Acceleration of Hessenberg Reduction for Nonsymmetric Matrix

by

Hesamaldin Nekouei

Bachelor of Science Degree in Electrical Engineering
Iran University of Science and Technology, Iran, 2009

A thesis
presented to Ryerson University
in partial fulfillment of the
requirements for the degree of
Master of Applied Science
in
Electrical and Computer Engineering Program

Toronto, Ontario, Canada, 2013

©Hesamaldin Nekouei, 2013

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Hesamaldin Nekouei

Acceleration of Hessenberg Reduction for Nonsymmetric Eigenvalue , Master of Applied Science, 2013, Hesamaldin Nekouei, Electrical and Computer Engineering Program with Specialization in telecommunication engineering, Ryerson University

Abstract

The worth of finding a general solution for nonsymmetric eigenvalue problems is specified in many areas of engineering and science computations, such as reducing noise to have a quiet ride in automotive industrial engineering or calculating the natural frequency of a bridge in civil engineering. The main objective of this thesis is to design a hybrid algorithm (based on CPU-GPU) in order to reduce general non-symmetric matrices to hessenberg form. A new blocks method is used to achieve great efficiency in solving eigenvalue problems and to reduce the execution time compared with the most recent related works. The GPU part of proposed algorithm is thread based with asynchrony structure (based on FFT techniques) that is able to maximize the memory usage in GPU. On a system with an Intel Core i5 CPU and NVIDIA GeForce GT 635M GPU, this approach achieved 239.74 times speed up over the CPU-only case when computing the Hessenberg form of a $256 * 256$ real matrix. Minimum matrix order (n), which the proposed algorithm supports, is sixteen. Therefore, supporting this matrix size is led to have the large matrix order range.

Acknowledgment

I dedicate this study to spirit of my father and brother.

I wish to acknowledge those who I feel have greatly aided me in completing this thesis.

I would like to thank Professor Lian Zhao for providing an open study and discussion environment, having faith in me through out the project, constant encouragement and willingness to provide advice, and very clear guidance towards the success of this thesis project. I would also like to thank Professor Minco He for his enlightening guidance, constructive suggestions, high-standard requirement, and unconditional support.

Special thanks to my classmates, without whose help I would not have been able to get through this difficult and emotional time. Thanks to all friends who ever helped me over the past two years.

A special thanks to my family. Words cannot express how grateful I am to my mother and sister for all of the sacrifices that you have made on my behalf. Your prayer for me was what sustained me thus far. I would also like to thank all of my friends, Specially Ahad Yarazavi, who supported me in writing, and incented me to strive towards my goal. At the end I would like express appreciation to my beloved fiancée, Pantea, who spent sleepless nights with and was always my support in the moments when there was no one to answer my queries.

Contents

1	Introduction	1
1.1	Thesis Motivation	2
1.2	Research Contributions	3
1.3	Thesis Outline	5
2	Related Work	7
2.1	Background Knowledge	7
2.1.1	Graphics Processing Unit (GPU)	7
2.1.2	Dense Linear Algebra	18
2.1.3	Two-sided Factorizations	19
2.1.4	One-sided Factorizations	20
2.2	Literature Survey	22
3	General Algorithm Procedure	25
3.1	Solve Eigenvalue Problem from Nonsymmetric Matrix	25
3.1.1	Block Annihilation Method	27
3.2	Block Method Algorithm to Reach Hessenberg Matrix Form	28
3.3	Implementation procedure in Serials	31
3.4	Implementation procedure in Serials/Parallel	38
4	Experiment Results	45
4.1	Processing Time	45
4.2	CPU Processing Time vs CPU/GPU Processing Time	46

4.3	Speedup Ratio	49
4.4	Comparison	51
5	Discussion and Conclusion	55
5.1	Future Work	55
	Bibliography	57
A	Abbreviation List	61

List of Tables

4.1	Execution time (in seconds) of Hessenberg reduction.	46
4.2	Execution time (in micro-second) for serials and serials/parallel implementation in different stages with different size of n	50
4.3	Execution time (in seconds) of Hessenberg reduction.	50

List of Figures

2.1	CPU vs. GPU comparison of floating point operations per second. [10]	8
2.2	CPU vs. GPU bandwidth comparison. [10]	9
2.3	Transistor allocation for CPU and GPU. [10]	9
2.4	CUDA Grid layout. [10]	12
2.5	CUDA memory heirarchy. [10]	15
2.6	Usage of the CUBLAS Library.[2]	23
2.7	An overview of the CUDA programming model in [29]	24
3.1	Execution time of the four steps. [2]	27
3.2	Convert to Hessenberg Matrix Form	29
3.3	Block Method Annihilation.[4]	30
3.4	Schematic diagram of annihilation for HH	37
3.5	Maximum implementation procedure for each thread in UGZ stage	40
3.6	Schematic diagram of implementation in CUDA	42
3.7	Schematic diagram of Hybrid algorithm in CUDA	43
4.1	Block algorithms based on CPU.	47
4.2	new block algorithms based on CPU/GPU.	48
4.3	Execution time (in seconds) of Hessenberg reduction.	51
4.4	Speedup ratio based on matrix order (n).	52
4.5	compare algorithm efficiency based on matrix order(n).	53

Chapter 1

Introduction

Eigenvalue problems are demonstrated with $Ax = \lambda x$ where scalars λ and vectors $x \neq 0$ are called eigenvalues and eigenvectors of matrix A , respectively [1]. Eigenvalue is a significant figure for many applications in the areas of physical sciences and engineering. Building a bridge in civil field, reducing noise to have a quiet ride in industrial engineering, characteristic mode analysis in antenna and optical transmission line field in wire communication are several examples that demonstrate the importance of this topic.

The worth of eigenvalue problems can be understood better, with the following examples:

- Statical properties of subclass of random matrices determine performance of multiple-input multiple-output (MIMO) systems. Based on [2], the probability density function (pdf) of eigenvalues of these matrices will be used in this particular system.
- High-Speed and high-performance computing environments have crucial roles for Linear Algebraic operations and therefore reducing operation time in this field will be appreciated [3].

The worth of eigenvalue problems topics motivate us to research about latest achievement in this field and promote the latest algorithms with a kind of hybrid model (which is a combination of serial and parallel processing). The goal is to design high efficient algorithms compared with the latest algorithms available in the literature compared.

1.1 Thesis Motivation

There are many software companies that have applications their customers are always seeking fast runtime. There has always been the pressure to make such applications run faster. As processors have increased in speed, the requested speed up could be achieved by tuning the single CPU performance of the program and by utilizing the latest and fastest hardware. For example, the speed and memory capability of the newest machines have always been a reason to design the next generation chips in the Electronic Design Automation industry. One of the best ways to save power is parallel processing. The implementation of the parallel program on the several processors can be more efficient. Note that if the parallel coding is inefficient, it means that the parallel program will use more power on the slower processors than the serial program running on the fast single processor. However, there are limitations to make new faster processors. Parallel processing is one solution for faster implementation. One of the software developer responsibilities is to write programs that are as efficient as possible and which make use of N processors. This is something really new and difficult task for most developers.

Matrix multiplication is a very important operation in numerical computation. Therefore, speeding up matrix multiplication can be an important parameter in numerical computation topic. Basic Linear Algebra Subprograms (BLAS) is used as a basic numerical calculation library. These libraries have a great performance enhancement on Central Processing Units (CPUs).

A Graphic Processing Unit (GPU) is better to use for parallel processing compared with a CPU. It has ability to perform various types of computation, including numerical computations. General-purpose computations on GPUs (GPGPU) have been examined for various applications. The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library is

a GPU-accelerated version of the complete standard BLAS library for GPUs. There are some reasons that we cannot choose GPUs for all matrix multiplications such as solving eigenproblems of a general nonsymmetric matrix.

There are some limitations to use GPUs because of following reasons:

- Blocks (block of threads) synchronization.
- Limitation to use shared memory and distributed memory when using built libraries such as CULA.
- Threads synchronization.

The methods that use only GPUs generally work well for very large matrices but are inefficient for small matrix sizes. However, the methods that use only CPU have better performance for small matrix sizes. Recently, some new methods were introduced, which combine CPU and GPU, to achieve better results but still there is not significant improvement in these methods. These problems motivate us to investigate and propose an algorithm with great efficiency (in both small and large matrix sizes) for solving eigenvalue problems. It is very important for us that this algorithm can cover both symmetric and nonsymmetric matrices.

1.2 Research Contributions

There are two steps to achieve eigenvalues of a general nonsymmetric matrix. First step is to convert the general matrix to hessenberg form and then convert the hessenburg form matrix to upper triangular matrix. Hessenberg matrix form reduction is a big step procedure because of high complexity processing in this step. The main objective of this research is to design efficient techniques to reduce a general nonsymmetric matrix to hessenberg form without using shared memory and synchronization. We also try to accelerate this step.

In this thesis, we improve block algorithm with implementing in CUDA. This algorithm is a combination of block method, which is introduced in [4, 5], and Fast Fourier Transform (FFT) algorithm. Practically, it is hard to formulize the blocks independently without any connection between them and using shared memory, which we did as combination of block method and FFT algorithms. This idea is completely new that in each stage of proposed algorithm, the variables (blocks) in the current stage do not affect each other and are independent. Each variable is updated only based on previous stage variables. It means that a variable in the current column is a function of the previous stage variables. Therefore, this proposed algorithm can work asynchronously. Each matrix block is coded independently for implementing the proposed algorithm.

