Bisection and Twisted Bidiagonal SVD on GPU for Big Matrices

Lu He[1], Yan Luo[1], Rui Liu[1], Hengyong Yu[1], Yu Cao[1], Xuzhou Chen[3] and Seung Woo Son[1]

[1] University of Massachusetts Lowell, Lowell MA 01854, USA
[2] Fitchburg State University, Fitchburg MA 01420, USA

Abstract. Singular value decomposition (SVD) is one of the most important factorizations in matrix computation. However, computing the singular values and their vectors is still time-consuming, especially when the dimension of matrices exceeds tens of thousands. In this paper, we present a high performance approach called "Bisection and Twisted" (BT) for solving bidiagonal SVD. Based on the bidiagonal SVD method and with a complexity of $O(n^2)$, the BT algorithm has three salient features: (1) it is shown to have weak data dependency, therefore can solve large SVD problems by splitting them into smaller ones; (2) it can compute any subsets of singular values and corresponding vectors directly with computational complexity of $O(kn)$, making it suitable for applications that do not require the whole SVD calculation; and (3) it needs only $O(kn)$ memory space for temporary variables, thus can work on memory-constrained platforms. As modern general purpose GPUs have shown their extreme computational advantages in parallel computing, we implement the BT algorithm on single and multiple GPUs. With our carefully designed GPU kernels, the BT algorithm takes only 0.8 second to obtain all the singular values and vectors on a GeForce 750Ti for a single-precision matrix size of $10k$ by $10k$, and 1.5 seconds on Tesla K40c for a double-precision matrix of the same size. This is about 10 times faster than MKL divide-and-conquer routine DBDSDC on 8 general purpose cores, and 36 times faster than CULA QR routine DBDSQR on the same GPUs. Additionally, the BT algorithm is able to compute SVD for matrices of size 1 million by 1 million with only two GPUs. To the best of our knowledge, no implementation has achieved such a scale.

1 Introduction

There is rapidly growing interest in singular value decomposition (SVD) in various fields, including noise reduction in signal processing, low-rank approximations in linear algebra, objects classification in computer vision, latent semantic indexing in information retrieval, supervised and unsupervised algorithms in machine learning, and data compression in information theory. Yet, most algorithms for computing SVD are designed to solve problems with only small-scale data in acceptable time and space requirements due to their algorithmic complexities.

With the advent of the "big data" era, these algorithms are no longer adequate in terms of computational complexity and required memory space.

A typical SVD algorithm can be broken down into two steps[3]. The first step is to reduce the initial matrix to bidiagonal form using Householder transformation. The second step is to diagonalize the resulting matrix using bidiagonal SVD algorithms. Most of the literature focuses on the second step as algorithms in second step typically use iterative approaches[14–16] and thus take majority of the computation time depending on the accuracy requirement.

There are many algorithms for solving the bidiagonal SVD. QR iteration is regarded as a powerful and effective approach for finding all the singular values. Due to its complexity of $O(n^3)$, QR's execution time increases rapidly as the size of a matrix increases. Jacobi algorithm is the one with the highest accuracy in practice[1]. However, its $O(n^3)$ complexity with a big constant causes the algorithm to be much slower than other algorithms, making the iteration times much larger than those of QR algorithm.

Divide-and-conquer (DC) is assumed to be the fastest algorithm for finding singular values for large matrices[18]. It takes $O(n^{2.3})$ flops on average[1]. In the worst case, it may require up to $O(n^3)$. But the major drawback of DC is the relatively low accuracy of the singular values when merging, let alone singular vectors. In summary, the prior works on SVD computations are either time-consuming or inaccurate.

In addition, all the algorithms discussed above have two common disadvantages: (1) heavy data dependence makes SVD algorithm not suitable for parallelization and extension for other architectures; and (2) large memory space required for temporary variables dramatically limits the capability for computing singular values for very large matrices.

Many SVD applications, such as principal component analysis (PCA), need only a small subset of the singular values and vectors. However, the aforementioned algorithms lack the flexibility of calculating the subsets directly. Bisection and inverse (BI) iteration method could calculate the subset of the singular values and vectors[2] as well as the complete SVD. In this method, bisection is responsible for obtaining singular values, while inverse iteration is responsible for singular vectors with obtained singular values. BI takes $O(kn)$ to find k singular values and singular vectors. However, the inverse iteration does not guarantee the accuracy and orthogonality of the computed singular vectors in the case of clustered singular values.

In this paper, we present a new SVD approach called "Bisection and Twisted" (BT) algorithm. We use the twisted algorithm [5] to replace the inverse iteration in BI, because the twisted algorithm is able to calculate accurate and orthogonal singular vectors. Comparing to other algorithms, BT approach only requires $O(n^2)$ to complete the whole SVD[5, 6], and $O(kn)$ to calculate k singular values and corresponding vectors. Most importantly, there are three salient features that make BT algorithm attractive to obtaining SVD of large matrices. First, the data dependency is weak in the BT algorithm because it does not need to synchronize intermediate results for the following calculation (as in other

algorithms), making it an excellent candidate for taking advantage of the parallel computing elements (e.g., multicore CPUs or GPUs). Second, the algorithm can obtain a subset of k singular values and its corresponding vectors in $O(kn)$ time. This is particularly useful for applications that do not require a complete SVD. Third, the algorithm needs only $O(kn)$ memory space to store temporary variables, which is important for extending to large scale matrices in big data applications on memory constrained platforms.

We then design GPU kernels to implement the BT algorithm on GPU platforms. We evaluate its performance on a variety of GPUs to study its scalability. We also design a multi-GPU version of BT to demonstrate the effectiveness of weak data dependency and scale it to compute the SVD of very large matrices. We are able to solve SVD of a matrix of 1 million by 1 million using only two GPUs.

In this paper, we make the following contributions:

1. We propose a novel SVD algorithm called Bisection and Twisted for fast SVD computation. It requires $O(n^2)$ time to solve the complete SVD, and $O(kn)$ for calculating a subset of k singular vectors.
2. We show that the data dependency of BT algorithm is weak. As a result, BT algorithm is highly suitable for massively parallel computing architecture on which we can effectively partition a large problem into smaller ones.
3. We not only implement the BT algorithm on different GPUs, but also design a multi-GPU version that can scale well with the matrix size. To the best of our knowledge, we are the first to achieve SVD on a one million by one million matrix using just two GPUs.
4. We perform in-depth analysis on the GPU kernels for singular vector calculation, and present multiple optimization methods to further improve the SVD performance on GPUs.

