International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 2, pages 227–247, July 2017

An Efficient GPU Implementation of Bulk Computation of the Eigenvalue Problem for Many Small Real Non-symmetric Matrices[1]

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

Received: February 14, 2017
Revised: May 4, 2017
Accepted: June 2
Communicated by Akihiro Fujiwara

Abstract

The main contribution of this paper is to present an efficient GPU implementation of bulk computation of eigenvalues for many small, non-symmetric, real matrices. This work is motivated by the necessity of such bulk computation in designing of control systems, which requires to compute the eigenvalues of hundreds of thousands non-symmetric real matrices of size up to 30×30. Several efforts have been devoted to accelerating the eigenvalue computation including computer languages, systems, environments supporting matrix manipulation offering specific libraries/function calls. Some of them are optimized for computing the eigenvalues of a very large matrix by parallel processing. However, such libraries/function calls are not aimed at accelerating the eigenvalues computation for a lot of small matrices. In our GPU implementation, we considered programming issues of the GPU architecture including warp divergence, coalesced access of the global memory, utilization of the shared memory, and so forth. In particular, we present two types of assignments of GPU threads to matrices and introduce three memory arrangements in the global memory. Furthermore, to hide CPU-GPU data transfer latency, overlapping computation on the GPU with the transfer is employed. Experimental results on NVIDIA TITAN X show that our GPU implementation attains a speed-up factor of up to 83.50 and 17.67 over the sequential CPU implementation and the parallel CPU implementation with eight threads on Intel Core i7-6700K, respectively.

Keywords: GPGPU, CUDA, eigenvalue problem, bulk computation, QR algorithm

1 Introduction

Given an $n \times n$ matrix A, *the eigenvalue problem* is to find all eigenvalues λ satisfying

$$A\mathbf{x} = \lambda\mathbf{x},$$

[1]The preliminary version of this paper has been presented at the First International Workshop on GPU Computing and Applications (GCA16) held in conjunction with the Fourth International Symposium on Computing and Networking (CANDAR16) [38].

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 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. The computation of the eigenvalue problem for single matrix can be solved efficiently. However, the computation of eigenvalues for real matrices is a time-consuming task. For example, such issue occurs in the parameter space design method with volume rendering proposed in [34]. 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 using the result of the volume rendering. The black point in the figure shows one of the parameter sets visually selected by the designer. The parameter meets a condition in the target control system. This numerical method is expected to treat more practical specifications than the previous analytical method in [4]. For further details of this method, the interested reader may refer to [31] and the references within.

Control design problems are reduced to problems of finding a controller that satisfies design specifications of pole assignment, transient response, and frequency response. In [34], the method with rendering is studied for the specification of transient response. This method can also be adapted for pole assignment. It 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 many 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 large number of small, non-symmetric real matrices is of great interest.

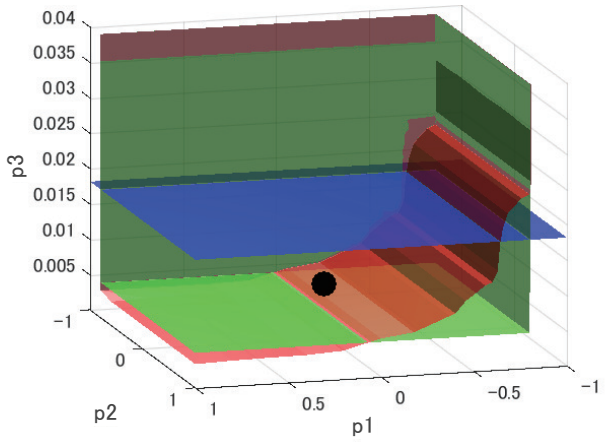


Figure 1: Volume rendering of the parameter space design in the pole assignment problems [34] using eigenvalues obtained by the proposed method

In classical numerical linear algebra, to compute eigenvalues of a non-symmetric matrix, the QR algorithm [10, 12] is usually employed. 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, variants have been proposed [11, 12]. Especially, in this work, we use *the implicit double-shift QR algorithm* [11] 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 applied.

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [17, 39]. 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 [17, 20, 21, 30, 40]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [28, 29], 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 [22], since they have thousands of processor cores and very high memory bandwidth.

The main contribution of this work is to propose a GPU implementation of bulk computation of eigenvalues of small, non-symmetric, real matrices of maximum size 30×30. Many works have been devoted to accelerating the eigenvalue computation and countless computer languages, systems, and environments supporting matrix manipulation offer libraries/function calls for this task. Some of them are optimized for computation of the eigenvalues of a very large matrix by parallel processing. However, such libraries/function calls are not aimed at accelerating the eigenvalues computation for a lot of small matrices. In the eigenvalue computation, several parallel algorithms have been proposed such as small bulge multi shift QR algorithm and two-tone QR sweep [6, 14]. These methods are to concurrently perform the iteration, called *QR sweep*, not to destroy the order of the iterations. Actually, in LAPACK [2], that is a linear algebra library, small bulge multi shift QR algorithm is employed [7]. These parallel algorithms can be applied to any size of matrices. However, the number of parallel executions for an $n \times n$ matrix is limited to at most n. Therefore, it is difficult for small matrix efficiently to utilize all processing cores on the CPU and the GPU. Also, since the computing time increases with the cube of n, the overhead cost of launching multiple threads cannot be ignored when the size of a matrix is small. Thus, in the existing software libraries/function calls, when the eigenvalue computation is executed for a small matrix, the sequential algorithm is used or the parallel algorithm is performed inefficiently. On the other hand, several fundamental operations in linear algebra that are often used for a large set of small matrices are supported by recent libraries. For example, MAGMA [18] supports parallel computation of matrix multiplication, LU factorization, and so forth. However, there is no libraries/function calls do not support eigenvalue computation for many small matrices. In our GPU implementation, we considered programming issues of the GPU architecture including warp divergence, coalesced access of the global memory, utilization of the shared memory, and so forth. We focused on the thread assignment to obtain the optimal parallel execution with many threads on the GPU. Apparently, running parallel threads as much as possible is an easy way to achieve high performance computation. However, this is not always correct due to various factors such as memory access latency and utilization of local registers [41]. Additionally, the optimal parameters including the number of threads and utilized shared memory differ among GPU architectures. To obtain optimal parameters automatically, auto-tuning techniques have been proposed [15, 23, 36]. Consequently, in this work, we propose two thread-assignment methods to perform the bulk execution of eigenvalues computation, *single-warp-based* (SWB) method and *multiple-warp-based* (MWB) method. In our GPU implementation, the optimal parameters have been obtained by evaluating the computation time for various parameters. Also, to improve the memory access efficiency, we introduce memory arrangements in the device memory on the GPU for each of the thread assignments. Furthermore, to hide CPU-GPU data transfer latency, overlapping computation on the GPU with the transfer is employed. We evaluated the performance of computing eigenvalues of 500000 matrices of size 5×5 to 30×30. The experimental results on NVIDIA TITAN X show that our GPU implementation attains a speed-up factor of up to 83.50 and 17.67 over the sequential CPU implementation and the parallel CPU implementation with eight threads on Intel Core i7-6700K, respectively.

Additionally, for ease of use, we have built an execution environment of our proposed GPU implementation in MATLAB. More specifically, we have made it possible for MATLAB users to utilize the bulk computation of eigenvalues with our proposed GPU implementation. MATLAB users who are not familiar with GPU computation can easily use it in a similar way to typical function calls in MATLAB.

This paper is organized as follows. Related work is summarized in Section 2. Section 3 introduces

the implicit double-shift QR algorithm for computing eigenvalues of a non-symmetric real matrix. Section 4 briefly describes about GPUs and CUDA architecture. In Section 5, our GPU implementation of the bulk computation of the eigenvalue problem for many small matrices is proposed. Experimental results are shown in Section 6. Finally, Section 7 concludes the paper.