The proposed algorithm is a hybrid algorithm that is a combination of serials and parallel processing. The algorithm is proposed by assigning larger computational tasks to GPU and assigning smaller ones to CPU. The procedure for each column's stage can be introduced as following steps:

1. The assigned blocks in the current column are updated with QR decomposition, in parallel (in GPU).
2. All other blocks (which should be affected) are updated in parallel.
3. The current column reduces by implementing general annihilation in serials (in CPU).
4. Other blocks should be updated because of the local and general annihilation in parallel.

For implementing this algorithm, we have used CUDA software which is based on parallel processing. CUDA is used for accelerating the reduction to upper hessenberg forms for solving eigenvalues problems.

The key contributions of this thesis to the field of hessenberg reduction of a general non-symmetric matrix are summarized as follows:

- Proposing a new hybrid algorithm which uses CPU and GPU for serials and parallel implementations, respectively.
- Applying the FFT techniques in the proposed algorithm to have both synchronization and shared memory, independently.
- Proposing a thread based algorithm in the GPU part of the general algorithm to have access to the maximum GPUs memory capacity.
- Applying blocks method to have independent structure from built-libraries (such as CULA) to achieve great efficiency (in both small and large matrix sizes) for solving eigenvalues problems.
- Looking for higher speedup ratio (with implementing the proposed algorithm) as comparison with others.
- Evaluating the proposed algorithm to support a higher matrix order range compared with others.

1.3 Thesis Outline

The remaining chapters of this thesis are structured as follows:

Chapter 2: Related Work. Briefly describe other works and what we need to know for new works. Present an overview of the related approaches to the algorithm proposed in this thesis.

Chapter 3: General Algorithm Procedure. Describe the methods which we use for the proposed new algorithm. Present the proposed algorithm. Implementation of the related algorithm in serials and the proposed algorithm in serials/parallel operations are presented.

Chapter 4: Experiment Results. Illustrate the proposed algorithm in details. Compare the proposed algorithm results with the related work results. Demonstrate the amount of improvement based on logical parameters.

Chapter 5: Discussion and Conclusion. Conclude the discussions, and propose future works to improve the approach of this thesis.

Chapter 2

Related Work

In the previous chapter, we mentioned thesis motivations and contributions and we briefly introduced our job. In this chapter, we concisely discuss related works and several approaches that we have used in our project.

2.1 Background Knowledge

2.1.1 Graphics Processing Unit (GPU)

A Programmable Graphic Processor Unit (GPU) can be described as highly parallel, multi-thread and multi-core processor with high computational power and very high memory bandwidth [6]. GPU is used in computers and gaming consoles. In recent years, GPUs are more programmable and therefore can be used for performing much more than graphics specific computations. The general purpose GPU (GPGPU) utilizes computational power of a GPU for performing computations in applications [6], [7]. This advantage is primarily due to the transistor allocation for the GPU vs. the CPU. Figure 2.1 and Figure 2.2 demonstrate how floating point operations per seconds and bandwidths are increased in recent years, respectively. It also shows that GPUs are much faster than CPUs. The majority of the transistors on the GPU are devoted to data processing rather than flow control and data caching. The specialized rendering hardware provides an advantage for the GPU over the CPU when performing compute-intensive, highly parallel computations. Figure 2.3 presents the structure of CPU and GPU. In both cases there is only one DRAM used. But GPU has

Figure 2.1: CPU vs. GPU comparison of floating point operations per second. [10]

several cache and control units. There is a specific cache and control unit for each bunch of ALUs that has been implemented in GPU.

GPGPU Programming Frameworks

High performance of GPUs for processing huge amount of data in a short time is one of the important features of this processor unit. For programming the GPU, software and some interfaces are requested to connect the hardware.

GPU Environments Earlier GPGPUs are based on low level languages such as OpenGL (Open Graphics Library) which is used for programming the devices. It is a multi-platform application programming interface (API) for programming 2D and 3D graphic applications. GPGPU programming on OpenGL requires a huge amount of knowledge about the hardware such as shaders and textures. Sh and Brook are known as two of the earliest high-level languages and programming environments for GPUs (BrookGPU). Brook is a programming

Figure 2.2: CPU vs. GPU bandwidth comparison. [10]

Figure 2.3: Transistor allocation for CPU and GPU. [10]

environment that presents GPU as a streaming co-processor [8]. These programming frameworks provide a level of abstraction from the graphics hardware. Therefore the programmer

does not require in-depth knowledge of GPU textures and shaders. BrookGPU comes from Stanford University graphics group. It is a compiler and runtime implementation of the Brook stream programming language for general purpose computations. It is implemented as an extension to the C programming language. BrookGPU can also be used in ATI Stream. “Sh” is a meta-programming language. This program is implemented as a C++ library. “Sh” has been commercialized and expanded with additional support for the cell processor and multi-core CPUs.

ATI Stream ATI Stream by AMD same as NVIDIA’s CUDA provides a high level interface for programming the stream processors on the GPUs. ATI Stream utilizes a high level language, ATI Brook+. It is a compiler and runtime package for GPGPU programming to provide control over GPU hardware. The Brook+ compiler and runtime layer handle the low-level details of program. Brook+ is built on top of ATI Compute Abstraction Layer (CAL). ATI Stream is cross-platform but only runs on AMD GPUs. ATI Stream and CUDA both have their positive and negative features. For the work of this thesis, NVIDIA’s CUDA is preferred over ATI Stream for its previous use at the institution.

OPENCL NVIDIA and AMD both support OpenCL (Open Computing Language). OpenCL is the first open standard for general-purpose parallel programming of heterogeneous systems. OpenCL supports not only GPU programming. It also supports a mix of multi-core CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs. OpenCL will provide a programming framework and environment most closely related to NVIDIA’s CUDA. During the time period of this work, OpenCL was still very new. It was not considered for implementation. [9] introduces OpenCL framework with four models which are the platform model, the execution model, the memory model and the programming model. A host connects to one or several OpenCL compatible devices in the platform model. The OpenCL execution model helps to define how kernels are executed. OpenCL memory model can be used to map for three levels GPU memory hierarchical structure. The data parallel programming model can be used to design OpenCL. However, it is believed

that the transition from CUDA to OpenCL would be a relatively straightforward process.

NVIDIA CUDA NVIDIA looked for an easy method to program GPUs. This company produced a new method which is called CUDA. CUDA, which has the general purpose parallel computing architecture, is one of the best choices to use for increasing performance in systems with ability to use both CPUs and GPUs. It is stated in [10] that “In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture - with a new parallel programming model and instruction set architecture - that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU ”. C programming language is a framework for CUDA, therefore CUDA applications can be easily implemented.

NVIDIA CUDA

As the majority of work required for this thesis is the implementation of a new algorithms using CUDA, some details regarding CUDA development must first be introduced. The concepts paraphrased below are covered with more details in [10].

CUDA Programming Model CUDA programmers write code for the GPU by creating C functions called kernels. Remember that only one kernel can be run on the device at a moment, and all configured threads execute the kernel in parallel. The threads are split into several thread blocks. Several blocks are also located into a grid. Figure 2.4 shows these organizations in details.

When a kernel runs, the blocks of the grid are sent to multiprocessors with available execution capacity. All threads of a block execute in parallel on a single multiprocessor. When all threads in each block complete their work, a new block is launched in its place. For managing the large amount of threads, the multiprocessor uses a single-instruction, multiple-thread (SIMT) architecture. This architecture allows each thread to execute independent of the other threads on one of scalar processors. Instructions are issued to groups of 32 threads called warps, which execute one common instruction at a time. If the instructions assigned

Figure 2.4: CUDA Grid layout. [10]

to threads within a warp differ due to conditional branching, the warp executes each path sequentially while disabling threads that are not on the path. When all branch paths are complete, the threads join back to the common execution path. This is the reason that code within conditional statements such as if/else should be limited. With the above information, it is better to know the following specifications for our GPU:

- The maximum number of active threads per block is 1024
- The maximum number of active threads per multiprocessor is 1536
- The number of multiprocessors in GPU is 2
- The maximum number of threads per block is 512
- The maximum number of active warps per multiprocessor is 32
- The maximum size of each dimension of a grid of thread blocks is 65535
- The maximum number of active blocks per multiprocessor is 8

There is an advantage in CUDA for synchronization. CUDA provides limited synchronization between threads of the same block via the “syncthreads” function call. It means that when an execution in one thread is finished, it will wait until all remaining executions in other threads are done. “Syncthreads” is primarily used to coordinate communication between the threads within a block. It prevents reading/writing data hazards with global or shared memory. There are some new functions that help synchronize threads between different blocks. But, still the best way to synchronize across thread blocks is by breaking the computation into multiple kernels. When all executions of one kernel finishes, next kernel begins to launch.

CUDA Program Flow Most CUDA applications follow a set of program flows, which are:

- The host first loads data from a source such as a text file and stores it into a data structure in host memory.
- The host allocates device memory for the data and copies the data to the allocated space.
- Kernels are launched to process the data and produce results.
- Results are copied back to the host for display or other processing.

Memory Hierarchy As presented in Figure 2.5, there are five main parts of memory on the device.

