GPU-Accelerated Bulk Computation of the Eigenvalue Problem for Many Small Real Non-symmetric Matrices

Hiroki Tokura, Takumi Honda, Yasuaki Ito, and Koji Nakano
Department of Information Engineering,
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 JAPAN

Mitsuya Nishino, Yushiro Hirota, and Masami Saeki
Department of Mechanical System Engineering,
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 JAPAN

***Abstract*—The main contribution of this paper is to present a very efficient GPU implementation of bulk computation of eigenvalues for a large number of small non-symmetric real matrices. This work is motivated by the necessity of such bulk computation in design of control systems, which requires to compute the eigenvalues of hundreds of thousands non-symmetric real matrices of size up to 30×30. In our GPU implementation, we considered programming issues of the GPU architecture including warp divergence, coalesced access of the global memory, bank conflict of the shared memory, etc. In particular, we present three types of assignments of GPU threads to matrices and introduce three memory arrangements in the global memory. The experimental results on NVIDIA GeForce GTX TITAN X show that our GPU implementation for 500000 matrices of size 5×5 to 30×30 attains a speed-up factor of approximately 15 over the CPU implementation on Intel Core i7-4790.**

***Keywords*-GPGPU; CUDA; The eigenvalue problem; Bulk computation; The implicit QR algorithm**

I. Introduction

Given an $n \times n$ matrix A, *the eigenvalue problem* is to find all eigenvalues λ satisfying $A\mathbf{x} = \lambda\mathbf{x}$, where $\mathbf{x}$ is a nonzero vector of size n. The computation of eigenvalues has many applications in the field of science and engineering such as image processing, control engineering, quantum mechanics, economics, among the others.

In control system design, the computation of the eigenvalue problem is widely used, e.g. stability analysis and Riccati equation. The numerical algorithm is well-developed and the eigenvalue problem of a single matrix can be solved efficiently. However, it requires much computation time to calculate the eigenvalues of real matrices for more than the-thousands times. For example, this issue occurs in the parameter space design method with volume rendering proposed in [1]. In this novel method, a scalar index for a design specification is calculated for each grid point in 3D space to get volume data, and the permissible set is visualized as iso-surfaces in 3D space by volume rendering (Figure 1). The designer can visually select an appropriate parameter. This numerical method is expected to treat more practical specifications than the previous analytical method in [2].

Control design problems are reduced to problems of finding a controller that satisfies design specifications of pole assignment, transient response, and frequency response. In [1], the method with rendering is studied for the specification of transient response. It is also very useful to develop the method for the specification of pole assignment. This requires calculation of the eigenvalues of non-symmetric real matrices for all the grid points. The matrix size is small, e.g. 15×15, and the number of grid points is more than ten-thousands, e.g. $50^3 = 125000$. Therefore, the eigenvalue problem for a large number of matrices needs to be computed, and the computing time of the eigenvalue problems dominates the processing time in the parameter space design with volume rendering. Thus, accelerating the computation of the eigenvalue problem for many small non-symmetric real matrices is needed.

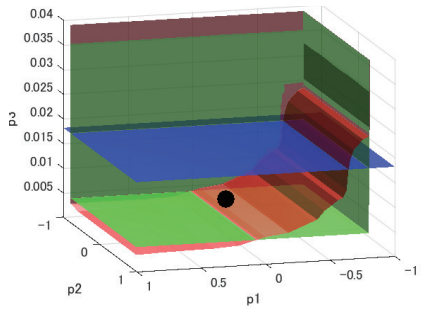

Figure 1. Volume rendering of the parameter space design in the pole assignment problems using eigenvalues obtained by the proposed method

In classical numerical linear algebra, to compute eigenvalues of a non-symmetric matrix, the QR algorithm [3], [4] is used. This algorithm is based on the factorization, called the QR decomposition, of a matrix A by division as a product of an orthogonal matrix Q and an upper triangular matrix R, that is, $A = QR$. To reduce the computing time of the QR algorithm, its variants have been proposed [4], [5]. Especially, in this work, we use *the implicit double-shift QR algorithm* [5] used in modern computational practice. The implicit double-shift QR algorithm is based on the implicit

Q theorem. Instead of the iterative QR decomposition, in this algorithm, *the double-shift QR sweep* is repeatedly performed.

