A Method for Parallel Non-Negative Sparse Large Matrix Factorization

Anatoly Anisimov, Oleksandr Marchenko, Emil Nasirov, and Stepan Palamarchuk

Faculty of Cybernetics, Taras Shevchenko National University of Kyiv, Ukraine

Abstract. This paper proposes parallel methods of non-negative sparse large matrix factorization. The described methods are tested and compared on large matrices processing.

Keywords: computational linguistics, parallel computations, non-negative matrix factorization

1 Introduction

Non-negative matrix and tensor factorization are very popular techniques in computational linguistics. With the help of non-negative matrix and tensor factorization within the paradigm of latent semantic analysis [1] computational linguists solve applied problems such as classification, clustering of texts and terms [2,3], construction of measures of semantic similarity [4,5], automatic extraction of linguistic structures and relations (Selectional Preferences) and Verb Sub-Categorization Frames), etc. [6]

This work describes the construction of a model for parallel non-negative factorization of a sparse large matrix. Such a model can be used in large NLP systems not limited to narrow domains.

The problem of non-negative factorization for a sparse large matrix emerged in the development of a measure of semantic similarity between words with Latent Semantic Analysis usage. To cover a wide range of topics a great amount of articles from the English Wikipedia was processed to construct the similarity measure. Lexical analysis of the various Wikipedia articles was performed to calculate the frequency of using words and collocations. As a result, a large matrix Terms $\times$ Articles was constructed. It contains frequency estimations of using terms in texts. The precise size of the matrix equals to 2,437,234 terms $\times$ 4,475,180 articles of the English Wikipedia. The frequency threshold T=3 was set to remove the noise. The resulting matrix contains 156,236,043 non-zero elements. To factorize a sparse matrix of such size it is necessary to develop a specific model for parallelizing matrix computations. The model has been implemented using distributed and parallel computing on the GPU. Recently a plenty number of powerful parallel models for Non-Negative Matrix Factorization (NMF) have been developed [7,8,9,10]. However none of the developed applications for them is an acceptable solution for the defined task. Some of them do not satisfy the requirements of the matrix dimensions [7,8,9]. The model presented in work [10] performs NMF for sparse matrices of required dimensions in an acceptable time, but it requires excessively large computational resources and it is not always affordable.

P. Sojka et al. (Eds.): TSD 2014, LNAI 8655, pp. 344–352, 2014.

2 NMF Algorithm

Non-negative matrix factorization of matrix V of size $[n; m]$ is a process of calculating two matrices W and H of size $[n; k]$ and $[k; m]$ respectively, such that $V \approx WH$.

$$F(W, H) = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}(V_{i,j} - (WH)_{i,j})^2} \tag{1}$$

The goal of the algorithm is to minimize the cost function quantifying the approximation quality. There are a lot of different cost functions. In this paper the root-mean-square distance between V and WH is used 1.

In [11], the authors proposed a simple iterative algorithm to approximate the matrices. It consists of two consequent updates of matrices W and H given by and .

$$(H_t)_{ij} = (H_{t-1})_{ij} \frac{(W_{t-1}^T V)_{ij}}{(W_{t-1}^T W_{t-1} H_{t-1})_{ij}} \tag{2}$$

$$(W_t)_{ij} = (W_{t-1})_{ij} \frac{(V H_t^T)_{ij}}{(W_{t-1} H_t H_t^T)_{ij}} \tag{3}$$

In 2 and 3, H_t and W_t are the matrices obtained on iteration t. H_0 and W_0 are initialized with random values from [0; 1) range.

The algorithm continues until either a stationary point is reached or a certain number of iterations is performed.

3 Model Analysis

The goal is to solve the NMF problem for different k values and compare results for all of them. Table 1 shows memory requirements for storing W and H for different k. On each iteration the algorithm described in Section 2 requires twice as much memory as required for matrices storage. This does not include memory required for V. Due to such excessive memory requirements of the algorithm it is difficult to execute it on a single machine, without dumping data to the hard drive. Two variants of the algorithm implementation are described below: local (with intensive hard drive usage) and distributed (with intensive network usage).

