Exploiting Kernel Compression on BNNs

Franyell Silfa
Universitat Politècnica de Catalunya
Barcelona, Spain
fsilfa@ac.upc.edu

Jose Maria Arnau
Universitat Politècnica de Catalunya
Barcelona, Spain
jarnau@ac.upc.edu

Antonio González
Universitat Politècnica de Catalunya
Barcelona, Spain
antonio@ac.upc.edu

***Abstract*—Binary Neural Networks (BNNs) are showing tremendous success on realistic image classification tasks. Notably, their accuracy is similar to the state-of-the-art accuracy obtained by full-precision models tailored to edge devices. In this regard, BNNs are very amenable to edge devices since they employ 1-bit to store the inputs and weights, and thus, their storage requirements are low. Moreover, BNNs computations are mainly done using xnor and pop-counts operations which are implemented very efficiently using simple hardware structures. Nonetheless, supporting BNNs efficiently on mobile CPUs is far from trivial since their benefits are hindered by frequent memory accesses to load weights and inputs.**

In BNNs, a weight or an input is stored using one bit, and aiming to increase storage and computation efficiency, several of them are packed together as a sequence of bits. In this work, we observe that the number of unique sequences representing a set of weights or inputs is typically low (i.e., 512). Also, we have seen that during the evaluation of a BNN layer, a small group of unique sequences is employed more frequently than others. Accordingly, we propose exploiting this observation by using Huffman Encoding to encode the bit sequences and then using an indirection table to decode them during the BNN evaluation. Also, we propose a clustering-based scheme to identify the most common sequences of bits and replace the less common ones with some similar common sequences. As a result, we decrease the storage requirements and memory accesses since the most common sequences are encoded with fewer bits.

In this work, we extend a mobile CPU by adding a small hardware structure that can efficiently cache and decode the compressed sequence of bits. We evaluate our scheme using the ReAacNet model with the Imagenet dataset on an ARM CPU. Our experimental results show that our technique can reduce memory requirement by 1.32x and improve performance by 1.35x.

***Index Terms*—Machine Learning, Binary Neural Networks**

I. Introduction

Deep Neural Networks (DNNs) have recently achieved tremendous success in various tasks such as image classification and speech recognition. Not surprisingly, they have become the most critical component for Machine Learning Applications. Nonetheless, the cost of storing and evaluating DNNs is high since they usually are composed of millions of parameters. Mainly, evaluating DNNs on edge devices is quite challenging due to the constrained resources on edge devices. In this regard, Binary neural networks (BNNs) are an attractive solution to reduce the cost of evaluating DNNs on edge devices. In BNNs, the weights and input features can only take the value of 1 and -1; thus, one bit is used to store them. Therefore, the cost of storing a BNN layer can be reduced dramatically compared to those that use integer or floating point values. Moreover, BNNs can be executed very efficiently with modest hardware since the matrix multiplications required to evaluate a DNN, can be performed using a series of xnor and popcount operations.

While BNNs are very computationally efficient, their accuracy tends to be relatively low compared to a similar state-of-the-art full-precision network. However, recent BNN models, such as ReactNet [1] and FracBNN [2], achieve a top-1 accuracy comparable to state-of-the-art full-precision networks. For instance, ReActNet obtains a top-1 accuracy of 69% on the Imagenet dataset while MobileNet V2 [3] (i.e., CNN model tailored to edge devices) has a top-1 accuracy of 72%. Nevertheless, the number of bits required to store ReactNet is 29 million, whereas it is 27 million for MobileNet V2. Hence, evaluating these models on edge devices is still challenging due to their limited storage and computational resources. Most of the storage and computational requirements of BNNs are due to convolutional operations.

