A Divide-and-Conquer Approach for Solving Singular Value Decomposition on a Heterogeneous System

Ding Liu[*,†], Ruixuan Li[*], David J. Lilja[†], and Weijun Xiao[‡]
ldhust@gmail.com, rxli@hust.edu.cn, lilja@umn.edu, wxiao@umn.edu

[*]School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, Hubei 430074
P. R. China

[†]Department of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN 55455
United States

[‡]Department of Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, VA 23284
United States

ABSTRACT

Singular value decomposition (SVD) is a fundamental linear operation that has been used for many applications, such as pattern recognition and statistical information processing. In order to accelerate this time-consuming operation, this paper presents a new divide-and-conquer approach for solving SVD on a heterogeneous CPU-GPU system. We carefully design our algorithm to match the mathematical requirements of SVD to the unique characteristics of a heterogeneous computing platform. This includes a high-performance solution to the secular equation with good numerical stability, overlapping the CPU and the GPU tasks, and leveraging the GPU bandwidth in a heterogeneous system. The experimental results show that our algorithm has better performance than MKL's divide-and-conquer routine [18] with four cores (eight hardware threads) when the size of the input matrix is larger than 3000. Furthermore, it is up to 33 times faster than LAPACK's divide-and-conquer routine [17], 3 times faster than MKL's divide-and-conquer routine with four cores, and 7 times faster than CULA on the same device, when the size of the matrix grows up to 14,000. Our algorithm is also much faster than previous SVD approaches on GPUs.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Hetergeneous Systems; G.1.3 [**Numerical Linear Algebra**]: Eigenvalues and Eigenvectors; D.1.3 [**Concurrent Programming**]: Parallel Programming

General Terms

Algorithms, Performance

Keywords

Heterogeneous Architecture, Singular Value Decomposition (SVD), Divide-and-Conquer, Performance Evaluation

1. INTRODUCTION

Singular value decomposition (SVD) is a fundamental matrix decomposition algorithm widely used in various areas. It exposes useful and interesting properties of the original matrix, and it is a basic mathematic tool for various applications such as principal component analysis (PCA), signal processing and automatic control. Unfortunately, this process is quite expensive with computational complexity ranging from $12n^3$ to $\frac{4}{3}n^3$ depending on the algorithm used [1, 2]. When scientists or engineers need to use SVD in a time-sensitive context, the computation time will be a significant bottleneck. In this paper, we introduce a novel high-performance approach for solving SVD based on a heterogeneous CPU-GPU system.

To apply a heterogeneous system for solving SVD, the performance not only relies on the parallelism of the heterogeneous system, but also is highly related to the mathematical structure and characteristics of the SVD algorithm. The most widely used SVD algorithm is based on QR-iteration [5, 6, 7]. However, the iteration structure of this algorithm has substantial data dependencies that make it unsuitable for parallelization. Previous researchers tried to accelerate the SVD process using the QR-iteration algorithm. However, they were not able to achieve very good performance. GPU-based linear algebra libraries are already available, e.g. CULA [21], for solving SVD. Nevertheless, it seems that the existing GPU libraries have not yet achieved significant improvements for SVD.

Compared with the QR-iteration algorithm, the divide-and-conquer approach is supposed to be the fastest mathematical algorithm. With this approach, the global SVD problem can be naturally divided into small sub-problems, and the final result is obtained by merging the results of the sub-problems. This is a useful property for achieving high parallelism and scalability. The main components of the divide-and-conquer algorithm are blocked matrix multiplication and solving the secular equation. Both of them can be further divided and solved using multiple smaller sub-problems. These main steps are suitable for both multi-threaded execution models in conventional CPUs and the GPU's SIMT (Single Instruction Multiple Threads) execution structure.

In this paper, we present a new approach for solving SVD that exploits a heterogeneous computing system using a divide-and-conquer algorithm. To achieve the best performance on the heterogeneous system, we develop several new enhancements. First, we execute the divide-and-conquer algorithm level-by-level rather than recursively. We overlap the algorithm at task and procedure levels based on both the characteristics of the divide-and-conquer algorithm and the structure of the heterogeneous system. The CPU and GPU are assigned different tasks and cooperate to reduce the synchronization overhead. Second, to solve the secular equation, which is a key step of the divide-and-conquer algorithm, we introduce a GPU-based approach that achieves similar or even better performance than other CPU-optimized methods. More importantly, it has better numerical stability and robustness. Finally, we analyze the performance and accuracy, and discuss the adaptability of our approach. The experimental results show that our new approach can achieve a speedup of 33 against LAPACK's single threaded routine. It additionally is 3 times faster than MKL's routine with four cores, and also about 7 times faster than CULA when the matrix size is to 14,000.

The rest of the paper is organized as follows. Section 2 discusses the related work. A high-level illustration of the problem is given in Section 3. Section 4 describes our SVD approach on a heterogeneous system, and Section 5 discusses the GPU-based solver for the secular equation. Experiments and results are presented in Sections 6 and 7, respectively. Finally, the conclusions are drawn in Section 8.

2. RELATED WORK

Singular value decomposition algorithms can be generally classified based upon their basic schemas. The QR-iteration based algorithm is the most developed and widely used SVD algorithm. It can calculate the singular values and vectors with high accuracy and numerical stability [1]. To obtain higher performance than QR-iteration, especially on parallel platforms, Cuppen [10] introduced a divide-and-conquer algorithm that had previously been used for solving eigen problems. It was subsequently developed and improved by Arbenz *et al* for SVD [11, 12, 13, 14]. The divide-and-conquer algorithm is reported to be the fastest SVD algorithm and can be more than 10 times faster than the traditional QR-iteration based algorithm [2, 17]. Compared with the QR-iteration based algorithm, the divide-and-conquer algorithm is more complex, and is more sensitive to accuracy issues. Jacobi's method is the third main approach to solve SVD. While it is the slowest method, for some types of matrices it can compute the singular values and singular vectors much more accurately than the other algorithms [1].

