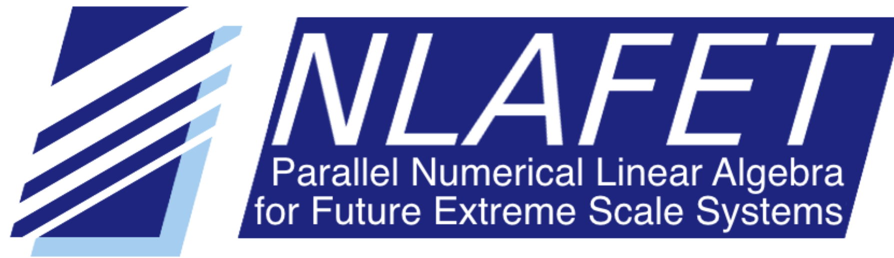

H2020–FETHPC–2014: GA 671633

D2.9

Novel SVD Algorithms

April 2019

Document information

Scheduled delivery 2019-04-30
Actual delivery 2019-04-29
Version 2.0
Responsible partner UNIMAN

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2019-04-29	Pierre Blanchard	Complete	2.0	Revision for final version
2019-04-26	Pierre Blanchard	Draft	1.1	Feedback from reviewers added.
2019-04-18	Pierre Blanchard	Draft	1.0	Feedback from reviewers added.
2019-04-11	Pierre Blanchard	Draft	0.1	Initial version of document produced.

Author(s)

Pierre Blanchard (UNIMAN)
Mawussi Zounon (UNIMAN)
Jack Dongarra (UNIMAN)
Nick Higham (UNIMAN)

Internal reviewers

Nicholas Higham (UNIMAN)
Sven Hammarling (UNIMAN)
Bo Kågström (UMU)
Carl Christian Kjelgaard Mikkelsen (UMU)

Copyright

This work is ©by the NLAFET Consortium, 2015–2018. Its duplication is allowed only for personal, educational, or research uses.

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

1 Introduction **4**

2 Review of state-of-the-art SVD algorithms **5**
2.1 Notation . . . 5
2.2 Computing the SVD . . . 5
2.3 Symmetric eigenvalue solvers . . . 6
2.4 Flops count . . . 7

3 Polar Decomposition based SVD **7**
3.1 Polar Decomposition . . . 8
3.2 Applications . . . 8
3.3 The `Polar-SVD` Algorithm . . . 8
3.4 The QR trick . . . 9

4 Optimized Polar Decomposition algorithms **9**
4.1 Computing the Polar Factor U_p . . . 10
4.2 Optimized Halley's iterations: the `QDWH-PD` algorithm . . . 10
4.3 Estimating $K_2(A)$ to ensure optimal convergence . . . 13
4.4 Complexity . . . 14

5 Implementation in Shared and Distributed Memory **15**
5.1 `QDWH-SVD` for Multi– and Many–core Architectures . . . 15
5.2 `QDWH-SVD` in Distributed Memory . . . 16
5.3 `QDWH-PD` on Accelerators . . . 17

6 Numerical Experiments **17**
6.1 Benchmarking the Polar–SVD . . . 17
6.2 Experiments on a Single Node . . . 19
6.3 Experiments in Distributed Memory . . . 21
6.4 Experiments with Accelerators . . . 22

7 Conclusions **22**

8 Appendix **26**
8.1 Remarks on Accuracy and Precision . . . 26
8.2 Benchmarking SLATE . . . 28

List of Figures

1 Schematic view on the 2-stage reduction. . . . 6
2 Time to solution and performance of `QDWH-PD` using QUARK or OpenMP. 20
3 Time to compute SVD using Plasma's `2-stage SVD` or `QDWH-SVD`. . . . 20
4 Time to compute SVD using MKL's `2-stage SVD` or Plasma's `QDWH-SVD` on Haswell and KNL. . . . 21
5 Scalability of `2-stage SVD` and `QDWH-SVD`. . . . 22
6 Detailed timing of `QDWH-SVD` on 8 Broadwell or Skylake nodes. . . . 23
7 Performance (TFlops/s) of the GPU accelerated `QDWH-PD`. . . . 24

8 Accuracy of QDWH-SVD and 2–stage approaches. 27
9 Ratio between the time to solution in double and single precision. 27
10 Performance of QDWH-PD's building blocks in SLATE. 28

1 Introduction

The *Description of Action* document states for deliverable D2.9:

> "*D2.9 Novel SVD Algorithms*
>
> Prototypes for the standard SVD algorithm, the symmetric eigenvalue problem and the ODWH-based SVD algorithm."

