High-Dimensional Computing with Sparse Vectors

4 authors, including:

Jussi Poikonen
Rolls-Royce Marine
65 PUBLICATIONS **580** CITATIONS

Eero Lehtonen
University of Turku
54 PUBLICATIONS **937** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project MinMotion (Academy of Finland) View project

Project EUROPEAN COST 2100 View project

High-Dimensional Computing with Sparse Vectors

Mika Laiho*‡, Jussi H. Poikonen*, Pentti Kanerva‡, Eero Lehtonen*

* Technology Research Center, University of Turku, Finland. E-mail: mlaiho@utu.fi

‡Redwood Center for Theoretical Neuroscience, University of California, Berkeley, U.S.A

Abstract—Computing with high-dimensional vectors in a manner that resembles computing with numbers is based on Plate's Holographic Reduced Representation (HRR) and is used to model human cognition. Here we examine its hardware realization under constraints suggested by the properties of the brain's circuits. The sparseness of neural firing suggests that the vectors should be sparse. We show that the HRR operations of addition, multiplication, and permutation can be realized with sparse vectors, making an energy-efficient implementation possible. Furthermore, we propose a processor that has both data and instructions embedded in the same high-dimensional vector. The operation is highlighted with a sequence memory example.

I. INTRODUCTION

The brain uses sparse representation of information and computes in high-dimensional (High-D) spaces. Work towards understanding the related representation of information and computing has yielded new computing methods inspired by the brain, of which artificial neural networks, or parallel distributed processing, is the most prominent example. In contrast to neural networks, where learning is inherently statistical, in this work we focus on high-dimensional arithmetic which allows logical manipulation of structured distributed data, specifically High-D random vectors. High-D computing, also known as vector symbolic arithmetic (VSA) has been demonstrated to facilitate for example indirect reasoning, analogy and learning from (a single) example [1]. In High-D computing the idea is to represent information objects with High-D random vectors and to compose structure and logical operations between such objects using a set of vector-arithmetic operations. A key benefit in computing with random vectors is their tolerance to error: information is stored holistically, i.e., distributed so that rather than assigning significance to individual vector elements (bits), the meaning of a data item is represented by a population of bits. Thus it is possible to change the values of some of the vector elements without changing the perceived meaning of the vector.

The key operations in high-dimensional arithmetic are *addition/sum*, *multiplication/product* and *permutation/scramble*. Addition takes two (or more) vectors as input and outputs a vector that resembles (in terms of inner product) each of the input vectors, whereas multiplication takes two (or more) vectors as input and outputs a vector that resembles none of the inputs. Permutation is like multiplication in that its output

does not resemble its input. Addition is used for encoding sets, permutation for encoding sequences, and multiplication for binding variables to values. Multiplication/binding operations realized in earlier studies (e.g. [2], [1]) have worked with *dense* vectors. For example, in real-valued holographic reduced representation (HRR) of [2] the elements of the vectors are i.i.d. with mean zero and variance $1/N$, where N is the vector dimension, and the binding operation is circular convolution. Binary Spatter Codes (BSC) [1] use dense High-D binary vectors with on average half of the bits having value 1, and bind with elementwise XOR. However, with *sparse* vectors, which contain only a small fraction of non-zeros, binding is problematic with both HRR and BSC — it is easy to see for example that with sparse binary vectors XOR reduces sparseness, and the result resembles the input vectors.

In this paper we aim at High-D computing with a heteroassociative processor that operates on sparse random binary vectors. There are two main motivations for using sparse data. 1) *Energy efficiency*: when using dense vectors for example as inputs to search operations, on average half of the bit values must be flipped between searches, whereas with sparse vectors only a small fraction of bits need to be flipped. With a bus thousands of bits wide, this has a great impact on power consumption. 2) *Memory capacity*. Sparse vectors can be stored in a distributed fashion so that the vector capacity (number of correctly recoverable stored vectors) is much higher than the number of rows in the memory [3]). Also, distributed storage is robust to e.g. device defects and noise.

