send each bitstream to each of the 16 DUs. Although our prototype is implemented with 16 DUs, the number of DUs is one of the design parameters which can be flexibly determined by the designer considering the available hardware resources (we will demonstrate the latency versus resource usage tradeoff in Section V-C). If we implement our decoding hardware with application-specific integrated circuits (ASICs), we could implement more number of DUs, resulting in a better decoding throughput. According to our implementation, our single DU hardware takes 6.45 clock cycles per one weight element decoding, on average, though the total number of decoding cycles depends on the pattern and sequence of the weight elements in the bitstream.

[5]Hardware resources that we have used for our 16-DU decoder in FPGA are as follows: 141,568 LUTs; 49,379 flip-flops; 1,152 DSP48E blocks; and 512 BRAM blocks. If we share the decoded weight buffer with the CNN accelerator, we could further reduce the resource usage though we design a separate decoded weight buffer in our implementation.

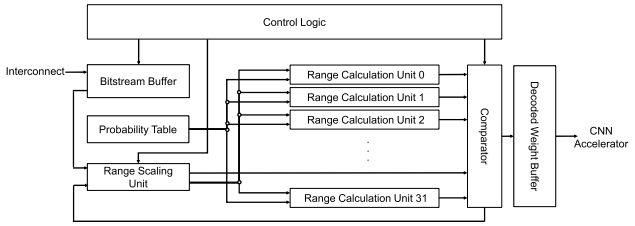


FIGURE 10. Decoding unit (DU) architecture.

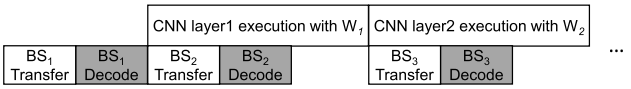


FIGURE 11. Latency hiding for our decoding hardware.

When executing the CNN inference, the weight decoding must be performed prior to the convolution operations. Without hiding the decoding latency, the latency overhead would be non-negligible, which is not desirable. To minimize the decoding latency, we can overlap the transfer and decoding latency for weights in the i-th CNN layer with the $(i-1)$-th CNN layer execution latency in the CNN accelerator (illustrated in Figure 11). In this case, the latency overhead will be only decoding latency of the weights for the first CNN layer (where the weight decoding latency cannot be overlapped) as long as the following two conditions are satisfied. First, the input and output feature maps are reused across the CNN layers in the CNN accelerator (hence, the input and output feature maps do not need to be transferred between the memory and the PLM of the accelerator). Second, data transfer and decoding latency of CNN layer i are fully hidden by the execution time of CNN layer $i-1$ where integer $i > 1$. To satisfy the second condition, more DUs in the decoding hardware can be desirable. This is because the decoding latency can be further reduced as we have more DUs in the decoder.

V. EVALUATION RESULTS

We show our evaluation results in terms of three metrics: compression ratio, energy consumption in the main memory, and latency overhead. For benchmarks, we use five CNN models: Network-in-Network (NiN) [23], SqueezeNet [24], GoogleNet [25], AlexNet [26], and CaffeNet [27]. We use 32 for N in our evaluations.

We have used the trained 32-bit floating-point (FP32) weights provided by Caffe framework [27]. For 5-bit quantization, we have employed an incremental network quantization (INQ) [6] method to generate 5-bit power-of-two (i.e., utilizing binary logarithm or logarithm to the base 2) quantized weights from the 32-bit full-precision weights. The main reason we choose INQ for our baseline is that it shows comparable or even better accuracy with 5-bit quantized weights as compared to 32-bit full-precision weights when running CNN inferences. Though we use a specific method (INQ) for 5-bit power-of-two quantization, our technique can be deployed with any 5-bit quantization method. For AlexNet and CaffeNet, we have additionally employed weight pruning (we used [28] and [29] for weight pruning of AlexNet and CaffeNet, respectively) to figure out how weight pruning affects the compression ratio. Although the top-1/top-5 accuracies show a little fluctuation by applying the weight pruning,[6] our compression technique itself does not adversely affect the inference accuracy as our technique is based on lossless compression. In other words, accuracy losses are not attributed to our compression technique, but attributed to the quantization and/or weight pruning. For AlexNet and CaffeNet, we demonstrate the results for two separate cases: (1) only 5-bit quantization (denoted as Q) and (2) pruning and 5-bit quantization (denoted as P+Q).

