# GPU 상에서 다중행렬처리를 이용한 Hessenberg 축약 알고리즘의 고속 수행 기법

## (A Fast Solution of Hessenberg Reduction Algorithm Using Multi-Matrix Processing on GPU)

윌 리 엄 † 이 영 주 †† 박 인 규 †††
(Williem) (Youngjoo Lee) (In Kyu Park)

**요 약** 본 논문에서는 GPU 병렬처리 환경에서 다중행렬처리를 통한 Hessenberg Reduction 문제의 고속 처리 기법을 제안한다. Hessenberg Reduction 문제는 고유치 문제를 해결하기 위한 필수적인 단계로서, 이를 해결하기 위해 많은 계산량이 필요하다. 기존의 GPU 기반의 처리 기법은 작은 크기의 행렬 처리에서 CPU에서의 처리에 비해 효율적이지 못하다. 본 논문에서는 GPU의 자원이 허용하는 최대의 행렬을 동시에 처리함으로써 CPU와 GPU간의 메모리 전송률을 높이고 또한 GPU의 대용량 병렬처리 구조에 부합하여 고속 수행이 가능하도록 한다. 실험 결과 기존의 최적화된 상용 라이브러리와 비교하여 제안하는 알고리즘의 성능이 우수함을 보인다.

**키워드** : GPU 병렬처리, 다중행렬처리, 선형대수, 고유치 문제, Hessenberg Reduction

***Abstract*** In this letter, we propose a multi-matrix programming model in GPU computation to deal with Hessenberg reduction problem. Hessenberg reduction problem is the most important step in eigenvalue problem, which is computationally expensive. Conventional method using GPU is inefficient in GPU resource usage when it deals with small matrix. The proposed method computes as many matrices as possible which maximally utilizes the GPU resources. Therefore, the proposed method achieves higher memory transfer rate between CPU and GPU, and massive parallel computation that is well fit for GPU computation. Experimental results show that the proposed outperforms the optimized commercial packages.

**Key words** : GPU computation, multi-matrix programming, linear algebra, eigenvalue problem, Hessenberg Reduction

· 이 논문은 삼성전자와 인하대학교의 지원에 의하여 연구되었음

† 학생회원 : 인하대학교 정보통신공학과
williem_060689@hotmail.com
†† 비 회 원 : 삼성전자 생산기술연구소 책임연구원
yj6.lee@samsung.com
††† 종신회원 : 인하대학교 정보통신공학부 교수
pik@inha.ac.kr
(Corresponding author임)

논문접수 : 2012년 3월 2일
심사완료 : 2012년 4월 26일


정보과학회논문지: 컴퓨팅의 실제 및 레터 제18권 제7호(2012.7)

## 1. Introduction

Hessenberg reduction problem (computing Hessenberg matrix of a square general matrix) in linear algebra plays an important role in computing non-symmetric eigenvalue problem. Note that non-symmetric eigenvalue problem is utilized for many different fields [1]. Hessenberg reduction problem becomes important because it is the first step before computing of Schur decomposition in non-symmetric eigenvalue problem. In addition, Hessenberg reduction computation reduces the complexity of Schur decomposition by simplifying the matrix. However, it is expensive to solve Hessenberg reduc-

tion problem, especially when it is computed in CPU architecture. Therefore, there are many studies to reduce the computational complexity of Hessenberg reduction problem [2-4].

There exist a few open sources or commercial packages for linear algebra, which handle Hessenberg reduction problem. The standard library is known as LAPACK (Linear Algebra PACKage) [5,6]. LAPACK is based on BLAS (Basic Linear Algebra Subprograms) package [7] which consists of the standard routines for matrix and vector computation. Based on LAPACK, Intel released the optimized linear algebra library, *i.e.* Intel MKL (Math Kernel Library) [8], for Pentium CPU architecture.

Recently, GPGPU (General Purpose GPU) computation has become popular in various fields in science and engineering, which aims to achieve significant speedup of computationally heavy problems by means of massive parallel computing on the GPU architecture. During last a few years, CUDA [9] has been a parallel computing framework that makes GPGPU implementation much easier and efficient than before. To deal with linear algebra computation on GPU, NVIDIA introduced CUBLAS [10], which is in fact a BLAS implementation on CUDA programming model. Based on CUBLAS, EM Photonics released CULA library, which consists of optimized linear algebra algorithm on GPU hardware [11].

