# A GPU-based Approximate SVD Algorithm

Blake Foster, Sridhar Mahadevan, and Rui Wang

Department of Computer Science
Univ. of Massachusetts, Amherst, MA 01003, USA
{blfoster,mahadeva,ruiwang}@cs.umass.edu

**Abstract.** Approximation of matrices using the Singular Value Decomposition (SVD) plays a central role in many science and engineering applications. However, the computation cost of an exact SVD is prohibitively high for very large matrices. In this paper, we describe a GPU-based approximate SVD algorithm for large matrices. Our method is based on the QUIC-SVD introduced by [6], which exploits a tree-based structure to efficiently discover a subset of rows that spans the matrix space. We describe how to map QUIC-SVD onto the GPU, and improve its speed and stability using a blocked Gram-Schmidt orthogonalization method. Results show that our GPU algorithm achieves 6∼7 times speedup over an optimized CPU version of QUIC-SVD, which itself is orders of magnitude faster than exact SVD methods. Using a simple matrix partitioning scheme, we have extended our algorithm to out-of-core computation, suitable for very large matrices that exceed the main memory size.

**Keywords:** SVD, GPU, cosine trees, out-of-core computation

## 1 Introduction

The Singular Value Decomposition (SVD) is a fundamental operation in linear algebra. Matrix approximation using SVD has numerous applications in data analysis, signal processing, and scientific computing. Despite its popularity, the SVD is often restricted by its high computation cost, making it impractical for very large datasets. In many practical situations, however, computing the full-matrix SVD is not necessary; instead, we often need only the $k$ largest singular values, or an approximate SVD with controllable error. In such cases, an algorithm that computes a low-rank SVD approximation is sufficient, and can significantly improve the computation speed for large matrices.

A series of recent work has studied matrix sampling to solve the low-rank matrix approximation (LRMA) problem. These algorithms construct a basis made up of rows or linear combinations of rows sampled from the matrix, such that the projection of the matrix onto the basis has bounded error. Sampling-based methods include length-squared sampling [4, 2] and random projection sampling [3, 11]. In this paper, we focus on a particular method called QUIC-SVD, recently introduced by [6]. QUIC-SVD exploits a tree-based structure to perform fast sampled-based SVD approximation with automatic error control.

The main benefit compared to previous work is that it iteratively selects samples that are both adaptive and representative.

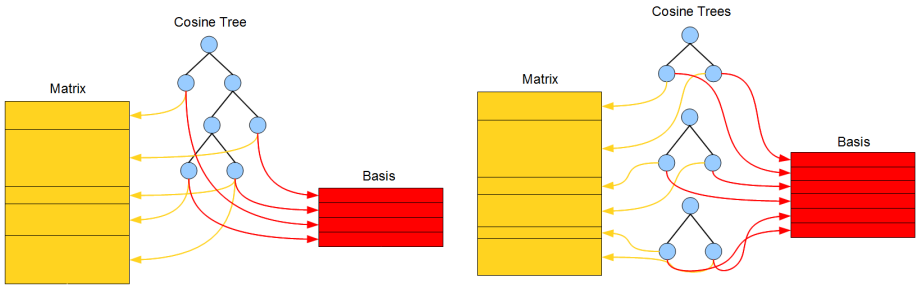
Our goal is to map the QUIC-SVD algorithm onto the graphics processing unit (GPU) to further improve its efficiency. Modern GPUs have emerged as low-cost massively parallel computation platforms that provide very high floating point performance and memory bandwidth. In addition, the availability of high-level programming languages such as CUDA has significantly lowered the programming barrier for the GPU. These features make the GPU a suitable and viable solution for solving many computationally intensive tasks in scientific computing. We describe how we implemented the QUIC-SVD algorithm on the GPU, and demonstrate its speedup (about 6∼7 times) over an optimized CPU version, which itself is orders of magnitude faster than exact SVD methods. We also describe a matrix partitioning scheme that easily adapts the algorithm to out-of-core computation, suitable for very large matrices. We have tested our algorithm on dense matrices up to 22,000× 22,000, as reported in Section 3.

