A HIGH PERFORMANCE QDWH-SVD SOLVER USING HARDWARE ACCELERATORS

DALAL SUKKARI†, HATEM LTAIEF†, AND DAVID KEYES†

Abstract. This paper describes a new high performance implementation of the QR-based Dynamically Weighted Halley Singular Value Decomposition (QDWH-SVD) solver on multicore architecture enhanced with multiple GPUs. The standard QDWH-SVD algorithm was introduced by Nakatsukasa and Higham (SIAM SISC, 2013) and combines three successive computational stages: (1) the polar decomposition calculation of the original matrix using the QDWH algorithm, (2) the symmetric eigendecomposition of the resulting polar factor to obtain the singular values and the right singular vectors and (3) the matrix-matrix multiplication to get the associated left singular vectors. A comprehensive test suite highlights the numerical robustness of the QDWH-SVD solver. Although it performs up to two times more flops when computing all singular vectors compared to the standard SVD solver algorithm, our new high performance implementation on single GPU results in up to 3.8x improvements for asymptotic matrix sizes, compared to the equivalent routines from existing state-of-the-art open-source and commercial libraries. However, when only singular values are needed, QDWH-SVD is penalized by performing up to 14 times more flops. The singular value only implementation of QDWH-SVD on single GPU can still run up to 18% faster than the best existing equivalent routines. Integrating mixed precision techniques in the solver can additionally provide up to 40% improvement at the price of losing few digits of accuracy, compared to the full double precision floating point arithmetic. We further leverage the single GPU QDWH-SVD implementation by introducing the first multi-GPU SVD solver to study the scalability of the QDWH-SVD framework.

Key words. Singular Value Decomposition, Polar Decomposition, Symmetric Eigensolver, Mixed Precision Algorithms, GPU-based scientific computing.

AMS subject classifications. 15A18, 65Y05, 65Y20, 68Q25, 68W10

1. Introduction.

Computing the singular value decomposition (SVD) is a critical operation for solving least square problems, determining the pseudoinverse of a matrix, or calculating low-rank matrix approximations, with direct application to signal processing, pattern recognition and statistics [8, 9, 25]. This paper proposes a new high performance implementation SVD solver featuring the QR-based Dynamically Weighted Halley (QDWH) on multicore architecture equipped with GPU accelerators. First introduced by Nakatsukasa and Higham [24], this new spectral divide and conquer algorithm for SVD is composed of three successive computational stages: (1) computing the polar decomposition of the original matrix using the QDWH algorithm, (2) calculating the symmetric eigendecomposition of the resulting polar factor to obtain the eigenvalues (which correspond to the singular values) and their associated eigenvectors (which correspond to the right singular vectors), and (3) applying the matrix-matrix multiplication to get the remaining left singular vectors. The QDWH-SVD framework presents interesting concepts compared to previous approaches [24]. It achieves backward stability thanks to the polar decomposition, reveals communication-reducing properties, and is implemented with conventional linear algebra building blocks for which quality vendor tunings are often available. However, QDWH-SVD exhibits overhead factors up to 14-fold and up to twofold in terms of number of extra floating-point operations (flops) compared to the standard SVD algorithm, as implemented in LAPACK [2], when only singular values are needed

†Extreme Computing Research Center, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia. Contact: Dalal.Sukkari@kaust.edu.sa, Hatem.Ltaief@kaust.edu.sa, David.Keyes@kaust.edu.sa

and additionally singular vectors, respectively. Although the total number of flops is very sensitive to the matrix conditioning and the convergence rate can greatly improve when the matrix is well-conditioned, the overhead due to the extra flops can still make the algorithm unattractive, especially if only singular values are to be calculated.

On the other hand, when it comes to high performance parallel implementations of numerical algorithms, flops are often no longer a sound proxy for execution time or energy expenditure. One must examine what are the limiting factors with respect to the hardware for a given execution, i.e., cache memory sizes, bus bandwidth, available cores (which may otherwise be idle), number of floating point units per core, etc. This step is paramount because it can help to assess the best implementation on specific hardware for given objectives, e.g., time to solution, energy to solution, data distribution for the next step of an overall procedure, etc. Most of the linear algebra operations of QDWH-SVD are expressed through Level 3 BLAS [5]. This level of BLAS deals with highly compute-intensive and parallel operations (mostly based on matrix-matrix operations), which stress the floating-point units of the underlying hardware and presents a high data reuse rate for the processor caches. As highlighted in the International Exascale Software Project [7], to exploit opportunities for energy-efficient performance, algorithms must be designed to boost concurrency and reduce data motion. Over the past decade, accelerators (e.g., GPUs) epitomize such hardware evolution. Indeed, today, because of the PCIe bottleneck, GPUs can process data beyond an order of magnitude faster than the speed with which they are fed. In other words, numerical algorithms performing extra flops should not usually be looked at as "show-stoppers" as long as data motion is confined.

This paper proposes a new, efficient implementation of the QDWH-SVD algorithm on multicore and GPUs hardware architecture operating at high flop/s rates. Using the high performance Matrix Algebra on GPU and Multicore Architectures (MAGMA) library [1, 22], all three computational stages of QDWH-SVD are accelerated using GPUs, while limiting and hiding data transfers between the host and the device and therefore, favoring *in situ* GPU processing. The resulting high performance GPU-based QDWH-SVD implementation outperforms by up to four-fold, and up to three-fold for asymptotic random matrix sizes, the equivalent routines (DGESVD) from existing state-of-the-art commercial (Intel MKL [16]) and open-source (MAGMA) libraries, respectively, when computing all singular values and vectors. If only singular values are required, the algorithm we implement is excessively penalized by performing up to 14 times more flops, depending on the matrix conditioning, compared to the standard SVD solver. The singular value-only QDWH-SVD implementation is still able to run up to 18% faster than the best existing equivalent routine. Moreover, if the end user's application is tolerant to loss of few digits, the singular value only implementation using mixed precision techniques can execute up to 50% faster than the standard full double precision floating point arithmetic SVD solver. A comprehensive testing framework checks on the numerical accuracy of singular values, the orthogonality of singular left/right vectors, and the accuracy of the computed singular values and the associated left/right vectors on various matrix conditioning types, which highlights the robustness of the overall method. Finally, a multi-GPU implementation is introduced as a first attempt to study the scalability of the QDWH-SVD framework.

The rest of the paper is organized as follows. Section 2 highlights our contributions. Section 3 presents related work. Section 4 briefly recalls the standard SVD algorithm. Section 5 describes the QDWH-SVD framework and presents its different

computational stages. Section 6 introduces the algorithmic complexity. Section 7 illustrates the QDWH-SVD numerical accuracy with various matrix conditioning. Section 8 shows the QDWH-SVD performance results and compares against the state-of-the-art commercial and open-source high performance SVD solver implementations, and we conclude in Section 9.

2. Contributions. We here delineate the research contributions of this paper for developing a new high performance SVD solver on x86 multicore and GPUs hardware architecture:

- We present a high performance implementation of the polar decomposition (QDWH), which is one of the main components for the symmetric eigensolver and the SVD solver.
- A high performance implementation of the symmetric eigensolver based on the spectral divide and conquer approach (QDWH-EIG) is proposed and performance compared against the two-stage symmetric eigensolver [12, 13, 21], previously implemented by some of the authors of this paper.
- A high performance implementation of the overall QDWH-SVD solver framework is described, which integrates the three computational stages: polar decomposition, symmetric eigensolver and matrix-matrix multiplication.
- A multiple GPU implementation of the resulting QDWH-SVD solver has been developed to study its scalability, which corresponds to the first SVD solver on multiple hardware accelerators (to our knowledge).
- A mixed precision version of the algorithm is developed for the case where only approximate singular values are required, enabling a trade between accuracy and time complexity.
- To show the numerical robustness of the newly implemented QDWH-SVD solver, the paper emphasizes on a comprehensive test suite for testing the numerical accuracy of the singular values, the orthogonality of the left/right singular vectors, and the accuracy of the computed singular values and the associated left/right vectors.

The combination of highly optimized dense linear algebra operations from MAGMA [1, 22] to form the new high performance QDWH-SVD solver represents the crux of the research work presented in this paper.

3. Related Work. The standard SVD algorithm first reduces a general dense matrix to bidiagonal form, then extracts the singular values from the condensed form using the QR [6, 17] or divide-and-conquer [10] algorithms and, finally, accumulates the subsequent orthogonal transformations if singular vectors are also required. While the back transformation phase to calculate the corresponding singular vectors is rich in Level 3 BLAS operations, the reduction to bidiagonal form is often seen as a major bottleneck for parallel performance due to the inefficient Level 2 BLAS kernels predominance.

The authors of the Successive Band Reductions (SBR) software package [4, 18] introduced an intermediary computational stage before getting to the final bidiagonal form. Originally applied to the symmetric eigensolver [12, 21], the main ideas have been extended to SVD solver. The matrix is first reduced to band bidiagonal form, where most of the original Level 2 BLAS operations are now cast into Level 3 BLAS, thus increasing the level of concurrency, as highlighted in [19]. The second stage annihilates the extra off-diagonal elements using an efficient bulge chasing technique, until the final condensed structure is obtained. Although operations in the latter stage are memory-bound, data reuse at the high level of caches is possible, thanks

to some locality scheduling heuristics [11]. The standard column-major data format of the matrix has to be translated into a tile data layout. Indeed, the matrix is split into tiles, where elements of a given tile are now contiguous in memory. The algorithm of the SVD solver operates now on tiles and exposes fine-grained tasks to be scheduled by a dynamic runtime system [1]. The overall tile SVD solver also generates substantial extra flops [3], especially during the back transformation, in case of singular vectors are desired. All the aforementioned high performance SVD solver implementations based on a two-stage matrix reduction run on x86 architecture only (accelerators are currently not supported), due to the challenges of porting non-conventional computational kernels on such complex platforms.

