Sampling Algorithms to Update Truncated SVD

Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra
University of Tennessee, Knoxville, Tennessee, U.S.A.

***Abstract*—**

A truncated singular value decomposition (SVD) is a powerful tool for analyzing modern datasets. However, the massive volume and rapidly changing nature of the datasets often make it too expensive to compute the SVD of the whole dataset at once. It is more attractive to use only a part of the dataset at a time and incrementally update the SVD. A randomized algorithm has been shown to be a great alternative to a traditional updating algorithm due to its ability to efficiently filter out the noises and extract the relevant features of the dataset. Though it is often faster than the traditional algorithm, in order to extract the relevant features, the randomized algorithm may need to accesses the data multiple times, and this data access creates a significant performance bottleneck. To improve the performance of the randomized algorithm for updating SVD, we study, in this paper, two sampling algorithms that access the data only two or three times, respectively. We present several case studies to show that only a small fraction of the data may be needed to maintain the quality of the updated SVD, while our performance results on a hybrid CPU/GPU computer demonstrate the potential of the sampling algorithms to improve the performance of the randomized algorithm.

***Index Terms*—sample; randomize; update SVD; out-of-core;**

I. Introduction

To analyze the modern datasets with a wide variety and veracity, a truncated singular value decomposition (SVD) [1] of the matrix representing the data is a powerful tool. The ability of the SVD to filter out noises and extract the underlying features of the data has been demonstrated in many data analysis tools, including Latent Semantic Indexing (LSI) [2], recommendation systems [3], population clustering [4], and subspace tracking [5]. Also, as the modern datasets are constantly being updated and analyzed, we develop a good understanding of the data (e.g., the singular value distribution), which can be used to tune the performance or the robustness of computing the SVD for that particular application (e.g., the required numerical rank for the accurate data analysis, or the number of data passes needed to compute the SVD). Furthermore, these tuning parameters stay roughly the same for different datasets from the same applications.

With the increase in the external storage capacity, the amount of data generated from the observations, experiments, and simulations has been growing at an unprecedented rate. These phenomena have led to the emergence of numerous massive datasets in many areas of studies including science, engineering, medicine, finance, social media, and e-commerce. The specific applications that generate the rapidly-changing massive datasets include the communication and electric grids, transportation and financial systems, personalized services on the internet, particle or astro physics, and genome sequencing. Hence, beside the variety and veracity of the dataset, the data analysis tool must address the challenges associated with the volume and velocity of the changes made to the dataset. For instance, the computers may not have enough compute power to accommodate such a rapidly growing or changing data if the computational complexity of the data analysis tool grows superlinearly with the data size. In addition, accessing the data through the local memory hierarchy is expensive, and accessing these data in the external storage is even more costly. Therefore, the data analysis tool needs to be data-pass efficient. In particular, it may become too costly to compute the SVD of the whole dataset at once, or to recompute the SVD every time the changes are made to the dataset. In some applications, recomputing the SVD may not even be possible because the original data, for which the SVD has been already computed, is no longer available. To address these challenges, an attractive approach is to update (rather than recompute) the SVD. For example, we can incrementally update the SVD using only a part of the matrix that fit in the core memory at a time. Hence, the whole matrix is moved to the core memory only once.

A randomized algorithm has been shown to be an efficient method to update SVD [6]. To reduce both the computational and data access costs, it projects the data onto a smaller subspace before computing the updated SVD. Compared with the state-of-the-art updating algorithm [7], the randomized algorithm often compresses the data into a smaller projection subspace with a lower communication latency cost. As a result, the randomized algorithm could obtain much higher performance on a modern computer, where the communication has become significantly more expensive compared with the arithmetic operations, both in terms of time and energy consumption. In addition, the randomized algorithm accesses the data only through the dense or sparse matrix-matrix multiplication (*GEMM* or *SpMM*) whose highly-optimized implementations are provided by many vendors. In other applications, the external storage (e.g., database) may provide a functionality to compute the matrix multiplication and only transfer the resulting vectors to the memory, thus avoiding the explicit generation and transfer of the matrix into the memory.

To filter out the noises and extract the relevant features, however, the randomized algorithm may require multiple data passes that become the performance bottleneck. In this paper, we use two methods to reduce this bottleneck. 1) We integrate data sampling into the randomized algorithm. Namely, we first sample the new data using the information gathered while compressing the previous data. Then, the randomized algorithm only uses the sampled data (which fits in the memory) to update the SVD. We present two sampling algorithms,

requiring only two or three data-passes, respectively (one pass to sample the data, and one or two additional passes to generate the projection subspace). 2) We study a randomized algorithm to incrementally update the SVD using a subset of the sampled rows. The algorithm does not access the rows that have been already compressed and uses only the sampled rows that can fit in the memory at a time. This becomes attractive when we fail to sample enough rows or all the sampled rows do not fit in the memory at once.

We present several case studies, in which we needed to sample only a fraction of the data to maintain the quality of the updated SVD. We also show the potential of the sampling algorithm to improve the performance of the randomized algorithm on multicore CPUs with an NVIDIA GPU using different implementations and data configurations. As the cost of data access grows due to the properties of both data and hardware, such sampling algorithms would likely play more critical roles in analyzing the modern datasets. Throughout this paper, we use $a_{i,j}$ and v_i to denote the (i,j)-th entry of a matrix A and the i-th entry of a vector $\mathbf{v}$, respectively.

II. Problem Statement

We assume that a rank-k approximation $A_k = U_k \Sigma_k V_k^T$ of an m-by-n matrix A has been computed where Σ_k is a k-by-k diagonal matrix whose diagonal entries approximate the k dominant singular values of A, and the columns of U_k and V_k approximate the corresponding left and right singular vectors, respectively. We then consider computing a rank-k approximation of a matrix $\widehat{A}$,

$$\widehat{A} \approx \widehat{U}_k \widehat{\Sigma}_k \widehat{V}_k^T, \tag{1}$$

where $\widehat{A} = [A \ D]$ and an m-by-d matrix D represents the new set of the columns being added to A. This problem is of particular interest for the term-document matrices from the latent semantic indexing (LSI) [8], and it is commonly referred to as the *updating-documents* problem. Two other updating problems exist, *updating-terms* and *updating-weights*. They add a new set of matrix rows and update a set of the matrix entries, respectively. Though we focus only on updating documents, all these three problems can be expressed as low-rank corrections to the original matrix. In many cases, $\widehat{A}$ is tall and skinny, having more rows than columns (i.e., $m \gg n, d$).

All the algorithms studied in this paper belong to a class of subspace projection methods:

Alg. 1. Subspace Projection Method:

1) Generate a pair of $k+\ell$ orthonormal basis vectors $\widehat{P}$ and $\widehat{Q}$ that approximately span the range and domain of the matrix $\widehat{A}$, respectively,
$$\widehat{A} \approx \widehat{P}\widehat{Q}^T, \tag{2}$$
where ℓ is an *oversampling* parameter [9] selected to enhance the performance or robustness of the algorithm.
2) Use a standard deterministic algorithm to compute the SVD of the projected matrix $B := \widehat{P}^T \widehat{A} \widehat{Q}$,
$$X \widehat{\Sigma} Y^T := \text{SVD}(B).$$
3) Compute the low-rank approximation (1) such that the diagonal entries of $\widehat{\Sigma}_k$ are the k dominant singular values of B, and $\widehat{U}_k := \widehat{P} X_k$ and $\widehat{V}_k := \widehat{Q} Y_k$, where X_k and Y_k are the corresponding left and right singular vectors of B, respectively.

The first step of generating the projection subspace typically dominates the performance, and is the focus of this paper.

III. Previous Algorithms

Two algorithms have been previously used to generate the basis vectors $\widehat{P}$ and $\widehat{Q}$ of (2).

