A Performance Evaluation of Computing Singular Value Decomposition of Matrices on Central and Graphics Processing Units

Dušan B. Gajić*, Đorđe Manoilov**
* Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
** Mathematical Institute of the Serbian Academy of Sciences and Arts, Belgrade, Serbia
dusan.gajic@uns.ac.rs, manoilov88@gmail.com

***Abstract*—This paper presents a comparison of processing times for different implementations of the singular value decomposition (SVD) of matrices, performed on multicore central processing units (CPUs) and manycore graphics processing units (GPUs). The aim of the paper is to identify which of the considered computing platforms and programming environments leads to the fastest computation of the SVD. The results show that the MATLAB implementation, processed in parallel on the multicore CPU, outperforms all other considered approaches for computing the SVD. It can also be concluded that the GPU implementation of the SVD, which is part of the Nvidia CUDA cuSolver/cuBLAS library, does not efficiently use the computational resources available on GPUs. Therefore, it would be beneficial to develop a custom GPU implementation for computing the SVD, based on some of the more advanced algorithms for this purpose, which would better exploit the advantageous features of the GPU architecture. Plans for future work include the development of this implementation and its use for efficient latent semantic analysis of large term-document matrices.**

I. INTRODUCTION

The singular value decomposition (SVD) factorizes a real or complex rectangular matrix into a product of three matrices – two orthogonal and one diagonal [9, 20, 22]. This method is widely used for problems in areas as diverse as linear algebra (e.g. solving homogeneous linear equations, low-rank matrix approximation), statistics (principal components analysis - PCA), natural language processing (latent semantic analysis - LSA), information retrieval (latent semantic indexing - LSI), signal processing (Karhunen–Loève transform), and machine learning (recommender systems) [9, 13, 17, 18, 22]. Time required for computing the SVD of a matrix is a limiting factor in many of these practical applications, since the size of a typical problem instance that needs to be handled is, currently, of the order of gigabytes. Therefore, efficient computation of the SVD is of critical importance for its use in science and engineering.

Two main contemporary computing platforms, which are available for performing numerical algorithms such as computing the SVD of a matrix, are central processing units (CPUs) and graphics processing units (GPUs). CPUs are still built using the single instruction, single data (SISD) architecture, although they now feature multiple cores [18]. GPUs, on the other hand, have a single instruction, multiple data (SIMD) manycore and highly-parallel architecture [1, 3, 15]. Their computational resources became accessible for performing non-graphics general-purpose algorithms (a technique known as GPGPU – general-purpose computing on the GPU) only in the last decade. GPGPU has been a topic of a very fast-growing research interest, with researchers showing significant speed-ups of different programs after porting them to GPUs [3, 5, 6, 7, 15]. Further, different programming approaches, such as using C/C++ (or CUDA C on the GPU) combined with BLAS/LAPACK (Basic Linear Algebra Subprograms/Linear Algebra PACKage) routines or performing computations in MATLAB, can be used both on CPUs and GPUs [7, 8, 11, 12, 15, 16]. Therefore, it is compelling to perform a comparison of different implementations of the SVD processed on CPUs and GPUs, in order to establish the most efficient method for its computation. These results can then be used for developing new implementations of the SVD with improved performance. This can, in turn, help to extend the set of problem instances that can be efficiently handled in practice.

In this paper, we present a performance comparison of computing the SVD of matrices using different programming frameworks for both CPUs and GPUs. The main goal of the presented research is to identify the most suitable environment, out of the considered computational platforms and programming approaches, for performing the SVD. This evaluation is performed in terms of processing times for different implementations on CPUs and GPUs. Further, the research offers answers to the question of how efficiently do different implementations of the same algorithm use the computational resources available on distinct architectures, such as the ones of central and graphics processing units.

The specific problem that motivated the research presented in this paper is the performance of latent semantic analysis of large term-document matrices that mainly relies on the fast computation of SVD [22, 23]. Therefore, the presented research sheds light on the steps required to develop a high-performance GPU implementation of LSA.