The rest of the paper is organized as follows. The Bisection and Twisted algorithm is given in Section 2. Section 3 describes the implementation of the BT algorithm on GPUs, as well as the GPU specific optimizations. Section 4 presents the experimental results and profiling analysis of GPU kernels. Section 5 discusses the related work. Conclusion and future work are in Section 6.

2 Bisection and Twisted Algorithm

For an arbitrary matrix $A \in R^{m \times n} (m > n)$ all SVD algorithms discussed in section 1 consist of two steps: The first step is to reduce the initial matrix A to bidiagonal form using Householder transformation. The second step is to reduce the bidiagonal matrix into a diagonal matrix.

By applying the Householder transformation, we can decompose the matrix A as follows:

$$A = QBP^T, \tag{1}$$

where $B \in R^{m \times n}$ is a bidiagonal matrix of the following form:

$$B = \begin{pmatrix} B_1 \\ 0 \end{pmatrix} \tag{2}$$

Here $B_1 \in R^{n \times n}$ is a bidiagonal matrix. Since

$$B^T B = \begin{pmatrix} B_1^T & 0 \end{pmatrix} \begin{pmatrix} B_1 \\ 0 \end{pmatrix} = B_1^T B_1,$$

we assume, without loss of generality, that B is a $n \times n$ bidiagonal matrix in this paper.

The first step reduces the time complexity of QR, Jacobi and bisection and inverse algorithms from $O(n^4)$ to $O(n^3)$. However, Householder transform has only one iteration, which takes a small portion of the time spending on solving the SVD. Thus, we focus only on the second step that reduces a bidiagonal matrix to a diagonal matrix, for which the "bisection and twisted" algorithm is designed.

The bisection and twisted algorithm is divided into two phases: (1) obtain the singular values of the bidiagonal matrix by using bisection approach; and (2) obtain the corresponding left and right singular vectors of every singular value by using twisted factorization. Next we illustrate these two phases in the following subsections.

2.1 Bisection Algorithm

Bisection algorithm is widely used in many applications to find the roots of a sophisticated equation that cannot be solved directly. It repeatedly bisects an interval and then selects a subinterval in which a root lies for further processing until convergence. Bisection is an approximation algorithm for solving sophisticated equation and can reach certain relative accuracy solution.

Suppose B is an upper bidiagonal $n \times n$ matrix with elements $b_{i,j}$ reduced by Householder transform [22]. The matrix $T = B^T B - \mu^2 I_n$ can be decomposed as

$$T = B^T B - \mu^2 I_n = LDL^T, \tag{3}$$

where μ is a shift variable, D is $diag(d_1, \cdots, d_n)$, and

$$L = \begin{pmatrix} 1 & & & \\ l_1 & 1 & & 0 \\ & l_2 & \ddots & \\ & 0 & \ddots & \ddots \\ & & & l_{n-1} \ 1 \end{pmatrix}. \tag{4}$$

By substituting L and D into Eq. (3), we can obtain the following equations

$$\begin{cases} b_{1,1}^2 - \mu^2 = d_1 \\ b_{k-1,k-1} b_{k-1,k} = d_{k-1} l_{k-1} \\ b_{k-1,k}^2 + b_{k,k}^2 - \mu^2 = l_{k-1}^2 d_{k-1} + d_k \end{cases} \tag{5}$$

where $k = 2, 3, \cdots, n$. Next, we define an auxiliary values t_k as

$$t_k = t_{k-1} * (b_{k-1,k}^2/d) - \mu^2, \tag{6}$$

and then all the equations in Eq. (5) reduce to

$$\begin{cases} d_1 = b_{1,1}^2 - \mu^2 \\ d_k = b_{k,k}^2 + t_k \end{cases} \tag{7}$$

It is clear that matrices D and T are two congruent symmetric matrices. According to the Sylvester's law of inertia, both matrices D and T have the same numbers of positive, negative, and zero eigenvalues. We define a function denoted by $NegCount(\mu)$ to count the number of negative eigenvalues of matrix T. Algorithm 1 describes how $NegCount$ function works. Lines 3-7 calculate the negative number of singular values that are less than μ. Since matrices B and B^T have the same singular values, $NegCount(\mu)$ is the number of the singular values of B which are less than μ. It is clear that if the floating point arithmetic is monotonic, then $NegCount(x)$ is a monotonically non-decreasing function of x[2].

Algorithm 1 NegCount in Bisection Algorithm

```
1: procedure NegCount(n, B, μ)
2:     d = 1, t = 0, cnt = 0, b_{0,1} = 0;
3:     for k = 1 → n do
4:         t = t * (b_{k-1,k}^2/d) − μ^2;
5:         d = b_{k,k}^2 + t;
6:         if d < 0 then
7:             cnt++;
8:     return cnt;
```

Algorithm 2 is the bisection algorithm for calculating the singular values in interval $[l, u)$. Based on the definition of $NegCount$ the total number of singular values in $[l, u)$ is $n_u - n_l = NegCount(u) - NegCount(l)$. Lines 5-17 can be parallelized in bisection algorithm.

Weak Data Dependency of Bisection Algorithm

The efficiency of our algorithm is based on the assumption that the data dependencies for both separating and calculating phases are weak so that we could divide the big problem into smaller but loosely coupled components. Since the singular values are all positive, we assume that the boundary containing all the singular values is $[0, ug)$, where the upper bound ug can be calculated by Gershgorin circle theorem[19].

