Neural, Parallel, and Scientific Computations 16 (2008) 543-562

Memory Hierarchy Exploration for Accelerating the Parallel Computation of SVDs

Mostafa I. Soliman
Computer & System Section, Electrical Engineering Department, Faculty of Engineering,
South Valley University, Aswan, Egypt

Abstract
The performance of many applications on modern computers is often limited by memory latency rather than by processor speed. For computers with memory hierarchy, it is preferable to perform the computation on blocks of data to reduce the impact of memory latency by reusing the loaded data in cache memories. This paper proposes a fast algorithm for parallel computing the extremely useful singular value decomposition (SVD) based on one-sided Jacobi on multi-level memory hierarchy architectures. On P parallel processors, the given matrix is divided into super-rows and then these super-rows are partitioned into $2P$ blocks. One key point of the proposed algorithm is the highly exploitation of memory hierarchy by performing all computations on super-rows loaded in cache memory rather than on rows. Another key point is that the number of sweeps required for convergence is very close to cyclic one-sided Jacobi. Third key point of the proposed algorithm is that the number of sweeps required for convergence does not depend drastically on the size of the input matrix. On two dual-core Intel Xeon processors, our results show that the performance of parallel implementation of the proposed algorithm is around 11 times higher than the sequential implementation on the same hardware. Moreover, a performance of around 10 GFLOPS (double-precision) can be achieved on the target system using multi-threading, Intel SIMD instructions, matrix blocking, and loop unrolling techniques.
Keywords - memory hierarchy, multi-core computation, multi-threading techniques, parallel algorithms, performance evaluation, SIMD, SVD, one-sided Jacobi.

1. INTRODUCTION

The gap between CPU speed and memory speed is increasing rapidly as new generations of computer systems are introduced [1, 2]. To address this memory access bottleneck, most modern computers use multi-level memory hierarchies in their architectures. The key to improve the performance of applications on multi-level memory hierarchies is to avoid unnecessary memory references as well as to exploit locality by reusing the loaded data into a higher-level cache [3]. In computers with multi-level memory hierarchies, data flows hierarchy from/to main memory to/from registers through cache memories (off/on-processor). Then data flows from registers into and out of

Received June 1, 2007

functional units, which perform the given instructions on the data. Therefore, an algorithm performance can be dominated by the amount of memory traffic rather than by the number of arithmetic operations involved. The movement of data between memory and registers can have the same (or even more) cost as arithmetic operations.

The cost of moving data to and from main memory provides a considerable motivation to restructure existing algorithms and to devise new algorithms that minimize data movement by reusing the loaded data into cache memories [4]. A number of researchers have demonstrated the effectiveness of block algorithms on a variety of modern computer architectures (see [3] for more detail). On these computers potentially high performance can easily be degraded by excessive transfer of data between different levels of memory (registers, cache, or main memory). In particular, for computers with memory hierarchy, it is often preferable to partition the input data and to perform the computation on the blocks. This approach provides for full reuse of data while the block is held in cache memory. It avoids excessive movement of data to/from main memory and gives a surface-to-volume effect for the ratio of arithmetic operations to data movement, i.e., $O(n^3)$ arithmetic operations to $O(n^2)$ data movement [5]. In addition, on architectures that provide for parallel processing, parallelism can be exploited in two ways: (1) operations on distinct blocks may be performed in parallel; and (2) within the operations on each block [4]. For example, on multi-core Intel processors [6], the given problem can be partitioned into blocks. Parallel proceeding of these blocks are performed on multiple cores to reduce the total execution time. Besides, Intel SIMD instructions [7] may be used to perform operations on a block in parallel to further improve the performance.

In linear algebra, the singular value decomposition (SVD) is an important factorization of a real matrix, with several applications in signal processing, data mining, statistics, etc [8-10]. SVD problem is a very computationally intensive problem that needs to exploit the growing availability of parallel hardware. The sequential implementation of the standard Golub-Kahan SVD algorithm [11] on an $m \times n$ matrix takes $O(mn^2)$ time. This amount of time may not be acceptable especially when the data size is large. Thus, many researchers have worked on designing efficient techniques to compute SVDs on parallel to reduce the execution time especially for real time applications [12-16]. This paper proposes a new algorithm for parallel computing SVD on multi-level memory hierarchy architectures by restructuring the well-known one-sided Jacobi method. Hestenes one-sided Jacobi method [17] is selected because it is the best approach for achieving efficient parallel SVD computation (see [12] for more detail).

The proposed algorithm, which is called hierarchal block Jacobi (HBJ), partitions the given matrix into super-rows (panels of rows) to exploit the memory hierarchy by

performing all computations on super-rows instead of on rows. Each super-row consists of a set of consecutive rows of the input matrix. On P parallel processors, these super-rows are partitioned into $2P$ blocks to be processed in parallel. Not all block sizes are necessarily have the same number of super-rows (the number of super-rows in a block depends on P, however, the number of rows in a super-row is constant). This results in partitioning the input matrix hierarchy from rows to super-rows, and then to blocks of super-rows. Our result shows that the hierarchal partitioning of the input matrix reduces the number of sweeps required for convergence. Besides, in the proposed HBJ algorithm the number of sweeps required for convergence does not depend drastically on the size of the input matrix or on the number of parallel processors but depends on the size of super-rows.

This paper implements the proposed HBJ algorithm for parallel computing SVD on multi-core Intel processors. The target system is Dell Precision 690 running Microsoft Windows Vista operating system. It has two dual-core Intel Xeon processors with hyper-threading technology running at 3.0 GHz, 2 GB memory, and unified, shared 16-way second-level cache of 4 MB [6]. Our results show that on Intel Xeon processors, a good performance of the proposed HBJ algorithm can be obtained by exploiting the multi-core hardware, memory hierarchy, and Intel SIMD instruction set. Besides, the use of matrix blocking and loop unrolling techniques further improves the performance of the proposed algorithm. To be specific, the execution time of the HBJ algorithm on the target system is around 11 times faster than the sequential implementation on the same hardware.

This paper is organized as follows. Section 2 reviews of the singular value decomposition problem briefly. Since most of parallel algorithms for computing SVDs are based on Jacob technique, one-sided and two-sided Jacobi methods are reviewed. The proposed algorithm for fast implementation of SVD on multi-level memory hierarchy systems is presented in Section 3. Section 4 shows the implementation and performance evaluation of the proposed algorithm on dual-core Intel Xeon processors. Finally, Section 5 concludes this paper and points out the key points of the proposed algorithm.

