GPU-accelerated Large-Scale Non-negative Matrix Factorization Using Spark

Bing Tang(✉), Linyao Kang, Yanmin Xia, and Li Zhang

School of Computer Science and Engineering,
Hunan University of Science and Technology, Xiangtan 411201, China
btang@hnust.edu.cn

Abstract. Non-negative matrix factorization (NMF) has been introduced as an efficient way to reduce the complexity of data compress and its ability of extracting highly-interpretable parts from data sets, and it has also been applied to various fields, such as recommendations, image analysis, and text clustering. However, as the size of the matrix increases, the processing speed of non-negative matrix factorization algorithm is very slow. To solve this problem, this paper proposes a parallel algorithm based on GPU for NMF in Spark platform, which makes full use of the advantages of in-memory computation mode and GPU Single-Instruction Multiple-data Streams mode. The new GPU-accelerated NMF on Spark platform is evaluated in a 4-nodes Spark heterogeneous cluster using Google Compute Engine by configuring each node a NVIDIA K80 GPU card, and experimental results indicate that it is competitive in terms of computational time against the existing solutions on a variety of matrix orders. It can achieve a high speed-up, and also can effectively deal with the non-negative decomposition of higher-order matrices, which greatly improves the computational efficiency.

Keywords: Non-negative matrix factorization · GPU · CUDA · Spark

1 Introduction

Non-negative matrix factorization is a matrix decomposition approach which decomposes a non-negative matrix into two low-rank matrices constrained to have nonnegative elements [4,5]. This results in a reduced representation of the original data that can be seen either as a feature extraction or as a dimensionality reduction technique. The widespread usage of the NMF is due to its ability of providing new insights and relevant information about the complex latent relationships in experimental data sets. Since Lee and Seung's Nature paper [4,5], NMF has been extensively studied and has a great deal of applications in science and engineering. It has become an important mathematical method in machine learning and data mining, and has been widely used in feature extraction, image

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2019
Published by Springer Nature Switzerland AG 2019. All Rights Reserved
H. Gao et al. (Eds.): CollaborateCom 2018, LNICST 268, pp. 189–201, 2019.
https://doi.org/10.1007/978-3-030-12981-1_13

analysis, recommendation systems, pattern recognition, signal analysis, bioinformatics and etc. [6–8]. Unlike other factorization methods (e.g., PCA, ICA, SVD, VQ, etc.), NMF can be interpreted as a parts-based representation of the data because only additive combinations are allowed. In contrast to PCA and ICA, NMF is strictly required that the entries of both resulting matrices are non-negative. Such a constraint is very meaningful in many applications, in which the data representation is purely additive, for instance, the user ratings of e-commerce websites are usually non-negative values, and image pixels are non-negative values.

The main problem of NMF is that the original matrix is usually high-order matrix, which makes the computational complexity very high. Therefore, the parallel algorithm of NMF gradually attracts more attentions, and some parallel NMF algorithms have been proposed. Although the parallelization of NMF can improve the computational efficiency to a certain extent, parallel algorithms should be matched to the machine hardware architecture, and should have strong scalability, that is, the ability to effectively utilize increased processor resources.

Accelerating HPC applications is currently under extensive research using new hardware technologies such as the recent Central Processing Units (CPUs) that are getting multiple processor cores for parallel computing, Graphics Processing Units (GPUs) that process huge data blocks in parallel, hybrid CPUs/GPUs computing that is a very common solution for HPC. GPUs are getting more attention than other HPC accelerators due to their high computation power, strong performance, functionality and low price. The modern GPU is not only a powerful graphic engine, but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth [10]. They are now used to accelerate graphics and some general applications with high data parallelism (GPGPU) due to the availability of Application Programming Interfaces (APIs), such as Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL).

Spark is a distributed in-memory computation framework which was proposed by AMPLab of University of California at Berkeley in 2009, and is based on a framework of processing large amounts of data in memory [12,13]. It supports four programming languages, Scala, Java, Python, and R. Resilient Distributed Datasets (RDD) is a new concept proposed by Spark for data collections. RDD can support coarse-grained write operations [11]. Spark caches a particular RDD into memory, and the next operation can read directly from memory. The data is not written to disk, saving a lot of disk I/O overhead. Experimental performance evaluation confirmed that Spark's performance has increased by dozens or even 100 times compared to Hadoop, which relies on MapReduce model [1] and data being stored in a distributed file system called HDFS, rather than in memory.

