Block Randomized Singular Value Decomposition on GPUs

Yuechao LU†(a), Yasuyuki MATSUSHITA†, Nonmembers, and Fumihiko INO†, Member

SUMMARY Fast computation of singular value decomposition (SVD) is of great interest in various machine learning tasks. Recently, SVD methods based on randomized linear algebra have shown significant speedup in this regime. For processing large-scale data, computing systems with accelerators like GPUs have become the mainstream approach. In those systems, access to the input data dominates the overall process time; therefore, it is needed to design an out-of-core algorithm to dispatch the computation into accelerators. This paper proposes an accurate two-pass randomized SVD, named block randomized SVD (BRSVD), designed for matrices with a slow-decay singular spectrum that is often observed in image data. BRSVD fully utilizes the power of modern computing system architectures and efficiently processes large-scale data in a parallel and out-of-core fashion. Our experiments show that BRSVD effectively moves the performance bottleneck from data transfer to computation, so that outperforms existing randomized SVD methods in terms of speed with retaining similar accuracy.

key words: randomized singular value decomposition, GPU

1. Introduction

Singular value decomposition (SVD) is an essential tool in computer vision, machine learning, data analysis, and various other scientific computing. Studies on improving the performance and numerical stability of SVD have been ongoing ever since its advent [2]–[5], and have been successfully applied to applications such as principal component analysis (PCA) [6], [7]. Recently, a randomized SVD (RSVD) method has been proposed to further accelerate SVD by exploiting the low-rank structure of data [1], [8] and now appears a method of choice for fast approximate SVD computation.

While SVD has been made efficient in terms of its computational complexity based on these findings, the transition of modern computing architectures, such as graphics processing units (GPUs), allow us to develop even faster methods by taking advantage of high parallelism. Nevertheless, to fully benefit from the modern computing architectures, there are a few major challenges. Although computers’ arithmetic operations are becoming ever efficient, communication between memory hierarchies or through networks is emerging as the bottleneck for a lot of applications in distributed memory systems equipped with accelerators [9]–[11]. The gap between communication cost and computational cost is expected to increase, where arithmetic operations are fast and highly parallelized but data communication remains slow [12]. Note that the communication refers to not only the data transfer between computing nodes but also the data transfer of CPUs/accelerators to their memory. A series of communication avoiding algorithms have been proposed to tackle this gap [13]–[15]. These studies aim to redesign linear algebraic algorithms to reduce data communication among memory hierarchies.

For fast SVD computation, this trend requires a locality-aware method with less access to the input data. Especially, for large-scale data, the traditional SVD computation cannot fully benefit from a fast level-3 basic linear algebra subprograms (BLAS3) computation, or its main computational kernel known as general matrix-matrix product (GEMM), due to that (1) the data may not fit in a single memory space, (2) its computation pattern includes vast data accesses, and (3) communication between distinct memory hierarchy levels [16]. As for RSVD, input data will be accessed for 2q + 2 times, where q denotes the number of power iterations [1], [17], to attain a high accuracy approximation. Single-pass algorithms [18], [19], also called streaming algorithms, are proposed, which access the target with “one touch.” Mostly, they include iterations to construct a sketch [1], [18], [19], which have data dependency and is difficult be made in parallel. Furthermore, there exists an accuracy-performance trade-off in these algorithms [18], [19].

To our best knowledge, few works have been done to accelerate single-pass RSVD on GPUs. Regarding accelerating multi-pass RSVD [1] on GPUs, the computations of RSVD are mainly matrix-matrix multiplication. It means that implementing RSVD on multi-core CPUs or GPUs is relatively easier than implementing deterministic SVD [5]. Yamazaki et al. [20] proposed exploiting random sampling to update partial SVD on a hybrid CPU/GPU cluster. Their work showed that a random sampling algorithm achieved a speedup of up to 14.1× compared with a standard deterministic SVD implementation on multi-core CPUs. They assumed that GPU memory can hold all the working data. Voronin et al. proposed a comprehensive randomized linear algebra library named RSVDpack [8]. While effective, their GPU implementation is in-memory, and efficient computation can only be achieved when the data fits in the space of GPU memory.

This study considers redesigning the RSVD algorithm...
especially for large matrices that do not fit in the GPU memory and a limited CPU-GPU communication bandwidth. We propose a two-pass RSVD algorithm named block randomized SVD (BRSVD), which accesses the input data only twice in the whole computation. Similar to the GPU-only strategy [21], BRSVD uses GPUs for all computations which fully utilizes the power of accelerators and efficiently processes data without burdening the host CPU. BRSVD decomposes the original power method into independent block executions to reduce access to the target matrix. Different from the previous works [8], [20], our proposed out-of-core algorithm frees the memory capacity limitation on the input data. Furthermore, BRSVD decomposes the original power method into independent block executions to reduce the communication between CPUs and GPUs.

We compare the efficiency with an in-memory (i.e., all the working data can be held on the GPU memory) implementation, which is the performance upper-bound of RSVD. For large-scale data, BRSVD achieves a significant speedup in comparison to the original algorithm. We then assess the accuracy of the proposed method using both synthetic and real data and compare with existing algorithms. Our experiment shows that with a moderate partition size of the input data, BRSVD gives a close approximation to the original algorithm. The empirical results also indicate that the proposed algorithm outperforms the single-pass algorithm in terms of accuracy.

2. Preliminaries

RSVD has been made popular by Halko et al.’s work [1] built upon the previous studies on randomized linear algebra [4], [17], [22]–[24]. The randomization approach outperformed classical deterministic SVD methods in terms of speed while maintaining equivalent accuracy and robustness.