In this work, we focus on improving the performance and storage requirement of BNN models using convolutional operations. In this regard, in a binary kernel (i.e., weights are 1 or -1), for any given *n by n* channel stored as a sequence of bits, there are 2^{n*n} possible sequences of bits. However, we observe that during the evaluation of a convolution operation for a given kernel, some sequences of bits are used more frequently than others. For instance, in some layers of ReActNet, 32 individual sequences of bits account for more than 50% of all the possible sequences. To exploit this observation, we propose to employ a variable length compression algorithm to encode each sequence of bits. Then, we assign an encoding with fewer bits to the most common sequences since they are more likely to be used when evaluating a kernel. For simplicity in the rest of the paper, we refer to the sequence of bits formed with the values of a given channel as *bit sequence*.

A significant challenge in implementing a variable-length encoding is providing a software- and hardware-friendly solution while also being efficient [4]. A software-only implementation is insufficient since the performance is severely affected by the overhead of decoding the compressed bit sequences. We address this issue by providing hardware support to decompress the *bit sequences*. Moreover, when evaluating BNNs on mobile CPUs, the matrix multiplications are done very fast since the latency of computing a xnor and a popcount is very low. Therefore, the loads to fetch the weights are in the critical path. Hence, we also provide hardware support to stream the weights automatically to mitigate this issue.

Another important observation is that for some layers, a less

978-3-9819263-7-8/DATE23/© 2023 EDAA

frequently used *bit sequence* can be replaced by one that is employed more often without negatively impacting the accuracy of the network. Accordingly, we propose a clustering algorithm that substitutes some of the less common *bit sequences* with others that are similar to them but also very common. By reducing the less common *bit sequences*, we can compress the binary kernels more efficiently since the number of bits to store the most common ones is smaller.

In summary, the main focus of this paper is improving the performance and memory requirements of BNNs on mobile CPUs. We claim the following contributions:

- We analyze the frequency distribution of all the *bit sequences* in the network and show that some of them are used more frequently than the rest.
- We propose to use a simple variable-length encoding to compress the binary kernels. Also, we describe a novel clustering algorithm that increases the frequency of the most common *bit sequences*. Our scheme compresses the BNN layers by 1.32x on average and improves the model storage requirements by 1.2x.
- We propose simple hardware extensions and evaluate them on top of a modern ARM CPU. Our proposals improve the performance by 1.35x.

II. Background

A. Binary Neural Networks

Like full-precision DNNs, Binary Neural Networks (BNNs) are composed of many layers (i.e., convolution, dense, etc) stacked on top of each other. BNNs use one-bit weights and inputs constrained to $+1$ and -1, but they usually are trained using full-precision weights. Then, the weights and inputs are binarized using the following equation:

$$x^b = \begin{cases} +1 & \text{if } x >= 0, \\ -1 & \text{otherwise,} \end{cases} \quad (1)$$

where x is a full-precision weight or input and x^b is the binarized value.

Convolution operations with binary inputs and weights are computed as their full-precision counterpart. However, to calculate an output feature Equation 2 is employed. In this equation, o is a feature of the output feature map, whereas w^b and x^b are the binarized kernel (weights) and input vectors. Note that the kernel and the input are composed of 1 and -1, but they are stored as 1 and 0, respectively.

$$o = popcount(xnor(w^b, x^b)) \quad (2)$$

Equation 2 is essentially computing a multiply-accumulate operation (MAC) which is the main computation on DNNs. Hence, as mentioned earlier, BNNs are very efficient in computations and memory requirements since MAC operations can be computed using xnor and popcount functions, and weights are stored using 1 bit.

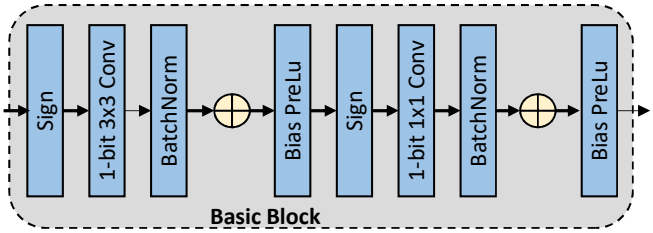

Fig. 1. ReActNet basic block and Architecture.

TABLE I
ReActNet storage and execution time breakdown. Others refers to activation and normalization layers,etc.

