Takagi Factorization on GPU using CUDA

Gagandeep S. Sachdev, Vishay Vanjani and Mary W. Hall
School of Computing, University of Utah, UT, 84102
Email: {singhs,vishayv,mhall}@cs.utah.edu

***Abstract*—Takagi factorization or symmetric singular value decomposition is a special form of SVD applicable to symmetric complex matrices. The computation takes advantage of symmetry to reduce computation and storage requirements. The Jacobi method with chess tournament ordering was used to perform the computation in parallel on a GPU using the CUDA programming model. We were able to achieve speedups of over 11x and 7x over CPU serial and Pthreads implementations, respectively, for matrix sizes greater than 512×512.**

I. Introduction

Takagi factorization or symmetric singular value decomposition (SSVD) is a special form of singular value decomposition (SVD) applicable to complex symmetric matrices. An advantage of this form is that it accounts for symmetry and thus saves computation and storage. Takagi factorization has applications in the Grunsky inequalities, computation of the near-best uniform polynomial, rational approximation of a high degree polynomial on a disk, complex independent component analysis problems [2], complex coordinate rotation method, optical potential method [1], nuclear magnetic resonance [3] and diagonalization of mass matrices of Majorana fermions [4].

Takagi factorization for a complex symmetric matrix A is defined as:

$$A = USU^T$$

U is a complex unitary matrix and its columns are called Takagi vectors of A. S is a diagonal singular value matrix of Takagi values where each value corresponds to a Takagi vector from U. On the other hand, in SVD, the third matrix is V^* and not U^T. This means that it is a conjugate transpose of a different matrix V. Given the SVD result, we can generate Takagi vectors by modifying the singular vectors corresponding to non-zero singular values. This methodology is explained in [3]. The operation of computing SVD and then generating Takagi vectors is however inefficient because two matrices of singular vectors have to be computed rather than one.

There exist a variety of algorithms for performing SSVD namely: Jacobi[1], Divide & Conquer [3] and Twisted factorization method[2]. An analysis of the amount of data parallel computations in each of these algorithms was done. Both of the latter two have a complexity of n^2 and are comprised of two stages of tridiagonalization via Householder transforms followed by diagonalization by the respective algorithms. These two algorithms are not as parallelizable as the Jacobi algorithm. The Jacobi method with a complexity of n^3 is not suited for sequential implementation but proves to be the best method for parallel execution. It is also the most numerically stable method of computing Takagi factorization [8].

In this paper, we attempt to accelerate Symmetric SVD for dense matrices using GPUs and the CUDA programming model. The Jacobi method with chess tournament ordering was used to accomplish this. We also parallelized the serial Jacobi implementation [4] using Pthreads for comparison against the GPU version. We were able to achieve a speedup of more than 11x and 7x over serial CPU and Pthreads implementations, respectively, for matrix sizes greater than 512×512. The rest of the paper is organized as follows. Section II describes related work. Section III describes the Jacobi eigenvalue algorithm. Section IV explains the GPU mapping and optimizations. Section V presents experimental results and Section VI concludes the paper.

II. Related work

The concept of porting general purpose computing applications including matrix decompositions on GPUs is not new. Fast SVD was implemented [6] on a 7900 GTK, a generation before CUDA enabled GPUs. Recently, Lahabar and Narayanan [5] implemented SVD on a GPU using CUDA. It was done by bidiagonalization using a series of Householder transforms followed by diagonalization using an implicitly shifted QR algorithm. It is an O (mn^2) algorithm and implemented only for real matrices. As earlier stated, additional computations are needed to get Takagi vectors from unitary matrices generated by SVD and real matrices can generate complex Takagi vectors too. This suggests that there is a need for an implementation that accelerates the symmetric SVD of complex symmetric matrices.

A CPU implementation of Takagi factorization by the Jacobi method has been done by Hahn [4]. This is the serial version implementation of the methods provided in [1]. The paper [1] however also presents a mutithreaded version of the Jacobi algorithm. Qiao [7] gives a Matlab implementation of Takagi factorization using Lanczos tridiagonalization methods and Factorization of symmetric tridiagonal matrix using implicit QR, divide and conquer [3], and the twisted factorization [2] method.