2 Related work

Several works have been devoted to accelerate the computation for matrix calculations for many small matrices using GPUs [5, 8, 9, 16]. Anderson *et al.* [5] presented implementations of parallel computation of the LU decomposition and the QR decomposition for many small matrices on the GPU. In this paper, two parallel implementations have been proposed. The first implementation assigns one thread to each matrix and each thread performs the computation in a serial fashion. The second implementation assigns one thread block to each matrix and threads in a block perform the computation in parallel. In [16], a GPU implementation for the QR decomposition of many small dense matrices is presented. The GPU implementation reduces the memory access latency by increasing data locality. Dong *et al.* [9] proposed a GPU implementation of the LU decomposition with pivoting for many dense matrices. Also, Cosnuau [8] proposed a GPU implementation of computing eigenvalues for many small matrices. However, the GPU implementation can compute eigenvalues only for Hermitian matrices.

Software libraries and tools for numerical linear algebra are widely available and can be used to accelerate matrix multiplication. Indeed, GSL [1] and Intel MKL [19] are software libraries for the CPU implementations. These libraries support the computation of matrix factorizations, multiplications, eigenvalues, and so forth. For GPU implementations, MAGMA [18] are cuBLAS [25], cuSOLVER [26] are available. In cuBLAS, we can use to compute matrix factorizations and multiplications for a large matrix. Also, cuBLAS supports *the bulk computation* of matrix factorizations for many matrices, where the bulk computation is to compute a problem for many different inputs in turn or at the same time. However, it does not include the computation of the eigenvalue problem. Besides, cuSOLVER supports the computation of a pair of the maximum eigenvalue and eigenvector for a sparse matrix. However, it cannot be used for dense matrices and the computation of all eigenvalues. MAGMA [18] is a software library for heterogeneous computing platforms with multicore CPUs and GPUs. It supports the computation of eigenvalues for dense matrices and bulk computation of the computation of matrix multiplications and the LU decomposition for many matrices. Although it supports the bulk computation for many matrices, the computation of eigenvalues is not included. We can also utilize MATLAB [37] to compute the eigenvalue problem as a linear algebra software. MATLAB supports the computation of the eigenvalue problem for a matrix, but it does not support the bulk computation of the eigenvalue problem. The Intel MKL, MAGMA, and MATLAB that support eigenvalue computation. They select the optimum algorithm and parameters including the number of threads in parallel computation depending on the size of matrices.

3 Eigenvalues Computation of a Non-symmetric Real Matrix

This section reviews the QR algorithm to compute the eigenvalues of a matrix [12]. 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* [11, 35]. 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 [11, 12, 35] 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: Directly compute eigenvalues of the matrices of size 1×1 or 2×2

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. In other words, an $n \times n$ matrix $A = a_{i,j}$ $(1 \leq i, j \leq n)$ such that $a_{i,j} = 0$ $(i > j + 1)$ is upper Hessenberg form.

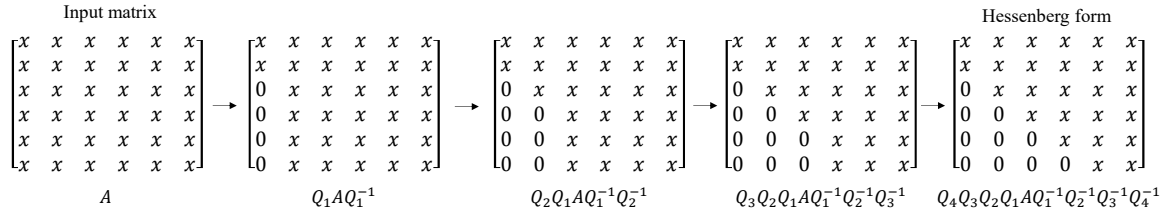


Figure 2: The Hessenberg reduction for a square matrix of size 6×6

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 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

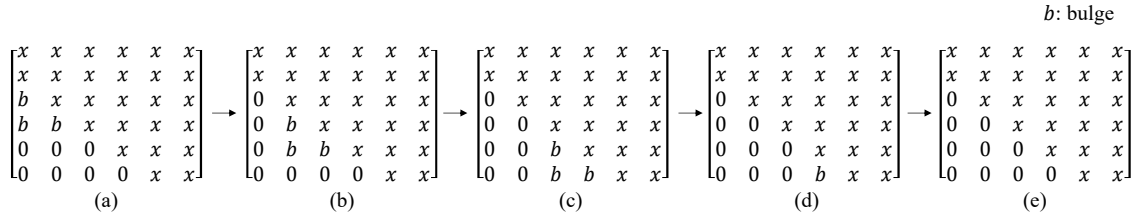


Figure 3: Bulge-generating and bulge-chasing in the double-shift QR sweep

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. More specifically, a subdiagonal element $a_{k+1,k}$ $(1 \leq k \leq n-1)$ converges to zero when $|a_{k+1,k}| \ll |a_{k,k}| + |a_{k+1,k+1}|$.

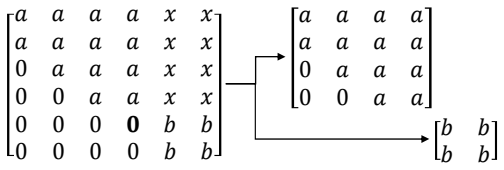


Figure 4: Matrix division by deflation

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.

Before the explanation in details about the above steps, we introduce *Householder transformation* and *similarity transformation* to be used in matrix transformations in the Hessenberg reduction and the double-shift QR sweep. Householder transformation is a linear transformation defined by a Householder matrix Q in the following equations;

$$\begin{aligned} Q &= I - 2\mathbf{v}\mathbf{v}^T \\ \mathbf{v} &= \frac{\mathbf{x}-\mathbf{y}}{||\mathbf{x}-\mathbf{y}||} \end{aligned}$$

where $\mathbf{x}$ and $\mathbf{y}$ are two distinct vectors such that $||\mathbf{x}|| = ||\mathbf{y}||$, I is a unit matrix, and $\mathbf{v}$ is called a Householder vector. The Householder matrix obtained by the above is symmetric and orthogonal. Namely, for a Householder matrix Q, we have $Q = Q^{-1} = Q^T$. On the other hand, similarity transformation is a transformation such that a square matrix A is transformed by $A \leftarrow BAB^{-1}$, where B is a regular matrix. The characteristic of this transformation is holding the eigenvalues before and after the transformation. In the following, we use Householder transformation to transform a matrix by multiplying the Householder matrix from left. However, the eigenvalues are changed by the transformation. Therefore, after that, to hold the eigenvalues, similarity transformation is applied to the matrix by multiplying the Householder matrix from right. We note that since a Householder matrix is identical to its inverse matrix, it is not necessary to compute the inverse matrix to perform similarity transformation. In the following, we explain the details of each step with Householder transformation and similarity transformation.

3.1 The Hessenberg Reduction

The Hessenberg reduction transforms a square matrix A of size $n \times n$ to a Hessenberg form matrix H. In the Hessenberg reduction, we change the values of elements below the subdiagonal to zero from left to right as shown in Figure 2. More specifically, an input matrix A is reduced to the Hessenberg form matrix H by Householder transformations from left and similarity transformations from right:

$$H = Q_{n-2}Q_{n-3}\cdots Q_2Q_1AQ_1^{-1}Q_2^{-1}\cdots Q_{n-3}^{-1}Q_{n-2}^{-1},$$

where each Q_k $(1 \leq k \leq n-2)$ is a Householder matrix to change the values of elements below the subdiagonal in k-th column to zero by Householder transformation and similarity transformation. This transformation matrix Q_k can be computed only from the elements in k-th column of A. The reader can find that each Q_k is identical to the identity matrix except the $(n-k)\times(n-k)$ sub-matrix at the right-bottom elements.

Algorithm 1 shows the Hessenberg reduction by Householder transformation and similarity transformation, where $\mathbf{v}_k$ is a Householder vector for k-th column. Let $A_{a:b,c:d}$ denote the sub-matrix of A of which the top-left 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. In this algorithm, we transform the input matrix such that the values of elements below the subdiagonal are changed to zero from left to right as shown in Figure 2. Since each Q_k is identical to the identity matrix except the $(n-k)\times(n-k)$ sub-matrix at the right-bottom elements in Q_k, we apply a Householder vector $\mathbf{v}$ without directly multiplying a Householder matrix Q_k. In lines 2–4, a Householder vector $\mathbf{v}$ is computed. Using $\mathbf{v}$, Householder transformation and similarity transformation are performed in lines 5 and 6, by multiplying $\mathbf{v}$ from left and right, respectively. In these two transformations, we compute only the elements of which values have changed. After performing the above operations for $k = 1, \ldots, n-2$, we obtain the Hessenberg form matrix of the input matrix.