In this paper, we propose a fast and efficient algorithm for Hessenberg reduction problem using massive parallel computing on the state-of-the-art GPU architecture. In order to utilize GPU capability maximally, instead of computing only one matrix, we propose multi-matrix programming model. Unlike the existing work [4], which has slow data transfer speed and inefficient computation on small matrices, the proposed approach uses multi-matrix programming that is more efficient.

This paper is organized as follows. In Section 2 and Section 3, we describe the existing linear algebra libraries and algorithms, respectively. Section 4 describes the multi-matrix programming model. Experimental results are shown in Section 5. Finally, we give a conclusive remark in Section 6.

## 2. Existing Linear Algebra Libraries and Implementation

### 2.1 BLAS

BLAS is the standard linear algebra routines to perform matrix and vector operations [7]. BLAS has three levels of routines based on the type of operations. BLAS level 1 is the group of vector operations, while BLAS level 2 is the set of matrix-vector operations. BLAS level 3 consists of matrix-matrix operations.

### 2.2 LAPACK

LAPACK is the standard library that performs the linear algebra algorithms on top of BLAS routines [5]. LAPACK is written in Fortran 90 language. LAPACK provides the linear algebra algorithms, such as eigenvalue decomposition, singular value decomposition, and others. Based on distinct architecture or programming language, LAPACK has derived versions including ScaLAPACK [12] and CLAPACK [13].

### 2.3 Intel MKL

Intel MKL is an optimized library that provides not only linear algebra functions but also fast Fourier transforms, vectorized mathematics, and random number generation functions. Intel MKL is based on LAPACK library and is optimized for the latest architecture of Intel's multicore processors by using OpenMP technology [8], which shows much improved performance than standard LAPACK.

### 2.4 CUBLAS

CUBLAS is the GPU version of BLAS package and is implemented on CUDA programming model [10]. It contains the optimized version of BLAS that exploits the massive parallelism of GPU. It provides considerably faster speed than CPU computation when the matrix size is large enough.

### 2.5 CULA

CULA is a high-performance linear algebra library that executes in a CPU/GPU hybrid environment [11]. CULA employs Intel MKL as default library for CPU computation, while CUBLAS and a few modified kernel functions are used for GPU computation. CULA is more powerful than Intel MKL when it handles large matrices since GPU is more utilized with higher occupancy.

### 2.6 Hybrid Implementation

Computation in GPU architecture would have powerful performance when it handles massive data computation with high floating-point operation intensity. In case the floating-point operation intensity of each thread is small or the parallel thread has much dependency, the performance decreases dramatically. To avoid this problem, the hybrid programming was proposed previously [3]. The idea is to separate the algorithm into sub algorithms so that some of them run on CPU and the others run on GPU according to the nature of the computation in sub algorithms. However, there should be inevitable memory copy whenever GPU and CPU switch their role, which is the major bottleneck of performance throughput of hybrid implementation.

## 3. Numerical Algorithm for Eigenvalue Problem

### 3.1 Complex Non-Symmetric Eigenvalue Decomposition

Eigenvalue problem is used to decompose a $N \times N$ complex matrix $\mathbf{A}$ to eigenvalues $\lambda_i$ and eigenvectors $\mathbf{u_i}$ which satisfy $\mathbf{Au_i} = \lambda_i \mathbf{u_i}$ $(0 \le i \le N-1)$. The standard algorithm consists of four main steps as follows.

- Step 1. Reduction to upper Hessenberg form. This function converts $\mathbf{A}$ into an upper Hessenberg matrix by applying Householder transformation.
- Step 2. Generating a unitary matrix by accumulating Householder reflectors computed in the previous step.
- Step 3. Performing QR iteration and accumulating Schur vectors. This function computes the eigenvalues by means of Schur decomposition.
- Step 4. Computation of eigenvectors. This function computes the eigenvectors of input matrix.

In parallel processing of eigenvalue problem, step 1 is the most important step since there is less dependency between parallel tasks. Note that step 1 consists of not only BLAS level 3 but also BLAS level 1 and 2 of which performance is usually bounded by memory throughput. Therefore, Hessenberg reduction algorithm is considered as the main target in this paper.