TABLE 1. Compression ratio (for only CONV layers) comparison across five CNN models.

CNN models	32-bit	16-bit	8-bit	5-bit	HC-5bit	**AC-5bit**	IE-5bit
NiN (Q)	1.000	2.000	4.000	6.400	9.335	**9.744**	9.747
SqueezeNet (Q)					8.817	**9.118**	9.157
GoogleNet (Q)					8.801	**9.434**	9.454
AlexNet (Q)					9.652	**9.829**	9.835
CaffeNet (Q)					9.524	**9.672**	9.678
Average (Q)					9.226	**9.560**	9.574
AlexNet (P+Q)					26.675	**57.498**	57.536
CaffeNet (P+Q)					28.806	**112.154**	112.333
Average (P+Q)					27.744	**84.826**	84.934

Firstly, we present compression ratio (Section V-A), memory energy consumption (Section V-A), and latency overheads for CNN inference (Section V-B). We only compare the compression ratio and memory energy consumption for CNN convolutional (CONV) layers while excluding the fully connected (FC) layers. For latency overheads, we assume that we only compress CONV layers' weights while the weights of the other layers such as FC layers are not compressed. In this case, we do not have the latency overheads of the layers other than the CONV layers because it does not need to be decompressed. Although early CNN models have a large number of weights in FC layers (e.g., AlexNet [26]), recent CNN models have FC layers only in the last layer of the CNN model (e.g., ResNet [1]), meaning that the CNN layer architecture is mostly composed of CONV layers. For deployment of our proposed technique to highly resource-constrained systems, we also demonstrate a trade-off between the CNN inference latency and hardware resource usage (Section V-C). In addition, we further show a system-level energy consumption for a CNN inference in highly resource-constrained systems (Section V-D).

[6]Weight pruning results in top-1/top-5 = –0.0%/+0.4% and top-1/top-5 = –1.5%/–1.4% for AlexNet and CaffeNet, respectively. The accuracy results are obtained by using Caffe framework [27] with ImageNet dataset [30]. The weight dataset used for evaluations is available at: https://github.com/tig0422/Arithmetic_Coding.

TABLE 2. AlexNet (full layers including CONV and FC layers) compression ratio comparison of our proposed technique with the state-of-the-art methods.

Method	Compression ratio
[4]	34.8
[12]	40.7
[14]	21.9
[16] (w/o bzip2)	57.5 (37.9)
Ours (AC-5 bit)	**43.3**

TABLE 3. Memory energy consumption (μJ) when transferring the weight data to the CNN accelerator or NPU on-chip memory.

CNN model	32-bit	16-bit	8-bit	5-bit	HC-5bit	**AC-5bit**	IE-5bit
NiN (Q)	17532.72	8766.36	4383.18	2924.72	1907.95	**1814.19**	1798.76
SqueezeNet (Q)	2874.67	1437.34	718.67	503.87	353.31	**341.66**	313.94
GoogleNet (Q)	14089.60	7044.80	3522.40	2398.83	1626.18	**1536.88**	1490.30
AlexNet (Q)	5388.55	2694.27	1347.14	902.44	567.10	**556.90**	547.88
CaffeNet (Q)					574.74	**565.21**	556.77
AlexNet (P+Q)					201.99	**93.71**	93.65
CaffeNet (P+Q)					187.06	**48.05**	47.97

A. COMPRESSION RATIO AND MEMORY ENERGY CONSUMPTION

The *compression ratio* (CR) is defined as the ratio of uncompressed data size S_u to the compressed data size S_c, that is, $CR = S_u/S_c$. Table 1 summarizes the compression ratio across five CNN models. In Table 1, 32-bit, 16-bit, 8-bit, and 5-bit correspond to the cases of uncompressed 32-bit FP, 16-bit quantized, 8-bit quantized, and 5-bit quantized weights, respectively. HC-5bit and AC-5bit (our proposed technique) denote the cases of 5-bit quantized weight compressed with Huffman coding and arithmetic coding, respectively. IE-5bit denotes the theoretical bound of compression ratio (information entropy) when losslessly compressing the 5-bit quantized weight data.