3.2 The double-shift QR sweep

The double-shift QR sweep first makes an initial transformation that produces a bulge at the top in a Hessenberg form matrix (*bulge-generating*). After that, it sweeps the matrix from the top to bottom by chasing the bulge (*bulge-chasing*). Once the sweep is finished, the matrix is returned to

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: $\mathbf{v} \leftarrow A_{k+1:n,k}$
3: $\mathbf{v} \leftarrow \mathbf{v} + \mathrm{sign}(v_1)||\mathbf{v}||\mathbf{e}_1$
4: $\mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||}$ ▷ Householder vector
5: $A_{k+1:n,k:n} \leftarrow A_{k+1:n,k:n} - 2\mathbf{v}(\mathbf{v}^T A_{k+1:n,k:n})$ ▷ Householder transformation
6: $A_{1:n,k+1:n} \leftarrow A_{1:n,k+1:n} - 2(A_{1:n,k+1:n}\mathbf{v})\mathbf{v}^T$ ▷ Similarity transformation
7: **end for**
8: **return** $H \leftarrow A$

upper Hessenberg form. By repeating the sweep, some element in subdiagonal converges to zero. However, due to a computational error, the value may not become zero exactly. Therefore, if a subdiagonal element becomes sufficiently small, we divide the matrix by deflation.

In bulge-generating, first, a 2×2 sub-matrix from the lower-right corner of H is extracted and its two eigenvalues σ_1 and σ_2 are computed. After that, we obtain a Householder matrix R_0 such that the first column of $(H - \sigma_1 I)(H - \sigma_2 I)$ except the diagonal element is introduced to zero. This operation is used to reduce the number of iterations of the sweeps. The Householder matrix R_0 is an $n \times n$ matrix such that

$$R_0 = \left[\begin{array}{ccc|ccc} r_{1,1} & r_{1,2} & r_{1,3} & & & \\ r_{2,1} & r_{2,2} & r_{2,3} & & O & \\ r_{3,1} & r_{3,2} & r_{3,3} & & & \\ \hline & & & 1 & & \\ & O & & & \ddots & \\ & & & & & 1 \end{array}\right]$$

Since R_0 is a Householder matrix, we have $R_0^{-1} = R_0$. Therefore, we perform Householder transformation and similarity transformation to H by multiplying R_0 from left and right, respectively. The structure of B obtained by these transformations is Hessenberg form with three non-zero elements in $b_{3,1}$, $b_{4,1}$, and $b_{4,2}$, called a *bulge*, as shown in Figure 3(a). Algorithm 2 shows bulge-generating by Householder transformation and similarity transformation, where $\mathbf{v}$ is a Householder vector. In Algorithm 2, a Householder vector $\mathbf{v}$ is computed in lines 1–5. We directly compute only the values to be used in the following computation without the computation of eigenvalues σ_1 and σ_2, and the matrix multiplication $(H - \sigma_1 I)(H - \sigma_2 I)$. We also apply the Householder vector $\mathbf{v}$ without directly multiplying a Householder matrix R_0 in the same way as Algorithm 1. After that, Householder transformation and similarity transformation are performed in lines 6 and 7, by multiplying $\mathbf{v}$ from left and right, respectively. After the above operations, we obtain the Hessenberg form matrix with a bulge. We note that if the size of the input matrix is 3×3, the range of the sub-matrix is beyond the size of the matrix. The elements out of the matrix need not to be computed.

Bulge-chasing sweeps the matrix obtained by bulge-generating from the top to bottom by chasing the bulge as shown in Figure 3. To chase the bulge, we perform Householder transformations from left and similarity transformations from right:

$$H = R_{n-2}R_{n-3}\cdots R_2 R_1 B R_1^{-1} R_2^{-1} \cdots R_{n-3}^{-1} R_{n-2}^{-1},$$

where each R_k $(1 \leq k \leq n-2)$ is a Householder matrix to move the bulge to the bottom one by

Algorithm 2 Bulge-generating

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

one. Each Householder matrix R_k is a square matrix of size $n \times n$ such that

$$R_k = \left[\begin{array}{c|ccc|c} 1 & & & & \\ & \ddots & & O & O \\ & & 1 & & \\ \hline & & r_{k+1,k+1} & r_{k+1,k+2} & r_{k+1,k+3} & \\ O & & r_{k+2,k+1} & r_{k+2,k+2} & r_{k+2,k+3} & O \\ & & r_{k+3,k+1} & r_{k+3,k+2} & r_{k+3,k+3} & \\ \hline & & & & 1 & \\ O & & O & & \ddots & \\ & & & & & 1 \end{array}\right]$$

Since R_k is a Householder matrix, we perform Householder transformation and similarity transformation to B by multiplying R_k from left and right, respectively. Algorithm 3 shows bulge-chasing by Householder transformation and similarity transformation, where $\mathbf{v}$ is a Householder vector. In Algorithm 3, a Householder vector $\mathbf{v}$ is computed in lines 2–4. We also apply Householder vectors $\mathbf{v}$ without directly multiplying Householder matrices R_k in the same way as bulge-generating. Householder transformation and similarity transformation in lines 5 and 6 are performed by multiplying $\mathbf{v}$ from left and right, respectively. After the above operations, we obtain the Hessenberg form matrix without the bulge. We note that when the size of the input matrix is 3×3 and/or $k \geq n - 3$, the range of the sub-matrix is beyond the size of the matrix. In such case, the computation of the elements out of the matrix is skipped.

Algorithm 3 Bulge-chasing

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

4 CUDA Architecture

NVIDIA provides a parallel computing architecture, called CUDA, on NVIDIA GPUs. Figure 5 illustrates the CUDA hardware architecture. CUDA uses three types of memories: *the global memory*,

the shared memory and *the register file*. 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. Also, the register file is on chip memory and the fastest memory among them. Registers in the register file can be accessed to a processor core, and they can be accessed only by a thread running on the processor core.

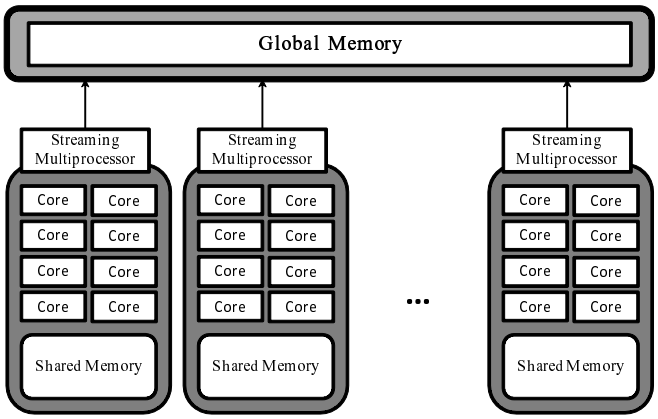


Figure 5: CUDA hardware architecture

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. However, as we can see in Figure 5, 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 threads, 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 go 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.

There is a metric, called *occupancy*, related to the number of active warps on an SM. The occupancy is the ratio of the number of active warps per SM to the maximum number of possible active warps. It is important in determining how effectively the hardware is kept busy. The occupancy depends on the numbers of threads and blocks and the size of the shared memory used in a block. Namely, utilizing over utilizing resources per thread or block may limit the occupancy. To obtain good performance with the GPUs, occupancy should be considered.

CUDA C extends C language by the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in SMs, and threads in each block are executed by processor cores in a single SM. The kernel calls terminate, when threads in all blocks finish the computation. Since all threads in a single block are executed a single SM, the barrier synchronization of them can be done by calling CUDA C `syncthreads()` function. However, there is no direct

way to synchronize threads in different blocks. One of the indirect methods of inter-block barrier synchronization is to partition the computation into kernels. Since continuous kernel calls can be executed such that a kernel is called after all blocks of the previous kernel terminate, execution of blocks is synchronized at the end of kernel calls. On the other hand, all threads of a warp perform the same instruction at the same time. More specifically, any synchronizing operations are not necessary to synchronize threads within a warp.