### 3.2 Hessenberg Reduction

In order to generate an upper Hessenberg matrix $\mathbf{H}$, the general matrix $\mathbf{A}$ is computed by an orthogonal similarity transformation known as Hessenberg reduction algorithm as follows.

$$\begin{aligned} \mathbf{H} &= \mathbf{Q}^{\mathbf{T}}\mathbf{A}\mathbf{Q} \\ \mathbf{Q} &= \mathbf{Q_1}\mathbf{Q_2}\cdots\mathbf{Q_{n-2}} \\ \mathbf{Q_i} &= \mathbf{I} - \tau_i \mathbf{v_i}\mathbf{v'_i} \end{aligned} \tag{1}$$

Householder reflector $\mathbf{Q_i}$ is computed by subtracting identity matrix $\mathbf{I}$ with the multiplication of scalar $\tau_i$ and vector $\mathbf{v_i}$. The product of Householder reflector $\mathbf{Q_i}$ produces the matrix $\mathbf{Q}$. However, Hessenberg reduction algorithm does not contain any BLAS level 3 functions which have more possibility of optimization on multicore architecture. Therefore, block Hessenberg reduction algorithm is employed in this paper to solve the problem more efficiently. In the block Hessenberg reduction algorithm, Householder reflectors are accumulated together as many as n block size before they are accumulated with matrix $\mathbf{A}$ as shown in (2).

$$\mathbf{Q_1}\mathbf{Q_2}\cdots\mathbf{Q_n} = \mathbf{I} - \mathbf{VTV'} \tag{2}$$

where $\mathbf{V}$ is a set of n vectors and $\mathbf{T}$ is a $n \times n$ upper triangular matrix. This transformation is known as compact WV transform [14,15] and performs more BLAS level 3 functions.

## 4. Multi-Matrix Programming Model

In GPU computing, it is strongly recommended to have massive parallel task so that GPU is fully occupied. However, smaller matrix computation cannot fully utilize GPU's computing units (resources). Hence, in this paper, multi-matrix programming is proposed to overcome the problem. Multi-matrix programming computes as many matrices as possible simultaneously. Consequently, GPU resources are utilized as maximally as possible. In addition, all functions that have been computed inside CPU architecture in conventional approaches are computed concurrently in GPU computation. Therefore, the amount of data transfer between CPU and GPU can be reduced significantly. The block diagram of multi-matrix programming model is illustrated in Fig. 1. The implementation and optimization on GPU are described in the following subsections.

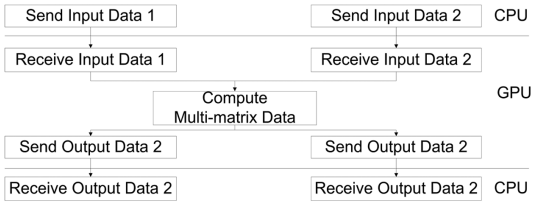



Fig. 1 Multi-matrix programming flowchart

### 4.1 Hessenberg Reduction Implementation

The accumulations of Householder reflectors in block Hessenberg reduction algorithm is computed by BLAS level 1, BLAS level 2, and a few functions that are not included in BLAS. Note that those functions do not fully utilize GPU resources, especially in small matrix computation. On the other hand, the proposed method is suitable even for small-sized multi-matrix processing since it can utilize GPU resources maximally. In addition, block size used in accumulation of Householder reflectors influences the amount of data, which is computed in kernel. Each kernel function simply computes some part of the whole data. In order to parallelize multi-matrix processing, multi-matrix data is transferred from CPU to GPU before accumulating the Householder reflectors. Then, the matrices are computed by calling kernel functions. Each kernel function is designed to handle more than a single matrix by separating the data computation based on thread blocks. After all computations have been done, the resultant multi-matrix data is returned back to CPU architecture.

Table 1 Hardware Specification

| Type | Specification |
|---|---|
| CPU | Intel Core i5 750 @ 2.67 GHz |
| Operating System | Windows 7 Enterprise 32-bit |
| Memory (RAM) | 4096 MB |
| GPU | NVIDIA Tesla C1060 |
| CUDA version | 3.2 |
| CUBLAS version | 3.2 |
| Intel MKL version | 10.3 |
| CULA version | Premium R11 |

### 4.2 Optimization & Limitation

In implementing Hessenberg reduction algorithm, several optimization methods are used. Memory coalescing is performed in this implementation by designing each kernel function to access GPU global memory sequentially. Therefore, CUDA thread warp is able to access the global memory simultaneously. Shared memory, which has much faster speed access than global memory, is divided into banks. In order to access shared memory effectively, bank conflicts, which occur if there are multiple accesses to the same bank, have to be avoided. Therefore, shared memory is designed to be accessed in sequence to avoid the bank conflicts.

