IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 11 August 2023, accepted 8 September 2023, date of publication 19 September 2023, date of current version 3 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3317293

RESEARCH ARTICLE

Efficient and Compact Representations of Deep Neural Networks via Entropy Coding

GIOSUÈ CATALDO MARINÒ, FLAVIO FURIA, DARIO MALCHIODI, AND MARCO FRASCA
Department of Computer Science, University of Milan, 20133 Milan, Italy

Corresponding author: Marco Frasca (marco.frasca@unimi.it)

This work has been supported by the Italian MUR PRIN project ''Multicriteria data structures and algorithms: from compressed to learned indexes, and beyond'', under Grant 2017WR7SHH.

ABSTRACT Matrix operations are nowadays central in many Machine Learning techniques, including in particular Deep Neural Networks (DNNs), whose core of any inference is represented by a sequence of dot product operations. An increasingly emerging problem is how to efficiently engineer their storage and operations. In this article we propose two new lossless compression schemes for real-valued matrices, supporting efficient vector-matrix multiplications in the compressed format, and specifically suitable for DNNs compression. Exploiting several recent studies that use weight pruning and quantization techniques to reduce the complexity of DNN inference, our schemes are expressly designed to benefit from both, that is from input matrices characterized by low entropy. In particular, our solutions are able to take advantage from the depth of the model, and the deeper the model, the higher the efficiency. Moreover, we derived space upper bounds for both variants in terms of the source entropy. Experiments show that our tools favourably compare in terms of energy and space efficiency against state-of-the-art matrix compression approaches, including Compressed Linear Algebra (CLA) and Compressed Shared Elements Row (CSER), the latter explicitly proposed in the context of DNN compression.

INDEX TERMS Neural network compression, space-conscious data structures, weight pruning, weight quantization, source coding, sparse matrices.

I. INTRODUCTION

Deep neural networks (DNNs) achieved state-of-the-art performance in several real world applications [1], ranging from speech [2] and image [3] recognition, to self-driving cars [4] or playing complex games [5]. To achieve such notable results, DNN models have often been over-parameterized [6], which hampers their efficiency due to both an increase of their memory demand, and of the complexity of matrix multiplications required in their inference (forward) step, typically in most layers of the network. Note that tensor products, characterizing convolutional layers, fall in this category, since they can be suitably reshaped into matrix products. As a well-known example, the *VGG19* network [7], made up by 16 convolutional and 3 fully-connected (FC) layers, requires 19 matrix operations for a single inference.

The associate editor coordinating the review of this manuscript and approving it for publication was Mostafa M. Fouda.

As a consequence, lowering the algorithmic complexity of this operation and increasing the efficiency of its storage is receiving lots of attention [8], [9], and several challenges arose w.r.t. this target. Indeed, attempting to reduce the space required to store the matrix, by leveraging some suitable compressed formats, is not enough to enable efficient and fast matrix-vector operations, if the compressed format does not support it. To achieve this goal, instead, a frequent approach is that of complying with two tasks: *lossy* compression of a DNN weight layer matrix to lower the complexity, and its subsequent *lossless compact representation* supporting the matrix-vector multiplication. The first task consists in altering or approximating the information contained in the matrix to attain a size reduction, e.g., via low precision storage, pruning/sparsification (that is, increasing the number of zero-valued entries in the matrix), or quantization (reduction of the number of distinct values). The second task, instead, does not alter the matrix information, and it can achieve space

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

reduction by reorganizing this information into appropriate data structures, e.g., those used by the CSR or CSC formats for sparse matrices. However, in order to be more efficient, the input matrix should exhibit specific characteristics about the probability mass distribution of its elements, which are indeed induced by the prior lossy compression.

Many studies have already shown how most state-of-the-art DNNs in various contexts can often undergo to lossy compression to the detriment of negligible accuracy drops [10], [11], [12], which encouraged researchers to explore the field of DNNs compression. Among the different approaches proposed, a plethora of works focused on lossy compression, see for instance [11], [13]. In particular, pruning and quantization, which induce weight matrices characterized by low entropy, allow to achieve higher compression ratios when combined with suitable lossless formats. We will get back to such methods through the paper.

However, most existing approaches are not bound to specifically designed storage formats able to maximally leverage their peculiar characteristics, or they do not support the vector-matrix multiplication in the compressed format, necessary for the network inference. To overcome such limitations, in this work we propose two lossless matrix storage formats tailored to be coupled with weight pruning and quantization methods, and which allows the network inference without re-expanding the network. We adhere in this study to the idea of designing lossless compression formats under the implicit assumption that the entropy of the distribution of the matrix elements is low. Indeed, being the minimal bit length of a data representation bounded by the entropy of its distribution [14], it is reasonable to aim at storing low-entropy matrices via data structures that do not require high memory and computational resources. This task, however, is very challenging, and only a few works describe lossless compression formats allowing to perform efficient operations on the matrices without their re-expansion [15], [16], [17], [18], [19]. Traditional lossless compression techniques, such as Gzip, are not applicable because decompression is too slow, while lightweight methods like Snappy or LZ4 achieve only modest compression ratios on matrices [16]. In addition, they usually require the full-matrix decompression in order to perform linear algebra operations, thus resulting in no space reduction in the computation phase.

Recently, new lossless compression schemes for matrices in the ML context have been proposed, with the twofold advantage of saving space and speeding up linear algebra operations [15], [16]. This approach, named *Compressed Linear Algebra* (CLA), leverages an appropriate grouping of matrix columns to be compressed together through a suitable scheme chosen, via an approximate compression planning, among a set of predefined simple compression schemes. CLA allows to efficiently execute matrix operations in the compressed format, although it requires the columns to present regularities and repeated patterns in order to attain high compression ratios. Another recent study proposed an extension of the Compressed Sparse Row (CSR) format, named *Compressed Shared Elements Row* (CSER), to matrices which are quantized as well as sparse [18]. This method, specifically proposed in the context of DNN compression, is able to benefit from repeated and consecutive elements on the same row, both to save space and effectively reduce the number of multiplications by leveraging the distributive property of multiplication. However, it requires 0 to be the most frequent element in the matrix.

In this work we propose two new lossless compression formats, called *Huffman Address Map* (HAM) and *sparse Huffman Address Map* (sHAM), specifically tailored to large and low-entropy matrices, which do not need any further assumptions on the distribution of the elements on the rows or on the columns of the matrix, as well as on the frequency of elements. These formats are based upon Huffman coding, address maps, and compressed sparse representations. Our main contributions can be summarized as follows:

1) we introduce new efficient lossless compression data structures, supporting matrix-vector multiplication in the compressed format, which leverage low-entropy characteristics of the input matrices in order to reduce storage space and energy costs required to execute matrix operations;
2) unlike most existing methods, our formats do not necessarily need the input matrix to be sparse, since sparsity can be considered as a subclass of the more general family of low-entropic distributions [20];
3) we provide a detailed analysis of space requirements (including the corresponding upper bounds), as well as of the algorithmic complexity and energy consumption of performing a matrix-vector multiplication;
4) a multicriteria design regulating the space/time trade-off can be exploited via dedicated features of the proposed compression schemes;
5) the proposed data structures are validated against state-of-the-art compression schemes, including CLA and CSER, in terms of memory requirement, energy and time efficiency for computing the matrix-vector multiplication. Three different scenarios are considered: (i) compressing synthetic matrices; (ii) compressing benchmark matrices; (iii) compressing two publicly available DNNs.

Our experiments show that when the input matrix is dense (or just slightly sparse) and quantized, HAM achieves the highest compression ratio and energy cost reduction in most experiments, and competitive results in terms of matrix-vector multiplication execution time. For a fair comparison with CLA, we implemented the HAM and sHAM dot procedures by leveraging multi-threading execution and pre-compiled code. When the input matrix is sparser, HAM is still among the best choices in terms of per-element storage requirements (even till 90% of sparsity on DNNs data), but its execution time becomes less competitive. Indeed, methods

specifically designed for sparse data (like CSER) are able to speed up more with an increase of the sparsity level. In such cases, our HAM variant for sparse data, sHAM, becomes more efficient, the best option along with CSER method in the majority of cases, especially for extreme sparsity levels. Our source code is publicly available at https://github.com/AnacletoLAB/sHAM.

The paper is organized as follows: Sec. II describes state-of-the-art methodologies for: (i) post-hoc processing of the learnt weight matrices of a DNN in order to reduce their entropy (Sec. II-B), and (ii) low-entropy matrices representation formats able to reduce the resource demand in terms of RAM and the computational burden of linear algebra operations (Sec. II-C). Sections III-A and III-B introduce two novel representation formats which we propose, namely HAM and sHAM, whereas a multi-threaded version of their dot product procedures and some clues about their potential extension via GPU devices are respectively illustrated in Sects. III-C and III-D. The energy consumption required by the compared compressed formats to execute matrix-vector multiplication is analyzed in Sec. III-E. Finally, the experimental comparison among the compression formats is described and discussed in Sec. IV in terms of memory requirements, as well as energy and time complexity of their dot product procedures. Some concluding considerations end the paper.

II. BACKGROUND AND RELATED WORK

The aim of this section is 1) to sketch existing DNN compression approaches, 2) to briefly mention two main lossy techniques, weight pruning and quantization, used in the literature to reduce the DNN layer weights entropy, and 3) to describe some state-of-the-art lossless matrix storage formats along with their per element storage requirements.

A. COMPRESSION OF DEEP NEURAL NETWORKS

The research advance in Deep Learning of the last twenty years has led to increasingly sophisticated and resource-hungry architectures, posing the problem of how to use available computing resources more sparingly. Accordingly, the interest of the scientific community towards the compression of deep neural networks has grown enormously in the last decade. In principle, since the DNN inference is performed through a sequence of vector-matrix multiplications, all the general purpose methods to compress matrices can be exploited for this task. Nevertheless, a plethora of strategies specific for DNNs have also been introduced. *Structural compression*, for instance, refers to a set of 'lossy' strategies which focus on detecting and removing less relevant components from the architecture, like hidden units in FC layers, or filters and channels in convolutional layers, at the expense of contained accuracy drops [21], [22], [23]. Despite their state-of-the-art performance in several application domains, such approaches are limited by the fact that no exact and tractable paradigm to estimate an optimally or near-optimally performing neural network structure is known [24], and much effort is to be dedicated to find the best topological design. Indeed, most works in this domain propose manual or automatic heuristics [25], which typically ties their success also to the individual expertise in choosing most promising topologies. The same limitation affects also the so-called *knowledge distillation*, where an initial "master", large model is trained, with the idea to get rid of it after a smaller "student" model is trained from the master outputs [26], [27]. Again, the bottleneck is represented by the absence of clear and efficient paradigms to choose the optimal architecture for the student network. To overcome this limitation, several studies focused on reducing the DNN resource usage by keeping the network topology unaltered. In this sense, low-rank factorization of the weight matrix of each network layer, which uncovers the latent compact structure of the weight matrix, can be seen as an hybrid between structure altering methods and the remaining ones, since it preserves the input and out dimension of the layer, while introducing an hidden lower-dimensional latent layer, allowing to reduce the total number of parameters [28], [29]. Among lossy structure-preserving approaches, weight pruning and quantization are the most widely adopted approaches. The former consists in removing neuronal connections likely to be irrelevant for the overall network behavior [30], usually performed by removing those having lowest absolute magnitude [31], in a one-shot or an iterative scenario [32], [33]. The latter, instead, refers to the process of reducing the number of bits used to represent individual wights: one bit per weight in the extreme case of *binarization* [34], and < 32 bits for a "smoother" quantization, where the most common approach is to group weights by magnitude into k groups, and use a unique representative weight for each group (see Section II-B3 for more details). Both approaches are frequently followed by a moderate weight re-training, and they can also be jointly applied to further improve the compression rate [12]. In this direction, some interesting studies are investigating how to directly embed multiple lossy compression techniques in the framework devoted to optimizing the model parameters, that is during model training, rather than after this phase [35]. Unfortunately, many of the methods described are not accompanied by appropriate formats to store the resulting network, so that the network inference can be done directly in the compressed format. Among the very few methods able to do that, it is worth mentioning Index Map and Compressed Shared Elements Row Format, that we describe in detail in Section II-C. In light of the above-mentioned limitations, we first focus on the structure-preserving compression of existing pre-trained models, exploiting the huge effort already done during the architecture optimization, and at the same time attempting at mitigating their sub-optimality. In particular, we propose two novel lossless formats for storing DNN matrix weights and to efficiently perform the vector-matrix multiplication, able to reduce the resource demand of existing networks under the specific assumption that

their weight matrices exhibit low-entropy features. In the following, we first describe the methodologies we adopted to reduce the entropy of layer weight matrices, then we introduce our DNN lossless storage formats, along with some state-of-the-art strategies in the same domain, that we included in the theoretical and experimental evaluation.

B. INDUCING LOW-ENTROPY IN DNN LAYER CONNECTIONS

To ensure the generality of our approach, we focus on techniques which operate on a pretrained network, without altering its topology. Subsequently, the obtained weight matrices will be stored using the compact formats described in Section II-C, directly supporting the matrix multiplication without re-expansion.

1) PRELIMINARY DEFINITIONS

Given a matrix $\boldsymbol{W}^{\text{o}} \in \mathbb{R}^{n\times m}$, whose distinct entries have relative frequencies denoted by $(p_1, \ldots, p_k)$, the *entropy* of this matrix/source is $\mathcal{H}_{\boldsymbol{W}^{\text{o}}} = -\sum_{i=1}^{k} p_i \log p_i$. The entropy assumes its maximum value when all symbols have the same probability/relative frequency, that is $p_1 = p_2 = \ldots = p_k = 1/k$. In such a case, $\mathcal{H}_{\boldsymbol{W}^{\text{o}}} = -\log k$, and accordingly, for a given matrix $\boldsymbol{W}^{\text{o}}$ with k distinct entries, we can define its *normalized entropy* as $\sum_{i=1}^{k} p_i \log p_i / \log k \in [0, 1]$. When clear from the context, in the following we will omit the subscript and denote the entropy simply by $\mathcal{H}$.

We will distinguish two experimental settings: 1) the input matrix already shows regularities that induce a relatively low normalized entropy; 2) the input matrix needs to undergo specific *lossy* techniques (i.e., techniques that partially loose information) that alter its content so as to induce low entropy characteristics. Here, the distinction between low and high entropy sources is not crisp, in the sense that we are able to reduce the source entropy via lossy techniques (e.g., sparsification), and the efficiency of the subsequent *lossless* formats (no information is lost) adopted to represent the input matrix is strictly related to the matrix entropy. In case 2), which is for instance the case of layer connection matrix in a DNNs, we can denote by $\boldsymbol{W}$ the low-entropy matrix obtained from $\boldsymbol{W}^{\text{o}}$ after applying lossy compression methods. Symbols w^{o} and w will denote generic entries of $\boldsymbol{W}^{\text{o}}$ and $\boldsymbol{W}$, respectively. We assume that each element of $\boldsymbol{W}^{\text{o}}$ is stored in one memory word, whose size is b bits. The *per-element storage requirement* of a matrix $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ is defined as $\psi = \text{size}(\boldsymbol{X})/(nm)$ bits, where $\text{size}(\cdot)$ is an operator returning the size of its argument, measured in bits. For instance, the per-element storage requirement of the (dense) matrix $\boldsymbol{W}^{\text{o}}$ is b. Italic boldface font is used for matrices and vectors (e.g., $\boldsymbol{X}$ and $\boldsymbol{x}$), while $\boldsymbol{X}^{\boldsymbol{I}}$ denotes the submatrix obtained from $\boldsymbol{X}$ by only considering the columns whose indices belong to the set $\boldsymbol{I}$. According to the context, $|\cdot|$ is an operator either denoting absolute value or cardinality, returning in the latter case the length of a string or the number of elements in a vector. The log function always refers to the binary logarithm. Assuming a given lossless matrix storage format F isable to exploit the characteristics of $\boldsymbol{W}$ to reduce its bit-size, the *compression ratio* obtained by F is defined as $\text{size}(\boldsymbol{W})/\text{size}(F(\boldsymbol{W}))$ in setting 1), and as $\text{size}(\boldsymbol{W}^{\text{o}})/\text{size}(F(\boldsymbol{W}))$ in setting 2).