Last but not least, a previous framework for computing the SVD via the polar decomposition and the eigendecomposition has already been presented in [15]. This algorithm requires three building blocks: matrix multiplication, matrix inversion, and solution of the Hermitian eigenproblem. The QDWH-SVD algorithm follows the core idea of this framework by implementing a new SVD solver replacing the expensive matrix inversion with a QDWH-based algorithm [24], which increases parallel concurrency and is rich in Level 3 BLAS.

The high performance QDWH-SVD implementation presented in this paper is inverse free and communication friendly, and therefore suitable for multicore and hybrid high performance computing systems. It operates on column-major format and uses building blocks (QR/Cholesky factorizations and matrix-matrix multiplication) well-known by the scientific community and often well-optimized by the vendor community on shared-memory as well as distributed-memory systems, which makes the portability of such codes possible across various architectures.

4. The Standard SVD Algorithm.

This section recalls the standard algorithm of the SVD solver for dense matrices, as implemented in LAPACK [2] and MAGMA [22] through the API function DGESVD.

4.1. Block Algorithms.

Introduced in the 1990's [2] with the emergence of hierarchical cache-based shared-memory systems, LAPACK block algorithms allow to recast vector operations to matrix operations so that efficient data reuse can be achieved on the memory subsystem. Matrix computations are split into two successive phases: (1) panel factorization, in which Level 2 BLAS operations are accumulated within a block of columns (called panel) for later usage and (2) update of the trailing submatrix, in which the accumulated transformations from the panel are applied by means of Level 3 BLAS operations on the unreduced part of the matrix. These two phases represent the core algorithmic methodology to solve linear systems of equations as well as eigenvalue problems and singular value decompositions.

4.2. Three Computational Stages.

The singular value decomposition (SVD) of $A \in \mathbb{R}^{m \times n}$ is $A = U \Sigma V^\top$, where Σ is the matrix containing all the singular values, and U and V are the orthogonal matrices containing the left and right singular vectors, respectively.

The standard SVD algorithm (a) reduces the dense matrix to condensed bidiagonal form through a series of orthogonal transformations, (b) applies an iterative solver (QR or divide-and-conquer algorithms) to calculate the singular values and its associated left/right singular vectors of the bidiagonal matrix, and (c) multiplies the freshly computed singular vectors with the accumulated subsequent orthogonal transformations during the reduction phase (also called back transformation), as shown in Figure 1.

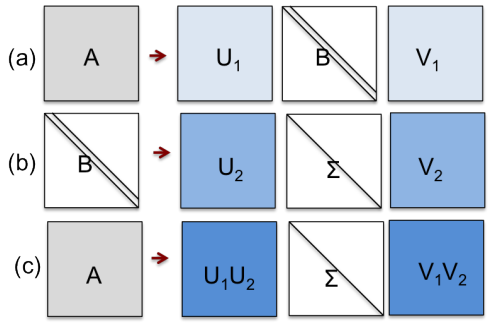


Fig. 1: Computational stages of the standard SVD algorithm: (a) bidiagonal reduction, (b) bidiagonal SVD solver and (c) back transformation.

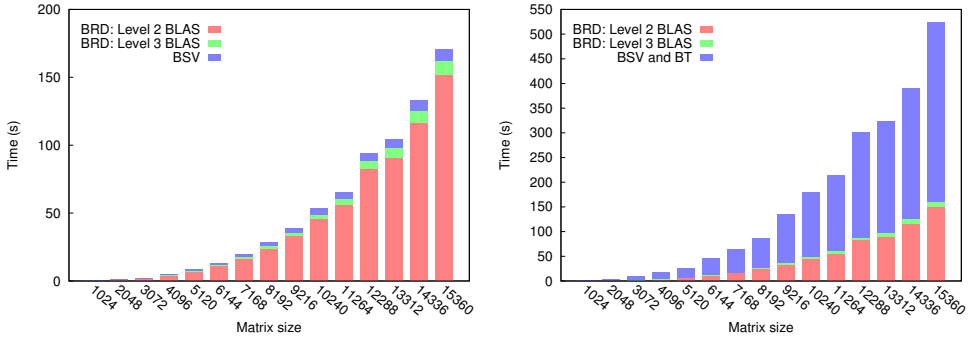
4.3. Algorithmic Drawbacks. To understand the algorithmic drawbacks of the standard SVD solver, it is important to distinguish two classes of transformations. One-sided transformations are typically needed for solving linear systems of equations and least-square problems: the matrix is reduced and factorized during the panel factorization phase and the latter does not require accessing matrix data outside of the current panel. Resulting transformations are then applied to the trailing submatrix on the left side. On the other hand, two-sided transformations are necessary for eigenvalue and SVD solvers. Their panel factorization is much more expensive than the one-sided transformations because it requires reading the entire trailing submatrix at each column update of the current panel. Once the panel is reduced, the accumulated transformations are applied from both sides, left and right, of the unreduced submatrix, which make, for instance, look ahead techniques for increasing concurrency difficult to apply, as opposed to one-sided transformations.

To illustrate the bottleneck of the panel factorization, Figure 2 describes the profiling of the SVD solver DGESVD, using MAGMA implementation (single GPU only). Although only 50% of the total number of operations of the bidiagonal reduction are expressed through Level 2 BLAS operations, they still count toward 90% of the elapsed time, as shown in Figure 2(a). Furthermore, when the overall SVD solver is executed, the trend represented in Figure 2(b) highlights that even though level 2 BLAS operations count only for 6% of the total number of flops, these still translate into 30% of the entire execution time.

5. The QDWH-SVD Framework. This section recalls the general QDWH-SVD algorithm and describes the three computational stages: the polar decomposition, the symmetric eigensolver and the matrix-matrix multiplication.

5.1. The Polar Decomposition. The polar decomposition is an important numerical algorithms for various applications, including aerospace computations, chemistry, multidimensional scaling in statistics, computation of block reflectors in numerical linear algebra, factor analysis and signal processing. In this paper, the polar decomposition is used as a first computational phase toward computing the SVD of a general dense matrix.

5.1.1. Background. The polar decomposition of the matrix $A \in \mathbb{R}^{m \times n}$ $(m \geq n)$ is written $A = U_p H$, where U_p is an orthogonal matrix and $H = \sqrt{A^* A}$ is a symmetric positive semidefinite matrix. To find the polar decomposition, the original DWH



(a) Computing the singular values.

(b) Computing additionally the singular vectors.

Fig. 2: Profiling of MAGMA-DGESVD: bidiagonal reduction (BRD), bidiagonal SVD solver (BSV) and back transformation (BT).

iteration can be derived as follows:

$$X_0 = A/\alpha,\ X_{k+1} = X_k(a_k I + b_k X_k^* X_k)(I + c_k X_k^* X_k)^{-1}, \tag{5.1}$$

where $\alpha = \|A\|_2$. The scalars (a_k, b_k, c_k) are critical parameters since they drive the convergence speed at each iterate. The formulas of (a_k, b_k, c_k, l_k) are as follows:

$$\begin{aligned} &a_k = h(l_k),\ b_k = (a_k - 1)^2/4,\ c_k = a_k + b_k - 1, \\ &l_0 \lesssim \sigma_{min}(X_0),\ l_k = \frac{l_{k-1}(a_{k-1} + b_{k-1} l_{k-1}^2)}{1 + c_{k-1} l_{k-1}^2},\ k = 1, 2, \ldots, \end{aligned} \tag{5.2}$$

where $\sigma_{min}(X_0)$ is the minimum singular value of X_0, $l_k = 1/\kappa_2(X_k)$ is the inverse of the condition number of the matrix iterate X_k and $h(l) = \sqrt{1+d} + \frac{1}{2}\sqrt{8 - 4d + \frac{8(2-l^2)}{l^2\sqrt{1+d}}}$, $d = \sqrt[3]{\frac{4(1-l^2)}{l^4}}$. However, this original QDWH iteration (Equation 5.1) requires an expensive matrix inversion at each iteration.

Fortunately, there exists an attractive practical QDWH formulation, which is based on matrix-inversion free [23] and with proper dynamically calculated (a_k, b_k, c_k), the algorithm converges after six iterations at most.

5.1.2. The *Practical* QR-based Dynamically Weighted Halley Method. This section highlights the practical QDWH iterative algorithm and makes the paper algorithmically self-contained. Convergence proofs can be found in [23].

Convergence. The convergence of the iterate X_{k+1} to the polar factor is measured by the closeness of its ith largest singular values $\sigma_i(X_{k+1})$ to 1. They can be found as follows. Let the SVD of $X_k = U_k \Sigma_k V_k^\top$. All singular values are now bounded such that $[\sigma_{min}(X_k), \sigma_{max}(X_k)] \subseteq [l_k, 1] \subset (0, 1]$. Hence,

$$\begin{aligned} X_{k+1} &= U_k \Sigma_k V_k^\top (a_k I + b_k V_k \Sigma^2 V_k^\top)(I + c_k V_k \Sigma_k^2 V_k^\top)^{-1} \\ &= U_k \Sigma_k (a_k I + b_k \Sigma^2)(I + c_k \Sigma_k^2)^{-1} V_k^\top \\ &= U_k \Sigma_{k+1} V_k^\top. \end{aligned}$$

Therefore, the singular values $\sigma_i(X_{k+1})$ are given by:

$$\sigma_i(X_{k+1}) = x\frac{a_k + b_k x^2}{1 + c_k x^2}, \tag{5.3}$$

where $x = \sigma_i(X_k)$. Hence, the solution of the following optimization problem $\max_{a_k,b_k,c_k}\{\min_{l_k \le x \le 1} \sigma_i(X_{k+1})\}$ gives optimal values of (a_k, b_k, c_k) (Equation 5.2) and bounds $\sigma_i(X_{k+1})$ as follows: $0 < \sigma_i(X_{k+1}) \le 1$, $l_k \le x \le 1$. The maximum number of DWH iterations for convergence can be calculated by determining the first k such that $|1 - l_k| < \epsilon$ (machine epsilon). Assuming double precision arithmetic with $\epsilon = 10^{-16}$, unrolling Equations 5.1, 5.2, and 5.3 with $l_0 = 1/\kappa_2(X_0)$ and a fairly large condition number $\kappa_2(X_0) = 10^{16}$, the number of DWH iterations is six. Therefore, DWH converges within at most six iterations for any matrix with condition number $\kappa_2 \le 10^{16}$.