Algorithm 2 Bisection Algorithm

```
1: procedure Bisection(val, n, B, l, u, n_l, n_u, τ)
2:     if n_l ≥ n_r or l > u then
3:         No singular values in [l, u);
4:     Enqueue (l, u, n_l, n_u) to Worklist;
5:     while Worklist is not empty do
6:         Dequeue (a, b, n_a, n_b) from Worklist;
7:         if b − a < τ then
8:             for i = n_a + 1 → n_b do
9:                 val[i] = min(max((a + b)/2, a), b);
10:        else
11:            m = MidPoint(a, b);
12:            NegCount(n_m, B, m);
13:            n_m = min(max(n_m, n_a), n_b);
14:            if n_m > m_a then
15:                Enqueue (a, m, n_a, n_m) to Worklist;
16:            if n_m < m_b then
17:                Enqueue (m, b, n_m, n_b) to Worklist;
```

After obtaining the boundary, we divide the interval $[0, ug)$ into p subintervals, and each subinterval $[l_i, u_i)$ can be calculated as follows:

$$l_i = (ug/p) * (i-1) \tag{8}$$

$$u_i = (ug/p) * i, \tag{9}$$

where $i = 1, 2, \cdots, p$. Then we can calculate the numbers of singular values n_{l_i} and n_{u_i} that are less than l_i and u_i as follows, respectively:

$$n_{l_i} = NegCount(l_i) \tag{10}$$

$$n_{u_i} = NegCount(u_i). \tag{11}$$

It is easily seen that the number of singular values in interval $[l_i, u_i)$ is $n_{u_i} - n_{l_i}$ and the indices of these singular values are $n_l, n_l+1, ..., n_u-1$. We thus illustrate that the data dependence is weak when dividing the interval to subintervals.

Now, we show that the data dependency in bisection algorithm is also weak in the calculation phase (Algorithm 2). Suppose $[l, u)$ is an subinterval containing several singular values of a matrix. n_l and n_u are the numbers of singular values that are less than l and u, respectively. How is the bisection algorithm able to calculate an arbitrary singular value v with index (n_l+k) in an subinterval $[l, u)$? After several bisection iterations, the subinterval $[l, u)$ reduces to $[v_1, v_1 + \tau)$, whose $NegCount$ are $(n_l + k - 1)$ and $(n_l + k)$, respectively. Since v is in the subinterval $[v_1, v_1 + \tau)$, we know that $|v_1 - v| < \tau$, and if τ is small enough, we stop the iteration and take v_1 as calculated singular value with index $(n_l + k)$. The whole procedure of bisection algorithm requires only $NegCount$ function without the information of other singular values.

Error Analysis

In theory, the bisection algorithm is bound to the upper limit of error between actual value and calculating value, which can be set according to particular applications. The selection of an error tolerance τ greatly impacts the execution time of the algorithm. Large error tolerance will allow us to solve the problem quick but result in low accuracy. On the other hand, too small value for τ may fail to reach the required accuracy. Based on our tests, we can obtain the singular values with error tolerance less than 10^{-18}.

2.2 Twisted Algorithm

The second phase of our algorithm, twisted algorithm, calculates the corresponding left and right singular vectors. It has advantages over existing approaches such as the power method[20], which can only find the largest singular value and the corresponding singular vector, and inverse iteration[21], which does not guarantee the accuracy and orthognality of the singular vectors when the singular values are clustered. In contrast, the twisted algorithm [5] can calculate orthogonal singular vectors with adequate accuracy.

Suppose λ is a singular value of bidiagonal matrix $B \in R^{n\times n}$. Then the matrix $B^TB - \lambda^2 I$ can be decomposed as

$$B^TB - \lambda^2 I_n = LD_LL^T = UD_UU^T, \tag{12}$$

where $D_L = diag(\alpha_1, \cdots, \alpha_n)$, $D_U = diag(\beta_1, \cdots, \beta_n)$. Matrix L is the same form as in Eq. (4)

$$U = \begin{pmatrix} 1 & u_1 & & & \\ & 1 & u_2 & 0 & \\ & & 1 & \ddots & \\ & 0 & & \ddots & u_{n-1} \\ & & & & 1 \end{pmatrix} \tag{13}$$

Given the LDL^T and UDU^T decomposition, the twisted factorization of the shifted matrix is shown as

$$B^TB - \lambda^2 I = N_kD_kN_k^T, \tag{14}$$

where k is the index of the minimum γ defined in Eq. (15).

$$k = \arg\min_{1\le i\le n} \gamma_i = \begin{cases} \beta_1 & \text{if } i = 1 \\ \beta_i - \alpha_i * l_{i-1} & \text{if } 2 \le i \le n-1 \\ \alpha_n & \text{if } i = n \end{cases} \tag{15}$$

D_k is diagonal matrix, N_k is the twisted matrix with form

$$N_k = \begin{pmatrix} 1 & & & & & & & \\ l_1 & \ddots & & & 0 & & & \\ & \ddots & 1 & & & & & \\ & & l_{k-2} & 1 & & & & \\ & & & \eta_k & 1 & u_k & & \\ & & & & & 1 & \ddots & \\ & & & 0 & & & \ddots & u_{n-1} \\ & & & & & & & 1 \end{pmatrix} \tag{16}$$

$$\eta_k = \begin{cases} 0 & \text{if } i = 1 \\ \frac{l_{k-1}\alpha_{k-1}}{(\alpha_{k-1} - v_{k-1}^2 \beta_{k+1})} & \text{if } 2 \leq i \leq n-1 \\ l_{n-1} & \text{if } i = n \end{cases} \tag{17}$$

Thus, the corresponding singular vector of λ can be obtained by solving the following matrix equations and then normalizing the solution:

$$N_k^T Z_k = E_k, \tag{18}$$

where each column in matrix Z_k is the non-normalization solution of singular vector corresponding to λ, and E_k is a unitary matrix.

One advantage of BT algorithm is that it can be applied to cases where singular values are clustered together. Suppose m is the multiplicity of singular values of matrix B. The algorithm selects the indices of the first m minimum values of γ. Each index k in m has a different twisted factorization, and thus the singular vector can be obtained by solving its own Eq. (18). These singular vectors are also orthogonal to each others[5]. Algorithm 3 is the algorithm to obtain the corresponding singular vectors of given singular values. Lines 4-7 are two obtain the parameter γ. Obtain the multiplicity of a singular value and their corresponding minimum values are described in Lines 8-9. Lines 10-16 are to calculate the singular vectors and normalize them. The cost for every singular vector transformation is $O(n)$, and the total cost of the transformations is $O(n^2)$.

