An Implementation of the MRRR Algorithm on a Data-Parallel Coprocessor

Christian Lessig*

Abstract

The Algorithm of Multiple Relatively Robust Representations (MRRR) is one of the most efficient and most accurate solvers for the symmetric tridiagonal eigenvalue problem. We present an implementation of the MRRR algorithm on a data-parallel coprocessor using the CUDA programming environment. We obtain up to 50-fold speedups over LAPACK's MRRR implementation and demonstrate that the algorithm can be mapped efficiently onto a data-parallel architecture. The accuracy of the obtained results is currently inferior to LAPACK's but we believe that the problem can be overcome in the near future.

1 Introduction

The eigenvalue problem $\mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_i$[1] for a real symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ with eigenvalues $\lambda_i \in \mathbb{R}$ and (right) eigenvectors $\mathbf{u}_i \in \mathbb{R}^n$ is important in many disciplines. This lead to the development of a variety of algorithms for the symmetric eigenvalue problem [33, 51]. The efficacy of these eigen-solver depends on the required computational resources and the accuracy of the obtained results. With the advent of mainstream parallel architectures such as multi-core CPUs and data-parallel coprocessors also the amenability to parallelization is becoming an important property.

Many eigen-solver are however difficult to parallelize. Notable exceptions are the Divide and Conquer algorithm [14] and the Algorithm of Multiple Relatively Robust Representations (MRRR) [21] for which efficient parallel implementations exist. Both algorithms provide additionally highly accurate results [19]. The computational costs of the Divide and Conquer algorithm are with $\mathcal{O}(n^3)$ however significantly higher than the $\mathcal{O}(n^2)$ operations required by the MRRR algorithm. The Divide and Conquer algorithm also does not allow to efficiently compute a subset of eigen-pairs $(\lambda_i, \mathbf{u}_i)$ which is possible with MRRR. Task-parallel implementations of the MRRR algorithm are available [8] but to our knowledge it has not yet been realized on a data-parallel architecture.

In this report we present an implementation of the MRRR algorithm on a data-parallel coprocessor using the CUDA programming environment [48]. We show that the MRRR algorithm can be mapped efficiently onto a data-parallel architecture and obtain up to 50-fold speedups compared to `sstemr`, LAPACK's [3] implementation of the MRRR algorithm.

*lessig@dgp.toronto.edu

[1] In this report we will employ Householder's notation. Matrices will be denoted by bold capital letters such as $\mathbf{A}$ and $\mathbf{T}$; vectors by bold small letters such as $\mathbf{a}$ and $\mathbf{b}$; and scalars by non-bold letters such as a, b, and α.

Although our implementation does currently not provide the same accuracy and robustness as `sstemr`, we believe that these problems can be overcome in the near future.

The remainder of the report is organized as follows. In the next section we will review algorithms for the symmetric eigenvalue problem. A brief introduction to data-parallel coprocessors and the CUDA programming environment is given in Section 3, before the mathematical background of the MRRR algorithm is presented in Section 4. A detailed discussion of our implementation and design choices is provided in Section 5, and in Section 6 we present experimental results comparing our implementation of the MRRR algorithm to LAPACK's `sstemr` routine. We conclude the report with a discussion of possible directions of future work. Appendix B provides a liner algebra primer with most of the mathematical background which underlies our implementation. Readers not familiar with the theoretical aspects of eigen-solvers are encouraged to consult the appendix before reading the remainder of the report.

2 Related Work

Algorithms for the symmetric eigenvalue problem have been studied extensively in numerical analysis, applied mathematics, physics, and many other fields. To limit the scope of this section we will restrict ourselves to approaches which have been implemented on parallel architectures. More complete discussions can be found in the books by Golub and van Loan [32, 33] and Parlett [51], and various LAPACK working notes [44].

The first algorithm for the symmetric eigenvalue problem was proposed by Jacobi as early as 1846 [40]. Jacobi iteration performs a sequence of similarity transformations[2] on the full symmetric input matrix $\mathbf{A}$ annihilating at each step one off-diagonal element. $\mathbf{A}$ is guaranteed to converges to a diagonal matrix $\mathbf{D}$ and the eigenvalues of the input matrix are the non-trivial elements of $\mathbf{D}$ [12]. The eigenvectors can be obtained by accumulating the similarity transformations starting with the identity matrix [12]. The eigenvalues and eigenvectors obtained with Jacobi iteration are usually highly accurate [15]. The algorithm suffers however from high computational costs of at least $\mathcal{O}(n^3)$ and in practice usually requires an order of magnitude more work than for example the QR algorithm [5]. Today it is therefore only used for special problems [21]. Due to its simplicity, Jacobi iteration was the first symmetric eigen-solver which was implemented on a parallel architecture [10, 4, 63, 7, 6]. We are however not aware of any recent parallel implementations.

In contrast to Jacobi iteration which operates on an unreduced matrix, most algorithms for the symmetric eigenvalue problem in use today determine the eigen-analysis for a tridiagonal matrix.[3] Householder transformations [33, pp. 206] are usually employed to reduce the input to tridiagonal form, requiring $\frac{4}{3}n^3$ multiplications and additions for the reduction and $2n^3$ additional operations for the back-transformation. The BLAS library [25], for which various processor and architecture specific implementations exist, including one for data-parallel coprocessors [41], can be employed to compute the transformations.

The canonical algorithm for the symmetric tridiagonal eigenvalue problem is *QR iteration* [33, pp. 414] which was discovered independently by Francis [30] and Kublanovskaya [43]. The QR algorithm applies a sequence of orthogonal transformations, each formed by $(n-1)$ elementary Givens rotations, to the input matrix $\mathbf{T}$. The eigenvalues are again the non-trivial elements of the resulting diagonal matrix $\mathbf{D}$ and the eigenvectors

[2] Let $\mathbf{X} \in \mathbf{R}^{n \times n}$ and be non-singular, and $\mathbf{B} = \mathbf{X}^{-1}\mathbf{A}\mathbf{X}$. It can be shown that the spectrum of $\mathbf{A}$ and $\mathbf{B}$ [33, p. 311]. $\mathbf{X}$ is therefore called a similarity transformation.

[3] A notable exception are algorithms which determine only a specific eigen-pair such as power iteration. Problems where this is of interest arise in many applications, for example internet search algorithms. In this report we restrict ourselves however to algorithms which can efficiently determine the whole spectrum of a matrix.

can be obtained by accumulating the transformations. The QR algorithm requires $\mathcal{O}(n^2)$ operations to determine all eigenvalues and $\mathcal{O}(n^3)$ operations to compute all eigenvectors. These costs remain approximately constant even if only a subset of the eigen-pairs $(\lambda_i, \mathbf{u}_i)$ is required. Despite its serial nature, different parallel implementation of the QR algorithm have been proposed [26, 59, 63, 42]. Today, the accumulation of the Givens rotations is usually distributed equally across all processors, and the eigenvalues are computed redundantly on all nodes [5].

In the early 1980's, Cuppen developed the *Divide and Conquer Algorithm* as an efficient and accurate eigen-solver for parallel architectures [14]. Given a symmetric tridiagonal matrix $\mathbf{T}$, a solution is obtained recursively by considering sub-matrices until these are small enough to efficiently determine eigenvalues and eigenvectors. The eigen-pairs of the sub-matrices are then related back to the original matrix in a back-propagation step [60]. Initially, the Divide and Conquer algorithm could not guarantee that the obtained eigenvectors are orthogonal but Gu and Eisenstat [38] devised a strategy which overcomes this problem. Similar to QR iteration, the Divide and Conquer algorithm requires $\mathcal{O}(n^3)$ operations. In contrast to most other eigen-solvers it demands however also $\mathcal{O}(n^2)$ additional memory. A parallel implementation of the Divide and Conquer algorithm can be obtained by distributing the work for different sub-matrices across different nodes [13]. It is interesting to note that although Cuppen developed the algorithm for parallel architectures in many cases it also outperforms other eigen-solvers on a single processor [5, 19].

An alternative algorithm for the symmetric tridiagonal eigenvalue problem combines *bisection* [31] and *inverse iteration* [56]. Although the eigen-pairs obtained with this technique satisfy $\mathbf{T}\mathbf{u}_i - \lambda_i \mathbf{u}_i$ to high relative accuracy, the orthogonality of the $\mathbf{u}_i$ cannot be guaranteed. In practice the eigenvectors of clustered eigenvalues have therefore to be orthogonalized, for example with the Gram-Schmidt algorithm. It is also known that inverse iteration can fail entirely [20] although this seems to be a mostly theoretical problem [19]. Bisection and inverse iteration both require $\mathcal{O}(n^2)$ operations but the orthogonalization requires $\mathcal{O}(n^3)$ work. An efficient parallel implementation of bisection and inverse iteration is possible but orthogonalizing the eigenvectors requires a considerable amount of communication. ScaLAPACK [8] computes the eigenvectors of a cluster of eigenvalues therefore on one node. This avoids excessive communication but can lead to load imbalance [5].

In the 1990's, Dhillon and Parlett proposed the *Algorithm of Multiple Relatively Robust Representations* (MRRR) [21, 52, 53, 22, 23, 24] for the symmetric tridiagonal eigenvalue problem. The MRRR algorithm also employs bisection and a version of inverse iteration but can guarantee the orthogonality of the computed eigenvectors to high relative accuracy without explicit orthogonalization. Parlett and Vömel [54] showed that the original MRRR algorithm can fail on tight eigenvalue clusters but Dhillon [54] devised a simple strategy to circumvents this problem. In contrast to other eigen-solvers which can require $\mathcal{O}(n^3)$ operations, MRRR has a worst case complexity of $\mathcal{O}(n^2)$ for obtaining all eigen-pairs, and it requires only $\mathcal{O}(kn)$ operations to determine a subset of k eigenvalues and eigenvectors. Similar to the Divide and Conquer algorithm, the MRRR algorithm generates a computation tree which can guide a parallel implementation.

The previous discussion raises the question what the most efficient algorithm for the symmetric tridiagonal eigenvalue problem is. The efficacy of an eigen-solver depends however on the input matrix, the hardware architecture on which the algorithm is implemented, the desired accuracy, and many other parameters so that in general no answer exist. Nonetheless, Demmel et al. [19] recently compared LAPACK's implementations [3] of QR iteration, bisection and inverse iteration, the Divide and Conquer algorithm, and the MRRR algorithm on a variety of single processor systems. They concluded that Divide and Conquer and MRRR are the fastest algorithm, and that QR iteration and Divide and Conquer are the most accurate once, although the MRRR algorithm provides the relative accuracy of $\mathcal{O}(\epsilon\, n)$ it promises.

The parallelization strategies discussed above, which have been used for example in ScaLAPACK [8], PLAPACK [1], or PeIGS [27], have been developed for distributed memory systems. Our target hardware is however a task-parallel array of data-parallel processing units. Disregarding some preliminary work on the Connection Machine [28, 57, 58], we are not aware of implementations of eigen-solvers on data-parallel architectures; in particular the Divide and Conquer algorithm and the MRRR algorithm, which naturally lend themselves to parallel implementations [14, 21] and which are highly efficient [19], have not been implemented on such systems. In this report we will present an implementation of the MRRR algorithm on a data-parallel coprocessor using the CUDA programming environment.

3 Data Parallel Coprocessors

Traditionally, highly parallel architectures such as the Connection Machine [62] and Cray's vector computers were prohibitively expensive and available only to large institutions and companies. With the transition from graphics supercomputers to off-the-shelf graphics processing units (GPUs) in the late 1990's highly parallel architectures with thousands of threads in flight simultaneously became however commodity. Today, these provide a readily available data-parallel coprocessor available in almost every computer with a raw compute power which exceeds the latest CPUs by more than an order of magnitude [49].

In the early 2000's, the widespread availability and high performance of GPUs combined with emerging programmability spurred the interest of the research community to employ the processors not only for calculations in computer graphics but also for general-purpose computations (GPGPU). Significant speedups have been reported for a variety of applications, ranging from numerical analysis to databases [49], where the algorithms mapped well onto the GPU. From the beginning, the practicality of GPGPU was however compromised by a variety of shortcomings; graphics APIs had to be used to write programs which made it inaccessible for most people; the hardware was optimized for direct lighting calculations and the parallelism was exposed only implicitly which led to cumbersome program design and often to large inefficiency for general-purpose computations; and synchronization between threads, which is necessary for most non-trivial computations, was prohibitively expensive; to name a few examples.

In the last years, a variety of approaches have been proposed to overcome these limitations. Middleware solutions such as SH [47] and Brook [11] provide an additional software layer on top of the graphics API. In most cases this simplifies program design considerably, in particular for people outside of computer graphics. It does however not overcome hardware limitations such as the lack of scatter writes and in fact might incur additional costs at runtime. ATI [55] proposed Close-To-Metal (CTM), a low-level general-purpose API for its GPUs, which is for many computations more efficient than OpenGL or DirectX. The software-only nature of CTM limits however its impact and it shares many of the limitations of traditional GPGPU and middleware solutions. NVIDIA's Compute Unified Device Architecture (CUDA) [48] improves on GPGPU with both software and hardware enhancements and exposes the GPU as a highly multi-threaded data-parallel coprocessor with a single-program multiple-data (SPMD) execution model which is programed with a superset of ANSI C, providing the most compelling approach to mainstream data-parallel programming so far.

In the remainder of the section we will provide an overview of the CUDA programming environment. A detailed discussion can be found in the CUDA programming guide [48]. We refer to the survey article by Owens et al. [49] for a more complete treatment of GPGPU.

3.1 CUDA Hardware

The latest generation of NVIDIA "graphics" hardware provides for the first time functionality specifically designed for general-purpose computations. Depending on the API used – OpenGL / DirectX or CUDA – it is therefore either employed as a graphics processing unit or as a general-purpose data-parallel coprocessor. In fact, time slicing allows to use the hardware (virtually) simultaneously for image synthesis and general-purpose computations.

The computational elements of CUDA-enabled hardware are arranged as a task-parallel array of data-parallel multiprocessors. Each multiprocessor has an independent SPMD execution unit and can run up to 512 threads simultaneously. Barriers are available for synchronization. User-controlled shared memory, which is also local to each multiprocessor and has thus a very low latency, allows to efficiently share data between threads and to cache it close to the computational units. Although the practicality of shared memory is often compromised by the small size of 16kB, it remains one of the distinguishing features of CUDA. Next to on-chip memory, CUDA hardware also provides three types of DRAM memory: global memory, texture memory, and constant memory. These are accessible from the host[4] and the device and thus serve as interface between the execution units. In contrast to texture and constant memory which are similar to their analogs on traditional graphics hardware, global memory provides read *and* write access on the device. This overcomes the gather-only limitation of traditional GPUs. The access is however un-cached which makes the efficient use of global memory often difficult, although with enough threads in flight in parallel on a multiprocessor it is possible to hide the memory latency [48].

3.2 CUDA Software

The most important components of the CUDA software layer are an extension of ANSI C [2] for data-parallel coprocessors, which supports also many useful features of C++ [61], and a meta-compiler which separates program parts executed on the host and device. The host code is compiled with a standard CPU compiler such as `gcc` and the device code compiler is again part of the CUDA software layer. In contrast to traditional GPGPU programming [49], this allows to develop data-parallel programs in a way very similar to CPU applications; for example if a function is executed on the device or on the host is determined by an intrinsic which is part of the function declaration and on the GPU the same standard math library is available as on the CPU.

CUDA programs, one often refers to them as kernels, are executed as a grid of thread blocks and each block runs on one multiprocessor. The execution environment for a kernel, that is how many blocks and how many threads per block are used when the program is run, is specified at launch time.

4 The MRRR Algorithm

The *Algorithm of Multiple Relatively Robust Representations* (MRRR) [21] computes k eigen-pairs $(\lambda_i, \mathbf{u}_i)$ of a symmetric tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n\times n}$ in $\mathcal{O}(nk)$ time while guaranteeing small residuals of the eigen-decomposition

$$\|\mathbf{T}\mathbf{u}_i - \lambda_i \mathbf{u}_i\| = \mathcal{O}\left(n\epsilon\|\mathbf{T}\|\right) \tag{1}$$

and orthogonality of the eigenvectors

$$|\mathbf{u}_i^T \mathbf{u}_j\| = \mathcal{O}\left(n\epsilon\right) \quad , \quad i \neq j. \tag{2}$$

[4] In the literature it is common practice to denote the CPU as *host* and the data-parallel coprocessor as *device*.

Algorithm 1: MRRR algorithm for the tridiagonal symmetric eigenvalue problem.

Input: Tridiagonal symmetric matrix $\mathbf{T} \in \mathbb{R}^{n \times n}$
Output: List of eigen-pairs $(\lambda_i, \mathbf{u}_i)$ of $\mathbf{T}$.