The paper is organized as follows. In Section II we present the SVD and its mathematical foundations, as well as a short introduction to the use of the SVD for

latent semantic analysis. Implementations of the SVD on CPUs and GPUs are described in Section III. Experimental settings and results are presented in Section IV. The final section of the paper offers main conclusions and offers some directions for further work.

II. THEORETICAL BACKGROUND

The singular value decomposition (SVD) is often used in situations where techniques such as Gaussian elimination or LU decomposition don't offer satisfactory answers [9, 20]. The application of SVD decomposes a real or complex rectangular matrix into a product of three matrices – two orthogonal and one diagonal. For a matrix **A** of size $M \times N$, the SVD is the factorization of the following form:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}, \tag{1}$$

where **U** is a column-orthogonal unitary matrix of size $M \times M$, **Σ** is a diagonal matrix of size $M \times N$ with elements $s_{ij} = 0$ if $i \neq j$ and $s_{ij} \geq 0$ in descending order along the diagonal, and $\mathbf{V}^{\mathrm{T}}$ is an $N \times N$ unitary matrix [9, 19, 20]. Elements on the diagonal of **Σ** are the singular values of **A**.

Therefore, using the SVD, a matrix **A** of size $M \times N$ is factorized as

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{bmatrix} =$$

$$= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}} =$$

$$= \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1M} \\ u_{21} & u_{22} & \cdots & u_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ u_{M1} & u_{M2} & \cdots & u_{MM} \end{bmatrix} \cdot$$

$$\cdot \begin{bmatrix} \sigma_{11} & 0 & \cdots & 0 & \cdots 0 \\ 0 & \sigma_{21} & \cdots & 0 & \cdots 0 \\ \vdots & \vdots & \ddots & \vdots & \cdots \vdots \\ 0 & 0 & \cdots & \sigma_{MN} & \cdots 0 \end{bmatrix} \cdot$$

$$\cdot \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1N} \\ v_{21} & u_{22} & \cdots & u_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ v_{N1} & v_{N2} & \cdots & u_{NN} \end{bmatrix}.$$

Depending on the values for M and N, the matrices **U**, **Σ**, and $\mathbf{V}^{\mathrm{T}}$ can take various shapes [20].

Example 1. A given (2×3) rectangular matrix **A**, taken from [24],

$$\mathbf{A} = \begin{bmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{bmatrix}$$

can be factorized by using the SVD as follows

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}} =$$

$$= \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \sqrt{12} & 0 & 0 \\ 0 & \sqrt{10} & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} & 0 \\ \frac{1}{\sqrt{30}} & \frac{2}{\sqrt{30}} & -\frac{5}{\sqrt{30}} \end{bmatrix}.$$

A. The Reduced SVD

The reduced SVD (also known as latent semantic indexing – LSI in information retrieval, or latent semantic analysis – LSA in natural language processing) is a mathematical method that projects queries and documents into a space with latent semantic dimensions [22]. It is based on the assumption that there is some underlying latent semantic structure in the data that is corrupted by the wide variety of used words and that this semantic structure can be discovered and enhanced by projecting the data (the term-document matrix and the queries) onto a lower-dimensional space using the SVD [22, 23, 24]. A term-document matrix describes the frequency of terms that occur in a collection of documents. In a term-document matrix, columns correspond to documents in the collection and rows correspond to terms.

