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

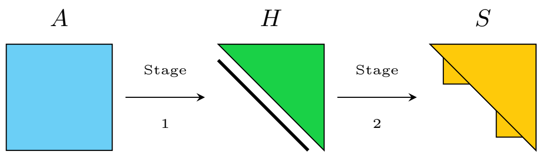


Figure 1: Main stages of the distributed multi-shift QR algorithm.

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

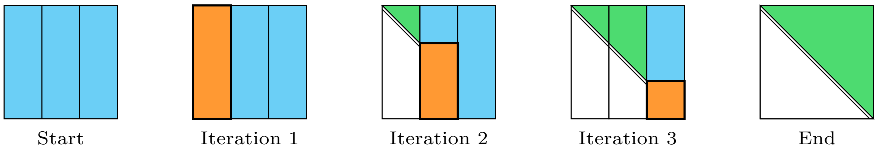


Figure 2: Partitioning of the input matrix into panels and reducing them one by one in the blocked Hessenberg reduction algorithm.

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.

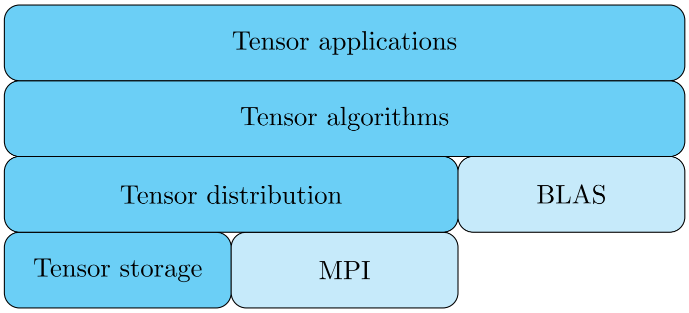


Figure 3: Software stack for tensor computation applications.

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.

Paper I

On the Tunability of a New Hessenberg Reduction Algorithm Using Parallel Cache Assignment

Mahmoud Eljammaly, Lars Karlsson, and Bo Kågström

Proceeding of the 12th International Conference on Parallel Processing and Applied Mathematics (PPAM 2017), LNCS. Springer, (to appear).

On the Tunability of a New Hessenberg Reduction Algorithm Using Parallel Cache Assignment

Mahmoud Eljammaly, Lars Karlsson, and Bo Kågström

Umeå University, SE 901 87 Umeå, Sweden,
{mjammaly,larsk,bokg}@cs.umu.se

Abstract. The reduction of a general dense square matrix to Hessenberg form is a well known first step in many standard eigenvalue solvers. Although parallel algorithms exist, the Hessenberg reduction is one of the bottlenecks in AED, a main part in state-of-the-art software for the distributed multishift QR algorithm. We propose a new NUMA-aware algorithm that fits the context of the QR algorithm and evaluate the sensitivity of its algorithmic parameters. The proposed algorithm is faster than LAPACK for all problem sizes and faster than ScaLAPACK for the relatively small problem sizes typical for AED.

Keywords: Hessenberg reduction, parallel cache assignment, NUMA-aware algorithm, shared-memory, tunable parameters, off-line tuning.

1 Introduction

This work is motivated by a bottleneck in the distributed parallel multi-shift QR algorithm for large-scale dense matrix eigenvalue problems [7]. On the critical path of the QR algorithm lies an expensive procedure called *Aggressive Early Deflation* (AED) [1, 2]. The purpose of AED is to detect and deflate converged eigenvalues and to generate shifts for subsequent QR iterations. There are three main steps in AED: Schur decomposition, eigenvalue reordering, and Hessenberg reduction. This work focuses on the last step while future work will investigate the first two steps.

In the context of AED, Hessenberg reduction is applied to relatively small problems (matrices of order hundreds to thousands) and, since AED appears on the critical path of the QR algorithm, there are relatively many cores available for its execution. The distributed QR algorithm presented in [7] computes the AED using a subset of the processors. We propose to select one shared-memory node and use a shared-memory programming model (OpenMP) for the AED. The aim is to develop a new parallel Hessenberg reduction algorithm which outperforms the state-of-the-art algorithm for small problems by using fine-grained parallelization and tunable algorithmic parameters to make it more efficient and flexible. Tuning the algorithmic parameters of the new algorithm is not one of the main concerns in this paper. Rather, this work focuses on the tunability potential of the algorithmic parameters.

A shared-memory node within a distributed system commonly has a *Non-Uniform Memory Access* (NUMA) architecture. Since Hessenberg reduction is a memory-bound problem where matrix–vector multiplications typically account for most of the execution time, high performance is obtained when the cost of memory accesses is minimized. Therefore, our algorithm employs the *Parallel Cache Assignment* (PCA) technique proposed by Castaldo and Whaley [4, 5, 8]. This technique leads to two benefits. First, the algorithm becomes NUMA-aware. Second, the algorithm uses the aggregate cache capacity more effectively.

The rest of the paper is organized as follows. Section 2 reviews a blocked Hessenberg reduction algorithm and the PCA technique. Section 3 describes how we applied the PCA technique to the blocked algorithm. Section 4 evaluates the impact of tuning each parameter. Section 5 shows the new algorithm's performance after tuning and compares it with state-of-the-art implementations. Section 6 concludes and highlights future work.

2 Background

2.1 Blocked Hessenberg Reduction

In this section we review the basics of the state-of-the-art algorithm in [11] on which our algorithm is based. Hessenberg reduction transforms a given square matrix $A \in \mathbb{R}^{n\times n}$ to an upper Hessenberg matrix $H = Q^T AQ$, where Q is an orthogonal matrix. A series of Householder reflections applied to both sides of A are used to zero out—*reduce*—the columns one by one from left to right.

The algorithm revolves around block iterations, each of which reduces a block of adjacent columns called a *panel*. After reducing the first $k-1$ columns, the matrix A is partitioned as in Fig. 1, where b is the *panel width*.

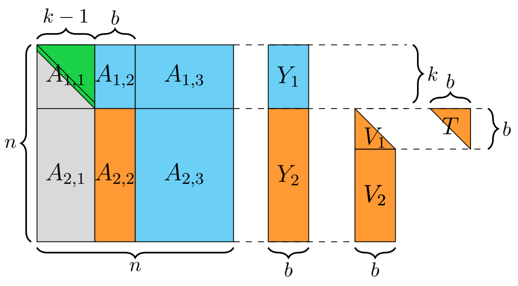


Fig. 1. Partitioning of A after reducing the first $k-1$ columns, and Y, V and T to be used for reducing $A_{2,2}$.

The panel $A_{2,2}$ (starting at the sub-diagonal) is reduced to upper triangular form by constructing and applying a transformation of the form

$$A \leftarrow (I - VTV^T)^T A(I - VTV^T) \ ,$$

where $I - VTV^T$ is a compact WY representation [12] of the b Householder reflections that reduced the panel. In practice, the algorithm incrementally builds an intermediate matrix $Y = AVT$ to eliminate redundant computations in the updates from the right. The matrix Y is partitioned as in Fig. 1. Each block iteration consists of two phases. In the first phase, the panel $A_{2,2}$ is reduced and fully updated. This gives rise to a set of b Householder reflections, which are accumulated into a compact WY representation $I - VTV^T$. The first phase also incrementally computes $Y_2 \leftarrow A_{2,2:3}VT$. In the second phase, $Y_1 \leftarrow A_{1,2:3}VT$ is computed, and blocks $A_{1,2}$, $A_{1,3}$, and $A_{2,3}$ are updated according to

$$A \leftarrow (I - VTV^T)^T (A - YV^T) \ , \tag{1}$$

where the dimensions of A, V, T and Y are derived from Fig. 1 according to which block is to be updated.

Other Variants of Hessenberg Reduction. A multi-stage Hessenberg reduction algorithm exists [9]. In this variant, some of the matrix-vector operations are substituted by matrix-matrix operations for the cost of performing more compute-bound computations overall. Applying PCA to this variant will be much less efficient since PCA is useful when we have repetitive memory-bound computations, as explained in Sect. 2.2.

2.2 PCA: Parallel Cache Assignment

Multicore shared-memory systems have parallel cache hierarchies with sibling caches on one or more levels. In such systems, the aggregate cache capacity might be able to persistently store the whole working set. To exploit this phenomenon, Castaldo and Whaley proposed the PCA technique and applied it to the panel factorizations of one-sided factorizations [5] as well as to the unblocked Hessenberg reduction algorithm [4]. They argued that PCA is able to turn memory-bound computations of small problems into cache-bound (or even compute-bound) computations by utilizing the parallel caches to transform the vast majority of memory accesses into local cache hits.

The main idea of PCA is to consider sibling caches as local memories in a distributed memory system and to assign to each core a subset of the data. Work is then assigned using the owner-computes rule. In addition, one may explicitly copy the data assigned to a specific core into a local memory to that core.

A pivotal aspect to benefit from using PCA is having a repeated memory-bound computation for the same memory region. Applying PCA allows fetching a large block of data from the main memory into several caches and use it repeatedly while still in the cache, which eliminates the slowdown penalty presented by repeatedly using the memory buses.

3 Hessenberg Reduction Using PCA

The proposed algorithm (Algorithm 1) is a parallel variant of [11] using PCA and aimed at small matrices. The algorithm consists of two nested loops. The inner loop, lines 7–24, implements the first phase while the remainder of the outer loop, lines 25–30, implements the second phase. In the following, we briefly describe the parallelization of each phase. For more details see the technical report [6].

3.1 Parallelization of the First Phase

The first phase is memory-bound due to the large matrix–vector multiplications on lines 17–18. The objective is to apply PCA to optimize the memory accesses. We partition A, V, and Y as illustrated in Fig. 1. This phase consists of four main steps for each column $\mathbf{a} = A_{2,2}(:, j)$ of the panel: update $\mathbf{a}$ from the right (lines 9–10), update $\mathbf{a}$ from the left (lines 11–15), reduce $\mathbf{a}$ (line 16), augment Y and T (lines 17–24). Two parallelization strategies are considered for this phase. In the *full strategy*, all multiplications except triangular matrix–vector are parallelized. In the *partial strategy*, only the most expensive computational step, lines 17–18, is parallelized. The full strategy exposes more parallelism at the cost of more overhead which makes it suitable only for sufficiently large problems.

To apply PCA, before each first phase the data are assigned to threads where each thread mainly works on data it owns. The matrix–vector multiplications in this phase involve mostly tall–and–skinny or short–and–fat matrices. For efficient parallelization in the full strategy, the matrices are partitioned along their longest dimension into p_1 parts assigned to p_1 threads. To parallelize the costly step in lines 17–18, $A_{2,2:3}$ is first partitioned into p_1 block rows then each thread *explicitly copies* its assigned block into local memory, (line 6). Having the assigned data from this block in a buffer local to the thread will reduce the amount of remote memory accesses, cache conflicts and false sharing incidents, which make the algorithm NUMA-aware. So even if the data did not fit into the cache, the algorithm will still benefit from the data locality. In general, all matrices are distributed among the threads in a round-robin fashion based on memory-pages.

3.2 Parallelization of the Second Phase

The second phase is compute-bound and mainly involves matrix–matrix multiplications. The objective is to balance the workload and avoid synchronization as much as possible. There are four main steps: updating $A_{2,3}$ from the right (lines 26–27), updating $A_{2,3}$ from the left (line 28), computing Y_1 (line 29), and updating $A_{1,2:3}$ (line 30). With conforming block partitions of the columns of $A_{2,3}$ and V_2^T, and of the block rows of $A_{1,2:3}$ and Y_1 (line 25) the computation can be performed without any synchronization.