As described in [1], given a matrix $A \in \mathbb{R}^{m \times n}$, an orthonormal basis $Q$ can be constructed such that $A \approx QQ^T A$. The factorization (SVD, QR, etc.) then can be efficiently computed using a relatively small sketch matrix $B = Q^TA$, when the basis matrix $Q$ has few columns. In other words, when the rank of matrix $A$ is small, i.e., $\text{rank}(A) = k \ll \min(m, n)$, a small matrix $B$ can be created and an SVD of the small matrix $B$ reveals the SVD of the original matrix $A$ as long as the range of the projector $Q$ retains the action of the original matrix $A$.

The process of randomized factorization has two stages: (1) Construction of the basis $Q$ with a random projection of the original matrix $A$, and (2) factorization of the small matrix $B$ with a standard deterministic method. In stage (1), it is important to construct $Q = (q^{(1)}, \ldots, q^{(l)})$, in which $q^{(i)}$ denotes the $i$-th column vector of $Q$, such that $Q$ covers the range of $A$. To achieve this, a random vector $\omega$ can be used to form a sample vector $y$ as

$$y^{(i)} = A\omega^{(i)}, \quad i = 1, \ldots, l,$$

where $l = k + o$, and $o$ denotes the oversampling parameter. With $l$ samplings, a sample matrix $Y = (y^{(1)}, \ldots, y^{(l)})$ can be constructed. In some cases, the singular spectrum of matrix $A$ may decay slowly, power iteration is used to overcome this issue by projecting more information of $A$ into the sample matrix $Y$ so as to accelerate the spectrum decay:

$$Y = (AA^\top)^q A\Omega,$$

(2)

where the random matrix $\Omega = (\omega^{(1)}, \ldots, \omega^{(l)})$ is a standard Gaussian matrix of $i.i.d$ normal random variables with mean 0 and variance 1. The acceleration of the power method can be achieved with

$$Y = (AA^\top)^q A\Omega \quad = (U \Sigma V^\top V \Sigma U^\top)^q U \Sigma V^\top \Omega.$$

(3)

Afterward, the basis of $Y$ is computed by $Q = \text{orth}(Y)$, where the operator $\text{orth} \cdot$ represents orthonormalization. In stage (2), matrix $B$ is formed as $B = Q^TA$ and factorized by a conventional deterministic factorization method. The RSVD algorithm is summarized in Algorithm 1.

3. Proposed Method: Block Randomized SVD (BRSVD)

In modern computing architectures, flop counts become rather irrelevant due to the greatly increased communication cost between storage and processor. A measure of algorithmic communication performance is called parallel efficiency [1], which counts how many times the data is accessed by a specific algorithm. In lines 2 and 4 of Algorithm 2, the power method [17] requires totally $2q + 2$ passes
mainly for matrix-matrix multiplication, which translates to communication cost of $(2q + 2)mn$. However, if the memory of processor cannot hold all working data for matrix-matrix multiplication, the algorithm has to be implemented with out-of-core GEMM routines, which will increase communication cost significantly. High communication cost will be the bottleneck for processing large-scale data (We will give detailed analysis in Sect. 3.1). Our goal is to design a new RSVD algorithm by reducing the communication cost without hurting the algorithm’s accuracy.

Note that the sampling process in Eq. (1) is a matrix-matrix multiplication, which can be decomposed into block operations and conducted independently. We wish to design a block power method so as to reduce the data access to $A$. With a moderate partitioning of the input matrix, we can accelerate the singular value decay for each submatrix independently. Therefore, BRSVD can avoid out-of-core GEMM in processing large-scale data.

We give our assumption before elaborating our proposed algorithm. Suppose that the input matrix $A$ is partitioned into $s$ column submatrices. We use the submatrix notation in [1], [25] here, then each column submatrix is expressed as $A_{(c,\beta)} \in \mathbb{R}^{mn \times l}$, where $\beta$ is an ordered set of indices defined as $\beta_j = \{j', \ldots, (j+1)n' - 1\}$ for the $j$-th submatrix with $n'$ columns. Each submatrix $A_{(c,\beta)}$ has its own singular values $\Sigma_j$. Our assumption is that the singular spectrums of submatrices do not highly deviate from each other. With similar singular spectrums, the spectrum decay of each submatrix can be accelerated independently by the power method. This assumption reflects the practical situation. In data processing, it is not recommended to combine matrices with extremely different singular values into a single matrix. Therefore, it is a rare case to have an extremely heterogeneous matrix. Regarding the accuracy of BRSVD, it is experimentally confirmed in Sect. 4.4.

Figure 1 illustrates the overall pipeline of the proposed algorithm. A Gaussian matrix $\Omega_j \in \mathbb{R}^{n' \times d}$ is drawn to sketch each $A_{(c,\beta)}$. The resulting matrix is further refined via a power method with exponent $q$ by reusing the transferred column block $A_{(c,\beta)}$, and the sample matrix $Y_j \in \mathbb{R}^{m \times l}$ is calculated independently as

$$Y_j = \left( A_{(c,\beta)} A_{(c,\beta)}^\top \right)^q A_{(c,\beta)} \Omega_j,$$

which we call a block power method. The submatrix $A_{(c,\beta)}$ can be orthonormalized before applying the power method on sample matrix $A_{(c,\beta)} \Omega_j$. The orthonormalization will reduce the magnitude of the projection so that batches with large magnitude will not overwhelm the final sample matrix $Y$. Note that with the power iteration number $q = 0$, BRSVD and RSVD become equivalent.

