Accelerating Low-Rank Matrix Completion on GPUs

Achal Shah* and Angshul Majumdar†
*Indian Institute of Technology, Guwahati
†Indraprastha Institute of Information Technology, Delhi

***Abstract*—Latent factor models formulate collaborative filtering as a matrix factorization problem. However, matrix factorization is a bi-linear problem with no global convergence guarantees. In recent years, research has shown that the same problem can be recast as a low-rank matrix completion problem. The resulting algorithms, however, are sequential in nature and computationally expensive. In this work we modify and parallelize a well known matrix completion algorithm so that it can be implemented on a GPU. The speed-up is significant and improves as the size of the dataset increases; there is no change in accuracy between the sequential and our proposed parallel implementation.**

***Keywords*—Matrix Completion, Recommendation Systems, Collaborative Filtering, Graphics Processing Units**

I. Introduction

In this work, we revisit the collaborative filtering approach to recommender systems that has, in recent years, enjoyed the attention of the research community. The problem of recovering a (approximately) low-rank matrix from its under-sampled projections is prevalent in many areas of engineering and applied science, including machine learning [1], [3], [5] and computer vision [31]. As an example, let us consider the Netflix problem [2], where the users provide ratings for a subset of the available items, and the objective is to infer their preference for the remaining content. The available ratings matrix, in such a scenario, is very sparse since the users can only be expected to rate a few items each. Recovering the matrix, given this limited sample of its entries, is extremely ill-posed as we may have infinitely many completions.

The latent factor model assumes that there are f underlying factors, contributing to a user's taste or preference, that are responsible for the user's rating on an item. The item is characterized by a vector $v_{f\times 1}$ which encapsulates the extent to which each of the f factors are present in it. The user is characterized by a vector $u_{f\times 1}$ which consists of the user's affinity towards each of the f factors. Thus, the rating of user i on the item j is simply the inner product between the corresponding v and u;

$$x_{i,j} = v_j^T u_i \tag{1}$$

We now consider the matrix of ratings from all users on all items. Let us assume that there are M users and N items. If we stack all the user' affinities (u's) as rows of a matrix $U_{M\times f}$ and all the items' v's as rows of a matrix $V_{M\times f}$, the full ratings matrix is represented as,

$$X_{M\times N} = U_{M\times f} V_{N\times f}^T \tag{2}$$

When the number of users (M) and number of items (N) is much larger than the number of factors (which is typically the case in any practical collaborative filtering problem) the matrix X is low-rank (of rank f). Since all the ratings are not available, X is only partially full. Thus, in collaborative filtering, the problem is to find all the ratings in X knowing that it is low-rank.

Let Ω be the set of indices where the matrix X is observed, i.e. where the ratings are available. Then we can define a sampling operator A on the set Ω. The observation model for collaborative filtering can thus be succinctly represented as,

$$y_{m\times 1} = A(X_{M\times N}),\ A: \mathbb{R}^{M\times N} \to \mathbb{R}^m \tag{3}$$

The problem is to recover X given the observations y and the sampling operator A.

Since the matrix X is known to be low-rank, the straightforward approach to solve the above is to minimize X's rank subject to data constraints.

$$\min_X \mathit{rank}(X) \text{ subject to } y = A(X) \tag{4}$$

Unfortunately solving the above formulation exactly is an NP-hard problem with doubly exponential complexity. Theoretical studies [8], [9] argue that it is possible to recover the correct solution (a low rank matrix) by replacing the NP-hard objective function (rank of a matrix) in (4) by its closest convex envelope - the nuclear norm:

$$\min_X \|X\|_* \text{ subject to } y = A(X);\ \|X\|_* = \sum_i |\sigma_i| \tag{5}$$

where σ_i's are the singular values of X and there are $r(\ll M, N)$ such singular values.

This paper contributes to the line of work concerned with developing algorithmic frameworks to solve (5) for large-scale problems. The amount of available information has increased faster than our ability to process it. It is no longer feasible to rely on improvements in processors to speed-up matrix estimation algorithms, and as a result, we have moved towards a model where we have more processors that can work in parallel. In this paper, we modify a well-known matrix completion algorithm using the principles of parallel stochastic gradient descent, and then accelerate it by harnessing the 'processing parallelism' of modern Graphics Processing Units (GPUs).