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [6], [7]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [6]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [8], [9], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [10], since they have thousands of processor cores and very high memory bandwidth.

Many researchers have been devoted to accelerate the computation for matrix calculations for many small matrices using GPUs [11]–[14]. On the other hand, we can utilize software libraries and softwares for numerical linear algebra. LAPACK [15] and GSL [16] are software libraries for the CPU implementations. These libraries support the computation of matrix factorizations, multiplications, eigenvalues, etc. For GPU implementations, cuBLAS [17], cuSOLVER [18] and MAGMA [19] are available. We can also utilize MATLAB [20] to compute the eigenvalue problem as a linear algebra software. As far as we know, there are not any GPU implementations and software libraries for the bulk computation of eigenvalues for many small non-symmetric matrices.

The main contribution of this work is to propose a GPU implementation of bulk computation of eigenvalues of small real non-symmetric matrices. We focus on matrices of sizes no more than 32×32. However, as mentioned in the above, there are not any GPU implementations and software libraries for the computation of eigenvalues for many small non-symmetric matrices. To compute eigenvalues of matrices, we compute eigenvalues using *the implicit double-shift QR algorithm* that iterates matrix transformation. In this work, we propose three assignments of GPU threads to matrices. Also, to make the memory access efficient, we introduce memory arrangements in the device memory on the GPU for each of the thread assignments. We have implemented them on the GPU and evaluated the performance. The experimental results on NVIDIA GeForce GTX TITAN X show that our GPU implementation for 500000 matrices of size 5×5 and 30×30 attains a speed-up factor of 15.20 and 14.73 over the CPU implementation on Intel Core i7-4790, respectively.

II. Eigenvalues Computation of a Non-symmetric Real Matrix

This section reviews the QR algorithm to compute the eigenvalues of a matrix [4]. Especially, we focus on the eigenvalues computation for a square non-symmetric real matrix. There are several algorithms of computing eigenvalues for non-symmetric matrices. In this work, we use *the implicit double-shift QR algorithm* [5]. This algorithm uses *the double-shift QR sweep* instead of the QR decomposition to reduce the computation cost. For further details on this algorithm, the interested reader may refer to [5], [21] and the references within. The implicit double-shift QR algorithm consists of three steps:

Step 1: Perform *the Hessenberg reduction*
Step 2: Repeat the following operations until the size of the matrices becomes 1×1 or 2×2
- Iterate the double-shift QR sweep until a subdiagonal element is sufficiently small
- Split into two smaller matrices by *deflation* and apply Step 2 recursively

Step 3: Compute eigenvalues

In Step 1, the Hessenberg reduction makes a square matrix to an upper *Hessenberg form* matrix. An upper Hessenberg form matrix has zero entries below the first subdiagonal as shown in Figure 2. Algorithm 1 shows the Hessenberg reduction by Householder transformation and similarity transformation, where $\mathbf{v}$ is a Householder vector for k-th column. Let $A_{a:b,c:d}$ denote the sub-matrix of A of which the left-top element is $a_{a,c}$ and the right-bottom element is $a_{b,d}$. In the following, for simplicity, if the range that denotes a sub-matrix is out of the size of the matrix, the range is reduced to the size of the matrix.

Algorithm 1 The Hessenberg reduction

Input: $n \times n$ non-symmetric matrix A
Output: $n \times n$ Hessenberg form matrix H
1: **for** $k = 1$ **to** $n - 2$ **do**
2: // Householder vector creation
3: $\mathbf{v} \leftarrow A_{k+1:n,k}$
4: $\mathbf{v} \leftarrow \mathbf{v} + \mathrm{sign}(v_1)||\mathbf{v}||\mathbf{e}_1$
5: $\mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||}$
6: // Householder transformation
7: $A_{k+1:n,k:n} \leftarrow A_{k+1:n,k:n} - 2\mathbf{v}(\mathbf{v}^T A_{k+1:n,k:n})$
8: // similarity transformation
9: $A_{1:n,k+1:n} \leftarrow A_{1:n,k+1:n} - 2(A_{1:n,k+1:n}\mathbf{v})\mathbf{v}^T$
10: **end for**
11: **return** $H \leftarrow A$

