Received: 24 May 2019 | Revised: 14 January 2020 | Accepted: 16 February 2020
DOI: 10.1002/cpe.5754

RESEARCH ARTICLE

Reducing the amount of out-of-core data access for GPU-accelerated randomized SVD

Yuechao Lu[1] | Ichitaro Yamazaki[2] | Fumihiko Ino[1] | Yasuyuki Matsushita[1] | Stanimire Tomov[2] | Jack Dongarra[2]

[1]Graduate School of Information Science and Technology, Osaka University, Osaka, Japan

[2]Innovative Computing Laboratory, University of Tennessee, Knoxville, Tennessee, USA

Correspondence
Yuechao Lu, Department of Computer Science, Osaka University, 1-5 Yamadaoka, Suita, Osaka, Japan, 565-0871.
Email: yc-lu@ist.osaka-u.ac.jp

Summary

We propose two acceleration methods, namely, Fused and Gram, for reducing out-of-core data access when performing randomized singular value decomposition (RSVD) on graphics processing units (GPUs). Out-of-core data here are data that are too large to fit into the GPU memory at once. Both methods accelerate GPU-enabled RSVD using the following three schemes: (1) a highly tuned general matrix-matrix multiplication (GEMM) scheme for processing out-of-core data on GPUs; (2) a data-access reduction scheme based on one-dimensional data partition; and (3) a first-in, first-out scheme that reduces CPU-GPU data transfer using the reverse iteration. The Fused method further reduces the amount of out-of-core data access by merging two GEMM operations into a single operation. By contrast, the Gram method reduces both in-core and out-of-core data access by explicitly forming the Gram matrix. According to our experimental results, the Fused and Gram methods improved the RSVD performance up to 1.7× and 5.2×, respectively, compared with a straightforward method that deploys schemes (1) and (2) on the GPU. In addition, we present a case study of deploying the Gram method for accelerating robust principal component analysis, a convex optimization problem in machine learning.

KEYWORDS
divide and conquer, GPU, out-of-core computation, singular value decomposition

1 | INTRODUCTION

Singular value decomposition (SVD)[1] is a matrix approximation algorithm that finds two orthogonal matrices **U** and **V** and a diagonal matrix $\mathbf{\Sigma}$ of an input matrix **A** such that $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top$, where **A** can be approximated with matrices smaller than itself by truncating **U**, **V**, and $\mathbf{\Sigma}$. Studies on improving the performance and numerical stability of SVD algorithms have been ongoing ever since its advent,[2-4] and successfully applied to various fields, such as bioinformatics,[5,6] physics,[7,8] and machine learning.[9-11] In particular, SVD of dense *tall-skinny matrices*, whose the height (m rows) is at least one magnitude larger than the width (n columns), is of great interest to researchers in computer vision,[12] image compression,[13] facial recognition,[14] and data analysis.[15]

Recently, randomized SVD (RSVD) algorithms[16-18] have been proposed to further accelerate SVD by exploiting the low-rank structure inherent in matrix data. Compared with classical deterministic algorithms,[1] randomized algorithms have been shown to access the input data less number of times, while maintaining the desirable accuracy of the approximation.[16-18] RSVD is typically build on random sampling[18,19] and power iteration methods.[18,20,21] The sampling method constructs a subspace of the input matrix, which reduces dimension. The power method takes powers of $\mathbf{A}$ (ie, $\mathbf{A}^\top\mathbf{A}$) to increase the approximation accuracy. Both underlying methods can be implemented using general matrix-matrix multiplication (GEMM) routines, whereas general matrix-vector multiplication (GEMV)[22] is required for deterministic SVD computation. GEMM is more suitable for modern parallel computers, where it achieves 20 to 40 times higher flop/s than GEMV.[22,23]

While matrix approximation, such as SVD, has been made efficient based on randomization methods, modern computing architectures, such as the use of graphics processing unit (GPU) accelerators,[24] enable the development of even faster algorithms by taking the advantage of parallel computing. Nevertheless, there are a few major challenges that prevent matrix approximation algorithms to fully benefit from the modern computing architectures. First, large matrix data may not fit into the GPU memory due to its limited capacity. Second, communication across distinct memory hierarchies or networks often constitutes a performance bottleneck[23,25] due to the increasing gap between arithmetic and communication performance.[26,27]

The above challenges have been addressed using the following two strategies: communication-avoiding algorithms[28,29] reduce the communication of intermediate data during computation, whereas pass-efficient algorithms[19,30] save the memory bandwidth by reducing the number of passes over data. As for the multipass RSVD,[18] target data is passed over $2q + 1$ times to attain a high-accuracy approximation, where q denotes the number of iterations in the power method.[18,21] By contrast, single-pass algorithms[31,32] access the target data in just one pass. However, single-pass algorithms include iterations to construct a *sketch*,[18,31,32] which have data dependency that avoids parallelization required by the modern computing architectures. Furthermore, there exists an accuracy-performance tradeoff in single-pass algorithms.[31,32] Due to this accuracy issue, multipass RSVD is preferred in convergence-sensitive applications such as robust principal component analysis (RPCA), where the noise and low-rank component in input data are typically separated in an iterative manner.[33]

In this study, we focus on a multi-pass RSVD algorithm proposed by Martinsson et al,[16,18] which has a higher accuracy than single-pass algorithms according to solid error-bound analysis. We extend this algorithm so that large tall-skinny matrices can be rapidly decomposed using a divide-and-conquer method that reduces *out-of-core data access* on a CPU-GPU heterogeneous system. Out-of-core data access here involves CPU-GPU data transfer in our target CPU-GPU system, where the GPU is regarded as the computing core. By contrast, in-core data (ie, data on the GPU memory) can be rapidly accessed without additional CPU-GPU data transfer. Compared with the previous in-core algorithms,[34,35] which made the assumption that input and intermediate data can be fully stored in the GPU memory, we consider an RSVD algorithm at scale, where the matrices include more than 10^6 entries. Our out-of-core approach relaxes the limitation on the data size by allowing the data to be stored in both the CPU and GPU memories.

The main contributions of this research include:

- **Highly tuned, out-of-core GEMM with theoretical performance model**.

  Since GEMM is a building block of RSVD algorithms, we extensively tuned the GEMM operation with theoretical performance analysis based on an extension of the roofline model.[36] The extended model shows that GPU-accelerated out-of-core GEMM is bandwidth bound for tall-skinny matrices. In addition, we present experimental results where our out-of-core scheme achieved higher performance than previous GEMM schemes.

- **Two out-of-core RSVD methods, namely, Fused and Gram**.

  Both methods are based on three common schemes: (1) the abovementioned out-of-core GEMM scheme; (2) a data-access reduction scheme based on one-dimensional (1D) data partitioning; and (3) a first-in, first-out (FIFO) scheme that reduces CPU-GPU data transfer by reverse iteration. The Fused method is a communication-avoiding algorithm because the method merges GEMM operations to reduce the amount of the CPU-GPU data transfer (ie, out-of-core data access). By contrast, the Gram method is a pass-efficient algorithm because the method explicitly computes the Gram matrix to reduce the number of data passes (ie, both in-core and out-of-core data access) from $2q + 1$ to 3.

- **Case study with a practical application**.

  We apply the Gram method to nuclear norm minimization in an RPCA algorithm that heavily relies on SVD computation. We found that our GPU-based implementation provided 23.3× faster RSVD computation compared with that of a CPU-based implementation, doubling the RPCA performance for the video background subtraction.

Our source code, which will be included in the MAGMA package,[37] is freely available at http://www-ppl.ist.osaka-u.ac.jp/research/code/.

The rest of this article is organized as follows. Section 2 provides an overview of related studies. Section 3 presents a technical background regarding RSVD. Section 4 outlines the proposed highly tuned out-of-core GEMM scheme with its theoretical performance analysis and preliminary evaluation. Section 5 details our proposed methods constructed over the underlying GEMM scheme. Section 6 compares and contrasts the proposed methods with the existing methods. Section 7 describes the case study with an RPCA application. Conclusions and prospects for future work are provided in Section 8.

2 | RELATED WORK

In this section, we brief the related work regarding deterministic SVD algorithms, randomized algorithms, and GPU-accelerated randomized algorithms.

2.1 | Deterministic SVD algorithms

In 1965, Golub and Kahan[2] proposed the first stable SVD algorithm for computers using a bidiagonlization method. EISPACK[38] first implemented the bidiagonlization method in Fortran. EISPACK was designed to run on a single-core CPU and was replaced by LINPACK,[39] which first implemented the SVD algorithm with basic linear algebra subprogram (BLAS) interface. The performance of LINPACK was limited by the BLAS1 implementation and benefited little from multicore architectures.[23] LAPACK[40] redesigned the SVD algorithm to use BLAS3 routines wherever possible to improve the performance on the multicore CPUs. Recently, a two-stage bidiagonal reduction method has been proposed to adapt SVD to new computer architectures like GPU accelerators.[22]

Theoretically, deterministic SVD of a matrix $\mathbf{A}$ can be calculated with its Gram matrix $\mathbf{A}^\top\mathbf{A}$, which is a well-known method.[41] However, forming the Gram matrix is usually avoided due to its high computational cost in linear algebra packages like LAPACK[40] or MAGMA.[37] In this work, we apply this idea to out-of-core RSVD computation; we introduce a method which explicitly forms the Gram matrix to reduce the number of data passes, that is, the amount of in-core and out-of-core data access.

2.2 | Randomized algorithms

Randomized algorithms[4,16-18,21,42,43] have been proposed to reduce the time and space complexities required for the approximation of high-dimensional data. The efficiency of such algorithms in tackling large-scale data has led to an increased interest from the high-performance computing community.

In general, randomized algorithms have a typical computation flow. First, they construct a subspace of the input data using random sampling. Computation is then performed only on the sampled subspace to reduce the costs of computation, communication, and storage. Randomized algorithms are popular in the field of big data analytics, where large quantities of data are missing or contain noise. There is a strong demand for low-precision approximation that is useful for the restoration of the entire data and elimination of irrelevant data hindering data analysis. Randomized algorithms typically employ one of the two sampling methods: namely, uniform and nonuniform sampling methods.

The uniform sampling method uses independent and identically distributed random numbers as the entries of sampling matrices, which are then multiplied with the target matrix. This method is also called random projection; it has a strong relative-error bound and widely used in randomized matrix factorization.[18] The major objective of random projection is to project the high-dimensional data onto a low-dimensional subspace by exploiting the low-rank characteristics of the data.[18] Deterministic decomposition is then performed on this subspace, and the decomposed results are projected back to form the full factorization. However, uniform sampling has relatively higher computational cost compared with that of nonuniform sampling. In particular, given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ and sampling matrix $\mathbf{Q} \in \mathbb{R}^{n\times \ell}$, it takes $\mathcal{O}(mn\ell)$ time to compute the sampled matrix $\mathbf{AQ}$.

By contrast, the nonuniform sampling method constructs a subspace by selecting a certain set of vectors from the target data. Given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, an importance sampling distribution[19] of the input matrices is first computed to perform the selection. The distribution and selection have the time complexity of $\mathcal{O}(mn)$, which is lower than the overhead of the uniform sampling mentioned above. Moreover, nonuniform sampling has a higher accuracy compared with that of uniform sampling. Hence, our base RSVD algorithm[16,18] deploys uniform sampling.

2.3 | GPU-accelerated randomized algorithms

Randomized algorithms have been implemented on GPUs to achieve further acceleration. For example, low-rank approximations of dense matrices were computed and evaluated with a truncated SVD[44] and a truncated QR factorization with column-pivoting.[45] The RSVDPACK library[35] contains a set of randomized algorithms for computing low-rank matrix approximations on a single GPU. These studies assume that the matrix data are small enough to fit into the GPU memory. Therefore, the maximum data size was limited by the capacity of the GPU memory, which is an order of magnitude smaller than that of the CPU memory.

As for large matrices whose data size exceeds the capacity of the GPU memory, traditional matrix factorization algorithms have been investigated for more than a decade.[37,46] For example, divide-and-conquer methods have been proposed to perform out-of-core LU,[47] QR, or Cholesky factorization.[48,49] Similarly, underlying BLAS routines have been extended to deal with large matrices. To the best of our knowledge, cuBLAS-XT[50] is the first library, implementing the out-of-core BLAS routines. BLASX[51] deploys a least recently used cache management scheme to implement the BLAS routines for out-of-core matrices, which aims at reducing the amount of the data transfer between the CPU and GPU.

While GPUs are widely used as accelerators for memory- or compute-intensive applications, GPU programming is still not easy partly due to its complex memory hierarchy levels. In particular, application developers are required to manually manage CPU and GPU memories to gain a high performance of GPU-based systems. To deal with this issue, NVIDIA introduced Unified Memory,[52] that realizes an integrated memory space available from both the CPU and GPU. This capability frees application developers from explicitly managing data movement between the CPU and GPU. Furthermore, Unified Memory accepts large data that may exceed the capacity of the GPU memory.

