Efficient FCM Computations Using Sparse Matrix-Vector Multiplication

Michal Puheim[1], Ján Vaščák[2], Kristína Machová[3]
Center for Intelligent Technologies
Department of Cybernetics and Artificial Intelligence
Technical University of Kosice
Letná 9, 042 00 Košice, Slovak Republic
[1]michal.puheim@tuke.sk, [2]jan.vascak@tuke.sk, [3]kristina.machova@tuke.sk

***Abstract*—Fuzzy cognitive maps (FCM) are often represented and implemented using matrix-vector multiplication (MxV). Since the multiplication operation is critical to the performance of the FCM computations, it is important to secure its efficient implementation. Considering the connection matrix used to represent the FCM is often static and since it often contains only several nonzero elements, it is viable to transform it into another particular representation suitable to perform sparse matrix-vector multiplication (SpMxV). This paper shows a performance benchmark for the most common SpMxV representations, namely the CRS and CCS. It also examines the sparsity threshold at which it is more efficient to use naïve dense MxV.**

Keywords— Sparse Matrix-Vector Multiplication; Benchmark; Fuzzy Cognitive Maps; Comparison;

I. INTRODUCTION

A wide range of fundamental computational problems in various scientific fields, can be transformed to a computational kernel cast as a matrix-vector multiplication (MxV). MxV operations are run every day on a huge number of processors all over the world, from embedded microcontrollers to supercomputers, performing computations of signal and image processing, computer science, natural science, engineering, economics and everyday graphical visualizations [1]. It is used in system modeling [2], kernel estimation methods of nonparametric statistics and many common probabilistic graphical models. Specific applications of MxV include statistical regression methods, radial-basis function networks, Gaussian processes [3][4] and calculation of ontology based dissimilarity measures [5][6]. MxV is also widely utilized in several well-known machine learning methods including neural networks, Bayesian networks, support vector machines [7], and last but not least, fuzzy cognitive maps [8].

The majority of modern applications often require high performance and efficient computations. This is especially important with emergent trend in deployment of control systems such as [9] using cloud computing frameworks where each extra computation has inherent financial cost. Therefore it is necessary to optimize the used algorithms and the same applies to the computational kernel methods based on MxV.

Most of the recent research considering MxV was conducted in the area of dense matrix-matrix multiplication and dense matrix-vector multiplication, which deals with matrices and vectors with low number of zero elements. Approaches proposed in [10] and [11] used BLAS [12] subroutines to divide matrix multiplication to smaller sub-problems which can be processed on CPU cache, effectively speeding up the computation process. Another trend is usage of parallelization, often on GPUs and also multicore CPUs. The authors of [1] proposed a new methodology for computing dense MxV for both embedded single core processors and general purpose multi-core processors.

The next important branch of research deals with sparse matrix-vector multiplication (SpMxV). The matrix sparsity is defined as a fraction of non-zero elements over the total number of elements. SpMxV represents a class of algorithms which exploit the lack of nonzero elements within the matrix. The multiplication is in the form $Y=A\times X$, where the matrix A is sparse, however the vectors X and Y may be dense. In case of repeated multiplication involving the same input matrix A, it can be preprocessed to reduce both the parallel and sequential runtime of the SpMxV operation [13]. Improvement in SpMxV speed was achieved in [14] and [15] by exploiting structure of a matrix consisting of multiple, irregularly aligned rectangular blocks. Performance evaluation of SpMxV on modern computer architectures was done in [16] and concluded, that greatest bottleneck in SpMxV computation is memory and CPU cache utilization. Parallel SpMxV using a new storage format of compressed sparse blocks was proposed in [17]. Optimization and evaluation of SpMxV on multicore platforms was performed in [18]. Study [13] explored ways to exploit cache locality in both sequential and parallel implementation of SpMxV and confirmed that temporal locality is more important than spatial locality in operations that involve irregularly sparse matrices.