2. SINGULAR VALUE DECOMPOSITION

This section provides a brief review of singular value decomposition (SVD) that is well known in matrix algebra [10]. SVD of a real matrix $A_{m\times n}$ ($m \geq n$) is its factorization into the product of three matrices $U_{m\times m}$, $\Sigma_{m\times n}$, and $V_{n\times n}$ such that:

$$A_{m\times n} = U_{m\times m}\, \Sigma_{m\times n}\, V^{\mathrm{T}}{}_{n\times n}$$

where $U_{m\times m}$ and $V_{n\times n}$ are orthogonal matrices (i.e. $U^{\mathrm{T}}U = \mathrm{I}_m$ and $V^{\mathrm{T}}V = \mathrm{I}_n$), and $\Sigma_{m\times n}$ is a diagonal matrix $diag(\sigma_0, \sigma_1, \sigma_2, \ldots\ldots, \sigma_{n-1})$ on top of $(m\text{-}n)$ rows of zeros, assuming that $m \geq n$. The σ_i are the singular values of $A_{m\times n}$. Matrix $U_{m\times m}$ contains n left singular

vectors, and matrix $V_{n\times n}$ consists of n right singular vectors. The singular values and singular (column) vectors of $U_{m\times m}$ and $V_{n\times n}$ form the relations:

$$Av_i = \sigma_i u_i \qquad \text{and} \qquad A^{\mathrm{T}}u_i = \sigma_i v_i.$$

This decomposition has many important scientific and engineering applications [8-10].

There are various ways to compute the SVD. Two of the most commonly used classes of algorithms are bidiagonalization-based and Jacobi-based [10]. The standard bidiagonalization-based method was introduced by Golub and Kahan in 1965 [11]. It uses first the Householder transformation to bidiagonalize the given matrix and then the QR method to compute the singular values of the resultant bidiagonal form. In sequential computing, the bidiagonalization-based algorithms are usually preferred because they are faster than the sequential implementation of Jacob algorithms. However, on an $m\times n$ matrix, $O(mn^2)$ clock cycles are needed for implementing the standard Golub-Kahan SVD algorithm sequentially. On a large matrix size (large values of m and n), the execution time is unacceptable especially for real-time applications. Moreover, the bidiagonalization-based algorithms have been found to be difficult to parallelize. In [18], Luk gives three reasons why the standard SVD method of Golub-Kahan may be undesirable on a parallel processor.

On the other hand, the Jacob-based SVD algorithms may be more accurate and have a higher degree of potential parallelism [19]. Thus, most of parallel algorithms are based on Jacob technique. There are two varieties of Jacobi-based algorithms, one-sided and two-sided. As shown below, the two-sided Jacobi algorithms are computationally more expensive than the one-sided algorithms. Moreover, to implement the two-sided Jacobi method, it needs to traverse both row and column of the given matrix; however, matrices are stored either in row-major or column-major format. Thus, one of the two traversals will be less efficient on conventional memory architectures. In other words, one of the two traversals accesses the elements of the input matrix $A_{m\times n}$ with unit stride, which is efficient, however, the other traversal performs stride n accesses, which is expensive because of cache miss handling time [7]. On the other hand, the one-sided rotation modifies rows only, which is more suitable for memory hierarchy architectures. Thus, to achieve efficient parallel SVD computation the best approach may be to adopt the Hestenes one-sided Jacobi transformation method [17] as advocated in [12, 20].

The two-sided Jacobi iteration algorithm transforms a symmetric matrix $A_{n\times n}$ into a diagonal matrix $\Sigma_{n\times n}$ by a sequence of Jacobi rotations (J), where each transform attempts to zero-out a given off-diagonal element of $A_{n\times n}$.

$$\Sigma_{n\times n} = (J_n^T \cdots (J_3^T (J_2^T (J_1^T A J_1) J_2) J_3) \cdots J_n) = (J_1 J_2 J_3 \cdots J_n)^T A (J_1 J_2 J_3 \cdots J_n).$$

The Jacobi rotation $J(i, j, \theta)$ for an index pair (i, j) and a rotation angle θ is a square matrix that is equal to the identity matrix I plus four additional entries at the intersections of rows and columns i and j:

$$J(i, j, \theta) = \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$. It is clear that the Jacobi rotation is an orthogonal matrix (i.e. $J(i, j, \theta)^{\mathrm{T}} J(i, j, \theta) = \mathrm{I}$) using the fact of $\cos^2(\theta) + \sin^2(\theta) = 1$. The rotation parameters c and s are computed such that the resultant matrix $B = J^T A J$ is diagonal, *i.e.*, $b_{ij} = b_{ji} = 0$.

$$\begin{pmatrix} b_{ii} & b_{ij} \\ b_{ji} & b_{jj} \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

By solving this equation and taking the smaller root, c and s are obtained by:

$$c = \frac{1}{\sqrt{1 + t^2}}$$

and

$$s = tc$$

where

$$t = \frac{sign(\tau)}{|\tau| + \sqrt{\tau^2 + 1}}$$

and

$$\tau = \frac{a_{jj} - a_{ii}}{2a_{ij}}.$$

Depending on the order of choosing the element to be zeroed, there are classic Jacobi and cyclic Jacobi algorithms. In the classic Jacobi iteration algorithm, each transformation chooses the off-diagonal element of the largest absolute value. However, searching for this element requires expensive computations. Cyclic Jacobi algorithm sacrifices the convergence behavior and steps through all the off-diagonal elements in a row-by-row or column-by-column fashion. It needs to perform $n(n - 1)/2$ rotations attempting to diagonalize an $n \times n$ symmetric matrix. These $n(n - 1)/2$ rotations constitute a sweep. Note that when an off-diagonal element is zeroed it may not continue to be zero when another off-diagonal element is zeroed. Since the norm of the off-diagonal elements

decreases after each sweep, a finite number of sweeps are required for Jacobi algorithms to converge. There are two important implementation details which determine the speed of convergence of the Jacobi based algorithms for computing the SVD. The first is the method of ordering, i.e., how to order the $n(n - 1)/2$ rotations in one sweep of computation. Various orderings have been introduced in the literature. The second important detail is the method for generating the plane of rotation parameters c and s in each iteration.