3 | RSVD ALGORITHM

Algorithm 1 lists a pseudocode of the RSVD algorithm.[16,18] Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the target rank k, oversampling parameter o, and power iteration count q, the algorithm outputs matrices $\mathbf{\Sigma}$, $\mathbf{U}$, and $\mathbf{V}$, where $\mathbf{\Sigma}$ is the diagonal matrix whose diagonal entries approximate the k-largest singular values of $\mathbf{A}$, and $\mathbf{U}$ and $\mathbf{V}$ approximate the corresponding left and right singular vectors, respectively. The oversampling parameter o is added to the target rank k to guarantee the approximation accuracy for matrices with a slow singular value decay.

Algorithm 1. Multipass RSVD. Function orth(·) orthogonalizes the column vectors while functions qr(·) and svd(·) return the QR factorization and deterministic SVD of a matrix, respectively. We use $\tilde{\Sigma}(1:k, 1:k)$ and $\tilde{U}(:, 1:k)$ to denote the leading $k \times k$ submatrix of $\tilde{\Sigma}$ and the submatrix consisting of the first k columns of $\tilde{U}$, respectively.

```
   Input  : matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, target rank $k$, oversampling parameter $o$ and power iteration count $q$.
   Output: $\Sigma$, $\mathbf{U}$ and $\mathbf{V}$ such that $\mathbf{A} \approx \mathbf{U}\Sigma\mathbf{V}^{\mathsf{T}}$ with $k \times k$ diagonal $\Sigma$, and orthonormal column vectors $\mathbf{U}$ and~ $\mathbf{V}$.
 1 Generate a random matrix $\mathbf{Q} \sim \mathcal{N}(0,1)^{n \times \ell}$,where $\ell = k + o$.
 2 for $i = 1$ to $q$ do
 3     $\mathbf{P} = \mathbf{AQ}$;
 4     $\mathbf{P} = \mathrm{orth}(\mathbf{P})$ ;                                  // if needed
 5     $\mathbf{Q} = \mathbf{A}^{\mathsf{T}}\mathbf{P}$;
 6     $\mathbf{Q} = \mathrm{orth}(\mathbf{Q})$
 7 end
 8 $\mathbf{P} = \mathbf{AQ}$;
 9 $[\mathbf{P}, \mathbf{B}] = \mathrm{qr}(\mathbf{P})$ ;
10 $[\tilde{U}, \tilde{\Sigma}, \tilde{V}] = \mathrm{svd}(\mathbf{B})$ ;
11 $\Sigma = \tilde{\Sigma}(1:k, 1:k)$;
12 $\mathbf{U} = \mathbf{P}\tilde{U}(:, 1:k)$ ;
13 $\mathbf{V} = \mathbf{Q}\tilde{V}(:, 1:k)$ ;
```

First, the RSVD algorithm generates the basis vectors $\mathbf{P}$ and $\mathbf{Q}$ that approximate the range and domain of the matrix $\mathbf{A}$, respectively. The power iteration method in lines 2 to 7 improves the approximation accuracy. During these q power iterations, basis vectors of $\mathbf{P}$ and $\mathbf{Q}$ are orthogonalized to maintain the numerical stability. After the power iterations, QR factorization is performed on $\mathbf{P}$ in line 9 such that the upper triangular matrix $\mathbf{B}$ is the projected matrix of dimension $\ell \times \ell$ (ie, $\mathbf{B} = \mathbf{P}^T\mathbf{AQ}$). In other words, a small matrix $\mathbf{B}$ can be created by GEMM when matrix $\mathbf{A}$ is low-rank, that is, $\mathrm{rank}(\mathbf{A}) = k \ll \min(m, n)$. This small matrix B is useful for revealing the SVD of the original matrix $\mathbf{A}$ at low cost. The SVD of $\mathbf{B}$ is then computed by deterministic SVD in line 10. Finally, the left singular vector $\tilde{U}$ and right singular vector $\tilde{V}$ are projected back onto $\mathbf{P}$ and $\mathbf{Q}$ to generate the left and right singular vectors of $\mathbf{A}$ in lines 12 and 13, respectively.

The orthogonalizations in lines 4 and 6 of Algorithm 1 ensure that the different columns of $\mathbf{P}$ or $\mathbf{Q}$ converge to different dominant singular vectors. However, the original RSVD algorithm[16] was later improved to skip the orthogonalization of $\mathbf{P}$.[18] Halko et al[18] used a diverse collection of real applications to illustrate the accuracy and stability of RSVD with skipping the orthogonalization of $\mathbf{P}$. Their test cases included the adaptive range approximation in physics, the graph Laplacian approximation in image processing, the face recognition in machine learning, and so on. Similar studies[12,53] were presented based on the improved algorithm.[16] Thus, the orthogonalization of $\mathbf{P}$ can be skipped depending on the accuracy required by the target application; this improvement is widely accepted for practical applications.

Algorithm 1 indicates that RSVD is dominated by GEMM computation in lines 3 and 5 with access to the large tall-skinny matrix $\mathbf{A}$, which we assume not to fit in the GPU memory. Therefore, a straightforward solution deploys the out-of-core GEMM routines that offload heavy computation to GPUs. In this case, the amount of CPU-GPU data transfer increases linearly with the number q of power iterations because $\mathbf{A}$ is accessed $2q + 1$ times in Algorithm 1. Thus, RSVD solvers for large data rely on out-of-core GEMM, whose performance fluctuates drastically according to divide-and-conquer strategies. In the following section, we provide theoretical analysis and empirical results to reveal that out-of-core RSVD is bandwidth bound requiring a data-access reduction scheme to minimize the amount of data transfer between the CPU and GPU.

4 | GPU-ACCELERATED OUT-OF-CORE GEMM

In this section, we first extend the roofline model[36] to investigate the upper bound of the GPU-accelerated out-of-core GEMM performance. We then propose a 1D partition scheme for tall-skinny matrices and compare our GEMM implementation with several existing implementations.

The following analysis and experiments were conducted in double precision using an experimental system equipped with two Intel 8-core Xeon Silver 4110 CPUs and two NVIDIA Tesla V100 (Volta) GPUs.[24] These CPUs had 96 GB of DDR4-2133 main memory and provided a double precision peak performance of 0.67 Tflop/s in total. Each GPU had 16 GB of GPU memory with theoretical peak performance of 7.0 Tflop/s in double precision. Each GPU was connected to the CPU via a PCIe 3.0 link allowing bidirectional transfer between the CPU and the GPU. In our system, we observed 6.9 Tflop/s for in-core GEMM computation, and the effective transfer bandwidth per PCIe link was 13.1 and 12.8 GB/s for CPU-to-GPU and GPU-to-CPU, respectively. The CPU and GPU implementations used Intel MKL 2018.0.3[54] and cuBLAS 9.2,[50] respectively, for processing in-core GEMM operations. In addition, we used MAGMA 2.4[37] and LAPACK 3.7[40] to evaluate the approximation accuracy and initialize multithreaded computation.

4.1 | Performance model

We now propose a performance model based on the roofline model[36] to investigate the upper bound of the GPU-accelerated out-of-core GEMM performance. The roofline model is useful for locating the performance bottleneck, that is, either arithmetic or memory operations, which limits the entire performance of linear algebra algorithms.[55,56] Considering the roofline model, the attainable performance in flop/s can be defined as

$$T = \min\left(\frac{F}{D}B,\ C\right), \tag{1}$$

where F denotes the number of floating point operations, D denotes the amount of memory access, B denotes the peak memory bandwidth and C denotes the peak computational performance of the target hardware. The term F/D is called operational intensity, which represents the ratio of floating point operations to total data access. The operational intensity is the key algorithmic factor that determines attainable flop/s, whereas the remaining parameters B and C depend on the target hardware.

In the following discussion, we consider DGEMM, $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$, to investigate the performance of the tall-skinny GEMM, $\mathbf{P} = \mathbf{AQ}$, where $\mathbf{P} \in \mathbb{R}^{m\times\ell}$, $\mathbf{A} \in \mathbb{R}^{m\times n}$ $(m \gg n)$ and $\mathbf{Q} \in \mathbb{R}^{n\times\ell}$, which appears in the power iteration in line 3 of Algorithm 1. Since our focus is on the tall-skinny $\mathbf{A}$, we assume that (1) the large matrix $\mathbf{A}$ must be partitioned into smaller blocks to be processed with a divide-and-conquer strategy, and (2) the small projection matrix $\mathbf{Q}$ fits into the GPU memory and hence is broadcasted to all GPUs. In addition, we assume that (3) the input and output matrices exist in the CPU memory; (4) m, n, and ℓ are multiplies of the block dimension b to simplify the discussion; and (5) the DGEMM implementation sends the output matrix $\mathbf{C}$ to the GPU even if $\beta = 0$.

To illustrate the impact of matrix shapes on the out-of-core GEMM performance, we substituted the operational intensity with the matrix size parameters such as m, n, and ℓ. The flop F of GEMM was fixed to $2mn\ell$ for double precision GEMM. By contrast, the memory access cost D was interpreted as CPU-GPU data transfer cost to consider the out-of-core execution on the GPU. Furthermore, the cost D was appropriately selected according to the two data partition schemes as follows.

1. Row-wise 1D partition scheme with block dimension b (Figure 2). According to this scheme, m/b blocks in total must be computed for the matrix $\mathbf{P}$. Each block requires $b \times n$ entries in $\mathbf{A}$ and $n \times \ell$ entries in $\mathbf{Q}$ to compute all entries in the block. Providing $\mathbf{Q}$ can be reused on the GPU, and $\left(\sum_{j=1}^{m/b} bn\right) + n\ell + m\ell = mn + n\ell + m\ell$ entries are transferred from the CPU to GPU and $m\ell$ entries for the opposite direction. Assuming bidirectional transfer between the CPU and GPU, the data transfer cost can be calculated as $D = mn + n\ell + m\ell$. Therefore, the attainable performance T_1 for the 1D partition scheme can be written as

$$T_1 = \min\left(\frac{2mn\ell}{mn + n\ell + m\ell}B,\ C\right). \tag{2}$$

2. 2D partition scheme with the block dimension $b \times b$, where b is the block size. According to this scheme, $m\ell/b^2$ blocks in total must be computed for the matrix $\mathbf{P}$. Each block requires $b \times n$ entries in $\mathbf{A}$ and $n \times b$ entries in $\mathbf{Q}$ to compute all entries in the block. Providing $\mathbf{Q}$ can be reused on the GPU, and $\left(\sum_{j=1}^{m\ell/b^2} bn\right) + n\ell + m\ell = mn\ell/b + n\ell + m\ell$ entries are transferred from the CPU to GPU and $m\ell$ entries for the opposite direction. Assuming the bidirectional transfer scheme mentioned above, the data transfer cost can be calculated as $D = mn\ell/b + n\ell + m\ell$. The attainable performance T_2 for the 2D partition scheme can be written as

$$T_2 = \min\left(\frac{2mn\ell}{mn\ell/b + n\ell + m\ell}B,\ C\right). \tag{3}$$

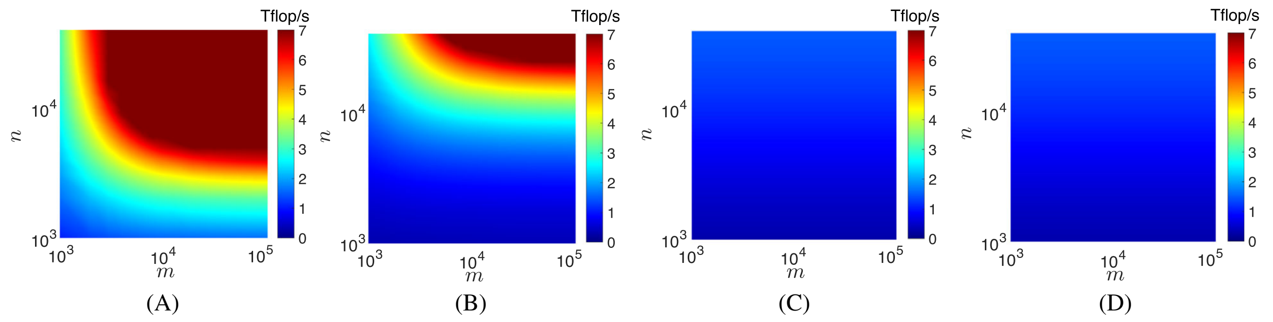

FIGURE 1 Performance upper bound of GPU-accelerated out-of-core GEMM: $\mathbf{P} = \mathbf{AQ}$, where $\mathbf{P} \in \mathbb{R}^{m\times\ell}$, $\mathbf{A} \in \mathbb{R}^{m\times n}$, and $\mathbf{Q} \in \mathbb{R}^{n\times\ell}$. Row-wise one-dimensional partitioning results for, A, square ($\ell = n$) and B, tall-skinny matrices $\mathbf{Q}$ ($\ell = n/10$). Two-dimensional partition results for, C, square ($\ell = n$) and D, tall-skinny matrices $\mathbf{Q}$ ($\ell = n/10$). Hardware specific parameters were set for the Tesla V100 GPU: $B = 13$ GB/s and $C = 7$ Tflop/s. Both partition schemes used the block size b of 1024, which was the default setup of cuBLAS-XT.[50] GEMM, general matrix-matrix multiplication; GPU, graphics processing unit

Equations (2) and (3) can be further simplified by considering the shape of the matrix $\mathbf{Q}$. In more detail, the parameter ℓ can be eliminated for the following two cases: (1) square matrix ($\ell = n$) and (2) tall-skinny matrix ($\ell = n/10$, for example). After this elimination, Equations (2) and (3) can be rewritten as functions of m and n. Consequently, the performance upper bound can be shown on a 2D heatmap, where the vertical and horizontal axes are the matrix dimensions m and n of $\mathbf{A}$, respectively (Figure 1).

