

Received 22 March 2023, accepted 21 April 2023, date of publication 1 May 2023, date of current version 4 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3271640

RESEARCH ARTICLE

Linearization Weight Compression and In-Situ Hardware-Based Decompression for Attention-Based Neural Machine Translation

MIJIN GO¹, (Student Member, IEEE), JOONHO KONG^{1,2}, (Member, IEEE),
AND ARSLAN MUNIR³, (Senior Member, IEEE)

¹School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, South Korea

²School of Electronics Engineering, Kyungpook National University, Daegu 41566, South Korea

³Department of Computer Science, Kansas State University, Manhattan, KS 66506, USA

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

This work was supported in part by the Samsung Electronics, and in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant NRF-2021R111A3A04037455.

ABSTRACT As recent machine translation models are mostly based on the attention-based neural machine translation (NMT), many well-known models such as Transformer or bidirectional encoder representations from Transformers (BERT) have been proposed. Along with algorithmic advancements, hardware acceleration methods for those attention-based neural machine translation models have also been introduced. However, the size of the parameters for attention-based NMT is also becoming larger to guarantee the satisfactory machine translation quality. Among various weights, linearization weights (W^Q , W^K , W^V , and W^O) account for a non-negligible portion (by up to 30%) among the entire parameters in the modern NMT models. In this paper, we propose a method for linearization weight compression and near-memory hardware decoder for fast and in-situ weight decompression. Our weight compression method exploits the fixed-point quantization along with Huffman coding which is selectively applied depending on the weight value distribution. Our hardware decoder decompresses the Huffman-coded weights near-memory to minimize the weight decoding latency. Our compression method shows 4.9–10.0 compression ratio with small NMT score drops across the five widely used attention-based NMT models (Transformer, Transformer-XL-base, Transformer-XL-large, BERT-base, and BERT-large). In addition, due to the reduced linearization weight size, our proposed method with near-memory decoding enables multi-head attention (MHA) execution latency reduction by 11.8%, on average, as compared to the baseline when considering the weight loading and initialization. In terms of the memory data transfer energy consumption, our proposed method leads to a memory energy saving of 16.1%, on average, as compared to the baseline.


INDEX TERMS Neural machine translation, multi-head attention, quantization, Huffman coding, hardware-based near-memory decoding.

I. INTRODUCTION

Recent trends in machine learning-based translation mostly focus on the neural machine translation (NMT). In the early stages of the NMT, recurrent neural networks (RNNs) or long short-term memories (LSTMs) have been typically used. However, the advent of attention-based models such as Transformer [1], bidirectional encoder representations from Trans-

formers (BERT) [2], etc., has opened a new era of the NMT because these attention-based models provide better accuracy for machine translations. Even further, recent advancement of the Transformer-based large language models (e.g., GPT-3 [3]) enables revolutionary artificial intelligence (AI)-based services such as AI-based chatbots (e.g., ChatGPT [4], LaMDA [5], etc.).

The attention-based models generally consist of the combination of multiple encoder and decoder layers. Inside of the encoder and decoder layers, multi-head attention (MHA)

The associate editor coordinating the review of this manuscript and approving it for publication was Bing Li .

is a key operation. In the MHA, linearization, which is performed to map tokens (outputs) into matrices before (after) the scaled dot-product attention, requires access to a huge amount of the weights (W^Q , W^K , W^V , and W^O). This high memory requirement for MHA linearization necessitates high capacity and bandwidth for both memory and storage. In this paper, we refer to these weight (W^Q , W^K , W^V , and W^O) as ‘linearization weights’ because these weights are used to project the input matrix into another matrix with linear transformation. It is also known that linearization weights account for 30.0% of the total attention-based weights in the case of BERT-large [2] model. Thus, reducing the linearization weight size can be beneficial for storage and memory capacity usage and energy efficiency. In particular, for enabling NMT inferences in resource-constrained systems, the weight size reduction is of paramount significance. As revealed in [6] and [7], the compression of the weights significantly reduces storage requirements and energy consumption for data transfer.

The recent efforts on attention-based NMT model acceleration or system design have mostly focused on the MHA acceleration. Since the MHA operations incorporate many complex operations, hardware acceleration of MHA could be beneficial for improving performance of the attention-based NMT models. Compared with the graphics processing unit (GPU) based execution, the customized hardware accelerators can significantly improve performance and energy efficiency [8], [9]. Apart from the hardware acceleration, weight quantization and data compression have also been explored [10], [11], [12], [13]. For instance, 3-bit quantization of the attention weights and its hardware-based acceleration have been introduced in [13]. While the most of the recent works have focused on the multi-head attention acceleration and data reduction, the linearization weights reduction has been largely overlooked.

In this paper, we focus on the linearization weights (i.e., W^Q , W^K , W^V , and W^O) used in the linearization step (i.e., linear projection of the input tokens and outputs: see Section II-A for details) before the MHA operations. We propose a linearization weight quantization and Huffman coding-based compression method. We employ fixed-point weight quantization for all linearization weights (i.e., for the entire encoder and decoder layers in the model). In addition, to achieve far better data compression ratio, we also employ Huffman coding for over 98% of the linearization weights, which are distributed within a certain range $[X, Y]$ where X and Y are determined by considering the weight distribution. By employing our weight compression method, we can reduce the linearization weight size by up to 10 \times as compared to the single precision floating point 32-bit (FP32) or tensor float 32-bit (TF32) element-based linearization weights. Although our method could also be employed for weights used in feed-forward networks in attention-based NMT models, too aggressive quantization might adversely affect the accuracy of the NMT model, which is another reason why we focus on the linearization weights in this work.

For fast in-situ weight decompression, we also introduce a near-memory hardware decompression accelerator, which enables seamless decoding of the Huffman-coded weights at runtime. We demonstrate the employment of our compression method and hardware decompression accelerator for high-bandwidth memory (HBM) enabled GPUs (e.g., [14]). After training the model, the linearization weights are quantized and compressed with our method and stored in the storage or memory. During runtime inference, the compressed weights are decompressed to the fixed-point weights in near-memory logic die (the 1st floor base logic die in the HBM) and stored in the HBM (as GPU memory) while also delivered to the GPU for the linearization operation. Experimental results reveal that our proposed method results in a compression ratio of 4.9–10.0 with only a marginal drop in NMT scores. Our proposed method also reduces memory data transfer energy consumption by 16.1%, on average, as compared to the case without our compression method (i.e., baseline). As our method employs a near-memory hardware decompression accelerator, the weight decoding latency is hidden by the latency required for other computations during the MHA operation. It means that the latency overhead (by up to 15.1%) for decompression can be minimized. Furthermore, our compression method results in the data transfer latency reduction when loading the weights from storage or host memory to the GPU memory. In this case, the MHA execution latency can be reduced by 11.8%, on average, as compared to the baseline.

We summarize our contributions as follows:

- We propose a quantization and selective Huffman coding-based weight compression method to reduce the size of the linearization weights, which results in a compression ratio of 4.9–10.0 with only marginal NMT score drops;
- For fast in-situ weight decoding at runtime, we also propose a near-memory processing hardware architecture;
- Our near-memory decoding enables memory energy savings by 16.1%, on average, as compared to the baseline (i.e., without using our proposed compression method);
- When considering the weight loading and initialization, our weight compression method with near-memory decoding enables MHA execution latency reduction by 11.8%, on average, as compared to the baseline.

The remainder of this paper is organized as follows. Section II presents background related to our work. Section III explains our proposed linearization weight compression and hardware decoder architecture. In Section IV, we present evaluation results. In Section V, we review the related works recently introduced and published. In Section VI, we conclude our work.

II. BACKGROUND

A. ATTENTION-BASED NEURAL MACHINE TRANSLATION

NMT is one of the most widely used artificial intelligence-based applications in modern data centers or servers.