Operation	Storage (%)	Precision (bits)	Execution time(%)
Input Layer	0.02	8	4.0
Output Layer	22.17	8	18.7
Conv 1x1	8.5	1	6.9
Conv 3x3	68.0	1	66.8
Others	1.31	32	3.6

B. BNN model

Recently, on image classification tasks, BNN models can achieve top-1 accuracy, similar to the accuracy of full-precision models tailored to edge devices. In this regard, ReActNet [1] is a BNN model based on MobileNet [3] with a top-1 accuracy of 69.4% on the ImageNet dataset. Note that the accuracy of Mobilenet V2 for the same task is 72%. Since the accuracy of ReActNet is very close to the full-precision counterpart, we employ this model as the baseline in this work.

The main building block of ReActNet is shown in Figure 1. This block comprises two convolutional operations that employ binarized input and kernels. Also, it performs other operations such as batch-norm and Prelu activation functions which are computed using full-precision. One of the critical contributions of this model is that the Prelu activation is biased by shifting and reshaping its input. By performing this transformation, the accuracy of the network is substantially improved [1].

ReActNet is composed of 15 layers which include one input, one output layer, and 13 layers that are constructed using the basic block shown in Figure 1. The input layer is convolutional, whereas the output layer is fully-connected. Both layers are computed using full-precision values, and in this work, we quantize them using 8 bits.

Table I shows the storage requirements and execution time for the main operations in ReActNet. The storage requirement due to the 3x3 convolutions accounts for 68% of the total. Also, these 3x3 convolutions represent 67% of the execution time; hence, we focus on improving them.

III. Kernel Compression

We observe that for the 3x3 binarized kernels found in ReActNet, each channel is composed of nine values that are either one or zero; thus, there are only 512 unique *bit sequences*. However, some *bit sequences* are rarely used, while other sequences tend to appear very frequently. Therefore, we could exploit this observation by employing a variable encoding that assigns fewer bits to the most common *bit sequences*. Moreover, we experimentally found that some rarely used

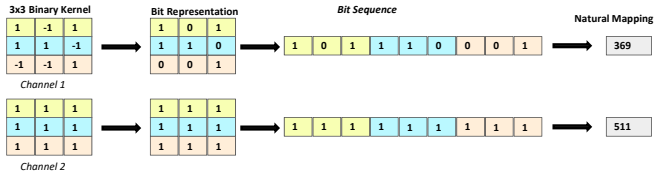

Fig. 2. 3x3 binary kernel with 2 channels. Conceptually the values of a channel are mapped to an integer number.

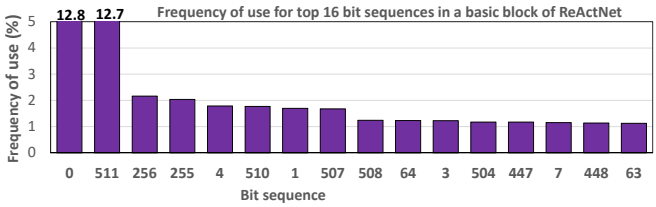

Fig. 3. Frequency of use for the top 16 *bit sequences* for the 3x3 binary kernels in one of ReActNet's basic blocks.

bit sequences can be substituted by some others that are more common without negatively affecting the overall network accuracy.

Figure 2 shows a binarized 3x3 kernel with 2 channels. As mentioned earlier, each of these channels is composed of nine values which are either 1 or -1, and to represent one channel a least 9 bits are needed. For clarity, to refer to a given *bit sequence*, we employ a Natural mapping, as shown in Figure 2, that assigns a Natural number to each *bit sequence*. In this regard, we assign the value in position 0,0 to the most significant bit and the value in position 2,2 to the least significant bit. For example, a channel where all the values are *zeros* (i.e., -1) will be referred to as 0. Similarly, a channel where all the values are *ones* will be mapped to 511. Note that this is a conceptual representation, and the actual layout of these bits in memory can be different.