In Step 2, we repeatedly execute the iterative double-shift QR sweep and deflation. The double-shift QR sweep consists of two steps: *bulge-generating* and *bulge-chasing*. Figure 3 shows the outline of the double-shift QR sweep. Bulge-generating transforms a Hessenberg form matrix to a matrix such that a bulge is added to the top left corner of a Hessenberg form matrix shown in Figure 3(a). After that, bulge-chasing moves the bulge down and to the right until it disappears (Figure 3(b)-(e)). By repeatedly performing the

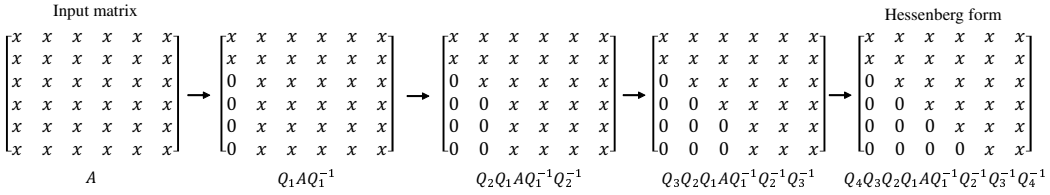

Figure 2. The Hessenberg reduction for a square matrix of size 6×6

double-shift QR sweep, a value of a subdiagonal element converges to zero. After converging, we split the matrix into the two smaller matrices by *deflation*. Deflation is decomposing an upper Hessenberg form matrix into the two smaller upper Hessenberg form matrices when a subdiagonal element converges to zero as illustrated in Figure 4. However, due to a computational error, the value may not become zero exactly. Therefore, in general, we consider a subdiagonal element converges to zero when the value is sufficiently small by comparing with the two neighboring diagonal elements. Algorithms 2 and 3 show bulge-generating and bulge-chasing, respectively.

Algorithm 2 Bulge-generating

Input: $n \times n$ Hessenberg form matrix H
Output: $n \times n$ Hessenberg form matrix with a bulge B
$x_1 \leftarrow (h_{1,1} - h_{n,n})(h_{1,1} - h_{n-1,n-1}) - h_{n-1,n}h_{n,n-1} + h_{1,2}h_{2,1}$
$x_2 \leftarrow h_{2,1}(h_{1,1} + h_{2,2} - h_{n-1,n-1} - h_{n,n})$
$x_3 \leftarrow h_{2,1}h_{3,2}$
// Householder vector creation
$\mathbf{v} \leftarrow \mathbf{x} + \mathrm{sign}(x_1)||\mathbf{x}||\mathbf{e}_1$
$\mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||}$
// Householder transformation
$H_{1:3,1:n} \leftarrow H_{1:3,1:n} - 2\mathbf{v}(\mathbf{v}^T H_{1:3,1:n})$
// similarity transformation
$H_{1:4,1:3} \leftarrow H_{1:4,1:3} - 2(H_{1:4,1:3}\mathbf{v})\mathbf{v}^T$
return $B \leftarrow H$

Algorithm 3 Bulge-chasing

Input: $n \times n$ Hessenberg form matrix with a bulge B
Output: $n \times n$ Hessenberg form matrix H
for $k = 1$ **to** $n - 2$ **do**
  // Householder vector creation
  $\mathbf{v} \leftarrow B_{k+1:k+3,k}$
  $\mathbf{v} \leftarrow \mathbf{v} + \mathrm{sign}(v_1)||\mathbf{v}||\mathbf{e}_1$
  $\mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||}$
  // Householder transformation
  $B_{k+1:k+3,k:n} \leftarrow B_{k+1:k+3,k:n} - 2\mathbf{v}(\mathbf{v}^T B_{k+1:k+3,k:n})$
  // similarity transformation
  $B_{1:k+4,k+1:k+3} \leftarrow B_{1:k+4,k+1:k+3} - 2(B_{1:k+4,k+1:k+3}\mathbf{v})\mathbf{v}^T$
end for
return $H \leftarrow B$

In Step 3, eigenvalues of the deflated matrices are computed one by one. Since the size of the matrices is 1×1 and 2×2, the eigenvalues can be computed easily.