The rest of the paper is organized into several sections. The existing literature is briefly reviewed in the following section. The proposed methodology is discussed in section 3. We examine the experimental results in section 4. Finally, we discuss the conclusions of this work in section 5.

II. Related Work

There have been a handful of attempts in the past [7], [18], [21], [22], [30], [25] to address the matrix completion problem, but these techniques are computationally intensive and prove to be impractical as the size of the data approaches the scale commonly used in practice (such as by internet retailers). A few methods, such as [27], try to factorize the ratings matrix into a user latent factor matrix and an item latent factor matrix. These methods rely on an inherent low-rank parameterization, which leads to a bi-linear (and hence non-convex) formulation of the optimization problem. While such problems are computationally more efficient to solve, they lack global convergence guarantees and the solutions, in general, may be highly sensitive to initialization.

One way to scale-up these algorithms to larger datasets, is by leveraging the computational horse-power provided by Graphics Processing Units. Modern GPUs have gained popularity as low-cost platforms for massively parallel computation. The availability of high-level programming languages such as CUDA [26] and OpenCL [19] has lowered the programming barrier for the GPU. As a result, GPUs are now seen as high performance multi-core processors and have found extensive use in general-purpose computation [13]. In this work, we modify the well-known iterative thresholding algorithm, and present a method to parallelize and implement the algorithm on a GPU.

III. Proposed Algorithm

In this section, we develop the algorithmic framework to solve the matrix completion problem for large matrices.

The optimization problem (5) can be approximated as the following relaxed problem:

$$\min_X \|X\|_* \text{ subject to } \|y - A(X)\|_F^2 < \epsilon \tag{6}$$

where $\| * \|_F^2$ denotes the Frobenius norm of a matrix, and ϵ is the tolerable limit of error in reconstruction. We initially consider the unconstrained lagrangian version of the problem, which is easier to solve:

$$\min_X \|y - A(X)\|_F^2 + \lambda\|X\|_* \tag{7}$$

A more convenient representation is:

$$\min_x \|y - Ax\|_F^2 + \lambda\|X\|_* \tag{8}$$

where $x_{MN\times 1}$ is the vectorized form of matrix $X_{M\times N}$, $A : \mathbb{R}^{M\times N} \to \mathbb{R}^m$ is a restriction operator, and y_m is the vector of measurements.

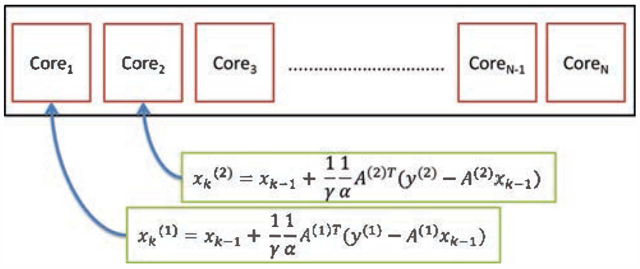
A. Optimization Transfer

The required optimization problem (8) can be solved by minimizing the following at each iteration:

$$G_k(x) = \|x - x_k\|_2^2 + \frac{\lambda}{\alpha}\|X\|_* \tag{9}$$

where

$$x_k = x_{k-1} + \frac{1}{\alpha}A^T(y - Ax_{k-1})$$



Core$_1$
Core$_2$
Core$_3$
Core$_{N-1}$
Core$_N$
$x_k^{(2)} = x_{k-1} + \frac{1}{\gamma}\frac{1}{\alpha}A^{(2)T}(y^{(2)} - A^{(2)}x_{k-1})$
$x_k^{(1)} = x_{k-1} + \frac{1}{\gamma}\frac{1}{\alpha}A^{(1)T}(y^{(1)} - A^{(1)}x_{k-1})$

Fig. 1. Parallelization strategy for the landweber iteration

In the above formulation, x_k is defined in each landweber iteration based on the value of x from the previous iteration; α is the step size and usually is set at max eigenvalue (A^TA).

The equivalence of (8) and (9) is not trivial, and the derivation is based on the majorization-minimization approach [28], [33]. We have refrained from presenting the derivation here due to the limited availability of space within this conference paper. The interested reader is encouraged to go through the relevant references.

