Bisection and Twisted Bidiagonal SVD on GPU

Authors Name/s per 1st Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

***Abstract*—Singular value decomposition (SVD) has many useful applications in various fields, while current implementations are still time-consuming, especially when matrix size becomes to ten thousands of order of magnitude. Modern General Purpose GPUs have shown their extreme computational advantages in parallel computing. In our paper, we present an implementation of one effective bidiagonal SVD method, bisection and twisted algorithm, with CUDA programming on Nvidia GPU. The algorithm can calculate all singular values and vectors at the time cost of $O(n^2)$ in serial. Additionally, It is able to obtain any subsets of singular values and vectors directly. This feature is suitable for current applications that does not require the whole SVD calculation. We design our GPU kernels carefully. It only takes 0.8 second to obtain all the singular values and vectors with single procision on GeForce 750Ti when matrix size is $10k*10k$, and 1.3 seconds on Tesla K40c with double precision. It is up to 10 times faster than MKL divide-and-conquer routine DBDSDC with 8 cores 16 threads, and 36 times faster than CULA QR routine DBDSQR on the same GPU. Our results also show that our algorithm has better performance than previous SVD approaches on CPU and GPU. We also evaluate the effect of execution time when different number of block is allocated, and analyze the performance when error tolerance becomes strict. In addition, we improve our implementation to a huge matrix size with 1 million $\times$ 1 million. To our knowledge, No implementation has achieved such size.**

***Index Terms*—Singular Valu Decomposition; GPU; Twisted fraction; Bisection algorithm.**

I. INTRODUCTION

Singular value decomposition (SVD) has been widely used in various fields in recent years, such as 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, data compression in information theory. Current SVD approaches could solve small-scale data in acceptable time and space requirements. Unfortunately, with the advent of the era of data explosion, small-scale data processing has been difficult to meet the needs of big data. Thus, it is still a problem to extend the SVD algorithms on a very large data set.

The SVD can be broken down into two steps [3]: The first step is to reduce the initial matrix to bidiagonal form using Householder transformations. The second step is to diagonalize the resulting matrix using bidiagonal SVD algorithms. Most of researchers focus on the second step with iteration methods [13]–[15]. This is because the first step executes only one time by Householder transform, while the second step executes much more than once dependent on accuracy requirement.

Three algorithms has been introduced to solve the bidiagonal SVD. QR algorithm is recognized as a powerful and effective method. However, it is only the fastest algorithm for small matrices whose sizes are less than 25 [1]. The complexity of $O(n^3)$ flops will give the execution time a rapid increment, as matrix size becomes large. Also, the iteration time rises to a large integer number when matrix size is large. Jacobi algorithm is the most accuracy method in practise [1]. However, the $O(n^3)$ flops with big constant will cause the algorithm much more slower than other algorithms. Also, the iteration time of Jacobi algorithm is much larger than those of QR algorithm.

Divide-and-conquer algorithm is assumed as the fastest method of SVD when matrices are large [17]. It takes $O(n^{2.3})$ flops on average [1]. But the singular values are not relative accuracy when merging, let alone singular vectors. If all the singular values are distributed in the worst case, the time cost will increase to $O(n^3)$.

In addition to the speed and relative accuracy, all of these three algorithms above have two common disadvantage for parallel computing:

1) These algorithms have heavy data dependence. The heavy data dependence makes SVD algorithm not suitable for parallelization and architecture extension.
2) These algorithms require $O(n^2)$ memory locations to save temporary variables. The large memory locations needed in these algorithms will not be able to calculate the SVD when the matrix is large enough.

Most of SVD applications, such as principle component analysis (PCA), need only a small subset of the singular values and vectors. However, the algorithms above are not able to calculate the subset directly. Fortunately, bisection and inverse (BI) algorithm could find the subset easily. Bisection and Inverse iteration takes $O(nk)$ flops to find k singular values and singular vectors, and $O(nk^2)$ in the worst case of k singular values are clustered. It is much faster than other algorithms, especially when only a small subset of singular values and vectors are needed in a huge data size. But the inverse iteration has its drawback. It does not guarantee the accuracy and orthogonality of the computed singular vectors in the case of singular values clustered.

In this paper, we present a new SVD approach, bisection and twisted (BT) algorithm. It inherits the advantages of BI algorithms. It has been proved that the singular vectors are accurate and orthogonal in twisted algorithm [5]. Comparing to other algorithms, BT approach only requires $O(n^2)$ flops to obtain all singular values and singular vectors [5], [6], even in the worst case. It is faster than any other algorithms metioned above. The data dependence is weak in bisection and twisted algorithm. It is excellent for parallelism on multi-threads and extention to multi-GPU. The algorithm can also obtain one singular value and its conresponding vector in $O(n)$ flops. Additionally, the algorithm needs only $O(kn)$ memory location to store temporary memory. It is good to extend to many threads or many cores.