1 Find an RRR for $\mathbf{T}$.
2 Compute approximation $\tilde{\lambda}_i$ of eigenvalue λ_i and classify as singleton or cluster.
3 **for** *each singleton* λ_i **do**
4 Compute the eigenvalue and eigenvector to high relative accuracy.
5 **for** *each cluster* $\lambda_{k_1:k_m}$ **do**
6 Shift $\mathbf{T}$ with a shift index μ close to the cluster to obtain $\bar{\mathbf{T}}$.
7 Let $\mathbf{T} \equiv \bar{\mathbf{T}}$ and go to line 2.

An overview of the MRRR algorithm is given in Algo. 1. Its three parts, eigenvalue classification (line 2), eigen-pair computation (line 3), and cluster shift (line 5), will be detailed in the following. A more complete discussion can be found in the thesis by Dhillon [21].

4.1 Eigenvalue Classification

Definition 1 (Relative Distance and Relative Gap [22])**.** *Let λ_i and λ_j be two eigenvalues of a tridiagonal symmetric matrix $\mathbf{T} \in \mathbb{R}^{n \times n}$, with $\lambda_i < \lambda_j$. The* relative distance $\operatorname{reldist}(\lambda_i, \lambda_j)$ *between two eigenvalues is*

$$\operatorname{reldist}(\lambda_i, \lambda_j) \equiv \frac{|\lambda_i - \lambda_j|}{|\lambda_i|}.$$

The relative gap $\operatorname{relgap}(\lambda_i)$ *of an eigenvalue λ_i is the minimum over all relative distances:*

$$\operatorname{relgap}(\lambda_i) \equiv \min \{\operatorname{reldist}(\lambda_i, \lambda_j) \,|\, \lambda_i \neq \lambda_j \in \lambda(\mathbf{T})\}.$$

An eigenvalue λ_i is (relatively) isolated, or a *singleton*, if its relative gap exceeds a given threshold t_c. It can be shown that for a singletons λ_i the eigen-pair $(\lambda_i, \mathbf{u}_i)$ can be computed to high relative accuracy [21]. A group $\lambda_{k_1:k_m}$ of m non-isolated eigenvalues forms an *eigenvalue cluster* of multiplicity m.

For the practicality of the MRRR algorithm it is important that an approximation $\tilde{\lambda}_i$ of the eigenvalue λ_i is sufficient to classify it as singleton or cluster. A common methods to obtain $\tilde{\lambda}_i$ is the bisection algorithm [29, 16] which approximates the eigenvalue as an interval around λ_i with size t_c. Singletons thus correspond to intervals containing one eigenvalue and cluster to intervals containing multiple eigenvalues. Bisection relies on Sturm count computations[5] to determine the number of eigenvalues in an interval. For a symmetric tridiagonal matrix the Sturm count can be obtained with a $\mathbf{qds}$ transform similar to those in Algo. 2 and Algo. 3.

4.2 Cluster Shift

For eigenvalues that are part of a cluster the eigen-pairs $(\lambda_i, \mathbf{u}_i)$ cannot be computed accurately [21]. Matrix shifts are employed to increase the relative distance between the eigenvalues in a cluster until these become singletons.

[5] Given a symmetric matrix $\mathbf{A}$ and a shift μ, the *Sturm count* $s_{\mathbf{A}}(\mu)$ is the number of eigenvalues smaller than μ (cf. Def. 4 in Appendix B).

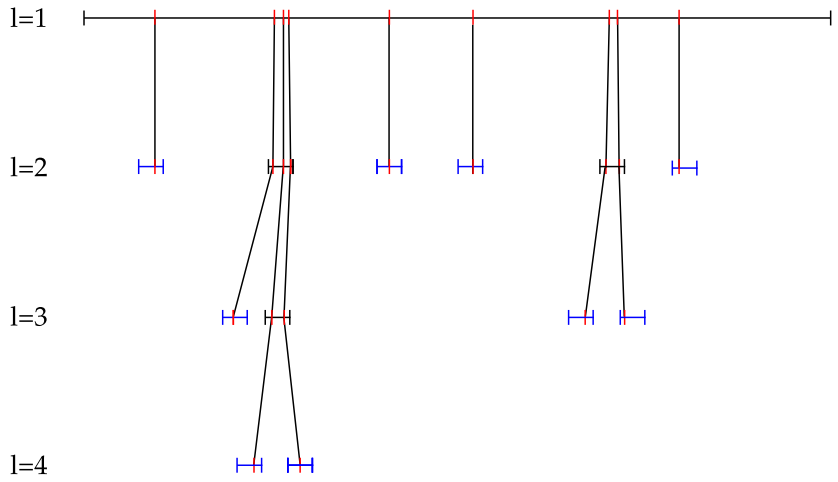

Figure 1: Representation tree $\mathcal{T}$ for a matrix $\mathbf{T}_{8\times 8}$. Singletons are the leaf nodes of the tree, shown in blue, and clusters are inner nodes, shown in black. Here we assumed that bisection was used to classify and approximate the eigenvalues so that $\bar{\lambda}_i$ is the midpoint of the blue intervals shown in the figure. The interval at level $l = 1$ is the Gerschgorin interval $\mathrm{G}_{\mathbf{T}}$ of the input matrix. Through the matrix shifts which are used for every cluster on level $j - 1$ to create the corresponding matrix $\mathbf{T}_{j,k}$ the relative distance of clustered eigenvalues is increased from level to level until all eigenvalues are singletons.

Let the eigenvalues λ_{k_1} to λ_{k_m} form the k^{th} cluster $\lambda_{k_1:k_m}$ of $\mathbf{T}$, and let μ be an approximation of the cluster location. The relative gap of an eigenvalue $\bar{\lambda}_{k_i} \in \bar{\lambda}_{k_1:k_m}$ of the shifted matrix $\bar{\mathbf{T}} = \mathbf{T} - \mu\mathbf{I}$[6] is then

$$\mathrm{relgap}_{\bar{\mathbf{T}}}\left(\bar{\lambda}_{k_i}\right) = \mathrm{relgap}_{\mathbf{T}}\left(\lambda_{k_i}\right) \frac{|\lambda_{k_i}|}{|\lambda_{k_i} - \mu|}.$$

With the choice $\mu \approx \lambda_{k_i}$ the denominator $|\lambda_{k_i} - \mu|$ becomes small and

$$\mathrm{relgap}_{\bar{\mathbf{T}}}\left(\bar{\lambda}_{k_i}\right) \gg \mathrm{relgap}_{\mathbf{T}}\left(\lambda_{k_i}\right).$$

The shifted eigenvalues $\bar{\lambda}_{k_1}$ to $\bar{\lambda}_{k_m}$ are thus likely to be singletons which has to be verified by classifying the $\bar{\lambda}_{k_i}$ with respect to $\bar{\mathbf{T}}$. For eigenvalues $\bar{\lambda}_{k_i}$ that are again part of a cluster, now with respect to $\bar{\mathbf{T}}$, a new shift $\bar{\mu} \approx \bar{\lambda}_{k_i}$ is applied to $\bar{\mathbf{T}}$. This process is repeated until all eigenvalues are classified as singletons and the corresponding eigen-pairs have been computed to high relative accuracy. The eigenvalues of the shifted matrices can be related back to those of the input matrix $\mathbf{T}$ by accumulating the shifts.

Classifying eigenvalues and shifting clusters generates a *representation tree* of matrices

$$\mathcal{T} = \{\mathbf{T}_{j,k} \mid j \in \mathcal{J}, k \in \mathcal{K}(j)\}, \tag{3}$$

where singletons are leaf nodes and clusters are inner nodes, and the root node $\mathbf{T}_{1,1} \equiv \mathbf{T}$ is the original input matrix (cf. Fig. 1). The representation tree describes all computations necessary to determine the eigenvalues and eigenvectors of $\mathbf{T}$.

Important for the practicality of the MRRR algorithm is that no precision is lost when the representation tree is traversed and shifted matrices $\mathbf{T}_{j,k}$ are obtained. Dhillon showed

[6] It is easy to show that the eigenvalues of $\mathbf{T} - \mu\mathbf{I}$ are those of $\mathbf{T}$ shifted by μ, and that the eigenvector are unaffected by the shift (cf. Theorem 2 in Appendix B).

Algorithm 2: DSTQDS

Input: $\mathbf{L}$, $\mathbf{D}$, μ
Output: $\mathbf{L}^+$, $\mathbf{D}^+$, $\mathbf{S}^+$

$s_1^+ = -\mu$;
for $i = 1 : n-1$ **do**
  $d_i^+ = d_i + s_i^+$;
  $l_i^+ = d_i l_i / d_i^+$;
  $s_{i+1}^+ = l_i^+ l_i s_i^+ - \mu$
$d_n^+ = d_n + s_n^+$

Algorithm 3: DPQDS

Input: $\mathbf{L}$, $\mathbf{D}$, μ
Output: $\mathbf{U}^+$, $\mathbf{R}^+$, $\mathbf{P}^+$

$p_n^+ = d_n - \mu$;
for $i = n-1 : -1 : 1$ **do**
  $r_{i+1}^+ = d_i l_i^2 + p_{i+1}^+$;
  $u_i^+ = l_i d_i / r_{i+1}^+$;
  $p_i^+ = p_{i+1}^+ d_i / r_{i+1}^+ - \mu$
$r_1^+ = p_1^+$

Figure 2: Differential stationary and differential progressive **qds** transforms which yield the $\mathbf{L}^+\mathbf{D}^+\left(\mathbf{L}^+\right)^T$ and $\mathbf{U}^+\mathbf{R}^+\left(\mathbf{U}^+\right)^T$ factorizations of $\mathbf{LDL}^T - \mu\mathbf{I}$, respectively.

that this is satisfied when *relatively robust representations* (RRR)[7] are employed for the computations. For a symmetric tridiagonal matrix the LDL^T factorization[8] is in most cases an RRR [21] and the **dstqds** transform in Algo. 2 can be employed to perform the matrix shifts. Care is however required to avoid element growth. An appropriate choice of the shift μ can usually circumvent this problem.

4.3 Eigenvector Computation

After $\lambda_s \equiv \lambda_i$ has been classified as singleton an accurate approximation $\bar{\lambda}_s$ of the eigenvalue is obtained from $\tilde{\lambda}_s$, for example again by using the bisection algorithm with a refinement threshold t_r, with $t_r \leq t_c$. With $\bar{\lambda}_s$, we seek $\bar{\mathbf{u}}_s$ such that Eq. 1 holds. Then the classic gap theorem [50] guarantees that the eigenvectors are orthogonal and Eq. 2 is satisfied [66]. A vector $\mathbf{q} \equiv \bar{\mathbf{u}}_s$ with elements $\mathbf{q} = \{q_i\}_{i=1}^n$ can be obtained by first determining the double factorization

$$\mathbf{LDL}^T - \bar{\lambda}_s\mathbf{I} = \mathbf{L}^+\mathbf{D}^+\left(\mathbf{L}^+\right)^T = \mathbf{U}^+\mathbf{R}^+\left(\mathbf{U}^+\right)^T$$

using the differential stationary and differential progressive **qds** transforms, **dstqds** and **dpqds** in Algo. 2 and Algo. 3, respectively, and then computing

$$\begin{aligned} q_k &= 1, \\ q_i &= -l_i^+ q_{i+1} \quad \text{for } i = k-1, \ldots, 1 \\ q_{i+1} &= -u_i^+ q_i \quad \text{for } i = k, \ldots, n-1. \end{aligned} \tag{4}$$

The remaining question is which index k should be employed in Eq. 4. Dhillon showed in his dissertation [21] that Eq. 1 is satisfied when k is chosen such that $|\gamma_k|$ is minimal or sufficiently small, with

$$\gamma_k = s_k^+ + p_k^+ + \mu, \tag{5}$$

and s_k^+ and p_k^+ being the non-trivial elements of $\mathbf{S}^+$ and $\mathbf{P}^+$ from the **dstqds** and **dpqds** transforms, respectively.

[7] A representation of a matrix, that is any set of numbers that uniquely identifies the matrix, is relatively robust (w.r.t the eigen-decomposition) if small relative changes to the elements of the representation cause only small relative changes to the eigenvalues and eigenvectors.

[8] For a symmetric tridiagonal matrix $\mathbf{T}$ the LDL^T factorization $\mathbf{T} = \mathbf{LDL}^T$ is formed by the diagonal matrix $\mathbf{D}$ and $\mathbf{L} = \mathbf{I} + \bar{\mathbf{L}}$, where $\bar{\mathbf{L}}$ has nonzero elements only on the first lower diagonal.

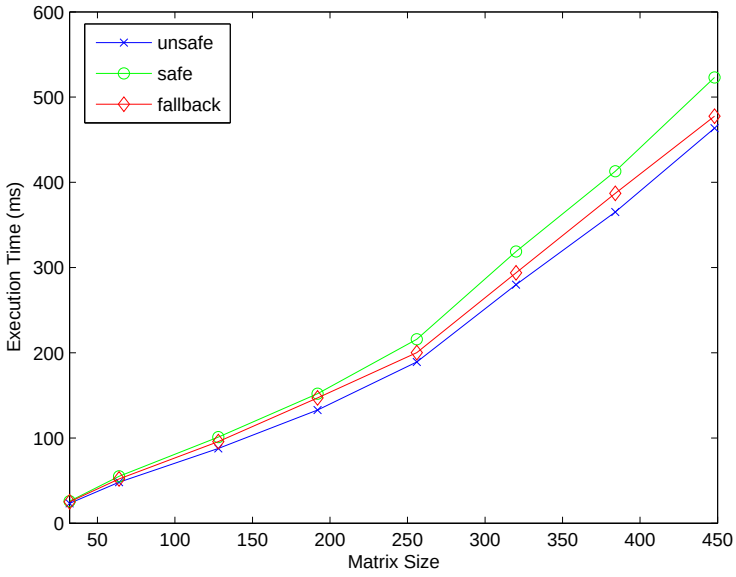

Figure 3: Execution time with different implementations of the **dstqds** and **dpqds** transforms. The try-and-fallback strategy outperforms the safe implementation. Using only the unsafe implementation is the fastest approach but can lead to incorrect results.

5 Implementation

In this section our CUDA implementation of the MRRR algorithm on a data-parallel coprocessor will be detailed. We will first discuss how the eigenvalues and eigenvectors for matrices with at most 512×512 elements can be determined to high relative accuracy, and then explain how this implementation can be used to compute the eigen-pairs for arbitrary size matrices.

5.1 Small Matrices

5.1.1 Overview

Input The diagonal and off-diagonal elements $\mathbf{a}$ and $\mathbf{b}$ of a symmetric tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n \times n}$ with $n \leq 512$, thresholds t_c and t_r for classifying and refining eigenvalue approximations, and the number of thread blocks K used for the computations.[9]

Output All eigenvalues and eigenvectors of $\mathbf{T}$ computed to high relative accuracy.

[9] If the specified number of thread blocks exceeds the number of multiprocessors available in hardware then some of the work will be serialized. Assume for example that $K = 8$ but that only two multiprocessors are available in hardware. Then the computations for two thread blocks are executed in parallel (assuming equal execution time) and a four-way serialization occurs. CUDA handles such serializations transparently, exposing to the programmer a device with an virtually infinite number of multiprocessors. In the remainder of the report we will therefore allow any value for K, disregarding possible performance penalties.

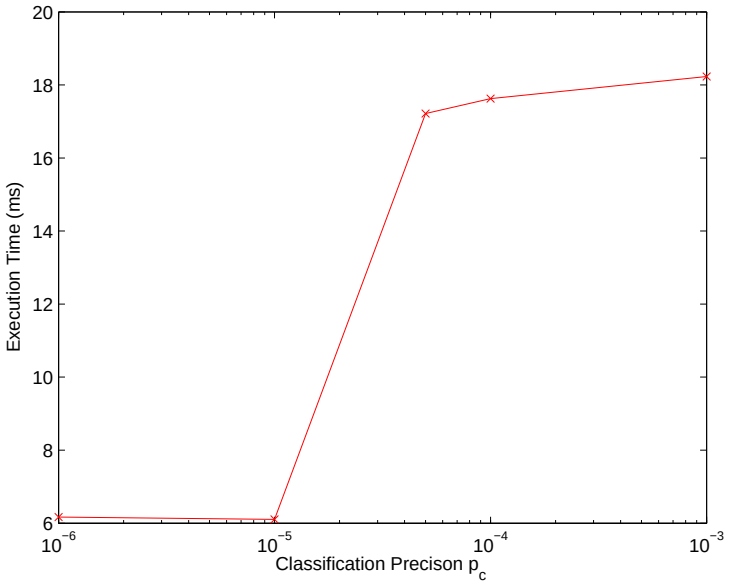

Figure 4: Execution time as a function of the classification threshold t_c for a matrix with 512×512 elements. A significant performance improvement can be observed for $t_c \leq 0.00001$ compared to $t_c \geq 0.00005$.