Given the definition of the Frobenius norm, and the property that both x and x_k have the same left and right singular vector, minimizing the above equation is the same as minimizing the following:

$$G'_k(s) = \|s - s_k\|_2^2 + \frac{\lambda}{\alpha}\|s\|_1 \tag{10}$$

where s and s_k are the singular values of the matrices corresponding to x and x_k respectively. This can now be decoupled, and minimized using term-wise differentiation, by the following (soft-thresholding):

$$s = \text{signum}(s_k)\max(0, |s_k| - \frac{\lambda}{2\alpha}) \tag{11}$$

This is a modified form of the iterative soft-thresholding algorithm used in compressive sensing for l_1 norm minimization.

B. The Shrinkage Algorithm

We find that the algorithm consists of the following four major steps:

1) Landweber iteration (to solve the partial least squares problem)
2) Computing the singular value decomposition (SVD)
3) Soft-thresholding the singular values
4) Recombination of the thresholded singular values with the left and right singular matrices

We initially focus on parallelizing the landweber iteration, which is inherently sequential in nature. To do so, we make use of a gradient descent optimization method based on stochastic optimization, known as the stochastic gradient descent. A similar approach was used in [29] as well, for parallelizing sparse recovery algorithms.

The landweber iteration is given by:

$$x_k = x_{k-1} + \frac{1}{2\alpha}\,\Delta_x\|y - Ax\|_2^2\big|_{x=x_{k-1}} \tag{12}$$

Such an iteration can be represented as the following gradient descent step:

$$x_k = x_{k-1} + \frac{1}{2\alpha}\,\Delta_x \|y - Ax\|_2^2\big|_{x=x_{k-1}} \tag{13}$$

In stochastic gradient descent [6], [11], [10] the gradient of a function is approximated by its stochastic version. This means that in each iteration of the algorithm, the gradient is computed on a small batch of samples instead of the entire data. An extreme case is when the gradient is computed on a single sample. The relation between the gradient on the entire data, given by g, and the stochastic gradient, given by g_s, can be expressed as follows:

$$g = g_s + e \tag{14}$$

The error between the two gradients is represented as a random variable e. In accordance to the theory of stochastic gradient descent, as shown in [6], [11], [10], we have:

$$E(e) = 0 \tag{15}$$

where $E(e)$ denotes the expected value of the random variable e. It has been further shown that the stochastic gradient converges to the gradient computed on the entire data as determined by the second order moment.

The principles of stochastic gradient descent can be applied to the landweber iteration. Therefore, we can run stochastic landweber iterations in parallel, on parts of the data, instead of calculating the landweber update on the entire data. We can then closely approximate the full landweber update by aggregating the individual stochastic landweber updates (as the error is expected to asymptotically reach zero).

We propose the use of the following two-step process in place of the original iteration:

1) Compute stochastic landweber updates, in parallel, on small samples of the data.
2) Aggregate the stochastic landweber updates to estimate the landweber update on the entire data.

Each stochastic landweber update can computed independently in a separate core. The proposed stochastic version of the iteration is illustrated in Algorithm 1.

Algorithm 1 Parallel Stochastic Landweber Iteration

```
1: $N \leftarrow$ number of processing units available
2: $m \leftarrow$ total number of samples (each row of $y$ is considered
   a sample)
3: $\gamma \leftarrow$ sampling factor ($\frac{1}{N} \leq \gamma \leq 1$)
4: for all $i \in \{1, ..N\}$, in parallel do
5:     $x_k^{(i)} \leftarrow x_{k-1} + \frac{1}{\gamma}\frac{1}{\alpha} A^{(i)T}(y^{(i)} - A^{(i)}x_{k-1})$, where $y^{(i)}$
   denotes the randomly chosen subset of the rows in $y$ and
   $A^{(i)}$ denotes the corresponding rows in $A$
6: end for
7: Aggregate from all the units, $x_k \leftarrow \frac{1}{N}\sum_{i=1}^{N} x_k^{(i)}$
8: return $x_k$
```

The next steps consist of a singular value decomposition followed by the soft-thresholding of the singular values of X. Soft-thresholding is an element-wise operation and is embarrassingly parallel. The computational bottleneck here is the singular value decomposition. The mathematical structure of the SVD makes it a suitable candidate for parallelization on GPUs, and a variety of approaches have been explored in literature [4], [12], [14], [15], [20].

In our implementation, we make use of the commercial GPU linear algebra library, CULA [17], for SVD computation. The CULA library was chosen as it provides stable and reliable GPU algorithms. In addition, it enables greater control on the transfer of data to and from the GPU, by exposing a *device* interface to operate directly on matrices residing within the GPU memory.

