A Study on Efficient GPU Implementations for Many Small Problems

(大量の小さな問題のための効率的な GPU 実装に関する研究)

Hiroki Tokura

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Engineering in Information Engineering

Under Supervision of
Professor Koji Nakano

Department of Information Engineering
Graduate School of Engineering
Hiroshima University

March, 2019

Abstract

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images. Since latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU, GPUs have recently attracted the attention of many application developers.

Some applications need to compute many instances. The computation of Summed Area Table (SAT) is needed many prefix-sum computations. In control design problems, the computation of large number of the small eigenvalue problem is necessary. However, GPU can not efficiently compute these computation due to programming issues.

The first contribution of our works is to present *the Look-back Column-wise Prefix-sum (LCP) algorithm*, which computes the column-wise prefix-sums of a matrix very efficiently on the GPU. It partitions the matrix into small tiles and the column-wise sums and prefix-sums of every tile are computed using one CUDA block for each tile in parallel. The LCP algorithm does not perform stride access to the global memory, shared memory access with bank conflicts, or separated kernel calls for global synchronization, which involve large overhead. Clearly, no GPU implementation of column-wise prefix-sum computation of an $n \times n$ matrix can be faster than matrix duplication, in which n^2 elements are read and written. Thus, we can say that a column-wise prefix-sum algorithm is optimal if the computing time is equal to matrix duplication. Quite surprisingly, the experimental results on NVIDIA TITAN X GPU show that our LCP algorithm runs only 2-6% slower than matrix duplication. Thus, our LCP algorithm is almost optimal.

The second contribution of our works is to propose a GPU implementation of bulk computation of eigenvalues of small, non-symmetric, real matrices of maximum size 30×30. 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. 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.

Contents

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Contributions 2

1.2.1 The column-wise prefix-sums computation 2

1.2.2 The eigenvalues computation for many small matrices 3

1.3 Dissertation Organization 5

2 GPU and CUDA **6**

3 A GPU implementation of column-wise prefix-sum computation **13**

3.1 Introduction 13

3.2 Preliminary 17

3.3 The Look-back Column-wise Prefix-sums (LCP) algorithm on the GPU . . 21

3.4 Experimental results 27

4 A GPU implementation of the eigenvalue problem for many small real non-symmetric matrices **30**

4.1 Introduction 30

4.2 Related work 33

4.3 Eigenvalues Computation of a Non-symmetric Real Matrix 35

4.3.1 The Hessenberg Reduction 38

4.3.2 The double-shift QR sweep 40
4.4 GPU Implementation 43
4.5 Performance Evaluation 50
5 Conclusion **58**
Bibliography **66**
Acknowledgement **67**
List of publications **68**

List of Figures

2.1 The outline of the GPU 7

2.2 If warp divergence is occured, instructuions are executed in serial. 8

2.3 Not warp divergence 9

2.4 A hierarchy of threads groups 9

2.5 Coalesced access 10

2.6 Stride access 11

2.7 Overlapped execution on the GPU using multiple streams 12

3.1 Row-wise and column-wise prefix-sums of a 4×4 matrix 14

3.2 Coalesced access to the global memory (the column-wise prefix-sums) . . . 15

3.3 Stride access to the global memory (the row-wise prefix-sums) 15

3.4 Coalesced access to the global memory (CUB-prefix for each row) 16

3.5 Stride access to the global memory (CUB-prefix for each column) 16

3.6 Illustrating warp prefix scan and diagonal arrangement for $w = 8$ 20

3.7 Serial numbers assigned to tiles 22

3.8 The LS, the GS, and the GP of a tile and the computation of three steps . . 23

4.1 Volume rendering of the parameter space design in the pole assignment problems [1] using eigenvalues obtained by the proposed method 32
4.2 The Hessenberg reduction for a square matrix of size 6×6 36
4.3 Bulge-generating and bulge-chasing in the double-shift QR sweep 36
4.4 Matrix division by deflation 37
4.5 Data arrangement for multiple matrices in the global memory 44
4.6 Single-warp-based method (SWB) 46
4.7 Matrix data loading from the global memory to the shared memory with coalesced access in SWB 47
4.8 Thread assignment in SWB and MWB for $n = 6$ and $p = 2$ 47
4.9 Multiple-warp-based method (MWB) 49
4.10 The computing time of Step 1 for 500000 matrices of size 5×5 to 30×30 . 52
4.11 The computing time of Steps 2 and 3 for 500000 matrices of size 5×5 to 30×30 54

Chapter 1

Introduction

1.1 Background and Motivation

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images[2, 3]. Since latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU, GPUs have recently attracted the attention of many application developers [2, 4, 5, 6, 7]. CUDA (Compute Unified Device Architecture)[8] is a parallel computing architecture provided by NVIDIA and we can develop the general purpose applications running on the GPU with scalability.

Some applications need to compute many instances. The computation of the row-wise and the column-wise prefix-sum which have many applications in the area of image processing and deep learning [9] needs many prefix-sum computations. Summed Area Table (SAT) [10, 11, 12] can be computed by computing the row-wise prefix-sums and the column-wise prefix-sums. If we compute SAT of 4096×4096 matrix, 4096 row-wise prefix-sum and 4096 column-wise prefix-sums should be computed. Since GPU has thousands of cores and high memory bandwidth, GPU can compute many prefix-sums at the same

time. GPU can efficiently compute the row-wise prefix-sum because each row is stored in consecutive memory address. On the other hand, GPU can not efficiently compute the column-wise prefix-sum because each column is stored in non-consecutive memory address.

Control design problems or MRI need to compute large number of the small eigenvalue problems[1, 13]. Especially, the computation of large number of the small non-symmetric eigenvalue problems needs long time. Namely, accelerating of computation of large number of the small non-symmetric eigenvalue problems are necessary. However, there are no GPU implementations of computing large number of the small non-symmetric eigenvalue problem, because the non-symmetric eigenvalue problem computation is too complicated algorithm.

1.2 Contributions

In this dissertation, we present the following two GPU implementations of computations.

1.2.1 The column-wise prefix-sums computation

The main contribution of this work is to present *the Look-back Column-wise Prefix-sum (LCP) algorithm*, which computes the column-wise prefix-sums of a matrix very efficiently on the GPU. It partitions the matrix into small tiles and the column-wise sums and prefix-sums of every tile are computed using one CUDA block for each tile in parallel. The LCP algorithm involves several GPU computing techniques including *the warp prefix scan* [14], *the diagonal arrangement of a matrix* [15], and *the decoupled look-back* [16] to minimize memory access and synchronization overhead. The LCP algorithm does not perform stride access to

the global memory, shared memory access with bank conflicts, or separated kernel calls for global synchronization, which involve large overhead. Clearly, no GPU implementation of column-wise prefix-sum computation of an $n \times n$ matrix can be faster than matrix duplication, in which n^2 elements are read and written. Thus, we can say that a column-wise prefix-sum algorithm is optimal if the computing time is equal to matrix duplication. Quite surprisingly, the experimental results on NVIDIA TITAN X GPU show that our LCP algorithm runs only 2-6% slower than matrix duplication. Thus, our LCP algorithm is almost optimal.

1.2.2 The eigenvalues computation for many small matrices

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 [17, 18]. These methods are to concurrently perform the iteration, called *QR sweep*, not to destroy the order of the iterations. Actually, in LAPACK [19], that is a linear algebra library, small bulge multi shift QR algorithm is employed [20]. 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 [21] 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 [22]. 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 [23, 24, 25]. 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.

1.3 Dissertation Organization

This dissertion is organized as followes. In Chapter 2, we describe GPU and CUDA. In Chapter 3, we show the GPU implementation of column-wise prefix-sum computation, ans its performance. In Chapter 4, we show the GPU implementation of the eigenvalue problem for many small real non-symmetric matrices, and the performance evaluations. Finalliy, we conclude this dissertion in Chapter 5.

Chapter 2

GPU and CUDA

In this chapter, the GPU architecture and CUDA programming guide are described. Figure 2.1 shows the outline of GPU. GPU has one or more Streaming Multiprocessors (SMs) and the global memory. Each SM consists of multiple cores , the register file and the shared memory. For example, NVIDIA TITAN X [26] has 28 SMs with 128 core and 96KB shared memory each and 12 GB global memory which has 480GB/s bandwidth. Each SM can access the global memory but the latency of the global memory access is quite large. Cores in a SM can access the shared memory in only the same SM, however, the latency of the shared memory access is quite small. That is, the appropriate usage of the shared memory is the key to achieve high GPU computing performance.

CUDA (Compute Unified Device Architecture)[8] is a parallel computing architecture provided by NVIDIA and we can develop the general purpose applications running on the GPU with scalability. CUDA has a hierarchy of threads groups, a CUDA block is a group of up to 1024 threads and a CUDA kernel is a group of one or more CUDA blocks 2.4. When a CUDA kernel is lunched, a CUDA kernel executes one or more CUDA blocks. Each CUDA block is assigned to one of the SMs and threads in a CUDA block are assigned

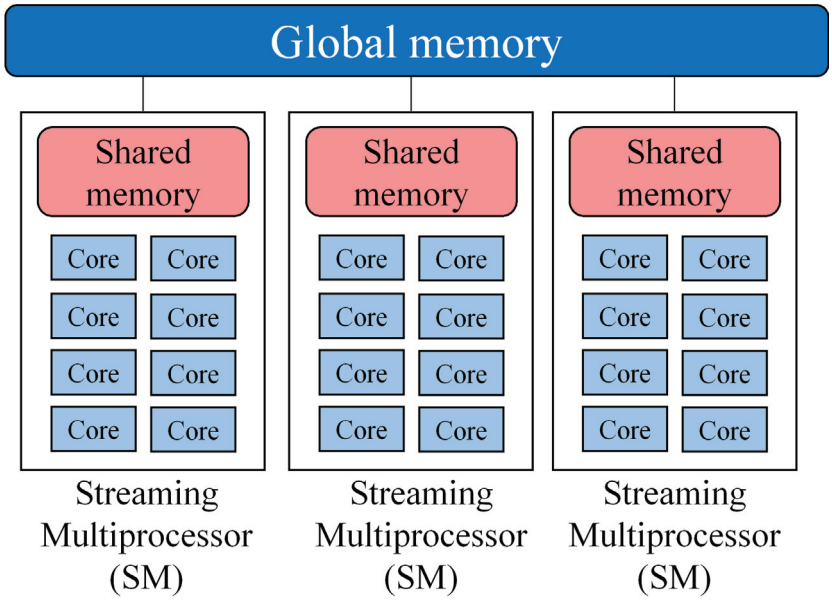


Figure 2.1: The outline of the GPU

to a core. Up to 2048 threads can be assigned cores in a SM at the same time, so large number of CUDA blocks cannot be assigned to a SM simultaneously. A CUDA block can be assigned to a SM after another CUDA block process is completed. Threads can access the Shared memory in a CUDA block Threads in a CUDA block is partitioned into several groups of 32 threads called warp. All threads in a warp work synchronously and execute the same instruction.

Threads in a warp can execute the different instruction using if-statement. We assume that threads in a warp execute instructions such as Figure 2.2. Even-numbered threads execute instructions A and the other threads execute instructions B. Odd-numbered threads execute instructions B after even-numbered threads execute instructions A because all threads in a warp work synchronously (Figure 2.2). Such execution is called warp divergence and warp divergence decreases the performance. On the other hands, it is not warp divergence

if all threads in a warp execute the same instruction like fig. 2.3.