- **Registers:** Each multiprocessor has reading/writing access to a limited number of 32 bit hardware registers.
- **Constant Memory:** A read-only constant cache is shared by all scalar processor cores and speeds up reads from the constant memory while all threads of a half warp access the same location. There are 64 KBytes constant memory in total. Note that the cache working set for constant memory is 8 KBytes per multiprocessor.
- **Device Memory:** All threads have write/read access ability to the device DRAM.
- **Texture Memory:** A read-only texture cache is shared by all scalar processor cores and speeds up reads from the texture memory space. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve the best performance. The cache working set for texture memory varies between 6 and 8 KBytes for each multiprocessor.
- **Shared Memory:** All threads within a block have access to a common shared memory region. The amount of shared memory available per multiprocessor is limited to 16 KBytes with a small quantity reserved for built in variables.

CUDA has another memory called local memory. It has the same speed as device memory. This memory is also usable for storing local scope arrays and additional variables when there are insufficient registers available.

“Zero copy” is an important topic when discussing the memory. Zero copy allows threads to access host memory directly. When data is written to zero copy memory from the device, the data transfer is overlapped with kernel execution. For solving this problem, the host should synchronize explicitly with the device before trying to read any zero copy memory. Zero copy asks the device to map host memory. It can be checked by calling “cudaGetDeviceProperties” function and also checking the “canMapHostMemory” property.

Figure 2.5: CUDA memory heirarchy. [10]

Multiple GPUS CUDA is flexible to use multiple GPUs in a single application. The GPUs, with their own memory space and instructions, are completely independent of each other. Each GPU should be programmed and it should be set up separately. Note that a CPU's threads have responsibility to manage each GPU and the OpenMP API have responsibility to manage the host threads.

OPENMP As it is mentioned before, OpenMP is a shared memory multiprocessing API that was selected to manage the host level parallelism, which has the following proper-

ties:

- A C/C++ interface
- Multi-Platform
- Used by OpenCV1
- Portable and Scalable.

preprocessor directives is used for OpenMP to signify parallel blocks of code in CUDA. For this work, the typical OpenMP usage contains:

- Join threads
- Set number of OpenMP threads to be equal to the number of GPUs in the system using “ompsetnumthreads” function.
- Perform serial code block
- Execute serial code block
- Execute parallel code block via preprocessor directive “programparallel” function.

Compute Capability The compute capability of a device is introduced by a major and minor revision numbers. Devices with the same major revision numbers belong to the same core architecture. The minor revision number corresponds to an incremental improvement to the core architecture.

The algorithms of this work were developed to employ the features of new devices with “2.1 compute compatibility”. The most important advantages of new GPUs with compute capability of “1.3” over earlier devices include:

- It supports for atomic functions operating on 64-bit words in global memory.
- There are 16384 registers per multiprocessor vs. 8192.

- Double-precision floating point numbers are supported.
- It supports for atomic functions operating in shared memory.
- There is an enhanced memory controller with more relaxed memory coalescing rules.

Occupancy Occupancy can be introduced by the ratio of the number of active warps per multiprocessor to the maximum number of active warps. A higher occupancy results in the GPU hardware being more utilized. One of the greatest benefits related to high occupancy is the latency hiding during global memory loads. Increasing occupancy does not guarantee to achieve higher performance. As we mentioned before, each multiprocessor has limited registers and shared memory. These resources are shared between all thread blocks running on a multiprocessor.

Occupancy can be increased by decreasing the resources used by each thread of block, or by decreasing the number of threads used in each block. As shared memory is manually managed, local or global memory can be substituted instead. Register usage is more difficult to manage because registers are automatically used during memory transfers and calculations. There are two additional mechanisms for limiting register usage in CUDA which are described as:

- The “maxrregcount” compiler flag is used for specifying the maximum number of registers each kernel can use. If specified, the compiler uses local memory instead of the extra registers.
- The volatile keyword can also be used to limit register usage. Actually, the volatile keyword is a method for offering the compiler to assess the variable and drop it into a register, immediately. CUDA compiler may postpone it to be evaluated later instead of immediately evaluating a variable. Each time the variable is computed, the register count is increased and additional registers may be used.

2.1.2 Dense Linear Algebra

Dense Linear Algebra (DLA) can be chosen to design new architectures in field of computational science for several reasons. First of all, a wide range of science and engineering applications related to linear algebra and the applications of which may not execute properly without good performance of DLA libraries. Secondly, DLA has an understandable structure for software developers.

The Matrix Algebra on GPU and multi-core Architectures (MAGMA) project and the libraries in [11] are used to demonstrate the algorithmic techniques and their effect on systems efficiency. It is designed similar to LAPACK in data storage, functionality and interface. Developers can use MAGMA libraries to effortlessly port their LAPACK-relying software components and also to achieve benefit of each component of the new hybrid architectures.

Developing of high performance DLA algorithms in homogeneous multi-cores has been successful in some cases such as the one-sided factorizations [12]. To code Dense Linear Algebra in GPUs, several parameters should be considered such as choosing a language, programming model, developing new kernels, programmability, reliability, and user productivity. To code DLA in GPU, following methods are suggested:

- **CUDA and OpenCL:** As mention before, CUDA is the language for programming GPUs. It simplifies a data-based parallel programming model which is a remarkable fit for many applications. In addition, new results demonstrate its programming model allows applications to scale on many cores [10]. DLA is an algorithm that can be typified in terms of Level 2 and level 3 BLAS. Basically, it is a data parallel set of operations that are scaling on current GPUs. OpenCL structure is based on the data-based parallelism similar to CUDA. Both languages are going to support task-based parallelism. OpenCL is based on a programming model which has the potential of providing portability across heterogeneous platforms consisting of CPUs, GPUs, and other processors. These parameters make OpenCL a great candidate for coding hybrid

algorithms.

- GPU BLAS: DLA with acceptable performance needs to access of fast BLAS, particularly on the most compute intensive kernel, i.e., the Level 3 BLAS matrix-matrix multiplication. Performance of older generation GPUs are dependent to high bandwidth because they do not have memory. As a result, although some works are released in the field, the use of older GPUs has not led to significantly accelerated DLA algorithms. For example, K.Fatahalian et al. and Galoppo et al. studied SGEMM and LU factorization, respectively. They concluded that CPU implementations outperform most GPU implementations. But, the introduction of memory hierarchy in current GPUs has changed the situation completely. Now, with memory hierarchy, GPUs can be programmed for memory reuse and as a result not depend to their high bandwidth. Implementing fast BLAS is an outstanding key because algorithms for GPUs can have high priority in DLA developments.
- Hybrid Algorithms: New GPUs have ability of massive parallelism but they are based on serial kernel execution. At the same time, kernels are executed serially. It means that after execution of one kernel, the next kernel can be executed and only one kernel has permission to run at a moment. It is advised to developers to use a hybrid coding approach only for large data-parallel kernels on the GPU, which we decline this recommendation. New GPUs are going to support task based on parallelism. It is preferred that small task execute on the CPU with existing softwares such as LAPACK.

2.1.3 Two-sided Factorizations

The reductions to upper Hessenberg, tridiagonal, and bidiagonal forms [13], also known as two-sided matrix factorizations, are very important linear algebra problems for solving eigenvalue problems. As it is mentioned before, the Hessenberg reduction is the first step in computing the Schur decomposition of a non-symmetric square matrix. The operation count for the reduction of an $(n \times n)$ matrix is estimated around $(10/3) \cdot n^3$. Therefore, the reduction is a very desirable goal for acceleration. Note that solving a Hessenberg matrix form of a

system is very cheap compared to the corresponding algorithms for general matrices, which is making the factorization applicable in other areas as well [14]

The problem in accelerating the two-sided factorizations comes from the fact that they have many Level 2 BLAS operations. It can limit the system bandwidth and as a result it cannot scale on multi-core architecture. Dense linear algebra techniques can help to replace Level 2 BLAS operations with Level 3 BLAS, i.e., in LU, QR, and Cholesky factorizations. The application of consecutive Level 2 BLAS operations, which are occurred in the algorithms, can be postponed and accumulated at a later moment when the accumulated transformation should be applied. Then, a Level 3 BLAS is requested (LAPACK [15]). This act removes Level 2 BLAS from Cholesky, and also reduces its amount to $O(n^2)$ in LU and QR. The same technique can be used for HR [16]. Note that as comparison with the one-sided factorizations, it leaves about 20% of the total number of operations as Level 2 BLAS. Also note that 20% of Level 2 BLAS can approximately take 70% of the total execution time on a single core. The amount of Level 2 BLAS operations in the other two-sided factorizations is higher, i.e., 50% of the flops in both the bidiagonal and tridiagonal reductions are in Level 2 BLAS.

2.1.4 One-sided Factorizations

Now we want to describe the hybridization of LAPACKs one-sided factorizations on dense matrices. LAPACK uses a kind of block algorithm based on partitioning the matrix. This idea is used for hybrid algorithms. A dense linear system can be solved with two steps and one-side factorizations are the first step of it. It would shows the bulk of the computation and as a result has to be optimized. The second step includes triangular solvers or multiplication with orthogonal matrices. Consider that when developing algorithms for GPUs, some part of operations of factorization are faster on CPU rather than GPU, that is caused to the development of highly efficient, one-sided hybrid factorizations for a single CPU core and a GPU [17], [18], multiple GPUs [18], [19], and multi-core with GPU systems [20]. Hybrid

DGEMM and DTRSM for GPU-enhanced clusters were developed in [21]. They were used to accelerate the Linpack benchmark. For hybridization of LAPACKs one-sided factorizations three kind of factorization is recommended which are known as LU, QR, and Cholesky factorizations.