**Related Work.** Acceleration of matrix decomposition algorithms on modern GPUs has received significant attention in recent years. Galoppo et al. [5] reduced matrix decomposition and row operations to a series of rasterization problems on the GPU, and Bondhugula et al. [1] provided a GPU-based implementation of SVD using fragment shaders and frame buffer objects. Since then, the availability of general programming language such as CUDA has made it possible to program the GPU without relying on the graphics pipeline. In [12], a number of matrix factorization methods are implemented using CUDA, including LU, QR and Cholesky, and considerable speedup is achieved over optimized CPU algorithms. GPU-based QR decomposition was also studied by [8] using blocked Houserholder reflections. Recently, Lahabar et al. [9] presented a GPU-based SVD algorithm built upon the Golub-Reinsch method. They achieve up to 8× speedup over an Intel MKL implementation running on dual core CPU. Like most existing work (including commercial GPU-based linear algebra toolkit such as CULA [7]), their focus is on solving the exact SVD. In contrast, our goal is to solve approximate SVD on the GPU, which can provide additional performance gain for many large-scale problems in practical applications.

## 2 Algorithm

### 2.1 Overview

Given an $m \times n$ matrix $A$ (where $n$ is the smaller dimension), the SVD factors the matrix into the product of three matrices: $A = U\Sigma V^{\mathrm{T}}$ where $U$ and $V$ are both orthonormal matrices ($U^{\mathrm{T}}U = I$ and $V^{\mathrm{T}}V = I$) and $\Sigma$ is a diagonal matrix storing the singular values. An exact SVD takes $O(mn^2)$ time to compute and thus is expensive for large matrices. To approximate the SVD, we can construct a subspace basis that captures the intrinsic dimensionality of $A$ by sampling rows or taking linear combinations of rows. The final SVD can be extracted by performing an exact SVD on the subspace matrix, which is a much smaller




(a) Single cosine tree (b) Partitioned cosine trees

**Fig. 1.** *(a) shows a single cosine tree; (b) shows a set of cosine trees constructed using our partitioning scheme, such that all trees collectively build a common basis set. Yellow arrow indicates the matrix rows that a tree node owns; red arrow indicates a vector inserted into the basis set, which is the mean vector of the rows owned by a node.*

than the original matrix. For example, if the intrinsic dimensionality of $A$ is approximately $k$, where $k \ll n$, the computation cost is now reduced to $O(mnk)$.

The QUIC-SVD [6] is a sample-based approximate SVD algorithm. It iteratively builds a row subspace that approximates $A$ with controlled L2 error. The basis construction is achieved using a binary tree called *cosine tree*, as shown in Figure 1(a), where each node represents a collection (subset) of the matrix rows. To begin, a root node is built that represents all rows (i.e. the entire matrix $A$), and the mean (average) vector of the rows is inserted into the initial basis set. At each iteration, a leaf node $\mathbf{n}_s$ in the current tree is selected for splitting. The selection is based on each node's estimated error, which predicts whether splitting a node is likely to lead to a maximal reduction in the matrix approximation error. To perform the splitting, a pivot row $\mathbf{r}_p$ is sampled from the selected node $\mathbf{n}_s$ according to the length-squared distribution. Then, $\mathbf{n}_s$ is partitioned into two child nodes. Each child node owns a subset of the rows from $\mathbf{n}_s$, selected by their dot products with the pivot row. Specifically, rows closer to the minimum dot product value are inserted into the left child, and the remaining nodes are inserted into the right child. Finally, the mean vector of each subset is added to the basis set, replacing the mean vector contributed by the parent node.