Host Computations Given $\mathbf{a}$ and $\mathbf{b}$, first the Gerschgorin interval $\mathrm{G}_\mathbf{T}$ of $\mathbf{T}$ is computed [33, p. 395] and K subintervals $\{\mathrm{I}_k\}_{k=1}^K \subset \mathrm{G}_\mathbf{T}$ with $\mathrm{I}_k = (l_k, u_k]$ and $\bigcup_{k \in K} \mathrm{I}_k = \mathrm{G}_\mathbf{T}$ are determined.[10] Then the initial relatively robust representation is obtained

$$\mathbf{L}_{1,1}\mathbf{D}_{1,1}\mathbf{L}_{1,1}^T = \mathbf{T} - \mu_{1,1}\mathbf{I}$$

where $\mu_{1,1}$ is chosen close to the end of the spectrum of $\mathbf{T}$ which contains the most eigenvalues [23]. Next, the data of the initial LDL^T factorization is copied to the device and the device kernel is launched with a one-dimensional grid of thread blocks where the number of threads per block equals the matrix size n. The parameters passed to the kernel are the pointers to the initial LDL^T factorization, the bounds for the intervals I_k, the initial shift $\mu_{1,1}$, the matrix size n, the thresholds t_c and t_r, and pointers to arrays in global memory which are used to store the result and intermediate data; for the eigenvalues a one-dimensional array is employed, and for the eigenvectors and the intermediates two-dimensional arrays are used.

Device Computations We compute the eigen-pairs $(\lambda_i, \mathbf{u}_i)$ for all eigenvalues $\lambda_i \in \mathrm{I}_k$ in an interval I_k on one multiprocessor. Instead of creating one representation tree we thus create a forest of trees. In the remainder of the report it will therefore be sufficient to consider only one interval I_k for arbitrary k.

At the heart of the MRRR algorithm is the $\mathbf{qds}$ transform in its different forms. It is used to determine shifted matrices when the representation tree is traversed and to compute eigenvectors for leaf nodes. Additionally, it also underlies the Sturm count computation;

[10] In practice we currently also compute the Sturm counts $\mathrm{s}_\mathbf{T}(l_k)$ and $\mathrm{s}_\mathbf{T}(u_k)$ but this could be avoided if necessary.

```
struct SharedMem {

  // interval bounds
  float  left[MAX_THREADS_BLOCK];
  float  right[MAX_THREADS_BLOCK];

  // number of eigenvalues that are smaller than left / right
  short  left_count[MAX_THREADS_BLOCK];
  short  right_count[MAX_THREADS_BLOCK];

  // shift for each interval / representation tree node
  float  shift[MAX_THREADS_BLOCK];

  // helper for stream compaction
  short  row_index[MAX_THREADS_BLOCK];

  // helper for compaction of arrays
  short  compaction[MAX_THREADS_BLOCK + 1];
};
```

Figure 5: Arrays wrapped in a C struct are used to store the representation tree nodes in shared memory.

an integral part of bisection, our algorithm of choice for classifying and refining eigenvalue approximations. Parallelizing the transform is however not possible because it consists of a loop with dependent iterations (cf. Algo. 2 and Algo. 3).

The tree structure of the MRRR algorithm – the representation tree $\mathcal{T}$ – suggests however that the computations for all nodes on the same level of $\mathcal{T}$ are independent and thus amendable to parallelization. Moreover, for nodes of the same type, inner nodes or leaf nodes, the computations are identical and differ only by their input data, providing a natural mapping onto a data-parallel architecture. Instead of parallelizing the $\mathbf{qds}$ transform we thus compute the transform for all nodes of $\mathcal{T}$ on the same level and of the same type in parallel using one thread per node. This leads to a breadth-first traversal of the representation tree.

The performance of CUDA programs does however not only depend on sufficient parallelism but also on an effective use of the memory hierarchy. In particular shared memory with its very low latency has to be employed efficiently. In our implementation we use shared memory as user-managed cache and store the interval information of the representation tree nodes on the current tree level (cf. Fig. 5):

- left and right interval bounds,
- Sturm counts of the bounds,
- accumulated shift index of the representation tree node,
- a row pointer providing an offset for the global memory arrays which are used to store intermediate results.

The intervals data is needed for most computations of the MRRR algorithm and it is thus beneficial to retain it close to the computational units.

At execution time, first the pointers to the two-dimensional global memory arrays are initialized with the Sturm count $s_{\mathbf{T}}(l_k)$ as offset to guarantee that different multiprocessors are operating on non-overlapping parts of the memory. Next, the parameters of I_k are stored in shared memory and bisection is employed to classify the eigenvalues as singletons or clusters. The generated intervals are then sorted according to the number of eigenvalues they

```
// make sure the shared memory is not used
__syncthreads();

// read data into shared memory
if( threadIdx.x < s_num_intervals) {
  smem.d[threadIdx.x] = gmem.d[threadIdx.x];
  smem.l[threadIdx.x] = gmem.l[threadIdx.x];
}

// make sure every thread read the data before it is used
__syncthreads();

// compute qds transform
...
```

Figure 6: On level 2, for an $n \times n$ matrix n threads are employed to load $\mathbf{d}_{1,1,}$ and $\mathbf{l}_{1,1}$ in parallel from global to shared memory. For the computation of the $\mathbf{qds}$ transform then only (fast) shared memory has to be accessed.

contain and for singletons the eigenvalue approximations are refined and the eigenvectors are obtained by first computing the $\mathbf{dstqds}$ and $\mathbf{dpqds}$ transforms (Algo. 2 and Algo. 3) and determining the twist index k with $\text{argmin}_k|\gamma_k|$ (Eq. 5), and then solving Eq. 4 for the eigenvectors.

After singletons have been resolved, for each cluster k a new LDL$^{\text{T}}$ factorization

$$\mathbf{L}_{2,k}\mathbf{D}_{2,k}\mathbf{L}_{2,k}^T = \mathbf{L}_{1,1}\mathbf{D}_{1,1}\mathbf{L}_{1,1}^T - \mu_{1,k}\mathbf{I}$$

is computed using the $\mathbf{dstqds}$ transform in Algo. 2. The non-trivial elements of $\mathbf{L}_{2,k}$ and $\mathbf{D}_{2,k}$ are stored in global memory in two of the arrays for intermediates. The generated tree nodes are then processed analogously to the root node, although for $l > 2$ there are in general multiple nodes as input. This process is continued until all eigenvalues and eigenvectors have been determined to high relative accuracy, or until a maximum tree level has been reached.

5.1.2 qds Transform

The $\mathbf{qds}$ transform in its different forms is employed at all stages of the MRRR algorithm (cf. Sec. 4). It takes as input vectors $\mathbf{d}$ and $\mathbf{l}$ formed by the non-trivial elements of the LDL$^{\text{T}}$ factorization of a symmetric tridiagonal matrix and the result is determined with a loop over the elements of $\mathbf{d}$ and $\mathbf{l}$. The low arithmetic intensity of the computations in the loop body makes an efficient memory management thereby particularly important for high performance.

Except for the eigenvector computation, on level $l = 2$ the inputs $\mathbf{d}$ and $\mathbf{l}$ are identical for all representation tree nodes (or threads). Thus, instead of loading the elements of $\mathbf{d}$ and $\mathbf{l}$ separately for every thread we can employ shared memory as user-managed cache. The expensive reads from global memory have then only to be performed once and we can hide latency further by using multiple threads in parallel to load the data as shown in Fig. 6.

However, even for small matrices shared memory is in general too small to store both $\mathbf{d}$ and $\mathbf{l}$ and the interval information for the current level of $\mathcal{T}$ (cf. Fig. 5). We therefore use `ShareMem.left` and `ShareMem.right` to cache the non-trivial matrix elements. The interval data is in the meantime stored in registers. Although "swapping" the data to registers has considerable overhead it is still beneficial because it allows to access $\mathbf{d}$ and $\mathbf{l}$ from shared instead of global memory during the computation of the $\mathbf{qds}$ transform.

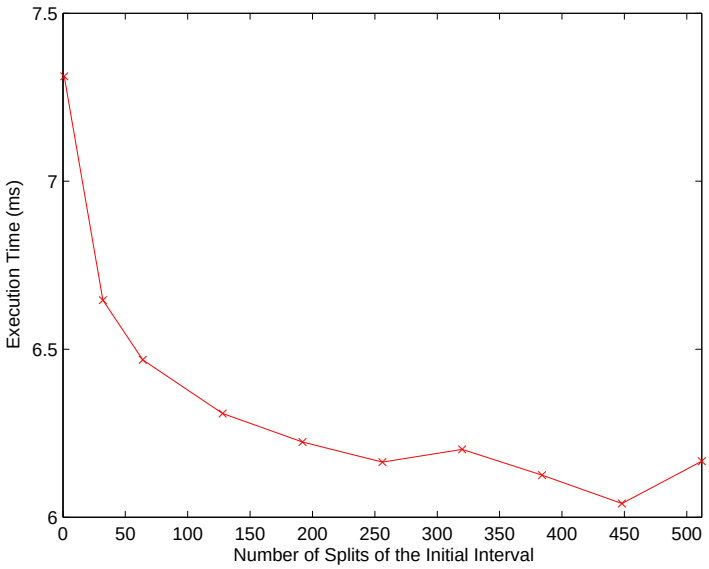

Figure 7: Execution time as a function of the number of interval splits used for the initial bisection step for a matrix with 512×512 elements. Employing 448 initial splits provides a speedup of 21%.

For all computations on levels $l > 2$ and for the eigenvector computation on levels $l \geq 2$, every representation tree node has a different shifted matrix as input. There is thus no shared information which can be cached and every thread has to read its input directly from global memory.

The **qds** transform includes a division (line 4 in Algo. 2 and Algo. 3) which can generate a NaN in the $i + 1$ iteration if the denominator becomes very small in iteration i [46]. Although in the literature it has been reported that this occurs only in rare cases [46] we observed the problem quiet frequently. Marques et al. [46] discuss different strategies to circumvent NaNs and in our current implementation we employ an approach devised by Demmel and Li [17]: First a naïve but fast implementation is used which does not account for the possibility of a division by zero and only if the final value of d^+ or r^+ is NaN (cf. Algo. 2 and Algo. 3) then a safe but more expensive version is executed which tests in every iteration if the denominator is less than some threshold t_{qds} and in this case sets it to the negative threshold value [46]. Note that we do *not* employ the try-and-fallback strategy for $l > 2$. There, the costs of the if-statement in the loop body are hidden by the high latency of reads from global memory.

5.1.3 Bisection

We employ bisection [33, p. 439] to classify eigenvalues as singletons or clusters and to refine eigenvalue approximations. The algorithm takes as input an interval $\mathrm{I} \subseteq \mathrm{G}_\mathbf{T}$ and recursively subdivides I until every non-empty interval is smaller than a predefined threshold. The resulting intervals are leaf nodes of a bisection tree. The Sturm count $s_\mathbf{T}(\mu)$ at the interval bounds is used to determine the number of eigenvalues in the interval. See for example the paper by Demmel et al. [16] for a more detailed discussion.

During the refinement of singletons, after every bisection step it is guaranteed that exactly one child interval is non-empty. The number of intervals remains thus constant and refined approximation can be computed by choosing the number of threads equal to the number of singletons.

When bisection is employed for classification, from all intervals generated on level l of the (unbalanced) bisection tree only non-empty ones are retained for level $l+1$. We use stream compaction [39] to remove empty intervals. First, the prefix sum algorithm [9] is employed to generate an index for every interval containing eigenvalues and during the compaction step this index provides the address for the data in shared memory [45].

We employ two optimizations in our implementation. With a two-fold subdivision every interval is guaranteed to have one non-empty child interval. Thus, only the set of second child intervals has to be compacted, reducing costs by half. Additionally, the compaction is only performed when necessary and on level $l + 1$ intervals with two non-empty child intervals exist; in particular on deep tree levels intervals are often only refined and no new ones are generated.

The convergence of an interval is currently determined using a combination of a relative and an absolute interval size criterion:

```
float t0 = right - left;
float t1 = max( abs(left), abs(right)) * t_c;

is_converged = ( t0 <= max( MIN_ABS_INTERVAL, t1)) 1 : 0;
```

where `left` and `right` are the left and right interval bounds, and `MIN_ABS_INTERVAL` is a minimal absolute interval size which is necessary if eigenvalues are close to zero; this occurs frequently for shifted matrices at representation tree levels $l > 2$.

The classic bisection algorithm creates an unbalanced *binary* tree starting with a single interval as root node. Parallelism arises thus only slowly and the hardware is exploited inefficiently at low tree levels. The problem can be alleviate by using multi-section but we believe that a k-nary tree would generate a considerable overhead, possibly outweighing the performance benefit obtained by additional parallelism. In our implementation we therefore subdivide only the root node of the bisection tree k times. This leads to an speedup of about 20% when k is close to n as shown in Fig. 7. After the set of intervals generated by the k-fold subdivision is compacted and empty intervals are removed, the regular bisection algorithm with two-fold subdivision is employed. For nodes at levels $l > 2$ the input to the bisection algorithm are multiple intervals containing usually only a small number of eigenvalues. We therefore do not employ multi-section in this case.

For the MRRR algorithm two thresholds t_c and t_r, for classification and refinement, respectively, have to be specified. In particular an appropriate choice of t_c is critical both for the accuracy and the performance of the MRRR algorithm. Choosing t_c too small can lead to non-orthogonal eigenvectors but choosing a large value for t_c can significantly increase the running time. We will discuss this issue in more detail in Sec. 6.

5.1.4 Distribution of the Computations across Multiprocessors

The representation tree of the MRRR algorithm has a single root node and it would therefore be natural to perform the computations on one multiprocessor. This would however exploit only a fraction of the available compute power. To circumvent this inefficiency we split the initial Gerschgorin interval $\mathrm{G_T}$ heuristically into K subintervals I_k and process each with one thread block, thereby distributing the work across multiprocessors. When $K = 2$, $\mathrm{G_T}$ is split symmetrically but with $K = 8$ the first and the last subinterval are chosen to be larger. The Gerschgorin interval provides only upper and lower bounds on the

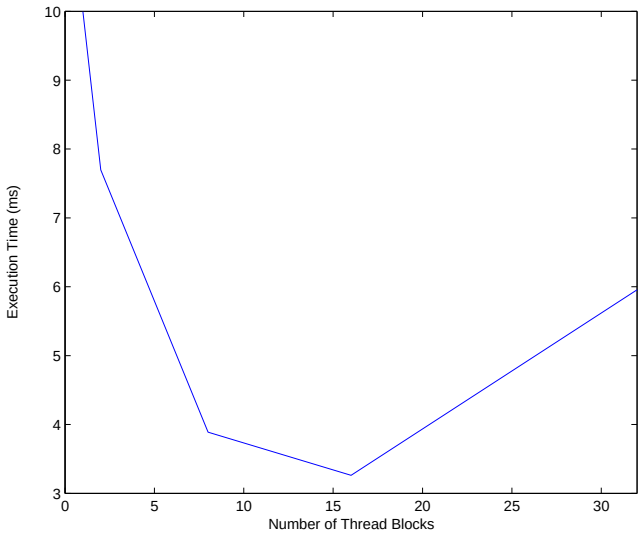

Figure 8: Execution time as function of thread blocks K for a matrix with 512×512 elements. On our data-parallel coprocessors with eight multiprocessors using 16 blocks provides a more than three-fold speedup over a naïve implementation with only one thread block.

spectrum of $\mathbf{T}$ and the "fringes" of $\mathrm{G}_{\mathbf{T}}$ contain usually far less eigenvalues than its center, making an asymmetric split beneficial.

Using multiple thread blocks (and thus multiprocessors) results in up to three-fold performance improvements as shown in Fig. 8. Interestingly, the highest speedup was obtained when $K = 16$, despite the fact that our test hardware had only eight multiprocessors. Note that the splitting of the initial Gerschgorin interval also allows to easily distribute the work across multiple coprocessors.

5.1.5 Memory Management

LAPACK's CPU implementation of the MRRR algorithm, `sstemr`, has memory requirements of $(n^2 + \mathcal{O}(n))$. For our implementation we require $(7n^2 + 4n)$ (global) memory. The large memory footprint is a direct consequence of the data-parallelism which is employed to speed up computations. For example, if all intervals on level 2 of the extended interval tree are classified as singletons then n double factorizations are obtained in parallel and for all of them $\mathbf{L}^+$ and $\mathbf{U}^+$ have to be stored for the eigenvector computation, requiring $2n^2$ memory. Additional n memory is necessary to determine the twist index k with $\mathrm{argmin}_k|\gamma_k|$. The indices in the $\mathbf{dstqds}$ and $\mathbf{dpqds}$ transforms (Algo. 2 and Algo. 3) run in opposite direction but for the computation of the twist index $\mathbf{s}^+$ and $\mathbf{p}^+$ are indexed in lock-step so that at least one of the auxiliary vectors $\mathbf{s}^+$ or $\mathbf{p}^+$ has to be stored in memory. In our implementation, we currently store $\mathbf{s}^+$ and determine k during the computation of $\mathbf{U}^+\mathbf{R}^+(\mathbf{U}^+)^T$. The remaining $4n^2$ memory are required to store the non-trivial elements of the representation tree nodes as well as the precomputed quantity lld, and the eigenvectors.