The most well-known model for modern NMT is Transformer [1], which is basically composed of sequence-to-sequence (seq2seq) model. The seq2seq model is generally composed of multiple encoders and decoders where the numbers of encoders and decoders are tunable hyper parameters depending on the detailed model configuration. The internal architecture of the encoder and decoder is same though the encoders are executed first while the decoder layers generally accept the outputs from the encoder layers. For the internal architecture of the attention, the most widely used attention mechanism is MHA. The MHA accepts the query, key, and value as inputs. They are multiplied with the linearization weights (W^Q , W^K , and W^V) in the linearization step ('linear' in Fig. 1). This step performs a linear projection of the input tokens, translating the words into the query, key, and value matrices. The results from the linearization are fed into the scaled dot-product attention. The attention results are concatenated ('concat' in Fig. 1) and finally linearized again with the linearization weight W^O (projecting the concatenated output matrices into a format that can be fed into the following feed-forward network).

Modern attention-based NMT models have a huge size of the linearization weights. For example, in Transformer model [1], there are 18 MHAs (6 and 12 in the encoders and decoders, respectively) and there are 2^{20} weight elements for each linearization weight (W^Q , W^K , W^V , and W^O) in each of MHA. Since there are four types of the weights, $2^{20} \times 2^2 \times 18$ linearization weight elements exist. As a single element size is 4-byte (FP32), the total size of the linearization weights is 288MB, which is very huge and hard to be fit into the on-chip memory of the accelerators, GPUs, or central processing units (CPUs). It inevitably entails a large amount of the data transfer due to the limited on-chip data reuse. In addition, the linearization weights are very frequently used in the inference of attention-based NMT models as they are used in the linearization of the multi-head attention (MHA) in all the layers of the model.

B. DATA COMPRESSION FOR NMT MODELS

For lightweight NMT models, one of the most widely used methods is data compression. The data compression methods can be broadly classified into two types: lossy versus lossless compression. The lossy compression for NMT models is typically done by quantization which approximates high-precision data to low-precision data. The low-precision data has the less number of bits as compared to the high-precision data, leading to data losses. The data size and precision reduction can also result in better performance and energy efficiency as it simplifies the operation and reduces the amount of data transfer. However, since the data is lost with quantization, it is important to find the best trade-off between the data size and accuracy (i.e., NMT score).

Apart from the lossy compression, we can also employ lossless compression. One of the most well-known lossless compression methods is an entropy-based approach, for which Huffman coding and arithmetic coding are the

most prominent techniques. Huffman coding considers symbols' occurring probability in the data stream, generating a Huffman tree (or often stored in the form of the table) for codebook information during compression. For example, the higher occurring probability a certain symbol shows, the less bit sequence it requires, meaning that we can obtain better compression ratio. Since the symbol occurring probability will be different across the data, a unique Huffman tree is built for each data stream. When encoding the data, based on the Huffman tree, each symbol is encoded to a pre-defined bit sequence. Finally, the sequence of the symbols will be encoded as a form of the bitstream. When decompressing (decoding) the encoded bitstream, we also need Huffman tree which was built during the encoding. The bitstream is sequentially decoded by referring to the codebook defined by the Huffman tree. On the other hand, arithmetic coding maps the data into a real number space (0, 1]. Thus, a certain bitstream can be translated into one real number. However, when encoding the data into the bits, there could be an underflow, meaning that we cannot map a certain symbol into the number space due to the limited bit precision. Thus, we can utilize a range scaling method (e.g., [7]) to prevent the underflow.

C. BASELINE ARCHITECTURE

Modern NMT workloads are typically executed in GPU-based or machine learning (ML) accelerator-based systems. Since many ML accelerators only support reduced precision such as fixed-point or integer while many NMT models are still based on FP32 precision, the GPUs are primarily used for NMT acceleration in data centers or servers [15], [16]. For baseline system architecture, we assume a generic GPU-based system, which employ GPUs with GDDR-based off-chip memory (or HBM-enabled 3D-stacked memory [14]). For attention layer execution, the weights are loaded into the GDDR memory (or HBM-enabled 3D-stacked memory) and GPU processing cores (e.g., tensor cores) are used for computation.

III. LINEARIZATION WEIGHT COMPRESSION AND NEAR-MEMORY DECOMPRESSION

A. OVERALL EXECUTION FLOW

Fig. 2 illustrates the overall execution flow and system architecture of our method. In the offline phase, after the linearization weights (W^Q , W^K , W^V , and W^O) are generated through the model training, we apply our weight quantization and compression with Huffman coding to the linearization weights. We then store the compressed weight bitstream and metadata to the memory or storage. A Huffman table, which has also been generated during the Huffman coding-based compression, is stored as a form of the lookup table in our proposed hardware decoder for decompressing the bitstream. In the online phase (i.e., runtime NMT inference), our compression method requires decompression before the linearization because the compressed bitstream cannot be directly used in the linearization. Hence, the compressed weight bitstream and metadata are fed into our proposed

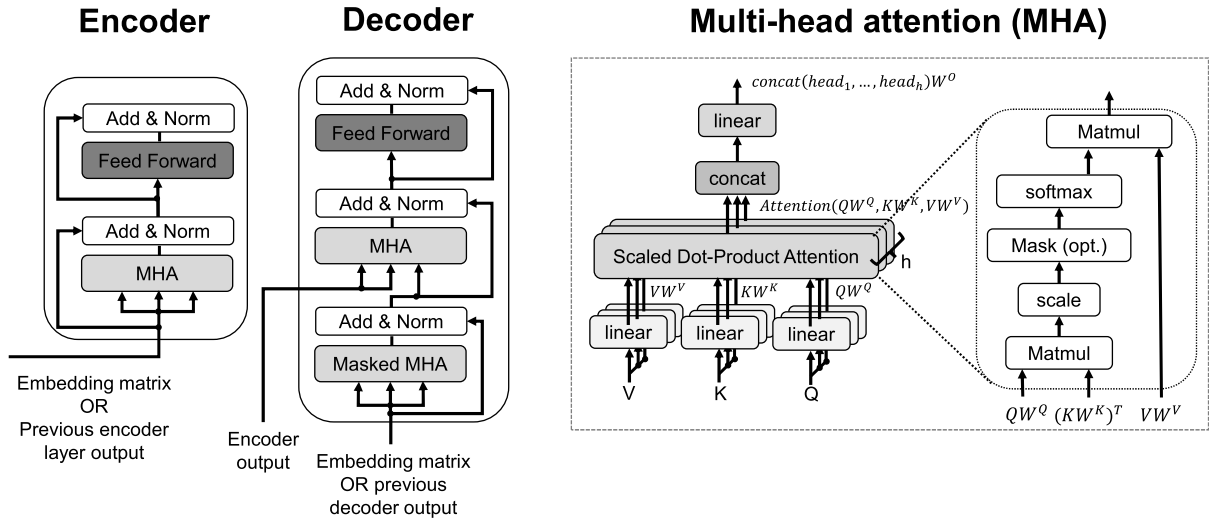


FIGURE 1. Encoder-decoder architecture of the attention-based neural machine translation.

hardware decoder that resides in the near-memory logic. The hardware decoder decompresses the quantized weights (i.e., recovers the weights to the status where only quantization is applied) from the compressed bitstream. The decompressed, yet quantized weights are delivered to the processing elements such as GPU cores for linearization.

Our system architecture is established as follows. We use the GPU system configuration as in [14], [17]. We use 2.5D architecture where HBM2 DRAM¹ and logic dies are stacked on the silicon interposer as shown in Fig. 3. Through the high-speed interconnection, the data is transmitted between HBM2 DRAM/logic and GPU. After decoding, the decoded weights (or a part of the decoded weights) are temporarily stored in the HBM2 DRAM. Since the HBM2 DRAM works as the GPU memory, the decoded weights are directly served from the GPU memory to the GPU core, which will be used for the inputs of the linearization.

B. LINEARIZATION WEIGHT QUANTIZATION AND COMPRESSION

Our proposed weight quantization and compression method is performed in two steps. Firstly, we quantize each FP32 weight element into a fixed-point element and selectively apply Huffman coding for most of the elements (i.e., elements which reside in the middle part of a Gaussian distribution representing the distribution of linearization weights). In the following subsections, we explain our quantization and selective Huffman coding-based compression method in details.