The proposed Shrinkage Algorithm, for the unconstrained optimization problem (8), is given in Algorithm 2.

Algorithm 2 Proposed Shrinkage Algorithm

```
1: Initialize $x_0$, and set $k \leftarrow 0$
2: while $k < k_{max}$ do
3:     $k \leftarrow k + 1$
4:     Compute the objective function, $J_{k-1}$:
       $J_{k-1} \leftarrow \|y - Ax_{k-1}\|_2^2 + \lambda\|X_{k-1}\|_*$
5:     $x_k \leftarrow$ parallel stochastic landweber iteration
6:     Reshape $x_k$ to form the matrix $X_k$
7:     GPU Accelerated SVD: $X_k = U\Sigma V^T$
8:     Soft-threshold: $\hat{\Sigma} \leftarrow soft(diag(\Sigma), \frac{\lambda}{2\alpha})$
9:     Reconstruct $X_k$: $X_k \leftarrow U\hat{\Sigma}V^T$
10:    Form $x_k$ by vectorizing $X_k$
11:    Compute the objective function, $J_k$:
       $J_k \leftarrow \|y - Ax_k\|_2^2 + \lambda\|X_k\|_*$
12:    if $(J_{k-1} - J_k)/(J_{k-1} + J_k) < Tol$ then
13:        break
14:    end if
15: end while
16: return $X_k$
```

C. Constrained Optimization via Cooling

We have, so far, discussed a solution for the unconstrained optimization problem (7). Our objective remains, however, to solve the constrained problem (6).

The parameters λ and ϵ are related, but for most problems, this relation is not analytic and is difficult to find without loss of generalization; it is not possible to obtain the parameter λ if ε is available. To address this issue, a cooling strategy is usually employed [23], [24]. We start with a high value of λ and solve the quadratic programming problem (QP) for the given value. In the next iteration we decrease the value of λ and solve the QP once more. The complete constrained optimization algorithm, along with the cooling technique, is illustrated in Algorithm 3.

The above mentioned algorithm makes use of the following two loops: The first loop is present within the shrinkage algorithm and minimizes (8) for a given value of λ. This loop terminates when the relative change in the objective function, $(J_{k-1} - J_k)/(J_{k-1} + J_k)$, is less than the allowed tolerance, or after running for a fixed number of iterations. The second is the outer loop that decreases the value of λ and exits when the error in reconstruction falls below the tolerable limit of error, $\|y - Ax\|_2 \leq \epsilon$, or if λ attains its minimum value.

Algorithm 3 Contrained Optimization via Cooling

```
1: Initialize $x \leftarrow 0$, $\lambda < \max(A^T y)$
2: Choose $DecFac$, the decrease factor for cooling $\lambda$
3: Transfer the sampled entries $y$, the initial $x$ and other
   required parameters to the GPU
4: while $\|y - Ax\|_2 > \epsilon$ do
5:     Obtain $X$ by using the proposed shrinkage algorithm
   for the current value of $\lambda$
6:     $\lambda \leftarrow \lambda * DecFac$
7:     if $\lambda < \lambda_{min}$ then
8:         break
9:     end if
10: end while
11: Transfer the recovered matrix $X$ back to the CPU
```

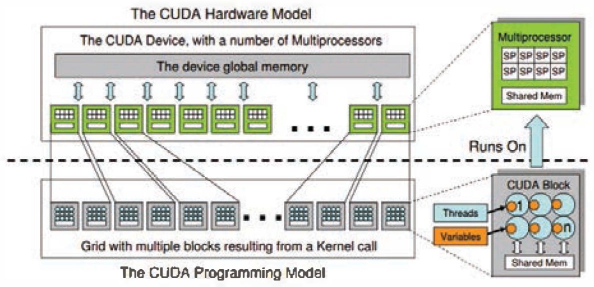


Fig. 2. The CUDA hardware model and programming model [16]

D. CUDA Implementation

We briefly discuss the CUDA programming and hardware models, and then describe the implementation of our proposed approach.

The CUDA Architecture: In the CUDA programming model, a software CUDA block is mapped to a hardware CUDA multiprocessor, as shown in Figure 2. When multiple blocks are assigned to a multiprocessor, the CUDA programming environment determines how the blocks are shared within the multiprocessor. Each multiprocessor consists of a series of processing units along with a small shared memory.