In Hestenes one-sided Jacobi algorithm [16], SVD of a real matrix $A_{m\times n}$ ($m \geq n$) is computed by generating an orthogonal matrix $U_{m\times m}$ such that the transformed matrix $B_{m\times n}$ has orthogonal rows.

$$U_{m\times m} A_{m\times n} = B_{m\times n},$$

where rows of $B_{m\times n}$ satisfy:

$$b_i^T b_j = 0 \text{ for } i \neq j.$$

By normalizing the Euclidean length of each non-zero row to unity, the following relation can be obtained:

$$B_{m\times n} = S\, V_{n\times n},$$

where $S_{n\times n} = diag(s_1, s_2, \ldots, s_n)$, and $s_i = b_i^T b_i$ and $V_{n\times n}$ is a matrix whose non-zero rows form an orthogonal set of vectors. An SVD of $A_{m\times n}$ is given by

$$A_{m\times n} = U^T{}_{m\times m}\, S_{m\times n}\, V_{n\times n.}$$

Hestenes [17] suggested that the orthogonal matrix $U_{m\times m}$ should be constructed as a sequence of plane rotations.

$$A_{m\times n} = (J_n^T \cdots (J_3^T (J_2^T (J_1^T A))) \cdots) = (J_1 J_2 J_3 \cdots J_n)^T A$$

where, J_k is a plane rotation. Each plane rotation affects only two rows. For a given i and j, rows i and j are orthogonalized by $B_{m\times n} = J^T A_{m\times n}$ where $J = J(i, j, \theta)$ is the same matrix as in the two-sided Jacobi:

$$\begin{pmatrix} b_i^T \\ b_j^T \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_i^T \\ a_j^T \end{pmatrix}$$

here c and s are chosen such that $b_i^T b_j = 0$. The solution of them is:

$$c = \frac{1}{\sqrt{1+t^2}}$$

and

$$s = tc,$$

where

$$t = \frac{sign(\tau)}{|\tau| + \sqrt{\tau^2 + 1}}$$

and

$$\tau = \frac{a_j^T a_j - a_i^T a_i}{2a_i^T a_j}.$$

It is clear that there is a close similarity between one-sided and two sided versions of the Jacobi algorithm.

3. THE PROPOSED ALGORITHM: HIERARCHAL BLOCK JACOBI (HBJ)

Traditionally, parallel algorithms for computing SVD on $n \times n$ matrix partition the $n(n - 1)/2$ rotations of a sweep into rotation sets. Each rotation set consists of some number of independent rotations. Therefore, all the rotations of a rotation set can be performed in parallel. Most of the parallel SVD algorithms in the literature employ $(n - 1)$ rotation sets, with each rotation set consisting of $n/2$ independent rotations. The Stream Hestenes SVD algorithm [15] is an exception, where all rotations can be performed in parallel even though they are dependent.

The proposed HBJ algorithm employs the idea of calculating rotations parameters (c and s) at once and then applying all of them also at once to exploit hierarchal memory. This is a good step for switching from vector operations, which require $O(n)$ memory accesses for $O(n)$ FLOPs, to matrix operations, which require $O(n^2)$ memory accesses for $O(n^3)$ FLOPs. Matrix operations are performed based on matrix blocking technique, which is an efficient technique to improve the performance of many matrix-based applications on a variety of modern computer architectures with parallel processing capabilities [3, 4].

The main problem of the direct applying this idea of Stream Hestenes SVD algorithm [15] is the increase of the number of sweeps required for convergence. To be specific, Figure 1 shows the number of sweeps for parallel computing SVD of real matrices on eight processors. Based on the partitioning method described in [21], the input matrix $A_{n \times n}$ is portioned into 16 ($2P$ and $P = 8$) blocks, with each block consisting of $\lceil n/16 \rceil$ consecutive rows of $A_{n \times n}$. It is clear that as the matrix size increases, the number of sweeps required for convergence increases. For a large matrix size, the number of sweeps using Stream Hestenes SVD algorithm is orders of magnitude greater than that required by cyclic Jacobi algorithm (see Figure 1). However, we observe that the number of sweeps is close to the cyclic one-sided Jacobi when the number of rows per block is small (see Figure 2). Approximately, the total execution time for computing SVD equals the execution time of a sweep times the number of sweeps. Thus, our approach to reduce

the total execution time for computing SVD is that: (1) parallel processing techniques would be used to reduce the execution time per sweep; and (2) the use of a small number of rows per block would reduce the number of sweeps required for convergence.

The idea of our proposed HBJ algorithm is as follows. Instead of partitioning the input matrix $A_{n\times n}$ based on the number of parallel processors (P) into $2P$ blocks, $n/2P$ rows each, the input matrix is partitioned into N super-rows (panels of rows). Each super-row consists of S ($1 \le S \le \lceil n/2P \rceil$) consecutive rows of the input matrix $A_{n\times n}$. These N super-rows are then partitioned into $2P$ blocks; each block has $\lfloor N/2P \rfloor$ or $\lceil N/2P \rceil$ super-rows. One advantage of the HBJ algorithm is the highly exploitation of memory hierarchy by performing all computations on super-rows using matrix operations (based