1) QUANTIZATION

For quantization of the linearization weights (W^Q , W^K , W^V , and W^O), we utilize the fixed-point quantization. We use the

¹High-bandwidth memory (HBM) 2 is a 3D-stacked DRAM technology. Since it provides much higher memory bandwidth as compared to the commodity DRAM modules, it is typically used for data-intensive workloads.

Q-format to represent fixed-point numbers where the notation of $Qm.n$ means that the fixed-point numbers in this format have m number of bits for the sign+integer part and n number of bits for the fractional part. As in typical quantization methods, we map each 32-bit floating point (FP32) weight element into the nearest fixed-point values. We use the fixed-point conversion method as shown in Fig. 4. We multiply 2^n to the FP32 value and apply the *round* operation to make it into a signed binary integer format with $m+n$ bits.² With the $m+n$ -bit binary, we set the integer and fractional parts with m bits in the upper part and n bits in the lower part, respectively.

Considering the distribution of the weight values in attention-based NMT models such as Transformer [1], we set the ranges of m and n as $1\sim3$ and $3\sim5$, respectively, and quantize them with certain m and n for each model (for detailed configuration, see Table 3). By using the fixed-point quantization, we can obtain at least $4\times$ theoretical compression ratio (i.e., $32\text{-bit}/8\text{-bit} = 4$ (in the case of $m=3, n=5$)).

Please note that we can also apply the quantization method to the tensor float 32-bit (TF32) format. Though the TF32 format has reduced bitwidth in the fractional part (from 23-bit to 10-bit), we can also approximate a TF32-formatted element into the $Qm.n$ format fixed-point value by using the method shown in Fig. 4.

2) SELECTIVE HUFFMAN CODING

Along with quantization, we additionally apply Huffman coding for elements whose values are distributed within a certain range. By applying Huffman coding, we can further achieve even higher compression ratio for linearization weights. As shown in Fig. 5, the weights typically show

²In case we cannot fit the value into the range (i.e., out of range or overflow) with $m+n$ bits, we cannot quantize this FP32 value. In this case, we need to set higher m and n values for fixed-point quantization.

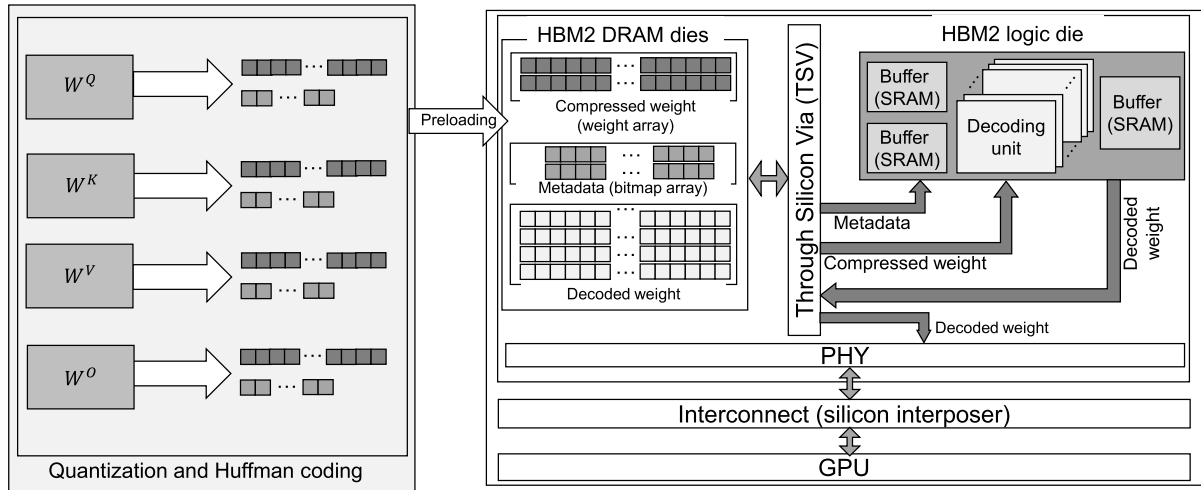


FIGURE 2. System architecture and overall flow of our method.

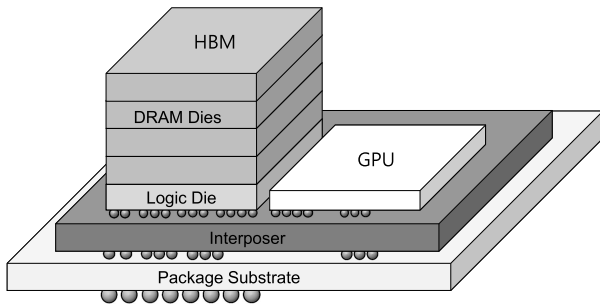


FIGURE 3. The HBM-GPU integrated architecture with 2.5D integration.

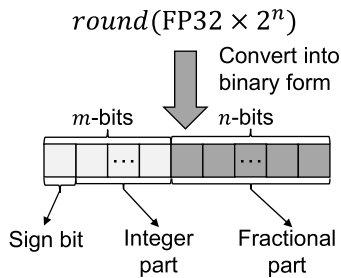


FIGURE 4. The illustration of the quantization method used in this paper. $FP32$ and $Qm.n$ correspond to the floating point 32-bit value and fixed-point value with m sign+integer bits and n fractional bits, respectively.

Gaussian distribution and most of the elements are distributed within the center range. Thus, we apply Huffman coding to the weight values distributed within the range of $[X, Y]$ while the rest of the elements (i.e., out of the range) are maintained with the fixed-point elements. In the example shown in Fig. 5, we apply Huffman coding to the weight elements distributed within the range of $[-0.2, 0.2]$. The X and Y are the tunable parameters which can be determined by considering the distribution of the linearization weights and desired compression ratio.

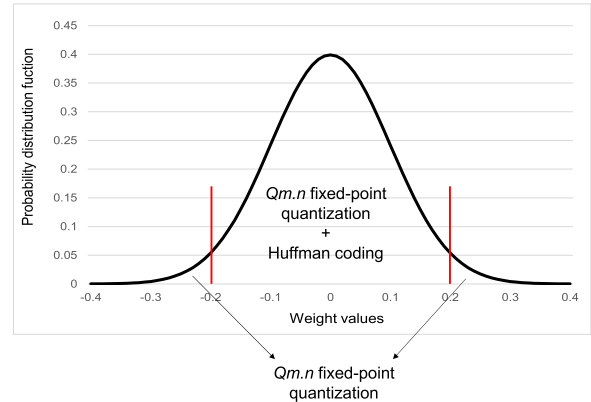


FIGURE 5. The illustrative figure of the linearization weight distribution. In this example, we apply Huffman coding to the weight values distributed within the range of $[-0.2, 0.2]$ (i.e., $X=-0.2, Y=0.2$ if we denote the range as $[X, Y]$).

For lossless compression, we also need to generate metadata during the compression. We first require a table which maintains the information of mapping between the fixed-point weight and Huffman-coded bit sequence (i.e., a codebook for Huffman coding). The second information we need is the bitmap that indicates whether the corresponding element is Huffman-coded or not. If the bit in the bitmap is 1, the corresponding element is Huffman-coded and vice versa. To reduce the decompression hardware complexity, we divide the compressed bitstream in fixed size chunk(s). In this paper, we generate each weight chunk with a size of 128B (1024b), meaning that there can be non-usable bits in the last part of the weight chunk. In this case, we put the zero-bit padding in the remaining part of the chunk. It simplifies the hardware logic, which makes a uniform size for each weight chunk (128B). To designate how many unusable bits exist in a certain compressed weight chunk, we also introduce an integer number N_{zp} which indicates the number of zero-padded bits in each weight chunk.