For a term-document matrix **A**, we can perform an SVD using (1) as described in the previous section. Some of the computed singular values are very small in magnitude and, thus, negligible. Therefore, these values can be ignored and replaced by zeros. If we keep only K singular values, then **Σ** will contain all zeros, except for the first K entries along its diagonal. As such, we can reduce matrix **Σ** into $\mathbf{\Sigma}_{\mathbf{k}}$, which is an $K \times K$ matrix containing only the k singular values that we keep. We can also reduce **U** and $\mathbf{V}^{\mathbf{T}}$ into $\mathbf{U}_{\mathbf{K}}$ and $\mathbf{V}_{\mathbf{K}}{}^{\mathbf{T}}$, respectively [23]. Therefore, **A** can now be approximated by:

$$\mathbf{A}_{\mathbf{K}} = \mathbf{U}_{\mathbf{K}}\mathbf{\Sigma}_{\mathbf{K}}\mathbf{V}_{\mathbf{K}}{}^{\mathrm{T}}.$$

The decomposition of **A** using the SVD and the reduced SVD, which is obtained by keeping K singular values, is presented in Fig. 1. The following example illustrates the case of computing the reduced SVD of a (5×5) square matrix when $K = 3$.

Example 2. A given (5×5) square matrix **A**, taken from [24],

$$\mathbf{A} = \begin{bmatrix} 2 & 0 & 8 & 6 & 0 \\ 1 & 6 & 0 & 1 & 7 \\ 5 & 0 & 7 & 4 & 0 \\ 7 & 0 & 8 & 5 & 0 \\ 0 & 10 & 0 & 0 & 7 \end{bmatrix},$$

can be factorized by using the SVD in the following way

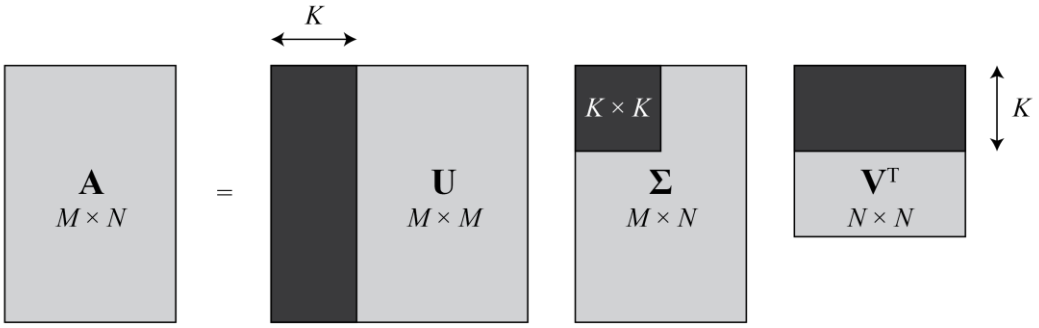

Figure 1. Decomposition of a rectangular matrix using the SVD and the reduced SVD.

$$\mathbf{U} = \begin{bmatrix} -0.54 & 0.07 & 0.82 & -0.11 & 0.12 \\ -0.1 & -0.59 & -0.11 & -0.79 & -0.06 \\ -0.53 & 0.06 & -0.21 & 0.12 & -0.81 \\ -0.65 & 0.07 & -0.51 & 0.06 & 0.56 \\ -0.06 & -0.8 & 0.09 & 0.59 & 0.04 \end{bmatrix},$$

$$\mathbf{\Sigma} = \begin{bmatrix} 17.92 & 0 & 0 & 0 & 0 \\ 0 & 15.17 & 0 & 0 & 0 \\ 0 & 0 & 3.56 & 0 & 0 \\ 0 & 0 & 0 & 1.98 & 0 \\ 0 & 0 & 0 & 0 & 0.35 \end{bmatrix},$$

$$\mathbf{V}^{\mathrm{T}} = \begin{bmatrix} -0.46 & 0.02 & -0.87 & 0 & 0.17 \\ -0.07 & -0.76 & 0.06 & 0.6 & 0.23 \\ -0.74 & 0.1 & 0.28 & 0.22 & -0.56 \\ -0.48 & 0.03 & 0.4 & -0.33 & 0.7 \\ -0.07 & -0.64 & -0.04 & -0.69 & -0.32 \end{bmatrix}.$$