The rest of this section presents an analysis of the use frequency for each *bit sequence* in ReActNet. Then, we describe our methodology to replace uncommon *bit sequences*. Finally, we detail our compressing scheme.

A. Frequency of use for bit sequences

Figure 3 shows the frequency of use for the 16 more common *bit sequences* in all the 3x3 binary kernels in one of the basic blocks of ReActNet. As shown in the plot, *bit sequences* containing only zeros or ones account for 25%. In contrast, the most common 16 and 64 *bit sequences* account for 46% and 75% of all of them, respectively. Regarding the kernels in other basic blocks, their behavior is similar to the one in Figure 3. As shown in Table II, in all the blocks, the top 64 *bit sequences* account for more than 50% of the sequences. Moreover, for all the blocks, the 256 most frequently used *bit sequences* account for up to 92%. Henceforth, instead of employing nine bits to store each channel, we could use 5 bits to store half of them and nine or less for the rest.

B. Compressing bit sequences

Aiming to increase storage efficiency and to exploit the variability in the frequency of use of the *bit sequences*, we propose to use Huffman Encoding to encode them. When using Huffman encoding to compress the kernels, we must first create a Huffman tree, where each *bit sequence* is assigned to a tree node. Then, before using a *bit sequence*, we must decode it with the help of lookup tables that contain the uncompressed value. Note that the tree is created offline, whereas the decoding is done at runtime.

A significant issue when decoding a Huffman encoded sequence of bits is that it requires optimized lookup tables or complex hardware if high throughput is needed [4]. In this regard, we proposed to use a simplified Huffman tree where we limit the tree size to a small number of nodes (i.e., four), as shown in Figure 4. In this tree, several encoded *bit sequences* are assigned to each node. Note that we could employ a table to store the encoded *bit sequences* belonging to a given node. Hence, during the decoding process, we use the first bits of the encoded *bit sequence* to find out to which table (node) it belongs and then employ the remaining bits to address the table and get the uncompressed *bit sequence*. This simplified version provides a good trade-off between simplicity and compression rate, as shown in Section VI.

TABLE II
DISTRIBUTION OF *bit sequences* FOR THE 3X3 KERNEL IN EACH BASIC BLOCK.

Layer	Top 64 (%)	Top 256 (%)
Block 1	53.4	90.6
Block 2	64.5	95.1
Block 3	56.3	87.11
Block 4	64.8	92.7
Block 5	63.2	88.3
Block 6	63.1	90.86
Block 7	62.4	91.64
Block 8	60.8	90.24
Block 9	55.2	92.9
Block 10	62.2	89.9
Block 11	67.97	92.
Block 12	75.3	93.4
Block 13	58.3	86.9

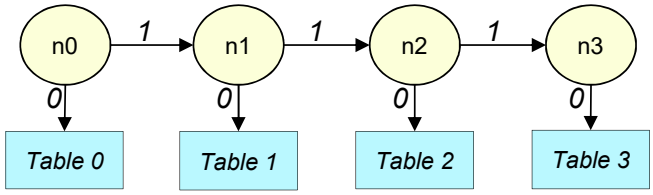

Fig. 4. Simplified Huffman tree with four nodes. The prefix of an encoded bit sequence is used as a pointer to a table with the uncompressed value.

C. Removing less frequent bit sequences

We observe that in some cases, a rarely used *bit sequence* (sa) can be replaced by one employed more frequently (sb) without negatively affecting the model accuracy. For this, we constrain the hamming distance between sa and sb to 1 (i.e., only a one-bit difference) to keep the error introduced during the evaluation low. Hence, we can further improve the compression ratio by increasing the frequency of use of the most common *bit sequences* and removing some of the less common ones. To exploit this observation, we employ the following algorithm.