In the case of only quantization (Q), our AC-5bit reduces the weight data size by 9.6× as compared to the uncompressed 32-bit weight data. In addition, our technique reduces the weight data size by 29.8%–34.9% as compared to the uncompressed 5-bit quantized weight data size. It means our proposed technique can significantly reduce the required storage and memory. In addition, our arithmetic coding-based compression technique shows better compression ratio as compared to the Huffman coding-based compression technique. On average, when compressing the 5-bit quantized weight data, our technique results in 3.7% (up to 7.2%) better compression ratio compared to the Huffman coding-based technique (HC-5bit). Moreover, our arithmetic coding-based compression technique shows near-optimal compression ratios. As compared to IE-5bit, our AC-5bit shows only 0.1% less compression ratio, on average.

In the case of pruning and quantization (P+Q), our AC-5-bit obtains 57.5× and 112.2× higher compression ratio over 32-bit FP for AlexNet and CaffeNet, respectively. The pruning significantly increases the number of zero-valued elements in the weights, which also significantly contributes to the compression ratio. On the other hand, HC-5-bit obtains 26.7× and 28.8× (as compared to the 32-bit FP) better compression ratios for AlexNet and CaffeNet, respectively. However, our AC-5bit results in 2.2× and 3.9× better compression ratios for AlexNet and CaffeNet, respectively, as compared to HC-5bit. As shown in the results (Table 1), our technique shows more promising results when applying the weight pruning which is a widely used technique for CNN inferences.

We also compare our work with other state-of-the-art techniques in terms of the compression ratio when using AlexNet [26]. Though the comparison in Table 1 is based on the compression of the weight in the CONV layers, for a fair comparison, we have compared our technique to other techniques based on the compression of the weights in the entire layers including the fully-connected (FC) layers. Before applying our compression technique, the entire CONV and FC layers are pruned by [28] and quantized to 5-bit format by [6]. As shown in Table 2, our technique shows better compression ratio as compared to the techniques or methods presented in [4], [12], and [14]. As compared to the method presented in [16], however, our technique shows a little lower compression ratio by 24.7%. In [16], the bzip2 compression is additionally applied to pruned and quantized weights. In this case, there would be a non-negligible latency overhead for decompressing the compressed weights during the runtime CNN inference. Please note that without bzip2 compression in [16], the compression ratio of our technique is better than that of [16] by 14.2%. Since our work has also proposed a hardware decoder and pipelining which minimizes latency overhead of runtime weight decompression, our technique would be more practical and suitable for real-world deployment.

Higher weight compression ratio translates into less memory energy consumption when transferring the weight data to the PLM of the CNN accelerator or the NPU. Table 3 presents memory data transfer energy comparison results when the device uses LPDDR4 (5pJ per bit [31] and 0.43W static power [32]) dynamic random access memory. We have estimated the dynamic energy by multiplying the accessed data size (in bits) by the per-bit access energy and static energy by multiplying the static power by the memory transfer time. The total estimated energy consumption is a summation of the dynamic and static energy. When only employing the 5-bit quantization (Q), our AC-5bit reduces the memory data transfer energy by 89.2% as compared to the case of using uncompressed 32-bit FP weight data. Our AC-5bit also leads to 36.4% and 3.4% less memory energy consumption for weight transfer as compared to the cases of uncompressed 5-bit weight and HC-5bit, respectively. When employing both 5-bit quantization and pruning (P+Q), our AC-5bit results in 98.3% and 99.1% less memory energy consumption for AlexNet and CaffeNet, respectively, as compared to the case of 32-bit FP weights. Even compared to HC-5bit, our AC-5bit

TABLE 4. Inference latency (in milliseconds) and overhead comparison across the baseline (w/o our technique) and our technique (16-DU decoder) w/ latency hiding (LH) and w/o LH when employing various state-of-the-art CNN accelerators [9]–[11]. When estimating the decoding latency, we assume that our decoder operates at the same clock frequency as the CNN accelerator.