Table 1. Memory requirements for storing of W and H for different k , based on 32-bit float.

k	100	200	300
W	0.98Gb	1.95Gb	2.92Gb
H	1.79Gb	3.58Gb	5.37Gb
$total$	2.76Gb	5.53Gb	8.29Gb

4 A GPU Version of the Algorithm

To simplify explanations the substitution ($H' = H^T$) and transformation of 2 and 3 result in:

$$(H'_t)_{ij} = (H'_{t-1})_{ij} \frac{(V^T W_{t-1})_{ij}}{(H'_{t-1} W_{t-1}^T W_{t-1})_{ij}} \tag{4a}$$

$$(W_t)_{ij} = (W_{t-1})_{ij} \frac{(V H'_t)_{ij}}{(W_{t-1} H_t'^T H'_t)_{ij}} \tag{4b}$$

It allows for treating both formulas in the same way, by simply substituting either H', W and V^T or W, H' and V instead of A, B and S into 5.

$$A_{ij} = A_{ij} \frac{(SB)_{ij}}{(AB^T B)_{ij}} \tag{5}$$

From this point, only evaluation of 5 with a configuration W, H' and V is discussed, since other configuration can be obtained in the same way.

Formula 5 can be calculated as a series of four steps as in 6.

$$C = SB \tag{6a}$$

$$K = B^T B \tag{6b}$$

$$D = AK \tag{6c}$$

$$A_{ij} = A_{ij} \frac{C_{ij}}{D_{ij}} \tag{6d}$$

This order of computation 5 requires a minimal number of calculations. The steps have computational complexity of $O(k * (nnz(S) + n))$, $O(k^2 m)$, $O(k^2 n)$ and $O(kn)$ correspondingly, where $nnz(S)$ is a number of non-zero cells in matrix S. The first three steps are natively supported by CUDA cuSPARSE [12] and cuBLAS [13] libraries (or other similar libraries for AMD). The fourth step requires custom GPU kernel implementation, but at the same time it is a relatively cheap operation and thus it can be performed on CPU.

Also these matrices are too large to be stored in the memory of GPU, thus operations should be performed by parts in a manner that reduces amount of excessive memory copying.

So for 6a matrices can be written as $S = (S'_1|S'_2|...|S'_t)^T$ and $B = (B_1|B_2|...|B_r)$ and each cell of C calculated as shown in 7. Since B is larger than S in terms of memory usage multiplications should be grouped by pieces of B (to upload them only once). Also it is rational to minimize r and keep t reasonably small, otherwise most of GPU cores will be idle. C is matrix of size $[m; k]$ for H' and $[n; k]$ for W.

$$C = \begin{pmatrix} S'_1 B_1 & \ldots & S'_1 B_r \\ \ldots & & \ldots \\ S'_t B_1 & \ldots & S'_t B_r \end{pmatrix} \tag{7}$$

For 6b it is preferable to write matrices as $B = (B'_1|...|B'_t)^T$ and $K = B'^T_1 B'_1 + ... + B'^T_t B'_t$, because it doesn't require any redundant matrix uploads to GPU. K is matrix of size $[k;k]$.

For 6c matrix K should be kept in memory and A should be multiplied by blocks of rows. D is matrix of size $[m;k]$ for H' and $[n;k]$ for W.

There is no need to store D in memory if 6d is applied on the piece of matrix A that was used to obtain a piece of matrix D.

The complexity of operation is $O(nm)$ and is straightforward to implement with CUDA toolkit.

5 Distributed Algorithm

The next step to improve the performance is to use a distributed grid of PCs of same configuration. There are several distribution models. In the case of 2 nodes in the grid, there are next three distribution models:

1. W **and** H' **are separately calculated on different nodes.** Both nodes work in one of the two modes alternatively. They either support the other node (supplying data to the other node) or lead (calculating by using the data received form supporting node). In this distribution model, on each iteration it is necessary to transmit over the network amount of data equal to $sizeof(W) + sizeof(H)$, where $sizeof(X)$ is the amount of memory required to store matrix X. Also lead node will be mostly idle, because 6a is the most resource-demanding step out of all the 4.
2. W **and** H' **are split in chunks of rows and evenly distributed between nodes.** Where $H' = (H'_1|H'_2)$ and $W = (W_1|W_2)$, the first node responsible for H'_1, W_1 and the second node for H'_2, W_2. In this model each node behaves as supporting and leading node at the same time. Nodes need to transmit amount of data equal to $1.5 * (sizeof(W) + sizeof(H))$ over the network.
3. W **and** H' **are split in chunks of columns and evenly distributed between the nodes.** Where $H' = (H_1|H_2)^T$ and $W = (W'_1|W'_2)^T$, the first node is responsible for H_1, W'_1 and the second for H_2, W'_2. In this model, similarly to the previous one, each node works in both modes at the same time. Nodes need to transmit the amount of data equal $sizeof(W) + sizeof(H)$ over the network because it is possible to calculate pieces of $H^T * H$, $W^T * W$ on each node separately and there is no need to transmit H'_2 to H'_1 and W_2 to W_1 as in second model.