Figure 1 clearly demonstrates that a higher performance illustrated as a red area can be expected only with the 1D partition scheme. With respect to the 2D partition scheme shown in Figure 1C,D, the performance upper bounds are strictly limited for both the square and tall-skinny shapes of the matrix $\mathbf{Q}$ with less than 1 Tflop/s due to the narrow bandwidth of the CPU-GPU data transfer. The same limitation can also be found for the 1D partition scheme; however, the maximum performance reached up to 7 Tflop/s in this case. Another remarkable point here is that the shape of the matrix $\mathbf{A}$ also strongly impacts the performance upper bound if the matrix $\mathbf{Q}$ is tall-skinny, which is demonstrated by a larger blue area in Figure 1B compared with that in Figure 1A. Thus, the transfer bandwidth between the CPU and GPU limits the GPU-accelerated out-of-core GEMM performance for the tall-skinny $\mathbf{A}$ ($m \gg n$), which frequently appears in big data analytics.

In summary, we make the following observations about the extended performance model.

- Row-wise 1D data partition is a promising solution for the GEMM operations of large tall-skinny matrices because this solution minimizes the amount of CPU-GPU data transfer.
- 2D data partition inevitably increases the amount of CPU-GPU data transfer limiting the out-of-core GEMM performance. The performance will deteriorate, especially for the tall-skinny GEMM operations, which are the main focus of our research.

4.2 | Row-wise 1D partition scheme for out-of-core GEMM

We propose a row-wise 1D partition scheme for the two GEMM operations applied to tall-skinny matrices in lines 3 and 5 of Algorithm 1. We use row-wise partition rather than column-wise partition for the following two reasons.

- Lower data transfer cost. The outer-product update of the matrix $\mathbf{P}$ results in a large amount of CPU-GPU data transfer for each GEMM computation if the tall-skinny $\mathbf{A}$ is partitioned into 1D column blocks. Furthermore, the column-wise partition requires multiple buffers and a complicated synchronization mechanism to accumulate the updates from different blocks. By contrast, the row-wise partition allows per-block GEMM operations to be data-independent without the race condition. This asynchronous property enables the pipelined execution of partitioned blocks, where data transfers are efficiently overlapped with GEMM computation. In more detail, software pipelining can be implemented using compute unified device architecture (CUDA) streams.[52]
- Higher GEMM performance. GEMM routines generally run faster for square matrices rather than tall-skinny matrices. Compared with the column-wise partition, the row-wise partition generates square-like blocks, which are useful for maximizing the performance of per-block GEMM operations.

Figure 2 illustrates how we apply our 1D partition scheme to the GEMM operations in the RSVD algorithm. As shown in Figure 2A, we partition the tall-skinny matrices $\mathbf{A}$ and $\mathbf{P}$ into blocks for the first GEMM, $\mathbf{P} = \mathbf{AQ}$. Given the block size b, this partition scheme generates m/b blocks for each $\mathbf{A}$ and $\mathbf{P}$, each having dimensions of $b \times n$ and $b \times \ell$, respectively. The small $n \times \ell$ matrix $\mathbf{Q}$ is initialized on the CPU and then broadcasted to all GPUs to allow them to independently call the cuBLAS routines for applying the in-core GEMM operations to blocks. The second GEMM, $\mathbf{Q} = \mathbf{A}^\top\mathbf{P}$, is based

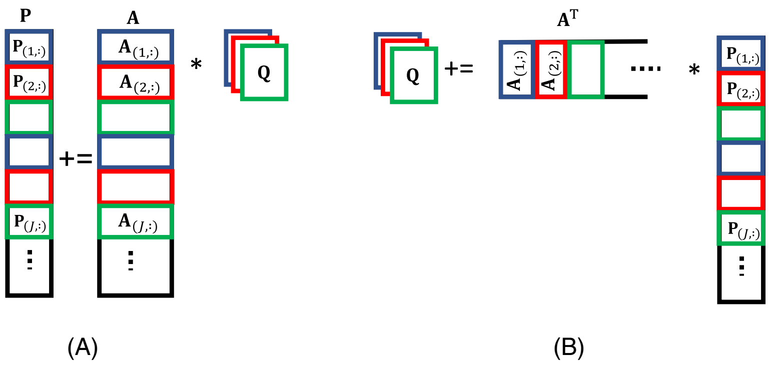

FIGURE 2 Proposed row-wise one-dimensional partition scheme for tall-skinny GEMM. A, $\mathbf{P} = \mathbf{AQ}$ (line 3 of Algorithm 1), where $\mathbf{A}$ is partitioned into blocks and $\mathbf{Q}$ is broadcasted among GPUs. B, $\mathbf{Q} = \mathbf{A}^{\mathrm{T}}\mathbf{P}$, where $\mathbf{Q}$ is accumulated by reduction. Blocks are assigned to CUDA streams[52] in a round-robin fashion (ie, block-cyclic distribution as illustrated with different colors). Because $\mathbf{P}$ is computed in a block-wise manner, the GPU is allowed to store the part of $\mathbf{P}$. GEMM, general matrix-matrix multiplication; GPU, graphics processing unit

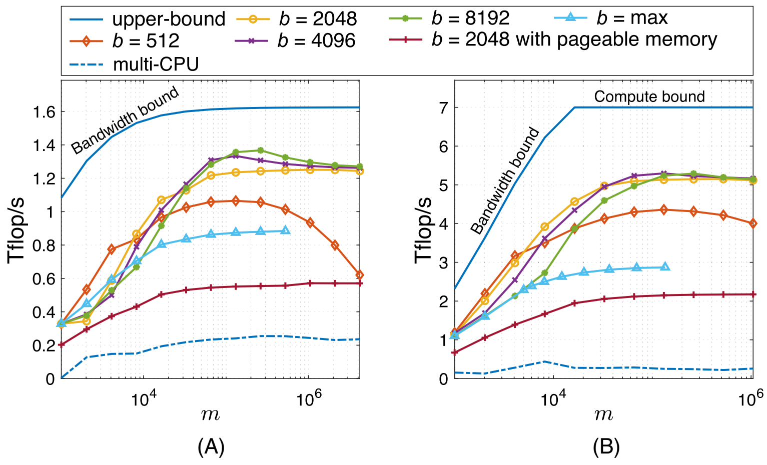

FIGURE 3 Out-of-core GEMM performance with different block sizes b on a single Tesla V100 GPU. Measured results for, A, small ($\ell = n = 1000$) and B, large square matrices ($\ell = n = 5000$). The maximum block size ($b = \max$) indicates the performance without data partition. A multithreaded CPU version was also evaluated on two 8-core CPUs. Note that the upper-bound line is curved because the horizontal axis of our extended model is the height of matrix $\mathbf{A}$, which is different from that (ie, the operational intensity) of the original roofline model. GEMM, general matrix-matrix multiplication; GPU, graphics processing unit

on an outer-product of $\mathbf{Q}$. Similar to the process shown in Figure 2A, we broadcast $\mathbf{Q}$ to all GPUs for processing blocks in parallel (Figure 2B). A reduction of small $\mathbf{Q}$ is followed after finishing all GEMM operations.

As mentioned above, GEMM is not universally efficient for tall-skinny matrices. This low efficiency is due to the computation of tall-skinny GEMM, which is closer to GEMV (BLAS2 routines) than GEMM (BLAS3 routines). The BLAS2 routines are less efficient than the BLAS3 routines due to vector accesses that degrade cache hit rate on both the multicore CPU and GPU; BLAS3 routines are 20 to 40 times more efficient than BLAS2 routines.[22,23] Regarding the in-core performance of tall-skinny GEMM, Chen et al[57] achieved 1.1 to 3.0× speedups over cuBLAS for tall-skinny matrices with up to 16 columns. Their GEMM solution can be easily integrated into our RSVD solver, but the maximum number of columns is limited by 16, mainly due to the limitation on computational resources, such as register files.

In the following discussion, we use the submatrix notation[1,18] to denote the block matrix; each block of $\mathbf{A}$ is expressed as $\mathbf{A}_{(J,:)} \in \mathbb{R}^{b\times n}$, where J is an ordered set of indices defined as $J = \left[jb, jb+1, \dots, jb+b-1\right]$ for the jth block ($j \geq 0$).

4.3 | Performance tuning and comparison of out-of-core GEMM

We first tuned our out-of-core GEMM implementation with respect to the block size b. Figure 3 illustrates the measured performance with different block sizes, ranging from 512 to 8192. The block size was also maximized to measure the performance without data partition; due to the lack of data partition, we failed to execute large matrices of $m \geq 2 \times 10^5$ entries. Similarly, we measured the performance of a multithreaded CPU implementation for reference. All except one of the setups allocated matrices in the pinned CPU memory.[52]

Figure 3 demonstrates that the pinned memory was faster than the pageable memory; the effective bandwidth from the pinned memory to the GPU memory was 1.7 times higher than that from the pageable memory. Hence, we allocated matrices in the pinned memory for the rest of the experiments. We also observed that the performance for the maximum block size ($b = \max$) was significantly degraded due to the lack of overlapped execution; there were no partitioned blocks that could be processed in the pipeline. Therefore, data partition schemes are necessary to maximize the performance of the out-of-core GEMM operations.

As mentioned in Section 4, the in-core GEMM computation ran at 6.9 Tflop/s, which was close to the theoretical peak of 7.0 Tflop/s. With respect to the out-of-core GEMM computation, Figure 3 demonstrates that there is some margin between the theoretical upper bound and the measured results. This margin occurred due to data partition, which applies GEMM operations to the partitioned blocks. In other words, partitioned blocks

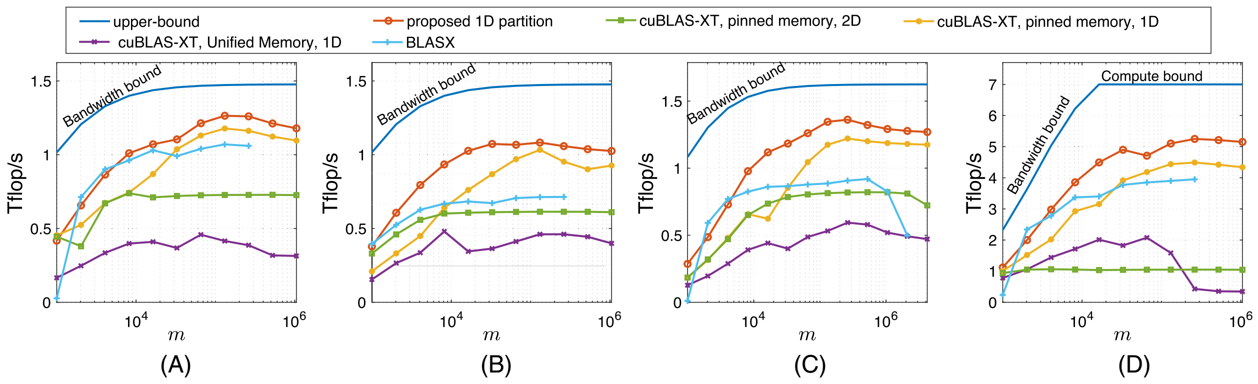

FIGURE 4 Performance comparison of the out-of-core GEMM implementations on a single Tesla V100 GPU. Results with different shapes for the matrix **Q**: A, tall-skinny ($n = 5000, \ell = 500$), B, short-wide ($n = 500, \ell = 5000$), C, small square ($\ell = n = 1000$), and D, large square ($\ell = n = 5000$). BLASX is a high-level library that hides specific data partition and memory allocation methods. GEMM, general matrix-matrix multiplication; GPU, graphics processing unit

are not sufficient large to maximize the effective performance; in-core GEMM runs without such data partition. However, small blocks are required to overlap CPU-GPU data transfer with GPU computation.

As for the block size, a tradeoff point must be found to maximize the performance of 1D partition scheme. For small block sizes of $b \leq 2048$, we observed only 8 to 11 GB/s of CPU-GPU transfer bandwidth and 8.3 to 9.3 Gflop/s of arithmetic performance, because the parallelism in each small block was not sufficient to achieve the maximum efficiency on the GPU. On the other hand, the maximum block size $b = \max$ resulted in a poor performance due to inefficient execution of the pipeline. Weighing the tradeoffs here, we selected $b = 4096$ for the following experiments.