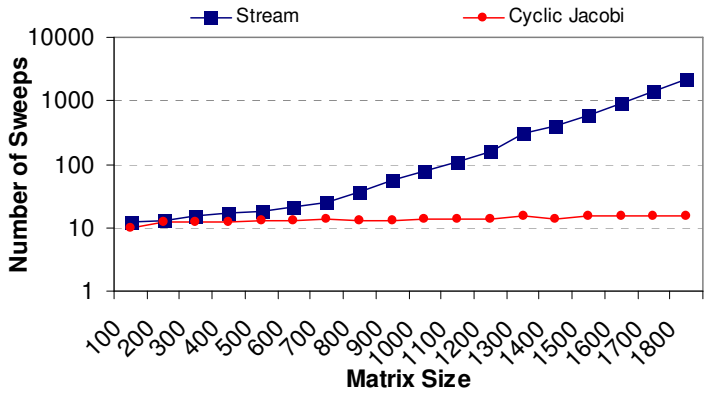

Figure 1: Number of sweeps of SVD on 8 parallel processors

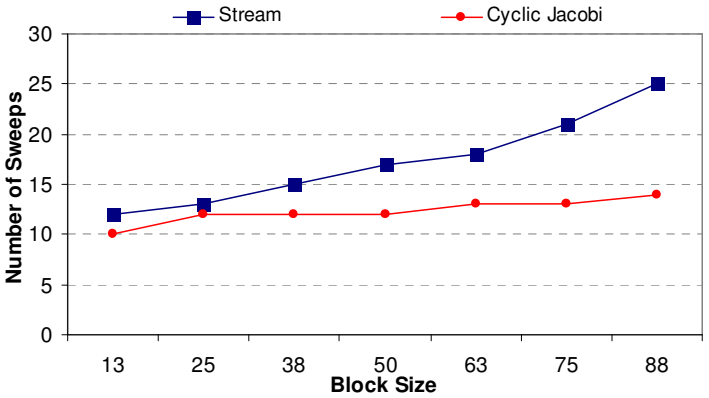

Figure 2: Number of sweeps for Stream and Cyclic Jacobi

on matrix blocking technique) instead of on rows using vector operations (based on strip mining technique). Performing matrix operations on super-rows loaded in cache memories exploits the memory hierarchy by reusing these loaded data. Another advantage is that the number of sweeps required by HBJ for parallel computing SVD is

very close to cyclic one-sided Jacobi, as shown in details in Section 4. Moreover, the number of sweeps required for converging the proposed HBJ algorithm does not depend drastically on the size of the input matrix or the number of parallel processors but on the size of super-rows. These advantages lead to reducing the total execution time required for parallel computing the SVD problem on multi-level memory hierarchy architectures.

There are two implementation approaches when S does not divide n. One approach is extending the input matrix with rows of zeros, which results in N equals $\lceil n/S \rceil$ and S is constant. The other approach uses super-rows with different sizes (S is variable), which results in N equals $\lfloor n/S \rfloor$ and S varies from $\lfloor n/N \rfloor$ to $\lceil n/N \rceil$ rows. On other words, $(n - S*\lfloor n/S \rfloor)$ super-rows have a size of $\lceil n/N \rceil$ rows and the remaining super-rows have a size of $\lfloor n/N \rfloor$ rows. For example, on 100×100 matrix, $P = 8$, and $S = 6$, the first approach results in 17 super-rows, six rows each. However, the second approach results in 16 super-rows; four super-rows with seven rows each, and the remaining 12 super-rows with six rows each. Both approaches are implemented and evaluated, and the performance of one is close to the other. On P parallel processors, these N super-rows are portioned into $2P$ blocks. Each processor performs computations on a pair of blocks. Not all block sizes are necessarily have the same number of super-rows; some blocks have $\lfloor N/2P \rfloor$ super-rows while the others have $\lceil N/2P \rceil$ super-rows.

In the proposed HBJ algorithm, the computation of SVD is done in two main parts. In the first part, which is the beginning of a sweep, each processor orthogonalizes the rows of a block. As shown in Listing 1 lines 3 to 10, the i^{th} processor ($1 \le i \le P$) fetches the super-rows of the $(2i\text{-}1)^{th}$ block sequentially and orthogonalizes the rows of a super-row with each other exactly once. The idea of Steam Hestenes SVD algorithm [15] is used to calculate the rotation parameters of a super-row (set of rows) at once and then apply all of them as follows. All rotations parameters of the j^{th} super-row are computed and stored in arrays c[] and s[] by calling `CalAll(SupRow[j],c[],s[])` routine. Then, all rotation parameters (c[] and s[]) are applied to the corresponding super-row at once by calling `ApplyAll(SupRow[j],c[],s[])` routine (see Listing 1 lines 4 to 6). Note that the super-rows assigned to the b^{th} block can be accessed using a data structure called *Block[b]*. The indices of the first and last super-rows of the b^{th} block are stored in *Block[b].low* and *Block[b].high* respectively. In short, the j^{th} super-row of b^{th} block means $Block[b].low \le j \le Block[b].high$. The same is done to access the rows of a super-row j; *SupRow[j].low* and *SupRow[j].high* are used to get the indices of the first and last rows of the j^{th} super-row, respectively.

After performing the rotations within each super-row of a block, the rotation parameters are computed and then applied between super-rows of the same block by calling `CalAll(SupRow[],SupRow[],c[],s[])` and `ApplyAll(SupRow[], SupRow[],c[],s[])` routines, respectively, as shown in Listing 1 lines 7 to 10. Each

row in one super-row must be orthogonalized with each row in the other super-row once but no rows in the same super-row are orthogonalized. The same is done for $(2i)^{th}$ block by i^{th} processor (again, each processor works on a pair of blocks).

The second part of the proposed scheme iterates ($2P$ - 1) steps (see Listing 1 lines 11 to 17). $P(2P - 1)$ block pairs can be generated in ($2P$ - 1) steps on P parallel processors. The well-kwon round-robin algorithm is used to generate ($2P$ - 1) steps, P block pairs each, required for implementing SVD on P parallel processors. Figure 3 shows that the generation of seven steps on four parallel processors using the round-robin method. In the computations of each step, each super-row in one block must be orthogonalized with each super-row in the other block once but no super-rows in the same block are orthogonalized. This is done by calling `CalAll(SupRow[],SupRow[],c[],s[])` and `ApplyAll(SupRow[],SupRow[],c[],s[])` routines. The round-robin ordering subroutine (see Figure 3) is applied to get the next step, and so forth.

Note that in step 0 of Listing 1, the Frobenius norm of the input matrix is calculated, which remains unchanged under orthogonal transformations. The Frobenius norm times the machine epsilon ($\varepsilon = 10^{-15}$) is used for checking the orthogonalization between two rows. Besides, the new order of the round-robin routine is stored on *up*[] and *dn*[] arrays. Initially *up*[] and *dn*[] arrays are initialized with even (2, 4, 6, …) and odd (1, 3, 5, …) numbers, respectively, as shown in Figure 3 step 1. The contents of these arrays are changed by calling the round-robin routine, as shown in Figure 3 steps 2 to 7.

