International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 2, pages 132–143, July 2011

Acceleration of Hessenberg Reduction for Nonsymmetric Eigenvalue Problems in a Hybrid CPU-GPU Computing Environment

Jun-ichi Muramatsu, Takeshi Fukaya, Shao-Liang Zhang
Department of Computational Science & Engineering, Nagoya University,
Furo-cho, Chikusa, Nagoya, Aichi, 464-8603, Japan

Kinji Kimura
Department of Applied Mathematics and Physics, Kyoto University
Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan

Yusaku Yamamoto
Department of Computational Science, Kobe University
1-1 Rokko-Dai, Nada, Kobe, 657-8501, Japan

Received: January 16, 2011
Revised: May 20, 2011
Accepted: June 20, 2011
Communicated by Yasuaki Ito

Abstract

Solution of large-scale dense nonsymmetric eigenvalue problem is required in many areas of scientific and engineering computing, such as vibration analysis of automobiles and analysis of electronic diffraction patterns. In this study, we focus on the Hessenberg reduction step and consider accelerating it in a hybrid CPU-GPU computing environment. Considering that the Hessenberg reduction algorithm consists almost entirely of BLAS (Basic Linear Algebra Subprograms) operations, we propose three approaches for distributing the BLAS operations between CPU and GPU. Among them, the third approach, which assigns small-size BLAS operations to CPU and distributes large-size BLAS operations between CPU and GPU in some optimal manner, was found to be consistently faster than the other two approaches. On a machine with an Intel Core i7 processor and an NVIDIA Tesla C1060 GPU, this approach achieved 3.2 times speedup over the CPU-only case when computing the Hessenberg form of a 8,192 $\times$ 8,192 real matrix.

Keywords: Eigenvalue problem, Hessenberg reduction, GPU

1 Introduction

The nonsymmetric eigenvalue problem $A\boldsymbol{x} = \lambda\boldsymbol{x}$ with a dense coefficient matrix A arises in various fields of science and engineering [5]. For example, analysis of electronic diffraction patterns requires the solution of dense complex nonsymmetric eigenvalue problem. In vibration analysis of automobiles by finite element methods, nonsymmetric eigenvalue problem with sparse coefficient matrices of order more than several millions need to be solved. In that case, the original eigenproblem is reduced to

a smaller eigenproblem with a dense coefficient matrix by aggregating the contributions from the nodes. Even after reduction, the resulting dense matrix has order of several thousands and requires large computation time. So there is a great need for speeding up the solution of such large-scale nonsymmetric dense eigenvalue problem.

Recently, General Purpose GPU (GPGPU), which exploits the large computing power and high memory throughput of graphic processing units to accelerate general-purpose computation, has attracted much attention [3]. In particular, the advent of the CUDA programming environment [1] has made it possible to use the computing power of GPU with a simple extension to the standard C language. In addition, the CUBLAS library, which is an optimized library for performing BLAS (Basic Linear Algebra Subprograms) [7][8] operations in the CUDA environment, has also been provided, facilitating development of matrix computation programs using GPU.

In this paper, we consider accelerating the solution of the nonsymmetric eigenvalue problem in a hybrid CPU-GPU computing environment. In particular, we focus on the Hessenberg reduction step, which accounts for considerable computing time in the whole procedure. To achieve high performance in such environment, it is critical to distribute the work judiciously between CPU and GPU. Considering that the Hessenberg reduction algorithm consists almost entirely of BLAS operations, we study three approaches. The first one executes almost all BLAS operations on GPU, using the CUBLAS library. In the second approach, only large-size BLAS operations are executed on GPU and small-size BLAS operations are executed on CPU. In the third approach, large-size BLAS operations are divided into two portions of unequal size and they are assigned to CPU and GPU, respectively. The size of each portion is determined by a division parameter. This gives more flexibility of work distribution between CPU and GPU. The small-size BLAS operations are executed on CPU. We describe implementations of these three approaches in detail and compare their performance experimentally.

There are attempts to accelerate the Hessenberg reduction using GPU. For example, the CULA library [2], which is a matrix computation library for the CUDA environment, contains a routine for Hessenberg reduction. Tomov et al. propose an efficient approach for computing the Hessenberg reduction in a hybrid CPU-GPU environment [15]. They report very large speedup thanks to the use of highly optimized BLAS routines. However, in both of these works, distribution of the computational work between CPU and GPU is done in some fixed manner. More precisely, some of the BLAS operations are assigned to CPU, while others are assigned to GPU. In contrast, our third approach described above is more flexible in that one BLAS routine can be distributed between CPU and GPU and the ratio of work distribution can be specified by a parameter. Concerning work distribution between CPU and GPU for other linear algebra computations, such as FFT and matrix-matrix multiplication, the readers are referred to [12] and [13]. See also [11] for performance of various numerical algorithms on GPU.