3 Implementation and Optimization on GPU

In this section, we describe the implementation of bisection and twisted algorithm on a GPU. A GPU with CUDA architecture typically uses Single Program Multiple Threads (SPMT) model to execute its workload. A GPU thread is the fundamental working unit of a parallel program and a GPU warp consists of 32 concurrent threads. A GPU block is a group of threads sharing data via the shared memory, and the number of threads per block is typically a multiple of 32 and ranges from 256 to 1024. The number of threads per block determines the number of warps, which are assigned to GPU's cores by its internal scheduler.

Algorithm 3 Twisted Factorization

procedure Twisted(q, n, B, μ)
  ▷ l is the number of different singular values, $l \leq n$;
  for $i = 1 \rightarrow l$ **do**
    Computes matrix $S = B - \mu_i^2 I$;
    Computes LDL' decomposition $S = LD_L L'$;
    Computes UDU' decomposition $S = UD_U U'$;
    Computes γ based on Eq 15;
    Find the number m of clustered singular values μ;
    Find m-th minimum $k = min_j|\gamma(j)|$;
    for each k **do**
      $z_k = 1$, $z_{k-1} = -L_{k-1,k}$, $z_{k+1} = -U_{k,k+1}$;
      for $j = k + 2 \rightarrow n$ **do**
        $z_j = -U_{j-1,j} * z_{j-1}$
      for $j = k - 2 \rightarrow 1$ **do**
        $z_j = -L_{j+1,j} * z_{j+1}$
      Scale vector $q = z/||z||_2$;

The goal of implementing our BT algorithm is to maximize the warp execution efficiency, which is defined as the average percentage of active threads in each executed warp to take full advantage of GPU resources.

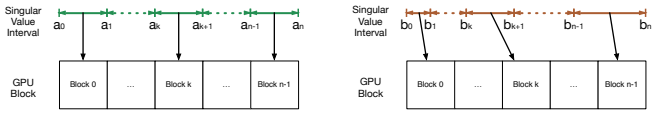
We implement our BT algorithm in two steps: we design singular value kernels in the first step (Section 3.1) and singular vector kernels in the second step (Section 3.2), and then map them onto the appropriate GPU components. We also leverage GPU-specific optimizations for handling huge matrix from big data applications.

3.1 Singular Value Kernels

To obtain the singular values, we divide the whole interval calculated with Gershgorin circle theorem into several subintervals, which can be computed in parallel. Each subinterval can then be assigned to one GPU block for calculation. Because there is a limit on the maximum number of threads for each GPU block, the number of singular values in a subinterval must not exceed that maximum number. Otherwise, the computation will have to be wrapped into two or more sequential phases to complete. Therefore how to divide the interval is critical for the performance of computing singular values. We consider two different partition strategies as described below.

The first (naïve) strategy is to divide the interval into chunks of equal length as shown in Fig.1(a). The whole interval $[a_0, a_n)$ is divided into n subintervals with the same length (i.e. same number of matrix elements). However, each such subinterval might not contain the same number of singular values. Mathematically, this strategy can be expressed as

$$a_{k+1} - a_k = a_k - a_{k-1} \tag{19}$$



(a) Equal Length Division (b) Equal Number Division

Fig. 1. Division Strategies for Partitioning Interval $[a_0, a_n)$.

However,

$$NegCount(a_{k+1}) - NegCount(a_k) = NegCount(a_k) - NegCount(a_{k-1}) \quad (20)$$

is not necessarily true. We call this strategy as *equal length division*.

Such "equal length division" strategy has issues in load balancing on parallel computing resources, as the number of singular values in each subinterval is not uniform. As a result, the SVD workload cannot be easily balanced on GPU's processing cores to achieve a satisfactory speedup because the optimal number of subintervals is dependent on multiple factors, such as the maximum number of threads, the number of threads in a GPU warp, and the size of matrix. The maximum number of threads per block determines the minimum number of subintervals that are assigned to corresponding GPU blocks. In addition, the number of threads in a GPU warp assigned to a subinterval affects the GPU utilization level.

We have conducted experiments to understand the trade-offs between subinterval length and the GPU execution efficiency. Smaller subintervals will lead to a larger number of subintervals, which cause the GPU warp execution to be less efficient. For example, our experiments show that if 512 subintervals are used, the efficiency of GPU warp execution ranges from only 12.38% to 52.15% when the matrix size increases from $1,000$ to $17,000$. That is, only half of the threads in a warp work in parallel at most. This is because (1) there are fewer singular values than the number of warps, leading the cores under-utilized; and (2) the warps are stalled due to memory accesses.

On the other hand, we can divide the whole interval into larger subintervals, making the total number of subintervals small. Then some of them may require much more processing capability because these subintervals may contain more singular values than others. In order to achieve a higher speedup, we could try to find the optimal number of subintervals, or the number of blocks, based on experimental results. However, because of the uneven number of singular values within subintervals, the speedup for equal length division remains very limited, which motivates us to search for a better partition method.

Our novel strategy is to divide the whole interval by the number of singular values, as shown in Fig.1(b). We call this approach *equal number division*. Specifically, we divide the interval $[b_0, b_n)$ into n subintervals, each of which has the same number of singular values. However, the length of a subinterval is not necessarily equal to others. The mathematical representation can be written as

$$NegCount(b_{k+1}) - NegCount(b_k) = NegCount(b_k) - NegCount(b_{k-1}) \quad (21)$$

but $b_{k+1} - b_k = b_k - b_{k-1}$ does not hold anymore. This equal number division significantly improves the load balancing on GPU cores, helping achieve a better speedup.

The convergence of bisection algorithm on a splitting point makes it possible to compute the number of singular values before actually computing the values themselves. The splitting point can be easily found by applying several $NegCount$ functions on the interval. Specifically, we design a method to find the boundary points of such subintervals, as shown in Algorithm 4. Lines 4-5 find the midpoint of interval and its corresponding $NegCount$. Lines 6-15 make use of loop to find the split point of number division by bisection algorithm.