Matrix Inversion Free QDWH. Equation 5.1 can be reformulated using the mathematically equivalent QR-based implementation [24]:

$$\begin{gathered} X_0 = A/\alpha, \\ \begin{bmatrix} \sqrt{c_k}X_k \\ I \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R,\ X_{k+1} = \frac{b_k}{c_k}X_k + \frac{1}{\sqrt{c_k}}\left(a_k - \frac{b_k}{c_k}\right) Q_1 Q_2^*,\ k \ge 0. \end{gathered} \tag{5.4}$$

If the condition number of the matrix X_k improves during execution and X_k becomes well-conditioned, it is possible to replace Equation 5.4 using a Cholesky-based implementation as follows:

$$\begin{gathered} X_{k+1} = \frac{b_k}{c_k}X_k + \left(a_k - \frac{b_k}{c_k}\right)(X_k W_k^{-1})W_k^{-*}, \\ W_k = chol(Z_k),\ Z_k = I + c_k X_k^* X_k. \end{gathered} \tag{5.5}$$

This algorithmic switch at runtime allows to further speed up the overall computation, thanks to a lower algorithmic complexity, while still maintaining numerical stability. In particular, in the subsequent experiments, our implementations switch from Equation 5.4 to Equation 5.5 if c_k is smaller than 100, as suggested in Section 5.6 of [24].

5.2. Computing the Symmetric Eigensolver. The second computational stage of QDWH-SVD consists in computing the symmetric eigendecomposition of the matrix $H = V\Sigma V^\top$ from Equation 5.1, where V corresponds to the eigenvectors of H as well as to the right singular vectors of the original dense matrix A. This Section describes two different eigensolver algorithms: a symmetric eigensolver based (again) on the QDWH procedure and a two-stage symmetric eigensolver.

5.2.1. The QDWH-Based Symmetric Eigensolver. The QDWH-based symmetric eigensolver (QDWH-EIG) [24] is a spectral divide-and-conquer algorithm to solve the symmetric eigenvalue problem. It is built on the polar decomposition (see previous Section 5.1) and operates by recursively decoupling the problem into independent subproblems through finding invariant subspaces. It is noteworthy to mention that this QDWH series of iterations operates in the inner loop of the QDWH-SVD algorithm.

Algorithm 1 outlines the overall QDWH-EIG algorithm. In order to provide a balanced division during the recursions, QDWH-EIG shifts the diagonal of the matrix

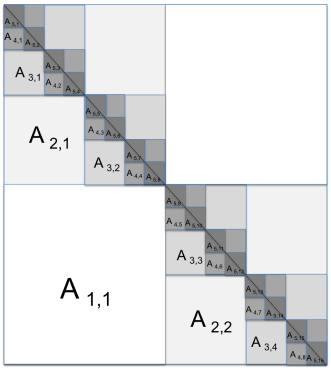


Fig. 3: The recursive QDWH-EIG algorithm. The matrix $A_{i,j}$ corresponds to the submatrix indexed j at the ith level of recursion.

Fig. 4: Reduction of a symmetric dense matrix to tridiagonal form using a two-stage approach.

by an estimate of the median of the eigenvalues. After computing the polar decomposition, the polar factor U_p permits eventually to generate two orthogonal matrices V_1 and V_2 through a subspace iteration procedure. Applying these orthogonal matrices to the original matrix permits to discriminate among eigenvalues along the median and consequently, to evenly span the eigenvalue spectrum across subsequent subproblems. The algorithm proceeds then recursively on each new subproblems until the submatrices get diagonalized, as depicted in Figure 3. Finally, all subsequent orthogonal transformations need to be accumulated in order to generate the eigenvectors corresponding to the eigenvalues.

Algorithm 1 The QDWH-Based Symmetric Eigensolver

1: Choose σ, an estimate of the median of eig(A).
2: Compute the orthogonal polar factor U_p of $A - \sigma I$ by the QDWH algorithm.
3: Use subspace iteration to compute an orthogonal $V = \begin{bmatrix} V_1 & V_2 \end{bmatrix}$ ($V_1 \in R^{n \times k}$) such that $\frac{1}{2}(U_p + I) = V_1 V_1^*$.
4: Compute $A_1 = V_1^* A V_1 \in R^{k \times k}$ and $A_2 = V_2^* A V_2 \in R^{(n-k) \times (n-k)}$.
5: Repeat steps $1 \rightarrow 4$ with $A \leftarrow A_1$ and $A \leftarrow A_2$ until A is diagonalized.

5.2.2. Two-Stage Symmetric Eigensolver. Previous works from some of the authors have shown a two-stage tridiagonal reduction phase necessary for implementing an efficient symmetric eigensolver on standard x86 architecture [12, 20, 21] and later on hardware accelerators [13], in the context of the PLASMA and MAGMA [1] numerical libraries, respectively. Similarly to the bidiagonal reduction for the SVD solver, the tridiagonal reduction phase can take up to 90% of the overall elapsed time of the symmetric eigensolver when only eigenvalues are needed and up to 50% when additionally eigenvectors are required. As sketched in Figure 4, since the matrix is symmetric, the upper (or lower) part is not referenced (colored in gray). The first stage reduces the original symmetric dense matrix to a symmetric band form. The second stage applies the bulge chasing procedure, which annihilates all the extra off-diagonal entries. The two-stage approach casts expensive memory operations

occurring during the panel factorization into faster compute intensive ones. Once the tridiagonal reduction is achieved, a divide-and-conquer eigensolver calculates the eigenvalues and its associated eigenvectors of the condensed matrix structure. These eigenvectors need to be further aggregated against all subsequent orthogonal transformations, which occurred during the two-stage tridiagonal reduction phase. This back transformation procedure (similar to the SVD solver, see previous Section 4.2) produces the final eigenvectors of the original symmetric dense matrix.

In fact, the overall two-stage symmetric eigensolver algorithm can be expressed into fine-granularity tasks to further expose parallelism, which eventually generates a directed acyclic graph (DAG), where nodes represent tasks and edges describe the data dependencies between them. An efficient dynamic runtime system named QUARK [26] is then employed to schedule the different tasks from both stages in an out-of-order fashion, as long as data dependencies are not violated to ensure numerical correctness.

5.3. Matrix-Matrix Multiplication. At this stage, the polar factor U_p and the right singular vectors V^T have already been calculated from the polar decomposition in Section 5.1 and from the symmetric eigensolver in Section 5.2, respectively. $A = U_pH$ (with $H = V\Sigma V^\top$) can then be rewritten as $A = U_p(V\Sigma V^\top) = (U_pV)\Sigma V^\top$. The last stage simply invokes a matrix-matrix multiplication kernel to calculate the remaining left singular vectors $U = U_pV$.

All in all, the iterative QDWH-SVD procedure relies primarily upon communication friendly and compute-intensive matrix operations, such as the QR/Cholesky factorization and the matrix-matrix multiplication, which are very often further optimized by the vendors (Intel MKL, AMD ACML, IBM ESSL, etc.). These types of operations also exhibit a high degree of parallelism. However, although the algorithm performs at most six iterations from the outer loop of the QDWH-SVD framework, it is still important to assess the algorithmic complexity of the overall QDWH-SVD and to compare it against the standard SVD approach (Section 4).

6. Algorithmic Complexity. In this Section, we present the algorithmic complexity of the standard SVD algorithm (DGESVD/DGESDD) and the two variants of the QDWH-SVD procedure, that is, using the QDWH-EIG framework or the two-stage approach as the symmetric eigensolver. We investigate the complexity of all aforementioned SVD algorithms in terms of number of floating-point operations (flops). We assume that the matrix A is square of size n.

6.1. The Standard SVD Solver. The standard approach to compute the SVD of a dense matrix is to first reduce it to bidiagonal form $A = U_1BV_1^\top$. This reduction can be done using Householder reflectors for a cost of $\frac{8}{3}n^3$ and is a common step for both standard DGESVD and DGESDD functions, as implemented in LAPACK. If only singular values are needed, computing them from the bidiagonal form using the QR (DGESVD) or the divide-and-conquer (DGESDD) algorithms requires $O(n^2)$ flops. If singular vectors are additionally required from $B = U_2\Sigma V_2$, the LAPACK subroutines DBDSQR and DBDSDC implement an iterative method and a recursive approach based on QR algorithm and divide-and-conquer, respectively. The subsequent left and right singular vectors are then accumulated during the back transformation phase, i.e., $U = U_1U_2$ and $V = V_1V_2$, to calculate the singular vectors of the original matrix A. The final estimated flop count to calculate the SVD is $22n^3$ for either DGESVD or DGESDD [14].

