Bringing High Performance Computing to Big Data Algorithms

H. Anzt, J. Dongarra, M. Gates, J. Kurzak, P. Luszczek, S. Tomov and I. Yamazaki

Abstract Many ideas of High Performance Computing are applicable to Big Data problems. The more so now, that hybrid, GPU computing gains traction in mainstream computing applications. This work discusses the differences between the High Performance Computing software stack and the Big Data software stack and then focuses on two popular computing workloads, the Alternating Least Squares algorithm and the Singular Value Decomposition, and shows how their performance can be maximized using hybrid computing techniques.

1 Introduction

1.1 High Performance Computing Meets Big Data

High Performance Computing (HPC), meaning scientific and engineering computing, with emphasis on simulation, offers decades of experience in crunching numbers

H. Anzt · J. Dongarra · M. Gates · J. Kurzak (✉) · P. Luszczek · S. Tomov · I. Yamazaki
Innovative Computing Laboratory, University of Tennessee,
1122 Volunteer Blvd, Knoxville, TN 37996, USA
e-mail: kurzak@icl.utk.edu

H. Anzt
e-mail: hanzt@icl.utk.edu

J. Dongarra
e-mail: dongarra@icl.utk.edu

M. Gates
e-mail: mgates3@icl.utk.edu

P. Luszczek
e-mail: luszczek@icl.utk.edu

S. Tomov
e-mail: tomov@icl.utk.edu

I. Yamazaki
e-mail: iyamazak@icl.utk.edu

A.Y. Zomaya and S. Sakr (eds.), *Handbook of Big Data Technologies*,
DOI 10.1007/978-3-319-49340-4_23

at the highest speeds, using machines form the high end of the hardware spectrum. Big Data, meaning data analytics, has been shifted more toward the lower end of that spectrum, where the price/performance ratio is more favorable. Now that Big Data problems enter the mainstream of computing, many solutions from HPC can be applied to Big Data.

This chapter opens with a discussion of the main differences between the hardware/software stacks of Big Data and HPC. Then two prominent HPC workloads are introduced, which happen to be in widespread use in the Big Data domain, the Alternating Least Squares (ALS) algorithm and the Singular Value Decomposition (SVD). Then the main techniques for maximizing the performance of the implementations are discussed. A comprehensive discussion of the implementation details of the ALS algorithm follows. Then a thorough presentation of the implementation details of the SVD algorithm is given. The chapter is concluded with the summary of the most important points.

High Performance Computing: In the 1980s, vector supercomputing dominated high-performance computing, as embodied in the eponymously named systems designed by the late Seymour Cray. The 1990s saw the rise of massively parallel processing (MPPs) and shared memory multiprocessors (SMPs) built by Thinking Machines, Silicon Graphics, and others. In turn, clusters of commodity (Intel/AMD x86) and purpose-built processors (such as IBM's BlueGene), dominated the previous decade.

Today, these clusters are augmented with computational accelerators in the form of coprocessors from Intel and graphical processing units (GPUs) from NVIDIA; they also include high-speed, low-latency interconnects (such as InfiniBand). Storage area networks (SANs) are used for persistent data storage, with local disks on each node used only for temporary files. This hardware ecosystem is optimized for performance first, rather than for minimal cost.

Atop the cluster hardware, Linux provides system services, augmented with parallel file systems (such as Lustre) and batch schedulers (such as PBS and SLURM) for parallel job management. MPI and OpenMP are used for internode and intranode parallelism, augmented with libraries and tools (such as CUDA and OpenCL) for coprocessor use. Numerical libraries (such as LAPACK and PETSc) and domain-specific libraries complete the software stack. Applications are typically developed in Fortran, C, or C++. Figure 1 (right) shows the mainstream HPC system stack.

Big Data: Just a few years ago, the very largest data storage systems contained only a few terabytes of secondary disk storage, backed by automated tape libraries. Today, commercial and research cloud-computing systems each contain many petabytes of secondary storage, and individual research laboratories routinely process terabytes of data produced by their own scientific instruments.

As with high-performance computing, a rich ecosystem of hardware and software has emerged for big data analytics. Unlike scientific computing clusters, data-analytics clusters are typically based on commodity Ethernet networks and local storage, with cost and capacity the primary optimization criteria. However, industry is now turning to FPGAs and improved network designs to optimize performance.

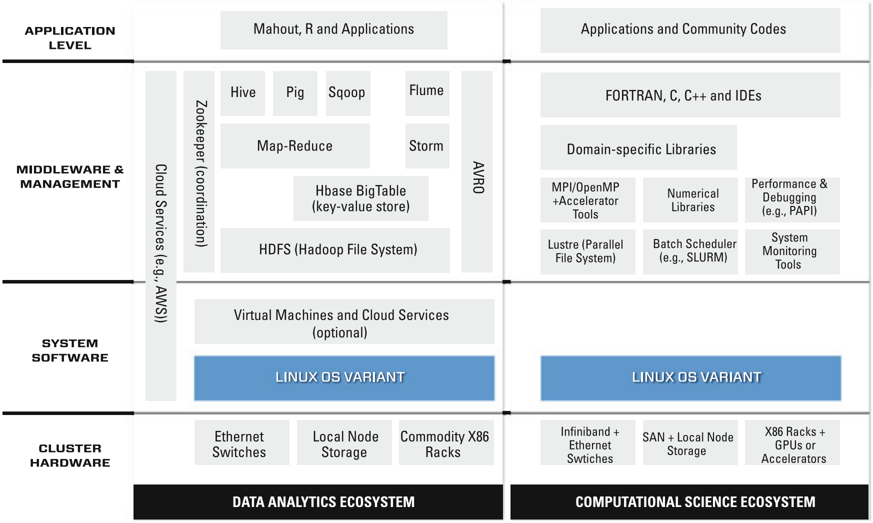

APPLICATION LEVEL
MIDDLEWARE & MANAGEMENT
SYSTEM SOFTWARE
CLUSTER HARDWARE
Mahout, R and Applications
Cloud Services (e.g., AWS))
Zookeeper (coordination)
Hive
Pig
Sqoop
Flume
Map-Reduce
Storm
Hbase BigTable (key-value store)
HDFS (Hadoop File System)
AVRO
Virtual Machines and Cloud Services (optional)
LINUX OS VARIANT
Ethernet Switches
Local Node Storage
Commodity X86 Racks
DATA ANALYTICS ECOSYSTEM
Applications and Community Codes
FORTRAN, C, C++ and IDEs
Domain-specific Libraries
MPI/OpenMP +Accelerator Tools
Numerical Libraries
Performance & Debugging (e.g., PAPI)
Lustre (Parallel File System)
Batch Scheduler (e.g., SLURM)
System Monitoring Tools
LINUX OS VARIANT
Infiniband + Ethernet Swtiches
SAN + Local Node Storage
X86 Racks + GPUs or Accelerators
COMPUTATIONAL SCIENCE ECOSYSTEM

Fig. 1 The mainstream big data stack (*left*) versus the mainstream HPC stack (*right*)

Atop this hardware, the Apache Hadoop system implements a MapReduce model for data analytics. Hadoop includes a distributed file system (HDFS) for managing large numbers of large files, distributed (with block replication) across the local storage of the cluster. HDFS and HBase, an open source implementation of Google's BigTable key-value store, are the big data analogs of Lustre for computational science, albeit optimized for different hardware and access patterns.

Atop the Hadoop storage system, tools (such as Pig) provide a high-level programming model for the two-phase MapReduce model. Coupled with streaming data (Storm and Flume), graph (Giraph), and relational data (Sqoop) support, the Hadoop ecosystem is designed for data analysis. Moreover, tools (such as Mahout) enable classification, recommendation, and prediction via supervised and unsupervised learning. Unlike scientific computing, application development for data analytics often relies on Java and Web services tools (such as Ruby on Rails). Figure 1 (left) shows the mainstream Big Data system stack.

1.2 Application Areas

This chapter discusses HPC implementations of two mainstream Big Data algorithms. While the first one, Alternating Least Squares (ALS), has primarily commercial applications, the second one, Singular Value Decomposition (SVD), is uniformly applicable to a wide range of problems in science, engineering, and commerce.

The **Alternating Least Squares** algorithm provides a classic solution for building a recommender system for e-commerce, and was one of the more successful approaches to the Netflix Prize challenge. The importance of the algorithm is in its ability to deal with systems with implicit feedback, when the user's preference towards some products or content is known, while it is unknown for others. The weighted regularization process employed in the ALS algorithm allows for attaching higher weights to the known values and lower weight to the unknown values, therefore effectively reconstructing the unknown values, as opposed to treating them as lack of interest. This approach leads to much more accurate recommendations than the simpler similarity-based algorithms.

One of the first open source implementations of the ALS algorithms was produced in Java, as part of the Mahout machine learning package [48], which relied on the MapReduce paradigm provided by the Hadoop framework [34, 60]. As the MapReduce approach is being ousted by the Resilient Distributed Datasets (RDD) of the Spark framework [67], a faster implementation showed up in the Spark MLlib library [44]. Also, ALS was one of the first algorithms implemented in the GraphLab package [37], and also, for some time now, has been available in the Data Analytics Acceleration Library (DAAL) from Intel [26]. Finally, the first state of the art GPU implementation was produced by the authors of this chapter [16], and followed by similar developments from other groups [57].

The **Singular Value Decomposition** is ubiquitous in statistics and scientific computing and commonly applied to problems where the matrices are large and substantial computational power is required. Prime examples of application areas include astrophysics, genomics, climate data analysis, and information retrieval systems. In astrophysics, the SVD is used on massive datasets from astronomical surveys for spectral classification, e.g., to predict morphological types using galaxy spectra, and to select quasar candidates from sky surveys. In genomics, the SVD is routinely used to analyze genome-wide single-nucleotide polymorphism (SNP) data, for detecting population structure and potential outliers. In climate data analysis, Empirical orthogonal function (EOF) and the SVD are the methods of choice for analyzing spacial and temporal variability of geophysical data. The SVD is also the primary tools for latent semantic indexing (LSI) in information retrieval systems, where it is used to find low-rank approximations to term-document matrices, enabling computation of query-document similarity scores in low-rank representation, as well as automated document categorization.