Algorithm 4 Equal Number of Singular Value Subinterval Algorithm

```
1: procedure Number_Divide(n, B, t, τ)
2:     Obtain singular value boundary [l, u) of matrix B;
3:     Define the thread ID i, i < t;                    ▷ t is total threads
4:     mid = inside(l, u, B);
5:     n_m = NegCount(n, B, mid);
6:     while n_m ≠ (i + 1)n/t and mid − l > τ do
7:         if n_m ≥ (i + 1)n/t then
8:             u = mid;
9:             n_u = n_m;
10:        else
11:            l = mid;
12:            n_l = n_m;
13:        mid = inside(l, u, B);
14:        n_m = NegCount(n, B, mid);
15:    save the division point mid and n_m.
```

The significant advantage for the equal number division strategy is that, more singular values can be calculated with the equal number division strategy than equal length division strategy, especially when the distribution of singular values is not uniform. Our experiment results show that equal number division outperforms length division strategy. The warp execution efficiency in equal number division version can reach 92.97%-96.13%.

3.2 Singular Vector Kernel

Since the singular vectors are two $n \times n$ dense matrices, as the matrix size increases, there are significant challenges on memory storage because the size of the fast shared memory is limited in GPU architectures. It is impracticable to move all the matrix entries to the shared memory when matrix size becomes extremely large.

Based on the twisted algorithm, the required memory size M on GPU is determined by matrix size n and the size of floating number S in Equation (22).

$$M = 5 * S * n^2 \tag{22}$$

For example, the maximum matrix size with 4-bytes floating number values is about 10000 for a GPU with 2GB GPU memory. For matrices larger than that, we need to explore multiple GPUs to scale its performance as described in Section 3.4.

We initially implement the twisted algorithm with row-major matrix stored in the global memory of a GPU system. We observe the heavy usage of global memory, which is the slowest memory component on GPUs. Further analysis reveals that about 50% of global memory transfers are read-after-write. Since it is a waste of time to read from global memory after writing to global memory, we could improve the memory access performance by copying these values into the local memory and shared memory of the GPU. The way to store matrix elements (column major or row major) is also critical as the memory accesses could be coalesced if the matrix elements can be read and computed in large chunks. We present the results of such optimizations in Section 4.

3.3 Solution to Huge Matrices

In our design, the maximum number of singular values can be processed on a singular GPU is $m_b = subinterval\_size \times thread\_block\_size$, while the maximum number of singular vectors are determined by Equation (23).

$$m_t = \sqrt{U/(5 * S)} \tag{23}$$

where U is the memory size of GPU, 5 is the number of arrays required for a pair of singular vectors. Usually, m_b is much larger than m_t. For Tesla K40c with 16GB memory, $m_t = 24K$, while $m_b = 262K$. However, when the matrix size is larger than m_t, even larger than m_b, the GPU kernels cannot obtain all singular values and vectors any more with a single GPU. Therefore we derive a divide-and-conquer architecture to solve the huge matrix size as explained in this section.

When the size of a matrix is less than m_b but larger than m_t, we only need to divide the singular vectors into small sets, each of which can be processed by a single GPU. However, if the matrix size is larger than m_b, we should divide both singular values and singular vectors into smaller partitions. The singular value computations can be partitioned using m_b directly, i.e., there are $l_b = \lfloor (n/m_b) \rfloor + 1$ partitions, each of which has $\lfloor (n/l_b) \rfloor$ singular values.

The division of singular vectors should take memory size into consideration. The maximum number of partitions l_t can be derived from Equation (24) as follows:

$$l_t = \sqrt{U/(5 * n * 4)} \tag{24}$$

where n is matrix size. Thus, there should be $\lfloor (n/l_t) \rfloor + 1$ partitions with $\lfloor (n/l_t) \rfloor$ singular vectors in each partition.

3.4 Multiple GPUs on a server

We implement the multiple-GPU version with Pthread libraries on one server to control data partition and assignment to GPUs. Pthreads is a nature choice for controlling multiple GPUs on a single server because it uses shared memory architecture, dramatically reducing overheads on data sharing. We also design an extensible interface between CPU and GPU, which is easy to scale to physical distributed GPUs, compared to CUDA asynchronization interface. In our design, each thread takes control of one GPU.

When matrix size becomes huge, the load balancing among multiple GPUs will be a problem for performance as shown in column 2 of Table 3. To conquer this issue we also introduce a dynamic load-balancing method for a better speedup, where a GPU will take new tasks on unfinished subintervals as soon as it finishes its prior task.

4 Experimental Results

In this section, we evaluate the performance of our bisection and twisted algorithm and compare it with prior SVD implementations on CPUs and GPUs. We also evaluate the performance of BT on huge matrices of size more than $100K$. To our knowledge, this has not been done in any of the prior work so far. In addition, we discuss the GPU kernel profiling results to show how to further optimize our implementations.

We implement the proposed BT algorithm on three different GPUs: GeForce 750 Ti, Quadro 600 and Tesla K40c, which differ in architecture, memory size and bandwidth. Their specifications are listed in Table 1. In our implementations, we set the number of threads per GPU block as 512, which brings the better performance than other possible block sizes. For the multi-GPU version of BT, we use a mixture of these GPUs that reside in the same server to scale up the size of matrices. It is worth noting that our multi-GPU version of BT algorithm can be extended to physically distributed GPUs, i.e., GPUs on different hosts connected via a high speed network. This part is however beyond the scope of this paper.

Table 1. Specifications of Different GPUs

Specifications	GeForce 750	Quandro 600	Tesla K40
Architecture	Maxwell	Fermi	Kepler
CUDA Cores	640	96	2880
TFLOPS	1.306	0.246	4.29
GPU Clock	1268 MHz	1280 MHz	745 MHz
Mem Size	2 GB	1 GB	12 GB
Mem Bandwidth	86.4 GB/s	25.6 GB/s	288 GB/s

4.1 Comparison to Existing SVD Implementations

We generate random bidiagonal matrices with double precision numbers in the range between 0 and 1. In order to achieve high confidence on the results, we generate 10 random matrices, and for each matrix, our SVD algorithm is executed 10 times on a GPU (or two GPUs). The standard deviation of their execution time is very small, so we report the average execution time across the 100 runs as the performance results.

We compare our algorithm with CULA GPU library [11], Intel MKL library [13], Sheetal's QR implementation on S1070 [7], and Liu's DC implementation on M2070 [8]. We measure the performance of CULA on Tesla K40c, and that of Intel MKL on an 8-core 2.53GHz CPU. Until now, CULA library only has a QR based routine called culaDbdsqr. Intel MKL library has both DC routine DBDSDC and QR routine DBDSQR. Usually DC routine is faster than QR routine, so we select DBDSDC for a comparison instead of DBDSQR. For Sheetal's implementation, we use the experimental results of diagonalization listed in the table of their paper. The results of Liu's are derived from their results presented in [8].