Combined CNN Accelerator	[9]	[10]	[11]	
CNN Model	AlexNet	AlexNet	AlexNet	GoogleNet
Platform	Stratix-V GXA7	ZC 706	Xilinx ZU9 MPSoC	
Clock(MHz)	155	156.25	100	
Precision(bits)	5-bits	5-bits	8-bits	
Peak Throughput (GOP/s)	206.87	155.10	14.97	15.63
Decode + Transfer Latency	3.79	6.70	16.06	42.62
Decode + Transfer Latency (pruned)	3.45	3.42	10.94	N/A
Baseline Inference Latency	8.83	52.80	90.01	202.02
Inference Latency of our technique w/ LH (overhead)	**9.31 (5.48%)**	**52.886 (0.16%)**	**91.16 (1.27%)**	**204.85 (1.40%)**
Inference Latency of our technique w/ LH (overhead) w/ pruned weights	**8.91 (0.91%)**	**52.884 (0.16%)**	**90.15 (0.16%)**	N/A
Inference Latency of our technique w/o LH (overhead)	12.62 (42.94%)	59.50 (13.26%)	106.07 (17.84%)	244.64 (21.10%)
Inference Latency of our technique w/o LH (overhead) w/ pruned weights	12.28 (39.07%)	56.24 (6.48%)	100.95 (12.15%)	N/A

reduces memory energy consumption by 53.6% and 74.3% for AlexNet and CaffeNet, respectively. Considering that the energy consumption in memory-related parts accounts for a large portion in resource-constrained edge systems [33], our proposed technique will enable more energy-efficient resource-constrained edge devices.

B. LATENCY OVERHEAD

Our technique incurs decoding latency before the weight data is loaded to the accelerator on-chip memory. As explained in Section IV-C, the decoding latency can be hidden by overlapping bitstream decoding in the decoding hardware with the CNN layer execution in the accelerator or the NPU. Even with the latency hiding, the weight decoding latency for the first layer in the CNN model cannot be hidden, thus incurring the latency overhead. Table 4 summarizes the latency overhead results when using our decoding hardware combined with various state-of-the-art CNN accelerators [9]–[11] ('Combined CNN Accelerator' in Table 4). We present the latency overhead results when our proposed decoding hardware (Figure 10) is combined with the FPGA-based CNN accelerators because our decoding hardware prototype is implemented and verified in an FPGA.

There is also a CNN accelerator [11] with 8-bit precision in Table 4 while our main target for weight compression is 5-bit quantized weights. Though we have presented our technique for 5-bit weights, our arithmetic coding-based encoding and decoding technique can also be used along with 8-bit precision-based CNN accelerators. In this case, we only compress 5-bits within each 8-bit weight element by using our arithmetic coding while the remaining bits (i.e., 3-bits) of the weights remain uncompressed (the remaining 3-bits can be transferred directly from the memory to the CNN accelerator without passing through the hardware decoder). In this case, we will have a lower compression ratio, and a higher weight decoding and transfer latency as compared to the case of 5-bit weight compression due to the uncompressed part in each weight element. Nonetheless, to present the versatility of our technique, we also present the latency overhead results when adopting our technique with the 8-bit precision accelerator [11] in Table 4.

When using our proposed decoding hardware and various CNN accelerators without the latency hiding, the latency overhead is 13.26%–42.94%. On the other hand, with the latency hiding,[7] our proposed technique without pruning shows 0.16%–5.48% latency overheads when performing the CNN inferences, implying that the latency overhead from the decoding hardware is small. In the case of our proposed technique with pruning and latency hiding, the latency overheads are almost negligible (0.16%–0.91%). When focusing on the case with 8-bit precision accelerator [11] in Table 4, the latency overhead is only up to 1.40%, implying that our technique can also be deployed with 8-bit precision accelerator with a negligible latency overhead.