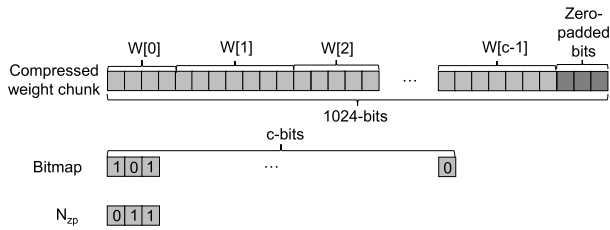


FIGURE 6. A composition of the compressed weight chunk with the metadata.

Fig. 6 shows a structure of a single compressed weight chunk with metadata. A single compressed weight chunk has c weights, which can be various depending on the compression ratio (i.e., the compression ratios will vary depending on how many weights can be compressed within a single chunk). For remaining parts of the chunk, we fill it with the 0_2 s as shown in Fig. 6 (3-bits are zero-padded in this example). In the metadata, we have c -bit bitmap to designate whether the corresponding weight element is Huffman-coded or not in the compressed weight chunk. In Fig. 6, we also have N_{zp} of $(011)_2$ ($=3$) as we have 3-bits for zero-padded bits.

We could also employ Huffman coding to the entire linearization weights (i.e., W^Q , W^K , W^V , and W^O). However, there are several reasons why we selectively employ Huffman coding. First, restricting the number of Huffman-coded weights can make the hardware decoder (for restoring the Huffman-coded bitstream to the fixed-point weights) less complicated. It can reduce the required number of the Huffman codebook table entries, resulting in less cycles and energy consumption for decoding. Second, the more the number of possible Huffman-coded weights we have, the longer the bit sequence we need to encode the weights with Huffman coding. It means that employing Huffman coding to all the weights may lead to suboptimal compression ratios due to the weights that have relatively long bit sequence. Moreover, these long bit sequence weights will also lead to more latency for weight decoding due to the complex decoding procedure for longer length of the bit sequences (for details of our proposed hardware decoder architecture, please see Section III-C). Thus, we selectively employ Huffman coding to the weights distributed within a certain range (e.g., $[-0.2, 0.2]$).

C. NEAR-MEMORY HARDWARE DECODER ARCHITECTURE
1) ARCHITECTURE AND DESIGN

Since we compress the linearization weight elements with Huffman coding, we need to decompress the elements when executing the NMT inference. During compression, we selectively compress the weight elements with Huffman coding. By referring to the bitmap values, we can specify whether a certain element is Huffman-coded or not. When decompressing the elements with only quantization (i.e., not compressed with Huffman coding), we can just pass them to the computation units such as GPU. However, when decompressing

the elements with quantization and Huffman coding, we need to search for the codebook table stored in the hardware and convert them into the quantized (i.e., fixed-point) weight elements. Decompressing the Huffman-coded bitstream requires non-negligible latency because the Huffman-coded bitstream should be decompressed sequentially. As we need the entire weight elements (W^Q , W^K , W^V , and W^O) for the linearization, the fast in-situ weight decoding is crucial for system performance and energy efficiency.

As shown in Fig. 7, the decoding procedure works as follows. The compressed weight bitstream is managed as multiple chunks while a single weight chunk size is $128B^3$ (please refer to Section III-B2 for further details). The corresponding bitmap is sliced and managed as multiple blocks of the bitmaps, which is maintained in a pair-wise manner with the compressed weight chunk. The access to the weight bitstream is performed with the sliding window, which designates a sliced weight bitstream. The sliced weight bitstream is fed into the lookup table (or just passed in the case of non-Huffman-coded weight elements) and decoded weight elements are delivered to the GPU memory for the matrix multiplication (i.e., linearization) with input token or concatenated matrix.

Fig. 7 illustrates the hardware architecture of a single hardware decoder unit for in-situ weight decompression. There are two input memory arrays: compressed weight chunk and bitmap. The compressed weight chunk and bitmap arrays act as an on-chip input buffer, which temporarily contains the compressed weight chunk and corresponding bitmap. The compressed weight chunk and bitmap are accessed by the idx_w and idx_b , respectively (① in Fig. 7), which are set to zero at the beginning of the weight chunk decoding. The compressed weight chunk is accessed with max -bits⁴ and $m + n$ -bits because the compressed weights have a variable length depending on whether Huffman-coded or not and how many bits are used for Huffman coding (②-1 and ②-2 in Fig. 7). Please note that the registers indicated by ②-1 and ②-2 in Fig. 7 are required for non-Huffman-coded and Huffman-coded elements, respectively. The appropriate bit length will be selected later by referring to the bit from the bitmap and codebook lookup table (LUT). The accessed compressed bit sequence (max -bits) are fed into the LUT⁵ that contains the codebook information between the Huffman-coded bits and original fixed-point weights (③ in Fig. 7). Once the LUT access finishes, the decoded weight element from the LUT and the passed weight element are fed into the rightmost MUX as inputs (④ in Fig. 7). By referring to the corresponding bit from the bitmap, we select either only

³Though we use 128B for the compressed weight chunk size in this paper, it is a tunable design parameter that can be determined considering the hardware decoder design.

⁴The size of max is determined by the maximum length of the Huffman-coded weight bit sequence.

⁵Even though we cannot find the corresponding bit sequence from the table in the case of non-Huffman-coded weights, the MUX in front of the decoded weight array can filter out the wrong values, avoiding the malfunction.

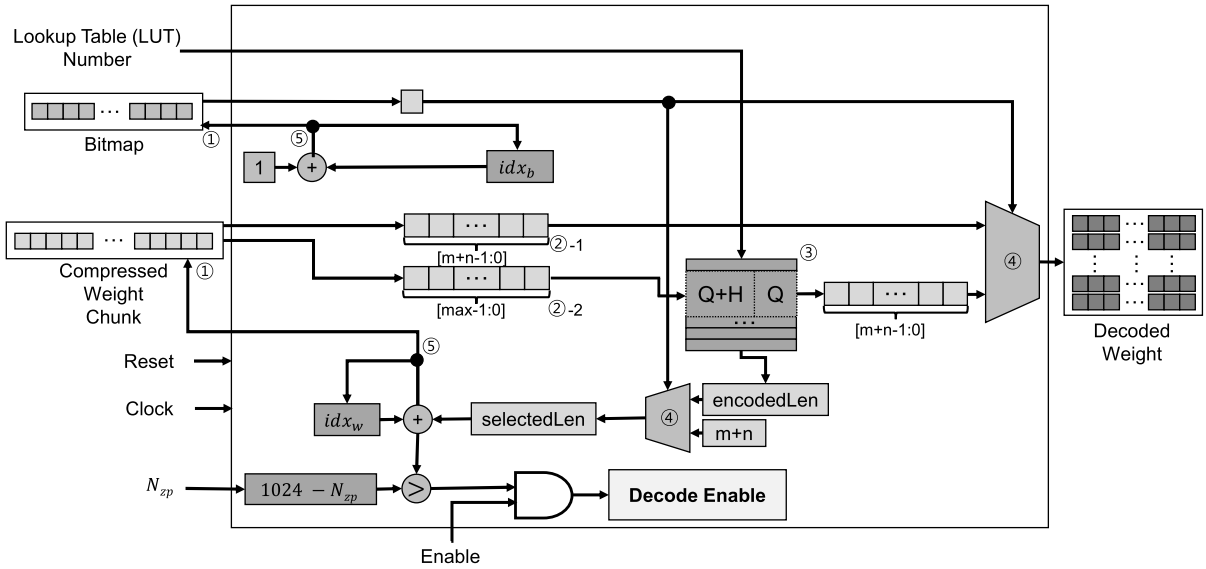


FIGURE 7. Near-memory decoder processing element architecture. Please note that m and n refer to the bitwidth of the sign+integer part and fractional part, respectively, in the fixed-point value ($Qm.n$ notation).

quantized or quantized+Huffman-coded weight element as output. In the case of the Huffman-coded weight element decoding, the length of the element is also determined by the MUX (below) and we increment the idx_w by the selected length (④ in Fig. 7). The decoded weight elements are stored into a temporary on-chip buffer shared across the decoder unit and finally delivered to the memory in the computation units (i.e., GPUs), which are used as the inputs for linearization (W^Q , W^K , W^V , and W^O). After that, the sliding windows in the weight chunk and bitmap move towards the next element decoding by adjusting the idx_w and idx_b (⑤ in Fig. 7).