Cholesky Factorization: Matrix Algebra on GPU and multi-core Architectures uses the left-looking version of the Cholesky factorization. It has the feature of simplicity and similarity between the hybrid Cholesky factorization code and the LAPACK code.

QR Factorization: Static scheduling and a right looking version of the block QR factorization are used recently. The panel factorizations are scheduled on the CPU with calling LAPACK but the Level 3 BLAS updates on the trailing sub-matrices are assigned to implement on the GPU. The trailing matrix updates are divided into two parts. First part is to update the next panel and a second part is to update the rest. However, when the next panel update is done and sent to the CPU, the panel factorization on the CPU will be overlapped with the second part of the trailing matrix. This technique is called look-ahead technique, i.e., used in the Linpack benchmark.

LU Factorization: MAGMA also uses a right looking version of the LU factorization, similar to QR factorization. The scheduling is using the look-ahead technique similar to QR method. Interchanging rows of a matrix, which are stored in column major format, need to implement in the pivoting process and it is not efficient to execute on current GPUs. It is possible to use the LU factorization algorithm in [18] which can remove the above bottleneck. Using coalescent memory, which has access on the GPU, is recommended for row interchanges efficiently. The panels should be transposed before sending to the CPU for factorization.

2.2 Literature Survey

In the previous section we described one side and two side factorizations for doing DLAs on GPU. In this section, recent works, which lead to acceleration DLAs operation with hybrid methods, are presented. Recent articles which focus on hessenberg reduction acceleration using CUDA are reviewed.

[23, 26, 34] proposed new algorithms for reaching certain communication optimal bounds in DLA fields. [22, 24, 35] focus to develop algorithms which use blocking the data structures and also localizing matrix transformation in the field of one side matrix factorization. One of the recent algorithms [35], which use blocking the data structures and localizing matrix transformation, works on thread-level parallelism. In their algorithm, they divide data to submatrices (blocks) as units of data and algorithms as operating on these blocks and finally, schedules the operations on blocks using out-of-order techniques.

Blocking data method and localized matrix transformations are useful for two-sided matrix factorizations. It is used for the Householder transformation, which is described in [36], for annihilating matrix elements away from the diagonal of the matrix. This idea leads to two-sided factorizations to band matrix forms [27, 33]. [25] has presented two times performance improvement for tridiagonalization on multi-core architectures. The first stage is indicated in Level 3 BLAS in their algorithm but note that its execution did not scale by increasing the number of cores. Better performance is achieved by using GPU for the stage.

CUDA was used for accelerating the reduction to upper hessenberg forms and solving eigenvalues problems [2, 28]. They used BLAS library for matrix-matrix and vector-matrix implementation. In article [2], transforming procedure of general matrix to achieve eigenvectors is divided to four steps. The first step is defined as hessenberg reduction formula. Next step uses parameter of mentioned formula to reach orthogonal matrix and the third step uses schur transformation to achieve block matrix T which is a diagonal block matrix that

contains the same eigenvalues as the original matrix. The last step is calculating eigenvectors using parameters from previous steps. This article focuses on the first two steps to reduce to

Figure 2.6: Usage of the CUBLAS Library.[2]

hessenberg matrix and accelerating this method with CUDA and CULA. They used block algorithm for hessenberg reduction. In this kind of block method, which the original matrix is called “ A ” with size of n by n , they consider the block size as “ L ” and then split the method to two parts. First part is updating the column one through L and achieve some parameters. The second step update column “ $L+1$ ” through n using the parameters which is calculated in step one. This article claims that they accelerate both steps with the GPU program, CUDA. [2] also used CUBLAS to implements the algorithm, where CUBLAS is a BLAS library for GPU. The CUBLAS consists of routines to transfer data between the CPU memory and the GPU memory, and routines to perform basic linear algebra operations on data residing on the GPU memory.

In general, the algorithm is implemented based on the following procedures in [2]: Send the matrix A to GPU. At each step k , after updating the k th column of A , the updated column is returned to CPU, construct the Householder transformation, and send back requested parameters to GPU. All other computations are performed on GPU using CUBLAS. Finally, the reduced matrix is returned to CPU.

Then in two other implementations, the algorithm is improved by assigning larger computational tasks to GPU and assigning smaller ones to CPU. In our proposed work, we use different algorithm for hessenberg form reduction with implementing in CUDA to achieve better performance.

Figure 2.7: An overview of the CUDA programming model in [29]

[29] introduces an algorithm without using CUBLAS, which is faster rather than using CUBLAS for matrix-vector multiplications on CUDA. In the proposed algorithm of this thesis, we will not use NVIDIA's BLAS library.

Chapter 3

General Algorithm Procedure

Most recent and related works are represented in Chapter 2. In this Chapter, our proposed algorithm which has better performance compared to the related works is represented. Implementations of the related algorithms in serials and our algorithms in serials/parallel are also shown. Before presenting the proposed algorithm, basic formulas for solving eigenvalue problem are introduced.

3.1 Solve Eigenvalue Problem from Nonsymmetric Matrix

The standard procedure for solving the eigenvector problem “ $Ax = \lambda x$ ” is divided into four steps [2, 32] as listed below.

1. Reduce general matrix to hessenberg form:

One of the best ways to reach a hessenberg matrix form from nonsymmetric matrix is using QR decomposition. The following formula shows how the QR decomposition works:

$$W_{n-2}^T \cdots W_2^T \cdot W_1^T \cdot A \cdot W_1 \cdot W_2 \cdots W_{n-2} = H, \quad (3.1)$$

where W represents orthogonal matrix and “ n ” represents matrix order. There are several ways to convert a matrix to QR form. Two of the most popular methods are householder

transformation and given rotation transformation. A combination of these two methods were discussed in [4, 5]. We also use the same idea in this thesis.

2. The orthogonal matrix W can be calculated with

$$W_1 \cdot W_2 \cdots W_{n-2} = W. \quad (3.2)$$

3. Compute eigenvalues and eigenvectors with Schur decomposition

Based on [2], the Hessenberg matrix is transformed into a block upper triangular matrix with diagonal blocks of size at most 2. With Schur decomposition which is shown in (3.3) and (3.4) we can achieve another matrix which is called “Y”, that the eigenvalues of the diagonal blocks of “Y” are the same as the eigenvalues of the original matrix.

$$J_n^T \cdots J_2^T \cdot J_1^T \cdot H \cdot J_1 \cdot J_2 \cdots J_n = Y. \quad (3.3)$$

$$J_1 \cdot J_2 \cdots J_n = J. \quad (3.4)$$

4. Regarding to compute the eigenvectors, the eigenvectors k of “Y” are computed and they are transformed into the eigenvectors of A by

$$U = W \cdot J \cdot k. \quad (3.5)$$

The most popular method to implement these steps uses a software with LAPACK or CUBLA packages. LAPACK is a library which is written in Fortran. This library is produced for solving systems of linear equations, eigenvalue problems, and several other problems. LAPACK routines are built from the Basic Linear Algebra Subprograms (BLAS). BLAS is divided into three levels. The levels contain vector operations, “*matrix * vector*” operations and “*matrix * matrix*” operations make level 1, level 2 and level 3 BLAS operations, respectively. When we use these packages to solve eigenvalues of a matrix, all 3-levels BLAS are called. Based on [31], the level-2 BLAS performance is related to the memory throughput of the system. GPU was used in [2] instead of CPU, which has higher memory throughput to implement step 1, to reduce to hessenberg form.

Figure 3.1: Execution time of the four steps. [2]

The computational time of each step is shown in Figure 3.1 for a matrix with random order “ n ” in a system. It can be seen that steps 1, 3 and 4 occupy a large fraction of the computational time. In this case, step 1 requires less work than step 4. Step 4 is slow because the corresponding LAPACK routine is written without using level-2 and level-3 BLAS. Because of this reason, there is limitation to speed up the overall performance of the nonsymmetric eigensolver. However, there are some cases where only the eigenvalues are requested. In these cases, steps 2 and 4 are unnecessary and step 1 occupies most of the execution time. In our work, we implement step 1 without using build-in libraries by proposed algorithm and compared the speedup ratio and order number of general matrix with related works.

3.1.1 Block Annihilation Method

One of the best ways to speed up the procedure of solving eigenvalues is using parallel processing based on block method. In this method original matrix should be partitioned to several submatrices(blocks) and then each block will be assigned to a processor/thread to work in parallel. It is possible to split block method to two parts. First step is to convert the general matrix to hessenberg form and then convert the hessenburg form matrix to upper triangular matrix which contains eigenvalues. More calculation is requested for the first step which is our focus in this thesis.

There are two stages to reduce a general matrix to hessenberg form [4, 5]:

1. QR decomposition; QR decomposition method is used for changing some particular submatrices to triangular form, R , and updates some others with Q . householder transformation is used for QR decomposition.
2. Block Annihilation; the second stage is block annihilation which is implemented by given rotation method, which we call GZ decomposition.

Same as QR decomposition procedure, when each submatrices eliminate with Z , some others will be updated with G . In [4, 5], after dividing a general matrix to submatrices, the blocks under first low subdiagonal blocks should become zeros. Therefore blocks under diagonal blocks need to be implemented with two stages. Implementation begins from left column to right one. In each column the assigned blocks need to be updated with QR decomposition from up to down. Note that when each block is affected by QR decomposition, blocks in the same row and column with same row number should be updated. When the first stage is done, next stage should be implemented by two levels, local and general annihilation, with the procedure similar to stage one. Obviously, blocks will be affected in serials, not in parallel [4, 5].