A. Updating Algorithm

The *updating algorithm* [7] computes the basis vectors $\widehat{P}$ and $\widehat{Q}$ by first orthogonalizing D against the current approximate left singular vectors U_k,

$$\widehat{D} := (I - U_k U_k^T) D, \tag{3}$$

and then computing its QR factorization to orthonormalize the resulting $\widehat{D}$,

$$PR := \text{QR}(\widehat{D})$$

such that P is an m-by-d orthonormal matrix and R is a d-by-d upper-triangular matrix. The basis vectors are then given by

$$\widehat{P} = \begin{bmatrix} U_k & P \end{bmatrix} \quad \text{and} \quad \widehat{Q} = \begin{bmatrix} V_k & 0 \\ 0 & I_d \end{bmatrix}, \tag{4}$$

where I_d is a d-by-d identity matrix. The resulting $(k+d)$-by-$(k+d)$ projected matrix $B \equiv \widehat{P}^T \widehat{A} \widehat{Q}$ is given by

$$B = \begin{bmatrix} \Sigma_k & U_k^T D \\ & R \end{bmatrix}.$$

The algorithm is shown to compute a good approximation to the truncated SVD of the matrix $\widehat{A}$, especially when its singular values have so-called "low-rank-plus-shift" distribution [10]. Since the "low-rank" and "shift" respectively correspond to the relevant features and the noises of the underlying data, the matrices from many applications of our interests have this type of singular value distributions.

In (3), the algorithm first accesses the matrix D through *SpMM* (or *GEMM*) to compute $U_k^T D$, and then accesses it again to accumulate the results of *SpMM* and compute $\widehat{D}$. In practice, to reduce the large amount of memory needed to store the m-by-d dense vectors P, we incrementally update the SVD by adding a subset of the new columns D at a time. However, all the columns of D are still orthogonalized against U_k. In addition, the accumulated cost of computing the SVD of the matrices B and updating U_k and V_k could still be significant.

B. Randomized Algorithm

To reduce the cost of the updating algorithm, a *randomized algorithm* [6] applies the normalized power iterations to the matrix $\widehat{D}$ of (3) without explicitly forming $\widehat{D}$:

Alg. 2. Randomized Algorithm for Adding Columns

1. Generate ℓ Gaussian random vectors Q
2. Compute QR factorization of Q, $QR := \text{QR}(Q)$
   `for` $j = 1, 2, \ldots, s$ `do`
3. Approximate the matrix range, $P := (I - U_k U_k^T) D Q$
4. Compute QR factorization of P, $PR := \text{QR}(P)$
   `if` $j < s$ `then`
5. Approximate the matrix domain, $Q := D^T P$

6. Compute QR factorization of Q, $QR := \text{QR}(Q)$
   end if
end for

After the power iteration, the basis vectors $\widehat{P}$ and $\widehat{Q}$ are computed as in (4), but using P generated by the iteration. To further reduce the cost of solving the projected system, a smaller right projection subspace was also proposed where the basis vectors Q generated by the power iteration was used,

$$\widehat{Q} = \begin{bmatrix} V_k & 0 \\ 0 & Q \end{bmatrix}.$$

We refer to these two projection schemes as the randomized algorithms with one-sided and two-sided approximation, or as Rnd_1 and Rnd_2 in short, respectively. The respective $(k+\ell)$-by-$(k+d)$ and $(k+\ell)$-by-$(k+\ell)$ projected matrices $B \equiv \widehat{P}^T \widehat{A} \widehat{Q}$ are given by

$$B = \begin{bmatrix} \Sigma_k & U_k^T D \\ 0 & P^T D \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \Sigma_k & U_k^T DQ \\ 0 & R \end{bmatrix}. \qquad (5)$$

The smaller B of Rnd_2 leads to a lower computational cost. More importantly, however, if the result of DQ is saved at Step 3 of the above algorithm, the projected matrix B of Rnd_2 can be computed without an additional *SpMM*. On the other hand, to compute B, Rnd_1 requires the additional *SpMM* to multiply D with the vectors U_k and P.

When the updating algorithm of Section III-A incrementally adds ℓ columns at a time, it performs $\frac{d}{\ell}$ orthogonalization and projection. Hence, if the randomized algorithm converges in less than $\frac{d}{\ell}$ iterations, it has a lower cost of orthogonalization than the updating algorithm. Since the randomized algorithm typically requires only a couple of iterations, it could obtain significant speedups over the updating algorithm (see Section VII). However, while the updating algorithm only accesses the matrix D twice, the randomized algorithm accesses D $(2s-1)$ times over the s iterations. Though D is accessed only through *SpMM*, this data access often dominates the performance of the randomized algorithm.

IV. Sampling Algorithm

To lower the data access cost of the randomized algorithm, in this section, we integrate *sampling*. For instance, instead of iterating with the new data D, we may iterate with the row-sampled matrix $\widetilde{D}_r$ that contains only a subset of its rows,

$$\widetilde{D}_r = S_r C_r D, \qquad (6)$$

where the c-by-m matrix C_r samples the rows of the matrix D, and the c-by-c matrix S_r scales the sampled rows (i.e., $S_r C_r$ has a single nonzero entry on each row, $(S_r C_r)_{i,c_i} = s_{i,i}$, where c_i is the index of the i-th sampled row). The following algorithm generates our sampling matrix C:

Alg. 3. Algorithm to Sample rows:
1. Generate m-length probabilistic distribution p such that
   p_i is the probability that i-th row is sampled, and $\sum_{i=1}^{m} p_i = 1$
2. Compute probabilistic interval t such that $t_i = \sum_{j=1}^{i} p_j$
3. Sample c rows of D following the distribution p
   for $i = 1, 2, \ldots, c$ do
   3.1. Draw a uniformly distributed random number γ in the interval (0,1)
   3.2. Select the sampled row c_i such that $c_i = \arg\max_i (\gamma < t_i \mid i = 1, 2, \ldots, m)$
   if sampling without replacement and c_i has been previously selected then
   3.3. Draw a new γ and go to Step 3.2
   end if
end for

In this paper, we use the following two types of distributions p. The first is the uniform distribution (i.e., $p_i = \frac{1}{m}$) while the second uses the *leverage scores* such that $p_i = \frac{1}{k}\sum_{j=1}^{k} u_{i,j}^2$, where $u_{i,j}$ is the (i,j)-th entry of the current approximate left singular vectors U_k. Then, when sampled with replacements, we use the scaling matrix $S_r = \sqrt{\frac{m}{c}} I_c$, while without replacement, we let $S_r = \text{diag}(\frac{1}{\sqrt{cp_{c_1}}}, \ldots, \frac{1}{\sqrt{cp_{c_c}}})$.

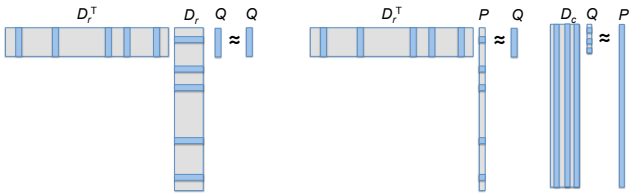
Though we could have used any sampling algorithm, we focused on these two distributions that are readily available without an additional data pass over D. Theoretical studies have been conducted on these two distributions including an upper bound on the approximation error of the sampled Gram matrix [11]. Since we iterate on the Gram matrix, this provides a good theoretical motivation for using these distributions. Finally, these two distributions have been used in many studies, and provide a baseline performance of our frameworks. If a more effective sampling, or sketching [12], scheme exists for a particular application, then it can be easily integrated in our framework. Our focus is to use these two basic distributions to sample our specific matrix $\widehat{D} = (I - U_k U_k^T) D$ and study their effectiveness for *updating* the SVD.