The rest of the paper is organized as follows. Section 2 discusses the related work. A high-level serial algorithm is given in Section 3. Section 4 describes the implementation of our bisection and twisted algorithm on GPU. Section 5 presents the experimental results and profiling data of GPU kernels. Future work and conclusion are given in Section 6.

II. RELATE WORK

General purpose GPU (GPGPU) becomes the research focus for its parallel computing power these years. It provides a functionally complete set of operations to compute any computable value. Linear algebra is the first batch of object to accelerate on GPU.

CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) library supported by NVIDIA [10]. Many solutions of simple linear equations are provided in CUBLAS. However, it does not provides the complex linear algebra problems, such as QR decomposition, LU decomposition, and SVD. CULA is an commercial hybrid GPU accelerated linear algebra routines [11]. It provides high-level linear algebra operations. The singular value decomposition employs QR algorithm. It is not a effective algorithm when matrix scale becomes larger. MAGMA is the project that aims to achieve high performance and portability across a wide range of multi-core architectures and hybrid systems respectively [12].

SVD algorithms is a high-level linear algebra routines. It is still a research direction because of its various applications.

The QR algorithm is the most commonly used SVD algorithm. It is able to calculate the singular values and singular vactors with high relative accuracy and numerical stability [1]. Sheetal et al [7] firstly introduces SVD algorithm on CUDA programming. In their method, they apply QR iteration algorithm and parallelize the algorithm on GPU. They use CUBLAS library to accelerate matrix and vector operations, and design an architecture for bidiagonalizaiton and diagonalization. Their implementation shows a speedup of upto 8 over the Intel MKL QR implementation. However, the QR-iteration algorithm has its defect on parallelization. The heavy data dependency makes QR algorithm not suitable for parallelization. The QR-iteration algorithm requires $O(n^3)$ to complete the diagonalization. The decomposition time is not intolerable when the matrix size scale is huge. Additionally, calling too much CUBLAS kernels also reduces the performance in the GPU design.

Ding et al [8] implement the divide-and-conquer approach for solving SVD on heterogeneous CPU-GPU system. It is up to 7 time faster than CULA QR algorithm executing on the same device M2070, and up to 33 times faster than LAPACK. However, the divide-and-conquer approach is not relative accuracy in the merging phase, especially when the data scale is huge. In the worst case, if the singular values are in the dense distribution, it will require $O(n^3)$ to complete SVD [1]. The implementation in the paper utilizes pipelines in the heterogeneous architercture. It does not make full use of resources on GPUs. The frequent data transfer between GPU and CPU will drag the speedup in the execution time. Additionally, their pipeline architecture is difficult to extend to multi-core architecture.

Vedran [9] presents a hierarchically blocked one-sided Jacobi algorithm for the singular value decomposition, targeting both single and multiple GPUs. It is the first paper to introduce multi-GPU to calculate SVD. But due to the speed limitation of the algorithm, even it is fully optimization and has a high speedup compared to the same algorithm on CPU, the execution time is still more than that of any other algorithms. It could not be the best algorithm for speedup except when extreme relative accuracy is required.

III. BISECTION & TWISTED ALGORITHM

The singular value decomposition of an arbitrary matrix $A \in R^{mn}(m > n)$ consists 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 diagonal matrix.

All algorithms have to make use of Householder transformation to reduces the time cost of SVD. The time complexity of QR algorithm, Jacobi algorithm and bisection and inverse algorithm will change from $O(n^3)$ to $O(n^4)$ without Householder transformation, while divide-and-conquer algorithm cannot work any more. Thus, in this paper, we only focus on the second step with bisection and twisted algorithm.

The bisection and twisted algorithm is separated into two phases:

1) Obtain the singular values of bi-diagonal matrix by using bisection approach.
2) Obtain the conresponding left and right singular vectors of each singular values by using twisted factorization.

We will illustrate these two phases in the following subsections.

A. Bisection Algorithm

Bisection algorithm is widely used in many application to find the root of a sophisticated equation which can not be solved directly. It repeatedly bisects an interval and then selects a subinterval for further processing until convergence. The algorithm is an approximate approach to solve the sophisticated equations. It can obtain relative accuracy solution when the toleration is relative strict.

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

$$T = B^T B - \mu^2 I = LDL^T \tag{1}$$

where D is $diag(d_1, \cdots, d_n)$,

$$L = \begin{pmatrix} 1 & & & & \\ l_1 & 1 & & 0 & \\ & l_2 & \ddots & & \\ & 0 & \ddots & \ddots & \\ & & & l_{n-1} & 1 \end{pmatrix} \tag{2}$$

Substitute L and D into Equation 1, we can obtain the following equations

$$\begin{cases} b_{1,1}^2 - \mu^2 = d_1 \\ b_{k-1,k-1} b_{k-1,k} = d_{k-1} l_{k-1} \\ b_{k-1,k}^2 + b_{k,k}^2 - \mu^2 = l_{k-1}^2 d_{k-1} + d_k \end{cases} \tag{3}$$

where $k = 2, 3, \cdots, n$. We define a temperary value t_k

$$t_k = t_{k-1} * (b_{k-1,k}^2 / d) - \mu^2. \tag{4}$$

After equivalent transformation, the Equations 3 can be rewrited as

$$d_k = b_{k,k}^2 + t_k \tag{5}$$

It is clear that matrix D and matrix T are two congruent symmetric matrices. According to the Sylvester's law of inertia, matrix D and matrix T has have the same numbers of positive, negative, and zero eigenvalues. We define $NegCount(\mu)$ function to be the number of negative eigenvalues in matrix T with μ^2 shift. Since matrix B and matrix B^T have the same singular values, $NegCount(\mu)$ is also 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 increasing function of x [2]. The $NegCount$ function is in Algorithm 1.