This deliverable is in the context of Task 2.4 Singular value decomposition algorithms.

Emerging computer architectures show increasing floating point capabilities. Additionally, current and upcoming supercomputers have extremely heterogeneous computing resources as they consists of many NUMA nodes with increasing numbers of cores and SIMD instructions with increasing lengths, e.g., Intel's AVX2 and AVX512 or ARM's SVE. They are also massively accelerated by graphic processing units (GPUs) and co–processors (Intel Knights Landing or KNL). Scalability of algorithms is always crucial, however in such an heterogeneous distributed memory environment it is very difficult to guarantee. Therefore, algorithms with higher levels of concurrency should be preferred and new parallel paradigms should be considered such as asynchronous execution and task scheduling. Furthermore, in order to use modern CPUs at full efficiency algorithms in numerical linear algebra must be redesigned to make better use of cache–efficient matrix operations such as level–3 BLAS routines.

In this report we are interested in algorithms to perform the Singular Value Decomposition (SVD), a keystone of scientific computing and data analysis. SVD algorithms based on an initial bidiagonal reduction are known to be memory bandwidth–bound and communication intensive. Thus, they fail to use the full compute capabilities of novel architectures and exhibit poor scalability. Splitting the reduction phases in 2 stages in order to make better use of level–3 BLAS operations is now considered the state-of-the-art approach. Please note that deliverable 2.8 *Bidiagonal factorization* is concerned with optimization of the first stage of the reduction in general bidiagonal factorizations with focus on SVD. Simpler approaches like Jacobi are easier to parallelize but usually perform worse than methods based on bidiagonal reduction.

Recent papers [9, 12] have pointed out the relevance of using the polar decomposition as a tool to compute standard matrix factorization such as the SVD, eigenvalue decomposition and matrix functions. These algorithms use many more flops than the standard implementations but make better use of the cache memory hierarchy as they rely mainly on matrix multiplications, Cholesky and QR factorizations. As a result, for sufficiently large problems and on sufficiently recent architectures they can provide a full SVD factorization in less time than even 2–stage reduction based variants.

In this report we evaluate the performance of a stable and scalable algorithm known as QDWH iterations to perform the SVD and compare it to state-of-the-art implementations of the 2–stage SVD. We show comparative numerical results using implementations in several numerical linear algebra libraries: for multicore architecture (PLASMA), distributed memory (ScaLAPACK) and heterogeneous architectures involving accelerators (SLATE).

2 Review of state-of-the-art SVD algorithms

SVD algorithms usually divide in 2 main classes: algorithms based on bidiagonal reduction or based on Jacobi method. In this section we briefly describe the former and highlight some numerical limitations arising at large scale. An extensive description of all approaches can be found in the recent review paper [1].

In this report we focus on optimized algorithms to compute singular values and vectors as opposed to singular values only. In fact, as our first numerical experiments will clearly show novel algorithms based on the polar decomposition such as `QDWH-SVD` are too expensive for the computation of singular values only.

2.1 Notation

For any integer m and n, we denote $\mathbb{R}^{m\times n}$ and $\mathbb{C}^{m\times n}$ the sets of real and complex valued m-by-n matrices, respectively. We denote $\mathcal{O}^{m\times n}$ and $\mathcal{U}^{m\times n}$ the set of real and complex matrices with orthonormal columns, respectively. If $m = n$ then they denote the sets of orthogonal and unitary matrices, respectively. Finally, $\mathcal{D}^{m\times n}$ denotes the set of real diagonal m-by-n matrices.