A. Row Sampling

We now describe our first randomized sampling framework to update the SVD. Given the row-sampling and scaling matrices C_r and S_r, we approximate the Gram matrix of $(I - U_k U_k^T)D$ using two row-sampled matrices $\widetilde{D}_r$ of (6) and $\widetilde{U}_k$, which is generated through the QR factorization of the row-sampled matrix $S_r C_r U_k$, i.e., $\widetilde{U}_k \widetilde{R} := \text{QR}(S_r C_r U_k)$. Then, we generate the right-projection subspace Q through power iterations on the approximate normal equation without explicitly forming the Gram matrix:

Alg. 4. Row-sampling for Adding Columns (Smp_1):
1. Sample and scale rows of the matrices D and U_k,
   $\widetilde{D}_r := S_r C_r D$ and $\widetilde{U}_k \widetilde{R} := \text{QR}(S_r C_r U_k)$
2. Generate right projection subspace Q
   by power iterating with Gram matrix of $(I - \widetilde{U}_k \widetilde{U}_k^T)\widetilde{D}_r$
   2.1. Generate ℓ Gaussian random vectors Q
   2.2. Compute QR factorization $QR := \text{QR}(Q)$
   for $j = 1, 2, \ldots, s-1$ do
   2.3. Approximate the matrix range, $Q := \widetilde{D}_r^T (I - \widetilde{U}_k \widetilde{U}_k^T) \widetilde{D}_r Q$
   2.4. Compute QR factorization, $QR := \text{QR}(Q)$
   end if
3. Generate left projection subspace,
   $P := (I - U_k U_k^T) DQ$
   $PR := \text{QR}(P)$
4. Generate projected matrix B of (5).

After the power iteration, the left-projection subspace P is computed through *SpMM* with the original matrix D (Step 3 of Alg. 4). To generate B of Rnd_1, we need one more *SpMM* to compute $D^T[U_k\ P]$. On the other hand, if we store the result



(a) Smp_1. (b) Smp_2.

Fig. 1. Illustration of two sampling approachs, where the regions colored in blue represent the sampled rows or columns.

	Rnd_1	Rnd_2	Smp_1	Smp_2
Matrix operation with D				
# of Sp/GEMM	s	s	$(s-1)\tau+1$	$(s-1)\tau$
# of Sp/GEMMt	s	s-1	$(s-1)\tau$	$(s-1)\tau+1$
Dense computation (flop count)				
Orth	$\ell^2 ms$	$\ell^2 ms$	$\ell^2((s-1)\tau+1)m$	$\ell^2((s-1)\tau+1)m$
SVD(B)	$(k+d)(k+\ell)^2$	$(k+\ell)^3$	$(k+\ell)^3$	$(k+\ell)^3$

Fig. 2. Complexities of algorithms to update SVD, s is the number of power iterations, m and d are the respective numbers of rows and columns in D, k is the rank of approximation, ℓ is the oversampling parameter, and τ is the sampling rate, e.g., $\tau = \frac{c}{m}$ and c is the number of sampled rows. We assume that the matrices are tall and skinny (i.e., $m \gg n, d$). When the updating algorithm incrementally adds ℓ columns at a time, it performs $\ell^2 m \frac{d}{\ell}$ flops for the orthogonalization.

of DQ at Step 3, then we can generate B of Rnd_2 without any additional *SpMM*. We refer to Alg. 4 as Smp_1, and use $\text{Smp}_{1,1}$ and $\text{Smp}_{1,2}$ to distinguish Smp_1 with the projection schemes of Rnd_1 and Rnd_2, respectively.

B. Row-column Sampling

Our second sampling framework approximates the results of *SpMM* with D and D^T using $\widetilde{D}_c$ and $\widetilde{D}_r$ that sample the columns and rows of D, respectively. Though we do not have the leverage scores for the columns of D, we can still, for example, sample the columns based on their norms (requiring a data pass over D) or using the uniform distribution p.

Alg. 5. Row-column sampling for Adding Columns (Smp_2):

1. Sample and scale the rows of the matrices D and U_k
   $\widetilde{D}_r := S_r C_r D$ and $\widetilde{U}_k \widetilde{R} := \text{QR}(S_r C_r U_k)$
2. Sample and scale the columns of the matrix D
   $\widetilde{D}_c := D C_c S_c$
3. Generate projection subspaces P and Q
   using the randomized algorithm on $(I - \widetilde{U}_k \widetilde{U}_k^T)\widetilde{D}$
   3.1. Generate ℓ Gaussian random vectors Q
   3.2. Compute QR factorization $QR := \text{QR}(Q)$
   `for` $j = 1, 2, \ldots, s-1$ `do`
   3.3. Approximate the matrix range, $P := (I - U_k U_k^T)\widetilde{D}_c(S_c C_c Q)$
   3.4. Compute QR factorization of P, $PR := \text{QR}(P)$
   `if` $j < s$ `then`
   3.5. Approximate the matrix domain, $Q := \widetilde{D}_r^T(I - \widetilde{U}_k \widetilde{U}_k^T)(S_r C_r P)$
   3.6. Compute QR factorization of Q, $QR := \text{QR}(Q)$
   `end if`
   `end for`
4. Generate projected matrix B of (5).

Since this approach uses power iteration to generate both the right- and left-projection subspaces P and Q, it does not require the extra *SpMM* that is needed by Smp_1 to generate P. However, we still need to perform *SpMM* with D to generate the projected matrix B. It also has the additional cost to orthogonalize P during the power iteration. We refer to this as Smp_2, and use $\text{Smp}_{2,1}$ and $\text{Smp}_{2,2}$ to refer to Smp_2 with the projection schemes of Rnd_1 and Rnd_2, respectively.

Fig. 1 illustrates these two sampling schemes, and Fig. 2 lists their computational and data access costs.

V. Randomization to add rows

In many applications, we have a good understanding of the data including how much data should be sampled. However, in some cases, we may fail to sample enough rows for the updated SVD to satisfy the desired accuracy. In such cases, we could discard the updated SVD, increase the sampling size, and recompute the updated SVD using the new set of the sampled rows. Instead, in this section, we look at updating the already-updated SVD using the additional sampled rows. Such a scheme is also attractive when all the sampled rows do not fit in the memory at once because it allows an incremental update of the SVD using only a subset of the sampled rows at a time, which fit in the memory.

Assuming that we have updated the SVD using Smp_1, we use the randomized algorithm to update the projection basis vectors P and Q by adding more sampled rows to $\widetilde{U}_k$ and $\widetilde{D}_r$. The algorithm is based on the power iteration on the Gram matrix, as shown below where $\overline{D}_r$ and $\overline{U}_k$ represent the new set of sampled rows, while $\overline{P}$ and $\overline{Q}$ are the new set of basis vectors to be generated.

Alg. 6. Randomized Algorithm for Adding Sampled Rows:

1. Sample more rows and generate $\overline{D}_r$ and $\overline{U}_k$
2. Generate ℓ Gaussian random vectors $\overline{P}$
3. Compute QR factorization of $\overline{P}$, $\overline{P}R := \text{QR}(\overline{P})$
   `for` $j = 1, 2, \ldots, s$ `do`
4. Approximate the matrix range,
   $\overline{Q} := \overline{D}_r^T(I - \overline{U}_k \overline{U}_k^T)\overline{P}$
   $\overline{Q} := (I - QQ^T)\overline{Q}$
5. Compute QR factorization, $\overline{Q}R := \text{QR}(\overline{Q})$, if requested
   `if` $j < s$ `then`
6. Approximate the matrix domain, $\overline{P} := (I - \overline{U}_k \overline{U}_k^T)\overline{D}_r \overline{Q}$
7. Compute QR factorization, $\overline{P}R := \text{QR}(\overline{P})$
   `end if`
   `end for`