In the following we propose a novel binding operation that works robustly with sparse binary vectors, and describe related permutation, sumset and thinning operations. The representation used in this paper is a generalization of Plate's HRR in the frequency domain [2]. Plate's HRR in frequency domain can be approximated by limiting the phase angles to a discrete set as explained in [5]. There the vector components are phase angles (i.e., complex numbers) and multiplication (binding) is equivalent to the addition of the phase angles. Our approach allows for a constellation of angles to be represented by an L-bit vector (segment), and binding is realized as a circular bit shift (rotation of the constellation). We also propose a sparse heteroassociative processor model that has both data and instructions embedded in the High-D vector. The operation is highlighted with a sequence memory Matlab simulation.

II. HIGH-D ARITHMETIC WITH SPARSE VECTORS

Let **A** be a binary vector of length N, made of S L-bit segments ($N = SL$). We refer to bit $j \in 0, 1, \ldots, L-1$
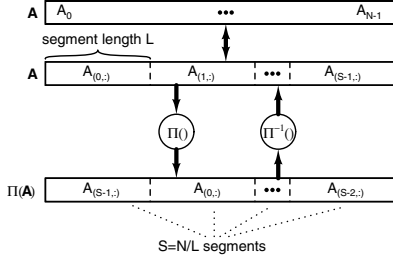
Fig. 1. Division of vector into segments and illustration of permutation and inverse permutation operations.

in segment $s \in 0, 1, \ldots, S - 1$ by $\mathbf{A}_{(s,j)}$, and to an entire segment by $\mathbf{A}_{(s,:)}$. Fig. 1 shows an illustration of this division into segments. A vector is in the following called *maximally sparse* if only a single bit location in each of the segments $\mathbf{A}_{(s,:)}$ is one – this represents an angle in terms of frequency domain HRR. Segments with multiple ones represent multiple (constellation of) angles. Given two segmented random vectors with a sufficient number and length of segments, the inner product between these is close to zero — the vectors do not resemble each other. Next we describe permutation, sumset, thinning, and binding operations for sparse segmented vectors.

A. Permutation Operation

We assume a block permutation operation $\Pi\mathbf{A}$, which shifts data segments circularly as

$$\Pi\mathbf{A}_{(s,:)} :\equiv \mathbf{A}_{(s+1 \ (S),:)}, \tag{1}$$

where (S) denotes a modulo S operation. The corresponding inverse permutation operation moves the data segments circularly in the opposite direction.

$$\Pi^{-1}\mathbf{A}_{(s,:)} :\equiv \mathbf{A}_{(s-1 \ (S),:)}. \tag{2}$$

These operations are illustrated in Fig. 1

B. Sumset Operation and Thinning

The sumset operation of sparse vectors $\mathbf{A}$ and $\mathbf{B}$ can be computed as the elementwise sum of the vectors, and is denoted by the + symbol. The resulting vector resembles both $\mathbf{A}$ and $\mathbf{B}$. Even though this sum operation increases the number of nonzero entries from the defined target S, the considered permutation and binding operations can be applied also with vectors of reduced sparsity. However, after aggregating information from multiple vectors and performing arithmetic operations, it is usually desirable to perform *thinning* to reach desired sparsity level. This can be achieved simply by selecting randomly one of the nonzero bits in each segment, or by a sumset operation which realizes integer (counter) summation instead of binary OR. In this case, the maximum value(s) of each segment are assigned value 1 (note that some entries can have values larger than 1), and all other entries are set to 0. The resulting vector has at least one 1 per segment — one of these is randomly chosen to remain 1, while others are set to 0. Alternatively, the bits themselves can decide which one remains 1. For example, the indices of the bits

$$\mathbf{a}_s = \arg\max_j(\mathbf{A}_{(s,j)}), \tag{3}$$

can be summed modulo q, where q is the length of $\mathbf{a}_s$; let us denote the value of this modulo sum by p. Then the pth element of $\mathbf{a}_s$ is the index of the bit of $\mathbf{A}_{(s,j)}$ which is set to 1, while all the rest of the bits of $\mathbf{A}_{(s,j)}$ are set to zero. The benefit of this over a random selection is that it produces the same result every time (easier to analyse). Such a thinning is an example of Context-Dependent Thinning (CDT) of sparse vectors [4].