CUDA is based on the Single Instruction Multiple Data (SIMD) model, which means that the same instruction is executed by the processor on different data at a time. The CUDA platform is accessible to developers through CUDA C/C++ APIs. The CUDA device acts as a multi-core processor, where threads run in parallel in batches of warp size.

If a group of blocks is assigned to a single multiprocessor, the available shared memory and registers are split equally amongst the blocks. A block consists of a group of threads, that run in parallel when the block is executed. Each thread executes a single instruction set called the kernel. Each block is assigned an ID, and each thread is assigned a unique ID within the block. These IDs can be accessed from within the kernel function, and are used together to operate on a part of the data.

The interested reader is encouraged to refer to [26] for more details about CUDA programming.

TABLE I. MovieLens 100K: 1682×943

Matrix	Proposed		Sequential		Speedup
	Time(s)	NMAE	Time(s)	NMAE	
M1	719.94	0.70138	1632.6	0.70142	2.2676
M2	722.17	0.70196	1630.2	0.70197	2.2573
M3	698.65	0.69742	1618.8	0.69743	2.3170
M4	716.62	0.69863	1633.7	0.69868	2.2797
M5	705.04	0.69849	1604.0	0.69854	2.2750

Implementation Details: We make use of the CUDA-based CUBLAS library to implement linear algebra routines, and the (dense) free version of the commercial GPU-based linear algebra toolkit, CULA [17], for the SVD computation.

The performance of GPU libraries, and BLAS in general, is highly dependent on data placement and its movement. The bandwidth between the CPU and the GPU is usually much lower than the internal bandwidth of the GPU device, and it therefore important that data transfers to and from the GPU device be minimized. We transfer the input data (the sampled entries, the initialized matrix and other parameters) to the GPU at the very beginning of the algorithm. The data is then kept in the GPU memory and operated upon directly using the available *device* interfaces, to avoid the cost of transferring data between the host and the GPU device in each iteration. Upon convergence of the algorithm, the recovered matrix is transferred back to the host. The input matrix $X_{M \times N}$ is stored on the GPU as a 1-D array of size $M \times N$, in a column-major format. This eliminates the need to reshape the vector x_k to matrix X_k (as in step 6 of the shrinkage algorithm) and to vectorize the matrix X_k (as in step 10) in each iteration.

The parallel stochastic landweber iteration is implemented using a custom kernel. Each element of $x_k^{(i)}$ is updated by a single thread. The aggregated x_k is then obtained by a custom aggregation kernel.

Following the SVD computation using the CULA library, a 1-D vector of singular values, Σ, is obtained. Since soft-thresholding is an element-wise operation, it is aptly suited to the massively multi-threaded CUDA SIMD architecture. The soft-threshold operation on the singular values is implemented using a custom kernel, where each singular value is acted upon by a single thread.

The subsequent reconstruction of the matrix X_k comprises of two linear algebra routines, implemented using the CUBLAS library. The first routine scales the columns of the matrix U with the corresponding, thresholded singular values. The second routine multiplies the scaled matrix $\hat{U}$ with V^T. These routines accelerate the reconstruction by leveraging the speed-ups in matrix-vector products offered by the GPU.

IV. Experimental Evaluation

To evaluate the performance of the proposed algorithm, we used the freely available MovieLens datasets [32]. The MovieLens datasets consist of user-provided ratings for movies, and were collected by the GroupLens Research Project at the University of Minnesota.

For our experiments, we used the MovieLens 100K and the MovieLens 1M datasets. The smaller dataset (100K) consists

TABLE II. MovieLens 1M: 6040×3952

Matrix	Proposed		Sequential		Speedup
	Time(s)	NMAE	Time(s)	NMAE	
M1	11391	0.70914	99287	0.70919	8.7162
M2	11218	0.70901	97878	0.70906	8.7250
M3	11201	0.70877	96326	0.70882	8.5997
M4	11460	0.70925	92997	0.70935	8.1149
M5	11272	0.70909	94673	0.70916	8.3989

of 100,000 ratings from 943 users on 1682 movies, while the latter (1M) contains approximately 1,000,000 anonymous ratings of 3,952 movies made by 6,040 users. The ratings are on a 5-star scale (whole-star ratings only), and each user has rated atleast 20 movies. For each dataset, we have 5 pairs of training and test matrices obtained by splitting the data in a ratio of 4:1. Each of these pairs have disjoint test sets; this is to allow 5 fold cross validation.