**Algorithm 2: Block Randomized SVD (BRSVD)**

```
Input : matrix $A \in \mathbb{R}^{mn}$, target rank $k$, oversampling parameter $\alpha$, power iteration exponent $q$, and partition number $s$
Output: SVD of $A$: matrix $U \in \mathbb{R}^{m \times k}$, $\Sigma \in \mathbb{R}^{k \times k}$, and $V \in \mathbb{R}^{n \times k}$

1. $n' = \lceil n/s \rceil$; $l = k + \alpha$; $Y = 0_{mn}$
2. for $j \leftarrow 0$ to $s - 1$ do in parallel
   3. Generate a Gaussian matrix $\Omega_j \in \mathbb{R}^{n' \times d}$; // sampling & block power iteration
   4. $Y_j = \left( A_{(c,\beta)} A_{(c,\beta)}^\top \right)^q A_{(c,\beta)} \Omega_j$;
5. end
6. $Q = \text{orthonormalize} \left( \sum_j Y_j \right)$; // reduction and orthonormalization
7. for $j \leftarrow 0$ to $s - 1$ do in parallel
   8. $B_{(c,\beta)} \leftarrow Q^\top A_{(c,\beta)}$; // form each $B_{(c,\beta)}$
9. end
10. $[U, \Sigma, V^\top] = \text{svd}(B)$; // gather $B_{(c,\beta)}$ into $B$ and SVD
11. $U = QU$; // form $U$
```

Fig. 1 Diagram of proposed block randomized SVD (BRSVD) method. Column blocks $A_{(c,\beta)}$ in local memory are reused in the RSVD computation pipeline.
Table 1: Computational and communication costs comparison. #flops refers to the arithmetic computational cost in floating point operations. #word indicates communication cost between CPU and GPU. Line # indicates the corresponding operation blocks in Algorithm 2. The non-dominant terms of #flops for random number generation in (1), QR in (4) and SVD in (6) are dropped. (For a detailed #flops count, please refer to Matrix Computations [25] and LAPACK working note [26].) We show the summation cost for out-of-core GEMM in item (2), (3) and (5). b denotes the partition size in out-of-core GEMM. The value of b varies in different implementations (e.g. LAPACK or cuBLAS) and hardware architectures. Note that if b ≥ l, #words (RSVD) are mn, 2qmn and mn for item (2), (3) and (5), respectively.

| Line # | Operation | #flops | #words (RSVD) | #words (BRSVD) |
|--------|-----------|--------|---------------|---------------|
| (1)    | Random number generation | 3      | nq       | 0             |
| (2)    | Sampling   | 4      | 2ml      | 0             |
| (3)    | Power Iterations | 4      | 4qmn     | 0             |
| (4)    | Orthonormalization | 6      | 2ml^2    | 0             |
| (5)    | Form B     | 8      | 2ml      | 0             |
| (6)    | SVD        | 10     | 6ml^2    | 0             |
| (7)    | Form U     | 11      | 2ml^2    | 0             |
| Total  |           | 4(q+1)ml + 4ml^2 + 6nl^2 + nl | 2(q+1)ml/ b + ml | 2mn + ml |

We can observe that if s = 1, Eq. (4) is equivalent to Eq. (2). For an extreme case where s = n, each submatrix is shrunk to a column of A and results in Y_j = (a_i a_i^T) a_i Ω_j = a_i (a_i a_i^T) a_i Ω_j = |a_i|^2 a_i Ω_j. For a moderate batch s ∈ (1, n), the block power method will weaken the effect of accelerating the spectrum decay of sample matrix Y. We will use extensive experiments to verify its accuracy in the next section. Because the sampling and power iteration of each block are independent operations, the data transfer and computation can be overlapped and executed concurrently.

The sample matrix Y can then be computed by reduction as

\[ Y = \sum_{j=0}^{s-1} Y_j \]

After each update of Y, the transferred column block A_βj can be discarded from the GPU for avoiding memory overflow. Once the sample matrix Y is created, its orthonormalized basis Q ∈ R^ml can be constructed by QR decomposition. After acquiring the basis Q, the input matrix A is accessed for the second pass to compute a small core matrix B ∈ R^ml:

\[ B_{(\beta_j)} \leftarrow Q^T A_{(\beta_j)}. \]

As the above operation shows, submatrix B_{(\beta_j)} also can be independently computed from the corresponding column block A_{(\beta_j)}. Finally, all submatrices of B is gathered and an SVD of B is computed to yield its decomposition U, Σ, V^T, and by reprojecting the obtained basis U via Q, the left singular vectors U = QU of the input matrix A can be obtained.

3.1 Efficiency Analysis

For now, let us look at the efficiency analysis summarized in Table 1. We suppose that the data size exceeds the GPU memory, therefore RSVD has to be implemented in an out-of-core fashion. Table 1 compares the computational and communication costs of the proposed algorithm with a straightforward implementation for RSVD. #flops refers to the arithmetic computational cost in floating point operations. #words refers to the communication cost between CPU and GPU memory. We first show the #flops of the original and proposed algorithm. In Algorithm 1, the first GEMM operation for AΩ (line 1) requires 2ml^2 flops, and the following power iteration requires 4qmnl. The QR operation to orthonormalize Y requires 2ml^2 flops. Forming B requires 2ml^2 flops. The SVD operation and forming U requires 6ml^2 and 2ml^2, respectively. Regarding BRSVD, each block sampling A_{s(\beta_j)} requires 2mn^2 flops. The overall flops for s blocks are s × 2mn^2 = 2mn^2, which equals the original algorithm. The other GEMM operations also have the same #flops as the original algorithm. The remaining operations have slightly different #flops for BRSVD, while they only contribute to a small portion of overall computation (We will show the time breakdown in the next section). Therefore, we come to the conclusion that BRSVD have the same theoretical flop counts as the RSVD algorithm.