Algorithm 1: Parallel blocked Hessenberg reduction using PCA.

```
1  for $k \leftarrow 1 : b : n-2$ do // Outer loop over panels
2      $V \leftarrow 0_{n-k\times 0}$, $T \leftarrow 0_{0\times 0}$, $Y \leftarrow 0_{n\times 0}$// Initialize intermediate matrices
3      if $s = full$ then $\hat{p} \leftarrow p_1$ else $\hat{p} \leftarrow 1$ // Select strategy
4      Partition $A$, $V$, and $Y$ as in Fig. 1
5      Partition $A_{2,2:3}$ into $p_1$ row blocks $A^{(i)}_{2,2:3}$ for $i = 1 \ldots p_1$
6      Thread $i$ copies $A^{(i)}_{2,2:3}$ to local memory
       // First Phase
7      for $j \leftarrow 1 : \min\{b, n-k-1\}$ do
8          Partition $A_{2,2}(:,j), V, V_2, \mathbf{v}_j, Y_2$ and $\mathbf{y}_j$ into $\hat{p}$ row blocks
             $A^{(i)}_{2,2}(:,j), V^{(i)}, V^{(i)}_2, \mathbf{v}^{(i)}_j, Y^{(i)}_2$ and $\mathbf{y}^{(i)}_j$ for $i = 1 \ldots \hat{p}$
           // Update column $j$ of $A_{22}$ from both sides
9          parfor $i \leftarrow 1 : \hat{p}$ do
10             $A^{(i)}_{2,2}(:,j) \leftarrow A^{(i)}_{2,2}(:,j) - Y^{(i)}_2 V_2(1,:)^T$
11             $\mathbf{w}^{(i)} \leftarrow V^{i^T} A^{(i)}_{2,2}(:,j)$
12         $\mathbf{w} \leftarrow \mathbf{w}^{(1)} + \cdots + \mathbf{w}^{(\hat{p})}$
13         $\mathbf{w} \leftarrow T^T \mathbf{w}$
14         parfor $i \leftarrow 1 : \hat{p}$ do
15             $A^{(i)}_{2,2}(:,j) \leftarrow A^{(i)}_{2,2}(:,j) - V^{(i)}\mathbf{w}$
16         Construct a Householder reflection $(\mathbf{v}_j, \tau_j)$ that reduces $A_{2,2}(j+1:n, j)$
           // Augment $Y$, $T$, and $V$
17         parfor $i \leftarrow 1 : p_1$ do
18             $\mathbf{y}^{(i)} \leftarrow A^{(i)}_{2,2:3}(:, j+1:n)\mathbf{v}_j$
19         parfor $i \leftarrow 1 : \hat{p}$ do
20             $\mathbf{t}^{(i)} \leftarrow V^{(i)^T}_2 \mathbf{v}^{(i)}_j$
21         $\mathbf{t} \leftarrow \mathbf{t}^{(1)} + \cdots + \mathbf{t}^{(\hat{p})}$
22         parfor $i \leftarrow 1 : \hat{p}$ do
23             $\mathbf{y}^{(i)} \leftarrow \tau \mathbf{y}^{(i)} - Y^{(i)}_2 \mathbf{t}$
24         $Y \leftarrow \begin{bmatrix} Y_1 & 0 \\ Y_2 & \mathbf{y} \end{bmatrix}$, $T \leftarrow \begin{bmatrix} T & -\tau_j T\mathbf{t} \\ 0 & \tau_j \end{bmatrix}$, $V \leftarrow \begin{bmatrix} V & \mathbf{v}_j \end{bmatrix}$
       // Second Phase
25     Partition $A_{2,3}$ into $p_2$ column blocks $A^{(i)}_{2,3}$ and $A_{1,2:3}(:, 2:n), Y_1$ and $V_2$ into
         $p_2$ row blocks $A^{(i)}_{1,2:3}(:, 2:n), Y1^{(i)}$ and $V^{(i)}_2$ for $i = 1 \ldots p_2$
26     parfor $i \leftarrow 1 : p_2$ do
           // Update $A_{2,3}$ from the right
27         $A^{(i)}_{2,3} \leftarrow A^{(i)}_{2,3} - Y_2 V^{(i)^T}_2$
           // Update $A_{2,3}$ from the left
28         $A^{(i)}_{2,3} \leftarrow A^{(i)}_{2,3} - V T^T V^T A^{(i)}_{2,3}$
           // Compute the top block of $Y$
29         $Y^{(i)}_1 \leftarrow A^{(i)}_{1,2:3}(:, 2:n) V T$
           // Update $A_{1,2:3}$ from the right
30         $A^{(i)}_{1,2:3}(:, 2:n) \leftarrow A^{(i)}_{1,2:3}(:, 2:n) - Y^{(i)}_1 V^T$
```

3.3 Algorithmic Parameters

There are four primary algorithmic parameters: the panel width, the parallelization strategy, and the thread counts for both phases. The panel width b can be set to any value in the range $1, \ldots, n-2$. The first phase can be parallelized using either the full or the partial parallelization strategy, as described in Sect. 3.1. The strategy $s \in \{\text{full, partial}\}$ can be set independently for each iteration of the outer loop. Using all available cores can potentially hurt the performance, especially near the end where the operations are small-sized. The synchronization overhead and cache interference may outweigh the benefits of using more cores. Therefore, the number of threads to use in each phase (p_1 and p_2) are tunable parameters that can be set independently in each outer loop iteration. If the thread count is less than the number of available cores, then threads are assigned to as few NUMA domains as possible to maximize memory throughput.

4 Evaluation of the Tuning Potential

This section evaluates the tuning potential of each algorithmic parameter while keeping all the others at their default setting.

The experiments were performed on the Abisko system at HPC2N, Umeå University. During the experiments, no other jobs were running on the same node. One node consists of four AMD Opteron 6238 processors each containing two chips with six cores each for a total of 48 cores. Each chip has its own memory controller, which means that the node has eight NUMA domains. The PathScale (5.0.0) compiler is used together with the following libraries: OpenMPI (1.8.1), OpenBLAS (0.2.13), LAPACK (3.5.0), and ScaLAPACK (2.0.2). The default parameter values in Table 1 were used in the experiments unless otherwise stated. All reported data points is the median of 100 trials, unless otherwise stated.

Tuning Potential for the Panel Width. The panel width plays a key role in shaping the performance since it determines the distribution of work. To find how b depends on the problem size we used $n \in \{500, 1000, \ldots, 4000\}$. Figure 2 shows the execution time of the new algorithm for different problem sizes and panel widths. The stars correspond to the best b found for each problem size. The algorithm execution time is sensitive to the choice of b which means b need tuning.

Tuning Potential for the Parallelization Strategy. The *partial strategy* is expected to be faster for small panels due to its lower parallelization overhead, while the *full strategy* is expected to be faster for large panels due to its higher degree of parallelism. Figure 3 shows the execution times per iteration of the outer loop for both strategies for $p = 48$ and $n = 4000$. For the first 20 or so iterations, the *full strategy* is faster, while the opposite is true for the remaining iterations. Hence, s needs tuning to find which strategy to use for each iteration of a reduction. For a smaller n and the same fixed parameters, the resulting figure is a subset of Fig. 3, e.g., for $n = 2000$, the resulting figure consists of iterations 40 to 80 of Fig. 3.

Table 1. Default values for the algorithmic parameters.

Parameter	*Default*
Panel width	$b = 50$
Thread count	$p_1 = p_2 = p$
Parallelization strategy	$s =$ partial

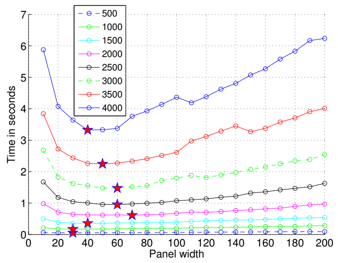


Fig. 2. Effect of the panel width on the execution time for $p = 48$ and $n \in \{500, 1000, \ldots, 4000\}$ with all other parameters as in Table 1. The stars represent the best b for each n.

Tuning Potential for the Thread Counts. The number of threads used in each phase affects the performance since it affects both the cache access patterns and the parallel overhead. To find the optimal configuration it suffices to know the execution time of each of the two phases in every iteration for each thread count since the phases do not overlap. These data can be obtained by repeating the same execution with different fixed thread counts. The time measurements are collected in two tables: T_1 for the first phase and T_2 for the second phase (not explicitly showed). One row per thread count and one column per iteration. To find the optimal thread count for a particular phase and iteration, one scans the corresponding column of the appropriate table and selects the thread count (row) with the smallest entry. Figure 4 compares the effect of varying the thread counts as opposed to always using the maximum number (48). The result shows that varying the thread counts is better, which means we need to tune the thread counts for each phase and iteration.

More evaluation results. A more thorough evaluation is discussed in the technical report [6]. Specifically, the report includes an evaluation of varying the panel width at each iteration of the reduction. The results show that the gain is insignificant compared to varying the panel width once per reduction. The evaluation of either performing the explicit data redistribution (copying to local buffers) or not is also included. The results show that it is always useful to redistribute the data. In addition, more cases for evaluating the effect of varying the thread counts are considered. The cases include experimenting with varying either p_1 or p_2 while fixing the other to the max, varying both but keeping $p_1 = p_2$, testing for a different problem size ($n = 4000$), and distributing the threads to the cores in two scheme: packed and round-robin. The general conclusion of all these cases is that p_1 and p_2 need to be tuned independently.

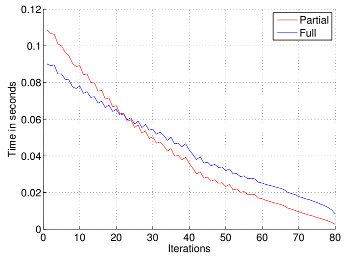


Fig. 3. Comparison of the full and partial strategies for $p = 48$ and $n = 4000$ with all other parameters as in Table 1.

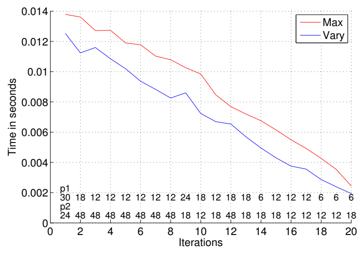


Fig. 4. Comparison of varying the thread counts and using maximum number of cores (48) for $n = 1000$ with all other parameters as in Table 1. The numbers at the bottom of the figure are the thread counts used in each iteration for each phase.

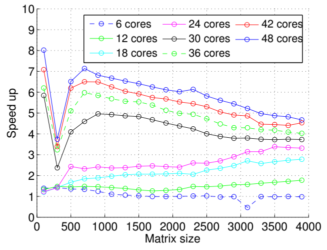
5 Performance Comparisons

This section illustrates the performance of the new parallel algorithm after tuning and compares it with LAPACK and ScaLAPACK over a range of problem sizes.

Off-Line Auto-tuning. To tune the parameters we used several rounds of *univariate search*. Our objective is not to come up with the best off-line auto-tuning mechanism but rather to get a rough idea how the new algorithm performs after tuning. Univariate search works by optimizing one variable at a time, in this case through exhaustive search, while fixing the other variables. The parameters are tuned separately for each problem size and number of cores.

Hessenberg reduction with and without PCA. Figure 5 shows the speed up of the Hessenberg algorithm with PCA against without PCA. The LAPACK routine `DGEHRD` was used as the variant without PCA since it is the closest in its implementation to the new algorithm. The comparison made for square matrices of size $n \in \{100, 300, \dots, 3900\}$ using $p \in \{6,\ 12, \dots, 48\}$. To have a fair comparison, the parameters of the PCA variant are fixed to the default values in Table 1. The results show that for most cases the PCA variant is faster.

Performance of The New Algorithm. To measure the new algorithm performance, tests are run on square matrices of size $n \in \{100, 300, \dots, 3900\}$ using $p \in \{6,\ 12, \dots, 48\}$ threads with 15 rounds of tuning. Figure 6 shows the performance measured in GFLOPS of the new algorithm after tuning on different numbers of cores. It is inconvenient to present all the parameter values in all tests since there are thousands of them. The results show that for small problems ($n \lesssim 2000$), it is not optimal to use the maximum number of cores (48).



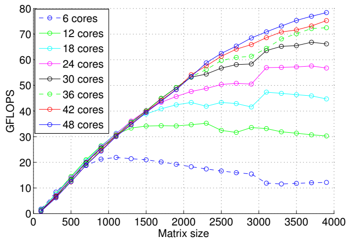


Fig. 5. Speed up comparison between the Hessenberg reduction algorithm with PCA, using the default parameters in Table 1, and without PCA.

Fig. 6. Performance of the new algorithm using 1-8 NUMA domains.

Comparison with LAPACK and ScaLAPACK. Figure 7 shows the speed up of the new algorithm after tuning against the `DGEHRD` routine from LAPACK and the `PDGEHRD` routine from ScaLAPACK. The three routines are run using $p \in \{6, 12, 18, \cdots, 48\}$ threads for each problem of size $n \in \{100, 300, \cdots, 3900\}$. The numbers in the figure indicate for each implementation which p gives the best performance for each n. The comparison for each n is then made between the best case of the three implementations. Table 2 shows the values of b and s which are used in the new algorithm for each best case. For $n \geq 3100$, the *full strategy* is used for the first few iterations then the *partial strategy* is used. It is inconvenient to present the values of p_1 and p_2 for each case. Instead, we summarize how they change during the reduction. Generally, any reduction starts with $p_1 = p_2 = p$, then p_1 gradually decreases until it reaches the minimum number of threads (6), while p_2 decreases but less gradually and does not necessarily reaches the minimum. The results show that the new algorithm outperforms LAPACK for all the tested problems while it outperforms ScaLAPACK only for small problems ($n \lesssim 1500$), a possible reason is that ScaLAPACK might be using local memory access for both phases.

Comparison with other libraries. There are other libraries for numerical linear algebra than LAPACK and ScaLAPACK. The latest release (2.8) of the PLASMA [3] library does not support Hessenberg reduction, while MAGMA [13] uses GPU which is not our focus. On the other hand, libFLAME [14] uses the LAPACK routine for a counterpart implementation, while the implementation from Elemental library [10] produces comparable results to ScaLAPACK in the best case speed up comparison.

Table 2. The panel widths and strategies of the new algorithm after tuning for the cases used in the comparison in Fig 7.

n	b	s	n	b	s
100	30	partial	2100	60	partial
300	30	partial	2300	60	partial
500	30	partial	2500	60	partial
700	30	partial	2700	50	partial
900	40	partial	2900	60	partial
1100	40	partial	3100	60	full until 4
1300	40	partial	3300	60	full until 7
1500	40	partial	3500	60	full until 11
1700	50	partial	3700	60	full until 14
1900	60	partial	3900	60	full until 19

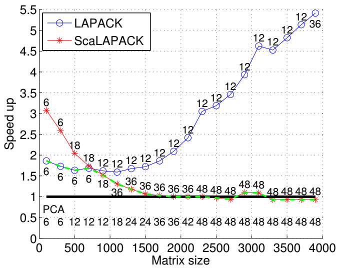


Fig. 7. Best case speed up comparison between our new algorithm after tuning and its counterparts in LAPACK and ScaLAPACK (block size 50×50). The numbers in the figure show the value of p which gives the best performance for each n.

6 Conclusion

We presented a new parallel algorithm for Hessenberg reduction which applies the PCA technique to an existing algorithm. The algorithm is aimed to speed up the costly AED procedure which lies on the critical path of the distributed parallel multi-shift QR algorithm [7]. The proposed algorithm has a high degree of flexibility (due to tens or hundreds of tunable parameters) and memory locality (due to the application of PCA). The impact of various algorithmic parameters of the new algorithm were evaluated. The panel width, the parallelization strategy and the thread counts found to have a significant impact on the algorithm performance and though they need tuning. A basic off-line auto-tuning using univariate search is used to tune the parameters. The proposed solution with tuning outperforms LAPACK's routine `DGEHRD` for all cases and ScaLAPACK's routine `PDGEHRD` for small problem sizes.