3.2 Block Method Algorithm to Reach Hessenberg Matrix Form

A block method to achieve hessenberg matrix form is proposed in [4, 5]. This algorithm begins by splitting a general matrix of n by n , to several k by k sub-matrices (the minimum amount for k is two). As a result, there is a block matrix that each block, sub-matrix, consist of several elements. Note that depending on the number of processors, we allocate blocks to the processors. Now, from left to right, each column should be processed with the following stages:

1. Calculate Q and R for QR decomposition based on householder transformation for specific blocks, which is shown in Figure 3.2, one by one. Next, update all blocks after proceed block in the current row and then update all blocks in the column which has

the same row number. Householder transformation for each block in a column is not affected to other blocks in the same column. Therefore these blocks can be processed in parallel. The remaining blocks in further columns are updated with Q of transformed blocks. Some of them with transposed Q and some other with Q and the rest with both, as shown in Figure 3.2. The point is that the blocks can be updated in parallel, as well, without any synchronization. We just have to send right Q matrix to right block for multiplications.

Figure 3.2: Convert to Hessenberg Matrix Form

2. When above procedure is done for the last block of current column then a new procedure should be applied for affected current column blocks. The given rotation transformation for annihilating blocks, which are placed under first sub-diagonal blocks, is used. In [4], for this aim, two levels are requested. First level is local annihilation that works with blocks in the same processor in current column. For each operation two blocks are taken. Upper block is considered as pivoting block and lower block is annihilated with pivot block. Level two is global annihilation that works with blocks in a separate processor in current column.

Figure 3.3: Block Method Annihilation.[4]

Figure 3.3 shows two level annihilations and how to annihilate with two blocks. Actually, it is not practical to do this step in parallel because high synchronization between processors and high complexity for transferring data between processors are requested for large number of blocks annihilation. In addition, the shared memory between processors is requested for doing this part. Therefore, there is limitation to use memory because of using only shared memory.

In the proposed algorithm we implement this part in serials and we have a level annihilation. The blocks in current column are annihilated one by one from bottom to top, until first sub-diagonal block. In later sections, the number of blocks iteration are calculated and we can observe that this number for given rotation in a column is not very large if it is compared with the other parts.

3.3 Implementation procedure in Serials

Before using parallel processing with CUDA, the block method need to be implemented in serials and then analyze the main algorithm to check which parts can be processed in parallel. Algorithm 1 shows the procedures for block method in serials.

Algorithm 1 : General Block Method Algorithm In serials

```

1: Input Matrix
2: Convert it to  $(n \times n)$  Block Matrix
3: for (column 1 to column  $(n-1)$  ) do
4:   Do Householder Transformation
5:   Update Blocks in further columns
6:   Do Given Rotation Transformation
7:   Update Blocks in further columns
8: end for
9: exit:

```

Regarding the implementation of step 4 of Algorithm 1, as mentioned in the previous section, Householder transformation is used for performing QR decomposition. At the first time, Alston Scott introduced Householder transformation. This transformation method is completely proposed in [30]. We implemented this transformation based on [30]. There are two functions for transforming a block matrix to two matrices, Q and R . First function is $HHQR$, which uses Householder Reflections to factorize $F = Q \cdot R$. As a result, R is upper-triangular matrix and Q matrix has orthonormal columns, $Q' \cdot Q = I$. Note that this function works when F has no more columns than rows.

Algorithm 2 Householder Transformation (PART1)

Function $[F,R] = \text{HHQR}(T)$

```
[m,n] = size(F)
2: if ( $m < n$ ) then
    Error
4: goto exit
end if
6:  $z = \text{zeros}(1, n)$ 
    $w = \text{zeros}(m, 1)$ 
8: for ( $j = 1$  to  $j = n$ ) do
     $[w, z(j)] = \text{HHW}(F(j:m, j))$ 
10:  $F(j:m, j) = w$ 
    if ( $j < n$ ) then
12:  $F(j : m, j + 1 : n) = F(j : m, j + 1 : n) - w \cdot (w' \cdot F(j : m, j + 1 : n))$ 
    end if
14: end for
     $R = \text{diag}(z) + \text{triu}(F(1:n, 1:n), 1) ;$ 
16: for ( $j = n : -1 : 1$ ) do
     $w = F(j:m, j) ;$ 
18:  $F(:, j) = \text{zeros}(m, 1)$ 
     $F(j, j) = 1$ 
20:  $F(j : m, j : n) = F(j : m, j : n) - w \cdot (w' \cdot F(j : m, j : n))$ 
    end for
22: exit
```

The second function is HHW that is used inside of $HHQR$. HHW output is a parameter which is called w with $w' \cdot w = 2$ or 0 values. Therefore, $W = I - w \cdot w' = W' = W^{-1}$ reflects the given column x to $(W \cdot x = [z; 0; 0; \dots; 0])$ with $|z| = \text{norm}(x)$.

Algorithm 3 Householder Transformation (PART2)

Function HHW(F(j:m, j))

```
w = x(:)
m = length(w)
3: x1 = w(1)
   a1 = |x1|
   if (m < 2) then
6:   w = 0
     z = x1
     goto exit
9:   end if
     if (a1) then
       s = x1/a1
12:  else
      s = 1
     end if
15:  vv = w(2 : m)' · w(2 : m)
     ax =  $\sqrt{(a1 \cdot a1 + vv)}$ 
     z = (-s · ax)
18:  a1 = (a1 + ax)
     w(1) = (s · a1)
     dd2 = (a1 · ax)
21:  if (dd2) then
     w =  $\frac{w}{\sqrt{(dd2)}}$ 
     end if
24:  exit
```

When Q and R are taken from $HHQR$ for each block in the current column, some specific blocks in the current column are replaced with R_i , that i represents the block number in the current column. As it is shown in Figure 3.2, we use the following Algorithm 4 for updating the blocks in further columns:

Algorithm 4 : Updating blocks algorithm after QR transformation of a column in serials

```
for (i = d : n) do
   A(i): = A(i): · Q(i)
   A:(i) = Q'(i) · A:(i)
4: end for
exit:
```

Note that in Algorithm 4, d notifies with the following stages:

- Stage 1, $d = 2$
- Stage 2, $d = 3$
- ...
- Stage (n-1), $d = n$.

In the above algorithm in “for” loop row’s range is $\{1 \text{ to } n\}$ and column’s range is $\{(i+1) \text{ to } n\}$.

Given Rotation transformation can also be used for QR decomposition. We use this transformation after householder transformation for annihilating some blocks, which will be presented in later sections. General idea of a Given Rotation is a rotation of a point or points around another point. Given Rotation transformation matrix can be presented as:

$$G(s, c, \theta) = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & -s & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \quad (3.6)$$

where $s = \sin\theta$, $c = \cos\theta$ and θ is the degree of rotation. Formula 3.7 shows how an element of a vector can annihilate with given rotation transformation matrix. Two functions are considered for implementing this transformation. The function of Algorithm 5 generates a matrix which contains two matrices, where one of them is considered as pivoting matrix and another one is considered as a matrix that should be annihilated. The duty of this function is sending the specific elements to *GivRot* function until the requested matrices produce and send to the main algorithm.

Algorithm 5 GivenRotation Transformation (PART1)

Function HesRot(A)

```
    GG1 =1;
    input = A;
    [m,n]= size(A);
    for ( $d = 1 : n$ ) do
5:   sign1 = 0 and sign2 = 0 and sign3 = 1 and mequn = 0
    end for
    if ( $m = n \ \& \ d = n$ ) then
        mequn = 1;
    end if
10: while ( $sign3 = 1 \ \& \ mequn = 0$ ) do
    e = m;
    while  $e \geq 1$  do
        if  $A(e, d) < 0.0000000001 \ \& \ A(e, d) > -0.0000000001$  then
            A(e,d)= 0;
15:        end if
        if  $A(e, d) \neq 0 \ \& \ sign1 = 0 \ \& \ (e > d)$  then
            B = A(e,d);
            b1 = e;
            sign1 = 1;
20:        else
            if ( $A(e, d) \neq 0 \ \& \ sign1 = 1 \ \& \ e \geq d \ \& \ e > (d - 1)$ ) then
                C = A(e,d);
                c1 = e;
                sign2 = 1;
25:            end if
            end if
            if  $sign2 = 1$  then
                [GG] = GivRot(B,C,b1,c1,m)
                GG1 = (GG * GG1)
30:             $sign1 = 0 \ \& \ sign2 = 0 \ \& \ b1 = 0 \ \& \ c1 = 0$ 
                A = (GG · A)
                e = e+1
            end if
            if  $A(d + 1 : m, d) = 0$  then
35:                sign3 = 0;
            end if
            e = e-1;
        end while
    end while
40: G = GG1;
    TRi = (GG1) · input;
    exit
```

Second function, Algorithm 6, which is given as:

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \quad (3.7)$$

is implemented to annihilate specific elements. In (3.7), from left to right, matrices are given as rotation matrix, initial vector and transformed vector, respectively. Note that Algorithm 6 is used inside of Algorithm 5.