This paper is organized as follows: in section 2, we explain the standard algorithm for the nonsymmetric eigenvalue problem. Section 3 gives a brief summary of CUDA and CUBLAS. Section 4 describes our approaches to accelerate Hessenberg reduction using CUBLAS. Experimental results that compare the performance of our three implementations are given in section 5. Finally, section 6 gives some concluding remarks.

2 Algorithm for the dense nonsymmetric eigenvalue problem

2.1 Four steps for the solution of the nonsymmetric eigenvalue problem

Let A be an $n \times n$ real nonsymmetric matrix. The standard procedure for solving the eigenvalue problem $A\mathbf{x} = \lambda\mathbf{x}$ consists of the following four steps [10]. Although the procedure given here is for real matrices, the procedure for complex matrices is almost the same.

Step 1. Hessenberg reduction The input matrix A is reduced to an upper Hessenberg matrix H by applying Householder transformations $Q_1, Q_2, \ldots, Q_{n-2}$ from left and right.

$$Q_{n-2}^T \cdots Q_2^T Q_1^T A Q_1 Q_2 \cdots Q_{n-2} = H. \tag{1}$$

Step 2. Accumulation of Householder transformations The Householder transformations used in Step 1 are accumulated as an orthogonal matrix Q.

$$Q_1 \cdots Q_{n-1} Q_{n-2} = Q. \tag{2}$$

Step 3. Schur decomposition The Hessenberg matrix H is transformed into a block upper triangular matrix T with diagonal blocks of size at most 2 by applying orthogonal transformations from left and right (the QR algorithm). The orthogonal matrices used in the transformation are accumulated as an orthogonal matrix P.

$$P_N^T \cdots P_2^T P_1^T H P_1 P_2 \cdots P_N = T, \tag{3}$$
$$P_1 P_2 \cdots P_N = P. \tag{4}$$

The eigenvalues of A are given as the eigenvalues of the diagonal blocks of T.

Step 4. Computation of the eigenvectors The eigenvectors $\boldsymbol{y}$ of T are computed and they are transformed into the eigenvectors of A by $\boldsymbol{x} = QP\boldsymbol{y}$.

The computational time of each step is shown in Fig. 1 for the $n = 4,800$ case. The computation is performed using LAPACK [4] on the Intel Core i7 920 (2.66GHz) processor using four cores. It can be seen that steps 1, 3 and 4 occupy a large fraction of the computational time. In terms of computational work, step 1 usually requires much less work than step 3. However, in contrast to step 3, which can be performed almost entirely with the level-3 BLAS [8], step 1 requires both level-2 and level-3 BLAS operations. As is well known, the level-2 BLAS cannot reuse the matrix data and its performance is usually bounded not by the peak FLOPS value of the CPU but by the memory throughput [10][6]. This is the main reason why step 1 requires long time. Step 4, which occupies the largest time, is slow because the corresponding LAPACK routine is written without using level-2 nor level-3 BLAS. The reason is that various exception handling is required in the computation. To speed up this routine, overall modification of the algorithm is necessary.

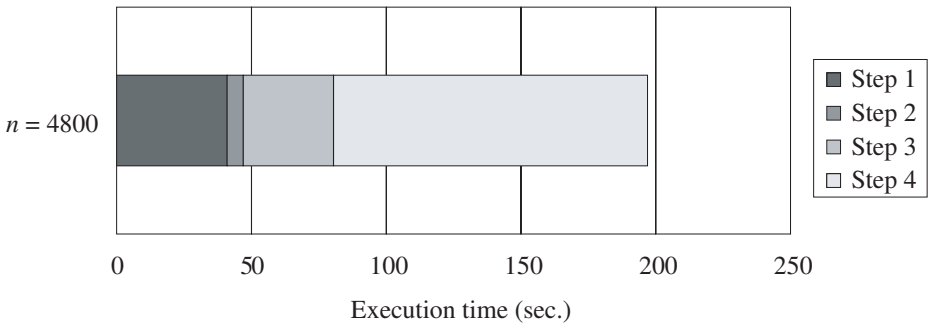

Figure 1: Execution time of steps 1 to 4.

In this paper, we focus on speeding up step 1 using GPU. GPUs have much higher memory throughput than CPUs, so we can expect that the level-2 BLAS can execute much faster on GPUs. As a preparation for porting step 1 to GPU, we will explain the algorithm of Hessenberg reduction in more detail in the rest of this section.