Even though both standard SVD implementations operate at the same flop counts, DGESDD is usually faster than DGESVD thanks to the recursion formulation. However, it necessitates larger workspaces to cast most of operations in terms of Level 3 BLAS, which permit to achieve further performance optimizations thanks to a higher rate of data reuse.

6.2. The QDWH-SVD Solver. The QDWH-SVD flop count includes the cost of the polar decomposition, the solution of the symmetric eigenvalue problem (which produces the singular values and the right singular vectors) and the matrix-matrix multiplication kernel (which computes the left singular vectors).

As shown in Equation 5.4, the QDWH flops using QR-based iteration includes the QR decomposition of $2n \times n$ matrix for a cost of $(3+\frac{1}{3})n^3$ flops. Then, forming $\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}$ explicitly, needs $(3+\frac{1}{3})n^3$ flops. The product $Q_1 Q_2'$ needs $2n^3$ flops. Therefore, the arithmetic cost of each QR-based iteration is $(8+\frac{2}{3})n^3$ flops. For the Cholesky-based iteration in Equation 5.5, matrix-matrix multiplication involves $2n^3$, the Cholesky factorization needs $\frac{1}{3}n^3$, and solving two linear systems requires $2n^3$. Therefore, the arithmetic cost of Cholesky-based iteration is $(4+\frac{1}{3})n^3$. Computing the positive semidefinite matrix $H = U_p^\top A$ requires $2n^3$. Hence, the overall cost of QDWH is $(8+\frac{2}{3})n^3 \times \#it_{QR} + (4+\frac{1}{3})n^3 \times \#it_{Chol} + 2n^3$, where $\#it_{QR}$ and $\#it_{Chol}$ correspond to the number of QR-based and Cholesky-based iterations, respectively. As discussed in [24], the flop count of QDWH depends on the matrix condition number κ_2, which involves during the QDWH iteration. The total flop count for QDWH ranges then from $(10+\frac{2}{3})n^3$ (for $\kappa_2(A) \approx 1$ with $\#it_{Chol} = 2$) to $(36+\frac{2}{3})n^3$ (for $\kappa_2(A) \gg 1$ with typically $\#it_{QR} = 2$ and $\#it_{Chol} = 4$).

This is followed by finding the eigenvalue decomposition of the resulting matrix H. In the original QDWH-SVD, the default variant of the eigensolver is QDWH-EIG (see Section 5.2.1). The flop count of a single execution of QDWH-EIG includes the cost of the polar decomposition, the subspace iteration and forming the submatrices A_1, A_2. It is assumed, following [24], that during the QDWH-EIG execution, the shift is always taken so that A_1 and A_2 are both approximately of dimension $\frac{n}{2}$. Since one spectral division results in two submatrices of size $\approx n/2$ and the arithmetic cost scales cubically with the matrix size, the overall arithmetic cost is approximately $\sum_{i=0}^{\inf}(2.2^{-3})^i\beta = \frac{4}{3}\beta$, where β is the number of flops needed for one run of QDWH-EIG for $n \times n$ matrix. The cost of the invariant subspace is $\frac{7}{6}n^3$ to obtain $\frac{n}{2}$ Householder reflectors. Applying them to A needs $(\frac{3}{2}+\frac{3}{4})n^3 = \frac{9}{4}n^3$. The final step of one recursion of QDWH-EIG is to update the eigenvectors $V_1 = V_1 V_{21}$ and $V_2 = V_2 V_{22}$, each requiring $\frac{1}{2}n^3$ flops. The flop count of QDWH-EIG depends also on the matrix condition number κ_2 and ranges from $(16+\frac{1}{9})n^3$ (for $\kappa_2(A) \approx 1$) to $(50+\frac{7}{9})n^3$ (for $\kappa_2(A) \gg 1$) if only the eigenvalues are needed and $(17+\frac{4}{9})n^3$ (for $\kappa_2(A) \approx 1$) to $(52+\frac{1}{9})n^3$ (for $\kappa_2(A) \gg 1$) if the eigenvectors are additionally needed.

The other variant of the symmetric eigensolver is based on a two-stage procedure (two-stage-EIG), as previously explained in Section 5.2.2. For finding the eigenvalues only, it needs $\frac{4}{3}n^3$ flops to reduce the symmetric matrix into a band matrix (with the matrix bandwidth $nb \ll n$) and a lower-order amount for the subsequent bulge-chasing step (bandwidth reduction) and for solving the resulting tridiagonal eigenvalue problem. If the eigenvectors are additionally needed, the reduction phase costs $\frac{8}{3}n^3$ due to the accumulation overhead of the intermediary orthogonal transformations, and solving the tridiagonal eigenvalue problem using divide and conquer algorithm D&C needs $\frac{4}{3}n^3$. Due to the two-stage tridiagonal reduction, recovering the original eigen-

vectors needs $4n^3$. This complexity very often overestimates the computational costs of the divide-and-conquer due to significant deflation that is observed surprisingly often. Finally, forming the left singular vectors $U = U_p V$ requires an additional $2n^3$ flops. Thus, the total arithmetic cost of QDWH-SVD with two-stage-EIG is up to $38n^3$.

6.3. Flops Comparison. Table 1 summarizes the flop counts of the standard DGESVD and DGESDD LAPACK functions and QDWH-SVD with its two variants of symmetric eigensolvers (QDWH-EIG and two-stage-EIG) for finding the singular values only and all singular vectors, additionally. QDWH-SVD with QDWH-EIG seems to be prohibitive due to the extra flops (33x and 4x more flops for singular values and additionally singular vectors, respectively), compared to the standard DGES{VD,DD} SVD algorithms. Although still expensive, QDWH-SVD with the two-stage-EIG seems to be more reachable with factors up to 14x and 2x in terms of the number of extra flops, compared to DGES{VD,DD}, when singular values and additionally singular vectors are needed, respectively.

QDWH-SVD	Σ	$U\Sigma V^\top$
DGES{VD,DD}	$(2+\frac{2}{3})n^3$	$22n^3$
w/QDWH-EIG	$(26+\frac{7}{9})n^3 \leq ... \leq (87+\frac{4}{9})n^3$	$(30+\frac{1}{9})n^3 \leq ... \leq (90+\frac{7}{9})n^3$
w/ two-stage-EIG	$12n^3 \leq ... \leq 38n^3$	$(20+\frac{2}{3})n^3 \leq ... \leq (46+\frac{2}{3})n^3$

Table 1: Algorithmic complexity ranges for various SVD solvers.

Before looking into performance results and how the GPU architecture allows QDWH-SVD to respond from the substantial extra flops, the next Section evaluates its numerical robustness against several synthetic matrices with various condition numbers.

7. Numerical Accuracy. In this Section, the orthogonality of the corresponding right and left singular vectors, the accuracy of the singular values and the backward error of the computed overall SVD for different matrix types are analyzed for the SVD algorithms (DGESVD, DGESDD and QDWH-SVD).

7.1. Synthetic Matrix Types. The numerical robustness of the SVD algorithms is assessed against a series of matrix types: (1) random matrices generated from DLARNV function in LAPACK, (2) well-conditioned synthetic matrices (*cond* equal to the 1) and (3) ill-conditioned synthetic matrices. Each dense synthetic matrix $A = QDQ^\top$ of the latter matrix type is generated using the LAPACK routine DLATMS, by initially setting a diagonal matrix $D = diag(\Sigma)$ containing the singular values, which follows a certain distribution and and from an orthogonal matrix Q generated from random entries. Table 2 enumerates the matrix types based on six different singular value distributions, which are all ill-conditioned with *cond* equal to the $1/(ulp)$, where *ulp* is the machine double precision. LAPACK uses these synthetic matrices to test the whole library during the nightly builds, especially the eigensolver and the SVD algorithms.

Type	Description
Type 1	$\sigma_1 = 1, \sigma_i = 1/cond,\ i = 2, ..., n$
Type 2	$\sigma_i = 1,\ i = 1, ..., n-1, \sigma_n = 1/cond$
Type 3	$\sigma_i = cond^{\frac{1-i}{n-1}},\ i = 1, ..., n$
Type 4	$\sigma_i = 1 - (\frac{i-1}{n-1})(1 - 1/cond),\ i = 1, ..., n$
Type 5	n random numbers $\in [1/cond, 1]$ their logarithms are uniformly distributed
Type 6	n random numbers following a uniform distribution

Table 2: List of ill-conditioned synthetic matrices with different singular value distributions.

7.2. Environment settings. The numerical accuracy assessments as well as the performance experiments have been run on a shared-memory multicore architecture composed of a dual-socket 10-core Ivy Bridge Intel(R) Xeon(R) CPU E5-2680 v2 (20 cores total), operating at 2.8 GHz. The system has 2.5 MB of L2 cache, 25 MB of L3 cache and 256 GB of DDR3 main memory. There are three NVIDIA Tesla K40 GPUs (ECC off) with 12 GB of main memory, each connected to the CPU through a PCIe bus 16x. The different implementations (all written in C) have been compiled using the Intel Compiler Suite v13.0.1 and NVIDIA CUDA compilation tools v6.0. The codes have been linked against the following numerical libraries: NVIDIA CUBLAS 6.0, Intel BLAS/LAPACK MKL and MAGMA v1.4.1 for CPU only and GPU only implementations, respectively.