III. Jacobi method

The Jacobi method is an algorithm to find eigenvectors and eigenvalues of a symmetric matrix. It is a stable method and has been proved to be the most accurate [8] when compared to other algorithms which involve tridiagonalizing the matrix. Also,the Jacobi method guarantees a solution. Complexity of this algorithm is of the order n^3 and therefore only suited for small and medium sized matrices. To calculate the Takagi vectors and Takagi values using the Jacobi method, we start

with a complex symmetric matrix A and perform sweeps until convergence is reached. Convergence is checked by calculating the sum of all off-diagonal elements at the end of each sweep. At convergence, off-diagonal elements are close to zero and the eigen values and vectors are returned in the S and U matrix respectively. This algorithm is quadratically convergent; so in most cases the convergence happens within 30 sweeps.

Each sweep involves iterating over all possible pairs of rows and columns. Therefore, there are ${}^{N}C_2$ iterations in each sweep. Each iteration involves applying a transformation to two rows and two columns. For example, if (i,j) is the unique pair of indices for the current iteration, then we update the i^{th} row and j^{th} row followed by i^{th} column and j^{th} column. For each iteration a 2×2 complex symmetric sub-matrix is created and diagonalized by applying the Jacobi rotation such that:

$$\begin{bmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{bmatrix} = V(i,j) \begin{bmatrix} d_i & 0 \\ 0 & d_j \end{bmatrix} V^T(i,j)$$

This is followed by applying the V(i,j) transform to all row elements. This stage is called the row update and is mathematically represented as:

$$\begin{bmatrix} A_{i,1:n} \\ A_{j,1:n} \end{bmatrix} \leftarrow V^H(i,j) \begin{bmatrix} A_{i,1:n} \\ A_{j,1:n} \end{bmatrix}$$

After the completion of row updates, columns are updated in the following way:

$$\begin{bmatrix} A_{1:n,i} & A_{1:n,j} \end{bmatrix} \leftarrow \begin{bmatrix} A_{1:n,i} & A_{1:n,j} \end{bmatrix} \overline{V}(i,j)$$

Serial implementation of Takagi factorization using Jacobi is done using the cyclic ordering [1]. As the name suggests, the first index is incrementally selected and the second index is selected by looping over all indices greater than the first index.

IV. GPU Implementation

The order of selection of index pairs in a Jacobi sweep does not alter the results as each operation is a unitary transform. This implies that pairs that do not share a common index can be computed in parallel. For example, in two subsequent pairs (1,2) and (1,3), which share a common index 1, the Jacobi update (1,3) cannot start until the (1,2) update finishes. However, a similar pair (3,4) can be computed in parallel with (1,2). So, given a symmetric matrix of dimension n and n being even, n/2 Jacobi updates can be carried out in parallel. In the CUDA programming model, this implies that there will be n/2 threads across blocks. We will however need a mechanism of generating n/2 unique pairs over n-1 times such that all possible (${}^{N}C_2$) combinations of indices are covered. [1] describes chess tournament ordering for this purpose. It was modified as in Fig.1 to make programming of this logic easier. So, in a matrix of dimension ten, the first set of pairs will be (0,9), (1,8), (2,7), (3,6), (4,5). The next set of pairs will be (1,9), (2,0), (3,8), (4,7), (5,6) and so on. These indices can be generated trivially using algorithm 1.

