UMEÅ UNIVERSITY

Identification and Tuning of Algorithmic Parameters in Parallel Matrix Computations: Hessenberg Reduction and Tensor Storage Format Conversion

Mahmoud Eljammaly

Licentiate Thesis
February 2018

Department of Computing Science
Umeå University
Sweden

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

mjammaly@cs.umu.se

ISBN 978-91-7601-843-9
ISSN 0348-0542
UMINF 18.02

Printed by UmU Print Service, Umeå University
Umeå 2018

Abstract

This thesis considers two problems in numerical linear algebra and high performance computing (HPC): (*i*) the parallelization of a new blocked Hessenberg reduction algorithm using Parallel Cache Assignment (PCA) and the tunability of its algorithm parameters, and (*ii*) storing and manipulating dense tensors on shared memory HPC systems.

The Hessenberg reduction appears in the Aggressive Early Deflation (AED) process for identifying converged eigenvalues in the distributed multishift QR algorithm (state-of-the-art algorithm for computing all eigenvalues for dense square matrices). Since the AED process becomes a parallel bottleneck it motivates a further study of AED components. We present a new Hessenberg reduction algorithm based on PCA which is NUMA-aware and targeting relatively small problem sizes on shared memory systems. The tunability of the algorithm parameters are investigated. A simple off-line tuning is presented and the performance of the new Hessenberg reduction algorithm is compared to its counterparts from LAPACK and ScaLAPACK. The new algorithm outperforms LAPACK in all tested cases and outperforms ScaLAPACK in problems smaller than order 1500, which are common problem sizes for AED in the context of the distributed multishift QR algorithm.

We also investigate automatic tuning of the algorithm parameters. The parameters span a huge search space and it is impractical to tune them using standard auto-tuning and optimization techniques. We present a modular auto-tuning framework which applies: search space decomposition, binning, and multi-stage search to enable searching the huge search space efficiently. The framework using these techniques exposes the underlying subproblems which allows using standard auto-tuning methods to tune them. In addition, the framework defines an abstract interface, which combined with its modular design, allows testing various tuning algorithms.

In the last part of the thesis, the focus is on the problem of storing and manipulating dense tensors. Developing open source tensor algorithms and applications is hard due to the lack of open source software for fundamental tensor operations. We present a software library `dten`, which includes tools for storing dense tensors in shared memory and converting a tensor storage format from one canonical form to another. The library provides two different ways to perform the conversion in *parallel*, in-place and out-of-place. The conversion involves moving blocks of contiguous data and are done to maximize the size of the blocks to move. In addition, the library supports tensor matricization for one or two tensors at the same time. The latter case is important in preparing tensors for contraction operations. The library is general purpose and highly flexible.

Preface

This licentiate thesis consists of an introduction, a summary and the following three papers.

Paper I	M. Eljammaly, L. Karlsson, B. Kågström. On the Tunability of a New Hessenberg Reduction Algorithm Using Parallel Cache Assignment[1]. *Proceeding of the 12th International Conference on Parallel Processing and Applied Mathematics (PPAM 2017)*, LNCS. Springer, (to appear).
Paper II	M. Eljammaly, L. Karlsson, B. Kågström. An Auto-Tuning Framework for a NUMA-Aware Hessenberg Reduction Algorithm. *NLAFET Working Note 18, 2017, and as Report UMINF 17.19,* Department of Computing Science, Umeå University, Sweden, 2017, (a condensed version with the same title has been accepted to the International Conference on Performance Engineering (ICPE 2018)).
Paper III	M. Eljammaly, L. Karlsson. A Library for Storing and Manipulating Dense Tensors. *Report UMINF 16.22*, Department of Computing Science, Umeå University, Sweden, 2016.

[1]Reprinted by permission of Springer.

Acknowledgements

First of all, I thank the almighty God for giving me the strength to finish this work.