For correct decoder operations, we need two more inputs: LUT table numbers and N_{zp} . The LUT table number is required to select the correct Huffman codebook LUT among the different types of the weights. For example, in our case of Transformer model, we have two types (encoder and decoder) of the layers \times 4-type weights = 8 codebook LUTs.⁶ The integer number N_{zp} specifies how many bits are zero-padded for each weight chunk as explained in Section III-B2. We compare the current idx_w with N_{zp} and decoding is performed as long as the $idx_w + selectedLen$ ($selectedLen$ is determined by how many bits in the current weight element exist) is less than 1024 (the weight chunk bit size) - N_{zp} .

2) TIMELINE ANALYSIS AND LATENCY HIDING

During NMT model inferences, we need to execute multiple layers, which are sequentially executed in general. Since we need the decoding of the W^Q , W^K , W^V , and W^O weights before the linearization execution, there would be additional latency overhead if we do not overlap the weight decoding

latency. In order to hide the decoding latency, we try to hide the latency of the weight decoding to that of the other computations in the MHA, which is a similar approach to [7].

Before linearization of the input tokens with W^Q , W^K , and W^V , we need to decode those W^Q , W^K , and W^V weights, respectively. In the encoder or decoder layers except the first layer, we can hide the W^Q , W^K , and W^V decoding latency with the latency of the previous layers in the GPU. For W^O decoding in the certain attention layer, it can be done during the linearization of W^Q , W^K , and W^V of the layer itself. Consequently, the only latency overhead comes from the W^Q , W^K , and W^V decoding latency of the first layer. However, modern attention-based NMT models are adopting many layers in the encoder and decoder, which makes the latency overhead of the first layer decoding marginal.

In this paper, we consider two scenarios: w/ and w/o weights loading and initialization, which are referred to as scenario A and B, respectively. In the case of multi-tenant GPU environments where the multiple models can be executed simultaneously, the latency for data transfer between the storage/host memory and GPU memory is significant due to a large amount of the data transfer. In addition, GPU initialization and warmup also require non-negligible latency. In this case, our proposed method can effectively reduce the data transfer latency due to our proposed data compression method.

As shown in Fig. 8, in the case of Transformer, the data transfer time is reduced by 9.5% before starting the decoding and computation (from 1429.5ms in Fig. 8 (a) to 1294.2ms in Fig. 8 (b)). During the NMT inference (i.e., after the data transfer and initialization), the decoding latency can be hidden by other computation latency, making the decoding latency overhead negligible. As shown in Fig. 8 (b), the decoding can begin as soon as the data transfer and initialization finishes and the linearization can also begin as soon as the

⁶In other words, we apply Huffman coding for each W^Q , W^K , W^V , and W^O of all the encoder layers and all the decoder layers, deriving total eight Huffman tables in the case of Transformer model.

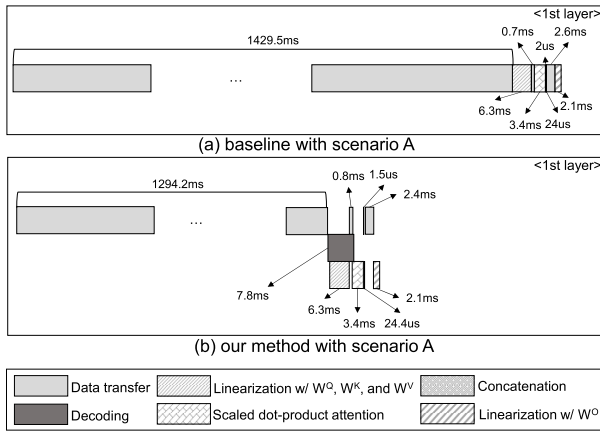


FIGURE 8. Timeline analysis when executing the Transformer model in the case of scenario A. The execution times for data transfer, linearization, concatenation, and scaled dot-product attention are obtained from the GPU execution with NVIDIA RTX 3080TI.

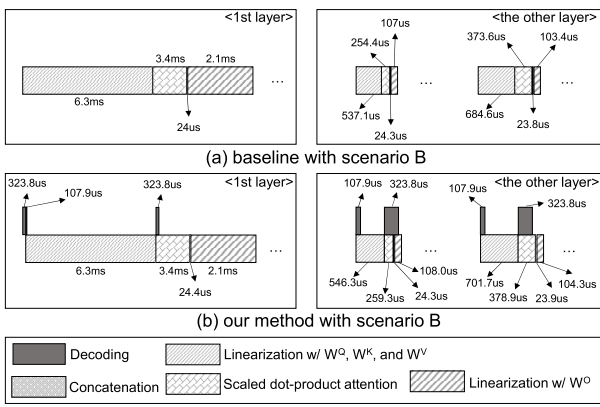


FIGURE 9. Timeline analysis when executing the Transformer model in the case of scenario B. The execution times for linearization, concatenation, and scaled dot-product attention are obtained from the GPU execution with NVIDIA RTX 3080TI.

decoding of the W^Q , W^K , and W^V for the first layer finishes. For decoding of the remaining weights, it can be hidden by the latencies of the linearization for W^Q , W^K , and W^V and scaled dot-product attention. Please note that this scenario can occur very often in the case of multi-tenant GPU in modern data centers. As the multiple DNN workloads will compete with each other to reside in the GPU memories, resulting in the shortage of the memory. In this case, weight loading from the host memory (or storage) will also occur very often. It makes our weight compression effective due to the reduction of the size and the loading latency of the weights.

In the case where the compressed weights are stored in the GPU memory while not decoded yet (i.e., in the case without weight loading and initialization), we may not be able to obtain performance improvement. The decoding of the W^Q , W^K , and W^V for the first layer incurs latency overhead because it cannot be hidden (323.8 μ s in Fig. 9 (b) <1st layer>). On the contrary, for the other layers, we can hide the decoding latency to that of other operations such as linearization and scaled dot-product as shown in Fig. 9 (b)

TABLE 1. Logic synthesis and CACTI-based SRAM array estimation results.

	Latency (cycle)	Power (mW)	Area (mm ²)
Logic	8.7 (avg.)	0.550	0.016
Buffer	Input	1	77.189
	Output	1	477.834

<the other layer>. Though we cannot obtain performance gain in this scenario, weight compression is still effective when maintaining the weight in the memory or storage by reducing the required capacity and energy for maintaining the weights.

3) LOGIC SYNTHESIS AND SRAM ARRAY ESTIMATION RESULTS

We have implemented our proposed hardware decoder with Verilog hardware description language and synthesized with 32nm process library at 500MHz clock frequency. Table 1 shows the latency, power, and area of our hardware decoder. Our single decoder takes 8.7 cycles per single weight element decoding,⁷ on average, with consuming 0.55mW. Since we need real-time decoding during the NMT inference, to meet the latency requirements, we employ multiple decoders which perform the decoding of the multiple compressed weight bitstreams in parallel. In this paper, we use 1,408 decoders, which can fully overlap the decoding latency to the data transfer and other computations (e.g., scaled dot-product attention) latency. Since our single decoder decompresses 128B weight bitstream at once, decoding the whole weights for linearization (W^Q , W^K , W^V , and W^O) requires 7.8ms for Transformer model. As we explained in Section III-C2, the other execution steps such as scaled dot-product or other linearization steps can sufficiently hide the decoding latency, meaning that the performance overhead of the runtime decoding is marginal. The logic synthesis results are summarized in Table 1.

For SRAM arrays in the HBM2 logic die, we have used CACTI [18] for latency, power, and area estimation with 32nm process technology. The results are also summarized in Table 1. We use 176KB and 660KB for input and output buffers, respectively, which are shared across multiple decoders. Considering the clock cycle time of the synthesized logic, accessing the input and output buffers can be done within a single clock cycle. For area, when employing 1,408 decoders with 836KB SRAM arrays, the total area will be 25.1mm², implying that the decoders can be placed in the processing-in-memory (PIM) cluster of the HBM2 logic die (26.27mm² [17]).