Researchers have tried to use GPUs to accelerate solutions to the SVD problem. Sheetal et al [3] first introduced a GPU-based SVD approach based on the classical QR-iteration algorithm. In their method, they assume that the input data is already on the GPU device and the data is kept on the GPU to avoid the cost of transferring data between the host and the GPU device. This work supported no cooperation between the CPUs and the GPU. By using CUBLAS to accelerate the level-2 and level-3 matrix and vector operations, and designing a thread schema for bidiagonalization and diagonalization, they achieved an 8X speedup for single precision compared to MKL's SVD routine running on an Intel Dual Core. However, their paper did not mention which routine they had used. It should be noted that the MKL's general routine, xGESVD, is much slower than the divide-and-conquer routine, xBDSDD. Using a divide-and-conquer algorithm, our approach in this paper performs much faster than their algorithm. Additionally, we compare our work with the MKL's fastest routine, xBDSDD. In other related work, Anderson et al. recently presented a fast QR method using GPUs [4]. They described an implementation of the communication-avoiding QR (CAQR) factorization that achieved a 13X improvement compared to CULA when the matrix is extremely tall and skinny. They also purposed to use their approach for SVD. However, their approach is still a QR-iteration based SVD. Furthermore, it can only perform well for special matrices. In addition, Vedran [15] and Yamamoto [16] introduced Jacobi-based approaches using GPUs and other accelerators that focus more on accuracy rather than performance.

Besides these proposed approaches in the literature, there are several libraries available for SVD computation, such as LAPACK and MKL. MKL is a commercial math library based on LAPACK. These highly developed libraries are CPU-based and use BLAS [20] for the low level linear operations. LAPACK and MKL have QR-based and divide-and-conquer SVD implementations. CULA [21], as a commercial GPU linear algebra library, provides stable and reliable GPU algorithms. MAGMA [22] is an open source library. To the best of our knowledge, both CULA and MAGMA have only QR-based implementations of SVD, and there is no GPU based or heterogeneous system based approach using a divide-and-conquer algorithm.

3. DIVIDE-AND-CONQUER SVD ALGORITHM

Generally, the SVD of a general matrix $A \in \mathbf{R}^{m*n}(m > n)$ is computed in two phases:

- **I:** Orthogonally reduce A to a bi-diagonal matrix B:
$$A = \begin{pmatrix} U_1 & U_2 \end{pmatrix} * \begin{pmatrix} B \\ 0 \end{pmatrix} * V^T \tag{1}$$
- **II:** Compute the SVD of B:
$$B = Q * \Sigma * W^T \tag{2}$$

The SVD of A is then computed as
$$A = \begin{pmatrix} U1 & Q & U2 \end{pmatrix} * \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} * (VW)^T \tag{3}$$

Phase II takes the majority of the computation time. It can be implemented using several different algorithms, as we described previously [5, 6, 7, 8, 9, 2]. In this paper, we focus on the divide-and-conquer approach.

The basic idea of the divide-and-conquer algorithm is to merge the SVD results of two similar sub-matrices from a larger matrix. Given the $(N+1)*N$ lower bi-diagonal matrix B, divide-and-conquer recursively divides B into two submatrices as:
$$B = \begin{pmatrix} B_1 & \alpha_k e_k & 0 \\ 0 & \beta_k e_1 & B_2 \end{pmatrix} \tag{4}$$

where $B1$ and $B2$ are $k*(k-1)$ and $(N-k+1)*(N-k)$ lower bi-diagonal matrices, respectively, and e_j is the jth unit vector of appropriate dimension. Assume that the SVD of B_i is

$$B_i = \begin{pmatrix} Q_i & q_i \end{pmatrix} \begin{pmatrix} D_i \\ 0 \end{pmatrix} W_i^T \tag{5}$$

Let l_1^T and λ_1 be the last row and last component of Q_1 and q_1, respectively, and let f_2T and φ_2 be the first row and first component of Q_2 and q_2, respectively. By plugging into Formula (4), and applying Givens rotation, we obtain

$$B = Q\overline{M}W^T \tag{6}$$

Where

$$Q = \begin{pmatrix} c_0q_1 & Q_1 & 0 & -s_0q_1 \\ s_0l_1 & 0 & Q_2 & c_0q_2 \end{pmatrix} \tag{7}$$

$$\overline{M} = \begin{pmatrix} \tilde{M} \\ 0 \end{pmatrix} = \begin{pmatrix} r_0 & 0 & 0 \\ \alpha_k l_1 & D_1 & 0 \\ \beta_k f_2 & 0 & D_2 \\ 0 & 0 & 0 \end{pmatrix} \tag{8}$$

$$W^T = \begin{pmatrix} 0 & W_1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & W_2 \end{pmatrix} \tag{9}$$

and r_0, c_0, s_0 are the variables of the applied Givens rotation.

Assume $U\Sigma V^T$ is the SVD of middle matrix $\tilde{M}$. Then we obtain

$$B = \begin{pmatrix} QU q \end{pmatrix} \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} \begin{pmatrix} WV \end{pmatrix}^T = X \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} Y^T \tag{10}$$

The singular values of B are the same as the middle matrix $\tilde{M}$, and the singular vectors of B can be formed by the singular vectors of $\tilde{M}$.

To solve the SVD of $\tilde{M}$, we re-write and permute the middle matrix $\tilde{M}$ as matrix M:

$$\tilde{M} \Rightarrow M = \begin{pmatrix} z_1 & & & \\ z_2 & d_2 & & \\ \vdots & & \ddots & \\ z_n & & & d_n \end{pmatrix} \tag{11}$$

Define $D = diag(d_1, d_2, ...d_n)$ [1] with $0 \equiv d_1 \leq d_2 \leq ...d_n$. According to Jessup and Sorensen [23], the singular values of M are the roots of the secular equation

$$\begin{gathered} f(\omega) = 1 + \sum_{i=1}^{n} \frac{z_k^2}{d_k^2 - \omega^2} = 0 \\ 0 \equiv d_1 < \omega_1 < d_2 < ... < d_n < \omega_n < d_n + \|z\|_2 \end{gathered} \tag{12}$$

and the singular vectors satisfy

$$u_i = \frac{\left(\frac{z_1}{d_1^2-\omega_i^2}, \cdots \frac{z_n}{d_n^2-\omega_i^2}\right)^T}{\sqrt{\sum_{i=1}^{n} \frac{z_k^2}{\left(d_k^2-\omega_i^2\right)^2}}}, v_i = \frac{\left(-1, \frac{d_2z_2}{d_2^2-\omega_i^2}, \cdots \frac{d_nz_n}{d_n^2-\omega_i^2}\right)^T}{\sqrt{\sum_{i=1}^{n} \frac{z_k^2}{\left(d_k^2-\omega_i^2\right)^2}}} \tag{13}$$

On the other hand, if given D and all the singular vectors, we can construct a matrix with the same structure as M.