7.3. Definition of Accuracy Metrics. For a given general matrix $A \in \mathbb{R}^{n \times n}$, the computed singular values are $\Sigma = diag(\sigma_1, \sigma_2, ..., \sigma_n)$. The corresponding left and right singular vectors are given by U and V. The various accuracy tests are as follows:

$$\frac{\|I - UU^\top\|}{n} \text{ and } \frac{\|I - VV^\top\|}{n}, \tag{7.1}$$

for the orthogonality of the left and right singular vectors (U and V),

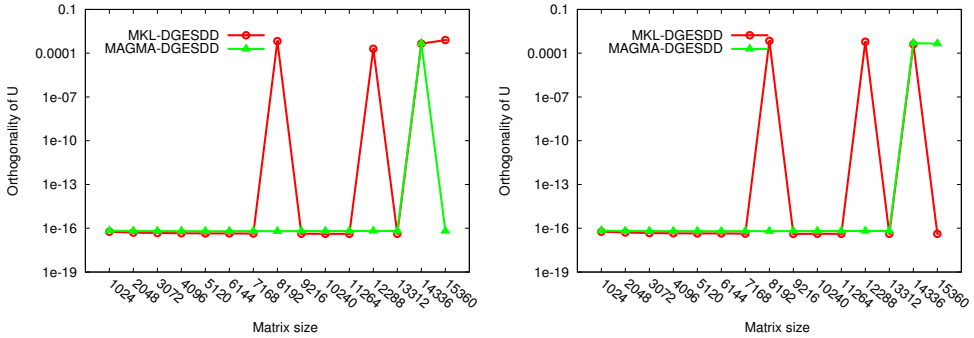
$$\frac{\|\sigma_i - \delta_i\|}{\|\sigma_i\|}, \tag{7.2}$$

for the accuracy of computed singular values, where δ_i are the exact singular values (analytically known or computed using LAPACK-DGESVD), and

$$\frac{\|A - U\Sigma V^\top\|}{\|A\| \times n}, \tag{7.3}$$

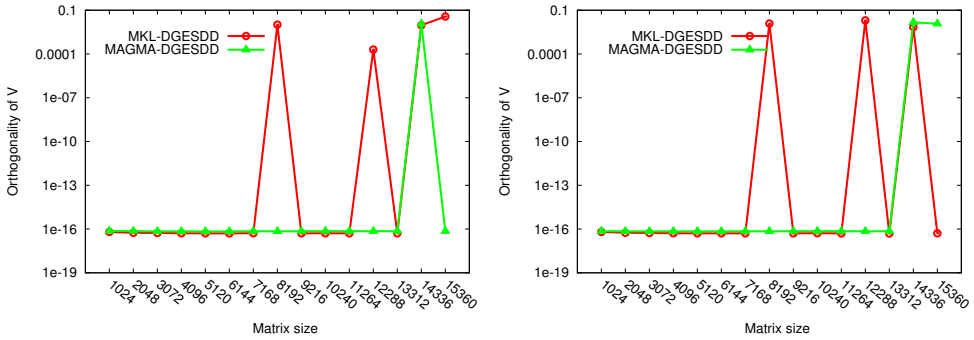
for the accuracy of the overall computed singular value decomposition. These standard accuracy metrics permit to numerically evaluate the stability of these new SVD algorithms compared to the existing state-of-the-art SVD implementations.

7.4. Accuracy Issues of DGESDD. This section highlights some of the issues encountered specifically when calling the SVD solver DGESDD. Figure 5 shows how the orthogonality of the left and right singular vectors (Equation 7.1) when using MKL-DGESDD and MAGMA-DGESDD fails to meet the double precision floating point arithmetic for well and ill-conditioned matrices (type 2) for certain sizes. This



(a) Well-conditioned matrix.

(c) Ill-conditioned matrix (type 2).



(b) Well-conditioned matrix.

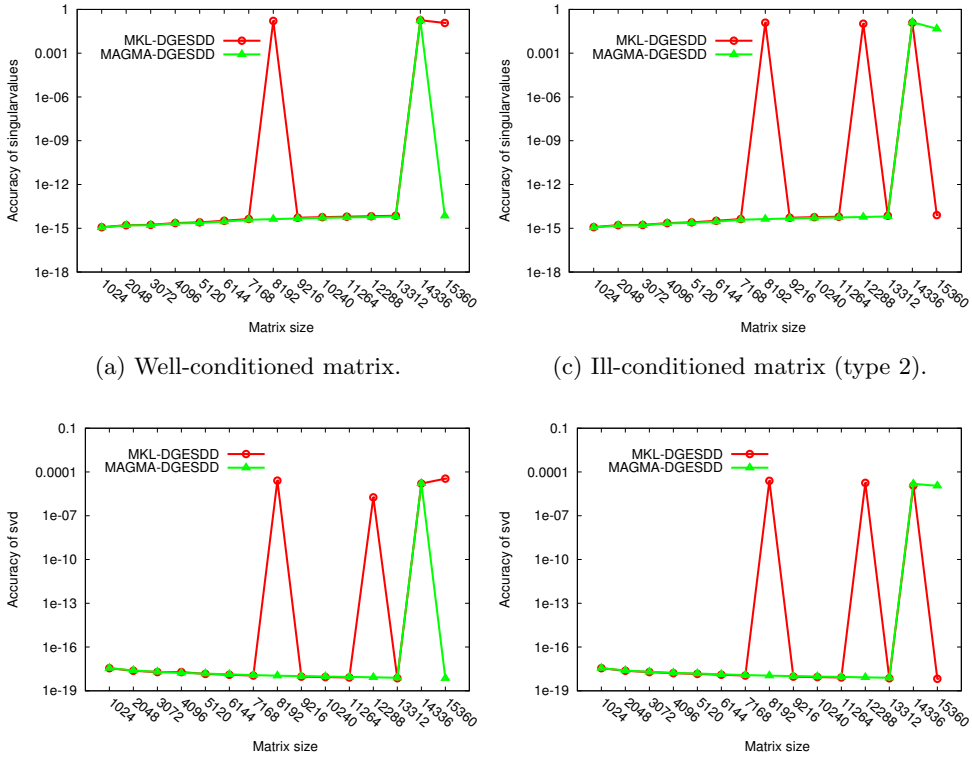
(d) Ill-conditioned matrix (type 2).

Fig. 5: Orthogonality of the left (a-c) and right (b-d) singular vectors.

precision loss obviously propagates to the accuracy of the singular values (Equation 7.2) and the overall computed SVD (Equation 7.3), as shown in Figure 6, due to convergence failure during the divide-and-conquer SVD solver. These accuracy problems are not surprising as these convergence issues are often reported online and in the literature. In particular, DGESDD is often not capable to converge when dealing with small singular values, as it is the case for matrix of type 2.

7.5. Accuracy Assessments of SVD Solvers. This section compares the numerical accuracy of DGESVD and DGESDD, as implemented in MKL (CPU) and MAGMA (GPU) libraries, against MAGMA-QDWH-SVD (GPU) on different matrix types. Since from an accuracy point of view MAGMA-QDWH-SVD with QDWH-EIG or two-stage-EIG gives a similar order of accuracy, only a single curve (MAGMA-QDWH-SVD with two-stage-EIG) is shown on the various graphs, for clarity purposes. By the same token, except for DGESDD and matrix type 2 (see previous Section 7.4), type 4 has been selected as a representative matrix type from Table 2 since the trend of the accuracy curves for all SVD solvers turn out to be quasi identical.

7.5.1. Orthogonality of the Singular Vectors. Figure 7 shows the orthogonality of all singular vectors. In particular, Figures 7(a), 7(b), 7(c) and Figures 7(d), 7(e), 7(f) present the orthogonality of the left and right singular vectors, respectively,



(a) Well-conditioned matrix.

(c) Ill-conditioned matrix (type 2).

(b) Well-conditioned matrix.

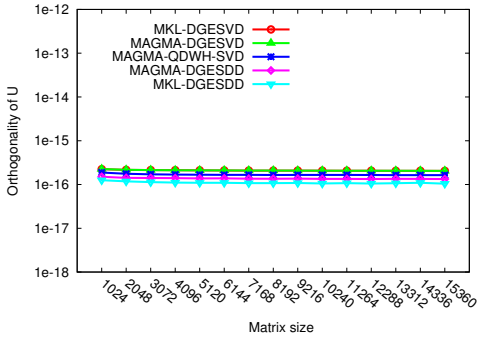
(d) Ill-conditioned matrix (type 2).

Fig. 6: Accuracy of the singular values (a-c) and the overall SVD (b-d).

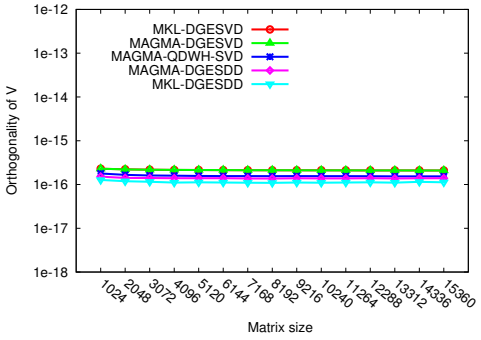
for an ill-conditioned matrix of type 4, a well-conditioned matrix, and a random matrix. All SVD solvers pass the orthogonality test (Equation 7.1) and present comparable orders of accuracy. Figures 7(b) and 7(e) do not show the orthogonality of the left and right singular vectors obtained from DGESDD, due to its accuracy issues revealed in Section 7.4.

7.5.2. Accuracy of the Singular Values and SVD Backward Error. Figure 8 presents the accuracy of the singular values (Equation 7.2) and the backward error of the overall SVD (Equation 7.3). In particular, Figures 8(a), 8(b), 8(c) and Figures 8(d), 8(e), 8(f) present the accuracy of the singular values and the backward error of the overall SVD solver, respectively, for an ill-conditioned matrix of type 4, a well-conditioned matrix, and a random matrix. The accuracy curves for all SVD solvers show that all numerical accuracy tests pass, thanks to an order of accuracy close to the double precision floating-point arithmetic. Similarly, Figures 8(b) and 8(e) do not show the accuracy of the singular values and the backward error of the overall SVD solver obtained from DGESDD, due to its accuracy issues described in Section 7.4.