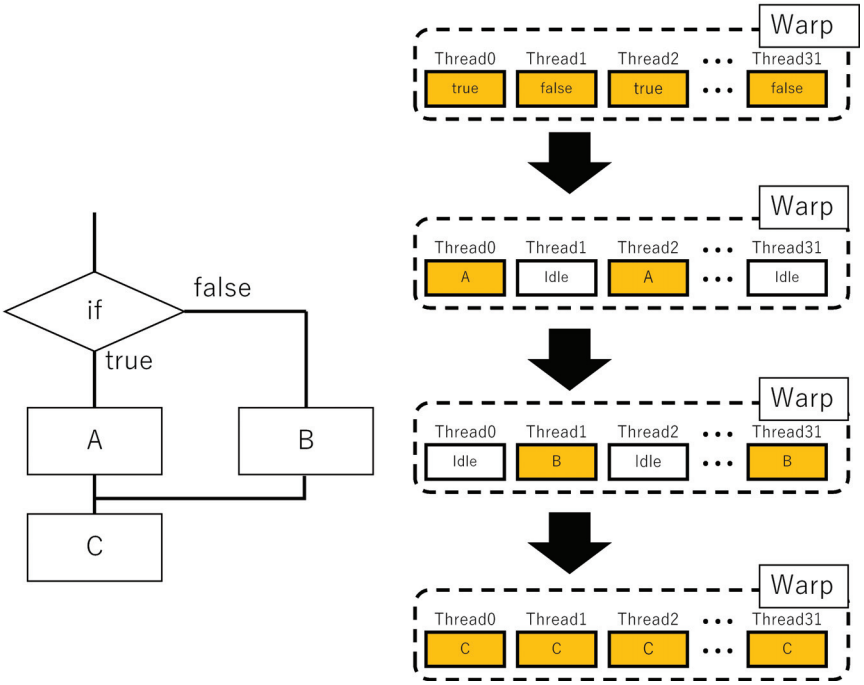


Figure 2.2: If warp divergence is occured, instructuions are executed in serial.

Threads in different warps work asynchronously so may not be executed the same instruction simultaneously. *__syncthreads()* guarantees that threads in a CUDA block are synchronized. *__syncthreads()* has a overhead so the frequent usage of *__syncthreads()* decreases the performance.

Since the shared memory has small capacity but the latency is quite low, the appropriate usage of the shared memory improves the overall performance. The shared memory consists of 32 memory banks, and two or more threads access different address in the same bank decreases the shared memory access performance, it is called bank conflict. If bank conflict is occurred, the shared memory access to the bank is serialized. That is, bank conflict should be avoided.

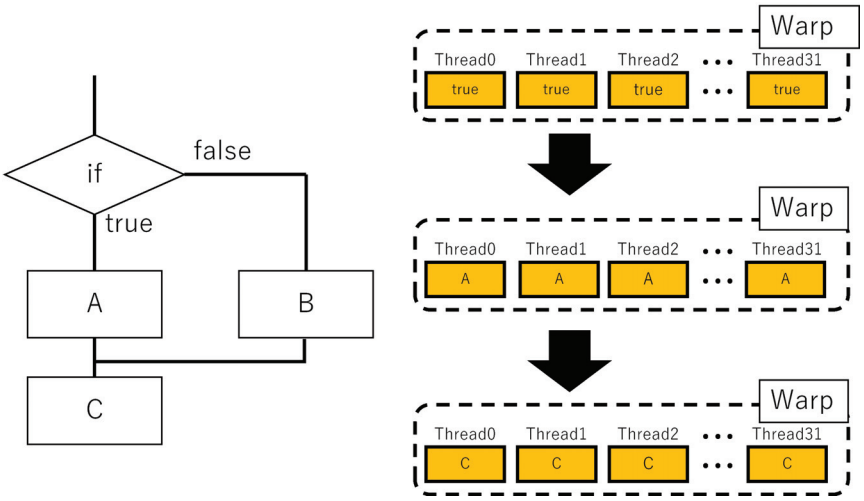


Figure 2.3: Not warp divergence

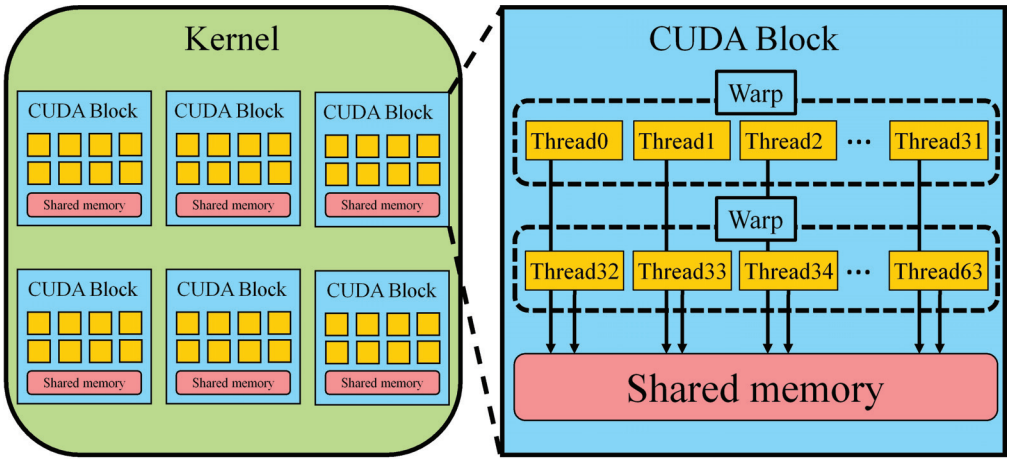


Figure 2.4: A hierarchy of threads groups

The global memory has large capacity but the latency is large. So Efficient access to the global memory improves the overall performance. Multiple threads on GPU simultaneously access global memory. So, there are two memory access patterns, coalesced access and stride access. We suppose that Elements in a matrix are stored in row major order. Coalesced access (Figure 2.5) is threads in the same warp access the same row at the same time that is, threads simultaneously access the consecutive addresses of the global memory . All threads get data in the same time. So, coalesced access is efficiently. Stride access (Figure 2.6) is threads in the same warp access the same column at the same time that is, threads simultaneously access distinct location of the global memory Threads get data sequentially. So, stride access is not efficiently.

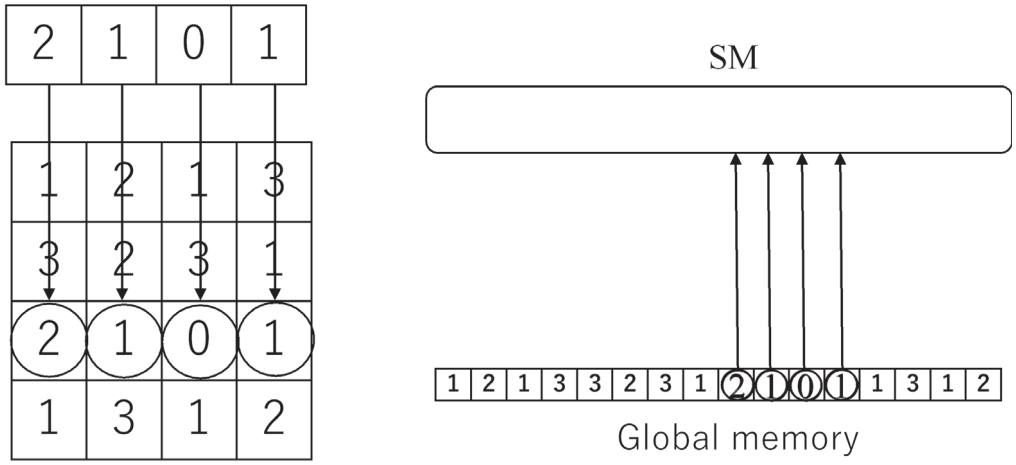


Figure 2.5: Coalesced access

The latest GPUs such as NVIDIA TITAN X support warp shuffle functions which threads in a warp can exchange data. Threads in a warp can get data of register a of i-th thread with *__shfl(a,i)*. Also, k-th thread can get data of register a of (k+i)-th threads in a warp using *__shfl_up(a,i)*.

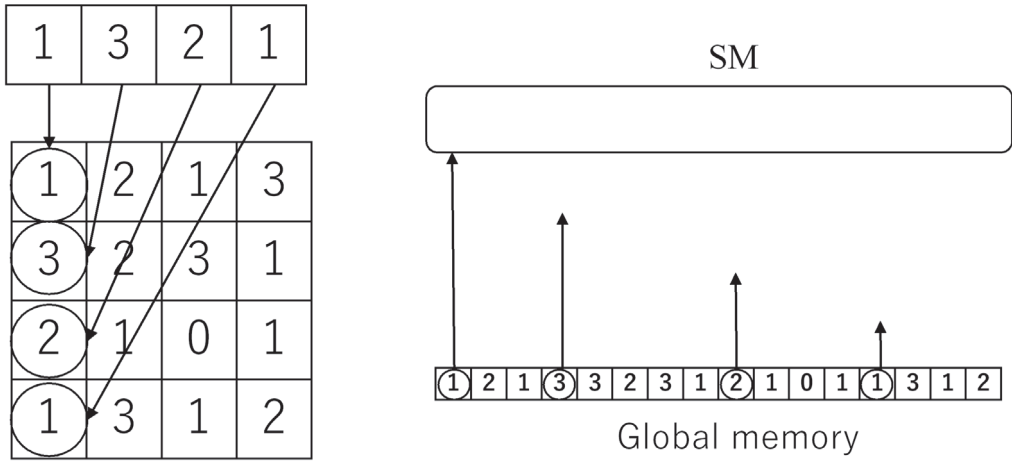


Figure 2.6: Stride access

Since multiple threads on GPU simultaneously access the global memory or the shared memory, atomic functions are supported. *atomicAdd(&c,1)*, which is one of atomic functions, exclusively increments *c* ans returns the value of *c* before addition.

To reduce CPU-GPU communication overhead, recent GPUs can perform an asynchronous memory copy to or from the GPU concurrently with kernel execution [27, 28]. 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 2.7. The data transfer hidden improves the performance if the data communication overhead is not small.

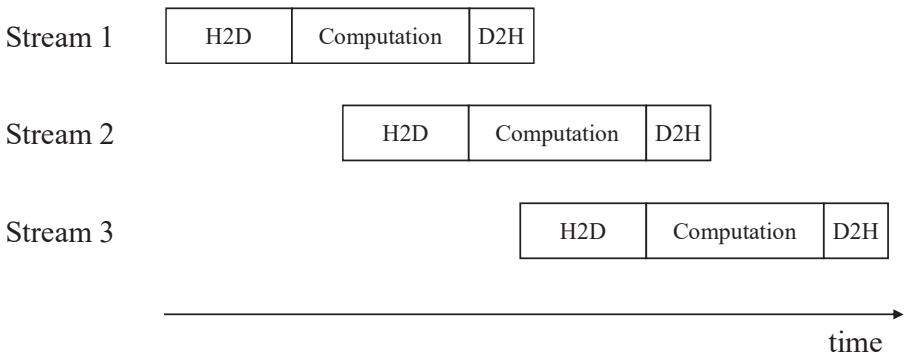


Figure 2.7: Overlapped execution on the GPU using multiple streams

Chapter 3

A GPU implementation of column-wise prefix-sum computation

3.1 Introduction

Let a_0, a_1, ..., a_{n-1} be n numbers. The prefix-sums $\hat{a}$ of a are n numbers such that $\hat{a}_i = a_0 + a_1 + \cdots + a_i$ for all i ($0 \leq i \leq n-1$). Suppose that each variable $A[i]$ stores a_i. After executing $A[i] \leftarrow A[i] + A[i-1]$ for all i ($1 \leq i \leq n-1$) in turn, each $A[i]$ stores the prefix-sum $\hat{a}_i$. The computation of the prefix-sums of a 1-dimensional array is one of the most important computation for many algorithms. For example, list ranking problem which determines the position of each item in a linked list can be solved by computing the prefix-sums. It is also used for computing the positions of keys in radix sort. In [14], several fundamental algorithms for computing the prefix-sums on the GPU have been shown. Also, Merrill *et al.* [29, 16] has presented a more sophisticated GPU implementation for the prefix-sums using decoupled look-back technique. The source program of this GPU implementation is available in [29]. As far as we know this algorithm is the most efficient GPU implementation for computing the prefix-sums of a 1-dimensional array. For later reference, we call this algorithm *CUB-prefix* in this paper.

Suppose that a matrix a with $n \times n$ elements $a_{i,j}$ $(0 \leq i, j \leq n-1)$ is given. As usual, we assume that each $a_{i,j}$ is an element in the i-th row and j-th column. The row-wise prefix-sums correspond to a matrix r of the same size such that $r_{i,j} = a_{i,0} + a_{i,1} + \cdots + a_{i,j}$ for all i and j $(0 \leq i, j \leq n-1)$. Similarly, the column-wise prefix-sums correspond to a matrix c such that $c_{i,j} = a_{0,j} + a_{1,j} + \cdots + a_{i,j}$ for all i and j $(0 \leq i, j \leq n-1)$. Figure 3.1 illustrates the row-wise and the column-wise prefix-sums of a 4×4 matrix. They have many applications in the area of image processing. For example, the summed area table [10, 11, 12] can be obtained by computing the row-wise prefix-sums and the column-wise prefix-sums. Also, in the computation of Euclidean distance map of a binary image, the column-wise prefix-minima is computed [30].