I thank my supervisors Professor Bo Kågström and Assoc. Professor Lars Karlsson for their great support and guidance, without them this work would not have been done.

I thank my friends in the Parallel and Scientific Computing research group, my colleagues at the Department of Computing Science and HPC2N, and everyone in UMIT Research Lab for their help and support, and for creating a great working environment. It was both exciting and fun to work in such environment.

I thank my parents, my siblings, and all my family for their great support during my long journey.

Financial support has been received from the European Unions Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633, and by eSSENCE, a strategic collaborative e-Science programme funded by the Swedish Government via VR.

Mahmoud Eljammaly
Umeå February 2018

Contents

Chapter 1

Introduction

1.1 QR Algorithm

Eigenvalue problems (EVPs) appear in many applications in Science and Engineering. Different methods exist for solving various types of EVPs and with matrices of different structure (e.g., dense, sparse, non-symmetric, symmetric). In this thesis we are interested in the standard eigenvalue problem (SEP) for dense non-symmetric matrices, which has the form:

$$Ax = \lambda x \quad (x \neq 0), \tag{1}$$

where A is a dense square matrix, λ is an eigenvalue (scalar) and x is a corresponding eigenvector. With A of size $n \times n$, SEP has n eigenvalues and at most n eigenvectors; if A has multiple eigenvalues it may not exist a full set of eigenvectors. The classical and still most popular algorithm for computing *all* eigenvalues of A is the *QR algorithm* [17, 18]. Given a dense square matrix A with real entries, the QR algorithm computes the eigenvalues by transforming A to a real Schur form in two main stages, illustrated in Figure 1. The first stage performs a similarity transformation of the form:

$$Q_1^T A Q_1 = H, \tag{2}$$

where Q_1 is orthogonal and H is an upper Hessenberg matrix, i.e. $H(i,j) = 0$ for $i > j + 1$. This stage called *Hessenberg reduction* is performed in a finite number of steps. The second stage computes the real Schur form such that:

$$Q_2^T H Q_2 = S, \tag{3}$$

where Q_2 is orthogonal and S is blocked quasi-triangular matrix, i.e. each diagonal block is either 1×1 or 2×2. This stage called *Hessenberg QR algorithm* is performed in an iterative manner using so called *QR iterations.* The eigenvalues of the input matrix A are then the eigenvalues of all the diagonal

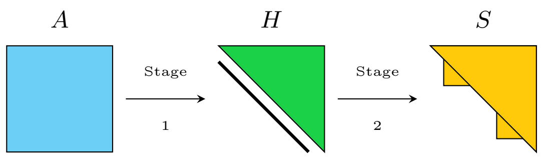

Figure 1: Main stages of the distributed multi-shift QR algorithm.

blocks of S, where the 1×1 blocks are real eigenvalues and the 2×2 blocks correspond to pairs of complex conjugate eigenvalues.

The QR algorithm has passed through many changes and improvements. The state-of-the-art algorithm (blocked but sequential) is known as the *multi-shift QR algorithm* [4, 5]. The QR iteration in the multi-shift QR algorithm involves a robust but costly process called *Aggressive Early Deflation* (AED) [4, 5]. The purpose of this process is to detect and deflate the converged eigenvalues much faster than the classical process of only identifying tiny subdiagonal elements in the Hessenberg form. However, the AED process becomes a bottleneck in the parallel variant of the state-of-the-art algorithm, the *distributed multi-shift QR algorithm* [20]. The AED process consists of three main components acting on a so called AED window (diagonal submatrix): Schur decomposition, eigenvalue reordering, and Hessenberg reduction. In Paper I and II we focus on speeding up the Hessenberg reduction which is part of the AED.