In this paper we examine open questions which have not been answered in available literature yet. We are specifically interested in SpMxV performance during computations of fuzzy cognitive maps, which are specific in a way they use the *same matrix* and *same input* and even *output vector* during course of all iterations. We also compare SpMxV with *varying level of sparsity* to determine a threshold for feasible utilization of SpMxV, i.e. the cases in which dense multiplication algorithm are more efficient. Along with this, we aim to compare naïve dense multiplication algorithm implemented using multidimensional and jagged arrays for matrix storage and determine which implementation is more efficient.

978-1-5090-1897-0/16/$31.00 ©2016 IEEE

II. Matrix-Vector Multiplication

The general definition of matrix vector multiplication is following: Let the matrix be represented by A. The elements of A are $A[i,j]$, where i is the row index which takes values from 1 to m and j is the column index which takes values from 1 to n. Let the vector be represented by X. The elements of X are $X[j]$, where j is the index and takes values from 1 to n. The product Y is defined as:

$$Y = A \times X, \tag{1}$$

This definition can be solved programmatically using various algorithms [19][20][21][22][23], however the question which of these algorithms is "fastest" has not been answered till this day. In the next part of this paper we will present examples of the most frequently used matrix-vector multiplication algorithms, while we will specifically focus on approaches exploiting sparsity within the matrix, since sparse matrices are the most common in applications of fuzzy cognitive maps.

A. Dense Matrix-Vector Multiplication

The naïve iterative approach to perform matrix-vector multiplication (i.e. dense matrix-vector multiplication) is based on direct application of mathematical definition of matrix-vector multiplication (1) and it computes the individual elements $Y[i]$ as:

$$Y[i] = \sum_{j=1}^{m} A[i,j] \cdot X[j], \tag{2}$$

From this, a simple algorithm can be constructed which loops over the indices j from 1 to m and i from 1 to n, computing the individual vector elements $Y[i]$ using a nested loop, see Fig. 1.

Algorithm 1 Dense Matrix-Vector Multiplication

```
1: for i ← 1 to m do
2:     Y[i] ← 0;
3:     for j ← 1 to n do
4:         Y[i] ← Y[i] + A[i,j] * X[j];
5:     end
6: end
```

Fig. 1. Dense Matrix-Vector Multiplication Algorithm

This direct application of the mathematical definition of the matrix-vector multiplication gives an algorithm that takes time in the order of n^2 to multiply $n \times n$ matrix with vector (or $\Theta(n^2)$ in big O notation).

The loops in the algorithm can be arbitrarily swapped with each other without an effect on correctness or asymptotic running time. However, its order can have a considerable impact on practical performance due to the memory access patterns and cache use of the algorithm [19]. The best order also depends on whether the matrices are stored in row-major order, column-major order, or other representation.

B. Compressed Row Storage

The most common matrix representation used for sparse matrix-vector multiplication is *Compressed Row Storage* (CRS). It stores the matrix as a sequence of compressed rows. It uses three arrays: *Ac*, *Colind* and *Rowptr*. The m nonzero elements of the sparse matrix A are compressed into an array *Ac* in a rowwise manner (left-to-right, then top-to-bottom) [14]. The column index of each i-th nonzero entry is stored in the array *Colind*, i.e., *Colind*[i] is the column index of i-th nonzero entry stored in *Ac*[i]. Finally, *Rowptr* stores the index of the first nonzero of each row [14]. Fig. 2 presents a SpMxV algorithm using CRS.

Algorithm 2 CRS SpMxV

```
1: for i ← 1 to m do
2:     Y[i] ← 0;
3:     for j ← Rowptr[i] to Rowptr[i+1] – 1 do
4:         Y[i] ← Y[i] + Ac[j] * X[Colind[j]];
5:     end
6: end
```

Fig. 2. SpMxV algorithm for Compressed Row Storage

According to [14], most SpMxV algorithms tend to exhibit poor memory performance. One reason for this is ineffective cache utilization, since temporal locality is limited to right and left-hand-side vectors and the *Rowptr* array. No temporal locality is present in arrays *Ac* and *Colind*. In CRS, there is good temporal locality in the left-hand-side vector Y, but the access pattern for the right-hand-side vector X is still quite irregular, causing excessive cache misses.