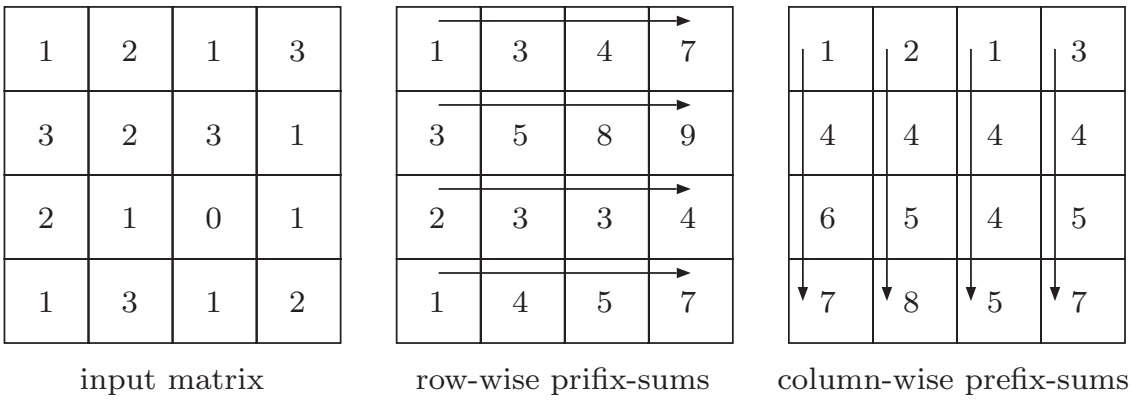


Figure 3.1: Row-wise and column-wise prefix-sums of a 4×4 matrix

The row-wise and the column-wise prefix-sums of an $n \times n$ matrix stored in the global memory can be computed in an obvious way using n threads on the GPU. More specifically, we assign a thread to each row/each column and compute the prefix-sums of each row/each column as illustrated in Figure 3.1. Since an $n \times n$ matrix is arranged in row-major order, memory access to the same row is coalesced. Thus, memory access by n threads to the

same row is coalesced and the column-wise prefix-sums can be computed efficiently by coalesced memory access to the global memory (Figure 3.2). On the other hands, thus, memory access by n threads to the same column is stride and the row-wise prefix-sums can be computed nonefficiently by stride memory access to the global memory (Figure 3.3).

However, since only n threads are used, the latency overhead of the global memory access is not negligible. Thus, we should use cn^2 threads for some constant $c > 0$ to hide the latency overhead.

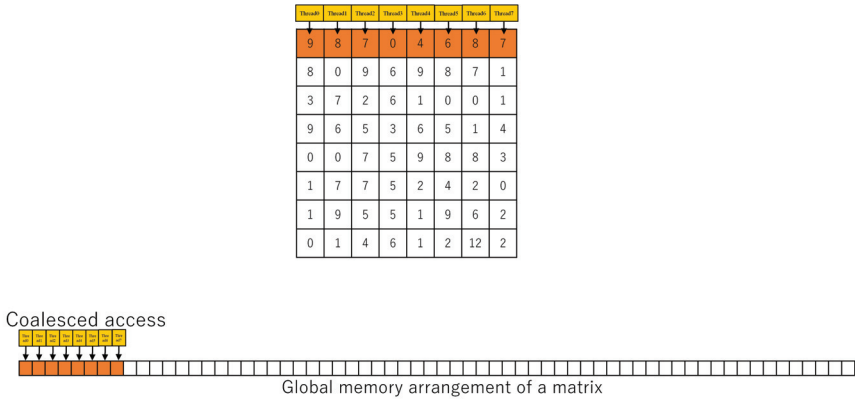


Figure 3.2: Coalesced access to the global memory (the column-wise prefix-sums)

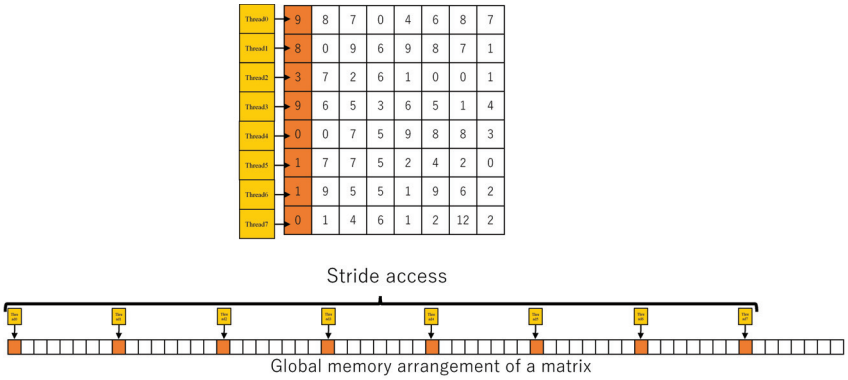


Figure 3.3: Stride access to the global memory (the row-wise prefix-sums)

To reduce the latency overhead, we can use CUB-prefix to compute the row-wise/column-

wise prefix sums on the GPU. The row-wise prefix-sums can be computed very efficiently by executing CUB-prefix for each row in parallel on the GPU. Since memory access to the global memory is coalesced, this approach works very efficiently (Figure 3.4). Similarly, the column-wise prefix-sums can be computed by executing CUB-prefix for each column in parallel. However, memory access is not coalesced (Figure 3.5), it runs much slower than the row-wise prefix-sum computation by CUB-prefix. Hence, it is not obvious to find an efficient GPU algorithm for computing the column-wise prefix-sums.

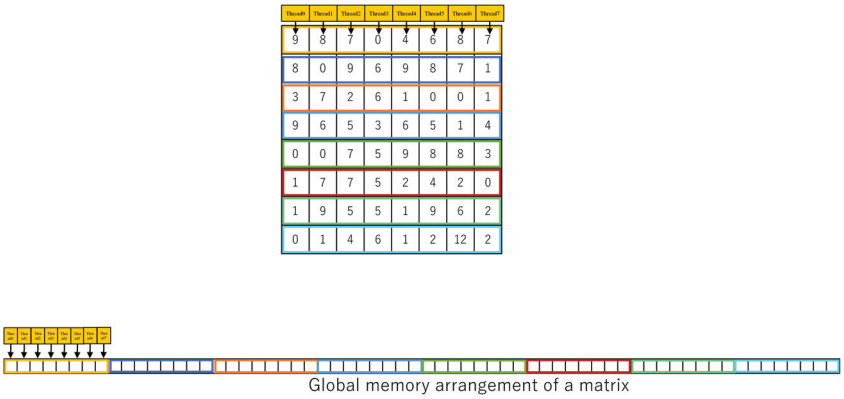


Figure 3.4: Coalesced access to the global memory (CUB-prefix for each row)

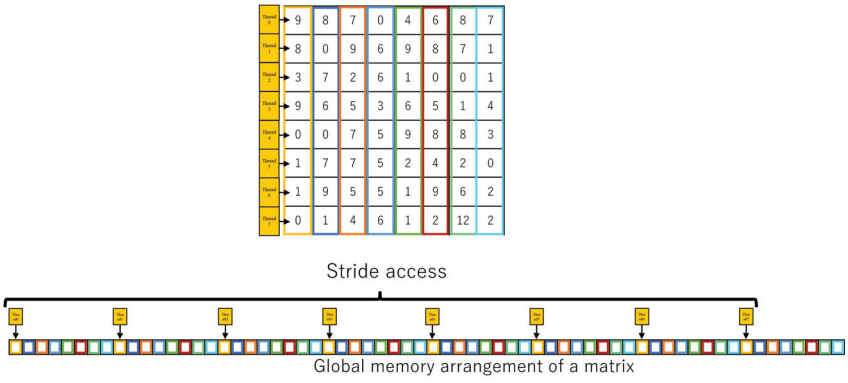


Figure 3.5: Stride access to the global memory (CUB-prefix for each column)

The main contribution of this paper is to present *the Look-back Column-wise Prefix-*

sum (LCP) algorithm, which computes the column-wise prefix-sums of a matrix very efficiently on the GPU. It partitions the matrix into small tiles and the column-wise sums and prefix-sums of every tile are computed using one CUDA block for each tile in parallel. The LCP algorithm involves several GPU computing techniques including *the warp prefix scan* [14], *the diagonal arrangement of a matrix* [15], and *the decoupled look-back* [16] to minimize memory access and synchronization overhead. The LCP algorithm does not perform stride access to the global memory, shared memory access with bank conflicts, or separated kernel calls for global synchronization, which involve large overhead. Clearly, no GPU implementation of column-wise prefix-sum computation of an $n \times n$ matrix can be faster than matrix duplication, in which n^2 elements are read and written. Thus, we can say that a column-wise prefix-sum algorithm is optimal if the computing time is equal to matrix duplication.

3.2 Preliminary

This section shows several fundamental techniques on the GPU necessary to understand our LCP algorithm and naive algorithms for computing the column-wise prefix-sums.

For theoretical analysis of the performance, we use the memory machine model [15], which capture the essence of global memory access on the GPU. Let w be the number of threads of a warp. For simplicity, we assume that the bandwidth of the global memory is also w, that is, the global memory is partitioned into groups of w numbers in consecutive addresses, and w numbers in the same group can be accessed at the same time. Also, let l be the latency of the global memory, that is, memory access requests to the global memory are

processed through l-stage pipeline registers, in which each stage can store w memory access requests to the same address group. As a simple example, we evaluate the time necessary to duplicate an $n \times n$ 2-dimensional array in the global memory. We use n^2 threads for this task such that each thread duplicates an element in an obvious way. For reading n^2 elements, $\frac{n^2}{w}$ warps send memory access requests though the l-stage pipeline registers. All read requests can be completed in at most $\frac{n^2}{w} + l$ time units. Similarly, writing n^2 elements by $\frac{n^2}{w}$ warps takes $\frac{n^2}{w} + l$ time units. Thus, an $n \times n$ 2-dimensional array in the global memory can be duplicated in at most $2\frac{n^2}{w} + 2l$ time units.

Let A be an $n \times n$ 2-dimensional array in the global memory storing a matrix of $n \times n$ numbers. We assume that each $A[i][j]$ storing $a_{i,j}$ is arranged in offset $i \cdot n + j$ of the memory space for A. Suppose that A stores the values of an $n \times n$ matrix a. Let R_i $(0 \leq i \leq n-1)$ be a register of thread i. The row-wise prefix-sums of a can be computed using n threads as follows:

[Naive row-wise prefix-sum algorithm]

for $i \leftarrow 0$ to $n-1$ do in parallel

  thread i performs $R_i \leftarrow A[i][0]$;

  for $j \leftarrow 1$ to $n-1$ do

    thread i performs $R_i \leftarrow R_i + A[i][j]$; $A[i][j] \leftarrow R_i$;

Clearly, n threads read $A[0][j], A[1][j], \ldots, A[n-1][j]$ for each i $(0 \leq i \leq n-1)$. Also, they write in $A[0][j], A[1][j], \ldots, A[n-1][j]$ for each i $(1 \leq i \leq n-1)$. If $n \geq w$, then these n variables are in distinct groups of the global memory, and each pipeline stage can store one read access request. Thus, access to these n numbers takes $n + l$ time. Since such

memory access is performed $2n-1$ times, the naive row-wise prefix-sum algorithm runs in $2(n-1)(n+l) < 2n^2 + nl$ time units using n threads.

The column-wise prefix-sums can be computed in the same way. In the naive column-wise prefix-sum algorithm, memory access to $A[i][0], A[i][1], \ldots, A[i][n-1]$ is performed for each i. Clearly, such memory access is coalesced, it takes at most $\frac{n}{w} + l$ time units to access these numbers. Thus, the column-wise prefix-sum algorithm runs at most takes $2\frac{n^2}{w} + 2nl$ time units using n threads.

Suppose that each thread in a warp has a register a storing a number and we write $A[i]$ $(0 \leq i \leq w-1)$ to denote register A of thread i. We assume that a 1-dimensional array a of size w are stored in register A's such that each $A[i]$ stores a_i. The prefix-sums of a can be computed in $\log_2 w$ steps as follows:

[Warp prefix scan]

for $k \leftarrow 0$ to $\log_2 w - 1$ do

  for $i \leftarrow 0$ to $w-1$ do in parallel

    thread i performs $A[i] \leftarrow A[i] + A[i-2^k]$ if $i \geq 2^k$;

Figure 3.6 illustrates how the warp prefix scan computes the prefix-sums. The reader should refer [14, 31, 32] for the details. In the warp prefix scan, each thread i $(0 \leq i \leq w-1)$ must read register $A[i-2^k]$ of thread $i-2^k$. This register read can be done very efficiently by warp shuffle function __shfl_up(A,2^k), which directly reads the value of register A of thread $i-2^k$. Since no memory access to the shared memory or the global memory is performed, the warp prefix scan runs very efficiently on a streaming multiprocessor of the GPU.

