Singular Value Decomposition on GPU using CUDA

Sheetal Lahabar
Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India
`sheetal@students.iiit.ac.in`

P. J. Narayanan
Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India
`pjn@iiit.ac.in`

Abstract

Linear algebra algorithms are fundamental to many computing applications. Modern GPUs are suited for many general purpose processing tasks and have emerged as inexpensive high performance co-processors due to their tremendous computing power. In this paper, we present the implementation of singular value decomposition (SVD) of a dense matrix on GPU using the CUDA programming model. SVD is implemented using the twin steps of bidiagonalization followed by diagonalization. It has not been implemented on the GPU before. Bidiagonalization is implemented using a series of Householder transformations which map well to BLAS operations. Diagonalization is performed by applying the implicitly shifted QR algorithm. Our complete SVD implementation outperforms the MATLAB and Intel ®Math Kernel Library (MKL) LAPACK implementation significantly on the CPU. We show a speedup of upto 60 over the MATLAB implementation and upto 8 over the Intel MKL implementation on a Intel Dual Core $2.66GHz$ PC on NVIDIA GTX 280 for large matrices. We also give results for very large matrices on NVIDIA Tesla S1070.

1. Introduction

The singular value decomposition (SVD) is an important technique used for factorization of a rectangular real or complex matrix. Matrix computations using the SVD are more robust to numerical errors. It is used for computing the pseudoinverse of a matrix, solving homogeneous linear equations, solving the total least square minimization problem and finding approximation matrix. It is also widely used in applications related to principal component analysis, signal processing, pattern recognition and image processing for singular value spectral analysis. The SVD also has a variety of applications in scientific computing, signal processing, automatic control and many other areas.

A SVD of a $m \times n$ matrix A is any factorization of the form
$$A = U\Sigma V^T, \tag{1}$$
where U is a $m \times m$ orthogonal matrix, V is a $n \times n$ orthogonal matrix and Σ is a $m \times n$ diagonal matrix with elements $s_{ij} = 0$ if $i \neq j$ and $s_{ii} \geq 0$ in descending order along the diagonal.

The rapid increase in the performance of graphics hardware has made GPU a strong candidate for performing many compute intensive tasks, especially many data-parallel tasks. GPUs now include fully programmable processing units that follow a stream programming model and support vectorized floating-point operations. High level languages have emerged to support the new programmability. NVIDIA's 8-series GPU with CUDA computing environment provides the standard C like language interface for the programmable processors, which eliminates the overhead of learning an inadequate API [21]. The Close-To-Metal (CTM) from ATI/AMD [1] is another interface for programming GPUs which treat them as massively parallel co-processors. GPUs provide tremendous memory bandwidth and computational horsepower. For example, the NVIDIA GeForce 8800 GTX can achieve a sustained memory bandwidth of 86.4 GB/s and a theoretical maximum of 346 GFLOPS. It has 768 MB of storage space. NVIDIA GTX 280 can achieve a theoretical maximum of 933 GFLOPS. The GPU performance has been growing at a faster rate than Moore's law. NVIDIA Tesla S1070 Computing System has four Tesla T10 computing processors with 4 GB memory per processor. A Tesla T10 processor can a achieve a theoretical peak performance of a teraflop.

Recently, GPUs have been extensively used for scientific computations. However, little work has been done to solve problems like SVD which has numerous applications. In this paper, we present an implementation of SVD for dense matrices on the GPU using the CUDA model. Our implementation uses NVIDIA's CUBLAS library and CUDA kernels. We achieve a speedup of 3 to 60 over the MATLAB implementation and 3 to 8 over the Intel MKL implementation of SVD on a Intel Dual Core $2.66GHz$ PC. We are able to compute the SVD of very large matrices which is otherwise impossible on the CPU due to memory limitations. We also demonstrate the ease of programmability with the availability of CUDA libraries for complex mathematical applications.

Section 2 describes the related work done for solving numerical problems. Section 3 describes the SVD algorithm

in detail and its GPU implementation. Section 4 presents the experimental results and its comparison with other available sources. Future work and conclusion are given in Section 5.

2. Related Work

Several algorithms have been developed on the GPUs for mathematical computations like sorting [16], geometric computations, matrix multiplications, FFT [20] and graph algorithms [17], [23]. Kruger *et al.* [18] introduced a framework for the implementation of linear algebra operators on vectors and matrices that exploits the parallelism on GPUs. Galoppo *et al.* [14] reduced the matrix decomposition and row operations to a series of rasterization problems on the GPU. Christen *et al.* [9] proposed algorithms to accelerate sparse direct factorization and non-linear interior point optimization on the GPU using CUDA. Barrachina *et al.* [4] proposed two high-level application programming interfaces that use the GPU as a co-processor for dense linear algebra operations. There have been many efforts towards optimizing and tuning of the Level 3 CUBLAS Graphics Processors. Barrachina *et al.* [5] proposed several alternative implementations that are competitive with those in CUBLAS. Fujimoto [13] proposed a new algorithm for matrix-vector multiplication on NVIDIA CUDA architecture. Barrachina *et al.* [3] presented several algorithms to compute the solution of a linear system of equations. Fatica *et al.* [12] proposed how MATLAB could be extended to take advantage of the computational power offered by the latest GPUs. NVIDIA's CUDA library [21] comes with an implementation of simple Basic Linear Algebra Subprograms (BLAS) on GPU, called the CUBLAS.

There have been many efforts towards parallelizing the SVD algorithm on architectures like the FPGA, Cell Processors, GPU, etc., which have scalable parallel architecture. Ma *et al.* [19] proposed the implementation of two-sided rotation Jacobi SVD algorithm on a two million gate FPGA. They proposed a mesh connected array structure of simple 2×2 processors to compute SVD of a large matrix. Bobda *et al.* [6] proposed an efficient implementation of the SVD for large matrices and the possibility of integrating FPGA's as a part of a Distributed Reconfigurable System (DRS). Baker [2] described a parallel algorithm to compute the SVD of block circulant matrices on Cray-2. Dickson *et al.* [11] designed a programmable processor for computing the Givens rotation using approximate rotation method. The processor can also be programmed for SVD computation. Yamamoto *et al.* [25] proposed a method to speed up the SVD of very large rectangular matrices using the CSX600 floating point co-processor. They achieve up to 3.5 times speedup over the Intel MKL on 3.2GHz Xeon processor for a 100000×4000 matrix but was not efficient on smaller matrices. Zhang Shu *et al.* [22] presented the implementation of One Sided Jacobi method for SVD on GPU using

CUDA. The performance of their algorithm is limited by the availability of shared memory and works well only for small size matrices. Bondhugula *et al.* [7] proposed a hybrid GPU based implementation of singular value decomposition using fragment shaders and frame buffer objects in which the diagonalization would be performed on the CPU.

There are several numerical libraries such as ATLAS and the Intel Math Kernel Library which are widely used for different applications on the CPU. They are designed to achieve high accuracy as well as high memory bandwidth and computational throughput on the CPUs, e.g. Intel MKL is optimized for Intel processors. Intel MKL LAPACK gives better performance than standard LAPACK on Intel processors.