IV. EVALUATIONS

A. METHODOLOGY

For accuracy (i.e., NMT score) evaluations of our proposed method, we have used the PyTorch framework [19]

⁷In our implementation, LUT is implemented as a form of finite state machine, meaning that the number of cycles required for decoding is proportional to the length of the encoded bit sequences.

with five widely used attention-based NMT models: Transformer (Tf) [1], Transformer-XL-base (Tf-XL-b) and Transformer-XL-large (Tf-XL-l) [20], and BERT-base (BERT-b) and BERT-large (BERT-l) [2]. To quantify model accuracy, we have used various metrics: SacreBLEU score for Transformer, perplexity for Transformer-XL-base and Transformer-XL-large, and F1 score for BERT-base and BERT-large. The reason why we use different metrics for models is that we have used in-built accuracy quantification codes in the framework, which report more stable and trustworthy accuracy results. Please note that WMT14 English-German [21], wikiText-103 [22], and SQuAD v1.1 [23] have been used as the dataset for Transformer, Transformer-XL (both base and large), and BERT (both base and large), respectively. Since our proposed quantization and compression method can also be employed to any FP32-based weights, the arbitrary (or trained by ourselves) weight element values could also be used for our evaluations. However, we have used the linearization weights provided by NVIDIA NGC Catalog [24], [25], [26], [27], [28] because they are more stable, trustworthy, and widely used for research.

For performance evaluation, we have used GPU-based emulation. We have assumed that the operations for the attention layers are performed in the GPU. Therefore, we have measured the execution time of each layer by using NVIDIA RTX3080 TI [29]. For the baseline, we use architecture explained in Section II-C. For the baseline performance, we use the execution time measured from the GPU. For the case with our proposed method, the decoding time is estimated from our synthesis results and the layer execution time is measured from the GPU. The full decoding time is added to the layer execution time when the latency cannot be fully hidden. In the case where the latency hiding can be employed, we assume that the decoding latency can be hidden by execution latency of other parts such as MHA. By transferring the decoded weights from the decoder output buffer to the GPU memory, which can also be performed in background, the GPU can seamlessly access the linearization weights.

B. PERFORMANCE

We compare performance of our method and hardware architecture to that of the GPU-based system explained in Section II-C. We set the GPU-based execution as our baseline because recently, the execution for NMT models is typically done in the GPUs. For GPU-based execution, we use non-quantized FP32-based weights for Tf, Tf-XL-b, Tf-XL-l, and BERT-b, and FP16 weights for BERT-l. By employing our proposed method, we perform the quantization and Huffman coding-based compression offline. As presented in Section III-C2, we consider two different scenarios: w/ (A in Fig. 10) and w/o weights loading and initialization (B in Fig. 10). For the scenario B, we show the speedup when executing the entire layers, only the first layer, and the rest of the layers except for the first layer. Please note that we have

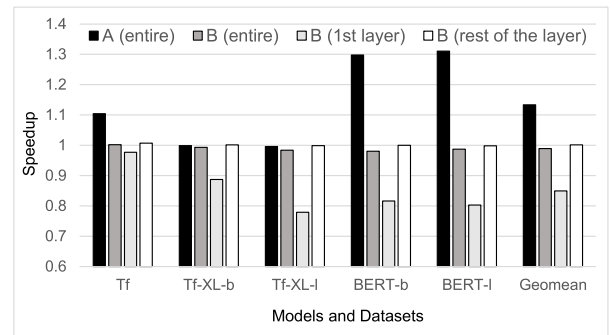


FIGURE 10. Speedup of our proposed method with near-memory processing as compared to the baseline. Please note that 'A' and 'B' refer to the scenario A and B, respectively.

only measured the execution time of multi-head attention because it is a key part of the modern NMT models. The speedup of our proposed method as compared to the baseline can be calculated as follows:

$$Speedup = \frac{T_{base}}{T_{prop}} \quad (1)$$

where T_{prop} and T_{base} correspond to the execution time when employing proposed method and the baseline, respectively.

In the case of scenario A where we consider the weight loading and initialization, our NMP architecture with the quantization and compression leads to a performance improvement of 13.3%, on average, across various NMT models. The reason why we can obtain the performance improvements is that we can reduce the weight loading and initialization latency due to the reduced weight size. As presented in Section III-C2, even with the first layer decoding latency which cannot be hidden, the performance gain from our weight compression when executing the weight loading and initialization is significant. Results depicted in Fig. 8 reveal that most performance gain from our proposed method comes from the first layer execution in scenario A. This is because the weight loading and model initialization latency is included in the first layer execution time. For the other layers, our proposed technique shows similar performance as compared to the baseline due to the decoding latency hiding, thus minimizing the decoding latency overhead. As an outlier, in the case of Transformer-XL models (Tf-XL-b and Tf-XL-l), our method leads to similar performance as compared to the baseline. The linearization weights account for only a 3%–9% of the entire weights in the Transformer-XL models, meaning that the performance gain from the loading and initialization latency reduction from linearization weight compression is small.

In the case of scenario B, our proposed method leads to the performance comparable to the baseline across the NMT models (a performance loss of only 1.1% for the entire layer execution). Since we have decoding latency overhead in the first layer, there is an additional latency overhead of 15.1% when executing the MHA execution in the first layer. On the contrary, for the other layers, there is a negligible latency

TABLE 2. Memory energy consumption (μJ) of the baseline and the proposed method across the models.

	Baseline			Our proposed method		
	Static	Dynamic	Total	Static	Dynamic	Total
Tf	956.3	570.2	1526.6	853.2	252.2	1105.4
Tf-XL-b	239.1	142.6	381.7	225.3	62.4	287.7
Tf-XL-l	956.3	570.2	1526.6	852.9	244.8	1097.7
BERT-b	537.9	320.9	858.8	487.4	141.1	628.5
BERT-l	478.2	285.1	763.3	854.1	247.5	1101.6

difference between the baseline and our proposed method. It means that we can successfully hide the decoding latency with other computation latency.

C. MEMORY ENERGY DURING DATA TRANSFER

We also present the results of the memory energy consumption in the case of the baseline and our proposed method. For the baseline, the memory energy consumption can be calculated as follows:

$$E_{base} = E_{w_r} + E_{static} \quad (2)$$

where E_{base} , E_{w_r} , and E_{static} correspond to the memory data transfer energy consumption in the case of baseline, weight read energy from the memory, and static energy during the memory data transfer, respectively. In case of our proposed method, the memory energy consumption is calculated with the following equation:

$$E_{prop} = E_{w_{comp_r}} + E_{dec} + E_{quant_w} + E_{static} \quad (3)$$

where E_{prop} , $E_{w_{comp_r}}$, E_{dec} , and E_{quant_w} correspond to the memory data transfer energy consumption in the case of our proposed method, compressed weight read energy from the memory, decoding energy from our hardware decoder, and quantized weight write energy to the memory, respectively. For energy evaluations, we have used HBM2 as our baseline memory for conservative evaluations.

As shown in Table 2, our method leads to an energy savings of 26.8%, on average, as compared to the baseline in the cases of Tf, Tf-XL-b, Tf-XL-l, and BERT-b. Since our proposed method reduces the size of the weights stored in the memory, our proposed method leads to a reduction of the memory read and static energy. Consequently, even with the decoding energy and weight write energy, total energy consumption can be reduced by leveraging our proposed method.

As an outlier, in the case of BERT-l, the energy consumption of our proposed method is rather increased as compared to the baseline. The main reason for this result is that the BERT-l uses FP16 as the baseline. When employing our proposed compression method, using FP16 as the baseline leads to a relatively lower compression ratio as compared to the case of using FP32 as the baseline. It makes the energy overhead from our decoders and buffers relatively higher than the case of using FP32, leading to a higher energy consumption. However, many contemporary NMT models are still using FP32 for the default precision, and thus our proposed method can still lead to energy savings in many cases.