Suppose that we have a $w \times w$ 2-dimensional array A stored in the shared memory with

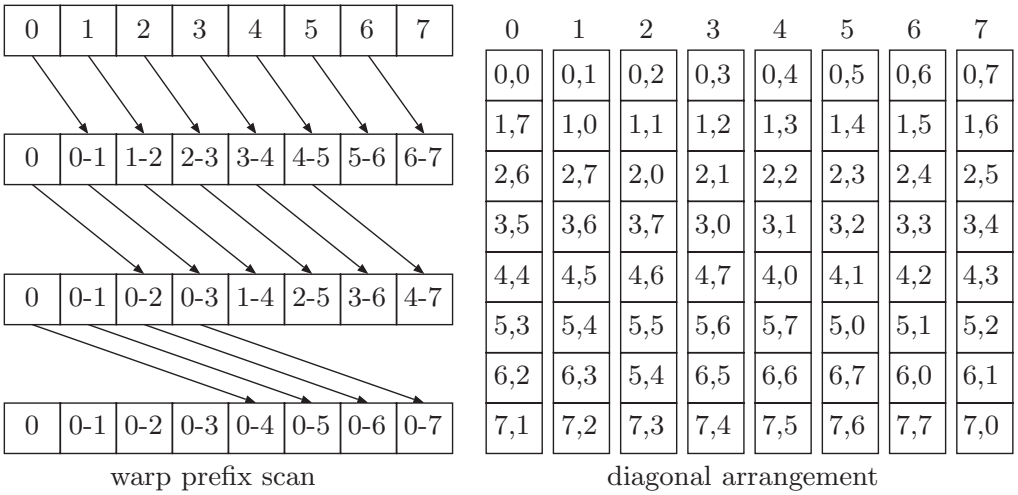


Figure 3.6: Illustrating warp prefix scan and diagonal arrangement for $w = 8$

w memory banks. Each $A[i][j]$ is in offset $wi + j$ of S, which is arranged in bank $(wi + j) \bmod w = j$. Thus, the row-wise memory access to $A[i][0], A[i][1], \ldots, A[i][w-1]$ has no bank conflict while the column-wise memory access to $A[0][j], A[1][j], \ldots, A[w-1][j]$ is destined for the same bank j. By the diagonal arrangement which maps each $A[i][j]$ to offset $wi + ((i + j) \bmod w)$, both the row-wise memory access and the column-wise memory access have no bank conflict. Figure 3.6 illustrates the diagonal arrangement for $w = 8$. Since w elements $A[i][0], A[i][1], \ldots, A[i][w-1]$ are arranged in banks $i \bmod w$, $(i+1) \bmod w$, ..., $(i+w-1) \bmod w$, respectively, the row-wise memory access is conflict-free. Similarly, the column-wise memory access has no bank conflict. Thus, the column-wise/row-wise prefix-sums can be computed very efficiently by executing warp prefix scan for each column/row in parallel. For later reference, we call the column-wise prefix-sum computation by this technique *column-wise warp scan*.

As we have mentioned, the CUB-based row-wise prefix-sum computation runs very efficiently, while the CUB-based column-wise prefix-sum performs stride memory access

with large overhead. To avoid stride memory access, we can transpose the input matrix in advance. More specifically, the column-wise prefix-sums can be computed, by matrix transposition, the row-wise prefix-sum computation, and matrix-transposition. Since matrix transposition can be done very efficiently by coalesced memory access to the global memory [15], this 3-step algorithm may run more efficiently on the GPU.

3.3 The Look-back Column-wise Prefix-sums (LCP) algorithm on the GPU

This section shows our LCP algorithm that computes the column-wise prefix-sums on the GPU. Again, let $w = 32$ denote the number of threads. We use CUDA blocks with $w^2 = 1024$ threads each and let $t_{i,j}$ $(0 \leq i, j \leq w-1)$ denote thread j in warp i, *i.e.* thread $iw + j$ in a CUDA block. Suppose that an $n \times n$ matrix a in the global memory is partitioned into $\frac{n}{wd} \times \frac{n}{w}$ tiles of size $wd \times w$ each as illustrated in Figure 3.7, where $d \geq 1$ is an integer parameter. Let $T(i, j)$ $(0 \leq i \leq \frac{n}{wd} - 1$ and $0 \leq j \leq \frac{n}{w} - 1)$ denote a tile. We assume that serial numbers from 0 to $\frac{n^2}{w^2 d} - 1$ are assigned to tiles in row major order, that is, each $T(i, j)$ is assigned a serial number $i\frac{n}{w} + j$. We also call $T(i, j)$ *tile* k for $k = i\frac{n}{w} + j$ and the computation performed for tile k *task* k. Each tile is further partitioned into w *strips* 0, 1, ..., $w - 1$ of size $d \times w$ each as illustrated in Figure 3.7.

Each tile k $(0 \leq k \leq \frac{n^2}{w^2 d} - 1)$ is assigned a CUDA block, which performs task k in three steps. Let $a[i][j]$ $(\alpha \leq i \leq \alpha + wd - 1$ and $\beta \leq j \leq \beta + w - 1)$ be elements of tile k. In Step 1 of task k, *the local column-wise sums (LS)* of tile k, $a[\alpha][j] + a[\alpha + 1][j] + \cdots + a[\alpha + wd - 1][j]$ for all j $(\beta \leq j \leq \beta + w - 1)$, are computed and written in the global memory. Step 2

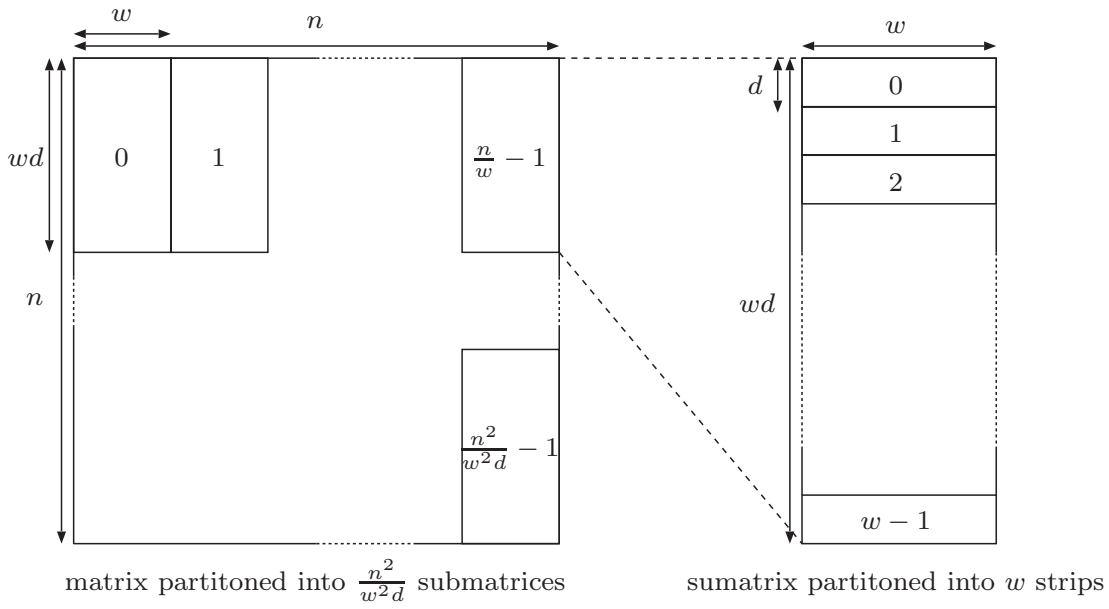


Figure 3.7: Serial numbers assigned to tiles

computes *the global column-wise sums (GS)*, $a[0][j]+a[1][j]+\cdots+a[\alpha+wd-1][j]$ for all j ($\beta \le j \le \beta+w-1$) and writes them in the global memory. Finally, *the global column-wise prefix-sums (GP)*, $a[0][j] + a[1][j] + \cdots + a[i][j]$ for all i and j ($\alpha \le i \le \alpha + wd - 1$ and $\beta \le j \le \beta + w - 1$) are computed and written in the global memory in Step 3. Thus, when Step 3 of all tasks is completed, all column-wise prefix-sums of a are stored in the global memory. Figure 3.8 illustrates the LS, the GS, and the GP of a tile.

For later reference, we define the state of a tile in the LCP. Initially, all tiles are in null state. A tile changes to State LS when values of the LS are written in the global memory in Step 1. After that, it changes to State GS when values of the GS are written in the global memory in Step 2. The LCP algorithm uses a 2-dimensional array of size $\frac{n}{wd} \times \frac{n}{w}$ in the global memory to store the states of all $\frac{n}{wd} \times \frac{n}{w}$ tiles. A CUDA block assigned to a tile updates the corresponding element of this 2-dimensional array when the tile changes the

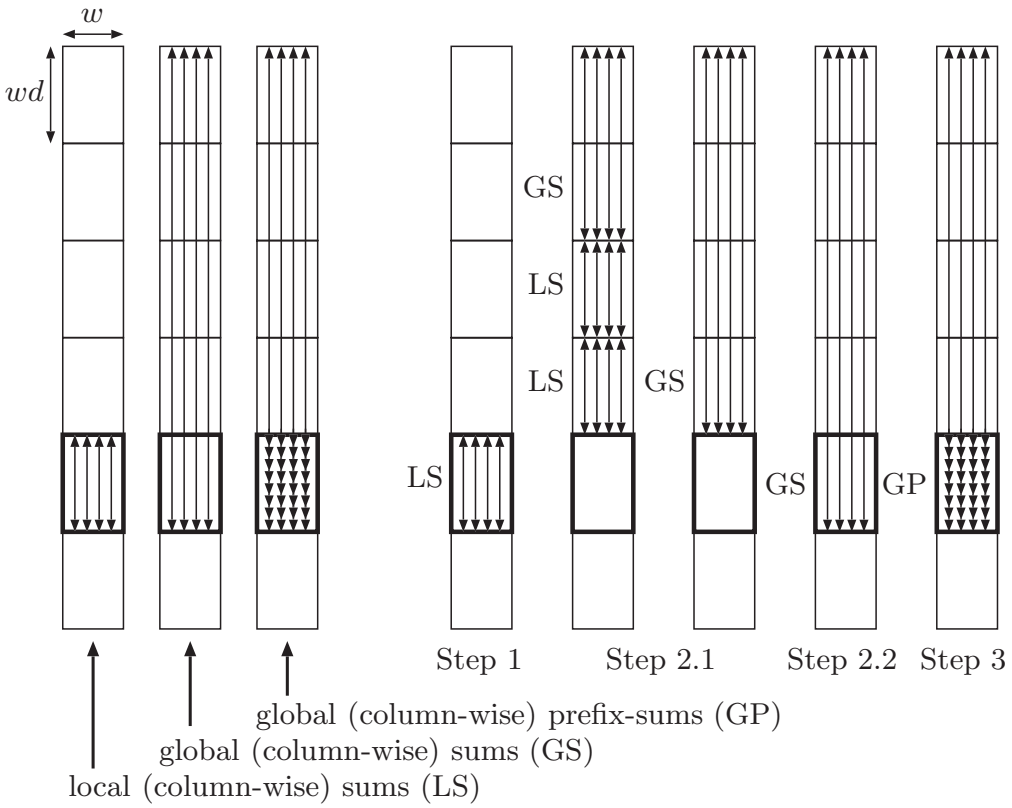


Figure 3.8: The LS, the GS, and the GP of a tile and the computation of three steps

state.