Algorithm 6 GivenRotation Transformation (PART2)

Function GivRot(BB,CC,bb1,cc1,mm)

```

    r =  $\sqrt{BB^2 + CC^2}$ 
    cos1 = ( $\frac{CC}{r}$ );
    sin1 = ( $-\frac{BB}{r}$ );
    G1 = size(mm,mm);
    for j = 1 : mm do
6:   for i = 1 : mm do
        if ((i = cc1 & j = cc1) || (i = bb1 & j = bb1)) then
            G1(i,j) = cos1;
        else
            if (((i  $\neq$  cc1 & j  $\neq$  cc1) || (i  $\neq$  bb1 & j  $\neq$  bb1)) & (i = j)) then
                G1(i,j) = 1
12:           end if
        else
            if (i = bb1 & j = cc1) then
                G1(i,j) = sin1
            end if
        else
18:         if (i = cc1 & j = bb1) then
                G1(i,j) = -sin1;
            end if
        else
            G1(i,j) = 0
        end if
24:   end for
    end for
    exit

```

After finishing the given rotation for each couple of blocks in the current column and replacing them with TR_i (i represents the block number in current column). The remaining blocks in further columns should be updated based on step 7 of the Algorithm 1. Update

procedure for the remaining blocks is similar to householder transformation but there is one difference that every annihilation operation affects on two blocks. Therefore two rows and columns blocks are revised in the update procedure that should be mentioned in serials updating procedure. In serials processing, as mentioned before, we change the block method algorithm in [4]. We eliminate the global annihilation and also the blocks in the current column are annihilated from bottom to top until first lower sub-diagonal block. Figure 3.4 shows the new annihilation method.

Figure 3.4: Schematic diagram of annihilation for HH

For updating further columns, we propose Algorithm (7).

Algorithm 7 : Updating blocks algorithm after GZ transformation of a column in serials

```

for ( $i = n : -1 : k$ ) do
   $A_{(i-1):} = G_{(1)}(i) \cdot A_{(i-1:)} + G_{(2)}(i) \cdot A_{(i:)}$ 
   $A_{(i):} = G_{(3)}(i) \cdot A_{(i-1:)} + G_{(4)}(i) \cdot A_{(i:)}$ 
   $A_{:(i-1)} = A_{:(i-1)} \cdot G'_{(1)}(i) + A_{:(i)} \cdot G'_{(2)}(i)$ 
   $A_{:(i)} = A_{:(i-1)} \cdot G'_{(3)}(i) + A_{:(i)} \cdot G'_{(4)}(i)$ 
end for
7: exit:

```

Note that in Algorithm 4, k clarifies with the following stages:

- Stage 1, $k = 3$
- Stage 2, $k = 4$
- ...
- Stage ($n-2$), $k = n$.

Also consider that in the above algorithm in the “*for*” loop, the range of the row is $\{ 1 \text{ to } n \}$ and that for column is $\{ (i+1) \text{ to } n \}$.

3.4 Implementation procedure in Serials/Parallel

The proposed algorithm uses the idea of FFT algorithm that is compatible with CUDA software. In each stage of FFT algorithm, the variables (blocks) in the current stage do not affect each other and are independent. Each variable is updated only based on previous stage variables. It means that a variable in the current column is a function of the previous stage variables. Therefore, this algorithm can work asynchronously.

After blocking the general matrix and making n columns blocks, totally, $(n-1)$ columns will be processed and n columns will be updated. There are $2(n-1)-1$ stages, which are considered for updating the general matrix in CUDA. In this procedure, $(n-1)$ and $(n-2)$ stages are used

for updating blocks related to householder and given rotation transformations. In CUDA, main routine should be written in the HOST. Some subroutines for parallel processing should be applied in DEVICE. The proposed general algorithm with combination of serials and parallel processing is shown below:

Algorithm 8 : General Block Method Algorithm In Serials/Parallel

```

Input Matrix.
Convert it to  $(n \times n)$  Block Matrix.
for (column 1 to column  $(n-1)$  ) do
    Do Householder Transformation;
    Send General Matrix To GPU;
    Update Blocks in further columns in Parallel;
    Send General Matrix To CPU;
8:   Do Given Rotation Transformation;
    Send General Matrix To GPU;
    Update Blocks in further columns in parallel;
    Send General Matrix To CPU;
end for
exit:

```

In comparing Algorithm 8 with Algorithm 1, four steps are added for sending and receiving data between GPU and CPU. Two steps are also changed that they are related to Updating blocks for both QR and GZ transformation. Other steps stay the same and are processed in serials.

More details of Step 6 and Step 10 for parallel processing are presented below.

Step 6:

Updating a block in the “updating blocks based on householder transformation” stage (UHH) is shown in Figure 3.2 and Algorithm 4. There are two steps to update the block. As a result, in this stage, the block matrix with updating matrices, Q and Q' , are sent to a thread. Thread can give the updated block to output individually. All blocks in UHH stage are updated with this method in parallel because each thread can work by itself and there is no need in sharing data between threads.

Step 10:

Updating a block in the “updating blocks based on given rotation transformation” stage (*UGZ*) owns higher complexity, which is a consequence of the following factors.

- The input and output of given rotation transformation in Algorithm 5 is a matrix which contains two, pivoting and annihilating, matrices.
- Figure 3.4 shows that each block matrix is affected twice, once for pivoting and once for annihilating.
- Algorithm 7 have two more steps comparing with Algorithm 4.

If these factors are considered for updating a block in a thread for *UGZ* stage, Figure 3.5 can be shown as the maximum implementation in a thread. As it is shown in Figure

Figure 3.5: Maximum implementation procedure for each thread in *UGZ* stage

3.5, at most $2^4 = 16$ blocks(sub-matrices) should be called to update a block in stage

UGZ .

In this figure, four steps of Algorithm 7 are shown with black and white. X_{A1} is considered as a block which should be updated in Figure 3.5. To achieve matrix X_{A1} , we have to begin the procedure from Step 4 to Step 1. In each step, two matrices update a matrix with G or G' matrices. Note that G or G' are calculated in Algorithm 5 and the size of them are $2L \times 2L$, where L is block matrix size.

The most important feature of using this method is that threads in the DEVICE only communicate with global memory. Therefore, there is no need to use shared memory to transfer data between each other and to communicate with other threads. As a result, there is much more DATA memory available to generate a larger matrix.

Figure 3.6 shows the structure of our proposed program. It declares how CPU and GPU are connected together with CUDA software. For each column except the last one, the following procedure is requested:

- Householder transformation for each column in serials.
- Update the rest blocks.
 1. Send all Q_{ij} s and all blocks as a matrix to global memory in the DEVICE.
 2. Global memory sends the required sub-matrices to each thread.
 3. Each thread do its own job, individually.
 4. Each thread put its own updated matrix on global memory.
 5. Receive operated blocks as a matrix from global memory.
- Given Rotation transformation for each two nonzero blocks consequently in serials until the main sub-diagonal block be triangular and the rest blocks be zeroes.

- Update the rest blocks
 1. Send all G_{ij} s and all blocks as a matrix to global memory in the DEVICE.
 2. Global memory sends the required sub-matrices to each thread.
 3. Each thread do its own job, individually.
 4. Each thread put its own updated matrix on global memory.
 5. Receive operated blocks as a matrix from global memory.

Figure 3.6: Schematic diagram of implementation in CUDA

As a conclusion, we summarize the procedure in CUDA for column $\{1 \text{ to } n-2\}$ with following steps:

1. Grids, blocks and threads sizes are defined in the Host.
2. Householder transformation for column in Host.
3. Data is sent to the device and then device is called with kernel function name.
4. In Device, Each matrix block is assigned to a thread and each thread receives their initial DATA from global memory and update itself with assigned function. Note that thread function output is the updated matrix block and is send to global memory.
5. At last, data is copied from DEVICE to HOST.
6. Given Rotation transformation for column in Host.
7. Repeat Step 3,4 and 5 for UGZ

Note that only Step 2 to Step 5 are applied for column $(n-1)$.

Figure 3.7: Schematic diagram of Hybrid algorithm in CUDA

The hybrid algorithm represents in a more detailed way in Figure 3.7. In the CPU partition, each block represent one of the transform functions, such as HH and GZ . In the GPU partitions, only one of the updating function, i.e. UHH and UGZ , is launched at a time.

Chapter 4

Experiment Results

In the previous chapter, the proposed algorithm in serials/parallel is presented. In this chapter, we explain our algorithm in details and also we compare our results with the related works and this will demonstrate the amount of improvement we gain based on logical parameters.

The proposed Algorithm (8) is implemented in Microsoft Visual Studio 2010. The system uses Intel(R) Core (TM) i5 CPU@1.7GHz and GeForce GT 635M GPU with Compute Capability 2.1 and 2 GB RAM. Visual Studio Debugger is used for Debugging in HOST which contains the Main routine. NVIDA Nsight is installed on Visual Studio for debugging the Device part of codes. It also helps us to track our program and check all grids, blocks and threads one by one.

4.1 Processing Time

In this section, the processing time will be calculated in detail. Execution time for all parts of algorithm will be identified step by step.

Table 4.1 shows the number of affected blocks in separate functions during each matrix column operation. For example, when householder transformation is done for column 2, $n \cdot (n - 2)$ blocks in further columns should be updated. As Table 4.1 presents, the number of

Table 4.1: Execution time (in seconds) of Hessenberg reduction.

Operation/column number	Col 1	Col 2	...	Col (n-2)	Col (n-1)
<i>Householder</i>	(n-1)	(n-2)	...	2	1
<i>HouseholderUpdating</i>	n·(n-1)	n·(n-2)	...	n·2	n·1
<i>GivenRotation</i>	2·(n-2)	2·(n-3)	...	2·1	–
<i>GivenRotationUpdating</i>	2n·(n-2)	2n·(n-3)	...	2n·1	–

total blocks iterations for updating the householder and given rotation parts are significantly large. If we can do these parts in parallel, significant amount of time can be saved at total execution time.