While the #flops remains the same in both algorithms, BRSVD significantly reduces the communication cost. As shown in Table 1, the #words for RSVD in sampling is mn/ b, where b denotes the partition size in out-of-core GEMM. RSVD requires overall 2(q+1)mn/ b + ml to calculate the out-of-core GEMM. The best case is that if b ≥ l, the communication cost is 2(q+1)mn + ml. We leave b in those equations because GEMM library other than cuBLAS may not allow users to set the partition size b. On the other hand, the communication cost of the proposed BRSVD is fixed to 2mn + ml, which is independent of the power iteration number q. This reduction comes from the fact that BRSVD avoids out-of-core GEMM entirely and reuses the column blocks A_{(\beta_j)} on the local memory (see Fig. 1). As we will see in the next section, this reduction of communication cost significantly improves the efficiency of RSVD computation for large-scale data.

3.2 Implementation Details

Here we describe implementation details of BRSVD that
will be needed to reproduce the work.

For generating Gaussian random matrices $\Omega_j$ on the GPU, we have used cuRAND library [27]. The random number generation is performed in parallel with transferring submatrix $A_{i,j}$ and sampling of the previous submatrix. The GEMM calculation sequence in line 4 of Algorithm 2 is reversed from right to left based on the associative law of matrix-matrix multiplication so as to avoid generating a large projection matrix of size $m \times m$ in the process:

$$Y_j \leftarrow A_{i,j}A_{i,j}^T \cdots A_{i,j}A_{i,j}^T A_{i,j}^T \Omega_j,$$

in which the long arrow on the top represents the order of matrix multiplication.

To orthonormalize the sample matrix $Y$, instead of using a classical Gram-Schmidt (p.254 of [25]) or Cholesky QR (p.163 of [25]), we use the Communication avoiding QR (CAQR) factorization proposed by Demmel et al. [14]. CAQR has a lower communication cost compared to Householder QR [25]. The in-memory GPU implementation [28] achieves fewer data accesses between GPU and CPU memory. In our test, the in-memory CAQR runs roughly 1.5x faster than MAGMA library [29]. Note that we used the geqrf() routine for QR factorization and orgqr() routine for generating matrix $Q$ in MAGMA implementation. In addition, since CAQR is built on the block Householder QR [14], it has intrinsically higher numerical stability than Gram-Schmidt and Cholesky QR.

For computing SVD of the small matrix $B$ on the GPU, we compared gesvd() routines provided by MAGMA and cuSolver [30] libraries and found that there were not much performance difference in terms of both speed and accuracy. We therefore chose cuSolver included in the CUDA library to keep the implementation simple and portable. We implemented BRSVD as a general solver, which can be extended to different precisions. Note that all the following experiments were conducted in double-precision.

4. Experiments

In this section, we use numerical examples to compare the proposed algorithm with several existing ones in terms of computational performance and accuracy. In Sect. 4.1, we use synthetic low-rank matrices with different sizes to compare BRSVD and RVSD in various computing environments. Section 4.4 focuses on validating the numerical stability using matrices with different singular spectrum decay patterns.

4.1 Performance Comparison

To assess the performance of the proposed BRSVD method, we compare with other RSVD implementations listed below. All the implementations are carefully optimized so as to yield the best performance in each setting.

1. RSVD by cuBLAS-XT: This implements Algorithm 1 using the cuBLAS-XT package [31]. cuBLAS-XT is a BLAS3 routines provided by the vendor that can process data larger than the GPU memory. cuBLAS-XT uses 2-dimensional partition in calculating out-of-core GEMM. cuBLAS-XT frees users from dealing with GPU memory allocation and CPU-GPU communication. However, users cannot control the reuse of the transferred data.

2. CPU: This is a straightforward implementation of Algorithm 1 on multi-core CPUs, with which all the working data is processed on CPUs.

3. in-memory on GPU: This is an in-memory GPU implementation, with which all the working data is held on the GPU memory. Since there is no data communication between CPU and GPU, we expect this implementation to give a reference to the performance peak for accelerating CPU and GPU. However, this implementation only works for small scale data that can well fit in the GPU memory.

4.2 Experiment Environment and Setup

For evaluation, we used an NVIDIA V100 (Volta) GPU with 16 GB memory. A V100 was connected to the host via PCIe 3.0 interface. The theoretical peak bandwidth is 7.5 Tflop/s. Although V100 can access its own memory at 900 GB/s, the speed of CPU to GPU data transfer is limited to 15.8 GB/s at maximum. cuBLAS 10.1 and cuSolver 10.1 were used for BLAS and solver routines, respectively. For CPU implementation, we used a system equipped with two Intel 8-core Xeon Silver 4114 processors with 384 GB DDR4-2666 memory. The theoretical peak in double precision is 0.9 Tflop/s for two CPUs. Intel MKL 2018.0.3 [32] was used for CPU BLAS and solver routines.

Regarding the data, we generated matrices with various shapes and sizes with different ranks. A low-rank input matrix $A \in \mathbb{R}^{m \times n}$ with rank-$k$ was created by a product of two low-dimensional matrices $A_i \in \mathbb{R}^{m \times l}$ and $A_r \in \mathbb{R}^{k \times n}$ that were both random Gaussian matrices. For selecting the block size $n'$, we use the maximal block size by querying available memory size at runtime.

4.3 Performance Comparison Results

We first tuned the performance of RSVD by cuBLAS-XT. As shown in Table 1, the communication cost of out-of-core GEMM for double precision is $8nnm/b$ in item (2). If $b \geq l$, cuBLAS-XT is set to do 1-D partition on the large input matrix $A$, which yields the minimum communication cost of $8nm$. According to the roofline model [33], the maximum flop/s = $(\text{flops/\text{words}}) \times \text{bandwidth} = (l/4) \times \text{bandwidth}$. The theoretical peak bandwidth is 15.8 GB/s for CPU-GPU communication. The largest parameters used in our experiments were set to $m = 589,824$, $n = 18,432$, and $l = 1152$ for tall-skinny test cases. The largest parameters for square cases are set to $m = 104,267$, $n = 104,267$, and $l = 814$. 
Therefore, the maximum theoretical peaks are limited to 4.5 TFin/s and 3.2 TFin/s for tall-skinny and square cases, respectively. The theoretical peak performance of V100 GPU is 7.5 TFin/s, which means the performances in all test cases are bandwidth bound. Items (3) and (5) in Table 1 have the same results as item (2). In the performance tuning, we set the \( b \) with different sizes and found that \( b = 4096 \) gave the best performance for both tall-skinny and square cases. We then set \( b = 4096 \) for cuBLAS-XT in all the following experiments. Note that 4096 is larger than the maximum \( l \) value in the following setups.