Randomized algorithms have been developed for the singular value decomposition [36, 42]. Great surveys of recent developments in randomization algorithms were published by Halko [21] and Mahoney [41]. In terms of software, singular value solvers are available in Skylark and Mahout. Skylark is an open-source software project launched by IBM Research with the objective to develop a set of randomized machine learning algorithms that support distributed memory and are accessible through Python interfaces. Skylark uses a number of sketching transforms to implement a few randomized linear algebra solvers, including a singular value solver based on the work by Halko et al. [21]. Mahout is a project of the Apache Software Foundation to produce free implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collaborative filtering, clustering, and classification [48]. In addition to a classic Lanczos SVD algorithm, Mahout also contains an implementation of a stochastic (randomized) SVD routine [40].

1.3 Tricks of the Trade

Two techniques discussed here and borrowed from the field of High Performance Computing, are automated software tuning and randomization algorithms. The technique of automated software tuning mostly addressed the challenges of programming modern computing devices, such as GPU accelerators, in a way that provides portable performance, i.e., not only allows getting maximum performance from a particular device, but also allows for porting to a new device by retuning rather than rewriting/redesigning the code. The technique of randomization allows dealing with one of the most burning problems of processing Big Data, which is the lagging of IO capabilities behind processing capabilities in modern hardware.

Automated Software Tuning: Although Moore's Law has still been in effect in the last few years, the multicore revolution initiated the trend, in processor design, of going away from architectural features that do not directly contribute to processing throughput. This means preference towards shallow pipelines with in-order execution and cutting down on branch prediction and speculative execution. On top of that, virtually all modern architectures require some form of vectorization to achieve top performance, whether it being short-vector SIMD (Single Instruction Multiple Data) extensions of CPU cores, or SIMT (Single Instruction Multiple Thread) pipelines of GPU accelerators. With the landscape of future High Performance Computing populated with complex, hybrid, vector architectures, automated software tuning may provide a path towards portable performance without heroic programming efforts.

Automated software tuning was pioneered by projects like ATLAS and Spiral, and is the objective of numerous academic projects, and is also practiced by hardware vendors providing libraries like BLAS for their devices. The basic premise is to explore a search space and find the best performers. The search space can be defined by a set of tunable parameters, code transformations, implementation variants, hardware switches, etc. It can then be pruned by applying a set of constraints that eliminate obvious underperformers. Finally, it can be searched to find the winners. Exhaustive search, steepest descent methods, genetic algorithms are all valid approaches.

Randomization Algorithms: The landscape of future High Performance Computing presents an explosive growth in the volume of data, and a relatively dismal growth in the capabilities of communication and IO systems. Under such conditions, it becomes increasingly important to find algorithms that communicate less, and perform IO operations even less. For an important set of problems in numerical computing, a class algorithms emerges that seem to be an answer to these challenges—randomization algorithms.

The new classes of random sampling and random projection algorithms offer numerous advantages when dealing with large datasets coming from both scientific applications (astrophysics, genomics, climate modeling), as well as commercial applications (social networks, information retrieval systems, financial transactions). In many cases, randomized algorithms beat their classical counterparts in terms of accuracy, speed, and robustness. They utilize modern computer architectures better by exposing higher levels of parallelism than traditional numerical methods. At the same time, they often produce more numerically robust solvers by introducing implicit regularization.

2 GPU Acceleration of Alternating Least Squares

Web-based services such as movie databases and online retailers increasingly rely on recommendation systems to suggest products to their customers. Collaborative Filtering (CF) is a class of recommendation systems that recommends products based on what other customers with similar interests have enjoyed [17]. It harvests information collected from a large set of users, which can be either explicit feedback, such as "likes" or product ratings; or implicit feedback, such as purchases, time spent, or search patterns. This yields a large dataset to process, for instance, the Netflix Prize dataset has over 100 million ratings [4].

Collaborative Filtering algorithms are based on observation data in a relation matrix R, where each entry denotes how a user rated or interacted with an item. As each user rates only a small subset of the items, most entries are unknown, i.e., the matrix R is sparse. The goal is to determine the unknown values in R for how a user would hypothetically rate every item. Thus it is an instance of the *matrix completion problem* [9], to determine the unknown entries of a sparsely sampled matrix. In recent years, latent feature models have assumed a small set of features—such as movie genres—drive users' interest. However, these latent features are determined by the algorithm, without any explicit, a priori assigned meaning. This small set of features implies the matrix R is (approximately) low-rank.

Besides providing an algorithm to complete R, an added benefit of the low-rank model is that it determines R in a compact representation, $R = X^T Y$, taking $O(fm + fn)$ space instead of $O(mn)$ space for m users, n items, and rank $f \ll m, n$. For a site with millions of users and millions of products, this compact representation makes storing and accessing the recommendations database tractable.

In addition to recommendation systems, the matrix completion problem occurs in numerous other contexts. Examples include recovery of missing pixels of an image [27], inferring 3D structure from motion of images [10], and determining sensor positions from incomplete distance measurements [8].

Various methods exist for computing the matrix completion. Many CF systems used neighborhood models [30]. For low-rank models, Candès and Recht [9] used convex relaxation, and proved that R can be completed if sufficient entries are known. Stochastic gradient descent [8, 50] and alternating least squares (ALS) [27, 70] are popular methods. We will focus on the ALS method, which has adaptations for both explicit [70] and implicit feedback [23].

We propose both multi-core CPU and GPU implementations that are able to exploit the computing power of state-of-the-art processors and accelerators. We compare performance with the open source implementations available in Mahout [1], GraphLab [12], and Spark MLlib [2, 44, 67], and report significant speedups for selected benchmark datasets.

2.1 Explicit Feedback

For explicit feedback, entry r_{ui} of R denotes how user u rated item i. Since users have not rated all items, the goal is to complete the missing entries of R. We assume that R is approximately low-rank, such that $R \approx X^T Y$, where X is $f \times m$ and Y is $f \times n$ for m users, n items, and rank or feature space size f. This latent feature space is small compared to the number of users and items, e.g., from 10 to 100, depending on the application. Column x_u of X represents user u, and column y_i of Y represents item i, such that their inner product yields the rating, $r_{ui} \approx x_u^T y_i$.

Determining X and Y is commonly expressed as an optimization problem, with a summation over known r_{ui} entries,

$$\min_{X,Y} \sum_{\substack{u,i \\ r_{ui} \text{ is known}}} \left(r_{ui} - x_u^T y_i\right)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right). \tag{1}$$

Here, λ is a regularization term to avoid overfitting. This can be solved with stochastic gradient descent or alternating least squares.

To solve using ALS, we observe that if X or Y is fixed, the cost function (1) becomes a linear least squares problem. ALS iterates two steps: fixing Y and solving for X, then fixing X and solving for Y. In the first step, fixing Y and finding where the gradient is zero yields

$$\left(Y D^u Y^T + \lambda I\right) x_u = Y r_u \qquad \text{for } u = 1, \ldots, m$$

to solve for each user-factor x_u. Each of the m user-factors can be solved independently, providing a large amount of parallelism. Here, r_u is row u of the R matrix, and D^u is a binary diagonal matrix that selects columns of Y corresponding to known r_{ui} values. Similarly, in the second step, fixing X yields

$$\left(X D^i X^T + \lambda I\right) y_i = X r_i \qquad \text{for } i = 1, \ldots, n$$

to solve for each item-factor y_i, where r_i is column i of the R matrix, and D^i selects columns of X for known r_{ui} values. Experiments have shown that the user- and item-factors typically converge after a few iterations of these two steps [70].

2.2 Implicit Feedback

For implicit feedback, Hu et al. [23] note that a large r_{ui} value does not necessarily indicate a higher preference, but instead gives a higher confidence. For instance, a user may enjoy watching a moderately good TV show every week, yielding a large r_{ui} value, but watch a beloved movie just once or twice, yielding a small r_{ui} value, despite its stronger preference. Therefore, they propose a preference matrix P with binary values,

$$p_{ui} = \begin{cases} 1 & \text{if } r_{ui} > 0, \\ 0 & \text{if } r_{ui} = 0, \end{cases}$$

to indicate whether user u has a preference for item i. Larger r_{ui} values indicate greater confidence in this preference, so a matrix C with entries $c_{ui} = 1 + \alpha r_{ui}$ is introduced that measures the confidence of the preference p_{ui}. Here some minimal confidence is given even to zero entries, while α weights known values more. Hu et al. found $\alpha = 40$ to work well. For implicit feedback, instead of completing the relation matrix R, the goal is to complete the preference matrix as $P \approx X^T Y$. Again, X and Y can be computed by minimizing a cost function,

$$\min_{X,Y} \sum_{u,i} c_{ui} \left(p_{ui} - x_u^T y_i\right)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right). \tag{2}$$

The major difference compared to explicit feedback is that the sum is over all u and i, not just those with nonzero r_{ui} values, since some minimal confidence is given even to zero entries. This means there are mn terms, making stochastic gradient descent prohibitively expensive for implicit feedback, whereas for explicit feedback only the nonzero r_{ui} values have terms in (1). Therefore, we apply the alternating least squares algorithm, similar to the explicit feedback case above, yielding

$$\begin{aligned} \left(YC^uY^T + \lambda I\right) x_u &= YC^u p_u && \text{for } u = 1, \ldots, m; \\ \left(XC^iX^T + \lambda I\right) y_i &= XC^i p_i && \text{for } i = 1, \ldots, n; \end{aligned}$$

to solve for each x_u and for each y_i, where C^u is a diagonal matrix of row u of the confidence matrix C, C^i is a diagonal matrix of column i of C, p_u is row u of the preference matrix P, and p_i is column i of P. Pseudocode is given in Algorithm 1.