Listing 1: The proposed HBJ algorithm

```
00. δ = ε ∑ A[i]^T A[i], 1 ≤ i ≤ n
01. repeat
02. converged = true
03. for s = 1 to (2*P)
04.   for i = Block[s].low to Block[s].high
05.     CalAll(SupRow[i],c[],s[])
06.     ApplyAll(SupRow[i],c[],s[])
07.   for i = Block[s].low to Block[s].high-1
08.     for j = i+1 to Block[s].high
09.       CalAll(SupRow[i],SupRow[j],c[],s[])
10.       ApplyAll(SupRow[i],SupRow[j],c[],s[])
11. for iteration = 1 to (2*P - 1)
12.   for s = 1 to P
13.     for i = Block[up[s]].low to Block[up[s]].high
14.       for j = Block[dn[s]].low to Block[dn[s]].high
15.         CalAll(SupRow[i],SupRow[j],c[],s[])
16.         ApplyAll(SupRow[i],SupRow[j],c[],s[])
17.   Round-Robin(up[], dn[])
18. until converged = true
19. for i = 1 to n
20. σ[i] = sqrt(A[i]^T A[i]
```

```
CalAll(SupRow[r],c[],s[])
  index = 0
 for i = SupRow[r].low to SupRow[r].high-1
   for j = i+1 to SupRow[k].high
    dii = A[i]^TA[i] , djj = A[j]^TA[j] , dij = A[i]^TA[j]
    if |dij| > δ then converged = false
    if dij ≠ 0 then
     τ = (djj - dii)/(2 * dij) , t = sgn(τ)/(|τ| + sqrt(τ^2 + 1)
     c[index] = 1 / sqrt(t^2 + 1)  , s[index] = t * c[index]
    else
     c[index] = 1 , s[index] = 0
    index = index + 1
ApplyAll(SupRow[r],c[],s[])
  index = 0
  for i = SupRow[r].low to SupRow[r].high-1
   for j = i+1 to SupRow[k].high
    for k = 1 to n
     temp = c[index] * A[i][k] - s[index] * A[j][k]
     A[j][k] = s[index] * A[i][k] + c[index] * A[j][k]
     A[i][k] = temp
    index = index + 1
CalAll(SupRow[r],SupRow[q],c[],s[])
  index = 0
  for i = SupRow[r].low to SupRow[r].high-1
   for j = SupRow[q].low to SupRow[q].high-1
    dii = A[i]TA[i] , djj = A[j]TA[j] , dij = A[i]TA[j]
    if |dij| > δ then converged = false
    if dij ≠ 0 then
     τ = (djj - dii)/(2 * dij), t = sgn(τ)/(|τ| + sqrt(τ^2 + 1))
     c[index] = 1/ sqrt(t^2 + 1)  , s[index] = t * c[index]
    else
     c[index] = 1 , s[index] = 0
    index = index + 1
ApplyAll(SupRow[r],SupRow[q],c[],s[])
  index = 0
  for i = SupRow[r].low to SupRow[r].high-1
   for j = SupRow[q].low to SupRow[q].high-1
    for k = 1 to n
     temp  = c[index] * A[i][k] - s[index] * A[j][k]
     A[j][k] = s[index] * A[i][k] + c[index] * A[j][k]
     A[i][k] = temp
    index = index + 1
```

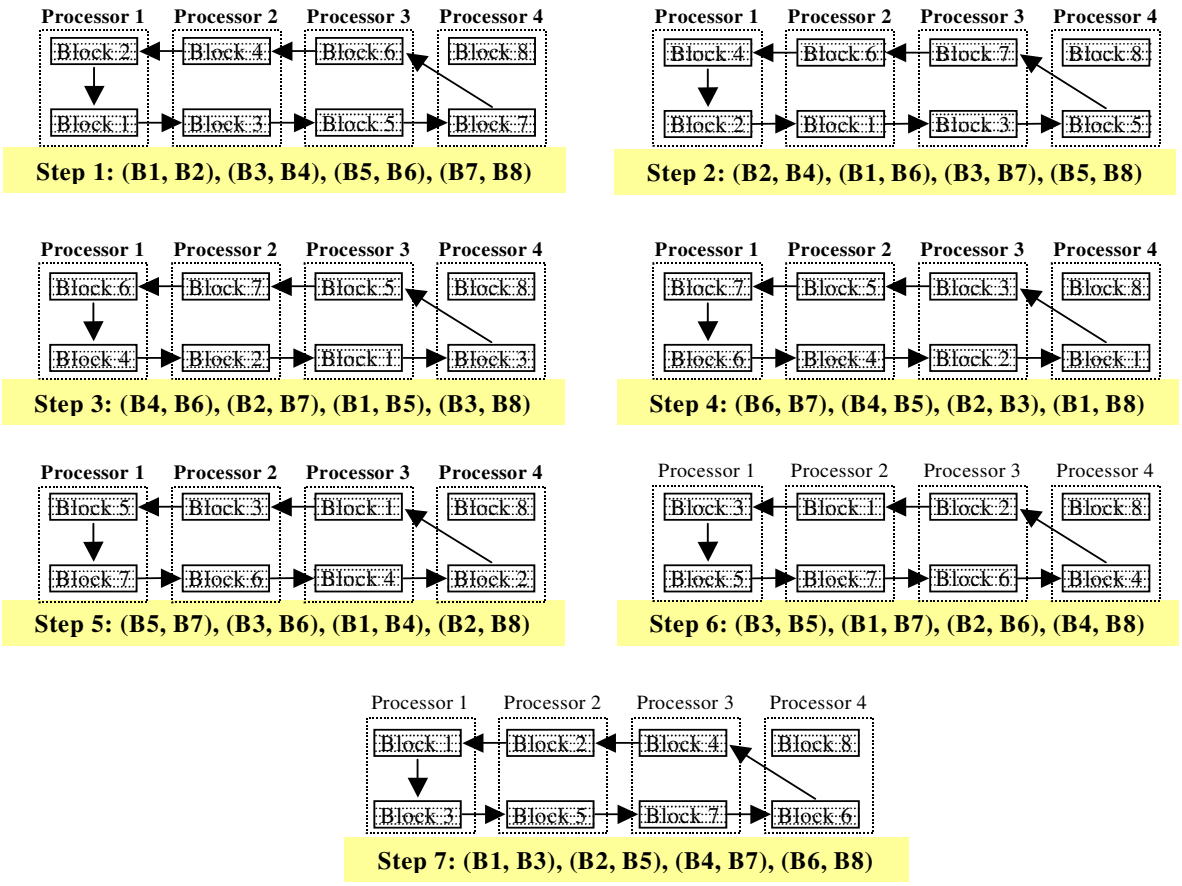

Figure 3: Round-robin method for generating various orders

4. PERFORMANCE EVALUATION OF THE PROPOSED HBJ ALGORITHM ON MULTI-CORE PROCESSORS

Multi-core technology is a form of hardware multi-threading capability in Intel 64 and IA-32 processor families [6]. Multi-core technology enhances hardware multi-threading capability by providing two or more cores in a physical package. Each core may support Hyper-Threading (HT) technology. A core that supports HT technology consists of two or more logical processors (see Figure 4). HT technology leverages the thread-level parallelism found in high-performance applications by allowing two or more threads to be executed simultaneously on each core (i.e., each thread is executed on a logical processor) [7, 22]. Each logical processor executes instructions from an application thread using shared resources in the processor core. However, each logical processor has its own architectural state (data registers, segment registers, control registers, etc.). Moreover, each logical processor has its own advanced programmable interrupt controller (APIC), which provides interrupt handling.