Currently, some parallel approaches for non-negative matrix factorization have been proposed, for example, high performance approaches using message passing interface [2], GPU-accelerated approaches [3,9], and Hadoop-based MapReduce approaches [6,7], etc. These approaches mainly utilize the multi-

core characteristics of the system, and there is still the potential to improve performance by utilizing memory, CPU and GPU resources together.

This paper proposes a Spark-based in-memory computing model and a GPU-based acceleration model to develop scalable NMF parallel algorithm, which takes advantages of both GPU and in-memory computing, to obtain a highly scalable parallel NMF algorithm. The algorithm can be automatically extended to support the processing of large-dimensional non-negative matrices, so that the algorithm can easily adapt to Internet big data processing.

The rest of the paper is organized as follows. Section 2 introduces the mathematical fundamental of NMF. Section 3 describes the general parallel principle of NMF. Section 4 describes the architecture of GPU-accelerated Spark platform. Section 5 presents GPU-accelerated NMF on Spark. Section 6 presents performance evaluation results, which is followed by the final section concludes the whole paper.

2 Non-negative Matrix Factorization

Non-negative matrix factorization (NMF) seeks to approximate a non-negative $n \times m$ matrix V (in this context, a matrix is called non-negative if all of its elements are non-negative) by a product $V \approx WH$ of non-negative matrices W and H of dimensions $n \times r$ and $r \times m$, respectively, with a given and typically low maximal rank r. Usually, r is chosen to satisfy $r \ll min\{m, n\}$ such that WH can be thought of as a compressed form of the original data. It forms the basis of unsupervised learning and data reduction algorithms with applications to image recognition, speech recognition, data mining and collaborative filtering, etc.

NMF is able to represent a large input dataset as the linear combination of a reduced collection of elements named *factors*. In this way, W contains the reduced set of r factors, and H stores the coefficient of the linear combination of such factors that rebuilds V. NMF iteratively modifies W and H until their product approximates to V. Such modifications, composed by matrix products and other algebraic operations, are derived from minimizing a cost function that describes the distance between WH and V. Lee and Seung presented two NMF algorithms based on multiplicative update rules whose objective functions are *Square of Euclidean Distance* (SED) and *Generalized Kullback-Leibler Divergence* (GKLD), respectively:

$$E(V||WH) = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\left(v_{ij} - \sum_{k=1}^{r} w_{ik}h_{kj}\right)^2 \tag{1}$$

$$D(V||WH) = \sum_{i,j}\left(v_{ij}\log\frac{v_{ij}}{(WH)_{ij}} - v_{ij} + (WH)_{ij}\right) \tag{2}$$

Then, the objective of NMF is converted to optimize the following: $\min_{W,H} E(V||WH)$ or $\min_{W,H} D(V||WH)$, and $s.t.\ W, H \geq 0,\ \sum_{i=1}^{n} w_{ij} = 1 \quad 1 \leq j \leq r$.

For the purpose of this paper, we define SED as the objective function, so we have $\min(\|V - WH\|_F^2)$, which leads to the updating rules for matrices H and W:

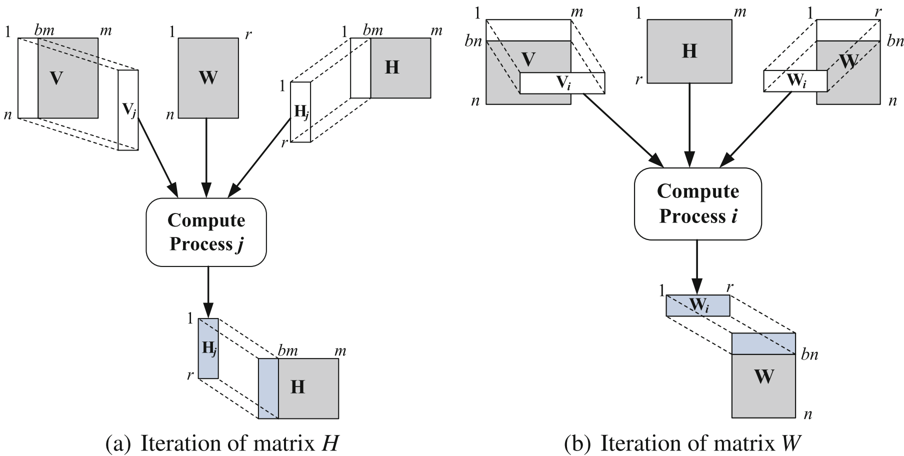
$$h_{ij} = h_{ij} \frac{(W^T V)_{ij}}{(W^T W H)_{ij}} \tag{3}$$