When we set $K = 3$, i.e. if we consider only the first three singular values and try to reconstruct the original matrix, we get:

$$\mathbf{\Sigma} = \begin{bmatrix} 17.92 & 0 & 0 \\ 0 & 15.17 & 0 \\ 0 & 0 & 3.56 \end{bmatrix},$$

$$\hat{\mathbf{A}} = \begin{bmatrix} 2.29 & -0.66 & 9.33 & 1.25 & -3.09 \\ 1.77 & 6.76 & 0.9 & -5.5 & -2.13 \\ 4.86 & -0.96 & 8.01 & 0.38 & -0.97 \\ 6.62 & -1.23 & 9.58 & 0.24 & -0.71 \\ 1.14 & 9.19 & 0.33 & -7.19 & -3.13 \end{bmatrix},$$

where $\hat{\mathbf{A}}$ is a thus-obtained reconstruction of **A**. However, in practical applications, the typical purpose of these computations is not to actually reconstruct the original matrix, but to use the reduced dimensionality representation to identify similar words and documents [22]. Documents are then represented by row vectors in **V**, and document similarity is obtained by comparing rows in the matrix **VΣ** (note that documents are represented as row vectors since we are working with **V**, not $\mathbf{V}^{\mathbf{T}}$). Words are represented by row vectors in **U**, and word similarity can be measured by computing row similarity in **UΣ** [22].

III. Implementation

On the CPU, we compute the singular value decomposition using two programming frameworks - C/C++ (with Intel Math Kernel Libraries - MKL) and MATLAB. The MKL is a library of mathematical functions for Intel and other compatible CPUs, composed of BLAS and Linear LAPACK routines, fast Fourier transform (FFT) algorithms, and a number of other mathematical operations [12]. MATLAB is an interactive numerical computing environment and a programming language which can be used for tasks such as numerical computations, simulations, and data analysis and visualization [15].

On the GPU, we also use two different frameworks for performing the computation of the SVD. The first is MATLAB Parallel Computing Toolbox and the second is Nvidia CUDA (Compute Unified Device Architecture), extended with the cuSolver programming package featuring cuBLAS and cuSPARSE libraries [15, 16]. MATLAB's Parallel Computing Toolbox allows the use of multicore CPUs, GPUs, and clusters, for parallel processing of computationally intensive algorithms in MATLAB [15]. It includes special high-level data types, parallel loops, and numerical algorithms, which permit concurrent execution of programs. CUDA is a parallel programming architecture, framework, and language, developed by Nvidia, for the purposes of implementing and processing general-purpose algorithms on graphics processing units [16]. It supports a highly-parallel programming model, constructed around the high-throughput, high-latency GPU architecture.

IV. Experiments

A. Experimental Settings

The experiments were performed as follows.

First, we generate densely-populated square matrices containing single-precision floating numbers (each element uses 4 bytes of memory). The choice of square

matrices in the experiments corresponds to the use cases of computing the LSA where the number of terms is approximately equal to the number of documents. The matrix elements are generated randomly using the *rand* function, available in MATLAB and MATLAB's Parallel Computing Toolbox. For the same purposes in the C/C++ implementation using the Intel MKL, as well as in the CUDA/cuSolver implementation, we used the *rand* function from *cstdlib*, with pseudo-random generator number seed set using *srand*(*time*(*NULL*)) function.

For computing on the multicore CPU, we use the *LAPACKE_sgesvd* routine from Intel MKL [12]. For performing the SVD using MATLAB's Parallel Computing Toolbox on the GPU, we used the *gpuArray* data structure for storing matrix [15]. In CUDA, we first transferred the matrix to the GPU using pinned memory, in order to effectively use the PCIe bus between the CPU and the GPU, and then called the *cusolverDnSgesvd* function from the cuBLAS library [16].