We now compare and contrast the performance of out-of-core GEMM implementation against that obtained with previous out-of-core GEMM implementations: cuBLAS-XT[50] and BLASX.[51] Figure 4 illustrates the performance comparison when using a single Tesla V100 GPU. Among these implementations, the proposed GEMM was the fastest in most cases. Note that cuBLAS-XT was evaluated with the following three setups: (1) 2D partition with pinned memory (default); (2) 1D partition with pinned memory; and (3) 1D partition with Unified Memory. As for 1D partition, we set the block size as $b = n$, which enforced the row-wise 1D block partition. Among these setups, the highest performance was obtained when using the second setup, that is, 1D partition with pinned memory. The default setup failed to achieve high performance for the out-of-core GEMM operations; this behavior is consistent with Equations (2) and (3), which imply that 2D partition transfers more entries than 1D partition. Therefore, excessive data transfers saturated the CPU-GPU bandwidth, resulting in a lower performance. While enforced 1D partition significantly increased the performance, there was still a performance gap compared with that of the proposed GEMM. The advantage of the proposed GEMM implementation comes from manual broadcast of the small matrix **Q** to GPUs (Figure 2A), which reduces the amount of CPU-GPU data transfer for all block GEMM operations. We obtained similar results for another GEMM operation in the power method ($\mathbf{Q} = \mathbf{A}^{\top}\mathbf{P}$ in Figure 2B).

With Unified Memory, we failed to increase the performance for larger m even with 1D partition. We speculate that Unified Memory failed to efficiently deal with the complicated memory access pattern. It is not easy to automate CPU-GPU data transfer without explicitly managing the memory. Finally, BLASX performed stably for all setups (Figure 4). However, the maximum matrix size was limited to $m \leq 2 \times 10^6$ entries; matrices larger than the maximum size resulted in an execution failure without any error message.

We also evaluated the speedups of the considered implementations on two Tesla V100 GPUs (Figure 5). For each implementation, we used the same implementation as the baseline, which ran on a single V100 GPU. All implementations except Unified Memory demonstrated increased speedups as m grows; the speedups reached around 1.8× for $m \geq 2 \times 10^4$ demonstrating a scalable performance on two GPUs. By contrast, small matrices of $m < 1 \times 10^4$ resulted in low speedups, due to the overheads of GPU initialization that took up around 30% of the overall execution time. Similarly, cuBLAS-XT with Unified Memory degraded the GEMM performance when using two GPUs.

5 | PROPOSED OUT-OF-CORE RSVD METHODS

We first describe the basic scheme that uses 1D partition for out-of-core RSVD computation. We then present a FIFO scheme that reduces the amount of CPU-GPU data transfer by employing the reverse iteration. We further elaborate on two methods, namely, Fused and Gram, which are our main contribution to this work. Both the Fused and Gram methods are built on the basic and FIFO schemes.

Table 1 summarizes the computational and communication costs of all proposed variations. This table considers only the power method, which is the main contribution of the article. As shown in Table 1, our main concern is to reduce the amount of CPU-GPU data transfer, which limits the performance of out-of-core computation on the GPU. We also show a pipelined mechanism to overlap kernel execution with data transfer. As for kernel optimization, our approach is to deploy vendor's optimized GEMM kernels with tuned execution setups, as investigated in Section 4.3.

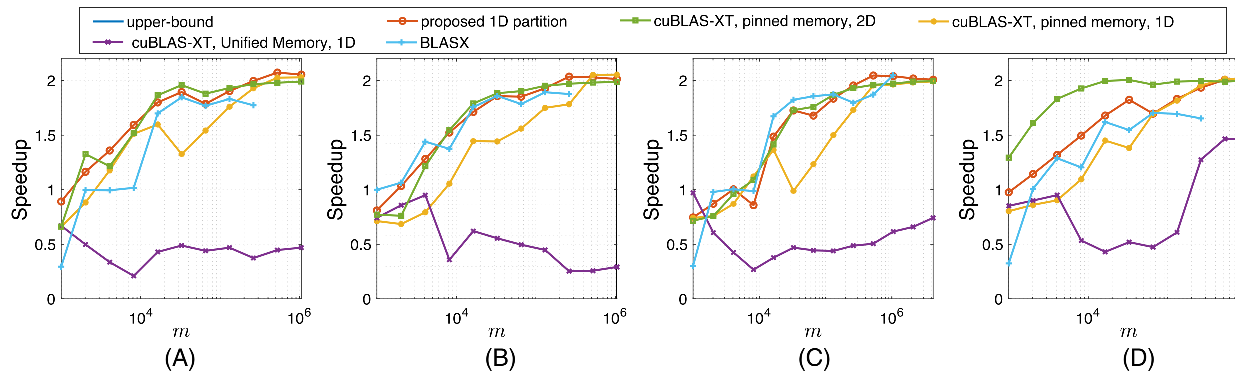

FIGURE 5 Speedup of the out-of-core GEMM implementations on two Tesla V100 GPUs. Results with different shapes for the matrix **Q**. A, tall-skinny ($n = 5000$, $\ell = 500$), B, short-wide ($n = 500$, $\ell = 5000$), C, small square ($\ell = n = 1000$), and D, large square ($\ell = n = 5000$). For each implementation, we executed the same implementation on a single GPU to compute the speedup. GEMM, general matrix-matrix multiplication; GPU, graphics processing unit

TABLE 1 Comparison of the proposed methods in terms of computational cost, the number of data passes, and CPU-GPU data transfer cost

Method	*F*: # of floating point operations	# of data passes	*D*: CPU-GPU data transfer cost
Basic (data-access reduction)	$(2q+1)mn\ell$	$2q+1$	$(2q+1)mn$
FIFO (reverse iteration)	$(2q+1)mn\ell$	$2q+1$	$< (2q+1)mn$
Fused	$(2q+1)mn\ell$	$2q+1$	$< (q+1)mn$
Gram	$mn^2 + qn^2\ell + mn\ell$	3	$< 2mn$

Note: The number of data passes and CPU-GPU data transfer cost correspond to in-core access cost and out-of-core access cost, respectively. We consider matrix **A** to evaluate the number of data passes. Both the Fused and Gram methods adopt the FIFO scheme to reduce the CPU-GPU data transfer cost.
Abbreviations: FIFO, first-in, first-out; GPU, graphics processing unit.

5.1 | Basic and FIFO schemes for reducing out-of-core data access

As summarized in Table 1, a straightforward divide-and-conquer implementation of RSVD passes **A** for $2q + 1$ times with the proposed 1D scheme, where **A** and **P** are partitioned, whereas **Q** is broadcasted to all GPUs (Figure 2). The QR factorization of **P** in line 9 of Algorithm 1 is computed using Cholesky factorization[58] shown in Algorithm 2. In Algorithm 2, the GEMM operation at line 4 can be processed by calling the GEMM function, such as the `cublasDgemm()` kernel of the cuBLAS library. Algorithm 2 only transfers small $\ell \times \ell$ matrices **B** and **R** between the CPU and GPU because Algorithm 1 stores the input matrix **P** in the GPU memory (at line 8) before processing the QR factorization. Note that the data transfers are omitted in Algorithm 2 to simplify its description. Both SVD and QR procedures in line 10 of Algorithm 1 and line 6 of Algorithm 2, respectively, can be implemented using the standard LAPACK routines. We denote this implementation as the basic scheme.

Algorithm 2. QR Factorization. Function chol(·) returns the Cholesky factorization of a matrix

```
   Input  : $\mathbf{P} \in \mathbb{R}^{m\times\ell}$.
   Output: $\mathbf{P} \in \mathbb{R}^{m\times\ell}$ and $\mathbf{R} \in \mathbb{R}^{\ell\times\ell}$.
 1 $\mathbf{B} = \mathbf{0}^{\ell\times\ell}$ ;                         // Initialize B with all 0
 2 $s = m/b$ ;                                 // # of blocks
 3 for $j = 1$ to $s$ do
 4     $\mathbf{B}$ += $\mathbf{P}^{\mathsf{T}}_{(J,:)}\mathbf{P}_{(J,:)}$
 5 end
 6 $\mathbf{R} = \mathrm{chol}(\mathbf{B})$ ;                      // factored on CPU
 7 for $j = 1$ to $s$ do
 8     $\mathbf{P}_{(J,:)} = \mathbf{P}_{(J,:)}\mathbf{R}^{-1}$
 9 end
```

The basic scheme can be easily improved by reorganizing the loop structure. Algorithm 3 presents the FIFO scheme that requires less access to **A**. As shown in lines 7 to 9, the execution order of iterations for computing **Q** is reversed such that blocks of $\mathbf{A}_{(J,:)}$ from the first GEMM in line 5 can be reused for that for the second GEMM in line 8; the data transfer of $\mathbf{A}_{(J,:)}$ occurs only before the first GEMM in line 5, where the CUDA kernel is invoked from the CPU. As compared with the basic scheme, this data reuse on the GPU reduces the amount of CPU-GPU data transfer; at the same time, the reduced amount depends on the number of blocks that can be stored at once in the GPU memory (Table 1). In addition to this data reuse, blocks can be further reused across different iterations, that is, the second GEMM at the current ith loop can be reused for the first GEMM at the next $(i + 1)$th loop. Note that the worst case of $mn + 2q(m - b)n$ occurs when the GPU memory can hold only a single block of $\mathbf{A}_{(J,:)}$; all blocks of the total size mn are sent to the GPU at the first data access to **A** (i = 1, lines 4-6), then a block of $b \times n$ entries is reused $2q$ times by the following GEMM. By contrast, the best case of mn can be obtained when the GPU memory can hold all blocks of $\mathbf{A}_{(J,:)}$; however, this contradicts to our assumption that data size exceeds the GPU memory capacity.

Algorithm 3. FIFO scheme. This method replaces lines 2 to 8 of Algorithm 1

```
Input   : $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{Q} \in \mathbb{R}^{n \times \ell}$, block size $b$ and power iteration count $q$.
Output: $\mathbf{P} \in \mathbb{R}^{m \times \ell}, \mathbf{Q} \in \mathbb{R}^{n \times \ell}$ and $\mathbf{B} \in \mathbb{R}^{\ell \times \ell}$.
1  $\mathbf{B} = \mathbf{0}^{\ell \times \ell}$ ;                          // Initialize B with all 0
2  $s = m/b$ ;                                          // # of blocks
3  for $i = 1$ to $q$ do
4      for $j = 1$ to $s$ do
5          $\mathbf{P}_{(J,:)} = \mathbf{A}_{(J,:)}\mathbf{Q}$;
6      end
7      for $j = s$ to $1$ do
8          $\mathbf{Q}$ += $\mathbf{A}^{\top}_{(J,:)}\mathbf{P}_{(J,:)}$ ;                  // reuse 1D blocks
9      end
10     $[\mathbf{Q}, \sim] = \mathrm{qr}(\mathbf{Q})$;
11 end
12 for $j = 1$ to $s$ do
13     $\mathbf{P}_{(J,:)} = \mathbf{A}_{(J,:)}\mathbf{Q}$;
14     $\mathbf{B}$ += $\mathbf{P}^{\top}_{(J,:)}\mathbf{P}_{(J,:)}$;
15 end
```

5.2 | Fused method for reducing out-of-core data access

The amount of CPU-GPU data transfer can be further reduced by taking the advantage of the fact that the orthogonalization of **P** in line 4 of Algorithm 1 can be omitted in many practical applications, as mentioned in Section 3. Algorithm 4 shows the Fused method that skips the orthogonalization of **P**. Consequently, the two block GEMM operations in lines 5 and 8 of Algorithm 3 can be processed in a single Fused loop, as shown in lines 6-7 and 11-12 of Algorithm 4. After this loop fusion, the two GEMM operations are performed with the same block $\mathbf{A}_{(J,:)}$ to compute $\mathbf{P}_{(J,:)}$ and $\tilde{\mathbf{Q}}$ before accessing the next block; the `if-else` statement in lines 4 to 14 is used to process $\mathbf{A}_{(J,:)}$ in inverse order such that the blocks can be reused in the next i loop. Consequently, the matrix **A** is transferred only once in every iteration, and thus, the amount of CPU-GPU data transfer is reduced to $(q + 1)mn$. Combined with the FIFO scheme, the worst case of $mn + q(m - b)n$ occurs when only a single block of $\mathbf{P}_{(J,:)}$ is reused, whereas the best case of mn is obtained when all blocks are reused at all GEMM operations.

Recall here that each block GEMM operation is processed by calling the GEMM kernel, which runs on the GPU. Therefore, the `if-else` statement in Algorithm 4, which exists outside the CUDA kernel function, is executed on the CPU. Consequently, there is no concern on warp divergence issues,[52] which degrade the performance on the GPU.

The Fused method requires an additional memory space for storing $\tilde{\mathbf{Q}}$, which has the same size as the sampling matrix **Q**. However, this additional cost is negligible because the size of **Q** is assumed to be small ($\ell \times \ell$). With respect to the number of data passes, the Fused method requires the same number $2q + 1$ of data passes as the basic scheme.