Algorithm 1 Pseudocode of alternating least square algorithm iterating user-factors and item-factors.

```
function ALS( input: $\alpha$, $\lambda$, $R$; output: $X$, $Y$ )
    set $Y$ to random initial guess
    while not converged
        // update user-factors $X$
        for $u = 1, \ldots, m$
            solve $\left(YC^uY^T + \lambda I\right) x_u = YC^u p_u$ for $x_u$
        end
        // update item-factors $Y$
        for $i = 1, \ldots, n$
            solve $\left(XC^iX^T + \lambda I\right) y_i = XC^i p_i$ for $y_i$
        end
    end
end function
```

The two steps, updating the user-factors and the item-factors, are identical except for swapping the input and output matrices. Therefore, we will subsequently focus on updating the user-factors, and the item-factors will follow similarly. The explicit and implicit feedback ALS algorithms are also very similar; we will concentrate on implicit feedback.

For computational efficiency, the product can be factored as

$$YC^uY^T = YY^T + \alpha YR^uY^T,$$

where R^u is a diagonal matrix of row u of R, as shown schematically in Fig. 2. Since YY^T is the same for all users, it can be computed once per iteration [23], which is done efficiently using the syrk (symmetric rank-k update) BLAS routine. (Explicit feedback lacks the YY^T term.) The remaining term, αYR^uY^T, involves

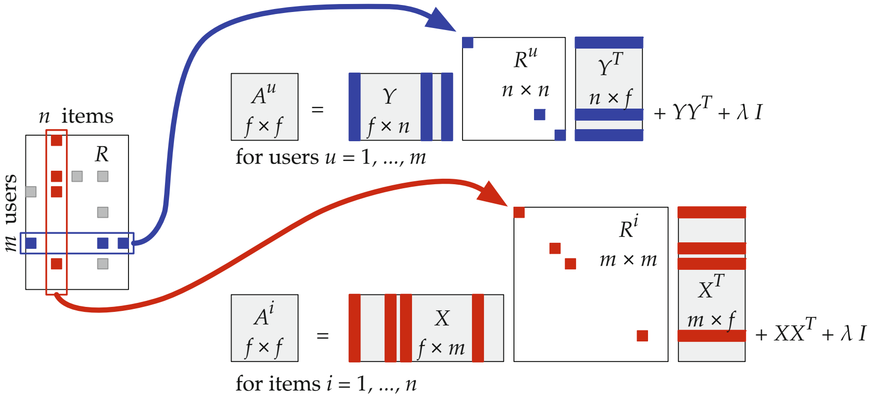


Fig. 2 Diagram of computation of user-factors and item-factors. R is general sparse, R^u and R^i are sparse diagonal, X, Y, A^u, A^i are dense

a dense matrix Y and the sparse diagonal matrix R^u, which will require a custom kernel. Under mild assumptions, $YC^uY^T + \lambda I$ is symmetric positive definite (SPD), allowing us to solve it with the Cholesky factorization.

2.3 CPU Implementation

In the product YR^uY^T, the sparse diagonal matrix R^u selects and scales a few columns of Y, as shown in Fig. 3. Columns of Y corresponding to zeros in R^u are ignored. As k, the number of nonzeros in R^u, is typically much less than n, the number of columns of Y, the kernel should take advantage of this sparsity, reducing the cost from a rank-n update to a rank-k update, with $k \ll n$.

For instance, with the Netflix dataset and $f = 64$, the problem is to generate and solve $m = 480190$ systems, each formed by a 64×64 rank-k update, with the average $k = 209$ (see Fig. 5). There is not enough parallelism in computing a single system for an efficient multi-core implementation. Instead, we do a batched implementation that generates and solves the m systems in parallel. For this, we use OpenMP to parallelize the loops in Algorithm 2.

High efficiency can be attained by relying on optimized Level 3 BLAS routines, which operate on matrices instead of individual vectors, enabling data reuse and optimizations for cache efficiency, improving performance to be compute-bound instead of memory-bound. To use Level 3 BLAS, we copy the relevant columns of Y to workspaces $\hat{Y}$ and V, with the R^u column scaling included in V, as shown in Fig. 3, then use a gemm (general matrix-matrix multiply) BLAS call. Since A is symmetric, work could be reduced by using an extended BLAS routine such as gemmt in Intel MKL [25] or syrkx in NVIDIA cuBLAS [46] instead of gemm.

Updating the item-factors is exactly the same, except it uses columns of R instead of rows of R. For updating the user-factors, we store R in CSR (compressed sparse row) format, which gives efficient, contiguous access to each row of R, but slow access to columns of R. For efficiency in updating the item-factors, we also store R

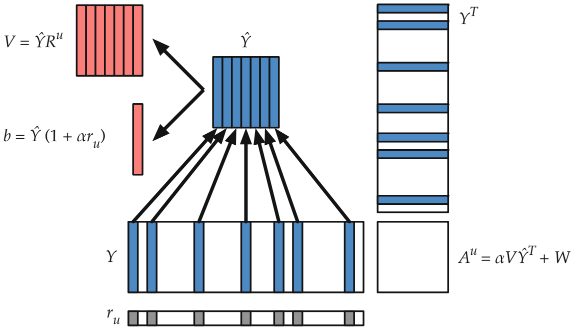

$V = \hat{Y}R^u$
$\hat{Y}$
Y^T
$b = \hat{Y}\,(1 + \alpha r_u)$
Y
$A^u = \alpha V\hat{Y}^T + W$
r_u

Fig. 3 Schematic of $A^u = YR^uY^T$ and $b = YC^u p_u$. Shaded boxes in row r_u represent nonzeros; only corresponding shaded columns of Y and rows of Y^T contribute to A^u and b

Algorithm 2 Multi-core CPU ALS algorithm.

```
function ALS_CPU( input: $\alpha$, $\lambda$, $R$; output: $X$, $Y$ )
    set $Y$ to random initial guess
    while not converged
        // update user-factors $X$
        $W = YY^T + \lambda I$ using syrk BLAS
        parallel for $u = 1, \dots, m$
            copy columns of $Y$ corresponding to nonzeros in $r_u$ to $\hat{Y}$
            copy and scale columns of $\hat{Y}$ as $V = \hat{Y} R^u$
            accumulate scaled columns of $\hat{Y}$ as $b_u = \hat{Y}(1 + \alpha r_u)$
            $A^u = \alpha V \hat{Y} + W$ using gemm BLAS (single-threaded)
            solve $A^u x_u = b_u$ using Cholesky (single-threaded)
        end
        // update item-factors $Y$
        $W = XX^T + \lambda I$ using syrk BLAS
        parallel for $i = 1, \dots, n$
            copy columns of $X$ corresponding to nonzeros in $r_i$ to $\hat{X}$
            copy and scale columns of $\hat{X}$ as $V = \hat{X} R^i$
            accumulate scaled columns of $\hat{X}$ as $b_i = \hat{X}(1 + \alpha r_i)$
            $A^i = \alpha V \hat{X} + W$ using gemm BLAS (single-threaded)
            solve $A^i y_i = b_i$ using Cholesky (single-threaded)
        end
    end
end function
```

in CSC (compressed sparse column) format, which gives efficient, contiguous access to each column of R.

Because the number of nonzeros per row can vary significantly (see Fig. 5), there will be a load imbalance between different processors. This is easily solved by using the OpenMP dynamic scheduler, adding `schedule(dynamic,NB)`, with a block size NB. We set NB = 200, but performance is not sensitive to the exact value.

2.4 GPU Implementation

A brief summary of the GPU architecture will help to understand the GPU implementation. A GPU kernel divides its computation into a grid of thread blocks, and each thread block into a grid of threads. Within each thread block, threads are not independent, but execute the same instructions on different data. Threads can synchronize and communicate via shared memory, which is a kind of fast, user-controlled cache. Each thread's local variables are stored in a large register file. Different thread blocks execute asynchronously, without an easy way to synchronize or communicate. An NVIDIA Kepler GPU contains up to 15 multiprocessors, each with 192 cores.

Due to this GPU architecture, the GPU implementation shown in Algorithm 3 is structured differently than the CPU implementation in Algorithm 2. Each thread block computes one tile of a matrix A^u and its right-hand side b_u. As with the CPU implementation, a single system has insufficient parallelism to fully occupy all the GPU's cores. Filling a GPU requires hundreds of thread blocks and tens of thousands of threads. Therefore, we use a batched implementation, where a single GPU kernel generates a batch of s matrices using the BATCHED_SPARSE_SYRK routine, then a batched Cholesky routine factors them, and finally batched triangular solvers solve the resulting systems. We use the batched Cholesky and triangular solves from the BEAST project [33]. We used a batch size of $s = 4096$ to balance parallelism with GPU memory requirements. However, performance is not sensitive to the exact batch size.