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:          end if
9:      end for
10:     return cnt;
11: end procedure
```

Algorithm 2 is the serial bisection algorithm. It is able to calculate the singular values in interval $[l, u)$, whose $NegCount$ are n_l and n_u, seperately. The basic steps of the algorithm for singular value are as follows:

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:      end if
5:      Enqueue (l, u, n_l, n_u) to Worklist;
6:      while Worklist is not empty do
7:          Dequeue (a, b, n_a, n_b) from Worklist;
8:          if b − a < τ then
9:              for i = n_a + 1 → n_b do
10:                 val[i] = min(max((a + b)/2, a), b);
11:             end for
12:         else
13:             m = MidPoint(a, b);
14:             NegCount(n_m, B, m);
15:             n_m = min(max(n_m, n_a), n_b);
16:             if n_m > m_a then
17:                 Enqueue (a, m, n_a, n_m) to Worklist;
18:             end if
19:             if n_m < m_b then
20:                 Enqueue (m, b, n_m, n_b) to Worklist;
21:             end if
22:         end if
23:     end while
24: end procedure
```

1) Divides one interval containing singular values into two subintervals.
2) Utilizes $NegCount$ algorithm to get the number of singular values in subintervals.
3) Selects the subinterval(s) which contain singular values for further bisection.
4) Repeated 1-3 until all subintervals becomes convergence.
5) The singular values are considered as the midpoint of the subintervals.

The algorithm provides theoretical basis to calculate the singular values in a specified interval. The interval containing all the singular values can be calculated by Gershgorin circle theorem.

B. Twisted Algorithm

After the first step of obtaining the singular values, it is still necessary to get their corresponding left and right singular vectors. The simplest method to obtain the singular vectors is the power method, which can find only the largest singular value and the corresponding singular vector [1]. The second method is inverse iteration. However, there is no guarantee that the singular vectors are accurate or orthogonal. Additionally, the inverse iteration requires $O(n^3)$ to obtain all the singular vectors.

We utilize the improved twisted factorization to calculate the singular vectors. The algorithm improves the accurate or orthogonal problems in general bisection and inversed

algorithm. It can also solve the singular vectors whose singular values are clustered together.

Suppose λ is one singular value of bidiagonal matrix B. Then the matrix $B^TB - \lambda^2 I$ can be decomposed as

$$B^TB - \lambda^2 I = LD_LL^T = UD_UU^T \tag{6}$$

where $D_L = diag(\alpha_1, \cdots, \alpha_n)$, $D_U = diag(\beta_1, \cdots, \beta_n)$, L is the same form with Equation 2

$$U = \begin{pmatrix} 1 & u_1 & & \\ & 1 & u_2 & 0 \\ & & 1 & \ddots \\ & 0 & & \ddots & u_{n-1} \\ & & & & 1 \end{pmatrix} \tag{7}$$

Given the LDL^T and UDU^T decomposition, we consider the twisted factorization of the shifted matrix

$$B^TB - \lambda^2 I = N_kD_kN_k^T \tag{8}$$

where N_k is the twisted matrix. k is the index of the minimum γ in Eq. 9.

$$k = \arg\min_{1\le i\le n} \gamma_i = \begin{cases} \beta_1 & \text{if } i = 1 \\ \beta_i - \alpha_i * l_{i-1} & \text{if } 2 \le i \le n-1 \\ \alpha_n & \text{if } i = n \end{cases} \tag{9}$$

Thus, the corresponding singular vector of λ is solving the following matrix equations and normalize the solution.

$$N_k^T z_k = e_k \tag{10}$$

where e is the unitary matrix, and z is the non-normalization solution of singular vectors.

Our algorithm has a small modification when singular values are clustered together. Suppose r is the multiplicity of singular values of matrix B. The algorithm selects the indices of the first r minimum value of $gamma$. Each index k has one different twisted factorization, and thus has a different Eq. 10 to solve. The singular vectors are also orthogonal to each others [5]. Algorithm 3 is the algorithm to obtain the corresponding singular vectors of given singular values in serial. The cost for every singular vector transformation is $O(n)$, and the total cost of the transformations is $O(n^2)$.