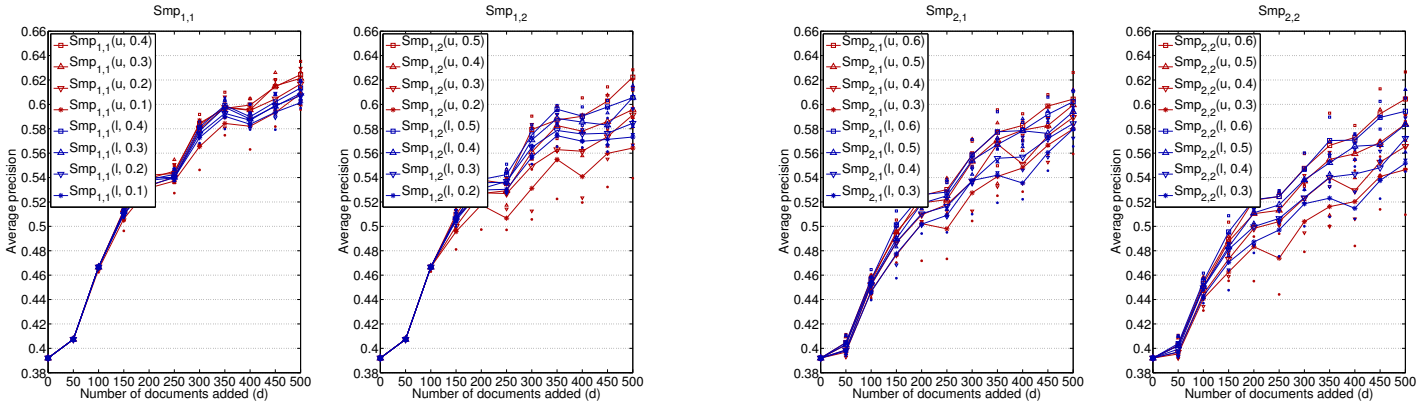
Then, Q is updated by compressing the new projection subspace $[Q\ \overline{Q}]$, where the projected matrix B is given by

$$B = \begin{pmatrix} I & 0 \\ \overline{P}^T \widetilde{D}_r Q & R^T \end{pmatrix}$$

since $\overline{P}$ and $\overline{Q}$ are orthogonal to $\overline{U}_k$ and Q, respectively. The main motivation of the above algorithm is to avoid accessing the previously sampled rows $\widetilde{D}_r$ that have been already compressed into a low-rank representation P and Q.

VI. Case Studies

A different application uses a different error measurement and requires a different approximation accuracy. In this section, we examine a few test cases to study the effectiveness of the sampling and randomized algorithms to update the truncated SVD. To this end, we focus on a powerful data analysis tool, the principal component analysis (PCA) [13]. In PCA,



(a) Smp_1. (b) Smp_2.

Fig. 3. Average 11-point interpolated precision with different sampling rate for `medline`, where the first argument of (u, τ) or (ℓ, τ) in the legend indicates that the uniform probabilistic distribution or the leverage score is used, respectively, and τ is the sample rate, $\frac{c}{m}$, while we fixed $n = 533$ and $s = 3$ ($m = 5735$). The line shows the mean precisions of ten runs, while the markers above and below the line show the highest and lowest precisions, respectively.

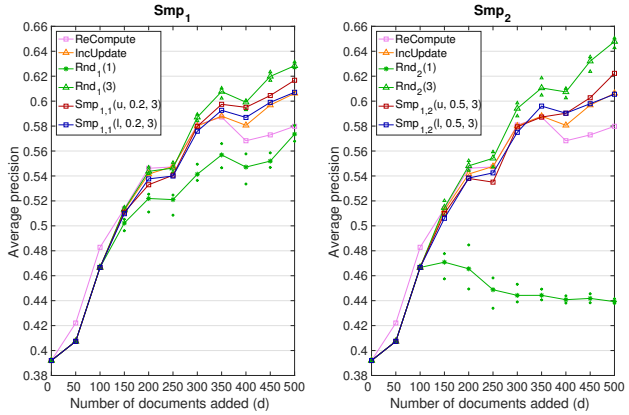


Fig. 4. Average 11-point interpolated precision for `medline` with Rnd(s) and Smp(ℓ or u, τ, s) where the first argument ℓ or u specifies either the leverage score or uniform distribution is used to sample the rows.

multidimensional data is projected onto a low-dimensional subspace given by the truncated SVD such that related items are close to each other in the low-dimensional subspace. Here, we examine three particular applications of PCA: LSI, data clustering or classification, and image processing. The matrices used for LSI and classification are sparse, while the matrices used for the clustering and image processing are dense. The results with the randomized and sampling algorithms are the mean of ten runs.

A. Sampling to Add Columns

We first investigate the required sampling rate (i.e., how much data needs to be sampled) to maintain the quality of the updated SVD. We also compare the effectiveness of different sampling schemes.

1) Latent Semantic Indexing: For text mining [14], Latent Semantic Indexing (LSI) [2] is an effective information retrieval tool since it can resolve the ambiguity due to the synonymy or polysemy, which are difficult to address using a traditional lexical-matching [15]. To study the effectiveness of the proposed sampling algorithms for LSI, we generated the term-document matrices using the Text to Matrix Generator (TMG) with the TREC dataset[1]. We then preprocessed the matrices using the `lxn.bpx` weighing scheme [16]. To compare the different combinations of sampling and projection schemes, Fig. 3 shows the average 11-point interpolated precisions [16] of the updated SVD for the `medline` dataset. Specifically, the figure shows the average precision when the randomized sampling was used for adding d new documents to the rank-30 approximation of the first 533 documents. The first sampling scheme Smp_1 obtained higher precisions than the second scheme Smp_2, while the first projection scheme was slightly more effective obtaining the higher precisions than the second scheme (e.g., $\text{Smp}_{1,1}$ was more effective than $\text{Smp}_{1,2}$). Then, Fig. 4 compares the results with the three previous algorithms: 1) recomputing SVD, 2) the updating algorithm (adding incremental of 500 documents at a time), and 3) the randomized algorithm without sampling. Overall, only about $20 \sim 50\%$ of the new data was needed to obtain the precisions that were equivalent to those obtained using the previous algorithms. We have observed similar results for the other datasets like `cranfield`.

2) Data Clustering and Classification: PCA has been successfully used to extract the underlying genetic structure of human populations [17], [18], [19]. To study the potential of the sampling algorithm, we used it to update the SVD, when a new population is incrementally added to the dataset from the HapMap project[2]. We randomly filled in the missing data with either -1, 0, or 1 with the probabilities based on the available information for the SNP. Fig. 5(a) shows the correlation coefficient of the resulting population cluster, which is computed using the MATLAB's k-mean algorithm

[1] http://scgroup20.ceid.upatras.gr:8000/tmg, http://ir.dcs.gla.ac.uk/resources
[2] http://hapmap.ncbi.nlm.nih.gov

	JPT+MEX	+ASW	+GIH	+CEU	+LWK	+CHB
Recompute	1.00	1.00	1.00	0.99	0.76	0.72
No update	1.00	0.81	0.59	0.67	0.56	0.47
Inc-Update	1.00	1.00	1.00	0.98	0.76	0.75
Rnd_2	1.00	1.00	1.00	0.99	0.76	0.73
$\text{Smp}_{1,2}(u)$	1.00	1.00	1.00	0.99	0.76	0.60
$\text{Smp}_{1,2}(\ell)$	1.00	1.00	1.00	0.99	0.76	0.71

(a) Population clustering where 83 African ancestry in south west USA (ASW), 88 Gujarati Indian in Houston (GIH), 165 European ancestry in Utah (CEU), 90 Luhya in Webuye, Kenya (LWK), and 84 Han Chinese in Beijing (CHB) 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). We used the fixed parameters $(s = 3, \tau = 0.9\%)$.

	crude+interest	+money-fx	+trade	+ship	+grain
Recompute	1.00	0.74	0.74	0.61	0.59
No update	1.00	0.62	0.42	0.44	0.39
Inc-Update	1.00	0.75	0.74	0.60	0.58
Rnd_2	1.00	0.75	0.75	0.62	0.60
$\text{Smp}_{1,2}(u)$	1.00	0.76	0.76	0.63	0.60
$\text{Smp}_{1,2}(\ell)$	1.00	0.75	0.75	0.62	0.60

(b) Document classification where there are 253, 190, 206, 251, 108, and 41 documents of `crude`, `interest`, `money-fx`, `trade`, `ship`, and `grain` categories, with $19,368$ terms. We used the fixed parameters $(s = 3, \tau = 25\%)$.