1.2 Hessenberg Reduction

Hessenberg reduction is a similarity transformation (2) which transforms a given dense square matrix A to upper Hessenberg form H. The algorithm reduces the input matrix one column at a time from left to right. The state-of-the-art algorithm [24], which our implementation is based on, performs the reduction in a blocked manner. It divides the input matrix into groups of adjacent columns, called *panels*, and iterates over the panels to reduce them one by one, see Figure 2. In each iteration the algorithm performs two phases. In the first phase (the *reduction phase*) a panel (the orange blocks in Figure 2) is reduced one column at a time using a Householder reflector for each column. The reflectors used to reduce the panel are accumulated and then used in the second phase (the *update phase*) to update the trailing matrix (the cyan blocks in Figure 2). The white trapezoidal blocks in Figure 2 only have zero entries.

In the context of AED, the Hessenberg reduction is applied to relatively small problems (matrices of order hundreds) and within the distributed QR algorithm it is supplied with relatively many more cores than needed. The AED in the distributed QR algorithm uses only a subset of these cores. Hence, we propose to use one shared-memory node with a shared-memory programming model for the AED process. Based on that, in Paper I (which is a condensed version of [14]) we present a new parallel Hessenberg reduction algorithm for

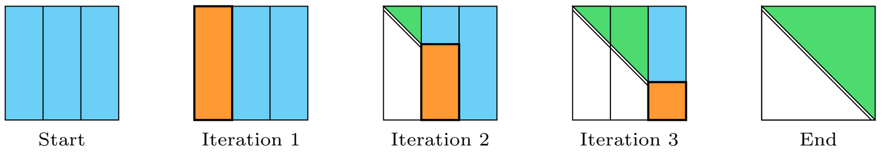

Figure 2: Partitioning of the input matrix into panels and reducing them one by one in the blocked Hessenberg reduction algorithm.

small problems on shared memory. The new algorithm is more efficient and flexible compared to the state-of-the-art algorithms.

To improve on the parallel efficiency, we applied a technique known as *Parallel Cache Assignment* (PCA) [6, 7, 21]. The blocked Hessenberg reduction is a memory-bound problem and the PCA technique is used to transform memory-bound computations to cache-bound computations. The main idea behind the PCA technique is to consider the cores' aggregate cache memory as local memory in a distributed memory system, and to assign work and data to cores such that each core works on data it owns. The PCA technique fits very well the modern shared-memory nodes architecture, which are mostly Non-Unified Memory Access (NUMA). If we bind each thread to a single core and each thread copies its assigned data to a buffer local to it, the algorithm which applies PCA will become NUMA-aware.

To gain flexibility, the new algorithm has many tunable parameters. There are four parameters at *each* iteration of the Hessenberg reduction: the panel width, the number of threads to use in the reduction phase, the number of threads to use in the update phase, and the parallelization strategy to use in the reduction phase (there are two strategies, one of them introduces more parallelism than the other, which is not desirable all the time).

1.3 Automatic Tuning

The performance of the new algorithm in Paper I depends greatly on the value chosen for its algorithm parameters, which need to be tuned for different machines and for different problem sizes. The parameters span a huge search space and they interact with each other such that it is impractical to tune them manually. Instead we need an automatic tuning (or auto-tuning for short) mechanism.

When it comes to auto-tuning we can divide the auto-tuning methods based on *when* the tuning occurs into two groups, off-line and on-line tuning. The *off-line tuning* occurs at installation or compile time, as in the ATLAS library [25], while the *on-line tuning* occurs at runtime as in the FFTW library [19]. Despite the difference between the two libraries in terms of when the tuning occurs, both use what is called *benchmark* computation, that is, computation which is not requested by the user and its results will be discarded. In on-line tuning, however, there is also another option which is *actual* computation. This means, one can use *only* computation requested by the user without any benchmark computation. In Paper I we implemented a simple off-line auto-tuning mechanism using *univariate search* to tune the parameters of the new algorithm. In Paper II, we propose an auto-tuning framework which provides an efficient way to search the huge search space and allows to test various auto-tuning methods easily, both off-line and on-line. We tested the framework using the Nelder-Mead method [23] in off-line tuning mode, but we aim to use the framework with on-line tuning methods in the actual computations.