IV. Implementation & Optimization on GPU

In this section, we will describe the implementation of bisection and twisted algorithm on GPU. GPU chooses Single Program Multiple Threads (SPMT) mode for its scheduling. A thread is the fundamental working unit of a parallel program. A warp with continuous 32 threads works concurrently. A block is a group of multiple threads sharing the shared memory. Based on the algorithm and the GPU architecture, the implementation can be separated into two steps one after another. The first one is singular value kernels and the last one is singular vector kernels. In the end of this section, we also introduce an implementation when matrix size becomes huge.

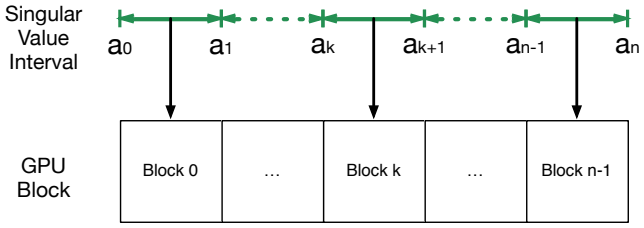
Algorithm 3 Twisted Factorization

```
procedure Twisted(q, n, B, μ)
    ▷ l is the number of different singular values, l ≤ n;
    for i = 1 → l do
        Computes the matrix S = B − μ_i^2 I;
        Computes the LDL' decomposition S = LD_L L';
        Computes the UDU' decomposition S = UD_U U';
        Computes gamma based on Eq 9;
        Find the multiplicity m of singular value μ;
        Find m-th minimum k = min_j|γ(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 → n do
                z_j = −U_{j-1,j} * z_{j-1}
            end for
            for j = k − 2 → 1 do
                z_j = −L_{j+1,j} * z_{j+1}]
            end for
            Scale vector q = z/||z||_2;
        end for
    end for
end procedure
```

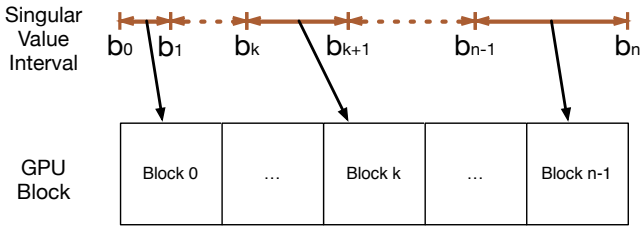
A. Singular Value Kernels

In the singular value design, we have two steps to obtain the singular values.

1) Seperate the whole interval into several subintervals in parallel.
2) Obtain the singular values in these subintervals in parallel.



(a) Equal Length Division



(b) Equal Number Division

Fig. 1. Division Strategies of interval

For the first step, we design the strategy to separate the interval by the length shown in Fig.1(a). In this strategy, we

divide the whole interval $[a_0, a_n)$ into n subintervals with the same length, while every subinterval may not have the same number of singular values usually. In mathematical word, this strategy can be expressed as

$$a_{k+1} - a_k = a_k - a_{k-1} \tag{11}$$

$$NegCount(a_{k+1}) - NegCount(a_k) \neq NegCount(a_k) - NegCount(a_{k-1}) \tag{12}$$

Algorithm 4 Equal Length Subinterval Algorithm

procedure $\mathbf{Length\_Divide}(n, B, t, \tau)$
  Obtain singular value boundary $[l, u)$ of matrix B;
  Define the thread ID i, $i < t$;
  Obtain average step $s = (u - l)/t$;
  Lower bound $\alpha = l + i * s$;
  $n_\alpha = NegCount(\alpha)$;
  save the division point α and n_α
end procedure

The parallel algorithm of length division working in GPU threads is shown in Algorithm 4. The length division strategy is easy to design. However, it is not flexible to determine the number of subintervals for a better speedup. This is because the determination of number of blocks is based on multiple aspects, such as the maximum number of threads, the number of threads in GPU warp, and the size of matrix. The maximum number of threads per block determines the minimum number of blocks should be allocated. The number in threads in GPU warp makes sure the suitable subintervals for best GPU utilization level.

Even we separate the interval with the best GPU utilization level, the speedup of length division cannot also reach a high level. To achieve a even better speedup, we use another strategy to spearate the whole interval by the number of singular values in Fig.1(b). In this strategy, we divide the interval $[b_0, b_n)$ into n subintervals with the same number of singular value. However, the length of the subinterval may not be equal to others. The mathematical representation can be writed as

$$b_{k+1} - b_k \neq b_k - b_{k-1} \tag{13}$$

$$NegCount(b_{k+1}) - NegCount(b_k) = NegCount(b_k) - NegCount(b_{k-1}) \tag{14}$$

The parallel algorithm with same number of singular value is shown in Algorithm 5.

We calculate the boundary points of the subintervals from the first step based on Algorithm 4 and Algorithm 5. Either algorithm works in one block with multiple threads. Every thread is responsible for one spliting point. The second step is to calculate the exact singular values in these subintervals with Algorithm 2.

In our design, one subinterval is allocated into one GPU block. In other words, one GPU block will calculate all the singular values in its corresponding subinterval. Inside the block, multiple threads are working concurrently. One thread will obtain one singular value in the subinterval.