In reality, even if we can compute an approximation $\hat{\omega_i}$ close to ω_i, $z_i/\left(d_i^2 - \hat{\omega_i}^2\right)$ and $d_iz_i/\left(d_i^2 - \hat{\omega_i}^2\right)$ can still have variances from the exact ratio $z_i/\left(d_i^2 - \omega_i^2\right)$ and $d_iz_i/\left(d_i^2 - \omega_i^2\right)$. Eventually, singular vectors bias u_i and v_i. To solve this problem, Gu[14] presented a common method to use $\hat{\omega_i}$ by adopting a high accuracy solver, and then $\hat{z_i}$ is calculated as follows,

$$|\hat{z_i}| = \sqrt{\left(\hat{\omega_n}^2 - d_i^2\right) \prod_{k=1}^{i-1} \frac{\left(\hat{\omega_k}^2 - d_i^2\right)}{\left(d_k^2 - d_i^2\right)} \prod_{k=i}^{n-1} \frac{\left(\hat{\omega_k}^2 - d_i^2\right)}{\left(d_{k+1}^2 - d_i^2\right)}} \tag{14}$$

Then we use $\hat{z_i}$ to replace z_i in Formula (13) to compute the left and right singular vectors with high accuracy.

The outline of the divide-and-conquer algorithm is shown below as *Algorithm* 1.

Algorithm 1 The divide-and-conquer algorithm.

```
1:  Define a threshold $\rho$ for the minimal size of the subproblem
2:  Divide the global problem into subproblem set $S$, and generate a tree-like merging schema $M$
3:  Perform each subproblem of $S$
4:  for each merging task m in M do
5:      Construct the middle matrix and permute it to sorted (Formula (11)), and record the transformation route $R$.
6:      Solve the secular equation to obtain the singular values $\Sigma$.
7:      Revise the $z$ vector.
8:      Compute the singular vectors of middle matrix $U$ and $V$.
9:      Construct the singular vectors of the original matrix $B$, $Q$ and $W$ by $U$, $V$ and $R$.
10: end for
11: Output: the result of the final merging.
```

4. THE HETEROGENEOUS DIVIDE-AND-CONQUER SVD ALGORITHM

Generally, GPUs can be used to accelerate highly parallelized operations of the SVD algorithm, such as processing the blocked matrix multiplication and solving the secular equation. Both of these two operations can be divided into multiple independent subproblems. All of the subproblems can be solved with the same code. Thus, only a few kernel functions are needed to perform the operations without any data transfers between the CPU and the GPU. However, some other operations are not always suitable for the GPU. For example, if we construct and permute the middle matrix on the GPU (see formula (11), we need to allocate global memory on the GPU, transfer data between the CPU and the GPU, and launch a GPU kernel, in addition to constructing and sorting operations. These are instruction-intensive tasks which are difficult to execute efficiently on GPUs. According to Amdahl's law [29], the portion of the application with low or no parallelism determines the maximum possible speedup. To overcome this problem, we use the CPUs to deal with the instruction-intensive tasks, and make the CPUs and GPUs work in parallel. The following principles guide our implementation:

- Keep the GPU running as long as possible.
- For tasks that do not perform well on the GPU, let the CPU execute while keeping the GPU executing other parallel tasks.

[1] d_1 is used to simplify the presentation.

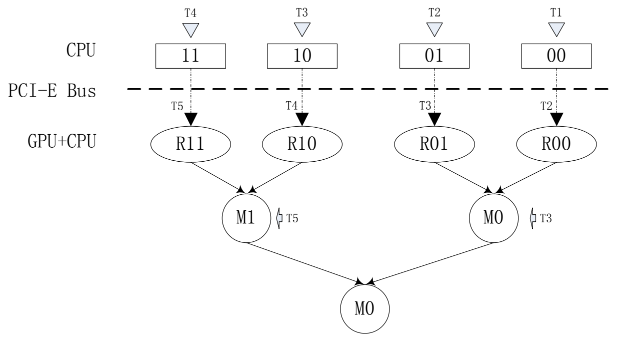

Figure 1: Overlapping the HOST and GPU at the Task Level

- For GPU-friendly tasks, let CPU execute a portion of the task, if the CPU can help without interfering with the GPU.
- For other tasks, use either the GPU or the CPU, whichever is available.

4.1 Overlapping at the Task Level

The divide-and-conquer algorithm organizes tasks recursively as a binary tree. In our implementation, each node is abstracted as a task, and each task consists of a set of procedures. A procedure is formed of several operations, such as matrix multiplications. The leaf tasks are the smallest tasks that should be handled by the QR-iteration-based algorithm, while the non-leaf tasks deal with the merging operations. Obviously, there is a direct approach to execute all of the tasks recursively using an in-order traversal. When we consider the GPU implementation, however, a recursive traversal requires a large amount of memory to store the temporary context. This is quite expensive for the GPU which typically does not have much memory available, compared to a CPU. In order to overcome this drawback, we execute the tasks breadth-first level-by-level. Thus, only one previous context needs to be stored as the input to the subsequent tasks.

Figure 1 shows the tree structure. We execute the smallest tasks using the traditional SVD method on the CPU, and then transfer the results of this level to the GPU. The GPU then focuses on merging the results of the smaller tasks. Since the tasks being merged depend on the outputs of the CPU tasks, the GPU needs to wait while the CPU completes its tasks and transfers the result to the GPU. In order to reduce the GPU's waiting time, we pipeline the CPU, system bus and the GPU at the task level. As shown in Figure 1, when the CPU deals with task ”00” at time point T1, the GPU and system bus have to wait. After the CPU completes task ”00”, it will deal with the following leaf task ”01”. At the same time, the system bus will transfer the result of task ”00” to the GPU, which is still waiting. After the CPU finishes task ”01”, it will execute task ”10”, and the bus will transfer the result of ”01” to the GPU. Once this transfer completes, the GPU starts to work until the overall algorithm has the output from the final merging on the GPU. To make the waiting time of the GPU as short as possible, we use a multi-threading approach to execute the leaf tasks concurrently. The approximate time line of the CPU, system bus, and GPU is shown in Figure 2.

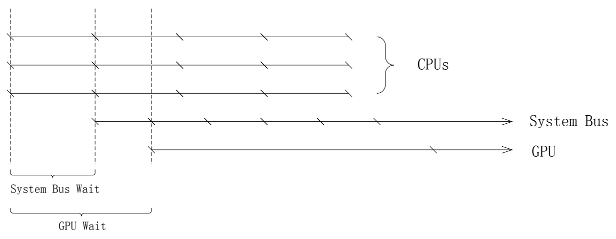

Figure 2: Timeline of the pipelined devices

From Figure 2 we can see that the wait time of the system bus is the time required to execute one leaf task, $\Delta_{bus} = T_{leaf}$, and the GPU's wait time is the time to execute one leaf task plus transfer two results to the GPU, $\Delta_{gpu} = T_{leaf} + 2 * T_{tran}$. The wait time of Δ_{bus} and Δ_{gpu} depend on the size of the leaf task. In practice, they comprise a very small part of the overall execution time, Δ_{all}. For example, with a matrix size of 14,000 and choose 800 as the size of leaf task, the GPU's waiting time is only 0.5 second, which is about 1.7% of the overall execution time. This means that the GPU is working almost all of the time during the execution of the program. Our experimental results show that this task-level overlapping reduces the overall execution time by more than 7%.