The Normalized Mean Absolute Error (NMAE) between ratings and predictions is a widely used metric to evaluate the statistical accuracy of a recommender system. NMAE measures the deviation of the predictions from their true user-specified values. For each pair $< p_i, q_i >$ of ratings and predictions, the metric treats the absolute error $|p_i - q_i|$ between them, equally. Formally, it is defined as

$$NMAE = \frac{\sum_{i=1}^{N} |p_i - q_i|}{N R_{max}} \tag{16}$$

where R_{max} is the maximum possible rating.

We report both the obtained speed-ups, as well as the NMAE, for our proposed approach on the two test sets.

We used MATLAB for the implementation of the algorithms, and ran the experiments on AMD Phenom II X3 710 2.6 GHz Triple-Core Processors. The proposed algorithm was parallelized on an NVIDIA GeForce GTX 680 graphics processor with 1536 CUDA cores and a standard memory of 2048 MB, using CUDA C/C++ APIs.

The results for the MovieLens 100K and the MovieLens 1M datasets have been illustrated in Tables I and II respectively. The proposed algorithm was able to recover the smaller matrices almost 2.28 times faster, at an average, than it's sequential counterpart, whereas an average speed-up of nearly 8.51 times was observed for the larger matrices. Moreover, the observed NMAE values are extremely close for the proposed parallel and existing sequential versions, which suggests that the proposed algorithm is able to estimate the ratings matrix without a deterioration in the reconstruction accuracy. This validates our expectation. An important observation is that the speed-ups increase with an increase in the size of the matrix. This can be attributed to the significant performance benefits provided by the GPU for computations involving matrix and vector operations.

These results suggest that the proposed approach is superior to the existing, sequential algorithm.

V. Conclusion

In collaborative filtering, latent semantic analysis factorize the ratings matrix into a user latent factor matrix and an item latent factor matrix. Such a matrix factorization problem, although efficient to solve is not an optimal approach since it is a bi-linear (and hence non-convex) problem with no global convergence guarantees. Nevertheless, traditional latent semantic analysis estimates these two matrices and finally computes the ratings matrix as a product of these two.

We do not actually need to estimate these user and item latent factor matrices; our goal is to estimate the final ratings matrix. The ratings matrix is low-rank; it has the same rank as the number of latent factors which is much less than the number of users or items. Thus finding the ratings can be recast as a low-rank matrix completion problem. All algorithms, that solve the matrix completion problem are computationally expensive; that is why they are not popular for practical collaborative filtering problems. In this work, we have taken a well known matrix completion algorithm and showed how it can be implemented on a GPU.

Any matrix completion algorithm consists of four major steps - solving a partial least squares problem, computing singular value decomposition (SVD), thresholding the singular values and recombination of the thresholded singular values with the left and right singular matrices to update the estimate of the low-rank matrix. The last two steps are inherently parallelizable on a GPU. There are also efficient algorithms to implement SVD on a GPU. We used the CULA library [17] for the same. For solving the partial least squares problem, we employed a stochastic technique that can be efficiently implemented on a GPU.

We have compared our proposed parallelized version with the sequential algorithm. We find that while there is no deterioration in recovery accuracy, our method yields significant improvements in speed, especially when the data size increases.

References

[1] Abernethy, J., Bach, F., Evgeniou, T. and Vert, J. P. "Low-rank matrix factorization with attributes". Technical Report N24/06/MM, Ecole des Mines de Paris, 2006.

[2] ACM SIGKDD and Netflix. Proceedings of KDD Cup and Workshop, 2007.

[3] Amit, Y., Fink, M., Srebro, N. and Ullman, S. "Uncovering shared structures in multiclass classification". International Conference on Machine Learning, 2007.

[4] Arbenz, P. and Golub, G. "On the spectral decomposition of Hermitian matrices modified by row rank perturbations with applications". SIAM Journal on Matrix Analysis and Applications, 1988.

[5] Argyriou, A., Evgeniou, T. and Pontil, M. "Multi-task feature learning". Neural Information Processing Systems, 2007.

[6] Bertsekas, D. P. and Tsitsiklis, J. N. "Gradient convergence in gradient methods with errors". SIAM Journal on Optimization, Vol. 10 , pp. 627-642, 2000.