Algorithm 5 Equal Number of Singular Value Subinterval Algorithm

procedure $\mathbf{Number\_Divide}(n, B, t, \tau)$
  Obtain singular value boundary $[l, u)$ of matrix B;
  Define the thread ID i, $i < t$;
  $mid = inside(l, u, B)$;
  $n_m = NegCount(n, B, mid)$;
  while $n_m \neq (i+1)n/t$ and $mid - l > \tau$ **do**
    if $n_m \geq (i+1)n/t$ **then**
      $u = mid$;
      $n_u = n_m$;
    else
      $l = mid$;
      $n_l = n_m$;
    end if
    $mid = inside(l, u, B)$;
    $n_m = NegCount(n, B, mid)$;
  end while
  save the division point mid and n_m.
end procedure

B. Singular Vector Kernel

The singular vector kernel is designed based on Algorithm 3. First, we achieve the algorithm without any optimization. But the 10X speedup compared to CPU is low because of the heavy usage of global memory and the low global memory throughput. After we utilize the local memory and shared memory instead of frequent-used global memory and change the matrix arrangement from row-major to column-major, The performance nearly doubled.

Since the singular vectors are two $n \times n$ matrices, the memory on GPU will effect the maximum matrix size. Based on our desing, all the singular vectors require $5 \times n^2$ floating numbers. Thus, the maximum matrix size is determined by

$$m_t = \sqrt{U/(5 * 4)} \tag{15}$$

where U is the memory size of GPU, 4 means the number of bytes of float number.

C. Huge Size Solution

In our design, the maximum singular values can be processed on a singular GPU is $m_b = block\_size \times thread\_block\_size$, while the maximum singular vectors are determined by Equation 15. 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 becomes larger than m_t, even larger than m_b, the GPU kernels cannot obtain all singular values and vectors any more. In this subsection, we will introduce a divide-and-conquer architecture to solve the huge matrix size.

When matrix size is less than m_b, but larger than m_t from Equation 15, we only need to separate the singular vectors

into small pieces which can be processed by a single GPU. When matrix size is larger than m_b, we should separate both singular values and singular vectors into small pieces.

The division of singular value can be divided by m_b directly. Thus, there are $l_b = \lfloor (n/m_b) \rfloor + 1$ pieces, and each pieces has $\lfloor (n/l_b) \rfloor$ singular values.

The division of singular vector should take memory size into consider. The maximum pieces l_t can be rewrite from Equation 15 as follow:

$$l_t = \sqrt{U/(5 * n * 4)} \tag{16}$$

where n is matrix size. Thus, There should be $\lfloor (n/l_t) \rfloor + 1$ pieces with $\lfloor (n/l_t) \rfloor$ singular vectors each pieces. For a single GPU, the execution between every pieces is serial. When one piece finished, GPU processes the next one, until the last one. Thus, the execution time is the summation of every subsection. Equation 16 also provides the theorical maximum matrix size can be processed on GPUs when set $l = 1$. For Tesla K40c, the theorical maximum matrix size is 600 million. However, the singular vector kernel will not have speedup compared to execution on CPU.

When matrix size is huge, only one GPU is not enough to speed up the SVD. We make use of pthread to control mutiple GPUs. Each thread created by pthread takes control of one GPU.

We also build a two-computer network to speed up the SVD. In our design, each computer has one GPU and communicated through sockets.

V. Experiment Result

In this section, we will analyze the performance of our algorithm compared to other SVD implementations on CPU and GPU. In addition, we will discuss the GPU kernel profiling result to show how to improve our implementations. We also show our performance on huge matrix size.

We tested our algorithm on GeForce 750 Ti, Quadro 600 and Tesla K40c. The specifications are in Table I.

TABLE I
Specifications of Different GPUs

Specifications	GeForce 750	Quandro 600	Tesla K40
Architecture	Maxwell	Fermi	Kepler
CUDA Cores	640	96	2880
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

A. Comparision to other implementations

We generated random bidiagonal matrices with double precision numbers in the range of 0 to 1. In order to obtain relative accuracy experimental results, we generates 10 random matrices. For each matrix, the SVD algorithm was executed 10 times on GPU. The average performance does not vary much when more matrices and more times are used.

We compare our algorithm with CULA GPU library, Intel MKL library, Sheetal's QR implementation on S1070, and Liu's divide-and-conquer implementation on M2070. We measure the performance of CULA on Tesla K40c, and that of Intel MKL on an 8-core cpu with 16 threads. Until now, CULA library only has QR routine culaDbdsqr. Intel MKL library has both divide-and-conquer DBDSDC and QR routine DBDSQR. Usually, divide-and-conquer routine is faster than QR routine, so we select DBDSDC instead of DBDSQR. We measure the performance of CULA on Tesla K40c, and that of Intel MKL on an 8-core cpu with 16 threads. For Sheetal's implementation, we use the experimental results of diagonalization listed in the table of their papers. The results of Liu's are estimated comprehensively according to the figures presented in their paper.