Figure 2 shows the running time of RSVD and BRSVD with different power iteration number \( q \) values. In each experiment, the ratio of matrix dimensions \((m,n)\) and rank \( k \) were fixed, and the performance was measured by varying the size of input data. For the attained performance, all measurements include the CPU-GPU data transfer time. As mentioned in Sect. 3, with \( q = 0 \), BRSVD and RSVD are exactly the same algorithm. The results for \( q = 0 \) in both Figs. 2 (a) and (b) are almost overlapped, which means that the performance results of both implementations are close, with the same computational and communication cost. The BRSVD shows its advantage over RSVD with a larger \( q \) value. With \( q = 3 \), BRSVD achieves 1.6x and 1.7x maximum acceleration against RSVD by cuBLAS-XT for tall-skinny and square matrices, respectively.

Figure 3 summarizes the performance comparison with multi-core CPU and in-memory results in TFin/s. The performance peaks of in-memory on GPU were 2.37 TFin/s and 1.88 TFin/s for tall-skinny and square matrices, respectively. These can be used as references for other out-of-core curious space. The experimental results in Fig. 3 (a) and (b) show stable performance of BRSVD compared to RSVD by cuBLAS-XT. For data size smaller than 20 GB, BRSVD and RSVD by cuBLAS-XT both showed lower performance than in-memory on GPU due to the high communication cost for initial and final data transfer between CPU and GPU. As we increased the data size, where the size of the working data exceeded the GPU memory capacity at 16GB, the performance of BRSVD gradually increased and reached 2.57 TFin/s for tall-skinny matrices and 2.17 TFin/s for square ones. We profiled each GEMM operation by the NVIDIA Visual Profiler [34] and found the performance of GEMM peaks at 2.9 TFin/s for tall-skinny and 2.3 TFin/s for square. This performance drop rooted from the different matrix shapes of two setups. The performance peak of GEMM varies according to the sizes and shapes of the input matrices. For a detailed GEMM performance analysis, please refer to [35]. Results of large matrices showed slightly higher performance compared to in-memory on GPU. This performance improvement came from the reduced proportion of execution overheads in processing large-scale data.

Regarding RSVD by cuBLAS-XT, the performance peaks were 1.82 TFin/s and 1.44 TFin/s for tall-skinny and square matrices, respectively. Figures 3 (c) and (d) show the breakdown analysis of the running time of BRSVD. Figure 3 (c) shows that GEMM routines in sampling, power iteration and forming \( \mathbf{B} \) dominate the running time for tall-skinny matrices. For square matrices, due to the increased portion of SVD computation and declined GEMM performance which render low flop/s for both CPU and GPU, the overall performance in Fig. 3 (d).
posed algorithm. In the setup, the dimension of matrix $A$.

Now we evaluate the accuracy and robustness of the proposed BRSVD. Figure 4 gives the profiling results in sampling and power iteration. Figure 4(a) shows that the data transfer of RSVR by cuBLAS-XT fully saturated the CPU-GPU bandwidth which rendered the performance to be data transfer bound. Therefore, the performance can not be easily improved by employing more GPUs. Figure 4(b) shows that BRSVD effectively reduced the amount of data transfer which moved the performance bottleneck to computation. Because the parallelizability of BRSVD, its implementation on multiple GPU accelerators is expected to achieve further acceleration.

4.4 Accuracy Evaluation

Now we evaluate the accuracy and robustness of the proposed algorithm. In the setup, the dimension of matrix $A$ was fixed to $10^3 \times 10^3$. We consider several synthetic matrices with different singular value decay patterns to assess the approximation accuracy of BRSVD.

1. Geometric and exponential decays of singular spectrum: We used two different singular value decay patterns here: geometric and exponential decays. For the geometric decay, the $j$-th singular value $\sigma_j$ was defined to have the form of $\sigma_j = \sigma_1 g^{j-1}$. For the exponential decay, it was defined as $\sigma_j = \sigma_1 \exp(-j/w)$. As illustrated in Fig. 5(a), the parameters were set to $g = [0.99, 0.9]$ and $w = [160, 50]$, respectively.

2. Low-rank patterns: In addition to the geometric and exponential decays, we used the experiment setup proposed in [19]. The matrices had a fixed $t$ leading

is reduced by 15.5% compared with the tall-skinny curious space.

To further evaluate the actual amount of communication between CPU and GPU, we profiled the communication and running time of RSVR by cuBLAS-XT and BRSVD. Figure 4 gives the profiling results in sampling and power iteration. Figure 4(a) shows that the data transfer of RSVR by cuBLAS-XT fully saturated the CPU-GPU bandwidth which rendered the performance to be data transfer bound. Therefore, the performance can not be easily improved by employing more GPUs. Figure 4(b) shows that BRSVD effectively reduced the amount of data transfer which moved the performance bottleneck to computation. Because the parallelizability of BRSVD, its implementation on multiple GPU accelerators is expected to achieve further acceleration.

Fig. 5 Singular spectrum decay pattern of matrices used in experiments. (a) shows geometric and exponential decay of singular value. (b) shows a low-rank with decaying tail.