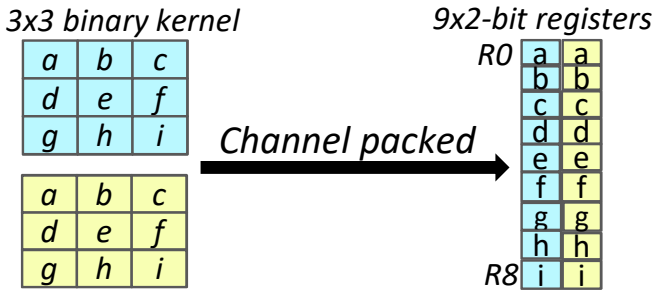

Fig. 5. Channel packing a 2-channel 3x3 binary kernel. In this case, a 2-bit register is used to store a bit from each channel.

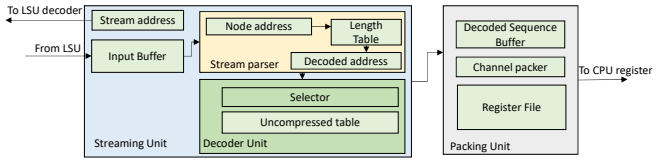

Fig. 6. Decoding Unit

First, for all the 3x3 binary kernels in a given basic block, we create a set st containing the M most commonly used $bit\ sequences$. Then, we create a second set su containing the N less commonly employed. After this, we try to replace the $bit\ sequences$ in su with $bit\ sequences$ from st. This process is done by first taking a $bit\ sequence$ (sa) from su. Then comparing its hamming distance with the hamming distance of the $bit\ sequences$ in st. When a $bit\ sequence$ (sb) is found, such that the hamming distance between sa and sb is one, we replace sa by sb. Note that if many $bit\ sequences$ match this criterion, we employ the $bit\ sequence$ with the highest frequency to replace sa. Finally, we repeat this process for all the $bit\ sequences$ in su. We empirically searched for some combinations of M and N; while this search is not exhaustive, we found some combinations that significantly improve the compression ratio, shown in Section VI.

IV. Hardware and Software Implementation

This section describes our proposal to improve BNN performance using compressed kernels. First, we present an overview of our proposal. Next, we detail the software baseline used to evaluate the model. Then, we motivate the need for some hardware optimizations. Finally, we discuss the hardware implementation of our scheme.

A. Overview

One of the goals of this work is to employ the compressed 3x3 binary kernels during the evaluation of the BNN model. Our overall proposal work as follows. First, we compute the frequency of use for each $bit\ sequence$ in all the 3x3 binary kernels. Then, based on those frequencies, we create a Huffman Tree and assign an encoding to each of the $bit\ sequences$. Note that the previous steps are done offline and only once. Next, when evaluating a 3x3 kernel, we store the Huffman tree corresponding to that kernel in memory. After this, we start fetching the compressed $bit\ sequences$ and inputs from memory. Then, the $bit\ sequences$ are decompressed, and the kernel evaluation is performed. This process is repeated for all $bit\ sequences$ in the kernel. Nonetheless, compressing the kernels decreases the storage requirements; it adds an overhead to decode the $bit\ sequences$ resulting in a slowdown when implemented in software. However, the overhead can be mitigated with additional hardware support. We detail these issues in the following sections.

B. Software Baseline

When evaluating BNNs in modern CPUs, one of the most critical components is laying out the kernels in memory such that the number of bits loaded into the internal register of the CPU [5], [6] is maximized. In this regard, one of the most efficient methods is to pack bits from different channels together so that the number of bits packed together fills an entire register when loaded from memory [5]. For simplicity, we refer to this method as *channel packing*.

In *channel packing*, the group of bits packed from each channel must belong to the same position (i.e., 0,0), as shown in Figure 5. Also, when the number of channels is larger than the CPU register's size, the channels are split into several sub-groups such that they can be loaded independently into the CPU registers. Note that *channel packing* works best when the number of channels is divisible by the size of CPU registers; otherwise, padding is needed. Padding BNN kernels is challenging since the zero value represents a -1. Subsequently, when padding is done, the kernel computation must account for the extra -1. In this work, we employ *channel packing* to store the uncompressed 3x3 binary kernels, and it is done offline, as in [5]. Also, since the number of channels for all the kernels is a power of two, we do not employ padding.