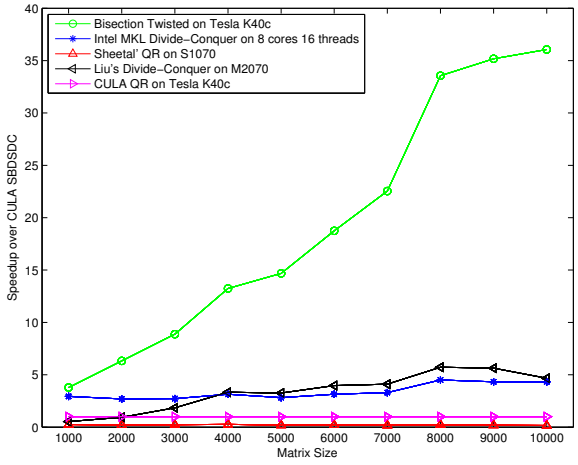


Fig. 2. Overall Performance Comparision

Figure 2 shows the performance of our implementation on Tesla K40c GPU to other existing libraries and implementations. We selects CULA QR routine BDBSQR as a baseline. From the figure, we achieve 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 has only 0.5 to 4.7 speedup over CULA library. Sheetal's implementation is about 3 to 5.3 times slower than CULA library. The performance goes up when matrix size becomes large. Overall, we achieve a speedup of 1.3 to 8.3 over the Intel MKL Divide-Conquer Implementation on CPU, 4 to 7.2 over the Liu's Divide-conquer method, 15 to 288 over the QR implementation Sheetal Paper.

B. Profiling Data

1) the Optimal Block Size: In our length division design on singular value, the number of block size determines the execution time of the whole singular values. An inappropriate block division will affect the performance on GPU heavily. To obtain the optimal block size, we make use of experimental method.

Figure 3 shows the elasped time of obtaining all the singular values with different number of blocks and different matrix

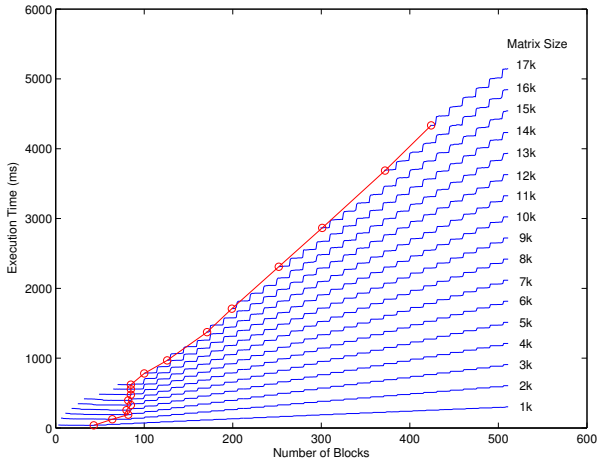


Fig. 3. The optimal block number of different matrix size with double precision on Tesla K40c

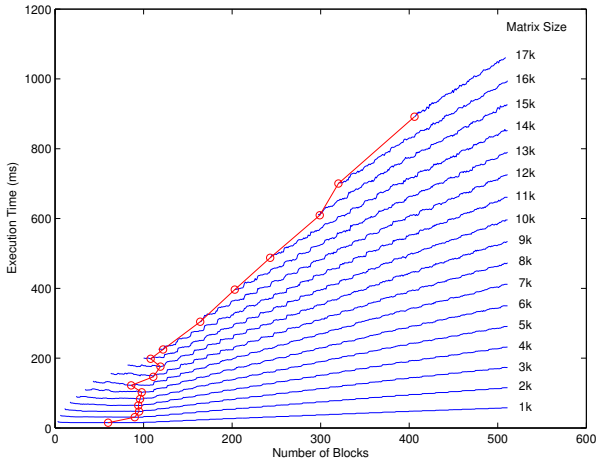


Fig. 4. The optimal block number of different matrix size with single precision on GeForce 750 Ti

size on GPU Tesla K40c with double precision. The blue wavy curves show the relationship between the execution time and the number of blocks on the matrix size in the right column. From the curves, we can see that the performance goes down slightly first, and then rises when the number of blocks increases. The left point of the wavy curve shows the minimal number of blocks should be allocated when the matrix size is identified. In other words, the number of blocks should not be less than the left point on the wavy curves. When matrix size becomes large, the left point in wave curve trends to a large number of blocks.

The red circle curve in the figure shows the optimal number of blocks with minimal execution time on different size. we can see that the optimal number of blocks increases when matrix size is less than $3k$, keeps stable when matrix size is in the range of $3k$ to $9k$, and becomes the minimal number of blocks when matrix size is lager than $9K$. Actually, when the block number is close to the optimal number, the execution time does not change too much. Due to the uncertainty of the input matrix, the optimal block number may vary slightly. Thus, it is not necessary to select the exact optimal block number. But it is better to select the number of blocks around the optimal one.