A CUDA block is assigned to one of the tiles in increasing order of serial number in turn. For this purpose, a global counter c initialized by zero in the global memory is used. A CUDA kernel for the LCP algorithm invokes $\min(\frac{n^2}{w^2 d}, m)$ CUDA blocks, where m is the maximum number of CUDA blocks that can be dispatched in the GPU at the same time. For example, NVIDIA TITAN X has 28 streaming multiprocessors with 2048 resident threads each and the LCP uses CUDA blocks with 1024 threads each, we have $m = \frac{28 \cdot 2048}{1024} = 56$. The first thread 0 of every CUDA block performs atomicAdd(&c,1), which exclusively increments c and returns the value of c before addition. Thus, atomicAdd(&c,1) returns 0,

1, ... in turn and no two threads receive the same return value. A CUDA block with the first thread receiving return value k performs task k for tile k if $k < \frac{n^2}{w^2 d}$. It terminates if $k \geq \frac{n^2}{w^2 d}$. After task k is completed, it executes $k \leftarrow$atomicAdd(&c,1) again and performs task k provided that return value k satisfies $k < \frac{n^2}{w^2 d}$. Otherwise, it terminates. The same procedure is repeated as long as return value k satisfies $k < \frac{n^2}{w^2 d}$.

We first show how w^2 threads in a CUDA block perform task 0 for tile 0. Tasks 1, 2, ..., $\frac{n}{w} - 1$ can be done in the same way. Let $a[i][j]$ ($0 \leq i \leq wd - 1$ and $0 \leq j \leq w - 1$) be elements in tile 0. A CUDA block assigned to tile 0 uses a $w \times w$ 2-dimensional array E with the diagonal arrangement. Thus, the row-wise/column-wise memory access to E is conflict-free. The details of the algorithm are spelled out as follows:

Step 1.1 Each thread $t_{i,j}$ ($0 \leq i, j \leq w - 1$) reads d elements $a[id][j]$, $a[id + 1][j]$, ..., $a[id + d - 1][j]$ one by one and store them in d registers.

Step 1.2 Each thread $t_{i,j}$ computes the sum $a[id][j] + a[id + 1][j] + \cdots + a[id + d - 1][j]$ of the d registers and write it in $E[i][j]$ in the shared memory.

Step 1.3 Execute the column-wise warp prefix scan for E. Clearly, each $E[i][j]$ stores the value of $a[0][j] + a[1][j] + \cdots + a[id + d - 1][j]$ for all i and j.

Step 1.4 and 2 The LS of tile 0, the values stored in $E[w - 1][0], E[w - 1][1]$, ..., $E[w - 1][w - 1]$ are written in the global memory. Since tile 0 is in the topmost row, they are also the GS of tile 0.

Step 3 Each thread $t_{i,j}$ ($0 \leq i, j \leq w - 1$) computes the prefix-sums of $E[i - 1][j] + a[id][j], a[id + 1][j], a[id + 2][j], \ldots, a[id + d - 1][j]$ in an obvious way, and writes

them in the global memory. For simplicity, we assume $E[-1][j] = 0$ for all j. Since $E[i-1][j] = a[0][j] + a[1][j] + \cdots + a[id-1][j]$, these prefix-sums thus obtained are the GP of tile 0.

Clearly, all memory access operations to the global memory are coalesced, and those to the shared memory are conflict-free.

Next, we will show how tasks $\frac{n}{w}$ and larger are performed. For simplicity, we show the algorithm for task $r\frac{n}{w}$ for tile $T(r,0)$ such that $1 \leq r \leq \frac{n}{wd} - 1$. The other tasks can be done in the same way. Step 1, which computes the LS, can be done in the same way as task 0 for tile 0. Steps 2 and 3 of task $r\frac{n}{w}$ are spelled out as follows:

Step 2.1 The GS of tile $T(r-1,0)$, $a[0][j] + a[1][j] + \cdots + a[rwd-1][j]$ for all j ($0 \leq j \leq w-1$), are computed and stored in registers. We will show how these values are computed later. Let $g[j] = a[0][j] + a[1][j] + \cdots + a[rwd-1][j]$ be the GS of tile $T(r-1,0)$ thus obtained.

Step 2.2 Each thread $t_{w-1,j}$ ($0 \leq j \leq w-1$) computes $g[j] + E[w-1][j]$, which is equal to the GS of tile $T(r,0)$, $a[0][j] + a[1][j] + \cdots + a[(r+1)wd-1][j]$, and writes it in the global memory.

Step 3 Each thread $t_{i,j}$ ($0 \leq i, j \leq w-1$) computes the prefix-sums of $g[j] + E[i-1][j] + a[rwd+id][j], a[rwd+id+1][j], a[rwd+id+2][j], \ldots, a[rwd+id+d-1][j]$, which are equal to the GP of tile $T(r,0)$, and writes them in the global memory.

The reader should refer to Figure 3.8 illustrating computation performed in three steps. Step 2.1 is implemented by looking back above tiles. If $T(r-1,0)$ is in State GS, then the

GS of $T(r-1,0)$ can be obtained simply by reading the global memory. If not, it waits until $T(r-1,0)$ is in State LS, and read the LS of $T(r-1,0)$. After that we repeat the same procedure for $T(r-2)$. If $T(r-2,0)$ is in State GS, then the GS of $T(r-2,0)$ are read. By computing the pairwise sum of the GS of $T(r-2,0)$ and the LS of $T(r-1,0)$, we can obtain the GS of $T(r-1,0)$. If not, it waits until $T(r-2,0)$ is in State LS, and read the LS of $T(r-2,0)$. Again, we repeat the same procedure upwards. Let tile $T(r',0)$ ($r' \leq r-1$) be the first tile in State GS. The GS of tile $T(r-1,0)$ can be computed by summing the GS of $T(r',0)$, the LS of $T(r'+1,0)$, the LS of $T(r'+2,0)$, ..., and the LS of $T(r-1,0)$.

Let us evaluate the performance of this algorithm. Since the local computation performed in this algorithm is quite light, the memory access to the global memory is dominant in the computing time. Thus, we evaluate the time for the global memory access by this algorithm. In Step 1.1, each of $\frac{n^2}{d}$ threads reads d numbers in the global memory. Since memory access is coalesced, it takes $\frac{n^2}{dw} + l$ time units to read $\frac{n^2}{d}$ numbers in the global memory by $\frac{n^2}{d}$ threads. This is repeated d times and Step 1.1 takes $d(\frac{n^2}{dw} + l) = \frac{n^2}{w} + dl$ time units. In Steps 1.2 and 1.3, no thread accesses the global memory. In Step 1.4, w threads assigned for each warp writes w numbers. Thus, totally $\frac{n^2}{dw^2}$ threads write $\frac{n^2}{dw^2}$ numbers in the global memory. This takes $\frac{n^2}{dw^2}+l$ time units. In Step 2.1, the above tile is always in State GS in our experiment shown in Section 3.4. Thus, only the GS of the above tile is read if this is the case. Since less than $\frac{n^2}{dw^2}$ threads writes one number in the global memory, Step 2.1 takes $\frac{n^2}{dw^2} + l$ time units. In Step 2.2, each of $\frac{n^2}{dw^2}$ threads writes one number in the global memory in $\frac{n^2}{dw^2} + l$ time units. Similarly to Step 1, Step 3 writes n^2 numbers by $\frac{n^2}{d}$ threads in $\frac{n^2}{w} + dl$ time units. Consequently the LCP runs in at most $2(\frac{n^2}{w}+dl)+3(\frac{n^2}{dw^2}+l) = 2\frac{n^2}{w}+3\frac{n^2}{dw^2}+(2d+3)l$

time units. Note that we can select the value of d to minimize the running time of the LCP. The running time is minimized when $3\frac{n^2}{dw^2} = 2dl$, that is, $d = \sqrt{\frac{3n^2}{2w^2l}}$.

3.4 Experimental results

We have used NVIDIA TITAN X GPU, which has 28 streaming multiprocessors with 128 processor cores each to evaluate GPU implementations of column-wise prefix-sum computation.

Table 3.1 shows the running time in milliseconds for an $n \times n$ matrix with 4-byte single precision floating point numbers from n =1K (1024) to 32K (32768). In "duplicate" the input matrix is duplicated using cudaMemcpy, which reads all n^2 elements of the matrix and writes them in another space of the global memory. As we have shown in Section 3.2, "duplicate" takes $2\frac{n^2}{w} + 2l$ time units. Clearly, no column-wise prefix-sum algorithm cannot be faster than "duplicate", we can say that the running time of "duplicate" is the lower bound of that of any column-wise prefix-sum computation.

In "naive", the naive column-wise prefix-sum algorithm executed using $\frac{n}{w}$ CUDA blocks with $w = 32$ threads each. As we have shown in Section 3.2, it runs $2\frac{n^2}{w} + 2nl$ time units. In the table, "ratio" is the running time ratio over "duplicate", that is, the running time of "naive" divided by that of "duplicate." Thus, the ratio indicates the overhead, that is, the algorithm has ϵ overhead if it is $1 + \epsilon$. From theoretical analysis, the ratio is $\frac{2\frac{n^2}{w}+2nl}{2\frac{n^2}{w}+2l}$. The latency overhead $2nl$ of "naive" is dominant for smaller n. Since "naive" uses only n threads, the ratio is much larger than 1. In particular, the ratio is more than 10 when $n = 1$K, because it uses only 1024 threads on the GPU with 3584 cores. Since fewer threads iterate

Table 3.1: The running time (in milliseconds) on TITAN X of the column-wise prefix-sums computation for a $n \times n$ matrix and the ratio of the running time over that of matrix duplication

	n	1K	2K	4K	8K	16K	32K
duplicate	time	0.0274	0.0974	0.379	1.50	6.00	22.3
naive	time	0.277	0.557	1.26	3.12	8.48	41.0
	ratio	10.1	5.72	3.33	2.07	1.41	1.84
column-wise	time	0.138	0.376	1.38	5.25	20.5	83.3
CUB	ratio	5.04	3.86	3.64	3.49	3.41	3.74
transposed	transpose time	0.101	0.246	0.877	3.90	15.8	61.9
CUB	row-wise time	0.0527	0.125	0.433	1.62	6.32	25.6
	total time	0.154	0.370	1.31	5.52	22.1	87.5
	ratio	5.60	3.80	3.46	3.67	3.68	3.93
LCP	time	0.0281	0.101	0.392	1.58	6.33	23.5
	ratio	1.02	1.04	1.04	1.05	1.05	1.06

global memory access, the latency overhead degrades the performance.

Table 3.2: The running time (in milliseconds) of the LCP with parameter d for $n \times n$ matrix

d \n	1K	2K	4K	8K	16K	32K
1	0.0591	0.207	0.813	3.24	13.1	52.8
2	0.0349	0.126	0.497	1.92	7.72	29.4
4	0.0286	0.105	0.404	1.59	6.46	24.1
8	0.0281	0.101	0.392	1.58	6.33	23.5
16	0.0316	0.104	0.404	1.59	6.35	24.2
32	0.0321	0.102	0.395	1.67	6.65	27.1

In "column-wise CUB", CUB-prefix is executed for every column in parallel. Since memory access performed by "column-wise CUB' is not coalesced it takes at least $2n^2$ time units to read/write n^2 numbers. Thus, the ratio is more than 3 for all n. In "transposed CUB", matrix transpose, row-wise CUB, and matrix transpose are executed to compute the column-wise prefix-sums. We can implement matrix transposition by block-wise matrix transposition. Each of n^2 numbers in the global memory is read and write once, and all memory access operations are coalesced. Thus, the matrix transposition can be done in

$2\frac{n^2}{w} + 2l$ time units. Also, "row-wise CUB" takes at least $2\frac{n^2}{w} + 2l$ time unit. In the table, "transpose time" is the time for performing matrix transpose twice and "row-wise time" is that for executing CUB-prefix for every row in parallel. Therefore, "transposed CUB" takes $6\frac{n^2}{w} + 6l$ time units and the ratio is also more than 3 as shown in the table.

To implement our LCP in NVIDIA TITAN X, we invoked $\min(\frac{n^2}{dw^2}, 56)$ CUDA blocks with 1024 threads each, because each of 28 streaming multiprocessors has 2048 resident threads. We have measured the running time with parameter $d = 1, 2, 4, 8, 16$ and 32 as shown in Table 3.2. We can see that the running time is minimized when $d = 8$ for all n. From Table 3.1, we can see that the LCP is much faster than "naive", "column-wise CUB", and "transposed CUB." In particular, the ratio is very close to 1 and the overhead is only 2-6%. Thus, we can say that computation performed by our LCP is almost hidden by necessary coalesced memory access to the global memory.