$$w_{ij} = w_{ij} \frac{(V H^T)_{ij}}{(W H H^T)_{ij}}. \tag{4}$$

3 Parallel Non-negative Matrix Factorization

Before describing our experimental study, we briefly introduce the main existing parallel techniques of NMF. By analyzing Eqs. (1) and (2), we can get the basic principle of iteration calculation of NMF in parallel manner. Matrix operations are performed in blocks. The block-based parallel updating rules for matrices H and W over multi-processes has shown in Fig. 1, and the size of b_m can be adjusted according to the hardware configurations. At the time of initialization, initial W and H are produced. As you see, the size of matrix W is $n \times r$, the size of the matrix block V_j is $n \times b_m$, and the size of the matrix block H_j is $r \times b_m$, and finally the updated matrix block H_j is obtained. As shown in Fig. 1(b), the new matrix H is used to compute the new matrix block W_i, and so on. Matrix H and W are updated alternatively.

It can be seen from the analysis, the original matrix V is equivalent to a read-only variable, which is shared among all processes. With the iteration, matrix W and H need to be synchronized among all processes. The algorithm works by iteratively all-gathering the entire matrix H or W to each processor and then performing the Local Update Computations to update the W_i or H_j.



(a) Iteration of matrix H (b) Iteration of matrix W

Fig. 1. Block-based parallel updating rules for matrices H and W over multi-processes.

4 Architecture of GPU-accelerated Spark Platform

4.1 Spark

Conceptually, Apache Spark is an open-source in-memory data analytics cluster computing framework. As a MapReduce-like cluster computing engine, Spark also possesses good characteristics such as scalability, fault tolerance as MapReduce does. The main abstraction of Spark is resilient distributed datasets (RDDs), which make Spark be well qualified to process iterative jobs, including PageRank algorithm, K-means algorithm and etc. RDDs are unique to Spark and thus differentiate Spark from conventional MapReduce engines. In addition, on the basis of RDDs, applications on Spark can keep data in memory across queries and reconstruct automatically data lost during failures. RDD is a read-only data collection, which can be either a file stored in an external storage system, such as HDFS, or a derived dataset generated by other RDDs. RDDs store much information, such as its partitions, and a set of dependencies on parent RDDs called lineage. With the help of the lineage, Spark recovers the lost data quickly and effectively. Spark shows great performance in processing iterative computation because it can reuse intermediate results, keep data in memory across multiple parallel operations.

4.2 Introduction to Architecture

Modern GPUs are now capable of general computing. Due to the popularity of the CUDA on Nvidia GPUs, which can be considered as a C/C++ extension, we will mostly follow CUDA terminologies to introduce GPU computing. Current generations of GPUs are used as accelerators of CPUs and data are transferred between CPUs and GPUs through PCI-E buses. NVIDIA GPU programming is generally supported by the NVIDIA CUDA environment. A program on the host (CPU) can call a GPU to execute CUDA functions called kernel.

GPU is a multi-core processor designed to parallelizable computational intensive tasks. It has very high computational processing power and data throughput. In scientific research and practical applications, the parallelizable computing task modules with less logical processing in the system are often transplanted to the GPU for execution, and a large execution performance improvement can usually be achieved.

However, Spark cluster will slow down when processing extremely large-scale data sets, especially when the node number is not very high. At the same time, more and more developers use GPUs for parallel computing to obtain high throughput and performance. Combing Spark with GPU, the mixed architecture is quickly becoming an emerging technology, which embeds the GPU into Spark, implements CPU/GPU integration, and builds an efficient heterogeneous parallel system.