Algorithm 3 GPU implementation of ALS, using batched operations.

```
function ALS_GPU( input: $\alpha$, $\lambda$, $R$; output: $X$, $Y$ )
    // workspaces: $A$ is $f \times f \times s$, $B$ is $f \times s$
    set $Y$ to random initial guess
    while not converged
        // update user-factors $X$
        $W = YY^T + \lambda I$ using syrk from cuBLAS
        for $k = 1, \ldots, m$ by batch size $s$
            BATCHED_SPARSE_SYRK computes $A^u = \alpha Y R^u Y^T + W$ and $b_u = YC^u p_u$
                for $u = k, \ldots, k + s$
            BATCHED_CHOLESKY factors $A^u$ for $u = k, \ldots, k + s$
            BATCHED_SOLVE solves $A^u x_u = b_u$ for $u = k, \ldots, k + s$
        end
        // update item-factors $Y$
        $W = XX^T + \lambda I$ using syrk from cuBLAS
        for $i = 1, \ldots, n$ by batch size $s$
            BATCHED_SPARSE_SYRK computes $A^i = \alpha X R^i X^T + W$ and $b_i = XC^i p_i$
                for $i = k, \ldots, k + s$
            BATCHED_CHOLESKY factors $A^i$ for $i = k, \ldots, k + s$
            BATCHED_SOLVE solves $A^i y_i = b_i$ for $i = k, \ldots, k + s$
        end
    end
end function
```

The implementation of the BATCHED_SPARSE_SYRK GPU kernel is conceptually similar to the CPU kernel. Like the CPU kernel, it copies the relevant columns of Y to a workspace $\hat{Y}$, in this case stored in GPU shared memory. Instead of copying all the relevant columns at once, it copies just one block of kb columns at a time and multiplies these, storing the results in registers, then continues with the next block. Unlike the CPU version, here the copy and multiply are fused into one kernel. The multiply is based an optimized gemm GPU kernel [32], which sub-tiles the output matrix A^u, with each GPU thread computing one entry in each sub-tile (Fig. 4).

A few optimizations can be made. Since A^u is symmetric, only the tiles on or below the diagonal need to be computed; tiles above the diagonal are known by symmetry. Also, since matrix Y is read-only, it is beneficial to bind its memory to GPU *texture memory*, which has optimized caching for read-only data. Texture memory also simplifies the code by dealing with out-of-bounds memory accesses

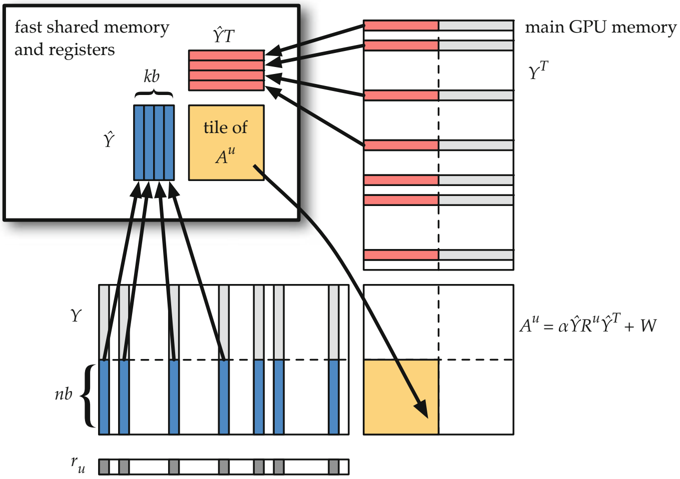


Fig. 4 Schematic of sparse-syrk GPU kernel. A^u is divided into $nb \times nb$ tiles. Block of kb relevant columns are loaded into shared memory and multiplied in registers at a time. At end, tile of A^u in registers is written back to main GPU memory. b_u is also computed (not shown)

in hardware—the software can pretend that Y is bigger than it actually is. This allows for fixed loop bounds and eliminates cleanup code, enabling more compiler optimizations.

2.5 Setup and Datasets

For performance comparison, we chose three ALS implementations from popular data analytics software packages: Mahout version 0.9 [1, 48], GraphLab version 1.3 [12, 37, 38], and Spark MLlib version 1.5 [2]. All results used single precision and were obtained on a two-socket 2.6 GHz Intel Sandy Bridge E5-2670 with 8 cores per socket. CPU implementations were linked with Intel's Math Kernel Library (MKL) version 11.1.2 [25]. Our GPU implementation ran on an NVIDIA Kepler K40c GPU with CUDA version 7.0 [47].

To compare performance, we target several recommendation datasets that are available online: Netflix Prize [4], Million Song [6], and Yahoo! Song [53]. For tuning parameters of the GPU implementation, we employ an autotuning sweep using the BEAST framework [24], with the EachMovie dataset [43, 53], a smaller dataset that permits executing a comprehensive set of kernel configurations in a moderate runtime. Table 1 summarizes properties of the datasets.

For the Netflix Prize dataset, we show histograms in Fig. 5 of the number of nonzeros per row (left) and per column (right). The minimum, median, mean, and maximum number of nonzeros per row and column are annotated in each graph. As

Table 1 Dataset properties

Dataset	# users	# items	# nonzeros
Netflix prize	480,190	17,771	100,480,508
Million song	1,019,318	384,546	48,373,586
Yahoo! song	130,558	136,736	49,770,695
EachMovie	1,623	61,265	2,811,717

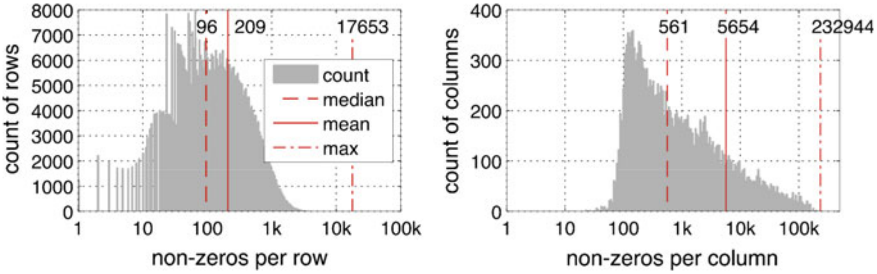


Fig. 5 Nonzero distribution of rows (*left*) and columns (*right*) of Netflix Prize dataset

previously noted, the wide range of nonzeros per row and column means different users and items incur widely different costs in computing YC^uY^T and XC^iX^T, potentially leading to load imbalance.

2.6 Auto Tuning

The sparse-syrk GPU kernel has four tunable parameters: tile size nb, block size kb, and thread block dimensions dx and dy. The kernel is generalized so that any value of nb can be used for any feature space size f. The optimal parameters are not obvious and not easy to derive by an analytical formula. Therefore the factorization calls for a real autotuning sweep. To achieve high performance, classic heuristic automatic software tuning methodology is applied, where a large number of kernels are generated and run, and the fastest ones identified.

The BEAST autotuning framework [39] enumerates and tests all possible kernel configurations. Various constraints are applied to limit the search space. Configurations violating correctness constraints—such as exceeding the maximum shared memory, or nb not divisible by the thread block dimensions—are eliminated. Several heuristic constraints are also applied, for instance, ensuring a compute-intensive kernel by requiring the ratio of multiply-add instructions to load instructions is at least 2. While kernels that violate these soft constraints will run correctly, they will not keep the GPU fully occupied, leading to lower performance.

After applying these constraints, BEAST generated 330 kernel configurations to test. The kernels were tested on the modest sized EachMovie dataset, timing the

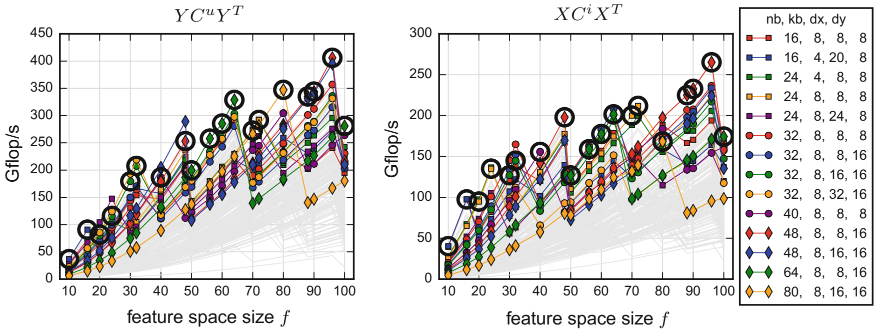


Fig. 6 Performance of all kernels (*gray lines*), highlighting ones that are best for some size. Circled kernel is chosen as best for each size(color figure online)

sparse-syrk for both the user-factor and the item-factor matrix generation. Due to differences in the size of Y and X and the sparsity of R^u and R^i, the performance was not identical between these two. We ran tests for sizes of f that are multiples of 8 and multiples of 10, from 8 to 100.

The performance of all these kernels is plotted in gray in Fig. 6. Kernels that were best for some size are highlighted with colored markers. For each size f, the circled kernel was chosen as the best overall kernel.

Inspecting the data reveals that no one configuration was optimal across all feature space sizes. Taking the yellow diamond (80, 8, 16, 16) kernel as an example: for small f it is a poor performer, but the performance increases as f increases, until it is the best kernel for $f = 80$, where $f = nb$. For the next size, $f = 88$, its performance plummets to less than half the optimal performance. This occurs because it goes from one tile to four tiles covering each matrix A, wasting three large tiles to cover the extra 8 rows and columns. This saw tooth pattern is evident for all the configurations.

While often the best kernel for user-factors (left in Fig. 6) and item-factors (right) is the same, there are several instances where this is not true due to the difference in sparsity patterns. In these cases, the kernel with the best geometric mean performance is chosen as the best compromise between the two.

This analysis highlights the need for autotuning. The performance difference between the best and worst kernels is dramatic—between a factor of 6 and 72 times for a particular f. Also, the optimal kernel configuration depends heavily on the size f, and to a lesser extent on the actual dataset. While some kernel configurations make sense in retrospect, it was infeasible to predict optimal kernels in all cases.

2.7 Performance Evaluation

Execution time of a single ALS iteration (updating user-factors and item-factors once) for the three large benchmark databases—Netflix, Million Song, and Yahoo!