![Figure 5](image-url)

**Fig. 5** Singular spectrum decay pattern of matrices used in experiments. (a) shows geometric and exponential decay of singular value. (b) shows a low-rank with decaying tail.

![Figure 4](image-url)

**Fig. 4** Profiling results for RSVR by cuBLAS-XT (a) and BRSVD (b). Beige and dark blue denote communication time and computation time, respectively. The data size was set to $m = 4 \times 10^5$, $n = 10^4$ and $k = 400$ (40GB for the input matrix). Other parameter is set as: $q = 2$. Note that the time scales are different in (a) and (b) to make illustration clearer.

singular values and a tail with polynomial or exponential decay. For matrices with a polynomial tail, the spectrum have the form

$$\Sigma = \text{diag}(1, \ldots, 1, 2^{-p}, 3^{-p}, \ldots, (n-t+1)^{-p}).$$

For matrices with a exponential tail, the form of spectrum is defined as

$$\Sigma = \text{diag}(1, \ldots, 1, 10^{-th}, 10^{-2h}, \ldots, 10^{-(n-t)h}).$$

As plotted in Fig. 5 (b), the parameters were set to $t = 10$, $p = \{1, 2\}$ and $h = [0.25, 1]$. The left singular matrix $U$ and right singular matrix $V$ are generated as random orthogonal matrices. The test matrices are generated by matrix-matrix multiplication as $A = \Sigma \Phi \Sigma^\top$.

For assessing the accuracy, we used two different measures, namely, relative and actual approximation errors [19]. The relative approximation error is defined as

$$e_1 = \frac{||A - \hat{A}||_F}{\tau_{l+1}(A)} - 1,$$

where $\hat{A}$ denotes the approximations of $A$ obtained by a matrix decomposition algorithm and $\tau_{l+1}(A)$ denotes the root sum of squared singular values after $l$-th [19]. The actual approximation error is defined as

$$e_2 = \frac{||A - \hat{A}||_F}{||A||_F},$$

which evaluates the difference from the ground truth.

Figure 6 depicts the relative approximation errors $e_1$ of the BRSVD in comparison to RSVR using (a) geometric and exponential decay patterns and (b) low-rank patterns. For the parameters of this experiment, we used $k = q = 10$ and $q = 2$ in all the conditions. The result shows that the proposed BRSVD effectively gives a close approximation to RSVR regardless of the number of batches. In particular, BRSVD is shown effective for a matrix with a slow singular
value decay pattern ($\text{Exp} \ w = 160$). For matrices with a faster decay rate, the result indicates that a moderately small batch size can yield good approximation.

We now show three test patterns to take a closer look at actual approximation errors $e^2$ using different algorithms in Fig. 7. We include comparisons with a single-pass RSVD algorithm [18], which is the most accurate approximation among several state-of-the-art single-pass RSVD algorithms. We can see that BRSVD gives substantially closer to the optimal result (RSVD with power iterations) than RSVD without power iteration ($q = 0$ case) and single-pass algorithms. As the result shows, the proposed BRSVD achieves good accuracy with a reduced data accesses.

Table 2 shows the accuracy of each algorithm with different $q$ values. We set the partition number $s = 10$ for BRSVD, which is a normal partition number for the out-of-core GPU computation. We found that for the geometric and exponential decay patterns, BRSVD requires approximately one more iteration to acquire a comparable accuracy of RSVD. For the polynomial low-rank pattern, BRSVD failed to obtain the same level of accuracy as RSVD. Regarding the exponential low-rank pattern, power iteration decreased the error of BRSVD. In contrast, the error of RSVD increased with the increasing $q$ values. Those results also show that it requires the users to set sampling rate $k$ and power iteration number $q$ to find the optimal values for unknown matrix inputs.

5. Applications

In this section, we demonstrate the performance of the proposed method on a diverse collection of data. All data used in the experiments exceeded the GPU memory capacity (16GB). In Sect. 5.1, we take Eigenface [36] and computed tomography data as examples to assess the proposed BRSVD in comparison to both deterministic SVD and RSVD. The hardware used in the experiment was the same in Sect. 4.1.

5.1 Eigenfaces

We apply the proposed BRSVD method to two real datasets. First, we used The Extended Yale Face dataset [37] to assess the leading left singular vectors, which are known as eigenfaces [36]. The database contains 2383 cropped face images, and each image has the resolution of $168 \times 192$. Then a matrix of $32,256 \times 2383$ was constructed as the input matrix $A$.

Figure 8 (a) and (b) show the leading 4 eigenfaces calculated by BRSVD and deterministic SVD, respectively. The results show that our proposed algorithm has no discernible deviation from the deterministic method.

Second, we used a large dataset derived from the FERET dataset [38]. 11,333 images are of size $512 \times 768$. The remaining 14,056 images were resized to the resolution of $512 \times 768$ so that all images can be processed in the same matrix; thus the input matrix size becomes $393,216 \times 25,389$. This matrix with double precision entries will take up about 80 GB for storage. As shown in Fig. 8 (c), the eigenfaces are

| Data     | $q = 0$ | $q = 1$ | $q = 2$ | $q = 3$ |
|----------|---------|---------|---------|---------|
| Geo $g = 0.99$ | 0.898   | 0.898   | 0.850   | 0.867   | 0.836   | 0.858   | 0.829   | 0.854   |
| Exp $\omega = 160$ | 0.936   | 0.936   | 0.905   | 0.918   | 0.896   | 0.913   | 0.891   | 0.910   |
| Poly $p = 1$ | 0.178   | 0.178   | 0.093   | 0.125   | 0.091   | 0.128   | 0.090   | 0.131   |
| Exp $h = 1$ | 4.21e-11 | 4.21e-11 | 1.42e-06 | 3.33e-11 | 1.30e-04 | 3.33d-11 | 3.36e-04 | 1.71e-11 |
Fig. 8 Comparison of BRSVD and deterministic rank-$k$ SVD. (a) and (b) shows the eigenfaces from the Extended Yale Face dataset [37] approximated by BRSVD and deterministic rank-$k$ SVD, respectively. (c) shows the eigenfaces computed from the FERET dataset [38] approximated by BRSVD. (d) shows the singular values calculated by BRSVD and RSVD for the FERET dataset.