C. Compressed Column Storage

The *Compressed Column Storage* (CSC) is similar to CSR except that values are read first by column. It is sometimes also called *Harwell-Boeing sparse matrix format* [21]. In the CCS, the nonzero values A are stored in columns instead of the rows. In other words, the CCS format is the CRS format for A^T [20]. The matrix is again stored in three arrays (*Ac*, *Rowind*, *Colptr*), where *Ac* is an array of the non-zero values of the matrix (stored top-to-bottom, then left-to-right); *Rowind* contains the row indices corresponding to the values; and, *Colptr* stores the index of the first nonzero of each column. CCS SpMxV algorithm is shown on Fig. 3.

Algorithm 3 CCS SpMxV

```
1: for i ← 1 to m do
2:     Y[i] ← 0;
3: end
4: for j ← 1 to n do
5:     for i ← Colptr[i] to Colptr[i+1] – 1 do
6:         Y[Rowind[j]] ← Y[i] + Ac[i] * X[i];
7:     end
8: end
```

Fig. 3. SpMxV algorithm for Compressed Column Storage

This format enables very efficient processing of arithmetic operations, column slicing, and matrix-vector products. It is the traditional format for specifying a sparse matrix in MATLAB (using sparse function) and it is also commonly used in Scientific Python. However it suffers from poor temporal locality in left-hand-side vector *Y* due to the indirect indexing of vector elements.

D. Incidence List Representation

The incidence list (IL) or adjacency list representation exploits the graph representation of the matrix specifically geared towards efficient matrix-vector multiplication. It can be found in many variations [24][22][23] where the graph is implemented with different associations between vertices and varying collections used to store these associations. There are also object-oriented approaches which represent the vertices and the edges as objects [23]. The following paragraphs present our interpretation of this idea.

Each element $Y[i]$ of the output vector is calculated according to (2) as a sum of scalar products of corresponding nonzero elements of the matrix $A[i,j]$ and elements of the input vector $X[j]$. Therefore, the IL representation associates each element $Y[i]$ (or vertex within the graph) with the collection of its neighboring vertices $X[j]$ and corresponding edges $A[i,j]$. Consequently, the matrix is stored using two jagged arrays (or lists of lists) – the index array of vertices *Vind*, and the value array of edges *Eval*.

Each element $Vind[i]$ of the vertex array stores the list of j-th indices of the corresponding input vector elements $X[j]$ (i.e. vertices). Similarly, each element $Eval[i]$ of the edge array stores the list of values of corresponding matrix elements $A[i,j]$ (i.e. edges). Algorithm 41: shows a SpMxV algorithm using IL representation.

Algorithm 4 Incidence List SpMxV

```
1: for i ← 1 to m do
2:     Y[i] ← 0;
3:     for k ← 0 to length(Vind[i]) do
4:         Y[i] ← Y[i] + Eval[i][k] * X[Vind[i][k]];
5:     end
6: end
```

Fig. 4. SpMxV algorithm for Incidence List Representation

The IL representation can be very easily expanded to efficiently perform several matrix-vector multiplications within various applications (such as [25]). Very perspective is also the object-oriented implementation proposed by [23] which utilize special classes of vertex objects and edge objects. Each vertex object has member variable pointing to a collection that lists the neighboring edge objects. In turn, each edge object points to the two vertex objects at its endpoints. While this variation uses more memory than direct listing of vertices, the existence of explicit edge object allows enables more flexibility in storing additional information about edges.