To accelerate the computation, the coalesced access to the global memory is a key issue. As illustrated in Figure 6, when threads access to continuous locations in a row of a 2-dimensional array (*horizontal access*), the continuous locations in address space of the global memory are accessed at the same time (*coalesced access*). However, if threads access to continuous locations in a column (*vertical access*), the distant locations are accessed at the same time (*stride access*). From the structure of the global memory, the coalesced access maximizes the bandwidth of memory access. On the other hand, the stride access needs a lot of clock cycles. Thus, we should avoid the stride access (or the vertical access) and perform the coalesced access (or the horizontal access) whenever possible.

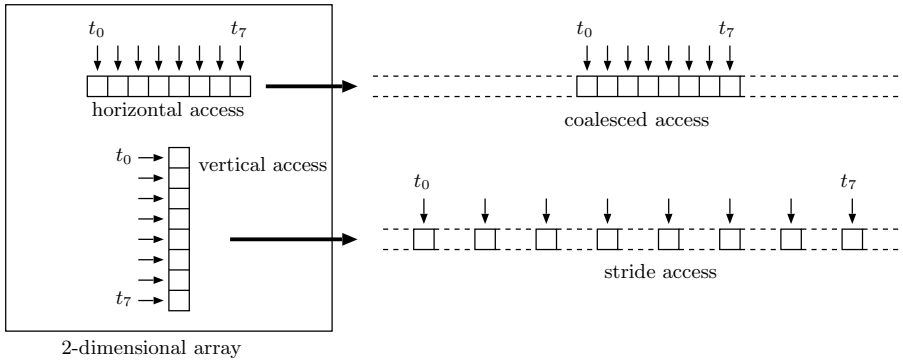


Figure 6: Coalesced and stride access

The shared memory is a sort of on-chip memory located within each SM. It has almost no access latency and only visible to the thread block which is executed by the corresponding SM. Therefore, the shared memory is often used as a cache to hide the access latency of the global memory.

In typical computation using the GPU, input data is transferred from the main memory on the host PC to the global memory on the GPU. Also, the computation results are copied back to the main memory. In general, the data transfer is performed via PCI Express. However, the throughput of PCI Express is not sufficiently high compared with that of the global memory on the GPU. For example, the throughput of PCI Express 3.0 (x16) that the recent GPUs support is up to 15.75GB/s [31]. On the other hand, the throughput of the global memory is up to 480GB/s [27]. Therefore, the data transfer time cannot be ignored occasionally when computing time is shorter than data transfer time.

To reduce CPU-GPU communication overhead, recent CUDA-enabled GPUs can perform an asynchronous memory copy to or from the GPU concurrently with kernel execution [13, 32]. The asynchronous memory copy is to hide CPU-GPU transfer latency by overlapping computation on the GPU with the transfer. CUDA defines a *stream* as a sequence of operations that are guaranteed to sequentially execute on the GPU. In general, a stream consists of memory copy of input data from host to device (H2D), execution of kernels (Computation), and memory copy of the results from device to host (D2H). Operations in different streams can be interleaved and concurrently run whenever possible as illustrated in Figure 7. In our approach, described in the next section, since eigenvalues of many matrices are computed, the data communication overhead is not small. Therefore, to hide the overhead, we apply this idea to our GPU implementation.

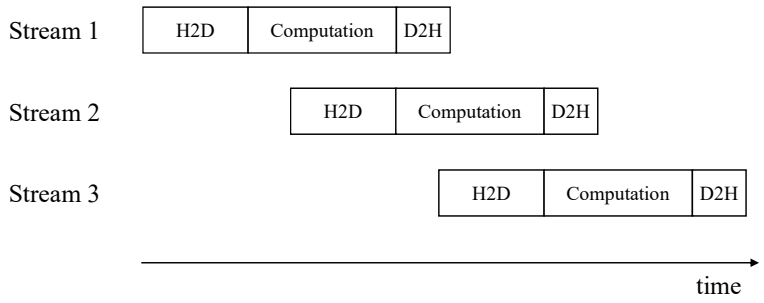


Figure 7: Overlapped execution on the GPU using multiple streams

5 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. In the following, let N be the number of input matrices and each size of matrix is $n \times n$. 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 8(a). This arrangement is generally used in numeric linear algebra tools and software libraries [18, 19, 37]. Therefore, 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 8(b). On the other hand, in the RW arrangement, each row taken from the matrices is stored row by row as illustrated in Figure 8(c). Two arrangements EW and RW are used in the global memory to make the memory access efficient.

Recall that according to Algorithms 1, 2, and 3, these algorithms mainly consist of Householder vector generation, Householder transformation, and similarity transformation. We consider that those computations are carried out on the GPU. A GPU can employ many threads working concurrently. Apparently, running parallel threads as much as possible is the easiest way to achieve high performance computation. However, this is not always correct due to various factors such as memory access latency and utilization of local registers [41]. Additionally, the optimal parameters such as the number of threads differ among GPU architectures. To obtain optimal parameters automatically, auto-tuning techniques have been proposed [15, 23, 36]. Consequently, in this work, we propose two thread-assignment methods to perform the bulk execution of eigenvalues computation, *single-warp-based* (SWB) method and *multiple-warp-based* (MWB) method. The idea of our approach is that the better method is selected step by step, and the best number of threads that compute one matrix is utilized by evaluating the computation time with various conditions. We explain these two methods using the above three data arrangements as follows.

SWB method In the SWB method, every warp is used to compute eigenvalues of one or more matrices as shown in Figure 9. More specifically, we allocate p ($1 \leq p \leq n$) threads to one matrix and every warp works for $\lfloor \frac{32}{p} \rfloor$ matrices in parallel. In this method, when $\frac{32}{p}$ is indivisible, $32 - p\lfloor \frac{32}{p} \rfloor$ threads in every warp are not employed. For example, when 7 threads are used to compute each matrix, in each warp, $\lfloor \frac{32}{7} \rfloor = 4$ matrices are computed 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 number of matrices for each warp is smaller than the MWB method, only in this method, all data of matrices can be located on the shared memory. Therefore, first of all the processes in

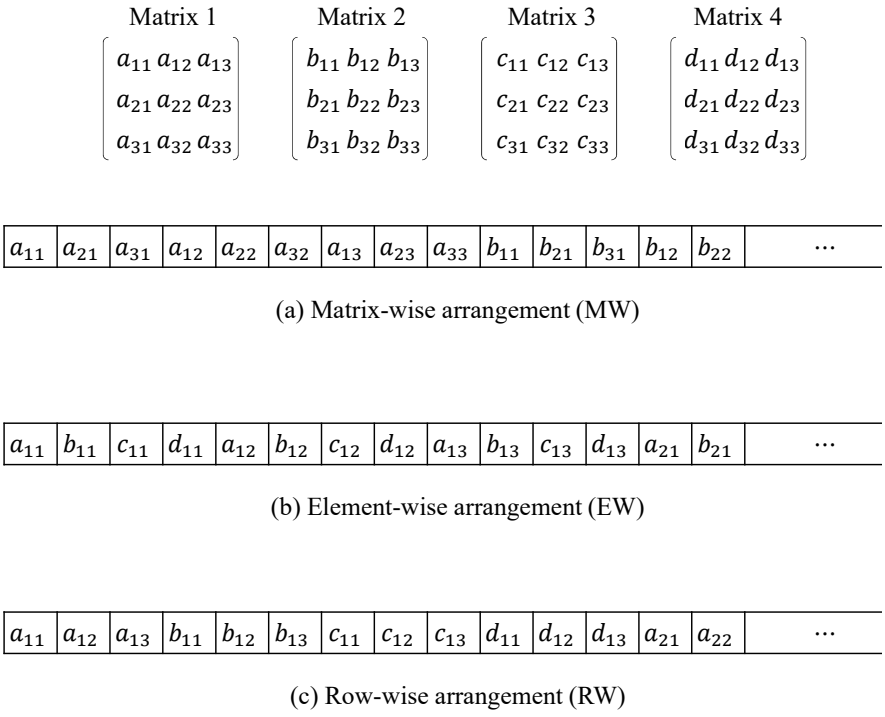


Figure 8: Data arrangement for multiple matrices in the global memory

this method, the data of matrices are loaded from the global memory to the shared memory. To make the access to the global memory coalesced, the matrix data loading from the global memory to the shared memory is performed by multiple threads as illustrated in Figure 10. We note that the eigenvalues computation of one matrix is performed by p threads. However, if the data loading is performed for each matrix using p threads, the memory access is not coalesced when $p \neq n$. Therefore, the data loading is performed regardless the number of p as shown in Figure 10. The following computation is performed on the shared memory until the resulting eigenvalues are stored to the global memory.