Finally, $s \in [0, 1]$ denotes the *ratio of non-zero elements* in a matrix $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ (number of non-zero entries divided by nm), and $1 - s$ is its *sparsity level*.

2) WEIGHT PRUNING

Weight pruning is one of the lossy techniques used in the literature to reduce the complexity of a matrix and also its entropy, and it consists in setting to 0 a fixed fraction of the matrix entries. This, in turn, reduces the entropy of the matrix, since one symbol (0) becomes more frequent than the other ones. There are many approaches to weight pruning (WP) in DNNs, but by WP we usually mean "magnitude-based" WP, that is removing the connection weights that are "small" in absolute value [31]. Operationally speaking, after having fixed the p-th empirical percentile w_p of the entries of $\boldsymbol{W}^{\text{o}}$, $\boldsymbol{W}$ is derived by setting $w = w^{\text{o}}$ if $|w| > w_p$, 0 otherwise. WP is often followed by a fine tuning of the remaining connections, involving a 'short' retraining of the DNN weights which have not been set to zero.

3) QUANTIZATION

Weight quantization is another strategy, adopted in particular for DNNs, to compress matrices and reduce their entropy. This strategy consists in reducing the space needed to store individual weights. . Quantization can be achieved in different ways, including the weight sharing (WS) approach, which expressly casts connection weights into k categories, and substitutes all weights in i-th category with a representative weight c_i, for $1 \leq i \leq k$. The way weights are partitioned distinguishes the different WS based quantization approaches. In particular, in the context of DNN compression, weight categories can be obtained using clustering techniques [36], a probabilistic partitioning preserving the original weights in expectation [10], uniform partitioning [37], [38], or by jointly optimizing the quantization distortion and the entropy of the resulting distribution of representatives [39], [40]. Also in this case, a fine tuning of the representatives is typically performed [10], [36], [41]. Here we adopt the clustering-based WS [36], which has been shown among the top performing methods in this category [12].

In the context of deep neural network compression, weight pruning and quantization can be applied in sequence, even in an iterated fashion, and, surprisingly, it has been shown that in some practical applications this does not deteriorate the model performance, while yielding a higher space reduction w.r.t. the sole application of pruning or quantization [12], [38].

C. LOSSLESS MATRIX COMPRESSION FORMATS

Hereafter, we refer to an input matrix $\boldsymbol{W}$ assuming it already shows characteristics like sparsity and/or quantization of

its values, and we focus on how lossless formats can allow its compression and the execution of vector-matrix products without re-expanding the matrix itself. In particular, we consider the CSC format as a baseline, and two state-of-the-art formats specifically proposed for DNN compression, Index Map and CSER. We selected CSC instead of the CSR format because it is more suitable for the layer shapes in the forward steps of DNNs, since non-zero values are stored by columns; moreover, the two formats would provide the same results up to simple transposing operations. In the following, we describe each format in detail, also analysing their performance in terms of per-elementstorage requirement, leaving their comparisons in terms of energy and time complexity to the next sections.

1) COMPRESSED SPARSE COLUMN

The *Compressed Sparse Column* (CSC) format [42] is a well established standard for storing sparse matrices. It is composed of 3 arrays:

- **nz**, containing the nonzero values, listed by columns;
- **ri**, containing the row indices of elements in **nz**;
- **cb**, where cb_i is the number of non-zero elements in column i.

Example 1: The CSC representation of the matrix

$$\boldsymbol{W} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

is $\mathbf{nz} = (1, 1, 1, 3, 1, 5, 5)$*,* $\mathbf{ri} = (0, 2, 1, 2, 0, 2, 4)$*, and* $\mathbf{cb} = (2, 2, 1, 0, 2)$.

Recalling that $s \in [0, 1]$ is the ratio on non-zero elements in $\boldsymbol{W}$, storing the CSC representation requires the use of $snmb$ bits for **nz**, $snm \log n$ bits for **ri**, and $m \log n$ bits for **cb**, leading to a per-element storage requirement

$$\psi_{\text{CSC}} = \frac{snm(b + \log n) + m \log n}{nm} = s(b + \log n)) + \frac{\log n}{n}. \quad (1)$$

From now on we assume that $\log x$ bits[1] are sufficient to store elements whose value is at most equal to x. In the theoretical discussion we prefer to use this notation, whereas in practice, when we run our simulations and experiments, $\log x$ is substituted with the smallest value among 8, 16 and 32 which is greater than the logarithm itself.

The dot product $\boldsymbol{x}^T \boldsymbol{W}$ can be computed when $\boldsymbol{W}$ is in CSC format by using a custom procedure, having a time complexity of $\mathcal{O}(snm)$ (see Algorithm S1 of Supplementary Material), and which can be sped up through parallel computing [42]. Using b bits for each element of **nz** constitutes the main limitation of the CSC representation.

[1] To be precise, we should write $\lceil \log x \rceil$, but throughout the paper we prefer to omit the ceiling operator to simplify the notation.

2) INDEX MAP

Index Map (IM) is a lossless format specifically designed to store quantized matrices, and proposed to compress fully-connected layers in DNNs [36]. It stores the representative weights in a vector $\boldsymbol{c}$, and then it populates an index matrix $\boldsymbol{\Pi}$, having the same dimension of $\boldsymbol{W}$, so that $\pi_{ij} = r$ when w_{ij} is associated with the representative c_r. Recalling that we denote by k the number of weight representatives, each entry of $\boldsymbol{\Pi}$ requires $\log k$ bits (since its value can be at most k); accordingly, we have

$$\psi_{\text{IM}} = \log k + \frac{kb}{nm}. \quad (2)$$

For low values of k, the first term is more relevant, whereas when k increases, the second term grows linearly with it and the format becomes inefficient. For instance, when $k = 16$, $b = 32$ and $n = m = 10^2$, we obtain $\psi_{IM} = 4.0512$ which means a compression ratio of around $7.9\times$ with regard to the space required by $\boldsymbol{W}$, but for $k = 256$, we have $\psi_{IM} = 8.8192$ and a compression ratio of around $3.6\times$. This comes at the price of two memory accesses in order to retrieve a weight to perform the matrix-vector product (Algorithm S2).

3) COMPRESSED SHARED ELEMENTS ROW FORMAT

Compressed Shared Elements Row (CSER) is a recent format, proposed in the context of DNN compression, to efficiently representlow-entropy matrices and execute the matrix-vector product directly in the compressed format [18]. It expressly exploits the assumption that few distinct values appear in the entire matrix, which is the case of quantized matrix. This format improves w.r.t. CSC because the latter repeatedly stores these distinct values, thus inducing high redundancies, whereas CSER stores each distinct element only once. The structures used by this format are described here below.

- $\boldsymbol{\Omega}$, vector containing the distinct elements of $\boldsymbol{W}$.
- **colI**, vector containing the column indices (starting from 0) of the elements of the matrix in row order, and, in each row, those relative to symbols in top-frequency order (excluding 0, which must be the most frequent symbol).
- $\boldsymbol{\Omega}$**I**, vector denoting the position (from 0) in $\boldsymbol{\Omega}$ of the distinct elements of $\boldsymbol{\Omega}$ the **colI** indices refer to. For each row, when an element occurs more than once, it is reported just one time. Therefore, for each row we have at most $|\boldsymbol{\Omega}| - 1$ entries in $\boldsymbol{\Omega}$**I**.
- ΩPtr, vector storing pointers that signal when, on each row, the positions in **colI** of the next new element of $\boldsymbol{\Omega}$ start, with the requirement that the last element of this vector be set to the first index outside **colI**.
- **rowPtr**, vector storing pointers that signal when a new row starts; **rowPtr** points to entries in ΩPtr, with a trailing element pointing to the last position of the same vector.

Example 2: The CSER representation of the matrix introduced in Example 1 is $\boldsymbol{\Omega} = (0, 1, 3, 5)$*,* $\mathbf{colI} = (0, 2, 1, 0, 4, 1, 4)$*,* $\boldsymbol{\Omega}\mathbf{I} = (1, 1, 1, 3, 2, 3)$*,* $\Omega Ptr = (0, 2, 3, 4, 5, 6, 7)$*,*

and **rowPtr** $= (0, 1, 2, 5, 5, 6)$. *The value* 5 *in* **rowPtr** *is repeated twice to allow detecting empty rows.*

Clearly, this format benefits from the presence of identical consecutive elements (interleaved by 0, which is not considered in the representation). Assuming that $\bar{k} \leq k$ is the average number of distinct elements per row, to store the CSER structures we need: (i) kb bits for $\boldsymbol{\Omega}$; (ii) $snm \log m$ bits for **colI**; (iii) $n\bar{k} \log k$ for $\boldsymbol{\Omega}\mathbf{I}$; (iv) $(n\bar{k}+1) \log snm$ bits for ΩPtr; (v) $(n+1) \log n\bar{k}$ bits for **rowPtr**. Overall we have the following per-element average number of bits:

$$\psi_{\text{CSER}} = s \log m + \frac{\bar{k}}{m} \left(\frac{\log n\bar{k}}{\bar{k}} + \log k + \log snm \right) + \frac{kb + \log snm + \log n\bar{k}}{nm}, \quad (3)$$

where the first term, which is the heaviest one, linearly increases with the fraction of non-zero elements in $\boldsymbol{W}$, the second term emphasizes the fact that this structure is efficient when few distinct elements appearin average on each row, and the last one is the least relevant, as long as the number of distinct elements k in $\boldsymbol{W}$ is much smaller than nm.

III. METHODS

We are now ready to introduce our novel lossless matrix compression formats.

A. HUFFMAN ADDRESS MAP COMPRESSION

This is the first novel method we propose here. It is based on the idea that the weight source might exhibit features that can be effectively compressed via Huffman coding after applying pruning and quantization to $\boldsymbol{W}^{\text{o}}$. This technique, that we name *Huffman Address Map compression* (HAM), exploits Huffman coding and address map logic to provide a lossless compression of $\boldsymbol{W}$. Here below we provide its description and derive its space complexity upper bound. Address maps organize the matrix entries as a row- or column-order based sequence of bits, in which 0 identifies any null entry, and each remaining element z is represented by a binary string $a(z)$ encoding its address.

Example 3: The bit sequence corresponding to the column-order based address map for the matrix $\boldsymbol{W}$ *of Example 1 is*

$$a(1)0\ a(1)000\ a(1)a(2)00a(1)00000000000a(3)0a(3).$$

In order to achieve efficiency, it is necessary to rely on compact representations for addresses. Taking inspiration from [36], we implement $a(z)$ via the corresponding Huffman coding of z, in view of its well-known properties: it is instantaneous, uniquely decodable and it has a near-optimal compression rate [43]. More precisely, let $Z = (z_1, \ldots, z_k)$ be a source of symbols, whose corresponding probabilities are $(p_1, \ldots, p_k)$, and $\mathcal{H}_Z = -\sum_{i=1}^{k} p_i \log p_i$ be the source entropy (see Section II-B1), which, by Shannon's source coding theorem, corresponds to the minimal average number of bits per symbol necessary to represent Z [14]. Moreover, let's denote by H_Z a Huffman code for this source, and by $H_Z(z)$ the codeword of a symbol z (abuse of notation). Finally, let $\overline{H}_Z := \sum_{i=1}^{k} p_i |H_Z(z_i)|$ be the average number of codeword bits per symbol used by H_Z. It can be shown that $\mathcal{H}_Z \leq |\overline{H}_Z| \leq \mathcal{H}_Z + 1$ [44]. To get uniquely decodable strings we also include zeroes in the source symbols.

In our setting, Z is composed of the distinct weights in the matrix $\boldsymbol{W}$. We denote by $HAM(\boldsymbol{W})$ the bitstream resulting by the column-order Huffman encode of entries in $\boldsymbol{W}$, and split the former into $N = \lceil |HAM(\boldsymbol{W})|/b \rceil$ memory words, in turn denoted as $HAM(\boldsymbol{W})_1, \ldots, HAM(\boldsymbol{W})_N$ and represented as an array $\mathcal{C}_{HAM}(\boldsymbol{W})$ of N unsigned integers. Zero-padding is added to the last word when $|HAM(\boldsymbol{W})|$ is not a multiple of b. We adopt the canonical variant of Huffman codes (CHCs), which allows fast decode and efficient use of memory, since it does not need to store pointers and tree structures. In particular, we leverage the implementation presented in [45], which we briefly report in the following. This selection is also supported by experiments conducted to evaluate other implementation variants of CHCs (Section IV-B). The multiplication of a matrix in the compressed format is based only on the decode phase, thus we need to store exclusively the Huffman code structures involved in this operation. For this reason, here we omit the description of the encode process. In particular, the following structures are used to decode codewords from the compressed bitstream, where $l_{\max}$ is the length of the longest codeword.

- **symbols**, vector of length k, where symbols$[i] = z_i$, for $1 \leq i \leq k$.
- *first_symbol*, vector indexed by a codeword length l, where first_symbol$[l]$ is the index (in **symbols**) of the symbol having the first codeword of length l.
- *first_code_l*, vector indexed by a codeword length l, where first_code_l$[l]$ contains the integer value of the codeword associated to first_symbol$[l]$, possibly zero-padded on the right to ensure a length equal to $l_{\max}$. The algorithm requires that the position $l_{\max} + 1$ of this structure contains a sentinel, initialized with $2^{l_{\max}}$.