Figure 1(a) shows the cosine tree constructed at an intermediate step of the algorithm. Each leaf node represents a subset of rows, and contributes a mean vector to the current basis set. As the tree is split further, the basis set expands. The process terminates when the whole matrix approximation error is estimated to fall below a relative threshold:

$$\|A - \widehat{A}\|_F^2 = \|A - A\widehat{V}\widehat{V}^{\mathrm{T}}\|_F^2 \leq \epsilon \|A\|_F^2$$

where $\widehat{V}$ is the row basis, $\widehat{A} = A\widehat{V}\widehat{V}^{\mathrm{T}}$ is the reconstructed matrix with the approximate SVD, and $\|\cdot\|_F$ denotes the Frobenius norm. This error is calculated using a Monte Carlo estimation routine, which evaluates the projection error of $A$ on to the row basis set $\widehat{V}$. The error estimation routine returns an upper

bound on the error with probability $1 - \delta$, where $\delta$ is an adjustable parameter. This routine is also used to estimate the error contributed by a node, in order to prioritize the selection of nodes for splitting as described above. Intuitively, nodes with large error are not well-approximated by the current basis, and thus splitting them is likely to yield the largest benefit.

Whenever a vector is inserted into the basis set, it is orthogonalized against the existing basis vectors using Gram-Schmidt orthogonalization. This is necessary for the Monte Carlo error estimation and SVD extraction. Once the tree building terminates, the current basis set accurately captures the row subspace of $A$, and the final SVD can be extracted from the basis set by solving a much smaller SVD problem.

In summary, the main computation loop involves the following steps: 1) select a leaf node with the maximum estimated error; 2) split the node and create two child nodes; 3) the mean vector of each child is inserted into the basis set and orthonormalized (while the one contributed by the parent is removed); 4) estimate the error of each child node; 5) estimate the error of the whole matrix approximation, and terminate when it's sufficiently small. For more details, we refer the reader to [6].

### 2.2 GPU Implementation

We implemented QUIC-SVD using the CUDA programming language, in conjunction with the CULA [7] library to extract the final SVD. We found that most of the computation is spent on the following two parts: 1) computing vector inner products and row means for node splitting; 2) Gram-Schmidt orthogonalization.

When we split a node, we need to compute the inner product of every row with the pivot row (which is selected by sampling length-squared distribution of the rows). Since a node does not necessarily span contiguous rows, we could not use a simple matrix-vector multiplication call to accomplish this step. Rearranging the rows of each node into contiguous chunks after each split was not an option, as this would incur excessive memory traffic. Instead, we maintain an index array at each node to point to the rows that the node owns, and then use a custom CUDA kernel to compute all inner products in parallel. This can be seen as a special case of sparse matrix and vector multiplication. Next, the rows are split into two subsets based on their inner product values, which can be done using parallel sorting. For each subset we again use a custom CUDA kernel to compute the mean vector, which will be inserted into the basis set.

When we add a new mean vector to the basis, it must be orthonormalized with respect to the existing basis vectors with the Gram-Schmidt process. Given a set of orthonormal basis vectors $\mathbf{v}_1, ..., \mathbf{v}_k$ and a new basis vector $\mathbf{r}$, the classical Gram-Schmidt process would compute

$$\mathbf{r}' = \mathbf{r} - p_{\mathbf{v}_1}(\mathbf{r}) - ... - p_{\mathbf{v}_k}(\mathbf{r}), \quad \text{and} \quad \mathbf{v}_{k+1} = \mathbf{r}'/\|\mathbf{r}'\|$$

where $p_{\mathbf{v}}(\mathbf{r}) = (\mathbf{r} \cdot \mathbf{v})\,\mathbf{v}$ denotes the projection of $\mathbf{r}$ onto $\mathbf{v}$. Both the projection and subtraction can be done in parallel, but the numerical stability is poor. The

modified Gram-Schmidt process subtracts the projection vector sequentially:

$$\mathbf{r}_1 = \mathbf{r} - p_{\mathbf{v}_1}(\mathbf{r}); \quad \mathbf{r}_2 = \mathbf{r}_1 - p_{\mathbf{v}2}(\mathbf{r}_1); \quad ... \quad \mathbf{r}' = \mathbf{r}_k = \mathbf{r}_{k-1} - p_{\mathbf{v}_k}(\mathbf{r}_{k-1})$$

This is mathematically the same, but the numerical stability is improved greatly. Unfortunately, this formulation serializes the computation and cannot be easily parallelized.