The dual-core Intel Xeon processor features multi-core, Hyper-Threading technology and supports multi-processor platforms. It provides four logical processors in a physical package (two logical processors for each processor core) based on the Intel Core microarchitecture [6]. As shown in Figure 4, the two cores on a Xeon processor share a smart second level cache, which enables efficient data sharing between two cores to reduce memory traffic bus.

Beside the use of parallel processing on a multi-core Intel Xeon processor, multiple data can be processed using a single instruction (SIMD) on each Xeon core [7, 23]. This further improves the performance of many data-parallel applications. Four packed single-precision (32-bit) or two packed double-precision (64-bit) floating-point data elements

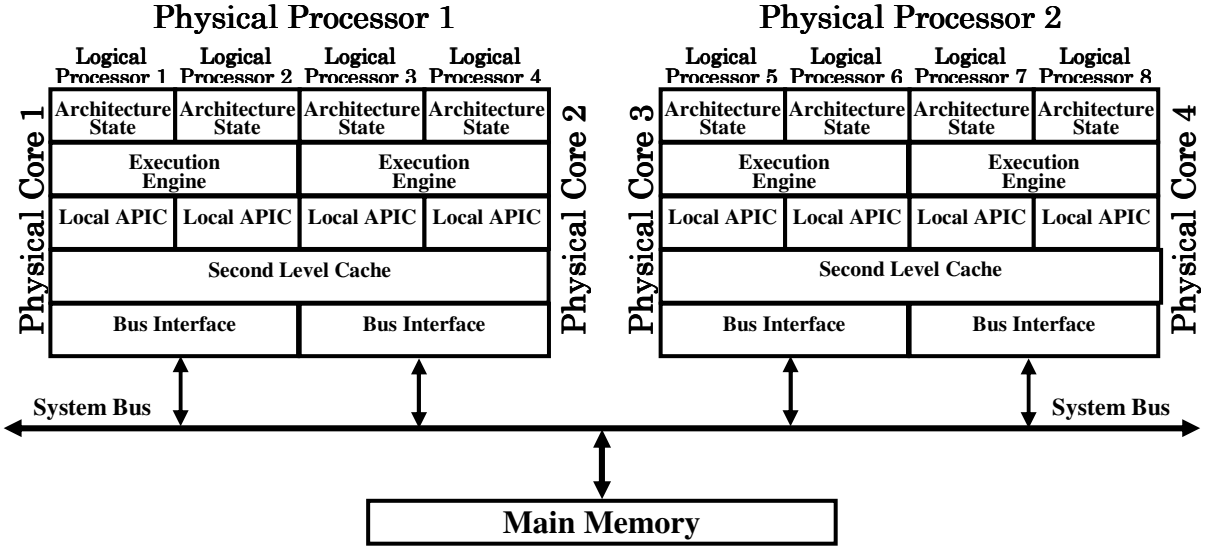

Figure 4: The target system, two dual-core Intel Xeon Processors with HT technology

can be processed using a single instruction. Moreover, Intel Xeon processors support 8/16/32/64/128-bit integer data types. The operands can be in memory or in a set of sixteen 128-bit XMM registers.

This section presents the performance evaluation of the proposed HBJ algorithm for parallel computing SVD on multi-core Intel Xeon processors. Figure 4 shows the block diagram of the target system, which has two dual-core Xeon processors with HT technology (i.e., the target system has eight logical processors). HBJ algorithm is implemented and evaluated on matrices with sizes vary form 100×100 up to 2000×2000 in step of 100. The content of these matrices are generated randomly to have a value in the interval [1, 10]. Besides, the norm of the input matrix times 10^{-15} is used as convergence condition.

Figure 5 shows the number of HBJ sweeps required for convergence, where SR*x* means that *x* rows are in a super-row. It is clear that the number of sweeps required for HBJ algorithm is very close to that needed for the cyclic one-sided Jacobi algorithm. To be specific, the average number of sweeps needed for cyclic one-sided Jacobi and HBJ algorithms are 14 and 15, respectively. This represents one of the main key points of the proposed HBJ algorithm because of its effect on the total execution time.