The "canonical" property imposes integers associated to codewords of the same length to be consecutive, thus allowing to store just the first codeword of a given length in the array *first_code_l*. Moreover, in this variant symbols need to be sorted in non-increasing probability order (for further details see [45]), and the bitstream is accessed by chunks of length $l_{\max}$, contained in a bit array **buff**. The decoding steps are represented at lines 5-10 of Algorithm 1. In order to correctly decode the next symbol, it is necessary to know the length l of the next codeword contained in **buff**. This can be achieved by scanning *first_code_l* to find the first l satisfying

$$\text{first_code_l}[l] \leq \text{dec}(\textbf{buff}) < \text{first_code_l}[l+1], \quad (4)$$

where dec(**buff**) is the decimal value of the binary string **buff**. Since the search of l can be time consuming, an efficient variant described in the same paper exploits a direct lookup table of size $2^{l_{\max}}$, addressed directly by dec(**buff**) to get the corresponding l. The entries in this table have value at most $l_{\max}$, hence requiring $\log l_{\max} \leq \log k$ bits per entry. We elect

TABLE 1. Example of a CHC having 7 codewords such that $l_{\max}=5$, together with the corresponding decode structures, using a hybrid lookup table $\mathcal{T}$ with $t=\lceil \log l_{\max} \rceil = 3$. An asterisk denotes an entry not directly corresponding to a codeword length, and the need of a scan in *first_code_l* to find it, starting at the position preceding the asterisk.

l	first_symbol	first_code_l
0	0	0
1	0	0
2	1	16
3	1	16
4	4	28
5	5	30
6	–	32

symbols	*codeword*
0	0
5	100
2	101
4	110
1	1110
3	11110
6	11111

Entry	$\mathcal{T}$
0	1
1	1
2	1
3	1
4	3
5	3
6	3
7	4∗

as final CHC an hybrid variant using a partial lookup table $\mathcal{T}$ having 2^t entries, with $t < l_{\max}$, which is referred using the first t bits of **buff**, and whose entries might contain a codeword length or the initial position in *first_code_l* from which starting the search as described above (see Eq. 4). Table 1 shows an example of a canonical Huffman code with hybrid lookup table. In [46], this solution has been shown to be a better compromise in practice between search speed-up and extra space complexity, even for very small values of t. This is confirmed by our experiments, as discussed in Section IV. To store the array *first_symbol* we need $l_{\max} \log k$ bits, which is upper bounded by $(k-1)\log k$, since $k-1$ is the maximum depth of the involved Huffman tree. The array *first_code_l* requires $(l_{\max}+1)l_{\max}$ bits, since we have $l_{\max}+1$ integers (including the above-mentioned sentinel), each taking $l_{\max}$ bits, which is upper bounded by $k(k-1)$. By fixing $t=\log l_{\max}$ (see Section IV-B for the rationale of this choice), the lookup table requires $l_{\max} \log k \leq (k-1)\log k$ bits. Moreover, this way the table size only linearly grows with $l_{\max}$, and accordingly, in the worst case, with k. This is a loose upper bound, and a tighter bound can be derived by knowing the minimum probability of a symbol (see Section III-F for details). Finally, we need kb bits to store the array **symbols**, leading to an overall overhead for the Huffman code of at most $B_k := k(k+2\log k + b - 1) - 2\log k$ bits. For the sake of completeness, it is worth noting that, when needed, the overhead can be still reduced at the expense of decoding speed, as there are methods storing a Huffman code for n symbols using at most $\lceil 10.75n \rceil - 3$ bits [47].

We are now ready to theoretically characterize the per-element storage requirements of the HAM format.

Fact 1 (HAM Worst Case): If $\boldsymbol{W}$ *is dense and it does not contain repeated entries,*

$$\psi_{HAM} \leq 3\log nm + nm + b - \frac{2\log nm}{nm}.$$

Proof: By hypothesis each of the nm symbols of $\boldsymbol{W}$ appear exactly once, thus $\mathcal{H} = \log nm$ and the corresponding Huffman code H_W has an average codeword length upper bounded by $1+\log nm$. Accordingly, at most $nm(1+\log nm)$ bits are needed for the bitstream $|HAM(\boldsymbol{W})|$. Considered that the decoding arrays need $nm(nm+2\log nm+b-1)-2\log nm$ bits, the thesis follows by summing up bitstream and Huffman overhead, and dividing by the number of elements nm. □

Fact 1 shows that alternative representations should be used when the matrix is dense and nearly composed of all distinct elements. Indeed, in this case ψ_{HAM} is always greater than b, the number of bits used to store each element of $\boldsymbol{W}$. On the other hand, in the next corollary we show that the HAM format becomes much more convenient when a relatively small number k of distinct values are contained in $\boldsymbol{W}$, as customary with quantized matrices (see Sect. II-B).

Corollary 1: If $\boldsymbol{W}$ *is dense and composed of* $k < nm$ *distinct values (including* 0*),*

$$\psi_{HAM} \leq 1 + \log k + \frac{B_k}{nm}. \tag{5}$$

Proof: In the worst case, all symbols are equally probable, the source entropy is $\mathcal{H} = \log k$, and the length of $|HAM(\boldsymbol{W})|$ is at most $nm(1+\log k)$ bits. Then, by adding the CHC overhead of B_k bits and dividing by nm, the thesis follows. □

As expected, the smaller k, the higher the compression. On the contrary, when k increases, the last term becomes bigger and dominates the overall bound. As an example, if we set $b=32$, and $nm=10^4$, from Corollary 1 all the values $k < 448$ yield $\psi_{HAM} < b$; when $nm = 10^5$, the same result is obtained for all $k < 1408$. This is quite impressive, considering that, in most real-world experiments, k has a magnitude of the order of hundreds, and we are assuming that the longest possible average codeword length is obtained.

1) DOT PRODUCT

The procedure $\texttt{Dot}_{HAM}$ (Algorithm 1) shows how the dot product $\boldsymbol{x}^T\boldsymbol{W}$ can be computed when $\boldsymbol{W}$ is stored using the HAM format. The dot product is realized in the outer loop (lines 2–18), where each iteration examines a column of $\boldsymbol{W}$. The inner loop (lines 4–15) computes the dot product between the current column (indexed by *col*) and $\boldsymbol{x}$. The variable $l_{\max}$ (line 5) contains at each step the first $l_{\max}$ bits not yet read from the bitstream. The procedure GetBuff also returns the next starting point in the bitstream (start) to be used to read the next nbits bits. The procedure GetEntry then accesses the partial lookup table $\mathcal{T}$, indexed by the first t bits of **buff**, and

Algorithm 1 Dot Procedure for Canonical Hybrid HAM Representation

Procedure Dot$_{HAM}$

```
Input:
– compressed array C_HAM(W)
– first_code_l array
– first_symbol array
– symbols array
– the partial lookup table T
– maximum codeword length l_max
– n, m ∈ ℕ, number of rows and columns of W
– vector x ∈ ℝ^{n×1}
begin algorithm
 1: Initialize: out ←zeros(n), col← 1, start ← 1
              nbits ← l_max, t ← ⌈log l_max⌉
 2: while col ≤ m do
 3:   sum ← 0,  row ← 1
 4:   while row ≤ n do
 5:     buff, start ← GetBuff(C_HAM(W), start, nbits)
 6:     l ← GetEntry(T, buff, t)
 7:     l ← SearchLength(l_max, first_code_l, l)
 8:     nbits ← l_max − l
 9:     oset ← (l_max −first_code_l[l]) ≫ (l_max − l)
10:     val ← symbols[first_symbol[l] + oset]
11:     if val ≠ 0 then
12:       sum ← sum + x[row] ∗ val
13:     end if
14:     row ← row + 1
15:   end while
16:   out[col] ← sum
17:   col ← col + 1
18: end while
end algorithm
Output: out = x^T W.
```

provides the next codeword length l. At line 7, in case the entry of $\mathcal{T}$ is not directly the length of the next codeword (remind that $\mathcal{T}$ is partial, see Table 1), l is used as starting position to operate a binary search in the vector *first_code_l*. Once the length of the next codeword is obtained, at line 8 the number of bits to be read in the next iteration is updated, and the offset oset to access the array **symbols** is computed (line 9). Here $\gg$ denotes the *right-shift* operator. Finally, the corresponding weight val is computed (line 10), and used to update the partial sum of the current column if val is not zero (lines $11-13$). Line 16 writes the result in the output vector **out**.

As previously stated, this procedure does not increase the order of memory requirements described by Corollary 1, since only one weight at a time is decoded and kept in memory (precisely, in the val variable). On the other side, the time complexity depends on the number of iterations of the inner loop, which is exactly the number of symbols encoded, that is, nm. Each iteration is made up by constant operations, except for line 7: when the entry of $\mathcal{T}$ contains the special character `'*'`, this line is executed in $\mathcal{O}(\log l_{\max}) < \mathcal{O}(\log k)$ time. Overall, this procedure has a time complexity upper bounded by $\mathcal{O}(nm \log k)$ (the case occurring when line 7 always performs the binary search).

In Sect. III-C we describe how the procedure Dot$_{HAM}$ can be reworked in order to speed up the computations.

2) RIGHT MULTIPLICATION FOR HAM-COMPRESSED MATRICES

The HAM format is designed to serve as a neural net compression format, where the left matrix multiplication $\boldsymbol{x}^T\boldsymbol{W}$ is run at each layer of the network in the forward step. However, it is easy to adapt this format for a generic right multiplication $\boldsymbol{Wz}$. Indeed, it suffices to symmetrically construct the bitstream $HAM(\boldsymbol{W})$ in row-order instead of column-order, and exchange the semantic of variables col and row in the dot procedure described in Algorithm 1. The complexity of both HAM construction and its dot procedure remains the same.

B. SPARSE HUFFMAN ADDRESS MAP COMPRESSION

One potential limitation of HAM is that it marginally benefits from sparsity: indeed, in such a case the per-element space is only indirectly reduced—typically the source entropy is lower, having the symbol 0 a higher frequency. But when $\boldsymbol{W}$ is sparse and very large, we would use in any case a high amount of memory to store the 0s: for instance, assuming a 1-bit codeword for the symbol 0 (which is the best case), more than $10(1-s)$ GBits for a $10^5 \times 10^5$ matrix are used. To address this issue, we introduce an extension of HAM, named *sparse Huffman Address Map compression* (sHAM), in which the bitstream and the Huffman code is computed excluding the symbol 0. More precisely, $\boldsymbol{W}$ is represented using a bitwise CSC format, obtaining **nz**, **ri**, **cb** as in Sect. II-C1: the first vector is stored using the HAM format, the others are kept uncompressed. The Huffman code $H_{\mathbf{nz}}$ for non-null elements is computed, subsequently obtaining by concatenation the bitstream $HAM(\mathbf{nz}) = H_{\mathbf{nz}}(nz_1)\ldots H_{\mathbf{nz}}(nz_q)$. The latter is stored in the array $\mathcal{C}_{HAM}(\mathbf{nz})$ of $N_1 = \lceil |HAM(\mathbf{nz})|/b \rceil$ memory words. Finally, we build the sHAM representation of $\boldsymbol{W}$, denoted by $sHAM(\boldsymbol{W})$, as the sequence of vectors $\mathcal{C}_{HAM}(\mathbf{nz})$, **ri**, **cb**. The following fact establishes an upper bound for $|sHAM(\boldsymbol{W})|$.

Fact 2: (sHAM worst case) If $\boldsymbol{W}$ contains snm non-null distinct elements (excluding 0), the per-element average number of bits of the *sHAM* format is

$$\psi_{sHAM} \leq 3s \log snm + s(snm + b + \log n) + \frac{\log n}{n} - \frac{2 \log snm}{nm}.$$

Proof: Being all snm symbols distinct, they are equally probable, accordingly the entropy of **nz** source is maximum, that is $\mathcal{H}_{nz} = \log snm$, and:

1) the average codeword length of H_{nz} is upper bounded by $1 + \log snm$, thus $|\mathcal{C}_{HAM}(\mathbf{nz})| \leq snm(1 + \log snm)$;
2) the Huffman decoding structures require $snm(snm + 2\log snm + b - 1) - 2\log snm$ bits;
3) vectors **ri** and **cb** need $snm \log n + m \log n$ bits.

The thesis follows by summing up the contributions in 1–3 and dividing by nm. □

For example, given $n = m = 10^2$, and assuming $b = 32$, Fact 2 implies that, in the worst case for the Huffman code, we need around $s < 0.01$ to have $\psi_{sHAM} < b$, which is also clearly less efficient than the CSC format in the same conditions (cfr. (1)). Notwithstanding, Fact 2, analogously to Fact 1 for HAM, induces the following corollary showing how sHAM benefits from the matrix quantization.

Corollary 2: Given a matrix $\boldsymbol{W}$ *containing snm non-null elements, and* $k < snm$ *distinct values (excluding the* 0*), it holds*:

$$\psi_{sHAM} \leq s(1 + \log k + \log n) + \frac{\log n}{n} + \frac{B_k}{nm}. \quad (6)$$

Proof: As in Corollary 1, in the worst case $\mathcal{H}_{nz} = \log k$, and the snm values can be represented using at most $snm(1 + \log k)$ bits. Then we add the B_k bits for the CHC and $snm \log n + m \log n$ bits for the CSC structures. The thesis follows by dividing the sum of each component space by nm. □

Considering for example $n = m = 10^2$, $b = 32$, and $s = 0.5$, in the hypotheses of Fact 2 it is guaranteed that $\psi_{sHAM} < b$ for all $k < 463$. Moreover, under the same assumptions, $\psi_{sHAM} < \psi_{CSC}$ when $k < 324$, which represent quite a feeble quantization, considering that it yields in average that each distinct weight is shared only around 30 times. Noticeably, this means that our method, even in its worst case, can gain over the CSC representation even with quite low quantization levels.

Corollary 2 points out that the per-element average number of bits of sHAM is similar to that of HAM (see Corollary 1), where the first term is scaled by s, and two additional quantities $s \log n$ and $\frac{\log n}{n}$ are added for representing the indices of the CSC format. Thus, as expected, the lower s, the more convenient is sHAM w.r.t. HAM.

1) DOT PRODUCT

Algorithm 2 describes how to perform the dot product $\boldsymbol{x}^T\boldsymbol{W}$ when $\boldsymbol{W}$ is in sHAM format. Like in Algorithm 1, the loop at lines 2–16 scans the columns of $\boldsymbol{W}$, whereas the inner loop 4–13 computes the dot product between $\boldsymbol{x}$ and the current column col. The only difference with the HAM dot procedure is that this loop only considers non-zero elements in the current column. When the latter has been parsed, line 14 writes the result in the output vector $\boldsymbol{out}$. The inner loop is executed snm times, and the heaviest operation in its body is SearchLength, having complexity $\mathcal{O}(\log k)$ when the entry of $\mathcal{T}$ does not contain an exact codeword length, for an overall time complexity $\mathcal{O}(snm \log k)$. Analogously to Algorithm 1, this procedure is sequential, and in Sect. III-C we show how to parallelize it.

Algorithm 2 Dot Procedure for sHAM Representation

Procedure Dot$_{sHAM}$

```
Input:
– compressed array C_HAM(nz)
– q = |nz| number of non-zero elements in W
– row index vector ri
– vector cb
– vector x ∈ R^{n×1}
– first_code_l array
– first_symbol array
– symbols array
– the partial lookup table T
– maximum codeword length l_max
– n, m ∈ N, number of rows and columns of W
begin algorithm
 1: Initialize: out ←zeros(m), read ← 1, col ← 1,
               t ← ⌈log l_max⌉, nbits ← l_max, pos ← 1
               start ← 0
 2: while col ≤ m do
 3:    celem ← 0, sum ← 0
 4:    while celem ≤ cb[col] do
 5:       buff, start ← GetBuff(C_HAM(nz), start, nbits)
 6:       l ← GetEntry(T, buff, t)
 7:       l ← SearchLength(l_max, first_code_l, l)
 8:       nbits ← l_max − l
 9:       oset ← (l_max − first_code_l[l]) ≫ (l_max − l)
10:       val ← symbols[first_symbol[l] + oset]
11:       sum ← sum + x[ri[pos]] ∗ val
12:       pos ← pos + 1, celem ← celem + 1
13:    end while
14:    out[col] ← sum
15:    col ← col + 1
16: end while
end algorithm
Output: out = x^T W.
```

2) RIGHT MULTIPLICATION FOR SHAM-COMPRESSED MATRICES

In analogy with the HAM format, also sHAM can be easily adapted to support right multiplications of the form $\boldsymbol{Wz}$. Indeed, it suffices to represent $\boldsymbol{W}$ using the Compressed Sparse Row (CSR) format instead of CSC, which means that also the vector **nz** will list non-zero values of $\boldsymbol{W}$ in row-order. The corresponding dot product will simply extend the CSR dot procedure analogously to what we have done in Algorithm 2 for the CSC dot procedure. Also in this case, this does not alter the complexity of building the sHAM representation of $\boldsymbol{W}$ and of running the related dot procedure.