The experiments were performed on a PC workstation specified in Table I. The software environment included the Windows 10 64-bit operating system running MATLAB 2015b, Intel MKL 2017, and cuSolver/cuBLAS libraries from the CUDA 7.5 SDK (Software Development Kit).

The SVD was performed on square matrices of size $N \times N$, for N = 256, 512, 1024, 2048, 4096, 8192, 16384. The computational times were recorded as average values for 10 program executions for each size of the input matrix.

B. Experimental Results

The results of the experiments are presented in Table II and Fig. 2. The sign '-' in Table II corresponds to situations where computations were not completed in 30 minutes, which was set as an upper time bound.

The summary of the results can be stated as follows. On the used computational platform, the MATLAB implementation of the SVD, processed in parallel on the multicore CPU, was from 1.5× to 3.8× faster than the C/C++ MKL implementation, which was also executed in parallel. The two considered GPU implementations, one using the MATLAB's Parallel Computing Toolbox and the other using CUDA with cuSolver/cuBLAS, were from 2.5× to 6.5× and from 20× to 70×, respectively, slower than the MATLAB CPU implementation of the SVD. The high-performance of the MATLAB SVD implementation can be attributed to its foundation in the Demmel and Kahan's improved version of Golub and Van Loan's algorithm for computing the SVD, which uses the technique of bi-diagonalization [9, 19, 20]. On the other hand, both of the GPU implementations of the SVD showed poor performance. Since both of them are based on the routines that are part of the Nvidia CUDA SDK, it can be suspected that the performance issues arise from the slow execution of these functions. Long computation times recorded using the implementations of these routines can be connected with the fact that they are currently optimized only for efficient processing of "tall" and "thin" matrices [16].

TABLE I
THE EXPERIMENTAL PLATFORM AND SOFTWARE VERSIONS

CPU	*Intel Xeon E5-1620*
core frequency	*3.6 GHz*
number of cores	*4*
number of threads	*8*
Memory	*32 GB DDR3 2133 MHz*
Operating System	*Windows 10 Pro 64-bit*
GPU	*Nvidia Quadro K620*
architecture	*Maxwell GM107*
memory	*2 GB DDR3*
number of cores	*384*
Library/Software version	
MATLAB	*2015b*
Intel MKL	*Intel Parallel Studio 2017XE*
CUDA	*7.5*

These experimental results stand in sharp contrast to significant speed-ups achieved using the GPU implementations of the LU and QR decompositions [7, 8], which are also available as part of the cuSolver/cuBLAS library [16]. When the considerable advantage in terms of raw processing power and memory bandwidth that GPUs have over CPUs is taken into account, it is clear that a custom GPU implementation of the SVD is required in order to efficiently use the computational resources available on graphics processors.

One of the approaches for developing this implementation can be founded on an algorithm for computing the SVD using the bi-diagonalization method, proposed by Lahabar and Narayanan in [11]. In this paper, a hybrid approach that uses both the CPU and the GPU for performing computation of the SVD is discussed. Two other algorithms that can be taken into consideration are an incremental, low-memory, large-matrix SVD algorithm, that has been proposed by Brand in 2006 [2], and a divide-and-conquer bi-diagonal SVD algorithm, proposed by Gu and Eisenstat in 1995 [10]. For the purposes of our research, this efficient GPU implementation of the SVD should be used as the core component of the system for the fast latent semantic analysis of large term-document matrices.