In each of the above models nodes also need to transmit one or several matrices of size $[k;k]$, but their total size is neglectable comparing to the size of W and H. For metrics calculation for both the first and the third model it is necessary to transmit the amount of data equal to $\frac{(sizeof(W)+sizeof(H))*K}{2}$, where K is the number of nodes in the grid, the second model requires $(K-1)$ times more transmitted data.

The last model is the most preferable, because it is better in both network and GPU utilization, thus it is used in the implementation.

Also it should be noticed that in case of the grid expansion, the total amount of the data transmitted over the network rises polynomially, but per node it will be limited by $2 * (sizeof(W) + sizeof(H))$.

Since V is a sparse matrix, it may contain an unevenly distributed amount of non-zero cells and this may badly impact the performance of the distributed algorithm. To optimize distribution of work between the nodes it is reasonable to rearrange the rows and columns of V in a way that equalizes amount of non-zero cells in each large cell of matrix V.

The third model is implemented to perform NMF of input matrix and used on a grid of four nodes, so this case will be described.

Where matrices W, H' and V are partitioned according to the selected model 3: $W = (W'_1|W'_2|W'_3|W'_4)^T$, $H' = (H_1|H_2|H_3|H_4)^T$, $V = \begin{matrix} V_{11} \dots V_{14} \\ \dots \quad \dots \\ V_{41} \dots V_{44} \end{matrix}$.

The algorithm consists of three main phases: initialization, iterations and metrics calculation. At initialization phase W, H and V are distributed between all the 4 nodes. Node i gets W'_i, H_i and V_{ki}, V_{ik}, $k = \overline{1,..,4}$, this phase is represented by the scheme in Figure 1 (left).

The iteration phase consists of two similar steps, one for calculation of H' and the other for W. Each of them is subdivided into 3 smaller sub-steps as was described further.

At the first sub-step each node calculates $k \times k$ matrix $W'_i * W'^T_i$ and sends it to the aggregator. The aggregator sums all received pieces into one matrix K_w and sends the aggregated result to all the nodes. This sub-step is represented by the scheme in Figure 1 (left).

At the next sub-step each node calculates its own $(V^T_{1i}|V^T_{2i}|V^T_{3i}|V^T_{4i})^T * W'_i$. The resulting matrix has the same size as H'. Finally each node divides its matrix according to the initial partitioning of matrix H and transmits these pieces to the corresponding nodes. This sub-step is represented on Figure 1 (right).

At the third sub-step the nodes calculate matrix $H_i * K_w$ and perform an in-place update of matrix H_i. This sub-step does not require any network communication.

These three sub-steps are intended for calculating matrix H'. After updating H', the same sub-steps should be made for W. Specifically next products should be calculated $H_i * H_i^T$, $(V_{i1}|V_{i2}|V_{i3}|V_{i4}) * H_i$ and $W'_i * K_h$.

At the metrics calculation phase each node transmits its piece of matrix H' to all other nodes. After receiving a piece of matrix H' each node calculates the corresponding part of the metrics. This phase is also represented by in Figure 1 (right).

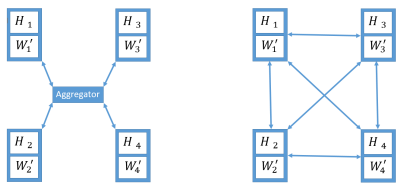

Fig. 1. Initial partitioning in the model with 4 nodes. (left) The iteration phase of the distributed model with 4 nodes. (right)

6 Results of Analysis

The previously described distributed algorithm with GPU usage has been implemented. The local GPU algorithm that dumps and reads data from the local hard drive has also been implemented to compare the performance of the models.

Table 2. Performance of local and distributed implementations for iteration and metrics calculation.

	Iteration		Metrics calculation	
	Local	Distributed	Local	Distributed
Data reads	34.44Gb	6.22Gb	13.66Gb	6.22Gb
Data writes	16.58Gb	6.22Gb	0	6.22Gb
Time (computation)	58s	15s	45865s	11371s
Time (data IO)	729s	287s	192s	280s

Both implementations are executed with the same input matrix.