In the CPU/GPU heterogeneous parallel cluster, the CUDA-based GPU acceleration technology is used, and the Spark computing tasks are accelerated

by GPU. The basic idea is that part of operations of the Spark RDD are transferred to the GPU cores. GPU code execution flows are: (1) copy data from main memory to GPU global memory; (2) GPU is driven by CPU instructions; (3) GPU parallel processing in each core; (4) GPU returns results to main memory. According to this idea and combined with Spark workflow, the GPU code is encapsulated, and then the data is transmitted between Spark Worker and GPU. The basic principle of Spark-GPU fusion is shown in Fig. 2.

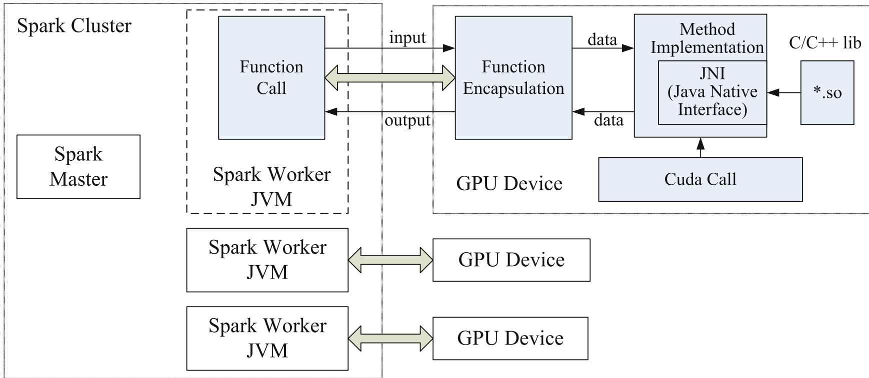


Fig. 2. Architecture of GPU-accelerated Spark platform.

From the perspective of programming language, since the GPU program is usually developed in C/C++ language, and the Spark platform uses Java language for program develop, Java's JNI (Java Native Interface) technology provides a solution to bridge the GPU and Spark, through code encapsulation to implement interfaces for the Worker to call. Several JNI tools for GPU programming can be used. For example, JCuda[1] is a development kit that provides bindings to the CUDA runtime, which currently includes multiple packages such as JCublas, JCufft, JCurand, JCusparse, JCusolver, JCudpp, JNpp, and JCudnn etc. It is convenient to write GPU programs in Java language. User-defined other GPU programs written in C/C++ can also be called after being packaged into Java functions.

For the developers, a bidirectional transmission channel between the main memory and the GPU global memory should be established. If the operation of the RDD is transferred to the GPU core, high-speed data transmission between the main memory and the GPU global memory is required, which is also implemented by function encapsulations, as is demostrated in Fig. 2.

[1] http://www.jcuda.org.

5 GPU-accelerated NMF on Spark

5.1 GPU-accelerated NMF

As we demostrated the matrix iterative process in Eqs. (3), (4) and Fig. 1, the main principle of GPU-based parallel NMF is presented in Fig. 3. The basic idea of GPU-based parallel NMF is to design several kernel functions to implement update rules for matrix H and W. H and W are blockwise transferred. In Fig. 3, circled operations denote CUDA kernels, and ".*" and "./" denote pointwise matrix operations, multiplication and division, respectively. Most of the matrix operations can be implemented using the library of Cublas and Cusparse, together with two self-defined operations, dot multiplication and dot division. In order to reduce the programming difficulty, JNI technology is used to transfer the CUDA programs to Java function encapsulations, which are called by Spark executors.

5.2 GPU-accelerated NMF on Spark

Spark has advantages in iterative computing, and GPU has advantages in numerical calculation of vectors and matrices. In the Spark-GPU fusion platform, fast memory read and write, combined with GPU acceleration, can play their respective advantages to improve performance. NMF calculation is started and controlled by Spark driver. The Workers calculate the parallel tasks iterately in a distributed manner. Workers are optimized with the highest speed using the GPU device and running the GPU kernel functions to complete the task. All intermediate results are written to the memory in each iteration, and exchanged among the Workers, and sent to the GPU global memory. Until the iterations are terminated, the tasks are completed and the results are written to HDFS.