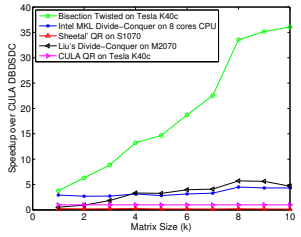


Fig. 2. Overall Performance Comparison

Figure 2 shows the performance comparison of BT implementation on Tesla K40c GPU to other existing libraries and implementations. The x-axis is the size of input matrix, and the y-axis is the speedup using CULA QR routine DBDSQR as the baseline. Our BT algorithm achieves a speedup of 3.8 to 36 over CULA culaDbdsqr routine, while Intel MKL DBDSDC routine has a 2.9 to 4.3 speedup on a 8 core CPU and Liu's implementation achieves only 0.5 to 4.7 speedup over CULA library. On the other hand, Sheetal's implementation is about 3 to 5.3 times slower than CULA library.

The performance of BT scales well when the matrix size becomes large. Overall, we achieve a speedup of 1.3 to 8.3 over the Intel MKL DC implementation on CPU, 4 to 7.2 over the Liu's DC method on GPU, and 15 to 288 over the QR implementation in the work by Sheetal et al. Now let us take a look at each of the algorithms from the perspective of matrix size, Sheetal's QR implementation and Liu's DC implementation do not work at all when the dimensions of matrices are larger than 14K by 14K on their GPU with memory size of 16GB

and 6GB, respectively. In contrast, in our implementation, the matrix size could reach 1 million by 1 million as shown in Table 3.

4.2 Performance Comparison on Different GPUs

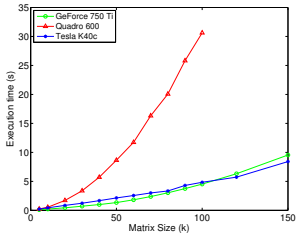


Fig. 3. Execution time of singular value kernel on different GPUs

Singular Value Computation Figure 3 shows the execution time of calculating singular values with our "equal number division" design on different GPUs with single-precision floating point. Quadro has the worst performance, while performance of GeForce and Tesla are close to each other. In particular, When the matrix size is less than $100K$, GeForce is slightly better. Otherwise, Tesla is better.

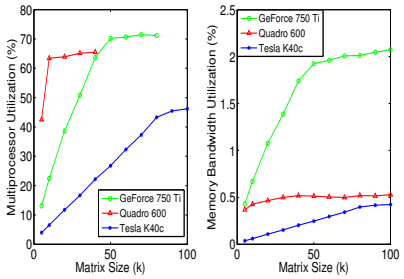


Fig. 4. Profiling Data of Singular Value Kernel on Different GPUs

To understand the reasons of such performance differences, we conduct a series of profiling experiments. Figure 4 shows the thread activity and memory bandwidth utilizations of singular value kernels on different GPUs, for matrices of size up to 100K (unfortunately we are unable to get profiling data for matrices of larger size due to the overflow of profiling counters). The figure shows that the thread utilization reaches 70% on Quadro and GeForce, and 50% on Tesla. But the memory bandwidth utilization is only 0.1%-2%. The main reason is that singular value computations rely on the fast shared memory of GPUs due to its

low memory requirements. That is, finding the singular values is rather CPU-bound than memory-bound. As a result, the performance is determined largely by the number of CUDA cores and the ratio of thread activity on a GPU.

Singular Vector Computation Figure 5 shows the execution time of singular vector kernel on different GPUs. It is easily seen that GeForce is about 8 times faster than Quadro. This is because GeForce has a much higher memory bandwidth than Quadro. In addition, the device memory read/write transactions of GeForce are only 1/6 and 1/4 of those of Quadro, respectively. The performance on Tesla is slightly better than that on GeForce. Tesla has nearly the same device memory read transactions as GeForce does, while 3 times more write transactions than GeForce. Yet Tesla is still the winner of the three because of its extremely high bandwidth as listed in Table 1.

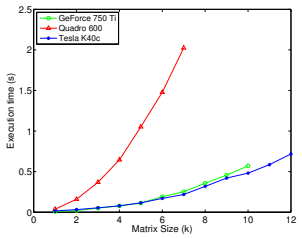


Fig. 5. Execution time of singular vector kernel on different GPUs

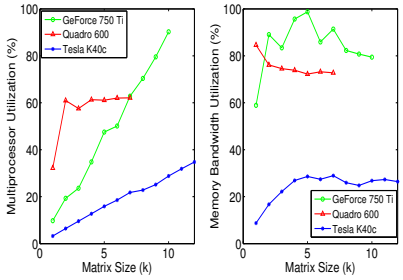


Fig. 6. Profiling Data of Singular Vector Kernel on Different GPUs

Figure 6 shows thread and memory bandwidth utilizations of singular vector design on different GPUs. We can see from the figure that Quadro and GeForce reach a high memory utilization (80%-99%), while the memory utilization of Tesla is around 30%. We also observe that when the matrix size is larger than 2K, the thread utilization keeps stable on Quadro. This is because the ratio of stall caused by memory is 70%, implying that there are a lot of threads waiting on data transfer.

4.3 Performance on a Multi-GPU Platform

Our solution for matrices of substantially larger sizes such as 1 million on each dimension is to scale the BT algorithm with multiple GPUs. As in our experimental platform, the multiple GPUs in a server may differ in architecture and performance, thus we need to balance the load on them to achieve the best speedup.

Load Balance Figure 7 shows the total execution times with load distributed to the two GPUs (Quadro 600 and GeForce 750 Ti) for different matrix sizes on our server. The x-axis represents the percentage of singular vectors calculated on Quadro 600 (with the rest on GeForce 750 Ti). The y-axis is the execution time. A red cycle highlights the spot where minimum execution time is reached for each matrix size. We can see that for matrices larger than 3K, a partition of 15% (Quadro) vs. 85% (GeForce) yields the best performance. We believe that this is because GeForce has 5 times more effective cores than Quadro.