2.2 Hessenberg reduction by Householder transformations

Let us denote the submatrix of A consisting of rows r_1 through r_2 and columns c_1 through c_2 by $A_{r_1:r_2,c_1:c_2}$. Also, denote the identity matrix of order m by I_m and the zero vector of length m by $\mathbf{0}_m$. At the kth step of the Hessenberg reduction ($k = 1, \ldots, n-2$), we determine a Householder transformation $\tilde{Q}_k = I_{n-k} - \tau_k \tilde{\boldsymbol{v}}_k \tilde{\boldsymbol{v}}_k^T$ to annihilate all but the first element of current $A_{k+1:n,k}$. The

Householder transformation Q_k in Eq. (1) is defined as $Q_k = I_k \oplus \tilde{Q}_k$, where $\oplus$ denotes the direct sum of two matrices.

Using these definitions of $\tilde{Q}_k$ and Q_k, the algorithm of Hessenberg reduction can be written as follows.

for $k = 1$ to $n-2$ **do**
  Generate $(\tau_k, \tilde{\boldsymbol{v}}_k)$ from $A_{k+1:n,k}$.
  [Application of Q_k from left]
  $\tilde{\boldsymbol{u}}^T := \tilde{\boldsymbol{v}}_k^T A_{k+1:n,k+1:n}$
  $A_{k+1:n,k+1:n} := A_{k+1:n,k+1:n} - \tau_k \tilde{\boldsymbol{v}}_k \tilde{\boldsymbol{u}}^T$
  [Application of Q_k from right]
  $\tilde{\boldsymbol{w}} := A_{1:n,k+1:n} \tilde{\boldsymbol{v}}_k$
  $A_{1:n,k+1,n} := A_{1:n,k+1,n} - \tau_k \tilde{\boldsymbol{w}} \tilde{\boldsymbol{v}}_k^T$
end for

This is the basic algorithm for Hessenberg reduction. In this algorithm, most of the computations are done with the level-2 BLAS, namely, matrix-vector multiplication and rank-1 update of a matrix ($A - \tau \boldsymbol{v}\boldsymbol{u}^T$). However, these operations cannot reuse the matrix data and their performance is usually limited by the memory throughput. To mitigate this problem, a blocked version of Hessenberg reduction has been proposed [9], as we will see below.

2.3 Block algorithm for Hessenberg reduction

In this subsection, we overview the blocked version of the Hessenberg reduction algorithm. The main idea is to accumulate several Householder transformations and apply them at once, using matrix multiplications. More specifically, let L be the block size and consider the 1st to the Lth step of Hessenberg reduction. Let $\boldsymbol{v}_k = \boldsymbol{0}_k \oplus \tilde{\boldsymbol{v}}_k$ and define an n by k matrix V_k by

$$V_k = [\boldsymbol{v}_1 \, \boldsymbol{v}_2 \, \ldots \, \boldsymbol{v}_k]. \tag{5}$$

Also, let T_k and Y_k be matrices such that

$$(I - \boldsymbol{v}_1 \boldsymbol{v}_1^T) \cdots (I - \boldsymbol{v}_k \boldsymbol{v}_k^T) = I - V_k T_k V_k^T, \tag{6}$$
$$Y_k = A V_k T_k. \tag{7}$$

Eq. (6) is known as the compact-WY representation of the k Householder transformations [14]. Then, if we have V_L, T_L and Y_L, we can apply the 1st to the Lth Householder transformations to A from left and right as follows:

$$A := A(I - V_L T_L V_L^T) = A - Y_L V_L^T, \tag{8}$$
$$A := (I - V_L T_L^T V_L^T) A. \tag{9}$$

Clearly, these operations can be done entirely with the level-3 BLAS, or matrix multiplication. Note also that in computing Eq. (9), only rows 2 through n are affected due to the structure of $\boldsymbol{v}_k$.

The matrices V_k, T_k and Y_k can be generated recursively. First, given the 1st column of A, we can construct τ_1 and $\boldsymbol{v}_1$. Then we set

$$V_1 = [\boldsymbol{v}_1], \tag{10}$$
$$T_1 = [\tau_1], \tag{11}$$
$$Y_1 = A V_1 T_1. \tag{12}$$

Now, assume that we are given V_{k-1}, T_{k-1} and Y_{k-1}. Then we can update the kth column of A using Eqs. (8) and (9). From the updated column, we can compute τ_k and $\boldsymbol{v}_k$, constructing V_k. Then T_k can be computed using the update formula of the compact-WY representation as follows [14]:

$$T_k = \begin{bmatrix} T_{k-1} & -\tau_k T_{k-1} V_{k-1}^T \boldsymbol{v}_k \\ \boldsymbol{0}^T & \tau_k \end{bmatrix}. \tag{13}$$