The whole algorithm is described in Algorithm 1. The matrix V is broadcasted to all executors, and each worker obtains the corresponding matrix block W_i or H_j from RDD. In the Spark platform, after the Action operator is triggered, all accumulated operators form a directed acyclic graph. Task is splitted into different stages based on different dependencies between RDDs. One stage consists of a series of function execution pipelines. The stages of GPU-accelerated NMF through RDD are listed as follows:

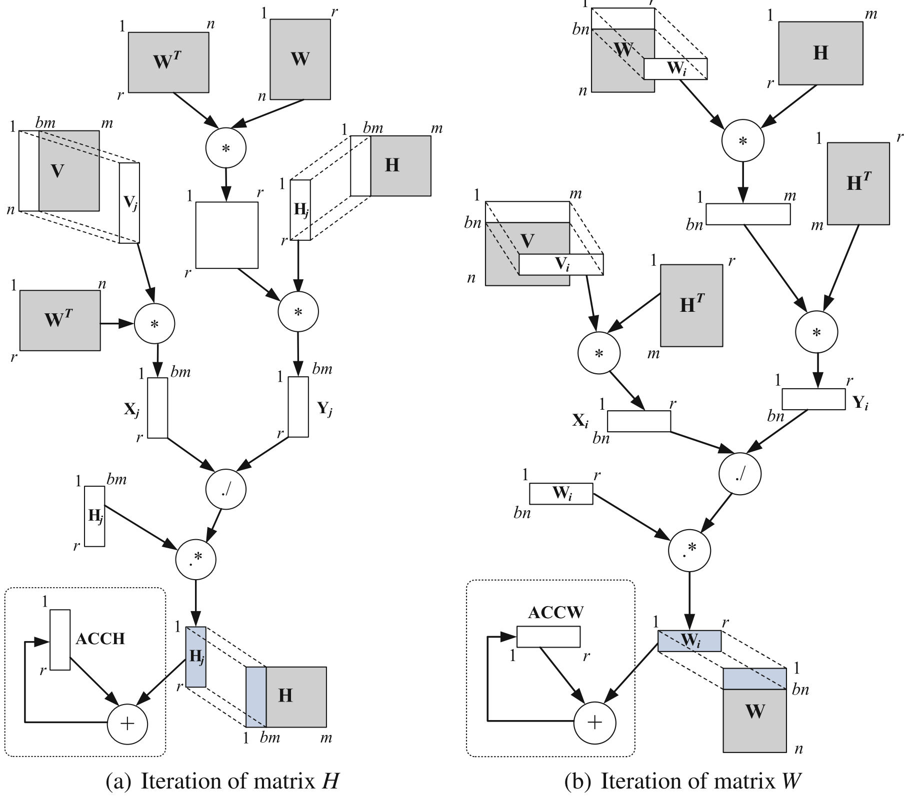
- **Stage 1:** Read and convert matrix W and H, perform `mapPartition` function to update H blocks;
- **Stage 2:** Splice all blocks of H after one iteration through perform a `collect` operation;
- **Stage 3:** Read and convert matrix W and H, perform `mapPartition` function to update W blocks;
- **Stage 4:** Splice all blocks of W after one iteration through perform a `collect` operation, and prepare for the next iteration.

Then, iteratively preform the above four stages. The method of caching data in memory is much faster than in file system for each iteration. When the convergence condition is reached, the matrices updating is terminated, and the results are then written to HDFS.

Algorithm 1. GPU-accelerated NMF on Spark

Input: Original matrix $V_{n \times m}$, low rank r and iteration times *iter*
Input: Context of Spark Environment *sc*
Input: Number of executors *en* and number of data partitions *pn*
Input: Data collection of matrix elements *dcM* and *dcH* for matrices M and H
Input: Data collection in the form of RDD *rddM* and *rddH* for matrices M and H
Output: Matrices $W_{n \times r}$ and $H_{r \times m}$ after decomposition