The execution time in seconds of the parallel implementation of HBJ algorithm on

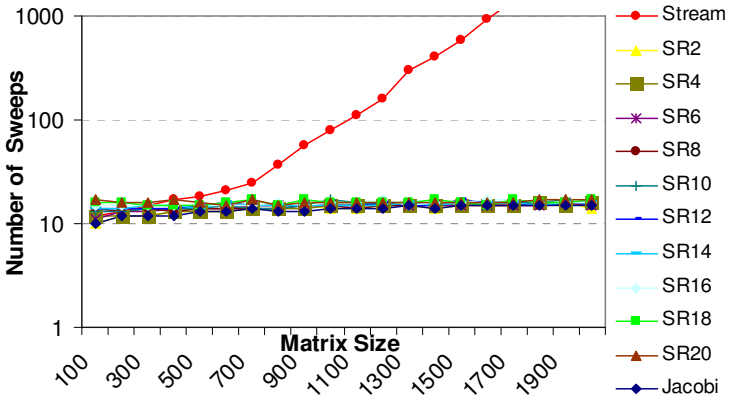

Figure 5: Number of sweeps for HBJ algorithm

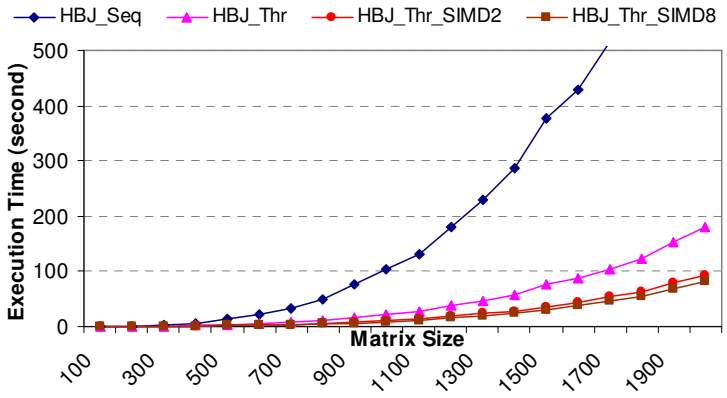

Figure 6: The performance of HBJ algorithm

eight logical processors is shown in Figure 6. The size of a super-row is selected to be eight rows. The curve "HBJ_Seq" represents the execution time of the sequential implementation of the HBJ algorithm on the target system. Around 3.5 clock cycles are needed to perform a double-precession floating-point operation (DP FLOP). The number of clock cycles per FLOP would be reduced by parallel processing threads of HBJ code on multi-core processors using multi-threading techniques [24]. The curve "HBJ_Thr" represents the execution time of the parallel implementation of HBJ algorithm on eight logical processors (four execution cores with HT technology) using eight threads. A speedup of around five is achieved due to the use of multi-threading technique (see Figure 7). Note that the ideal speedup on logical processors differs from that on multi-processor systems, which use a single physical processor in a single chip package [7]. Thus, the ideal speedup due to multi-threading on the target system is less than eight because the number of execution engines is only four (see Figure 4). However, the ideal speedup should be greater than four because each Xeon core supports HT technology. Intel HT technology improves the performance of many applications by around 30% [24]. In total, the ideal speedup of the use of multi-threading technique on four cores with HT technology is 5.2. On a reasonable matrix size, around 95% of the ideal speedup is achieved on the target system using multi-threaded HBJ algorithm.

The execution time of HBJ algorithm on Intel processors can be further reduced using SIMD instruction sets [23]. The curve "HBJ_Thr_SIMD2" in Figure 6 shows the parallel performance of HBJ algorithm using multi-threading and SIMD techniques. Two double-precision floating-point elements are processed using a single SIMD instruction. It is known that the ideal speedup of using Intel SIMD instruction on double-precision numbers is two. Our result shows an average speedup of 1.9 (see Figure 7) due to the use of double-precision Intel SIMD instructions on the HBJ algorithm.

To further improve the performance of the proposed HBJ algorithm, loop unrolling technique is used to reduce the number of branch instruction and to keep the pipeline full for a long time [23]. The curve "HBJ_Thr_SIMD8" in Figure 6 shows the effect of

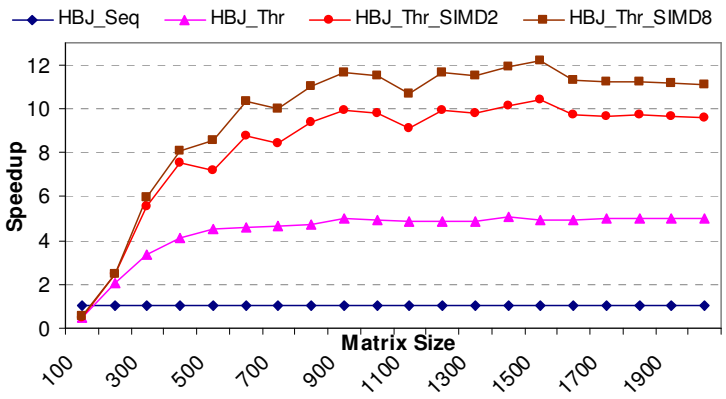

Figure 7: The speedup of HBJ over the sequential

unrolling loops by a factor of four on the performance of the HBJ algorithm. The speedup of the unrolled, multi-threaded, SIMD HBJ algorithm on multi-core processors is shown in Figure 7. The proposed HBJ algorithm speeds up the parallel computations of SVD by around 11 times higher than the sequential implementation on the same hardware. Figure 7 shows that HBJ is a highly parallel algorithm, accelerates the computations of SVDs on Intel multi-core processors using multi-threading, SIMD, and loop unrolling techniques. Moreover, Figure 8 shows the speedup of using the proposed HBJ over using cyclic one-sided Jacobi algorithm. The parallel implementation of the HBJ algorithm gives a speedup of around eight times higher than the sequential implementation of the cyclic one-sided Jacobi algorithm. Note that the performance of multi-threaded HBJ algorithm is not good on a small matrix size because of the overheads of creating threads (see [22, 24] for more detail). A performance of around 10 GFLOPS (double-precision) can be achieved on four Intel Xeon cores with HT technology, using eight parallel threads as shown in Figure 9.

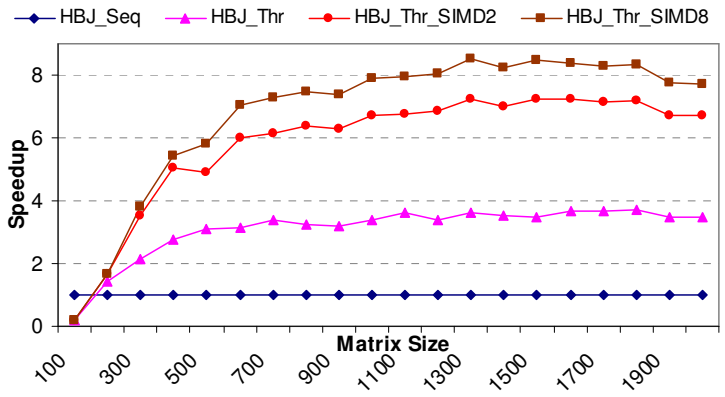

Figure 8: The speedup of HBJ over cyclic Jacobi

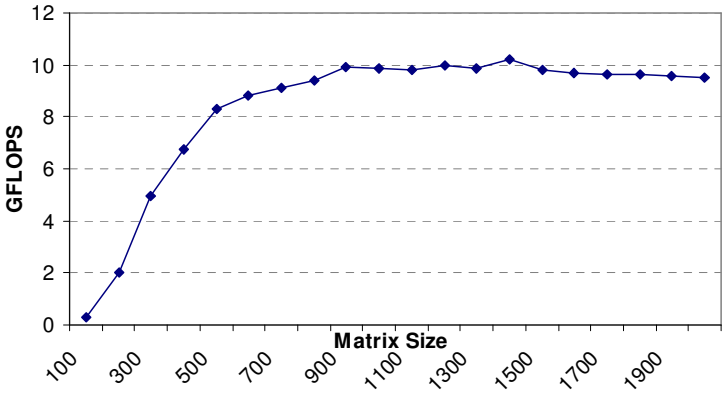

Figure 9: GFLOPS performance of the proposed HBJ algorithm

5. CONCLUSION

In this paper the well-known one-sided Jacobi for computing SVD is restructured to be implemented efficiently on multi-level memory hierarchy architectures. An efficient algorithm is proposed to reduce the parallel execution time of computing SVDs by exploiting memory hierarchy and reducing the number of sweeps required for convergence. The given matrix is partitioned hierarchy into super-rows; each super-row consisting of a set of consecutive rows. On P parallel processors, these super-rows are partitioned into $2P$ blocks to be processed in parallel.