TABLE II
PROCESSING TIMES FOR DIFFERENT IMPLEMENTATIONS OF THE SVD DECOMPOSITION ON THE CPU AND THE GPU

N	Processing time [ms]			
	CPU		GPU	
	Intel MKL	MATLAB	MATLAB GPU	CUDA/ cuSolver
256	29	10	52	229
512	142	51	233	1036
1024	676	179	1163	8661
2048	3941	1335	6158	65705
4096	26573	11692	33480	848227
8192	140343	92614	232721	-
16384	-	688150	-	-

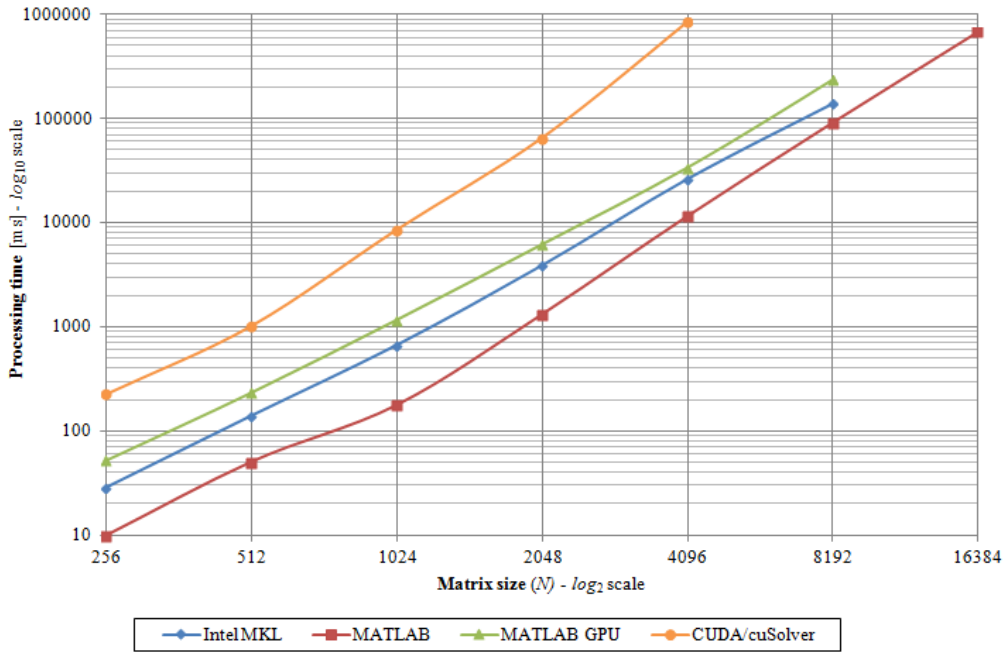

Figure 2. Processing times for different implementations of the SVD on the CPU and the GPU.

V. CONCLUSIONS

In this paper we presented a performance comparison of time efficiency of computing the singular value decomposition of matrices using four different implementations - two processed on central processing units and two on graphics processing units. The experiments were performed on dense square matrices containing single-precision floating point numbers that were generated randomly. The results show that the MATLAB implementation, performed on the CPU, offers best performance for all considered sizes of square matrices ($N \leq 16384$).

We can conclude that the GPU implementation of the SVD, which is part of the nVidia CUDA cuSolver/cuBLAS library, does not efficiently use the computational resources available on GPUs. Therefore, it would be beneficial to develop a custom GPU implementation for computing the SVD, based on some of the more advanced algorithms, which would better use advantageous features of the GPU architecture. Further work includes the development of this efficient SVD implementation for the GPU and its application for fast latent semantic analysis of large term-document matrices.

ACKNOWLEDGMENT

The research reported in this paper is partly supported by the Ministry of Education and Science of the Republic of Serbia, projects ON174026 (2011-2017) and III44006 (2011-2017).

REFERENCES

[1] T. M. Aamodt, "Architecting graphics processors for non-graphics compute acceleration", in *Proc. 2009 IEEE Pacific Rim Conf. Communications, Computers & Signal Processing*, Victoria, BC, Canada, 2009.

[2] M. Brand, "Fast Low-Rank Modifications of the Thin Singular Value Decomposition", Mitsubishi Electric Research Laboratories, Technical Report TR2006-059, May 2006.

[3] A. R. Brodtkorb, M. L. Sætra, T. R. Hagen, "Graphics processing unit (GPU) programming strategies and trends in GPU computing", *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, 2013, pp. 4-13.