1. generate initial W, H by random
2. $dcW \leftarrow W, dcH \leftarrow H$
3. broadcast V
4. **for** k=1: *iter* **do**
5. $rddH \leftarrow sc.parallelize(dcH, pn)$
6. //update H
7. call $rddH.mapPartition($`mapH`$)$
8. **function** mapH(data, result)
9. $X \leftarrow gpu_multiply(W^T, V)$
10. $WW \leftarrow gpu_multiply(W^T, W)$
11. $Y \leftarrow gpu_multiply(WW, data)$
12. $data \leftarrow gpu_dot_multiply(data, X)$
13. $result \leftarrow gpu_dot_divide(data, Y)$
14. *return result*
15. **end function**
16. $dcH \leftarrow rddH.collect()$
17. $rddW \leftarrow sc.parallelize(dcW, pn)$
18. //update W
19. call $rddW.mapPartition($`mapW`$)$
20. **function** mapW(data, result)
21. $X \leftarrow gpu_multiply(V, H^T)$
22. $WH \leftarrow gpu_multiply(W, H)$
23. $Y \leftarrow gpu_multiply(WH, H^T)$
24. $data \leftarrow gpu_dot_multiply(data, X)$
25. $result \leftarrow gpu_dot_divide(data, Y)$
26. *return result*
27. **end function**
28. $dcW \leftarrow rddW.collect()$
29. **end for**
30. $W \leftarrow dcW, H \leftarrow dcH$



(a) Iteration of matrix H (b) Iteration of matrix W

Fig. 3. GPU implementation of iteration.

6 Performance Evaluations

6.1 Experiment Configurations

For our experiments we have used four n1-standard-4 instances of Google Compute Engine, and each instance is configured with 4 vGPU, 15 GB memory and 100 GB SSD hard disk in asia-east1 district. Each instance is also configured with a NVIDIA K80 GPU with 2496 CUDA cores and 12 GB global memory. In the 4-nodes cluster, 64 bits Ubuntu 16.04 LTS is installed, and other software packages include Hadoop 2.7, Spark 2.3, JDK 1.8 and CUDA 9.0.

6.2 Algorithms for Comparison

Serial NMF. Serial NMF algorithm is performed in a single thread using CPU only. According to the Eqs. (3) and (4), the method of alternately updating W and H are used to obtain the decomposition results by performing multiple iterations.

GPU-based NMF. GPU-based NMF algorithm is also performed in a single thread but with one GPU device support. As you see in Fig. 3, alternately updating W and H are accelerated by GPU, implemented using the library of Cublas and Cusparse, together with two self-defined operations, dot multiplication and dot division.

Spark-Based NMF Without GPU Support. For this algorithm, NMF is computed in a Spark cluster, and each node has no GPU device. Similar to Algorithm 1, in the two stages of *rddH.mapPartition* and *rddW.mapPartition*, there is no GPU support for the updating of H and W, and only CPU for matrix operations in each iteration.

6.3 Result Analysis

In the experiments, we conducted performance evaluations using four algorithms: (*i*) Serial NMF, (*ii*) GPU-based NMF, (*iii*) Spark-based NMF without GPU support, (*iv*) Spark-based NMF with GPU support which is proposed in this paper and developed on Spark-GPU fusion platform. We designed three performance comparisons to validate the proposed new algorithm. We select some typical matrix dimensions, and the low rank r is set to 10, the number of iterations is 100.

Performance of GPU Speedup. We performed GPU-based NMF in a single node, and we varied the matrix dimensions as you see in Fig. 4. We measured the computation time, and then we also perform the serial NMF in the same node so

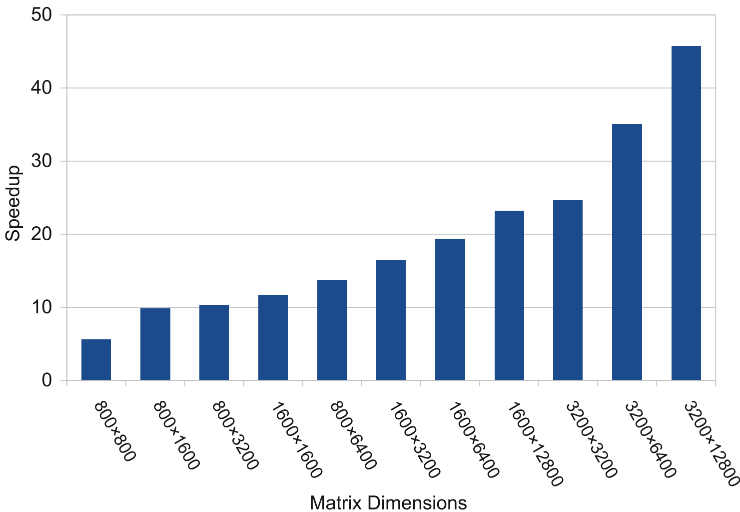


Fig. 4. Performance of GPU speedup.