Algorithm 4. Fused method. This method replaces lines 2 to 11 of Algorithm 3

```
Input  : $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{Q} \in \mathbb{R}^{n\times \ell}$, block size $b$ and power iteration count $q$.
Output: $\mathbf{P} \in \mathbb{R}^{m\times \ell}$ and $\mathbf{Q} \in \mathbb{R}^{n\times \ell}$.
1  $\tilde{\mathbf{Q}} = \mathbf{0}^{n\times \ell}$ ;                          // Initialize $\tilde{Q}$ with all 0
2  $s = m/b$ ;                                      // # of blocks
3  for $i = 1$ to $q$ do
4      if $i\%2 == 1$ then
5          for $j = 1$ to $s$ do
6              $\mathbf{P}_{(J,:)} = \mathbf{A}_{(J,:)}\mathbf{Q}$;
7              $\tilde{\mathbf{Q}} \mathrel{+}= \mathbf{A}^{\top}_{(J,:)}\mathbf{P}_{(J,:)}$ ;                // reuse 1D blocks
8          end
9      else
10         for $j = s$ to $1$ do
11             $\mathbf{P}_{(J,:)} = \mathbf{A}_{(J,:)}\mathbf{Q}$;
12             $\tilde{\mathbf{Q}} \mathrel{+}= \mathbf{A}^{\top}_{(J,:)}\mathbf{P}_{(J,:)}$ ;                // reuse 1D blocks
13         end
14     end
15     $[\mathbf{Q}, \sim] = \mathrm{qr}(\tilde{\mathbf{Q}})$;
16 end
```

5.3 | Gram method for reducing in-core and out-of-core data access

Similar to the Fused method, the Gram method skips the orthogonalization of **P** in line 4 of Algorithm 1. Therefore, every power iteration computes the following equation:

$$\mathbf{Q} = \mathrm{orth}(\mathbf{A}^{\top}(\mathbf{AQ})). \tag{4}$$

We explicitly form the Gram matrix as $\mathbf{G} = \mathbf{A}^{\top}\mathbf{A} \in \mathbb{R}^{n\times n}$, and thereby, Equation (4) is mathematically equivalent to the following equation:

$$\mathbf{Q} = \mathrm{orth}(\mathbf{A}^{\top}(\mathbf{AQ})) = \mathrm{orth}((\mathbf{A}^{\top}\mathbf{A})\mathbf{Q}) = \mathrm{orth}(\mathbf{GQ}). \tag{5}$$

As mentioned in Section 2.1, forming the Gram matrix is usually avoided due to its high computation cost; however, we do this to reduce the number of data passes, that is, the amount of in-core and out-of-core data access.

Algorithm 5 lists a pseudocode of our proposed Gram method. The power method is applied to the Gram matrix **G** without accessing **A** in lines 6 to 9 of Algorithm 5, so that the number of data passes is reduced to 3, which is independent from q. These q-independent passes prevent the performance degradation if some data present a slow singular decay pattern that requires more power iterations to form the approximation.[18] In this case, the communication cost with more passes to **A** incurs a tremendous burden on the narrow CPU-GPU bandwidth.

Algorithm 5. Gram method. This method replaces line 1 to 11 of Algorithm 3

```
Input  : $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{Q} \in \mathbb{R}^{n\times \ell}$ and block size $b$.
Output: $\mathbf{Q} \in \mathbb{R}^{n\times \ell}$.
1  $\mathbf{G} = \mathbf{0}^{n\times n}$ ;                          // Initialize $\mathbf{G}$ with all 0
2  $s = m/b$ ;                                      // # of blocks
3  for $j = 1$ to $s$ do          // $\mathbf{G} = \mathbf{A}^{\top}\mathbf{A}$: form Gram matrix with 1D partition of $\mathbf{A}$
4      $\mathbf{G} \mathrel{+}= \mathbf{A}^{\top}_{(J,:)}\mathbf{A}_{(J,:)}$
5  end
6  for $i = 1$ to $q$ do
7      $\mathbf{Q} = \mathbf{GQ}$;
8      $[\mathbf{Q}, \sim] = \mathrm{qr}(\mathbf{Q})$;
9  end
```

For 1D partition of **A**, GEMM operations for forming **G** (lines 3-5 of Algorithm 5) can be processed with the data transfer cost of mn. However, the Gram method increases the number of floating-point operations from $(2q + 1)mn\ell$ to $mn^2 + qn^2\ell + mn\ell$; mn^2 and $qn^2\ell$ correspond to lines 3-5 and 6-9 of Algorithm 5, respectively. Despite this extra computation, the Gram method reduces the amount of CPU-GPU data transfer to less than $2mn$ when combined with the FIFO scheme (Table 1). Considering the large gap between the communication cost and computational cost, we believe that forming **G** reduces the overall run time at the expense of increased floating-point operations. We experimentally validate this assumption in Section 6.

5.4 | Implementation details

We implemented all of the proposed methods with the underlying GEMM operations tuned in Section 4.3. In addition, we integrated the following techniques into our RSVD solver.

Kernel optimization: The MAGMA library,[37] which wraps the CUDA library, was deployed for all GEMM operations, GPU initialization, memory management, and data transfer. The MAGMA library assumes that (1) matrices are stored in column-major format and (2) the leading dimension of matrices are round up to multiples of 32. These assumptions are useful for maximizing effective memory bandwidth by achieving memory access coalescing[52] on the GPU. For QR and deterministic SVD computations on the CPU, we used the `potrf()` and `gesvd()` routines included in LAPACK,[40] respectively.

Software pipeline: A double buffering approach was implemented to realize a software pipeline mechanism to overlap CUDA kernel execution with CPU-GPU data transfer. In more detail, our solver creates two CUDA streams[52] per GPU. Each stream, wrapped inside the MAGMA queue structure,[37] runs asynchronously so that overlapping can be achieved to maximize the entire performance. For the GEMM operation at line 5 of Algorithm 3, a CUDA stream calls a single cuBLAS CUDA kernel on a buffer while another stream executes data transfer on another buffer.

Multi-GPU execution: Our solver assigns matrix blocks to GPUs in a round-robin fashion. For the FIFO and Fused methods, each GPU obtains intermediates of **B** and **Q** after processing assigned blocks. The solver then transfers **Q** to the CPU to process the QR factorization with the `potrf()` routine on the CPU. Finally, the CPU calls the `axpy()` routine to form **B** and **Q** from all the intermediates. A similar approach was used to obtain matrix **G** for the Gram method. We used a single GPU to compute the power iteration process in lines 6 to 9 of Algorithm 5 because **Q** and **G** are small enough to fit into a single GPU memory.

6 | EXPERIMENTAL RESULTS

We now evaluate the proposed methods in terms of the performance and numerical stability. In the following discussion, the CPU-GPU data transfer time was taken into account for the measured performance in flop/s.

We also compare the proposed methods with the previous methods. We implemented a CPU-based method using the LAPACK library for reference. The CPU-based method was multithreaded using OpenMP directives.[59] Intel MKL 11.3.1 was linked to LAPACK for BLAS routines. We thoroughly tuned the solver to gain the highest performance on our experimental CPUs. All implementations were compiled using GNU C++ 7.4.0 and CUDA 10.1.

All experiments were conducted in double precision using two Intel Xeon Silver 4114 CPUs and two NVIDIA Tesla V100 GPUs. These CPUs had 384 GB of DDR4-2666 main memory and provided a double precision peak performance of 0.9 Tflop/s in total.

6.1 | Performance evaluation

We applied the proposed schemes to the basic scheme step by step to investigate the performance impact of each scheme; (1) the basic method (the basic scheme in Section 5.1), (2) the FIFO method (the FIFO scheme in Section 5.1), (3) the Fused method (Section 5.2), and (4) the Gram method (Section 5.3). For the power iteration count, we used $q = 1, 4$, and 8, which covers from low to high-accuracy approximation. Note that with $q = 4$, RSVD achieved almost the same level of accuracy as the deterministic SVD (Section 6.3). Figure 6 shows the RSVD performance measured with different numbers of rows m. In Figure 6, the FIFO method slightly reduced the overall runtime by approximately 10%. Furthermore, the Fused method increased the basic performance by reducing the amount of CPU-GPU data transfer D from $(2q + 1)mn$ to less than $(q + 1)mn$, which corresponds to at most 33%, 42%, and 44% reductions for $q = 1, 4$, and 8, respectively. Accordingly, the overall execution time was reduced by 24.2%, 37.8%, and 42.5% for $q = 1, 4$, and 8, respectively. Thus, the gap between the reduction rate on execution time and that on data transfer amount became closer as we increased q. To understand this behavior, we investigated the time breakdowns of the Fused method; the proportion other than GEMM operations dropped from 12.5% to 6.7%, which implies that reducing data transfer amount increased its impact as q increased. Overall, the Fused method achieved up to 1.7× speedup over the basic method.

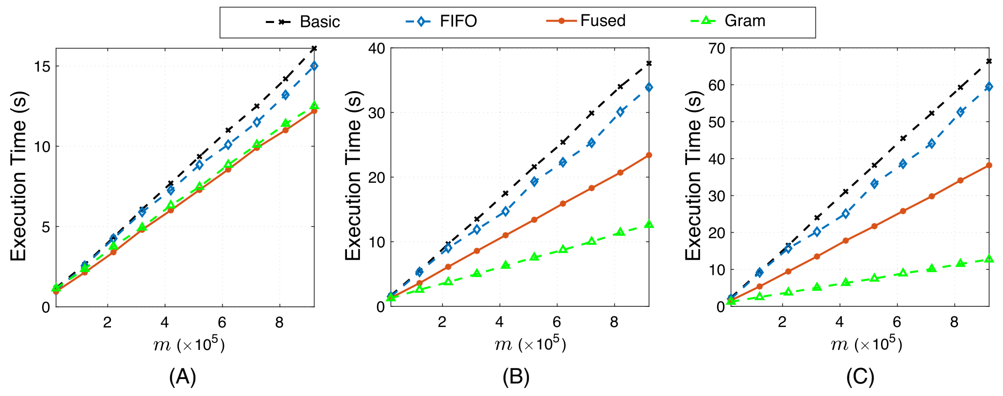

FIGURE 6 RSVD execution time a single Tesla V100 GPU with different numbers of rows m. Results for power iteration counts, A, $q = 1$, B, $q = 4$, and C, $q = 8$. Matrix sizes were $n = 5000$ and $\ell = 500$. The results are shown in execution time instead of flop/s because the flop counts of the Gram method are different from others. CPU-based results are omitted to focus on GPU-based results. GPU, graphics processing unit; RSVD, randomized singular value decomposition

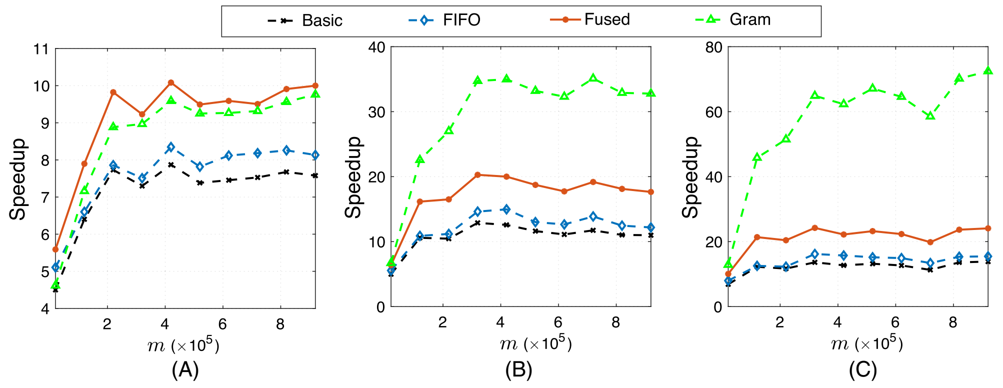

FIGURE 7 Speedup of one Tesla V100 GPU over two Xeon Silver 4114 CPUs with different numbers of rows m. Results for power iteration counts, A, $q = 1$, B, $q = 4$, and C, $q = 8$. Matrix sizes were $n = 5000$ and $\ell = 500$. For each method, we executed a multithreaded version of Algorithm 1 on two CPUs to compute the speedup. GPU, graphics processing unit

Regarding the performance of the Gram method, the execution time for $m = 9.2 \times 10^5$ remained at approximately 12 s, which was independent of q. In fact, the proportions of GEMM in the Gram method maintained at 93% from $q = 1$ to $q = 8$ with $m = 9.2 \times 10^5$. With $q = 1$, the Gram method performed slightly worse than the Fused method due to increased computation cost. Comparatively, the execution time of the Fused method increased from 12.2 s in Figure 6A to 38.2 s in Figure 6B with $m = 9.2 \times 10^5$. With $q = 8$ and $m = 9.2 \times 10^5$ in Figure 6C, the execution times of the Gram and Fused methods were 12.7 s and 38.2 s, respectively. This 3.0× speedup of the Gram method was achieved by further reducing D from $(q + 1)mn$ to $2mn$ via Gram matrix computation. Overall, the Gram method improved the performance by up to 5.2× over the basic method.