III. CUDA Architecture

NVIDIA provides a parallel computing architecture, called CUDA, on NVIDIA GPUs. CUDA uses two types of memories: *the global memory* and *the shared memory*. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-12 Gbytes, but its access latency is very long. The shared memory is an extremely fast on chip memory with lower capacity, say, 16-112 Kbytes. The efficiency usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [8], [9]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed in the same time. Thus threads should perform coalescing access when they access to the global memory.

CUDA parallel programming model has a hierarchy of thread groups, called *grid*, *block*, and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to SMs such that all threads in a block are executed by the same SM in parallel. All threads can access to the global memory. Threads in a block can access to the shared memory of the SM to which the block is allocated. Since blocks are arranged to multiple SMs, threads in different blocks cannot share data in the shared memories.

In the execution, threads in a block are split into groups of thread, called *warps*. A warp is an implicitly synchronized group of threads. Each of these warps contains the same number of threads and is executed independently. When a warp is selected for execution, all threads execute the same instruction. Any flow control instruction (e.g. if-statements in C language) can significantly impact the effective instruction throughput by causing threads if the same warp to diverge, that is, to follow different execution paths, called *warp divergence*. If this happens, the different execution paths have completed, the threads back to the same execution path. For example, for an if-else statement, if some threads in a warp take the if-clause and the others take the else-clause, both clauses are executed in serial. On the other hand, when all threads in a warp branch in the same direction, all threads in a warp take the if-clause, or all take the else-clause. Therefore, to improve the performance, it is important to make branch behavior of all threads in a warp uniform. When one warp is paused or stalled, other warps can be executed to hide latencies and keep the hardware busy.

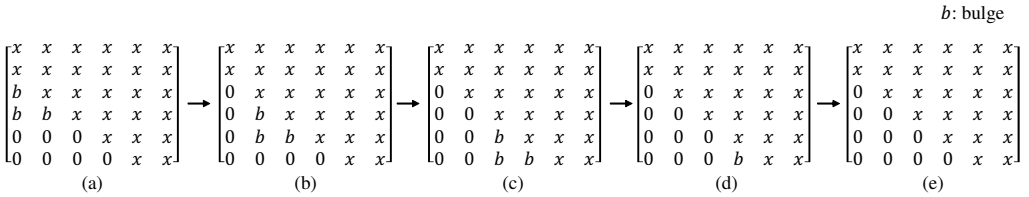

Figure 3. Bulge-generating and bulge-chasing in the double-shift QR sweep

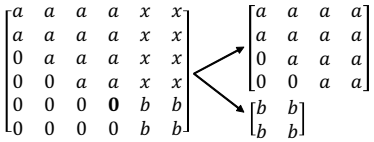

Figure 4. Matrix division by deflation

IV. GPU Implementation

This section presents the main contribution of this work, a GPU implementation of the implicit double-shift QR algorithm for many small matrices of size $n \times n$. Also, we use a 64-bit floating point number as a real number and two 64-bit floating point numbers as a complex number. In our implementation, we use only real numbers in Steps 1 and 2 during the computation. From Step 3, we use complex numbers.

Before the explanation about parallel execution on the GPU, we introduce three data arrangements for many matrices in the memory, *matrix-wise* (MW), *element-wise* (EW), and *row-wise* (RW). These three data arrangements show how to store multiple two-dimensional arrays in the memory that is a one-dimensional memory. In the MW arrangement, each matrix is stored one by one and elements of each matrix are stored in column-major order as shown in Figure 5(a). In this paper, the input data of matrices are stored to the main memory in the MW arrangement. In the EW arrangement, each element picked from the matrices in row-major order is stored element by element as illustrated in Figure 5(b). On the other hand, in the RW arrangement, each row taken from the matrices is stored row by row as illustrated in Figure 5(c). Two arrangements EW and RW are used in the global memory to make the memory access efficient. In this work, we propose three methods that compute them in parallel, *single-thread-based* (STB), *single-warp-based* (SWB), and *multiple-warp-based* (MWB). We explain the three methods using the above three data arrangements as follows.