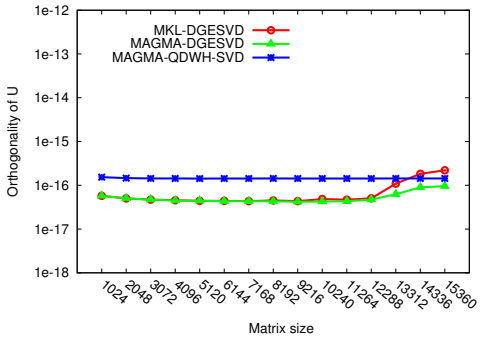
7.6. Accuracy Results Using Mixed Precision. Since the convergence speed of QDWH-SVD framework is remarkable, a mixed precision technique has been inte-



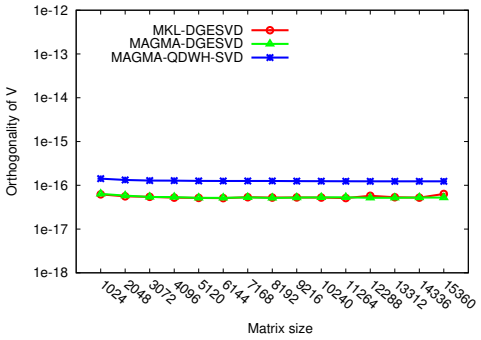
(a) Ill-conditioned matrix (type 4).



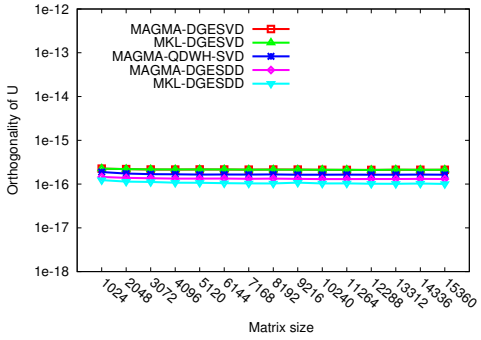
(d) Ill-conditioned matrix (type 4).



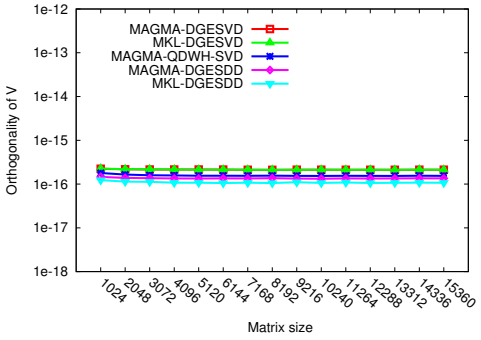
(b) Well-conditioned matrix.



(e) Well-conditioned matrix.



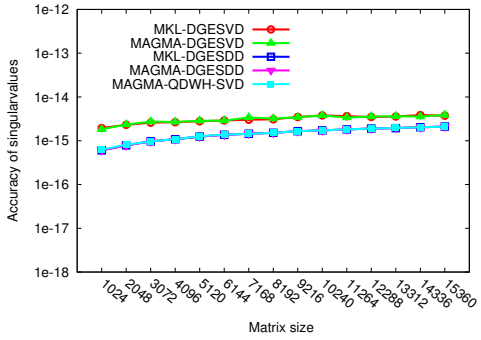
(c) Random matrix.



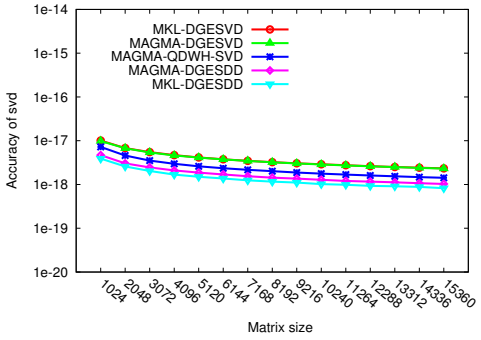
(f) Random matrix.

Fig. 7: Orthogonality of the left (a-b-c) and right (d-e-f) singular vectors.

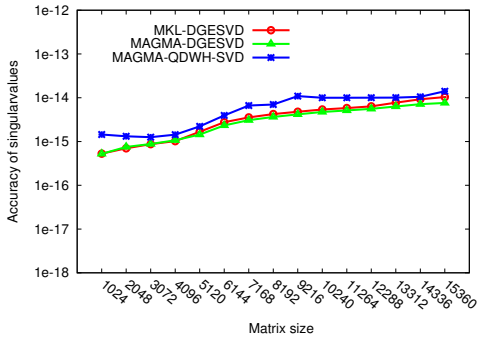
grated into the original QDWH-SVD algorithm so that the first iterations are done in single precision (SP) and the subsequent ones in double precision (DP) in order to be able to recover some of the lost digits. This type of precision play, to maintain most of the value of a strictly high precision implementation while doing a significant fraction of the work in faster and cheaper arithmetic, is typical of algorithmic adaptations to extreme architectures to come. Unfortunately, this novel contribution does not work



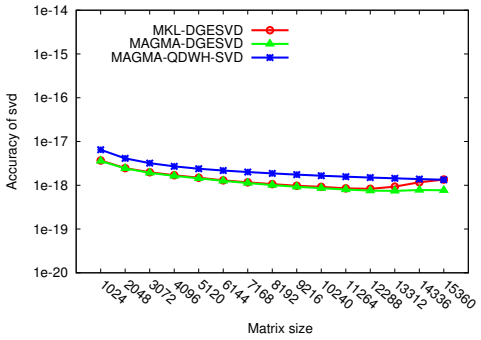
(a) Ill-conditioned matrix (type 4).



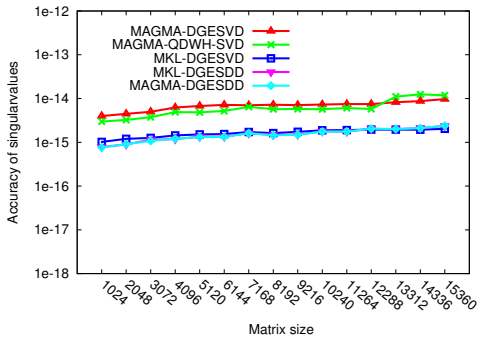
(d) Ill-conditioned matrix (type 4).



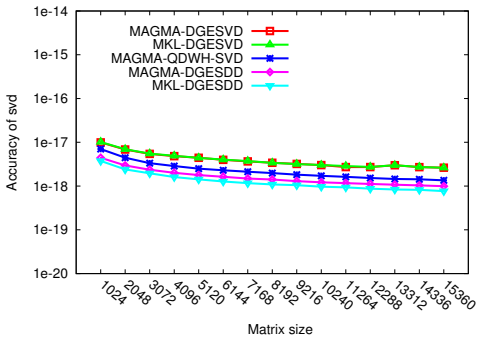
(b) Well-conditioned matrix.



(e) Well-conditioned matrix.



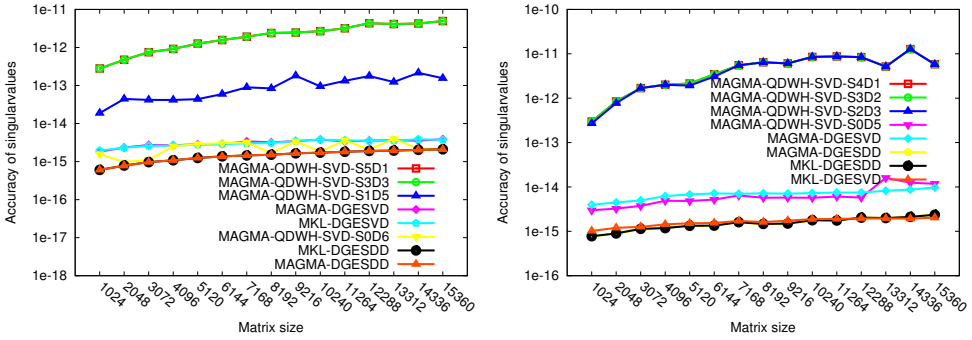
(c) Random matrix.



(f) Random matrix.

Fig. 8: Accuracy of the singular values (a-b-c) and backward error of the overall SVD (d-e-f).

when singular vectors are additionally needed, even though their orthogonality is preserved. The loss of digits in the singular values directly affects the backward error of the overall SVD solver (Equation 7.3), which makes the mixed precision technique suitable only for calculating singular values. Figure 9 shows the impact of the mixed precision technique on the numerical accuracy of the MAGMA-QDWH-SVD singular



(a) Ill-conditioned matrix (type 4). (b) Random matrix.

Fig. 9: Accuracy of the mixed precision MAGMA-QDWH-SVD solver.

values on an ill-conditioned matrix (type 4) and a random matrix only, since for a well-conditioned matrix, the QDWH-SVD framework rapidly converges after two iterations. By increasing the number of iterations in SP over DP, the accuracy of the singular values of MAGMA-QDWH-SVD degrades and it loses two to three digits at most, compared to the other SVD solvers, including the full DP MAGMA-QDWH-SVD-S0D6. This can still be of interest for applications that show tolerance to a loss of few digits.

8. Experimental Results. The performance results have been obtained on the system described in Section 7.2. All subsequent performance graphs show the total elapsed time (seconds) in logarithmic scale against the matrix size, unless mentioned otherwise.