Figure 7 shows the speedup of one GPU over two CPUs with different numbers of rows m. The speedup gradually increased for all methods as we increased q, because the proportion of GEMM increased with q and the GPU performed better than CPUs for GEMM operations. The Gram method achieved up to 70× speedup over two CPUs with $q = 8$ in Figure 7C.

Figure 8 shows that all proposed methods achieved similar speedups over one GPU, demonstrating efficient scaling on two GPUs. However, when $m \leq 1 \times 10^5$, the speedups were at most 1.7× because the GPU initialization time surpassed the performance gain provided by two GPUs.

We next focus on the Fused and Gram methods to investigate the performance with different numbers of columns n and different power iteration count q (Figure 9). As shown in Figure 9A, the Gram performance was independent from q, as explained in Section 5.3. Thus, the additional computational cost needed for forming the Gram matrix had a limited impact on the GPU-based RSVD performance. In fact, the number ($qn^2\ell$) of floating-point operations for power iteration is much smaller than that (mn^2) for forming the Gram matrix $\mathbf{G}$. Consequently, the measured run time for power iteration was less than 1% of that for forming the Gram matrix on our experimental machine.

As shown in Figure 9A, for a small power iteration number ($q = 1$), the Fused method outperformed the Gram method. In particular, their gap increased with n. The reason for this behavior is that the Gram method has a higher computation cost ($mn^2 + n^2\ell + mn\ell$) than the Fused method ($3mn\ell$) when $q = 1$. In particular, the computation cost for forming the Gram matrix, that is, mn^2, made the Gram method slower compared with the Fused method when $n \geq 1 \times 10^4$. As for the data transfer cost D, when $q = 1$, there was no significant difference between the Fused and Gram methods; the data transfer cost for both methods was approximately $2mn$ when $q = 1$ (Table 1). However, when we increased the power

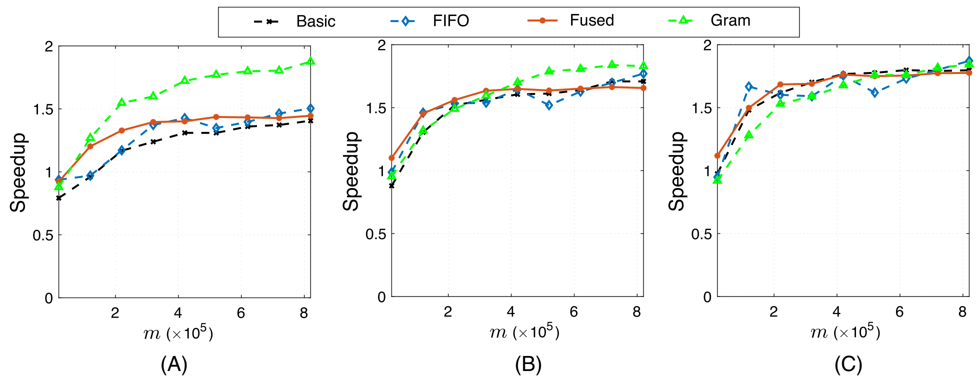

FIGURE 8 Speedup of two Tesla V100 GPUs over one V100 GPU with different numbers of rows m. Results for power iteration counts, A, $q = 1$, B, $q = 4$, and C, $q = 8$. Matrix sizes were $n = 5000$ and $\ell = 500$. For each method, we executed the same method on a single GPU to compute the speedup. GPU, graphics processing unit

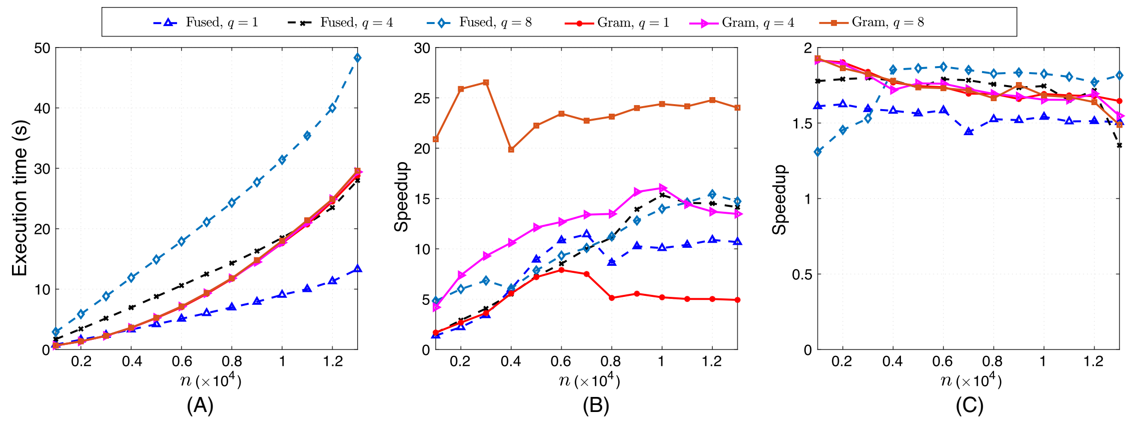

FIGURE 9 RSVD performance comparison of Fused and Gram methods with different numbers of columns n and different power iteration count q. A, Execution time on a single Tesla V100 GPU, B, speedup of one GPU over two Xeon Silver 4114 CPUs, and C, speedup of two GPUs over one GPU. Matrix sizes were $m = 4 \times 10^5$ and $\ell = n/10$. We executed a multithreaded version of Algorithm 1 to compute the speedup over two CPUs in B. For each method in C, the same method was executed to compute the speedup over a single GPU. GPU, graphics processing unit; RSVD, randomized singular value decomposition

iteration count to $q = 4$, the cost D for the Fused method increased to approximately $5mn$, whereas that for the Gram method remained $2mn$. Consequently, the Gram method was twice as fast as the Fused method when $q = 4$ and $n < 0.6 \times 10^4$ in Figure 9A. Thus, the Gram method is robust for large q values but sensitive to large n values, whereas the Fused method is robust for large n values but sensitive to large q values. Therefore, we think that the Gram method is useful especially when a large number of iterations is required to acquire a high-accuracy approximation.

Figure 9B shows the speedup of one GPU over two CPUs. The speedup of the Fused method was up to 17× for different q. Comparatively, the speedup of the Gram method gradually increased from 7× to 24× as we increased power iteration count from $q = 1$ to $q = 8$. Figure 9C shows the speedup of two GPUs over one GPU. The performance was slightly better than the growing height case in Figure 8, achieving 1.5 to 1.9× speedups.

Figure 10 shows the breakdown of the execution time obtained with the Gram method. We used two experimental setups: (a) increasing the height m of the matrix **A** with the fixed width and (b) increasing the width n of the matrix **A** with the fixed height. Figure 10A demonstrates that the proportion of GEMM execution gradually increased with m and reached 90%. This behavior was due to the computational cost of GEMM ($mn^2 + qn^2\ell + mn\ell$), which increases linearly with m, whereas that of SVD is fixed to $\mathcal{O}(n^3)$. Consequently, GEMM operations dominate the computation for extremely tall-skinny matrices.

By contrast, the proportion of SVD gradually increased with n and reached 24% of the total time when $n = 2.4{\times}10^4$ for **A** transformed from tall-skinny to square-like Figure 10B. This was due to the computational cost of SVD and GEMM, $\mathcal{O}(n^3)$ and $mn^2 + qn^2\ell + mn\ell$, respectively. Assuming that GEMM is parallelizable while other operations are not, parallel GEMM can achieve a linear speedup (Figure 10A). By contrast, the theoretical speedup for situations similar to that shown in Figure 10B is limited to at most 5× according to the Amdahl's law.[60]

6.2 | Performance comparison

We next compared the proposed methods with the previous methods in terms of the out-of-core RSVD performance. As a comparative method, we used cuBLAS-XT[50] as a GPU-accelerated method. According to our preliminary results reported in Section 4, we maximized

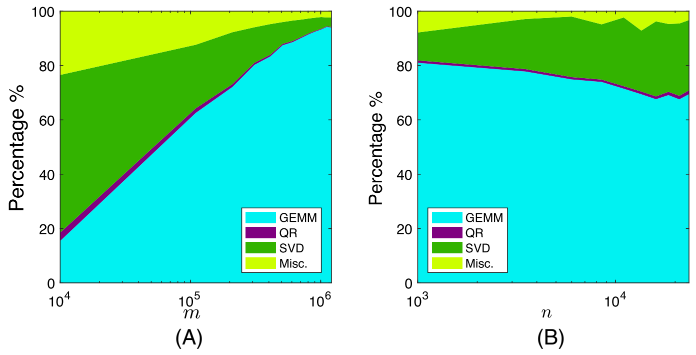

FIGURE 10 Breakdown of execution time on a single Tesla V100 GPU with $\ell = n/10$ and $q = 4$. Results of A, growing height m of **A** with fixed width $n = 5000$ and of B, growing width n of **A** with fixed height $m = 1 \times 10^5$. GEMM includes the time of CPU-GPU data transfer and GEMM operations. SVD denotes the deterministic SVD of matrix **B**. Misc. is composed of initialization and random matrix generation. GEMM, general matrix-matrix multiplication; GPU, graphics processing unit; SVD, singular value decomposition

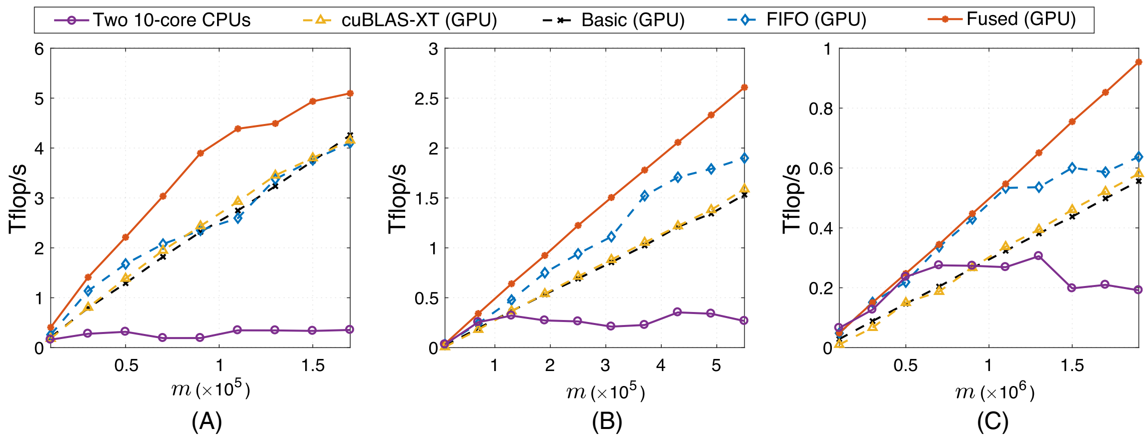

FIGURE 11 Comparison of CPU-based method, cuBLAS-XT,[50] and the proposed Fused method with different matrix shapes. A, Rectangular with $m : n : \ell = 100 : 10 : 1$, B, tall-skinny with $m : n : \ell = 1000 : 10 : 1$, and C, extremely tall-skinny with $m : n : \ell = 10000 : 10 : 1$. Results obtained on a single Tesla V100 GPU are presented in flop/s. Power iteration count was $q = 4$. The CPU-based method was multithreaded using two Xeon Silver 4114 CPUs. Note that the horizontal scale is different in three setups because we set the maximum input matrix size to the maximum memory size that our system can hold. GPU, graphics processing unit

the GEMM performance by applying the row-wise 1D partition to cuBLAST-XT. With respect to the proposed method, we used the Fused method to compare the RSVD performance in Tflop/s because all methods, except the Gram method, had the same number of floating-point operations.

We varied the matrix setup $m : n : \ell$ from $100 : 10 : 1$ to $10000 : 10 : 1$ to investigate the out-of-core RSVD performance (Figure 11). We first used a rectangular matrix setup where $m : n : \ell = 100 : 10 : 1$. Figure 11A demonstrates that all GPU-based methods achieved 4 Tflop/s, which outperformed the CPU-based method (approximately 0.35 Tflop/s); the basic, FIFO, and cuBLAS-XT methods showed similar performance, whereas the Fused method achieved the highest flop/s. The GPU-based methods achieved approximately 14× higher flop/s than the CPU-based method. This performance gap came from that the deployed CPUs provided a theoretical peak performance of 0.9 Tflop/s, which was one-fifteenth as compared with that of a single V100 GPU. The second setup in Figure 11B used tall-skinny matrices where $m : n : \ell = 1000 : 10 : 1$. The GPU performance dropped with the increasing ratio of $m : n$ and achieved a maximum of 2.6 Tflop/s. On the other hand, the CPUs still resulted in around 0.35 Tflop/s. The third setup in Figure 11C was obtained with the extremely tall-skinny matrices where $m : n : \ell = 10000 : 10 : 1$. The GPU performance dropped to 0.95 Tflop/s, but the achieved performance was still higher than that on the CPUs (0.3 Tflop/s). The CPU performance stagnated and dropped slightly after $m > 1.2 \times 10^6$. Thus, the CPU performance was less sensitive to input matrix shapes than the GPU performance. In summary, the GPU-based methods outperformed the CPU-based method for square-like input matrices. However, the performance advantage over multicore CPUs dropped as we increased the $m : n$ ratio. The main reason is that with the increasing $m : n$ ratio from $10 : 1$ to $1000 : 1$, the GEMM operations in RSVD get close to GEMV operations which are less efficient for GPUs.