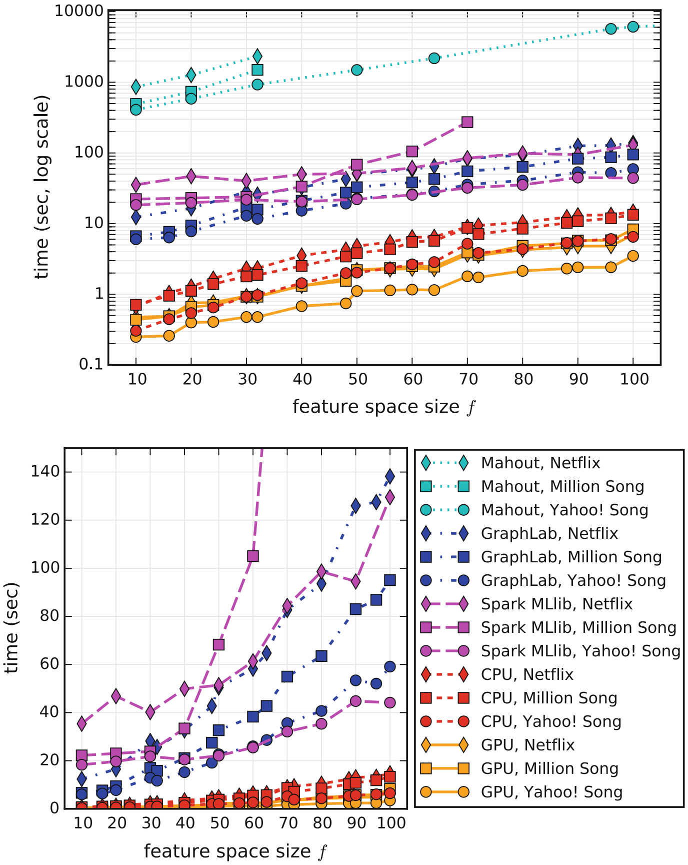


Fig. 7 Time in log scale (*top*) and linear scale (*bottom*) for single ALS iteration, using 16 CPU cores or GPU

Song—is presented in Fig. 7, in both log and linear scale. This covers a range of feature space sizes, all using 16 CPU cores or the GPU. A large performance difference between implements is evident. Mahout is nearly two orders-of-magnitude slower than GraphLab and Spark. This is not surprising, as Mahout is written in Java while GraphLab is a newer implementation written in C++. Spark, while written in Scala/Java, links with native optimized BLAS to achieve good performance. For $f \geq 50$ with the Yahoo and Netflix datasets, Spark had performance comparable to GraphLab. However, with the Million Song dataset, the Spark execution time increased markedly for $f \geq 50$, and it encountered an exception for $f \geq 80$. Our CPU implementation is 10 times faster than GraphLab and 19 times faster than Spark MLlib, on average.

The speedup of our GPU implementation over Mahout, GraphLab, Spark, and our CPU implementation is given in Fig. 8. The GPU achieves an average speedup of 2.1

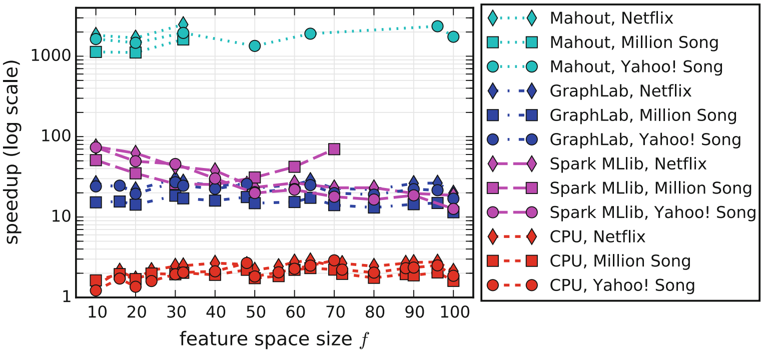


Fig. 8 Speedup in log scale of GPU implementation over Mahout, GraphLab, Spark, and CPU implementations using 16 cores

times over our CPU implementation. Compared to GraphLab, the GPU is on average 20.9 times faster, and compared to Spark it is 35.3 times faster. Mahout performs poorly, taking 1684 times longer, on average, to compute a single ALS iteration.

While speedups are similar across datasets, our GPU implementation consistently gets the best speedups for the Netflix dataset and the least speedups for the Million Song dataset. This may be because the Million Song dataset has the smallest average nonzeros-per-row and nonzeros-per-column, with a mean of 47 nonzeros per row and 126 per column, compared to 209 and 5654 for the Netflix dataset (Fig. 5). Having more nonzeros means a higher floating point operation count in the sparse-syrk routine to amortize memory reads.

We have presented both a multi-core CPU and a GPU implementation for the alternating least-squares algorithm to compute recommendations based on implicit feedback datasets. The central kernel involved is sparse_syrk, an algorithm-specific kernel achieving compute-bound performance for multiplying two dense matrices scaled by a sparse diagonal matrix. Our results demonstrate the advantage of fully exploiting the available parallelism by using a batched implementation, along with using optimized kernels, either from the vendor's BLAS library or custom auto-tuned kernels. This yields good performance over several different datasets and a range of feature space sizes.

3 GPU Acceleration of Singular Value Decomposition

3.1 Introduction

A partial singular value decomposition (SVD) [18] of a sparse matrix is a powerful tool for data analysis, where the data is represented as the sparse matrix. The ability of the SVD to filter out noise and extract the underlying features of the data has been demonstrated in many applications, including Latent Semantic Indexing (LSI) [5, 13], recommendation systems [13, 55], population clustering [49], and subspace tracking [28]. The SVD is also used to compute the leverage scores – statistical measurements for sampling the data in order to reduce the cost of the data analysis [21].

In recent years, the amount of data being generated from the observations, experiments, and simulations has been growing at unprecedented paces in many areas of studies, e.g., science, engineering, medicine, finance, social media, and e-commerce [11, 14]. The algorithmic challenges to analyze such "Big Data" are exacerbated by its massive volume and wide variety as well as its high veracity and velocity [35]. Though the SVD has the potential to address the variety and veracity of the modern data sets, the traditional approaches to computing the partial SVD access the data repeatedly, e.g., block Lanczos [19]. This is a significant drawback on a modern computer, where the data access has become significantly more expensive compared to arithmetic operations, both in terms of time and energy consumptions. The gap between the communication and computation costs is expected to further grow on future computers [15, 20], and this high cost of the communication is exacerbated by the Big Data. This hardware trend is certainly true for the GPU.

3.2 Randomized Algorithms to Compute SVD

To address this hardware trend, a randomized algorithm [21] has been gaining attention since compared to the traditional algorithms, it may require fewer data accesses to compute the SVD of the matrices arising from the modern applications (see Fig. 9 for an illustration of the algorithm). To compare the performance of different algorithms for computing the truncated SVD, we implemented the framework, which encapsulates these algorithms on multicore CPUs with multiple GPUs [64]. This framework not only allows us to develop software whose performance can be tuned based on domain specific knowledge, but it also allows a user from one discipline to test an algorithm from another, or to combine the techniques from different algorithms (see Fig. 10 for the list of the algorithms). For example, we studied the performance of a block Lanczos, combining it with communication-avoiding [22, 62] and thick-restarting [3, 61]; two techniques developed by two different disciplines

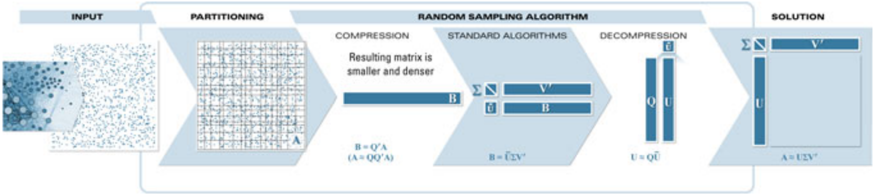


Fig. 9 Randomized algorithm to compute truncated SVD

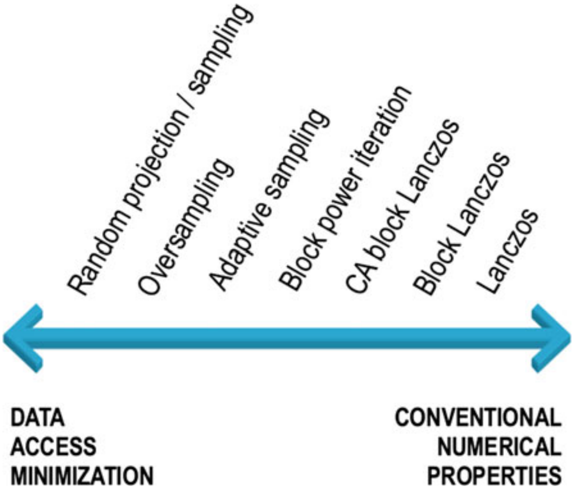


Fig. 10 Algorithms to compute truncated SVD

– computer science and numerical linear algebra. These two techniques allow us to build the projection subspace with the minimum data access and accelerate the solution convergence by retaining the useful information when restarting the iteration, respectively. Hence, compared to the randomized algorithm, Lanczos could build a projection subspace of the same dimension, which is richer in useful information with fewer communication phases, and potentially with about the same amount of data access. Unfortunately, this is possible only when the matrix can be partitioned well, while many of the matrices from the modern applications cannot be partitioned in such a way, leading to the significant overheads of the communication-avoiding technique in term of the computation and storage requirements, as well as the communication volume. Hence, there is a growing interest in a novel algorithm that can more efficiently compute the SVD of the massive data that are being generated from many modern applications, and the randomized algorithm is one of such algorithms with the potential.

3.3 Hybrid CPU/GPU Implementation

Figure 11 shows the pseudocode of a randomized algorithm to compute the SVD. Since the computational cost of the randomized algorithm is dominated by the cost of generating the projection basis vectors, $\widehat{P}$ and $\widehat{Q}$, we accelerate this step using GPUs, while the SVD of the projected matrix B is redundantly computed by each MPI process on CPU. To generate the basis vectors, the two main computational kernels of the randomized algorithm are the sparse-matrix dense-matrix multiply (SpMM) and the orthogonalization. In this subsections, we describe our implementations of these two kernels on a hybrid CPU/GPU cluster.

```
for $j = 1, 2, \ldots, s$ do
    1.  Orthogonalize $\widehat{Q}$
        $QR := \widehat{Q}$
    2.  Sample range of $A$
        $\widehat{P} := AQ$
    3.  Orthogonalize $\widehat{P}$
        $PB := \widehat{P}$
    4.  Prepare to iterate
        if $j < s$ then
          $\widehat{Q} := A^T P$
        end if
end for
```

(a) Power iteration to generate subspace.