3. SVD Algorithm

The SVD of a matrix A can be computed using the Golub-Reinsch (Bidiagonalization and Diagonalization) algorithm or the Hestenes method. We use the Golub-Reinsch method as it is simple and compact, and maps well to the SIMD GPU architecture. It is also popular in many numerical libraries. Hestenes algorithm is a Jacobi type approach that gives low performance and hence not popular. Golub-Reinsch algorithm is used in the LAPACK package which is a two step algorithm [24]. The matrix is first reduced to a bidiagonal matrix using a series of householder transformations. The bidiagonal matrix is then diagonalized by performing implicitly shifted QR iterations [10]. SVD is an $O(mn^2)$ algorithm for $m \geq n$. Algorithm 1 describes the SVD algorithm for a input matrix A.

Algorithm 1 Singular Value Decomposition

1: $B \leftarrow Q^T A P$ {Bidiagonalization of A to B}
2: $\Sigma \leftarrow X^T B Y$ {Diagonalization of B to Σ}
3: $U \leftarrow QX$
4: $V^T \leftarrow (PY)^T$ {Compute orthogonal matrices U and V^T and SVD of $A = U\Sigma V^T$}

3.1. Bidiagonalization

3.1.1. Algorithm. In this step, the given matrix A is decomposed as

$$A = QBP^T \tag{2}$$

by applying a series of householder transformations where B is a bidiagonal matrix and Q and P are unitary householder matrices. For a matrix of size $m \times n$ with $m \geq n$, we first select a householder vector $\mathbf{u}^{(1)}$ of length m for vector

$A(1:m,1)$ and $\mathbf{v}^{(1)}$ of length n for $A(1,2:n)$ such that

$$
\begin{aligned}
\hat{A}_1 &= (I - \sigma_{1,1}\mathbf{u}^{(1)}\mathbf{u}^{(1)^T})A(I - \sigma_{2,1}\mathbf{v}^{(1)}\mathbf{v}^{(1)^T}) \quad (3) \\
&= H_1 A G_1 = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \dots & 0 \\ 0 & x & x & \dots & x \\ \vdots & \vdots & & & \vdots \\ 0 & x & \dots & & x \end{bmatrix}.
\end{aligned}
$$

$\hat{A}_1$ has zeros below the diagonal and to the right of the superdiagonal of the first row and $A(1,1)$ is updated to α_1 and $A(1,2)$ is updated to β_1. This is the first column-row elimination.

We denote the left householder $m \times m$ matrices as H_i's and right householder $n \times n$ matrices as G_i's and the corresponding σ's as $\sigma_{1,i}$'s and $\sigma_{2,i}$'s respectively. The elimination procedure is then repeated for second column $A(2:m,2)$ and row $A(2,3:n)$ and so on. If $m > n$, n columns and $n-2$ rows must be eliminated. After all the columns and rows are eliminated we obtain a final bidiagonal matrix B such that

$$
B = Q^T A P, \quad (4)
$$

where

$$
Q^T = \prod_{i=1}^{n} H_i, P = \prod_{i=1}^{n-2} G_i. \quad (5)
$$

Here, $H_i = I - \sigma_{1,i}\mathbf{u}^{(i)}\mathbf{u}^{(i)^T}$ and $G_i = I - \sigma_{2,i}\mathbf{v}^{(i)}\mathbf{v}^{(i)^T}$. The $\mathbf{u}^{(i)}$'s are vectors of length m with $i-1$ leading zeros and $\mathbf{v}^{(i)}$'s are vectors of length n with i leading zeros. These are formed as $\mathbf{u}^{(i)} = [0\dots0,\hat{\mathbf{u}}^{(i)}]^T$ and $\mathbf{v}^{(i)} = [0\dots0,\hat{\mathbf{v}}^{(i)}]^T$, where $\hat{\mathbf{u}}^{(i)}$ is a vector of $m-i+1$ trailing components of $\mathbf{u}^{(i)}$ and $\hat{\mathbf{v}}^{(i)}$ is a vector of $n-i$ trailing components of $\mathbf{v}^{(i)}$. In general, for a vector $\mathbf{y} = [y_1, \dots, y_l]$ of length l the selection of householder vector $\mathbf{r}$ and scalars σ and α as given below

$$
\begin{aligned}
\alpha &= -\text{sign}(y_1)\,||\mathbf{y}||\,, \quad a = \text{sign}(y_1)\,||\mathbf{y}|| , \quad (6) \\
\sigma &= (y_1 + a)/a \quad (7) \\
\text{and} \quad \mathbf{r} &= \frac{\mathbf{y} + [a, 0, \dots, 0]^T}{y_1 + a}, \quad (8)
\end{aligned}
$$

such that $(I - \sigma\mathbf{r}\mathbf{r}^T)\mathbf{y} = [\alpha, 0, \dots, 0]^T$. $\hat{\mathbf{u}}^{(i)}$ of length $m - (i-1)$ and α_i for $A(i:m,i)$ and $\hat{\mathbf{v}}^{(i)}$ of length $n-i$ and β_i for $A(i,i+1:n)$ are computed similar to $\mathbf{r}$ and α in Equation 6 to 8.

The householder bidiagonalization can be achieved by alternating matrix vector multiplies with rank-one updates introduced by Golub and Kahan [15]. The multiplication of A by H_i updates $A(i:m,i+1:n)$ and $A(i,i)$ and multiplication by G_i updates $A(i+1:m,i+1:n)$ and $A(i,i+1)$. We can summarize the update of the trailing matrix A after i^{th} column-row elimination as two rank updates as

$$
\begin{aligned}
A(i+1:m,i+1:n) &= A(i+1:m,i+1:n) \\
&\quad -\hat{\mathbf{u}}^{(i)}\hat{\mathbf{z}}^{(i)^T} - \hat{\mathbf{w}}^{(i)}\hat{\mathbf{v}}^{(i)^T},
\end{aligned}
$$

where

$$
\begin{aligned}
\hat{\mathbf{z}}^{(i)^T} &= \mathbf{x}^T - \sigma_{2,i}(\mathbf{x}^T\hat{\mathbf{v}}^{(i)})\hat{\mathbf{v}}^{(i)^T}, \\
\hat{\mathbf{w}}^{(i)} &= \sigma_{2,i}A(i:m,i+1:n)\hat{\mathbf{v}}^{(i)} \\
\text{and} \quad \mathbf{x} &= \sigma_{1,i}A^T(i:m,i+1:n)\hat{\mathbf{u}}^{(i)}.
\end{aligned}
$$