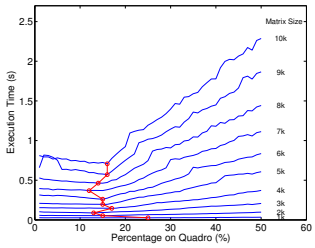


Fig. 7. Load Balance on GeForce and Quadro

Huge Size Result Table 2 gives the execution times for huge size matrices on a single GeForce, a single Quadra, and both of them with the optimal load balancing as shown in Figure 7. When the matrix size is larger than 100K, a single Quadro is not able to calculate the whole SVD. The GeForce can calculate the whole SVD until the matrix size reaches to 200K. When the matrix size becomes larger than 200K, our bisection algorithm on GeForce cannot converge due to the accuracy of IEEE single-precision floating point number on GeForce. When we use double-precision floating numbers on Tesla, our BT implementation works well even when the matrix size reaches one million.

Table 3 shows the performance of huge size matrix with double-precision floating-point numbers on a single Tesla and two Tesla GPUs on a server. For two Telsa GPUs, we compared static workload allocation (50%/50%) and dynamic allocation where each GPU is tracked constantly by the host CPU and assigned new workload as soon as it finishes the current kernel. When matrix size reaches 1 million by 1 million our BT algorithm reaches the results in 54801 seconds

Table 2. Performance with Very Large Matrix Sizes

Matrix Size	GeForce	Quadro	GeForce + Quadro
50K*50K	19s	96s	16s (84%) / 15s (16%)
80K*80K	50s	238s	43s (84%) / 40s (16%)
100K*100K	84s	393s	71s (83%) / 67s (17%)
120K*120K	134s	-	113s (82%) / 102s (18%)
150K*150K	245s	-	203s (81%) / 194s (19%)
180K*180K	412s	-	340s (80%) / 337s (20%)
200K*200K	555s	-	453s (80%) / 450s (20%)

with a single Telsa, and 35607 seconds with two Telsa GPUs. This is a 1.54X speedup.

Table 3. Performance of Huge Size Matrix with double floating-point on Tesla

Matrix Size	Tesla	Static	Dynamic
100K*100K	341s	217s / 189s	210s / 202s
120K*120K	524s	326s / 286s	311s / 311s
150K*150K	864s	524s / 467s	498s / 507s
200K*200K	1407s	955s / 827s	849s / 858s
250K*250K	1949s	1286s / 1118s	1199s / 1204s
300K*300K	3490s	2234s / 1906s	2123s / 2110s
400K*400K	6559s	4110s / 3709s	3853s / 3871s
500K*500K	12282s	7371s / 6916s	7148s / 7129s
800K*800K	40311s	22454s / 21627s	22046s / 22026s
1000K*1000K	54801s	36119s / 35071s	35587s / 35607s

4.4 Profiling Analysis of GPU Kernels

Comparison of Two Different Singular Value Designs We compare the execution time on two different singular value kernels: "equal length division" versus "equal number division". Each method has two phases: (1) divide the interval into subintervals and (2) calculate singular values in each subinterval. Figure 8 shows the detailed execution time breakdown for each phase of both methods ("Interval Division" time for "equal-length division" is negligible) on Tesla K40c. From the figure, we can see that when the matrix size is less than 9K, the equal length division version runs a little faster than equal number division version. However, when the matrix size exceeds 9K, the execution time of the equal length division version increases dramatically, while the execution time of equal number division version still rises linearly. And in this case, even though the time to divide the interval is noticeably large, the balanced number of singular values in a subinterval still yields much better performance. Thus, the equal number division version is obviously the winner when the matrix size becomes larger than 9K.

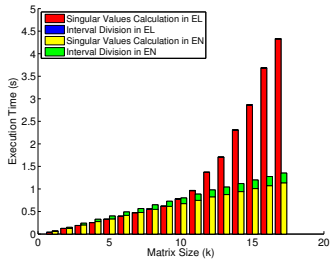


Fig. 8. Comparison of Equal Length Division and Equal Number Division. "Interval Division in EL" is negligible.

Memory Access Optimization We evaluate the memory optimization techniques on improving the performance of singular vector calculation. As depicted in Figure 9, in the baseline design, Global memory Load Transactions (GLT)

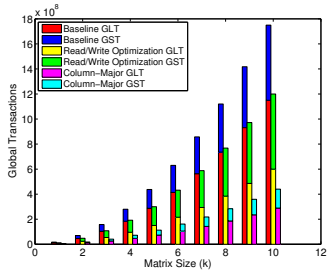


Fig. 9. Memory Transactions on Singular Vector Design

are about twice of the Global memory Store Transactions (GST) on the global memory. As there are 50% of global memory transfers are read-after-write, we improve the memory access performance by copying these values into the local memory and shared memory of the GPU. As a result, the GLT are reduced by 50% compared to the baseline, while the GST remains the same, labeled as "Read/Write Optimization" in Figure 9. The speedup on singular vector calculation reaches to 1.2X compared to the baseline. Changing the matrix arrangement from row-major to column-major in the global memory reduces GLT and GST by 50% and 25%, respectively, compared to "Read/Write optimization". This is because column-major matrix have coalesced global memory accesses, which saves hundreds of transactions per thread. The speedup rises up to 4.5X compared to the baseline.

5 Related Work

General purpose GPUs have become important processing engines for computationally intensive workloads due to their highly parallel computing architec-

tures. Linear algebra operations are among the first batch of libraries accelerated with GPUs. There are various such accelerated libraries, i.e., CUBLAS, CULA, MAGMA. The BLAS (Basic Linear Algebra Subprograms) are routines supported by Nvidia [10] that provide standard building blocks for performing basic vector and matrix operations. CUBLAS is an implementation of BLAS and provides many solutions for common linear algebra operations. However, it does not provide solutions to complex linear algebra problems, such as QR decomposition, LU decomposition and SVD. CULA consists of commercial hybrid GPU accelerated linear algebra routines[11] that support high-level linear algebra operations. Among them, the singular value decomposition employs QR algorithm, which is not an effective algorithm when the matrix size becomes significantly large. MAGMA aims to achieve high performance and portability across a wide range of multi-core architectures and hybrid systems respectively[12].