4.2 Overlapping at the Procedure Level

Since the merging operation consumes most of the execution time of the whole algorithm, we explore approaches for obtaining more independence inside of the merging task to parallelize the process at lower levels. In order to simplify the explanation of this process, we abstract the merging task to three procedures based on Algorithm 1.

- Preprocess, represented by **P**, includes processing of the middle matrix, solving the secular equation to obtain the singular values, and revising the z vector. It should be noted that this procedure depends on the result of the two subproblems, while relying only on their left singular matrices. The processing of the middle matrix has little parallelism available. However, the computational complexity of this step is only $\mathcal{O}(n^2)$. Thus, the CPU is adequate for computing this step. The other two operations have high parallelism and can be computed using either the CPU or the GPU. Hence, we execute them on the CPU or the GPU, depending on which is available at the time.
- Computing the left singular matrix Q, represented by **Q**, depends on the result of procedure **P**. Most of its operations are blocked matrix multiplications. We use only the GPU to execute this procedure to obtain the best performance.
- Computing the right singular matrix W, represented by **W**, also depends on the result of procedure **P**. Most of its operations are blocked matrix-matrix and matrix-vector multiplication. This procedure has no logical dependence with **Q**. We use only the GPU for this procedure since it also is GPU-friendly.

To illustrate the dependence among different tasks, Figure 3 shows that merging tasks ”00” and ”01” is the precursor of their parent node task ”0”. The **P** procedure of task ”0”

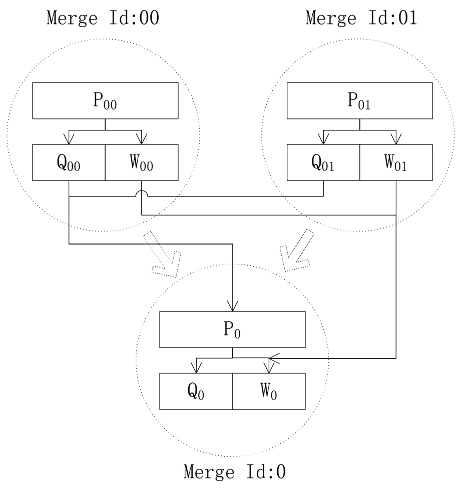

Figure 3: Procedure Level Dependence

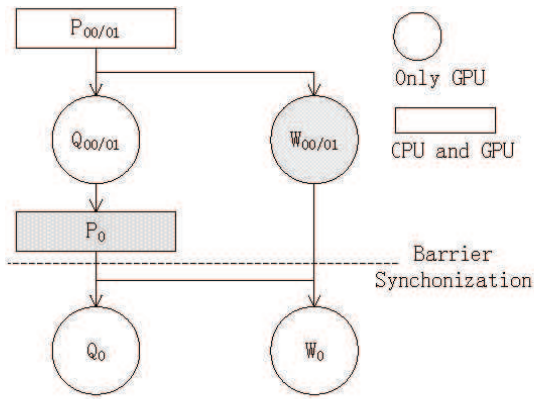

Figure 4: Overlapping the HOST and the GPU at the Procedure Level

depends on the **Q** procedure of task "00" and task "01", and the **W** procedure of tasks "00" and "01" are the only inputs of the **W** procedure of task "0". This also depends on the output of task 0's **P** procedure. Based on this observation, we can reorganize the execution order of these three tasks as shown in Figure 4. In this figure, we use a box to represent the procedure **P** to indicate that it can be executed by both the CPU and the GPU. A circle depicts procedures that are executed only by the GPU. It can be seen that the execution of $Q_{00/01}$ and P_0 can be in parallel with the execution of $W_{00/01}$. It should be noted that $Q_{00/01}$ and $W_{00/01}$ are both executed by the GPU for the best performance. However, we can use the CPU to process P_0 while the GPU is simultaneously executing $W_{00/01}$. If P_0 is still executing when the GPU finishes $W_{00/01}$, the GPU begins to execute P_0 together with the CPU. In our tests with matrix sizes from 1,000 to 14,000, when the GPU finished $W_{00/01}$, the CPU either has finished P_0 or has at least finished processing the middle matrix. By reorganizing this execution pattern, we can keep the GPU executing the GPU-friendly operations, and make the CPU handle the GPU-unfriendly parts without blocking the GPU. This optimization reduced the overall execution time by more than 8%.

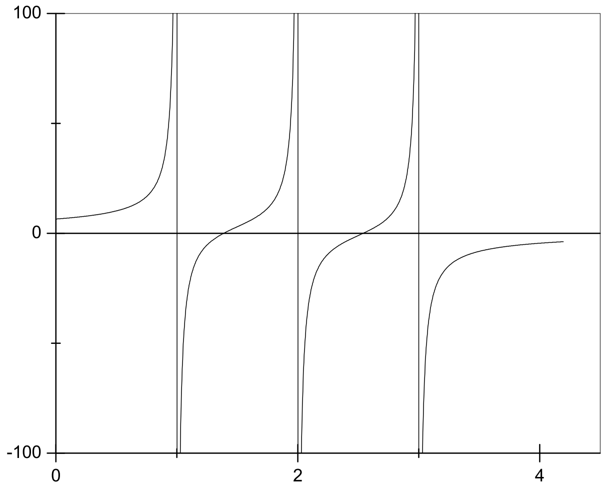

Figure 5: The distributions of the roots of $g(x) = 1 + 3/(1-x) + 3/(2-x) + 3/(3-x)$

5. SOLVING THE SECULAR EQUATION

Solving the secular equation efficiently is the key part of the divide-and-conquer algorithm. It directly determines the quality of the output singular values and greatly impacts computing the singular vectors. The mathematical form of the secular equation is shown in Formula (5). There are n roots of the equation to satisfy the conditions. Multiple iterations are needed to find each root where each iteration costs $\mathcal{O}(n)$. Therefore, the complexity is $f(\epsilon)\,\mathcal{O}\left(n^2\right)$. Here, $f(\epsilon)$ is the number of iterations required to solve each root when applying a root solver with a precision of ϵ.