Fig. 5. Average correlation coefficients of clustering based on the five dominant singular vectors.

d	Recomp	Update	Rnd_1	Rnd_2	$\text{Smp}_{1,1}$	$\text{Smp}_{1,2}$
0	0.013	0.013	0.013	0.013	0.013	0.013
500	0.013	0.013	0.013	0.043	0.013	0.014
1000	0.013	0.013	0.014	0.085	0.013	0.019
1500	0.014	0.014	0.014	0.121	0.014	0.020
1700	0.014	0.014	0.014	0.140	0.015	0.024

(a) Approximation error norm $\|\widehat{A} - \widehat{U}_k \widehat{S}_k \widehat{V}_k\|_2 / \|A\|_2$.

Fig. 6. Results with a 2250-by-2250 aerial image from the USC-SIPI Image Database with $n = 500$, $k = 100$, and $(s, \tau) = (3, 10\%)$.

in the low-dimensional subspace given by the dominant left singular vectors. The correlation coefficient of 1.00 indicates the perfect clustering result, while a lower coefficient indicates a lower quality of the cluster. Similarly, Fig. 5(b) shows the correlation coefficients for the document classification as a new set of documents belonging to a different category is added to the dataset[3]. Especially for the population clustering, only a small fraction of the data is needed to obtain the coefficients that are equivalent to those obtained by recomputing the SVD or by the randomized algorithm without sampling.

3) Image Processing: The truncated SVD has been a valuable component in image processing because many digital images can be represented as low-rank matrices where the components associated with small singular values correspond to noises [20]. When these images are too large to fit in the memory at once, it becomes attractive to bring a part of the image into the memory and incrementally update the SVD. Updating SVD is also of interest in the applications where the images are continually updated [21], [22]. Fig. 6(a) shows the relative error norm, where only a couple of digits of accuracies are often needed for the image processing. Compared to Rnd_2, Rnd_1 was more effective maintaining the accuracy. The sampling techniques obtained equivalent accuracies using only 10% of the data. We have observed similar results using other images from different applications. For this particular image, in some cases, the sampling techniques could even improve the accuracy of the approximation.

[3]http://www.cs.umb.edu/~smimarog/textmining/datasets

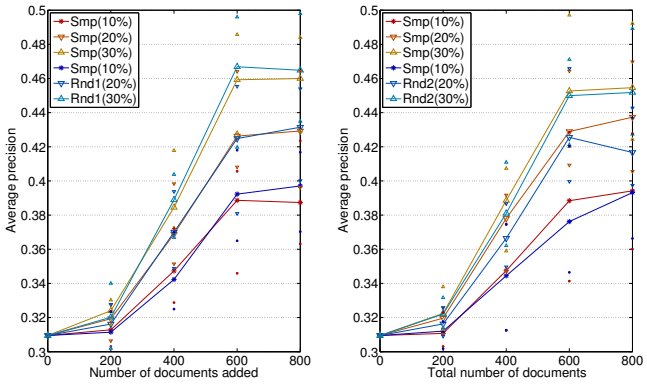


Fig. 7. Average 11-point interpolated precision for `medline` when adding sampled rows with $(n, k) = (233, 50)$.

B. Randomization to Add Rows

We now study the randomized algorithm to add more sampled rows. For the experiments in Fig. 7, we first updated the SVD of the first 233 documents of `medline` by sampling 10% of the rows (i.e, $d = 200 \sim 800$). Then we used the randomized algorithm to update the SVD by adding more sampled rows. The figure illustrates that without accessing the previously compressed sampled rows, both the randomized algorithms Rnd_1 and Rnd_2 could obtain the precisions close to the sampling scheme that computes the updated SVD using all of the sampled rows at once.

VII. Performance Results

A. Experimental Setups

For our performance studies in this section, we focus on Smp_1 combined with Rnd_2 (i.e., $\text{Smp}_{1,2}$) because compared with Smp_2, Smp_1 was more effective maintaining the quality of the SVD in Section VI. Smp_1 also accesses only the row-sampled matrix $\widetilde{D}_r$, while Smp_2 requires both the row and column sampled matrices. Our sampling algorithm greatly reduces both the storage and time complexities of updating the SVD. Hence, to study the effects of the sampling on the performance of updating SVD, in this paper, we use a single compute node as our testbed. The performance of the randomized algorithm without sampling on a hybrid cluster was studied in [6]. On a distributed-memory computer, sampling the matrix could lead to a greater performance improvement since it allows us to use a fewer compute nodes, reducing the inter-node communication cost.

We assumed that the compressed basis vectors P and Q fit in the GPU memory, where these vectors were orthonormalized using the vender-optimized BLAS-3 kernels: specifically, to orthogonalize the basis vectors, we used the Cholesky QR (CholQR) [23], which is communication optimal [24] and was

	GPU-resident		CPU-resident		
	('N', D^T)	('T', D)	('N', D^T)	('T', D)	in-place
multiply with D	0.34	0.33	0.70	0.70	0.72
multiply with D^T	0.52	2.40	2.10	0.34	0.52
Total	0.95	2.82	2.96	1.15	1.35

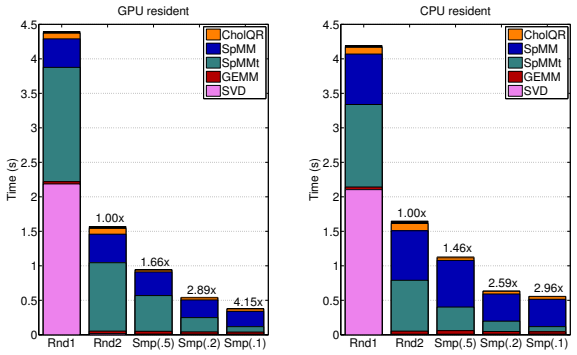
Fig. 8. Performance of $\text{Smp}_{1,2}(\tau = .5, s = 3)$ using different *SpMM* implementations with D^T: ('N', D^T) explicitly stores the matrix D^T, ('T', D) transposes D stored in the CSR format, and in-place performs *SpMM* with $\widetilde{D}_r$ using D stored in the CSR format, ($n = 5,000$, $d = 5,000$).

numerically stable in our experiments (the test matrices had the low-rank-plus-shift singular value distributions but they were not ill-conditioned). For our test matrices, we considered both dense and sparse matrix D. To store these matrices in the core memory, we used the standard LAPACK dense format in the column major order, or the Compressed Sparse Row (CSR) format. We then investigated two approaches to apply the matrix multiplication: either 1) using CUBLAS or CuSPARSE on the GPU, or 2) using dense or sparse BLAS of threaded MKL on the CPUs. To perform *SpMM* on a GPU, we designed two implementations: GPU-resident or non-GPU-resident, where the whole matrix fits in the GPU memory at once, or only a part of the matrix is copied from the CPU into the GPU memory at a time, respectively. Similarly, to perform *SpMM* on the CPUs, we used two implementations: CPU-resident or non-CPU-resident, where the whole matrix is resident in the CPU memory or only a part of the matrix is read from an external disk at a time, respectively.

All of our experiments were conducted using ten-core Intel Xeon E5-2650 (Haswell) CPUs and one NVIDIA K80 GPU. The CPU and GPU memories have achievable bandwidths of 40 GB/s and 205 GB/s, respectively, and the CPUs and the GPU are connected through the PCI-E x16 Gen3 PCI interface with 10 GB/s. Two external storages were available on this compute node: a HP Hard Drive (HD) with 200MB/s peak bandwidth, and a OCZ Solid State Drive (SSD) with 160MB/s. We compiled the code using `g++` (GCC) version 4.4.7 and `nvcc` of CUDA version 8.0 with the optimization flags `-O3`, and linked it with the threaded MKL version 2016.0.109. For our performance studies, we focus on two matrices: 1) the sparse movie-by-user `netflix` matrix with $480,189$ movies, $17,770$ users, and about 209 nonzero entries per row, and 2) dense random matrices of different dimensions. Unless otherwise stated, we performed three power iterations that were shown to be enough in Section VI.