C. Binding Operation

We define the binding operation of two sparse random binary vectors $\mathbf{A}$ and $\mathbf{B}$ to a result vector $\mathbf{C}$ as follows. Let

$$a_s = \min(\arg\max_j(\mathbf{A}_{(s,j)})), \tag{4}$$

i.e., a_s is the smallest of the indices of the maxima of segment s, and α is a parameter with default value 1 (see discussion below). We denote the binding operation by $\otimes$, and define $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ so that bits in each segment of vector $\mathbf{B}$ are circularly shifted a_s locations:

$$\mathbf{C}_{(s,j)} :\equiv \mathbf{B}_{(s,j-\alpha a_s \ (L))}. \tag{5}$$

With maximally sparse vectors this is equivalent to modulo sum of the indices of the two operands as in frequency domain HRR [2], [5].

The corresponding inverse binding operation $\mathbf{B} = \mathbf{A} \oslash \mathbf{C}$ is defined as

$$\mathbf{B}_{(s,j)} :\equiv \mathbf{C}_{(s,j+\alpha a_s \ (L))}. \tag{6}$$

This binding operation is robust since erroneous bits or noise in a particular segment only affect the binding result of that segment. The binding operation is essentially a segment-wise permutation of $\mathbf{B}$. It has been shown in [1] that permutation preserves distance, distributes over addition and is invertible. The same characteristics apply here, with the exception that the segment-wise permutation left-distributes over addition. Furthermore, as required from the binding operation, the resulting vector $\mathbf{C}$ does not resemble either of the input vectors $\mathbf{A}$ and $\mathbf{B}$. If both vectors are maximally sparse (number of ones in both vectors is S), the binding operation commutes, i.e., $\mathbf{A} \otimes \mathbf{B} = \mathbf{B} \otimes \mathbf{A}$. Interestingly, if we set α to -1, the arithmetic properties of the binding and unbinding operation are interchanged: binding with $\alpha = -1$ does not commute (subtraction of indices does not commute), whereas unbinding with $\alpha = -1$ associates.

It should be noted that the authors of [4] use the word binding with a different meaning – they call context-dependent thinning a binding operation. In this paper we refer to binding as an invertible multiplication operation that produces an output that is not similar to the inputs as in HRR and BSC.

III. Sparse Heteroassociative Processor

Fig. 2 shows the block diagram of an associative processor that operates on sparse vectors. The core of the processor is an associative memory that has an autoassociative part (columns 0 to $N - 1$) and a heteroassociative part (columns

N to $2N - 1$). Vector $\mathbf{Z}$ acts as the search key to the autoassociative part of the memory. The memory can be implemented as a distributed memory such as a Willshaw memory [3]. Combining autoassociative and heteroassociative operation by increasing the word length was proposed in [6]. In [6] the heteroassociative data was used as a pointer to the next address to be searched, whereas in this work it can also contain metadata that can be used for many purposes such as control instructions (see below).

Data vectors $\mathbf{W}$ are passed to and from the associative memory through a vector arithmetic logic (ALU) unit that is capable of doing the operations described in Section II. The ALU has also access to multiple registers for vector storage.

A. Storing Items to the Contents Matrix

We denote the ith row of the contents matrix as $\mathbb{C}_i$. Each row in the contents matrix is a concatenation of two sparse segmented vectors of length N as defined in the previous section: $\mathbb{C}_i = [\mathbf{C_i}^{(A)} \ \mathbf{C_i}^{(H)}]$. These correspond to an autoassociative search key $\mathbf{C_i}^{(A)}$, and the heteroassociative part $\mathbf{C_i}^{(H)}$, which contains metadata associated with the search item. The metadata $\mathbf{Q}$ can contain e.g., an instruction, a vector that points to memory, or other metadata associated with the vector. The metadata is programmed to $\mathbf{C_i^{(H)}}$ in row i as

$$\mathbf{C_i^{(H)}} = \mathbf{C_i^{(A)}} \otimes \mathbf{Q}. \tag{7}$$

Binding is here used to keep $\mathbf{C_i^{(H)}}$ random even if the same metadata repeats in different rows – vectors stored in a distributed memory need to be random in order to maximize the capacity. With elementary items (for example symbols), the search key $\mathbf{C_i^{(A)}}$ is a random vector. With sequence items, the first item is a random vector, and subsequent vectors in the sequence are obtained as

$$\mathbf{C_{i+1}^{(A)}} = \mathbf{C_i^{(A)}} \otimes \Pi \mathbf{C_i^{(A)}} \tag{8}$$