Figure 4 is another experimental results on GeForce 750 Ti GPU with single precision. The curves are very similar with the curves in Figure 3. However, the optimal number of blocks shifts right to be a large number because GeForce has a better architecture than Tesla.

2) Comparasion of two different Singular Value Design: In this part, we compare the execution time on two different singular value kernels. For the length division design, we selects the minimal execution time corresponding to red point curve in figure 3. For the number division design, we selects the minimal number of blocks that is able to allocated, since the execution time does not vary much between the minimal number of blocks and the optimal nubmer of blocks.

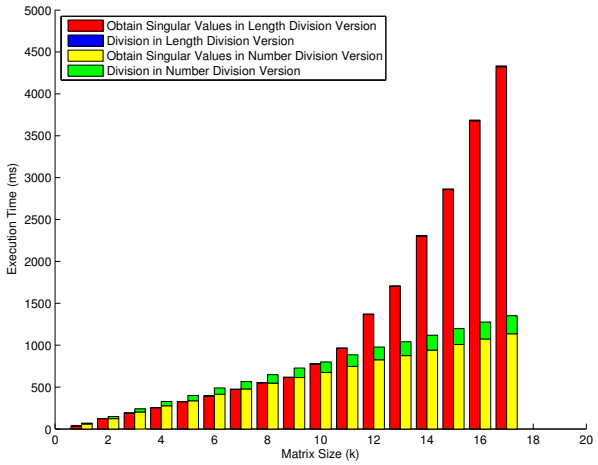


Fig. 5. The optimal block number of different matrix size with single precision on GeForce 750 Ti

Figure 5 shows the execution time of obtaining all singular values of two different implementations on GPU. The left bar is the execution time of length division design, while the right bar is that of number division design. From the figure, we can see that when matrix size is less than $9k$, length division version run a little faster than number division version. When matrix size is larger than $9k$, the execution time of length division version rises quickly, while the execution time of number division version only rises linearly. Thus, when matrix size becomes larger than $9k$, it is better to select number division kernel.

Also, figure 5 represents the average execution time on each GPU kernels of two different designs. When matrix size is smaller than $9k$, the difference between two designs is only

on the interval division method. The division time in length division version is negligible since it is two small, while the division time in number division version increases. The execution time on obtaining singular values in both version are almost the same when matrix size is less than $9k$.

3) Profiling Analysis on Different GPUs: **I want to analyze the bound**

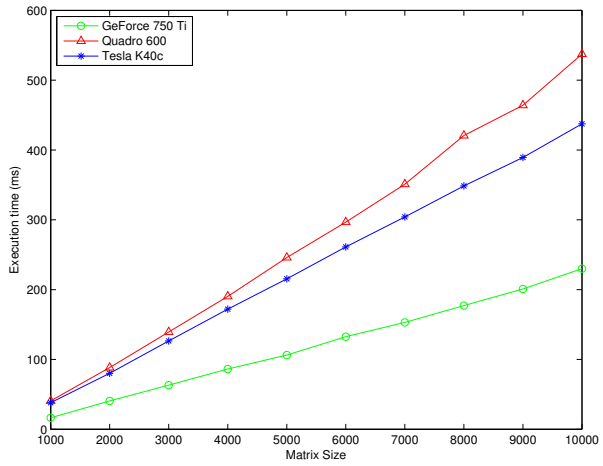


Fig. 6. Execution time of number-division singular value kernel on different GPUs with single-float precision

Figure 6 shows the execution time of singular value kernel on different GPUs.

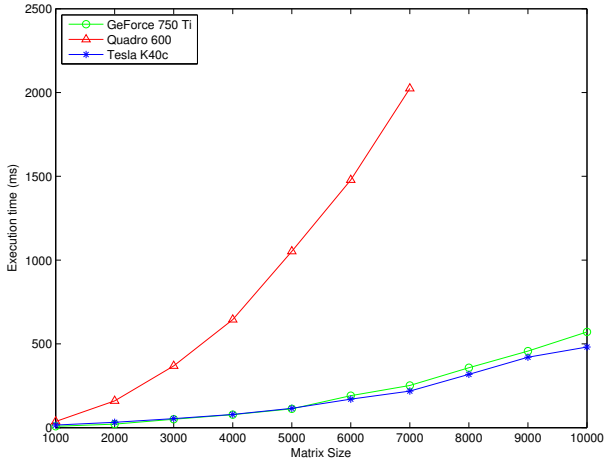


Fig. 7. Execution time of singular value kernel on different GPUs with single-float precision

Figure 7 shows the execution time of singular vector kernel on different GPUs.

4) Tolerance in Bisection Algorithm: Since the bisection algorithm is an approximate algorithm to calculate the singular values, we should test the effect of different error tolerance. The error tolerance err means that the error between the singular values of our algorithm and the actual singular values are less than err. It determines the accuracy of singular value and therefore the orthogonality of singular vectors. As we know, the more accuracy of singular values are, the more execution time should be spent. However, it is important to know the incremental execution time to determine which error tolerance is suitable for different applications. We test our algorithm on different error tolerance. The error tolerance is between 10^{-5} to 10^{-16} with tenfolder decreasing.