We used next hardware configuration for tests: Intel Core i7 CPU, NVIDIA GeForce GTX560 1Gb, 8Gb of RAM (available 6 Gb), 1Gbit LAN and SATA III hard drive.

Table 2 shows the time and resources required for each version of the algorithm to perform the iteration. The data for distributed model are per node, so the total data IO (read & write together) across all 4 nodes is 49.76Gb. Table 2 shows comparison of metrics calculation. The data in both tables are obtained for $k = 300$.

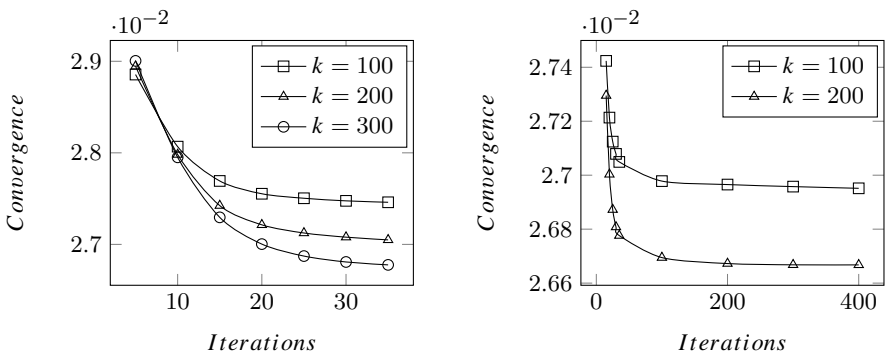

Fig. 2. Convergence of NMF with different k. Metrics value is calculated at each 5th iteration. (left) Convergence of NMF with k = 200 and k = 300. Metrics value is calculated at each 100th iteration. (right)

The experiments show that the process of matrices calculation converges after approximately 100 iterations. Therefore, the calculation of the non-negative factorization for the given sparse large matrix with the proposed model takes approximately 9.6 hours, for the distributed implementation and almost 21 hours for the local.

7 Parallelization of Non-negative Tensor Factorization

Distributed approach can be easily used for non-negative tensor factorization. To compute the non-negative component matrices A, B, C usually constrained optimization approach is applied as by minimizing a suitable cost function. Typically, the following global cost function is minimizing (with respect the component matrices):

$$D_F(\underline{Y}||\,[[A,B,C]]) = \|\underline{Y} - [[A,B,C]]\|_F^2 + \alpha_A\|A\|_F^2 + \alpha_B\|B\|_F^2 + \alpha_C\|C\|_F^2 \quad (8)$$

, where $\alpha_A, \alpha_B, \alpha_C$ are non-negative regularization parameters.

The most popular approach is to apply the ALS technique [14]. In this approach the gradient of the cost function for each component matrix is computed individually.

$$\nabla_A D_F = -Y_{(1)}(C \odot B) + A[(C^T C) \circledast (B^T B) + \alpha_A I] \quad (9)$$

$$\nabla_B D_F = -Y_{(2)}(C \odot A) + A[(C^T C) \circledast (A^T A) + \alpha_B I] \quad (10)$$

$$\nabla_C D_F = -Y_{(3)}(B \odot A) + A[(B^T B) \circledast (A^T A) + \alpha_C I] \quad (11)$$

ALS update rules for the NTF are obtained by equating the gradient components to zero:

$$A \leftarrow \left[Y_{(1)}(C \odot B) + \left[(C^T C) \circledast (B^T B) + \alpha_A I\right]^{-1}\right]_+ \quad (12)$$

$$B \leftarrow \left[Y_{(2)}(C \odot A) + \left[(C^T C) \circledast (A^T A) + \alpha_B I\right]^{-1}\right]_+ \quad (13)$$

$$C \leftarrow \left[Y_{(3)}(B \odot A) + \left[(B^T B) \circledast (A^T A) + \alpha_C I\right]^{-1}\right]_+ \quad (14)$$

These update rules 12, 13, 14 have the same form so they can be rewrited in one common rule 15 for further parallelizing:

$$M_i \leftarrow \left[Y_{(i)}(M_j \odot M_k) + \left[\left(M_j^T M_j\right) \circledast (M_k^T M_k) + \alpha_{M_i} I\right]^{-1}\right]_+ \quad (15)$$

Such update rule can be computed distributively in a similar way, like it was described for matrices factorization.