Each thread will diagonalize a 2×2 matrix and apply the transform to two rows and two columns. To make the Jacobi updates completely independent and parallelizable, we now need to store the full matrix A rather than storing just the upper half of the symmetric matrix. This is because in case of half storage, a column's lower half will continue into a row, which in turn will be part of some other pair's column. Also we will need to globally synchronize between the row and column updates. This is because one pair's rows can be another pair's column and would lead to race conditions if not made mutually exclusive. On the GPU, we do this through host synchronization [9] i.e. letting all threads finish and exiting out of the GPU kernel. In order to minimize copying of data between GPU and CPU, many auxiliary kernels are used to perform tasks like initialization of Takagi values and vectors on the GPU itself, calculation of threshold, and making diagonal elements non-negative. The time taken for computation is large enough to cover the overhead of multiple kernel calls. Algorithm 2 explains the GPU kernels and their use. Storage space taken in the GPU was $(4n^2+9n) \times 4$ bytes.

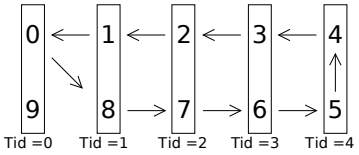

Fig. 1. Chess tournament ordering

Algorithm 1 CHESS TOURNAMENT ORDERING

Require: Thread ID - Tid, Dimension - N, Iteration - i
1: Index 1 = (Tid+i)%(N-1)
2: **if** (Tid != 0) **then**
3: Index 2 = ((N-Tid)+i)%(N-1)
4: **else**
5: Index 2 = N-1
6: **end if**

The target NVIDIA architecture has a compute capability [9] up to 1.3 and lacks caches to global memory. Memory coalescing [9] is one way of obtaining optimum memory bandwidth to global memory in these architectures. Although structures for complex numbers provide better readability and advantage in cached architectures, it does not provide coalescing if each thread accesses a subsequent structure from an array in memory. Thus, complex numbers were stored in two different arrays of float variables containing the real and imaginary parts. Effects of all optimizations done were verified in the CUDA visual profiler [9]. In our case, the two kernels which diagonalize and perform updates are called multiple times and consume most of the time taken by the overall computation. The common technique of getting chunks of data in shared memory from global memory in a coalesced way and then accessing it such that there are no bank conflicts [9], provided much benefit and was implemented wherever possible. Loop unrolling also improved timing and was done to a limit where registers are not spilled into local memory [9]. However, it was not possible to make all memory accesses in row and column updates coalesced. This is because the unique pair of indices handled by a thread in a given iteration might not be contiguous with adjacent threads.

Algorithm 2 GPU IMPLEMENTATION

```
1: Copy complex Matrix A from host to GPU
2: Initialize auxiliary data structures  real diagonal matrix D
   and complex Takagi vector matrix U (GPU)
3: while (Convergence not reached) do
4:     Calculate threshold
5:     for i = 0 to N − 1 do
6:         Diagonalization and row updates (GPU)
7:         Host synchronization
8:         Column and Unitary matrix updates (GPU)
9:         Host synchronization
10:    end for
11: end while
12: Make diagonal elements non-negative and sort(GPU)
13: Copy U and D from GPU
```

V. RESULTS

In this section, we compare the performance of our GPU implementation with an optimized serial [4] and Pthreads version. All experiments were done on Intel Core i7 - 920 PC and an NVIDIA GTX 260 with CUDA 2.3. NVIDIA GTX 260 has 216 stream processors divided into 27 streaming multiprocessors and a total memory of 896 MB. This GPU's peak double precision performance is 1/8x of its peak single precision performance. Thus we resorted to using single precision numbers on all CPU and GPU implementations. We verify the correctness of GPU implementation by comparing it with the CPU version term by term. Timing results are averaged over ten random dense matrices of single precision numbers for each size. This was done to avoid a particularly good or bad sample. Parallel Takagi factorization using Pthreads was implemented using the methodology described in [1].

Table 1 gives the average execution time of the serial, Pthreads and CUDA implementation of the code. All of the timing measurement includes the time to copy the matrices to and from GPUs. Matlab implementations [7], even with faster algorithms were much slower than the serial C implementation provided by [4] and so were not included in the comparison. As shown in Table 1, the GPU implementation surpasses the CPU serial version at dimension of 64. GPU speedup over serial CPU version increases linearly to a size of 512 and slows down after that. This is depicted in Fig. 2. Our GPU implementation achieved a maximum speedup of 11.69x for matrix size of 2048. Pthreads on the other hand reaches a maximum speedup of 1.64x over the serial CPU implementation of the same matrix size on an Intel four core machine. The GPU implementation was more than 7x faster than this Pthreads version. The serial CPU version of any size bigger than this becomes impractical and takes up lot of processing time.