Future work includes designing an on-line auto-tuning mechanism. The aim is to obtain an implementation that continuously improves itself the more it is being used. A major challenge is how to effectively handle the per-iteration parameters (thread count and parallelization strategy) as well as how to share information across nearby problem sizes.

Acknowledgements. We thank the High Performance Computing Center North (HPC2N) at Umeå University for providing computational resources and valuable support during test and performance runs. 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.

References

1. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. part I: Maintaining well-focused shifts and level 3 performance. SIMAX 23(4), 929–947 (2002), doi:10.1137/S0895479801384573
2. Braman, K., Byers, R., Mathias, R.: The multishift QR algorithm. Part II: Aggressive early deflation. SIMAX 23(4), 948–973 (2002), doi:10.1137/S0895479801384585
3. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing 35(1) (2009), doi:10.1016/j.parco.2008.10.002
4. Castaldo, A., Whaley, R.C.: Achieving scalable parallelization for the Hessenberg factorization. In: Cluster Computing (CLUSTER), 2011 IEEE International Conference on. pp. 65–73. IEEE (2011), doi:10.1109/CLUSTER.2011.16
5. Castaldo, A., Whaley, R.C., Samuel, S.: Scaling LAPACK panel operations using parallel cache assignment. ACM TOMS 39(4) (2013), doi:10.1145/2491491.2491493
6. Eljammaly, M., Karlsson, L., Kågström, B.: 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.
7. Granat, R., Kågström, B., Kressner, D., Shao, M.: ALGORITHM 953: Parallel Library Software for the Multishift QR Algorithm with Aggressive Early Deflation. ACM Trans. Math. Software 41(4), Article 29:1–23 (2015), doi:10.1145/2699471
8. Hasan, M.R., Whaley, R.C.: Effectively exploiting parallel scale for all problem sizes in LU factorization. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. pp. 1039–1048. IEEE (2014), doi:10.1109/IPDPS.2014.109
9. Karlsson, L., Kågström, B.: Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. Parallel Computing 37(12), 771 – 782 (2011), 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10), doi:10.1016/j.parco.2011.05.001
10. Poulson, J., Marker, B., van de Geijn, R.A., Hammond, J.R., Romero, N.A.: Elemental: A new framework for distributed memory dense matrix computations. ACM Trans. Math. Softw. 39(2), 13:1–13:24 (Feb 2013), doi:10.1145/2427023.2427030
11. Quintana-Ortí, G., van de Geijn, R.: Improving the performance of reduction to Hessenberg form. ACM TOMS 32(2), 180–194 (2006), doi:10.1145/1141885.1141887
12. Schreiber, R., Loan, C.V.: A storage efficient WY representation for products of Householder transformations. Tech. Rep. 1 (1989), doi:10.1137/0910005
13. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing 36(5-6), 232–240 (2010), doi:10.1016/j.parco.2009.12.005
14. Zee, F.G.V., Chan, E., van de Geijn, R.A., Quintana-Ortí, E.S., Quintana-Ortí, G.: The libflame library for dense matrix computations. Computing in Science and Engg. 11(6), 56–63 (Nov 2009), doi:10.1109/MCSE.2009.207

Paper II

An Auto-Tuning Framework for a NUMA-Aware Hessenberg Reduction Algorithm

Mahmoud Eljammaly, Lars Karlsson, and Bo Kågström

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)).

An Auto-Tuning Framework for a NUMA-Aware Hessenberg Reduction Algorithm*

Mahmoud Eljammaly
Department of Computing Science
`mjammaly@cs.umu.se`

Lars Karlsson
Department of Computing Science
`larsk@cs.umu.se`

Bo Kågström
Department of Computing Science and HPC2N
`bokg@cs.umu.se`

Abstract

The performance of a recently developed Hessenberg reduction algorithm greatly depends on the values chosen for its tunable parameters. The search space is huge combined with other complications makes the problem hard to solve effectively with generic methods and tools. We describe a modular auto-tuning framework in which the underlying optimization algorithm is easy to substitute. The framework exposes sub-problems of standard auto-tuning type for which existing generic methods can be reused. The outputs of concurrently executing sub-tuners are assembled by the framework into a solution to the original problem.

Keywords: Auto-tuning, Tuning framework, Binning, Search space decomposition, Multistage search, Hessenberg reduction, NUMA-aware.

1 Introduction

The motivation behind this work starts from the distributed parallel multishift QR algorithm [9], which is the key step in solving large dense unsymmetric eigenvalue problems. On the critical path of the distributed QR algorithm lies a costly process known as Aggressive Early Deflation (AED) [2, 3]. The purpose of AED is two-fold: to detect and deflate converged eigenvalues and to generate shifts for subsequent QR iterations. Aggressive early deflation is composed of three major parts: Schur decomposition, eigenvalue

*NLAFET Working Note 18. Report UMINF 17.19, Dept. Computing Science, Umeå University, SE 901 87 Umeå, Sweden.

reordering, and Hessenberg reduction. The AED process is currently a bottleneck in the distributed QR algorithm and we aim to accelerate it in the hopes of improving the performance and scalability of the QR algorithm. We recently developed a new NUMA-aware Hessenberg reduction algorithm [7] based on the Parallel Cache Assignment (PCA) technique [4, 5, 10]. The performance of the new algorithm depends greatly on the values chosen for its tunable parameters. Auto-tuning is required due to both the large number of parameters (four per iteration; see Section 2.1 ahead) and the interactions between different parameters.

In this paper, we propose a modular auto-tuning framework that helps with the tuning process. In particular, the framework tries to search the huge search space efficiently by partitioning the parameters into subsets that are tuned independently, grouping similar sub-problems into the same bin and tune them as one, and searching in multiple stages (first coarsely and then finely). The framework by itself is not a complete solution. At the heart of the framework is a generic module for optimizing a sub-problem of standard type. The framework provides a clean interface to generic optimization methods and extends them into an auto-tuner for the complex and non-standard original problem. Besides the main benefit of reducing the complex optimization problem into something more manageable, this architecture has the added benefit of making it easy to experiment with different search algorithms.

The framework works as pre- and post-processing layers around the Hessenberg algorithm. The interactions between the framework and the algorithm are as follows. The user provides to the framework an input matrix $A \in \mathbb{R}^{n \times n}$ and the number, p, of available cores. Based on n and p, the framework chooses specific values for all the algorithmic parameters of the Hessenberg algorithm. The framework then executes the Hessenberg algorithm on A with the specified parameters. The output matrices H and Q are returned to the user. Simultaneously, the Hessenberg algorithm feeds back internal time measurements to the framework for use in the tuning process.

The rest of the paper is organized as follows. Section 2 describes the details of the new implementation of blocked Hessenberg reduction. The algorithmic parameters and their interaction are identified and discussed in Section 2.1. Section 3 describes techniques used within the framework to efficiently search the huge search space. Section 4 describes the architecture of the auto-tuning framework. Section 5 shows experimental results. Finally, Section 6 sums up the paper and outlines future work.

2 NUMA-Aware Hessenberg Reduction

Hessenberg reduction is a similarity transformation that maps a matrix $A \in \mathbb{R}^{n \times n}$ to an upper Hessenberg matrix $H = Q^T AQ$, where Q is an orthogonal matrix. The current state-of-the-art algorithm [12] performs the reduction in a blocked manner. The matrix is reduced iteratively one block of columns (called a *panel*) at a time from left to right. Each panel is reduced column-by-column using Householder reflectors. The reflectors are also applied to the rest of the matrix to update it. Most of the work associated with the updates are delayed. More precisely, one iteration consists of two phases: a *(panel) reduction phase*, in which the panel is reduced, and an *(delayed) update phase*, in which the delayed updates are fully applied. Let $I - VTV^T$ denote the compact WY representation [13] of all reflectors from one panel reduction. One iteration logically applies the similarity transformation

$$A \leftarrow (I - VTV^T)^T A(I - VTV^T) = (I - VTV^T)^T (A - YV^T), \quad (1)$$

where $Y = AVT$; see [12] for details. Our recently developed NUMA-aware parallel variant of [12] is summarized in Algorithm 1.

Figure 1 shows the shapes of A, V, and Y after the first k columns of A have been reduced. Here b refers to the width (number of columns) of the next panel.

In the reduction phase, the panel A_{22} is reduced. To reduce a column in A_{22}, the column is first updated using (1), lines 6 to 7, and then reduced by a Householder reflection, line 8. The reflection is augmented into the compact WY representation. This process affects Y_2, T, and V, lines 9 to 15.

The operations in the reduction phase are mainly matrix–vector operations, which therefore makes the whole phase memory-bound. The most expensive operation is a large matrix–vector multiplication involving $A_{2,2:3}$ during the computation of $\mathbf{y}$, lines 9 to 10. To perform this multiplication efficiently, our NUMA-aware algorithm [7] uses the PCA technique [5, 4, 10]. This makes for efficient utilization of the aggregate cache capacity and more localized access to main memory. Applying PCA means to (logically or physically) distribute the data over the threads/cores and then distribute the work according to the owner-computes rule. Concretely, before the start of the reduction phase, $A_{2,2:3}$ is partitioned into uniform row blocks and each block is assigned to one thread.

The NUMA-aware algorithm provides two alternative parallelization strategies for the reduction phase. In the *partial parallelization strategy*, multi-threading is used only for the most expensive multiplication (i.e., lines 9 to 10) while in the *full parallelization strategy* multi-threading is used for most of the operations.

In the update phase, Y_1 is efficiently computed directly from its definition $Y = AVT$ and A_{12}, A_{13}, and A_{23} are updated using (1), lines 16 to 19. All operations in this phase are efficient matrix multiplications, which makes it compute-bound. All computations in the update phase are parallelized.

Algorithm 1: Parallel blocked Hessenberg reduction using PCA.

```
   // Outer loop over panels
1  foreach panel do
       // Select strategy
2      if $s = full$ then $p \leftarrow t_r$ else $p \leftarrow 1$
3      Partition $A$, $V$, and $Y$ as in Figure 1 with panel width $b$
4      Assign and redistribute data to workers
       // Reduction Phase
5      foreach column $\mathbf{a}$ in panel $A_{2,2}$ do
6          parfor $i \leftarrow 1 : p$ do
7              Update column $\mathbf{a}^{(i)}$ of $A_{2,2}$
8          Construct a Householder reflection that reduces column $\mathbf{a}$ of
            $A_{2,2}$
9          parfor $i \leftarrow 1 : t_r$ do
10             Compute column $\mathbf{y}^{(i)}$ of $Y_2$
11         parfor $i \leftarrow 1 : p$ do
12             Compute column $\mathbf{t}^{(i)}$
13         parfor $i \leftarrow 1 : p$ do
14             Update column $\mathbf{y}^{(i)}$ of $Y_2$ using $\mathbf{t}$
15         Augment $Y$, $T$, and $V$
       // Update Phase
16     parfor $i \leftarrow 1 : t_u$ do
17         Update $A_{2,3}$ from the right and left
18         Compute block $Y_1$ of $Y$
19         Update $A_{1,2:3}$ from the right
```

2.1 Algorithmic Parameters

There are four families of tunable parameters in the NUMA-aware algorithm (see Table 1). There is one instance of each parameter *per iteration* of the algorithm, which means that there are $4N$ parameters to tune if there are N iterations. A complicating factor is that N in turn depends on the values chosen for the panel width parameters (b). Since our particular context (as a part of AED) implies that n might be relatively small compared to p, it may turn out to be sub-optimal to use all available cores, especially towards the end of the computation. The parameters t_r and t_u therefore specify the number of threads/cores ($\leq p$) to use in the reduction and update phases, respectively.

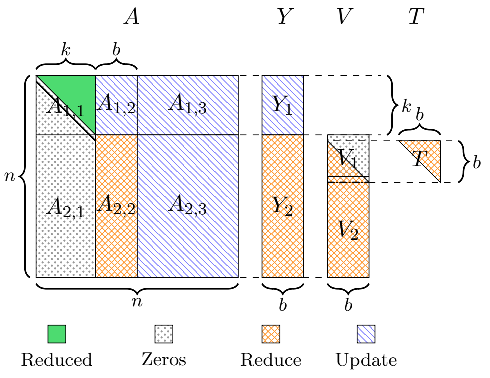


Figure 1: Partitioning of matrix A after reducing the first k columns, and Y and V will be used to reduce the panel A_{22}.

Table 1: The four families of algorithmic parameters.

Parameter name	*Type*	*Domain*	*Affected phases*
Panel width (b)	Integer	$\{1, \ldots, n-k\}$	Both
Parallelization strategy (s)	Category	{FULL, PARTIAL}	Reduction
No. of reduction threads (t_r)	Integer	$\{1, \ldots, p\}$	Reduction
No. of update threads (t_u)	Integer	$\{1, \ldots, p\}$	Update

3 Techniques Used within the Framework

At the heart of the framework is a *search module* (see Section 4.1 ahead), which abstracts any standard auto-tuning method behind a generic interface. The main aim of the framework is to extend the very limited capability of the tuning algorithm within the search module into a complete auto-tuner for the NUMA-aware algorithm. The framework achieves this by employing three specific techniques described in this section.

3.1 Decomposition into Independent Sub-Problems

Since the algorithm consists of an outer loop with non-overlapping iterations, it is reasonable to assume that parameters from different iterations are largely uncoupled. However, the four parameters within an iteration do strongly interact and must therefore be tuned together. This leads to the thought of decomposing the problem of tuning all $4N$ parameters at once into tuning N independent sets of 4 parameters. Yet, since the number of iterations, N, depends on one of the parameter families (the panel width) this idea cannot be directly applied.