Algorithm 3 Pseudocode of the parallel Matrix multiplication for HAM Representation.

Procedure `ParDot`$_{HAM}$

`Input`: Same input of Algorithm 1 and
– column offsets $\boldsymbol{o} = (o_1, \ldots, o_m)$
– number of computing units g
– sets of column indices $G_1, \ldots, G_g$

`begin algorithm`
1: Initialize: **out**$[G_j]$ ←zeros($|G_j|$), $o_0 \leftarrow 1$
2: **for** $j = 1$ to g in parallel **do**
3:
$$\text{out}[G_j] \leftarrow \text{DotHamCol}(\mathcal{C}_{HAM}(\boldsymbol{W}), \boldsymbol{o}[G_j], \mathcal{T}, l_{\max}, n, m \\ \boldsymbol{x}, first_code_l, first_symbol, \textbf{symbols})$$
4: **end for**
`end algorithm`

`Output`: **out** $= \boldsymbol{x}^T \boldsymbol{W}^{G_j}$ in parallel.

C. MULTI-THREADED MEMORY-SHARED DOT PRODUCT OFHAM AND SHAM

To take advantage of modern multi-core architectures, the procedures `Dot`$_{HAM}$ and `Dot`$_{sHAM}$ can be adapted to parallel computation by exploiting the nature of matrix multiplication, since $\boldsymbol{x}^T$ can undergo, in parallel, the dot product with individual columns of $\boldsymbol{W}$. The evaluation of $\boldsymbol{x}^T\boldsymbol{W}$ can be distributed across $1 < g \leq m$ computing units, each assigned to a chunk G_i of contiguous column indices, with $1 \leq i \leq m$, such that $G_i \cap G_j = \emptyset$ for any $1 \leq i, j \leq m$, with $i \neq j$, and $\cup_{i=1}^{g} G_i = \{1, \ldots, m\}$. In order to allow the g units to work in parallel, when growing the bitstream $HAM(\boldsymbol{W})$ we keep trace of the offsets o_r when column r ends, for each $r \in \{1, \ldots, m\}$, so that each unit can retrieve the portion of $HAM(\boldsymbol{W})$ corresponding to the columns assigned to it. For instance, if $G_1 = \{3, 4\}$, the unit 1 will parse the substring $HAM(\boldsymbol{W})$, suitably derived from the array $\mathcal{C}_{HAM}(\boldsymbol{W})$, starting from position $o_2 + 1$ and ending in position o_4.

The pseudocode of the parallel procedure is shown in Algorithm 3, where the procedure DotHamCol realizes the dot procedure of Algorithm 1, with two differences: at line 5 a variant of GetBuff is called, receiving the o_{j-1}, o_j offsets, fetching **buff** just in that portion of the bitstream, and returning the dot product $\boldsymbol{x}^T\boldsymbol{W}^{G_j}$. While this procedure allows a speed-up bounded by $g\times$ (ideal case without shared resources and parallel overhead), it just slightly increases the demand of memory, roughly up to $(g - 1)b$ bits for the weights fetched in parallel by individual units, plus the space for storing the offsets. We implement this strategy by means of CPU multi-threading, where the bitstream is split among the available threads in such a way that each thread is responsible for the same amount of contiguous columns. Being column-independent, the dot product is executed through the SIMT paradigm, and the result of each thread is then combined in order to obtain the final output. The implementation leverages the C multitreading *pthread* library, and shares memory across thread to strongly limit the memory overhead induced by multithreading. The design of a parallel version for sHAM is derived analogously.

D. EXTENSION TO GPU DEVICES

The efficient implementation of sparse matrix-vector multiplication (SpMV) is crucial in scientific computing applications, and the introduction of General Purpose Graphics Processing Units (GPGPUs) renewed the interest in high-performance computing architectures to solve this task [48]. Huge ongoing efforts are attempting to develop SpMV kernel on GPGPUs{- for existing SpMV formats, e.g., for CSR [49] and CSC [50]. This constitutes a big challenge which does not admit a *one-size-fits-all* solution, since GPGPU sparse matrix representations require to be tailored for particular sparsity patterns, or for specific architectures, with dedicated performance tuning procedures [48]. On the other hand, early research works are focusing also on extending Huffman coding to GPGPUs architectures [51], and merging this two lines of research deserves accurate and dedicated studies, which are beyond the scope on this work. We leave it to future extensions.

E. EVALUATION OF THE ENERGY EFFICIENCY

The compact representations considered in this paper have already been evaluated in terms of space and time complexity, but in this section, following emerging trends in algorithm performance assessment, we also investigate their energy consumption. Due to the ever increasing power demand of modern systems, energy has become a leading design constraint for both computing devices and algorithms, and nowadays the research community is paying more and more attention to efficient algorithms that reduce energy consumption while minimizing the related compromises to effectiveness [52]. However, exactly measuring the energy demand of an algorithm is rather difficult, given its connection with different factors, including the programming language used in its implementation, as well as the underlying hardware architecture [53].

For these reasons, we adhere in this study to an energy evaluation framework recently proposed in the context of neural network compression, which models the energy costs in a way which is easily adaptable across different software platforms, as well as hardware architectures [18].[2] In this model, the energy is computed on the basis of four elementary operations, namely: 1) *mul*, the binary multiplication operator, 2) *sum*, the binary addition operator, 3) *read*, which reads a value from memory, and 4) *write*,

[2]By modelling the energy costs of individual elementary operations, we can potentially adapt the framework to any different experimental setting, simulating—albeit approximately—the energy consumption variation of specific hardware platforms.

which writes a value into memory. This model assimilates the costs of read/write operations from/into low-level memory (like caches and registers) that stores temporary runtime values directly into the costs of the corresponding elementary operations.

Then, each of these operations is associated with an energy cost, and the total energy required for a given dot product algorithm sums up the overall costs of the elementary operations. Furthermore, this model also takes into account the possibility of operating on values of different size in bits, as happening for the compressed formats described in Section II-C. Indeed, the cost of elementary operations might vary when the size of the input values changes. To this end, four cost functions $f_m, f_s, f_r, f_w : \mathbb{N} \rightarrow \mathbb{R}^+$ are considered, taking as input the size in bits and returning the corresponding cost, respectively for the *mul*, *sum*, *read* and *write* operations. Finally, when two inputs have different size for *sum* and *mul*, the maximum size is elected as input for the cost function.

1) ENERGY EFFICIENCY OF CSC, IM AND CSER

Here we briefly report the energy computation of the dot procedures for the CSC, IM and CSER formats. The pseudocode of these procedures can be found in the Supplementary Section I-A, along with further details about their energy computation. As mentioned above, the energy cost is based upon the real need of memory for each structure required by compressed formats. To avoid an overly complex notation, in the following we denote by b_y the smallest size (expressed in bits) in the set {8, 16, 32} which is sufficient either to store an element of vector $\boldsymbol{y}$, or to store elements of a vector whose components are less or equal than y (when it is possible to estimate it), ensuring the distinction is always clear from the text.

a: CSC

To execute the dot product $\boldsymbol{x}^T \boldsymbol{W}$ when $\boldsymbol{W}$ is in CSC format, the following per-element energy costis needed:

$$E_{\text{CSC}} = s\Big(f_s(b) + f_m(b) + f_r(b_n) + f_r(b) + f_r(b_x)\Big) + \frac{f_w(b) + f_r(b_n)}{n} \tag{7}$$

where b_n is the size required to store an element of vectors **cb** (value $\leq n$) and **ri** (value $\leq n$), whereas b_x is the bit size to store an element of $\boldsymbol{x}$. Here and hereafter, to simplify the notation we assume the output size is b, since writing the output affects the energy only for a $\mathcal{O}(1/n)$ term. Reminding that s is the ratio on non-zero elements, the cost (7) is derived by considering the following steps, whose energy cost is written in brackets: (i) loading the elements of the input vector ($sf_r(b_x)$), which in turn requires $sf_r(b_n)$ for loading an index from **ri**; (ii) loading the elements of the matrix ($sf_r(b)$); (iii) multiplying them ($sf_m(b)$); (iv) summing the products ($sf_s(b)$); (v) writing the result ($f_w(b)/n$); (vi) checking a given column is ended ($f_r(b_n)/n$).

b: INDEX MAP

The per-element energy cost of IM format is

$$E_{\text{IM}} = f_r(b) + f_r(b_k) + s\Big(f_s(b) + f_m(b) + f_r(b_x)\Big) + \frac{f_w(b)}{n} \tag{8}$$

where b_k is the size necessary to store an element of matrix $\boldsymbol{\Pi}$ (value $\leq k$). The factor s derives from the check that the current weight W_{ij} is not null (line 6 of Algorithm S2, Supplementary Material).

c: CSER

For CSER format it holds

$$E_{\text{CSER}} = s\Big(f_s(b_x) + f_r(b_x) + f_r(b_n)\Big) + \frac{\bar{k}\Big(f_s(b) + f_m(b) + f_r(b_{snm}) + f_r(b_k) + f_r(b)\Big)}{n} + \frac{f_w(b) + f_r(b_{rowPtr})}{n} \tag{9}$$

where b_{rowPtr} and b_{snm} represent respectively the bit size of an element of the vectors **rowPtr** and of ΩPtr (value at most snm). Since CSER is designed to perform right matrix multiplications, for a fair comparison here we compute $\boldsymbol{W}^T\boldsymbol{x} = \boldsymbol{x}^T\boldsymbol{W}$. Accordingly, $\bar{k}$ is the average number of distinct weights per row of $\boldsymbol{W}^T$.

F. ENERGY EFFICIENCY OF HAM AND SHAM

This section discusses the energy computation of the dot procedures for the proposed methods HAM and sHAM, under the same assumptions made for the other formats. We can prove the following result.

Fact 3: Let $\boldsymbol{W} \in \mathbb{R}^{n\times m}$ *a sparse and quantized matrix, with a fraction* $s \in [0, 1]$ *of non-zero entries and* $k > 1$ *distinct weights, and* $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$ *be an input vector. The per-element energy cost for computing the product* $\boldsymbol{x}^T\boldsymbol{W}$ *when* $\boldsymbol{W}$ *is in the HAM format is*

$$E_{HAM} = \left(1 + \frac{N}{nm}\right) f_r(b) + f_r(b_{l_{\max}}) + f_r(b_k) + \left(1 + \frac{D \log l_{\max}}{nm}\right) f_r(b_{first_code_l}) + s\big(f_s(b) + f_m(b) + f_r(b_x)\big) + f_w(b)/n, \tag{10}$$

where N *is the number of memory words in* $\mathcal{C}_{HAM}$, *and* D *is the number of times we need a binary search in first_code_l at line 7 of Algorithm 1.*

The proof of Fact 3 is given in the Supplementary Section II. Here we remind that $l_{\max}$ is the maximum length in bits of a codeword, while $b_{first_code_l}$ and $b_{l_{\max}}$ are the sizes to store elements of *first_code_l* and $\mathcal{T}$, respectively. Also in this case, the factor s for the costs f_m, f_s and $f_r(b_x)$ is due to the check that the value read from the matrix is non null (line 11). The array *first_code_l* stores values having $l_{\max}$ bits, which in principle could induce $b_{first_code_l}$ to be larger than b.

Nevertheless, in practice this never happens: indeed, in [54] it was proved that $l_{\max}$ depends on the minimum symbol probability $p_{\min} := \min_{i\in\{1,\ldots,k\}} p_i$, and that in order to have $l_{\max} = 33$, it must hold $1/F_{33+3} < p_{\min} \leq 1/F_{33+2}$, where F_j is the j-th Fibonacci number. This means $6.697 \cdot 10^{-8} < p_{\min} \leq 1.084 \cdot 10^{-7}$. Accordingly, in most real cases we can expect $l_{\max} \leq 16$, since this happens when $p_{\min} > 0.00024$. This is confirmed by our experiments, where we frequently found even $l_{max} \leq 8$.

Overall, the energy efficiency of the HAM dot procedure mainly depends on N and D, which are related to the read operations on the two largest bit sizes, and which are not scaled by s. In turn, their value depends on the Huffman code. In particular, we know that, for a Huffman code, $\mathcal{H} + 1$ is an upper bound to the average codeword length in bits (see Section III-A), and accordingly $|HAM(\boldsymbol{W})| \leq nm(\mathcal{H} + 1)$ bits. Therefore,

$$N \leq \left\lceil \frac{nm}{b}(\mathcal{H} + 1) \right\rceil.$$

Instead, D depends on the chosen value of t. In our case, rather, we have $0 < t < l_{\max}$, as we used $t = \lceil \log l_{\max} \rceil$. Such a choice of t typically covers the majority of codeword lengths [46], as we also empirically verified in our experiments, meanwhile saving much space w.r.t. the full lookup table: as mentioned above, the number of entries increases only linearly with $l_{\max}$.

Analogously, for sHAM we can prove the following fact, whose proof is given in the Supplementary Section II.

Fact 4: If $\boldsymbol{W}$, $\boldsymbol{x}$, b_x, $b_{first_code_l}$ and $b_{l_{\max}}$ are defined as in Fact 3, the per-element energy cost for computing the product $\boldsymbol{x}^T\boldsymbol{W}$ when $\boldsymbol{W}$ is stored in the sHAM format is

$$\begin{aligned} E_{sHAM} = &\left(s + \frac{N_1}{nm}\right) f_r(b) \\ &+ \left(s + \frac{D_1 \log l_{\max}}{nm}\right) f_r(b_{first_code_l}) \\ &+ s\Big(f_s(b) + f_m(b) + f_r(b_x) + f_r(b_n) \\ &+ f_r(b_{l_{\max}}) + f_r(b_k)\Big) \\ &+ \frac{f_r(b_n) + f_w(b)}{n}, \end{aligned} \quad (11)$$

where N_1 is the number of memory words in $\mathcal{C}_{sHAM}$, and D_1 is the number of times we need a binary search in first_code_l at line 7 of Algorithm 2.

We point out that the symbols $b_{first_code_l}$, $l_{\max}$ and consequently $b_{l_{\max}}$ have the same semantics as for HAM, but in principle they can have different values, since in this case the Huffman code is constructed only on the non-zero elements. Comparing to HAM energy and coherently aiming to leverage sparsity, the terms $f_r(b), f_r(b_{first_code_l}), f_r b_{l_{\max}}$ and $f_r(b_k)$ are now scaled by s, whereas the additional costs $sf_r(b_n)$ and $f_r(b_n)/n$ are due to the accesses to vectors **ri** and **cb**. Furthermore, denoted by $\mathcal{H}_1$ the entropy of the source when the 0 symbol is excluded, similarly to HAM it follows

$$N_1 \leq \left\lceil \frac{snm}{b}(\mathcal{H}_1 + 1) \right\rceil,$$

which further benefits from the sparsity of $\boldsymbol{W}$. Concerning D_1, the same considerations made for D hold here.

IV. EXPERIMENTS AND RESULTS

In this section we initially test four variants of canonical Huffman coding on synthetic data, then we compare the various compression formats on some benchmark low-entropy matrices, and on two publicly available pretrained DNN models, proposed respectively for image classification and drug-target affinity prediction.

TABLE 2. Energy costs in pJ (Picojoule) of different elementary operations for a 45-nm CMOS processor [18], [55]. MB and KB denotes megabytes and kilobytes, respectively.