Such a sequence item (i.e., sequence of associations) makes it possible to sequence through algorithms.

B. Associative Processing

Figure 3 shows a flowchart that illustrates the operation of the processor. The instructions come from the input, which is a sequence of N-bit vectors $\mathbf{D}_1, \mathbf{D}_2, \ldots$, and from the memory. When the processor is turned on, it starts from the "start" state. Next it reads an input vector $\mathbf{D}_1$ and performs an associative search with that vector. If there is a match, vector $\mathbf{W}$ is updated with the matching vector retrieved from the memory. $\mathbf{W}$ contains an autoassociative part $\mathbf{W}^{(A)}$ and a heteroassociative part $\mathbf{W}^{(H)}$. With the unbinding operation, the ALU extracts the metadata vector $\mathbf{Q}$:

$$\mathbf{Q} = \mathbf{W}^{(A)} \oslash \mathbf{W}^{(H)} \tag{9}$$

The metadata vector is passed to an instruction and data decoding block. If $\mathbf{Q}$ is an instruction, it enters the controller. The controller uses the elementary instruction to control the ALU, registers and I/O operations.
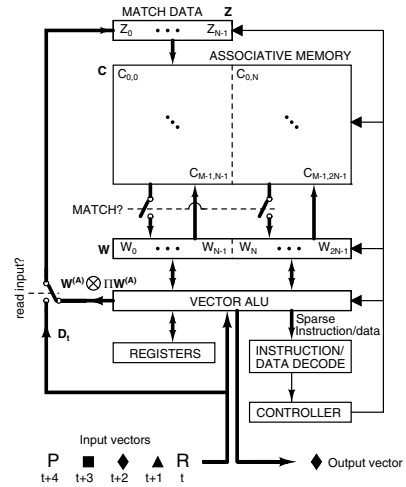

Fig. 2. Sparse heteroassociative processor. The thick lines convey the High-D vector data.
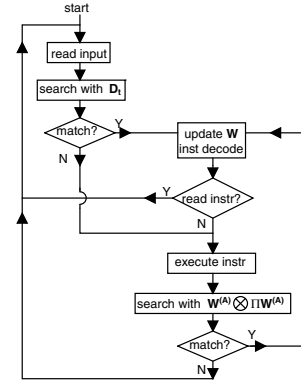

Fig. 3. Flowchart that illustrates the operation of the sparse heteroassociative processor.

If the decoded instruction is a read instruction, the processor returns to the start state, reads the next input vector, and performs another search. If the decoded instruction is other than a read, the controller executes the instruction. Then search is performed with $\mathbf{W}^{(A)} \otimes \Pi\mathbf{W}^{(A)}$. If there is a match, $\mathbf{W}$ is updated and the instruction is decoded. If there is not a match, $\mathbf{W}$ is not modified and the next input is read and processed. In other words, the processor sequences through a program until a new command is received from the input.

IV. PROCESSING EXAMPLE: SEQUENCE MEMORY

In order to demonstrate the binding operation in practice, we show how the processor can record a sequence of symbols, and then pick a symbol from the sequence. A similar experiment was solved with dense High-D vectors using HRR data representation on Semantic Pointer Architecture Unified Network (SPAUN) [7]; SPAUN recognizes task identifiers that switch the SPAUN to a state that runs through a pre-determined (fixed) sequence.