To reduce the number of floating point operations, we replace the previous formula with the following:

$$g(x) = 1 + \sum_{i=1}^{n} \frac{\bar{z}_k}{\bar{d}_k - x} = 0 \qquad (15)$$

The shape of this function is very special. For example, Figure 5 shows the shape and root distribution of an sample function: $g(x) = 1 + 3/(1-x) + 3/(2-x) + 3/(3-x)$.

Newton's method is a direct solution to this problem, but it cannot guarantee convergence. When $\bar{z}$ is a small number close to *zero*, for example, $g(x) = 1+0.01/(1-x)+0.01/(2-x) + 0.01/(3-x)$, a portion of the plot of $g(x)$ looks like Figure 15. It is clear that the first order derivative of most of each interval is close to zero. If we use Newton's method to find the roots of this function, the next approximation to the true root after the first iteration will be out of the interval.

Researchers have tried to improve Newton's method using different interpolation functions to approximate the original function. The first stable method, called BNS1(2), was proposed by Bunch et al [24]. They used a second-order interpolation function to approximate the original function. BNS1 and BNS2 converged from an initial guess to the root from the left and right, respectively. Li [25] proposed a method called the Middle-Way, which also uses a second-order inter-

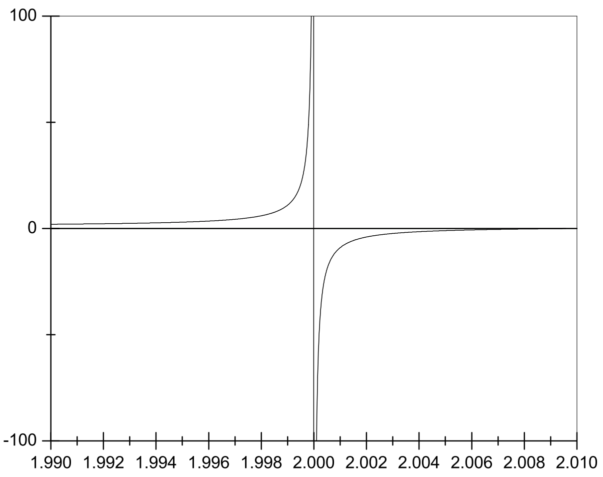

Figure 6: An expanded view of a root of $g(x) = 1 + 0.01/(1 - x) + 0.01/(2 - x) + 0.01/(3 - x)$

polation function. However, it converges from either the left or the right side to the true root. Unfortunately, all these improved methods cannot guarantee convergence. Since LAPACK has adopted the Middle-Way method [1], it is most likely the most widely used method. Additionally, Melman [26] and Gragg [28] also presented a different but similar method to solve the secular equation. In practice, regardless of whether or not these algorithms can mathematically guarantee convergence, they often slowly converge in numerical experiments due to limits of machine precision.

Compared with the algorithms based on Newton's method, the Bisection method can numerically guarantee convergence, and the next approximation to the true root is always in the middle of the upper and low bounds of the interval. As a result, out-of-interval or overflow will never occur with this method giving it good stability. However, the average convergence of the Bisection method is typically slower than previous methods. Therefore, it is usually used as a safeguard to find a solution when other methods cannot successfully generate a result. Fortunately, each iteration of the Bisection method is simpler than Newton-based methods. Here, we use the Middle-Way method as an example. This algorithm depends on a second-order interpolation function $G(x) = 1 + \phi_k(x) + \varphi_k(x)$ to approximate $g(x)$, where $k = 1, 2, ...n$ depends on which root needs to be solved, where

$$\phi_k(x) = \sum_{i=1}^{k} \frac{\bar{z}_k}{\bar{d}_k - x}, \varphi_k(x) = \sum_{i=k+1}^{n} \frac{\bar{z}_k}{\bar{d}_k - x} \tag{16}$$

Using these interpolations, the approach for finding an initial guess and the next approximation to the true root has been presented in [25]. Compared with the Bisection method, which needs to calculate only the middle point of the upper and lower bounds of an interval, Newton's method requires much more time to find the next approximation.

According to Li [25], solving the secular equation using the

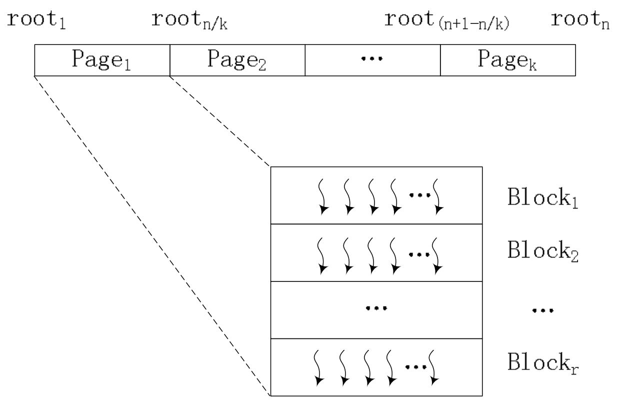

Figure 7: Solving the Secular Equation on the GPU

Middle-Way method with a length of 50 requires an average of 4.1 and 4.0 iterations for BNS1 and BNS2, respectively. The maximum number of iterations is 12 and 10, which is less than the Bisection method requires. However, it is clear that the Middle-Way needs Bisection's safeguard in several conditions so that a portion of the roots finally are found by using the Bisection method. When calculating on the CPU, all the roots are generated sequentially. When using the GPU, however, the situation is quite different. Since finding each root is independent, while the process to find a single root is iterative, the best way to solve the secular equation is parallelize the solution for each root separately. In our approach, one GPU thread is used to find one root.

Our GPU algorithm divides the problem as shown in Figure 7. The GPU solver is a loop with executing a GPU kernel function in each iteration for finding a group of roots called a "Page". When processing each page, the GPU divides the work into multiple blocks. Each block is divided into threads, which are the logical execution units. Due to the characteristics of the GPU, the threads cannot cooperate with each other when they are associated with different blocks. To avoid the issue of concurrent data accesses, a batch of threads should start and end synchronously. In other words, the slowest thread determines the execution time of each page. This feature greatly improves the time required to solve the secular equation on the GPU. When using Newton's method, if there is a need of a safeguard in a page, the overall execution time of this page is determined by the slowest thread, which actually consists of both Newton's method and the Bisection method. This is different from the implementation on the CPU.

Because of this characteristic of the GPU, we use the Bisection method instead of Newton-based methods. The Bisection method is the best algorithm for the GPU for solving the secular equation because:

- It is guaranteed to converge, which eliminates the need for a safeguard method.
- Our experimental results show that it achieve performance comparable to Newton-based methods.
- It has better numerical stability when computing the singular vectors. Formula (13) shows the process for computing singular vectors. When the solution of z is close to *zero*, the distance between d_i and ξ_i will be

very small. It can be even smaller than the precision of a float/double system. When applying Newton-based algorithms, the output ξ_i will be an infinite number because the denominator will be a numerical *zero*. The Bisection method does not have this drawback.