QR, the most commonly used algorithm for computing singular values and vectors, is regarded as a powerful and effective algorithm with high accuracy and numerical stability[1]. Their implementation shows a speedup of up to 8 over the Intel MKL QR implementation [13]. The first SVD algorithm with CUDA programming is implemented by Sheetal et al.[7]. With parallelized QR iteration algorithm using CUBLAS library on GPU, they achieved a speedup of up to 8 over the Intel MKL QR implementation. However, the QR-iteration algorithm has its drawback on parallelization due to its heavy data dependency. The algorithm requires $O(n^3)$ to complete the diagonalization and its decomposition time is intolerable as size of matrices increase.

Liu et al. use a divide-and-conquer approach to solve SVD on a heterogeneous CPU-GPU system [8]. It is almost 7 times faster than CULA QR algorithm executing on the same device M2070, and up to 33 times faster than LAPACK. However, DC approach is relatively inaccurate during the merging phase, especially when the data scale is huge. In the worst case, it will require $O(n^3)$ to complete SVD, if the singular values are in a dense distribution[1].

Vedran[9] presents a hierarchically blocked one-sided Jacobi algorithm for the singular value decomposition on both single and multiple GPU architectures. The algorithm maintains high accuracy in singular values and vectors. Due to the speed limitation of the algorithm, even with full optimizations and a high speedup compared to the same algorithm on CPU, the execution time is still more than that of Sheetal's QR implementation.

In [23] Drineas et al. provide a clustered SVD algorithm for large matrices. The algorithm divides a set of n points into k clusters, where k is much less than n on CPU. It is an approximation algorithm to obtain only one subset of singular values and vectors. Our BT algorithm can compute the complete SVD, thus is not directly compared with [23].

6 Conclusion

We present a novel algorithm for computing singular values and vectors called bisection and twisted (BT) algorithm, which is implemented on both single GPUs

and a multi-GPU platform. The experimental results show that BT outperforms a number of existing work on SVD acceleration. One of the major advantages for BT algorithm is its scalability: we are the first to perform SVD on a matrix of 1 million by 1 million, using only two GPUs. In the near future, we plan to extend our multi-GPU version of BT with network middleware such as MPI so that BT algorithm can be further extended to distributed GPUs.

Acknowledgment

This work is support in part by the National Science Foundation under grant number ACI-1440737.

References

1. James W. Demmel, *Applied Numerical Linear Algebra*, 1st ed. Philadelphia, USA: Society for Industrial and Applied Mathematics, 1995.
2. James W. Demmel, Inderjit Dhillon, and Huan Ren. *On The Correctness Of Some Bisection-Like Parallel Eigenvalue Algorithms In Floating Point Arithmetic.* Electronic Transactions on Numerical Analysis. Volume 3, pp. 116-149, December 1995.
3. G. Golub and W. Kahan *Calculating the Singular Values and Pseudo-Inverse of a Matrix.* SIAM Journal for Numerical Analysis; Vol. 2, #2; 1965.
4. Leslie Hogben, Kenneth H Rosen. *Handbook of Linear Algebra.* Taylor & Francis Group, LLC, 2007.
5. W. Xu, S. Qiao, *A Twisted Factorization Method for Symmetric SVD of a Complex Symmetric Tridiagonal Matrix.* Numerical Linear Algebra with Applications, 16 (10) (2009), pp. 801815.
6. P.R. Willems, B. Lang, and C. Vömel. *LAPACK Working Note 166: Computing the Bidiagonal SVD Using Multiple Relatively Robust Representations.* EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-05-1376, 2005.
7. Sheetal Lahabar, P. J. Narayanan. *Singular Value Decomposition on GPU using CUDA.* Proc. IEEE Int',l Symp. Parallel and Distributed Processing, pp. 1-10, 2009.
8. Ding Liu, Ruixuan Li, David J. Lilja and Weijun Xiao. *A Divide-and-Conquer Approach for Solving Singular Value Decomposition on a Heterogeneous System.* CF'13 Proceedings of the ACM International Conference on Computing Frontiers Article No. 36
9. Vedran Novakovic. *A Hierarchically Blocked Jacobi SVD Algorithm for Single and Multiple Graphics Processing Units.* http://arxiv.org/abs/1401.2720v2.
10. *NVIDIA CUDA CUBLAS Library*, 2nd ed, NVIDIA Corporation (August 2010).
11. John R. Humphrey, Daniel K. Price, Kyle E. Spagnoli, Aaron L. Paolini and Eric J. Kelmelis. *CULA: Hybrid GPU Accelerated Linear Algebra Routines.* Proc. SPIE 7705, Modeling and Simulation for Defense Systems and Applications V, 770502 (April 26, 2010); doi:10.1117/12.850538; http://dx.doi.org/10.1117/12.850538.
12. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S. *Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects.* Journal of Physics: Conference Series, Vol. 180, 2009.

13. *Reference Manual for Intel Math Kernel Library 11.1.* https://software.intel.com/en-us/intel-mkl.
14. M. R. Hestenes. *Inversion of Matrices by Biorthonalization and Related Results.* J. Soc. Indust. Appl. Math., 6 (1958), pp. 5190.
15. J. Demmel and W. Kahan. *Accurate Singular Values of Bidiagonal Matrices.* SIAM J. Sci. Stat. Comput., 11(5):873-912, 1990.
16. G. Golub and W. Kanhan.
17. Calculating the Singular Values and Pseudo-Inverse of a Matrix. SIAM J. Num. Anal. (Series B), 1965.
18. M. Gu, J. Demmel and I. 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.
19. Bell, H. E. *Gerschgorin's Theorem and the Zeros of Polynomials.* Amer. Math. Monthly 72, 292-295, 1965.
20. J. E. Gubernatis, and T. E. Booth. *Multiple Extremal Eigenpairs by the Power Method.* http://arxiv.org/pdf/0807.1261.pdf.
21. M. Panju. *Iterative Methods for Computing Eigenvalues and Eigenvectors.* Waterloo Math. Rev., Vol. 1, pp.9-18 2011.
22. Renan Cabrera, Traci Strohecker, and Herschel Rabitz. *The Canonical Coset Decomposition of Unitary Matrices through Householder Transformations.* Journal of Mathematical Physics 51 (8). doi:10.1063/1.3466798.
23. P. Drineas, A. Frieze, R. Kannan, S. Vempala and V. Vinay. *Clustering in Large Graphs and Matrices.* ACM-SIAM Symposium of Discrete Algorihtms 1999.