The householder matrices Q and P given in Equation 5 are computed similarly as it also involves multiplication by H_i's and G_i's respectively, but in reverse order. We use the term *partial bidiagonalization* to refer to the computation of the matrix B in Equation 4, without keeping track of the P and Q matrices. This is computationally less expensive than complete bidiagonalization and was the operation implemented on the GPU by Bondhugula *et al* [7]. The update rule after i^{th} column-row elimination is

$$
\begin{aligned}
Q(1:m,i:m) &= Q(1:m,i:m) - \hat{\mathbf{k}}^{(i)}\hat{\mathbf{u}}^{(i)^T} \quad \text{and} \\
P^T(i:n,1:n) &= P^T(i:n,1:n) - \hat{\mathbf{v}}^{(i)}\hat{\mathbf{l}}^{(i)^T}, \quad \text{where} \\
\hat{\mathbf{k}}^{(i)} &= \sigma_{1,i}Q(1:m,i:m)\hat{\mathbf{u}}^{(i)} \quad \text{and} \\
\hat{\mathbf{l}}^{(i)} &= \sigma_{2,i}P^T(i:n,1:n)^T\hat{\mathbf{v}}^{(i)}.
\end{aligned}
$$

The updates can be expressed using BLAS level 2 operations. After every column-row elimination, the trailing matrix is updated. This method is computationally expensive and involves many reads and writes to the memory. We can increase the computation to read ratio by deferring the update of the trailing matrix by bidiagonalizing a block of columns and rows together and then updating the trailing matrix as proposed in [8]. The LAPACK implementation also uses the blocking approach. It requires the computation of new rows and columns belonging to the block just before elimination due to the previous eliminations in the block.

The matrix A is divided into blocks of size L as shown in Figure 1 and the update occurs only after L columns and rows are bidiagonalized. Extra computations are performed for the updated columns and rows of the same block, which require $\hat{\mathbf{u}}^{(i)}$'s, $\hat{\mathbf{v}}^{(i)}$'s, $\hat{\mathbf{w}}^{(i)}$'s and $\hat{\mathbf{z}}^{(i)}$'s due to previous eliminations in the block. These vectors are required for updating the trailing matrix once L columns and rows are eliminated. As the set of update vectors is incremented with every elimination, more computations are required to update the columns and rows before elimination. The householder matrices Q and P^T are also block updated for which $\hat{\mathbf{k}}^{(i)}$'s and $\hat{\mathbf{l}}^{(i)}$'s are stored. The value of L is chosen depending on the performance of the BLAS routines. The block algorithm has a total floating point operation count of $O(mn^2)$ for $m \geq n$.
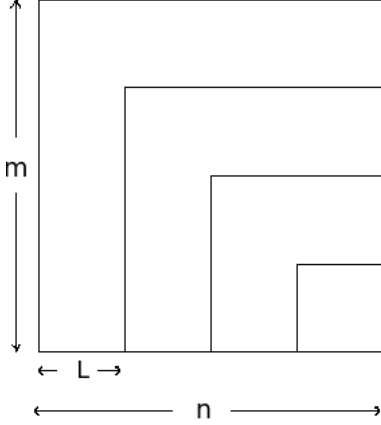
Figure 1. Subdivision of a matrix into blocks of size L

This method requires an additional storage of a $m \times L$ array U_{mat} to store $\hat{\mathbf{u}}^{(i)}$'s, a $L \times n$ array V_{mat} to store $\hat{\mathbf{v}}^{(i)}$'s, a $m \times L$ array W_{mat} to store $\hat{\mathbf{w}}^{(i)}$'s, a $L \times n$ array Z_{mat} to store $\hat{\mathbf{z}}^{(i)}$'s, a $m \times L$ array Q_{mat} to store $\hat{\mathbf{k}}^{(i)}$'s and a $L \times n$ array P_{mat} to store $\hat{\mathbf{l}}^{(i)}$'s. Since these are required only for updating the matrix, they can be reused after every block update. The trailing matrix is accessed only during the matrix update.

Algorithm 2 Bidiagonalization algorithm

Require: $m \geq n$
1: $kMax \leftarrow \frac{n}{L}$ {L is the block size}
2: **for** $i = 1$ to $kMax$ **do**
3:      $t \leftarrow L(i-1) + 1$
4:      Compute $\hat{\mathbf{u}}^{(t)}$, $\alpha_{1,t}, \sigma_{1,t}, \hat{\mathbf{k}}^{(t)}$
5:      Eliminate $A(t:m,t)$ and update $Q(1:m,t)$
6:      Compute new $A(t, t+1:n)$
7:      Compute $\hat{\mathbf{v}}^{(t)}$, $\alpha_{2,t}, \sigma_{2,t}, \hat{\mathbf{l}}^{(t)}$
8:      Eliminate $A(t, t+1:n)$ and update $P^T(t, 1:n)$
9:      Compute $\hat{\mathbf{w}}^{(t)}$, $\hat{\mathbf{z}}^{(t)}$ and store the vectors
10:      **for** $k = 2$ to L **do**
11:         $t \leftarrow L(i-1) + k$
12:         Compute new $A(t:m,t)$ using $k-1$ update vectors
13:         Compute $\hat{\mathbf{u}}^{(t)}$, $\alpha_{1,t}, \sigma_{1,t}, \hat{\mathbf{k}}^{(t)}$
14:         Eliminate $A(t:m,t)$ and update $Q(1:m,t)$
15:         Compute new $A(t, t+1:n)$
16:         Compute $\hat{\mathbf{v}}^{(t)}$, $\alpha_{2,t}, \sigma_{2,t}, \hat{\mathbf{l}}^{(t)}$
17:         Eliminate $A(t, t+1:n)$ and update $P^T(t, 1:n)$
18:         Compute $\hat{\mathbf{w}}^{(t)}$, $\hat{\mathbf{z}}^{(t)}$ and store the vectors
19:      **end for**
20:      Update $A(iL+1:m, iL+1:n)$, $Q(1:m, iL+1:m)$ and $P^T(iL+1:n, 1:n)$
21: **end for**

3.1.2. Bidiagonalization on the GPU. Algorithm 2 describes the bidiagonalization procedure. Each step can be performed using CUDA BLAS functions. CUBLAS [21] provides high performance matrix-vector, matrix-matrix multiplication and norm computation function. The blocking approach for bidiagonalization can be performed efficiently since CUBLAS gives high performance for matrix-vector, matrix-matrix multiplication even if one of the dimensions is small. Experiments prove that CUBLAS deliver much higher performance when operating on matrices with dimensions that are a multiple of 32 due to memory alignment issues [5]. Hence, we pad the vectors and matrices with zeros, transforming their dimensions to the next multiple of 32.