Loop unrolling is also performed in this implementation to remove the branch and increment operation in each iteration. Therefore, the loop operation can be computed efficiently by unrolling the loop in kernel functions. Each kernel function is separated into different thread blocks depends on the task or data. It is performed in order to avoid divergent branches, which occur if CUDA thread warp has different branch operation.

However, there is a limitation on the proposed method, which is caused by the limited number of arguments and instructions in each CUDA kernel function. Therefore, multi-matrix programming model has a maximum number of matrices can be computed together. The maximum number of matrices is up to eight matrices in our experimental environment.

## 5. Experimental Results

In order to evaluate the performance of the proposed approach, we evaluate the GPU memory bandwidth of vector data transfer between GPU global memory (CCOPY) and the execution time of Hessenberg reduction (CGEHRD). The experimental environment is listed in Table 1. We compare the performance with CULA and Intel MKL for randomly generated matrices with different dimension from 500×500 to 3000×3000.

### 5.1 Memory Bandwidth Test

We select one of BLAS functions used in Hessenberg reduction algorithm, i.e. CCOPY to test the advantage of multi-matrix processing in memory copy bandwidth. CCOPY could not utilize GPU resources maximally due to insufficient parallelism. Note that CCOPY is called a lot of times in Hessenberg reduction algorithm. Therefore, it is

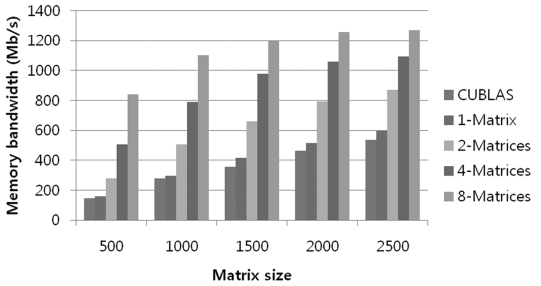



Fig. 2 CCOPY memory bandwidth result (in MB/s)

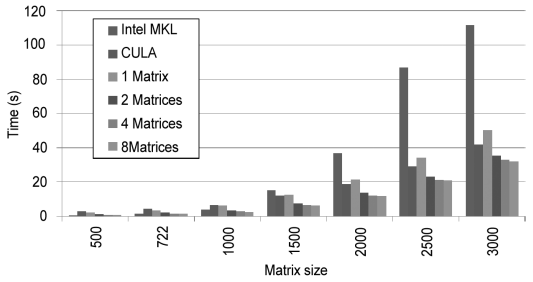



Fig. 3 Runtime comparison of Hessenberg reduction (in sec)

needed to have high memory bandwidth. Quantitative comparison is performed by using NVIDIA's Visual Profiler, which records the execution time. The execution time includes invocation and execution of GPU kernel (GPU memory copy) for eight different complex vectors with sizes varying from 500 to 2500 for each computation. We compute the corresponding GPU memory bandwidth by dividing data size with execution time. Fig. 2 illustrates the bandwidth comparison of the proposed method and CUBLAS for different level of multi-matrix processing. It is clearly shown that the proposed method outperforms CUBLAS implementation significantly. Note that the proposed method does not only improve the memory bandwidth but also reduce the kernel invocation overhead by minimizing the number of invoking kernel.

### 5.2 Hessenberg Reduction Result

Experiment result of Hessenberg reduction is shown in Fig. 3, along with the comparison with competing linear algebra libraries for different matrix sizes. It is shown that the proposed method gains significant speed-up over others. In addition, the proposed multi-matrix programming model shows better performance (in case of 8 matrices processing) than Intel MKL on small matrix computation. In this case, CULA, which is powerful in large matrix computation, is not adequate to handle small matrices. The experiment result proves that the proposed multi-matrix programming model overcomes the existing problem of inefficient GPU resources usage and slow data transfer speed. Note that 8 matrices computation does not have significant speed-up when it is compared with two matrices computation. It is primarily because GPU resources are fully utilized already in two matrices computation.