D. TRADEOFF BETWEEN ACCURACY AND DATA SIZE (COMPRESSION RATIO)

We have summarized the tradeoff between accuracy of the NMT models and compression ratio. The compression ratio can be calculated as $\frac{Size_B}{Size_M + Size_C}$ where the $Size_B$, $Size_M$, and $Size_C$ are size of the uncompressed data, metadata, and compressed data, respectively. In order to figure out which configuration of $Q_{m,n}$ shows the best tradeoff between the accuracy and data size for each model, we have profiled the accuracy (e.g., scores or perplexity) and data size of NMT models and select one configuration (i.e., m and n) for each model that shows the best tradeoff.

As summarized in Table 3, our fixed-point quantization and Huffman coding-based compression show significant compression ratios of 4.94 – 10.04. In the case of BERT-l, the compression ratio is lower than the other models. This is due to the fact that the BERT-l model uses FP16 while the other models use FP32, meaning that the baseline weight element size of the BERT-l model is a half of that of the other models. For Tf-XL-b, BERT-b, and BERT-l, we adopt $Q_{1.5}$ fixed-point weight elements. Based on our observation that most of the linearization weight elements of Tf-XL-b, BERT-b, and BERT-l are distributed between -1.0 and $+1.0$, we only need a sign-bit for the integer part.

Since quantization is a lossy compression, there would be an accuracy loss. For various NMT models, there are many quantifying metrics for accuracy. In this paper, the accuracy metrics used for NMT models are different across the models for the sake of fair comparison. In the case of Tf, we only see 0.61 SacreBLEU score drop with our fixed-point quantization, which is negligible. Tf-XL models' perplexity scores (the lower, the better) show only a marginal increase by 1.34 – 1.62. In the case of BERT, F1 score drop is only 0.50–0.74. By adopting our proposed quantization and Huffman coding-based compression, we can obtain significant data size reduction with only small accuracy losses.

V. RELATED WORK

Recent advancements of attention-based NMT models have primarily focused on increasing sparsity, eventually reducing the computation complexity with sparsity-aware hardware and storage requirements. In [30], a sparse attention mechanism is proposed to reduce the computation complexity and memory requirements. The sparse attention mechanism enables $8\times$ longer sequence modeling with the identical hardware as compared to the full attention mechanism. In [31], a rectified linear unit (ReLU)-based activation is used instead of the complex softmax activation, which increases the sparsity and yields better scalability for modeling the long sequences. In [32], a sparse attention mask is employed for video captioning, which exploits dense sampling of the frames while trying to minimize the processing of the redundant frames. In [33], content-awareness is considered in sparse attention mechanism by clustering the queries and keys. With k-means clustering, only in the case where the

TABLE 3. The accuracy and compression ratio comparisons across the five attention-based NMT models. Please note that the accuracy metrics used for NMT models are different across the models due to the lack of the required model parameters and for the sake of the fair comparison.

	Accuracy				Compression ratio
	Metric	Baseline	Quantization (fixed-point notation)	Accuracy change (ratio in %)	
Transformer (large)	SacreBLEU	27.77	27.16 (Q3.4)	0.61 (2.20%)	9.14
Transformer-XL (base)	Perplexity	23.19	24.53 (Q1.5)	1.34 (5.78%)	9.08
Transformer-XL (large)	Perplexity	18.24	19.86 (Q2.5)	1.62 (8.88%)	10.04
BERT (base)	F1 score	88.69	87.95 (Q1.5)	0.74 (0.83%)	9.43
BERT (large)	F1 score	91.42	90.92 (Q1.5)	0.50 (0.55%)	4.94

queries and keys are in the same cluster, the attention mechanism can be performed. In [34], instead of static sparsity, dynamic sparsity (i.e., sparsity patterns are changed along with the execution of attention mechanism) is considered for Transformer model. Their work demonstrates the acceleration of the dynamic sparsity-aware attention mechanism in GPUs and accelerators. For dynamic sparsity awareness and acceleration of the sparse attention, algorithm-hardware co-design approach is also proposed [35], which is called as Energon. It identifies the important query and key pairs at runtime with the filtering approach, which is performed by the dedicated hardware accelerator. The accelerator also performs attention layer execution, resulting in an $8.7\times$ speedup as compared to the GPU-based execution. Another algorithm-hardware co-design approach with online pruning is proposed in [36]. The bit-serial accelerator, which is called as LEOPARD, performs the online pruning and other computations required for the attention mechanism, leading to a $1.9\times$ speedup with a negligible loss in accuracy. In another work, an in-memory pruning approach with the hardware accelerator is proposed [37]. By converting the low attention score values (thus, can be regarded as less important) into zero with the thresholding circuitry inside of the resistive random-access memory (ReRAM) arrays, it simply implements dynamic runtime pruning in hardware. In addition, the dedicated hardware accelerator further expedites the attention computation, resulting in a $7.5\times$ speedup with a negligible loss in accuracy. Another way to reduce the memory requirements and computation complexity is low-rank approximation [38]. In [39], a method which exploits both sparsity and low-rank approximation is proposed, which is also called as Scatterbrain. It is shown that the Scatterbrain can outperform the methods which employ only either sparsity or low-rank approximation, illustrating that the sparsity and low-rank approximation can be exploited synergistically. To reduce the computation complexity, a hardware-software co-design approach (which is called as ELSA) is proposed in [40] to reduce the computation complexity of the self-attention. By calculating a similarity measure, their results show that the ELSA achieves approximately a speedup of $58\times$ with a negligible loss in accuracy.

As the NMT models get larger and more complex, the size of weights for attention-based NMT models has also been increased. Thus, reducing the weight size of NMT models has gained huge attention as a promising research topic. In [13], Zadeh et al. have proposed a quantization-based weight

compression technique, which utilizes post-training weight distribution information. The quantization is performed for each layer where each layer has a centroid and 3-bit indices. To obtain a sufficient accuracy, 0.1% outliers for each layer are maintained as FP32. In [8], all trainable variable matrices and activation matrices are quantized to 8-bit integer format, resulting in $4\times$ compression ratio with less than 1.0 BLEU score drop. In [12], a progressive quantization method is employed, which primarily maintains most significant bits (MSBs) for each weight. Since the computation can also be done with only MSB part, the computation complexity can also be lowered. For certain cases, the LSB parts can also be used to sustain the satisfactory accuracy. As a result, the method proposed in [12] reduces DRAM accesses by $10\times$ and computations by $2\times$ without accuracy drop. In [10], the fixed-point quantization method is employed for size reduction of the query and key matrices. In [11], low-bit quantization (i.e., 4-bit quantization) to the input query and key matrices is employed. In addition, computation complexity has been reduced by using the sparse attention mask due to the increased sparsity.

The works introduced above have mostly focused on the reducing the computation complexity and memory requirements via sparsity, low-rank approximation, algorithmic optimization, quantization, and hardware acceleration for attention layer operation. Those works have mostly focused on the scaled dot-product attention while the linearization step is largely overlooked. On the contrary, our work employs selective Huffman coding as well as quantization, resulting in a better compression ratio for linearization weights. In addition, we have proposed a fast in-situ near-memory decoding hardware, which makes the decoding procedure fast and further hides the decoding latency by the execution latency of other MHA operations (e.g., scaled dot-product attention), thus minimizing the potential latency overhead.

VI. CONCLUSION

The contemporary NMT models have huge weight size for satisfactory accuracy scores. Particularly, the weights (W^Q , W^K , W^V , and W^O) for the linearization occupy non-negligible memory and storage capacity when employing huge NMT models. In this paper, we propose a linearization weight quantization and Huffman coding-based compression method. Our proposed method results in compression ratios of 4.9–10.0 across the five widely used attention-based NMT models with only marginal accuracy

drops. For fast in-situ weight decompression, we also propose near-memory decompression hardware architecture. The weight decompression (decoding) latency can be easily hidden by the latency required for other computations during the MHA execution. As a result, when considering the weight loading and initialization, our proposed method leads to 13.3% performance improvement as compared to the baseline (i.e., without using our method). In the case without the weight loading and initialization, the only latency overhead comes from the first layer execution, which accounts for only 1.1% overall performance overhead as compared to the baseline. Moreover, our proposed method and near-memory hardware lead to memory data transfer energy savings by 16.1%, on average, as compared to the baseline. As our future work, we plan to employ our quantization and compression method to other types of weights in attention-based NMT models and extend our hardware decoder for in-situ decompression of these weights.