The performance of the GPU libraries depend on data placement and how the library is used. The movement of data is of consideration when using BLAS in general. We assume that initially the input matrix A is on the CPU and is transferred to the GPU. NVIDIA 8800 GTX has a sustained internal memory bandwidth of 86.4 GB/s but the bandwidth between the CPU and the GPU is an order of magnitude lower. Hence CPU to GPU transfers should be minimized. The matrices Q, P^T, U_{mat}, V_{mat}, W_{mat}, Z_{mat}, Q_{mat} and P_{mat} are initialized on the device. All the operations required for bidiagonalization are done on the data local to the GPU using CUBLAS library routines. Since the thread-processors of the GPU operate on GPU data, there is no expensive data transfer between the GPU and the CPU. The bidiagonalization is performed inplace, i.e., A becomes the bidiagonal matrix. After the bidiagonalization of matrix A on the GPU, only the diagonal and superdiagonal elements of the bidiagonal matrix are copied to the CPU to proceed with the diagonalization as described in the next section while matrices Q and P^T remain on the device. Our use of the latest CUBLAS 2.0 library keeps the GPU implementation very efficient on the given hardware. The total storage requirement for the algorithm is $(3(mL + Ln) + m^2 + n^2 + mn + 2\max(m,n)) \times 4$ bytes on the GPU.

3.2. Diagonalization of a bidiagonal matrix

3.2.1. Algorithm. The bidiagonal matrix can be reduced to a diagonal matrix by iteratively applying the implicitly shifted QR algorithm [10]. The matrix B obtained in the first step is decomposed as

$$\Sigma = X^T B Y, \qquad (9)$$

where Σ is a diagonal matrix, X and Y are orthogonal unitary matrices.

Algorithm 3 describes the diagonalization procedure. The $d(i)'s$ are the diagonal elements and $e(i)'s$ are the superdiagonal elements of the matrix B. Every iteration updates the diagonal and the superdiagonal elements such that the value of the superdiagonal elements becomes less than their

previous value. On convergence of the algorithm, $d(i)'s$ contains the singular values and X and Y^T contains the singular vectors of B.

Algorithm 3 Diagonalization algorithm

1: *iter* $\leftarrow 0$
2: *maxitr* $\leftarrow 12 * N * N$ {N is the number of main diagonal elements}
3: $k_2 \leftarrow N$ {k_2 points to the last element of unconverged part of matrix}
4: **for** $i = 1$ to *maxitr* **do**
5:    **if** $k_2 <= 1$ **then**
6:       break the loop
7:    **end if**
8:    **if** *iter* $>$ *maxitr* **then**
9:       return *false*
10:    **end if**
11:    *matrixsplitflag* $\leftarrow$ *false*
12:    **for** $l = 1$ to $k_2 - 1$ **do**
13:       $k_1 \leftarrow k_2 - l$ {Find diagonal block matrix to work on}
14:       **if** $abs(e(k_1)) <= thres$ **then**
15:          *matrixsplitflag* $\leftarrow$ *true*, break the loop
16:       **end if**
17:    **end for**
18:    **if** !*matrixsplitflag* **then**
19:       $k_1 \leftarrow 1$
20:    **else**
21:       $e(k_1) \leftarrow 0$
22:       **if** $k_1 == k_2 - 1$ **then**
23:          $k_2 \leftarrow k_2 - 1$, continue with next iteration
24:       **end if**
25:    **end if**
26:    $k_1 = k_1 + 1$
27:    **if** $k_1 == k_2 - 1$ **then**
28:       Compute SVD of 2×2 block and coefficient vectors $\mathbf{C}_1$, $\mathbf{S}_1$ and $\mathbf{C}_2$, $\mathbf{S}_2$ of length 1
29:       Apply forward row transformation on the rows $k_2 - 1$ and k_2 of Y^T using $\mathbf{C}_1$, $\mathbf{S}_1$
30:       Apply forward column transformation on the columns $k_2 - 1$ and k_2 of X using $\mathbf{C}_2$, $\mathbf{S}_2$
31:       $k_2 \leftarrow k_2 - 2$, continue with next iteration
32:    **end if**
33:    Select shift direction: *forward* if $d(k_1) < d(k_2)$, else *backward*
34:    Apply convergence test on the sub block, continue next iteration if any value converges
35:    Compute the $shift$ from 2×2 block at the end of the sub matrix
36:    *iter* $\leftarrow$ *iter*$+k_2 - k_1$
37:    Apply simplified/shifted forward/backward Givens rotation on the rows k_1 to k_2 of B and compute $\mathbf{C}_1$, $\mathbf{S}_1$ and $\mathbf{C}_2$, $\mathbf{S}_2$ of length $k_2 - k_1$
38:    Apply forward/backward transformation on the rows k_1 to k_2 of Y^T using $\mathbf{C}_1$, $\mathbf{S}_1$
39:    Apply forward/backward transformation on the columns k_1 to k_2 of X using $\mathbf{C}_2$, $\mathbf{S}_2$
40: **end for**
41: Sort the singular values and corresponding singular vectors in decreasing order

---

The algorithm finds indexes k_1 and k_2 with $k_1 < k_2$ in each iteration such that $e(k_1)$ is below a threshold which depends on the machine precision. If k_1 and k_2 differ by 1 or 2, one or two singular values can be extracted directly and k_2 moves up. Otherwise, a series of Givens rotations modify $d(i)$ and $e(i)$ in the range k_1 to k_2 such that $e(i)$'s become smaller than before. Each rotation is captured in the coefficient vectors $(\mathbf{C}_1, \mathbf{S}_1)$ and $(\mathbf{C}_2, \mathbf{S}_2)$. Corresponding inverse rotations are applied on Y^T and X using the coefficient vectors $(\mathbf{C}_1, \mathbf{S}_1)$ and $(\mathbf{C}_2, \mathbf{S}_2)$ respectively. Algorithm 4 and 5 describes the rotations applied on the rows of Y^T in the forward and backward direction respectively. Similar rotations are applied on the columns of X using $\mathbf{C}_2$ and $\mathbf{S}_2$. See [10] for more details on the steps. The computation converges when all the singular values are found.

Algorithm 4 Forward transformation on the rows of Y^T

Require: $k_1 < k_2$
1: **for** $j=k_1$ to $k_2 - 1$ **do**
2:    $\mathbf{t} \leftarrow Y^T(j+1, 1:n)\mathbf{C}_1(j - k_1 + 1)$
3:    $\mathbf{t} \leftarrow \mathbf{t} - Y^T(j, 1:n)\mathbf{S}_1(j - k_1 + 1)$
4:    $Y^T(j, 1:n) \leftarrow Y^T(j, 1:n)\mathbf{C}_1(j - k_1 + 1) + Y^T(j+1, 1:n)\mathbf{S}_1(j - k_1 + 1)$
5:    $Y^T(j+1, 1:n) \leftarrow \mathbf{t}$
6: **end for**

Algorithm 5 Backward transformation on the rows of Y^T

Require: $k_1 < k_2$
1: **for** $j=k_2 - 1$ to k_1 **do**
2:    $\mathbf{t} \leftarrow Y^T(j+1, 1:n)\mathbf{C}_1(j - k_1 + 1)$
3:    $\mathbf{t} \leftarrow \mathbf{t} - Y^T(j, 1:n)\mathbf{S}_1(j - k_1 + 1)$
4:    $Y^T(j, 1:n) \leftarrow Y^T(j, 1:n)\mathbf{C}_1(j - k_1 + 1) + Y^T(j+1, 1:n)\mathbf{S}_1(j - k_1 + 1)$
5:    $Y^T(j+1, 1:n) \leftarrow \mathbf{t}$
6: **end for**