Paper I (and its longer version [14]) and Paper II are developed under the Horizon 2020 project *Parallel Numerical Linear Algebra for Future Extreme-Scale Systems* (NLAFET) [1] and the *eSSENCE* Strategic Research Program.

1.4 Dense Tensor Storage Formats and Matricization

Tensors or multi-dimensional arrays are used in many multi-dimensional data analysis applications. Yet, most of these applications are either application-specific solutions or based on large commercial software environments. Developing open source tensor algorithms and applications which are independent of commercial software environments is hard due to the lack of open source software support for fundamental tensor operations.

Looking at the history of matrix computation applications we find many similarities with the current state of tensor computation applications, where most software include its own implementation of basic matrix operations. Libraries like BLAS [3, 8, 9, 10, 11, 12, 22] and LAPACK [2] have been developed to unify the usage of basic matrix operations. A great benefit is that software depending on matrix computations has become easier to maintain and now exhibit portable performance. Unfortunately, the field of tensor computations has not matured to the point that a standard interface can be settled on. In addition, tensor algorithms differ in their nature from matrix algorithms so following the same path may not be the best solution.

Nevertheless, we think that developing a software stack as in Figure 3 is a good starting point. The stack consists of six components. Two of them are already established components: the *Basic Linear Algebra Subprograms* (BLAS) and the *Message Passing Interface* (MPI). Most of the tensor computations can be expressed in terms of fundamental matrix operations provided by high-performance BLAS libraries, and MPI is the de-facto standard for communication between nodes in distributed memory HPC systems. The remaining four components are tensor related. From top down these are: the *tensor applications* component, which includes various types of complete applications that use large-scale tensor computations; the *tensor algorithms* component, which includes basic numerical tensor algorithms like tensor contraction and tensor decomposition algorithms; the *tensor distribution* component, which includes tools for (re)distribution and communications of tensor; and the *tensor storage* component, which includes storing and manipulating tensors in a distributed memory system.

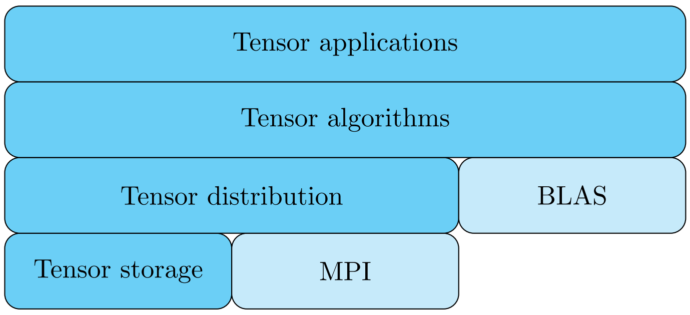

Figure 3: Software stack for tensor computation applications.

The two tensor related components at the bottom of the software stack (data distribution and storage) form the fundamental components of the stack. So, we propose to take a bottom up approach to realize the software stack. Paper III discusses the tensor storage component. Specifically, the storage and manipulation of dense tensors in a shared memory system, which is the fist step toward storing and manipulating dense tensors in a distributed memory system. One of the main points discussed is the relation between tensor matricization (unfolding a tensor into a matrix) and storage format conversions. Indeed, any tensor matricization can be realized as a storage conversion of the tensor from one canonical format to another.

Chapter 2

Summary of Papers

This chapter briefly summarizes the papers in this thesis. Papers I and II concern the tunability of a new Hessenberg reduction algorithm. Paper III presents a library for storing and manipulating dense tensors.

2.1 Paper I

In Paper I [16], we propose a new implementation of the blocked Hessenberg reduction algorithm and study the tunability of its parameters. The new algorithm is parallel, NUMA-aware, and flexible, which are required characteristics to reach high performance measures on different machines for various problem sizes.