The huge latency overhead reduction when applying the pruning is attributed to the reduced weight data size with arithmetic coding (due to the increase in the number of zero-valued weight elements), resulting in quicker decoding and shorter weight transfer latency. Considering that the main focus of our technique is resource-constrained edge devices, this small latency overhead is sufficiently acceptable as the benefits from the reduced memory and storage requirement and reduced memory energy consumption are much greater than the latency overhead.

C. LATENCY VERSUS RESOURCE USAGE TRADE-OFF

For systems or devices under tight resource constraints, we also present the latency versus resource usage trade-off when employing 2-DU, 4-DU, 8-DU, and 16-DU decoders.[8] The 2-DU, 4-DU and 8-DU designs require much less hardware resources than the 16-DU design. Thus, the 2-DU, 4-DU, and 8-DU designs can be suitable for small or tiny embedded edge devices. However, the smaller number of DUs will lead to a higher decoding latency overhead, which also results in increased CNN inference latency. As shown in Figure 12, in the case of the 4-DU and 8-DU decoders, performance overheads without pruning can be up to 34.2% and 8.73%, respectively, whereas the performance overheads with pruning can be up to 31.4% and 2.77%, respectively, even with the latency hiding. With the 2-DU decoder, which can be deployed for the systems with extremely stringent resource constraints, the latency overhead can be up to 126.1% without

[7]The latency hiding does not affect the compression ratio, but reduces latency overhead incurred by the runtime decoding.

[8]We have additionally implemented and verified 8-DU decoder design in ZCU106. We have also implemented and verified 2-DU and 4-DU decoder designs in Ultra96 platform, which is used in highly resource-constrained edge devices and/or embedded systems.

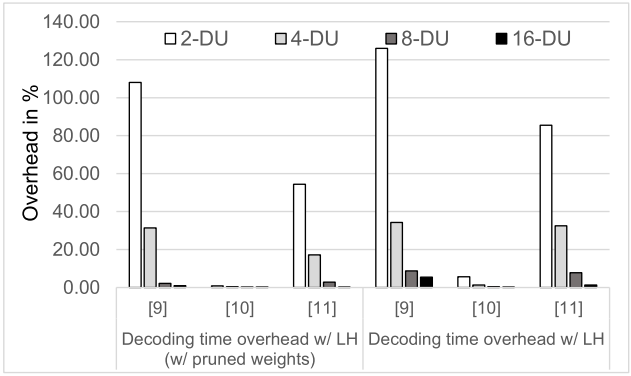


FIGURE 12. A comparison of decoding time overheads across the number of DUs in the decoder (2-DU, 4-DU, 8-DU, and 16-DU) when running AlexNet. Please note that LH stands for latency hiding.

TABLE 5. Inference latencies (in ms) across the state-of-the-art CNN accelerators without our technique (baseline) and with our technique (2-DU, 4-DU, 8-DU, and 16-DU).

	Combined CNN accelerator		
	[9]	[10]	[11]
Baseline	8.83	52.80	90.01
2-DU	18.37	53.23	138.96
4-DU	11.60	53.02	105.46
8-DU	9.02	52.91	92.50
16-DU	8.91	52.88	90.15

pruning and 108.0% with pruning. The reason why the decoding time overhead seems to be large when used with the CNN accelerator in [9] is that the baseline CNN inference latency in [9] is very small, which makes the latency overhead from our decoder relatively large. For the decoding overhead with the CNN accelerator in [11], even though the baseline inference latency of [11] is higher than that of [10], the latency overhead is larger as compared to the case of [10]. This is because the CNN accelerator in [11] uses the 8-bit precision accelerator, which implies that the compression ratio will be worse in [11] as compared to 5-bit precision accelerators as our arithmetic coding technique is optimized for 5-bit weight encoding (i.e., 5-bits within 8-bits element are compressed while the remaining 3-bits remain uncompressed). This results in relatively high transfer latency when using the CNN accelerator in [11] with our decoder. In the cases of 8-DU and 16-DU with the CNN accelerator in [11], the transfer latency and decoding latency can be mostly hidden by the CNN layer processing time. However, in the cases of 2-DU and 4-DU with the CNN accelerator in [11], the transfer latency and decoding latency cannot be hidden by the CNN layer processing time, leading to the large latency overhead. For [9], though relative decoding time overhead can be large, the absolute inference latency is negligibly affected (+9.54ms and +2.77ms with 2-DU and 4-DU decoders, respectively) as shown in Table 5. For [11], the decoding time overheads with 2-DU and 4-DU decoders can be decreased if we use 5-bit precision CNN accelerators.