3.2.2. Diagonalization on the GPU. In this section, we present the parallel version of the diagonalization algorithm and its implementation on the GPU. The diagonal and superdiagonal elements of B are copied to the CPU. Applying Givens rotations on B and computing the coefficient vectors is done sequentially on the CPU as it only requires access to the diagonal and superdiagonal elements. In Algorithm 4, the computations for every row of Y^T depends only on the next row, i.e., every element of a row depends only on the element below it and its corresponding coefficient vector element. In Algorithm 5, the operations on a row depends only on the row above it. Similarly for the columns of X. The computations for each row depends on the results from the previous row, making it difficult to parallelize across rows. However, the results for all the elements of the row can be computed in parallel. We use the thread processors of the GPU to process elements of each row in parallel. This

gives high performance on large matrices but also works well for medium sized matrices. The transformation of matrices Y^T and X is done in parallel on the GPU. In Algorithm 3, steps 29-30, 38-39 and 41 are executed on the GPU. A simple swap kernel is called for sorting the vectors. The matrices Y^T and X are initialized to identity on the device.

Our algorithm divides a row of the matrix into blocks as shown in Figure 2. Each thread operates on one element of the row. Since the transformations are applied on $k_2 - k_1 + 1$ rows, the kernel runs a loop of $k_2 - k_1$ similar to Algorithm 4 with each modifying two rows. This division of the row into blocks and looping can be done efficiently on CUDA architecture, since each thread performs independent computations. The data required for the block is stored in shared memory and the operations can be performed efficiently on a multiprocessor.
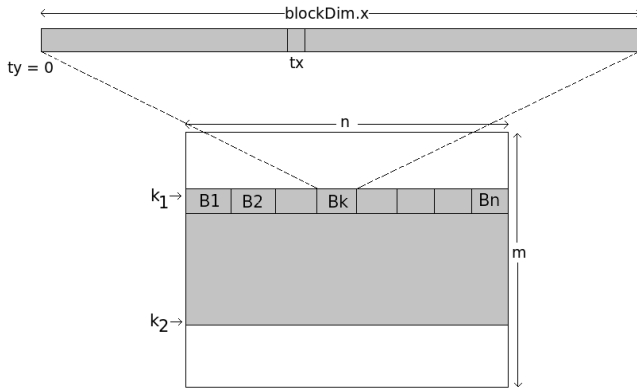

Figure 2. Division of a matrix row into CUDA thread blocks

The coefficient vectors $\mathbf{C}_1$ and $\mathbf{S}_1$ are copied from the CPU to the device memory. At any instant during the kernel execution, an element of the coefficient vector is required by all elements of two rows. Hence, we use the shared memory to store the coefficient vectors.

We allocate 64 to 256 threads for a block depending on the size of the matrix. This ensures that there are enough blocks and all the multiprocessors are allotted atleast 2 blocks. When the thread block contains T threads, it must use the shared memory of at most 2KB to keep 8 blocks active on a multiprocessor which will give optimal performance. At any instant, a block will require $2 \times (T \times 4)$ bytes of shared memory for the T elements of the two rows of the matrix it is working on as we use floating point arithmetic. Every iteration of the loop in the forward kernel modifies two rows of the matrix. Since the second row is again modified in the next iteration only the first updated row is copied back to the device. The second updated row remains in the shared memory for the next iteration. The shared memory is reused for copying the third row for the next iteration and the iteration proceeds.

The amount of shared memory that could be used to store coefficient vectors is $2\text{KB} - (2 \times T \times 4)$ bytes. However, the memory required for the coefficient vectors is $2 \times (k_2 - k_1) \times 4$ bytes which can exceed $2\text{KB} - (2 \times T \times 4)$ bytes for large matrices. In order to only use the available shared memory for the coefficient vectors we copy a fixed number of coefficient vector elements of $\mathbf{C}_1$ and $\mathbf{S}_1$ into $2\text{KB} - (2 \times T \times 4)$ bytes, process the same number of rows and then reuse the shared memory for copying the next set of vector elements. The backward row transformation kernel is similar to the forward row transformation kernel.

Since the column transformations are similar to row transformations, we use the row transformation kernel on the rows of X^T instead of the columns of X. This requires copying $\mathbf{C}_2$ and $\mathbf{S}_2$ to the GPU. As the elements are accessed sequentially there are no non-coalesced memory accesses. The access to the shared memory has no bank conflicts. All the threads in a block are used for copying the vectors from the global memory to the shared memory which requires that the vectors are padded to the nearest multiple of block size.

On convergence, $d(i)$'s contain the singular values. Y^T and X^T are on the device which are further used for computing the orthogonal matrices U and V. Our algorithm is efficient as it performs exactly the same number of operations on the GPU as the corresponding sequential algorithm. The total storage requirement for the algorithm is $(6 \min(m, n)) \times 4$ bytes on the CPU and $(m^2 + n^2) \times 4$ bytes on the GPU.

3.3. Complete SVD

We perform two matrix-matrix multiplications at the end to compute orthogonal matrices $U = QX$ and $V^T = (PY)^T$ as given in Equation 1. We use CUBLAS matrix multiplication routines. The matrices Q, P^T, X^T, Y^T, U and V^T are on the device. The orthogonal matrices U and V^T can then be copied to the CPU. $d(i)$'s contains the singular values, i.e., diagonal elements of Σ and is on the CPU.

4. Results

In this section, we analyze the performance of our algorithm with the optimized CPU implementation of SVD on MATLAB and Intel MKL 10.0.4 LAPACK. We enable dynamic threading in Intel MKL for good performance. We tested our algorithm on a Intel Dual Core 2.66GHz PC and a NVIDIA GeForce 8800 GTX graphics processor, a NVIDIA GTX 280 processor and NVIDIA Tesla S1070. NVIDIA Tesla S1070 has 4 Tesla T10 GPU processors and a total memory of 16 GB and can achieve a theoretical maximum of 4 TFLOPS performance. We used a GPU of Tesla S1070 with 240 cores and 4 GB memory with a compute power of 1 TFLOPS. The 8800 GTX has 128 stream processors

divided into 16 multiprocessors with 8 texture access units and a total of 768 MB of memory. The GTX 280 has 240 stream processors divided into 30 multiprocessors with 10 texture access units and a total of 1 GB memory. According to NVIDIA, GTX 280 can achieve a peak performance of 933 GFLOPS and 8800 GTX of 345.6 GFLOPS. However, GTX 280 gives 375 GFLOPS and 8800 GTX gives 120 GFLOPS performance for single precision matrix multiply using CUBLAS. We used Intel Core 2 Duo CPU E6750 @ 2.66GHz processor for our experiments which is said to be rated 22.4 GFLOPS.