In Step 1, this method performs the computation as follows. To obtain $||\mathbf{v}||$, the sum of squared values of $\mathbf{v}$ is computed. We use the parallel sum reduction method [24] on the shared memory. After that, the Householder vector is computed by one thread and stored to the shared memory. In Householder transformation, we assign p threads to one column each, and repeat it until all columns are computed as illustrated in Figure 11(a). Namely, the computation of the transformation is concurrently performed using p threads. In the parallel computation, each thread computes the multiplication of the elements in the assigned column from top to bottom. In similarity transformation, we assign p threads to one row each, and repeat it until all rows are computed as illustrated in Figure 11(b). Each thread computes the multiplication of the elements in the row from left to right.

On the other hand, in Step 2, the sum of only three squared values is computed to obtain the Householder vector. Therefore, the sum is directly computed instead of the parallel sum reduction method unlike Step 1. After that, in Householder transformation and similarity transformation, p threads are assigned to columns and rows, and work in the same way as Step 1. If a matrix is divided into smaller matrices by deflation, bulge-generating and bulge-chasing are repeatedly performed using p threads for each smaller matrix.

After all the matrices divided by deflation becomes 1×1 or 2×2, one thread computes eigenvalues of the matrices in serial. The resulting eigenvalues are temporarily stored to the shared memory. After that, all the threads within the warp store them to the global memory with the MW

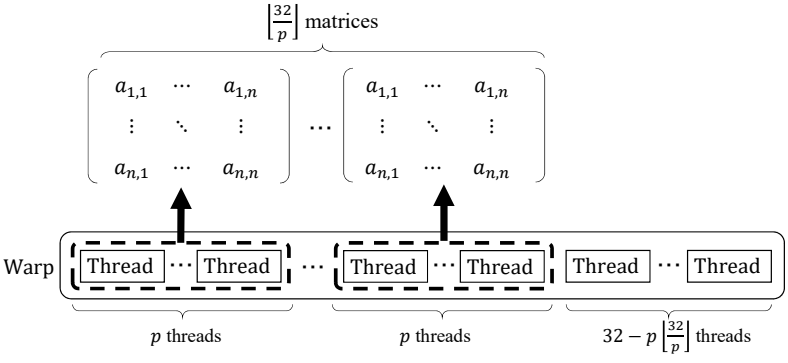


Figure 9: Single-warp-based method (SWB)

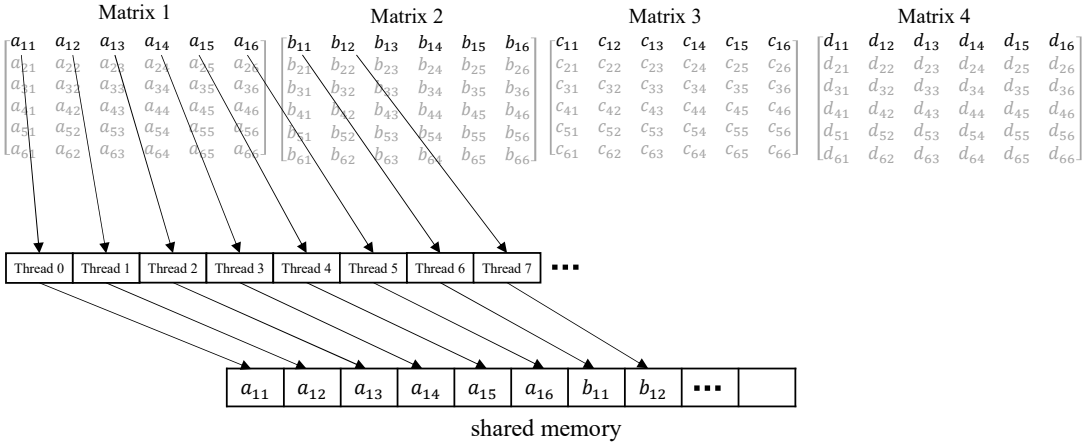
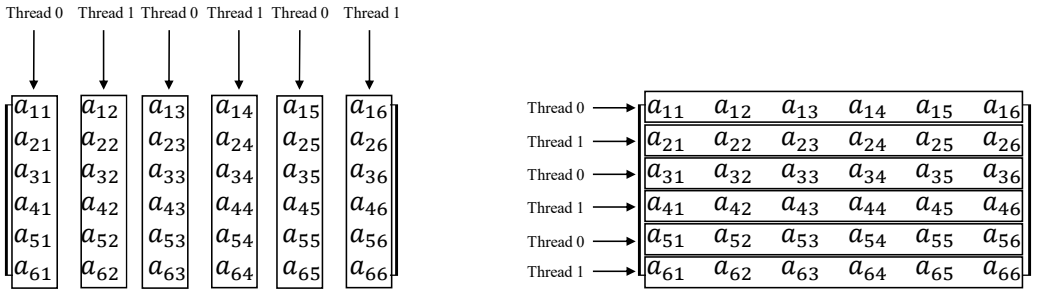


Figure 10: Matrix data loading from the global memory to the shared memory with coalesced access in SWB

arrangement using coalesced access.

MWB method In the MWB method, p ($1 \leq p \leq n$) warps are used to compute eigenvalues of 32 matrices as illustrated in Figure 12. More specifically, we allocate p threads in p different warps to one matrix computation. Each matrix is computed in parallel using p threads each of which is in p distinct warps. Since only the number of warps is depended on p, all threads in a warp are employed for any p unlike the SWB method. The parallel execution by p threads in the MWB method is the same as that in the SWB method except that the execution is basically performed on the global memory. Regarding the parallel execution with p threads, the thread-assignment and the computation are the same as the SWB method illustrated in Figure 11. Also, since multiple warps are used, it is necessary to synchronize the execution between warps using `syncthreads()` function. However, although multiple warps cooperate, the function is not frequently called. The synchronization is performed in Householder vector generation several times. After that, the execution needs to be synchronized only at the end of Householder transformation and similarity transformation each. Additionally, in this method, to access the global memory with coalesced access, we arrange data in the global memory using the EW arrangement.

We assume that the input data of matrices are stored in 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 two 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



(a) Householder transformation

(b) Similarity transformation

Figure 11: Thread assignment in SWB and MWB for $n = 6$ and $p = 2$

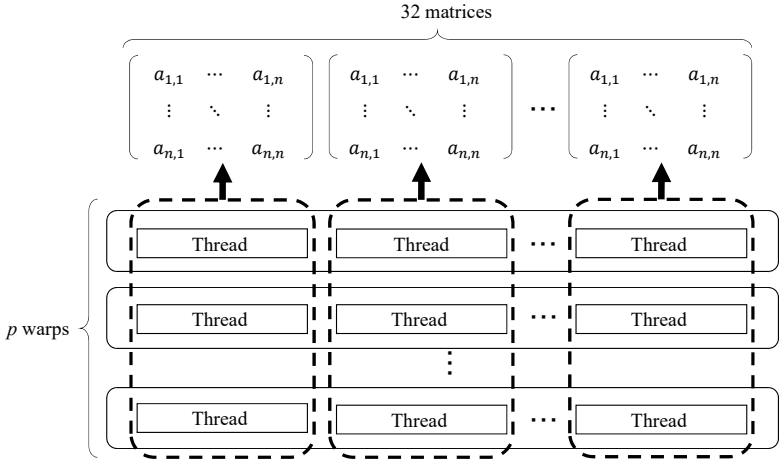


Figure 12: Multiple-warp-based method (MWB)

technique proposed in [33]. 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. In the next section, we evaluate the processing time of the rearrangement.

We note that in our preliminary work [38], we proposed another thread-assignment method, called *single-thread-based* (STB) method. This method is to assign each thread to one matrix. Each thread computes eigenvalues of the matrix in serial. Indeed, the STB method is the special case of the MWB method for $p = 1$. Therefore, in this paper, the STB method is included in the MWB method.