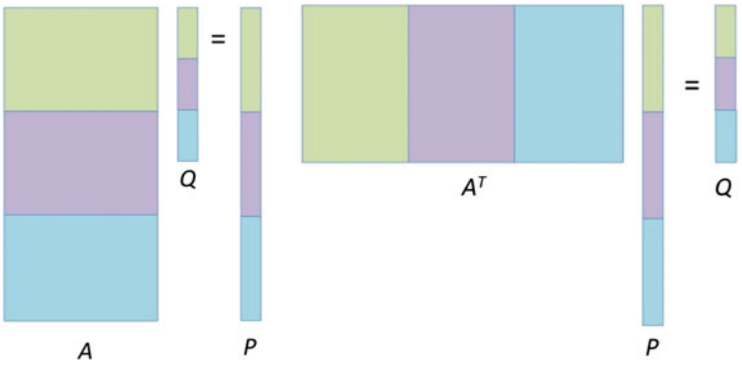
```
1. Generate $\widehat{P}_{k+\ell}$ and $\widehat{Q}_{k+\ell}$ that approximate $\widehat{A}$,
        $\widehat{A} \approx \widehat{P}\widehat{Q}^T$.
2. Compute SVD of the projected matrix $B$,
        $B = X\widehat{\Sigma}Y^T$,
   where $B = \widehat{P}^T\widehat{A}\widehat{Q}$.
3. Compute approximate partial SVD of $\widehat{A}$,
        $\widehat{A}_k \approx \widehat{U}_k\widehat{\Sigma}_k\widehat{V}_k^T$,
   where $\widehat{U}_k = \widehat{P}X_k$ and $\widehat{V}_k = \widehat{Q}Y_k$.
```

(b) Projection method to compute partial SVD.

Fig. 11 Randomized algorithm to compute partial SVD based on power iteration

3.3.1 Sparse Matrix Matrix Multiply

To perform SpMM with the matrix A on a hybrid CPU/GPU cluster, we distribute A among the GPUs in a 1D block row format (e.g., using a graph or hypergraph partitioning algorithm). The basis vectors $\widehat{P}$ and $\widehat{Q}$ are then distributed in the same formats. Then, to perform SpMM, each GPU first exchanges the required non-local vector elements with its neighboring GPUs. This is done by first copying the required local elements from the GPU to the CPU, then performing the point-to-point communication among the neighbors using the non-blocking MPI (i.e., `MPI_Isend` and `MPI_Irecv`), and finally copying the non-local vector elements back to the GPU. Then, each GPU computes the local part of the next basis vectors using the CuSPARSE SpMM in the compressed sparse row (CSR) format. This was an efficient communication scheme in our previous studies to develop a linear solver [65], where the coefficient matrix A arising from a scientific or engineering simulation is often sparse and structured, e.g., with three-dimensional embedding. Unfortunately, sparse matrices originating from the modern data sets such as social networks and/or commercial applications have irregular sparsity structures, and have wide ranges of nonzero counts per row. In fact, they often exhibit power-law distributions of nonzeros as they result from scale-free graphs. As a result, this point-to-point communication with all the neighbors at once could be inefficient (in term of time and buffer storage). To alleviate the problem, our current implementation is based on a collective communication scheme. For example, using `MPI_Allgatherv`, each process sends its local vector elements, which are needed by at least one of its neighbors, to all the processes. Though this all-to-all approach requires the buffer to store the receiving messages from all the processes at once, it could obtain a significant speedup over the point-to-point communication, especially when the nonzeros of the matrix follows the power-law distribution.



(a) 1DBR with neighborhood-collective before local SpMM.

(b) 1DBC with all-reduce after local SpMM.

Fig. 12 Illustration of matrix and vector distributions for SpMM with A and A^T. The submatrices distributed to the same GPU are colored in the same *color*. In Figure (**a**) or (**b**), the sparse matrices A and A^T are distributed either in 1D block row or block column (1DBR or 1DBC in short), respectively

Sine many matrices of our interests are tall-skinny, to perform SpMM with A^T, our current implementation keeps the input and output vectors, $\widehat{P}$ and $\widehat{Q}$, in the 1D block row distribution, but distribute A^T in the 1D block column (see Fig. 12b). Since the columns of A^T are the same as the rows of A on each GPU, we do not need to separately store A^T and A. In this implementation, each GPU first computes SpMM with its local parts of A^T and $\widehat{P}$, and then copies the partial result to the CPU. Then, the MPI process computes the final result $\widehat{Q}$ by a global all-reduce, and copies its local part back to the GPU. Hence, this requires each MPI process to store the global vectors $\widehat{Q}$. However, when A^T has the power-law distribution, performing SpMM with A^T in the 1D block row requires each GPU to store the much longer global vectors $\widehat{P}$. Our performance results have demonstrated the advantage of this all-reduce communication. Furthermore, partitioning A^T in the 1D block column often led to a higher performance of SpMM on each GPU as the local submatrix becomes more square than tall-skinny.

3.3.2 Orthogonalization

For our experiments in this paper, we used the block classical Gram-Schmidt (CGS) [18] to orthogonalize a set of vectors against another set of vectors (block orthogonalization, or BOrth in short) and the Cholesky QR (CholQR) [56] to orthogonalize the set of vectors against each other. In our previous studies, these algorithms obtained great performance on multiple GPUs on a single compute node [63] or on a hybrid CPU/GPU cluster [65]. This is because these algorithms can orthogonalize the basis vectors with a low communication cost. For example, CholQR requires only one global reduction between the GPUs, while most of the local computation is based on BLAS-3 kernels on the GPU.

3.4 Randomized Algorithms to Update SVD

Though the randomized algorithms have the potential to efficiently compute the SVD on the GPUs, there are several obstacles that need to be overcome. In particular, the randomized algorithm may require only a small number of data accesses, but each data access can be expensive due to the irregular sparsity pattern of the matrix and the power-law distribution of its nonzeros. Though several techniques to avoid such communication have been proposed [22], these techniques may not be effective for computing the SVD of the modern data because they often require a significant computational or communication overhead due to the particular sparsity structure of the matrix [64].

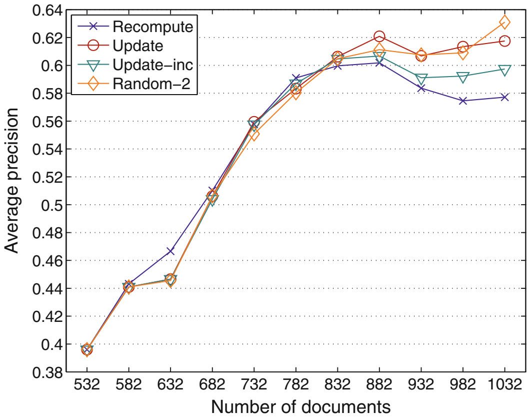
To address this challenge, we studied randomized algorithms to update (rather than recompute) the partial SVD as the changes are made to the data set [66]. This is an attractive approach because compared to recomputing it from scratch, the SVD may be updated more efficiently, while in modern applications, the existing data are being constantly updated and new data is being added. Moreover, in some applications, recomputing the SVD may not be possible because the original data, for which the SVD has been already computed, is no longer available. At the same time, in modern applications, the size of the update is significant even though it is much smaller than the massive data that has been already compressed. Therefore, an efficient updating algorithm is needed to address the large volume and high velocity of the modern data sets. Such applications with the rapidly changing data include the communication and electric grids, transportation and financial systems, personalized services on the internet, particle physics, astrophysics, and genome sequencing [11].

3.4.1 Case Studies

To study the potential of the randomized algorithm, we studied its performance for a popular statistical analysis tool, the principal component analysis (PCA) [7]. In PCA, a multidimensional dataset is projected onto a low-dimensional subspace given by the partial SVD such that related items are close to each other in the projected subspace. Here, we show the results from two particular applications of PCA, Latent Semantic Indexing (LSI) and population clustering.

For information retrieval by text mining [54], a variant of PCA, Latent Semantic Indexing (LSI) [13], has been shown to effectively address the ambiguity caused by the synonymy or polysemy, which are difficult to address using a traditional lexical-matching [31]. Figure 13a compares the average 11-point interpolated precisions [29] after adding different numbers of documents from the `MEDLINE` matrix. Our test matrices are the term-document matrices generated using the Text to Matrix

Fig. 13 Case studies with randomized algorithms for LSI ($k = 50$)



(a) Latent semantic indexing.

	Total number of documents, $n+d$							
Method	700	800	900	1000	1100	1200	1300	1400
Recompute	26.7	30.9	32.0	32.5	32.7	31.3	30.8	29.8
Update	26.7	29.8	30.1	30.7	31.5	30.7	30.4	29.7
Update-inc	26.7	29.8	30.1	30.6	30.9	30.1	29.8	29.5
Random-1	26.7	29.0	29.9	31.9	31.9	30.9	29.5	28.6
Random-2	26.7	29.6	29.6	30.0	31.0	30.1	30.0	29.7
Random-3	26.7	29.6	28.2	28.2	27.9	27.4	26.8	25.8

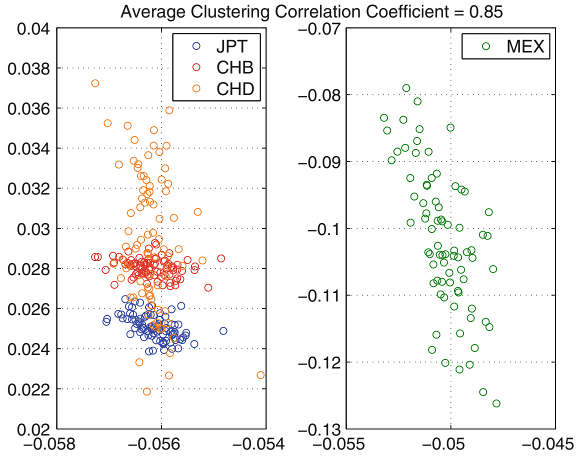
(b) Average 11-point interpolated precision for 6916-by-1400 CRANFIELD matrix with 225 queries, $n = 700$.