To compute Y_k, we use the following relationship:

$$\begin{aligned} Y_k &= AV_kT_k \\ &= A[V_{k-1}\ \boldsymbol{v}_k]\begin{bmatrix} T_{k-1} & -\tau_k T_{k-1}V_{k-1}^T\boldsymbol{v}_k \\ \mathbf{0}^T & \tau_k \end{bmatrix} \\ &= \left[AV_{k-1}T_{k-1}\ \ \tau_k\left(-AV_{k-1}T_{k-1}V_{k-1}^T\boldsymbol{v}_k + A\boldsymbol{v}_k\right)\right] \\ &= \left[Y_{k-1}\ \ \tau_k\left(-Y_{k-1}V_{k-1}^T\boldsymbol{v}_k + A\boldsymbol{v}_k\right)\right]. \end{aligned} \tag{14}$$

In summary, in the block algorithm, the computation is divided into the following two phases.

(a) Update of columns 1 through L of A and construction of V_L, T_L and Y_L. This is performed by repeating the following for $k = 1, 2, \ldots, L$:

(a1) $A_{*,k} := A_{*,k} - Y_k\left((V_k)_{k,*}\right)^T$,

(a2) $A_{*,k} := (I - V_kT_k^TV_k^T)A_{*,k}$,

(a3) Construct the Householder transformation $(\tau_k, \boldsymbol{v}_k)$ from $A_{*,k}$,

(a4) Compute T_k by Eq. (13),

(a5) Compute Y_k by Eq. (14).

(b) Update of column $L+1$ through n of A:

(b1) $A_{*,L+1:n} := A_{*,L+1:n} - Y_L\left((V_L)_{L+1:n,*}\right)^T$,

(b2) $A_{*,L+1:n} := (I - V_LT_L^TV_L^T)A_{*,L+1:n}$.

In this algorithm, step (b), which accounts for more than 60% of the computational work, can be done entirely with level-3 BLAS. However, the remaining part, step (a), consists of level-2 BLAS, namely, matrix-vector multiplications. This becomes a serious bottleneck, as can be seen from Fig. 1. We seek to accelerate both steps (a) and (b) using GPU with high floating-point performance and large memory throughput.

3 CUDA and CUBLAS

To speed up Hessenberg reduction with the computing power of GPU, we use the CUDA (Compute Unified Device Architecture) environment [1] provided by NVIDIA Corp. CUDA is an integrated software development environment for GPGPU and consists of a compiler, mathematical libraries and other software.

3.1 The nvcc compiler

When doing matrix computations in the CUDA environment, we have two options. One is to use the nvcc compiler. In this case, the user writes a subroutine to be executed on GPU using an extension of the C language. The nvcc compiler takes the main program and this subroutine as inputs and outputs executable files both for CPU and GPU. This provides the user with a large degree of freedom in programming. On the other hand, the user has to write the GPU program from scratch. This is not easy, because various optimization techniques must be employed to exploit the potential of GPU.

3.2 The CUBLAS

Another approach is to use the CUBLAS, which is a BLAS library for GPU. The CUBLAS consists of routines to transfer data between the CPU memory and the GPU memory, and routines to perform basic linear algebra operations on data residing on the GPU memory. To use the CUBLAS, the user first transfers the necessary data to the GPU memory, performs several BLAS operations on

that data, and gets back the data to the CPU memory (see Fig. 2). The CUBLAS is optimized for NVIDIA's GPUs and the user can exploit the computing power of GPU while minimizing the modification of the original program. In view of these advantages, we will use the CUBLAS in this study.

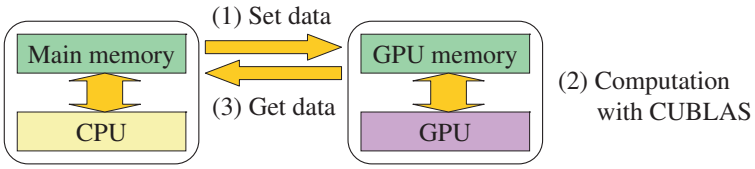

Figure 2: Usage of the CUBLAS library.

4 Implementation of Hessenberg reduction using CUBLAS

4.1 Basic principles for implementation

As we can see from Fig. 2, CPU and GPU have distinct memory spaces. The data transfer between these two memory spaces is usually very slow, achieving only a few gigabytes per second. Hence, to use the computing power of GPU effectively, it is crucial to minimize the data transfer between CPU and GPU. From this viewpoint, it would be the best to send all the data to GPU at first and perform all the subsequent computations on GPU.

However, there are operations that cannot be performed by CUBLAS. For example, construction of a Householder reflector in step (a3) is a function not provided by CUBLAS (or BLAS). In that case, we have to get the data to CPU, perform the necessary operation, and send back the data to GPU. Also, from the viewpoint of utilizing the computing power of the system to a maximum extent, it is sometimes more effective to assign some of the work to CPU or to distribute some of the work between CPU and GPU. Based on these observations, we consider three implementations in the following.