In our problem, the bulk computation of eigenvalues problem, we compute eigenvalues for a lot of matrices. Since the matrices are independent from each other, we can easily compute eigenvalues in parallel such that several streams are invoked and each stream computes eigenvalues of part of the matrices. Using such parallel computation with multiple streams, we hide CPU-GPU transfer latency by overlapping computation on the GPU with the transfer as described in Section 4.

6 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 TITAN X, which has 3584 cores in running on 1.531GHz [27]. Also, we have used Intel Core i7-6700K running on 4.2GHz, which has 4 physical cores each of which acts 2 logical cores by hyper-threading technology, on the

host PC. In the following, the running time is average of 10 times execution of computing eigenvalues for 500000 matrices of size from 5×5 to 30×30 that are dense matrices randomly generated.

First, we evaluate the performance of Step 1, the Hessenberg reduction, using the proposed SWB and MWB methods. Figure 13 shows the computing time of the Hessenberg reduction. The evaluation has been carried out for different values of p. We note that when p is small, the SWB method cannot be executed due to the limitation of the shared memory. 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 for each method as shown in Figure 8. Namely, we use the EW arrangement for the MWB method and the RW arrangement for the SWB method. Furthermore, in the SWB method, when p is small, the size of utilized shared memory in a warp is large since the number of matrices in a warp is large. Hence, for larger than 10×10 matrices, we could not evaluate the computing time due to the limitation of size of the shared memory when p is small. This limitation is applied to the next evaluation of Steps 2 and 3.

According to the graphs, the SWB method is faster for 15×15 or larger matrices on the whole, since the computation in the SWB method is carried out on the shared memory. On the other hand, since the memory access is not performed frequently for small matrices, the benefit of the computation on the shared memory is very small. Therefore, since the cost of the memory copy to the shared memory cannot be ignored, the computing time of the SWB method is longer than that of the MWB. In addition, when p is small in the SWB method, the number of matrices to be computed in one warp is large. Due to the large amount of used shared memory, the occupancy decreases. Therefore, the computing time of the SWB method is long when p is small.

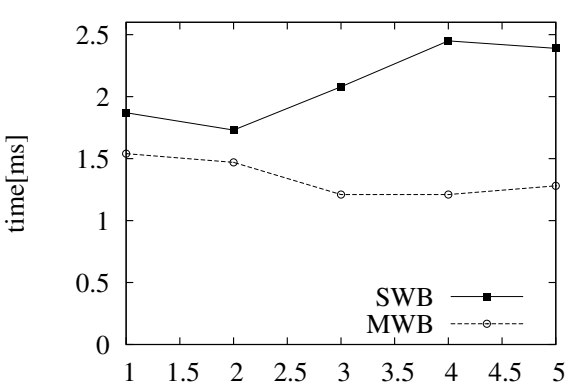
Figure 14 shows the computing time of Steps 2 and 3 for 500000 matrices of size from 5×5 to 30×30. Similarly, the computing time does not include data transfer time. Also, input data in the global memory are stored by the appropriate arrangement in Figure 8 for each method.

According to the graphs, the MWB method is faster than the SWB in most cases. This is because in Step 2, the number of active threads becomes small whenever matrices are divided by deflation and their size becomes small. In the SWB method, although the number of active threads is small, at least one thread in every warp is always active due to the assignment of threads. On the other hand, in the MWB method, although the number of active threads assigned to one matrix computation is small, it is possible that every thread in several warps becomes inactive, that is, no warp divergence occurs in such warps. Therefore, the warp divergence in the SWB method occurs more frequently than that in the MWB method. Thus, the MWB method is faster than the SWB in Step 2.

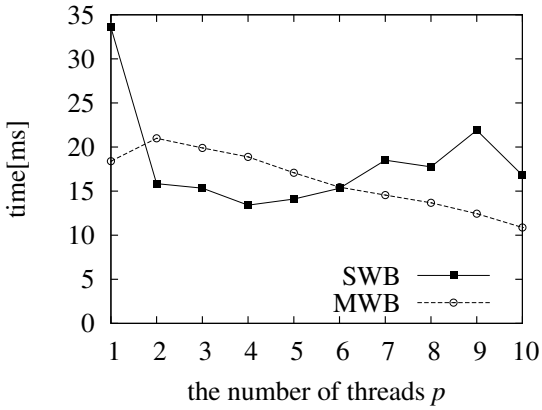
Table 1 shows the computing time of our GPU implementation for 500000 matrices of size $n \times n$. We assume that all input data are stored in the main memory on the host PC using the data arrangement in the MW arrangement. The input data are transferred from the main memory on the CPU to the global 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.

As regards the data transfer time between the CPU and the GPU, you can find that it is not small from the table. Especially, when the size of matrix is small, the data transfer time accounts for more than half of the total computing time. Therefore, we evaluate the overlapped execution by multiple streams shown in Section 4. Table 2 shows the computing time of eigenvalues for 500000 matrices using multiple streams when each stream computes eigenvalues of 1024 to 262144. According to the table, the running time is long when the number of matrices per stream is both small and large. This is because the overlapped execution is small since the computation time and the data transfer time is larger than the other, respectively. On the other hand, if the optimal number of matrices is selected, almost data transfer time can be hidden from Tables 1 and 2. According to the results, the computation time is reduced by approximately 25% to 59% using multiple streams. In the following, the GPU implementation selects the optimal number of matrices obtained by this evaluation.

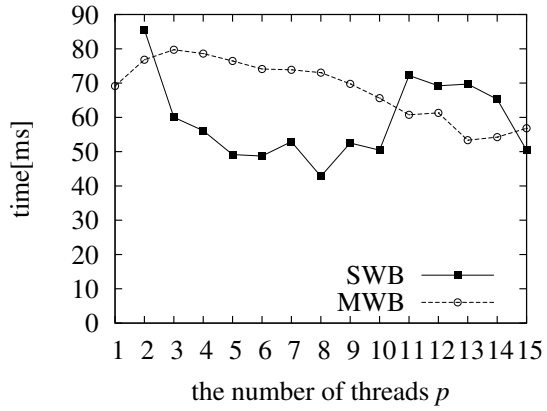
To compare the performance of our method, we have evaluated the computation time of Intel