With CUDA it is in general crucial to coalesce global memory access. Interestingly, we

observed in our implementation that un-coalesced access was slightly faster than partially coalesced access.

5.2 Arbitrary Size Matrices

The limited size of shared memory prevents that the implementation discussed in the last section can be used for arbitrary size matrices. Minor modifications are however sufficient to allow the computation of eigenvalues and eigenvectors for matrices with $n > 512$.

The interval information can still be stored in shared memory if we require that the $\mathrm{I}_k \subset \mathrm{G}_k$ contain not more than 512 eigenvalues. For this, we initially split G_T into a fixed number of intervals and then recursively subdivide all intervals with more than 512 eigenvalues until the constraint is met.

For arbitrary size matrices, shared memory is too small to serve as user-managed cache for the $\mathbf{qds}$ transform even for level $l = 2$. Although we could read the matrix elements directly from global memory, this would leave shared memory unexploited and thus be inefficient. In our current implementation we load the input matrix block-wise to shared memory so that the data can still be read from fast on-chip memory during the computation of the $\mathbf{qds}$ transform on level $l = 2$. Loading the data in blocks provides an almost three-fold performance benefit over a naïve implementation which does not use shared memory.

At this point it should be noted that the significant memory requirements of our implementation (cf. Sec. 5.1.5) impede currently its use for true arbitrary size matrices. This would require a dynamic memory management where eigenvectors and eigenvalues are read to the host after they have been computed and where the global memory arrays are reused. We believe however that only minor modifications are necessary to incorporate such an advanced memory management.

5.3 Discussion

In contrast to traditional GPGPU programming [49] CUDA allows to develop data-parallel programs in a way very similar to CPU applications (cf. Sec. 3). Particularly useful for our implementation proved the ability to use template classes and to employ them transparently in both the host and device code. We developed for example an abstraction for CUDA's two-dimensional arrays which are otherwise often cumbersome to use because of a pitch which has to be considered for every data access. Our implementation consists of a host class which provides a convenient and safe interface to initialize and access the array on the CPU, and a device class which can be passed directly to a kernel and automatically takes the necessary pitch into account when the data is accessed. The same pattern of tightly linked host and device classes was successfully used also for other purposes in our implementation. Templates proved also to be useful when we extended our initial implementation for small matrices to arbitrary size matrices. Since the two versions differ only at minor points (cf. Sec. 5.2) we used a template argument for the kernel functions and static branching to distinguish between the two code paths, minimizing code-duplication to almost zero without performance penalty at runtime.

6 Experimental Evaluation

6.1 Experimental Setup

To asses the efficacy of our CUDA implementation of the MRRR algorithm, `mrrr_dp`, we compared it to CLAPACK's `sstemr` routine. Our test system was a Intel Pentium D CPU (3 GHz) with 1 GB RAM, and an NVIDIA GeForce 8800 GTX (driver version 169.07).

The operating system employed for the experiments was Fedora Core 6 and all programs were compiled using `gcc` 4.1.2. We used CUDA version 1.1 and CLAPACK version 3.1.1.

To determine the numerical accuracy of the computed eigenvalues we used LAPACK's double-precision function `dsterf`, which implements the highly accurate QR / QL algorithm [19], to obtain a ground truth and compared the results of both `mrrr_dp` and `sstemr` to those of `dsterf`. We also computed the residual of the eigen-decomposition $\|\mathbf{T}\mathbf{u}_i - \lambda_i \mathbf{u}_i\|$ and the error of $\langle \mathbf{u}_i, \mathbf{u}_j \rangle$ over all eigen-pairs. Reported is always the ℓ_∞ norm.

6.2 Parameters

We employed random matrices (`rand`) with elements in $[-1, 1]$ and the Wilkinson matrix (`wilkinson`) [18] as input for our experiments. Random matrices are interesting because they allow to estimate the robustness of an algorithm when different random inputs are used for otherwise fixed parameter settings. In the figures in Appendix A we report the mean, minimal and maximal errors over a set of random matrices for each parameter configuration. Note that the range of the random matrix elements is fixed independent of the matrix size. The eigenvalue clustering – and therefore the degree of difficulty for the MRRR algorithm – increases therefore with increasing matrix size. It is known that the Wilkinson matrix has significant eigenvalue clustering and we therefore employed it as stress test for our implementation.

Two different sets of matrix sizes have been employed for the experiments, "arbitrary" size matrices with $n \in [32, 4096]$ and "small" matrices with $n \in [32, 512]$. The latter category gave us more flexibility in choosing the parameter t_c without affecting the correctness of our implementation, whereas the first category is better suited to show the asymptotic behaviour of `mrrr_dp` and `sstemr`. Matrix sizes were both generated randomly and predetermined.

The performance numbers shown in Appendix A are without preprocessing times on the CPU. These were usually negligible compared to the overall execution times.

6.3 Arbitrary Size Matrices

Fig. 9 to Fig. 16 show the experimental results for `rand` and `wilkinson` with $n \in [32, 4096]$ and 16 and 8 multiprocessors, respectively. Fig. 9 and Fig. 10 show clearly the superior performance of `mrrr_dp` with an up to 16-fold speedup over `sstemr` . The width of the quadratic is for `mrrr_dp` thereby significantly larger than for `sstemr` so that even higher speedups can be expected for larger matrices.

The accuracy of `mrrr_dp` and `sstemr` is shown in Fig. 11 to Fig. 16. The results reveal that `sstemr` is usually an order of magnitude more accurate than `mrrr_dp` . It is interesting to not that for the Wilkinson matrix the eigenvalues obtained with `mrrr_dp` are more accurate than those of `sstemr` .

6.4 Small Matrices

Performance graphs for `rand` with $n \in [32, 512]$ and different values of t_c are presented in Fig. 17 and Fig. 18. The results show that the choice of t_c significantly affects the performance of `mrrr_dp` and that a minimal execution time is obtained when classification and refinement threshold are chosen equal. Our implementation then provides a 50-fold speedup over the CPU. LAPACK's `sstemr` routine outperforms `mrrr_dp` only for very small matrices with $n = 32$. The accuracy of `mrrr_dp` and `sstemr` for `rand` and $n \in [32, 512]$ is shown in Fig. 19 to Fig. 24. It can be observed that the choice of t_c is here

largely irrelevant and `mrrr_dp` yields the same accuracy independent of t_c. `sstemr` is however again about an order of magnitude more accurate.

It is interesting to note that for $n \in [32, 512]$ the performance of `mrrr_dp` scales linearly in contrast to the quadratic complexity of the MRRR algorithm which clearly governs the performance of `sstemr`. This suggests that the data-parallel coprocessor is not fully used for small matrices. With increasing matrix size more and more parallelism is utilized which compensates for one $\mathcal{O}(n)$ factor.

In Fig. 25 to Fig. 32 results for the Wilkinson matrix with $n \in [32, 512]$ and different values of t_c are shown, largely resembling those for `rand` . The eigenvalues obtained with `mrrr_dp` are however again approximately as accurate as those of `sstemr` .

6.5 Discussion

Our experimental results demonstrate that the MRRR algorithm can be mapped efficiently onto a data-parallel coprocessor and that significant speedups over optimized CPU implementations are possible. Currently, `mrrr_dp` provides however not the accuracy and robustness which is required for many applications. `sstegr`, LAPACK's original implementation of the MRRR algorithm, suffered from similar problems and we believe that it will be possible to incorporate the improvements from `sstegr` to `sstemr` into our implementation. Note that the performance will not necessarily be affected negatively by these changes; it has for example be shown that an improved representation tree can reduce the computation time [64]. We currently do not believe that the limited accuracy of `mrrr_dp` results from CUDA hardware limitations but so far did not explore this question in detail.

The presented results show that the efficacy of `mrrr_dp` depends on an appropriate choice of t_c. Choosing a small value close or equal to the refinement threshold allows higher performance but incurs the risk of non-orthogonal eigenvectors, whereas a large value of t_c yields a more robust algorithm at the cost of reduced performance. For example choosing t_c and t_r equal for $n \in [32, 4096]$ would result in non-orthogonal eigenvectors for large n due to the increasing eigenvalue clustering for `rand`. More work will be necessary to – ideally automatically – determine values of t_c which are appropriate for the input matrix at hand.

In Sec. 5 we reported that the performance of `mrrr_dp` can be improved significantly by using multiple thread blocks (and hence multiple multiprocessors) and by using multi-section instead of bisection for the initial eigenvalue classification. Our experiments showed however that these optimizations have to be used with care. When multiple thread blocks are employed the intervals I_k are currently determined irrespectively of the eigenvalue distribution of the input matrix. One cluster can thus be distributed across multiple intervals. The correct classification of eigenvalues is however only possible if clusters are treated as being atomic. For the Wilkinson matrix this made it necessary to use only 8 instead of 16 multiprocessor for $n \in [32, 4096]$. When multi-section is performed too aggressively, the resulting intervals might be smaller than t_c, preventing that clustered eigenvalues are isolated before eigenvectors are computed. In the experiments we thus did not use multi-section for $n \in [32, 4096]$.

7 Future Work

The implementation of the MRRR algorithm presented in this report can be improved in a variety of ways. Of great importance for the practicality of an eigen-solver is the accuracy of the obtained results. In this respect our implementation is clearly inferior to LAPACK's `sstemr` routine. We believe however that it will be relatively easy to incorporate the improvements over the basic MRRR algorithm in `sstemr` into our own implementa-

tion. These changes might affect performance but we do not expect a dramatically different runtime or that the conclusions presented in this paper have to be altered (cf. Sec. 5.3).

Similar to our implementation, LAPACK's `sstemr` routine employs the try-and-fallback strategy proposed by Demmel and Li [17] for the **qds** transform (cf. Section 5.1.2). In `sstemr` the loop is however subdivided into blocks so that after every k^{th} iteration a test for a NaN is performed and accordingly only the previous block has to be re-computed with the safe implementation if that was the case. So far we have not investigated if such a blocking is also beneficial on a data-parallel coprocessor.

Efficient data and memory management is one of the most difficult aspects of CUDA and there are a variety of ways to improve our current implementation. For very small matrices it would be possible to use shared memory much more extensively, for example by reading the input matrix only once into shared memory and retaining it there, and also to employ more registers per thread which should reduce the spilling to global memory which occurs at the moment quiet extensively. Independent of any other factor, in our experience coalescing global memory access always improved performance. We believe that it will therefore be worthwhile to investigate in more detail why partial coalescing deteriorates performance for our MRRR implementation.

For an effective load balancing across multiprocessors an appropriate splitting of the Gerschgorin interval is crucial. Currently, we subdivide $\mathrm{G}_{\mathbf{T}}$ only heuristically and so far did not investigate different strategies. We believe however that a thorough analysis will be valuable; for example, to determine what an effective size for the "fringe" intervals is. The splitting of the Gerschgorin interval is currently performed on the CPU. Although this does not incur significant overhead, it will in some cases be beneficial to perform all computations on the device; for example when the input matrix is already in device memory or when the host is used for other computations.

A major limitation of our implementation are the significant memory requirements. A mild reduction could be obtained by not pre-computing lld although we believe that it will be very challenging to achieve further improvements without reducing the parallelism and possibly affecting performance. Note however that for current data-parallel coprocessors with at most eight multiprocessors the memory requirements are bound by $(7 \times (4096)^2 + 4 \times 4096)$ independent of the matrix size because at most $8 \times 512 = 4096$ eigen-pairs can be "in flight" at the same time on the device. With the modifications we hinted at in Sec. 5.1.5 it will thus in fact be possible to compute eigen-decompositions for true arbitrary size matrices even with the current memory requirements.

The MRRR algorithm operates on a symmetric, tridiagonal matrix. Eigenvalue problems arising in practice involve however often unreduced matrices. CPU implementations of Householder transformations are available but we believe that CUDA can provide significant performance benefits for these computations.

In many applications only a subset of k eigen-pairs is of interest. It would thus be worthwhile to extent the current implementation to allow an efficient subset computation. Care is however required when $k \ll n$ because the available parallelism might then not be sufficient to provide a performance benefit over an implementation on a serial processor. We observed a similar problem in the current implementation for deep levels of the representation tree where the number of nodes, and therefore the available parallelism, is usually very small. We believe that it is in this case more efficient to read an almost complete eigen-decomposition back to the host and to resolve the remaining clusters there.

We currently use bisection with its linear convergence to classify and refine eigenvalues. LAPACK's `sstemr` routine employs the **dqds** transform which has quadratic convergence if it is sufficiently close to the true eigenvalue and bisection is only employed when the **dqds** algorithm fails. Until now we favored the bisection algorithm over the **dqds**

transform for its simplicity and because it is guaranteed to succeed. A more thorough comparison of the two algorithms would however be valuable.

Recently, Vömel [64] suggested an improved representation tree for the MRRR algorithm which provides better results in particular for matrices with many clusters [65]. We believe that it would be beneficial to employ the new tree also in our data-parallel implementation.

The recent comparison of LAPACK's eigen-solvers by Demmel et al. [18, 19] and later results by Vömel [64] showed that clustered matrices which are particularly difficult for the MRRR algorithm are relatively easy for the Divide and Conquer algorithm, and that matrices which are difficult for Divide and Conquer are tackled with less effort by MRRR. It would therefore be interesting to investigate the possibility to implement the Divide and Conquer algorithm on a data-parallel coprocessor. With efficient CUDA implementations of the MRRR *and* the Divide and Conquer algorithm a viable alternative to LAPACK's eigen-solvers could be provided which needs only a fraction of the computation time.

Based on the MRRR algorithm, Großer and Lang [35, 34, 36, 37] developed an algorithm for the bidiagonal singular value decomposition (SVD). The importance of the SVD in many fields would justify to extend our current implementation to the algorithm proposed by Großer and Lang.

8 Conclusion

We reported on an implementation of the Algorithm of Multiple Relatively Robust Representations (MRRR) for the symmetric tridiagonal eigenvalue problem on a data-parallel coprocessor using the CUDA programming environment. Our results demonstrate that the algorithm maps well onto a data-parallel architecture and we achieved up to 50-fold speedups over LAPACK's `sstemr` routine. Although our accuracy lacks currently behind those of LAPACK we believe that these problems can be overcome in the near future.

The source code of our implementation is available online.

Acknowledgement

An initial version of the bisection algorithm was implement during an internship at NVIDIA in Summer 2007 and the idea to implement the MRRR algorithm with CUDA was also born during this time. I want to thank my colleagues at NVIDIA in London for fruitful discussions and for providing hardware.

I also wish to acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) for funding this basic research project.

References

[1] ALPATOV, P., BAKER, G., EDWARDS, C., GUNNELS, J., MORROW, G., OVERFELT, J., VAN DE GEIJN, R., AND WU, Y.-J. J. PLAPACK: Parallel Linear Algebra Package Design Overview. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 1997), ACM, pp. 1–16.

[2] AMERICAN NATIONAL STANDARDS INSTITUTE. *American National Standard Programming Language C, ANSI X3.159-1989"*. 1430 Broadway, New York, NY 10018, USA, Dec. 1989.

[3] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[4] BERRY, M., AND SAMEH, A. H. Multiprocessor Jacobi Algorithms for Dense Symmetric Eigenvalue and Singular Value Decompositions. In *ICPP* (1986), pp. 433–440.

[5] BIENTINESI, P., DHILLON, I. S., AND VAN DE GEIJN, R. A. A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations. *SIAM J. Sci. Comput. 27*, 1 (2005), 43–66.

[6] BISCHOF, C., AND VAN LOAN, C. The WY Representation for Products of Householder Matrices. *SIAM J. Sci. Stat. Comput. 8*, 1 (1987), 2–13.

[7] BISCHOF, C. H. The Two-Sided Block Jacobi Method on Hypercube Architectures. In *Hypercube Multiprocessors*, M. T. Heath, Ed. SIAM Publications, Philadelphia, 1987.

[8] BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[9] BLELLOCH, G. E. Prefix Sums and Their Applications. In *John H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann*. 1993.

[10] BRENT, R. P., AND LUK, F. T. The Solution of Singular Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays. 69–84.

[11] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3 (2004), 777–786.