We generated 10 random dense matrices of single precision numbers for each size. The SVD algorithm was executed for each matrix 10 times. To avoid a particularly good or bad sample, we averaged over the random matrices for each size. The average did not vary much if 10 or more matrices were used. Table 1 gives the overall average time in seconds on GPU, MATLAB and Intel MKL. SVD computation includes the time taken for bidiagonalization, diagonalization and computing the orthogonal matrices. We achieve a speedup of 3.04 to 8.2 over the Intel MKL implementation and 3.32 to 59.3 over the MATLAB implementation for various square and rectangular matrices on GTX 280. The CPU still out-performs the GPU implementation for small matrices. For large square matrices, the speedup increases with the size of the matrix. Figure 3 shows the time for computing the SVD of square matrices and Figure 4 shows the time for computing the SVD of rectangular matrices with leading dimension m=8K.
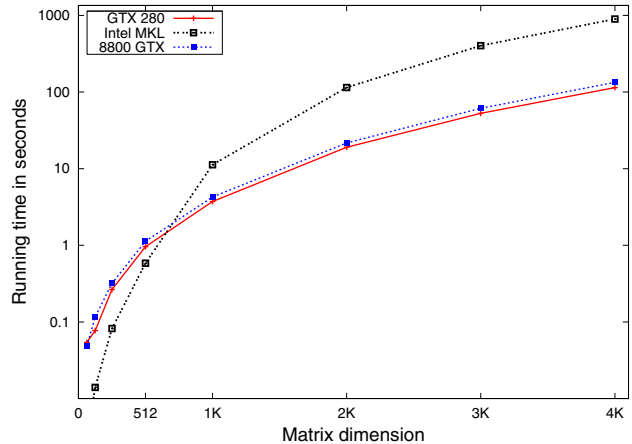

Figure 3. SVD computation time (in seconds) for square matrices on Intel MKL and GPU (GTX 280 and 8800 GTX)
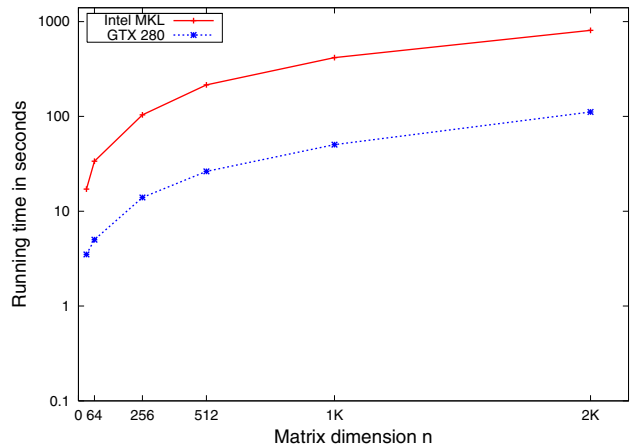

Figure 4. SVD computation time (in seconds) for rectangular matrices ($m \times n$) with leading dimension m=8K and varying n on Intel MKL and GPU (GTX 280)

| | SVD | SVD | SVD | SVD | Speedup |
SIZE	MATLAB	MKL	GTX 280	8800	MKL/280
64×64	0.01	0.003	0.054	0.048	0.05
128×128	0.03	0.014	0.077	0.116	0.18
256×256	0.210	0.082	0.265	0.319	0.31
512×512	3.19	0.584	0.958	1.129	0.61
$1K \times 1K$	72	11.255	3.725	4.28	3.02
$2K \times 2K$	758.6	114.625	19.6	21.656	5.84
$3K \times 3K$	2940	402.7	52.8	61.31	7.62
$4K \times 4K$	6780	898.23	114.32	133.68	7.85
$1K \times 512$	5.070	2.27	1.523	3.749	1.48
$2K \times 512$	10.74	12.8	3.118	4.072	4.11
$4K \times 512$	34.33	54.7	8.311	12.418	6.58
$8K \times 32$	24.310	17.112	3.506	-	4.88
$8K \times 64$	47.87	33.7	5.016	-	6.72
$8K \times 256$	107.57	103.8	13.96	-	7.4
$8K \times 512$	137.98	215	26.33	-	8.16
$8K \times 1K$	254.26	417	50.364	-	8.2
$8K \times 2K$	1371.9	808	111.3	-	7.25

Table 1. Total computation time for SVD (in seconds) for different matrices

Table 2 gives the timings for bidiagonalization on the GPU and Intel MKL. Since Intel MKL routine performs the partial bidiagonalization, refer Section 3.1.1, we compare it with the time required for partial bidiagonalization on the GPU. We achieve a speedup of 1.58 to 16.5 over Intel MKL on partial bidiagonalization. We experimented with different values of block size. We used the block size of 1 when n is small and 16 for large n. The performance for the square matrices increases with the size of the matrix. For rectangular matrices, the performance increases with the increase in n since blocking can be performed efficiently. The bidiagonalization by Bondhugula *et al.* [7] performs only partial bidiagonalization. Their timings given on `http://www.cs.unc.edu/~geom/Numeric/svd/` are best compared with partial bidiagonalization timings given in Table 2, Column 5. Our timing is comparable (11 seconds on GTX 280, 14 seconds on 8800 GTX, compared to 19

seconds on 7900). Raw rating of the GPU speed doesn't guarantee proportionate performance on this operation as can be seen from the minor speedup on GTX 280 over 8800 GTX.

SIZE	Bidiag. GTX 280	Bidiag. 8800	Partial Bidiag. MKL	Partial Bidiag. GTX 280	Partial Bidiag. 8800
128×128	0.060	0.075	0.003	0.050	0.063
512×512	0.570	0.637	0.1478	0.373	0.430
1K×1K	2.40	2.588	3.8122	1.068	1.304
2K×2K	14.40	15.47	47.9	4.6	5.4
3K×3K	41	51.80	184	11.114	14.088
4K×4K	92.7	105.071	361.8	21.8	27.576
8K×32	1.499	–	0.020	0.143	0.066
8K×256	11.8	–	2.721	1.245	1.276
8K×512	23.8	–	13.8	2.650	2.8
8K×2K	101	–	220.500	14.3	20.281

Table 2. Bidiagonalization and partial bidiagonalization time (in seconds) for different matrices

Bobda *et al.* in [6] report only the timing for SVD computation of $10^6 \times 10^6$ matrix which takes about 17 hours. We compute SVD for much smaller matrices. Bondhugula *et al.* [7] report only the time for the bidiagonalization of the matrix. Dickson *et al.* [11] presented a programmable processor design suitable for SVD, but do not give any SVD results. Yamamoto *et al.* [25] give the optimized algorithm only for large rectangular matrices. The maximum speedup they achieve is 4 with CSX600 board over the CPU implementation for a large rectangular matrix, but get little speedup on smaller matrices.

We also compare our work efficient diagonalization algorithm with the Intel MKL diagonalization algorithm. Table 3 gives the timings for diagonalization on the GPU and Intel MKL. We achieve a speedup of 1.41 to 17.72 on diagonalization over Intel MKL implementation. The performance of our kernel is limited by the availability of registers per thread. We used 64 threads in a block for small matrices and 128 for larger matrices. It is done to keep all the multiprocessors active. We could achieve 67% to 83% occupancy on diagonalization since only 8 blocks could be active at a time. The performance increases with the increase in the size of the matrix.