I'm still thinking how to draw this figure. Figure 8 shows

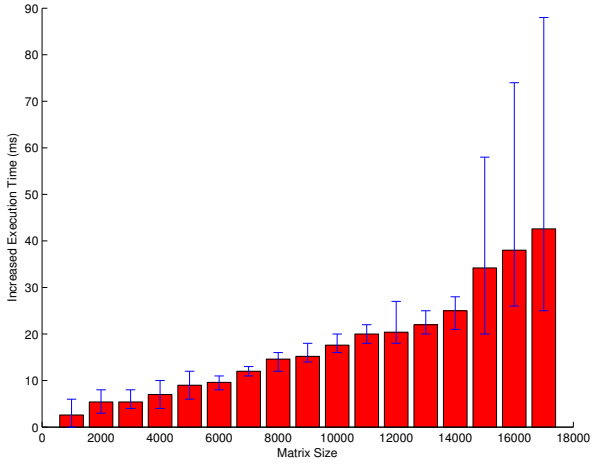


Fig. 8. Average Extra Execution Time When the accuracy increase Performance Comparision

the average increased execution time when the accuracy of singular values goes up a higher level on different matrix size. In other word, it shows the average increased execution time when the error tolerance becomes smaller from 10^{-x} to $10^{-(x+1)}$. From the figure, we can see that when matrix size is smaller than 12000, the additional execution time is only less than $20ms$ when the error tolerance rises a level. When the matrix size is larger than 15000, the additional execution time is a little higher about $40ms$ per level.

C. Pthread CUDA Result

1) Load Balance: Figure 9 shows the total execution time with different load balance on a system of two different GPUs. The horizontal axis represents the percentage of work on Quadro 600. The other percentage of work will be moved to GeForce 750 Ti obviously. The red points represent the minimum values in the blue curves. The blue curves can be separated into two parts by red points. The left ones are dominated by GeForce 750 Ti, while the right ones are contributed by Quadro 600. Since Quadro 600 has a worse performance than GeForce 750 Ti, less work is allocated on it. When around 15% of work is executed on Quadro 600, and 85% of that is processed on GeForce 750 Ti, the total performance of the system will be better than others.

A table to shows the Huge Size Result. This part do not compare to other results.

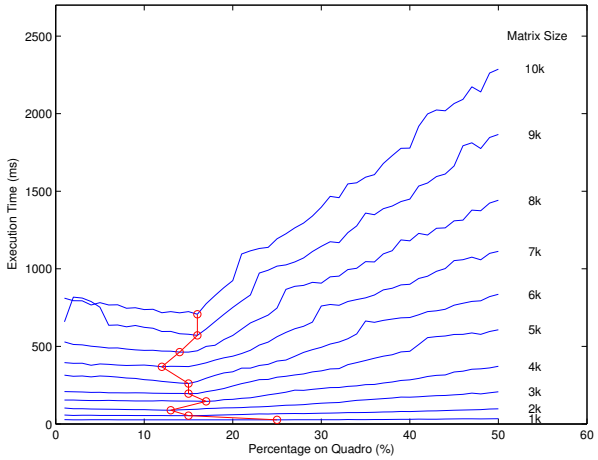


Fig. 9. Load Balance on GeForce and Quadro

TABLE II
PERFORMANCE OF HUGE SIZE MATRIX

Matrix Size	GeForce	Quadro	GeForce + Quadro
10K*10K	0.8s	4.6s	0.7s (85%) / 0.7s (15%)
20K*20K	2.9s	18s	2.6s (85%) / 2.7s (15%)
30K*30K	6.7s	37s	5.9s (85%) / 5.6s (15%)
40K*40K	12s	63s	10s (85%) / 9.5s (15%)
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	-	-

VI. CONCLUSION

ACKNOWLEDGMENT

The authors would like to thank... more thanks here

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.

TABLE III
PERFORMANCE OF HUGE SIZE MATRIX WITH DOUBLE FLOATING-POINT ON TESLA

Matrix Size	Tesla	Tesla (50%) + Tesls (50%)
10K*10K	s	1.2s / 0.6s
20K*20K	s	4.9s / 3.6s
30K*30K	s	12s / 9.3s
40K*40K	s	21s / 18s
50K*50K	s	41s / 36s
80K*80K	s	103s / 81s
100K*100K	s	190s / 154s
120K*120K	s	286s / 243s
150K*150K	s	469s / 399s
180K*180K	s	-
200K*200K	s	-

[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] M. R. Hestenes, *Inversion of matrices by biorthonalization and related results*, J. Soc. Indust. Appl. Math., 6 (1958), pp. 5190.

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

[15] G. Golub and W. Kanhan.

[16] Calculating the singular values and pseudo-inverse of a matrix. SIAM J. Num. Anal. (Series B), 1965.

[17] 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.