- It contributes to the robustness of the divide-and-conquer algorithm due to its simplicity.

6. EXPERIMENTAL METHODOLOGY

We evaluate the performance of our new algorithm using the following system configuration:

- CPU: Intel Xeon L5530 at 2.40GHz, with 8192KB cache.
- Memory: 12GB at 1066 MHz.
- Bus: PCI Express.
- GPU^1: Nvidia GeForce GTX 480.
- GPU^2: Nvidia Tesla C2050.
- GPU^3: Nvidia Tesla M2070.
- OS: Linux version 2.6.32-41-generic.
- Compiler: GCC version 4.4.3.

We use double-precision in all of the experiments. For each experiment, we choose at least five different random size matrices with multiple runs, and report the average execution time.

6.1 Baseline for Performance Comparison

We compare our algorithm with the following commercial and open-sourced CPU/GPU libraries:

- *LAPACK* (ver. 3.4.1) has two top-level SVD routines called DGESVD and DGESDD. They include both QR-based (DBDSQR) and divide-and-conquer (DBDSDC) routines for decomposing the bidiagonal matrix. LAPACK uses BLAS as its base linear matrix operations library. In our test, we use DBDSDC as a baseline and do not compare with DBDSQR since DBDSDC is a divide-and-conquer algorithm, while DBDSQR is not. In addition, DBDSQR is about 10x slower than DBDSDC.

- *Intel MKL* (Math Kernel Library, ver. 10.3) is a commercial linear software package based on LAPACK and BLAS. MKL has much better performance than directly using LAPCK on Intel's processors because Intel optimizes the library to use Intel-specific processor features. MKLâĂŹs divide-and-conquer routine is also called DBDSDC, but it provides multithreading support.

- *CULA* is an important commercial GPU library that is optimized to NVIDIA's GPU architecture. Unfortunately, there is no routine for directly decomposing a bidiagonal matrix. We use the top level SVD routine *culaDgesvd* for general matrices. We cannot directly measure the execution time cost for transforming a general matrix to a bidiagonal matrix. According to Gu [2], this part of the computation is about 60% to 70% of the overall execution time. In our test, we conservatively estimate this ratio to be 55%. Hence, we assume that 55% of the execution time of the routine *culaDgesvd* is spent computing the bidiagonal matrix.

- *Sheetal's method* [3] is also used for comparison. We simply use the experimental results presented in the paper rather than implementing the algorithm ourselves.

7. RESULTS AND ANALYSIS

We compare our new divide-and-conquer approach with the major CPU and GPU libraries, and use profiling to explain the performance results. Then we experimentally show that the Bisection method is the best approach for solving the secular equation on GPUs. We also conduct experiments to analyze the adaptability of our algorithm when changing the threshold for the minimal size of the subproblems.

7.1 Performance Comparison

Figure 8 compares the performance of our approach with the other existing libraries. While LAPACK's divide-and-conquer routine, DBDSDC, is much faster than commonly used methods, such as Matlab and DGESVD of LAPACK, MKL's divide-and-conquer DBDSDC (which is the same name as LAPACK's) runs 2-3 times faster than LAPACK on a single CPU core. This difference is due to the specific optimizations made for the Intel processor. CULA performs only 4-5 times faster than LAPACK's DBDSDC, and it is slightly faster than MKL's with one core. We believe that this difference is because CULA uses a QR-iteration approach. Even if it can achieve higher speedups than other CPU QR-iteration based routines, such as LAPACK's or MKL's DGESVD, CULA is not substantially better than the CPU-based divide-and-conquer routines. Sheetal's GPU method runs slower than MKL's DBDSDC routine with a single CPU thread. We note that the device Sheetal used was an S1070, which is an older GPU device. However, the main reason for the slowdown is the use of the QR-iteration algorithm. Our algorithm performs well when the matrix size is larger than 2000, and is even faster than MKL with 4 CPU cores (8 hardware threads) when the matrix size grows larger than 3000. As the matrix size further increases, our algorithm can achieve even higher speedups. When the matrix size is up to 14,000, our algorithm is 33 times faster than LAPACK, 3 times faster than MKL with 4 cores, and 7 times faster than CULA executing on the same device.

Figure 8 also shows that the speedup of our algorithm grows as the matrix size increases. For the largest size matrix, our approach has better efficiency than with smaller matrices. In order to explain this phenomenon, we profile the performance of our implementation. Figure 9 shows the execution times of the different types of operations compared to the total execution time. We classify the operations into three types: matrix multiplication, secular equation solving, and others. It is clear that, when the matrix size is small, most of the time is spent on other types of operations, including data transfers, launching the kernel, synchronization delays when dealing with minimal subproblem sizes, sorting, and so on. These operations have little parallelism and are not suitable for the GPU. The fraction of time spent doing matrix multiplication with high parallelism is only 20%. Solving the secular equation has high parallelism as each

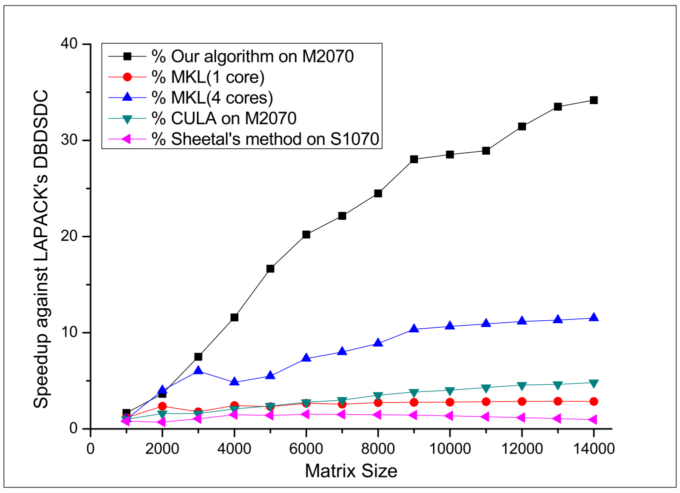

Figure 8: Overall Performance Comparison

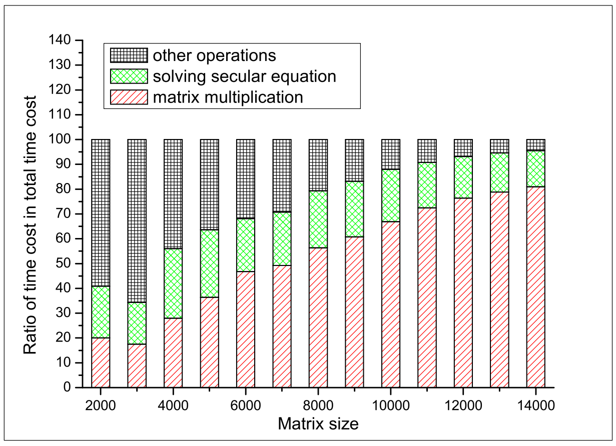

Figure 9: Execution Time Profile of the New Divide-and-Conquer Algorithm

root is found using a separate thread. It only requires only 19% of the total execution time. Hence, the overall parallelism is low when the matrix is small. When dealing with the large matrices, however, the ratio of matrix multiplication and secular equation solving grows. When the problem size grows to 14,000, matrix multiplication takes about 80% of the total execution time, while solving the secular equation takes about 10%. Obviously, there is a great benefit to executing these portions of the problem on the GPU. We conclude that our algorithm is able to exploit substantial parallelism when dealing with large matrices.