Figure 5 shows the speed up achieved on GTX 280 over Intel MKL for SVD, partial bidiagonalization and diagonalization. A sustained bandwidth of 2 GB/s can be easily obtained from the CPU to the GPU. This will translate to 2 milliseconds of transfer time for 1K×1K matrix and 32 milliseconds for 4K×4K matrix. The SVD computation time makes the data transfer time irrelevant. The timings in Table 1, 2, 3, 4 and 5 exclude the cost of transferring the matrix from the CPU to the GPU since the overhead of transfer of

SIZE	Diagonalization Intel MKL	Diagonalization GTX 280	Diagonalization 8800
128×128	0.010	0.017	0.041
512×512	0.5439	0.385	0.381
1K×1K	6.417	1.3	1.347
2K×2K	49.1	5.14	5.29
3K×3K	159.413	11.6	11.821
4K×4K	354.3	20	21.7
8K×32	0.022	0.007	–
8K×256	0.564	0.159	–
8K×512	2.239	0.530	–
8K×2K	100.000	8.2	–

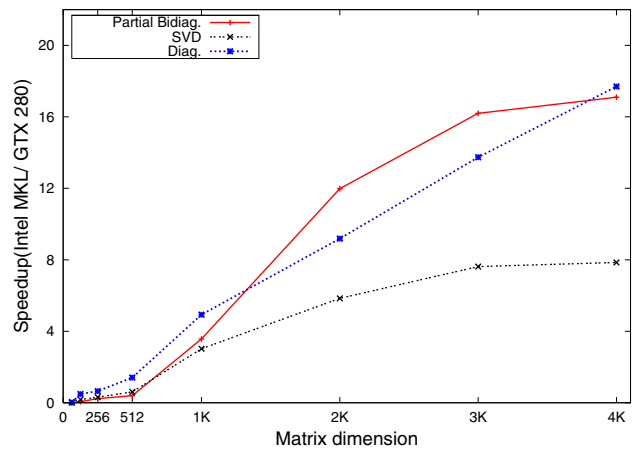Table 3. Diagonalization time (in seconds) for different matrices

Figure 5. Speedup for square matrices for SVD, Partial Bidiagonalization and Diagonalization on GTX 280 over Intel MKL

data from the CPU to the GPU and back is only to the order of tens of milliseconds.

Tables 1, 2 and 3 bring out the following points. The optimized Intel MKL does a very good job on smaller matrices, but the performance of the GPU improves with the size of the matrix. The diagonalization contributes a major share to the performance improvement on the GPU especially on larger matrices. The bidiagonalization step takes more time on the CPU than the diagonalization step. On the GPU, however, diagonalization is much faster. The GTX 280, surprisingly, improves the performance only by 10% to 15% over the 8800 GTX but is able to handle larger matrices due to the larger internal memory.

Table 4 and 5 gives the timing for computing the SVD of very large square and rectangular matrices on NVIDIA Tesla S1070. We report the timing for square matrices upto 14K×14K and rectangular matrices upto 16K×12K which is otherwise impossible to compute on the CPU due to memory limitations. SVD of a 14K×14K matrix takes about 76

SIZE	SVD	Bidiag.	Diag.	Partial Bidiag.
1K×1K	3.58	2.30	1.27	1.02
2K×2K	19.58	13.7	5.8	4.9
3K×3K	52.6	40.91	11.4	11.03
4K×4K	109.32	89.50	19.6	21.2
5K×5K	207.13	166.5	39.2	39.6
6K×6K	346.9	282.3	62.2	62
7K×7K	536.8	436.5	96.4	92.5
8K×8K	798.6	648.3	144.6	127.2
9K×9K	1131.5	911.7	211.47	176.4
10K×10K	1538.5	1226.4	301.9	234.1
11K×11K	2050	1608.8	426.6	306.5
12K×12K	2717	2104	593.3	397.3
13K×13K	3545.4	2702.1	818.38	507.2
14K×14K	4573.2	3428.8	1113	628.6

Table 4. Computation time (in seconds) for SVD, Bidiagonalization, Diagonalization and Partial Bidiagonalization for very large square matrices on NVIDIA Tesla S1070

SIZE	SVD	Bidiag.	Diag.	Partial Bidiag.
16K×1K	208.24	182.3	2.5	11.5
16K×2K	400.25	366.5	10.27	31.8
16K×3K	600.08	553.3	23.2	57.8
16K×4K	814.5	750.2	40.5	92.2
16K×5K	1042	953.8	64.1	134.4
16K×6K	1296.5	1177.4	94.5	178.9
16K×7K	1553.8	1394	134.42	230
16K×8K	1855.4	1641.1	188.2	292.1
16K×9K	2214.8	1931.3	255.9	366.6
16K×10K	2586.5	2213.7	344.23	454.5
16K×11K	2960.9	2475.6	454.3	550.2
16K×12K	3437.2	2813.2	590.7	662

Table 5. Computation time (in seconds) for SVD, Bidiagonalization, Diagonalization and Partial Bidiagonalization for very large rectangular matrices on NVIDIA Tesla S1070
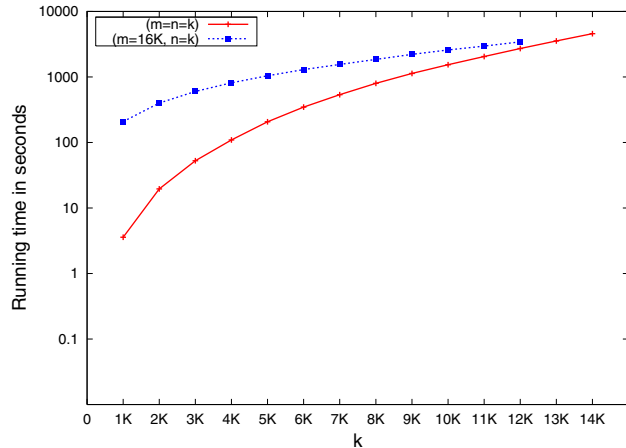

Figure 6. SVD computation time (in seconds) for very large square matrices ($m \times n$) with $m = n = k$ and rectangular matrices ($m \times n$) with $m = 16$K and $n = k$ on NVIDIA Tesla S1070

maximum difference in the singular values was 0.013% but the average was less than 0.00005%. Similarly, the maximum error of any entry in the U and V matrices was 0.01% with an average of 0.001%. Figure 7 shows the error distribution in computing the singular values of a 3K×3K matrix.
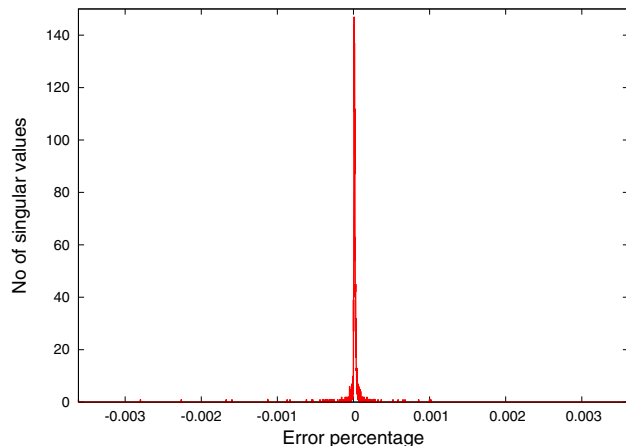