For any symmetric matrix $H \in \mathbb{R}^{n\times n}$, there exists an eigenvalue decomposition, i.e., a set of factors $(W, \Lambda) \in \mathcal{D}^{n\times n} \times \mathcal{O}^{n\times n}$ such that

$$H = W\Lambda W^* \tag{1}$$

where W contains the eigenvectors of H and Λ contains the associated eigenvalues.

For any matrix $A \in \mathbb{R}^{m\times n}$ the SVD of A is defined as the set of matrices $(U, \Sigma, V) \in \mathcal{O}^{m\times m} \times \mathcal{D}^{m\times n} \times \mathcal{O}^{n\times n}$ such that

$$A = U\Sigma V^* \tag{2}$$

where U and V contain the left and right singular vectors respectively, and Σ contains the associated singular values of A (eigenvalues of A^*A). We will denote $\sigma_i(A) = \Sigma_{ii}$ the i-th singular value of A, $\alpha = \max_k \sigma_k(A)$ and $\beta = \min_k \sigma_k(A)$.

If $m > n$, a more compact representation of A (known as economy size SVD) can be used where $U \in \mathcal{O}^{m\times n}$ and $V \in \mathcal{O}^{n\times n}$ and $\Sigma \in \mathcal{D}^{n\times n}$.

If $m \ll n$, i.e., the matrix A is *tall and skinny*, a typical trick consists in computing a preliminary QR factorization of A, followed by the SVD of the R factor that we write $R = U_R\Sigma V^*$. Hence, the left singular vectors of A are obtained by forming the matrix $U = QU_R$.

For the sake of clarity, we will mainly consider the case of real square matrices, i.e., $A \in \mathbb{R}^{n\times n}$. Extension of the method discussed in this report to the complex case is straightforward.

2.2 Computing the SVD

Bidiagonal SVD One of the most common approaches for computing the SVD of a dense matrix consists in first reducing the matrix to bidiagonal form, then using a direct solver to compute the SVD of the reduced matrix. Methods for computing the SVD of a bidiagonal matrix rely on either the QR decomposition or a Divide and Conquer process (D&C). The latter is usually preferred when both the singular values and vectors are required.

1–stage approach The reduction can be done in one stage by applying orthogonal transformations to the input matrix. Each transformation can be expressed as a product of elementary Householder reflectors. However this method is known to be memory–bound and thus shows very limited performance on modern computer architectures [1]. In order to address this issue a variant making use of blocked Householder transformations was introduced in LAPACK [2]. It involves a total of about $8n^3/3$ flops if A is square, where the flops count is equally split between level–2 and level–3 BLAS operations.

2–stage approach In order to further increase the use of BLAS 3 operations, in [3, 6] the bidiagonalization is itself split into 2 stages as described on Fig 1: a reduction to band format (stage 1) followed by a reduction to bidiagonal format (stage 2) based on *bulge chasing* a memory–bound routine. The first stage can be implemented in $8n^3/3$ flops and is rich in level–3 BLAS operations, thus reaching similar performance as the QR decomposition.

Multistage successive band reduction (SBR) can also be considered and have been implemented in the latest version of Intel MKL's LAPACK and ScaLAPACK. However they increase the cost of constructing the singular vectors (back transformation), hence they are not considered in this work.

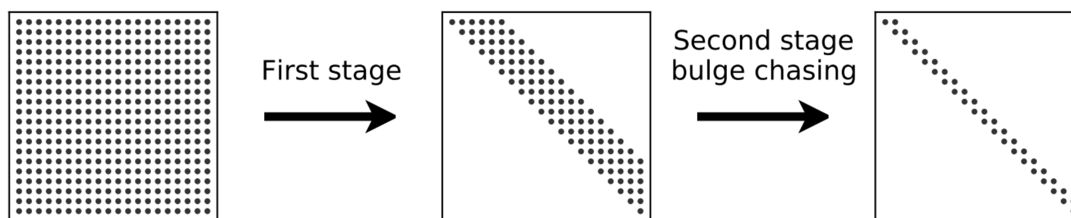

Figure 1: Schematic view on the 2-stage reduction: reduction to band format then to bidiagonal format using *bulge chasing*. Source: [1].