8.1. Performance Comparisons of QDHW-SVD Variants on Single GPU. Figure 10 shows the performance of both symmetric eigensolvers (MAGMA-QDWH-EIG and MAGMA-two-stage-EIG) on random matrices, when calculating all singular vectors. Because MAGMA-QDWH-EIG performs around 6.5x more flops than MAGMA-two-stage-EIG, there is almost an order of magnitude difference in terms of elapsed time between them, when looking at asymptotic matrix sizes. Figure 11 highlights the overall performance of the two QDWH-SVD variants, after plugging in QDWH-EIG or two-stage-EIG, and shows up an overall twofold speedup when selecting two-stage-EIG over QDWH-EIG as the symmetric eigensolver. For the subsequent performance graphs, MAGMA-QDWH-SVD will always refer to QDWH-SVD with the symmetric eigensolver two-stage-EIG.

8.2. Performance Comparisons of All SVD Solvers on Single GPU. Figure 12 presents the performance comparisons of all SVD solvers (i.e., DGESVD, DGESDD and QDWH-SVD) for an ill-conditioned matrix of type 4, a well-conditioned matrix , and a random matrix. In particular, Figures 12(a), 12(b), 12(c) and Figures 12(d), 12(e), 12(f) depict the performance comparisons when computing the singular values and additionally all singular vectors, respectively. Figures 12(b) and 12(e) do not draw the performance curves of MKL/MAGMA-DGESDD, again because of the accuracy issued mentioned in Section 7.4. When calculating only the singular values, the MKL variants (CPU) of DGESVD and DGESDD outperform

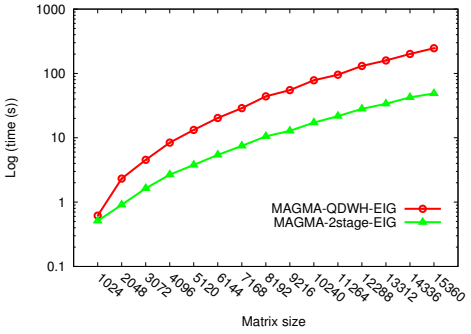


Fig. 10: Performance of both symmetric eigensolvers.

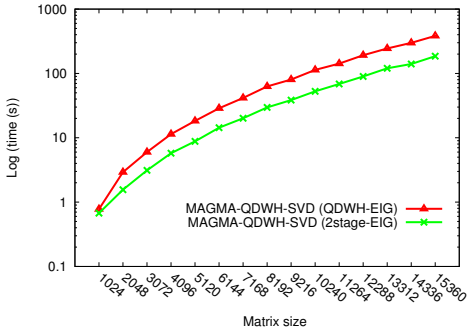


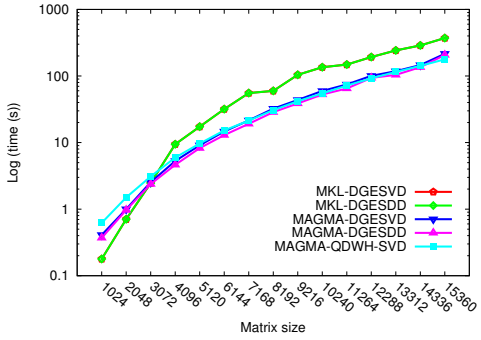
Fig. 11: Performance of various QDWH-SVD implementations.

all MAGMA-DGESVD/DGESDD/QDWH-SVD versions on small matrix sizes, due to the low arithmetic complexity which unveils the overhead of moving data to the device. The same analysis also applies when singular vectors are required additionally for MKL-DGESVD (CPU), only for small well-conditioned matrix sizes as in Figure 12(e), thanks to a faster convergence of the SVD solvers. For large matrix sizes, when only singular values are calculated, MAGMA-QDWH-SVD shows competitive performance compared to other SVD solvers on single GPU for ill-conditioned and random matrices, although MAGMA-QDWH-SVD performs more than 14x more flops than other SVD solvers, and even achieves a twofold and 3.5x speedups against MAGMA-DGESVD (GPU) and MKL-DGESVD (CPU) for well-conditioned matrices, thanks to a reduced number of flops. When singular vectors are required additionally, MAGMA-QDWH-SVD outperforms the SVD solvers on all matrix types, with up to 18% and 30% compared to MAGMA-DGESDD, for ill-conditioned and random matrices, respectively, and achieves up to twofold and 3.5x speedups against MAGMA-DGESVD (GPU) and MKL-DGESVD (CPU) for well-conditioned matrix type, respectively.

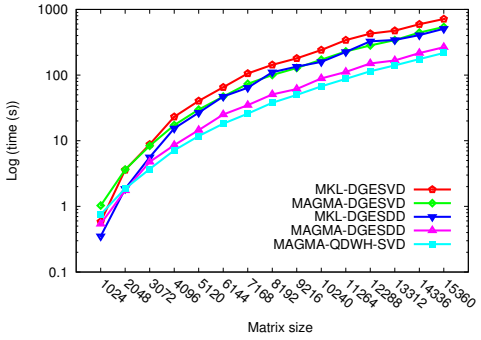
Figure 13 assesses the performance for computing singular values using the mixed precision technique. As expected, time to solution of MAGMA-QDWH-SVD decreases when executing more QDWH iterations in single precision floating-point arithmetic. The mixed precisions technique allows to further reduce time to solution and brings an additional 40% performance improvement compared to the full double precision floating-point arithmetic, at the expense of losing two to three digits of accuracy at most.

Last but not least, Figure 14 shows the time breakdown between the main phases of MAGMA-QDWH-SVD for random matrices. It is clear from this profiling Figure that the polar decomposition (QDWH) is the most time consuming stage and its elapsed time increases substantially as the matrix size gets larger.

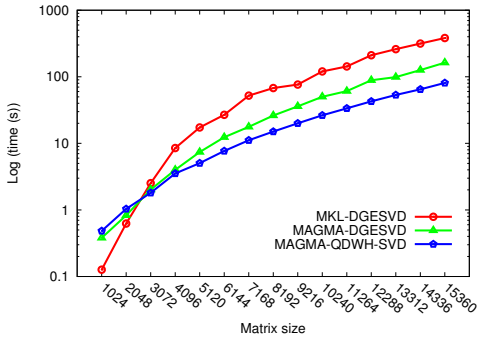
8.3. Scalability of MAGMA-QDWH-SVD on Multiple GPUs. Figure 15 shows the performance scalability of the multiple GPU implementation of MAGMA-QDWH-SVD. In fact, this corresponds to the first multiple GPU implementation of an SVD solver (to our knowledge). The curves show the speedup of MAGMA-QDWH-SVD up to three K40 GPUs. This first attempt presents a speedup of roughly 1.7 and 2.1 on two and three GPUs, respectively. The scalability is somewhat limited mainly



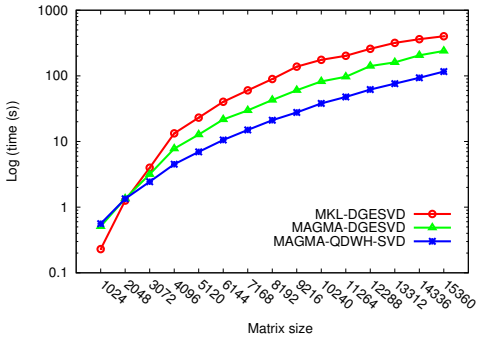
(a) Ill-conditioned matrix (type 4).



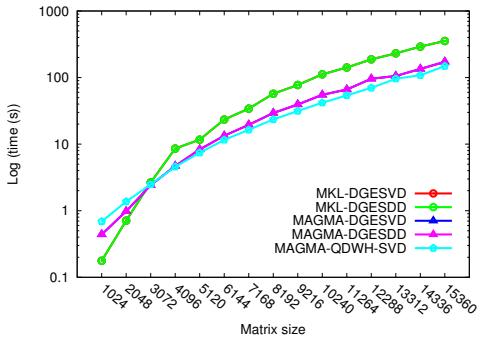
(d) Ill-conditioned matrix (type 4).



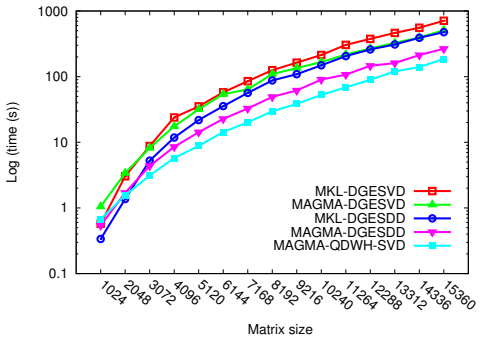
(b) Well-conditioned matrix.



(e) Well-conditioned matrix.



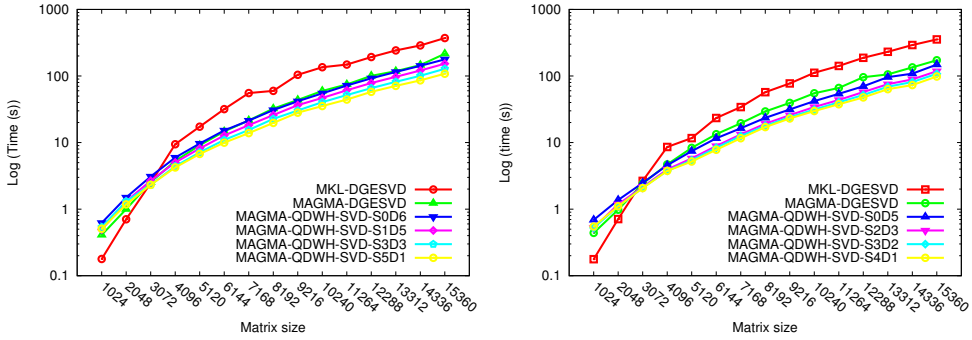
(c) Random matrix.