[12] BYRON, F. W., AND FULLER, R. W. *Mathematics of Classical and Quantum Physics*. Dover Publications, 1964 (1992).

[13] CLEARY, A., AND DONGARRA, J. Implementation in ScaLAPACK of Divide-and-Conquer Algorithms forBanded and Tridiagonal Linear Systems. Tech. rep., Knoxville, TN, USA, 1997.

[14] CUPPEN, J. J. M. A Divide and Conquer Method for the Symmetric Eigenproblem. 177–195.

[15] DEMMEL, J., AND VESELIC, K. Jacobi's Method is More Accurate Than QR. Tech. rep., Knoxville, TN, USA, 1989.

[16] DEMMEL, J. W., DHILLON, I., AND REN, H. On the Correctness of Some Bisection-Like Parallel Eigenvalue Algorithms in Floating Point Arithmetic. *Electron. Trans. Numer. Anal. 3* (1995), 116–149.

[17] DEMMEL, J. W., AND LI, X. Faster Numerical Algorithms Via Exception Handling. *IEEE Trans. Comput. 43*, 8 (1994), 983–992.

[18] DEMMEL, J. W., MARQUES, O. A., PARLETT, B. N., AND VÖMEL, C. LAPACK Working Note 182: A Testing Infrastructure for LAPACK's Symmetric Eigensolvers. Tech. rep., EECS Department, University of California, Berkeley, Apr 2007.

[19] DEMMEL, J. W., MARQUES, O. A., PARLETT, B. N., AND VÖMEL, C. LAPACK Working Note 183: Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers. Tech. rep., EECS Department, University of California, Berkeley, Apr 2007.

[20] DHILLON, I. Current Inverse Iteration Software Can Fail. *BIT Numerical Mathematics 38*, 4 (1998), 685–704.

[21] DHILLON, I. S. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*. PhD thesis, EECS Department, University of California, Berkeley, 1997.

[22] DHILLON, I. S., AND PARLETT, B. N. Orthogonal Eigenvectors and Relative Gaps. *SIAM J. Matrix Anal. Appl. 25*, 3 (2003), 858–899.

[23] DHILLON, I. S., AND PARLETT, B. N. Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices. *Linear Algebra and its Applications 387*, 1 (Aug. 2004), 1–28.

[24] DHILLON, I. S., PARLETT, B. N., AND VÖMEL, C. The Design and Implementation of the MRRR Algorithm. *ACM Trans. Math. Softw. 32*, 4 (2006), 533–560.

[25] DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., AND DUFF, I. Algorithm 679: A set of level 3 Basic Linear Algebra Subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software 16*, 1 (Mar. 1990), 18–28.

[26] ELDÈN, L. A Parallel QR Decomposition Algorithm. Tech. Rep. LiTh Mat R 1988-02, Mathematics, Linkoping University, Sweden, 1988.

[27] ELWOOD, D., FANN, G., AND LITTLEFIELD, D. PeIGS User's Manual. Tech. rep., Pacific Northwest National Laboratory, 1993.

[28] EMAD, N. Data Parallel Lanczos and Pad'e-Rayleigh-Ritz Methods on the CM5. Tech. rep., Laboratoire PRiSM, Université Versaille St. Quentin, 1999.

[29] FERNANDO, K. V. Accurately Counting Singular Values of Bidiagonal Matrices and Eigenvalues of Skew-Symmetric Tridiagonal Matrices. *SIAM J. Matrix Anal. Appl. 20*, 2 (1999), 373–399.

[30] FRANCIS, J. G. F. The QR transformation: A unitary analogue to the LR transformation, parts I and II. 265–272, 332–345.

[31] GIVENS, W. J. Numerical Computation of the Characteristic Values of a Real Symmetric Matrix. Tech. rep., Oak Ridge National Laboratory, 1954.

[32] GOLUB, G. H., AND LOAN, C. F. V. *Matrix Computations*, 2nd ed. Johns Hopkins University Press, Baltimore, 1983.

[33] GOLUB, G. H., AND LOAN, C. F. V. *Matrix Computations*, third ed. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[34] GROSSER, B. *Ein paralleler und hochgenauer $O(n^2)$ Algorithmus für die bidiagonaler Singulärwertzerlegung*. PhD thesis, Bergische Universität Wuppertal, Fachbereich Mathematik, Wuppertal, Germany, 2001. In German.

[35] GROSSER, B., AND LANG, B. Efficient Parallel Reduction to Bidiagonal Form. *Parallel Comput. 25*, 8 (1999), 969–986.

[36] GROSSER, B., AND LANG, B. An $O(n^2)$ Algorithm for the Bidiagonal SVD. *Linear Algebra and its Applications 358*, 1–3 (Jan. 2003), 45–70.

[37] GROSSER, B., AND LANG, B. On Symmetric Eigenproblems Induced by the Bidiagonal SVD. *SIAM J. Matrix Anal. Appl. 26*, 3 (2005), 599–620.

[38] GU, M., AND EISENSTAT, S. C. A Divide-and-Conquer Algorithm for the Symmetric TridiagonalEigenproblem. *SIAM J. Matrix Anal. Appl. 16*, 1 (1995), 172–191.

[39] HARRIS, M., SENGUPTA, S., AND OWENS, J. D. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Aug. 2007.

[40] JACOBI, C. G. J. Über ein leichtes verfahren die in der theorie der säculärstörungen vorkommenden gleichungen numerisch aufzulösen. 51–94.

[41] JUFFA, N. *CUDA CUBLAS Library*, first ed. NVIDIA Corporation, 2701 San Toman Expressway, Santa Clara, CA 95050, USA, 2007.

[42] KAUFMAN, L. A parallel QR algorithm for the symmetric tridiagonal eigenvalue problem. *J. Parallel Distrib. Comput. 23*, 3 (1994), 429–434.

[43] KUBLANOVSKAYA, V. N. On Some Algorithms for the Solution of the Complete Eigenvalue Problem. 637–657.

[44] LAPACK. LAPACK Working Note Directory. http://www.netlib.org/lapack/lawns/downloads/.

[45] LESSIG, C. Eigenvalue Computation with CUDA. Tech. rep., NVIDIA Corporation, August 2007.

[46] MARQUES, O. A., RIEDY, E. J., AND VÖMEL, C. LAPACK Working Note 172: Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers. Tech. Rep. UCB/CSD-05-1414, EECS Department, University of California, Berkeley, Sep 2005.

[47] MCCOOL, M. D., QIN, Z., AND POPA, T. S. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 57–68.

[48] NVIDIA CORPORATION. *CUDA Programming Guide*, first ed. NVIDIA Corporation, 2701 San Toman Expressway, Santa Clara, CA 95050, USA, 2007.

[49] OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A. E., AND PURCELL, T. J. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports* (Aug. 2005), pp. 21–51.

[50] PARLETT, B. N. The new *qd* algorithms. 1995, pp. 459–491.

[51] PARLETT, B. N. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[52] PARLETT, B. N., AND DHILLON, I. S. Fernando's solution to Wilkinson's problem: An application of double factorization. *Linear Algebra and its Applications 267* (1997), 247–279.

[53] PARLETT, B. N., AND DHILLON, I. S. Relatively Robust Representations of Symmetric Tridiagonals. *Linear Algebra and its Applications 309*, 1–3 (Apr. 2000), 121–151.

[54] PARLETT, B. N., AND VÖMEL, C. LAPACK Working Note 163: How the MRRR Algorithm Can Fail on Tight Eigenvalue Clusters. Tech. Rep. UCB/CSD-04-1367, EECS Department, University of California, Berkeley, 2004.

[55] PEERCY, M., SEGAL, M., AND GERSTMANN, D. A performance-oriented data parallel virtual machine for gpus. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches* (New York, NY, USA, 2006), ACM, p. 184.

[56] PETERS, G., AND WILKINSON, J. H. The Calculation of Specified Eigenvectors by Inverse Iteration. In *Handbook for Automatic Computation Vol. 2: Linear Algebra*, J. H. Wilkinson and C. Reinsch, Eds. New York, NY, USA, 1971, pp. 418–439.

[57] PETITION, S. G. Parallel QR Algorithm for Iterative Subspace Methods on the Connection Machine (CM2). In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing* (Philadelphia, PA, USA, 1990), Society for Industrial and Applied Mathematics, pp. 42–47.

[58] PETITON, S. G. Parallel subspace method for non-Hermitian eigenproblems on the Connection Machine (CM2). In *Selected papers from the symposia on CWI-IMACS symposia on parallel scientific computing* (Amsterdam, The Netherlands, The Netherlands, 1992), Elsevier North-Holland, Inc., pp. 19–35.

[59] RALHA, R. M. S. Parallel QR algorithm for the complete eigensystem of symmetric matrices. In *PDP '95: Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing* (Washington, DC, USA, 1995), IEEE Computer Society, p. 480.

[60] RUTTER, J. D. A Serial Implementation of Cuppen's Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem. Tech. Rep. UCB/CSD-94-799, EECS Department, University of California, Berkeley, 1994.

[61] STROUSTRUP, B. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[62] TUCKER, L. W., AND ROBERTSON, G. G. Architecture and applications of the connection machine. *Computer 21*, 8 (1988), 26–38.

[63] VAN DE GEIJN, R. A. Storage Schemes for Parallel Eigenvalue Algorithms. Tech. rep., Austin, TX, USA, 1988.

[64] VÖMEL, C. LAPACK Working Note 194: A Refined Representation Tree for MRRR. Tech. rep., EECS Department, University of California, Berkeley, Nov 2007.

[65] VÖMEL, C. LAPACK Working Note 195: ScaLAPACK's MRRR Algorithm. Tech. rep., EECS Department, University of California, Berkeley, Nov 2007.

[66] WILLEMS, P. R., LANG, B., AND VÖMEL, C. LAPACK Working Note 166: Computing the Bidiagonal SVD Using Multiple relatively robust representations. Tech. Rep. UCB/CSD-05-1376, EECS Department, University of California, Berkeley, May 2005.

A Evaluation Results

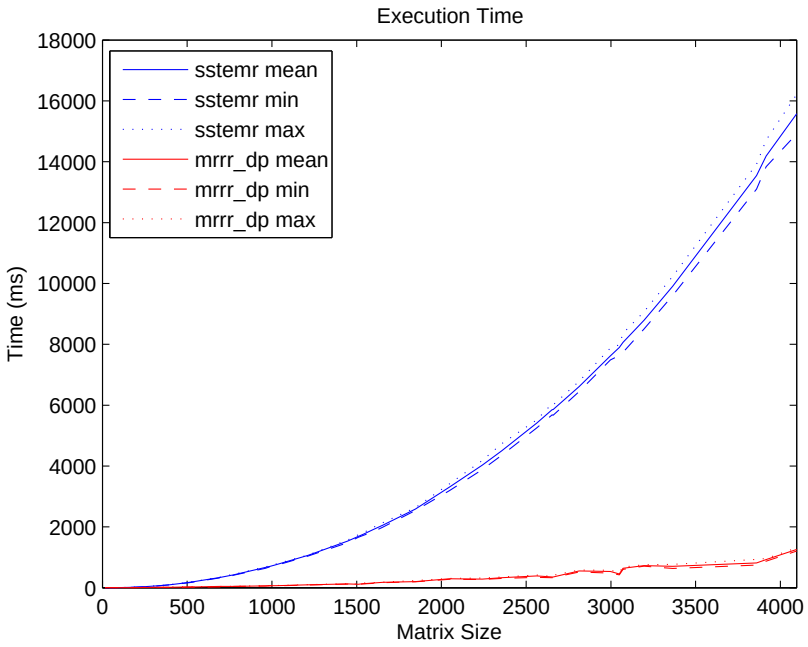

Figure 9: Mean / min / max execution time for 32 random matrices for each matrix size n and $n \in [32, 4096]$ with $t_c = 0.01$.

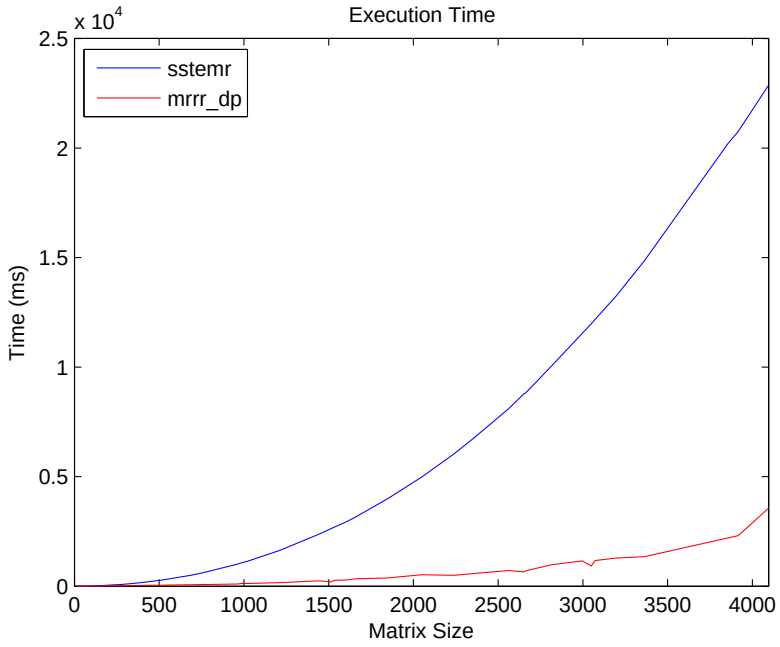

Figure 10: Execution time for the Wilkinson matrix and $n \in [32, 4096]$ with $t_c = 0.01$.

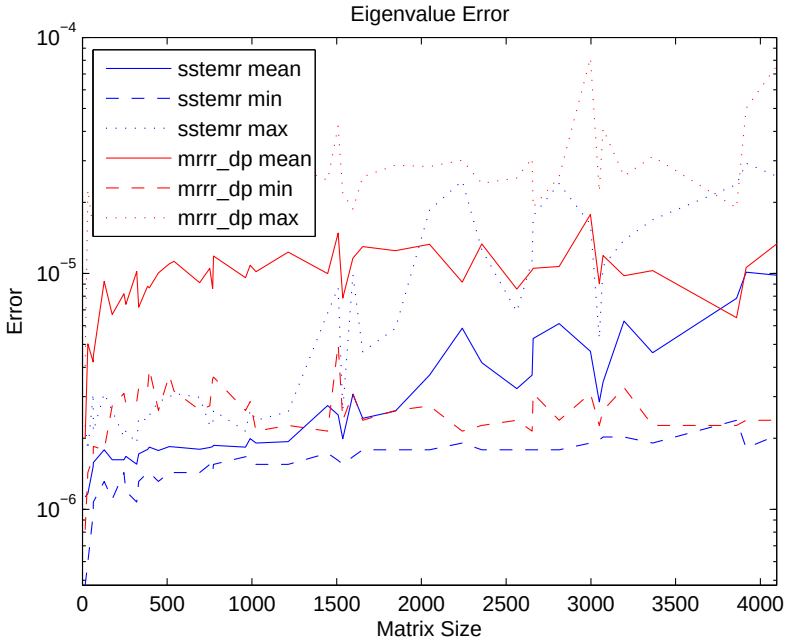

Figure 11: Mean / min / max ℓ_∞ error of eigenvalues for 32 random matrices for each matrix size n and $n \in [32, 4096]$ with $t_c = 0.01$.

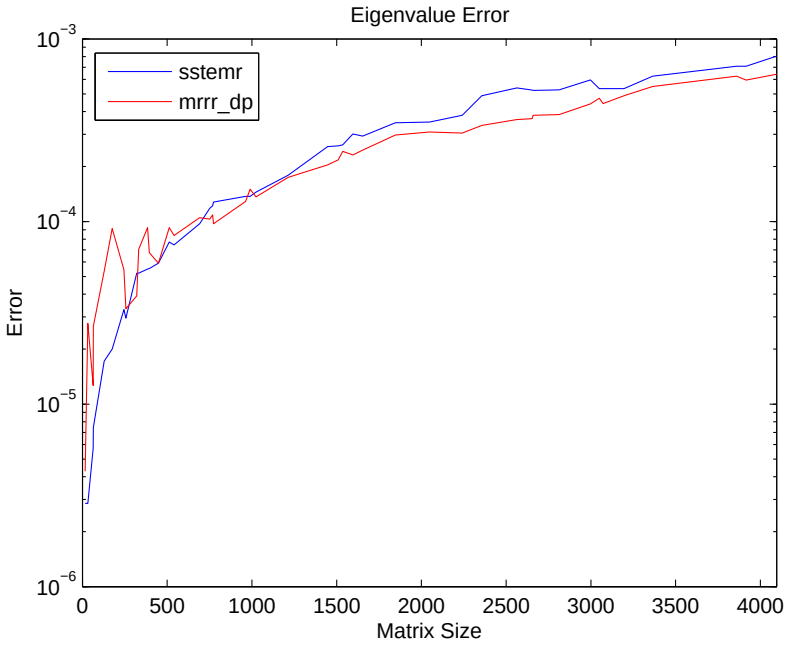

Figure 12: ℓ_∞ norm of the eigenvalues for the Wilkinson matrix and $n \in [32, 4096]$ with $t_c = 0.01$.