In the STB method, one thread is used to sequentially perform the computation for one matrix (Figure 6). In this method, to make the global memory access coalesced, the data in the global memory are stored in the EW arrangement. Therefore, it is necessary to change the arrangement of the global memory from the MW arrangement to the EW

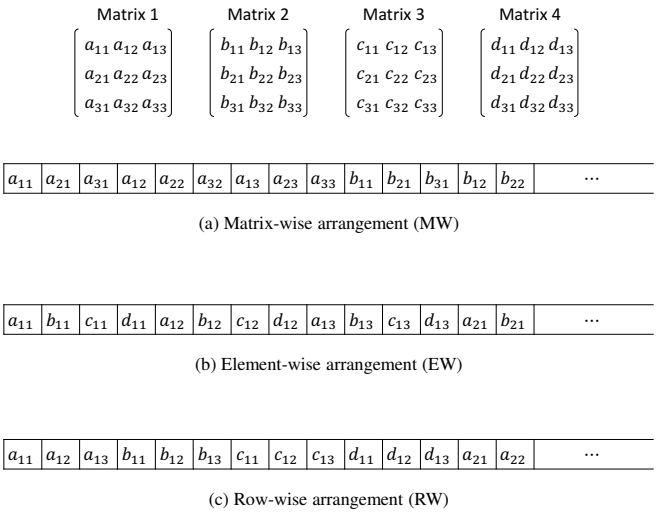

Figure 5. Data arrangement for multiple matrices in the memory

arrangement before the launch of kernels of this method. In Step 1, that is the Hessenberg reduction, the algorithm is a sequential oblivious algorithm [22] since all addresses accessed at each step is independent of the input. Namely, in computation of the Hessenberg reduction, all threads in a warp always can execute identical instructions. Therefore, the idea of GPU bulk execution technique [22]–[24] can be applied to the STB method for the Hessenberg reduction. On the other hand, in Step 2, since the size of matrix is changed after deflation, the sizes of matrices computed by threads in a warp may differ during the computation. In such case, warp divergence occurs and the global memory access is not coalesced. However, until deflation, threads in a warp can work without warp divergence.

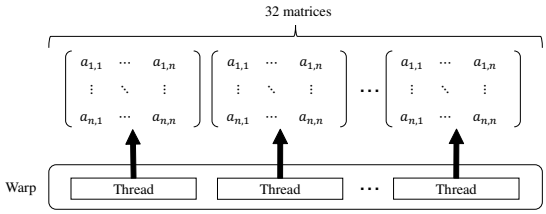

Figure 6. Single-thread-based method (STB)

In the SWB method, one warp is used to compute for one or more matrices as shown in Figure 7. In this method, when $\frac{32}{n}$ is indivisible, $32 - n\lfloor\frac{32}{n}\rfloor$ threads in every warp are not employed. For example, when the size of matrices is 7×7, $\lfloor\frac{32}{7}\rfloor = 4$ matrices are computed by one warp. In each warp, 7 threads are allocated to one matrix and 4 matrices are transformed in parallel. In this case, the remaining 4 threads are not used. In this method, the access to the global memory is made coalesced using the RW arrangement. Since the total size of matrices is smaller than other two methods, only in this method, all data of matrices can be located on the shared memory. Therefore, first of all the processes in this method, the data of matrices is loaded to the shared memory. Also, to avoid the bank conflict of the shared memory as much as possible, we use the padding technique [9]. The following operations are performed on the shared memory until the result is stored to the global memory. In Householder vector creation in Step 1, this method performs the computation on the shared memory as follows. To compute $||\mathbf{v}||$, the sum of squared values of $\mathbf{v}$ needs to be computed. We use the parallel sum reduction method [25] with the shared memory. After that, the Householder vector is computed by one thread and stored to the shared memory. On the other hand, in Householder vector creation in Step 2, the sum of only three squared values is computed. Therefore, in Step 2, the sum is directly computed instead of the parallel sum reduction method. In Householder transformation, we assign one thread to each column. Each thread computes the multiplication of the elements in the assigned column from top to bottom. In similarity transformation, we assign one thread to each row using n threads. Each thread computes the multiplication of the elements in the row from left to right. However, it is not always to employ n threads in these two transformations.