Operation	8 bits	16 bits	32 bits
float add	0.2	0.4	0.9
float mul	0.6	1.1	3.7
R/W (<8KB)	1.25	2.5	5.0
R/W (<32KB)	2.5	5.0	10.0
R/W (<1MB)	12.5	25.0	50.0
R/W (>1MB)	250.0	500.0	1000.0

A. PERFORMANCE CRITERIA

All compression methods are compared using the following three criteria.

1) *Memory Requirements*: the average number of bits per matrix entry required by the various matrix compression formats, namely according to the measure ψ as in Section II-C. We recall that, in practice, any value x is stored in a memory variable whose size in bits is the smallest among $8, 16, 32$ which is greater than $\log x$. This size is the one practically counted in the computation of ψ.
2) *Energy Complexity*: the average energy cost per-matrix element measured according to Equations (7–11). To fully adhere to the energy framework discussed so far and proposed in [18], we also utilize the energy cost estimates reported therein, and shown in Table 2. The costs of read and write operations largely depend on the size of the object which they are applied to. This means that the theoretical energy formulas do not suffice to compare two methods, since for instance the same function $f_r(b)$, for a given $b \in \{8, 16, 32\}$, actually depends on a second input, the size $\mathcal{S}$ of the object it is applied to. Thus a more correct notation would be $f_r(b, \mathcal{S})$, *de facto* adopted in practice. Notwithstanding, to not further complicate the discussion, we keep the notation originally proposed.

Note that this framework does not include the complexity of converting the dense representation into the different formats, with the rationale that this can

be considered a *una tantum* operation like for deep neural networks, where this step can be applied a priori, subsequently performing all inferences using the compressed format.

3) *Time Complexity*: the average time in seconds to perform the dot procedure once. It is measured on multiple runs, executed on the same hardware architecture for all compared compression formats. We differ in this from the original framework, where each operation is timed and then the overall time complexity is obtained from the energy by substituting the time of each operation. In this way, the time complexity would have been only a surrogate of the energy, reflecting also the same assumptions and simplifications (all operations but the four elementary ones are omitted). In addition, neither the estimated operation time, nor the procedure adopted for it has been made available in the original framework. Such an estimation is well-known being error prone, particularly to obtain the estimate for sizes smaller than one memory word [56].
The experiments have been carried out on an Ubuntu machine equipped with 2 Intel(R) Xeon(R) Bronze 3106 CPU @ 1.70GHz with a total of 16 cores, and 256 GB of RAM.

Finally, concerning criterion 1), when useful to better understand the results, we also report the *compression ratio*, as described in Section II-B1. In the following we use $b = 32$, as in most neural network architectures and as adopted in the original framework. However, on the benchmark matrices some studies used $b = 64$ (see, e.g., [57]), which should be taken into consideration since it clearly yields higher compression ratios than $b = 32$.

B. ANALYSING CANONICAL HUFFMAN CODE VARIANTS

We test some alternative variants of Canonical Huffman codes used in the HAM and sHAM formats. In particular, we refer henceforth to the CHC version described in Section III-A using the adjective *partial*, and we consider two alternatives of the latter: *full*, characterized by a full lookup table ($t = l_{\max}$), and *no-table*, in which no lookup table is used. Moreover, we also test a classical canonical variant which fetches one bit at a time from the bitstream, here named *bitwise*. Here below we briefly describe the latter, then we show a comparison on synthetic matrices suitably generated to test those variants on different levels of sparsity/quantization and source entropy. As a remark, for a fair comparison we always include 0 among the k distinct symbols, since HAM does so.

1) BITWISE CANONICAL HUFFMAN CODE

This variant [19] needs the following decoding structures: i) an array *fc*, whose size is $l_{\max}$, with fc$[l]$ being the first codeword of length l; ii) a table of symbols *symb*, where symb$[l]$ is the list of symbols having a codeword of l bits. Note that symb$[l]$ must be initialized with consecutive codewords starting from fc$[l]$ (in the sense of the integer value associated with them). We recall $l_{\max}$ is the maximum length in bits of a codeword. Like for the buffered variants, the decoding procedure is still fast since it does not require storing and traversing the tree structure (see [19] for details).

2) TIME/SPACE TRADE-OFFS FOR BUFFERED CANONICAL CODES

The choice of parameter t, leading to the three buffered CHC variants *full*, *partial* and *no-table*, allows meantime to partially control the trade-off between space and dot product time. With reference to Equations (1) (for HAM) and (2) (for sHAM), the term B_k is affected by the dimension of the lookup table $\mathcal{T}$, expressly with the upper bound term $(k-1)\log k$, when $t = \log l_{\max}$. Meanwhile, the choice of t affects the execution time of the dot procedures of HAM and sHAM (cfr. (10) and (11), respectively), specifically in the values of D and D_1. Therefore, when using $t = l_{\max}$ (*full* variant) we have $D = D_1 = 0$, being a binary search in array *first_code_l* necessary only for the codewords longer than t, while the size of the lookup table increases up to $2^{k-1}\log k$. Conversely, the *no-table* variant allows us to save room, since the lookup table is no longer used (formally equivalent to $t = -\infty$). Nevertheless, this choice of t slows down the dot execution, inducing the maximum possible number of searches in *first_code_l*, namely $D = nm$ and $D_1 = snm$. It is worth noting that, in principle, the energy consumption should behave similarly to execution time, but in practice this is not guaranteed, since the cost for read and write operations might vary with the size of the object on which they are performed on (see Section IV-A), in this case the lookup table.

3) EXPERIMENTAL COMPARISON OF CHC VARIANTS

Since the efficiency of Huffman coding depends on the source entropy, we aim at investigating the behavior of the four aforementioned CHC variants (*bitwise*, *no-table*, *partial*, and *full*) when the matrix weights induce sources with different levels of entropy. In addition, we also study their performance in relation to the quantization and pruning levels of the input matrix. The data generation procedure adopted is described here below.

a: SYNTHETIC DATA GENERATION

In order to compare only the storage formats, we already generate matrices showing low entropy characteristics. In particular, synthetic matrices are generated to exhibit three different entropy levels, named *low*, *medium* and *high*, and multiple quantization choices, initially without applying pruning. More precisely, for any fixed choice of $k \in \{5, 10, 15, 20, 25, 30\}$, we randomly generate three square matrices of dimension $n = 5000$ having different entropy levels, corresponding to sources $Z = (z_1, \ldots, z_k)$ whose probabilities $(p_1, \ldots, p_k)$ are fixed as follows:

- *low*: set two initial values $\bar{p}_1$ and $\bar{p}_2$, with $\bar{p}_1 < \bar{p}_2$; then, for a given small $0 < q \ll 1$, iteratively set $\bar{p}_i = \bar{p}_{i-1} + \bar{p}_{i-2} + q$, with $i = 3, \ldots, k$; finally, for each i

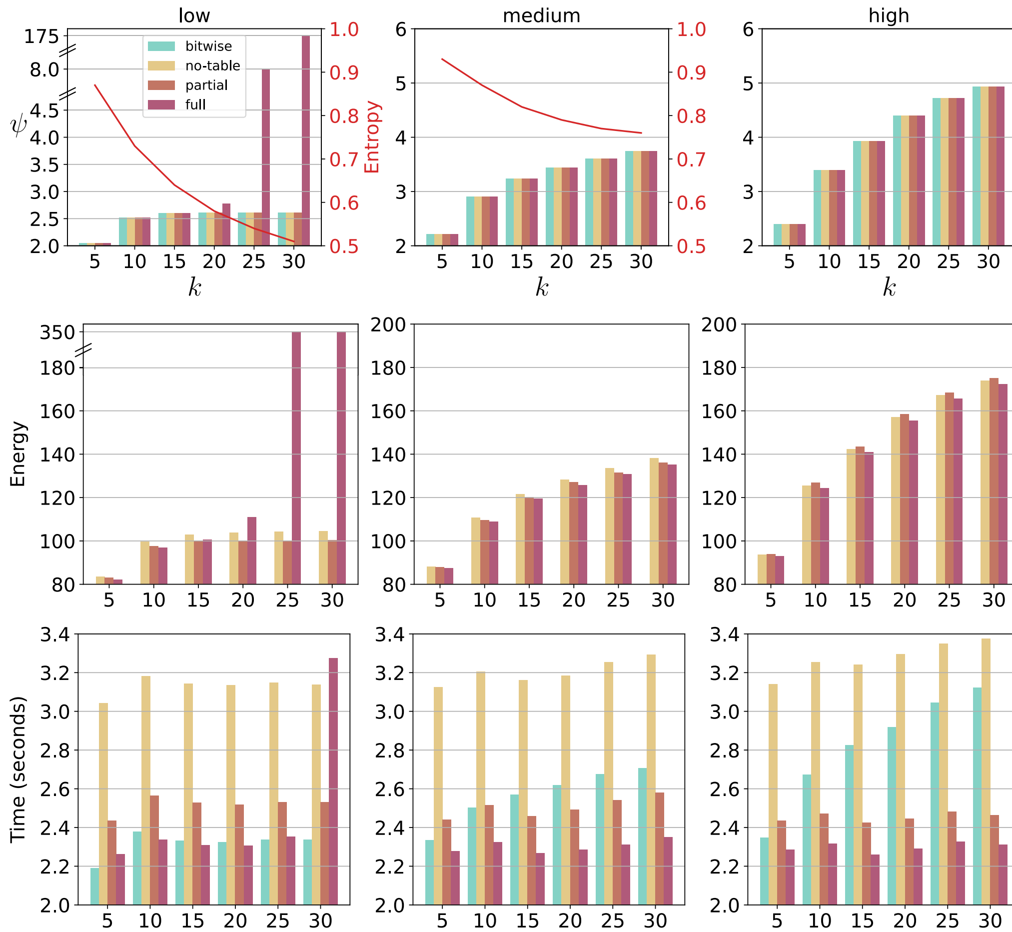

FIGURE 1. Per-element storage requirement ψ (above), per-element energy cost (middle), and dot time (below), averaged across multiple dot products $x^T W$ with different synthetically generated W. The matrix is represented via HAM CHC variants. Here, $n = m = 5000$, and k is the number of distinct weights in W. Red lines represent the normalized entropy of the generated matrices.

set $p_i = \bar{p}_i / \sum_j \bar{p}_j$; the resulting distribution is obviously unbalanced, and in particular it also yields a CHC with the maximum possible $l_{\max}$, that is $l_{\max} = k - 1$;

- *medium*: set p_i according to a truncated geometric distribution of parameter p, selected to ensure that the minimum probability is large enough to have at least one occurrence in the generated matrix of the corresponding symbol, and an intermediate normalized entropy between 0 and 1: namely $p_i = \left(p(1-p)^{(i-1)}\right) / \left(1 - (1-p)^k\right)$;
- *high*: set $p_i = 1/k$ for each i.

In order to assess the performance of the CHC variants in presence of sparsity along with quantization, we further generate matrices with different levels of sparsity $(1 - s)$: in particular, we keep $n = 5000$ and fix $k = 10$, a low enough value to guarantee each symbol has at least one occurrence. We generate three matrices having sparsity 0.6, 0.75 and 0.9, respectively. Note that by fixing n and k, the variation of s implicitly induces a variation in the entropy: the higher the sparsity, the lower the entropy of the corresponding source. As a consequence, fixed the probability of symbol 0 to $1 - s$, we extract the remaining symbols as in the case of *medium* entropy.

b: RESULTS

For dense matrices only HAM format has been tested, being sHAM inefficient in this case. The corresponding results are shown in Figure 1, in which the energy complexity of the *bitwise* CHC variant cannot be shown, as its energy computation is not available. Red lines show the normalized entropy of the matrix/source, computed as in Section II-B1: in the case of high entropy the distribution is uniform, hence we don't show the normalized entropy because it always equals 1. Moreover, being results in each column related to the same input matrices, the entropy value is shown only in the first row, in a second axis.

In terms of space, there is no significant difference across CHC variants, except for the full one, where for $k > 20$ we observe in the low-entropy case a sensible increment in the per element storage (ψ), due to the exponential growth of the lookup table. Indeed, its dimension increases with $l_{\max}$, that in this case is equal to $k - 1$, differently from the medium- and

high-entropy cases. This means that in remaining cases the bitstream takes up the large majority of the overall memory. The variant must be thereby selected banking upon time and energy behaviors. The dot product time (averaged across 40 repetitions) reveals that the *no-table* variant is always the slowest,[3] as expected, while the comparison between *bitwise* and buffered variants significantly depends on the source entropy. In particular, *bitwise* is faster in low-entropy cases, but it becomes much slower when the entropy increases. This behavior is likely to depend on the decoding procedure of the variant, which fetches one bit at a time, and consequently the higher the average codeword length, the higher the time needed to retrieve the next codeword. Indeed, in low-entropy cases we have small l_{max} values, which mitigates this effect. Moreover, the buffered variants using a lookup table (*partial* and *full*) are more robust than *bitwise* to the value of k, remaining their time almost the same when k increases. Finally, the energy complexity estimates reflect somehow the behavior noted w.r.t. space complexity, although now the differences between the three buffered variants are not negligible, with the best results attained by *partial* variant in the *low* case, and by *full* in the remaining settings.

In the experiment adding sparsity to the generated matrices, we apply both HAM and sHAM, since with increasing sparsity the latter becomes a more efficient option. Figure 2 shows the results of this experiment, w.r.t. three different levels of sparsity: HAM still attains the best results both in terms of space and energy. It is worth noting that some of the Y-axes of the involved plots are broken, and the correct interpretation is that HAM is still more efficient than sHAM, but their gap is going to reduce when the sparsity increases (till $1 - s = 0.9$). For larger values of sparsity, sHAM would become more efficient than HAM, as observed in the experiments involving benchmark matrices and neural networks (cfr. Sect. IV-C). Moreover, we are showing the entropy of the whole matrix, zeroes included, which does not directly impact of sHAM performance, since only the entropy of **nz** source affects its efficiency. Therefore, the entropy of **nz** can be high even in the lowest-entropy case for $\boldsymbol{W}$ (rightmost column), and accordingly we should not aim at definitive conclusions about the comparison of HAM and sHAM from this experiment.

Concerning dot time, instead, sHAM is consistently superior to HAM in all the tested variants, attaining in some cases a very relevant improvement. With reference to the behavior of the different variants, we note a very similar space and energy trends for both HAM and sHAM. The energy of HAM still slightly decreases when increasing the size of the lookup table (from *no-table* to *full* variants), confirming the behaviour observed in Figure 1, while sHAM is almost insensible to it. This might depend on the fact that, being $D_1 < D$ and $N_1 < N$, passing from *no-table* to *full* does not grant to save a significant number of accesses to *first_code_l*. Finally, we note that *bitwise* can exploit sparsity in a better way w.r.t. the ''chunked'' variants, although it becomes much slower when dealing with dense matrices (see Figure 1).

In summary, these results suggest the *partial* variant must be selected, because: 1) it has almost the same ψ and energy cost of the *no-table* variant, but it is much faster, mainly in dense cases; 2) the *full* variant is very unstable in low-entropy cases under all performance criteria. Further, in the sparse case it is only slightly slower that *bitwise* when using sHAM (variant to be used typically on sparse data); conversely, on dense data it is much faster than *bitwise* in conjunction with HAM, format preferable on dense inputs.

[3]The case low entropy and $k = 30$ is an exception, since the full lookup table becomes very large, with a consequent degradation even in direct access time.