Regarding the layout in memory of the compressed kernels, note that each of the encoded $bit\ sequence$ has a variable length, and thus we cannot perform channel packing offline. In this case, we store them consecutively in memory as a sequence of encoded words. As a result, during the kernel evaluation, once an encoded $bit\ sequence$ is decoded, we need to pack it into the CPU registers. Note that the size of a decoded $bit\ sequence$ is 9 bits, whereas the typical size of a CPU register is 32 bits or more; hence if packing is not done, the CPU registers are underutilized.

We implemented our compression scheme in software to evaluate its feasibility. Then, we compare the performance of this implementation against a baseline using uncompressed kernels. For the compression scheme, we implemented the simplified Huffman Tree described in Section III-B. In this case, we limit the size of each tree to four nodes and use four tables to store the decoded $bit\ sequence$. Then, we employ the first bits of each encoded $bit\ sequence$ to indicate the table where it belongs. Moreover, we use the remaining bits in the uncompressed $bit\ sequence$ to get the original from its corresponding table. After comparing these two implementations, the compressed kernel implementation is 1.47x slower than the baseline.

C. Hardware Support

The main reason for the slowdown of the software implementation of our scheme is the overhead of decoding and packing the $bit\ sequences$ at runtime. Hence, to mitigate this issue, we propose performing it in hardware. For this, we add a *decoding unit* to the load-store unit (LSU) of the CPU. Furthermore, we add two new instructions to manage this unit. The *decoding unit*, shown in Figure 6, is tailored to streaming and decoding the encoded *bit sequences*. As can be seen, it is composed of a *streaming unit* and a *packing unit*.

The *streaming unit* is employed to load a stream of encoded *bit sequences* from the main memory and decompress them. It works as follows, first, it uses the *stream address register* to store the base address of the *bit sequences* that will be fetched. Then, a request to load T bytes from the address in the *stream address register* is sent to the LSU decoder. Once the T bytes are fetched from memory, they are stored in the *input buffer*. After this, the decoding process starts by sending m-bits of data from the *input buffer* to the *stream parser*. Note that, we also send a new request to fetch more bytes while doing the decoding. In the *stream parser*, from this m-bits the first n bits are used to create the *node address* which identifies the node of the Huffman tree belonging to the current *bit sequence*. Note that since the length of the encoded *bit sequence* is variable, after finding the node, we need to find its length. It is done by reading it from the *length table*, which is addressed with the *node address*. Then, with the length information, we can get the remaining bits of the encoded *bit sequence* and store them in the *decoded address register*. Next, in the *decoder unit*, the uncompressed *bit sequences* are stored in a scratchpad memory (*uncompressed table*), which is partitioned into multiple banks. Then, we employ the *node address* and the value in the *decoded address register* to address the *uncompressed table* and get the uncompressed *bit sequence*. Finally, the uncompressed sequence is passed to the $packing\ unit$. We repeat the previous process until the whole stream of bits in the *input buffer* have been processed.

The $packing\ unit$ is in charge of *channel packing* each one of the *bit sequences* received from the $streaming\ unit$ into the *packing registers*. In this unit, after receiving a *bit sequence*, it is distributed into k register (i.e., 9) each of R bits (i.e., 128 bits). Note that R *bit sequences* are channel packed sequentially into k registers. Also, once R *bit sequences* are packed, a new set of k register is used to store the incoming *bit sequences*.

To use the *decoding unit*, the programmer first must configure it. For this, we add the instruction *lddu* (i.e., load decoder unit configuration). This instruction uses a pointer to a configuration structure to load its values in the *decoding unit*. Table III shows the main configuration values for the *decoding unit* which are stored in the main memory. Once the configuration values are loaded, the *decoding unit* is reset and it starts decoding the *bit sequences*.

For reading the packed and uncompressed *bit sequences* from the *decoding unit*, we add the instruction *ldps* (i.e., load packed bit sequence). To use this instruction, the programmer must specify the destination register to store the *bit sequence* being read. Note that this instruction will read the oldest *bit sequence* that was decoded in the *decoding unit*. Hence, the programmer is responsible for setting this unit before using *ldps*.