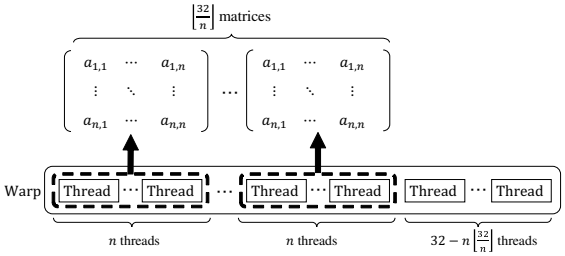

Figure 7. Single-warp-based method (SWB)

In the MWB method, n warps are used to compute for 32 matrices as shown in Figure 8. More specifically, we allocate n threads in n different warps to one matrix computation. Each matrix is computed in parallel using n threads. Since only the number of warps is depended on n, all threads in a warp are employed for any n unlike the SWB method. The parallel execution by n threads in the MWB method is same as that in the SWB method except that the execution is basically performed on the global memory. Also, since multiple warps are used, it is necessary to synchronize the execution with `syncthreads()` function. However, the synchronization is not performed frequently since it is done only at the end of Householder vector creation, Householder transformation, and similarity transformation. In this method, to access the global memory with coalescing access, we arrange data in the global memory using the EW arrangement. The arrangement is identical to that of the STB method.

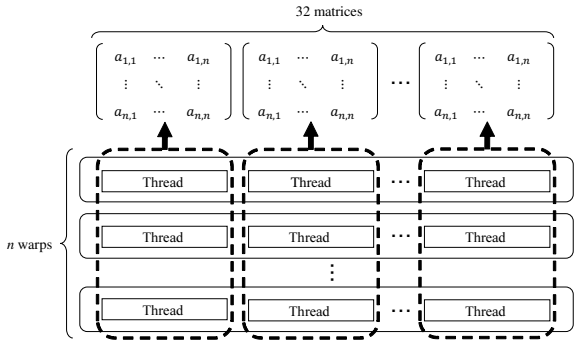

Figure 8. Multiple-warp-based method (MWB)

In this work, first, the input data of matrices are stored to the main memory on the host PC in the MW arrangement. The data are transferred to the global memory on the GPU as it is. In the above three methods, the data arrangement in the global memory needs to be rearranged for each utilized method. Therefore, we implemented kernels that mutually rearrange between the MW, EW, and RW arrangements on the global memory. In these kernels, we use the idea of the matrix transpose technique proposed in [26]. The idea is to efficiently transpose a two-dimensional array on the global memory with coalesced access using the shared memory. The rearrangements are not transposing, but this technique can be applied with small modification.

V. Performance Evaluation

The main purpose of this section is to show the performance evaluation of the proposed GPU implementation for the eigenvalues computation. We have used NVIDIA GeForce GTX TITAN X, which has 3072 cores in running on 1.075GHz. Also, we have used Intel Core i7-4790 running on 3.6GHz on the host PC. In the following, the computing time is average of 10 times execution.

First, we evaluate the performance of Step 1, that is, the Hessenberg reduction, using the proposed three methods, STB SWB and MWB. Table I shows the computing time of the Hessenberg reduction for 500000 matrices of size from 5×5 to 30×30 that are dense matrices randomly generated. The computing time does not include data transfer time between the main memory in the CPU and the device memory in the GPU. Also, input matrices in the global

memory are stored by the appropriate arrangement as shown in Figure 5. According to the table, the MWB method is faster for no more than 20×20 matrices, while the SWB method is faster for 25×25 or larger matrices.

Table I
THE COMPUTING TIME (IN MILLISECONDS) OF STEP 1 FOR 500000 MATRICES OF SIZE $n \times n$

n	5	10	15	20	25	30
STB	4.29	39.44	164.50	455.41	1006.50	1751.48
SWB	4.88	28.20	86.17	274.71	407.92	583.71
MWB	2.87	19.59	75.30	218.57	455.81	864.43

Table II shows the computing time of Steps 2 and 3 for 500000 matrices of size from 5×5 to 30×30. Similarly, the computing time does not include data transfer time. Also, input data in the global memory are stored by the appropriate arrangement in Figure 5 for each method. According to the table, the STB method is faster than the SWB and MWB methods.