2.3 Symmetric eigenvalue solvers

In this deliverable we are not directly concerned with symmetric eigensolvers, i.e., computing (1). However they play a central role in Polar Decomposition–based SVD algorithms. Therefore, we recall fundamentals of the state-of-the-art approach.

Tridiagonal reduction Implementation of symmetric eigensolvers are usually based on a reduction to tridiagonal format followed by a direct tridiagonal solve of the eigenvalue problem. Similarly to the SVD, this reduction is performed using orthogonal transformations. The 2–stage variants make use of level–3 BLAS routines by first reducing the matrix to block tridiagonal format then to tridiagonal format using *bulge–chasing*.

Tridiagonal eigensolver There exists many variants for the tridiagonal eigensolver using either QR (`syev` routine in LAPACK), D&C (`syevd`), MRRR (`syevr`) or bisection plus inverse iterations (`syevr`). The performance and accuracy of each method depends on the structure of the input matrix (e.g., clustering of eigenvalues) the necessity to compute

Table 1: Flops count ($/n^3$) for SVD algorithms (`gesvd`) and symmetric eigenvalue solvers (`syev`) for either all values only or all values and vectors.

Problem	`gesvd`		`syev`	
Output(s)	Σ	U, Σ, V	Λ	W, Λ
QR-based	$\frac{8}{3}$	≈ 17	$\frac{4}{3} = \mathbf{1} + \frac{1}{3}$	$\frac{4}{3} + 6 = \frac{22}{3} = \mathbf{7} + \frac{1}{3}$
D & C	$\frac{8}{3}$	≈ 9	$\frac{4}{3} = \mathbf{1} + \frac{1}{3}$	$\frac{8}{3} + \frac{4}{3} = \mathbf{4}$
2-Stage	$\frac{10}{3} = \mathbf{3} + \frac{1}{3}$	$\frac{10}{3} + 4 = \mathbf{7} + \frac{1}{3}$	$\frac{4}{3} = \mathbf{1} + \frac{1}{3}$	$\frac{4}{3} + (\frac{8}{3} + \frac{4}{3}) = \mathbf{5} + \frac{1}{3}$

eigenvectors and the number of MPI processes. As a reference, the Intel MKL team gives the following informations[1]: “The EV driver is the slowest but also the most robust. The EVX driver is the fastest but is more likely to fail for clustered eigenvalues. The EVR driver struggles to scale. The EVD driver is slightly slower than EVX and should be used only when eigenvectors are required.” In this report we will use the divide and conquer approach as it is a good compromise between speed and accuracy for computing eigenvalues and eigenvectors.

Implementation Robust symmetric eigensolvers can also be obtained using the polar decomposition. In particular, a spectral divide and conquer symmetric eigensolver based on QDWH is presented in [12]. However, preliminary experiments showed that it is not very competitive with state-of-the-art approach in practice.

2.4 Flops count

One of the main points we make in the present report is that a routine involving many more flops than another one does not necessarily result in a longer time to solution. In fact, depending on the nature of the operations (cache efficient/compute–bound) and the scalability of the algorithm it can actually run much faster. Table 1 summarizes the flop counts associated with several SVD algorithms and symmetric eigenvalue solvers. Although these numbers are much smaller than those associated with `QDWH-SVD` (Section 4), namely $7n^3$ for `2-stage SVD` and about 15 to $40n^3$ for `QDWH-SVD`, we show in Section 6 that on recent CPU architectures `QDWH-SVD` can run faster than `2-stage SVD`.

3 Polar Decomposition based SVD

This section provides some basic knowledge on the polar decomposition and explains why it is relevant to use it in SVD algorithms and eigensolvers. A generic algorithm is provided for rectangular matrices using a generic polar decomposition algorithm denoted `polar`().

[1]https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-2018-update-2-scalapack-symmetric-eigensolver

Algorithms for computing the polar decomposition are presented in Section 4, where focus is made on QDWH iterations.