Chapter 4

A GPU implementation of the eigenvalue problem for many small real non-symmetric matrices

4.1 Introduction

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

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

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 ren-

dering proposed in [1]. In this novel method, a scalar index for a design specification is calculated for each grid point in 3D space to get volume data and the permissible set is visualized as iso-surfaces in 3D space by volume rendering (Figure 4.1). 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 [33]. 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 [1], 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.

In classical numerical linear algebra, to compute eigenvalues of a non-symmetric matrix, the QR algorithm [34, 35] is usually employed. This algorithm is based on the factorization,

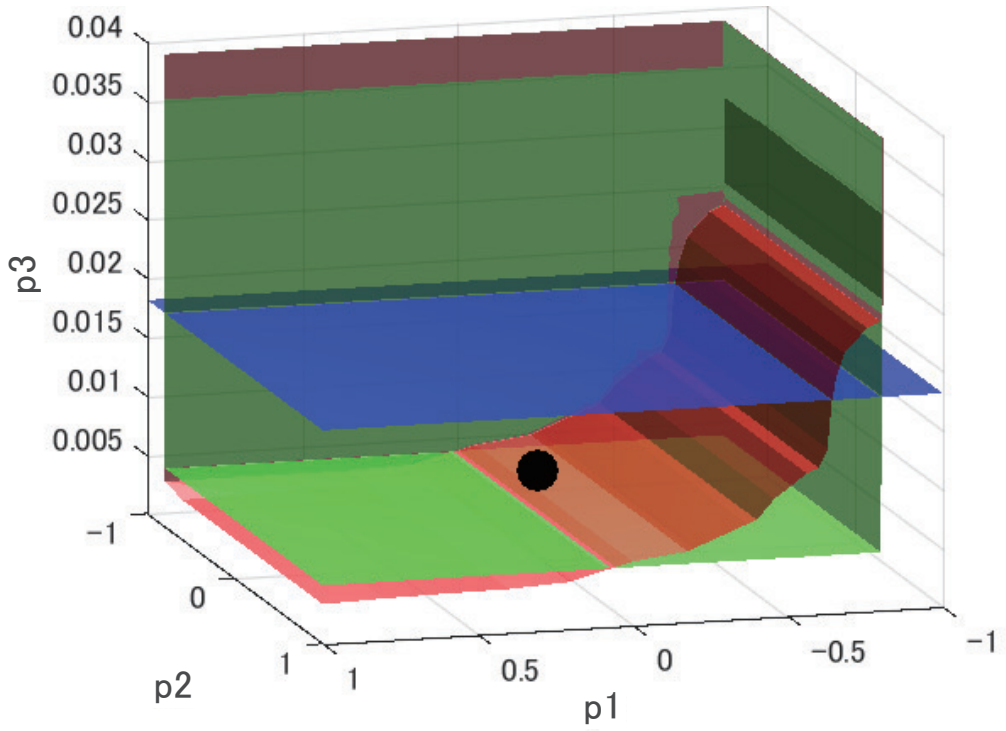


Figure 4.1: Volume rendering of the parameter space design in the pole assignment problems [1] using eigenvalues obtained by the proposed method

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 [36, 35]. Especially, in this work, we use *the implicit double-shift QR algorithm* [36] 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.

4.2 Related work

Several works have been devoted to accelerate the computation for matrix calculations for many small matrices using GPUs [37, 38, 39, 40]. Anderson *et al.* [37] 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 [40], 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.* [39] proposed a GPU implementation of the LU decomposition with pivoting for many dense matrices. Also, Cosnuau [38] 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 [41] and Intel MKL [42] are software libraries for the CPU implementations. These libraries support the computation of matrix factorizations, multiplications, eigenvalues, and so forth. For GPU implementations, MAGMA [21] , cuBLAS [43] and cuSOLVER [44] 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 [21] 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 [45] 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.

4.3 Eigenvalues Computation of a Non-symmetric Real Matrix

This section reviews the QR algorithm to compute the eigenvalues of a matrix [35]. 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* [36, 46]. 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 [36, 35, 46] 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 4.2. 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.

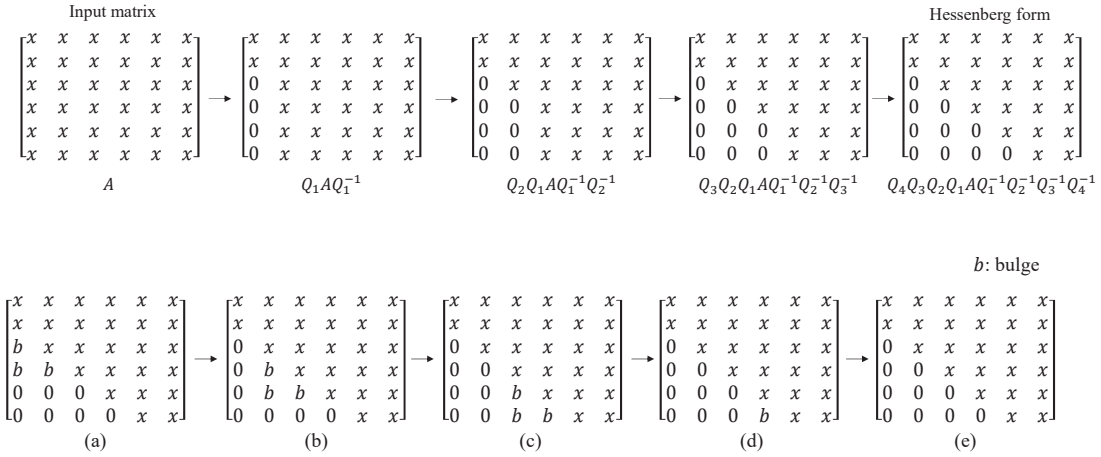


Figure 4.2: The Hessenberg reduction for a square matrix of size 6×6

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 4.3 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 4.3(a). After that, bulge-chasing moves the bulge down and to the right until it disappears (Figure 4.3(b)-(e)). By repeatedly performing the double-shift QR sweep, a value of a subdiagonal element converges to zero. After converg-

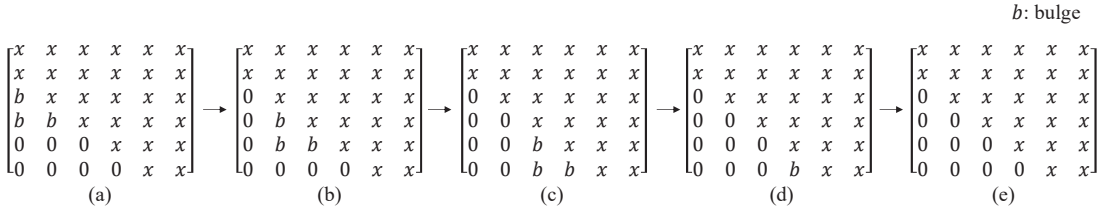


Figure 4.3: Bulge-generating and bulge-chasing in the double-shift QR sweep

ing, we split the matrix into the two smaller matrices by *deflation*. Deflation is decomposing an upper Hessenberg form matrix into the two smaller upper Hessenberg form matrices when a subdiagonal element converges to zero as illustrated in Figure 4.4. 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 $|a_{k+1,k}| \ll |a_{k,k}| + |a_{k+1,k+1}|$.

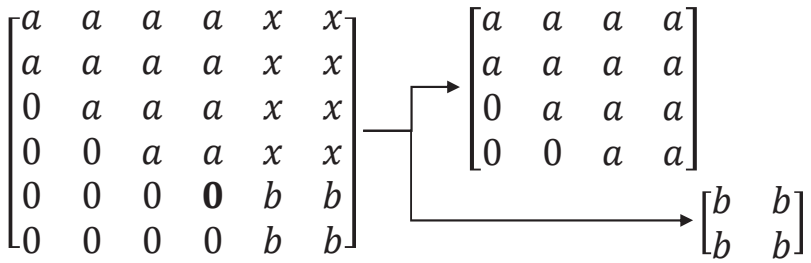


Figure 4.4: Matrix division by deflation

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.

4.3.1 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 4.2. 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 4.2. 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.

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

4.3.2 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 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 4.3(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.

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** $B \leftarrow H$

Bulge-chasing sweeps the matrix obtained by bulge-generating from the top to bottom by chasing the bulge as shown in Figure 4.3. To chase the bulge, we perform Householder

transformations from left and similarity transformations from right:

$$H = R_{n-2}R_{n-3}\cdots R_2R_1BR_1^{-1}R_2^{-1}\cdots R_{n-3}^{-1}R_{n-2}^{-1},$$

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

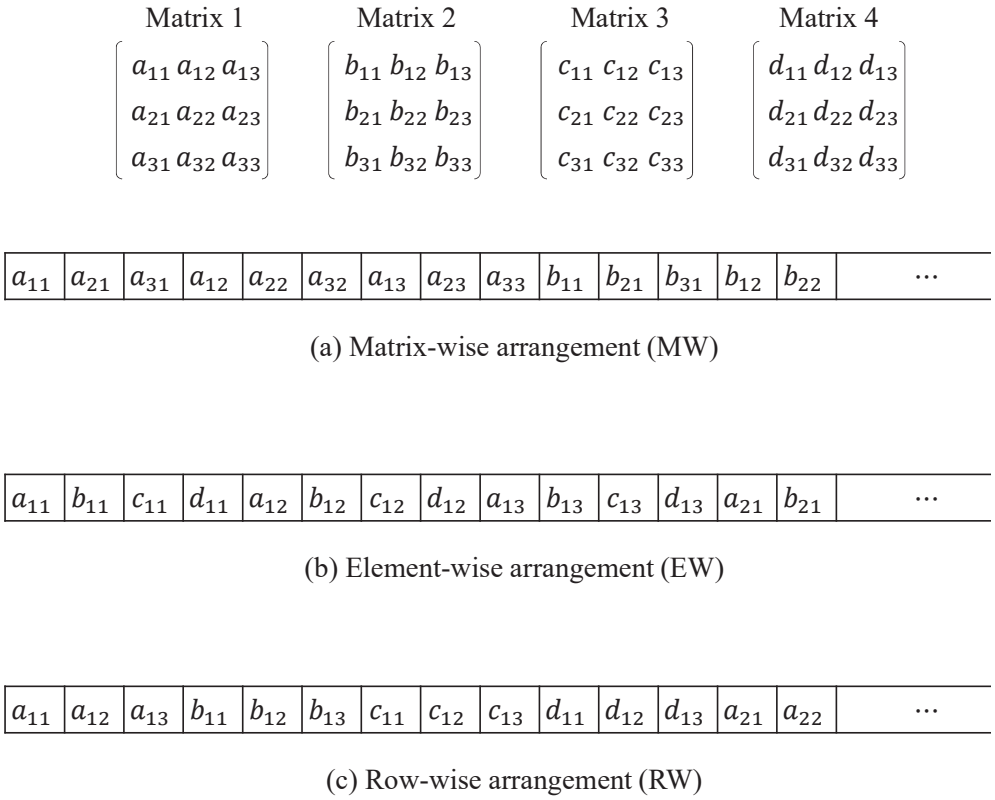
1: **for** $k = 1$ **to** $n - 2$ **do**
2: $\quad \mathbf{v} \leftarrow B_{k+1:k+3,k}$
3: $\quad \mathbf{v} \leftarrow \mathbf{v} + \text{sign}(v_1)||\mathbf{v}||\mathbf{e}_1$
4: $\quad \mathbf{v} \leftarrow \frac{\mathbf{v}}{||\mathbf{v}||}$ ▷ Householder vector
5: $\quad B_{k+1:k+3,k:n} \leftarrow B_{k+1:k+3,k:n} - 2\mathbf{v}_k(\mathbf{v}_k^T B_{k+1:k+3,k:n})$ ▷ Householder transformation
6: $\quad B_{1:k+4,k+1:k+3} \leftarrow B_{1:k+4,k+1:k+3} - 2(B_{1:k+4,k+1:k+3}\mathbf{v})\mathbf{v}^T$ ▷ Similarity transformation
7: **end for**
8: **return** $H \leftarrow B$

4.4 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 4.5(a). This arrangement is generally used in numeric linear algebra tools

and software libraries [21, 42, 45]. 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 4.5(b). On the other hand, in the RW arrangement, each row taken from the matrices is stored row by row as illustrated in Figure 4.5(c). Two arrangements EW and RW are used in the global memory to make the memory access efficient.