Table II
THE COMPUTING TIME (IN MILLISECONDS) OF STEPS 2 AND 3 FOR 500000 MATRICES OF SIZE $n \times n$

n	5	10	15	20	25	30
STB	25.35	182.14	519.39	1125.50	2167.39	3624.97
SWB	57.37	394.02	1506.90	4095.07	6690.20	10228.14
MWB	60.03	451.41	1461.45	3557.14	8816.74	14278.44

Table III shows the computing time of our GPU implementation for 500000 matrices of size $n \times n$. We assume that all input data are stored in the main memory on the host PC using the data arrangement in the MW arrangement. The input data are transferred from the main memory on the CPU to the global memory on the GPU as it is. In Step 1 and Steps 2 and 3, according to the result in the above, we select the fastest method for each size of the matrix. Therefore, we rearrange the data in the global memory to the appropriate arrangement before launching the kernels if necessary. We note that we select the methods by considering the computing time including the rearranging time. After executing Step 3, the output data are rearranged to the MW arrangement on the global memory and transferred to the main memory on the CPU.

Table IV shows the comparison of our GPU implementation with two software libraries, LAPACK version 3.6.0 [15] and MAGMA version 2.0.2 [19], MATLAB version R2015b [20], and the sequential CPU implementation, that corresponds to CPU in the table, for 500000 matrices of size 5×5 to 30×30. In the sequential CPU implementation, we made a C program that performs the serial computation in Section II from scratch. LAPACK, MAGMA and MATLAB support parallel computation of the eigenvalue problem with multithreads on the CPU. MAGMA also supports parallel computation on the GPU. However, since the size of matrices is small in this experiment, MAGMA performed the computation only by CPU with multithreads. Additionally, since LAPACK, MAGMA and MATLAB do not support bulk computation of the eigenvalue problem, each implementation with them calls a procedure of computing the eigenvalue problem for each matrix. Due to such execution, multiple threads are launched and stopped before and after each procedure call, respectively. Therefore, there is overhead between each procedure call and it is not negligible. On the other hand, in our sequential CPU implementation and GPU implementation, since the computation is executed continuously, such overhead is extremely small. Our method can compute the bulk computation of the eigenvalue problem approximately 15 times faster than the CPU implementation.

VI. CONCLUSIONS

In this paper, we have presented a GPU implementation of bulk eigenvalue computations for a large number of small non-symmetric real matrices. The idea of our GPU implementation is to consider programming issues of the GPU architecture including warp divergence, coalesced access of the global memory, and bank conflict of the shared memory. We proposed three assignments of the GPU threads to compute in parallel and data assignment in the global memory for them. The experimental results show that our GPU implementation on NVIDIA GeForce GTX TITAN X attains a speed up factor of approximately 15 over the CPU implementation on Intel Core i7-4790.

REFERENCES

[1] M. Saeki, Y. Kurosaka, N. Wada, and S. Satoh, "Parameter space design of a nonlinear filter by volume rendering (in Japanese)," *Transactions of the Institute of Systems, Control and Information Engineers*, vol. 28, no. 10, pp. 419–425, 2015.

[2] J. Ackermann, *Robust Control: The Parameter Space Approach*, 2nd ed., ser. Communications and Control Engineering. Springer-Verlag London, 2002.

[3] J. G. Francis, "The QR transformation a unitary analogue to the LR transformation–part 1," *The Computer Journal*, vol. 4, no. 3, pp. 265–271, 1961.

[4] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, 1996.

[5] J. G. Francis, "The QR transformation–part 2," *The Computer Journal*, vol. 4, no. 4, pp. 332–345, 1962.

[6] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[7] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 153–159.

[8] *CUDA C Programming Guide Version 7.0*, NVIDIA Corporation, 2015.