Let us define

$$h_1 = \sum_{x=1}^{n-1} x \quad (4.1)$$

$$h_2 = n \cdot \sum_{x=1}^{n-1} x \quad (4.2)$$

$$h_3 = 2 \cdot \sum_{x=1}^{n-2} x \quad (4.3)$$

$$h_4 = 2n \cdot \sum_{x=1}^{n-2} x \quad (4.4)$$

to represent the total number of blocks for HH , total number of blocks for UHH , total number of blocks for GZ and total UGZ blocks number, respectively.

4.2 CPU Processing Time vs CPU/GPU Processing Time

Figure 4.1 and Figure 4.2 identify the total operation time needed for reaching the hessenberg form. Figure 4.1 shows the total execution time for all kind of operations in a column based on block method algorithm of [4, 5]. Figure 4.2 displays the total execution time for all kind of operations in a column based on the proposed algorithm with parallel processing. The total operation time for HH is assumed as T and the line's slope (operations vs time)

Figure 4.1: Block algorithms based on CPU.

is represented with “ m ”. All other operations are calculated based on these two parameters. The total execution time is reduced from “ $(3n+3)T$ ” in Figure 4.1 to “ $3T$ ” in Figure 4.2 based on ideal mode. Ideal mode means the size of matrix is large enough where the transmission time between CPU and GPU can be ignored. Note that if the size of matrix and blocks are large enough the householder transformation part also can be done in parallel.

The execution time for our systems in different operations are measured and represented as:

- DATA transfer between CPU and GPU for each stage is 30-50 micro seconds.
- Maximum execution time for updating blocks after each householder transformed block is 0.2-0.4 micro seconds.
- Maximum execution time for updating blocks after each given rotation transformed

Figure 4.2: new block algorithms based on CPU/GPU.

block is 2-4 micro seconds.

- Maximum execution time for transforming a block based on householder is 1-1.4 micro seconds.
- Maximum execution time for transforming a block based on given rotation is 1.5-2.5 micro seconds.

Above calculated time is based on (3×3) block's matrix size. Now with the above time and also h_1 , h_2 , h_3 and h_4 definition, we can calculate the total execution time for different size of original matrix. For example, as we mentioned before, the execution time for HH of a block is between 1-1.4 microsecond. Therefore, the average time for HH of a block is 1.2 microsecond. This number multiplied with h_1 gives us the average total time for this stage. UHH total execution time, UGZ total execution time and GZ total execution time of general matrix are all calculated based on this concept. The average execution time for

UHH , 0.3 microsecond, multiplied with h_2 gives the total average time for this stage. The average execution time for GZ , 2 micro seconds, multiplied with h_3 gives the total average time for GZ stage. Also, the average execution time for UGZ is 3 micro seconds. This number multiplied with h_4 gives the total average time for UGZ stage. All these procedures for different “ n ” are calculated in Table (4.2). There are two columns in this table, which are called “ HH transmit” and “ GZ transmit”. “ HH transmit” is the transmission time between CPU and GPU for UHH . Based on previous Chapter, the matrix should be sent to GPU ($n-1$) times; updated matrix is returned to the CPU ($n-1$) times, as well. Therefore, in total, the average time for “ HH transmit” is $2 \cdot (n - 1) \cdot 40$ microseconds. The same procedure for “ GZ transmit” is requested, which is the transmission time between CPU and GPU for UGZ , but the matrix should be transferred between CPU and GPU $2 \cdot (n - 2)$ times instead of $2 \cdot (n - 1)$ times. As a result, the average time for “ GZ transmit” is $2 \cdot (n - 2) \cdot 40$ microseconds.

As we mentioned in the previous section, our algorithm for updating blocks is thread based and maximum number of threads per multiprocessors should be considered. The maximum number of threads multiplications per processor is 1536 and the number of processors is 2. Therefore, if the number of blocks, which want to process as same time, is greater than 3072 we have to consider this parameter in the execution time tables.

4.3 Speedup Ratio

It is not possible to compare execution time of our algorithm with others because the systems are different and the algorithms are not the same as each other. The best way to compare them is using ratios such as speedup ratio that we understand the amount of improvement achieved.

In Table (4.2), the total execution time based on serials and serials/parallel algorithms are represented for different value of “ n ”. In this table the average time is considered for the execution time of each stage.

		HH	UHH	HH Transmit	GZ	UGZ	GZ Transmit	Total Time
4	CPU	7.2	7.2	---	12	72	---	98.4
4	CPU-GPU	7.2	0.9	240	12	6	160	426.1
16	CPU	144	576	---	420	10080	---	11220
16	CPU-GPU	144	4.5	1200	420	42	1120	2930.5
32	CPU	595.2	4761.6	---	1860	89280	---	96496.8
32	CPU-GPU	595.2	9.3	2480	1860	90	2400	7434.5
64	CPU	2419.2	38707.2	---	7812	749952	---	798890.4
64	CPU-GPU	2419.2	18.9	5040	7812	186	4960	20436.1
128	CPU	9753.6	312115.2	---	32004	6144768	---	6498641
128	CPU-GPU	9753.6	114.3	10160	32004	1134	10080	63245.9
256	CPU	39168	2506752	---	129540	49743360	---	52418820
256	CPU-GPU	39168	841.5	20400	129540	8382	20320	218651.5
512	CPU	156979.2	20093337.6	---	521220	4E+08	---	4.21E+08
512	CPU-GPU	156979.2	6591.9	40880	521220	65790	40800	832261.1
1024	CPU	628531.2	160903987.2	---	2091012	3.21E+09	---	3.38E+09
1024	CPU-GPU	628531.2	52479.9	81840	2091012	524286	81760	3459909

Table 4.2: Execution time (in micro-second) for serials and serials/parallel implementation in different stages with different size of n

Table 4.3: Execution time (in seconds) of Hessenberg reduction.

n	4	16	32	64	128	256
<i>CPU</i>	0.0000984	0.01122	0.0964968	0.7988904	6.4986408	52.41882
<i>CPU – GPU</i>	0.000426	0.0029305	0.0074345	0.0204361	0.0632459	0.2186515
<i>SpeedupRatio</i>	0.231	3.82	12.98	39.09	102.82	239.74

Table (4.3) summarizes Table (4.2) for range of $\{n = 4 \text{ to } n = 256\}$. The total execution based on serial algorithm and serial/parallel algorithm is compared for different value of “ n ”. The speedup ratio represents the ratio of execution time in CPU over execution time in CPU/GPU. Speedup ratio shows the algorithm does not work for very small size of block matrices such as (4x4) matrix. But it works from (16x16) matrix and when the matrix size grows, the speed up number grows as well.

Figure 4.3 shows a better understanding of execution time comparison in Table (4.3). The CPU line represents the execution time based on serials Algorithm (1) that only used CPU,

Figure 4.3: Execution time (in seconds) of Hessenberg reduction.

and the CPU-GPU line shows the execution time in the proposed algorithm (8), that is based on CPU/GPU. The CPU line increases exponentially in compare to CPU-GPU line, which increases smoothly.

4.4 Comparison

In combination of (3.1) and (3.3), which is mentioned in the Chapter 3, eigenvalues for a general matrix with CUDA can be determined. This goal requested in [28] using QR method as the future work and in this thesis we have reached to first part of this procedure with outstanding results.

Comparing our results with others can be done in two ways. First, speedup ratio between different algorithms is compared. Figure 4.4 shows the speedup ratio for LAPACK,

Figure 4.4: Speedup ratio based on matrix order (n).

scalPACK, SMPS BHR, CULA, CUBLA and our proposed algorithm for reaching to hesenberg form. SMP Super Scalar (SMPS) is a parallel programming framework which is represented in [5]. LAPACK, scalPACK and SMPS BHR speedup ratio results come from [5]. These ratios are elapsed time of each algorithm over elapsed time of householder block reduction. Note that these results are approximate values. In Figure 4.4, the acceptable speedup ratios in closed matrix order with others are shown. It is shown that the proposed algorithm has the highest ratio compared with others. Note that the best performances of others are compared to the performance of the proposed algorithm with block size 3.

The block size can effect on the performance of the algorithm. When the block size is small, the process for each block is less. Also, more blocks process in parallel. On the other hand, as mentioned before, the maximum number of active threads should be considered for parallel threads processing because the process of each block is belong to one thread. A balance between these parameters is requested to achieve best performance for different matrix size. We don't need to look for the best performance of proposed algorithm based on different block size because our results, with this random block size (3), are much better

than others results.

Figure 4.5: compare algorithm efficiency based on matrix order(n).

In the following, the matrix orders among the referred algorithms are compared. Figure 4.5 shows minimum matrix order (n) of each algorithm which has speedup ratios bigger than 1. Because when the speedup ratios are less than one, it is useless to use these algorithms instead of householder block reduction in serials. The proposed algorithm can be used for matrix order bigger than 16. It clarifies that the proposed hybrid algorithm can be used in smaller size of matrices where previous algorithms were unable to give an acceptable result.

These significant results are logical and reasonable since we code each matrix block independently in our algorithm. Although, these items are the advantages of our algorithms, there are some disadvantages as well. In large size of matrices coding each block matrix will be complicated. For future works, the suggestion is to innovate a general formula for blocks of matrix. Other papers didn't use such a method because of the following reasons:

- It is hard to formulize the blocks independently without any connection between them and using shared memory, which we did as a combination of Block method and FFT algorithms.