We then used two GPUs to demonstrate the scalability of each implementation. As shown in Figure 12, the proposed methods scaled well on two GPUs, achieving speedups of up to 1.9× over one GPU. The performance gap between cuBLAS-XT and our methods increased significantly on two GPUs. The speedups of cuBLAS-XT were about 0.9 to 1.2× in all setups. While cuBLAS-XT was enforced to 1D partition, we failed to make cuBLAS-XT to perform similar to our 1D scheme that was free of reduction between GPUs. The excessive data transfer of both CPU-GPU and GPU-GPU was the main reason for the performance degradation.

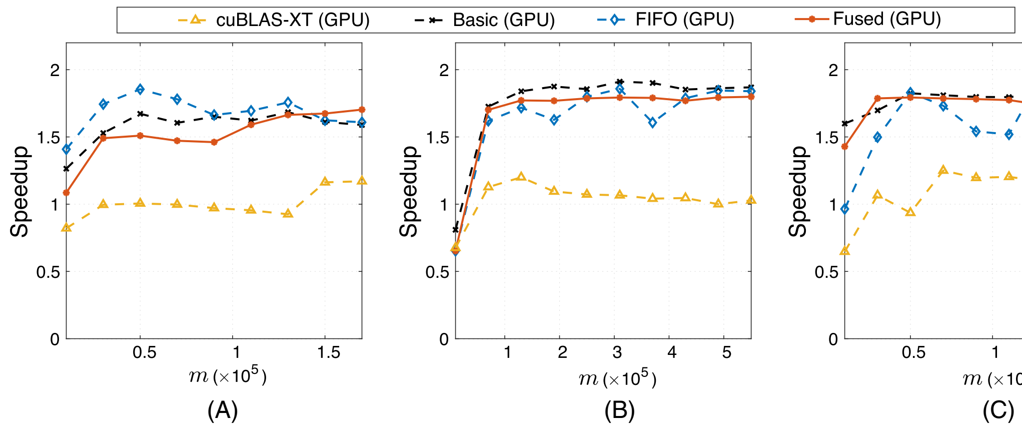

FIGURE 12 Speedup of two Tesla V100 GPUs over one V100 GPU with different matrix shapes: Results for, A, rectangular matrices with $m : n : \ell = 100 : 10 : 1$, B, tall-skinny matrices with $m : n : \ell = 1000 : 10 : 1$, and C, extremely tall-skinny matrices with $m : n : \ell = 10000 : 10 : 1$. Power iteration count was $q = 4$. For each method, the same method was executed to compute the speedup. GPU, graphics processing unit

TABLE 2 Comparison of approximation error $\left\|\mathbf{A}-\hat{\mathbf{A}}\right\|_F/\|\mathbf{A}\|_F$ with different SVD methods and different power iteration count q

		Original RSVD		Gram method	
Data	**Deterministic SVD**	$q = 1$	$q = 4$	$q = 1$	$q = 4$
Geometric distribution	0.5204	0.5297	0.5204	0.5302	0.5204
Exponential distribution	0.6622	0.6828	0.6623	0.6830	0.6623
FERET[61]	0.1661	0.1690	0.1661	0.1690	0.1661

Abbreviations: RSVD, randomized singular value decomposition; SVD, singular value decomposition.

6.3 | Numerical study with synthetic and real data

We used synthetic and real data to evaluate the numerical stability of the following three methods: (1) a deterministic SVD method provided by LAPACK,[40] (2) the original RSVD method[18] that orthogonalizes both **P** and **Q**, and (3) the proposed Gram method.

We used two different singular value distributions for the synthetic data: geometric and exponential distributions. For the geometric distribution, the *j-th* singular value σ_j was defined as $\sigma_j = \sigma_1 \gamma^{j-1}$ with the parameter $\gamma = 0.99$. For the exponential distribution, the *j-th* singular value σ_j was defined as $\sigma_j = \sigma_1 e^{-j/\beta}$ with the parameter $\beta = 160$. The matrix size $m \times n$ was set to 10000×5000 with the sampling parameters $k = 64$ and $o = 64$. The real data came from the facial recognition technology dataset[61] containing human face images. All 25 389 images were resized to the resolution of 512×768 pixels, and thus the input matrix consisted of 393216×25389 entries.

Table 2 shows the approximation error $\left\|\mathbf{A}-\hat{\mathbf{A}}\right\|_F / \|\mathbf{A}\|_F$, where $\|\cdot\|_F$ denotes the Frobenius norm. When $q = 4$, the Gram method achieved similar errors compared with those of the original RSVD and deterministic SVD methods. By contrast, when $q = 1$, there were small differences across these methods. Thus, increasing the number q of power iterations slightly improved the accuracy for randomized methods. Hence, the Gram method achieved the same level of accuracy as the deterministic SVD method without the orthogonalization of **P** (line 4 of Algorithm 1).

7 | CASE STUDY WITH RPCA

Finally, we demonstrate the performance of out-of-core RSVD with an RPCA application.[10] RPCA is a common method in computer vision and machine learning, which recovers a low-rank matrix with an unknown fraction of data corruption.[10] While being effective, RPCA algorithms are computationally demanding due to iterative SVD operations required to reveal the singular values for thresholding.

Algorithm 6 lists a pseudocode of the RPCA algorithm,[10,62] which separates the sparse corruptions **S** from the original data **M** so that a low-rank matrix **L** can be obtained as **L** = **M** − **S**. The problem can be written as

$$\min_{\mathbf{L},\mathbf{S}} ||\mathbf{L}||_* + \lambda||\mathbf{S}||_1 \quad \text{subject to} \quad \mathbf{M} = \mathbf{L} + \mathbf{S}, \tag{6}$$

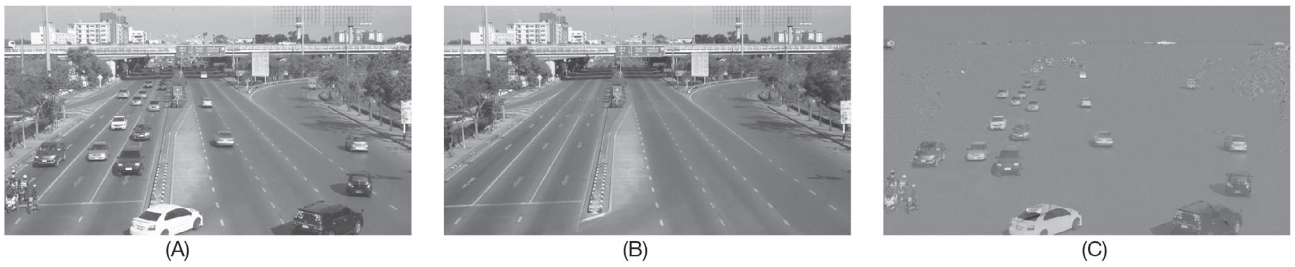

FIGURE 13 An execution example of RPCA. A, Input image, B, low-rank image, and C, sparse image. RPCA, robust principal component analysis

where $||\cdot||_*$ and $||\cdot||_1$ denote the nuclear norm and ℓ_1 norm of a matrix, respectively, and λ is a positive weighting parameter that is usually set to $\lambda = 1/\sqrt{\max(m,n)}$. Various solvers have been proposed for this convex optimization problem, and recent solution methods drastically improved the computation efficiency.[62-64] Similar to Oh et al,[12] we used an RSVD solver to replace the deterministic SVD solver in RPCA to accelerate the computation. The shrinkage operator $\mathcal{S}_\varepsilon[\cdot]$ in line 4 of Algorithm 6 is defined as

$$\mathcal{S}_\varepsilon[x] = \begin{cases} x-\varepsilon, & \text{if} \quad x > \varepsilon, \\ x+\varepsilon, & \text{if} \quad x < -\varepsilon, \\ 0, & \text{otherwise}, \end{cases} \tag{7}$$

where ε represents the threshold value.

Algorithm 6. RPCA by RSVD

```
Input : matrix $\mathbf{M} \in \mathbb{R}^{m\times n}$, target rank $k$, oversampling parameter $o$, power iteration count $q$, convergence condition $t$ and parameters $\mu_0$
        and $\rho$.
Output: low-rank matrix $\mathbf{L} \in \mathbb{R}^{m\times n}$ and sparse matrix $\mathbf{S} \in \mathbb{R}^{m\times n}$.
1  $\mathbf{Y}_0 = \mathbf{M} / \max(||\mathbf{M}||_2, \lambda^{-1}||\mathbf{M}||_\infty)$; $\mathbf{S}_0 = \mathbf{0}^{m\times n}$; $i = 0$;
2  while TRUE do
3      $[\mathbf{U}, \Sigma, \mathbf{V}^T] = \mathrm{rsvd}(\mathbf{M} - \mathbf{S}_i + {\mu_i}^{-1}\mathbf{Y}_i, k, o, q)$;
4      $\mathbf{L}_{i+1} = \mathbf{U}\mathcal{S}_{{\mu_i}^{-1}}[\Sigma]\mathbf{V}^T$;                 // shrinkage operation on Σ
5      $\mathbf{S}_{i+1} = \mathcal{S}_{\lambda{\mu_i}^{-1}}[\mathbf{M} - \mathbf{L}_{i+1} + {\mu_i}^{-1}\mathbf{Y}_i]$;
6      $\mathbf{Z}_{i+1} = \mathbf{M} - \mathbf{L}_{i+1} - \mathbf{S}_{i+1}$;
7      $\mathbf{Y}_{i+1} = \mathbf{Y}_i + \mu_i\mathbf{Z}_{i+1}$;
8      if ( $||\mathbf{Z}_{i+1}||_F / ||\mathbf{M}||_F < t$ ) break;               // evaluate convergence
9      $\mu_{i+1} = \rho\mu_i$;  $i = i + 1$;
10 end
```

In Algorithm 6, GEMM operations appear in lines 3 to 4, whereas all other computations are vector summations. GEMM operations have a higher operational intensity than vector summations, which cannot be efficiently offloaded to GPUs. Therefore, we decided to call LAPACK routines to implement vector summations on the CPU. On the other hand, we used the Gram method for RSVD in line 3 and the out-of-core GEMM with 1D partition for the reconstruction of **L** at line 4. Vector summations were implemented on the CPU with LAPACK routines.

All experiments were conducted in double precision using the same machine described in Section 4. The sampling parameters for RSVD were $k = 100$, $o = 100$, and $q = 4$. The parameters in Algorithm 6 were set as $t = 10^{-6}$, $\rho = 1.5$, $\mu_0 = 1.5/||\mathbf{M}||_\infty$, and $\varepsilon = \mu_i^{-1}$. As for data, we used a high definition traffic video that was heavily corrupted by camera jittering and a large traffic volume. We extracted 1000 frames with a resolution of 1920×1080. Therefore, the input matrix **M** is of size 2073600×1000. Figure 13 illustrates a sample frame output, which indicates that our implementation successfully separated the input data into sparse traffic noise and the low-rank background.

Table 3 compares the RPCA performance measured on the experimental machine. Compared with a multicore CPU implementation, our GPU-accelerated Gram method accelerated the RSVD computation by 23.3×, which halved the total RPCA time. This speedup is reasonable according to the out-of-core GEMM performance presented in Figure 7B; the acceleration on two GPUs over two 10-core CPUs was up to 35× for out-of-core RSVD. Figure 3A; the highest performances on two GPUs and two CPUs were about 2.5 and 0.22 Tflop/s, respectively, demonstrating a speedup of 11.4× for out-of-core GEMM. The highest performances on two GPUs and two CPUs were about 2.5 and 0.22 Tflop/s, respectively,

TABLE 3 Execution time of RPCA based on two Xeon Silver 4114 CPUs and two Tesla V100 GPUs

Breakdown	CPU (s)	GPU (s)
RSVD	1120	48
Misc.	951	46
Total RPCA	2071	994

Note: Both implementations converged with the same number (28) of iterations.
Abbreviations: GPU, graphics processing unit; RPCA, robust principal component analysis; RSVD, randomized singular value decomposition.

demonstrating a speedup of 11.4× for out-of-core GEMM in Figure 3A. However, the maximum performance of two Tesla V100 GPUs was close to 14 Tflop/s (Figure 3B), implying that the data size was not sufficiently large to maximize the performance on the GPU. By accelerating RSVD with GPUs, the performance bottleneck of RPCA moved to the vector summation part, which has a low operational intensity. The acceleration of out-of-core algorithms with a low operational intensity, such as vector summation, is still limited by the CPU-GPU transfer bandwidth, which is a challenge to fully utilize GPUs.