Table III
THE COMPUTING TIME (IN MILLISECONDS) OF OUR GPU IMPLEMENTATIONS OF THE EIGENVALUE PROBLEM FOR 500000 MATRICES OF SIZE $n \times n$

n		5	10	15	20	25	30
data transfer (host to device)		18.20	71.09	158.42	285.92	439.84	637.01
data rearrangement	time	0.82	3.69	8.65	14.03	19.97	27.81
	arrange	MW→EW	MW→EW	MW→EW	MW→EW	MW→RW	MW→RW
Step 1	time	2.87	19.59	75.30	218.57	407.92	583.71
	method	MWB	MWB	MWB	MWB	SWB	SWB
data rearrangement	time	—	—	—	—	21.45	30.76
	arrange	—	—	—	—	RW→EW	RW→EW
Steps 2 and 3	time	25.35	182.14	519.39	1125.50	2167.39	3624.97
	method	STB	STB	STB	STB	STB	STB
data rearrangement	time	0.32	0.64	0.95	1.26	1.58	1.90
	arrange	EW→MW	EW→MW	EW→MW	EW→MW	EW→MW	EW→MW
data transfer (device to host)		7.94	15.12	22.68	30.39	38.25	45.13
total		55.51	292.26	785.38	1675.66	3096.40	4951.29

Table IV
THE COMPUTING TIME (IN MILLISECONDS) OF EIGENVALUES FOR 500000 MATRICES OF SIZE $n \times n$

n	5	10	15	20	25	30
LAPACK [15]	9556.42	51099.73	112859.84	194458.98	299943.68	432478.17
MAGMA [19]	75061.60	80263.20	89930.79	104234.16	143672.27	202531.19
MATLAB [20]	3397.26	10198.21	22875.17	39356.65	63514.07	89707.41
CPU	843.86	4362.26	12542.57	26172.12	46031.23	72951.33
GPU	55.51	292.26	785.38	1675.66	3096.40	4951.29
speed-up (CPU / GPU)	15.20	14.93	15.97	15.62	14.87	14.73

[9] *CUDA C Best Practice Guide Version 7.0*, NVIDIA Corporation, 2015.

[10] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[11] M. J. Anderson, D. Sheffield, and K. Keutzer, "A predictive model for solving small linear algebra problems in GPU registers," in *Proc. of IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 2–13.

[12] A. Haidar, T. T. Dong, S. Tomov, P. Luszczek, and J. Dongarra, "A framework for batched and GPU-resident factorization algorithms applied to block Householder transformations," in *Proc.of 30th International Conference on ISC High Performance*, 2015, pp. 31–47.

[13] T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. Dongarra, "LU factorization of small matrices: Accelerating batched DGETRF on the GPU," in *Proc. of IEEE International Conference on High Performance Computing and Communications*, 2014, pp. 157–160.

[14] A. Cosnuau, "Computation on GPU of eigenvalues and eigenvectors of a large number of small Hermitian matrices," in *Proc. of 14th International Conference on Computational Science*, vol. 29, 2014, pp. 800–810.

[15] *LAPACK–Linear Algebra PACKage*, http://www.netlib.org/lapack/.

[16] *GSL - GNU Scientific Library*, http://www.gnu.org/software/gsl/.

[17] NVIDIA Corp., *cuBLAS*, https://developer.nvidia.com/cublas.

[18] ——, *cuSOLVER*, https://developer.nvidia.com/cusolver.

[19] Innovative Computing Laboratory, *MAGMA:Matrix Algebra on GPU and Multicore Architectures*, http://icl.cs.utk.edu/magma/.

[20] The MathWorks, Inc., "MATLAB," http://mathworks.com/products/matlab.

[21] D. C. Sorensen, "Implicit application of polynomial filters in a k-step Arnoldi method," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 357–385, 1992.

[22] K. Tani, D. Takafuji, K. Nakano, and Y. Ito, "Bulk execution of oblivious algorithms on the unified memory machine, with GPU implementation," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2014, pp. 586–595.

[23] D. Takafuji, K. Nakano, and Y. Ito, "A CUDA C program generator for bulk execution of a sequential algorithm," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, Aug. 2014, pp. 178–191.

[24] T. Fujita, K. Nakano, and Y. Ito, "Bulk execution of Euclidean algorithms on the CUDA-enabled GPU," *International Journal of Networking and Computing*, vol. 6, no. 1, pp. 42–63, 2016.

[25] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[26] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in CUDA," 2010.