TABLE III
CONFIGURATION STRUCTURE.

Number of bit sequences
Compressed sequences pointer
Compressed sequences length
Huffman tree nodes

TABLE IV
CONFIGURATION PARAMETERS.

Parameter	Value
CPU	ARM A53
Main Memory	4 GB DDR4
Vector Registers	32 (128 bits)
Frequency	1 GHz
CPU L1 Cache	32KB
CPU L2 Cache	256 KB
Decoding Unit	
Max number of Nodes	4
Uncompressed table	1 KB
Register file	256 bytes
Input Buffer	256 bytes

To summarize, the *decoding unit* works as follows. First, a *lddu* instruction is executed to start its configuration and the decoding process. Then, in the background, the *decoding unit* fetches and decodes the compressed *bit sequences*. Later, when evaluating a 3x3 convolution, the programmer uses the *lddu* instruction to load the uncompressed and channel-packed *bit sequences* into the CPU registers. This process is repeated during the computation of a kernel until all the *bit sequences* are evaluated.

V. EVALUATION METHODOLOGY

We employ the Pytorch implementation of ReActNet and the ImageNet dataset as our baseline to evaluate the accuracy. Also, we use Python to compress and analyze the frequency of use for each bit sequence. For the performance evaluation, we use the daBnn [5] C++ framework to implement the BNN model. DaBnn employs *channel packing* for the kernels and inputs. Moreover, it provides several C++ functions to compute the most common layers found on BNNs. This library is highly optimized for the ARMv8 architecture and uses the NEON extensions.

As a hardware baseline, we use an ARM A53 CPU and employ Gem5 to implement and evaluate our hardware optimizations. To assess the software baseline with the proposed hardware extensions, we rewrote the assembly functions used by daBnn, such that we exploit them when computing the binary kernels. Also, to estimate the latencies of our hardware structures, we implement them in Verilog and use the Synopsys Design Compiler for synthesis. We modeled a 4GB DDR4 DRAM for the main memory. The main configuration parameters for the CPU and the *decoding unit* are shown in Table IV.

VI. RESULTS

This section presents the evaluation of our scheme implemented on an ARM CPU. We limit the Huffman tree to four

TABLE V
COMPRESSION RATIO OF *bit sequences* FOR THE 3X3 KERNEL IN EACH BASIC BLOCK.

Layer	Encoding	Clustering
Block 1	1.18	1.30
Block 2	1.22	1.30
Block 3	1.21	1.31
Block 4	1.21	1.32
Block 5	1.19	1.30
Block 6	1.20	1.33
Block 7	1.18	1.33
Block 8	1.20	1.32
Block 9	1.20	1.31
Block 10	1.18	1.32
Block 11	1.19	1.33
Block 12	1.25	1.36
Block 13	1.22	1.35

nodes. Also, we restrict the number of *bit sequences* stored on each node to 32, 64, 64, and 256. Hence, we use 6 bits to store the 32 most common sequences. Similarly, we use 8, 9, and 12 bits for the other nodes.

Table V shows the compression ratio by layer. The Encoding column refers to encoding the 3x3 kernels without removing the less frequent $bit\ sequences$, whereas the 256 most uncommon are removed for the Clustering column. For the Encoding, the compression ratio is between 1.18x and 1.25. The improvements came from employing only 6 bits to store the 32 most common *bit sequences*, which have an average frequency of use of 46%. Also, the rest of the *bit sequences* are stored using 8, 9, and 12 bits with a frequency of use of 24%, 23%, and 5%, respectively. After removing some of the least used *bit sequences*, the compression ratio is 1.32x on average. These improvements are due to increasing the frequency of the top 32 *bit sequences* to 65% on average. In this case, the frequency of use of the stored sequences using 12 bits is 0.6%, whereas, for 8 and 9, it is 25% and 8%, respectively. The overall BNN model is compressed by 1.2x.