To exploit the benefits of both, we propose to use a *blocked* Gram-Schmidt process, which involves partitioning the basis vectors into $\kappa$ blocks (subsets). Within each block, we use the classical Gram-Schmidt to gain parallelism; and across blocks we use the modified Gram-Schmidt to gain numerical stability. Specifically, assume the current set of basis vectors is partitioned into the following $\kappa$ blocks: $V_1, ..., V_\kappa (\kappa \ll k)$. We will then compute

$$\mathbf{u}_1 = \mathbf{r} - \mathrm{GS}(\mathbf{r}, V_1),\ \mathbf{u}_2 = \mathbf{u}_1 - \mathrm{GS}(\mathbf{u}_1, V_2),\ ...\ \mathbf{u}_\kappa = \mathbf{u}_{\kappa-1} - \mathrm{GS}(\mathbf{u}_{\kappa-1}, V_\kappa) = \mathbf{r}'$$

where $\mathrm{GS}(\mathbf{u}, V)$ denotes the standard Gram Schmidt orthogonalization of $\mathbf{u}$ with respect to basis subset $V$. Note that when $\kappa = 1$ or $\kappa = k$, the algorithm degenerates to the classical or the modified Gram-Schmidt respectively. We set $\kappa$ such that each block contains approximately 20 basis vectors, and we have found that this provides a good tradeoff between speed and numerical stability.

Among the other steps, the Monte Carlo error estimation is straightforward to implement on the GPU, and its cost is insignificant. Selecting a splitting node is achieved with a priority queue [6] maintained on the CPU. The extraction of the final SVD is performed with the CULA toolkit. Again, the cost of this step is insignificant as it only involves computing the SVD of a $k \times k$ matrix.

### 2.3 Partitioned Version

To accommodate large datasets, we introduce a partitioned version of the algorithm that can process matrices larger than GPU or even main memory size. While the original QUIC-SVD algorithm [6] did not consider out-of-core computation, we found that the structure of the cosine tree lends itself naturally to partitioning. To begin, we split the matrix $A$ into $s$ submatrices $A_1, ..., A_s$, each containing $\lceil m/s \rceil$ consecutive rows from $A$. Next, we run QUIC-SVD on each submatrix $A_i$ sequentially. A naive algorithm would then simply merge the basis set constructed for each submatrix $A_i$ to form a basis for the whole matrix. While this would give correct results, it introduces a lot of redundancy (as each basis set is computed independently), and consequently reduce efficiency.

We make a small modification to the algorithm to eliminate redundancy. We build an individual cosine tree for each submatrix $A_i$, but all submatrices share a common basis set. The algorithm processes the submatrices sequentially in order. When processing submatrix $A_i$, the corresponding matrix rows are loaded into GPU memory, and a new cosine tree is constructed. The basis set from previous submatrices is used as the initial basis. If the error estimated from this basis is already below the given threshold, the algorithm will stop immediately and proceed to the next submatrix. Intuitively this means the submatrix $A_i$ is

already well represented by the current basis set, hence no update is necessary. Otherwise, the algorithm processes $A_i$ in the same way as the non-partitioned version, and the basis set is expanded accordingly. Once we are done with the current submatrix, the GPU memory storing the matrix rows is overwritten with the next submatrix.

After a complete pass through every subset, we observe that the whole matrix approximation error is equal to the sum of the each subset's approximation error, which is bounded by the given relative error threshold. In other words:

$$\|A - \widehat{A}\|_F^2 = \sum_{i=1}^{s} \|A_i - \widehat{A}_i\|_F^2 \leq \sum_{i=1}^{s} \epsilon \|A_i\|_F^2 = \epsilon \|A\|_F^2$$

where $\widehat{A}_i = A_i \widehat{V} \widehat{V}^{\mathrm{T}}$ is the submatrix $A_i$ reconstructed using the row basis $V$. Thus by controlling the relative error of each submatrix, we can guarantee the error bound on the whole matrix in the end. Figure 1(b) shows an example of three cosine trees sharing a common basis.