By analyzing Algorithm 1 and Figure 1 it becomes clear that the shape of A at the start of an iteration depends only on n and k. We (logically) associate a sub-problem with each (valid) pair (n, k). The sub-problem for (n, k) is defined as finding optimal parameter settings for the four parameters in the upcoming iteration. But optimal in what sense? Minimizing the time will not work since the panel width affects both the amount of work and the progress made. Instead the objective function (for the sub-problem) is to maximize the performance

$$P = \frac{F_\mathrm{r} + F_\mathrm{u}}{T_\mathrm{r} + T_\mathrm{u}},$$

where F_r and F_u are the flop counts for the reduction and update phases, respectively, and T_r and T_u are the corresponding wall clock times.

We collect the values of the parameters for one sub-problem into a 4-tuple referred to as a *parameter tuple*. We arrange all the N parameter tuples as columns (from left to right) of a table referred to as a *parameter table*. The objective for the auto-tuner represented by the framework is to find a parameter table that minimizes the total execution time.

3.1.1 Concurrent Solution of Several Sub-Problems.

The size of a parameter table depends on the number of iterations, which in turn depends on the chosen panel widths. For an input matrix of fixed size n, there are as many as $n-2$ possible sub-problems (n, k) for $k = 0, 1, \ldots, n-3$. Any particular parameter table therefore consists of parameter tuples extracted from some subset of the sub-problems.

One full execution of the Hessenberg algorithm uses N parameter tuples provided by the framework and in turn feeds back time measurements used by the framework to make progress on N sub-problems. In other words, the framework can concurrently solve several sub-problems. But note, however, that exactly *which subset* of the $n-2$ sub-problems are relevant for a given execution depends on the chosen panel widths. See Figure 2 for an illustration of the relationships between sub-problems, parameter tuples, and parameter tables. The framework logically keeps track of $n-2$ partially solved sub-problems and after each particular execution of the Hessenberg algorithm is able to make progress on some subset of them.

3.2 Binning Similar Sub-Problems

Two distinct sub-problems (n, k) and (n', k') are similar if $n \approx n'$ and $k \approx k'$ simply because the shapes of all operands are similar. What this means is that we could (with some loss of accuracy) treat the two as one single sub-problem. This has several benefits. First, it reduces the total number of

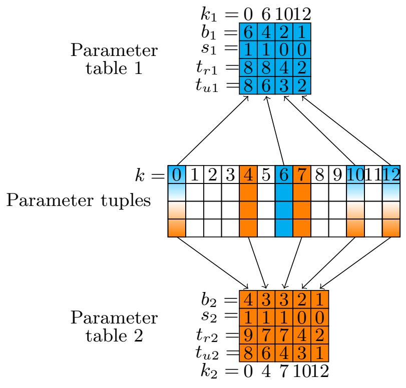


Figure 2: Two examples of parameter tables for $n = 15$.

sub-problems that need to be solved. Second, it allows the effort invested into making progress on one sub-problem to benefit also other (similar) sub-problems.

Specifically, we group adjacent sub-problems into bins and tune each bin as if it represents a single sub-problem. The bins are rectangular of size $\Delta_n \times \Delta_k$ as illustrated by the example in Figure 3 for $\Delta_n = 2$ and $\Delta_k = 3$. In particular, the sub-problems $(10, 4)$ and $(9, 6)$ belong to the same bin $(4, 2)$.

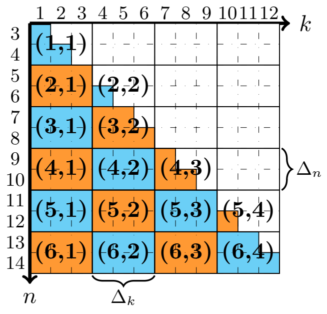


Figure 3: Binning of (12×12) space using bins of size (2×3).

3.3 Searching in Multiple Stages

Parameter tuples that yield good performance have a strong tendency (in this application) to cluster in one region of the search space. By performing the search in multiple stages, we can (potentially) more rapidly localize the search to this promising region. The idea is to start with a sparse but well distributed subset of the search space in the first stage of the search. Once (near-)convergence is reached, the search space is made denser and also restricted to a region around the converged point in subsequent stages.

For example, consider the two-stage search in Figure 4 which involves only t_r and t_u for simplicity. The goal is to optimize within the domain $\{1, \ldots, 10\}$. In the first stage, we choose the sparse but well distributed sub-domain $\{1, 4, 7, 10\}$ (large green dots). Suppose the search in the first stage converges to the point $(t_\mathrm{r}, t_\mathrm{u}) = (4, 7)$ (red cross). Then we include more points and restrict the search in the second stage to the sub-domain $\{2, 3, 4, 5, 6\}$ for t_r and $\{5, 6, 7, 8, 9\}$ for t_u.

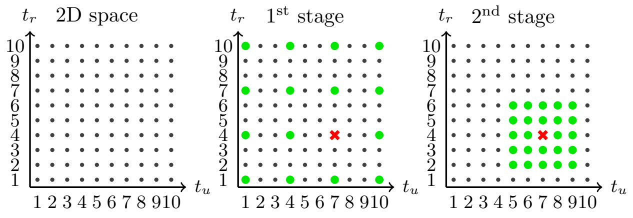


Figure 4: Two-stage search for 2D parameter space.

4 The Framework's Architecture

This section describes the software architecture of the framework. There are three modules: the Search Module, the Management Module, and the Database Module (see Figure 5).

4.1 The Search Module

The purpose of the Search Module is to encapsulate some standard auto-tuning method behind an abstract interface. The framework does not provide any implementation of this module by itself.

The Search Module has two primary functions: choose a parameter tuple for a given sub-problem and advance the search for a given sub-problem by one step in response to feedback. The module implementation itself is supposed to be state-less. A search state is instead encapsulated by the

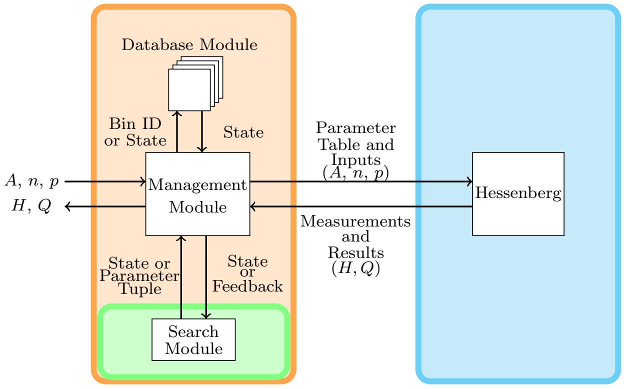


Figure 5: Modular diagram of the auto-tuning framework and the interface with the new Hessenberg reduction algorithm.

implementation into an opaque object[1] that is externally managed by the framework (see Sections 4.2 and 4.3 ahead). Since the specifics of what constitutes a "search state" depends entirely on the implementation of the Search Module, the framework views these objects as binary blobs[2] with no structure.

The Search Module exposes the following interface:

- Create–State: Creates a new state.
- Select–Parameters: Chooses a parameter tuple for the next iteration.
- Receive–Feedback: Receives feedback from the previous execution.
- Update–State: Performs one search step using previous feedback.
- Check–Convergence: Check if the search has converged.

4.2 The Management Module

The Management Module provides the glue that binds all the other modules together with the user input/output and the Hessenberg algorithm.

The core functions of the Management Module are as follows:

- Construct the next parameter table to use.
- Run the Hessenberg algorithm with the chosen parameter table.

[1]An object whose content and structure are not concretely known.
[2]Collection of data stored in binary as a single entry.

- Feed back measurements to the active sub-problems.

Construct parameter table. Starting from $k = 0$ and repeatedly calling the SELECT–PARAMETERS function of the Search Module (and updating $k \leftarrow k + b$ in between), a complete parameter table can be constructed column by column from left to right. The search state to use is either fetched from the Database Module or initialized using CREATE–STATE. Binning is applied before looking up a search state. The process of constructing a parameter table also implicitly selects the subset of *active sub-problems*, i.e., sub-problems which are going to be used in the next execution. So *before* calling SELECT–PARAMETERS, the function UPDATE–STATE is called on to make one step in the optimization algorithm (except initially when there is no feedback available). Furthermore, if the CHECK–CONVERGENCE function signals convergence, then the state is re-initialized with the search space used in the next stage of the multi-stage search.

Run the Hessenberg algorithm. The parameter table is passed alongside the other inputs to the Hessenberg algorithm. The computed matrices are output to the user.

Feed back measurements. The internal time measurements from each iteration are fed back to the active sub-problem search states using the RECEIVE–FEEDBACK function. The active states are kept in the Management Module until the measurements are fed back and afterwards they sent to the Database Module.

4.3 The Database Module

The Database Module stores the binary blobs representing the opaque search states. The search states are indexed by the bin coordinates (bin ID).

5 Experimental Results

The framework by itself cannot be meaningfully tested since it is dependent on an implementation of the Search Module. So in order to test the framework we implemented the Search Module using the *Nelder-Mead* algorithm [11]. This is neither the best nor the worst choice of algorithm. Ultimately the choice is not so important since the aim of this section is to show that the framework is able to make gradual improvements of the overall performance even though the actual optimization is only performed on small sub-problems. What the most effective implementation of the Search Module looks like is an open problem and something we do not contemplate in this paper.

In the experiments we used bins of size 10×10 and multi-stage search spaces as defined by Table 2, where b', t'_r, t'_u refer to the best values found in the first stage. Figure 6 shows the execution times (dots) of 500 executions for a matrix of order $n = 1000$. The curve shows a moving average of 50 consecutive measurements. The results indicate that in general the performance is indeed improving over time.

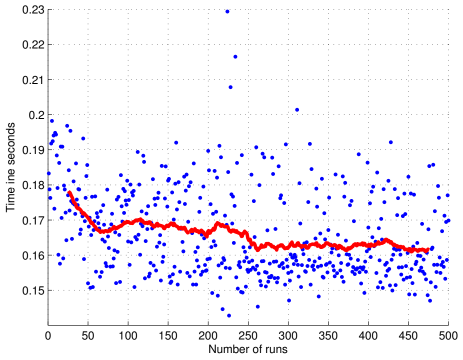


Figure 6: Execution time of 500 runs of the new Hessenberg reduction algorithm for $n = 1000$ using the framework. The red curve represents the moving average for a window of size 50.

Table 2: The search spaces used in multi-stage search.

Parameter symbol	*1st stage domain*	*2nd stage domain*
b	$10 : 10 : 100$	$b' - 9 : b' + 10$
s	{FULL, PARTIAL}	{FULL, PARTIAL}
t_r	$6 : 6 : 48$	$t'_r - 5 : t'_r + 6$
t_u	$6 : 6 : 48$	$t'_u - 5 : t'_u + 6$

6 Summary

In this paper we propose a modular auto-tuning framework that helps with tuning the parameters of a recently developed Hessenberg reduction algorithm. A brief description of the new algorithm and its parameters are presented. The algorithm's parameters interact with each other and span a huge search space which makes using generic tuning methods and tools like [1, 6, 8] not directly applicable. Such tools, despite been successfully used in solving other problems, can not deal with a problem which has a variable number of tunable parameters (like the one we have). Specially when this number depends on the value chosen for some of the tuned parameters them selves.

In contrast, the proposed framework facilitate that. The framework applies several techniques which allow searching the huge search space efficiently. Specifically, the framework decomposes the search space into smaller subspaces revealing standard auto-tuning sub-problems which can be tuned independently and concurrently. In addition, the framework groups similar sub-problems together in a single bin and tune them as one problem, which reduces the total number of sub-problems that need to be tuned, and propagate the progress made in tuning one sub-problem to other similar sub-problems. The framework also applies a multi-stage search, which, in one stage, allows for fast discovery of a promising region, where, in a later stage, the search is localized.

Besides solving the complex problem of the huge search space, the framework defines an abstract module with clear interface which can encapsulate any standard optimization methods or generic tuning tools, including [1, 6, 8], to expand its capabilities. This abstract module allows the experimentation with different tuning algorithms.

For testing the framework's ability to improve the overall performance of the new Hessenberg reduction algorithm, we used the Nelder-Mead algorithm in the search module. The results show that the performance of the new Hessenberg reduction algorithm is gradually improving over time.

Future work includes experimenting with both generic and specialized tuning algorithms in the search module and apply the idea underlying the framework to other linear algebra algorithms besides Hessenberg reduction.

Acknowledgements.

We thank the High Performance Computing Center North (HPC2N) at Umeå University for providing computational resources and valuable support during test and performance runs. 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.

References

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel*

Architectures and Compilation Techniques, Edmonton, Canada, August 2014.

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

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

[4] A. Castaldo and R. Whaley. Achieving Scalable Parallelization for the Hessenberg Factorization. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 65–73. IEEE, 2011.

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

[6] C. Ţăpuş, I. Chung, and J. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–11. IEEE Computer Society Press, 2002.

[7] 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.

[8] M. Gerndt, S. Benkner, E. César, C. Navarrete, E. Bajrovic, J. Dokulil, C. Guillén, R. Mijakovic, and A. Sikora. A Multi-Aspect Online Tuning Framework for HPC Applications. *Software Quality Journal*, May 2017.

[9] 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):29:1–29:23, October 2015.

[10] M. Hasan and R. 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.

[11] J. Nelder and R. Mead. A Simplex Method for Function Minimization. *The computer journal*, 7(4):308–313, 1965.

[12] 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.

[13] R. Schreiber and C. Van Loan. A Storage Efficient WY Representation for Products of Householder Transformations. Technical report, Cornell University, 1987.

Paper III

A Library for Storing and Manipulating Dense Tensors

Mahmoud Eljammaly and Lars Karlsson

Report UMINF 16.22, Department of Computing Science, Umeå University, Sweden, 2016.

A Library for Storing and Manipulating Dense Tensors[*]

Mahmoud Eljammaly
`mjammaly@cs.umu.se`

Lars Karlsson
`larsk@cs.umu.se`

Abstract

Aiming to build a layered infrastructure for high-performance dense tensor applications, we present a library, called `dten`, for storing and manipulating dense tensors. The library focuses on storing dense tensors in canonical storage formats and converting between storage formats in parallel. In addition, it supports tensor matricization in different ways. The library is general-purpose and provides a high degree of flexibility.