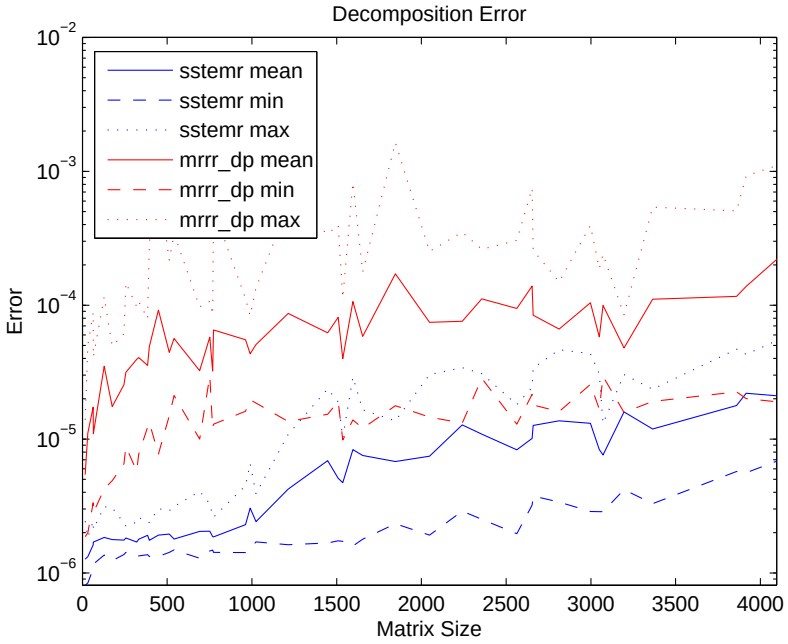

Figure 13: Mean / min / max ℓ_∞ error of the eigen-decomposition for 32 random matrices for each matrix size n and $n \in [32, 4096]$ with $t_c = 0.01$.

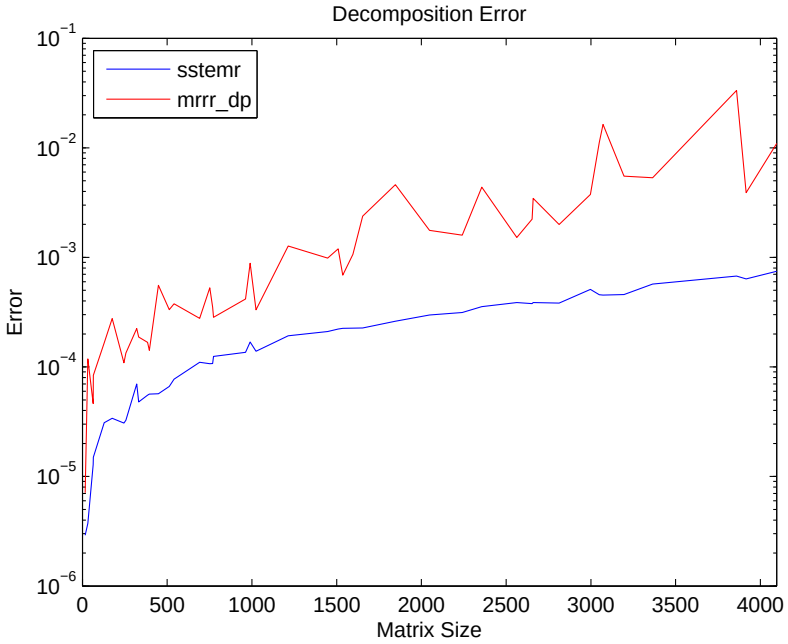

Figure 14: ℓ_∞ norm of the eigen-decomposition for the Wilkinson matrix and $n \in [32, 4096]$ with $t_c = 0.01$.

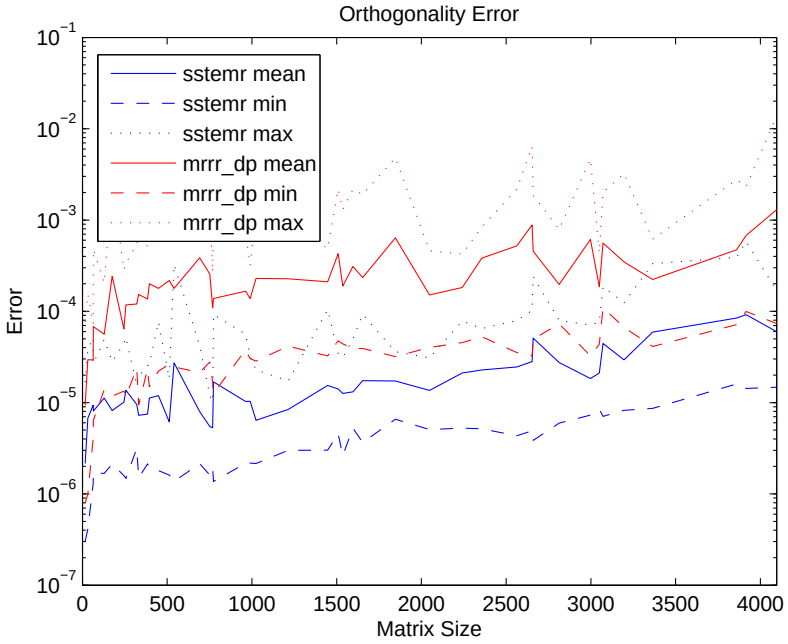

Figure 15: Mean / min / max ℓ_∞ error in the orthogonality of the eigenvectors for 32 random matrices for each matrix size n and $n \in [32, 4096]$ with $t_c = 0.01$.

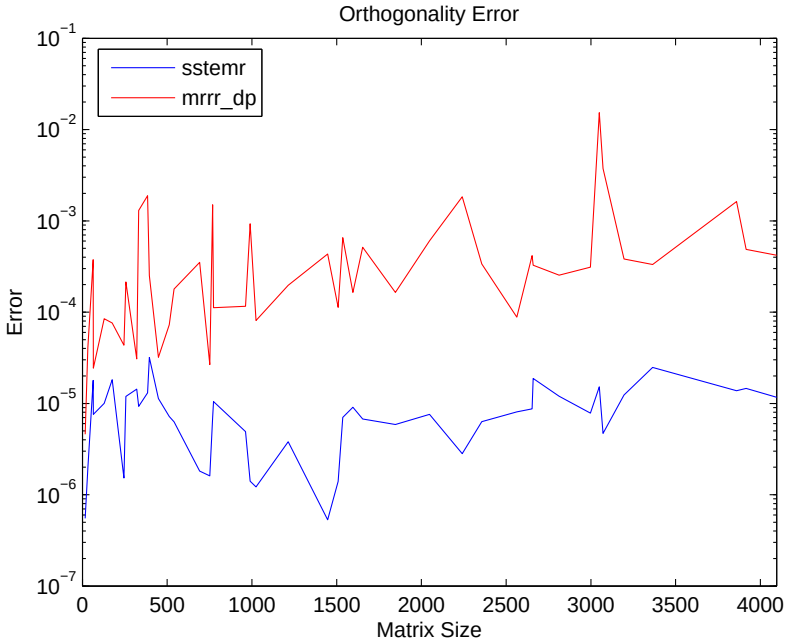

Figure 16: ℓ_∞ norm in the orthogonality of eigenvectors for the Wilkinson matrix and $n \in [32, 4096]$ with $t_c = 0.01$.

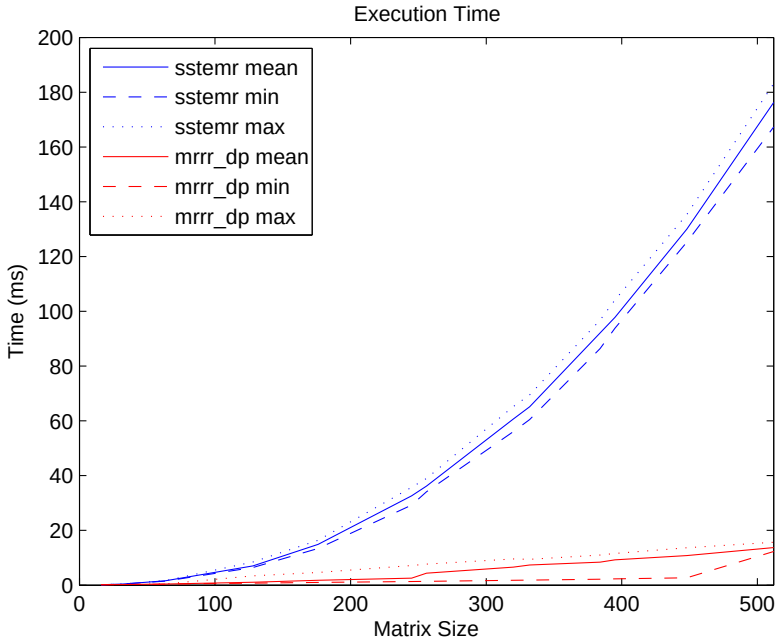

Figure 17: Mean / min / max execution time for 32 random matrices for each matrix size n and $n \in [32, 512]$ with $t_c = 0.0001$.

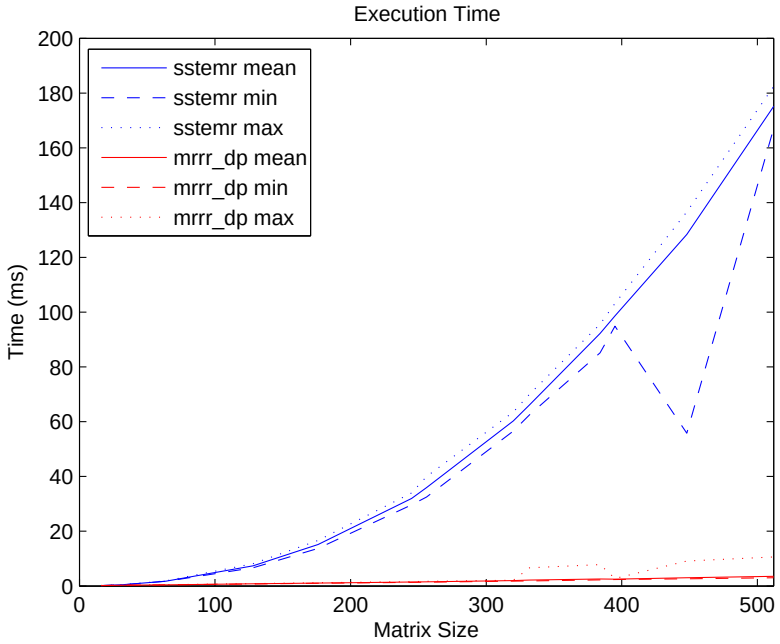

Figure 18: Mean / min / max execution time for 32 random matrices for each matrix size n and $n \in [32, 512]$ with $t_c = t_r = 0.000001$.

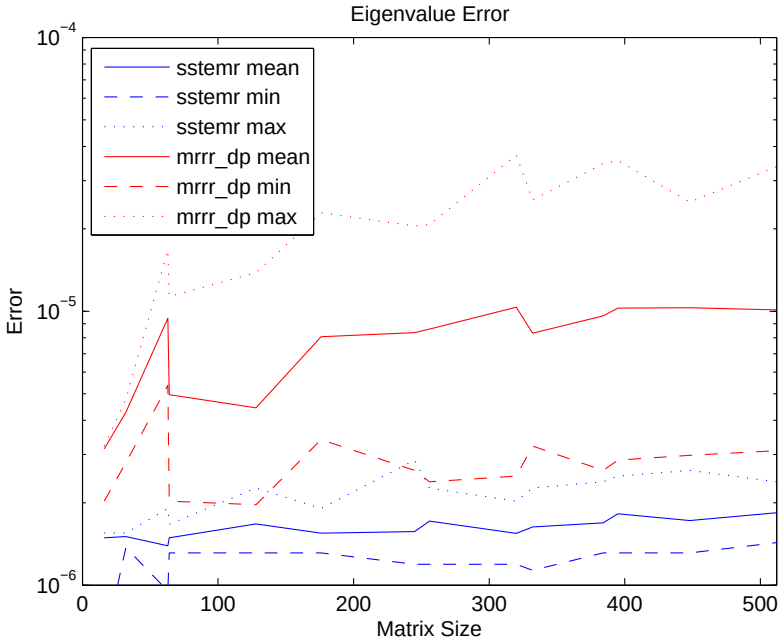

Figure 19: Mean / min / max ℓ_∞ error of the eigenvalues for 32 random matrices for each matrix size n and $n \in [32, 512]$ with $t_c = 0.0001$.

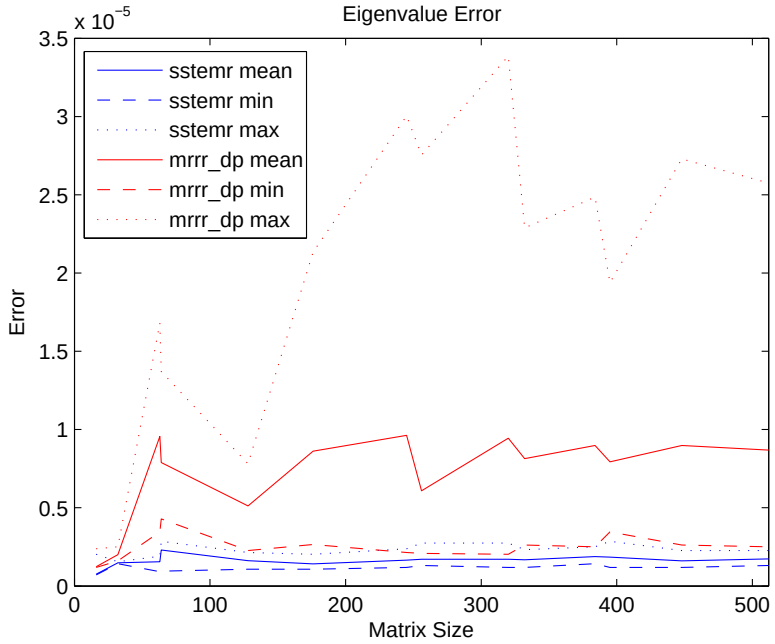

Figure 20: Mean / min / max ℓ_∞ error of the eigenvalues for 32 random matrices for each matrix size n and $n \in [32, 512]$ with $t_c = t_r = 0.000001$.

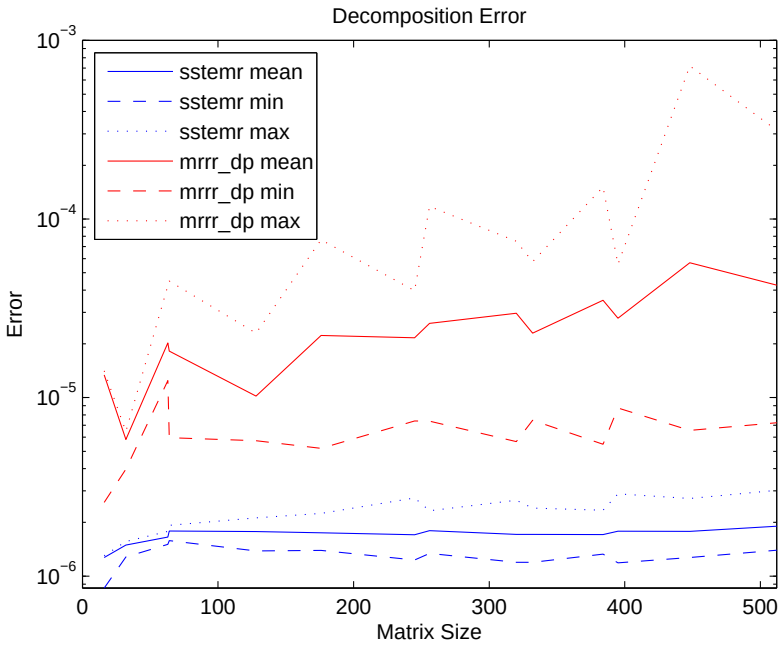

Figure 21: Mean / min / max ℓ_∞ error of the eigen-decomposition for 32 random matrices for each matrix size n and $n \in [32, 512]$ with $t_c = 0.0001$.