4.2 Implementation (I)

In this implementation, we try to minimize the amount of work assigned to CPU. As can be seen from the algorithm presented in subsection 2.3, all the computations except for the construction of Householder transformations can be done with BLAS. So we assign only the construction of Householder transformations to CPU. The schematic diagram of this implementation is shown in Fig. 3. At the beginning of Hessenberg reduction, we send the matrix A to GPU. At each step k, after updating the kth column of A using Eq. (8) and (9), we get the updated column to CPU, construct the Householder transformation, and send back τ_k and $\boldsymbol{v}_k$ to GPU. All other computations are performed on GPU using CUBLAS. Finally, the reduced matrix is returned to CPU.

This implementation requires one send and one receive per step, and the data size of each transfer is $O(n)$. Thus the total amount of data transfer is $O(n^2)$. Since the computational work of Hessenberg reduction is $O(n^3)$, we can expect that the cost of data transfer becomes negligible for large n.

4.3 Implementation (II)

The second approach is to assign larger computational tasks to GPU and assign smaller ones, including the construction of Householder transformations, to CPU. We consider this approach because generally GPU cannot attain high performance on small data, due to insufficient level of parallelism. There are several ways of dividing the computational tasks in Hessenberg reduction between CPU and GPU. From the structure of the algorithm described in subsection 2.3, we know

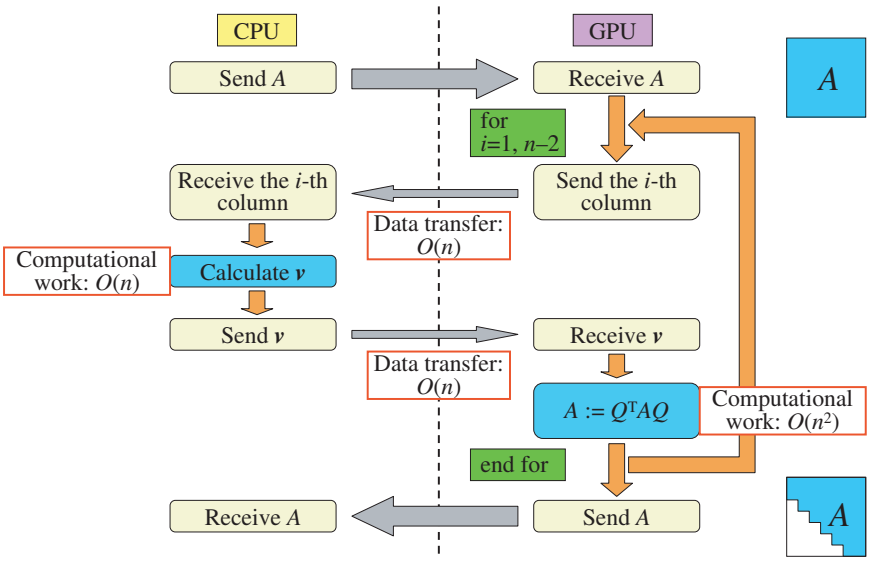

Figure 3: Schematic diagram of implementation (I).

that the phase (b), that is, update of column $L+1$ through n of A with aggregated Householder transformations, consists of large-size level-3 BLAS operations. On the other hand, the phase (a), that is, update of column 1 through L of A and construction of V_L, T_L and Y_L, consists of small-size level-1 and level-2 BLAS operations, except for the computation of $A\boldsymbol{v}_k$ in step (a5). In view of this, we divide the work as follows:

- Phase (a), except for the computation of $A\boldsymbol{v}_k$ in step (a5), is assigned to CPU.
- Phase (b) and the computation of $A\boldsymbol{v}_k$ in step (a5) are assigned to GPU.

As in the implementation (I), we send the whole matrix to GPU at the beginning of Hessenberg reduction. At each step k, the computation proceeds as follows:

- CPU receives the kth column of A from GPU, updates it as shown in steps (a1) and (a2), constructs the Householder transformation $(\tau_k, \boldsymbol{v}_k)$ and sends $\boldsymbol{v}_k$ to GPU.
- GPU computes $A\boldsymbol{v}_k$ and sends the result to CPU.
- CPU updates T_k and Y_k using Eqs. (13) and (14) and sends the kth columns of T_k and Y_k to GPU.

After these steps are repeated L times, GPU updates the columns $L+1$ through n of A as shown in steps (b1) and (b2).

The total amount of data transfer is twice that of the implementation (I), but it might be advantageous to perform small-size matrix computations on CPU.