Here our approach is that, similarly to SPAUN, we give a pointer (task identifier) to the heteroassociative processor that means start recording a sequence. The pointer steps through a procedure that records the sequence and eventually

outputs a desired symbol from the sequence. The difference to SPAUN is that it can be *programmed*; programming can be done just by changing the instructions embedded in the heteroassociative memory; the whole operation is defined by the memory contents and vector inputs.

The bottom of Figure 2 shows a sequence of input vectors $\mathbf{D} = [\mathbf{R}, \blacktriangle, \blacklozenge, \blacksquare, \mathbf{P}]$. Vector $\mathbf{R}$ is a command that starts recording the sequence of symbols, while $\mathbf{P}$ is a command that picks the second symbol from the sequence. $\mathbf{R}$, $\mathbf{P}$, $\blacktriangle$, $\blacklozenge$ and $\blacksquare$ are sparse random binary vectors. In this particular case, vector corresponding to symbol $\blacklozenge$ is output.

Table I shows the memory contents needed for the sequence memory example. Each row i shows the information that is retrieved when the memory is "addressed" with a vector that resembles $\mathbf{C}_i^{(A)}$. Rows 1–7 read the sequence of input data vectors into rows 10 and 11, and rows 8 and 9 put out the answer (second symbol in the sequence).

The heteroassociative data in row i is $\mathbf{C}_i^{(H)}$, and the corresponding $\mathbf{Q}_i$ can be obtained using (9). The instruction set has eight commands, namely *READ12, READ3, WRITEA, WRITEH, PERMR, JUMP, SEARCH* and *RESULT*. In a realistic application, the instruction set would be much larger. The fourth column of the table shows how the controller acts upon the decoded instructions.

The procedure of storing the sequence starts by initializing parameters $t = 0$ and $m = 10$. Then an input vector $\mathbf{R}$ enters the vector ALU and is used to make an associative search. The controller extracts $\mathbf{Q_i}$ from $\mathbf{C}_i^{(H)}$. The instruction decoder decodes the instruction which in row 1 is *READ12*; it increases the input pointer t and reads the vector, permutes it, and writes to REG1 and REG2. Row 2 reads the next input vector and writes it to REG3. Row 3 instructs the controller to write REG2 to $\mathbf{C}_m^{(A)}$, and Row 4 to write REG3 $\otimes$ REG2 to $\mathbf{C}_m^{(H)}$, where m is the next available memory location (10 and 11 in the example). Row 5 stores REG3 $\otimes$ REG2 to REG2. Row 6 is *JUMP* instruction, while row 7 holds the jump target address; the program jumps back to row 2 to store the next symbol in the sequence, essentially forming a loop. The loop continues until an input vector is recognised as an instruction. Rows 10 and 11 show how the sequence of symbols in Fig. 2 would be represented in the memory.

$\mathbf{P}$ is a command that picks the second symbol of the stored sequence. The permuted first symbol $\Pi\blacktriangle$ in the sequence is stored in REG1, and is readily available by inverse permutation. Rows 8 and 9 show how the second symbol in the series can be found by searching with the contents of REG1, and unbinding the resulting vector $\mathbf{W}^{(H)}$ ($\blacklozenge \otimes \Pi\blacktriangle$ in the example) with REG1. The third symbol could be obtained by searching with $\Pi(\blacklozenge \otimes \Pi\blacktriangle)$, and again permuting $\mathbf{W}(H)$ with it.

The heteroassociative processor depicted in Figures 2 and 3 was modeled with Matlab. We simulated the sequence task with different vector lengths and levels of sparsity, using $\alpha = 1$. A content-addressable memory with $M = 10000$ was initialised with the 9 rows as shown in Table I, while the rest of the rows were random vectors. The statistical simulation of the sequence memory revealed a tradeoff between N and

sparsity level $1/L$: a 0.01% error probability was obtained as N was 100, 300 and 500, with 10%, 2% and 1% sparsity level, respectively. In more practical tasks with multiple terms combined with the sumset operation, the vectors need to be much longer as highlighted in [1]. However, even this simple simulation reveals how the representative power of the vectors reduces with sparsity. Therefore, it may be beneficial to compute the critical parts of a particular algorithm with vectors of reduced sparsity. The optimal choice of sparsity level with different algorithms is a topic of future work.