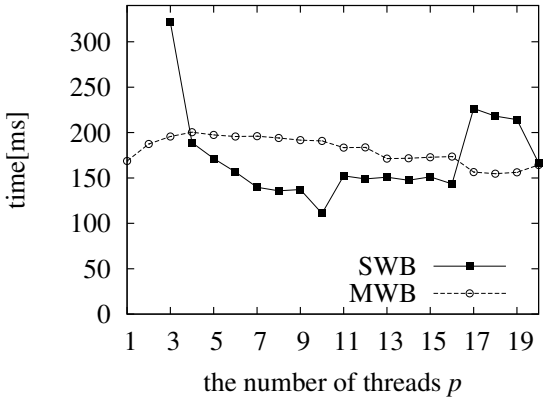
(a) 5×5



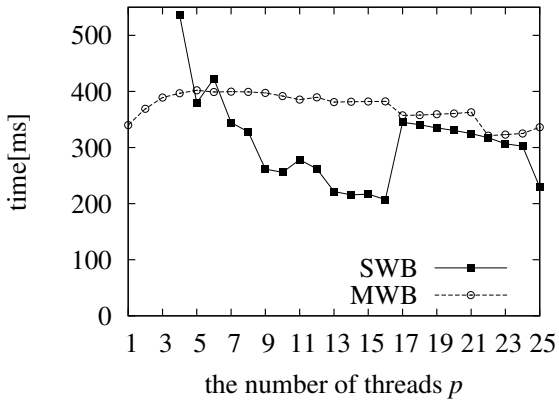
(b) 10×10



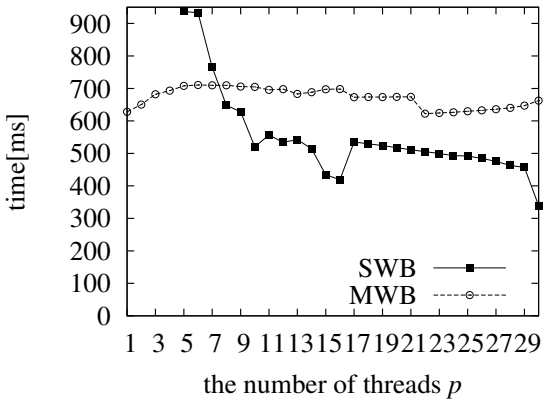
(c) 15×15



(d) 20×20

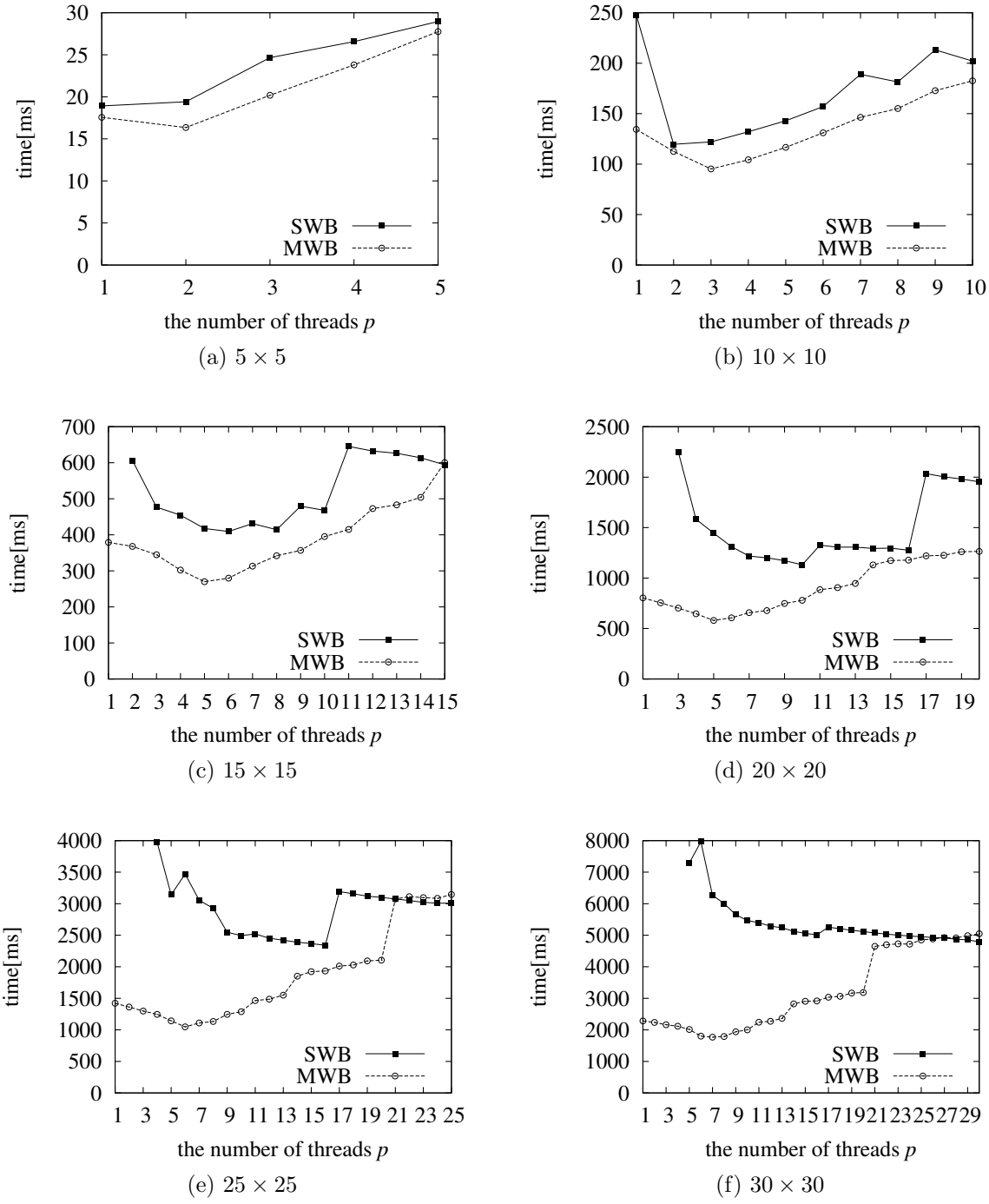


(e) 25×25



(f) 30×30

Figure 13: The computing time of Step 1 for 500000 matrices of size 5×5 to 30×30



(a) 5×5 (b) 10×10

(c) 15×15 (d) 20×20

(e) 25×25 (f) 30×30

Figure 14: The computing time of Steps 2 and 3 for 500000 matrices of size 5×5 to 30×30

Table 1: The computing time (in milliseconds) of our GPU implementations of the eigenvalue problem for 500000 matrices

size		5×5	10×10	15×15	20×20	25×25	30×30
data transfer (host to device)		21.22	80.81	178.15	314.37	494.52	714.42
data rearrangement	time	0.64	2.54	5.89	10.03	15.00	21.65
	arrange	MW→EW	MW→EW	MW→RW	MW→RW	MW→RW	MW→RW
Step 1	time	1.21	10.87	42.80	110.83	207.23	339.44
	method	MWB	MWB	SWB	SWB	SWB	SWB
data rearrangement	time	—	—	6.27	10.39	15.36	22.03
	arrange	—	—	RW→EW	RW→EW	RW→EW	RW→EW
Steps 2 and 3	time	16.35	95.20	269.98	580.21	1048.09	1767.47
	method	MWB	MWB	MWB	MWB	MWB	MWB
data rearrangement	time	0.23	0.46	0.68	0.91	1.14	1.36
	arrange	EW→MW	EW→MW	EW→MW	EW→MW	EW→MW	EW→MW
data transfer (device to host)		9.65	17.60	27.30	35.99	44.37	59.74
total		49.30	207.48	531.07	1062.73	1825.72	2926.11

Table 2: The computing time (in milliseconds) of eigenvalues for 500000 matrices using multiple streams each of which computes k matrices

k	5×5	10×10	15×15	20×20	25×25	30×30
1024	46.79	189.74	414.11	863.00	1498.60	2479.38
2048	33.16	156.94	369.96	804.59	1448.53	2350.33
4096	25.53	140.92	343.55	764.63	1337.03	2194.36
8192	22.64	132.22	338.79	758.22	1374.25	2191.16
16384	20.89	128.85	347.84	757.34	1367.70	2232.39
32768	20.01	126.85	350.80	751.42	1342.78	2215.02
65536	24.08	125.61	347.13	766.55	1568.12	2586.04
131072	27.00	150.85	406.32	870.03	1590.94	2627.18
262144	29.51	159.82	423.45	902.12	1649.10	2713.23

MKL 2017 Update 1 [19], MAGMA version 2.0.2 [18], and MATLAB version R2016b [37]. Table 3 shows the computing time of eigenvalues for 500000 matrices using existing tools and software libraries. Intel MKL supports two types of implementation, sequential and parallel execution for

Table 3: The computing time (in milliseconds) of eigenvalues for 500000 matrices using existing tools and software libraries

	5×5	10×10	15×15	20×20	25×25	30×30
Intel MKL (sequential)	1670.56	5765.72	13610.48	22867.30	38025.77	53680.96
Intel MKL (parallel)	1867.66	6588.52	14953.56	25720.05	41688.86	59750.73
Intel MKL (sequential+OpenMP)	353.56	1214.61	2854.39	4875.80	7915.52	11442.21
MAGMA	192925.18	227177.54	236115.41	270104.45	270104.45	371107.56
MATLAB	2663.68	7026.19	30586.44	54662.29	90959.14	127717.43

computing eigenvalues of a matrix. In the sequential execution, one thread is invoked and the thread computes eigenvalues in serial. On the other hand, the parallel execution, eigenvalues of one matrix are computed using multiple threads. According to the table, the parallel execution is slower than the sequential execution though multiple threads are used. This is because in the parallel execution, whenever eigenvalues are computed for each matrix, threads are invoked. Since the size of matrices is too small, the overhead caused by invoking threads cannot be ignored compared with the computing time of eigenvalues. Due to the result, we have implemented another parallel execution using Intel MKL and OpenMP 2.0 [3]. In the parallel execution, since a lot of matrices are computed, we have parallelized the bulk execution such that multiple threads are invoked and each thread performs the sequential execution in parallel using OpenMP. The behavior of each thread is equivalent to the thread in the sequential implementation. In the table, Intel MKL (sequential+OpenMP) corresponds to this parallel execution using OpenMP. In the evaluation, we have used 8 threads since the utilized

CPU has 8 logical cores.