(a) Matrix-wise arrangement (MW)

(b) Element-wise arrangement (EW)

(c) Row-wise arrangement (RW)

Figure 4.5: Data arrangement for multiple matrices in the global memory

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 em-

ploy 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 [22]. 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 [23, 24, 25]. 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 4.6. 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

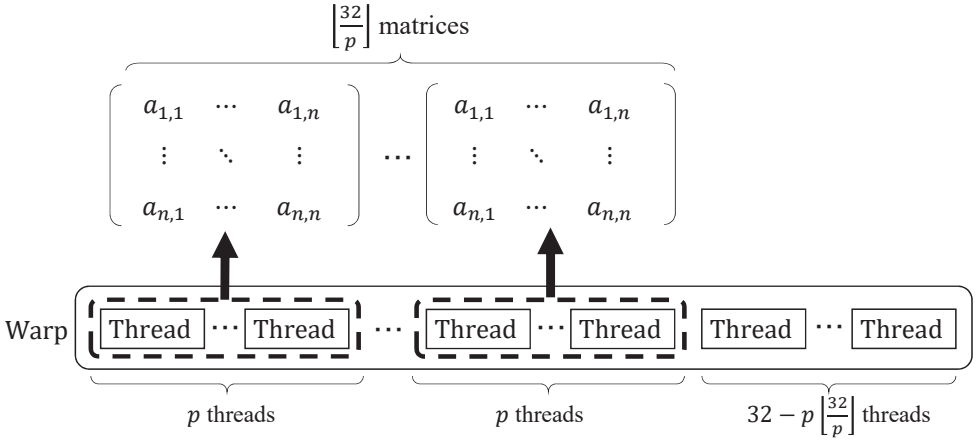


Figure 4.6: Single-warp-based method (SWB)

of all the processes in 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 4.7. 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 4.7. 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 [47] 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 4.8(a).

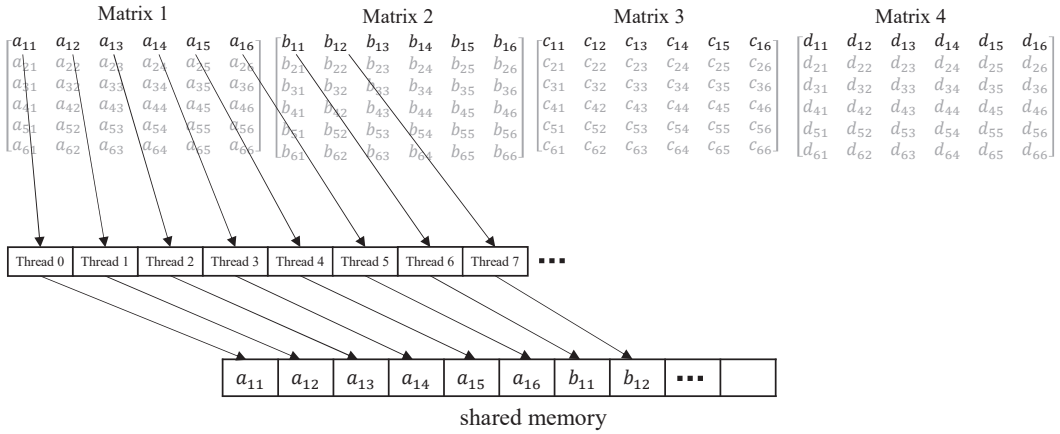
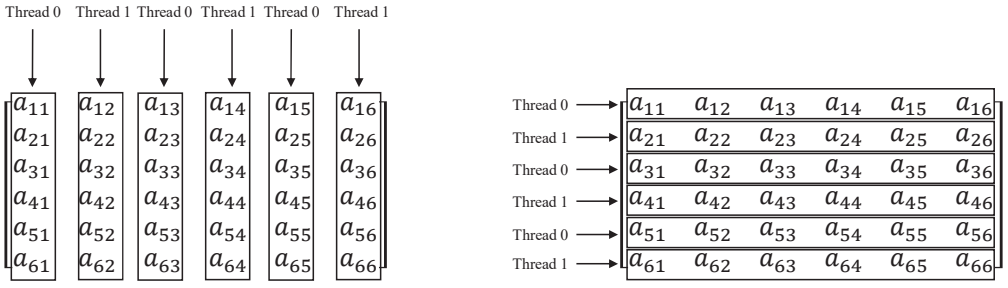


Figure 4.7: Matrix data loading from the global memory to the shared memory with coalesced access in SWB

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 4.8(b). Each thread computes the multiplication of the elements in the row from left to right.



(a) Householder transformation

(b) Similarity transformation

Figure 4.8: Thread assignment in SWB and MWB for $n = 6$ and $p = 2$

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 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 4.9. 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 4.8. 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.

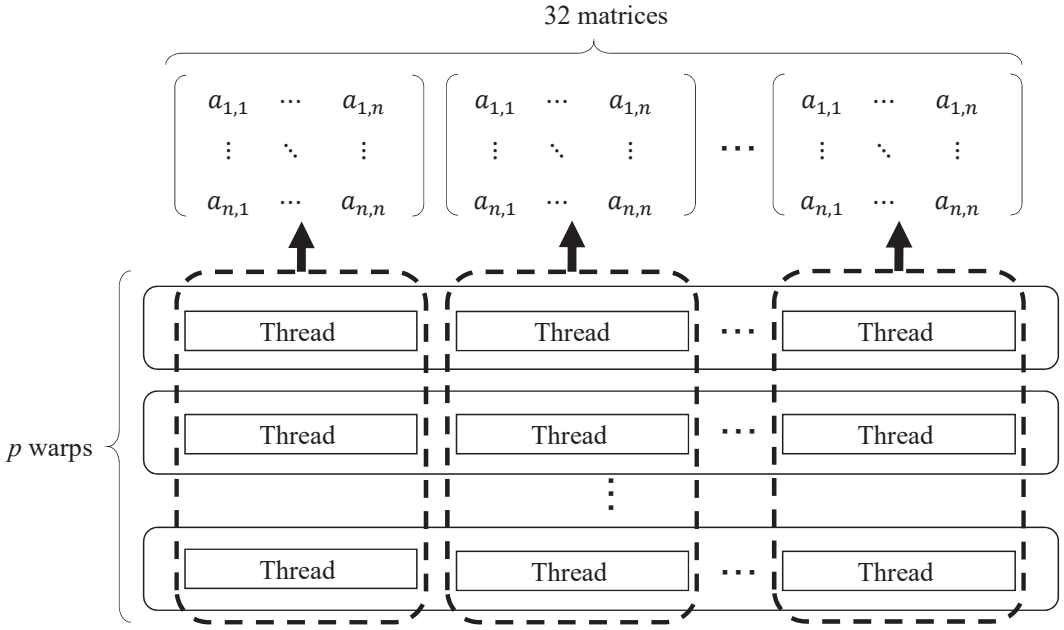


Figure 4.9: Multiple-warp-based method (MWB)

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 technique proposed in [48]. 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.

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 2.

4.5 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 [26]. 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.

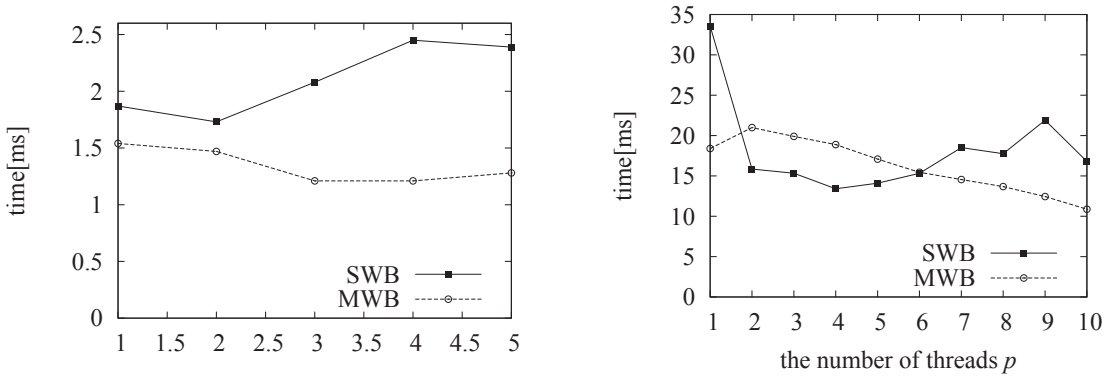
First, we evaluate the performance of Step 1, the Hessenberg reduction, using the proposed SWB and MWB methods. Figure 4.10 shows the computing time of the Hessenberg reduction. The evaluation has been carried out for different values of p. We note that when p is small, the SWB method cannot be executed 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 4.5. 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.

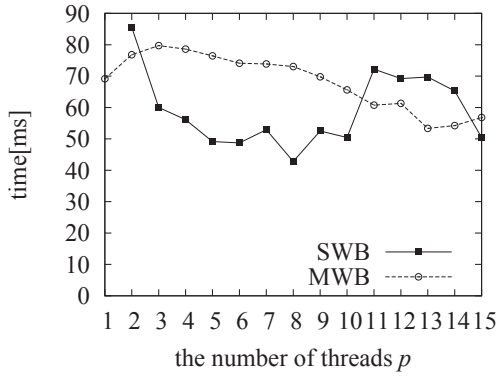
Figure 4.11 shows the computing time of Steps 2 and 3 for 500000 matrices of size from 5×5 to 30×30. Similarly, the computing time does not include data transfer time. Also, input data in the global memory are stored by the appropriate arrangement in Figure 4.5 for each method.

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

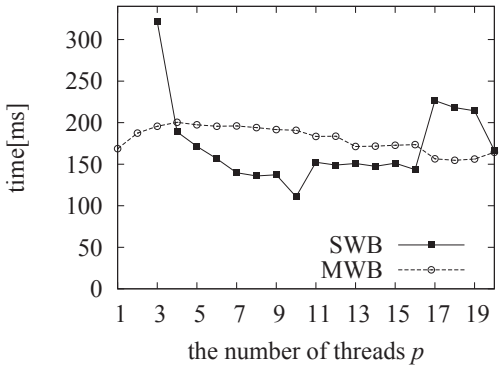


(a) 5×5

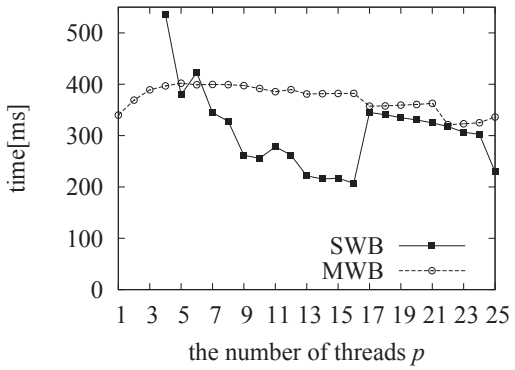
(b) 10×10



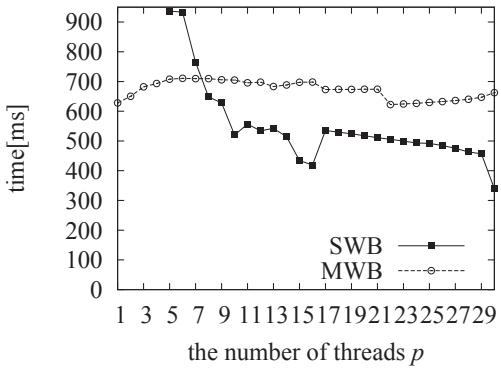
(c) 15×15



(d) 20×20



(e) 25×25



(f) 30×30

Figure 4.10: The computing time of Step 1 for 500000 matrices of size 5×5 to 30×30

Table 4.1: 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

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 4.1 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