III. FUZZY COGNITIVE MAPS

Fuzzy cognitive map (FCM) in general is an oriented graph, where vertices C_j (j = 1, 2,..., n) represent concepts (elements or attributes of the modeled system) and edges w_{ij} determine mutual relations between concepts C_i and C_j. Values of either concepts or edges are usually represented by fuzzy values from range [0, 1], or [–1, 1] respectively. An advantage of FCM is clear and comprehensible representation of knowledge, which can be easily visualized in graph form (see Fig. 5). However, edges w_{ij} in FCM are very often represented in a form of a connection matrix E (see Fig. 6) so that:

$$E = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{pmatrix} = \left(w_{ij}\right) \in [-1,1]^{n \times n}, \qquad (3)$$

and activation values a_j of concepts c_j as a column vector A:

$$A = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \left(a_j\right) \in [0,1]^n, \qquad (4)$$

If we define the initial state of A in the time t as $A(t)$ then we can use the matrix E in order to calculate the new state $A(t+1)$ according to formula:

$$A(t+1) = p\left(A(t) + E \times A(t)\right), \qquad (5)$$

where p is a bounding function, which keeps the results in a range [0, 1] and therefore provides stability measures for the FCM model [8][25].

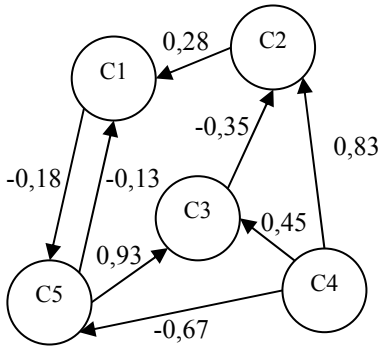

Fig. 5. FCM represented as fuzzy oriented graph.

$$E = \begin{bmatrix} 0 & 0{,}28 & 0 & 0 & -0{,}13 \\ 0 & 0 & -0.35 & 0.83 & 0 \\ 0 & 0 & 0 & 0{,}45 & 0.93 \\ 0 & 0 & 0 & 0 & 0 \\ -0{,}18 & 0 & 0 & -0{,}67 & 0 \end{bmatrix}$$

Fig. 6. FCM representation in form of connection matrix

IV. Problem Definition and Experimental Setup

The most computationally demanding operation of (5) is the matrix-vector product $E \times A(t)$. The insight into its computation efficiency is also the main motivation for this paper. In the Fig. 6 we can clearly see, that the matrix E is very sparse as it contains majority of zero elements. Therefore, the standard algorithms used for dense MxV will be apparently quite inefficient. However, since matrix E of the FCM is mostly static, it is feasible to perform even very demanding transformation to other suitable representation.

Our main goal is to compare the resulting performance of SpMxV after the transformation. Since we are not interested in transformation costs, we only measure the time of the multiplication operation using the specific representation. All the representations, which were described in section II of this paper, were included into the comparison. We have also implemented two versions of dense MxV, where the first one use multidimensional array representation and the second one use jagged array representation (or array of arrays), since no specific information about the influence of such array implementation is to be found in the available literature.

Random matrices of variable size and sparsity were used and several MxV operations were performed in order to balance the results. The performance was measured by a number of operations per second (or flops). A simplified algorithm of a benchmark for performance of MxV with fixed matrix size and variable matrix sparsity is shown on Fig. 7.

Algorithm 5 Variable Sparsity Benchmark

```
1:  time ← benchmarkStepTime;
2:  size ← fixedSize;
3:  for sparsity ← initial to final do
4:      foreach r in representations do
5:          stopwatch[r] ← new Stopwatch();
6:          time[r] ← 0;
7:          ops[r] ← 0;
8:      end
9:      stopwatchStep ← new Stopwatch();
10:     stopwatchStep.start();
11:     do
12:         E ← new RandMatrix(size, sparcity);
13:         X ← new RandDenseVector(size);
14:         foreach r in representations do
15:             E_r ← Transform(E, r);
16:             stopwatch[r].start();
17:             Y ← E_r * X;
18:             stopwatch[r].stop();
19:             ops[r] ← ops[r] + 1;
20:         end
21:     while time > stopwatchStep.time();
22:     foreach r in representations do
23:         time[r] ← stopwatch[r].time();
24:         flops[r][sparsity] ← ops[r] / time[r] ;
25:     end
26: end
```

Fig. 7. Sparsity benchmark algorithm

We have also conducted a second benchmark with fixed sparsity and varying size, which is of the same definition since it only differs in the lines 2 and 3, where in the line 2 there is a sparsity variable set to a fixed value and in the line 3 the loop is performed over a range of varying matrix sizes. This change is shown on Fig. 8.