(f) Random matrix.

Fig. 12: Performance comparisons for Σ (a-b-c) and additionally $U\Sigma V^\top$ (d-e-f).

because of the overhead of moving data between the host and the three GPUs. There is obviously still room for improvement by doing more GPU computations and further hiding the communication overhead by computations, which are doable thanks to the nature of operations (compute-bound) occurring in the polar decomposition step.



(a) Ill-conditioned matrix (type 4). (b) Random matrix.

Fig. 13: Performance comparisons (Σ only) using mixed precision techniques.

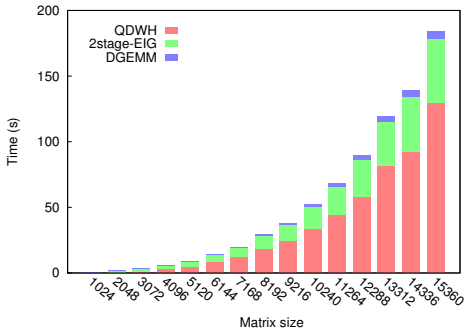


Fig. 14: Profiling the Computational Stages of MAGMA-QDWH-SVD

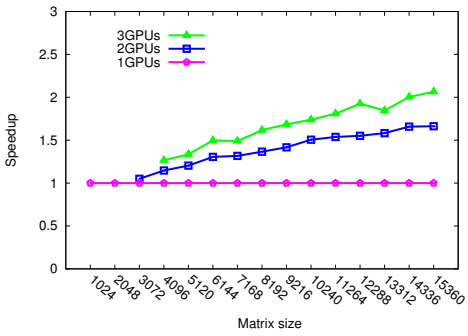


Fig. 15: Performance scalability of MAGMA-QDWH-SVD on multiple GPUs.

9. Conclusions and Future Work. A new high performance implementation of the QR-based Dynamically Weighted Halley Singular Value Decomposition (QDWH-SVD) solver has been presented on multicore architecture enhanced with GPUs. Based on three successive computational stages: (1) the polar decomposition calculation of the original matrix using the QDWH algorithm, (2) the symmetric eigendecomposition of the resulting polar factor to obtain the singular values and the right singular vectors, and (3) the matrix-matrix multiplication to get the associated left singular vectors, the GPU accelerated QDWH-SVD framework outperforms the equivalent routines from existing state-of-the-art commercial (Intel MKL) and open-source GPU libraries (MAGMA) by up to 3.5x and twofold speedups for asymptotic matrix sizes, respectively, although it performs up to two times more flops when computing all singular vectors. When only singular values are needed, the number of extra flops is further exacerbated (up to 14x) compared to the standard methods. The singular value only implementation of QDWH-SVD can still recover and run up to 18% faster than the best existing equivalent routines. Integrating mixed precisions techniques in the solver can further provide up to 40% improvement at the price of

losing two to three digits of accuracy, compared to the full double precision floating point arithmetic version. The comprehensive numerical testing confirms the robustness of the QDWH-SVD solver. The provided multi-GPU implementation shows a decent scalability of the overall framework and constitutes the first multi-GPU SVD solver. QDWH-SVD will be eventually integrated into the MAGMA library for further community dissemination.

The gains in execution time of QDWH-SVD over the current state of the art are poised to become more significant in computational environments that are yet more austere in memory access. Thanks to the conventional kernels (QR/Cholesky factorizations, GEMM, etc.) upon which QDWH-SVD relies, porting the framework to other architectures such as Intel Xeon Phi, AMD APU, or even ARM processors seems straightforward as long as an optimized vendor BLAS/LAPACK library is available on the underlying system. Another research direction will be to replace current block algorithms (coarse-grained) used in the QDWH-SVD framework by tile DAG-scheduled algorithms (fine-grained). This will allow to pipeline the various computational stages and to maintain useful computing in all processing cores. Finally, the authors are currently designing an implementation of QDWH-SVD for distributed memory systems.

Acknowledgement. This work was supported by the Extreme Computing Research Center at KAUST. The authors would like to thank Ahmad Abdelfattah for his help to integrate KBLAS into QDWH-SVD and NVIDIA for the hardware donations.

REFERENCES

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov, *Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects*, in Journal of Physics: Conference Series, vol. 180, 2009.

[2] Edward Anderson, Zhaojun Bai, Christian Heinrich Bischof, Laura Susan Blackford, James Weldon Demmel, Jack J Dongarra, Jeremy J Du Croz, Anne Greenbaum, Sven Hammarling, A McKenney, and Danny C Sorensen, *LAPACK User's Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 3rd ed., 1999.

[3] Grey Ballard, James Demmel, and Ioana Dumitriu, *Minimizing Communication for Eigenproblems and the Singular Value Decomposition*, CoRR, abs/1011.3077 (2010).

[4] Christian H. Bischof, Bruno Lang, and Xiaobai Sun, *Algorithm 807: The SBR Toolbox—Software for Successive Band Reduction*, ACM Transactions on Mathematical Software, 26 (2000), pp. 602–616.

[5] BLAS, *Basic Linear Algebra Subprograms.* `http://www.netlib.org/blas/`.

[6] James Demmel and W Kahan, *Computing Small Singular Values of Bidiagonal Matrices With Guaranteed High Relative Accuracy*, SIAM Journal on Scientific and Statistical Computing, 5 (1990), pp. 873–912.

[7] Jack Dongarra and P. Beckman et al., *The international exascale software project*, International Journal of High Performance Computer Applications, 25 (2011), pp. 309–322. ISSN 1094-3420.

[8] Gene H. Golub and C. Reinsch, *Singular Value Decomposition and Least Squares Solutions*, Numerische Mathematik, 14 (1970), pp. 403–420.

[9] Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, John Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, Baltimore, Maryland, third ed., 1996.

[10] Ming Gu and Stanley C. Eisenstat, *A Divide-and-Conquer Algorithm for the Bidiagonal SVD*, SIAM Journal on Matrix Analysis and Applications, 16 (1995), pp. 79–92.

[11] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek, *An Improved Parallel Singular Value Algorithm and Its Implementation for Multicore Hardware*, Proceedings of the Interna-

tional Conference on High Performance Computing, Networking, Storage and Analysis, (2013), pp. 90:1–90:12.

[12] Azzam Haidar, Hatem Ltaief, and Jack Dongarra, *Parallel reduction to condensed forms for symmetric Eigenvalue problems using aggregated fine-grained and memory-aware kernels*, in Proceedings of SC'11 Conference on High Performance Computing Networking, Storage and Analysis, Seattle, WA, USA, Nov. 2011, ACM SIGARCH/IEEE Computer Society, p. 8.

[13] Azzam Haidar, Stanimire Tomov, Jack Dongarra, Raffaele Solc, and Thomas C. Schulthess, *A Novel Hybrid CPU-GPU Generalized Eigensolver for Electronic Structure Calculations Based on Fine-Grained Memory Aware Tasks*, 2014, pp. 196–209.

[14] Per Christian Hansen, *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*, Mathematical Modeling and Computation, Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[15] Nicholas J. Higham and Pythagoras Papadimitriou, *Parallel Singular Value Decomposition via the Polar Decomposition*, Numerical Analysis Report No. 239, University of Manchester, England, Oct 1993.

[16] Intel, *Math Kernel Library*. Available at `http://software.intel.com/en-us/articles/intel-mkl/`.

[17] K Vince Fernando and Beresford N Parlett, *Accurate Singular Values and Differential QD Algorithms*, Numerische Mathematik, 67 (1994), pp. 191–229.

[18] Bruno Lang, *Efficient eigenvalue and singular value computations on shared memory machines*, Parallel Computing, 25 (1999), pp. 845–860.

[19] Hatem Ltaief, Jakub Kurzak, and Jack Dongarra, *Parallel Band Two-Sided Matrix Bidiagonalization for Multicore Architectures*, IEEE Transactions on Parallel and Distributed Systems, 21 (2010).

[20] Hatem Ltaief, Piotr Luszczek, Azzam Haidar, and Jack Dongarra, *Solving the Generalized Symmetric Eigenvalue Problem using Tile Algorithms on Multicore Architectures*, in PARCO, Koen De Bosschere, Erik H. D'Hollander, Gerhard R. Joubert, David A. Padua, Frans J. Peters, and Mark Sawyer, eds., vol. 22 of Advances in Parallel Computing, IOS Press, 2011, pp. 397–404.

[21] Piotr Luszczek, Hatem Ltaief, and Jack Dongarra, *Two-Stage Tridiagonal Reduction for Dense Symmetric Matrices using Tile Algorithms on Multicore Architectures*, in Proceedings of IPDPS 2011, Anchorage, AK USA, 2011, ACM.

[22] MAGMA, *Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee. Available at* `http://icl.cs.utk.edu/magma/`, 2009.

[23] Yuji Nakatsukasa, Zhaojun Bai, and Franois Gygi, *Optimizing Halley's Iteration for Computing the Matrix Polar Decomposition*, SIAM Journal on Matrix Analysis and Applications, (2010), pp. 2700–2720.

[24] Yuji Nakatsukasa and Nicholas J.Higham, *Stable and Efficient Spectral Divide and Conquer Algorithms for the Symmetric Eigenvalue Decomposition and the SVD*, SIAM Journal on Scientific Computing, 35 (2013), pp. A1325–A1349.

[25] Lloyd N. Trefethen and David Bau, *Numerical Linear Algebra*, SIAM, Philadelphia, PA, 1997.

[26] Asim YarKhan, Jakub Kurzak, and Jack Dongarra, *QUARK Users' Guide: QUeueing And Runtime for Kernels*, University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02, (2011).