C. STATE-OF-THE-ART COMPARISON

We compare all matrix compression formats described in Sect. II-C, with the addition of *Compressed Linear Algebra* (CLA) [15], [16], which is, as anticipated in the text, a compendium of effective column compression schemes, cache-conscious operations, and a sampling-based compression algorithm to select the compression scheme more suitable for each column or group of columns. CLA, available as a multi-threaded Java implementation, outperformed state-of-the-art methods like CSR-VI, D-VI, Gzip, LZ4, Snappy [16]. We set up two experiments: 1) compressing sparse and quantized benchmark matrices, 2) compressing deep neural networks.

TABLE 3. Benchmark matrices used in the experiments of Sect. IV-C1. The columns report, respectively, the number of rows and columns (n and m), the ratio of non-null entries (s), and the number of distinct entries (k).

Matrix	n	m	s	k
orsreg_1 [58]	2 205	2 205	2.907e−3	111
SiNa [58]	5 743	5 743	6.027e−6	24 317
Covtype [59]	581 012	54	2.200e−1	6 682
ImageNet [60]	1 262 102	900	3.099e−1	824
Census [59]	2 458 285	68	5.697e−1	45

1) COMPRESSING BENCHMARK MATRICES

We selected five matrices used as benchmark in previous works in this context [16], [61], and offering a wide spectrum of matrix characteristics: namely, dimension, sparsity, and number of distinct values, as detailed in Table 3. Since the matrices already are sparse and/or quantized at different levels, we just compare the algorithmic features and performance of all storage formats described so far, without applying any lossy compression. To compare the methods also in terms of dot product execution time, we ran 50 vector matrix multiplications $\boldsymbol{x}^T\boldsymbol{W}$ using 50 randomly generated $\boldsymbol{x}$, and averaged the results.

a: RESULTS.

The obtained compression ratios are shown in Table 4, while Figure 3 illustrates the analogous results in terms of per-element storage and energy requirements. The average

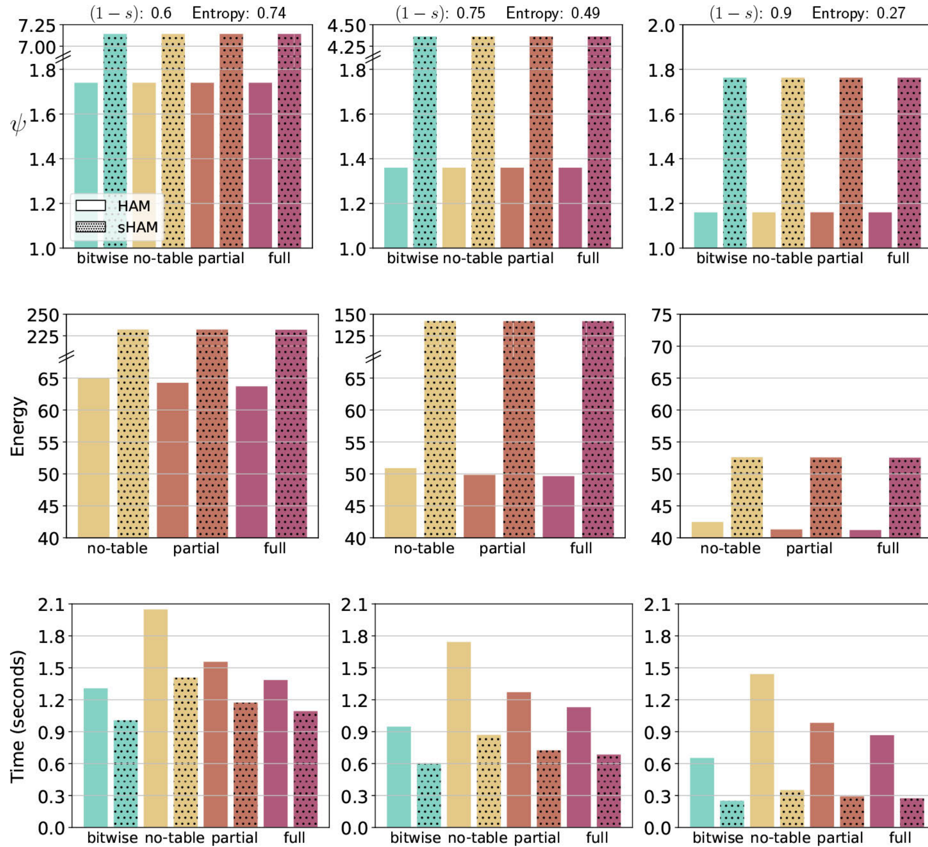

FIGURE 2. Per-element storage requirement ψ (above), per-element energy cost (middle) and dot time (below) averaged across multiple dot products $x^T W$ with different synthetically generated W, quantized (with $k = 5$) and sparsified (sparsity coefficients are 0.6, 0.75 and 0.9, from left to right, and shown as label of each column, alongside the normalized entropy value). The HAM and sHAM formats are represented via plain and dotted bars, respectively.

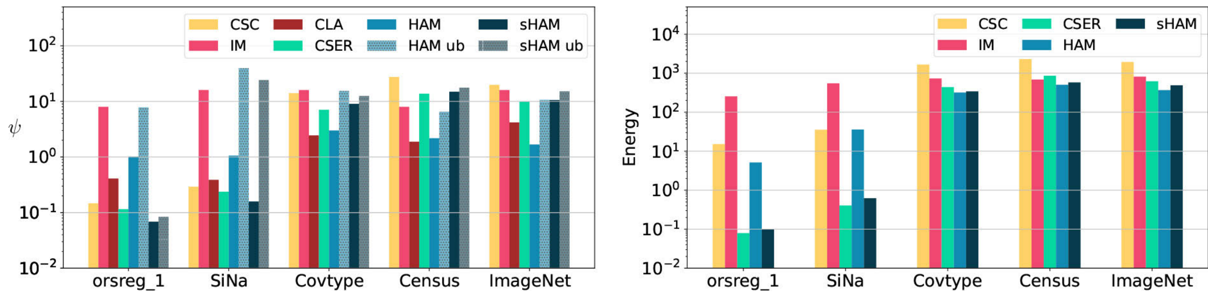

FIGURE 3. Per-element storage requirement ψ (left) and per-element energy cost (right) w.r.t. the representation of the benchmark matrices listed in Table 3. Bars having the suffix "ub" refer to the theoretical upper bounds computed for HAM/sHAM, respectively in Corollaries 1 and 2. Values are reported in logarithmic scale. For the uncompressed matrix, $\psi = b = 32$.

time required for executing a dot product (we remark, once again, without re-expanding the matrix) are shown in Figure 4, where CSER is not included because its implementation is not available. We can distinguish two main scenarios, basically related to sparsity, as described below.

- In case of the highly sparse matrices *SiNa* and *osreg_1*, sHAM always attains the best compression ratio, and only CSER has a comparable (though sensibly worse) performance. The situation is reversed when we focus on energy consumption: indeed, these two formats outperform the remaining ones, although CSER has slightly lower requirements than sHAM (CLA not shown because its energy consumption is not provided). sHAM is, by far, also the fastest method, with the only exception of CSC, which in any case has worse results in terms of space and, especially, energy.
- Within the remaining category of denser matrices, HAM is the top performer in terms compression ratio on

ImageNet data, and the second one after CLA on *Covtype* and *Census* data. It also exhibits the lowest energy consumption, at the detriment of an execution time for the dot product, which is around one order of magnitude higher than CLA.

As a further interesting behaviour, in Figure 3 we also computed the corresponding space upper bounds for HAM and sHAM according to (5) and (6), respectively. Since HAM depends more on the entropy than sHAM (the overhead for the CSC structures is entropy-invariant), the HAM upper bound tends to be looser in general w.r.t sHAM. *SiNA* data is an exception due to its very large k, that makes the space upper bound much higher than the actual per element space, even for sHAM: In this case, we observe a larger gap also for HAM, because the elevated sparsity induces entropy values much smaller than the maximum ($\log k$), assumed instead in the upper bounds.

TABLE 4. Compression ratios obtained by the methods of Sec. II-C on the benchmark matrices listed in Table 3. Values in bold highlight the best method for each matrix.

Matrix	HAM	sHAM	CSER	CSC	IM	CLA
orsreg_1	31.53	**466.08**	277.61	218.01	4.00	78.06
SiNa	30.21	**202.02**	134.66	109.56	1.99	82.36
Covtype	10.67	3.54	4.53	3.64	1.99	**13.08**
Census	14.72	2.14	2.32	1.86	4.00	**16.96**
ImageNet	**19.06**	3.02	3.23	2.15	2.00	7.65

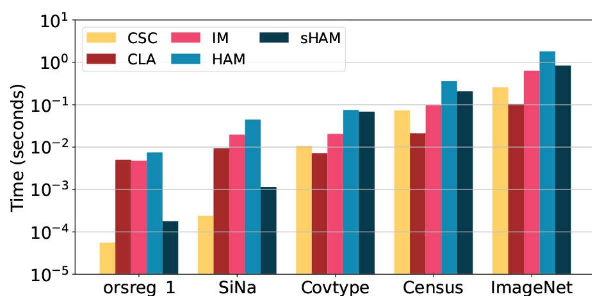

FIGURE 4. Average dot product time (in seconds) for the experiments shown in Figure 3.

2) COMPRESSING DEEP NEURAL NETWORKS

Being DNNs the most effective solution nowadays in several application domains, we test the efficiency of the compared methodologies in compressing publicly available DNNs. Using pre-trained networks allows a fair analysis of the compression and storage techniques, without introducing potential biases in the procedures devoted to training, model selection, and model assessment. Interestingly, two of the compared methods (precisely, IM and CSER) have been proposed specifically for this task. The comparison is performed by calculating the dot product without expanding the matrix. For this reason we exclude from this evaluation other DNN compression techniques that need to expand the matrix to perform the dot product, or to fine-tune weights, such as the *Tensorflow Lite Converter*.[4] In the following we provide details about the selected DNN models, the corresponding dataset and learning problems, some implementation details and the overall results which we obtain.

a: MODELS

We retrieved the following two publicly available pre-trained deep neural networks.

(i) *VGG19* [7], a classification model consisting of 16 convolutional layers followed by a FC block, in turn containing two dense layers of 4096 neurons each, and a 10-units softmax output layer.[5] It was proposed for the *ImageNet Large Scale Visual Recognition Challenge* [62], and subsequently also used in the realm of handwritten digits classification, which is the case we consider here, focusing in particular on the MNIST dataset [63].

(ii) *DeepDTA* [64], a regression model having two separate input blocks for proteins and ligands, both containing three convolutional layers followed by a max pool layer, and merged in a FC block consisting of three dense layers, respectively containing 1024, 1024 and 512 units, and a single-neuron output layer.[6] We use the same dataset on which the model was originally trained, namely the DAVIS dataset [65] storing affinities between proteins and ligands (that the models aim to predict).

Reminding that the storage formats which we compare directly support the vector-matrix multiplication required to compute the output of the FC layers, we only compress the latter, underlying that they contain a vast majority of the learnt parameters in both models. In addition, when reporting the compression ratios related to these layers, we also analyse how this impacts on the memory requirements of the overall network (that is including the space of convolutional layers, which are not compressed). Both networks undergo pruning and quantization of their FC layers via the techniques describes in Section II-B, with a subsequent fine-tuning of weights which preserves the same training configurations adopted in the original works. Finally, we represent the resulting weight matrices using the compressed formats under study and test the efficiency of the latter with regard to the criteria described in Sec. IV-A.

We adhere here with the following aspects of the framework suggested in [12]:

- we select the compression configurations (p-th empirical percentile of weights and number k of distinct weights) so as to guarantee no performance decay w.r.t. the original model (in terms of accuracy and mean squared error, respectively, for classification and regression), namely $p \in \{30, 40, 50, 60, 70, 80, 90, 95,$

[4]https://www.tensorflow.org/lite/convert
[5]https://github.com/BIGBALLON/cifar-10-cnn
[6]https://github.com/hkmztrk/DeepDTA

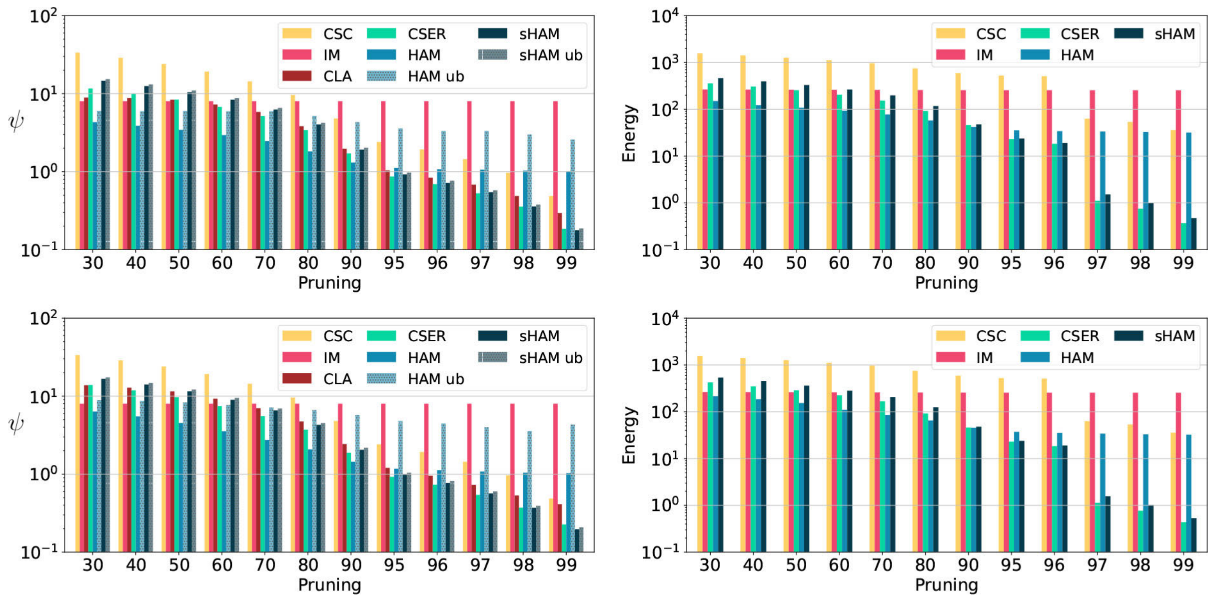

FIGURE 5. Performance obtained when compressing the dense layers of the *VGG19* network. Horizontal axis shows the empirical weight percentile of pruning. Matrices are quantized using $k = 32$ (first row) and $k = 256$ (second row) distinct weights. The "ub" suffix has the same meaning of Figure 3. Values are reported in logarithmic scale.