[4] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, R. Harshman, "Indexing by latent semantic analysis", *Journal of the American Society for Information Science and Technology*, vol. 41, no. 6, 1990, pp. 391-407.

[5] D. B. Gajić, R. S. Stanković, "Computing spectral transforms used in digital logic on the GPU", in *GPU Computing with Applications in Digital Logic*, J. Astola, M. Kameyama, M. Lukac, R. S. Stanković (editors), Tampere International Center for Signal Processing - TICSP, Tampere, Finland, 2012, pp. 25-62.

[6] D. B. Gajić, R. S. Stanković, "Computing the Vilenkin-Chrestenson transform on a GPU", in *Journal of Multiple-Valued Logic and Soft Computing*, Old City Publishing, Philadelphia, USA, vol. 25, no. 1-4, 2015, pp. 317-340.

[7] D. B. Gajić, R. S. Stanković, M. Radmanović, "A Performance Comparison of Computing LU Decomposition of Matrices on the CPU and the GPU", in *Proc. ICEST* 2015, Technical University Sofia, Bulgaria, June 24 – 26, 2015, pp. 109-112.

[8] D. B. Gajić, R. S. Stanković, M. Radmanović, "A Performance Analysis of Computing LU and QR Matrix Decompositions on CPUs and GPUs", submitted to *International Journal of Reasoning-Based Intelligent Systems*.

[9] G. Golub, C. Van Loan, *Matrix Computations*, 3rd edition, The Johns Hopkins University Press, 1996.

[10] M. Gu, S. C. Eisenstat, "A Divide-and-Conquer Algorithm for the Bidiagonal SVD", *SIAM Journal on Matrix Analysis and Applications*, Vol. 16, No. 1, 1995, pp. 79-92.

[11] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th edition, Morgan Kaufmann, 2011.

[12] Intel Corporation, Intel Math Kernel Library Reference Manual, available from: *https://software.intel.com/sites/default/files/managed/9d/ c8/mklman.pdf* [accessed March 10, 2017].

[13] S. Lahabar, P. J. Narayanan, "Singular Value Decomposition on GPU using CUDA", in *Proc. IEEE Intl. Symp. on Parallel and Distributed Processing - IPDPS 2009*, doi 10.1109/IPDPS.2009.5161058, May 23-29, 2009.

[14] C. Manning, P. Raghavan, H. Schutze, *Introduction to Information Retrieval*, 1st edition, Cambridge University Press, 2008.

[15] MathWorks, MATLAB, available from: *http://www.mathworks.com/products/matlab/*, [accessed March, 10, 2017].

[16] Nvidia, Nvidia CUDA Programming Guide, available from: *http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf* [accessed March, 10, 2017], version 8.0, January, 2017.

[17] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, "GPU computing", *Proc. of the IEEE*, vol. 96, no. 5, 2008, pp. 279–299.

[18] P. Pacheco, *An Introduction to Parallel Programming*, Elsevier, 2011.

[19] D. Poole, *Linear Algebra - A Modern Introduction*, 2nd edition, Brooks/Cole, Thomson, 2006.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd edition, Cambridge University Press, 2007.

[21] J. W. Suh, Y. Kim, *Accelerating MATLAB with GPU Computing – A Primer with Examples*, Morgan Kaufmann – Elsevier, 2014.

[22] L. Eldén, *Matrix Methods in Data Mining and Pattern Recognition*, Society for Industrial and Applied Mathematics, Philadelphia, 2007.

[23] A. Thomo, *Latent Semantic Analysis Tutorial*, available from: *http://webhome.cs.uvic.ca/~thomo/svd.pdf* [accessed March 12, 2017].

[24] K. Baker, *Singular Value Decomposition Tutorial*, available from: *http://davetang.org/file/Singular_Value_Decomposition_Tutorial.pdf* [accessed March 12, 2017].