3.1 Polar Decomposition

The polar decomposition generalizes the polar representation of a complex numbers to the case of matrices. More precisely, for any matrix $A \in \mathbb{R}^{m\times n}$ there exists a set of orthonormal columns $U_p \in \mathcal{O}^{m\times n}$ and an Hermitian matrix $H \in \mathbb{R}^{n\times n}$ such that

$$A = U_p H. \tag{3}$$

The matrix U_p is also known as the unitary *polar factor* and the H as *Hermitian factor*. The matrix representation (3) is called a polar decomposition (PD) and is equivalent to the SVD. If the matrix is non-singular, then the polar decomposition is unique.

3.2 Applications

The polar decomposition is a keystone in the computation of matrix functions [5], being very closely related to the matrix sign function [10]. Both the unitary and the Hermitian factors provide useful information regarding nearby structured matrices [4]. In fact, in the Frobenius norm U is the nearest orthonormal matrix to A, while $\frac{1}{2}(A + H)$ is the nearest Hermitian positive semi-definite matrix to A. As a result, both U and H have numerous practical applications in scientific computing. The polar factor U is used in optimization schemes to orthonormalize the search space in optimization algorithms such as the gradient descent. The Hermitian factor H has direct applications in instances of principal component analysis (PCA) such as factor analysis or multidimensional scaling (MDS). Finally, as pointed out in [5], not only is the polar decomposition ubiquitous in the computation of matrix functions but it is also a powerful tool to perform standard matrix factorizations. In this report we give some details on how it can be used to derive a competitive SVD algorithm.

3.3 The `Polar-SVD` Algorithm

The SVD and the PD are equivalent and as a result one can easily be obtained from the other. In this report we are interested in obtaining an SVD from a polar decomposition. Let us consider an arbitrary matrix $A \in \mathbb{R}^{m\times n}$ and its polar decomposition (3). Then, the SVD (2) is obtained by first computing an eigenvalue decomposition of the Hermitian factor $H \in \mathbb{R}^{n\times n}$, i.e., factors $\Sigma \in \mathcal{D}^{n\times n}$ and $V \in \mathcal{O}^{n\times n}$ such that

$$H = V\Sigma V^*. \tag{4}$$

Thus, the singular values of A are equal to the eigenvalues of H, while the right singular vectors are equal to the eigenvectors of H. Finally, the left singular vectors of A are equal to $U = U_p V \in \mathcal{O}^{m\times n}$. Hence, as described in Algorithm 1, an SVD can be obtained via a PD followed by a symmetric eigensolve and a rectangular matrix-to-matrix multiplication. We call this generic algorithm `Polar-SVD` where the polar decomposition can be implemented using the algorithm described in Section 4, namely QDWH iterations. For a square matrix this represents an extra $6n^3$ flops after polar decomposition if vectors are required and $2n^3$ otherwise.

3.4 The QR trick

As mentioned in Section 2.1 it is often convenient for $m > n$ to perform a preliminary QR factorization. This trick obviously applies to the SVD or even the polar decomposition itself, since we can compute the polar decomposition of the R factor as

$$R = \bar{U}_p \bar{H}$$

and obtain the polar factors of A as $U_p = Q\bar{U}_p$ and $H = \bar{H}$. For the `Polar-SVD` it is cheaper to first compute the left singular vectors of R as $\bar{U} = \bar{U}_p \bar{V}$ and then apply the Q factor to $\bar{U}$ in order to obtain the left singular vectors of A, namely $U = Q\bar{U}$.

As explained in [12], in order to minimize the number of flops in the standard bidiagonal SVD we need $m \geq 2n$ to apply this trick, while we only need $m \geq 1.15n$ for the `QDWH-SVD`. This threshold may differ slightly if we want to minimize the computational time. However, it is still a good indicator for `QDWH-SVD` as all flops in `QDWH-PD` can be computed with a similar efficiency and we always solve the eigenvalue problem on a small n-by-n matrix. Additionally, because the threshold is so close to the square case ($m = n$) for the `QDWH-SVD`, we will mainly consider square matrices in our experiments. However, we will give the detailed flops count for the general case of rectangular m-by-n matrices.