REFERENCES

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. U. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–10.
- [2] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
- [3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020, *arXiv:2005.14165*.
- [4] (2021). *Openai, Chatgpt*. [Online]. Available: <https://openai.com/blog/chatgpt>
- [5] R. Thoppilan et al., "Lamda: Language models for dialog applications," *CoRR*, vol. abs/2201.08239, 2022. [Online]. Available: <https://arxiv.org/abs/2201.08239>
- [6] J. Kwon, J. Kong, and A. Munir, "Sparse convolutional neural network acceleration with lossless input feature map compression for resource-constrained systems," *IET Comput. Digit. Techn.*, vol. 16, no. 1, pp. 29–43, Jan. 2022.
- [7] J. H. Lee, J. Kong, and A. Munir, "Arithmetic coding-based 5-bit weight encoding and hardware decoder for CNN inference in edge devices," *IEEE Access*, vol. 9, pp. 166736–166749, 2021.
- [8] S. Lu, M. Wang, S. Liang, J. Lin, and Z. Wang, "Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer," in *Proc. IEEE 33rd Int. Syst., Chip Conf. (SOCC)*, Dec. 2020, pp. 84–89.
- [9] M. Zhou, W. Xu, J. Kang, and T. Rosing, "TransPIM: A memory-based acceleration via software-hardware co-design for transformer," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Apr. 2022, pp. 1071–1085.
- [10] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, and D.-K. Jeong, "A3: Accelerating attention mechanisms in neural networks with approximation," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, May 2020, pp. 328–341.
- [11] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 977–991.
- [12] H. Wang, Z. Zhang, and S. Han, "SpAtten: Efficient sparse attention architecture with cascade token and head pruning," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 97–110.
- [13] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "GOBO: Quantizing attention-based NLP models for low latency and energy efficient inference," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 811–824.
- [14] M. Zhu, Y. Zhuo, C. Wang, W. Chen, and Y. Xie, "Performance evaluation and optimization of HBM-enabled GPU for data-intensive applications," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 1245–1248.
- [15] (2021). *Achieving Fp32 Accuracy for int8 Inference Using Quantization Aware Training With Nvidia Tensorrt*. [Online]. Available: <https://developer.nvidia.com/blog/achieving-fp32-accuracy-for-int8-inference-using-quantization-aware-training-with-tensorrt/>
- [16] A. Álvarez, V. R. Gómez, I. G. Torre, T. Etchegoyhen, H. Gete, and J. Arellano, "Iassist: Low-cost, portable and embedded assistants for on-premise automated transcription and translation services," in *Proc. Annu. Conf. Spanish Assoc. Natural Lang. Process.*, Sep. 2022, pp. 75–78.
- [17] S. Kim, S. Kim, K. Cho, T. Shin, H. Park, D. Lho, S. Park, K. Son, G. Park, S. Jeong, Y. Kim, and J. Kim, "Signal integrity and computing performance analysis of a processing-in-memory of high bandwidth memory (PIM-HBM) scheme," *IEEE Trans. Compon., Packag., Manuf. Technol.*, vol. 11, no. 11, pp. 1955–1970, Nov. 2021.
- [18] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 1–25, Jun. 2017.
- [19] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," in *Proc. NIPS*, Feb. 2019, pp. 8024–8035.
- [20] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-XL: Attentive language models beyond a fixed-length context," 2019, *arXiv:1901.02860*.
- [21] O. Bojar, C. Buck, C. Federmann, B. Haddow, P. Koehn, J. Leveling, C. Monz, P. Pecina, M. Post, H. Saint-Amand, R. Soricut, L. Specia, and A. Tamchyna, "Findings of the 2014 workshop on statistical machine translation," in *Proc. 9th Workshop Stat. Mach. Transl.*, Baltimore, MD, USA, May 2014, pp. 12–58.
- [22] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016, *arXiv:1609.07843*.
- [23] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," 2016, *arXiv:1606.05250*.
- [24] (2020). *Transformer EN-DE Checkpoint*. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/models/transformer_pyt_ckpt_tf32/version
- [25] (2021). *Transformer-XL Pytorch Checkpoint (Base, Amp)*. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/teams/dle/models/transformerxl_base_pyt_ckpt/files
- [26] (2021). *Transformer-XL Pytorch Checkpoint (Large, Amp)*. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/teams/dle/models/transformerxl_large_pyt_ckpt/files
- [27] (2021). *Bert Pytorch Checkpoint (Base, QA, Squad1.1, Amp)*. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/teams/dle/models/bert_base_pyt_ckpt_mode-qa_ds-squad11
- [28] (2021). *Bert Pytorch Checkpoint (Large, QA, Squad1.1, Amp)*. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/teams/dle/models/bert_large_pyt_ckpt_mode-qa_ds-squad11
- [29] (2021). *Nvidia RTX 3080 TI*. [Online]. Available: <https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080-3080ti/>
- [30] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," 2020, *arXiv:2007.14062*.
- [31] B. Zhang, I. Titov, and R. Sennrich, "Sparse attention with linear units," 2021, *arXiv:2104.07012*.
- [32] K. Lin, L. Li, C.-C. Lin, F. Ahmed, Z. Gan, Z. Liu, Y. Lu, and L. Wang, "SwinBERT: End-to-end transformers with sparse attention for video captioning," 2021, *arXiv:2111.13196*.
- [33] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, "Efficient content-based sparse attention with routing transformers," *Trans. Assoc. Comput. Linguistics*, vol. 9, pp. 53–68, Feb. 2021.
- [34] L. Liu, Z. Qu, Z. Chen, Y. Ding, and Y. Xie, "Transformer acceleration with dynamic sparse attention," 2021, *arXiv:2110.11299*.
- [35] Z. Zhou, J. Liu, Z. Gu, and G. Sun, "Energon: Towards efficient acceleration of transformers using dynamic sparse attention," 2021, *arXiv:2110.09310*.
- [36] Z. Li, S. Ghodrati, A. Yazdanbakhsh, H. Esmaeilzadeh, and M. Kang, "Accelerating attention through gradient-based learned runtime pruning," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, Jun. 2022, pp. 902–915.

- [37] A. Yazdanbakhsh, A. Moradifirouzabadi, Z. Li, and M. Kang, "Sparse attention acceleration with synergistic in-memory pruning and on-chip recomputation," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2022, pp. 744–762.
- [38] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," 2020, *arXiv:2006.04768*.
- [39] B. Chen, T. Dao, E. Winsor, Z. Song, A. Rudra, and C. Ré, "Scatterbrain: Unifying sparse and low-rank attention approximation," 2021, *arXiv:2110.15343*.
- [40] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, "ELSA: hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 692–705.



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.

He is currently an Associate Professor with the Department of Computer Science, Kansas State University. He was a Postdoctoral Research Associate with the Electrical and Computer Engineering Department, Rice University, Houston, TX, USA, from May 2012 to June 2014. From 2007 to 2008, he was a Software Development Engineer with Mentor Graphics Corporation, Embedded Systems Division. 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 received 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).

...



MJIN GO (Student Member, IEEE) received the B.S. degree in electronics engineering from Kyungpook National University, in 2021, where she is currently pursuing the M.S. degree with the School of Electronic and Electrical Engineering. Her research interests include neural machine translation model acceleration, data compression, near-memory processing, and FPGA-based design.



JOONHO KONG (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Korea University, in 2007, 2009, and 2011, respectively. He was a Postdoctoral Research Associate with the Department of Electrical and Computer Engineering, Rice University, from 2012 to 2014. Before joining Kyungpook National University, he was a Senior Engineer with 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, deep learning acceleration, and hardware/software co-design.