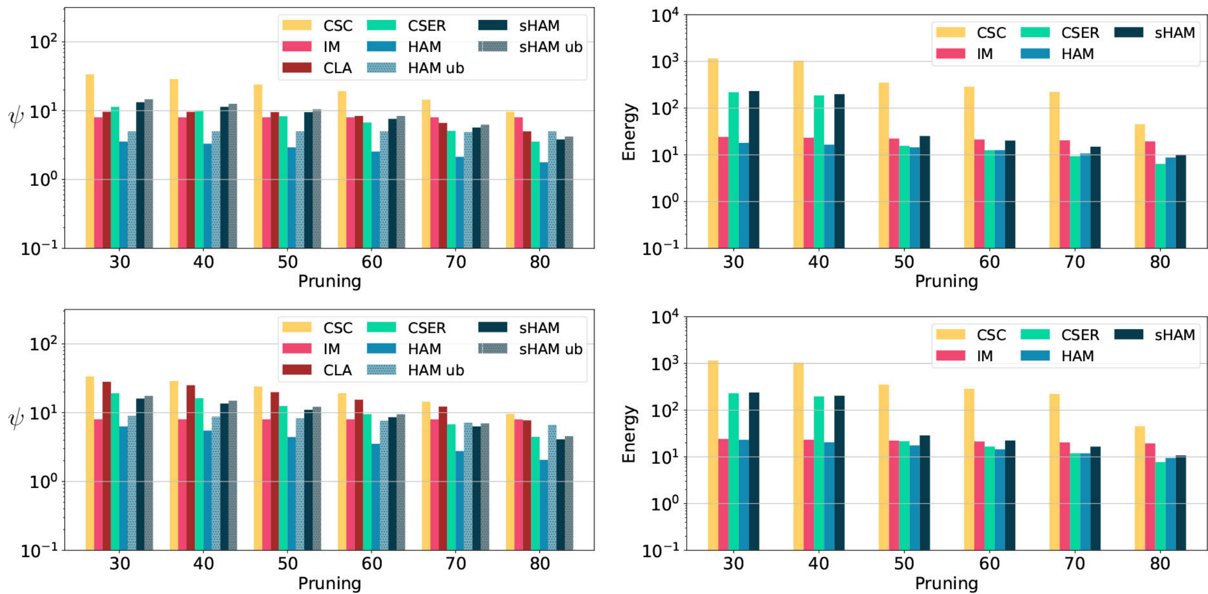

FIGURE 6. Performance obtained when compressing the dense layers of the *DeepDTA* network. Same notation used in Figure 5. Matrices are quantized using $k = 16$ (first row) and $k = 256$ (second row) distinct weights.

96, 97, 98, 99}, $k \in \{32, 256\}$ for *VGG19*, and $p \in \{30, 40, 50, 60, 70, 80\}$, $k \in \{16, 256\}$ for *DeepDTA*;
- we use *unified* quantization, i.e., sharing the k distinct weights across all the compressed network layers.

The latter property allows HAM and sHAM to use a unique Huffman code for all network layers, thereby sensibly reducing the related overhead.

b: SOFTWARE IMPLEMENTATION

The code retrieved *VGG19* and *DeepDTA* is implemented in Python 3, using Tensorflow and Keras. We use the same environment for implementing and applying the pruning and quantization techniques described in Section II-B. The resulting software is distributed as a standalone Python 3 package.[7]

c: RESULTS

Figures 5 and 6 summarize the results on *VGG19* and *DeepDTA*, respectively, in terms of space and energy requirements, whereas the execution time is shown in Figure 7. Moreover, in Tables 5 and 6 we show the obtained compression ratios. On both DNNs, HAM always has the lowest space

[7]Source code, datasets and trained baseline networks are available at https://github.com/AnacletoLAB/sHAM

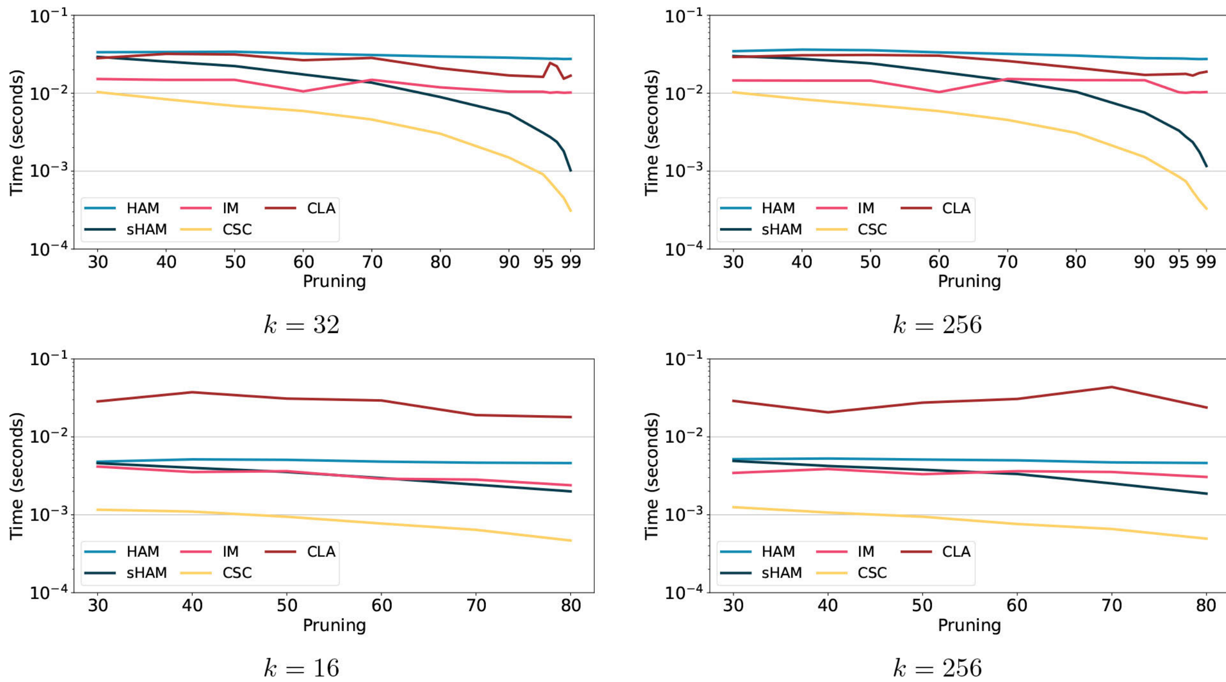

FIGURE 7. Average execution time (in seconds) for computing the dense layers of *VGG19* (first row) and *DeepDTA* (second row). Layer weights undergo pruning(with the levels shown on the X axis) and quantization using *k* distinct weights for all layers (shown under each graphic).

requirement, except when applying a very high pruning level (say, $p > 90$, or equivalently $s \leq 0.9$) to *VGG19*. To some extent, this confirms the results obtained on benchmark matrices, but here such a trend is even more marked, since CLA is no more competitive with HAM. The reasons of this result likely reside in the following properties: i) the sparsity is, in any case, not high enough to allow sparse formats to become more efficient; ii) HAM is indirectly able to leverage sparsity, since the source entropy becomes lower when one symbol is largely over-represented (as in this case 0 is always represented using just 1 bit); iii) HAM does not need any additional structure to store indices, whereas the other formats should use auxiliary index vectors for the matrix of each compressed layer (apart CLA, which adaptively selects the formats to be used); iv) lower k values induce shorter Huffman decoding arrays; v) HAM (like sHAM) can use a unique Huffman code for all compressed network layers, and the bigger the number of such layers, the higher the corresponding benefit.

The lack of additional structures, instead, makes the dot product computation in HAM slower; however, considering that our formats have been developed and optimized mainly to gain in terms of compression ratios when pruning and quantization is applied, and not directly to achieve time efficiency, it is quite surprising to see that the time of HAM dot product is close to that of CLA in this setting, or even better (*DeepDTA*). CSC and IM are faster, but they are the worst methods in terms of memory requirements (CSC for $p < 80$, IM in the remaining cases). An analogous situation occurs with reference to energy consumption: CSC is the worst method for all pruning levels (except for $p > 96$ on *VGG19*), while IM becomes the worst method in the remaining cases. It is worth nothing that there are some marked drops in the energy for contiguous choices of the pruning level, e.g., from $p = 96$ to $p = 97$ for both sHAM and CSC when using *VGG19*, and analogously in *DeepDTA* (now from $p = 70$ to $p = 80$ for CSC and from $p = 40$ to $p = 50$ for sHAM). Such sudden jumps are due to the change of the memory size of the operands involved in the elementary operations listed in Table 2, which, as already mentioned, directly impacts on the energy cost. HAM has the lowest energy consumption till $p = 90$ for *VGG19* and around $p = 70$ for *DeepDTA*.

When the weight matrices are highly sparse ($p > 90$) CSER and sHAM are, as expected, the formats which realize the best compression in terms of space. Indeed, the compression ratios obtained by the two methods are very similar, and they are definitely higher than those of the remaining compression formats. More precisely, CSER is the top performer when pruning is not pushed to an extreme level ($95 \leq p \leq 97$), while sHAM becomes the best method when sparsification is very high ($p \geq 98$). In general, CSER is more energy-efficient, behaving slightly better than sHAM for the overall class of sparse weight matrices (that is, $p > 90$), although sHAM is the fastest among all methods (with the only exception of CSC, which anyway is not viable in terms of space and energy). Interestingly, although we only compress dense layers, the overall size of VGG19 (Table 5) can be

TABLE 5. Compression ratios obtained by the storage formats compared in this study on *VGG19* model. Different configurations of pruning and quantization are applied to the dense layers of the network. Column *Config* is in the format *p-k*. Values outside (inside) parentheses are the compression ratios w.r.t. only dense layers (all layers). Index map method, whose space is constant *k* <= 256, achieves a compression ratio of ≃ 4.00 (1.57) on this data. In boldface font the best results for each configuration.

VGG19-MNIST					
Config	HAM	sHAM	CSER	CSC	CLA
30-32	**7.426** (**1.725**)	2.188 (1.358)	2.751 (1.448)	0.953 (0.976)	3.594 (1.540)
30-256	**5.034** (**1.638**)	1.918 (1.303)	2.300 (1.285)	0.953 (0.976)	2.317 (1.382)
40-32	**8.252** (**1.745**)	2.563 (1.421)	3.206 (1.502)	1.111 (1.051)	3.645 (1.544)
40-256	**5.820** (**1.673**)	2.269 (1.547)	2.694 (1.44)	1.111 (1.051)	2.490 (1.410)
50-32	**9.326** (**1.766**)	3.066 (1.487)	3.817 (1.469)	1.333 (1.138)	3.833 (1.560)
50-256	**7.084** (**1.716**)	2.777 (1.451)	3.299 (1.512)	1.333 (1.138)	2.781 (1.452)
60-32	**10.881** (**1.790**)	3.833 (1.560)	4.737 (1.621)	1.667 (1.241)	4.412 (1.602)
60-256	**9.001** (**1.760**)	3.571 (1.538)	4.283 (1.593)	1.667 (1.241)	3.443 (1.526)
70-32	**13.007** (**1.813**)	5.106 (1.641)	6.210 (1.688)	2.222 (1.365)	5.493 (1.659)
70-256	**11.639** (**1.799**)	4.881 (1.629)	5.775 (1.671)	2.222 (1.365)	4.563 (1.611)
80-32	**17.611** (**1.846**)	7.952 (1.738)	9.394 (1.767)	3.332 (1.515)	8.364 (1.747)
80-256	**15.390** (**1.832**)	7.466 (1.726)	8.604 (1.752)	3.332 (1.515)	6.762 (1.706)
90-32	**24.468** (**1.873**)	16.712 (1.841)	18.695 (1.851)	6.659 (1.703)	16.244 (1.838)
90-256	**22.150** (**1.865**)	15.597 (1.834)	17.022 (1.843)	6.659 (1.703)	13.158 (1.815)
95-32	28.697 (1.883)	34.705 (1.893)	**36.994** (**1.896**)	13.300 (1.816)	30.799 (1.887)
95-256	27.210 (1.880)	32.554 (1.890)	**34.634** (**1.893**)	13.300 (1.816)	26.578 (1.878)
96-32	29.845 (1.885)	44.498 (1.904)	**46.340** (**1.906**)	16.613 (1.840)	38.186 (1.898)
96-256	28.448 (1.882)	41.461 (1.902)	**43.696** (**1.904**)	16.613 (1.840)	33.508 (1.892)
97-32	30.204 (1.886)	58.564 (1.914)	**60.672** (**1.915**)	22.124 (1.865)	46.852 (1.906)
97-256	29.636 (1.885)	56.468 (1.913)	**58.938** (**1.914**)	22.124 (1.865)	43.896 (1.904)
98-32	31.033 (1.887)	**89.919** (**1.925**)	89.366 (1.925)	33.106 (1.891)	65.708 (1.917)
98-256	30.653 (1.887)	**86.277** (**1.924**)	86.029 (1.924)	33.106 (1.891)	59.925 (1.915)
99-32	31.683 (1.889)	**180.845** (**1.935**)	173.119 (1.934)	65.740 (1.917)	108.475 (1.928)
99-256	31.071 (1.888)	**162.559** (**1.934**)	141.204 (1.932)	65.740 (1.917)	77.670 (1.922)

almost halved, and that of DeepDTA (Table 6) reduced more than 5.5×.

Finally, it is worth noting that Figures 5 and 6 also show the space upper bounds introduced in Sects III-A and III-B, evaluated using Corollaries 1 and 2. Confirming the trends of benchmark data, HAM upper bounds are looser w.r.t. sHAM, and this gap consistently tends to augment with the sparsity, for the reasons already explained in the text. In this sense, the assumption of weights having the same frequency (used to compute upper bounds) seems far from being realistic, and in practice the proposed formats are definitely more effective. Tighter bounds might be found if more practical assumptions are considered, which could be the subject of future studies.

TABLE 6. Compression ratios on *DeepDTA* model. Same notation of Table 5. Index map in this case exhibits a (constant) compression ratio of ≃ 4.00 (2.93).

DeepDTA-DAVIS					
Config	HAM	sHAM	CSER	CSC	CLA
30-16	**8.993** (**4.555**)	2.412 (2.058)	2.812 (1.908)	0.952 (0.958)	3.319 (2.587)
30-256	**5.071** (**3.389**)	1.998 (1.781)	1.668 (1.657)	0.952 (0.958)	1.138 (1.119)
40-16	**9.657** (**4.698**)	2.807 (2.300)	3.252 (2.551)	1.110 (1.096)	3.332 (2.594)
40-256	**5.833** (**3.670**)	2.358 (2.023)	1.969 (1.761)	1.110 (1.096)	1.276 (1.235)
50-16	**10.924** (**4.943**)	3.367 (2.613)	3.853 (2.859)	1.332 (1.280)	3.356 (2.607)
50-256	**7.217** (**4.105**)	2.907 (2.864)	2.557 (2.149)	1.332 (1.280)	1.611 (1.500)
60-16	**12.577** (**5.216**)	4.206 (3.024)	4.753 (3.261)	1.665 (1.540)	3.813 (2.839)
60-256	**9.052** (**4.568**)	3.722 (2.794)	3.360 (2.609)	1.665 (1.540)	2.073 (1.833)
70-16	**14.959** (**5.536**)	5.623 (3.596)	6.303 (3.828)	2.219 (1.932)	4.816 (3.287)
70-256	**11.558** (**5.053**)	5.063 (3.386)	4.721 (3.247)	2.219 (1.932)	2.608 (2.181)
80-16	**18.008** (**5.859**)	8.382 (4.412)	9.012 (4.559)	3.327 (2.592)	6.399 (3.859)
80-256	**15.495** (**5.599**)	7.794 (4.263)	7.176 (4.094)	3.327 (2.592)	4.132 (2.990)

V. DISCUSSION

The experiments investigating the role of the CHC variants in the performances obtained by HAM and sHAM reveal that, in general, the *partial* variant emerges as a general purpose solution for building these compression formats, representing a good compromise among the used performance metrics. Indeed, it generally exhibits analogous results w.r.t. the other variants in terms of compression ratio and energy consumption, meanwhile optimizing the execution time of the dot product operations. As exception to this generic advice, the *bitwise* variant attains a faster dot product execution when working in a low entropy regime, while the *full* variant performs the best energy optimization.