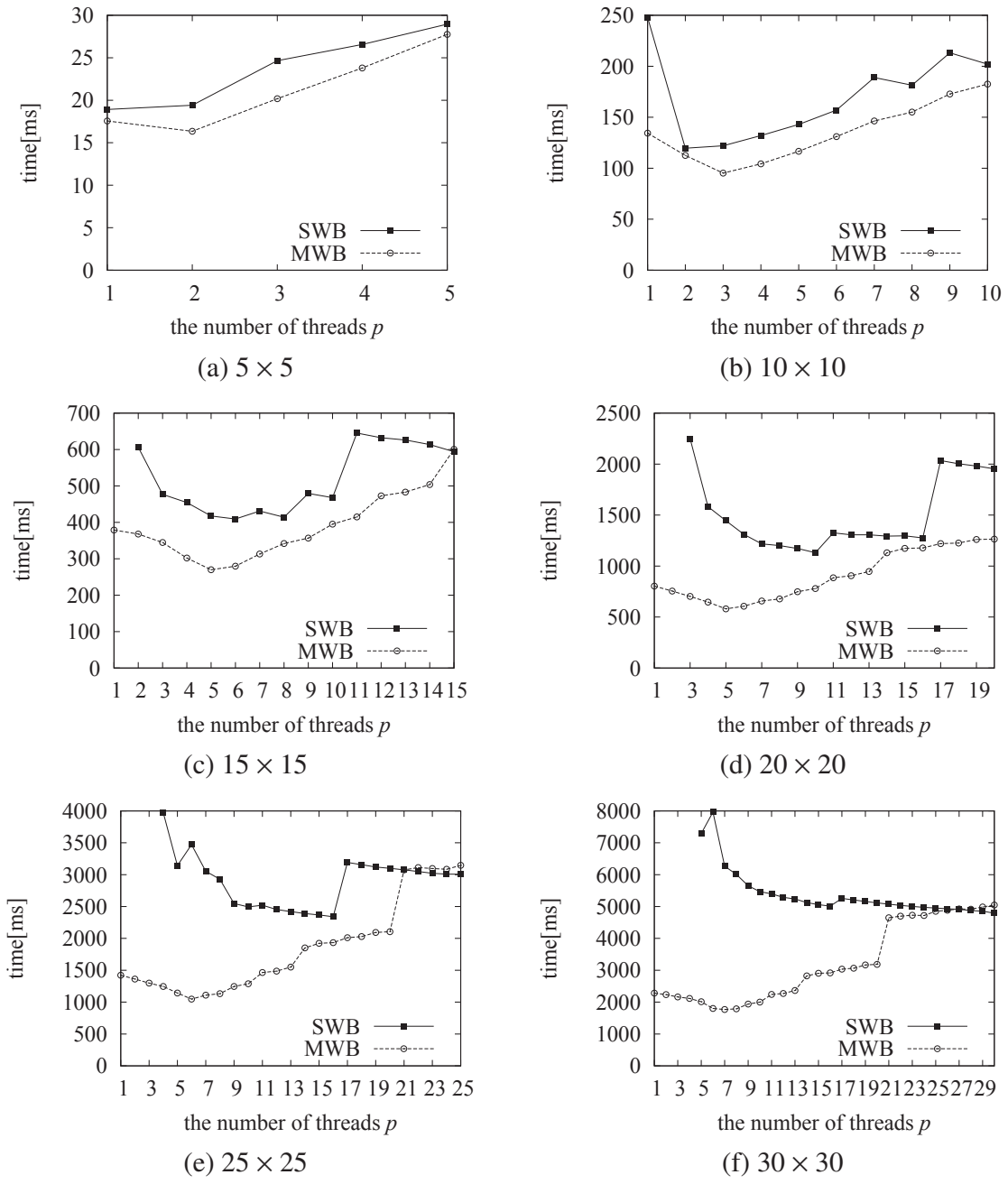


Figure 4.11: The computing time of Steps 2 and 3 for 500000 matrices of size 5×5 to 30×30

Table 4.2: 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

time accounts for more than half of the total computing time. Therefore, we evaluate the overlapped execution by multiple streams shown in Section 2. Table 4.2 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 4.1 and 4.2. 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 MKL 2017 Update 1 [42], MAGMA version 2.0.2 [21], and MATLAB version R2016b [45]. Table 4.3 shows the computing time of eigenvalues for 500000 matrices us-

ing existing tools and software libraries. Intel MKL supports two types of implementation,

Table 4.3: 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

sequential and parallel execution for 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 [49]. 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 4.4 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 4.4: 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

Chapter 5

Conclusion

In this dissertation, we have presented efficient GPU implementations for column-wise prefix sum computation and bulk computation of the eigenvalue problem for many small real non-symmetric matrices.

In Chapter 3, we have presented an almost optimal GPU implementation for column-wise prefix sum computation. The LCP algorithm involves several GPU computing techniques including the warp prefix scan, the diagonal arrangement of a matrix, and the decoupled look-back to minimize memory access and synchronization overhead. Quite surprisingly, experimental results using NVIDIA TITAN X show that our column-wise prefix-sum algorithm runs only 2-6% slower than matrix duplication. Thus, our column-wise prefix sum algorithm is almost optimal.

In Chapter 4, we have presented an efficient GPU implementations for bulk computation of the eigenvalue problem for many small real non-symmetric matrices. The ideas are to use the appropriate thread assignment and data arrangement for multiple matrices in the global memory. Also, data transfer hidden between host and device contributes to improve the performance. 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.

Bibliography

[1] 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*, Vol. 28, No. 10, pp. 419–425, 2015.

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

[3] 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*, pp. 153–159, Dec. 2011.

[4] 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. Vol. E96-D, No. 12, pp. 2617–2625, Dec. 2013.

[5] 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), pp. 1–10. IEEE CS Press, Oct. 2013.

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

[7] 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*, pp. 94–102. IEEE CS Press, Dec. 2012.

[8] NVIDIA Corporation. *CUDA Zone*. `https://developer.nvidia.com/cuda-zone`.

[9] Akihiko Kasagi, Tsuguchika Tabaru, and Hirotaka Tamura. Fast algorithm using summed area tables with unified layer performing convolution and average pooling. In *Machine Learning for Signal Processing (MLSP), 2017 IEEE 27th International Workshop on*, pp. 1–6. IEEE, 2017.

[10] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations. In *Proc. of International Conference on Parallel Processing (ICPP)*, pp. 251–250, Sept. 2014.

[11] Andrew Lauritzen. Chapter 8: Summed-area variance shadow maps. In *GPU Gems 3*. Addison-Wesley, 2007.

[12] Diego Nehab, André Maximo, Rodolfo S. Lima, and Hugues Hoppe. GPU-efficient recursive filtering and summed-area tables. *ACM Trans. Graph.*, Vol. 30, No. 6, p. 176, 2011.

[13] James R Lemen, David J Akin, Paul F Boerner, Catherine Chou, Jerry F Drake, Dexter W Duncan, Christopher G Edwards, Frank M Friedlaender, Gary F Heyman, Neal E Hurlburt, et al. The atmospheric imaging assembly (aia) on the solar dynamics observatory (sdo). In *The Solar Dynamics Observatory*, pp. 17–40. Springer, 2011.

[14] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Chapter 39. parallel prefix sum (scan) with CUDA. In *GPU Gems 3*. Addison-Wesley, 2007.

[15] Koji Nakano. Simple memory machine models for GPUs. *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 29, No. 1, pp. 17–37, 2014.

[16] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled look-back. Technical report, NVIDIA Technical Report NVR-2016-002, 2016.

[17] Karen Braman, Ralph Byers, and Roy Mathias. The multishift QR algorithm. Part II: Aggressive early deflation. *SIAM Journal on Matrix Analysis and Applications*, Vol. 23, No. 4, pp. 948–973, 2002.

[18] 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*, Vol. 41, No. 4, p. 29, 2015.

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

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

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

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

[23] 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*, pp. 1154–1157, 2010.

[24] 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*, pp. 136–145, 2011.

[25] 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*, pp. 619–629, 2011.

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

[27] 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*, Vol. 72, No. 9, pp. 1117–1126, 2012.

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

[29] Merrill D(2017). CUB: a library of warp-wide, block-wide, and device-wide gpu parallel primitives. https://nvlabs.github.io/cub/.

[30] 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*, Vol. 1, No. 2, pp. 260–276, July 2011.

[31] Koji Nakano. An optimal parallel prefix-sums algorithm on the memory machine models for GPUs. In *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, pp. 99–113. Springer, Sept. 2012.

[32] Koji Nakano. Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models. *IEICE Trans. on Information and Systems*, Vol. E96-D, No. 12, pp. 2626–2634, 2013.

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

[34] John GF Francis. The QR transformation a unitary analogue to the LR transformation–part 1. *The Computer Journal*, Vol. 4, No. 3, pp. 265–271, 1961.

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

[36] John GF Francis. The QR transformation–part 2. *The Computer Journal*, Vol. 4, No. 4, pp. 332–345, 1962.

[37] 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*, pp. 2–13, 2012.

[38] 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*, Vol. 29, pp. 800–810, 2014.

[39] 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*, pp. 157–160, 2014.

[40] 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*, pp. 31–47, 2015.

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

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

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

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

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

[46] Danny C Sorensen. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM Journal on Matrix Analysis and Applications*, Vol. 13, No. 1, pp. 357–385, 1992.

[47] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, Vol. 6, No. 2, pp. 40–53, 2008.

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

[49] OpenMP. http://www.openmp.org/.

Acknowledgement

First and foremost, I would like to show my deepest gratitude to my supervisor, Professor Koji Nakano for his continuous encouragement, advice and support. He is a respectable and resourceful scholar. His knowledge and research experience are in value through the whole period of my study. As a supervisor, he taught me skills and practices that will benefit my future research career.

I shall express sincere appreciation to my thesis committee members, Professor Satoshi Fujita and Associate Professor Yasuaki Ito for reviewing my dissertation.

I would also like to express my thanks to Assistant Professor Daisuke Takafuji for his support and continuous guidance in every stage of my study. My heartiest thanks go to all members of computer system laboratory. They were always kind and very keen to help.

I would like to offer my special thanks to Professor Masami Saeki, Mr. Mitsuya Nishino and Mr. Yushiro Hirota for their supports and providing technical guidance on the parameter space design method.

I would express thanks to all the faculty members of the Department of Information Engineering of Hiroshima University.

Last but not least, I wish to express my thanks to my parents who have always supported me.

List of publications

Journals

J-1: Hiroki Tokura, Toru Fujita, Koji Nakano, Yasuaki Ito, and Jacir L. Bordim, Almost optimal column-wise prefix-sum computation on the GPU, The Journal of Supercomputing, Vol. 74, No. 4, pp. 1510 - 1521, April 2018.

- Chapter 3: An GPU implementation of column-wise prefix-sum computation

J-2: Hiroki Tokura, Takumi Honda, Yasuaki Ito, Koji Nakano, Mitsuya Nishino, Yushiro Hirota, Masami Saeki, An Efficient GPU Implementation of Bulk Computation of the Eigenvalue Problem for Many Small Real Non-symmetric Matrices. IJNC Vol. 7, No. 2, pp. 227-247, July 2017.

- Chapter 4: An GPU implementation of the eigenvalue problem for many small real non-symmetric matrices

International Conferences

C-1: Hiroki Tokura, Toru Fujita, Koji Nakano and Yasuaki Ito, Almost Optimal Column-wise Prefix-sum Computation on the GPU, in Proc. of 12th International Conference of Parallel Processing and Applied Mathematics (PPAM 2017, LNCS 10778), pp.

224-233, Lublin, Poland, September 2017.

- Chapter 3: An GPU implementation of column-wise prefix-sum computation

C-2: 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, Proc. of International Symposium on Computing and Networking (CANDAR), pp. 490-496, November 2016.

- Chapter 4: An GPU implementation of the eigenvalue problem for many small real non-symmetric matrices

Invited Talks

T-1: 戸倉宏樹, 大量の小規模非対称行列の固有値計算のための GPU 実装, 科研費基盤 B 課題「O(1 億) コア環境におけるスケーラブルな数値計算ソフトウェアの理論と応用」ワークショップ, 札幌 (北海道)

- Chapter 4: An GPU implementation of the eigenvalue problem for many small real non-symmetric matrices

Others

C-3: Yutaro Emoto, Shunji Funasaka, Hiroki Tokura, Takumi Honda, Koji Nakano and Yasuaki Ito, An Optimal Parallel Algorithm for Computing the Summed Area Table on the GPU, Proc. of International Parallel and Distributed Processing Symposium Workshops, pp. 763-772, Vancouver, Canada, May 2018.

C-4: Hiroki Tokura, Yuki Kuroda, Yasuaki Ito, and Koji Nakano, A Square Pointillism Image Generation, and its GPU Acceleration, in Proc. of International Symposium on Computing and Networking (CANDAR), pp. 38-47, Aomori, Japan, November 2017.

C-5: Naoki Matsumura, Hiroki Tokura, Yuki Kuroda, Yasuaki Ito, Koji Nakano, Tile Art Image Generation Using Conditional Generative Adversarial Networks, in Proc. of International Symposium on Computing and Networking Workshops, pp. 209-215, Takayama, Japan, November 2018.