One key point of the proposed scheme is the highly exploitation of memory hierarchy by performing all computations on super-rows (matrix operations) instead on rows (vector operation). Our results show that the performance of the proposed algorithm does not degrade as the input matrix becomes large. Another key point is that the number of sweeps required by the proposed scheme very close to cyclic one-sided Jacobi. This key point helps to reduce the total execution time of the proposed algorithm. Our results show the speedup of the proposed algorithm over cyclic one-sided Jacobi is around eight on four Xeon cores with HT technology. Third key point of the proposed algorithm is that the number of sweeps required for convergence does not depend drastically on the size of the input matrix. This means that the proposed algorithm is suitable for implementation on small as well as large number of parallel processors.

On two dual-core Intel Xeon processors, our results show that the performance of parallel implementation of the proposed scheme is around 11 times higher than the sequential implementation on the same hardware. Moreover, a performance of around 10 GFLOPS (double-precision) can be achieved on four executions cores supporting Hyper-Threading technology, using eight parallel threads.

In conclusion, the proposed algorithm is highly parallel algorithm for computing SVD based on cyclic one-sided Jacobi method. A good performance of the proposed algorithm on multi-core processors like Intel Xeon processors can be achieved by exploiting the multi-core hardware, memory hierarchy, and Intel SIMD instruction set.

REFERENCES

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, 3rd Edition, 2003, ISBN 1558605967.

[2] J. Crummey, D. Whalley, and K. Kennedy, "Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings," *International Journal of Parallel Programming*, Vol. 29, No. 3, 2001, pp. 217-247.

[3] J. Demmel, J. Dongarra, J. Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, "Prospectus for the Development of a Linear Algebra Library for High-Performance Computers," *Argonne National Laboratory Report, ANL-MCS-TM-97*, Mathematics and Computer Science Division, September 1987.

[4] O. Brewer, J. Dongarra, and D. Sorensen, "Tools to Aid in the Analysis of Memory Access Patterns for FORTRAN Programs," *Parallel Computing*, Vol. 9, No. 1, December 1988, pp. 25-35.

[5] J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, L. Torczon, and W. Gropp, *The Sourcebook of Parallel Computing*, Morgan Kaufmann, November 2002, ISBN 1558608710.

[6] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 1: Basic Architecture, http://www.intel.com/products/processor/manuals/index.htm.

[7] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, http://www.intel.com/products/processor/manuals/index.htm

[8] G. Golub and F. Luk "Singular Value Decomposition: Applications and Computations," *ARO Rep. 77-1, Trans. 22nd Conf. Army Mathemattctans,* 1977, pp. 577-605.

[9] V. Klema and A. Laub, "The Singular Value Decomposition: Its Computation and Some Applications," *IEEE Transactions on Automatic Control*, Vol. AC-25, No. 2, 1980.

[10] G. Golub and C. Van Loan, *Matrix Computations*. John Hopkins University Press, Baltimore and London, 2nd edition, 1993.

[11] G. Golub and W. Kahan, "Calculating the Singular Values and Pseudo-Inverse of a Matrix," J SIAM Ser. B. Numer Anal 2, 1965, pp. 205-224.

[12] R. Brent, and F. Luk, "The Solution of Singular-Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays," *SIAM Journal on Scientific and Statistical Computing*, Vol 6, No.1, 1985, pp. 69-84.

[13] B. Zhou and R. Brent, "Parallel Computation of the Singular Value Decomposition on Tree Architectures," *Proc. 22nd International Conference of Parallel Processing (ICPP)*, CRC Press, Ann Arbor, 1993, Vol. 3, pp. 128-131.

[14] B. Zhou and R. Brent, "On Parallel Implementation of the One-Sided Jacobi Algorithm for Singular Value Decompositions," *Proc. Euromicro Workshop on Parallel and Distributed Processing*, San Remo, Italy, IEEE CS Press, 1995, pp. 401-408.

[15] V. Strumpen, H. Hoffmann, and A. Agarwal, "A Stream Algorithm for the SVD," *Technical Memo 641*, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, October 2003.

[16] S. Rajasekaran and M. Song, "A Novel Scheme for the Parallel Computation of SVDs," *Proc. High Performance Computing and Communications, 2006*, pp.129-137

[17] M. Hestenes, "Inversion of Matrices by Biorthogonalization and Related Results," *Journal of the Society for Industrial and Applied Mathematics*, Vol. 6, Issue 1, 1958, pp. 51-90.

[18] F. Luk, "Computing the Singular-Value Decomposition on the ILLIAC IV," *ACM Trans. Math. Softw*., 6, 1980, pp. 524-539.

[19] J. Demmel and K. Veselic, "Jacobi's Method is More Accurate than QR," *SIAM J. Matrix Anal. Appl*, Vol. 13, 1992, pp. 1204-1246.

[20] R. Brent, "Parallel Algorithms for Digital Signal," *Proc. Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, Springer-Verlag, 1991, pp. 93-110.

[21] R. Schreiber, "Solving Eigenvalue and Singular Value Problems on an Undersized Systolic Array," *SIAM. J. Sci. Statist. Comput*. Vol. 7, 1986, pp. 441-451.

[22] S. Akhter and J. Roberts, *Multi-Core Programming: Increasing Performance through Software Multithreading*, Intel PRESS, 2006, ISBN 0976483246.

[23] R. Gerber, A. Bik, K. Smith and X. Tian, *The Software Optimization Cookbook: High-Performance Recipes for IA-32 Platforms*, Second Edition, Intel PRESS, 2006, ISBN 0976483211

[24] A. Binstock and R. Gerber, *Programming with Hyper-Threading Technology: How to Write Multithreaded Software For Intel(r) IA-32 Processors*, Intel PRESS 2003, ISBN 0970284691.