7.2 Performance of the Secular Equation Solver

In this section, we experimentally validate the selection of the secular equation solver by comparing the performance of the Bisection method with the most important optimized Newton-based algorithm BNS and the Middle-Way method. The experiments have been conducted on both the CPU and multiple GPUs.

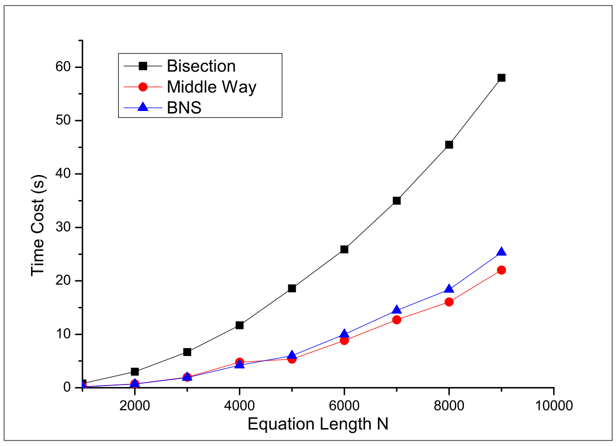

Figure 10: Performance of the Secular Equation Solver on the CPU

Figure 10 shows that both the Middle-Way method and BNS are faster than the Bisection method. These results are consistent with our previous analysis that the execution time on the CPU depends on the average number of iterations and the cost of each iteration. However, the situation is totally different on the GPU. As shown in Figure 11, the Middle-Way method and BNS do not have any advantage when executed on the GPU. As the length of the equation grows, their performance becomes even worse than the Bisection method. This result is due to several reasons. First, as the number of terms in the equation grows, the need for safeguarding grows. Specifically, there is a high probability of requiring at least one safeguard step using the Bisection method in a page. Second, no matter which algorithm is applied, if the result does not converge in an iteration, the program needs to find the next approximate point. This step is more expensive when using the Middle-Way and BNS methods. However, the Bisection method needs only one floating point operation to calculate the middle point of the upper and lower bounds. Third, if the memory to store all of the threads' context exceeds the available shared memory and registers, the GPU will use global memory instead, which is substantially slower than the block's local shared memory. Specifically, each GPU block has rather limited share memory and registers. For example, on GPU^1, the total amount of shared memory per block is 49152 bytes plus 32768 registers. Thus, the maximum local memory is sufficient for only 10K double floating-point numbers. Finding the next approximate point for both the Middle-Way and BNS methods [24, 25] requires substantial memory to store the program context since they need to calculate two interpolation functions and their first-order derivatives. Hence, in order to avoid accessing the global memory frequently, we must use fewer threads in a block for the Middle-Way and BNS methods, which compromises the thread-level parallelism. In contrast, the Bisection method uses a simple approach to generate the next approximate point. Therefore, each of its threads does much less work allowing more threads in a block, which then makes better use of the GPU resources.

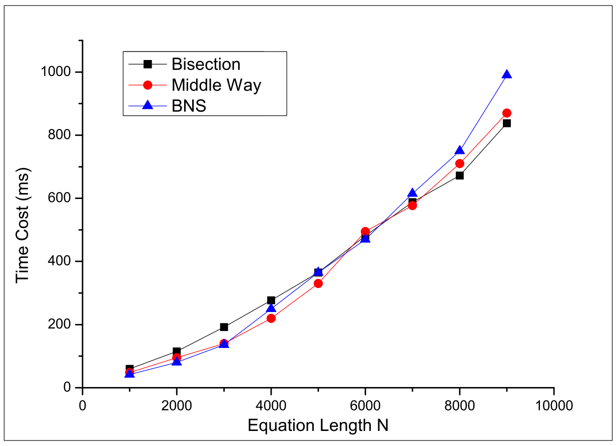

Figure 11: Performance of the Secular Equation Solver on the GPU

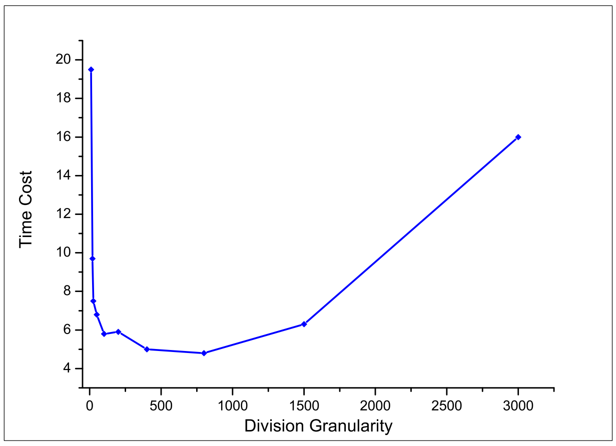

Figure 12: The sensitivity of the execution time to the divide-and-conquer granularity

7.3 Adaptability Analysis

To illustrate how the granularity of the divide-and-conquer algorithm affects performance, we measure the execution time required to solve a 6000-by-6000 element bidiagonal matrix on GPU^3. Figure 12 shows the relationship between the granularity and the performance. It is noted here that the granularity defines a threshold for the minimal size of the subproblem, which is also the matrix size of a leaf node in the task tree. From Figure 12, one can see that the performance is sensitive to very large and very small divide-and-conquer granularities, although it has a broad region of stable values in between these extremes. For example, when the granularity is 20 for the matrix size of a leaf node, there will be 512 leaf nodes that need to be handled by the CPU, 511 merging tasks on the GPU, and more than 512 data transfers on the system bus. Most of these tasks are very small, which leads to very low utilization of the system. At the other