For the grid of two nodes M_j and M'_k should be split in chunks of columns and evenly distributed between the nodes. Where $M'_k = (M_{k_1}|M_{k_2})^T$ and $M_j = (M'_{j_1}|M'_{j_2})^T$, the first node is responsible for M_{k_1}, M'_{j_1} and the second for M_{k_2}, M'_{j_2}.

In such distributed model every node has to calculate matrix $(M_{j_t}^T M_{j_t})$ of size $k \times k$ and send it to aggregator. Aggregator after receiving data from all nodes merges blocks into one matrix K_j and sends corresponding parts to the nodes. The same step should be done for calculation of matrix $(M_k^T M_k)$.

On the next step nodes have to calculate $S_t = \left[\left(M_j^T M_j\right)_t \circledast (M_k^T M_k)_t + \alpha_{M_i} I\right]_t^{-1}$ and send then to the aggregator which after receiving of all parts needs merge them, calculate S^{-1}, divide result on blocks and send them to corresponding nodes. After that nodes can calculate parts of the result matrix as $(M_j \odot M_k)_t + (S^{-1})_t$.

Therefore, the distributed model described in Section 5 can be easily transformed for non-negative factorization of large tensors.

8 Conclusion

We have combined the GPU-based and distributed algorithms, and also paid special attention to memory usage, which allows larger input matrices to be factorized. The experiments showed the constructed model is effective. It can be used to perform the tasks of industrial scale to factorize sparse matrices of large dimension with an acceptable time using available computing resources.

Proposed distributed model can be easily modified to speed up non-negative factorization of large tensors.

References

1. Deerwester, S., Dumais, S.T., Furnas G.W., Landauer T.K., Harshman, R.: Indexing by latent semantic analysis. JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE, 391–407, (1990)
2. Xu, Wei and Liu, Xin and Gong, Yihong: Document Clustering Based on Non-negative Matrix Factorization. Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval, SIGIR '03, ACM, New York, NY, USA, 267–273 (2003)
3. Shahnaz, Farial and Berry, Michael W. and Pauca, V. Paul and Plemmons, Robert J.: Document Clustering Using Nonnegative Matrix Factorization. Inf. Process. Manage., Pergamon Press, Inc., 373–386 (2006)
4. Landauer, T.K. and Foltz, P.W. and Laham, D.: An introduction to latent semantic analysis Discourse processes, ABLEX PUBLISHING CO, 259–284 (1998)
5. Mihalcea, R., Corley, C., Strapparava, C.: Corpus-based and knowledge-based measures of text semantic similarity IN AAAI06, AAAI Press, Menlo Park, CA; Cambridge, MA; London 775–780 (2006)
6. de Cruys, Tim Van: A non-negative tensor factorization model for selectional preference induction. Natural Language Engineering, 417-437,(2010)
7. Brett W. Bader and Tamara G. Kolda: MATLAB Tensor Toolbox Version 2.5 Available online, http://www.sandia.gov/ tgkolda/TensorToolbox/(2012)
8. Khushboo Kanjani: Parallel Non Negative Matrix Factorization for Document Clustering Texas A & M University (2007)
9. Kysenko, V., Rupp, K., Marchenko, O., Selberherr, S., Anisimov, A.: GPU-Accelerated Non-negative Matrix Factorization for Text Mining. LNCS, vol. 7337, Springer (2012)

10. Liu, Chao and Yang, Hung-chih and Fan, Jinliang and He, Li-Wei and Wang, Yi-Min: Distributed Nonnegative Matrix Factorization for Web-scale Dyadic Data Analysis on Mapreduce Proceedings of the 19th International Conference on World Wide Web, WWW '10, Raleigh, North Carolina, USA, ACM, 681–690 (2010)
11. Daniel D. Lee and H. Sebastian Seung: Algorithms for Non-negative Matrix Factorization In NIPS, MIT Press, 556–562 (2000)
12. M. Naumov, L. S. Chien, P. Vandermersch, U. Kapasi: CUDA CUSPARSE Library NVIDIA, San Jose, CA (2010)
13. NVIDIA: CUBLAS Library User Guide Availiable online, http://docs.nvidia.com/cublas/index.html, (2013)
14. Cickocki, A., Zdunek, R., Anh Huy Phan, Shun-Ichi Amari: Non-negative matrix and tensor factorizations: applications to exploratory multiway data analysis and blind source separation Fabulous, Singapore, 237-240 (2009)