On the other hand, MAGMA and MATLAB support parallel computation of the eigenvalue problem with multi-threads on the CPU. MAGMA also supports parallel computation on the GPU. However, since the size of matrices is too small in this experiment, MAGMA automatically selected the CPU execution without the GPU. Actually, since MAGMA basically expects computation on the GPU, whenever functions of MAGMA are called, some overhead to the GPU is necessary. Therefore, MAGMA is much slower than Intel MKL. Furthermore, since Intel MKL, MAGMA and MATLAB do not support bulk computation of the eigenvalue problem, a procedure that computes eigenvalues is called for each matrix. Due to such execution, multiple threads are launched and stopped before and after each procedure call, respectively. Therefore, there is an overhead between each procedure call and it is not negligible.

Table 4 shows the comparison between CPU sequential and parallel implementations and our GPU implementation. In the GPU implementation, the appropriate parameters in Tables 1 and 2 have been selected. According to the table, our GPU implementation attains a speed-up factor of up to 83.50 and 17.67 over the sequential CPU implementation and the parallel CPU implementation, respectively.

Table 4: The total computing time (in milliseconds) of eigenvalues for 500000 matrices

	5×5	10×10	15×15	20×20	25×25	30×30
CPU1 (Intel MKL sequential)	1670.56	5765.72	13610.48	22867.30	38025.77	53680.96
CPU2 (Intel MKL+OpenMP)	353.56	1214.61	2854.39	4875.80	7915.52	11442.21
GPU (proposed method)	20.01	125.61	338.79	751.42	1337.03	2191.16
speed-up (CPU1/GPU)	83.50	45.90	40.17	30.43	28.44	24.50
speed-up (CPU2/GPU)	17.67	9.67	8.43	6.49	5.92	5.22

7 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 utilization of the shared memory. We proposed two assignments of GPU threads to compute in parallel and data assignment in the global memory. Furthermore, to hide CPU-GPU data transfer latency, overlapping computation on the GPU with the transfer has been employed. The experimental results on NVIDIA TITAN X show that our GPU implementation attains a speed-up factor of up to 83.50 and 17.67 over the sequential CPU implementation and the parallel CPU implementation with eight threads on Intel Core i7-6700K, respectively. Additionally, for ease of use, we have built an execution environment for our proposed GPU implementation in MATLAB.

References

[1] *GSL–GNU Scientific Library*. `http://www.gnu.org/software/gsl/`.

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

[3] OpenMP. `http://www.openmp.org/`.

[4] Jürgen Ackermann. *Robust Control: The Parameter Space Approach*. Communications and Control Engineering. Springer-Verlag London, second edition, 2002.

[5] Michael J Anderson, David Sheffield, and Kurt Keutzer. A predictive model for solving small linear algebra problems in GPU registers. In *Proc. of IEEE 26th International Parallel and Distributed Processing Symposium*, pages 2–13, 2012.

[6] Karen Braman, Ralph Byers, and Roy Mathias. The multishift QR algorithm. Part II: Aggressive early deflation. *SIAM Journal on Matrix Analysis and Applications*, 23(4):948–973, 2002.

[7] Ralph Byers. LAPACK 3.1 xHSEQR: Tuning and implementation notes on the small bulge multi-shift QR algorithm with aggressive early deflation, 2007.

[8] Alain 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*, volume 29, pages 800–810, 2014.

[9] Tingxing Dong, Azzam Haidar, Piotr Luszczek, James Austin Harris, Stanimire Tomov, and Jack Dongarra. LU factorization of small matrices: Accelerating batched DGETRF on the GPU. In *Proc. of IEEE International Conference on High Performance Computing and Communications*, pages 157–160, 2014.

[10] John GF Francis. The QR transformation a unitary analogue to the LR transformation–part 1. *The Computer Journal*, 4(3):265–271, 1961.

[11] John GF Francis. The QR transformation–part 2. *The Computer Journal*, 4(4):332–345, 1962.

[12] Gene H. Golub and Chrles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[13] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil. Performance models for CUDA streams on NVIDIA GeForce series. *Journal of Parallel and Distributed Computing*, 72(9):1117–1126, 2012.

[14] Robert Granat, Bo Kågström, Daniel Kressner, and Meiyue Shao. Algorithm 953: parallel library software for the multishift QR algorithm with aggressive early deflation. *ACM Transactions on Mathematical Software*, 41(4):29, 2015.

[15] Ping Guo and Liqiang Wang. Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs. In *Proc. of International Conference on Computational and Information Sciences*, pages 1154–1157, 2010.

[16] Azzam Haidar, Tingxing Tim Dong, Stanimire Tomov, Piotr Luszczek, and Jack 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*, pages 31–47, 2015.

[17] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

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

[19] Intel Corporation. *Intel Math Kernel Library (Intel MKL)*. `https://software.intel.com/en-us/intel-mkl`.

[20] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. Offline permutation algorithms on the discrete memory machine with performance evaluation on the GPU. *IEICE Transactions on Information and Systems*, Vol. E96-D(12):2617–2625, Dec. 2013.

[21] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation. In *Proc. of International Conference on Parallel Processing (ICPP)*, pages 1–10. IEEE CS Press, Oct. 2013.

[22] Duhu Man, Kenji Uda, Hironobu Ueyama, Yasuaki Ito, and Koji Nakano. Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs. *International Journal of Networking and Computing*, 1(2):260–276, July 2011.

[23] Jiří Matela, Martin Šrom, and Petr Holub. Low GPU occupancy approach to fast arithmetic coding in JPEG2000. In *Proc. of International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 136–145, 2011.

[24] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[25] NVIDIA Corporation. *cuBLAS*. `https://developer.nvidia.com/cublas`.

[26] NVIDIA Corporation. *cuSOLVER*. `https://developer.nvidia.com/cusolver`.

[27] NVIDIA Corporation. *NVIDIA TITAN X*. `https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/`.

[28] NVIDIA Corporation. *CUDA C Best Practice Guide Version 8.0*, 2017.

[29] NVIDIA Corporation. *CUDA C Programming Guide Version 8.0*, 2017.

[30] Kohei Ogawa, Yasuaki Ito, and Koji Nakano. Efficient Canny edge detection using a GPU. In *Proc. of International Conference on Networking and Computing*, pages 279–280. IEEE CS Press, Nov. 2010.

[31] Peripheral Component Interconnect Special Interest Group. PCI Express. `http://pcisig.com/`.

[32] Steve Rennich. CUDA C/C++ streams and concurrency, 2011. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`.

[33] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in CUDA, 2010.

[34] Masami Saeki, Yusuke Kurosaka, Nobutaka Wada, and Satoshi Satoh. Parameter space design of a nonlinear filter by volume rendering (in Japanese). *Transactions of the Institute of Systems, Control and Information Engineers*, 28(10):419–425, 2015.

[35] Danny C Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM Journal on Matrix Analysis and Applications*, 13(1):357–385, 1992.

[36] Hans Henrik Brandenborg Sørensen. Auto-tuning dense vector and matrix-vector operations for Fermi GPUs. In *Proc. of International Conference on Parallel Processing and Applied Mathematics*, pages 619–629, 2011.

[37] The MathWorks, Inc. *MATLAB*. `http://mathworks.com/products/matlab`.

[38] Hiroki Tokura, Takumi Honda, Yasuaki Ito, Koji Nakano, Mitsuya Nishino, Yushiro Hirota, and Masami Saeki. GPU-accelerated bulk computation of the eigenvalue problem for many small real non-symmetric matrices. In *Proc. of the Fourth International Symposium on Computing and Networking*, pages 490–496, 2016.

[39] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. Fast and accurate template matching using pixel rearrangement on the GPU. In *Proc. of International Conference on Networking and Computing*, pages 153–159, Dec. 2011.

[40] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. An efficient GPU implementation of ant colony optimization for the traveling salesman problem. In *Proc. of International Conference on Networking and Computing*, pages 94–102. IEEE CS Press, Dec. 2012.

[41] Vasily Volkov. Better performance at lower occupancy. In *GTC Techmology Conference*, 2010.