TABLE 6. System-level energy comparison between the baseline (Q) and our technique (P+Q) with 4-DU decoder.

Combined CNN accelerator		[9]	[10]
FPGA platform		Stratix-GXA7	ZC706
Baseline (Q)	Latency (ms)	8.83	52.80
	FPGA Power (W)	8.69	12.02
	FPGA Energy (mJ)	76.73	634.66
	Total Energy (mJ)	130.27	707.10
Our technique (P+Q) with 4-DU decoder	Latency (ms)	11.60	53.23
	FPGA Power (W)	9.28	12.61
	FPGA Energy (mJ)	107.59	670.96
	Total Energy (mJ)	118.11	699.39

In typical edge devices, the baseline CNN inference latency will not be very small. This is because the CNN accelerator performance will be limited due to the tight hardware resource constraints. In addition, satisfying the deadline of the response time (i.e., latency) is more important in edge or embedded systems, which implies that the increased latency overhead is acceptable as long as it does not violate the response time deadline. Thus, the edge system designers can choose the appropriate number of DUs by considering the performance requirements and resource constraints of the system under design.

D. SYSTEM-LEVEL ENERGY ESTIMATION

We compare the system-level energy when using our arithmetic coding-based compression for the 5-bit quantized and pruned weights with 4-DU decoder and baseline (i.e., without our compression and decoder while only 5-bit quantization is employed since the combined CNN accelerators supports 5-bit precision arithmetic operations.). The system-level energy includes the CNN accelerator (with the 4-DU decoder in the case with our proposed technique) energy, DRAM-based main memory energy, and NVMe (Non-Volatile Memory Express) flash-based storage energy. Since the non-volatile flash storage will be accessed to load the weights into the main memory before CNN inferences, we have included the flash-based storage energy to our system-level energy estimation. Please note that we use the flash energy parameter reported in [34] (1J / 28MB = 4.26nJ per bit). Since CNN accelerator power is reported in [9] and [10], while it is not reported in [11], we only include the results with [9] and [10] for our system-level energy estimation. The reason why we choose the 4-DU decoder among the various configurations is that the 4-DU decoder can be accommodated in an edge/embedded platform (such as Ultra96) and shows a good tradeoff between the inference latency and resource usage.

As shown in Table 6, for the combined accelerators with our proposed technique, power consumption and inference latency are increased, which results in an increased energy consumption in the FPGA by 40.2% and 5.7% with the CNN accelerators in [9] and [10], respectively. However, when

considering the system-level energy consumption which includes the DRAM memory and flash-based storage energy consumption, our proposed technique with 4-DU configuration results in the system-level energy reduction by 9.3% and 1.1% with the CNN accelerators in [9] and [10], respectively. This is attributed to the reduced weight data size from our arithmetic coding-based weight compression.