B. In-core Sampling to Add Columns

In this section, we study the performance of the sampling algorithm using the GPU-resident or CPU-resident *SpMM*. The performance of the randomized algorithm is often dominated by *SpMM*. Though we rely on the vendor-optimized *SpMM*, the kernel may not be fully optimized for the particular shapes of our matrices. For instance, in Fig. 8, we compare the performance of *SpMM* with D^T of `netflix` matrix by using D stored in the CSR format or by explicitly storing D^T in a separate CSR format. In many cases, explicitly storing D^T benefited the performance on the GPU, but not on the CPU.



(a) $\text{Smp}_{1,2}$ using CuSPARSE or threaded MKL.

	Number of iterations					
	3	4	5	6	7	8
Rnd_2	1.6 (1.00)	2.2 (1.00)	2.8 (1.00)	3.5 (1.00)	4.1 (1.00)	4.8 (1.00)
Smp(0.5)						
in-place	1.3 (1.27)	1.6 (1.37)	2.0 (1.40)	2.4 (1.42)	2.8 (1.44)	3.2 (1.47)
explicit	1.1 (1.49)	1.4 (1.58)	1.7 (1.66)	2.0 (1.75)	2.3 (1.79)	2.6 (1.80)
Smp(0.2)						
in-place	0.8 (2.05)	1.0 (2.26)	1.2 (2.43)	1.4 (2.54)	1.5 (2.63)	1.8 (2.72)
explicit	0.6 (2.53)	0.8 (2.91)	0.9 (3.13)	1.0 (3.50)	1.1 (3.64)	1.3 (3.72)

(b) Time in second (speedups) with the number of iterations (CPU-resident).

Fig. 9. Performance of sampling algorithm, $\text{Smp}(\tau)$, when adding new users to `netflix` matrix, ($n = 5,000$, $d = 5,000$), using different sampling rates τ but fixed parameters ($k = 30$, $\ell = 30$, and $s = 3$).

Hence, for the rest of this section, we store both D and D^T on the GPU, while we only store D on the CPUs. On the CPUs, we used *SpMM* of MKL whose interface supports *SpMM* with the sampled matrix $\widetilde{D}_r$ using the original matrix D stored in the CSR format. Hence, only the matrix D needs to be stored in the memory. This in-place *SpMM* adds a small overhead since the rows of $\widetilde{D}_r$ are stored in noncontiguous locations, leading to more irregular memory accesses. However, it avoids the additional storage for $\widetilde{D}_r$.

In Fig. 9(a), we study the effects of sampling on the performance of the randomized algorithms using different sampling rates $\tau = \frac{c}{m}$. First, we see that the time spent computing the SVD of the projected matrix B could become significant in the performance of Rnd_1, whereas Rnd_2 reduces this bottleneck, significantly. For this particular setup, though not shown in the figure, the updating algorithm [7] (adding 60 user columns at a time) needed about 8.9 seconds to add the $5,000$ user columns (7.5 seconds for Orth). Hence, Rnd_2 obtained the speedup of about $5.7\times$ over the updating algorithm. Then, $\text{Smp}_{1,2}$ reduces both the computation and data traffic needed by Rnd_2 for *SpMM* by a factor of $\frac{2s-1}{2(s-1)\tau+1}$ (see Fig. 2). Hence, when the performance of Rnd_2 is dominated by *SpMM*, with $s = 3$, we expect the speedups of about $1.67\times$, $2.78\times$, and $3.57\times$ using $\text{Smp}_{1,2}$ with $\tau = 0.5, 0.2$, and 0.1, respectively. Fig. 9(a) shows that our implementation obtains the speedups close to these expectations.

Fig. 9(b) then shows the speedups obtained using the sampling technique with an increasing number of iteration counts. Using $\text{Smp}_{1,2}$ with $\tau = 0.5$, the reduction in the data access, and hence the expected speedups, are $1.67\times$, $1.75\times$, $1.80\times$,

$n/100$		10	25	50	75	100
COPY	(s)	0.03	0.07	0.13	0.19	0.25
	(GB/s)	7.78	7.61	7.91	7.99	8.11
GEMM('N', D)	(s)	0.01	0.02	0.03	0.04	0.05
	(Gflp/s)	608.2	695.1	926.1	943.7	911.5
GEMM('T', D)	(s)	0.01	0.02	0.03	0.05	0.07
	(Gflp/s)	366.5	526.2	767.7	781.8	759.0
GEMM('N', D^T)	(s)	0.01	0.02	0.03	0.04	0.05
	(Gflp/s)	384.1	594.9	762.8	836.3	877.1

(a) Non-GPU-resident.

$n/100$		10	25	50	75	100
SSD						
READ (contig)	(s)	1.2	2.7	5.1	7.9	10.6
	(MB/s)	177.1	185.9	197.5	188.7	193.4
READ (by row)	(s)	1.3	3.2	6.4	8.0	10.8
	(MB/s)	155.9	156.8	156.9	187.6	184.7
READ (noncont)	(s)	14.0	13.0	10.9	9.1	9.2
	(MB/s)	14.3	38.5	91.9	165.5	218.1
HD						
READ (contig)	(s)	1.29	3.18	5.02	7.09	9.13
	(MB/s)	155.6	157.3	199.4	211.5	219.1
READ (by-row)	(s)	1.28	3.20	5.62	6.90	10.2
	(MB/s)	155.7	156.4	177.9	217.4	196.8
READ (noncont)	(s)	10.0	11.3	12.2	12.0	11.9
	(MB/s)	20.0	44.2	81.7	125.4	167.5
GEMM('N', D)	(s)	0.02	0.05	0.10	0.18	0.23
	(Gflp/s)	232.1	260.4	248.5	272.4	275.4
GEMM('T', D)	(s)	0.03	0.05	0.09	0.13	0.17
	(Gflp/s)	184.2	240.4	280.6	296.1	297.7
GEMM('N', D^T)	(s)	0.03	0.06	0.10	0.17	0.20
	(Gflp/s)	185.6	216.6	252.4	226.1	250.0

(b) Non-CPU-resident.

Fig. 10. Time in seconds (average of five runs) for GPU and I/O data transfer ($m = 25,000$, $n + d = 10,000$, $\ell = 100$).

	SpMM(D)	*SpMM*(D^T)	total	speedup
Recompute	1.02 / 3.73	2.19 / 2.73	3.31 / 6.48	1.0 / 1.0
Rnd_2	0.25 / 0.83	0.55 / 0.63	1.00 / 1.50	3.3 / 4.3
$\text{Smp}_{1,2}$	0.17 / 0.44	0.15 / 0.04	0.34 / 0.50	10.3 / 13.0

Fig. 11. Performance with non-GPU-persident *SpMM* when adding 25% of the columns, assuming 25% of D fits in the main memory with `netflix` / dense matrices (i.e., $n = 7,500$, $d = 2,500$, and $s = 3$).

	SpMM(D)	*SpMM*(D^T)	total	speedup
Recomp	937.2 / 941.5	620.4 / 622.3	1558.1 / 1564.0	1.0 / 1.0
Rnd_2	231.2 / 232.4	154.3 / 155.0	385.7 / 387.6	4.0 / 4.0
$\text{Smp}_{1,2}$	97.1 / 96.9	0.1 / 0.1	97.2 / 97.1	16.0 / 16.1

(a) `netflix` matrix with local HD / SSD.