Algorithm 6 Part of Variable Size Benchmark

```
1:  time ← benchmarkStepTime;
2:  sparsity ← fixedSparsity;
3:  for size ← initial to final do
4:      … (continues same as Algorithm 5)
```

Fig. 8. Part of size benchmark algorithm

The full code for the both benchmarks is available at [26]. The results of the first benchmark, which compares MxV with fixed matrix size and variable matrix sparsity, are presented in Fig. 9-11. The results of the second benchmark, which compares MxV with fixed matrix sparsity and variable matrix size, are presented in Fig. 12-14.

V. Interpretation of the Results and Discussion

The results of both the sparsity benchmark and the size benchmark show, that the SpMxV algorithms are almost always better than dense MxV algorithms. The best performance is shown by CCS which is on pair with dense MxV even at 99% sparsity. It is clear that the threshold, at which the dense MxV performs better than SpMxV, is unreachable for most of the typical MxV applications.

The credibility and relevance of the benchmark could be improved in several ways. It is known [16] that in some cases the SpMxV algorithms can be optimized by exploiting information regarding the matrix structure and the processor's architectural characteristics. For example, we could include algorithms which utilize approaches to reorder of matrix elements in order to gain dense [14][15] or sparse [17] blocks which can be then efficiently computed. Such optimizations can be applied to a limited number of sparse matrices. Even though there is no guarantee that such reordering is possible with any given matrix, the comparison of performance of these algorithms could be interesting. It is also possible to employ parallelization of computations [1][18]. Indeed all of the tested methods are suitable for parallelization. On the other hand, according to [27], parallelization may not be always as beneficial as expected, since parallel architectures still suffer from memory bottleneck. It is also questionable, if parallelized algorithms can be objectively compared. And finally, in order to perform a truly objective benchmark, it is viable to randomize order of representations within the inner loop (i.e. lines 14 to 20 of the benchmark algorithm). Even though the specific order cannot affect the resulting performance, since each representation has its own generated matrix representation, the engagement of the garbage collector releasing memory used by previous representations may put additional workload on the CPU and consequently distort the results.