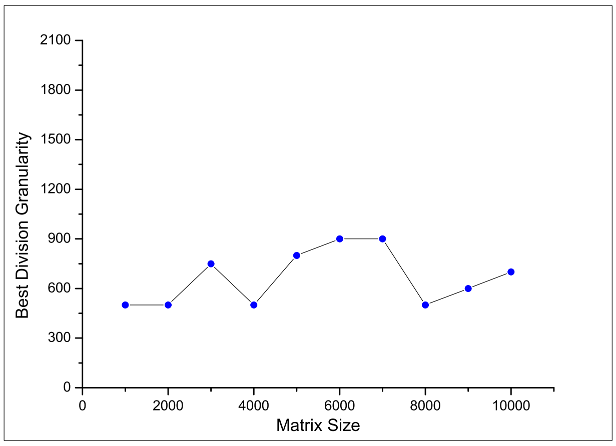

Figure 13: The best divide-and-conquer granularity for different matrix sizes

extreme, when the matrix size of a leaf node exceeds 1500, there is insufficient work to be performed on the GPU causing the CPU to become the bottleneck while executing the LAPACK code. When this parameter is between 100 and 1000, the CPU and GPU tasks are well balanced. Similar results are obtained when using the other GPU devices.

Figure 13 shows that the best granularity varies somewhat with the size of the matrix. Fortunately, the best granularity is typically between 500-1000 for the size of matrices tested. We conclude that the algorithm performs quite well over a broad range of granularity values on all of the GPUs.

8. CONCLUSION

We have presented a new divide-and-conquer algorithm for a heterogeneous CPU-GPU system for solving singular value decomposition (SVD). Compared with previous methods that use traditional QR-iteration methods with GPUs, our approach has higher parallelism and better performance. We have carefully designed our algorithm to match the mathematical requirements of SVD to the unique characteristics of this heterogeneous platform. These design enhancements include a high-performance solution to the secular equation with good numerical stability, overlapping the CPU and GPU tasks, and leveraging the CPU-GPU interconnection bandwidth.

We have conducted extensive experiments to evaluate the performance of this new approach. The results show that, when using an Nvidia M2070 GPU, our algorithm is up to 33 times faster than LAPACK's divide-and-conquer routine, 3 times faster than MKL's divide-and-conquer routine with 4 cores, and 7 times faster than CULA on the same GPU device.

Acknowledgments

This paper is based upon work supported by the US National Science Foundation (NSF) under Grant ITR-0937060 to the Computing Research Association for the CIFellows Project and by NSF grant no. EAR-0941666, the Minnesota Supercomputing Institute, and NVIDIA Corporation. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF. This work is also supported by National Natural Science Foundation of China under grants 61173170 and 60873225, and Innovation Fund of Huazhong University of Science and Technology under grants 2012TS052 and 2012TS053.

The authors would like to thank the anonymous reviewers for the valuable comments that improved the quality of the presentation of this paper.

9. REFERENCES

[1] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[2] Ming Gu, James Demmel and Inderjit Dhillon, *Efficient Computation of the Singular Value Decomposition with Applications to Least Squares Problems.* In Technical Report CS-94-257, Department of Computer Science, University of Tennessee,October 1994.

[3] Sheetal Lahabar, P. J. Narayanan. *Singular Value Decomposition on GPU using CUDA.* IPDPS, 2009.

[4] M. Anderson, G. Ballard, J. Demmel,K. Keutzer. *Communication-Avoiding QR Decomposition for GPUs.* IPDPS, 2011.

[5] J. Demmel and W. Kahan. *Accurate singular values of bidiagonal matrices.* SIAM J. Sci. Stat. Comput., 11(5):873-912, 1990.

[6] G. Golub and W. Kanhan. *Calculating the singular values and pseudo-inverse of a matrix.* SIAM J. Num. Anal. (Series B), 1965.

[7] G. Golub and C. Reinsch. *Singular value decomposition and least squares solutions.* Num. Math., 1970.

[8] B. Parlett and V. Fernando. *Accurate singular values and differnetial qd algorithms.* Math Department PAM-554, University of California, Berkeley, July 1992.

[9] H. Rutishauser. *Lectures on Numerical Mathematics.* Biskhauser, 1990.

[10] J. J. M. Cuppen. *A divide and conquer method for the symmetric tridiagonal eigenproblem.* Numer. Math., 1981.

[11] P. Arbenz and G. Golub. *On the spectral decomposition of Hermitian matrices modified by row rank perturbations with applications.* SIAM J. Matrix Anal. Appl. 1988.

[12] G. H. Golub. *Some modified matrix eigenvalue problems.* SIAM Review, 1973.

[13] M. Gu and S. Eisenstat. *A stable algorithm for the rank-1 modification of the symmetric eigenproblem.* Report YALEU/DCS/RR-916, Yale University, 1992.

[14] M. Gu and S. Eisenstat. *A divide-and-conquer algorithm for the bidiagonal SVD.* Report YALEU/DCS/RR-933, Yale University, 1992.

[15] Vedran Novakovi and Sanja Singer. *A GPU-based hyperbolic SVD algorithm.*BIT 51(2011), 1009-1030

[16] 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.

[17] E. Anderson, Z. Bai. *LAPACK Users' Guide.* $http://www.netlib.org/lapack/lug/lapack_lug.html$

[18] Intel MKL 10.3 *MKL Reference Manual*, available at $http://software.intel.com/$

[19] Intel MKL 10.3 *Singular Value Decomposition*, available at $http://software.intel.com/$

[20] *BLAS (Basic Linear Algebra Subprograms)*, available at $http://www.netlib.org/blas/$

[21] *CULA*, available at $http://www.culatools.com$

[22] *MAGMA*, available at $http://icl.cs.utk.edu$

[23] E. Jessup and D. Sorensen. *A parallel algorithm for computing the singular value decomposition of a matrix.* Mathematics and Computer Science Division Report ANL/MCS-TM-102, Argonne National Lab, 1987.

[24] J.R. Bunch, C.P. Nielsen, and D.C. Sorensen. *Rank-one modification of the symmetric eigenvalue problem.* Numer. 1981.

[25] R.C. Li, *Solving Secular Equation Stably and Efficiently.* Tech. Report UCB/CSD-94-851, 1994.

[26] A. Melman, *Numerical solution for a secular equation.* Numer. Math. 1995.

[27] A. Melman, *A numerical comparsion of methods for solving secular euqations.* J. of Computational and Applied Math., 1997.

[28] C.F. Borges, W.B. Gragg. *A parallel divide and conquer algorithm for the generalized real symmetric definite tridiagonal eigenproblem.* Numerical Linear Algebra and Scientific Computing, 1993.

[29] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.