VI. CONCLUSION

In this paper, we presented implementation of SSVD or Takagi factorization using the Jacobi method on commodity GPUs. It achieves a good speedup over serial and Pthreads CPU implementations. The Jacobi method on GPU proves to be the most stable and accurate method [8] of computing SSVD in parallel. With the advent of NVIDIA Fermi architecture, we expect to see much higher speedup because the few non-coalesced memory reads in our case will now be cached. Also, with a speedup of 8X double precision performance over present architectures, NVIDIA Fermi architecture will make Jacobi coupled with double precision numbers the most accurate and efficient method of computing symmetric SVD of a complex symmetric matrix.

TABLE I
EXECUTION TIME FOR SERIAL CPU, PTHREADS AND GPU VERSION

Size	Serial CPU Execution Time (s)	Pthreads Execution Time (s)	Pthreads Speedup (over CPU)	GPU Execution Time (s)	GPU Speedup (over CPU)
16	0.002	0.015	-	0.007	-
32	0.026	0.093	-	0.031	-
64	0.221	0.536	-	0.117	1.879
128	2.010	3.183	-	0 .512	3.92
256	21.343	23.347	-	2.527	8.45
512	171.683	167.479	1.025	15.497	11.07
1024	1486.223	1121.015	1.325	129.801	11.45
2048	11965.910	7255.588	1.649	1023.602	11.69

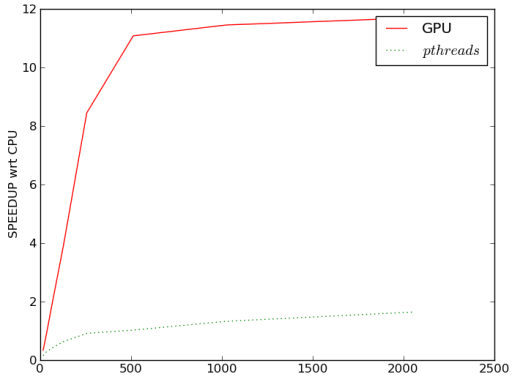

Fig. 2. Speedup of GPU and Pthreads over serial CPU for different matrix sizes

REFERENCES

[1] X. Wang and S. Qiao, *A Parallel Jacobi Method for the Takagi Factorization*. In Proceedings of the international Conference on Parallel and Distributed Processing Techniques and Applications - Volume 1 (June 24 - 27, 2002).

[2] X. Wang and S. Qiao, *A twisted factorization method for symmetric SVD of a complex symmetric tridiagonal matrix*. Technical Report No. CAS 06-01-SQ, McMaster University.

[3] Wei Xu, *Symmetric singular value decomposition of complex symmetric matrices*. ETD Collection for McMaster University (January 1, 2007).

[4] T. Hahn, *Routines for the diagonalization of complex matrices*. [physics/0607103].

[5] S. Lahabar and P. J. Narayanan, *Singular value decomposition on GPU using CUDA*. In Proceedings of the 2009 IEEE international Symposium on Parallel & Distributed Processing (May 23 - 29, 2009).

[6] V. Bondhugula, N. Govindaraju and D. Manocha , *Fast SVD on Graphic Processors*. http://gamma.cs.unc.edu/NUMERIC/SVD/.

[7] S. Qiao, *Fast SVD of Hankel/Toelpitz and Takagi Factorization (Matlab)*. http://www.cas.mcmaster.ca/ qiao/software/takagi/matlab/readme.html.

[8] Demmel, James; Veselic, Kresimir, *Jacobi's method is more accurate than QR*. SIAM J. Matrix Anal. Appl. 13 (1992), no. 4, 1204–1245.

[9] David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*,1st ed. Morgan Kaufmann, February 2010.