Regarding the performance, we compare our compression scheme with hardware support with the software implementation of ReAcNet. In this case, we obtain a speedup of 1.35x. The primary source for this improvement is the reduced latency for loading the 3x3 kernels. Note that since the *decoding unit* fetches some of the *bit sequences* while decoding and evaluating others, the load's latency overlaps with the computations.

VII. RELATED WORK

Hardware acceleration for BNNs has been an active area of research in recent years. In this regards, there have been several works targeting CPUs [6], FPGAs [2], [7], [8], and ASICs [9]. FracBnn [2] proposes a FPGA accelerator for ReActNet. In addition, it improves performance by employing up to two bits for activation and increasing the level of sparsity in the kernels. Also, they binarize the first layer of the model, wheres it is quantized in this work. The work in [7] proposes an FPGA accelerator that combines the xnor and accumulation to improve performance. BNNsplit [8] is another accelerator that divides the evaluation of the BNN model across multiple FPGAs. Our proposal is different from these works, since we focus on compressing the kernels and we target mobile CPUs.

The work in [6] proposes a blocking algorithm to partition the binary kernels such that the utilization of CPU registers is more efficient. In [10], Huffman encoding and pruning are applied to models running on GPUs. However, our proposal employs a hybrid software/hardware approach.

VIII. CONCLUSION

This work presents a novel compressing scheme for binary neural networks. We show that in the binary kernels, some *bit sequences* occur more frequently than others. Aiming to exploit this observation, we encode each sequence with a simplified Huffman Tree. We propose a simple hardware extension to conventional CPUs to support this scheme, since implementing it on software degrades performance. After applying our compression scheme, the binary kernels are compressed by 1.32x on average, whereas the BNN model is compressed by 1.2x. Also, compared to the software baseline, our proposal achieves a speedup of 1.35x.

ACKNOWLEDGMENT

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00, and the ICREA Academia program.

REFERENCES

[1] Z. Liu, Z. Shen, M. Savvides, and K.-T. Cheng, "Reactnet: Towards precise binary neural network with generalized activation functions," in *European Conference on Computer Vision (ECCV)*, 2020.

[2] Y. Zhang, J. Pan, X. Liu, H. Chen, D. Chen, and Z. Zhang, "Fracbnn: Accurate and fpga-efficient binary neural networks with fractional activations," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 171–182.

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

[4] Z. Aspar, Z. Mohd Yusof, and I. Suleiman, "Parallel huffman decoder with an optimized look up table option on fpga," in *2000 TENCON Proceedings. Intelligent Systems and Technologies for the New Millennium (Cat. No.00CH37119)*, vol. 1, 2000, pp. 73–76 vol.1.

[5] J. Zhang, Y. Pan, T. Yao, H. Zhao, and T. Mei, "dabnn: A super fast inference framework for binary neural networks on arm devices," in *Proceedings of the 27th ACM international conference on multimedia*, 2019, pp. 2272–2275.

[6] A. Trusov, E. Limonova, D. Nikolaev, and V. V. Arlazarov, "Fast matrix multiplication for binary and ternary cnns on arm cpu," *arXiv preprint arXiv:2205.09120*, 2022.

[7] G. Du, B. Chen, Z. Li, Z. Tu, J. Zhou, S. Wang, Q. Zhao, Y. Yin, and X. Wang, "A bnn accelerator based on edge-skip-calculation strategy and consolidation compressed tree," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 3, pp. 1–20, 2022.

[8] G. Fiscaletti, M. Speziali, L. Stornaiuolo, M. D. Santambrogio, and D. Sciuto, "Bnnsplit: Binarized neural networks for embedded distributed fpga-based computing systems," in *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*, ser. DATE '20. San Jose, CA, USA: EDA Consortium, 2020, p. 975–978.

[9] M. Hosseini and T. Mohsenin, "Binary precision neural network manycore accelerator," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 17, no. 2, pp. 1–27, 2021.

[10] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.