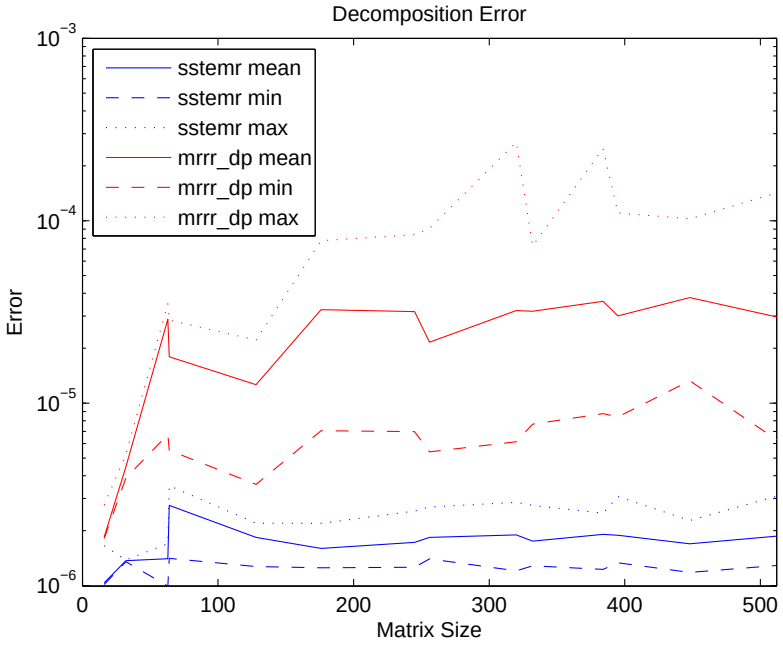

Figure 22: Mean / min / max ℓ_∞ error of the eigen-decomposition for 32 random matrices for each matrix size n and $n \in [32, 512]$ with $t_c = t_r = 0.000001$.

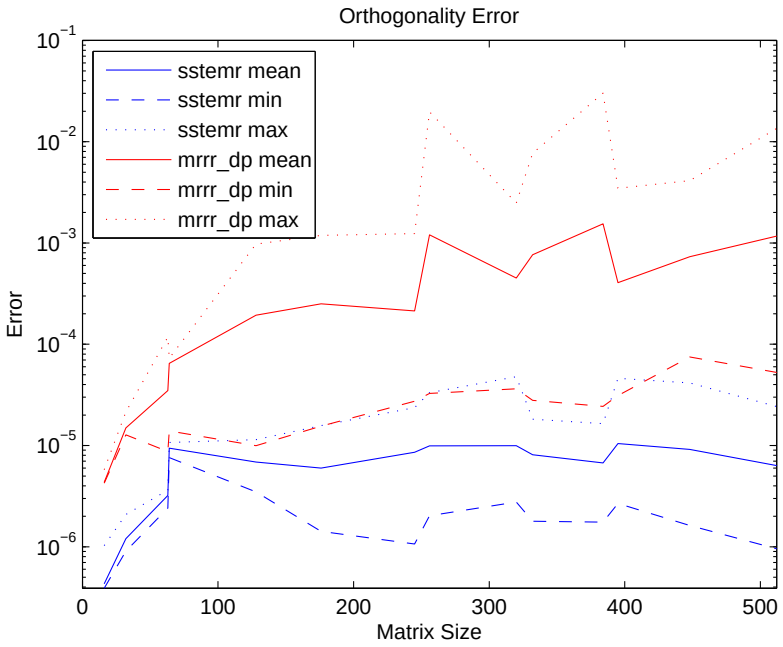

Figure 23: Mean / min / max ℓ_∞ error in the orthogonality of the eigenvectors for 32 random matrices for each matrix size n and $n \in [32, 512]$ with $t_c = 0.0001$.

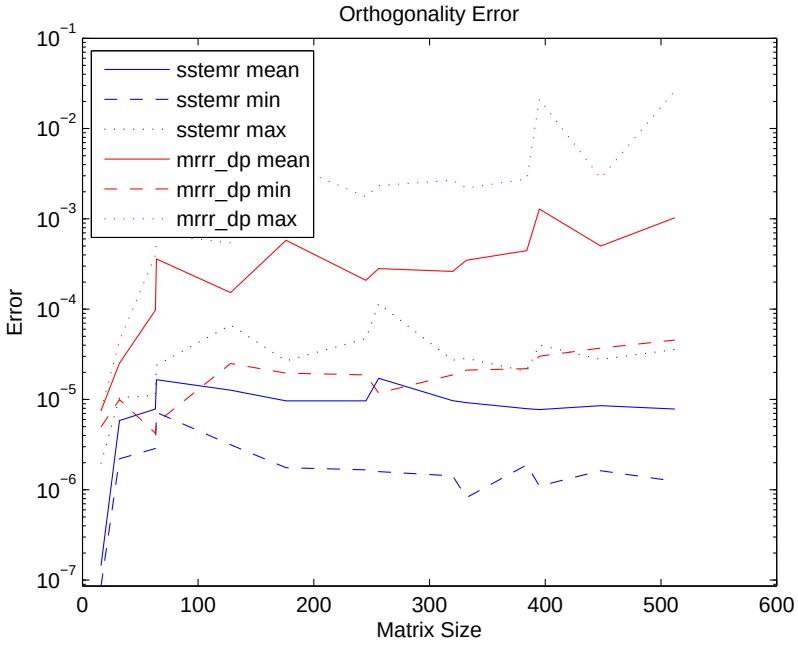

Figure 24: Mean / min / max ℓ_∞ error in the orthogonality of the eigenvectors for 32 random matrices for each matrix size n and $n \in [32, 512]$ with $t_c = t_r = 0.000001$.

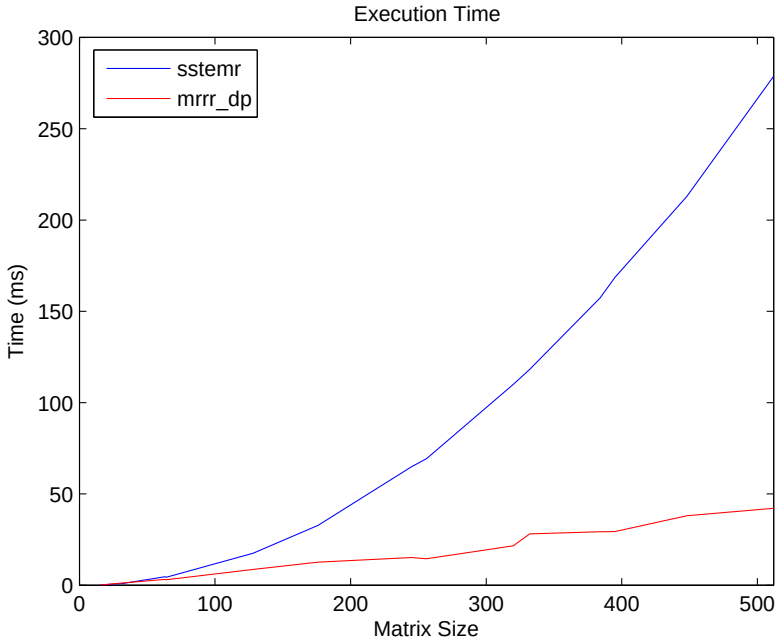

Figure 25: Execution time for the Wilkinson matrix for $n \in [32, 512]$ with $t_c = 0.01$.

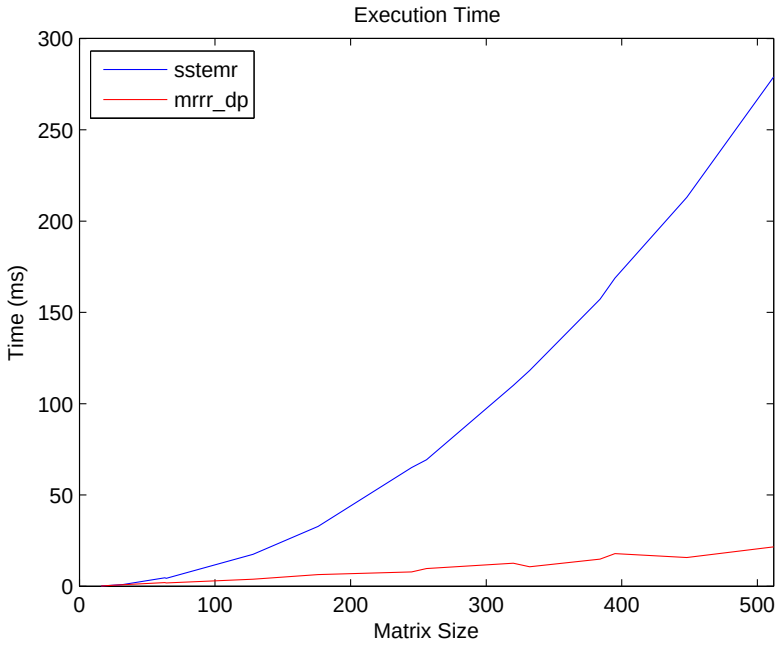

Figure 26: Execution time for the Wilkinson matrix and $n \in [32, 512]$ with $t_c = 0.000001$.

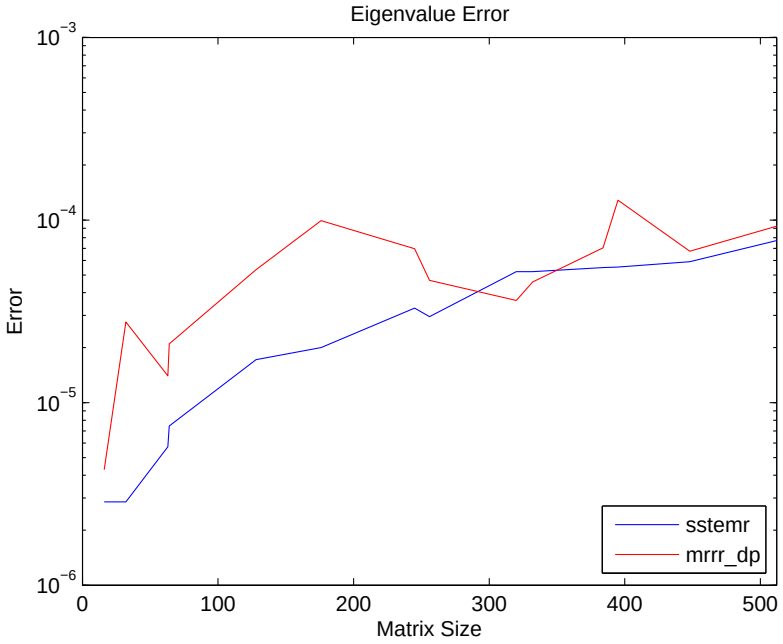

Figure 27: ℓ_∞ norm of the eigenvalues for the Wilkinson matrix and $n \in [32, 512]$ with $t_c = 0.01$.

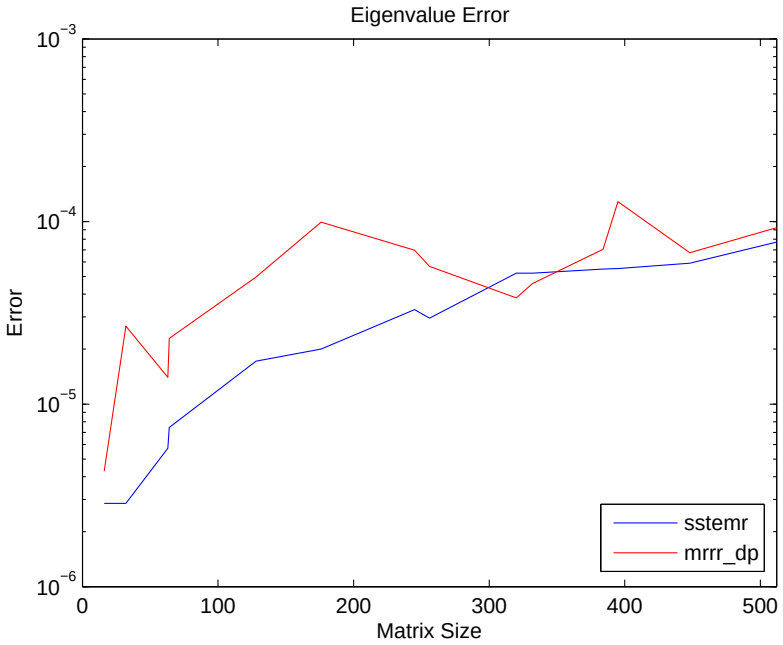

Figure 28: ℓ_∞ norm of the eigenvalues for the Wilkinson matrix and $n \in [32, 512]$ with $t_c = 0.000001$.

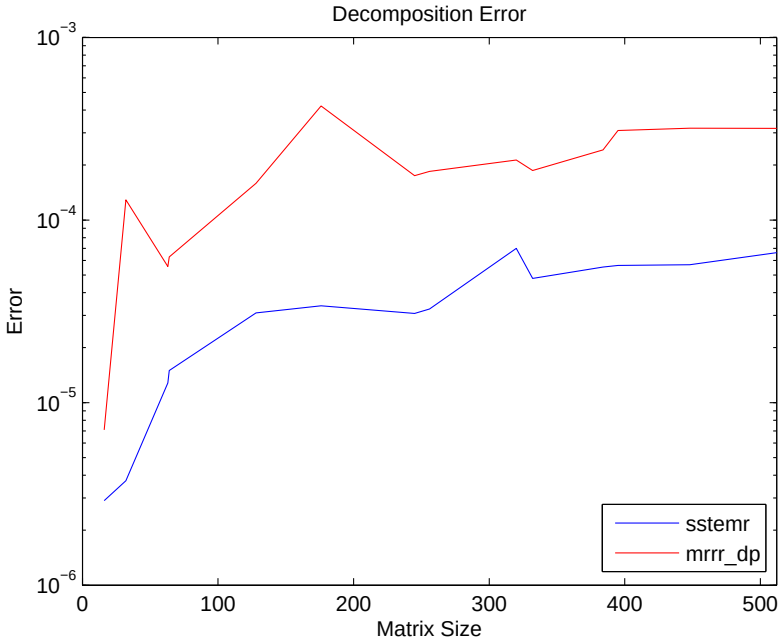

Figure 29: ℓ_∞ norm of the eigen-decomposition for the Wilkinson matrix and $n \in [32, 512]$ with $t_c = 0.01$.

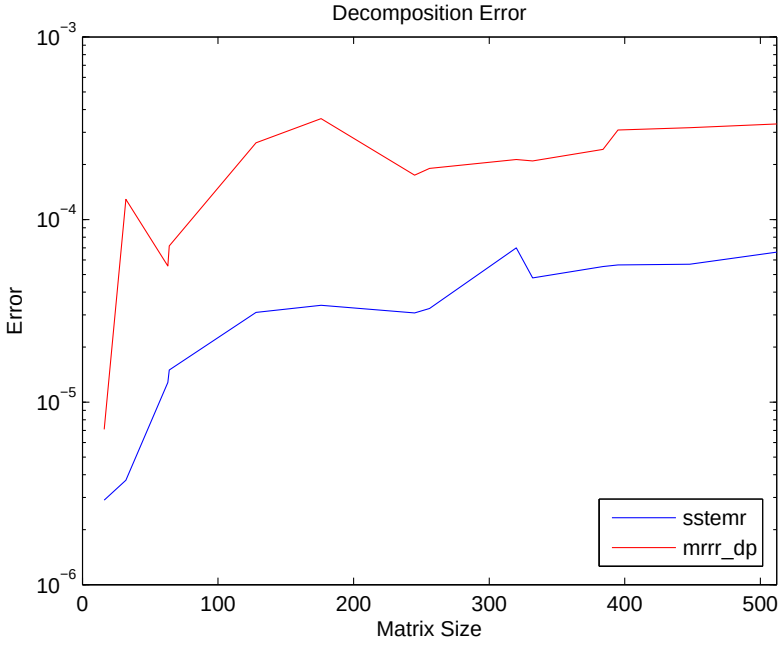

Figure 30: ℓ_∞ norm of the eigen-decomposition for the Wilkinson matrix and $n \in [32, 512]$ with $t_c = 0.000001$.

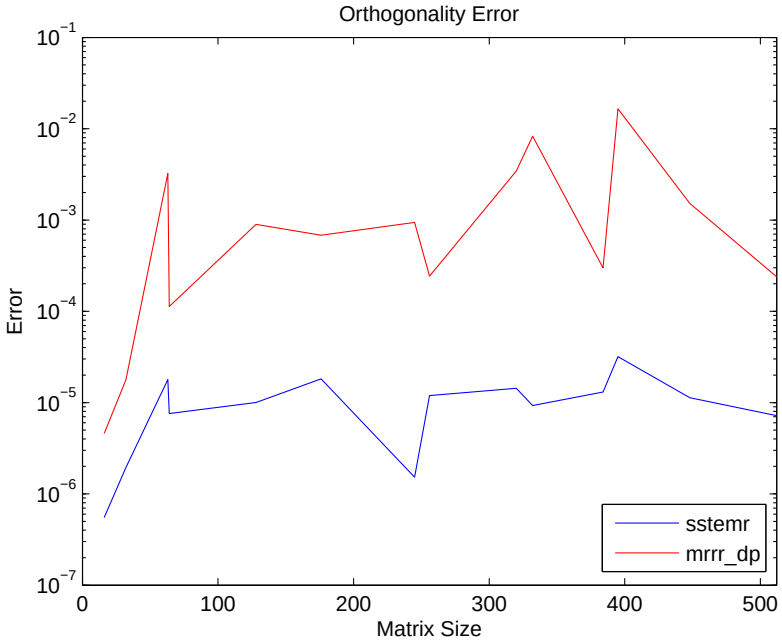

Figure 31: ℓ_∞ norm in the orthogonality of the eigenvectors for the Wilkinson matrix and $n \in [32, 512]$ with $t_c = 0.01$.