4.4 Implementation (III)

In implementation (II), large-size BLAS operations such as the computation of $A\boldsymbol{v}_k$ and steps (b1) and (b2) are performed exclusively by GPU. The CPU is idle while these operations are executed. The implementation (III) tries to improve this situation. To this end, we define a division parameter r $(0 \le r \le 1)$, which specifies the ratio of matrix $A_{*,L+1:n}$ assigned to CPU. More precisely, among the $n-L$ columns of $A_{*,L+1,n}$, the first $r(n-L)$ columns are allocated to CPU, while the remaining columns are allocated to GPU. In computing $A\boldsymbol{v}_k$ and steps (b1) and (b2), both CPU and GPU join the computation using the part of $A_{*,L+1:n}$ they own.

At each step k, the computation proceeds as follows:

- CPU receives the kth column of A from GPU (unless it is already owned by CPU), updates it as shown in steps (a1) and (a2), constructs the Householder transformation $(\tau_k, \boldsymbol{v}_k)$ and sends $\boldsymbol{v}_k$ to GPU.
- Both CPU and GPU compute partial result of $A\boldsymbol{v}_k$ using the part of A they own.
- CPU receives the partial result computed by GPU and combine it with the partial result of its own to get the full result of $A\boldsymbol{v}_k$.
- CPU updates T_k and Y_k using Eqs. (13) and (14) and sends the kth columns of T_k and Y_k to GPU.

After these steps are repeated L times, both CPU and GPU execute steps (b1) and (b2) and update the part of $A_{*,L+1:n}$ they own. After this point, the 1st through the Lth columns of A will no longer be involved in the computation. We therefore redistribute the columns of A so that the ratio of active columns allocated to CPU and GPU is $r : 1 - r$.

In the next section, we compare the performance of these two implementations experimentally.

5 Performance evaluation

5.1 Computational environments

Based on the approaches given in the previous section, we wrote programs that perform Hessenberg reduction using CPU and GPU and evaluated their performance. We used LAPACK routine DGEHRD (a subroutine for Hessenberg reduction) as a basis, translated it into the C language and rewrote it using CUBLAS to use GPU. For comparison, we also show the performance of routine DGEHRD using CPU only.

As test matrices, we used real random matrices whose entries follow uniform distribution in $[0, 1]$. The matrix size n was varied from 1,024 to 8,192 and the block size L was fixed to 32, which proved to be the optimal value from preliminary experiments.

The computational environments are as shown in Table 1. All of the four cores of the Core i7 CPU were used in all experiments.

item	data
CPU	Core i7 920 (2.66 GHz) [4 cores]
CPU Memory	6.0GB
Compiler	gcc ver.4.1.2 (option -O3) Intel Fortran Compiler ver.9.1
GPU	NVIDIA Tesla C1060
GPU Memory	4.0GB
CUDA version	2.0

Table 1: Computational environments.

5.2 Performance of Hessenberg reduction

We show the execution times of Hessenberg reduction in Table 2 and Fig. 4. In the table, "CPU" means the execution time of LAPACK routine DGEHRD using four cores of the CPU. "GPU1", "GPU2" and "GPU3" mean the execution times for the implementation (I), (II) and (III) described in subsections 4.2, 4.3 and 4.4, respectively. "Speedup" means their relative speed compared with the CPU-only case. For the implementation (III), we varied the division parameter r from 1/32 to 32/32 and chose the best value for each n. These values are denoted as r_{opt}.

n	1024	2048	3072	4096	5120	6144	7168	8192
CPU	0.29	2.96	10.04	23.83	46.10	80.94	127.70	192.51
GPU1	1.69	4.05	8.40	15.45	25.37	39.18	57.47	81.80
Speedup	0.17	0.73	1.20	1.54	1.82	2.07	2.22	2.35
GPU2	1.35	2.95	6.09	11.59	19.63	31.38	47.31	69.50
Speedup	0.22	1.00	1.65	2.06	2.35	2.58	2.70	2.77
GPU3	1.11	2.32	4.91	9.65	16.56	27.36	41.09	59.86
Speedup	0.27	1.27	2.04	2.47	2.78	2.96	3.11	3.22
r_{opt}	24/32	10/32	8/32	6/32	5/32	4/32	4/32	4/32

Table 2: Execution time (in seconds) of Hessenberg reduction.