Keywords: Dense tensors, canonical storage format, tensor matricization, tensor storage format conversion, out-of-place conversion, in-place conversion.

1 Introduction

Tensors or multi-dimensional arrays are used in a diverse set of multi-dimensional data analysis applications. Many software products suitable for tensor computations exist, such as the commercial MATLAB suite enhanced by various open source third party toolboxes. Unlike for computations with matrices where there is a long history of community developed high-performance software libraries being widely used and incorporated into commercial software products, there is yet no analog for computations with tensors. Developing tensor computation algorithms and applications that are open source and independent of large commercial software environments is difficult in large parts due to a lack of open source software support for fundamental tensor operations.

Tensor algorithms and applications tend to either depend on proprietary functions provided by a large software environment or their own application-specific software solutions. Many parallels can be drawn with the early history of the field of matrix computations where every software included its

[*]Report UMINF 16.22, Dept. of Computing Science, Umeå University, Sweden.

own code for matrix–vector multiplication, scalar products, matrix transposition, and so on. The introduction and widespread adoption of core interfaces such as the BLAS [6, 9, 10, 11, 12, 13, 17, 18, 19] and LAPACK [2] has meant that software reliant on matrix computations have become easier to maintain and now exhibit portable performance. In contrast, the field of tensor computations, especially parallel and high-performance computations, has not yet matured to the point where a standard set of interfaces can be settled. The algorithms used are also much different in nature compared to those used for matrix computations, so it is not clear that the best approach is to mimic what has worked for matrices in the last couple of decades.

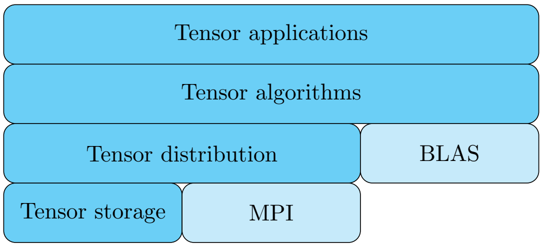


Figure 1: Software stack for tensor computation applications.

Learning from insights made for matrix computations, we nevertheless think that developing a software stack similar to the one depicted in Figure 1 would be a good starting point. The proposed stack consists of two established components: the Basic Linear Algebra Subprograms (BLAS) for high-performance fundamental matrix operations and the Message Passing Interface (MPI) for communication between nodes in a distributed memory system. Many fundamental tensor operations can be expressed largely in terms of the BLAS, and MPI is available on virtually any distributed memory system, so there is little doubt that these components will be a part of a future tensor computations software stack. In addition, there are four tensor-specific components in the stack. From the top down: the *tensor applications* component consists of complete applications that use large-scale tensor computations, the *tensor algorithms* component consists of numerical tensor algorithms such as tensor decomposition algorithms and tensor contraction, the *tensor distribution* component consists of such things as communication and (re)distribution of tensors, and finally the *tensor storage* component manages the local storage and manipulation of (sub)tensors on each node in a distributed memory system.

The two components at the top (applications and algorithms) are large and multi-faceted with new algorithms and applications being added as time goes by. But the other two components (distribution and storage) are more fundamental in nature and bounded in scope. We propose to start from the bottom up in an effort to realize a first seed for a tensor computations software stack.

This paper focuses on the *tensor storage* component (see Figure 1) and more specifically on the storage and manipulation of *dense* tensors, i.e., tensors whose elements are mostly non-zero. The sister problem of storing and manipulating *sparse* tensors has been recently addressed by Dahlberg, see [8] and the references within.

Some of the main points of this paper are:

1. Any one-mode or multi-mode tensor matricization is equivalent to converting the storage format of the tensor from one canonical format to another.

2. A tensor stored in a canonical tensor storage format can be interpreted as a matricization of that tensor stored in a canonical matrix storage format.

3. Any tensor storage format conversion can be performed either out-of-place by copying or in-place by in-place permutation.

4. The performance of matricization depends on whether the resulting matrix should be stored in column-major or row-major format.

5. If the ordering of the rows and columns in a matricized tensor is not important, then there exists an efficient way to matricize the tensor.

The rest of the paper is organized as follows. Section 2 gives a mathematical background of tensor storage formats. Tensor storage formats are defined and their relation to matrix storage formats is explained. In Section 3 we present the algorithms used in this paper. Different storage format conversion techniques are discussed in addition to the potential for parallelism. Section 4 provides details about the implementation and introduces the library. Section 5 presents experiments that demonstrate the performance and scalability of the library. Finally, in Section 6 some conclusions and related work are described.

1.1 Notation and terminology

Zero-based indexing is used in order to simplify many of the formulas. We denote by $S(n)$ the set $\{0, 1, \ldots, n-1\}$.

A *sequence* is denoted by angle brackets, e.g., $\langle 0, 1, 2\rangle$, and as a symbol we use a bold lower case letter. The notation $|\cdot|$ denotes the length of a sequence, i.e, the number of elements. The *concatenation* of two sequences $\mathbf{a}$ and $\mathbf{b}$ is denoted by $\mathbf{a} \oplus \mathbf{b}$, e.g., $\langle 0, 1\rangle \oplus \langle 2, 3\rangle = \langle 0, 1, 2, 3\rangle$. A *subsequence* is obtained by deleting zero or more elements from a sequence. An *extraction* of a sequence is denoted by σ and returns a permuted subsequence. An extraction is defined by a sequence of indices, e.g., $\sigma = \langle 2, 1\rangle$, that specify which elements to extract and in which order. The application of σ to a sequence $\mathbf{a}$ is denoted by $\sigma\mathbf{a}$. For example, applying the extraction $\sigma = \langle 2, 1\rangle$ to the sequence $\mathbf{a} = \langle a_0, a_1, a_2, a_3\rangle$ results in the permuted subsequence $\langle a_2, a_1\rangle$. Any extraction that selects the entire sequence degenerates into a permutation and is then denoted by π.

A *tensor* $\mathcal{A}$ of *order* d and *size* $n_0 \times n_1 \times \cdots \times n_{d-1}$ is a d-dimensional array. The size of $\mathcal{A}$ is denoted by the sequence $\mathbf{n} = \langle n_0, n_1, \ldots, n_{d-1}\rangle$. Each *element* of $\mathcal{A}$ is identified by a unique *index sequence* $\mathbf{k} = \langle k_0, k_1, \ldots, k_{d-1}\rangle$ where $k_i \in S(n_i)$ is the *index in mode* i. The $\mathbf{k}$'th element of $\mathcal{A}$ is denoted by $\mathcal{A}(\mathbf{k})$.

2 Canonical tensor storage formats

2.1 Definition

A *storage format* for a dense tensor $\mathcal{A}$ of order d and size $\mathbf{n}$ is a one-to-one mapping of the index sequence $\mathbf{k}$ to the set of integers $S(N)$, where $N = \prod_{i=0}^{d-1} n_i$ denotes the total number of elements. Formally, a tensor storage format is a bijective function parameterized by the size $\mathbf{n}$, i.e.,

$$\phi : S(n_0) \times \cdots \times S(n_{d-1}) \to S(N).$$

The function ϕ maps each index sequence $\mathbf{k}$ to a unique offset in a contiguous memory area of N memory locations.

There are many potential tensor storage formats that fit this definition, but only a few are interesting in practice. A particularly simple and useful tensor storage format is obtained by defining

$$\phi(\mathbf{k}; \mathbf{n}) = \sum_{i=0}^{d-1} k_i \prod_{j=0}^{i-1} n_j. \tag{1}$$

Here, $\mathbf{k}$ is the argument and $\mathbf{n}$ is the parameter to the function. For example, given tensor of size $\mathbf{n} = \langle n_0, n_1, n_2\rangle$, the function ϕ will map the index sequence $\mathbf{k} = \langle k_0, k_1, k_2\rangle$ to a continues memory area of size $N = n_0 n_1 n_2$ such that

$$\phi(\mathbf{k}; \mathbf{n}) = k_0 + k_1 n_0 + k_2 n_0 n_1.$$

A more general class of storage formats is obtained by permuting the index sequence $\mathbf{k}$ (and size $\mathbf{n}$) before applying ϕ. Formally, let π be any permutation of a sequence of length d. Then the mapping ϕ_π defined by

$$\phi_\pi(\mathbf{k}; \mathbf{n}) = \phi(\pi\mathbf{k}; \pi\mathbf{n}) \tag{2}$$

is also a valid tensor storage format. Since there are $d!$ permutations of a sequence of length d, there are $d!$ different storage formats of this type. These formats are known as the *canonical (dense) tensor storage formats* and are the focus of this paper.

The concept of a canonical tensor storage format generalizes the row- and column-major storage formats used for matrices. To see this, note that a matrix is a tensor of order $d = 2$ and size $\mathbf{n} = \langle n_0, n_1 \rangle$, where n_0 is the number of rows and n_1 the number of columns. Similarly, an index sequence of the matrix takes the form $\mathbf{k} = \langle k_0, k_1 \rangle$, where k_0 is the row index and k_1 the column index. Choosing the permutation $\pi = \langle 0, 1 \rangle$ in (2) gives the *column-major* matrix storage format, as can be seen by

$$\phi_{\langle 0,1 \rangle}(\mathbf{k}; \mathbf{n}) = \phi(\langle k_0, k_1 \rangle, \langle n_0, n_1 \rangle) = k_0 + k_1 n_0,$$

which we recognize as the column-major ordering. Conversely, choosing $\pi = \langle 1, 0 \rangle$ in (2) gives the *row-major* matrix storage format:

$$\phi_{\langle 1,0 \rangle}(\mathbf{k}; \mathbf{n}) = \phi(\langle k_1, k_0 \rangle, \langle n_1, n_0 \rangle) = k_0 n_1 + k_1.$$

In other words, applying the permutation π to the index sequence $\mathbf{k}$ decides which index of the two will vary the fastest as one scans the memory; $\pi = \langle 0, 1 \rangle$ means that index k_0 will vary faster than index k_1 and $\pi = \langle 1, 0 \rangle$ means that index k_1 will vary faster than index k_0. Generally, the leftmost element in a permutation is the fastest varying one and the rightmost element in a permutation is the most slowly varying one.

2.2 Matricization

A tensor of order d can be reshaped into a matrix, an operation that goes by many names in the literature (e.g., *matricization*, *unfolding*, or *flattening*). We prefer to use the term matricization in this paper. To view a tensor as a matrix, the modes of the tensor need to be partitioned into two disjoint subsets: one subset for the columns of the matrix and another subset for the rows of the matrix. For example, consider a tensor $\mathcal{A}$ of order $d = 4$ and choose the mode subset $\{0, 1\}$ for the rows of the matrix and the complementary subset $\{2, 3\}$ for the columns. The size of the resulting matrix is $\hat{n}_0 \times \hat{n}_1$, where $\hat{n}_0 = n_0 n_1$ and $\hat{n}_1 = n_2 n_3$. To map a tensor index sequence to a matrix index sequence, we need to define two mappings of the form (2) by specifying an extraction σ_{row} for the row dimension and another extraction σ_{col} for the column dimension. For example, we can define $\sigma_{\text{row}} = \langle 0, 1 \rangle$ and $\sigma_{\text{col}} = \langle 2, 3 \rangle$ to obtain the following translation from the tensor index sequence $\mathbf{k} = \langle k_0, k_1, k_2, k_3 \rangle$ to the matrix index sequence $\hat{\mathbf{k}} = \langle \hat{k}_0, \hat{k}_1 \rangle$:

$$\langle k_0, k_1, k_2, k_3 \rangle \mapsto \langle \phi_{\sigma_{\text{row}}}(\mathbf{k}; \mathbf{n}), \phi_{\sigma_{\text{col}}}(\mathbf{k}; \mathbf{n}) \rangle = \langle k_0 + k_1 n_0, k_2 + k_3 n_2 \rangle = \langle \hat{k}_0, \hat{k}_1 \rangle.$$

The matricization defined in this way is denoted by $A_{\sigma_{\text{row}}, \sigma_{\text{col}}}$.

2.3 Matricization and storage format conversion

It turns out that storing a matricized tensor in either the column- or the row-major storage format is equivalent to storing the tensor itself in a canonical tensor storage format of the form (2) for some permutation π. For example, storing the matricization $A_{\langle 0,1\rangle,\langle 2,3\rangle}$ in the column-major storage format is equivalent to storing the tensor as in (2) with $\pi = \langle 0,1,2,3\rangle = \langle 0,1\rangle \oplus \langle 2,3\rangle$. To see this, note that

$$\phi_{\langle 0,1,2,3\rangle}(\mathbf{k};\mathbf{n}) = \underbrace{\phi_{\langle 0,1\rangle}(\mathbf{k};\mathbf{n})}_{\hat{k}_0} + \underbrace{\phi_{\langle 2,3\rangle}(\mathbf{k};\mathbf{n})}_{\hat{k}_1} \cdot \underbrace{n_0 n_1}_{\hat{n}_0} = \hat{k}_0 + \hat{k}_1 \hat{n}_0,$$

which we recognize as the column-major ordering of $A_{\langle 0,1\rangle,\langle 2,3\rangle}$. Similarly, storing the matricization in the row-major storage format is equivalent to choosing $\pi = \langle 2,3,0,1\rangle = \langle 2,3\rangle \oplus \langle 0,1\rangle$ in (2).