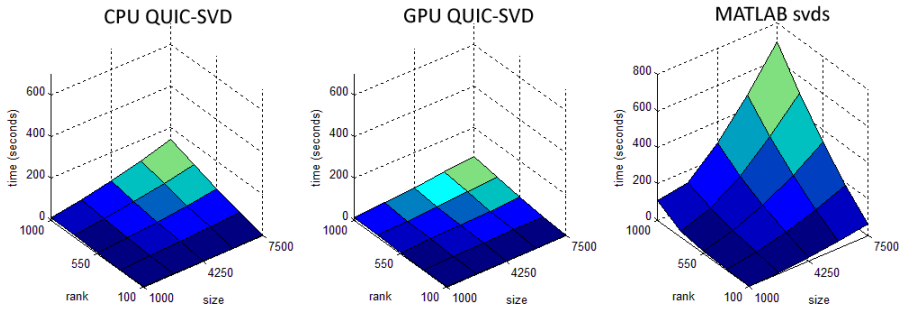
Note that by using partitioning, only a fraction of the matrix data is loaded to the GPU memory at a time, thus allowing for out-of-core computation.

**SVD Extraction** Given a matrix $A \in \mathbb{R}^{m \times n}$ and a basis $\widehat{V} \in \mathbb{R}^{n \times k}$, QUIC's SVD-extraction procedure first projects $A$ onto the basis, resulting in an $m \times k$ matrix $P = A\widehat{V}$. It then computes an exact SVD on the $k \times k$ matrix $P^{\mathrm{T}}P$, resulting in $U'\Sigma'V'^{\mathrm{T}} = P^{\mathrm{T}}P$. Finally, the approximate SVD of $A$ is extracted as $V = \widehat{V}V'$, $\Sigma = \sqrt{\Sigma'}$, and $U = PV'\Sigma^{-1}$.

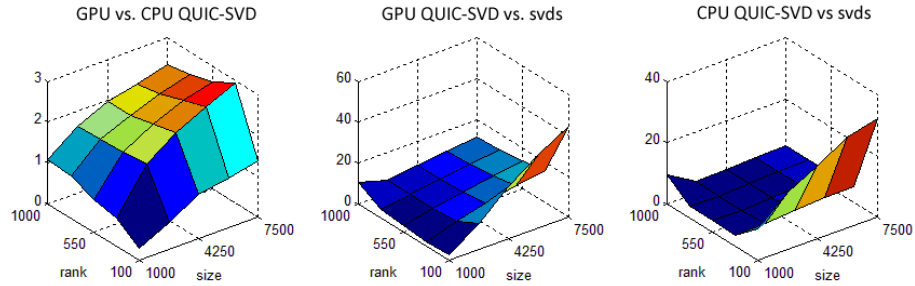
We assume that $P$ can fit in memory, since $k \ll n$. The matrix $A$ cannot fit in memory, so we once again load $A$ into memory a block at a time. Given a block $A_i$, the corresponding block of $P_i \subset P$ is $A_i\widehat{V}$. After we have completed a pass over all of $A$, the entire $P$ is in memory. We then proceed with the rest of the computation as described above.

## 3 Results

For testing and evaluation, we compared results of our GPU-based algorithm to the following three implementations: 1) a highly-optimized, multi-threaded CPU version of QUIC-SVD implemented using Intel Math Kernal Library; 2) MATLAB `svds` routine; and 3) the Tygert SVD [10], which is a fast CPU-based approximate SVD algorithm built upon random projection. In each test case, we plot the running time as well as the speedup over a range of matrix sizes and ranks. We use random matrices for testing. Given a rank $k$ and size $n$, we first generate an $n \times k$ matrix and a $k \times n$ matrix filled with uniform random numbers between $[-1, 1]$; we then multiply them to obtain an $n \times n$ matrix of rank $k$. Our experimental results were collected on a PC with an Intel Core i7 2.66 GHz CPU (with 8 hyperthreads), 6 GB of RAM, and an NVIDIA GTX 480 GPU. Both the CPU and GPU algorithms use double-precision arithmetic. For QUIC-SVD, we set the relative error threshold $\epsilon = 10^{-12}$, and $\delta = 10^{-12}$ (in Monte Carlo error estimation) for all experiments. All timings for the GPU




(a) Running time reported for each of the three algorithms listed.