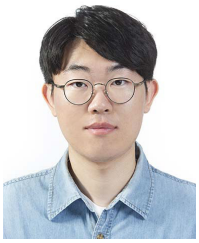
VI. CONCLUSION

In resource-constrained edge devices, one of the most serious challenges for deploying on-device CNN inferences is huge weight data size which can hardly be fully stored in an edge device. In this paper, we have proposed an arithmetic coding-based 5-bit quantized weight compression technique with range scaling for lossless 5-bit weight compression. In addition, we have also proposed a decoding hardware for fast, yet efficient runtime weight decoding (decompression). Our evaluation results reveal that employing our weight compression technique to 5-bit quantized weights (not pruned) achieves 9.6× better compression ratio as compared to the uncompressed 32-bit FP weights. When employing our technique to the pruned 5-bit quantized weights, our technique results in 57.5×–112.2× better compression ratio as compared to the uncompressed 32-bit FP weights. Due to the reduced weight data size, our technique also leads to memory data transfer energy reduction by 89.2% (by up to 99.1% for pruned weights), on average, as compared to the uncompressed 32-bit FP weight data. When combining our decoding hardware with various state-of-the-art CNN accelerators, the latency overheads of our proposed technique with 16-DU decoder along with the latency hiding are only 0.16%–5.48% and 0.16%–0.91% for non-pruned and pruned weights, respectively. In addition, our proposed technique with 4-DU decoder hardware reduces system-level energy consumption by 1.1%–9.3% as compared to the case without our proposed technique. As our future work, we plan to generalize our arithmetic coding-based compression/decompression technique for weight elements with arbitrary bit-widths.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Represent., (ICLR)*, 2015, pp. 1–14.
[3] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.
[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *Proc. 4th Int. Conf. Learn. Represent., (ICLR)*, 2016, pp. 1–14.
[5] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016, *arXiv:1603.01025*.
[6] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless CNNs with low-precision weights," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017, pp. 1–14.
[7] S. M. Darwish and Z. H. Noori, "Secure image compression approach based on fusion of 3D chaotic maps and arithmetic coding," *IET Signal Process.*, vol. 13, no. 3, pp. 286–295, May 2019.
[8] Z. Guo, J. Fu, R. Feng, and Z. Chen, "Accelerate neural image compression with channel-adaptive arithmetic coding," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2021, pp. 1–5.
[9] C. F. B. Fong, J. Mu, and W. Zhang, "A cost-effective CNN accelerator design with configurable PU on FPGA," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2019, pp. 31–36.
[10] M. Sit, R. Kazami, and H. Amano, "FPGA-based accelerator for losslessly quantized convolutional neural networks," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, Dec. 2017, pp. 295–298.
[11] R. Struharik, B. Vukobratovic, A. Erdeljan, and D. Rakanovic, "CoNNA—Compressed CNN hardware accelerator," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 365–372.
[12] Y. Choi, M. El-Khamy, and J. Lee, "Towards the limit of network quantization," 2016, *arXiv:1612.01543*.
[13] J. H. Ko, D. Kim, T. Na, J. Kung, and S. Mukhopadhyay, "Adaptive weight compression for memory-efficient neural networks," in *Proc. Design, Autom., Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 199–204.
[14] S. Ge, Z. Luo, S. Zhao, X. Jin, and X.-Y. Zhang, "Compressing deep neural networks for efficient visual inference," in *Proc. IEEE Int. Conf. Multimedia Expo. (ICME)*, Jul. 2017, pp. 667–672.
[15] B. Reagan, U. Gupta, B. Adolf, M. Mitzenmacher, A. Rush, G.-Y. Wei, and D. Brooks, "Weightless: Lossy weight encoding for deep neural network compression," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 4324–4333.
[16] Y. Choi, M. El-Khamy, and J. Lee, "Universal deep neural network compression," *IEEE J. Sel. Topics Signal Process.*, vol. 14, no. 4, pp. 715–726, May 2020.
[17] S. Young, Z. Wang, D. Taubman, and B. Girod, "Transform quantization for CNN compression," *IEEE Trans. Pattern Anal. Mach. Intell.*, early access, May 28, 2021, doi: 10.1109/TPAMI.2021.3084839.
[18] M. A. Qureshi and A. Munir, "NeuroMAX: A high throughput, multi-threaded, log-based accelerator for convolutional neural networks," 2020, *arXiv:2007.09578*.
[19] A. Shahbahrami, R. Bahrampour, M. S. Rostami, and M. A. Mobarhan, "Evaluation of Huffman and arithmetic algorithms for multimedia compression standards," 2011, *arXiv:1109.0216*.
[20] *Google Coral Dev Board*. Accessed: Nov. 9, 2020. [Online]. Available: https://coral.ai/products/dev-board/
[21] G. Yuan, X. Ma, C. Ding, S. Lin, T. Zhang, Z. S. Jalali, Y. Zhao, L. Jiang, S. Soundarajan, and Y. Wang, "An ultra-efficient memristor-based DNN framework with structured weight pruning and quantization using ADMM," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Design (ISLPED)*, Jul. 2019, pp. 1–6.
[22] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid, "Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.
[23] M. Lin, Q. Chen, and S. Yan, "Network in network," in *Proc. 2nd Int. Conf. Learn. Represent. (ICLR)*, 2014, pp. 1–10.
[24] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," 2016, *arXiv:1602.07360*.
[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.
[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NIPS*, vol. 1, 2012, pp. 1097–1105.
[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 2014, *arXiv:1408.5093*.
[28] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic DNN weight pruning framework using alternating direction method of multipliers," 2018, *arXiv:1804.03294*.
[29] T. Zhang, S. Ye, K. Zhang, X. Ma, N. Liu, L. Zhang, J. Tang, K. Ma, X. Lin, M. Fardad, and Y. Wang, "StructADMM: A systematic, high-efficiency framework of structured weight pruning for DNNs," 2018, *arXiv:1807.11091*.
[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[31] D. Stathis, C. Sudarshan, Y. Yang, M. Jung, S. Asad, M. H. Jafri, C. Weis, A. Hemani, A. Lansner, and N. Wehn, "EBrainII: A 3 kW realtime custom 3D DRAM integrated ASIC implementation of a biologically plausible model of a human scale cortex," 2019, *arXiv:1911.00889*.
[32] S. Lee, H. Cho, Y. H. Son, Y. Ro, N. S. Kim, and J. H. Ahn, "Leveraging power-performance relationship of energy-efficient modern DRAM devices," *IEEE Access*, vol. 6, pp. 31387–31398, 2018.
[33] K. Lee, J. Kong, Y. G. Kim, and S. W. Chung, "Memory streaming acceleration for embedded systems with CPU-accelerator cooperative data processing," *Microprocessors Microsyst.*, vol. 71, Nov. 2019, Art. no. 102897.
[34] B. Harris and N. Altiparmak, "Ultra-low latency SSDs' impact on overall energy efficiency," in *Proc. 12th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2020, pp. 1–25.