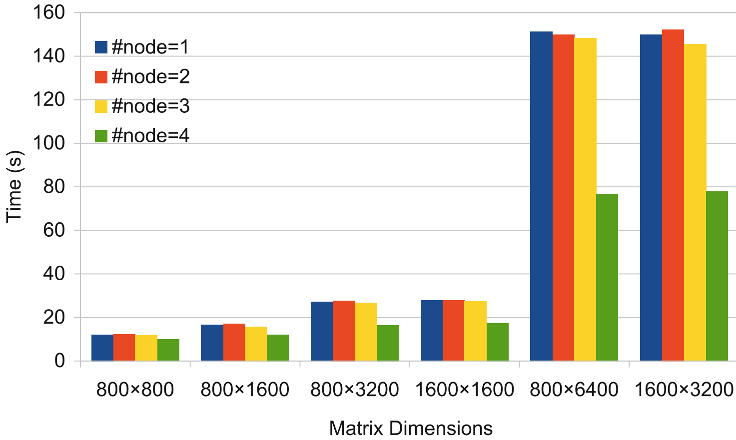


Fig. 5. Performance of NMF on Spark.

as to calculate the GPU speedup to validate the effective of GPU acceleration. The speedup is defined as the ratio of the computation time of the single node serial method to the computation time of the single node GPU method, that is, $Speedup = T_{serial}/T_{gpu_parallel}$. The speedup varies with matrix dimensions, and we have obtained maximum speedup of 45x for GPU when compared with CPU.

Performance of NMF on Spark. In this evaluation, we started the Spark cluster, and the number of worker nodes is varied from 1, 2, 3 to 4. We varied the matrix dimensions from $800 * 800$, $800 * 1600$, $800 * 3200$, $1600 * 1600$, $800 * 6400$ to $1600 * 3200$, and measured the computation time of NMF in Spark platform, and results are shown in Fig. 5. When the number of nodes is 4, we set the number of Spark executors to 16, and as the increase of the matrix dimensions, the advantages of 4 nodes are becoming more and more obvious. Compared with 3 nodes Spark platform, the computation time of 4 nodes saves about 50% of the time.

Performance of NMF on Spark with GPU Support. In the last evaluation, we started the Spark cluster, the number of nodes is 4, and we varied the matrix dimensions from $6400 * 6400$, $3200 * 25600$, $6400 * 12800$ to $6400 * 25600$, and compared GPU support with Non-GPU support. As can be seen from Fig. 6, in the 4-node Spark platform, the computation time of NMF with GPU is smaller than without GPU. When the size of matrix is $6400 * 25600$, NMF on Spark with GPU support saves about 10.8% of the time. NMF on GPU-accelerated Spark platform obviously shows execution efficiency.

Due to the mathematical fundamental of NMF and the blockwise-based parallel principle, there are frequent data distributions and data collections among all executors, the communication cost is very high for the NMF on Spark. However, compared with data distributions and data collections, the execution of

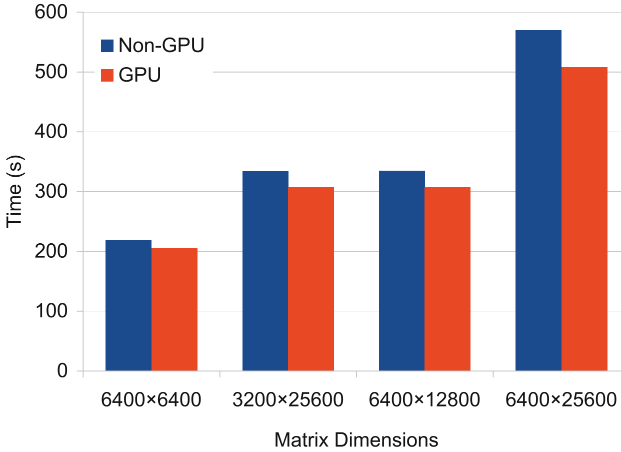


Fig. 6. Performance of NMF on Spark with GPU support.