Generator (TMG)[1] and the TREC dataset,[2] and are preprocessed using the lxn.bpx weighing scheme [29]. These are the standard test matrices and were used in the previous studies [59, 68]. For our studies, we first performed 20 power iterations of the randomized algorithm to compute the rank-k approximation of the matrix $\widehat{A}$ representing the first 700 documents Then, the figure shows the average precision after new columns are added (e.g., under the column labeled "1000," 300 documents were added). To recompute the partial SVD of the matrix, we performed 20 power iterations, while the randomized algorithm used the oversampling parameter set to be $\ell = k$ (i.e., $r = 2k$), and performed two iterations that access the matrix three times. Since the basis vectors $\widehat{P}$ and $\widehat{Q}$ approximate the ranges of $\widehat{A}$ and $\widehat{A}^T$, respectively, the randomized algorithm accesses the matrix at least twice. Then, they access the matrix one more time to compute the projected matrix B. We let the incremental update algorithm (Update-inc) add $k + \ell$ columns at a time such that it requires about the same amount of memory as the randomized algorithm. We see that with only three data passes, the randomized algorithm obtained similar precisions as those of the updating algorithm. In some cases, the updating and randomized algorithms obtained higher precisions than recomputing the SVD, while the precisions of

[1] http://scgroup20.ceid.upatras.gr:8000/tmg.

[2] http://ir.dcs.gla.ac.uk/resources.

Fig. 14 Case studies with randomized algorithms for population clustering



(a) Population clustering.

	JPT+MEX	+ ASW	+ GIH	+ CEU
Recompute	1.00	1.00	1.00	0.97
No update	1.00	0.81	0.84	0.67
Update-inc	1.00	1.00	0.89	0.70
Random-1	1.00	0.95	0.92	0.86

(b) Average correlation coefficients of population clustering based on the five dominant singular vectors, where 83 African ancestry in south west USA (ASW), 88 Gujarati Indian in Houston (GIH), and 165 European ancestry in Utah (CEU) were incrementally added to the 116,565 SNP matrix of 86 Japanese in Tokyo and 77 Mexican ancestry in Los Angeles, USA (JPT and MEX). Random-1 iterated twice with $\ell = k$.

the incremental update slightly deteriorated at the end. Such phenomena were also reported in the previous studies [58, 68].

PCA has been also successfully used to extract the underlying genetic structure of human populations [45, 51, 52]. To study the potential of the randomized algorithm, we used it to update the SVD, when a new population is added to the population dataset from the HapMap project.[3] Figure 14 shows the correlation coefficient of the resulting population cluster, which is computed using the k-mean algorithm of MATLAB in the low-dimensional subspace given by the dominant left singular vectors. We randomly filled in the missing data with either −1, 0, or 1 with the probabilities based on the available information for the SNP. We let the randomized algorithm iterate twice, and with only the three data passes, the randomized algorithm

[3] http://hapmap.ncbi.nlm.nih.gov.

improved the clustering results, potentially reducing the number of times the SVD must be recomputed.

3.4.2 Performance Studies

We now study the performance of the randomized algorithm on the Tsubame Computer at the Tokyo Institute of Technology.[4] Each of its compute nodes consists of two six-core Intel Xeon CPUs and three NVIDIA Tesla K20Xm GPUs. We compiled our code using the GNU `gcc` version 4.3.4 compiler and the CUDA `nvcc` version 6.0 compiler with the optimization flag `-O3`, and linked it with Intel's Math Kernel Library (MKL) version xe2013.1.046.

Figure 15a compares the strong parallel scaling of the randomized algorithm with that of the current state-of-the-art updating algorithm [68]. Clearly, the state-of-the-art algorithm can spend significantly longer time in the orthogonalization, leading to a great speedup obtained by the randomized algorithm (i.e., the speedups of up to 14.1). At the same time, the speedup decreased on a larger number of GPUs. This is because the execution time of the randomized algorithm is dominated by SpMM, whose strong parallel scaling suffered from the increasing inter-GPU communication cost for this relatively small-scale matrix that was used for this study. On the other hand, the updating algorithm was still spending a significant amount of its execution time for the orthogonalization which was still compute intensive and scaled over the small number of the GPUs. On a larger number of GPUs, compared to the randomized algorithm, the updating algorithm is expected to suffer from the greater communication latency.

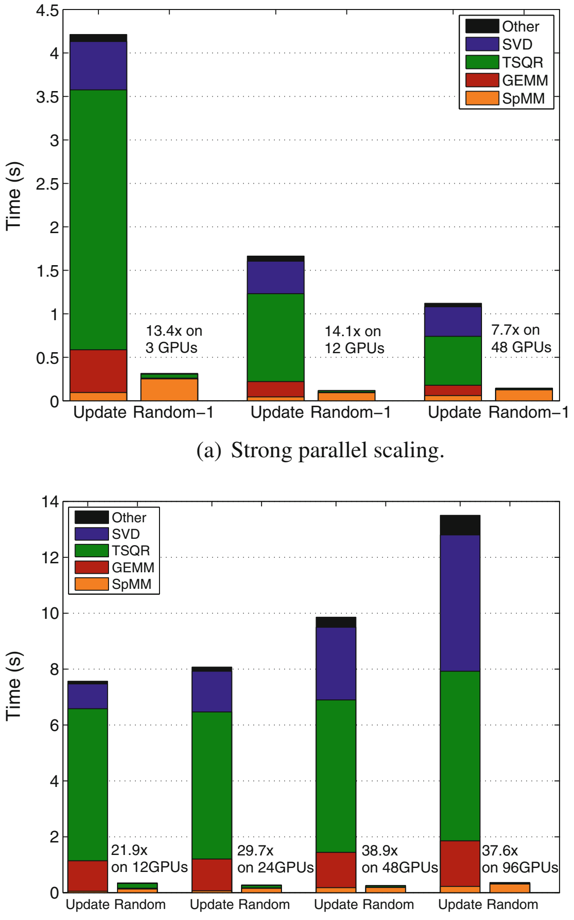
Figure 15b shows the weak parallel scaling results for the document-document matrix used in a previous LSI study [69]. The matrix row contains 2,559,430 documents, and each column contains about 4, 176 nonzero entries. The weak parallel scaling results, in particular, show the advantages of the randomized algorithm due to its ability to compress the desired information into a small projection subspace using a small number of data passes. For the updating algorithm, the accumulated cost of the SVDs of the projected matrices also became significant.

4 Conclusions

In this chapter, two mainstream Big Data algorithms were discussed: the Alternating Least Squares algorithm for solving the matrix completion problem and the Singular Value Decomposition algorithm for computing a low-rank approximation of a matrix, both of which pose significant challenges when offloading to a GPU or a computing cluster with multiple GPUs.

[4] http://tsubame.gsic.titech.ac.jp.

Fig. 15 Performance studies with randomized algorithms



(a) Strong parallel scaling.

(b) Weak parallel scaling.

In the case of the ALS algorithm, the technique of automatic software tuning was used to achieve top performance, leading to an order of magnitude performance advantage over mainstream open source packages, GraphLab and Spark MLlib, and three orders of magnitude advantage over Mahout (Hadoop), when using a single GPU as opposed to a multicore CPU (16 cores).

In the case of the SVD algorithm, the technique of random projection was applied to implement the algorithm efficiently on a computing cluster with up to 48 GPUs, and also to implement an algorithm for updating a previously computed factorization upon arrival of new data. In this case, the algorithmic innovations also lead to an order of magnitude performance advantage.

Both case studies show the kind of impact that cutting-edge HPC techniques can have on the world of Big Data by enabling efficient use of accelerators, which leads to massive performance improvements.