## 6. Conclusions

In this paper, we proposed a fast method for Hessenberg reduction problem by using multi-matrix programming model on GPU. Massive parallel processing was fully utilized even for small matrices. Experimental result showed that the proposed method achieved higher memory copy bandwidth as well as faster processing time compared with the existing commercial libraries.

## 참 고 문 헌

[1] Z. Bai, D. Day, J. Demmel, and J. Dongarra, "A test matrix collection for non-hermitian eigenvalue problems," *University Tennessee*, Technical Report CS-97-355, 1997.

[2] G. Quintana-Orti and R. Geijn, "Improving the performance of reduction to hessenberg reduction," *ACM Transactions on Mathematical Software*, vol. 32, pp.180-194, Jun. 2006.

[3] S. Tomov, R. Nath, and J. Dongarra, "Accelerating the reduction to upper hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing," *Parallel Computing*, vol.36, pp.645-654, Dec. 2010.

[4] J. Muramatsu, S. L. Zhang, and Y. Yamamoto, "Acceleration of hessenberg reduction for non-symmetric eigenvalue problems using GPU," *Proceeding First International Conference on Networking and Computing*, pp.215-219, Nov. 2010.

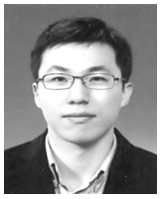
[5] *LAPACK: Linear Algebra PACKage*, http://www.netlib.org/lapack/.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK Users's Guide," *Society for Industrial and Applied Mathematics*, Philadelphia, 1992.

[7] L. Blackford, W. Demmel, J. Dongarra, I. Duff, S. Hammaring, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington,

and R. C. Whaley, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol.28, pp.135-151, Jun. 2002.
[8] Intel Corporation, *Intel Math Kernel Library (MKL)*.
[9] NVIDIA Corporation, *CUDA C Programming Guide version 3.2*.
[10] NVIDIA Corporation, *CUDA CUBLAS Library*.
[11] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini and E. J. Kelmelis, "CULA: Hybrid GPU accelerated linear algebra routines," *Proc. SPIE 7705 (Modeling and Simulation for Defense Systems and Applications V)*, p.770502, Apr. 2010.
[12] *SCALAPACK: Scalable Linear Algebra PACKage*, http://www.netlib.org/scalapack/.
[13] *CLAPACK: C Linear Algebra PACKage*, http://www.netlib. org/clapack/.
[14] R. S. Schreiber and C. Van Loan, "A storage efficient WY representation for products of householder transformations," *SIAM Journal on Scientific and Statistical Computing*, vol.10, pp.53-57, Jan. 1989.
[15] C. H. Bischof and C. Van Loan, "The WY representation for products of householder matrices," *SIAM Journal on Scientific and Statistical Computing*, vol.8, pp.S2-S13, 1987.

Williem
2011년 7월 인도네시아 비누스대학교 컴퓨터과학과 학사. 2011년 8월~현재 인하대학교 정보통신공학과 석박사통합과정. 주관심분야는 컴퓨터 그래픽스 및 비젼(영상기반 3차원 형상 복원, 3차원 카메라, GPGPU



이 영 주
2005년 2월 서울대학교 산업공학과 학사. 2007년 2월 서울대학교 산업공학과 석사. 2010년 2월 서울대학교 산업공학과 박사. 2010년 3월~현재 삼성전자 생산기술연구소 책임연구원. 주관심분야는 패턴인식, 영상처리, 컴퓨터 보조 진단, 의료영상, 고성능 컴퓨팅

박 인 규
1995년 2월 서울대학교 제어계측공학과 학사. 1997년 2월 서울대학교 제어계측공학과 석사. 2001년 8월 서울대학교 전기컴퓨터공학부 박사. 2001년 9월~2004년 3월 삼성종합기술원 멀티미디어랩 전문연구원. 2007년 1월~2008년 2월 미국 Mitsubishi Electric Research Laboratories (MERL) 방문연구원. 2004년 3월~현재 인하대학교 정보통신공학부 부교수. 주관심분야는 컴퓨터 그래픽스 및 비젼(영상기반 3차원 형상 복원, 3차원 카메라, Mobile computational photography), GPGPU