mapPartition function takes much less time due to the GPU acceleration. From the perspective of time analysis, communication and data exchange are the bottlenecks of NMF parallel algorithm. NMF on GPU-accelerated Spark platform still has great potential for improvement.

7 Conclusion

This paper implements the GPU-accelerated parallel NMF algorithm on Spark platform. Through the performance evaluations, experimental results proved that the combination of Spark-based in-memory computing and GPU has higher execution efficiency. In the heterogeneous CPU/GPU cluster, nodes have large memory resources and GPU multi-core resources, the advantages of distributed storage between nodes and data sharing within nodes should be utilized. This model can effectively improve the parallel computing performance in multi-cores environment. It is an efficient and feasible parallel programming strategy. It can support the processing of ultra-large-scale high-dimensional non-negative matrix factorization, and will further expand the application fields of non-negative matrix factorization. However, the GPU-accelerated NMF algorithm on Spark platform designed in this paper still needs to be improved. First, some improvements can be made to overlap calculations and data transmissions; Second, some optimizations can also be made for non-negative factorization of sparse matrices.

Acknowledgements. This work is supported by the National Natural Science Foundation of China under grant no. 61602169 and 61702181, and the Natural Science Foundation of Hunan Province under grant no. 2018JJ2135 and 2018JJ3190, as well as the Scientific Research Fund of Hunan Provincial Education Department under grant no.16C0643.

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
2. Kannan, R., Ballard, G., Park, H.: A high-performance parallel algorithm for non-negative matrix factorization. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, 12–16 March 2016, pp. 9:1–9:11 (2016)
3. Kysenko, V., Rupp, K., Marchenko, O., Selberherr, S., Anisimov, A.: GPU-accelerated non-negative matrix factorization for text mining. In: Bouma, G., Ittoo, A., Métais, E., Wortmann, H. (eds.) NLDB 2012. LNCS, vol. 7337, pp. 158–163. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31178-9_15
4. Lee, D.D., Seung, H.S.: Learning the parts of objects by non-negative matrix factorization. Nature **401**(6755), 788–791 (1999)
5. Lee, D.D., Seung, H.S.: Algorithms for non-negative matrix factorization. In: Leen, T.K., Dietterich, T.G., Tresp, V. (eds.) Advances in Neural Information Processing Systems, Papers from Neural Information Processing Systems (NIPS), Denver, CO, USA, vol. 13, pp. 556–562. MIT Press (2000)
6. Liao, R., Zhang, Y., Guan, J., Zhou, S.: CloudNMF: a MapReduce implementation of nonnegative matrix factorization for large-scale biological datasets. Genomics Proteomics Bioinf. **12**(1), 48–51 (2014)
7. Liu, C., Yang, H., Fan, J., He, L., Wang, Y.: Distributed nonnegative matrix factorization for web-scale dyadic data analysis on MapReduce. In: Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, 26–30 April 2010, pp. 681–690 (2010)
8. Luo, X., Zhou, M., Xia, Y., Zhu, Q.: An efficient non-negative matrix-factorization-based approach to collaborative filtering for recommender systems. IEEE Trans. Ind. Inf. **10**(2), 1273–1284 (2014)
9. Mejía-Roa, E., Tabas-Madrid, D., Setoain, J., García, C., Tirado, F., Pascual-Montano, A.D.: NMF-mGPU: non-negative matrix factorization on multi-GPU systems. BMC Bioinf. **16**, 43:1–43:12 (2015)
10. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM Comput. Surv. **47**(4), 69:1–69:35 (2015)
11. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, 25–27 April 2012, pp. 15–28 (2012)
12. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Nahum, E.M., Xu, D. (eds.) 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2010, Boston, MA, USA, 22 June 2010. USENIX Association (2010)
13. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)