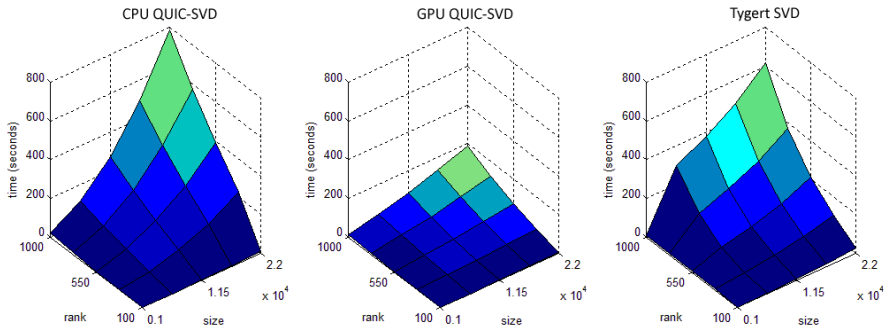



(b) Plots of speedup factors comparing each pair of algorithms.

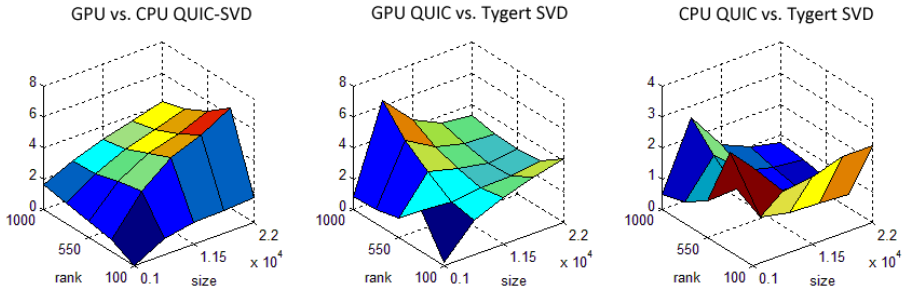
**Fig. 2.** *Performance and speedup comparison for the following three algorithms: CPU QUIC-SVD, GPU QUIC-SVD, and MATLAB* `svds`*. The input matrices are randomly generated with size ranging from* $1000^2$ *to* $7500^2$ *and rank ranging from 100 to 1000.*

implementation includes both the data transfer time (to and from the GPU) and actual computation time (on the GPU).

Figure 2(a) shows a performance comparison of our GPU implementation vs. the CPU implementation of QUIC-SVD as well as MATLAB `svds`. The matrix size ranges from $1,000^2$ to $7,500^2$ (the largest that `svds` could handle on our system), and the matrix rank ranges from 100 to 1000. All three algorithms were run with the same input and similar accuracy. Figure 2(b) plots the speedup factor for the same test cases. In addition, we show the speedup factor of the CPU version of QUIC-SVD over `svds`. We observe that the CPU QUIC-SVD is up to 30 times faster than `svds`, and our GPU implementation is up to 40 times faster. In both cases, the maximum speedup is achieved under a large and low-rank matrix. This makes sense because matrices with lower ranks favor the QUIC-SVD algorithm. From this plot we can see that the speedup primarily comes from the QUIC-SVD algorithm itself. If we compare the GPU and the CPU versions of QUIC-SVD alone, the maximum speedup of the GPU version is about 3 times (note that the two have their peak performances at different points). Although this is a moderate speedup, it will become more significant for larger matrices (shown below), as the GPU's parallelism will be better utilized.




(a) Running time reported for each of the three algorithms listed.




(b) Plots of speedup factors comparing each pair of algorithms.