Figure 7. Discrepancy plot for singular values of a 3K×3K matrix

minutes and 57 minutes for a 16K×12K matrix on the GPU. On the CPU, we could compute the SVD of upto 10K×10K square matrices. SVD of a 10K×10K matrix takes about 4.5 hours on the CPU using Intel MKL and 25.6 minutes on the GPU using our algorithm. Figure 6 shows the timing for computing the SVD of very large matrices on NVIDIA Tesla S1070. From Table 1 and 4 we observe that the timings on NVIDIA Tesla S1070 is slightly better than on GTX 280.

GPUs are today limited to single precision arithmetic mostly. The GTX 280 has very limited double precision support, but at a very heavy performance penalty. We explored the discrepancy or error due to the reduced precision by comparing the results of the GPU version with the CPU version which uses double precision term by term. The

5. Conclusion

In this paper, we presented the implementation of the complete singular value decomposition on commodity GPUs. The algorithm exploits the parallelism in the GPU architecture and achieves high computing performance on them. The bidiagonalization of a matrix is performed entirely on the GPU using the optimized CUBLAS library

to derive maximum performance. We used a hybrid implementation for the diagonalization of the matrix that splits the computations between the CPU and the GPU, giving good performance results. We could compute the SVD of very large matrices upto the order $14K$ which is impossible on the CPU due to memory limitations. The GPUs are limited to single precision numbers, though that is changing with the newer generations. The error due to the lower precision was less than 0.001% on the random matrices we experimented with. Our approach of using CUDA and the software libraries available with it can be used for solving many other graphics and non-graphics tasks.

References

[1] ATI Corporation. 2007. ATI CTM (Close To Metal) guide. Technical report. Available on:http://www.ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.

[2] Baker, J. 1989. Macrotasking the Singular Value Decomposition of Block Circulant Matrices on the Cray-2. In *Proc. of ACM*.

[3] Barrachina, S., Castillo, M., Igual, F., Mayo, R. and Quintana-Orti, E. 2008. Solving Dense Linear Systems on Graphics Processors. In *Proc. of the European Conference on Parallel Computing*.

[4] Barrachina, S., Castillo, M., Igual, F., Mayo, R. and Quintana-Orti, E. 2008. GLAME@lab: An M-script API for Linear Algebra Operations on Graphics Processors. In *Proc. of Para'08*.

[5] Barrachina, S., Castillo, M., Igual, F. and Mayo, R. 2008. Evaluation and Tuning of the Level 3 CUBLAS for Graphics Processors. In *Proc. of Parallel and Distributed Scientific and Engineering Computing*.

[6] Bobda, C. and Steenbock, N. 2001. Singular Value Decomposition on Distributed Reconfigurable Systems. In *Proc. of 12th IEEE Workshop on Rapid System Prototyping*.

[7] Bondhugula, V., Govindaraju, N. and Manocha, D. 2006. Fast Singular Value Decomposition on Graphics Processors. Technical report. University of North Carolina.

[8] Choi, J., Dongarra, J. and Walker, D. 1995. The design of a parallel dense linear algebra software library: Reduction to Hessenberg, tridiagonal and bidiagonal form. *Numer. Alg., 10:379-399.*

[9] Christen, M., Schenk, O. and Burkhart, H. 2007. General-Purpose Sparse Matrix Building Blocks using the NVIDIA CUDA Technology Platform. Workshop on General Purpose Processing on Graphics Processing Units, Boston.

[10] Demmel, J. and Kahan, W. 1990. Computing Small Singular Values of Bidiagonal Matrices with Guaranteed High Relative Accuracy. *SIAM J. Sci. Stat. Comput. 11, 873-912.*

[11] Dickson, K. and McCanny, J. 2006. Programmable Processor Design for Givens Rotations Bases Applications. In *Proc. of 4th IEEE Workshop on Sensor Array and Multi-Channel Processing*.

[12] Fatica, M. and Jeong. W. 2007. Accelerating MATLAB with CUDA. In *Proc. of High Performance Embedded Computing*.

[13] Fujimoto, N. 2008. Faster Matrix-Vector Multiplication on GeForce 8800 GTX. In *Proc. of IEEE International Parallel and Distributed Processing Symposium*.

[14] Galoppo, N., Govindaraju, N., Henson, M. and Manocha, D. 2005. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *Proc. of ACM/IEEE Super Computing Conference*.

[15] Golub, G. and Kahan, W. 1965. Calculating the Singular Values and Pseudo-Inverse of a Matrix. *SIAM J. Num. Anal., 2:205-24.*

[16] Govindaraju, N., Gray, J., Kumar, R. and Manocha, D. 2006. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proc. of ACM SIGMOD International Conference on Management of Data*.

[17] Harish, P. and Narayanan, P. J. 2007. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of High Performance Computing*.

[18] Kruger, J. and Westermann, R. 2003. Linear Algebra Operators for GPU implementation of Numerical Algorithms. In *Proc. of SIGGRAPH*.

[19] Ma, Weiwei., Kaye, M., Luke, D. and Doraiswami, R. 2006. An FPGA-Based Singular Value Decomposition Processor. In Proc. of Canadian Conference on Electrical and Computer Engineering.

[20] Moreland, K. and Angel, E. 2003. The FFT on a GPU. In *Proc. of SIGGRAPH/Eurographics Workshop on Graphics Hardware*. pp. 112119.

[21] NVIDIA Corporation. 2007. NVIDIA CUBLAS Library. http://developer.download.nvidia.com/compute/cuda/1_1/CUBLAS_Library_1.1.pdf.

[22] Shu, Z. One Sided Jacobi Method on CUDA for SVD. Application Research of computers. http://forums.nvidia.com/index.php?act=Attach&type=post&id=8958

[23] Vineet, V. and Narayanan, P. J. 2008. CUDA-Cuts: Fast Graph Cuts on the GPU. In *CVPR Workshop on Visual Computer Vision on GPUs*.

[24] Wilkinson, J. and Reinsch, C. 1971. *Handbook for Automatic Computation: Vol. II-Linear Algebra*. Springer-Verlag. New York.

[25] Yamamoto, Y., Fukaya, T., Uneyama, T., Takata, M., Kimura, K., Iwasaki, M. and Nakamura, Y. 2007. *Accelerating the Singular Value Decomposition of Rectangular Matrices with the CSX600 and the Integrable SVD*. LNCS, Vol. 4671/2007. Springer Berlin.