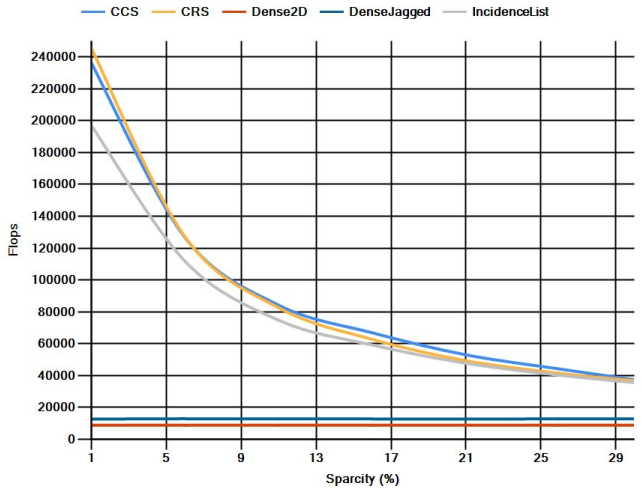

Fig. 9. Performance of SpMxV; matrix size 100x100; sparcity 1 to 30 %

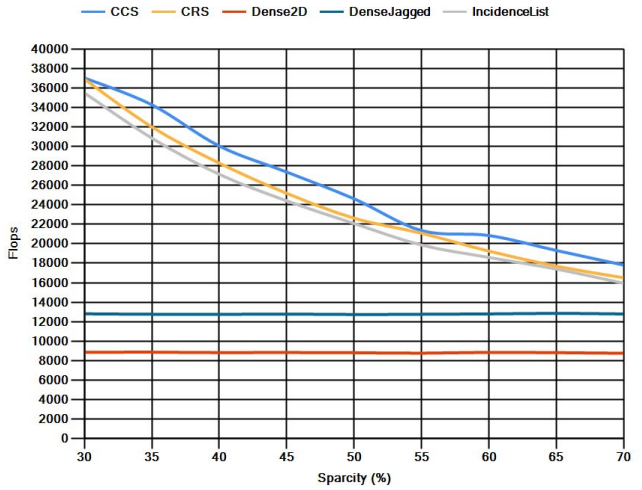

Fig. 10. Performance of SpMxV; matrix size 100x100; sparcity 30 to 70 %

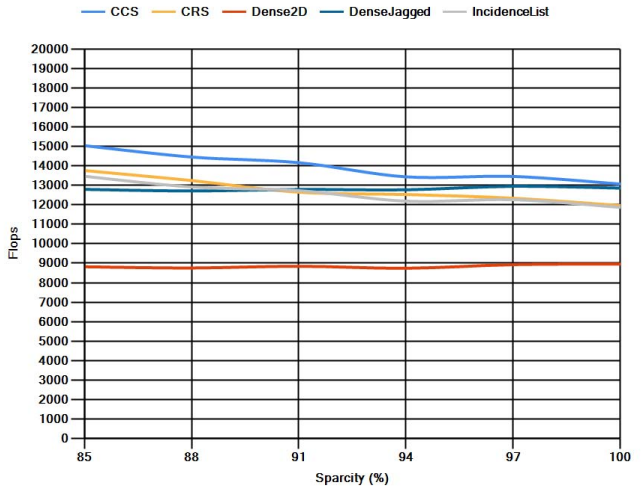

Fig. 11. Performance of SpMxV; matrix size 100x100; sparcity 85 to 100 %

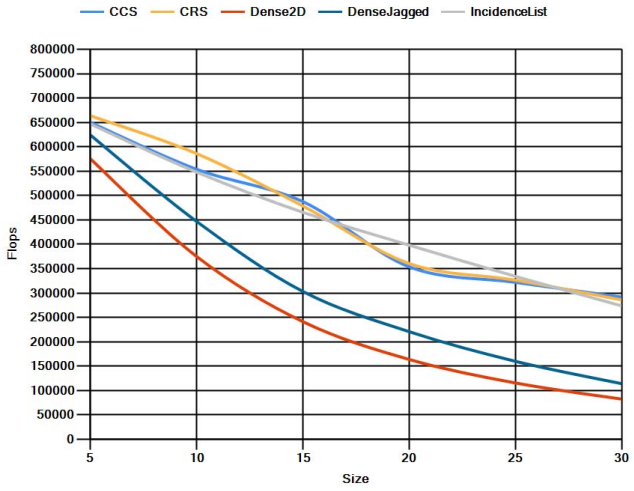

Fig. 12. Performance of SpMxV; matrix sparcity 20 % ; size 5^2 to 30^2

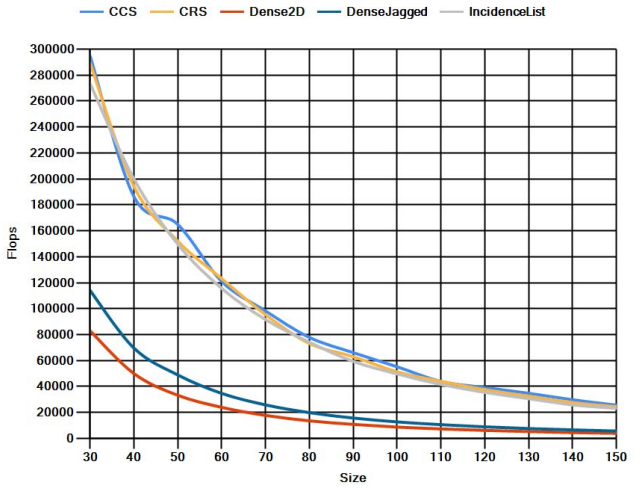

Fig. 13. Performance of SpMxV; matrix sparcity 20 % ; size 30^2 to 150^2

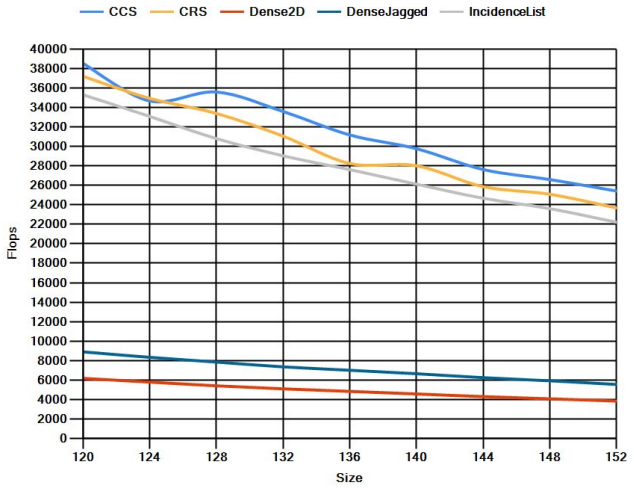

Fig. 14. Performance of SpMxV; matrix sparcity 20 % ; size 120^2 to 150^2

VI. Conclusion

This paper presents a benchmark of the most widely used SpMxV algorithms along with a naïve dense MxV, for the purpose of utilization in FCM computations. It shows that all tested SpMxV algorithms (CRS, CCS and Incidence List) are more efficient in terms of operations per second with matrices which are below 90% sparsity. However, the CCS is on pair with dense MxV even at 99% sparsity. The performance of the Incidence List representation is also very promising, since this representation is most suitable to implement advanced FCM structures. Finally, it has been confirmed, that naïve dense MxV implemented using jagged arrays is faster than MxV using 2D multidimensional array (by factor of 1.5).

Acknowledgment