In the remaining experiments, concerning the compression of benchmark matrices and of two DNNs, HAM emerges as the most efficient format to compress matrices with low to moderate sparsity levels (roughly $p \leq 90$). Indeed, while it is slightly outperformed by CLA (in terms of compression ratio) on two of the considered benchmark matrices (*Covtype* and *Census*), on DNNs data it is to a great degree better than CLA:the best compression for both formats is obtained on VGG19 when $p = 90$ and $k = 32$, with a compression ratio of 24.468× for HAM vs 16.712× for CLA, and 18.008× (HAM) vs 6.399× (CLA) on DeepDTA ($p = 80$, $k = 16$).

The dot execution time of HAM is related to the size of Huffman decoding structures. Indeed, HAM is slower than CLA on benchmark data where k values often are larger, but its execution time is similar (VGG19) or even faster (DeepDTA) than that of CLA when performing the DNN inference. IM needs denser matrices and values of $k \leq 256$ to be competitive.Indeed, on benchmark matrices, which do not have such characteristics, it poorly performs in terms of both space and energy. On the other side, it becomes in both metrics the second method on DNNs in all $p \leq 50$ settings. Conversely, sHAM and CSER are less competitive in such cases, but they can exploit increasing levels of sparsity more than CLA, HAM and IM. Actually, with high levels of sparsity (roughly $p > 90$), sHAM and CSER are, by far, the top methods, performing very similarly in terms of both space and energy, with sHAM compressing slightly more for more extreme pruning ($p > 97$ of VGG19 and benchmark matrices *orsreg_1* and *SiNa*), at the expense of a slender higher energy consumption. The best compression ratios for these two methods are 180.845× and 466.08× (sHAM) vs 173.119× and 277.61× (CSER), respectively on VGG19 ($p = 99$, $k = 32$) and *orsreg_1* data. In this setting, sHAM is even faster than CLA (we remind no implementation is provided for CSER), a nice feature considering also its compression capabilities.

A special mention is reserved for the DNN compression, where the possibility to share the Huffman code across all FC layers substantially boosts the space efficiency of HAM and sHAM: e.g., see how HAM improves w.r.t. CLA from benchmark to DNN setting. As in the last decade the most effective solutions in various application domains have been achieved through deeper and deeper neural networks, it becomes a fundamental feature that HAM and sHAM take more advantage in terms of compression ratios over the other methods as the depth of the models increases.

VI. CONCLUSION

We have presented two new lossless compression methods for real-valued matrices, Huffman Address Map (HAM) and sparse Huffman Address Map (sHAM), able to benefit from low-entropy inputs (e.g., induced via sparsification and quantization of the matrix) to attain high compression ratios and energy savings, and to perform the vector-matrix multiplication directly in the compressed format. We have derived theoretical upper bounds to their space usage in terms of the entropy of the input matrix. Moreover, the proposed formats are particularly suitable to compress deep neural networks, since the deeper the network, the higher their efficiency. An extensive set of experiments, to compress benchmark matrices and two public deep neural networks, favourably compared our proposals with regard to state-of-the-art solutions.

As a future work, an alternative source coding in place of Huffman coding could be investigated, like asymmetric numeral systems [66], largely used in data compression due to their elevated performance. Moreover, a potential limitation of HAM usage resides in the execution time of vector-matrix multiplication, inherently induced by the sequential nature of the decoding procedure. Although we have already made it more competitive with existing solutions through a multi-threaded extension, supplemental effort could be devoted in designing and implementing variants which can be executed on modern GPUs, so as to further increase their time efficiency.

REFERENCES

[1] A. Hazra, P. Choudhary, and M. S. Singh, "Recent advances in deep learning techniques and its applications: An overview," in *Advances in Biomedical Engineering and Technology*, A. A. Rizvanov, B. K. Singh, and P. Ganasala, Eds. Singapore: Springer, 2021, pp. 103–122.

[2] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, Y. Gong, and A. Acero, "Recent advances in deep learning for speech research at Microsoft," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 8604–8608.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25. Red Hook, NY, USA: Curran Associates, 2012, pp. 1097–1105.

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning affordance for direct perception in autonomous driving," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 2722–2730.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[6] Z. Allen-Zhu, Y. Li, and Y. Liang, "Learning and generalization in overparameterized neural networks, going beyond two layers," in *Advances in Neural Information Processing Systems*, vol. 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2019.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2015, pp. 1–14.

[8] S. Afroz, M. Tahaseen, F. Ahmed, K. S. Farshee, and M. N. Huda, "Survey on matrix multiplication algorithms," in *Proc. 5th Int. Conf. Informat., Electron. Vis. (ICIEV)*, May 2016, pp. 151–155.

[9] I. S. Duff, "A survey of sparse matrix research," *Proc. IEEE*, vol. 65, no. 4, pp. 500–535, Apr. 1977.

[10] G. C. Marinó, G. Ghidoli, M. Frasca, and D. Malchiodi, "Compression strategies and space-conscious representations for deep neural networks," in *Proc. 25th Int. Conf. Pattern Recognit. (ICPR)*, Jan. 2021, pp. 9835–9842, doi: 10.1109/ICPR48806.2021.9412209.

[11] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[12] G. C. Marinó, A. Petrini, D. Malchiodi, and M. Frasca, "Deep neural networks compression: A comparative survey and choice recommendations," *Neurocomputing*, vol. 520, pp. 152–170, Feb. 2023, doi: 10.1016/j.neucom.2022.11.072.

[13] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," 2017, *arXiv:1710.09282*.

[14] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, Jul. 1948.

[15] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for large-scale machine learning," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 960–971, Aug. 2016.

[16] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for declarative large-scale machine learning," *Commun. ACM*, vol. 62, no. 5, pp. 83–91, Apr. 2019.

[17] A. Francisco, T. Gagie, S. Ladra, and G. Navarro, "Exploiting computation-friendly graph compression methods for adjacency-matrix multiplication," in *Proc. Data Compress. Conf.*, Mar. 2018, pp. 307–314.

[18] S. Wiedemann, K.-R. Müller, and W. Samek, "Compact and computationally efficient representation of deep neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 3, pp. 772–785, Mar. 2020.

[19] P. Ferragina, *Pearls of Algorithm Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 2022.
[20] A. Rényi, "On measures of entropy and information," in *Proc. 4th Berkeley Symp. Math. Statist. Probab.*, vol. 1, 1961, pp. 547–561.
[21] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst. (NIPS)*. Red Hook, NY, USA: Curran Associates, 2016, pp. 2082–2090.
[22] J.-H. Luo, H. Zhang, H.-Y. Zhou, C.-W. Xie, J. Wu, and W. Lin, "ThiNet: Pruning CNN filters for a thinner net," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 10, pp. 2525–2538, Oct. 2019.
[23] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *Computer Vision—ECCV 2018*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds., Cham, Switzerland: Springer, 2018, pp. 191–207.
[24] D. Stathakis, "How many hidden layers and nodes?" *Int. J. Remote Sens.*, vol. 30, no. 8, pp. 2133–2147, 2009.
[25] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, "AutoCompress: An automatic DNN structured pruning framework for ultra-high compression rates," in *Proc. AAAI Conf. Artif. Intell.*, 2019, vol. 34, no. 4, pp. 4876–4883.
[26] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *Advances in Neural Information Processing Systems*, vol. 27. Red Hook, NY, USA: Curran Associates, 2014.
[27] R. Müller, S. Kornblith, and G. E. Hinton, "When does label smoothing help?" in *Advances in Neural Information Processing Systems*, vol. 32. Red Hook, NY, USA: Curran Associates, 2019.
[28] F. Mamalet and C. Garcia, "Simplifying ConvNets for fast learning," in *Artificial Neural Networks and Machine Learning—ICANN*, A. E. P. Villa, W. Duch, P. Érdi, F. Masulli, and G. Palm, Eds. Berlin, Germany: Springer, 2012, pp. 58–65.
[29] S. Swaminathan, D. Garg, R. Kannan, and F. Andres, "Sparse low rank factorization for deep neural network compression," *Neurocomputing*, vol. 398, pp. 185–196, Jul. 2020.
[30] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 2, 1989, pp. 598–605.
[31] M. Hagiwara, "Removal of hidden units and weights for back propagation networks," in *Proc. Int. Conf. Neural Net. (IJCNN-Nagoya, Japan)*, vol. 1, 1993, pp. 351–354.
[32] T. Zhang, X. Ma, Z. Zhan, S. Zhou, C. Ding, M. Fardad, and Y. Wang, "A unified DNN weight pruning framework using reweighted optimization methods," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 493–498.
[33] J. Chang, Y. Lu, P. Xue, Y. Xu, and Z. Wei, "Iterative clustering pruning for convolutional neural networks," *Knowl.-Based Syst.*, vol. 265, Apr. 2023, Art. no. 110386.
[34] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems*, vol. 29, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2016.
[35] M. Á. Carreira-Perpiñán and Y. Idelbayev, "Model compression as constrained optimization, with application to neural nets. Part V: Combining compressions," 2021, *arXiv:2107.04380*.
[36] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. ICLR*, 2016, pp. 1–14.
[37] H. Gish and J. Pierce, "Asymptotically efficient quantizing," *IEEE Trans. Inf. Theory*, vol. IT-14, no. 5, pp. 676–683, Sep. 1968.
[38] Y. Choi, M. El-Khamy, and J. Lee, "Universal deep neural network compression," *IEEE J. Sel. Topics Signal Process.*, vol. 14, no. 4, pp. 715–726, May 2020.
[39] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*. Norwell, MA, USA: Kluwer, 1991.
[40] P. A. Chou, T. Lookabaugh, and R. M. Gray, "Entropy-constrained vector quantization," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 37, no. 1, pp. 31–42, Jan. 1989.
[41] G. C. Marinò, G. Ghidoli, M. Frasca, and D. Malchiodi, "Reproducing the sparse Huffman address map compression for deep neural networks," in *Reproducible Research in Pattern Recognition*. Cham, Switzerland: Springer, 2021, pp. 161–166, doi: 10.1007/978-3-030-76423-4_12.
[42] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: SIAM, 2003.
[43] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
[44] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. Inst. Radio Eng.*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
[45] A. Moffat, "Huffman coding," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 1–35, Aug. 2019.
[46] A. Moffat and A. Turpin, "On the implementation of minimum redundancy prefix codes," *IEEE Trans. Commun.*, vol. 45, no. 10, pp. 1200–1207, Oct. 1997.
[47] Z. Sultana and S. Akter, "A new approach of a memory efficient Huffman tree representation technique," in *Proc. Int. Conf. Informat., Electron. Vis. (ICIEV)*, May 2012, pp. 731–736.
[48] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 1–49, Jan. 2017.
[49] N. Bell and M. Garl, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corp., Tech. Rep. NVR-2008-004, 2008.
[50] M. R. Hugues and S. G. Petiton, "Sparse matrix formats evaluation and optimization on a GPU," in *Proc. IEEE 12th Int. Conf. High Perform. Comput. Commun. (HPCC)*, Sep. 2010, pp. 122–129.
[51] J. Tian, C. Rivera, S. Di, J. Chen, X. Liang, D. Tao, and F. Cappello, "Revisiting Huffman coding: Toward extreme performance on modern GPU architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2021, pp. 881–891.
[52] S. Albers, "Energy-efficient algorithms," *Commun. ACM*, vol. 53, no. 5, pp. 86–96, May 2010.
[53] M. Rashid, L. Ardito, and M. Torchiano, "Energy consumption analysis of algorithms implementations," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2015, pp. 1–4.
[54] Y. S. Abu-Mostafa and R. J. McEliece, "Maximal codeword lengths in Huffman codes," *Comput. Math. With Appl.*, vol. 39, no. 11, pp. 129–134, Jun. 2000.
[55] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 10–14.
[56] N. Frid, D. Ivošević, and V. Sruk, "Elementary operations: A novel concept for source-level timing estimation," *Automatika*, vol. 60, no. 1, pp. 91–104, Jan. 2019.
[57] P. Ferragina, G. Manzini, T. Gagie, D. Köppl, G. Navarro, M. Striani, and F. Tosoni, "Improving matrix-vector multiplication via lossless grammar-compressed matrices," *Proc. VLDB Endowment*, vol. 15, no. 10, pp. 2175–2187, Jun. 2022.
[58] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Dec. 2011.
[59] D. Dua and C. Graff, "UCI machine learning repository," 2017. Accessed: Dec. 10, 2022. [Online]. Available: http://archive.ics.uci.edu/ml
[60] R. Chitta, R. Jin, T. C. Havens, and A. K. Jain, "Approximate kernel k-means: Solution to large scale kernel clustering," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 895–903.
[61] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *Proc. 5th Conf. Comput. Frontiers*, May 2008, pp. 86–97.
[62] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.
[63] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[64] H. Öztürk, A. Özgür, and E. Ozkirimli, "DeepDTA: Deep drug–target binding affinity prediction," *Bioinformatics*, vol. 34, no. 17, pp. i821–i829, Sep. 2018.
[65] M. I. Davis, J. P. Hunt, S. Herrgard, P. Ciceri, L. M. Wodicka, G. Pallares, M. Hocker, D. K. Treiber, and P. P. Zarrinkar, "Comprehensive analysis of kinase inhibitor selectivity," *Nature Biotechnol.*, vol. 29, no. 11, pp. 1046–1051, Nov. 2011.
[66] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, "The use of asymmetric numeral systems as an accurate replacement for Huffman coding," in *Proc. Picture Coding Symp. (PCS)*, May 2015, pp. 65–69.

GIOSUÈ CATALDO MARINÒ received the B.Sc. degree in computer science from the University of Milan, in 2021, where he is currently pursuing the master's degree in computer science. His research interests include machine learning and compression of neural network models.

FLAVIO FURIA received the master's and Ph.D. degrees in computer science from the University of Milan, in 2022. His master's dissertation was titled, "Centrality in Undirected Networks: Proofs and Counterexamples." His research interests revolve around the study of graph theory and network centrality. In particular, he is currently focusing on how different centrality measures react to edge additions in terms of score and rank monotonicity of nodes.

DARIO MALCHIODI received the M.Sc. degree in computing and the Ph.D. degree in computational mathematics and operations research from the University of Milan, in 1997 and 2000, respectively. Since 2002, he has been an Assistant Professor with the Department of Computer Science, University of Milan, where he was an Associate Professor, in 2011. He teaches statistics and data analysis and algorithms for massive datasets. He is the author of about 100 scientific publications and is actively involved in the popularization of computing. His research interests include the treatment of uncertainty in machine learning, with a particular focus on data-driven induction of fuzzy sets, compression of machine-learning models, mining of knowledge bases in the semantic web, negative example selection in bioinformatics, and application of machine learning to the medical, veterinary, and forensics fields.

MARCO FRASCA received the Ph.D. degree in computer science from the University of Milan, in 2012. Since 2017, he has been an Assistant Professor with the Department of Computer Science, University of Milan. He has been an Invited Research Visitor at several universities, including the Terrence Donnelly Centre for Cellular and Biomolecular Research, the University of Toronto, and the Institute of Molecular Biology, Johannes Gutenberg University of Mainz. He contributed to consolidating the application of Hopfield networks to classification and ranking problems with the development of single- and multi-task parametric Hopfield models. His research interests include the design and analysis of new machine-learning methods, with applications in bioinformatics, computational biology, and medicine.

Open Access funding provided by 'Università degli Studi di Milano' within the CRUI CARE Agreement