The motivation behind the work in this paper is a bottleneck in the distributed multi-shift QR algorithm [20], the state-of-the-art algorithm for computing the Schur form and all eigenvalues of dense matrices. On the critical path of this algorithm lies a component called *Aggressive Early Deflation* (AED) [4, 5] which identifies already converged eigenvalues in the Schur form and accounts for a considerable amount of the total execution time. Hessenberg reduction is one of three main components of the AED process.

The Hessenberg reduction is a memory-bound algorithm which makes it hard to scale well on modern high performance machines. Even the state-of-the-art algorithm [24], which our implementation is based on, suffers from that. To minimize the cost for memory accesses, and to achieve high performance, the new algorithm applies a technique called *Parallel Cache Assignment* (PCA) [6, 7, 21] which is used to transform memory-bound computations to cache-bound computations. In addition, applying PCA in a specific way makes the algorithm NUMA-aware.

To enable flexibility, the new algorithm has many tunable parameters. Specifically, the panel width, the number of threads, and the parallelization strategy must be chosen for each iteration in the reduction. The paper evaluates the tunability of these parameters to find their impact on the performance of the new Hessenberg reduction algorithm. Moreover, a simple off-line auto-tuning mechanism is used to evaluate the performance of the new algorithm after tuning these parameters.

A comparison between the new algorithm and its counterparts in LAPACK and ScaLAPACK is included in the paper. The results show that the new algorithm is faster than LAPACK for all the tested problem sizes and faster than ScaLAPACK for small problem sizes ($n \lesssim 1500$) but competitive with it for larger problems.

2.2 Paper II

In Paper II [15], we present a modular auto-tuning framework that gives support for tuning the parameters of the new algorithm in Paper I [16]. We concluded in Paper I that at each iteration of the new Hessenberg reduction algorithm there are parameters that need tuning, see Section 2.1. These parameters span a huge search space and they interact with each other which makes it impractical to apply standard tuning and optimization techniques directly.

The proposed framework applies different techniques which expose the underlying subproblems and allow us to search the huge search space *efficiently*. The main idea is to tune the original problem (the huge search space) by tuning the subproblems (which are lower-dimensional spaces) independently using standard tuning and optimizing techniques. Moreover, the modular design of the framework allows the testing of different tuning and optimization techniques.

The framework consists of three modules: management, database, and search modules, respectively. The *management module* binds things together, including other modules, the Hessenberg algorithm, and the user I/O. The *database module* keeps track of the subproblems' tuning processes. Finally, the *search module* performs the actual tuning for a subproblem. The search module does not implement a specific tuning algorithm but defines an abstract interface which allow us to encapsulate any tuning technique in the search module.

We implemented the *Nelder-Mead* algorithm [23] in the search module and tested the framework. The results show that the overall performance of the new Hessenberg reduction algorithm is improving over time when using the auto-tuning framework.

2.3 Paper III

In Paper III [13], we present `dten`, a library for storing and manipulating dense tensors (multi-dimensional arrays). The library provides tools for storing dense tensors in canonical storage formats and converting between them efficiently in parallel. In addition, `dten` provides different ways for tensor matricization. The library is generic and have *tunable parameters* to increase its flexibility.

There are many ways to convert the storage format of a dense tensor from one canonical format to another. `dten` finds the most efficient way, in the sense of moving the largest contiguous blocks of data, to perform the conversion. The library provides two ways to perform the conversion: out-of-place and in-place. Out-of-place conversion imposes more parallelism than in-place but uses much more memory while in-place conversion has the opposite characteristics. Moreover, the paper discusses two different ways to implement the in-place conversion.

When it comes to tensor matricization, `dten` performs the matricization as a storage format conversion. The library provides matricization for one or two tensors together. The latter is done to maximize the size of the moving blocks in both tensors together to get the highest overall performance. This is specially important when performing a tensor contraction.