Table 3 Accuracy comparison for the leading four left singular vectors. The sampling parameters were set as: $k = o = 32$ for Yale Face and $k = o = 128$ for FERET, respectively. Other parameters were set as: $q = 2$ and $s = 10$ for both datasets.

|      | Yale Face | FERET |
|------|-----------|-------|
|      | RSVD      | BRSVD | RSVD  | BRSVD |
| $v_1$| 1.47e-5   | 3.31e-5| 9.46e-7| 4.77e-6|
| $v_2$| 1.52e-4   | 3.39e-4| 8.22e-5| 3.20e-4|
| $v_3$| 3.43e-3   | 9.13e-3| 6.44e-5| 5.21e-4|
| $v_4$| 2.92e-3   | 9.91e-3| 5.72e-4| 2.05e-3|

Table 4 Performance comparison of BRSVD, RSVD by cuBLAS-XT, and RSVD on CPU. All experiments were conducted in double precision. The parameters used in the experiments were set as: $k = o = 64$, $q = 2$ and $s = 8$.

|      | BRSVD    | RSVD by cuBLAS-XT | RSVD on CPU |
|------|----------|--------------------|-------------|
| Time (s) | 23.1     | 39.4               | 145.3       |

5.2 Computed Tomography

SVD has been used in computed tomography (CT) reconstruction and denoising [39], [40]. To evaluate the performance on large scale data, we used a standard Shepp-Logan phantom [41] as the input dataset. Each entry of the data is a double precision gray-scale voxel of the phantom. 2048 slices of CT image with a resolution of $2048 \times 2048$ (= 4,194,304) were vectorized to form a 4,194,304 $\times$ 2048 double precision matrix. This matrix took up 64GB for storage. As shown in Table 4, BRSVD and RSVD achieved 6.3x and 3.7x, respectively, against the multi-core CPU implementation.

6. Conclusions

This paper presented a fast RSVD algorithm named BRSVD that fully utilizes GPU accelerators. We provided a detailed study of the data access efficiency of the proposed algorithm, and demonstrated the performance with detailed experimental results and real applications. The proposed algorithm allows us to see the directions of randomized algorithm development in evolving computing environments. Most immediate is the independent operations which fit accelerators. Regarding the accuracy of the proposed algorithm, our proposed two-pass algorithm has a higher accuracy than the single-pass one. However, it has a slightly lower accuracy than the standard multi-pass with the same computational cost. Our future work mainly focuses on showing the mathematical proof of BRSVD. Other work includes mixed-precision randomized algorithms and enabling BRSVD to run in a multi-GPU cluster environment to achieve further acceleration.

Acknowledgments

This research was in part supported by “Program for Leading Graduate Schools” of the Ministry of Education, Culture, Sports, Science and Technology, Japan and the Japan
Society for the Promotion of Science KAKENHI grant numbers 15H01687 and 16H02801. Finally, the authors thank reviewers for their valuable comments.

References

[1] N. Halko, P.G. Martinsson, and J.A. Tropp, “Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions,” SIAM Review, vol.53, no.2, pp.217–288, 2011.

[2] G. Golub and W. Kahan, “Calculating the singular values and pseudo-inverse of a matrix,” Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis, vol.2, no.2, pp.205–224, 1965.

[3] R.M. Larsen, “Lanczos bidiagonalization with partial reorthogonalization,” DAIMI Report Series, vol.27, no.537, 1998.

[4] A. Frieze, R. Kannan, and S. Vempala, “Fast Monte-Carlo algorithms for finding low-rank approximations,” Journal of the ACM, vol.51, no.6, pp.1025–1041, 2004.

[5] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, “The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale,” SIAM Review, vol.60, no.4, pp.808–865, 2018.

[6] K. Pearson, “On lines and planes of closest fit to systems of points in space,” Philosophical Magazine, vol.2, no.11, pp.559–572, 1901.

[7] H. Hotelling, “Analysis of a complex of statistical variables into principal components,” Journal of Educational Psychology, vol.24, no.6, pp.417–441, 1933.

[8] S. Voronin and P.G. Martinsson, “RSVDPACK: An implementation of randomized algorithms for computing the singular value, interpolative, and CUR decompositions of matrices on multi-core and GPU architectures,” arXiv preprint arXiv:1502.05366v3, vol.2, 2016.

[9] D. Irony, S. Toledo, and A. Tiskin, “Communication lower bounds for distributed-memory matrix multiplication,” Journal of Parallel and Distributed Computing, vol.64, no.9, pp.1017–1026, 2004.

[10] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, “Minimizing communication in numerical linear algebra,” SIAM Journal on Matrix Analysis and Applications, vol.32, no.3, pp.866–901, 2011.

[11] T.B. Jablin, P. Prabhu, J.A. Jablin, N.P. Johnson, S.R. Beard, and D.I. August, “Automatic CPU-GPU communication management and optimization,” ACM SIGPLAN Notices, vol.46, no.6, pp.142–151, 2011.

[12] J. Demmel, “Communication-avoiding algorithms for linear algebra and beyond,” Proc. 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS), p.585, IEEE, 2013.

[13] M. Hoemmen, “Communication-avoiding Krylov subspace methods,” Ph.D. thesis, University of California, Berkeley, Berkeley, CA, USA, 2010.