It is clear from the table that the use of GPU cannot accelerate the computation when the matrix size is one or two thousands. One possible reason for this is that the data transfer between CPU and GPU, which requires $O(n^2)$ time, is not negligible compared with the $O(n^3)$ time for arithmetic operations. Another reason is that the CUBLAS is not efficient on small matrices. As the matrix size becomes larger, the speedup increases, and when $n = 8,192$, the implementation (III) attains 3.2 times speedup over the CPU-only case. It is also apparent that the implementation (III) consistently outperforms the implementation (I) and (II). This shows that it is more efficient to assign operations on small matrices to CPU than doing as much work as possible on GPU and that work distribution between CPU and GPU works effectively. Note that the optimal division parameter r_{opt} is larger than 1/2 for small n and decreases monotonically as n increases. This indicates that small-size BLAS runs more efficiently on CPU, while large-size BLAS runs several times faster on GPU than on CPU. In the face of such performance characteristics, our implementation (III) can achieve optimal work distribution between CPU and GPU due to the existence of the division parameter.

5.3 Discussion

How to determine the optimal value of r In the above experiments, we determined the optimal value of the division parameter r by trying several values of r for each n and choosing the best one empirically. We believe that this is acceptable because this procedure needs to be done only once at the installation of the Hessenberg reduction program. An alternative approach would be to predict the execution time of the Hessenberg reduction program using some performance model and estimate an appropriate value of r in advance. In [16], a methodology is proposed to predict the execution time of linear algebra programs based on the performance models of BLAS routines. By using this approach, the time needed to tune the division parameter r will be greatly reduced. We are pursuing this approach.

Effect of suboptimal choice of r To see how suboptimal choice of r affects the performance, we plotted the execution time of the implementation (III) as a function of r. Here, we fixed the matrix size n to 4,096 and varied r from 2/32 to 16/32. The result is shown in Fig. 5. The optimal value is $r = 6/32$ in this case. By changing r to 2/32 or 10/32, the execution time increases more than 10%. Thus we know that it is important to optimize r to achieve the best performance.

Comparison with the CULA library Our implementations of the Hessenberg reduction are based on the CUBLAS. This approach has two advantages. First, we can exploit the performance of the optimized CUBLAS library. Second, it is easy to transport the program developed in this way to another GPU or accelerator. However, in this approach, operations not supported by BLAS must be done by CPU. Thus data transfer between CPU and GPU is necessary at each step of the algorithm. An alternative approach would be to perform all the operations on the GPU. To study how this approach compares with ours, we evaluated the performance of the CULA library, which adopts this approach. The execution times of the CULA's Hessenberg reduction routine are shown

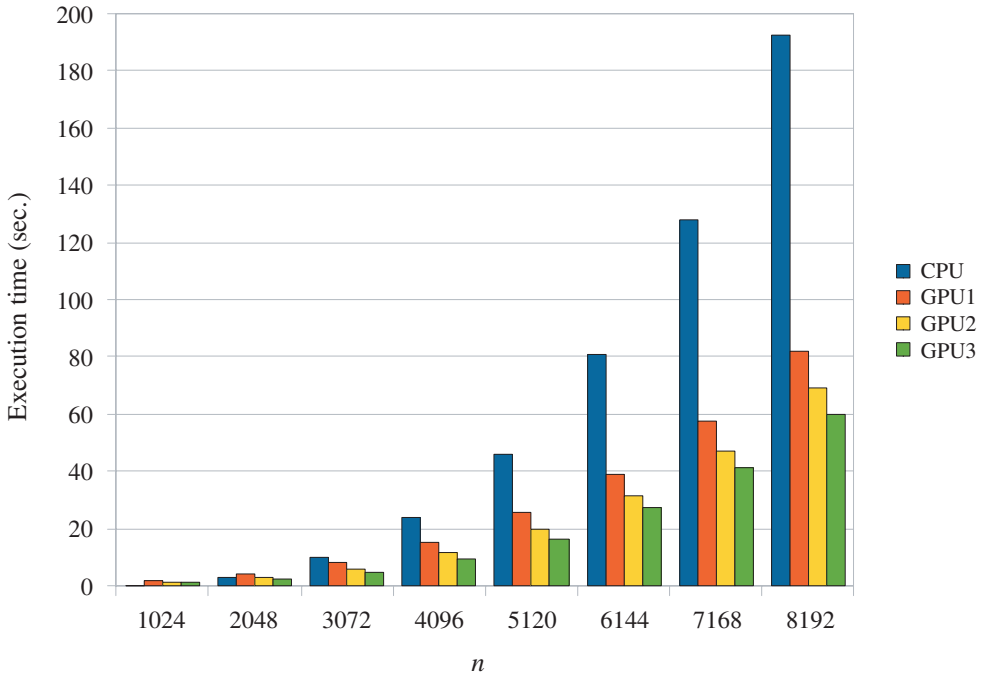

Figure 4: Execution time (in seconds) of Hessenberg reduction.

in Table 3. By comparing Table 3 with Table 2, we know that our implementation (III) with the best division parameter r_{opt} is faster than the CULA routine when $n \geq 3,072$. This is because our implementation uses both CPU and GPU efficiently. Note however that the difference is not large. This seems to be because our implementation requires data transfer between CPU and GPU more frequently than the CULA implementation.

n	1024	2048	3072	4096	5120	6144	7168	8192
CULA	0.64	2.13	5.21	10.58	18.02	28.42	42.19	64.44

Table 3: Execution time (in seconds) of CULA Hessenberg reduction.