	GEMM(D)	*GEMM*(D^T)	total	speedup
Recomp	139.7 / 158.1	101.0 / 105.4	240.9 / 263.4	1.0 / 1.0
Rnd_2	146.6 / 169.9	110.0 / 116.9	257.0 / 286.8	0.9 / 0.9
$\text{Smp}_{1,2}$	38.6 / 61.4	0.1 / 0.1	38.7 / 61.5	6.2 / 4.3

(b) dense matrix with local HD / SSD.

	GEMM(D)	*GEMM*(D^T)	total	speedup
Rnd_2	37.6	26.0	63.7	3.8×
$\text{Smp}_{1,2}$	24.2	0.1	24.3	9.9×

(c) dense matrix, D is stored in a separate file, with local HD.

Fig. 12. Performance with non-CPU-resident *SpMM* when adding 25% of columns, and assuming 25% of D fits in the main memory (i.e., $n = 7,500$, $d = 2,500$, and $s = 3$).

1.83×, 1.86×, 1.88× when $s = 3, 4, \ldots, 8$, respectively, while with $\tau = 0.2$, the respective reductions are 2.78×, 3.18×, 3.46×, 3.67×, 3.82×, and 3.95×. Due to the overhead associated with the irregular memory accesses, the sampling algorithm obtained smaller speedups using the in-place *SpMM* than those when the sampled matrix $\widetilde{D}_r$ is explicitly stored. The sampling algorithm obtained the speedups close to the expectations when $\widetilde{D}_r$ is explicitly stored in the memory.

C. Out-of-core Sampling to Add Columns

We now study the performance with the non-GPU-resident or non-CPU-resident *SpMM*, where the matrix does not fit in either the GPU or CPU memory, respectively. To perform the out-of-core *SpMM*, we transfer a block row of the matrix into the core memory at a time and perform the partial *SpMM* with the block row. For this, we first focused on dense matrices and profiled the bandwidth for transferring the matrix into the memory, and compared it with the performance of the required matrix multiplication. Fig. 10(a) shows the observed bandwidths when copying a 25,000-by-n block row from the CPU memory to the GPU memory for different numbers of columns, n. We observed the bandwidth of around 8.2 GB/s, while GEMM with D achieved the performance of about 920∼735 Gflop/s, and when multiplying with D^T, it reached about 889∼756 Gflop/s when D^T is explicitly stored, and 762∼684 Gflop/s when D is transposed on the fly. Overall, the data transfer took about 3× more time compared with the required computation.

Next, Fig. 10(b) shows the data-transfer rates between the CPU and external disk, where the dense matrix is stored in the row major order in a binary file on the disk. In the table, for "contig," we read the matrix, which is stored contiguously in the file, at once (i.e., a single call to `freed`), while for "by-row," the matrix is still stored contiguously, but we read the matrix one row at a time. Finally, for "noncont," the total of 10^4 columns are stored contiguously, but we read only the last n columns of these columns. Hence, for "noncont," we read the columns whose rows are not stored in the contiguous locations (each row is read using `fseek` followed by `fread`). We see that both "contig" and "by-row" obtained near-peak bandwidth, while "noncont" added significant overhead reading the noncontiguous data in the file, and lead to much lower bandwidth. We have also tried storing the matrix in the column major order. This allows us to access the subset of the columns in the contiguous locations. However, the sampled matrix must be read one matrix element at a time.

We now show the performance of the randomized and sampling algorithms using the out-of-core *SpMM* or *GEMM*. For these experiments, we split both the `netflix` and a 10^5-by-10^4 dense matrix such that A and D have 75% and 25% of the columns, respectively. We further assumed that 25% of D fits in the core memory. Hence, to perform *SpMM* with D for the randomized algorithm, we copy 25% of its rows into the memory at a time. On the other hand, when recomputing the SVD, for *SpMM* with the matrix $\widehat{A} = [A\ D]$, we bring in 12.5% of the matrix rows into the core memory at a time. In Fig. 12, we present the performance of the non-GPU-resident algorithms. Since we used the CSR format to store the sparse matrix in the CPU memory, we can directly access each block row of both the sparse and dense matrices on the CPU, and copy it to the GPU. However, since the whole matrix does not fit in the GPU memory, for both recomputing

the SVD and running the randomized algorithm, we must copy the matrix from the CPU memory for each *SpMM*. The randomized algorithm performs *SpMM* with D that is 25% of the matrix A. Therefore, the randomized algorithm reduces the amount of the data copy to the GPU by a factor of $4\times$, and was expected to obtain the speedup of about $4\times$ over recomputing the SVD. Then, the sampling algorithm samples 25% of D, which stays in the GPU memory during the power iterations, and then copies the whole D once into the GPU memory for the projection at the end. Hence, compared with Rand_2, $\text{Smp}_{1,2}$ reduces the amount of data copy by a factor of $2.5\times$, and was expected to obtain the speedups of about $2.5\times$. Our performance results confirm these expectations. In these experiments, we used a relatively large sampling rate (i.e., $\tau = 25\%$). The benefit of sampling is expected to increase with a smaller sampling rate.

Fig. 12(a) shows the performance with the non-CPU-resident *SpMM* for the `netflix` matrix. For the sparse matrix, each nonzero entry $a_{i,j}$ of the matrix is stored as an $(i, j, a_{i,j})$ triplet in an ASCII file. These nonzero entries may be stored in any order. Therefore, to read a block row of the matrix into the CPU memory, we scan the entire file for the row block.[4] For recomputing the SVD, only 12.5% of the matrix $[A, D]$ fits in the CPU memory at a time. Hence, for each *SpMM*, we need to scan the file sixteen times. On the other hand, 25% of D fits in the memory, and the randomized algorithm requires scanning the file only four times for each *SpMM*. Hence, we expect a speedup of four using the randomized algorithm. The sampling scheme samples 25% of the rows in the matrix D, and hence it scans the file once to sample the file, and then after the three power iterations with the sampled matrix in the memory, we read the file four times to perform *SpMM* to compute the projection space. Recomputing SVD requires reading the file 80 times over the three power iterations, while the randomized algorithm scan the file total of 20 times. Hence, using the sampling algorithm, which reads the file five times, we expect the speedups of about 16 and 4 over recomputing the SVD and the randomized algorithm, respectively. We see these speedups in Fig. 12(a).

For the non-CPU-resident *GEMM*, the dense matrix is stored by rows in a binary file, and hence, we can directly read each row of any submatrix.[5] Compared with recomputing the SVD that accesses the matrix $[A\ D]$, the randomized algorithm reads only D. Since D contains only 25% of the columns of $[A\ D]$, we expected the speedups of about four. However, this was not the case due to the different bandwidth obtained by each algorithm. For recomputing the SVD, we read the matrix stored contiguously in the file, and as seen in Fig. 10(b), we reached the near-peak bandwidth (about 160 MB/s). On the other hand, the randomized algorithm accesses the last 25% of the columns, where each row is stored in the contiguous location but separated from the previous row by 75% of the columns in the row. In many cases, compared with recomputing the SVD, the randomized algorithm obtained much lower bandwidth (about 30∼35 MB/s). As a result, as shown in Fig. 12(b), the randomized algorithm could not improve the performance of recomputing the SVD. When sampling the rows of D, we obtained even lower bandwidth than the randomized algorithm, especially on the HD (20 MB/s on HD and 40 MB/s on SSD). However, since the sampling algorithm accesses the file only five times (once to sample, and four more times to compute the projection), it was able to improve the performance. Finally, in Fig. 12(c), we show the performance where the matrices D and A are stored in separate files. Now, the randomized algorithm could obtain a near-peak bandwidth (around 170 MB/s) and achieve the expected speedup over recomputing the SVD. On the other hand, the speedup obtained by the sampling algorithm over the randomized algorithm was still smaller than the expected speedup of four due to the low bandwidth obtained when sampling the rows (around 40 MB/s). These experiments demonstrate that though sampling improves the performance, the performance may be further improved through the hardware supports for the random memory access.

[4]We scan the file twice (once to count the number of nonzeros and then to copy them to the memory), but we count them as one scan.