TABLE I
MEMORY CONTENTS FOR THE SEQUENCE MEMORY.

i	$\mathbf{C}_i^{(A)}$	$\mathbf{C}_i^{(H)} = \mathbf{C}_i^{(A)} \otimes \mathbf{Q}_i$	Decoded $\mathbf{Q}_i$ (Instruction)
1	$\mathbf{R}$	$\mathbf{R} \otimes READ12$	$t := t + 1$ REG1=ΠD_t REG2=ΠD_t
2	$\mathbf{R}(2) = \mathbf{R} \otimes \Pi\mathbf{R}$	$\mathbf{R}(2) \otimes READ3$	$t := t + 1$ REG3= D_t
3	$\mathbf{R}(3) = \mathbf{R}(2) \otimes \Pi\mathbf{R}(2)$	$\mathbf{R}(3) \otimes WRITEA$	$\mathbf{C}_m^{(A)}$ =REG2
4	$\mathbf{R}(4) = \mathbf{R}(3) \otimes \Pi\mathbf{R}(3)$	$\mathbf{R}(4) \otimes WRITEH$	$\mathbf{C}_m^{(H)} =$ REG3 $\otimes$ REG2
5	$\mathbf{R}(5) = \mathbf{R}(4) \otimes \Pi\mathbf{R}(4)$	$\mathbf{R}(5) \otimes PERMR$	REG2= $\Pi\mathbf{C}_m^{(H)}$ $m := m + 1$
6	$\mathbf{R}(6) = \mathbf{R}(5) \otimes \Pi\mathbf{R}(5)$	$\mathbf{R}(6) \otimes JUMP$	Jump to $\mathbf{Q}_{i+1}$
7	$\mathbf{R}(7) = \mathbf{R}(6) \otimes \Pi\mathbf{R}(6)$	$\mathbf{R}(7) \otimes \mathbf{R}(2)$	
8	$\mathbf{P}$	$\mathbf{P} \otimes SEARCH$	search w/ REG1 REG2=$\mathbf{W}^{(H)}$
9	$\mathbf{P}(2) = \mathbf{P} \otimes \Pi\mathbf{P}$	$\mathbf{P}(2) \otimes RESULT$	$OUT=$ REG1 $\oslash$ REG2
10	$\Pi\blacktriangle$	$\blacklozenge \otimes \Pi\blacktriangle$	
11	$\Pi(\blacklozenge \otimes \Pi\blacktriangle)$	$\blacksquare \otimes \Pi(\blacklozenge \otimes \Pi\blacktriangle)$	

V. CONCLUSIONS

In this paper we demonstrated that reliable symbolic processing that relies on high-dimensional, sparse, random binary vectors and a content-addressable memory is possible. The key to achieving this is a new permutation-based binding operation that operates on vectors that are divided to segments. Also, we proposed a heteroassociative processor model and highlighted that by embedding metadata into the vectors, the processor can be made programmable.

REFERENCES

[1] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation,*, vol. 1, no. 2, pp. 139–159, 2009.
[2] T. A. Plate, *Holographic Reduced Representation: Distributed Representation for Cognitive Structures*. Center for Study of Language and Information, 2003.
[3] D. J. Willshaw, O. P. Buneman, and H. C. Longuet-Higgins, "Non-holographic associative memory," *Nature*, vol. 222, pp. 960–962, 1969.
[4] D. A. Rachkovskij and E. M. Kussul, "Binding and normalization of binary sparse distributed representations by context-dependent thinning.," *Neural Computation*, vol. 13, no. 2, pp. 411–452, 2001.
[5] J. Snaider, *Integer Sparse Distributed Memory and Modular Composite Representation*. PhD thesis, University of Memphis, 2012.
[6] J. Snaider and S. Franklin, "Extended sparse distributed memory," in *Proceedings of the Biological Inspired Cognitive Architectures 2011*, 2011.
[7] C. Eliasmith, T. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen, "A large-scale model of the functioning brain," *Science*, vol. 338, pp. 1203–1204, November 2012.