Effect of speeding up the Step 1 In this paper, we picked up step 1 of the nonsymmetric eigenvalue solver, namely Hessenberg reduction, and showed how to speed up this step using GPU. Because step 4 requires longer time than step 1, it may appear that this speed up has limited impact on the overall performance of the nonsymmetric eigensolver. However, there are cases where only the eigenvalues are needed [5]. In those cases, steps 2 and 4 are unnecessary and step 1 occupies most of the execution time. Thus the speedup achieved in this paper will be more significant in such cases. Also, it is possible to speed up step 4 using GPU by modifying the current algorithm. If this has been achieved, the speed up of step 1 will become more important.

6 Conclusion

In this paper, we proposed three approaches for accelerating Hessenberg reduction in a hybrid computing environment with CPU and GPU. Among them, the third approach, which assigns small-size BLAS operations to CPU and distributes large-size BLAS operations between CPU and GPU in some optimal manner, was consistently faster than the other two approaches. On a platform with

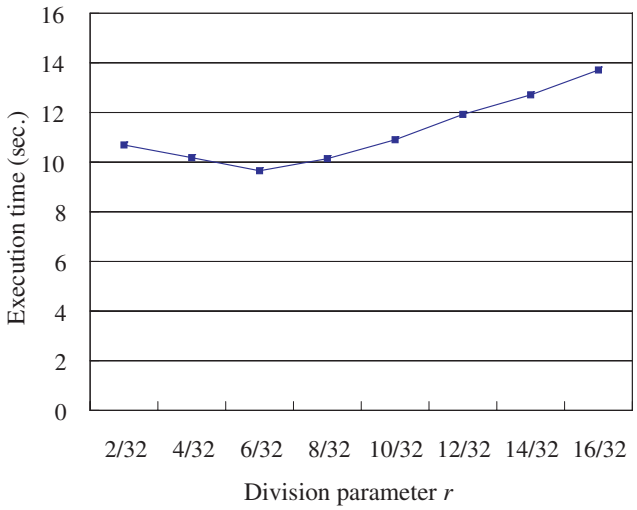

Figure 5: Execution time as a function of division parameter r (n=4,096).

an Intel Core i7 processor and an NVIDIA Tesla C1060 GPU, the third approach achieved up to 3.2 times speedup over the CPU-only case.

Our future work includes detailed analysis of the performance results and development of more efficient strategies for distributing the work between CPU and GPU. It also remains to speed up steps 3 and 4 of the nonsymmetric eigenvalue algorithm with the computing power of GPU.

Acknowledgments We are grateful to the anonymous referees for reading our initial manuscript carefully and giving us many valuable comments. We would like to thank Professor Yoshimasa Nakamura for continuous support. This work is supported in part by the Ministry of Education, Science, Sports, and Culture through a Grant-in-Aid for Scientific Research.

References

[1] CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html.

[2] CULA. http://www.culatools.com.

[3] GPGPU.org. http://gpgpu.org/.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, 1992.

[5] Z. Bai, D. Day, J. W. Demmel, and J. J. Dongarra. A test matrix collection for non-Hermitian eigenvalue problems (release 1.0). Technical Report CS-97-355, Department of Computer Science, University of Tennessee, Knoxville, 1997.

[6] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[7] J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Software*, 14(1):1–17, 1988.

[8] J. J. Dongarra, J. D. Croz, S. van Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[9] Jack Dongarra, S. J. Hammarling, and D. C. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comput. Appl. Math.*, 27:215–227, 1989.

[10] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[11] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News - ISCA '10*, 38(3), 2010.

[12] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *Proceedings of IPDPS 2008*, pages 1–10, 2008.

[13] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *Selected Paper of VECPAR'2006*, number 4395 in Lecture Notes in Computer Science, pages 305–318. Springer-Verlag, 2007.

[14] R. Schreiber and C. F. van Loan. A storage-efficient WY representation for products of householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, 1989.

[15] S. Tomov and J. J. Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. LAPACK Working Notes 219, 2010.

[16] Y. Yamamoto. Performance modeling and optimal block size selection for a BLAS-3 based tridiagonalization algorithm. In *Proceedings of HPC-Asia 2005*, pages 249–256, Beijing, December 2005.