[5]For our non-CPU-resident performance studies, to avoid the I/O cache-effects, we read a separate file, storing the same dense matrix, for each *GEMM*.

		τ_2 10%	20%	30%	40%	50%
GPU-resident						
Resample		0.05	0.09	0.16	0.21	0.26
Random	τ_1 =10%	x	0.06	0.11	0.17	0.22
	20%	x	x	0.06	0.12	0.16
	30%	x	x	x	0.07	0.10
	40%	x	x	x	x	0.06
non-GPU-resident, $\tau_1 = \tau_2 - 10\%$						
Resample		0.05	0.15	0.25	0.31	0.48
Random		x	0.06	0.07	0.06	0.06

Fig. 13. Performance comparison of resampling and randomized algorithms to add sampled rows of dense matrix, where τ_1 is the sampling rate of the number of rows that have been previously compressed, and τ_2 is the new sampling rate after the new rows were added ($m = 10^4$, $n = 2,500$, $d = 7,500$).

D. Randomization to Add Rows

Fig. 13 compares the performance of updating the SVD with all the sampled rows at once with that of incrementally updating with a subset of the sampled rows at a time. First, in the top of Fig. 13, we show the results with the GPU-resident *SpMM*, and hence once we copy the whole matrix into the GPU memory, it stays resident in the memory. For these experiments, we have already compressed c_1 sampled rows (i.e., $\tau_1 = \frac{c_1}{m}$), but we now want to increase the number of the sampled rows to be c_2 (i.e., $\tau_2 = \frac{c_2}{m}$). The randomized algorithm avoids accessing the previously compressed sampled rows, and we see that when the data access dominates the performance, the randomized algorithm could lead to significant performance improvement. In bottom of Fig. 13, we show similar results with non-GPU-resident *SpMM*. Here, we assumed that only 10% of the matrix D fits in the memory. Hence, to recompute the updated SVD using all of the sampled rows, we need to copy the matrix to the GPU memory for each

SpMM. On the other hand, since the new set of the sampled rows fit in the GPU memory, the randomized algorithm need to copy the sampled rows only once into the GPU memory.

VIII. Conclusion

In this paper, we studied two data sampling techniques to reduce the data access cost of the randomized algorithm to update a truncated SVD. Our case studies have shown that only a small fraction of the data may be needed to maintain the quality of the SVD. Our experimental results on multicore CPUs with an NVIDIA GPU demonstrated the potential of the sampling algorithm to improve the performance of the randomized algorithm, especially when the data access dominates the performance. Though we often have knowledge about the data (e.g., how much data needs to be sampled), we may fail to sample enough data to obtain the required accuracy. To address such a case, we also studied an updating scheme to add additional sampled rows without accessing the previously-compressed rows. We demonstrated the performance benefit of such algorithms when the data access is expensive. Theoretical results on the updating algorithm [10] and the sampling algorithms to approximate the matrix multiplication exist [11]. We would like to investigate if such studies can be extended for sampling to update SVD.

Our focus was on updating SVD. Such updating schemes are attractive for the applications where the data access is prohibitively expensive (e.g., the whole matrix does not fit in the main memory, or the whole dataset is not available at once because a part of the dataset is still to be generated or has been deleted). A few previous studies have sampled the matrices to compute (rather than update) the SVD [25], [26]. Computing the SVD would require at least one pass over the entire dataset, while our updating scheme reads only the new data to be used for updating the SVD, while the rest of the data is read only in the compressed form. Nevertheless, we would like to compare, and potentially combine, our approach with these previous approaches. There also exist many randomized sampling schemes beside the Gaussian randomization and the sampling schemes used in this paper. We plan to use larger datasets from different applications, potentially on a hybrid CPU/GPU cluster, for studying the effectiveness of our algorithms compared or combined with these other schemes and also with the original updating algorithm (which has a greater local computation cost but reads the data only once).

Acknowledgments

This research was supported in part by the National Science Foundation (NSF) OAC Award number 1708299.

References

[1] G. Golub and C. van Loan, *Matrix Computations*, 4th ed. The Johns Hopkins University Press, 2012.

[2] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Amer. Soc. Info. Sci.*, vol. 41, pp. 391–407, 1990.

[3] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Analysis of recommendation algorithms for e-commerce," in *Proceedings of the 2nd ACM Conference on Electronic Commerce*, 2000, pp. 158–167.

[4] P. Paschou, E. Ziv, E. Burchard, S. Choudhry, W. R.-Cintron, M. Mahoney, and P. Drineas, "PCA-correlated SNPs for structure identification in worldwide human populations," *PLoS Genetics*, vol. 3, pp. 1672–1686, 2007.

[5] I. Karasalo, "Estimating the covariance matrix by signal subspace averaging," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 8–12, Feb. 1986.

[6] I. Yamazaki, J. Kurzak, P. Luszczek, and 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. 59:1–59:12.

[7] H. Zha and H. Simon, "On updating problems in latent semantic indexing," *SIAM J. Sci. Comput.*, vol. 21, pp. 782–791, 2006.

[8] M. Berry, S. Dumais, and G. O'Brien, "Using linear algebra for intelligent information retrieval," *SIAM Rev.*, vol. 37, pp. 573–595, 1995.

[9] N. Halko, P. Martinsson, and J. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Rev.*, vol. 53, pp. 217–288, 2011.

[10] H. Zha and Z. Zhang, "Matrices with low-rank-plus-shift structure: partial SVD and latent semantic indexing," *SIAM J. Matrix Anal. Appl.*, vol. 21, pp. 522–536, 1999.

[11] P. Drineas, R. Kannan, and M. Mahoney, "Fast Monte Carlo algorithms for matrices I: Approximating matrix multiplication," *SIAM J. Comput.*, vol. 36, pp. 132–157, 2004.

[12] D. P. Woodruff, "Sketching as a tool for numerical linear algebra," *CoRR*, vol. abs/1411.4357, 2014.

[13] C. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

[14] G. Salton and M. McGill, *Introduction to modern information retrieval*. McGraw-Hill, 1983.

[15] R. Krovetz and W. B. Croft, "Lexical ambiguity and information retrieval," *ACM Trans. Inf. Syst.*, vol. 10, pp. 115–141, 1992.

[16] T. Kolda and D. O'Leary, "A semidiscrete matrix decomposition for latent semantic indexing information retrieval," *ACM Trans. Inf. Syst.*, vol. 16, pp. 322–346, 1998.

[17] P. Menozzi, A. Piazza, and L. C.-Sforza, "Synthetic maps of human gene frequencies in Europeans," *Science*, vol. 201, pp. 786–792, 1978.

[18] N. Patterson, A. Price, and D. Reich, "Population structure and eigenanalysis," *PLoS Genet.*, vol. e190, 2006.

[19] A. Price, N. Patterson, R. Plenge, M. Weinblatt, N. Shadick, and D. Reich, "Principal components analysis corrects for stratification in genome-wide association studies," *Nat. Genet.*, vol. 38, pp. 904–909, 2006.

[20] W. Pratt, *Digital image processing*. Wiley-Interscience, 1978.

[21] G. Zientara, L. Panych, and F. Jolesz, "Dynamically adaptive MRI with encoding by singular value decomposition," *Magnetic Resonance in Medicine*, vol. 32, pp. 268–274, 1994.

[22] E. Drinea, P. Drineas, and P. Huggins, "A randomized singular value decomposition algorithm for image processing applications," 2001.

[23] A. Stathopoulos and K. Wu, "A block orthogonalization procedure with constant synchronization requirements," *SIAM J. Sci. Comput.*, vol. 23, pp. 2165–2182, 2002.

[24] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM Journal on Scientific Computing*, vol. 34, 2012.

[25] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay, "Clustering in large graphs and matrices," in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999, pp. 291–299.

[26] P. Drineas, R. Kannan, and M. Mahoney, "Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix," *SIAM J. Comput.*, vol. 36, pp. 158–183, 2006.