In general, consider a tensor of order d stored in a canonical format defined by the permutation π and a general matricization of this tensor defined by the extractions σ_{row} for the row dimension and σ_{col} for the column dimension. If $\pi = \sigma_{\mathrm{row}} \oplus \sigma_{\mathrm{col}}$, then the storage mapping (2) for the tensor can be rewritten in the form

$$\phi_\pi(\mathbf{k};\mathbf{n}) = \phi_{\sigma_{\mathrm{row}}}(\mathbf{k};\mathbf{n}) + \phi_{\sigma_{\mathrm{col}}}(\mathbf{k};\mathbf{n}) \cdot \prod_{i \in \sigma_{\mathrm{row}}} n_i,$$

which means that the memory used to store the tensor can be reinterpreted as the matricization $A_{\sigma_{\mathrm{row}},\sigma_{\mathrm{col}}}$ stored in the column-major format. Similarly, if $\pi = \sigma_{\mathrm{col}} \oplus \sigma_{\mathrm{row}}$, then the same holds but with the matricization stored in the row-major format.

3 Tensor storage format conversion

Given a tensor stored in the format defined by π_{in} and a target format defined by π_{out}, the problem of tensor storage format conversion consists of permuting the tensor elements in memory such that the storage format changes from π_{in} to π_{out}. There are two main types of conversions: out-of-place (OOP) conversion involves the explicit copying of the tensor elements to a separate memory area, but in-place (IP) conversion changes the format by overwriting the old memory area and uses only a small constant amount of additional memory.

The mapping from input memory location ℓ_{in} to output memory location ℓ_{out} is a bijective function $f : S(N) \to S(N)$ and is defined by

$$f(\ell_{\mathrm{in}};\mathbf{n}) = \phi_{\pi_{\mathrm{out}}}(\phi^{-1}_{\pi_{\mathrm{in}}}(\ell_{\mathrm{in}};\mathbf{n});\mathbf{n}) = \ell_{\mathrm{out}}. \tag{3}$$

The mapping consist of two steps: first $\phi^{-1}_{\pi_{in}}$ maps the input memory location ℓ_{in} to the corresponding index sequence $\mathbf{k}$ and then $\phi_{\pi_{out}}$ maps this index sequence to the output memory location ℓ_{out}. The function f determines the memory transfer pattern and is completely determined by the size $\mathbf{n}$ and the input and output formats π_{in} and π_{out}.

3.1 Efficient conversion by moving blocks of memory

For many pairs of formats, the memory transfers implied by f can be arranged into a set of efficient copies of contiguous blocks of memory. Exploiting this feature of the problem whenever possible is important to obtain high performance in the conversion process, since it will benefit from the memory hierarchy and require fewer evaluations of f. In addition, the block transfers is a source of parallelism since different blocks can be transferred at the same time to speed up the process.

To see where the blocks come from and how big they are, suppose that the formats π_{in} and π_{out} have a common prefix σ_{pre} of length m. In other words, it is possible to write $\pi_{\text{in}} = \sigma_{\text{pre}} \oplus \sigma_{\text{in}}$ and $\pi_{\text{out}} = \sigma_{\text{pre}} \oplus \sigma_{\text{out}}$ for some σ_{in} and σ_{out}. Consider the set of elements in the input tensor for which $\sigma_{\text{in}}\mathbf{k}$ are the same but the indices in $\sigma_{\text{pre}}\mathbf{k}$ vary. This set consists of $\prod_{i\in\sigma_{\text{pre}}} n_i$ elements and is stored *contiguously in memory* due to the structure of (2) since the indices in σ_{pre} vary faster than the fastest varying index in σ_{in}. The same holds for the output tensor, and the relative order of the elements in each such block is preserved. Hence, the storage format conversion can be carried out using $\prod_{i\notin\sigma_{\text{pre}}} n_i$ block memory transfers of size $\prod_{i\in\sigma_{\text{pre}}} n_i$.

3.2 Out-of-place versus in-place conversion

The OOP conversion technique involves creating a second tensor and copying each block to its new location in the output tensor. In contrast, the IP conversion technique involves permuting the blocks inside the original memory area, thereby using roughly half of the memory required by the OOP conversion technique. The block transfers form cycles (see Section 3.3) where the blocks in a cycle are shifted within the cycle, see [14] and the references within.

Figure 2 illustrates both the OOP and the IP conversion techniques. The figure shows a tensor of order $d = 4$ and size $\mathbf{n} = \langle 5, 3, 2, 4\rangle$. The conversion changes the tensor storage format from $\pi_{\text{in}} = \langle 0, 1, 2, 3\rangle$ to $\pi_{\text{out}} = \langle 0, 3, 2, 1\rangle$. The in-place conversion consist of six cycles of which two are singleton cycles (i.e., containing only one block). Each cycle is shown in the figure with a unique color, while the singleton cycles share the same color (red).

3.3 In-place conversion techniques

The in-place conversion technique moves tensor blocks one by one from its initial position to its final position within the same memory area. To avoid overwriting the already occupied destination, that block must in turn first be

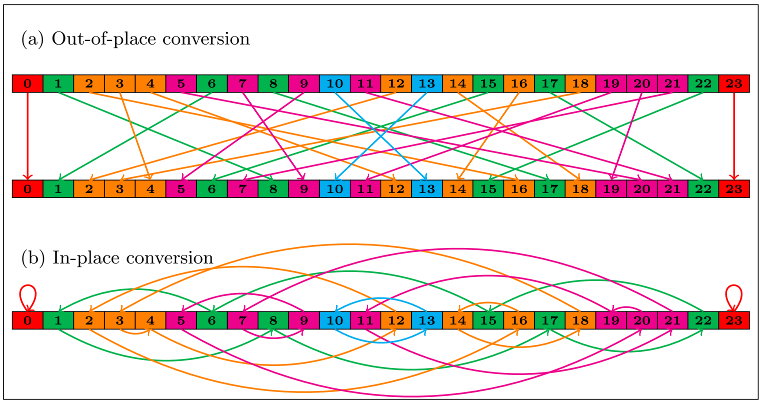


Figure 2: Illustration of the out-of-place and in-place tensor storage format conversion techniques for a tensor of size $5 \times 3 \times 2 \times 4$.

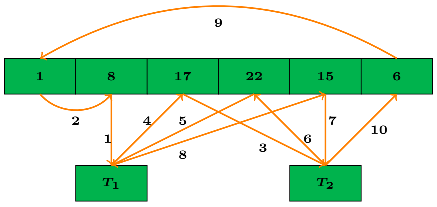
moved to its final position. This continues until a block is encountered whose final position is the initial position of the first block moved. This completes a cycle and the whole permutation consists of one or more such cycles of potentially diverse lengths. Some cycles involve moving/shifting only one block, which actually keeps the block in the same position and involves no data movement at all. Such degenerate cycles are called *singleton cycles* and for any permutation resulting from the mapping function f with at least two blocks there are at least two singletons. See Figure 2 for an example.

3.3.1 Forward versus backward cycle shifting

Different techniques can be used in in-place conversion to make it more efficient [14]. Specifically, a cycle can be shifted in one of two ways: forward or backward.

In the forward shifting technique, the destination block is first moved to a temporary memory area and then the source block is moved to the now vacant destination block. If the number of blocks in the cycle is at least four, then we need two temporary storage areas, each the size of one block. Figure 3(a) shows the forward shifting technique applied to a cycle consisting of six blocks. We start with block number 1 and its destination is block number 8, so we move block number 8 to the temporary storage area T_1. Then we move block number 1 to block number 8. Now to move block number 8 to its destination block number 17, we need to first save block number 17 to the second temporary area T_2. This process continues until we reach the block whose destination is the first block in the cycle (block number 1). For non-singleton cycles with $b > 2$ blocks, the forward shifting

(a) Forward shifting



(b) Backward shifting

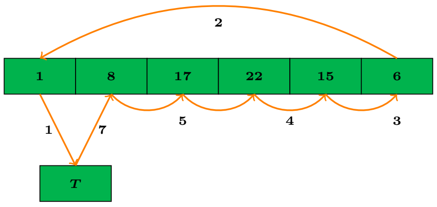


Figure 3: Illustration of forward and backward shifting for a cycle with six blocks. Numbers inside the blocks represent the block's position in the tensor (T is a temporary workspace block) while numbers on arrows represent the ordering of the steps.

technique requires $2(b-1)$ block memory transfers. In our case, the cycle has $b = 6$ blocks and we need $2(6-1) = 10$ steps.

In the backward shifting technique, we begin by moving the first block to a temporary storage area T. Then we loop backward in the cycle to the block whose destination is the block we just copied. We move that block (in this case block number 6) and repeat the procedure until we have traversed the entire cycle. We end by transferring the initial block from the temporary storage area to its now vacant destination. The number of steps required by the backward shifting technique is only $b+1$. In our case, we need $b+1 = 7$ steps.

In conclusion we prefer the backward shifting technique not only because it uses fewer steps to shift a cycle, but also because it uses the source block in one step as the destination block in the next. If the block fits into the cache, then the data will be reused.

3.3.2 Sub-blocking

If the blocks become too large to fit in the lowest level cache, then the nice cache effects inherent in the backward shifting technique do not apply. There is a simple scheme called *sub-blocking* that can be used to overcome this issue and retain the beneficial cache behavior. The sub-blocking scheme works by partitioning each block into smaller sub-blocks that fit inside some desired level of the cache hierarchy. The backward cycle shifting is replaced by several rounds of backward cycle shifting: one round for each sub-block. Figure 4 illustrates the sub-blocking scheme with three sub-blocks per block (and hence three rounds of cycle shifting per cycle). The figure shows the sub-blocking of a cycle consisting of six blocks into three sub-blocks per block. The first round moves the dark green blocks (subscript "1"). The second round moves the green blocks (subscript "2"). Finally, the third round moves the light green blocks (subscript "3").

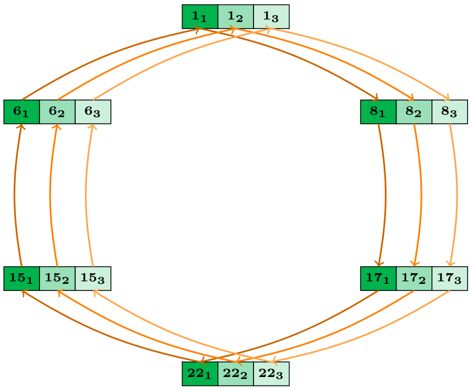


Figure 4: Illustration of the sub-blocking scheme to improve the cache behavior of the backward cycle shifting technique in in-place tensor storage format conversion. Each of the six blocks in the cycle have been partitioned into three sub-blocks. The numbers inside the blocks represent the block number in the tensor while the subscript numbers represent the sub-block number of that block.

3.4 Parallel conversion: Sources of concurrency

Both out-of-place and in-place conversion allow for parallel processing, but the former has a higher degree of (inherent) parallelism. In out-of-place conversion, parallelism is available when moving blocks since every block can be simultaneously copied to their respective destinations. In in-place conversion, the situation is quite different since the same memory area is used for the input and the output and the blocks can therefore not be moved simultaneously. But there is still potential for parallelism. Dependencies between blocks exist only within a given cycle. Two different cycles can be shifted at the same time and hence the cycles are a source of concurrency.

To exploit the available parallelism efficiently, we need to take care of the load balance since the cycles do not all have the same length in general. Distributing the blocks evenly over the processors increases the scalability of the out-of-place conversion. In the in-place case, the aim is to distribute the total work evenly over the processors. Such a balanced load can in many cases be well approximated by using a dynamic load balancing scheme.

3.5 Matricization by storage format conversion

As described in Section 2.3, obtaining an explicit matricization is equivalent to converting the tensor storage format. The input is a tensor stored in a specific format defined by the permutation π_{in} and a subset $M \subseteq S(d)$ of the modes to associate with the columns of the resulting matrix. The output is the same tensor but stored in a format defined by some permutation π_{out} such that the stored tensor can be reinterpreted as a matricization of the tensor in either the row-major or the column-major format with the modes in M associated with the columns of the matrix. Figure 5 illustrates for a third-order tensor that the choice of target matrix format can drastically change the cost of the resulting tensor storage format conversion. In this example, choosing the row-major storage format will result in no change of the input tensor and is therefore entirely free. On the other hand, choosing the column-major storage format will result in a costly conversion with blocks of size one, which is the worst possible case.

Since the precise ordering of the rows and columns of the resulting matrix is seldom important in applications, there are many candidate formats π_{out}. Specifically, the matrix may be stored in either the row-major or the column-major format and the modes associated with the rows and columns may be arbitrarily ordered. The choice of output format affects the performance of the conversion process primarily because it determines the block size of the conversion as described in Section 3.1.

For the column dimension of the matrix, we need to choose an extraction σ_{col} of $S(d)$ of length $|M|$, and for the row dimension we need a complementary extraction σ_{row} of $S(d)$ of length $|S(d) \setminus M|$. In addition, we need to choose the target matrix format: row- or column-major. The output format π_{out} is determined by these three choices as follows: if the target is the column-major format, then $\pi_{\text{out}} = \sigma_{\text{row}} \oplus \sigma_{\text{col}}$, otherwise $\pi_{\text{out}} = \sigma_{\text{col}} \oplus \sigma_{\text{row}}$.