There are extra benefits in applying polar decomposition algorithms to a square or even triangular input matrix. First, it reduces their relatively high memory footprint as more matrices can be recycled and packed format can be used. Second, it can help improve the accuracy and speed of the parameter estimation required in most polar decomposition algorithms (see Section 4.3).

Algorithm 1 Polar Decomposition-based SVD (`Polar-SVD`).

Input: An m-by-n complex-valued matrix A and a boolean `jobv` to indicate if vectors should be computed.
Output: Left and right singular vectors (U, V) and singular values Σ.

```
1: function POLAR–SVD(A, jobv)
2:     [U_p, H] = polar(A)              ▷ polar=newton,qdwh,zolo,...
3:     if jobv then
4:         [V, Σ] = syev(H, jobv)       ▷ 4n^3 flops
5:         U = U_p V                    ▷ 2mn^2 flops
6:         return U, Σ, V
7:     else
8:         [ , Σ] = syev(H, jobv)       ▷ 2n^3 flops
9:         return Σ
10:    end if
11: end function
```

4 Optimized Polar Decomposition algorithms

In this section we are interested in ways to efficiently compute a polar decomposition. We start by describing the main concepts behind the most commonly used algorithms then present an efficient variant designed to perform well at large scale, namely QDWH iterations (see Algorithm 2 in Section 4.2).

4.1 Computing the Polar Factor U_p

Algorithms for computing the polar decomposition can take many forms but they all rely on first computing the polar factor U_p using appropriate matrix iterations. Once an approximate polar factor is computed, the Hermitian factor H is simply obtained using a matrix to matrix multiplication, namely

$$H = U_p^* A.$$

Since accumulation of rounding errors can result in H not being symmetric, we can subsequently extract the symmetric part, namely $H := \frac{1}{2}(H^* + H)$.

Matrix iterations Since we know that the singular values of U_p are all equal to 1, any matrix iteration

$$X_{k+1} = f(X_k), \text{ with } X_0 = A$$

that maps the singular values of A to 1 while preserving the singular vectors is admissible. Applying Newton iterations to $A^*A = I$ is a simple way to do this, while Newton-Schulz would be its inverse–free counterpart. The main idea here is that we can reformulate matrix iterations for the polar decomposition as an optimization problem, where we seek to minimize the distance $\delta_k = |1 - \ell(X_k)|$ and

$$\ell_k = \ell(X_k) = \min_{i \leq n} \sigma_i(X_k)$$

denotes the minimum singular value of X_k. This criterion is very convenient as it provides an upper bound on the distance of U_p to orthogonality and ultimately some precious information on the backward stability of the algorithms [11].

Rational approximation of the sign function The relation between the polar factor and the matrix sign function is explained extensively in [5, Chap. 8]. In particular, it is shown in [5, Theo. 8.13] that for any matrix function g of the form $g(X) = Xh(X^2)$ such that $g(X)^* = g(X^*)$ and the matrix iteration $X_{k+1} = g(X_{k+1})$ converges to $\text{sign}(X_0)$ then

$$Y_{k+1} = Y_k h(Y_k^* Y_k), \quad Y_0 = A$$

converges to U_p. Once this relation is understood it is natural to prefer Padé iterations to compute the polar decomposition. In fact, a common backward stable method is scaled Newton iterations, the quadratically convergent iterations of the Padé family. Given the relatively high number of iterations required to reach convergence that approach can be computationally intractable at large scale. In this report, we will only consider the computationally efficient and scalable QDWH iterations [9].