This paper is supported by KEGA, Grant Agency of Ministry of Education of Slovak Republic, under *Grant No. 014TUKE-4/2015* – "Digitalization, virtualization and testing of a small turbojet engine and its elements using stands for modern applied lecturing."

References

[1] V. Kelefouras, A. Kritikakou, E. Papadima and C. Goutis, "A methodology for speeding up matrix vector multiplication for single/multi-core architectures," in *The Journal of Supercomputing*, vol. 71, no. 7, pp. 2644-2667, 2015, doi: 10.1007/s11227-015-1409-9

[2] R. E. Precup, C. A. Dragos, S. Preitl, M. B. Radac and E. M. Petriu, "Novel Tensor Product Models for Automatic Transmission System Control," in *IEEE Systems Journal*, vol. 6, no. 3, pp. 488-498, Sept. 2012. doi: 10.1109/JSYST.2012.2190692

[3] A. G. Gray, "Fast kernel matrix-vector multiplication with application to Gaussian process learning." Technical report, School of Computer Science, Carnegie Mellon University, 2004.

[4] I. Murray, "Gaussian processes and fast matrix-vector multiplies." Numerical Mathematics in Machine Learning Workshop, International Conference on Machine Learning ICML 2009, 2009. Available: http://homepages.inf.ed.ac.uk/imurray2/pub/09gp_eval/

[5] J. Vrana and M. Mach, "Ontology key concepts interpretation," in Proceedings of 8th Int. Symposium on Applied Machine Intelligence and Informatics, January 28-30, 2010, Herľany, Slovakia. Danvers: IEEE, 2010, pp. 215-219, ISBN 978-1-4244-6423-4.

[6] J. Vrana and M. Mach, "Key concepts extended by vector descriptions to interpret the meaning of ontologies," in Acta Electrotechnica et Informatica, vol. 11, no. 3, pp. 57-63, 2011, ISSN 1335-8243.