To maximize the block size in the conversion process, we need to maximize the length of the common prefix of the given π_{in} and the chosen π_{out} subject to its constraints. The choice of target matrix format is governed only by the first component of π_{in}. If that component is in M and hence associated with the columns and a member of σ_{col}, the only way to get a block size greater than one is to choose the row-major format since that places σ_{col} first in π_{out}. Conversely, if the component is in $S(d) \setminus M$, the only reasonable choice is the column-major format. With the target matrix

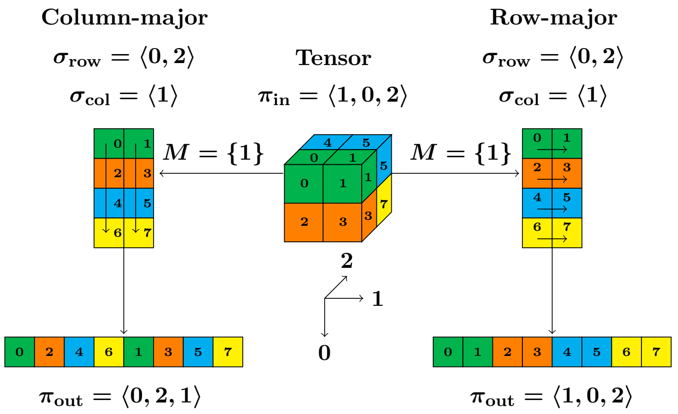
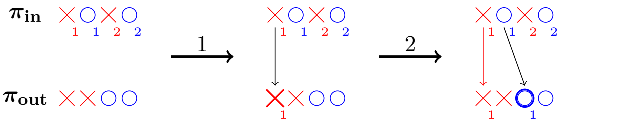

Column-major
$\sigma_{\text{row}} = \langle 0, 2 \rangle$
$\sigma_{\text{col}} = \langle 1 \rangle$
Tensor
$\pi_{\text{in}} = \langle 1, 0, 2 \rangle$
Row-major
$\sigma_{\text{row}} = \langle 0, 2 \rangle$
$\sigma_{\text{col}} = \langle 1 \rangle$
$M = \{1\}$
$M = \{1\}$
0 1 2 3 4 5 6 7
2
1
0
0 2 4 6 1 3 5 7
0 1 2 3 4 5 6 7
$\pi_{\text{out}} = \langle 0, 2, 1 \rangle$
$\pi_{\text{out}} = \langle 1, 0, 2 \rangle$

Figure 5: Tensor matricization can be performed in different ways depending on the choice of target matrix format. On the left, the matricization results in a matrix in column-major storage format but requires a very expensive tensor storage format conversion with blocks of size one. On the right, the same matricization results in a matrix in row-major storage format and requires no memory transfers at all.

format fixed, the remaining components of σ_{row} and σ_{col} need to be chosen such that the length of the common prefix of π_{in} and π_{out} is maximized. For example, if $\pi_{\text{in}} = \langle 0, 1, 2, 3 \rangle$ and $M = \{1, 3\}$, then by the reasoning above we should choose the column-major format (since $0 \notin M$) and place 0 first in σ_{row}. Since the next component is in M and hence not in σ_{row}, there is no way to create a match between the second components of π_{in} and π_{out}. There are in this particular case two solutions with the same block size n_0: $\pi_{\text{out}} = \langle 0, 2, 1, 3 \rangle$ and $\pi_{\text{out}} = \langle 0, 2, 3, 1 \rangle$.

To help visualize the matricization process, we represented π_{in} as a number of $\circ$'s and $\times$'s, Figure 6 (a). The $\circ$ denotes a component in M and $\times$ denotes a component in $S(d) \setminus M$. The subscript numbers represent the order of the components. For example, $\times_2$ is the second component in π_{in} from $S(d) \setminus M$ and $\circ_1$ is the first component in π_{in} from M. π_{out} is also represented as a number of $\circ$'s and $\times$'s but initially without subscript numbers, Figure 6 (a). The goal is to map π_{in} $\times$'s to π_{out} $\times$'s and π_{in} $\circ$'s to π_{out} $\circ$'s such that the block size is maximized. The mapping of a component from π_{in} to π_{out} is represented as an arrow. The arrow points to the component in π_{out} that will be numbered, Figure 6 (b,c). The length of the common prefix is defined by the number of leading contiguous vertical arrows between π_{in} and π_{out}. In other words, the prefix size is captured by the components coming before the first tilted arrow.



(a) Start (b) $\pi_{out} = \langle 0, ?, ?, ? \rangle$ (c) $\pi_{out} = \langle 0, ?, 1, ? \rangle$

Figure 6: Matricization algorithm: block size fixed after mapping 2 components.

Using this notation, Figure 6 represents the matricization in the previous example. Since the target format is column-major format, M components come last in π_{out}. In the first step, we map the first component in π_{in}, which is $\times_1$, 0 in the example. In this case the arrow is vertical. In the next step, we map the next component in π_{in}, which is $\circ_1$. The mapping arrow is tilted in this case, which means that the block size of this matricization is now known to be n_0.

Algorithm 1 formalizes the steps required to build π_{out} in a manner that maximizes the block size.

We start with an empty $\sigma_\times$ and $\sigma_\circ$. We check the first component in π_{in}, if it is from M we append it to $\sigma_\circ$ otherwise we append it to $\sigma_\times$. Then, we move to the next component in π_{in}. We keep doing that until we have mapped all the components in π_{in}.

After the mapping, we decide the order of $\sigma_\times$ and $\sigma_\circ$ in π_{out} based on $\pi_{\text{in}}(0)$: if it's from M then $\pi_{\text{out}} = \sigma_\circ \oplus \sigma_\times$ otherwise $\pi_{\text{out}} = \sigma_\times \oplus \sigma_\circ$.

3.6 Matricizing two tensors

A common operation in tensor computation is the process of combining a subset of indices from one tensor with a subset of indices from another tensor. This is called *tensor contraction*. A contraction can be performed using matrix-matrix multiplication. In this case, the two tensors need to be converted to matrices. We matricize each tensor over its contraction subset then multiply the two matrices.

Given a tensor $\mathcal{A}$ of size $\mathbf{n}_A$ stored in the format defined by π_{in}^A, a tensor $\mathcal{B}$ of size $\mathbf{n}_B$ stored in the format defined by π_{in}^B, a subset M_A of the modes of $\mathcal{A}$, and a corresponding subset M_B of the modes of $\mathcal{B}$. We must decide on a format π_{out}^A for $\mathcal{A}$ and a *compatible* format π_{out}^B for $\mathcal{B}$. The formats are compatible if and only if the modes in M_A occur in π_{out}^A in the exact same order as their corresponding elements in M_B occur in π_{out}^B.

The constraint on the ordering of the elements of M_A and M_B in the chosen formats implies that, at least in general, we cannot find optimal formats for $\mathcal{A}$ and $\mathcal{B}$ independently. In other words, we seek an algorithm

```
Input  : $\pi_{in}$ // Input storage format.
       : $M$ // Set of matricization modes.
Output : $\pi_{out}$ // Output storage format.
       : $\gamma$ // Block size.
       : $\sigma_\circ$ // Extraction specifying the order of the
         matricization modes.
begin
    $\sigma_\times \leftarrow \langle\rangle$
    $\sigma_\circ \leftarrow \langle\rangle$
    for $i = 0$ to $d - 1$ do
        if $\pi_{in}(i) \in M$ then
            $\sigma_\circ \leftarrow \sigma_\circ \oplus \pi_{in}(i)$
        else
            $\sigma_\times \leftarrow \sigma_\times \oplus \pi_{in}(i)$
        end
    end
    if $\pi_{in}(0) \in M$ then
        $\pi_{out} \leftarrow \sigma_\circ \oplus \sigma_\times$
    else
        $\pi_{out} \leftarrow \sigma_\times \oplus \sigma_\circ$
    end
    Compute the block size $\gamma$
    return $\pi_{out}, \gamma, \sigma_\circ$
end
```

Algorithm 1: Matricization algorithm.

for finding an optimal pair of formats.

Given two sets of compatible formats, which is better? When matricizing a single tensor, we assumed that a bigger block size implies a faster conversion, see Section 3.5. When matricizing a pair of tensors, we have two block sizes. Let $h(\alpha, \beta)$ be some measure of the execution rate of converting $\mathcal{A}$ with block size α and $\mathcal{B}$ with block size β. We can reasonably assume that h is non-decreasing in each parameter, i.e., that $h(\alpha + \Delta, \beta) \geq h(\alpha, \beta)$ and $h(\alpha, \beta + \Delta) \geq h(\alpha, \beta)$ for every $\Delta > 0$ because increasing only one of the block sizes improves the execution rate of that conversion while the other is unaffected. See Section 4.4 for more details about h.

Consider how Algorithm 1 builds an optimal output format. It starts off with an empty output and a block size of 1. In each iteration, one additional element of the output format is determined. The block size will increase in the first few iterations until the maximum block size is reached. After that point, the block size will remain constant as the missing elements of the output format are determined. Crucially, the leading elements (which determine the block size) are uniquely determined. This means that even though there could be several output formats that have the same optimal block size, they will all have the same *prefix* that determines the optimal block size.

Now consider running Algorithm 1 on both $\mathcal{A}$ and $\mathcal{B}$ at the same time. In general, the two outputs will not be compatible. This happens if the two executions produce a different ordering for the elements in M_A and M_B. Instead, start the two executions and run them until the first iteration that wants to map an element in M_A respectively M_B. Pause both executions. At this point, the two partially defined formats are compatible. From this point forward there are two cases, Figure 7. In the best case, the two elements that are about to be mapped are in correspondence with one another, Figure 7(a). In this case, we simply advance both executions to the next iteration; both block sizes will be increased and the formats will remain compatible. In the worst case, the two elements are not in correspondence with one another, Figure 7(b). In that case, we must accept that one of the block sizes need to be fixed in order to preserve compatibility. There are again two cases: we fix the block size of either $\mathcal{A}$ or $\mathcal{B}$. Let α and β denote the block sizes obtained thus far by Algorithm 1. If we fix the block size α, then we can proceed with the execution on $\mathcal{B}$ and obtain the block size $\beta + \Delta_B$ for $\mathcal{B}$. On the other hand, if we fix the block size β, then we can proceed with the execution on $\mathcal{A}$ and obtain the block size $\alpha + \Delta_A$ for $\mathcal{A}$. Which is better depends on h: if $h(\alpha + \Delta_A, \beta) < h(\alpha, \beta + \Delta_B)$, then we should fix α and otherwise we should fix β.

In conclusion, to compute an optimal pair of formats we use the following procedure:

1. Run Algorithm 1 on $\mathcal{A}$. Let α_1 denote the resulting block size.

2. Run Algorithm 2, a modified version of Algorithm 1, on $\mathcal{B}$ that produces a format compatible with that produced in Step 1. Let β_1 denote the resulting block size.

3. Run Algorithm 1 on $\mathcal{B}$. Let β_2 denote the resulting block size.

4. Run Algorithm 2 on $\mathcal{A}$ to ensure compatibility with the format produced in Step 3. Let α_2 denote the resulting block size.

5. If $h(\alpha_1, \beta_1) \geq h(\alpha_2, \beta_2)$ then use the formats from Steps 1 and 2 and otherwise use the formats from Steps 3 and 4.

Algorithm 3 formalizes the steps required to build π_{out}^{A} and π_{out}^{B} in a manner that maximizes the execution rate.

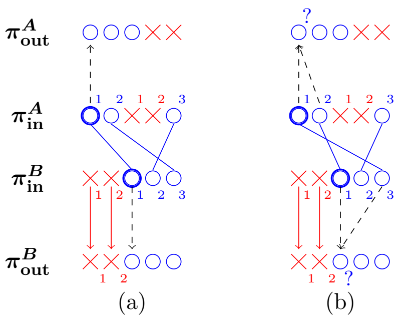


Figure 7: Different scenarios for mapping an element from M.

```
Input   : $\pi_{in}$ // Input storage format for tensor 2.
        : $M_1$ // Set of matricization modes of tensor 1.
        : $M_2$ // Set of matricization modes of tensor 2.
        : $\sigma_\circ^1$ // Extraction specifying the order of tensor 1
    matricization modes.
Output: $\pi_{out}$ // Output storage format for tensor 2.
        : $\gamma$ // Block Size for tensor 2.
1  begin
2      $\sigma_\times \leftarrow \langle\rangle$
3      Using $\sigma_\circ^1$ define an extraction $\sigma_\circ$ that orders the elements of $M_2$
        which keeps the correspondence between $M_1$ and $M_2$
4      for $i = 0$ to $d - 1$ do
5          if $\pi_{in}(i) \notin M_2$ then
6              $\sigma_\times \leftarrow \sigma_\times \oplus \pi_{in}(i)$
7          end
8      end
9      if $\pi_{in}(0) \in M_2$ then
10         $\pi_{out} \leftarrow \sigma_\circ \oplus \sigma_\times$
11     else
12         $\pi_{out} \leftarrow \sigma_\times \oplus \sigma_\circ$
13     end
14     Compute the block size $\gamma$
15     return $\pi_{out}, \gamma$
16 end
```

Algorithm 2: Modified version of Matricization algorithm.

```
Input  : $\pi_{in}^{A}$ // Input storage format of tensor A.
       : $\pi_{in}^{B}$ // Input storage format of tensor B.
       : $M_A$ // Set of matricization modes of tensor A.
       : $M_B$ // Set of matricization modes of tensor B.
Output: $\pi_{out}^{A}$ // Output storage format of tensor A.
       : $\pi_{out}^{B}$ // Output storage format of tensor B.
1  begin
2      $(\pi_{out}^{A|A}, \gamma_{A|A}, \sigma_{\circ}^{A|A}) \leftarrow$ Algorithm 1($\pi_{in}^{A}, M_A$)
3      $(\pi_{out}^{B|A}, \gamma_{B|A}) \leftarrow$ Algorithm 2($\pi_{in}^{B}, M_A, M_B, \sigma_{\circ}^{A|A}$)
4      $(\pi_{out}^{B|B}, \gamma_{B|B}, \sigma_{\circ}^{B|B}) \leftarrow$ Algorithm 1($\pi_{in}^{B}, M_B$)
5      $(\pi_{out}^{A|B}, \gamma_{A|B}) \leftarrow$ Algorithm 2($\pi_{in}^{A}, M_B, M_A, \sigma_{\circ}^{B|B}$)
6      if $h(\gamma_{A|A}, \gamma_{B|A}) < h(\gamma_{A|B}, \gamma_{B|B})$ then
7          $\pi_{out}^{A} \leftarrow \pi_{out}^{A|B}$
8          $\pi_{out}^{B} \leftarrow \pi_{out}^{B|B}$
9      else
10         $\pi_{out}^{A} \leftarrow \pi_{out}^{A|A}$
11         $\pi_{out}^{B} \leftarrow \pi_{out}^{B|A}$
12     end
13     return $\pi_{out}^{A}, \pi_{out}^{B}$
14 end
```

Algorithm 3: Matricize-pair algorithm.