**Fig. 3.** *Performance and speedup comparison for the following three algorithms: CPU QUIC-SVD, GPU QUIC-SVD, and Tygert SVD. The input matrices are randomly generated with size ranging from* $1000^2$ *to* $22000^2$ *and rank ranging from 100 to 1000.*

Figure 3(a) shows a performance comparison of our GPU and CPU implementations to Tygert SVD [10], which is a very fast approximate SVD algorithm that exploits random projection. Here we set the size of the test matrices to range from $1,000^2$ to $22,000^2$, and the rank to range from 100 to 1000. As $22,000^2$ matrix (double precision) is too large to fit in GPU memory, we used our partitioned version with 4 partitions. Again all three algorithms were run with the same input and comparable accuracy. Figure 3(b) plots the speedup factor for each pair of the test cases. Note that the CPU version and Tygert's algorithm have comparable performance, while the GPU version is up to 7 times faster than either algorithm. While the GPU version does not perform as well on small matrices due to its overhead, its benefits are evident for large-scale matrices.

## 4 Conclusions and Future Work

In conclusion, we have presented a GPU-based approximate SVD algorithm. Our method builds upon the QUIC-SVD algorithm introduced by [6], which exploits a tree-based structure to efficiently discover the intrinsic subspace of

the input matrix. Results show that our GPU algorithm achieves 6∼7 times speedup over an optimized CPU implementation. Using a matrix partitioning scheme, we have extended our algorithm to out-of-core computation, suitable for very large matrices.

In ongoing work, we are modifying our GPU algorithm to work with sparse matrices. This is important as large-scale matrices tend to be sparse. We will also test our algorithm in practical applications. One application we are particularly interested in is exacting the singular vectors of large graph Laplacians. This is instrumental for certain machine learning problems such as manifold alignment and transfer learning. Finally, we have found that the Monte Carlo error estimation is taking a considerable amount of overhead. We would like to investigate possible way to reduce or eliminate this overhead.

**Acknowledgments** We would like to thank Alexander Gray for providing details of the QUIC-SVD algorithm. This work is supported by NSF grant FODAVA-1025120.

## References

1. Bondhugula, V., Govindaraju, N., Manocha, D.: Fast SVD on graphics processors. Tech. rep., UNC Chapel Hill (2006)
2. Deshpande, A., Vempala, S.: Adaptive sampling and fast low-rank matrix approximation. In: RANDOM. pp. 292–303 (2006)
3. Friedland, S., Niknejad, A., Kaveh, M., Zare, H.: Fast Monte-Carlo low rank approximations for matrices. In: Proc. of IEEE/SMC International Conference on System of Systems Engineering (2006)
4. Frieze, A., Kannan, R., Vempala, S.: Fast monte-carlo algorithms for finding low-rank approximations. J. ACM 51, 1025–1041 (2004)
5. Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D.: Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In: Proc. of the 2005 ACM/IEEE conference on Supercomputing. pp. 3– (2005)
6. Holmes, M., Gray, A., Isbell, C.L.: QUIC-SVD: Fast SVD using Cosine trees. In: Proc. of NIPS. pp. 673–680 (2008)
7. Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J.: CULA: hybrid gpu accelerated linear algebra routines. In: Proc. SPIE. p. 7705 (2010)
8. Kerr, A., Campbell, D., Richards, M.: Qr decomposition on gpus. In: Proc. of the 2nd Workshop on GPGPU. pp. 71–78 (2009)
9. Lahabar, S., Narayanan, P.J.: Singular value decomposition on GPU using CUDA. In: Proc. of IEEE International Symposium on Parallel&Distributed Processing. pp. 1–10 (2009)
10. Rokhlin, V., Szlam, A., Tygert, M.: A randomized algorithm for principal component analysis. SIAM J. Matrix Anal. Appl. 31, 1100–1124 (August 2009)
11. Sarlos, T.: Improved approximation algorithms for large matrices via random projections. In: Proc. of the 47th Annual IEEE Symposium on Foundations of Computer Science. pp. 143–152 (2006)
12. Volkov, V., Demmel, J.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. Tech. rep., UC Berkeley (2008)