[14] 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, no.1, pp.A206–A239, 2012.

[15] J.W. Demmel, L. Grigori, M. Gu, and H. Xiang, “Communication avoiding rank revealing QR factorization with column pivoting,” SIAM Journal on Matrix Analysis and Applications, vol.36, no.1, pp.55–89, 2015.

[16] A. Haidar, K. Kabir, D. Fayad, S. Tomov, and J. Dongarra, “Out of memory SVD solver for big data,” Proc. 21st High Performance Extreme Computing Conference (HPEC), pp.1–7, IEEE, 2017.

[17] V. Rokhlin, A. Szlam, and M. Tygert, “A randomized algorithm for principal component analysis,” SIAM Journal on Matrix Analysis and Applications, vol.31, no.3, pp.1100–1124, 2009.

[18] W. Yu, Y. Gu, J. Li, S. Liu, and Y. Li, “Single-pass PCA of large high-dimensional data,” Proc. 26th International Joint Conference on Artificial Intelligence (IJCAI), pp.3350–3356, 2017.

[19] J.A. Tropp, A. Yurtsever, M. Udell, and V. Cevher, “Practical sketching algorithms for low-rank matrix approximation,” SIAM Journal on Matrix Analysis and Applications, vol.38, no.4, pp.1454–1485, 2017.

[20] I. Yamazaki, J. Kurzak, P. Luszczek, and J. Dongarra, “Random sampling to update partial singular value decomposition on a hybrid CPU/GPU cluster,” Proc. 27th International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2015.

[21] A. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra, “High-performance Cholesky factorization for GPU-only execution,” Proc. 10th Workshop on General Purpose GPUs (GPGPU), pp.42–52, 2017.

[22] P.-G. Martinsson, V. Rokhlin, and M. Tygert, “A randomized algorithm for the approximation of matrices,” Tech. Rep. YALEU/DCS/TR-1361, Yale University, Department of Computer Science, New Haven, CT, USA, 2006.

[23] T. Sarlós, “Improved approximation algorithms for large matrices via random projections,” Proc. 47th IEEE Symposium on Foundations of Computer Science (FOCS), pp.143–152, 2006.

[24] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert, “Randomized algorithms for the low-rank approximation of matrices,” Proc. National Academy of Sciences of the United States of America (PNAS), vol.104, no.51, pp.20167–20172, 2007.

[25] G.H. Golub and C.P. Van Loan, Matrix Computations, 4th ed., Johns Hopkins University Press, 2012.

[26] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Staney, D. Walker, and R. Williams, “LAPACK working note 95 ScalAPACK: A portable linear algebra library for distributed memory computers-design issues and performance,” Tech. Rep. UT CS-95-283, University of Tennessee, Knoxville, TN, USA, 1995.

[27] NVIDIA Corporation, “CURAND Library Programming Guide v10.1,” 2018.

[28] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, “Communication-avoiding QR decomposition for GPUs,” Proc. 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp.48–58, 2011.

[29] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, “Dense linear algebra solvers for multicore with GPU accelerators,” Proc. 24th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp.1–8, 2010.

[30] NVIDIA Corporation, “CUSOLVER Library v10.1,” 2018.

[31] NVIDIA Corporation, “CUBLAS Library User Guide,” 2019.

[32] Intel Corporation, “Developer Reference for Intel® Math Kernel Library - C,” 2019.

[33] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” Communications of the ACM, vol.52, no.4, pp.65–76, 2009.

[34] NVIDIA Corporation, “Profiler User’s Guide Version 10.1,” 2019.

[35] V. Volkov and J.W. Demmel, “Benchmarking GPUs to tune dense linear algebra,” Proc. 20th International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2008.

[36] M.A. Turk and A.P. Pentland, “Face recognition using eigenfaces,” Proc. 4th IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), pp.586–591, 1991.

[37] A. Georghiades, P. Belhumeur, and D. Kriegman, “From few to many: Illumination cone models for face recognition under variable lighting and pose,” IEEE Trans. Pattern Anal. Mach. Intell., vol.23, no.6, pp.643–660, 2001.

[38] P.J. Phillips, H. Moon, P. Rauss, and S.A. Rizvi, “The FERET evaluation methodology for face-recognition algorithms,” Proc. 10th IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), pp.137–143, 1997.

[39] D. Verhoeven, “Limited-data computed tomography algorithms for the physical sciences,” Applied optics, vol.32, no.20, pp.3736–3754, 1993.

[40] K. Abhari, M. Marsoussi, J. Alirezaie, and P. Babyn, “Computed to-
mography image denoising utilizing an efficient sparse coding algorithm,” Proc. 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA), pp.259–263, 2012.

[41] L.A. Shepp and B.F. Logan, “The fourier reconstruction of a head section,” IEEE Trans. Nucl. Sci., vol.21, no.3, pp.21–43, June 1974.

Yuechao Lu is currently working toward the Ph.D. degree in computer science at Osaka University. His current research interests include randomized algorithms and high performance computing.

Yasuyuki Matsushita received the B.S., M.S., and the Ph.D. degrees in EECS from the University of Tokyo, in 1998, 2000, and 2003, respectively. From April 2003 to March 2015, he was with Visual Computing group at Microsoft Research Asia. In April 2015, he joined Osaka University as a professor. His research area includes computer vision, machine learning and optimization. He is on the editorial board of the IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), the International Journal of Computer Vision (IJCV), the IPSJ Journal of Computer Vision and Applications (CVA), The Visual Computer Journal, and Encyclopedia of Computer Vision. He served/is serving as a program co-chair of the PSIVT 2010, 3DIMPVT 2011, ACCV 2012, ICCV 2017, and a general co-chair for ACCV 2014. He is a senior member of the IEEE.

Fumihiko Ino received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently a Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation.