2012 IEEE Seventh International Conference on Networking, Architecture, and Storage

Implementing the Jacobi Algorithm for Solving Eigenvalues of Symmetric Matrices with CUDA

Tao Wang[1], Longjiang Guo[1,2†], Guilin Li[3], Jinbao Li[1,2], Renda Wang[1], Meirui Ren[1,2], Jing (Selena) He[4]
[1]School of Computer Science and Technology, Heilongjiang University, Harbin, 150080, China
[2]Key Laboratory of Database and Parallel Computing of Heilongjiang Province, Harbin 150080, China
[3]Software School of Xiamen University, Xiamen, Fujian 361005, China
[4]Department of Computer Science, Kennesaw State University, Kennesaw, GA, 30144, USA
Email: longjiangguo@gmail.com

***Abstract*—Solving the eigenvalues of matrices is an open problem which is often related to scientific computation. With the increasing of the order of matrices, traditional sequential algorithms are unable to meet the needs for the calculation time. Although people can use cluster systems in a short time to solve the eigenvalues of large-scale matrices, it will bring an increase in equipment costs and power consumption. This paper proposes a parallel algorithm named Jacobi_on_gpu which is implemented by CUDA (Computer Unified Device Architecture) on GPU (Graphic Process Unit) to solve the eigenvalues of symmetric matrices. In our experimental environment, we have Intel Core i5-760 quad-core CPU, NVIDIA GeForce GTX460 card, and Win7 64-bit operating system. When the size of matrix is 10240×10240, the number of iterations is 10000 times, the speedup ratio is 13.71. As the size of matrices increase, the speedup ratio increases correspondingly. Moreover, as the number of iterations increases, the speedup ratio is very stable. When the size of matrix is 8192×8192, the number of iterations are 1000, 2000, 4000, 8000 and 16000 respectively, the standard deviation of the speedup ratio is 0.1161. The experimental results show that the Jacobi_on_gpu algorithm can save more running time than traditional sequential algorithms and the speedup ratio is 3.02∼13.71. Therefore, the computing time of traditional sequential algorithms to solve the eigenvalues of matrices is reduced significantly.**

***Keywords*-Matrix eigenvalue; CUDA; Jacobi iteration; GPU; Symmetric Matrix;**

I. INTRODUCTION

Solving the eigenvalues of matrices is one of the most common and important operation in linear algebra. It has wide applications in scientific computing [1]. Solving the eigenvalues of matrices not only can directly help to solve the nonlinear programming, optimization, ordinary differential equations, and a variety of math problems, but also plays an important role in structural mechanics, engineering design, computational physics, and quantum mechanics. Nowadays, because of its important and wide applications, the matrix eigenvalue problem becomes one of the main computational tasks of high-performance computers. However, with the development of advanced technologies, and the improvement of computing power, the size of matrices which people deal with has been increased dramatically. For example, in the field of computational fluid dynamics, statistical, structural engineering, quantum physics, chemical engineering, economic models, the aerospace industry, hydropower, weather forecasting, integrated circuit simulation, signal processing and control, network queuing simulation of Markov chains, and many other fields , it is needed to solve the eigenvalues with large-scale matrices. For some practical problems, the orders of matrices often reach thousands, tens of thousands, or even millions [2].

However, the majority of methods for solving the eigenvalues of matrices adopts the sequential algorithm, or parallel methods on cluster systems. The speed of sequential solutions is undoubtedly very slow for large matrices. The speed of cluster systems is much faster than the sequential methods in solving the eigenvalues of matrices. Nevertheless, using cluster systems brings an increase in equipment costs and power consumptions. Thus, people need to pay more expensive computational costs. For example, a cluster system consists of eight computers. The total cost includes the costs of the eight stand-alone machines, the costs of the connections among these eight machines, and the high maintenance costs. Additionally, the power consumption is eight times or more than using only one machine. The costs of using cluster systems usually range from tens of thousands to hundreds of thousands of dollars (*e.g.*, the Huaan-RAC cluster costs 185,000 RMB in 2011), and their power consumptions are very large. This expensive cost for accelerating computing ability is not affordable for some people.

In recent years, GPU is already famous for the programming capabilities for the large-scale fast calculations. The CUDA technology proposed by NVIDIA is an outstanding representative of this area [3]. CUDA programming gives people a new idea, and also provides us a new way to solve the eigenvalues of matrices. In recent years, the development of CUDA has become a focus on academic researches. People not only use CUDA to deal with graphics and images, but also use it to handle numerical calculations.

[1]†To whom correspondences should be addressed. Email: longjiangguo@gmail.com

978-0-7695-4722-0/12 $26.00 © 2012 IEEE
DOI 10.1109/NAS.2012.12

For example, R. R. Amossen et al. presented a novel data layout, BATMAP, which is particularly well suitable for parallel processing. Moreover BATMAP is compact even for sparse data [4]. Grand et al. proposed a broad-phase collision detection with CUDA [5]. A. A. Aqrawi et al. presented a method using compression for large seismic data sets on modern GPUs and CPUs [6]. Later, A. A. Aqrawi et al. presented the 3D convolution for large data sets on modern GPUs [7]. J. Barnat et al. designed a new CUDA-aware procedure for pivot selection and implemented parallel algorithms using CUDA accelerated computation [8]. A. Hagiescu et al. described an automated compilation flow that maps most stream processing applications onto GPUs by taking two important architectural features of NVIDIA GPUs into consideration , namely, interleaved execution, as well as the small amount of shared memory available in each streaming multiprocessor [9].

This paper proposes a parallel algorithm named Jacobi_on_gpu which is implemented by CUDA on GPU to solve the eigenvalues of symmetric matrices. In our experimental environment, we have Intel Core i5-760 quad-core CPU, NVIDIA GeForce GTX460 card, and Win7 64-bit operating system. When the size of matrix is 10240×10240, the number of iterations is 10000 times, the speedup ratio is 13.71. As the size of matrices increases, the speedup ratio increases. As the number of iterations increases, the speedup ratio is very stable. When the size of matrix is 8192×8192, the number of iterations are 1000, 2000, 4000, 8000 and 16000 respectively, the standard deviation of the speedup ratio is 0.1161. The experimental results show that the Jacobi_on_gpu algorithm can save more running time than traditional sequential algorithms. Moveover the speedup ratio is increased from 3.02 to 13.71. Therefore, the computation time is reduced significantly compared with traditional sequential algorithms.

The contributions of this paper are summarized as follows:

- To the best of our knowledge, this paper firstly proposes a parallel algorithm to solve the eigenvalues of symmetric matrices with CUDA on GPU.
- The paper implements the proposed parallel algorithm on Intel Core i5-760 quad-core CPU, NVIDIA GeForce GTX460 card, and Win7 64-bit operating system. The speedup ratio is $3.02\sim13.71$.
- The theoretical analysis show that the time complexity of the parallel algorithm is $\mathrm{O}(n)$.

The rest of the paper is organized as follows: Section II presents related work. Section III gives an overview of CUDA. In Section IV, we first describes the sequential Jacobi algorithm to solve the eigenvalues of symmetric matrices. Then we presents the proposed Jacobi_on_gpu algorithm. Finally, we gives the time complexity analysis. Section V shows the experimental results. Section VI summarizes the paper.

II. Relate Work

There are many algorithms proposed to solve the eigenvalues of matrices, such as, Jacobi iteration method, QR method, and Power method [10]. However, the majority of these algorithms are implemented sequentially. The speed of sequential solutions undoubtedly is very slow for large matrices.

B. Butrylo et al. presented a parallel algorithm called SSOR which preconditioning implemented on dynamic SMP clusters with communication on the fly [11]. In this paper, they use SMP cluster system to improve the efficiency of solving the matrices eigenvalues. T. Auckenthaler et al. presented a parallel solution of partial symmetric eigenvalue problems, by using the parallel computer system to solve the eigenvalues of matrices [12]. Cluster systems can be used to solve the eigenvalues of large-scale matrices in a short period of time. However, it brings an increasement in equipment costs and power consumptions.

J. Xia et al. presented a GPGPU (General Purpose computing on GPU) implementation for solving the eigenvalues of matrices [13]. In the paper, they presented a power method for solving the maximum eigenvalue of matrices, and QR method for solving all eigenvalues of matrices based on OpenGL (which is a professional graphics interface). Programmers must be very familiar with their proposed parallel algorithms and fully grasp the programming interface of the graphic hardware to implement the approach. As the implementation is difficult, this GPU development approach has not been widely applied to various fields [14]. Since the architecture differences between OpenGL and CUDA, it is very difficult to implement the algorithm [13] on CUDA platform. Their speedup ratio is $2.7\sim7.6$. The speedup ratio 7.6 is reached when using the power method for solving the maximum eigenvalue of the matrix with the size of 2048×2048, while the speedup ratio is 2.7 when using the QR method for solving all eigenvalues of the matrix with the size of 448×448. However, the algorithm Jacobi_on_gpu presented in our paper is based on CUDA, which is widely used in many fields. In addition, our proposed algorithm solves all eigenvalues of matrices with the speedup ratio of $3.02\sim13.71$. The speedup ratio 3.02 is reached when the size of the matrix is 1024×1024, while the speedup ratio 13.71 is reached when the size of the matrix is 10240×10240.

The graphic hardware can be easily applied because of its programmability and parallelism to implement the fast general-purpose computing of some complex models. It has become one of today's research focus. Because of GPU's excellent floating-point calculation capabilities, large memory bandwidth, and relatively low price, GPGPU plays a very important role in many applications fields [15]. Hence, in this paper we proposes a CUDA parallel implementation for the Jacobi algorithm to solve the eigenvalues of symmetric matrices. This provides people a new way when solving the

eigenvalues of symmetric matrices.

III. An overview of CUDA

Traditionally, GPU is only responsible for graphics rendering. Most of other processing are handed over to CPU. Hence there are plenty of wastes GPU's computing resources. GPU has obvious advantages over CPU in processing ability and memory bandwidth. Moreover, GPU does not need to pay too much effort in computing costs and power consumptions. Currently, the single precision floating point processing capacity of mainstream GPU has reached about 10 times more than the same period CPU.

NVIDIA had officially released CUDA in 2007. It is the first development environment and software system which using C-like language and do not need to use graphics API. Compared with the traditional GPU general computing development, CUDA has a very significant improvement. Moreover, since CUDA using C-like language, it can be quickly accepted and mastered by people. After several years of development, CUDA and CUDA-enabled GPU performances have been significantly improved, as well as the function.

Program codes developed on CUDA can be divided into two parts in actual execution. One is the host code which runs on CPU, and the other is the device code which runs on GPU. A parallel program that runs on GPU is called a kernel. The CUDA thread structure is shown in Figure 1. Threads of an executing kernel are organized as blocks, and blocks are organized as grids. A block is the execution unit of a kernel. A grid is a collection of blocks which can be executed in parallel. All blocks are executed in parallel. There is no communication and execution order among blocks. Within a block, all threads are also executed in parallel. The same kernel program can be executed in parallel by all the threads of the blocks which are contained in the same grid. Threads in the same block communicate with each other by using the shared memory and are synchronized by using the _syncthreads() function. This is the two-level block-thread parallel execution model of CUDA.

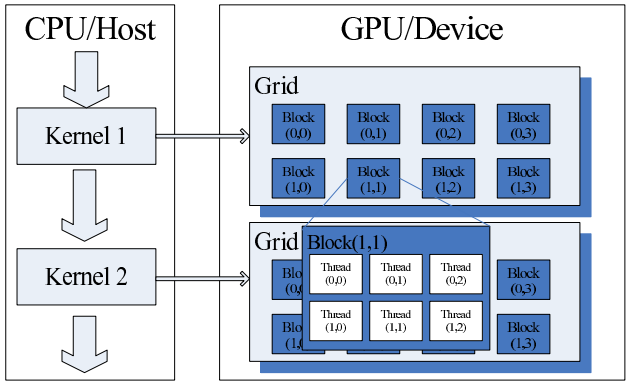

Figure 1. CUDA Thread Structure.

The memory space of CUDA is divided into register, local memory, shared memory, global memory, constant memory, and texture memory. It is shown in Figure 2. Each thread has its own memory, *i.e.*, registers and local memory, which can be read and written directly. Each block has a shared memory, which can be read and written by the threads in the same block. All the threads in the same grid can access the same global memory. In addition, there is ROM accessible by all threads, *i.e.*, constant memory and texture memory, which can help to do the optimization for different applications.

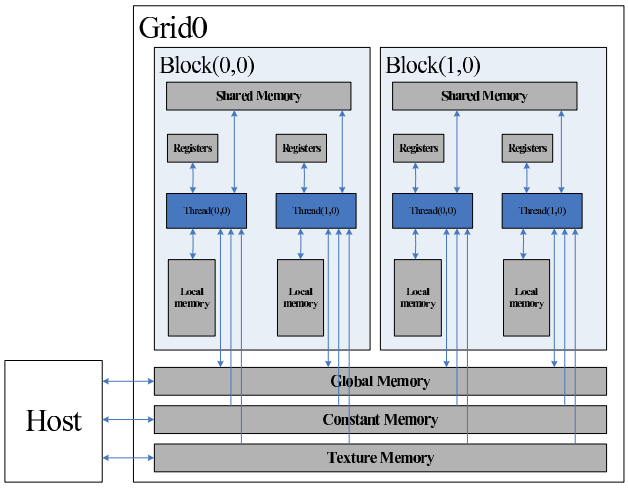

Figure 2. CUDA Memory Model.

IV. Algorithm design

A. Mathematical Basis

Let A be a complex square matrix. λ is a complex number. X is a non-zero complex column vector satisfying $AX = \lambda X$. X denotes an eigenvector of A, while λ is called an eigenvalue of A. X is also called the eigenvector corresponding to the eigenvalue λ.

As noted above, if λ is an eigenvalue of an $n \times n$ matrix A, with the corresponding eigenvector X, then from $(A - \lambda I_n)X = 0$, where $X \neq 0$, and $det(A - \lambda I_n) = 0$, we know that there are at most n distinct eigenvalues of A.

Conversely, if $det(A - \lambda I_n) = 0$, then $(A - \lambda I_n)X = 0$ has a non-trivial solution X. λ is an eigenvalue of A with X as a corresponding eigenvector.

Jacobi iteration method can only solve the eigenvalues of symmetric matrices. If people want to solve the eigenvalues of general matrices, there is a need to use the QR method and other methods [10].

This paper focuses on symmetric matrices.

B. Sequential Jacobi Algorithm

Suppose matrix A is an n-order symmetric matrix. The element which has the maximum absolute value among the

non-diagonal matrix elements of a n-order symmetric matrix A is denoted as a_{pq}, where (p, q) is the position of a_{pq} in A. Without loss of generality, this paper assumes that $p < q$.

We use the plane rotation transformation matrix $R(p, q, \theta)$ to do the orthogonal similarity transformation for matrix A. The transformation is described as Equation (1):

$$A_1 = R(p, q, \theta)^T A R(p, q, \theta) \tag{1}$$

Matrix $R(p, q, \theta)$ is given as follows:

$$R(p,q,\theta) = \begin{array}{c} \\ \\ \\ p \\ \\ \\ \\ q \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccccc} & & & p & & & & q & & & \end{array} \\ \begin{pmatrix} 1 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \cos\theta & 0 & \cdots & 0 & -\sin\theta & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \sin\theta & 0 & \cdots & 0 & \cos\theta & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \end{array}$$

Figure 3. Matrix $R(p, q, \theta)$.

Here, the angle θ is determined by Equation (2),

$$tan2\theta = \frac{2a_{pq}}{a_{pp} - a_{qq}} \tag{2}$$

All the elements in $R(p, q, \theta)$ are summarized as follows:

$$\begin{cases} r_{pp} = \cos\theta;\ r_{qq} = \cos\theta \\ r_{pq} = -\sin\theta;\ r_{qp} = \sin\theta \\ r_{ii} = 1\ (i \neq p, q;\ i = 0, 1, 2 \cdots, n-1) \\ r_{ij} = 0\ (i, j \neq p, q;\ i, j = 0, 1, 2 \cdots, n-1) \end{cases} \tag{3}$$

After the above transformation, the non-diagonal elements' quadratic sum of the symmetric matrix A will reduce $2a_{pq}^2$. Its diagonal elements' quadratic sum will increase $2a_{pq}^2$, while the quadratic sum of all other elements in the matrix will remain unchanged. It can be seen that the non-diagonal elements' quadratic sum of the symmetric matrix A approaches to zero with every transformation. Therefore, when the above transformation is repeated, the matrix A can be gradually changed to a diagonal matrix. The diagonal elements $\lambda_0, \lambda_1, \cdots, \lambda_{n-1}$ in the diagonal matrix are the eigenvalues. In summary, the steps of using sequential Jacobi algorithm to solve the eigenvalues of n-order symmetric matrix A are given as follows:

(1) $i \leftarrow 1$; $A_0 \leftarrow A$.

(2) Select a maximum absolute value element among the non-diagonal matrix elements of a n-order symmetric matrix A_{i-1} denoted by a_{pq}.

(3) If $|a_{pq}| < \varepsilon$, then the iterative process finishes. Here, ε is a predefined accuracy threshold. At this time the diagonal elements a_{jj} $(j = 0, 1, \ldots, n-1)$ are the eigenvalues λ_j $(j = 0, 1, \ldots, n-1)$. Otherwise, go to step (4).

(4) Do the orthogonal similarity transformation for matrix A_{i-1}, *i.e.*, $A_i \leftarrow R(p, q, \theta)^T A_{i-1} R(p, q, \theta)$. Algorithm 1 performs the orthogonal similarity transformation.

(5) $i \leftarrow i + 1$.

(6) Go to step (2).

Algorithm 1 : Sequential_Matrix_Transformation($a[\ ][\ ]$)

Input: Before transformation matrix A_{i-1} stored in the two dimensional array $a[\ ][\ ]$

Output: Matrix A_i which is the orthogonal similarity transformation of matrix A_{i-1}.

1: $x \leftarrow -a_{pq}^{(i-1)}, y \leftarrow \frac{1}{2}(a_{qq}^{(i-1)} - a_{pp}^{(i-1)})$

2: $\omega \leftarrow sign(y)\frac{x}{\sqrt{x^2 - y^2}}$

3: $\sin\theta \leftarrow \frac{\omega}{\sqrt{2(1+\sqrt{1-\omega^2})}}$

4: $\cos\theta \leftarrow \sqrt{1 - \sin^2\theta}$

5: $a_{pp}^{(i)} \leftarrow a_{pp}^{(i-1)}\cos^2\theta + a_{qq}^{(i-1)}\sin^2\theta + a_{pq}^{(i-1)}\sin 2\theta$

6: $a_{qq}^{(i)} \leftarrow a_{pp}^{(i-1)}\sin^2\theta - a_{qq}^{(i-1)}\cos^2\theta - a_{pq}^{(i-1)}\sin 2\theta$

7: $a_{pq}^{(i)} \leftarrow 0$; $a_{qp}^{(i)} \leftarrow 0$

8: $a_{pj}^{(i)} \leftarrow a_{pj}^{(i-1)}\cos\theta + a_{qj}^{(i-1)}\sin\theta (j \neq p, q)$

9: $a_{qj}^{(i)} \leftarrow -a_{pj}^{(i-1)}\sin\theta + a_{qj}^{(i-1)}\cos\theta (j \neq p, q)$

10: $a_{kp}^{(i)} \leftarrow a_{kp}^{(i-1)}\cos\theta + a_{kq}^{(i-1)}\sin\theta (k \neq p, q)$

11: $a_{kq}^{(i)} \leftarrow -a_{kp}^{(i-1)}\sin\theta + a_{kq}^{(i-1)}\cos\theta (k \neq p, q)$

The rest elements of the matrix which are not given in Algorithm 1 don't need to be updated, so these elements just stay the same. In Algorithm 1, the $sign(y)$ is given as follows:

$$sign(y) = \begin{cases} 1\ (y \geq 0) \\ -1\ (y < 0) \end{cases} \tag{4}$$

C. Parallel Jacobi Algorithm with CUDA on GPU

1) Preparation for the Algorithm: An absolute maximum element a_{pq} is selected from the non-diagonal matrix of a n-order symmetric matrix A in the second step. The elements of the transformed matrix which are computed in the fourth step can be solved by using GPU's multi-thread parallel implementation. The parallelization of the two steps can be solved by kernel functions. As starting a kernel function for many times incurs a lot of time overhead, there is a need

to design a parallel algorithm to solve the eigenvalues of matrices in only one kernel function. In addition, to achieve a higher speedup ratio, we should put the iteration into the kernel function and invoke the kernel function only once.

Suppose the size of matrix A is $n \times n$, there is a need to start $n(n-1)/2$ threads. By assigning 512 threads to each BLOCK, the total number of BLOCK is $n(n-1)/(2\times 512)$. Because the second step of the algorithm should seek a maximum absolute value element a_{pq} from the non-diagonal matrix elements of a n-order symmetric matrix A, it is necessary to do some appropriate treatment to save matrix A before starting a kernel function to facilitate the computation. A one-dimensional array a is used to store the elements of matrix A. Row is prior in storage. The first n positions of array a (the array index from 0 to $n-1$) are used to store the diagonal elements of matrix A. Starting from the array index n, the non-diagonal elements of matrix A are stored. Because matrix A is a symmetric matrix, it is only needed to store the elements of the upper triangular part (excluding the diagonal elements). The total number of non-diagonal elements stored in the array a is $n(n-1)/2$. Matrix storage structure is shown in Figure 4:

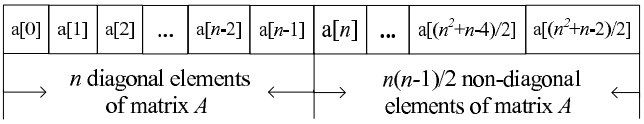

Figure 4. Matrix storage structure.

For example, Figure 5 is a matrix with the size of 4×4. Its row index and column index are both from 0 to 3. According to the above storage structure, we put the four diagonal elements {1, 4, 6, 7} in the first four positions of the array. Then we put the residual elements {2, -9, 5} of row0 in the next three positions. Subsequently we put the residual elements {3, 8} of row1 in the next two positions. At last, we put the residual element {-1} of row2 in the last position. This matrix storage structure is shown in Figure 6.

$$\begin{pmatrix} 1 & 2 & -9 & 5 \\ 2 & 4 & 3 & 8 \\ -9 & 3 & 6 & -1 \\ 5 & 8 & -1 & 7 \end{pmatrix}$$

Figure 5. Example of Matrix.

2) Parallel algorithm for finding the one from non-diagonal elements whose absolute value is the maximum: First of all, we compute the global index of each thread in the kernel function. Second, we use the while loop to

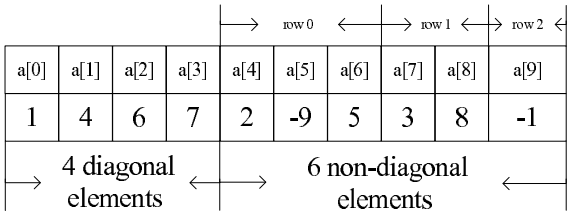

Figure 6. Matrix storage structure of Figure 5.

implement the iteration. In the while loop, we put the last $n(n-1)/2$ elements of array a in another one-dimensional array b, that is to store the non-diagonal elements of matrix A in array b. By using $n(n-1)/2$ threads, each thread can copy an element from array a to array b. The corresponding relationship is given as follows: $b[index] = a[n + index]$. $index$ stands for the global index of threads. After the copy process, threads need to be synchronized. The next step is to find the absolute maximum element of array b, which is the absolute maximum element a_{pq} from matrix A.

Because the total number of matrix A's non-diagonal elements is $n(n-1)/2$, only $n/2$ threads are needed. Each thread is used to find the absolute maximum element from its $n-1$ elements allocated to them. For example, thread 0 deals with $n-1$ elements from $b[0]$ to $b[n-2]$; thread 1 deals with $n-1$ elements from $b[n-1]$ to $b[2n-3]$; $\cdots$, thread j deals with $n-1$ elements from $b[j*n-j]$ to $b[(j+1)*n-(j+2)]$. So there is a need to do a judgement: if the global index of thread is less than $n/2$ (the global index of threads is from 0 to $(n(n-1)/2)-1$, and the total number of threads is $n(n-1)/2$), then let this thread work. Two one-dimensional arrays $maxvalue$ and $rowindex$ are used to store the absolute value of the absolute maximum element of each thread's $n-1$ elements and the index of this element in its $n-1$ elements, respectively ($rowindex$ is in the range from 0 to $n-2$). $maxvalue[j]$ stores the absolute value of the absolute maximum element of thread j's $n-1$ elements from $b[j*n-j]$ to $b[(j+1)*n-(j+2)]$. $rowindex[j]$ stores the index of the absolute maximum element in these $n-1$ elements.

For example, in the matrix of Figure 5, suppose that we start three threads and each thread deals with two elements. $maxvalue[0]$ stores 9 and $rowindex[0]$ store 1. $maxvalue[1]$ stores 5 and $rowindex[1]$ stores 0. $maxvalue[2]$ stores 8 and $rowindex[2]$ stores 0. After finishing the above work, it can obtain $n/2$ elements whose absolute values are maximum. Thread 0 is used to find the one from these $n/2$ elements whose absolute value is the maximum. It can find the absolute maximum element, which is the absolute maximum element in the one-dimensional array $maxvalue$, from these $n/2$ elements by using a for loop. This element is a_{pq} which is what we want to find. The next problem is how to compute the index p, q of this element in matrix A. This job can also be done by thread 0.

Suppose that $maxvalue[i]$ is the absolute maximum element in this array. We need to find the element which is stored in array $rowindex$. First, there is a need to compute the index of this absolute maximum element in array b according to the Equation (5):

$$b\_index = i * (n-1) + rowindex[i] \tag{5}$$

The next problem is that how to compute p and q according to the value of b_index. Because array b only stores the elements of matrix A's upper triangular part (excluding the diagonal elements), matrix A needs to store $n-1$ elements at row 0; $n-2$ elements at row 1; $\cdots$, $n-p-1$ elements at row p. It is obvious that this is a arithmetic progression. Now there is a need to search. To be specific, we are looking for a row, before which the total number of the elements from all rows are less than or equal to $b\_index+1$. Moreover, the total number of the elements from all rows after this row (include this row) are greater than or equal to $b\_index+1$. At this time, this row number p is what we are seeking for. The procedure is shown in Algorithm 2.

Algorithm 2 : Compute_p(b_index, n)

Input: b_index is the index of the element whose absolute value is the maximum among array $b[\ ]$.
n is the order of the matrix.

Output: p which is the row number of element $b[b\_index]$ in symmetric matrix A.
$lownum$ is the total number of the elements from all rows before this row in symmetric matrix A's upper triangular part(excluding the diagonal elements).

1: $low \leftarrow b\_index/n,\ high \leftarrow n-1$
2: **while** $low <= high$ **do**
3: $mid \leftarrow (low+high)/2$
4: $lownum \leftarrow (2*n-mid-1)*mid/2$
5: $highnum \leftarrow (2*n-mid-2)*(mid+1)/2$
6: **if** $(b\_index+1 \geqslant lownum) and (b\_index+1 \leqslant highnum)$ **then**
7: $p \leftarrow mid$
8: break
9: **else if** $b\_index+1 > highnum$ **then**
10: $low \leftarrow mid+1$
11: **else if** $b\_index+1 < lownum$ **then**
12: $high \leftarrow mid-1$
13: **end if**
14: **end while**

After getting the value p, q is very easy to compute. It is shown as Equation (6):

$$q = b\_index + 1 - lownum + p \tag{6}$$

Here, $lownum$ is the total number of the elements from all rows before this row. The reason to add p in Equation (6) is that there are $p+1$ elements at row p which are not be stored in array b. Since q is start from 0, we have $p+1-1=p$.

For example, in the matrix shown in Figure 5, we start three threads and each thread deals with two elements. The absolute value of element -9 is the maximum, and its absolute value is 9. Its index in array b is stored in $rowindex[0]$, and the value of $rowindex[0]$ is 1. Because thread 0 deals with element -9, $i=0$. This is already explained above. Then we have $b\_index = i*(n-1) + rowindex[i] = 0*(4-1)+rowindex[0] = 1$. According to the output value p of algorithm 2, **Compute_p**($rowindex[0]$, 4), we have $p=0$, $lownum$=0. $q = b\_index + 1 - lownum + p$=$rowindex[0]+1-lownum+p$=1+1-0+0=2. $(p,q)=(0,2)$ is the right index of element -9 in the matrix which is shown in Figure 5.

In order to accelerate the search speed, the binary search method is used. Binary search usually searches n elements with time complexity of $O(log(n-1))$. For our algorithm, there is no need to search starting from row 0. Our searching starts from row $\lfloor b\_index/n \rfloor$ with time complexity of $O(log(n-1-\lfloor b\_index/n \rfloor))$. The pseudo-code of the parallel algorithm to find absolute maximum value element from the non-diagonal elements is described in Algorithm 3.

Algorithm 3 : GPU_Maxvalue($b[\ ], n$)

Input: one-dimensional array b, n which is the order of the matrix

Output: fm which is the absolute maximum element in array b and p, q which are index of this element in matrix A

1: $ThreadID \leftarrow blockIdx.x * BLOCK\_SIZE + threadIdx.x$ // $blockIdx.x$ and $threadIdx.x$ are Built-in variables, $BLOCK\_SIZE$ is the number of threads in one BLOCK
2: **if** $ThreadID < n/2$ **then**
3: Each thread is used to find the absolute maximum element from its $n-1$ elements which are allocated to them. Each thread also stores this element and the index of this element in array b into the array $maxvalue$ and $rowindex$, respectively.
4: **end if**
5: **if** $ThreadID == 0$ **then**
6: Finding the absolute maximum element from these $n/2$ elements by a for loop which is the absolute maximum element in one-dimensional array $maxvalue$
7: Computing the index p, q of this element in matrix A by using the binary search method
8: **end if**

3) Parallel algorithm for matrix update: The fourth step of sequential Jacobi algorithm is to update the matrix after the calculation of Equation (1). Using n threads to parallelly

implement the Algorithm 1, each thread needs to compute the new elements $a_{pj}, a_{qj}, a_{kp}, a_{kq}(j, k \neq p, q)$. p and q are the index of a_{pq} which is found in the second step of sequential Jacobi algorithm. j and k are the index of threads, which is from 0 to $n-1$. n threads are used to in the parallel implementation, which can update the matrix within a shorter time than the original n-cycles process. Because it has already stored the matrix elements in a one-dimensional array before starting the kernel function. The first n positions of the array (the array index from 0 to $n-1$) are used to store the matrix's diagonal elements. The non diagonal elements of the matrix are stored from the array index n. So the next problem is how to find the correct position of the elements $a_{pj}, a_{qj}, a_{kp}, a_{kq}(j, k \neq p, q)$ in array a. The details of the storage structure of the matrix is already described in Figure 4. First of all, we need to compute the index of these elements in one-dimensional array when using the normal storage (that is to store all elements of the matrix) which row is preferred. The next step is to compute the corresponding index in the one-dimensional array b according to the above index. Array b stores the elements of the upper triangular part (excluding the diagonal elements) of matrix A. Because the elements $a_{pj}, a_{qj}, a_{kp}, a_{kq}(j, k \neq p, q)$ are non-diagonal elements, array a stores the non-diagonal elements of matrix A starting from the array index n. Finally, the index which is found from array b plus n is what we are seeking for. The details are described in the following Equations:

$$\begin{cases} u = p * n + j \\ u1 = u - \frac{(\lfloor u/n \rfloor + 1)(\lfloor u/n \rfloor + 2)}{2} \\ u2 = n + u1 \end{cases} \tag{7}$$

Equation (7) is used to compute the index of element a_{pj} in array a. Here, u stands for the index of the element in one-dimensional array when using the normal storage (that is to store all elements of the matrix) which row is preferred. $u1$ stands for the corresponding index in the one-dimensional array b. $u2$ stands for the index in array a which we are finally looking for.

$$\begin{cases} w = q * n + j \\ w1 = w - \frac{(\lfloor w/n \rfloor + 1)(\lfloor w/n \rfloor + 2)}{2} \\ w2 = n + w1 \end{cases} \tag{8}$$

Equation (8) is used to compute the index of element a_{qj} in array a. The description of its variables is similar to Equation (7).

$$\begin{cases} z = k * n + p \\ z1 = z - \frac{(\lfloor z/n \rfloor + 1)(\lfloor z/n \rfloor + 2)}{2} \\ z2 = n + z1 \end{cases} \tag{9}$$

Equation (9) is used to compute the index of element a_{kp} in array a. The description of its variables is similar to Equation (7).

$$\begin{cases} v = k * n + q \\ v1 = v - \frac{(\lfloor v/n \rfloor + 1)(\lfloor v/n \rfloor + 2)}{2} \\ v2 = n + v1 \end{cases} \tag{10}$$

Equation (10) is used to compute the index of element a_{kq} in array a. The description of its variable is similar to Equation (7).

The elements $a_{pj}, a_{qj}, a_{kp}, a_{kq}(j, k \neq p, q)$ of the matrix which are $a[u2], a[w2], a[z2], a[v2]$ in array a can be updated by Equation (11).

$$\begin{cases} m = a[u2],\ n = a[w2] \\ x = a[z2],\ y = a[v2] \\ a[u2] = m * \cos\theta + n * \sin\theta \\ a[w2] = -m * \sin\theta + n * \cos\theta \\ a[z2] = x * \cos\theta + y * \sin\theta \\ a[v2] = -x * \sin\theta + y * \cos\theta \end{cases} \tag{11}$$

The pseudo-code of the Jacobi algorithm to solve the eigenvalues of symmetric matrices is described in Algorithm 4:

Algorithm 4 : Kernel_Solving_Eigenvalue($a[\ ], lt, n$)

Input: Before transformation matrix A_{i-1} stored in the array $a[\ ]$, and Iterations lt, and n which is the order of the matrix

Output: After transformation matrix A_i stored in the array $a[\ ]$

1: $ThreadID \leftarrow blockIdx.x * BLOCK\_SIZE + threadIdx.x$ // $blockIdx.x$ and $threadIdx.x$ are Built-in variables, $BLOCK\_SIZE$ is the number of threads in one BLOCK
2: Putting the last $n(n-1)/2$ elements of array a to store in another one-dimensional array b(that is put the non-diagonal elements of matrix A to store in array b);
3: Call GPU_Maxvalue($b[\ ], n$)
4: **if** $ThreadID < n$ **then**
5: Each thread compute the new elements $a[u2], a[w2], a[z2], a[v2]$ using the Equation (11)
6: **end if**

The CUDA Parallel Implementation for the Jacobi Algorithm is described in Algorithm 5:

D. The Time Complexity Analysis

The Time Complexity of Sequential Algorithm: The second step of the sequential algorithm is to select the maximum absolute value element a_{pq} from the non-diagonal matrix elements of a n-order symmetric matrix A. The above process needs dual cycles, with the time complexity of $O(n^2)$. Computing the elements of the transformed matrix in the fourth step costs the most time in doing two single cycles. The time complexity is $O(n)$. The other steps of the sequential algorithm with time complexity of $O(1)$.

Algorithm 5 : Jacobi_on gpu($a[\ ], lt, n$)

Input: The initial matrix A stored in the array $a[\ ]$, and Iterations lt, and n which is the order of the matrix

Output: The eigenvalues of matrix A, that is the first n elements of after transformation array $a[\ ]$

1: Copy the array a from the main memory of the host to the global memory of the device;
2: Transfer the kernel function to start up $n(n-1)/2$ threads;
3: Call kernel function: Kernel_Solving_Eigenvalue($a[\ ], lt, n$)
4: Copy the array a which is after the final calculation and transformation from the global memory of the device to the main memory of the host;

In summary, the total time complexity of the sequential algorithm is $\mathrm{O}(n^2)$.

The Time Complexity of Parallel Algorithm: The parallel algorithm uses $n/2$ threads to find the element who has the maximum absolute value in the non-diagonal elements of matrix A. Each thread is used to find the absolute maximum element from its $n-1$ elements allocated to them, with time complexity of $\mathrm{O}(n-1)$. Then thread 0 is used to find the one from these $n/2$ elements whose absolute value is the maximum by using a loop, with time complexity of $\mathrm{O}(n/2)$. So the time complexity of finding the absolute maximum element a_{pq} from the non-diagonal matrix elements of matrix A is $\mathrm{O}(n-1)$+$\mathrm{O}(n/2)$. The next is the time complexity of using the binary search. This has already been analyzed in Section IV, C, which is $\mathrm{O}(log(n-1-\lfloor b\_index/n \rfloor))$. Moreover. in the parallel algorithm, the matrix transformation is implemented by starting n threads parallelly, which with time complexity of $\mathrm{O}(1)$. Because of the interactions between GPU and CPU, there is a need to copy the data from the main memory of the host to the global memory of the device first, and then copy the data from the global memory of the device to the main memory of the host after the final calculation. So the transmission time is $2t_{transfer}$, and the total time complexity of the parallel algorithm is $\mathrm{O}(n) + 2t_{transfer}$. However, it can be seen that the $t_{transfer}$ is very short from the Figure 11 in Section V. Therefore, the $t_{transfer}$ is neglected. In summary, the time complexity of the parallel algorithm is $\mathrm{O}(n)$.

V. Experimental Results and Analysis

The experimental environment of our work is: Intel Core i5-760 quad-core CPU, NVIDIA GeForce GTX460 graphics card, and operating system is Win7 64-bit systems. In order to use Nsight software to turn off TDR which can solve the problem of run time limit on kernels, we also install another graphics card on the main board. When the number of iterations is large, the kernel function timeout problem occurs. Therefore, there is a need to use Nsight software to solve this problem. NVIDIA Parallel Nsight software is the first industry developed environment for massively parallel computing integrated into Microsoft Visual Studio, which is the world's most popular development environment. Parallel Nsight is a powerful tool that allows programmers to develop for both GPUs and CPUs within Microsoft Visual Studio. This graphics card connects to the monitor for displaying. Moreover, NVIDIA GeForce GTX460 graphics card fully used for the computation. In our experiment, the data of matrices are randomly generated by a random function. In order to test more intuitively and obviously, we control the number of iterations to meet the accuracy of solving the matrices eigenvalues, that is ε which mentioned in Section IV, B.

Our environment of compiling and running is Visual Studio 2010.

In the first experiment, we fix the iterations of the test cases to 10000 and test the running times of the sequential algorithm and our parallel algorithm when the size of matrixes are 1024×1024, 2048×2048, 4096×4096, 6144×6144, 8192×8192 and 10240×10240, respectively. The results are shown in Figure 7. It can be seen that the running times of Parallel Algorithm and Sequential Algorithm increase, as the number of matrix order increases.

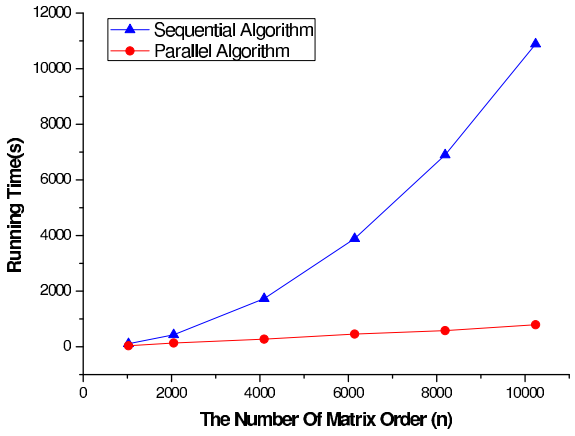

Figure 7. The Number Of Matrix Order and Running Time.

Figure 8 draw out the speedup results according to the running times of the sequential algorithm and the parallel algorithm. It can be seen that as the size of matrix increases, the speedup ratio is increased. It shows that the speedup ratio of our algorithm is larger when the size of matrix is larger.

We also test the running times of the sequential algorithm and the parallel algorithm, with the matrix size of 2048×2048 and 8192×8192, when the number of iterations are 1000, 2000, 4000, 8000 and 16000. The results are shown in Figure 9. It can be seen that the running times of Parallel Algorithm and Sequential Algorithm increase, as the number of iterations increases.

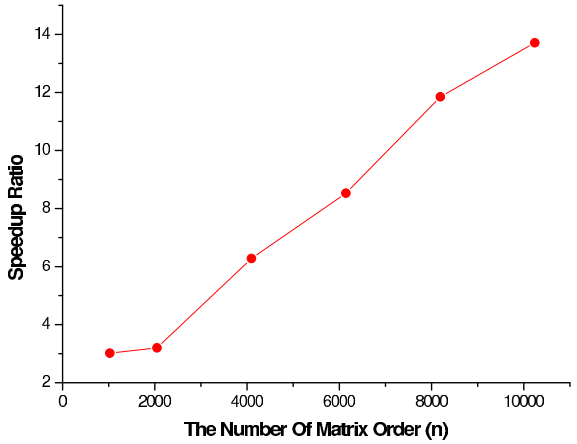

Figure 8. The Number Of Matrix Order and Speedup Ratio.

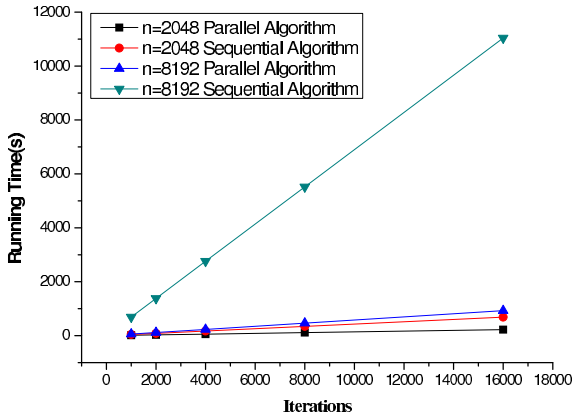

Figure 9. Iterations and Running Time.

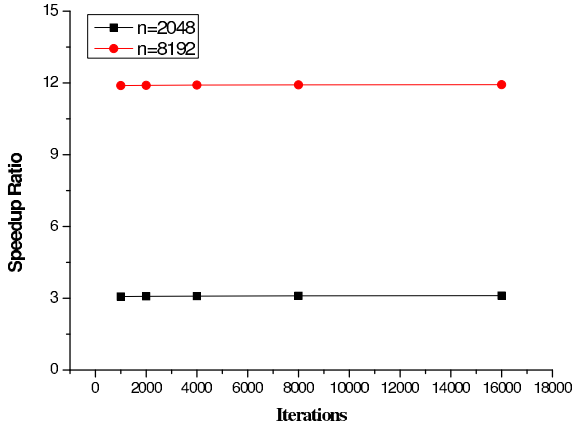

Figure 10. Iterations and Speedup Ratio.

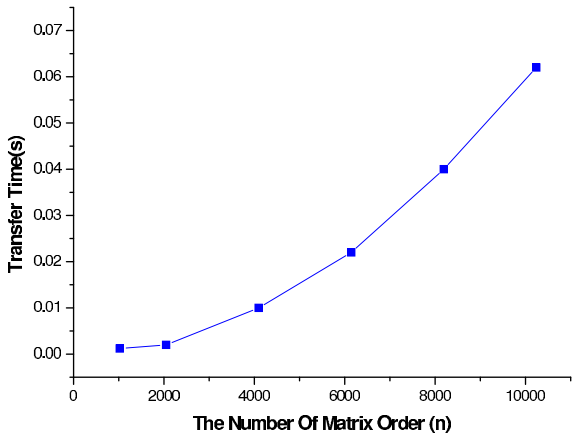

Figure 11. The Number Of Matrix Order and Transfer Time.

Figure 10 draw out the speedup results according to the running times of the sequential algorithm and the parallel algorithm. It can be seen that as the number of iterations increases, the speedup ratio of the proposed algorithm is very stable. When the size of matrix is 8192×8192, the number of iterations are 1000, 2000, 4000, 8000 and 16000, the standard deviation of the speedup ratio on this set of data is 0.1161. It shows that the speedup ratio of the algorithm is very stable when the number of iterations is very large.

$t_{transfer}$ which is the transmission time between CPU and GPU is also tested, which is shown in Figure 11. It can be seen that the $t_{transfer}$ is very short.

Through the results of the above experiments, it can be found that as the size of matrices increases, the speedup ratio also increases. The algorithm and the speedup ratio are very stable, as the number of iterations increases. This shows that our algorithm has good expandability. Our algorithm obtains a certain amount of speedup, and does not add too many computational costs. It solves the problem of the high computational costs of using cluster systems. Furthermore, the speed of traditional sequential algorithms are very slow which are unable to meet the needs of people. Our proposed algorithm overcomes this problem as well.

VI. Conclusion

According to the characteristic of CUDA, this paper proposes a parallel algorithm named Jacobi_on_gpu based on the Jacobi iteration method to solve the eigenvalues of matrices. The proposed algorithm solves the problem of high computational costs of using cluster systems, and the slow speed of traditional sequential algorithms. The new algorithm also provides us a new way when solving the eigenvalues of matrices. In addition, the experimental results show that the speedup ratio of the Jacobi_on_gpu algorithm is 3.02∼13.71, and its time complexity is $O(n)$. As the size of matrices increases, the speedup ratio also increases. Furthermore, as the number of iterations increases, the speedup ratio does not decrease.

The Jacobi iteration method can only solve the eigenvalues of the symmetric matrices. If people want to solve the eigenvalues of the general matrices, there is a need to use QR method and other methods. Therefore, our future work

is that how to use QR method to solve the eigenvalues of the general matrices with CUDA.

ACKNOWLEDGMENT

This work is supported by Program for New Century Excellent Talents in University under grant No.NCET-11-0955, Programs Foundation of Heilongjiang Educational Committee for New Century Excellent Talents in University under grant No.1252-NCET-011, Program for Group of Science and Technology Innovation of Heilongjiang Educational Committee under grant No.2011PYTD002, the Science and Technology Research of Heilongjiang Educational Committee under grant No.12511395, the Science and Technology Innovation Research Project of Harbin for Young Scholar under grant No.2008RFQXG107 and No.2011RFXXG014, the National Natural Science Foundation of China under grant No.61070193, 61100032, 60803015, Heilongjiang Province Founds for Distinguished Young Scientists under Grant No.JC201104, Heilongjiang Province Science and Technique Foundation under Grant No.GC09A109, Basal Research Fund of Xiamen University under Grant No.2010121072.

REFERENCES

[1] Andrea Brini, Marcos Marino, and Sebastien Stevan. The uses of the refined matrix model recursion. Journal of Mathematical Physics, 2011, 52(5): pp. 291-315.

[2] S. L. Foo and P. P. Silvester. Finite Element Analysis of Inductive Strips in Unilateral Finlines. IEEE Microwave Theory and Techniques, 1993, 41(2): pp. 298-304.

[3] NVIDIA. NVIDIA CUDA Programming Guide[EB /OL]. http: / /developer. download. nvidia. com / compute / cuda /2.2 / toolkit/docs/NVIDIA CUDA Programming Guide 2. 2. 1. pdf, 2009 - 5 - 26.

[4] Rasmus Resen Amossen, and Rasmus Pagh. A New Data Layout For Set Intersection on GPUs. IEEE IPDPS, 2011.

[5] S. L. Grand. Broad-phase collision detection with cuda. In GPU Gems 3, 2007.

[6] Ahmed A.Aqrawi, and Anne C.Elster. Accelerating Disk Access Using Compression for Large Seismic Datasets on modern GPU and CPU. Scientific and Parallel Computing, 2010.

[7] A. A. Aqrawi. 3d convolution of large datasets on modern gpus. Norwegian University of Science and Technology, 2009.

[8] J. Barnat, Petr Bauch, L. Brim, and M. Ceska. Computing Strongly Connected Components in Parallel on CUDA. IEEE IPDPS, 2011.

[9] Andrei Hagiescu, Huynh Phung Huynh, Wengfai Wong, and Rick Siow Mong Goh. Automated architecture-aware mapping of streaming applications onto GPUs. IEEE IPDPS, 2011.

[10] David Kincaid, and Ward Cheney. Numerical Analysis Mathematics of Scientific Computing(Third Edition). Thomson Publishers, 2003.

[11] Boguslaw Butrylo, Marek Tudruj, and Lukasz Masko. Parallel SSOR preconditioning implemented on dynamic SMP clusters with communication on the fly. Future Generation Computer Systems, 2010, 26(3): pp. 491-497.

[12] T. Auckenthaler, V. Blum, HJ. Bungartz, T. Huckle, R. Johanni, L. Kramer, B. Lang, H. Lederer, and P.R.WillemsParallel solution of partial symmetric eigenvalue problems from electronic structure calculations. Parallel Computing, 2011, 37 (12): pp. 783-794.

[13] Xiajian-Ming, and WeiDe-Min. GPU Implementation for Solving Eigenvalues of a Matrix. Acta Scientiarum Naturalium Universitatis Sunyatseni, 2008, 47(2): pp. 89-92.

[14] M Snyder. Solving the embedded OpenGL puzzle - making standards, tools, and APIs work together in highly embedded and safety critical environments. IEEE DASC, 2005.

[15] Charl van Deventer, Willem A.Clarke, and Scott Hazelhurst. BOINC and CUDA: Distributed High-Performance Computing for Bioinformatics String Matching Problems. IEEE ACMW, 2006.