[7] E. Nurvitadhi, A. Mishra and D. Marr, "A sparse matrix vector multiply accelerator for support vector machine," Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2015 International Conference on, Amsterdam, 2015, pp. 109-116, doi: 10.1109/CASES.2015.7324551

[8] D. Lorencik, J. Vascak, M. Vircikova, "Adaptive Fuzzy Cognitive Maps Using Interactive Evolution: A Robust Solution for Navigation of Robots" in Advances in Intelligent Systems and Computing, vol 208, pp 703-711, 2013, doi: 10.1007/978-3-642-37374-9_67

[9] R. Limosani, A. Manzi, L. Fiorini, F. Cavallo and P. Dario, "Enabling Global Robot Navigation Based on a Cloud Robotics Approach," in *International Journal of Social Robotics*. March 2016, doi: 10.1007/s12369-016-0349-8

[10] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," in ACM Trans. on Mathematical Software (TOMS), vol. 34, iss. 3, no. 12, May 2008, 25 pp. doi: 10.1145/1356052.1356053

[11] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 BLAS," in ACM Trans. on Mathematical Software (TOMS), vol. 35, iss. 1, no. 4, July 2008, 14 pp. doi: 10.1145/1377603.1377607

[12] B. Kagstrom, P. Ling, C. V. Loan, "GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark," in ACM Transactions on Mathematical Software (TOMS), vol. 24, no. 3, pp. 268-302, 1998, doi: 10.1145/292395.292412

[13] K. Akbudak, K. Enver and A. Cevdet, "Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication," in *SIAM Journal on Scientific Computing*, vol 35, no 3, pp. 237-262, 2013. doi: 10.1137/100813956

[14] A. Pinar, and M. T. Heath, "Improving performance of sparse matrix-vector multiplication." in Proc. of the 1999 ACM/IEEE conference on Supercomputing. ACM, 1999. doi: 10.1145/331532.331562

[15] R. W. Vuduc and M. Hyun-Jin, "Fast sparse matrix-vector multiplication by exploiting variable block structure." High Performance Computing and Communications. Springer Berlin Heidelberg, 2005. 807-816. doi: 10.1007/11557654_91

[16] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," in The Journal of Supercomputing, vol 50, no 1, pp 36-77, October 2009, doi: 10.1007/s11227-008-0251-8

[17] A. Buluc, et al, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks." Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. ACM, pp. 233-234, 2009, doi: 10.1145/1583991.1584053

[18] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms." in Parallel Computing, vol. 35, no. 3, 2009, pp. 178-194, doi:10.1016/j.parco.2008.12.006

[19] S. S. Skiena, "The algorithm design manual," Second Edition. Springer-Verlag London, pp. 720, 2008, doi: 10.1007/978-1-84800-070-4

[20] J. Dongarra, "Survey of Sparse Matrix Storage Formats," 1995. [Online]. Available: http://netlib.org/linalg/html_templates/node90.html

[21] I. Duff, R. Grimes and J. Lewis, "Sparse matrix test problems," in ACM Transactions on Mathematical Software (TOMS), vol.15, no 1, March 1989, pp. 1-14, doi: 10.1145/62038.62043

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms," Second Edition. MIT Press, Cambridge, MA, USA, 2001, ISBN:0-262-03293-7 9780262032933

[23] M. T. Goodrich and R. Tamassia, "Algorithm Design: Foundations, Analysis and Internet Examples," Second Edition. John Wiley & Sons, Inc., New York, NY, USA, 2009, ISBN:0470088540 9780470088548

[24] G. van Rossum, "Python Patterns – Implementing Graphs," 1998. [Online]. Available: https://www.python.org/doc/essays/graphs/

[25] M. Puheim, J. Vaščák, L. Madarász, "Three-Term Relation Neuro-Fuzzy Cognitive Maps." in Proceedings of 15th IEEE Int. Symposium on Computational Intelligence and Informatics. November 19-21, 2014, Budapest. Danvers: IEEE, 2014, pp. 477-482. ISBN 978-1-4799-5337-0.

[26] M. Puheim, "Sparse Matrix Vector Multiplication Benchmark". 2016. [Online]. Available: https://github.com/mpuheim/SMVM-Benchmark

[27] V. W. Lee, et al, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU." in ACM SIGARCH Computer Architecture News, vol. 38, no. 3, 2010, doi: 10.1145/1815961.1816021