References

1. Apache, Mahout version 0.9 (2015a). https://mahout.apache.org/
2. Apache, Spark version 1.5 (2015b). http://spark.apache.org/
3. J. Baglama, L. Reichel, Augmented implicitly restarted Lanczos bidiagonalization methods. SIAM J. Sci. Comput. **27**, 19–42 (2005)
4. J. Bennett, S. Lanning, The netflix prize, in *Proceedings of the KDD Cup Workshop 2007* (ACM, New York, 2007), pp 3–6. http://www.cs.uic.edu/~liub/KDD-cup-2007/NetflixPrize-description.pdf
5. M.W. Berry, Large scale sparse singular value computations. Int. J. Supercomput. Appl. **6**, 13–49 (1992)
6. T. Bertin-Mahieux, D.P. Ellis, B. Whitman, P. Lamere, The million song dataset, in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR)* (2011)
7. C. Bishop, *Pattern Recognition and Machine Learning* (Springer, New York, 2006)
8. P. Biswas, T.C. Lian, T.C. Wang, Y. Ye, Semidefinite programming based algorithms for sensor network localization. ACM Trans. Sensor Networks (TOSN) **2**(2), 188–220 (2006)
9. E.J. Candès, B. Recht, Exact matrix completion via convex optimization. Found. Comput. Math. **9**(6), 717–772 (2009)
10. P. Chen, D. Suter, Recovering the missing components in a large noisy low-rank matrix: application to SFM. IEEE Trans. Pattern Anal. Mach. Intell. **26**(8), 1051–1063 (2004)
11. Committee on the Analysis of Massive Data, Committee on Applied and Theoretical Statistics, Board on Mathematical Sciences and Their Applications, Division on Engineering and Physical Sciences, National Research Council (2013). Frontiers in Massive Data Analysis. The National Academies Press
12. Dato, GraphLab version 1.3 (2015). https://dato.com/products/create/open_source.html
13. S. Deerwester, S. Dumais, G. Furnas, T. Landauer, R. Harshman, Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. **41**, 391–407 (1990)
14. DOE Office of Science, Synergistic challenges in data-intensive science and exascale computing. DOE Advanced Scientific Computing Advisory Committee (ASCAC) (2013). Data Subcommittee Report
15. S.H. Fuller, L.I. Millett, *The Future of Computing Performance: Game Over Or Next Level?* (National Academy Press, Washington, DC, 2011)
16. M. Gates, H. Anzt, J. Kurzak, J. Dongarra, Accelerating collaborative filtering using concepts from high performance computing, in *2015 IEEE International Conference on Big Data (Big Data)* (IEEE, 2015), pp. 667–676
17. D. Goldberg, D. Nichols, B.M. Oki, D. Terry, Using collaborative filtering to weave an information tapestry. Commun. ACM **35**(12), 61–70 (1992)
18. G. Golub, C. van Loan, *Matrix Computations*, 4th edn. (The Johns Hopkins University Press, Baltimore, 2012)
19. G. Golub, F. Luk, M. Overton, A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix. ACM Trans. Math. Softw. **7**, 149–169 (1981)
20. S. Graham, M. Snir, C. Patterson, *Getting Up to Speed: The Future of Supercomputing* (The National Academies Press, Washington, DC, 2004)
21. N. Halko, P. Martinsson, J. Tropp, Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. SIAM Rev. **53**(2), 217–288 (2011)
22. M. Hoemmen, Communication-avoiding Krylov subspace methods. Ph.D. thesis, University of California, Berkeley (2010)

23. Y. Hu, Y. Koren, C. Volinsky, Collaborative filtering for implicit feedback datasets, in *IEEE International Conference on Data Mining (ICDM)* (2008), pp. 263–272
24. Innovative Computing Lab, BEAST (2015). http://icl.utk.edu/beast/
25. Intel Corp, Developer Reference for Intel Math Kernel Library (2015). https://software.intel.com/en-us/articles/mkl-reference-manual
26. Intel Corp, Intel Data Analytics Acceleration Library 2016, Developer Guide (2016)
27. P. Jain, P. Netrapalli, S. Sanghavi, Low-rank matrix completion using alternating minimization, in *Proceedings of the Forty-Fifth annual ACM Symposium on Theory of Computing* (ACM, 2013), pp 665–674
28. I. Karasalo, Estimating the covariance matrix by signal subspace averaging. IEEE Trans. Acoust. Speech Signal Process. **34**(1), 8–12 (1986)
29. T. Kolda, D. O'Leary, A semidiscrete matrix decomposition for latent semantic indexing information retrieval. ACM Trans. Inf. Syst. **16**(4), 322–346 (1998)
30. Y. Koren, Factorization meets the neighborhood: a multifaceted collaborative filtering model, in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'08* (ACM, New York, 2008), pp. 426–434
31. R. Krovetz, W.B. Croft, Lexical ambiguity and information retrieval. ACM Trans. Inf. Syst. **10**(2), 115–141 (1992)
32. J. Kurzak, S. Tomov, J. Dongarra, Autotuning gemm kernels for the Fermi GPU. IEEE Trans. Parallel Distrib. Syst. **23**(11), 2045–2057 (2012)
33. J. Kurzak, H. Anzt, M. Gates, J. Dongarra, Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs. Trans. Parallel Distrib. Syst. (2015). doi:10.1109/TPDS.2015.2481890
34. C. Lam, *Hadoop in Action* (Manning Publications Co., Stamford, 2010)
35. D. Laney, 3D data management: controlling data volume, velocity, and variety. Application Delivery Strategies by META Group Inc., File: 949 (2001)
36. E. Liberty, F. Woolfe, P.G. Martinsson, V. Rokhlin, M. Tygert, Randomized algorithms for the low-rank approximation of matrices. Proc. National Acad. Sci. **104**(51), 20167–20172 (2007)
37. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, j.M. Hellerstein, GraphLab: a new framework for parallel machine learning. CoRR abs/1006.4990 (2010). http://arxiv.org/abs/1006.4990
38. Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
39. P. Luszczek, M. Gates, J. Kurzak, A. Danalis, J. Dongarra, Search space generation and pruning system for autotuners, in *International Workshop on Automatic Performance Tuning (iWAPT 2016)* (2016, submitted)
40. D. Lyubimov, Command line interface, stochastic SVD. Technical report, The Apache Software Foundation (2014). https://mahout.apache.org/users/dim-reduction/ssvd.page/SSVD-CLI.pdf
41. M.W. Mahoney, Randomized algorithms for matrices and data. Found. Trends® Mach. Learn. **3**(2), 123–224 (2011)
42. P.G. Martinsson, V. Rockhlin, M. Tygert, A randomized algorithm for the approximation of matrices. Technical report, DTIC Document (2006)
43. P. McJones, Eachmovie collaborative filtering data set. DEC Systems Research Center 249 (1997)
44. X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen et al., MLlib: Machine learning in Apache Spark (2015). arXiv preprint arXiv:150506807
45. P. Menozzi, A. Piazza, L. C-Sforza, Synthetic maps of human gene frequencies in Europeans. Science **201**, 786–792 (1978)
46. NVIDIA Corp, cuBLAS Library User Guide, v7.0 (2015a)
47. NVIDIA Corp, CUDA C Programming Guide, v7.0 (2015b)

48. S. Owen, R. Anil, T. Dunning, E. Friedman, *Mahout in Action* (Manning Publications Co., Greenwich, 2011)
49. P. Paschou, E. Ziv, E. Burchard, S. Choudhry, W. R-Cintron, M. Mahoney, P. Drineas, PCA-correlated SNPs for structure identification in worldwide human populations. PLoS Genet. **3**, 1672–1686 (2007)
50. A. Paterek, Improving regularized singular value decomposition for collaborative filtering, in *Proceedings of KDD Cup and Workshop* (2007), pp. 39–42
51. N. Patterson, A. Price, D. Reich, Population structure and eigenanalysis. PLoS Genet. **2**(12), 2074–2093 (2006)
52. A. Price, N. Patterson, R. Plenge, M. Weinblatt, N. Shadick, D. Reich, Principal components analysis corrects for stratification in genome-wide association studies. Nature Genet. **38**(8), 904–909 (2006)
53. R.A. Rossi, N.K. Ahmed, The network data repository with interactive graph analytics and visualization, in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015). http://networkrepository.com
54. G. Salton, M. McGill, *Introduction to Modern Information Retrieval* (McGraw-Hill, New York, 1983)
55. B. Sarwar, G. Karypis, J. Konstan, J. Riedl, Analysis of recommendation algorithms for e-commerce, in *Proceedings of the 2nd ACM Conference on Electronic Commerce* (2000), pp 158–167
56. A. Stathopoulos, K. Wu, A block orthogonalization procedure with constant synchronization requirements. SIAM J. Sci. Comput. **23**(6), 2165–2182 (2002)
57. W. Tan, L. Cao, L.L. Fong, Faster and cheaper: Parallelizing large-scale matrix factorization on gpus. CoRR abs/1603.03820 (2016). http://arxiv.org/abs/1603.03820
58. J. Tougas, R. Spiteri, Updating the partial singular value decomposition in latent semantic indexing. Comput. Statist. Data Anal. **52**, 174–183 (2007)
59. E. Vecharynski, Y. Saad, Fast updating algorithms for latent semantic indexing. SIAM J. Matrix Anal. Appl. **35**(3), 1105–1131 (2014)
60. T. White, *Hadoop: The Definitive Guide* (O'Reilly Media, Inc., Sebastopol, 2012)
61. K. Wu, H. Simon, Thick-restart Lanczos method for large symmetric eigenvalue problems. SIAM J. Matrix Anal. Appl. **22**(2), 602–616 (2000)
62. I. Yamazaki, K. Wu, A communication-avoiding thick-restart lanczos method on a distributed-memory system, in *Proceedings of the 2011 International Conference on Parallel Processing, Euro-Par'11* (Springer, Berlin, 2012), pp. 345–354
63. I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, J. Dongarra Improving the performance of CA-GMRES on multicores with multiple GPUs, in *Proceedings of the IEEE International Parallel and Distributed Symposium (IPDPS)* (2014a), pp. 382–391
64. I. Yamazaki, T. Mary, J. Kurzak, S. Tomov, Access-averse framework for computing low-rank matrix approximations, in *Proceedings of the International Workshop on High Performance Big Graph Data Management, Analysis, and Minig* (2014b), pp. 70–77
65. I. Yamazaki, S. Rajamanickam, E. Boman, M. Hoemmen, M. Heroux, S. Tomov, Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2014c), pp. 933–944
66. I. Yamazaki, J. Kurzak, P. Luszczek, J. Dongarra, Randomized algorithms to update partial singular value decomposition on a hybrid CPU/GPU cluster, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2015), pp. 345–354
67. M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, vol. 10 (2010), p.10
68. H. Zha, H. Simon, On updating problems in latent semantic indexing. SIAM J. Sci. Comput. **21**(2), 782–791 (1999)

69. H. Zha, O. Marques, H. Simon, Large-scale SVD and subspace-based methods for information retrieval, in *Solving Irregularly Structured Problems in Parallel*, vol. 1457, Lecture Notes in Computer Science, ed. by A. Ferreira, J. Rolim, H. Simon, S.-H. Teng (Springer, Heidelberg, 1998), pp. 29–42
70. Y. Zhou, D. Wilkinson, R. Schreiber, R. Pan, Large-scale parallel collaborative filtering for the netflix prize in *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management, AAIM'08* (Springer, Berlin, 2008), pp. 337–348