4.2 Optimized Halley's iterations: the `QDWH-PD` algorithm

In order to accelerate the convergence of the polar decomposition, Halley's iterations (3rd order Padé iterations) can be considered, i.e.,

$$X_{k+1} = X_k(aI + bX_k^* X_k)(I + cX_k^* X_k)^{-1}, \quad X_0 = A/\alpha$$

where $(a, b, c) = (3, 1, 3)$ denote static weights and $\alpha > 0$ an approximate upper bound for $\|A\|_2 = \max_i \sigma_i(A)$. Thus, the singular values of X_0 are iteratively mapped from $[\ell_0, 1]$ to 1 using the recursive relation $\ell_{k+1} = h(\ell_k)$, where the rational function h reads as

$$h(x) = xg(x^2), \quad g(x) = \frac{a + bx}{1 + cx}.$$

Choosing dynamic weights (a_k, b_k, c_k) such as

$$a_k = h(\ell_k), \quad b_k = g(a_k), \quad c_k = a_k + b_k - 1.$$

gives a best rational approximation of the sign function and thus ensures optimal convergence of the scheme [9]. As a result these Dynamically Weighted Halley's iterations (DWH) can reach double precision accuracy in a maximum of 6 iterations, assuming the 2–norm condition number of A is lower than 10^{16} and parameters α and $\beta = \alpha\ell_0$ are relatively accurate estimations of the maximum and minimum singular values of A (see Section 4.3 for more details). Finally, the inverse matrix can be reformulated and implemented using only QR decompositions (see line 9–11 of Algorithm 2). Hence we obtain the following modified matrix iteration

$$X_{k+1} = (b_k/c_k)X_k + (1/c_k)(a_k - b_k/c_k)Q_1 Q_2^*, \tag{5}$$

where Q_1 and Q_2 are computed by mean of a $(m + n) - by - n$ QR decomposition such that

$$\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R = \left(\begin{bmatrix} \sqrt{c_k} X_k \\ I_n \end{bmatrix} \right). \tag{6}$$

The resulting method is called QR-based Dynamically Weighted Halley's iterations or QDWH iterations.

Optimizing the QDWH iterations Two crucial optimizations have been proposed in [12, Sec. 5.6] for the QDWH iterations. First, because the main computational bottleneck of the method lies in the computation of the $(m+n)$-by-n QR factorization, it is important to optimize it by first exploiting the identity structure of the last n-by-n block. By doing so we reduce the cost of dense QR iterations from $8mn^2 + \frac{2}{3}n^3$ flops to $7mn^2$ (See [7] for details of implementation). Second, QR iterations (5) can be reformulated in terms of the following Cholesky iterations (PO)

$$Z = I_n + c_k X_k^* X_k, \quad W = \texttt{chol}(Z)$$

$$X_{k+1} = (a_k/b_k)X_k + (a_k - b_k/c_k)\left(X_k W^{-1}\right) W^{-*}.$$

As shown in [12], switching from QR to PO iterations for c_k lower than 100 preserves the stability of the algorithm. In practice, c_k becomes lower than 100 after only 2 QR iterations for matrices with condition number up to 10^{16}. Additionally, for $K_2(A) < 10^6$ a maximum of 1 QR iteration is required and for $K_2(A) < 100$ only PO iterations are required. PO iterations require $4mn^2 + n^3/3$ flops each, which is half the flops required by a dense QR iteration (dense $(m + n)$-by-n QR factorization). Both types of iterations reach performance that are fairly close to the peak of the machine, hence we expect PO iterations to be about twice as fast as QR iterations.

The optimized QDWH–based polar decomposition `QDWH-PD` is described in Algorithm 2. The `QDWH-SVD` is obtained by combining `Polar-SVD` (Algorithm 1) with `QDWH-PD`.

Algorithm 2 Optimized QDWH based Polar Decomposition (`QDWH-PD`).

Input: An m-by-n complex-valued matrix A with $m > n$, an estimate for (α, β), a tolerance ε for the stopping criterion and a boolean `jobh` to indicate if H should be computed.
Output: A Polar factor U and an Hermitian factor H if `jobh` is true.