[7] Cai, J. F., Candes, E. J. and Shen, Z. "A singular value thresholding algorithm for matrix completion". SIAM Journal on Optimization, 20(4):1956-1982, 2008.

[8] Candes, E. J., and Recht, B. "Exact matrix completion via convex optimization". Foundations of Computational Mathematics., 9, (2008) 717-772.

[9] Candes, E. J., and Tao, T. "The power of convex relaxation: Near-optimal matrix completion". IEEE Transactions on Information Theory, 56 (5), (2009), 2053-2080.

[10] Friedlander, M. P. and Goh, G. "Tail bounds for stochastic approximation". arXiv:1304.5586, 2013.

[11] Friedlander, M. P. and Schmidt, M. “Hybrid deterministic-stochastic methods for data fitting”. SIAM Journal on Scientific Computing, Vol. 34, 2012.

[12] Golub, G. H. “Some modified matrix eigenvalue problems”. SIAM Review, 1973.

[13] GPGPU. “General purpose computation on Graphics Processing Units”. URL http://www.gpgpu.org.

[14] Gu, M. and Eisenstat, S. “A stable algorithm for the rank-1 modification of the symmetric eigenproblem”. Report YALEU/DCS/RR-916, Yale University, 1992.

[15] Gu, M. and Eisenstat, S. “A divide-and-conquer algorithm for the bidiagonal SVD”. Report YALEU/DCS/RR-933, Yale University, 1992.

[16] Harish, P., Vineet, V., and Narayanan, P. J. (2009). “Large graph algorithms for massively multithreaded architectures”. Centre for Visual Information Technology, I. Institute of Information Technology, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74.

[17] Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J. “CULA: hybrid gpu accelerated linear algebra routines”. Proceedings of SPIE. p. 7705, 2010

[18] Ji, S. and Ye, J. “An accelerated gradient method for trace norm minimization”. International Conference on Machine Learning, 2009.

[19] Khronos OpenCL Working Group. “The OpenCL Specification”. Version 1.0.29, 8 December 2008.

[20] Lahabar, S. and Narayanan, P. J. “Singular Value Decomposition on GPU using CUDA”. IPDPS, 2009.

[21] Liu, Z. and Vandenberghe, L. “Interior-point method for nuclear norm approximation with application to system identification”. SIAM Journal on Matrix Analysis and Applications, 31(3):1235-1256, 2009.

[22] Ma, S., Goldfarb, D. and Chen, L. “Fixed point and Bregman iterative methods for matrix rank minimization”. Mathematical Programming, pages 1-33, 2009.

[23] Majumdar, A. and Ward, R. K. “Synthesis and Analysis Prior Algorithms for Joint-Sparse Recovery”. IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 3421-3424, 2012.

[24] Majumdar, A. and Ward, R. K. “On the Choice of Compressed Sensing Priors: An Experimental Study”. Signal Processing: Image Communication, Vol. 27 (9), pp. 1035-1048, 2012.

[25] Mazumder, R., Hastie, T., and Tibshirani, R. “Spectral Regularization Algorithms for Learning Large Incomplete Matrices”. Journal of Machine Learning Research, 99:2287-2322, 2010

[26] NVIDIA. “NVIDIA CUDA Programming Guide”. NVIDIA Corporation, 2013.

[27] Recht, B. and Re, C. “Parallel Stochastic Gradient Algorithms for Large-Scale Matrix Completion”. Optimization Online, 2011.

[28] Selesnick, I. W. and Figueiredo, M. A. T. “Signal restoration with overcomplete wavelet transforms: comparison of analysis and synthesis priors”. Proceedings of SPIE, Vol. 7446 (Wavelets XIII), 2009.

[29] Shah, A. and Majumdar, A. “Parallelizing Sparse Recovery Algorithms: A Stochastic Approach”. International Conference on Digital Signal Processing, 2014.

[30] Toh, K. C. and Yun, S. “An accelerated proximal gradient algorithm for nuclear norm regularized least squares problems”. Pacific Journal of Mathematics, 6:615-640, 2010.

[31] Tomasi, C. and Kanade, T. “Shape and motion from image streams under orthography: a factorization method”. International Journal of Computer Vision, 9(2):137-154, 1992.

[32] http://grouplens.org/datasets/movielens/

[33] http://cnx.org/content/m32168/latest/