JONG HUN LEE (Student Member, IEEE) received the B.S. degree in electronics engineering from Kyungpook National University, in 2020, where he is currently pursuing the M.S. degree in electronics and electrical engineering. His research interests include convolutional neural network acceleration, data compression, and FPGA-based design.

JOONHO KONG (Member, IEEE) received the B.S. degree in computer science and the M.S. and Ph.D. degrees in computer science and engineering from Korea University, in 2007, 2009, and 2011, respectively. He worked as an Postdoctoral Research Associate with the Department of Electrical and Computer Engineering, Rice University, from 2012 to 2014. Before joining Kyungpook National University, he also worked as a Senior Engineer at Samsung Electronics, from 2014 to 2015. He is currently an Associate Professor with the School of Electronics Engineering, Kyungpook National University. His research interests include computer architecture, heterogeneous computing, embedded systems, and hardware/software co-design.

ARSLAN MUNIR (Senior Member, IEEE) received the M.A.Sc. degree in electrical and computer engineering from The University of British Columbia, Vancouver, Canada, in 2007, and the Ph.D. degree in electrical and computer engineering from the University of Florida, Gainesville, FL, USA, in 2012. From 2007 to 2008, he worked as a Software Development Engineer at the Embedded Systems Division, Mentor Graphics Corporation. He was a Postdoctoral Research Associate with the Department of Electrical and Computer Engineering, Rice University, Houston, TX, USA, from May 2012 to June 2014. He is currently an Associate Professor with the Department of Computer Science, Kansas State University. His current research interests include embedded and cyber-physical systems, secure and trustworthy systems, parallel computing, artificial intelligence, and computer vision. He received many academic awards, including the doctoral fellowship from the Natural Sciences and Engineering Research Council (NSERC) of Canada. He earned gold medals for best performance in electrical engineering, gold medals, and academic roll of honor for securing rank one in pre-engineering provincial examinations (out of approximately 300,000 candidates).