Chapter 3

Future Work

The framework in Paper II allows us to test various auto-tuning algorithms and techniques. We are interested to build an on-line auto-tuning mechanism to tune the parameters of the algorithm from Paper I at run time. Also we are interested to expose more flexibility and apply the ideas from the framework with the on-line tuning to other numerical linear algebra algorithms besides Hessenberg reduction. Creating a flexible linear algebra library which is capable of tuning the parameters of its routines at run time is crucial for high performance algorithms on the future extreme scale systems.

Moreover, the software stack for tensor computation applications is still not complete. We are interested in extending `dten` capabilities to handle tensors in distributed memory to cover both the tensor storage and tensor distribution components of the stack.

Bibliography

[1] http://www.nlafet.eu.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.

[4] K. Braman, R. Byers, and R. Mathias. The Multishift QR Algorithm. Part I: Maintaining Well-Focused Shifts and Level 3 Performance. *SIAM J. Matrix Anal. Appl.*, 23(4):929–947, 2002.

[5] K. Braman, R. Byers, and R. Mathias. The Multishift QR Algorithm. Part II: Aggressive Early Deflation. *SIAM J. Matrix Anal. Appl.*, 23(4):948–973, 2002.

[6] A. Castaldo and R. C. Whaley. Achieving scalable parallelization for the Hessenberg factorization. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 65–73. IEEE, 2011.

[7] A. Castaldo, R. C. Whaley, and S. Samuel. Scaling LAPACK Panel Operations Using Parallel Cache Assignment. *ACM Trans. Math. Softw.*, 39(4), 2013.

[8] J. Dongarra. Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. *International Journal of High Performance Computing Applications*, 16(1,2):1–111,115–199, 2002.

[9] J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[10] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Algorithm 656: An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Trans. Math. Softw.*, 14(1):18–32, March 1988.

[11] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, March 1988.

[12] J. J. Dongarra, J. Du Cruz, S. Hammerling, and I. S. Duff. Algorithm 679: A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Trans. Math. Softw.*, 16(1):18–28, March 1990.

[13] M. Eljammaly and L. Karlsson. *A Library for Storing and Manipulating Dense Tensors. Report UMINF* 16.22, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden, 2016.

[14] M. Eljammaly, L. Karlsson, and B. Kågström. Evaluation of the Tunability of a New NUMA-Aware Hessenberg Reduction Algorithm. *NLAFET Working Note 8*, December, 2016. Also as Report UMINF 16.22, Dept. of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.

[15] M. Eljammaly, L. Karlsson, and B. Kågström. An Auto-Tuning Framework for a NUMA-Aware Hessenberg Reduction Algorithm. In *Proceedings of the 9th ACM/SPEC on International Conference on Performance Engineering (ICPE 2018).* ACM, submitted.

[16] M. Eljammaly, L. Karlsson, and B. Kågström. On the tunability of a new Hessenberg reduction algorithm using parallel cache assignment. In *Proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics (PPAM 2017).* LNCS, Springer, To appear.

[17] J. G. F. Francis. The QR Transformation A Unitary Analogue to the LR Transformation—Part 1. *The Computer Journal*, 4(3):265–271, 1961.

[18] J. G. F. Francis. The QR Transformation—Part 2. *The Computer Journal*, 4(4):332–345, 1962.

[19] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[20] R. Granat, B. Kågström, D. Kressner, and M. Shao. ALGORITHM 953: Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation. *ACM Trans. Math. Softw.*, 41(4):Article 29:1–23, 2015.

[21] M. R. Hasan and R. C. Whaley. Effectively Exploiting Parallel Scale for all Problem Sizes in LU Factorization. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1039–1048. IEEE, 2014.

[22] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[23] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[24] G. Quintana-Ortí and R. van de Geijn. Improving the performance of reduction to Hessenberg form. *ACM Trans. Math. Softw.*, 32(2):180–194, 2006.

[25] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.