8 | CONCLUSION

Over the past decade, randomized algorithms have shown significant advancements in terms of the computation efficiency. However, there have not been many attempts to harness the modern computing architectures such as GPU accelerators. The likely explanation is that GPUs are still considered as specialized devices. At the same time, large-scale computing is now performed on supercomputers that are prevalently equipped with accelerators; hence, developing new algorithms for evolving computing architectures is becoming urgent.

We studied GPU-accelerated methods, namely, Fused and Gram, to reduce out-of-core data access for computing randomized SVD. The Gram method, which was especially effective for tall-skinny matrices, achieved up to 5.2× speedup compared with a straightforward method that deploys the highly tuned GEMM scheme and the 1D data partition scheme. The Fused method effectively accelerated the RSVD up to 1.7× compared with the straightforward method. This work allows us to see the directions of the randomized algorithm development as we move toward exascale computing. Most immediate direction is the independent block operations which fit accelerators.

Our analysis and empirical results also revealed that CPU-GPU transfer bandwidth limits the RSVD performance on a common workstation, especially for tall-skinny matrices that limits the scalability on GPUs. This constraint is expected to be mitigated with the introduction of the next generation PCIe and NVlink buses. Nevertheless, reducing the amount of data access and communication remains the major challenge in developing scalable linear algebra algorithms for both single- and multi-node systems. This is also our main focus in the future.

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 the reviewers for their valuable comments.

ORCID

Yuechao Lu https://orcid.org/0000-0002-8808-1559

REFERENCES

1. Golub GH, Van Loan CF. *Matrix Computations*. 4th ed. Baltimore, Maryland: Johns Hopkins University Press; 2012.
2. Golub G, Kahan W. Calculating the singular values and pseudo-inverse of a matrix. *J Soc Ind Appl Math Ser B Numer Anal*. 1965;2(2):205-224.
3. Larsen RM. Lanczos bidiagonalization with partial reorthogonalization. *DAIMI Rep Ser*. 1998;27(537):1-101.
4. Frieze A, Kannan R, Vempala S. Fast Monte-Carlo algorithms for finding low-rank approximations. *J ACM*. 2004;51(6):1025-1041.
5. Troyanskaya O, Cantor M, Sherlock G, et al. Missing value estimation methods for DNA microarrays. *Bioinformatics*. 2001;17(6):520-525.
6. Wall Michael E., Rechtsteiner Andreas, Rocha Luis M.. Singular value decomposition and principal component analysis. Paper presented at: Proceedings of the 27th International Conference for High Performance Computing, Networking, Storage and Analysis (SC); 2003:91-109.
7. Frankenberg C, O'Dell C, Berry J, et al. Prospects for chlorophyll fluorescence remote sensing from the orbiting carbon observatory-2. *Remote Sens Env*. 2014;147:1-12.
8. Höcker A, Kartvelishvili V. SVD approach to data unfolding. *Nucl Instrum Methods Phys Res, Sect A*. 1996;372(3):469-481.
9. Zhang Qiang, Li Baoxin. Discriminative K-SVD for dictionary learning in face recognition. Paper presented at: Proceedings of the 23rd IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2010:2691-2698; IEEE.

10. Candès Emmanuel J., Li Xiaodong, Ma Yi, Wright John. Robust principal component analysis?. *J ACM*. 2011;58(3):1-37. 11.
11. Liu Baiyang, Huang Junzhou, Yang Lin, Kulikowsk Casimir. Robust tracking using local sparse appearance model and *K*-selection. Paper presented at: Proceedings of the 24th IEEE Conference on Computer Vision and Pattern Recognition (CVPR):1313-1320; 2011:IEEE.
12. Oh, T. H., Matsushita, Y., Tai, Y. W., & So Kweon, I. Fast randomized singular value thresholding for nuclear norm minimization. Paper presented at: Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2015:4484-4493; IEEE.
13. Andrews H, Patterson C. Singular value decomposition (SVD) image coding. *IEEE Trans Commun*. 1976;24(4):425-432.
14. Turk Matthew A., Pentland Alex P.. Face recognition using eigenfaces. Paper presented at: Proceedings of the 4th IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR); 1991:586-591; IEEE.
15. Henry E.R., Hofrichter J. Singular value decomposition: Application to analysis of experimental data. *Methods in Enzymology*. 1992;210:(8), 129–192.
16. Martinsson P-G, Rockhlin V, Tygert M. *A Randomized Algorithm for the Approximation of Matrices YALEU/DCS/TR-1361*. New Haven, CT: Yale University, Department of Computer Science; 2006.
17. Liberty E, Woolfe F, Martinsson P-G, Rokhlin V, Tygert M. Randomized algorithms for the low-rank approximation of matrices. *Proc Nat Acad Sci U S Am (PNAS)*. 2007;104(51):20167-20172.
18. Halko N, Martinsson P-G, Tropp JA. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev*. 2011;53(2):217-288.
19. Drineas P, Kannan R, Mahoney MW. Fast Monte Carlo algorithms for matrices i: approximating matrix multiplication. *SIAM J Comput*. 2006;36(1):132-157.
20. Kuczynski J, Wozniakowski H. Estimating the largest eigenvalue by the power and lanczos algorithms with a random start. *SIAM J Matrix Anal Appl*. 1992;13(4):1094-1122.
21. Rokhlin V, Szlam A, Tygert M. A randomized algorithm for principal component analysis. *SIAM J Matrix Anal Appl*. 2009;31(3):1100-1124.
22. Gates M, Tomov S, Dongarra J. Accelerating the SVD two stage bidiagonal reduction and divide and conquer using GPUs. *Parallel Comput*. 2018;74:3-18.
23. Dongarra J, Gates M, Haidar A, et al. The singular value decomposition: anatomy of optimizing an algorithm for extreme scale. *SIAM Rev*. 2018;60(4):808-865.
24. NVIDIA Corporation. NVIDIA Tesla V100 GPU Architecture; 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.
25. Haidar Azzam, Kabir Khairul, Fayad Diana, Tomov Stanimire, Dongarra Jack. Out of memory SVD solver for big data. Paper presented at: Proceedings of the 21st High Performance Extreme Computing Conference (HPEC); 2017:1-7; IEEE.
26. Demmel James. Communication-avoiding algorithms for linear algebra and beyond. Paper presented at: Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2013:585; IEEE.
27. Shen Jingcheng, Shigeoka Kentaro, Ino Fumihiko, Hagihara Kenichi. GPU-based branch and bound method to solve large 0-1 knapsack problems with data-centric strategies. *Concurr Comput Pract Exp*. 2019;31(4):1-15. e4954.
28. Hoemmen Mark. Communication-avoiding Krylov subspace methods (PhD thesis). Berkeley, CA: University of California, Berkeley; 2010.
29. Anderson Michael, Ballard Grey, Demmel James, Keutzer Kurt. Communication-avoiding QR decomposition for GPUs; 2011:48-58; IEEE.
30. Drineas P, Mahoney MW. A randomized algorithm for a tensor-based generalization of the singular value decomposition. *Linear Algebra Appl*. 2007;420(2/3):553-571.
31. Yu Wenjian, Gu Yu, Li Jian, Liu Shenghua, Li Yaohang. Single-pass PCA of large high-dimensional data; 2017:3350-3356.
32. Tropp JA, Yurtsever A, Udell M, Cevher V. Practical sketching algorithms for low-rank matrix approximation. *SIAM J Matrix Anal Appl*. 2017;38(4):1454-1485.
33. De la Torre F, Black MJ. A framework for robust supspace learning. *Int J Comput Vis*. 2003;54(1/2/3):117-142.
34. Yamazaki Ichitaro, Kurzak Jakub, Luszczek Piotr, Dongarra Jack. Random sampling to update partial singular value decomposition on a hybrid CPU/GPU cluster; 2015:59; IEEE/ACM.
35. Voronin Sergey, Martinsson Per-Gunnar. RSVDPACK: an implementation of randomized algorithms for computing the singular value, interpolative, and CUR decompositions of matrices on multi-core and GPU architectures; 2016:2. arXiv preprint arXiv:1502.05366v3.
36. Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Commun ACM*. 2009;52(4):65-76.
37. Agullo E, Demmel J, Dongarra J, et al. Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *J Phys Conf Ser*. 2009;180:012037.
38. Smith BT, Boyle JM, Dongarra JJ, et al. Matrix eigensystem routines — EISPACK guide. *Lect Notes Comput Sci*. 1977;6:1-366.
39. Dongarra JJ, Moler CB, Bunch JR, Stewart GW. *LINPACK Users' Guide*. Philadelphia, Pennsylvania: SIAM; 1979.
40. Anderson E, Bai Z, Bischof C, et al. *LAPACK Users' Guide*. 3rd ed. Philadelphia, Pennsylvania: SIAM; 1987.
41. Strang G. *Introduction to Linear Algebra*. 5th ed. Cambridge, MA: Wellesley-Cambridge Press; 2016.
42. Sarlós Tamás. Improved Approximation Algorithms for Large Matrices via Random Projections; 2006:143-152; IEEE.
43. Mahoney MW. Randomized algorithms for matrices and data. *Found Trends® Mach Learn*. 2011;3(2):123-224.
44. Ji Hao, Li Yaohang. GPU accelerated randomized singular value decomposition and its application in image compression. Paper presented at: Proceedings of the Modeling, Simulation, and Visualization Student Capstone Conference; 2014:39-45.
45. Mary Théo, Yamazaki Ichitaro, Kurzak Jakub, Luszczek Piotr, Tomov Stanimire, Dongarra Jack. Performance of random sampling for computing low-rank approximations of a dense matrix on GPUs; 2015:60; IEEE/ACM.
46. Volkov Vasily, Demmel James W. Benchmarking GPUs to tune dense linear algebra; 2008:31; IEEE/ACM.
47. D'Azevedo E, Hill JC. Parallel LU factorization on GPU cluster. *Proc Comput Sci*. 2012;9:67-75.
48. Yamazaki I, Tomov S, Dongarra J. One-sided dense matrix factorizations on a multicore with multiple GPU accelerators. *Proc Comput sci*. 2012;9:37-46.
49. Yamazaki Ichitaro, Tomov Stanimire, Dongarra Jack. Non-GPU-resident symmetric indefinite factorization. *Concurr Comput Pract Exp*. 2017;29(5):1-12. e4012.
50. NVIDIA Corporation. Cublas library user guide; 2019. http://docs.nvidia.com/pdf/CUBLAS_Library.pdf.
51. Wang Linnan, Wu Wei, Xu Zenglin, Xiao Jianxiong, Yang Yi. BLASX: a high performance level-3 BLAS library for heterogeneous multi-GPU computing; 2016:20; ACM.
52. NVIDIA Corporation. CUDA C++ programming guide; 2019. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

53. Erichson NB, Donovan C. Randomized low-rank dynamic mode decomposition for motion detection. *Comput Vis Image Underst*. 2016;146:40-50.
54. Intel Corporation. Developer reference for intel® math kernel library – C; 2019. https://software.intel.com/en-us/mkl-developer-reference-c/.
55. Williams Samuel, Oliker Leonid, Vuduc Richard, Shalf John, Yelick Katherine, Demmel James. Optimization of sparse matrix-vector multiplication on emerging multicore platforms; 2007:38; IEEE/ACM.
56. Haidar A, Abdelfattah A, Zounon M, Tomov S, Dongarra J. A guide for achieving high performance with very small matrices on GPU: a case study of batched LU and Cholesky factorizations. *IEEE Trans Parall Distrib Syst*. 2018;29(5):973-984.
57. Chen Jieyang, Xiong Nan, Liang Xin, et al. TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs; 2019:106-116; ACM.
58. Stathopoulos A, Wu K. A block orthogonalization procedure with constant synchronization requirements. *SIAM J Sci Comput*. 2002;23(6):2165-2182.
59. OpenMP Architecture Review Board. OpenMP application programming interface, version 5.0; 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.
60. Amdahl Gene M.. Validity of the single processor approach to achieving large scale computing capabilities; 1967:483-485; AFIPS Press.
61. Phillips P. Jonathon, Moon Hyeonjoon, Rauss Patrick, Rizvi Syed A. The FERET evaluation methodology for face-recognition algorithms; 1997:137-143; IEEE.
62. Lin Zhouchen, Chen Minming, Ma Yi. The augmented lagrange multiplier method for exact recovery of corrupted low-rank matrices; 2013. arXiv preprint arXiv:1009.5055v3.
63. Yuan X, Yang J. Sparse and low-rank matrix decomposition via alternating direction methods. *Pacific J Optimiz*. 2013;9(1):167-180.
64. Lin Z, Ganesh A, Wright J, Wu L, Chen M, Ma Y. *Fast Convex Optimization Algorithms for Exact Recovery of a Corrupted Low-Rank Matrix UILU-ENG-09-2214*. Champaign, IL: University of Illinois at Urbana-Champaign; 2009.

How to cite this article: Lu Y, Yamazaki I, Ino F, Matsushita Y, Tomov S, Dongarra J. Reducing the amount of out-of-core data access for GPU-accelerated randomized SVD. *Concurrency Computat Pract Exper*. 2020;e5754. https://doi.org/10.1002/cpe.5754