- Based on SIMT architecture, using IF/Else functions are not allowed in the threads of each warp. If an algorithm is written with IF/Else functions inside of the threads, only one thread in each warp should be used. Because of the above restriction, there are no IF/Else functions in our algorithm and we can use maximum number of threads, which can be implemented in parallel, at a time.

We strongly believe that our proposed algorithm works better since there is no memory sharing needed between the blocks and very significant results can be achieved as it is mentioned before in this chapter.

Chapter 5

Discussion and Conclusion

In this thesis, we proposed an approach for accelerating Hessenberg reduction and increasing its matrix order range in a hybrid computing environment. Throughout this process, we innovated a hybrid algorithm which is based on a modified blocks method. In consequence, approached blocks method is stand on FFT algorithm while using both householder and given rotation transformations. The hybrid algorithm is introduced by a combination of serials processing (in CPU) and parallel processing (in GPU). The GPU part of our current proposed algorithm is thread based and functions asynchronously. Each GPU algorithm matrix block is coded independently for implementation. For calculating processing time of the proposed algorithm, we have introduced certain formulas to figure out the total number of each operation in hybrid algorithm, which is designed to be compatible with CUDA platform. The first approach on this platform with an Intel Core i5 processor and an NVIDIA GeForce GT 635M, is achieved by 239.74 times speedup over the CPU-only case. The second approach was the matrix order range, increased by supporting small size matrix ($n = 16$). Both planned targets were reached through this effort.

5.1 Future Work

As the next step, we plan to continue working on optimizing the proposed algorithm to support nonsymmetric complex matrix. Also by analyzing the GPU side of the proposed method to innovate a general formula, we will enhance the usability of this method. More-

over, in next processing part we will expand our algorithm to accelerate other steps to achieve eigenvectors.

Bibliography

- [1] Carl D.Meyer, “Matrix Analysis and Applied Linear Algebra” *Matrices, Algebras, Linear*, Issue: 2000.
- [2] J. Muramatsu, K. Kimura,Y.Yamamoto, “Acceleration of Hessenberg Reduction for Nonsymmetric Eigenvalue Problems in a Hybrid CPU-GPU Computing Environment” *International Journal of Networking and Computing*, Volume: 1, Number 2, Issue: July 2011, Page(s): 132-143.
- [3] S.Ohshima, K.Kise, T.Katagiri, T.Yuba, “Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment” *Computer Communication Networks*, Volume: 4395, Issue: 2007, Page(s): 305 - 318.
- [4] M.W. Berry, J. J. Dongarra,Y. Kim, “A parallel algorithm for the reduction of a non-symmetric matrix to block upper-Hessenberg form” , *Parallel Computing*, Volume: 21, Issue: 4, January 1995, Page(s): 1189-1211.
- [5] H.Ltaief, J.Kurzak, J.Dongarra, “Parallel Two-Stage Hessenberg Reduction using Tile Algorithms for Multicore Architectures” <http://www.netlib.org/lapack/lawnspdf/lawn208.pdf>
- [6] J. Sanders, E. Kandrot, “ CUDA by example: an introduction to general-purpose GPU programming” *Application software*, Issue: 2011,
- [7] “ General-purpose computing on graphics processing units” <http://en.wikipedia.org/wiki/GPGPU>,

- [8] I. Buck, T. Foley, D. Horn, J. Sugerman, “Brook for GPUs: Stream Computing on Graphics Hardware” *Computer Science Department Stanford University, SIGGRAPH 2004*, 2004 ACM 0730-0301/04/0800-0777
- [9] K. Zhang, G. Chen, “GPU accelerate parallel Odd-Even merge sort: An OpenCL method ” *Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design*, 2011 IEEE, Page(s): 76 - 83.
- [10] “ NVIDIA. Cuda Program Guide. [Online].”
[http : //docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf](http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf),
- [11] S. Tomov, R. Nath, P. Du, and J. Dongarra. “MAGMA version 0.2 User Guide.”
<http://icl.cs.utk.edu/magma>, 11/2009.
- [12] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. “ A class of parallel tiled linear algebra algorithms for multicore architectures.” *Parallel Computing*, 35(1):3853, 2009.
- [13] G.H. Golub, C.F. Van Loan, “Matrix Computations” , second ed., Baltimore, MD, USA, 1989.
- [14] C.F. Van Loan, “Using the Hessenberg Decomposition in Control Theory” , North-Holland, Amsterdam, 1982.
- [15] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, “ LAPACK Users Guide” , third ed., SIAM, 1999.
- [16] S. Hammarling, D. Sorensen, J. Dongarra, “ Block reduction of matrices to condensed forms for eigenvalue computations ” , J. Comput. Appl. Math. 27 (1987) Page(s): 215227.
- [17] M. Baboulin, J. Dongarra, and S. Tomov. “ Some issues in dense linear algebra for multicore and special purpose architectures.” Lapack working note 200, May 2008.

- [18] V. Volkov and J. Demmel. “Benchmarking gpus to tune dense linear algebra.” In SC 08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Piscataway, NJ, USA, 2008. IEEE Press. Page(s): 111.
- [19] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. “A scalable high performant Cholesky factorization for multicore with GPU accelerators” *Lawn* 223, November 2009.
- [20] S. Tomov, J. Dongarra, and M. Baboulin. “Towards dense linear algebra for hybrid GPU accelerated manycore systems.” *Lawn* 210, October 2008.
- [21] M. Fatica. “Accelerating Linpack with CUDA on heterogenous clusters.” In GPGPU-2, New York, NY, USA, Page(s): 4651.
- [22] M. Baboulin, J. Dongarra, S. Tomov, “Some issues in dense linear algebra for multicore and special purpose architectures,” *Proceedings of the International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, Trondheim, Norway, 2008.
- [23] G. Ballard, J. Demmel, O. Holtz, O. Schwartz, “Minimizing Communication in Linear Algebra,” *Tech. Report, LAPACK Working Note 218*, May 2009.
- [24] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, “A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures,” *Technical Report UT-CS-07-600, LAPACK Working Note 191*, University of Tennessee, 2007.
- [25] P. Bientinesi, F. Igual, D. Kressner, E. Quintana-Orti, “Reduction to Condensed Forms for Symmetric Eigenvalue Problems on Multi-core Architectures,” *Aachen Institute for Computational Engineering Science, RWTH Aachen, AICES-2009-11*, March 2009.
- [26] James W. Demmel, Laura Grigori, Mark Frederick Hoemmen, Julien Langou, “Communication-Optimal Parallel and Sequential QR and LU Factorizations,” *Tech. Report, LAPACK Working Note 204*, August 2008.

- [27] H. Ltaief, J. Kurzak, J. Dongarra, “Parallel Block Hessenberg Reduction using Algorithms-By-Tiles for Multicore Architectures Revisited,” Tech. Report, LAPACK Working Note 208, August 2009.
- [28] T.Wang, L.Guo, G.Li, J.Li, R.Wang, M.Ren, J.He, “Implementing the Jacobi Algorithm for Solving Eigenvalues of Symmetric Matrices with CUDA” *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, Issue: 2012, Page(s): 69 - 78.
- [29] N.FUJIMOTO, “DENSE MATRIX-VECTOR MULTIPLICATION ON THE CUDA ARCHITECTURE” *Parallel Processing Letters*, Volume: 18, Issue: 4, 2008, Page(s): 511.
- [30] Householder, Alston, “Unitary Triangularization of a Nonsymmetric Matrix” *Journal of the ACM (JACM)*. 10/1958, Volume: 10, Issue: 4, Page(s): 339 - 342.
- [31] James W. Demmel, “Applied Numerical Linear Algebra” *SIAM*, Aug 1, 1997,
- [32] G. H. Golub and C. F. van Loan, “Matrix Computations.” *Johns Hopkins University Press*,, third edition, 1996,
- [33] H. Ltaief, J. Kurzak, J. Dongarra, “Parallel Band Two-Sided Matrix Bidiagonalization for Multicore Architectures,” Tech. Report, LAPACK Working Note 209, October 2009.
- [34] G. Ballard, J. Demmel, O. Holtz, O. Schwartz, “Communication-Optimal Parallel and Sequential Cholesky Decomposition,” Tech. Report, LAPACK Working Note 215, February 2009.
- [35] G. Quintana-Ort, E.S. Quintana-Ort, R. van de Geijn, F.G. Van Zee, E. Chan, “Programming matrix algorithms-by-blocks for thread-level parallelism,” *ACM Trans. Math. Softw.* 36 (3) (2009) 126.
- [36] G.H. Golub, C.F. Van Loan, “Matrix Computations,” second ed., Baltimore, MD, USA, 1989.

Appendix A

Abbreviation List

HH	Householder Transformation
UHH	Update block with Householder transformation
GZ	Given Rotation Transformation
UGZ	Update block with Given Rotation Transformation
FFT	Fast Fourier transform
GPU	Graphic Processor Unit
CPU	Central Processing Unit
MIMO	Multiple-Input Multiple-Output
PDF	Probability Density Function
BLAS	Basic Linear Algebra Subprograms
API	Application Programming Interface
OpenGL	Open Graphics Library
OpenCL	Open Computing Language
CAL	Compute Abstraction Layer
SIMT	Single-Instruction, Multiple-Thread