4 Software

A software library called `dten` has been written in the C programming language with the OpenMP extension used to parallelize the core functions.

4.1 Features

Presently, `dten` contains two levels of functionality: basic and advanced. The basic functionality consists of essential functions for allocating, deallocating, initializing, printing, copying, and saving/loading to/from an HDF5-based file format [24]. In addition, primitive functions for obtaining information about the tensor such as its order and size are provided. The advanced functionality consists of parallel tensor storage format conversion and wrappers for efficient matricization. Both out-of-place (OOP) and in-place (IP) conversion functions are provided. In the out-of-place function, the user can choose which tensor (input or output) to traverse contiguously. The allocation and initialization of the output tensor in the out-of-place conversion is the responsibility of the user to potentially save the allocation and deallocation time, which may be a factor affecting the conversion performance specially for large tensors.

The matricization functionality, as described in Section 3.5, is split into three functions. The first matricizes over a single mode, $|M| = 1$. The second matricizes over a subset of the modes, $|M| > 1$. The third matricizes a pair of tensors such that a contraction over a subset of the modes can be performed afterwards using standard matrix–matrix multiplication routines. For each matricization function, the user can specify the target matrix format as well as the ordering of the modes associated with the rows and/or the columns or choose to leave one or more of these choices to the library.

In addition, if no initial permutation of a tensor is supplied to the allocation, then the library tries to maximize the potential block size by ordering the indices based on their size in a descending order.

4.2 Tunable parameters

The library contains a few parameters that affect the performance of some of the functions and can be tuned to give the best performance on a particular machine. As mentioned in Section 3.4, the parallelism in the in-place conversion function is done by shifting multiple cycles in parallel. The cycles need to be identified first and since the number of cycles is potentially very large, the number of cycles to generate before shifting them in parallel is bounded by a tunable parameter. The cycle cache should be large enough to enable effective load balancing (i.e., much larger than the number of threads), but not too big as to waste a lot of memory. The effect on performance is negligible unless the cache is very small in which case there will not be enough cycles to parallelize, leading to idle threads.

Another parameter is the size of the sub-blocks as described in Section 3.3.2. This parameter affects the core of the conversion function and has a large impact on the performance. The optimal choice depends on the size and characteristics of the memory hierarchy.

4.3 Cycle shifting strategy

As mentioned in Section 3.3.1, there are two ways of shifting a cycle in the in-place technique. The backward shifting technique is used due to the advantages explained in Section 3.3.1.

4.4 Matricize-pair heuristic function

The matricize-pair algorithm described in Section 3.6 uses a heuristic function h to find which tensor to optimize. The heuristic function takes as an input the block sizes, α and β, and returns an estimate of the execution rate when using these two block sizes. For a given set of formats and corresponding block sizes, we assume that the execution rates of the two conversions are limited by the smallest block size among the two; so the heuristic function returns the smallest block size, i.e., $h(\alpha, \beta) = \min\{\alpha, \beta\}$, as an estimate of the execution rate. In Algorithm 3 we get two sets of compatible formats to matricize two tensors together. We evaluate the two sets and we choose the one with the highest execution rate, i.e., the one that gives the highest value for the heuristic function. In case the execution rates are equal, we use the following tie-breaker:

1. Choose the set which leads to applying the smallest block size on the shorter tensor.
2. If both tensors are equally large, maximize the largest block size.

5 Performance

This section presents the performance and the scalability of the conversion function. The experiments were performed on one node of the high performance computer Abisko, which is operated by High Performance Computing Center North (HPC2N) at Umeå University. One node consists of four AMD Opteron 6238 processors clocked at 2.6 GHz. Each processor contains two chips with six cores each for a total of 48 cores per node. Each chip has its own memory controller, which leads to eight NUMA domains per node. Each group of six cores share a memory bus on Abisko, for that reason the number of cores in our tests are multiple of six.

The main factor affecting the conversion performance is the block size. In addition, the performance of the OOP conversion is affected by which tensor will be accessed contiguously. While the performance of the IP conversion is affected by whether the sub-blocking is used or not.

Figure 8 shows the change in memory bandwidth with respect to changing the block size. The figure contains different plots for four different cases, OOP with contiguous access to the output tensor, OOP with contiguous access to the input tensor, IP without sub-blocking and IP with sub-blocking. The four cases were tested on 48 cores. The tensor chosen as a study case is of order 6 with size $\mathbf{n} = \langle x, 8, 4, 4, 5, 2 \rangle$, where x was changed to give different block sizes. The initial storage format was defined by $\phi_{\pi_{in}}(\mathbf{k}; \mathbf{n})$ where $\pi_{in} = \langle 0, 1, 2, 3, 4, 5 \rangle$ and the target storage format was defined by $\phi_{\pi_{out}}(\mathbf{k}; \mathbf{n})$ where $\pi_{out} = \langle 0, 3, 2, 1, 4, 5 \rangle$. The conversion contains 200 cycles, most of them involve moving 7 blocks while the rest are singleton cycles. The sub-blocking size used for the IP conversion is 8KB. The tested block sizes were 8KB, 16KB, 32KB, ..., 3.2MB.

The aggregate cache for the 48 cores on one node of Abisko is 96MB. If the tensor size is more than that it will not fit into the cache. This explains why there is a drop in the memory bandwidth for tensors of sizes larger than the aggregate cache size. Recall that OOP conversion doubles the memory used for tensor storage, see Section 3.2.

It is clear from Figure 8 that accessing either tensor contiguously is not affecting the performance dramatically for the OOP conversion. On the other hand, the IP conversion with sub-blocking improves the performance drastically compared to omitting the sub-blocking for large size tensors. These results influenced us to use OOP conversion with input tensor accessed contiguously and IP conversion with sub-blocking for the rest of the conducted experiments.

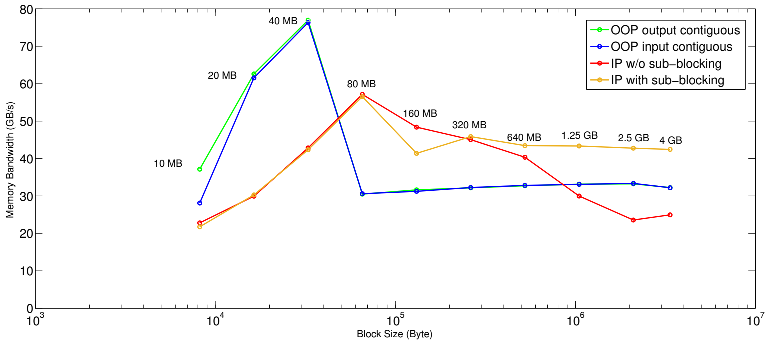


Figure 8: The effect of block size on performance.

To study the scalability of the conversion functions, different tensors are tested. The tensor sizes were taken from the previous described experiment. We chose the case where the aggregate cache is almost full, which is 40MB tensor for the OOP conversion and 80MB tensor for the IP conversion. We will call this case the cache fit case. In addition, 10MB, 640MB and 4GB tensors were tested for both OOP and IP conversion. The real factor that affects the scalability is the memory buses.

Figures 9 and 10 present the scalability of the OOP conversion and the IP conversion functions, respectively. The figures show the number of cores versus the efficiency E given by

$$E = \frac{t_s}{p \times t_p}, \tag{4}$$

where t_s is the sequential execution time, t_p is the parallel execution time and p is number of cores. Both figures show acceptable efficiency for large tensor sizes. The cache fit case gives slightly better performance due to efficient use of cache memory while the small tensor case behave wildly because it is too small to be parallelized.

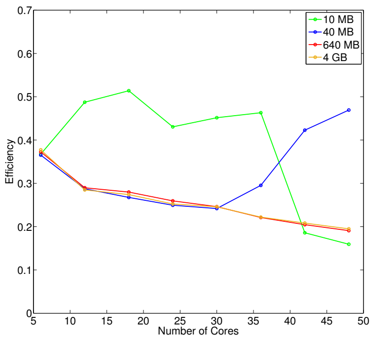


Figure 9: The scalability of OOP conversion.

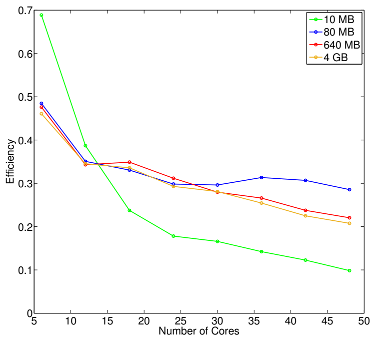


Figure 10: The scalability of IP conversion.

6 Conclusion and related work

An n-dimensional tensor has $n!$ canonical tensor storage formats. Converting a tensor from one format to another can, in many cases, be done efficiently by transferring memory blocks. But sometimes the blocks can degenerate and consist of a single element. The maximum block size is determined by the pair of formats and the size of the tensor. Putting the tensor dimensions in a descending order of size can maximize the potential block size.

Converting a tensor format can be done using an out-of-place technique or an in-place technique. The former uses another memory location to perform the conversion where the latter shifts the tensor blocks within the same memory in cycles. Also the in-place technique requires almost half of the memory but at the expense of exploiting lower degree of inherent parallelism.

We showed that shifting cycles in the in-place technique can be done in two ways, from which the backward shifting is chosen because, compared to the forward shifting, it is more cache efficient and requires less steps. In addition, to benefit from the memory hierarchy and be more cache efficient, the blocks could be divided into sub-blocks in the in-place conversion.

Furthermore, tensor matricization can always be performed using non-degenerate blocks if the output matrix format (row- or column-major) can be chosen freely.

6.1 Related work

Some work has been done in tensor computation and some tools are presented which target storage format of dense tensors. The MATLAB Tensor Toolbox [4] provides a set of tensor related functions such as tensor multiplication, matricization and various tensor decompositions. Another

MATLAB toolbox is Tensorlab [25] which supports complex optimization, tensor factorization and tensor optimization. While the MATLAB toolbox TT-Toolbox [22] provides basic tensor operations for tensors stored in tensor-train format. The python library Scikit-tensor [21] supports basic tensor operations and factorization. A famous software for symbolic computation, Wolfram Mathematica [1], represents tensors as a list of lists. The software supports many tensor operations and tensor related algorithms. In Torch7 [7], a scientific computing framework for machine learning on GPU, a tensor represents a view to a storage. The data in the tensor may not be contiguous in memory. Yet, the framework provides tools for manipulating and rearranging the data in storage. In [15] Albert Hartono et al. present a way for permuting the indices of a tensor by generating multiple code versions optimized during the library installation stage. Another automatic code generator widely used is So Hirata's Tensor Contraction Engine (TCE) [16]. While the main focus is to generate optimized code for tensor contraction, Hirata proposed a way for tensor permutation where the tensor is divided into tiles and the position of the tile for each required permutation during a contraction is precomputed and stored in the memory. Ballared G. et al. in [5] address the problem of symmetric tensors storage format in their proposal for computing tensor eigenvalues on GPU. While Beverly A. Sanders et al. in [23] are focusing more on distributed memory. Also related to distributed memory, Austin W. et al. in [3] propose to distribute the tensor among a processor grid and perform the matricization logically without data movement. While Dmitry I. Lyakh in [20] is considering the case where the block size is one.

Acknowledgments

The work was supported by eSSENCE, a strategic collaborative e-Science program funded by the Swedish Government via the Swedish Research Council (VR). The authors thank the High Performance Computing Center North (HPC2N) for providing computational resources and valuable support during test and performance runs.

References

[1] Wolfram Mathematica: symbolic computation software. https://www.wolfram.com/mathematica/.

[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] W. Austin, G. Ballard, and T. Kolda. Parallel tensor compression for large-scale scientific data. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 912–922. IEEE, 2016.

[4] B. W. Bader and T. G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, December 2006.

[5] G. Ballard, T. Kolda, and T. Plantenga. Efficiently computing tensor eigenvalues on a GPU. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium*, pages 1340–1348. IEEE, 2011.

[6] 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 Transactions on Mathematical Software*, 28(2):135–151, 2002.

[7] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[8] T. Dahlberg. *Compact Representation and Efficient Manipulation of Sparse Multidimensional Arrays.* Bachelor thesis, Ume University, Department of Computing Science, 2014.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] F. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)*, 38(3):17, 2012.

[15] A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D.E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry A*, 113(45):12715–12723, 2009.

[16] S. Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[17] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Software*, 24(3):268–302, 1998.

[18] B. Kågström, P. Ling, and C. Van Loan. Algorithm 784: GEMM-Based Level 3 BLAS: Portability and Optimization Issues. *ACM Trans. Math. Software*, 24(3):303–316, 1998.

[19] 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.

[20] D. I. Lyakh. An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. *Computer Physics Communications*, 189:84 – 91, 2015.

[21] M. Nickel. Scikit-tensor: Python library for multilinear algebra and tensor factorizations. https://github.com/mnick/scikit-tensor.

[22] I. Oseledets, S. Dolgov, V. Kazeev, O. Lebedeva, and T. Mach. TT-Toolbox 2.2, 2012. http://spring.inm.ras.ru/osel.

[23] B. A. Sanders, R. Bartlett, E. Deumens, V. Lotrich, and M. Ponton. A block-oriented language and runtime system for tensor algebra with very large arrays. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.

[24] The HDF Group. Hierarchical Data Format, version 5, 1997-2017. http://www.hdfgroup.org/HDF5/.

[25] N. Vervliet, O. Debals, and L. De Lathauwer. Tensorlab 3.0numerical optimization strategies for large-scale constrained and coupled matrix/tensor factorization. In *2016 Conference Record of the 50th Asilomar Conference on Signals, Systems and Computers. IEEE*, 2016.