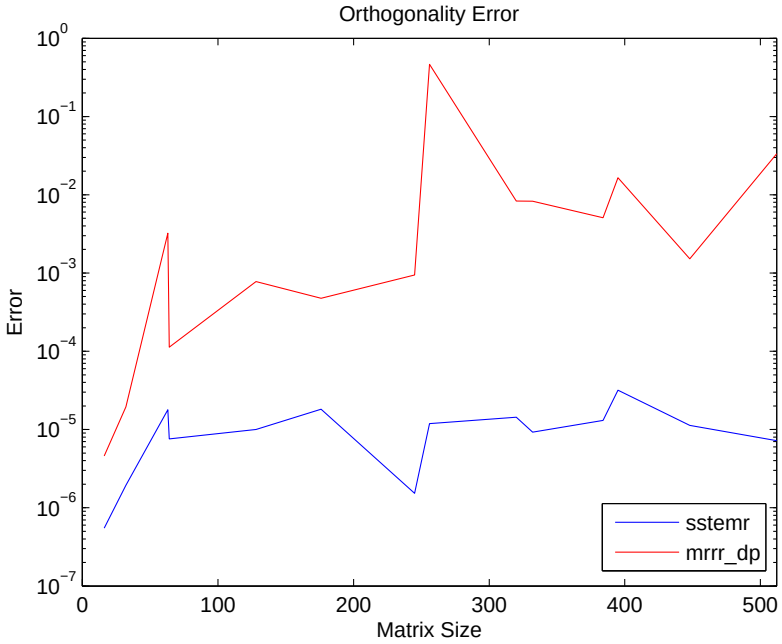

Figure 32: ℓ_∞ norm in the orthogonality of the eigenvectors for the Wilkinson matrix and $n \in [32, 512]$ with $t_c = 0.000001$.

B Linear Algebra Primer

B.1 Notation

In this report we employ Householder's notation. Scalars are denoted by lowercase roman or greek letters such as a, b, and c and α, β, and γ. Lowercase bold letters such as $\mathbf{a} = \{a_i\}_{i=1}^n$, $\mathbf{b} = \{b_i\}_{i=1}^n$, and $\mathbf{c} = \{c_i\}_{i=1}^n$ are used to represent vectors. If not mentioned otherwise, all vectors are assumed to be column vectors. Matrices are denoted by uppercase bold letters such as $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{T}$. Matrix elements are scalars indexed with two indices i and j, where i denotes the row index and j the column index. The matrix $\mathbf{B} \in \mathfrak{F}^{n\times m}$ defined over the field $\mathfrak{F}$ can thus be written as

$$\mathbf{B} = \begin{pmatrix} b_{11} & \dots & \dots & b_{1m} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ b_{n1} & \dots & \dots & b_{nm} \end{pmatrix}.$$

In this report we will be only concerned with real-valued matrices, that is $\mathfrak{F} = \mathbb{R}$. A matrix $\mathbf{A}$ is symmetric if $a_{ij} = a_{ji}, \forall(i,j)$. We will use symmetric letters such as $\mathbf{A}$ and $\mathbf{T}$ to denote symmetric matrices.

The *main diagonal* of a matrix $\mathbf{B} \in \mathbb{R}^{n\times n}$ is formed by the elements b_{ii}. In most cases we will represent the diagonal by a vector $\mathbf{b} = \{b_i\} \equiv \{b_{ii}\}$ of length n. The elements b_{ij} with $i + k = j$, $k \in \{1, \dots, n-1\}$, form the k-th *upper diagonal*, and the elements b_{ij} with $i = j + k$ the k-th *lower diagonal* of $\mathbf{B}$. Analogous to the main diagonal, we can represent k-th upper and lower diagonals by vectors of length $n - k$. $\mathbf{B}$ can therefore be written as

$$\mathbf{A} = \begin{pmatrix} a_{ii} & \dots & c_{ij} & & & & & \\ b_{ij} & a_{ii} & & c_{ij} & & & & \\ & b_{ij} & \ddots & & \ddots & & & \\ & & & \ddots & \ddots & & c_{ij} & \\ & & & & \ddots & \ddots & & c_{ij} \\ & & & & & b_{ij} & a_{ii} & \vdots \\ & & & & & & b_{ij} & a_{ii} \end{pmatrix},$$

where the elements a_{ii} form the main diagonal, the elements b_{ij} the first lower diagonal, and the element c_{ij} the k-th upper diagonal of $\mathbf{B}$. A matrix which has non-zero elements only on the main diagonal is called a *diagonal matrix*. An important diagonal matrix is the identity matrix $\mathbf{I}$ with $a_{ij} = \delta_{ij}$, where δ_{ij} is the Kronecker delta.

The *transpose* of a matrix $\mathbf{B}$ will be denoted with $\mathbf{B}^T$. We will sometimes also transpose vectors. In this case we treat a vector of length n as a matrix of size $n \times 1$.

A *symmetric tridiagonal matrix*, which is of particular importance for this report, has non-zero elements only on the main diagonal and the upper *and* lower first diagonals. We will usually use $\mathbf{T}$ to denote such a matrix.

B.2 Eigenanalysis

Definition 2 (Eigenanalysis)**.** *Let* $\mathbf{B} \in \mathbb{R}^{n\times n}$. *A scalar* λ *is an* eigenvalue *of* $\mathbf{B}$ *if*

$$\mathbf{B}\mathbf{u} = \lambda\mathbf{u}. \tag{6}$$

where $\mathbf{u} \neq \mathbf{0}$ *is a* right eigenvector. *A vector* $\mathbf{v} \neq \mathbf{0}$ *satisfying*

$$\mathbf{v}^T\mathbf{B} = \lambda\mathbf{v}^T. \tag{7}$$

is a left eigenvector *of* $\mathbf{B}$. *All left and right eigenvectors are orthogonal*

$$\langle \mathbf{u}_i, \mathbf{u}_j \rangle = \delta_{i,j} \quad \textit{and} \quad \langle \mathbf{v}_i, \mathbf{v}_j \rangle = \delta_{i,j}, \tag{8}$$

where $\langle \cdot, \cdot \rangle$ *denotes the inner product.*

Unless stated otherwise, in the following "eigenvector" refers to a right eigenvector. The set of all eigenvalues of a matrix is denoted as spectrum.

Definition 3 (Spectrum)**.** *The* spectrum $\lambda(\mathbf{M})$ *of a matrix* $\mathbf{M} \in \mathbb{R}^{n \times n}$ *is the set of all of its eigenvalues*

$$\lambda(\mathbf{M}) = \{\lambda_i\}.$$

If $\mathbf{M}$ *is not degenerate then the cardinality of* $\lambda(\mathbf{M})$ *is* n*, that is* $|\lambda(\mathbf{M})| = n$.

Corollary 1. *The spectrum of a non-degenerate diagonal matrix* $\mathbf{D} \in \mathbb{R}^{n \times n}$ *is the set of diagonal elements*

$$\lambda(\mathbf{D}) = \{d_i\}_{i=1}^{n}.$$

Alternatively to Eq. 6 and Eq. 7, eigenvalues can also be understood as the roots of the characteristic polynomial of a matrix.

Remark 1 (Characteristic Root)**.** *Let* $\mathbf{B} \in \mathbf{R}^{n \times n}$ *and* $\lambda(\mathbf{M}) = \{\lambda_i\}_{i=1}^{n}$ *be its spectrum. The roots of the characteristic polynomial*

$$\det(\mathbf{M} - \lambda_i \mathbf{I}) = 0 \tag{9}$$

of $\mathbf{M}$ *are the eigenvalues* λ_i*. Eigenvalues can thus in general be real or complex. For a symmetric matrix the eigenvalues are guaranteed to be real [32, p. 393].*

The Gerschgorin interval $\mathrm{G}_{\mathbf{A}}$ provides lower and upper bounds for the spectrum $\lambda(\mathbf{A})$ of a matrix.

Theorem 1 (Gerschgorin Circle Theorem)**.** *Let* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *be a symmetric matrix and* $\mathbf{Q} \in \mathbb{R}^{n \times n}$ *be orthogonal. If* $\mathbf{Q}^T \mathbf{A} \mathbf{Q} = \mathbf{D} + \mathbf{F}$*, where* $\mathbf{D}$ *is diagonal and* $\mathbf{F}$ *has zero diagonal entries, then*

$$\lambda(\mathbf{A}) \subseteq \bigcup_{i=1}^{n} [d_i - r_i, d_i + r_i]$$

with $r_i = \sum_{j=1}^{n} |f_{ij}|$ *for* $i = 1, \ldots, n$*, where* d_i *are the non-zero entries of* $\mathbf{D}$ *and* f_{ij} *the off-diagonal elements of* $\mathbf{F}$.

Proof. See [32, pp. 395]. □

Note that one can always choose $\mathbf{Q}$ to be the (trivially orthogonal) identity matrix to satisfy $\mathbf{Q}^T \mathbf{A} \mathbf{Q} = \mathbf{D} + \mathbf{F}$ in Theorem 1. In practice one wants to employ a matrix $\mathbf{Q}$ such that $\mathbf{Q}^T \mathbf{A} \mathbf{Q}$ is diagonally dominant. This improves the bounds provided by the Gerschgorin interval $\mathrm{G}_{\mathbf{A}} \equiv \bigcup_{i=1}^{n} [d_i - r_i, d_i + r_i]$ which can otherwise be rather pessimistic.

Corollary 2 (Gerschgorin Circle Theorem for Symmetric Tridiagonal Matrices)**.** *Let* $\mathbf{T} \in \mathbb{R}^{n \times n}$ *be symmetric and tridiagonal, and let* $\mathbf{a}$ *and* $\mathbf{b}$ *the vectors containing the diagonal and off-diagonal elements of* $\mathbf{T}$*, respectively. The spectrum of* $\mathbf{T}$ *is then bound by*

$$\lambda(\mathbf{T}) \subseteq \bigcup_{i=1}^{n} [a_i - r_i, a_i + r_i]$$

with $r_i = b_i + b_{i-1}$, for $i = 2, \ldots, (n-1)$, $r_1 = b_1$, and $r_n = b_{n-1}$.

It is often convenient to arrange the eigenvalues of $\mathbf{B} \in \mathbb{R}^{n \times n}$ in a diagonal matrix $\mathbf{D}$, and to define $\mathbf{U}$ to be the matrix whose columns are the eigenvectors. Eq. 6 can then be written as

$$\mathbf{BU} = \mathbf{UD}, \tag{10}$$

and, analogously, we can restate Eq. 7 as

$$\mathbf{V}^T\mathbf{B} = \mathbf{DV}^T. \tag{11}$$

It follows from the orthogonality of the left and right eigenvectors that $\mathbf{UU}^T = \mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\mathbf{VV}^T = \mathbf{V}^T\mathbf{V} = \mathbf{I}$.

Theorem 2 (Eigenvalue Shift)**.** *Let $\mathbf{M} \in \mathbb{R}^{n \times n}$ be a matrix with eigenvalues λ_i, and let $\mu \in \mathbb{R}$ be a* shift index. *The eigenvalues $\bar{\lambda}_i$ of the shifted matrix $\bar{\mathbf{M}}_\mu = \mathbf{M} - \mu\mathbf{I}$ are $\bar{\lambda}_i = \lambda_i - \mu$, and the eigenvectors of $\bar{\mathbf{M}}$ are those of $\mathbf{M}$.*

Proof. Consider the characteristic polynomial of $\mathbf{M}$

$$\det\left(\left(\bar{\mathbf{M}}_\mu + \mu\mathbf{I}\right) - \lambda_i\mathbf{I}\right) = 0.$$

Substituting $\mathbf{M}$ by $\mathbf{M} = \bar{\mathbf{M}}_\mu + \mu\mathbf{I}$ and rearranging the terms shows the desired result

$$\begin{aligned} \det\left(\bar{\mathbf{M}}_\mu - (\lambda_i\mathbf{I} - \mu\mathbf{I})\right) &= 0, \\ \det\left(\bar{\mathbf{M}}_\mu - (\lambda_i - \mu)\,\mathbf{I}\right) &= 0. \end{aligned}$$

For the second claim, simplifying

$$(\mathbf{M} - \mu\mathbf{I})\,\mathbf{U} = (\mathbf{D} - \mu\mathbf{I})\,\mathbf{U}.$$

which is Eq. 10 for $\bar{\mathbf{M}}_\mu$ with $\bar{\mathbf{M}}_\mu = \mathbf{M} - \mu\mathbf{I}$ and $\bar{\mathbf{D}} = \mathbf{D} - \mu\mathbf{I}$, shows that the eigenvectors of $\mathbf{M}$ and $\bar{\mathbf{M}}$ are identical. □

In practice, it is often numerically more stable and more efficient to determine an eigenvalue of a shifted matrix, and then to employ Theorem 2 to relate the computed eigenvalues to those of the original matrix.

Definition 4 (Sturm Count)**.** *Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be symmetric and let $\mu \in \mathbb{R}$ be a shift index. The* Sturm count $s_\mu(\mathbf{A})$ *is the number of eigenvalues of $\mathbf{A}$ smaller then μ.*

Sturm counts are important for the *bisection algorithm* which determines the eigenvalues of a real symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ [31]. Starting with the Gerschgorin interval $\mathrm{G}_\mathbf{A}$ as root node, bisection generates an unbalance binary tree by splitting intervals on level $l-1$ and retaining for level l only those containing eigenvalues. The Sturm counts at the interval bounds are employed to determine if an interval is empty. The algorithm converges when the size of all intervals falls below a thresholds t, where t governs the accuracy of the obtained eigenvalues. See for example the book by Golub and van Loan [32, pp. 437], and the papers by Demmel et al. [16] and Marques et al. [46] for a more detailed introduction.

Most eigenanalysis algorithms do not operate on general matrices but first reduce the input matrix to a canonical form. Orthogonal transformations are employed for this reduction.

Theorem 3. *Let $\mathbf{Q}_i \in \mathbb{R}^{n \times n}$ with $i = 1, \ldots, m$ be a sequence of orthogonal matrices, and let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be symmetric. Then, every matrix $\mathbf{A}_i$ with*

$$\mathbf{A}_0 = \mathbf{A} \quad \textit{and} \quad \mathbf{A}_{i+1} = \mathbf{Q}_i^T\mathbf{A}\mathbf{Q}_i$$

has the same eigenvalues as A, and computing the orthogonal transformation $\mathbf{Q}_i^T\mathbf{A}\mathbf{Q}_i$ *is numerically stable.*

Proof. The proof follows directly from Theorem 7.1.3 in [32]. □

In practice a common canonical representation is a symmetric tridiagonal matrix, and one then seeks the solution to the *symmetric tridiagonal eigenvalue problem*. Householder transformations are employed to reduce an arbitrary symmetric matrix to tridiagonal form [33, pp. 206].

Theorem 4 (LDL^T Factorization)**.** *Let* $\mathbf{T} \in \mathbb{R}^{n \times n}$ *be symmetric, tridiagonal, and positive definite, and let* $\mathbf{a}$ *and* $\mathbf{b}$ *be the vectors containing the diagonal and off-diagonal elements of* $\mathbf{T}$*, respectively. If* $a_i b_i \neq 0$ *for* $i = 1, \ldots, n$ *then there exists a* LDL^T factorization *such that*

$$\mathbf{T} = \mathbf{L}\mathbf{D}\mathbf{L}^T,$$

where $\mathbf{D}$ *is diagonal and* $\mathbf{L}$ *is lower bidiagonal with all diagonal elements being* 1.

Proof. Consider the Cholesky factorization $\mathbf{T} = \bar{\mathbf{L}}\bar{\mathbf{L}}^T$ of $\mathbf{T}$. The result then follows immediately from $\bar{\mathbf{L}} = \mathbf{L}\mathbf{D}^{1/2}$. □

The Cholesky factorization of $\mathbf{T}$, and therefore also its LDL^T factorization, exists only if $\mathbf{M}$ is positive definite. In practice we can shift $\mathbf{T}$ to ensure its positive definiteness.