1: **function** QDWH($A, \alpha, \beta, \varepsilon,$ `jobh`)
2: $X_0 = A/\alpha$, $\ell_0 = \beta/\alpha$
3: $k = 0$
4: **while** $\|X_k - X_{k-1}\|_F \leq u^{1/3}\|X_k\|_F$ **do**
5: $a_k = h(\ell_k)$, $b_k = g(a_k)$, $c_k = a_k + b_k - 1$
6: //
7: **if** $c_k \geq 100$ **then** ▷ QR-based it.: $\mathbf{8mn^2 + \frac{2}{3}n^3}$ or $\mathbf{7mn^2}$
8:
9: $\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R = \texttt{qr}\left(\begin{bmatrix} \sqrt{c_k} X_k \\ I_n \end{bmatrix}\right)$
10:
11: $X_{k+1} = (b_k/c_k)X_k + (1/c_k)(a_k - b_k/c_k)Q_1Q_2^*$
12:
13: **else** ▷ PO-based it.: $\mathbf{4mn^2 + n^3/3}$
14:
15: $Z = I_n + c_k X_k^* X_k$
16:
17: $W = \texttt{chol}(Z)$
18:
19: $X_{k+1} = (a_k/b_k)X_k + (a_k - b_k/c_k)\left(X_k W^{-1}\right) W^{-*}$
20:
21: **end if**
22: //
23: $\ell_{k+1} = \ell_k(a_k + b_k\ell_k^2)/(1 + c_k\ell_k^2)$
24: $k = k + 1$
25: **end while**
26: $U = X_{k+1}$
27: **if** `jobh` **then**
28: $H = U_p^* A$
29: $H = \frac{1}{2}(H^* + H)$
30: **return** U, H
31: **else**
32: **return** U
33: **end if**
34: **end function**

4.3 Estimating $K_2(A)$ to ensure optimal convergence

In order to ensure fast convergence, it is crucial to properly initialize the iterations with accurate estimation of parameter α and $\beta = \alpha \ell_0$. First, scaling A by $\|A\|_2$ (or at least a tight upper bound) in the definition of X_0 ensures that the singular values of X_0 are in the interval $[\ell_0, 1]$, where $\ell_0 = \min_i \sigma_i(X_0)$. Second, providing an estimate for ℓ_0 (or at least a tight lower bound) will allow us to estimate the number of iterations required to reach a given convergence before actually starting the iterations.

Let us focus on the case where A is a square matrix. For the reasons mentioned in Section 3.4, if A is rectangular we will use a preliminary QR decomposition. As a result the parameters estimation will be done on the triangular matrix R rather than A, which offers more alternatives for the computation of α and β.

The 2–norm of A can be approximated efficiently using power iterations; see the `normest` function in MATLAB. Setting the maximum number of iterations to 100 proved very reliable in all cases of interest. Parameter estimation represents only a small fraction of the computational cost but its outcome dramatically affects the overall cost of the method. In particular, overestimating ℓ_0 can slow down convergence [9].

For ℓ_0 the estimator suggested in [12] reads as

$$0.9 \times \frac{1}{\texttt{condest}(X_0)}$$

where $\texttt{condest}(X_0)$ is an approximation of $K_1(X_0) = \|X_0\|_1 \|X_0^{-1}\|_1$. Alternatively, $K_1(X_0)$ can be bounded using the 1–norm of X_0 and X_0^{-1}, and the relation $\|X_0^{-1}\|_1 \geq \|X_0^{-1}\|_2/\sqrt{n} = (\sqrt{n}\ell_0)^{-1}$. Hence, the estimate reads as

$$\ell_0 \geq \frac{1}{\sqrt{n}\|X_0^{-1}\|_1}. \tag{7}$$

This bound may be satisfactory for small matrices but as n grows we are more likely to underestimate ℓ_0. In [12] the authors made it clear that underestimating ℓ_0 is somehow harmless to the convergence and stability, however it can still incur one more iteration. In a high performance implementation given the cost of each iteration it seems crucial to get optimal convergence of the iterations. Finally, a similar bound is considered in [14] without much justifications, namely

$$\ell_0 \geq \frac{\alpha}{1.1} \times \frac{1}{\|X_0\|_1 \|X_0^{-1}\|_1} \tag{8}$$

In [14] the authors suggest to use the triangular factor of the QR factorization of X_0 to compute $\|X_0^{-1}\|_1$. Not only is that inverse-free approach more efficient, scalable and stable but it is also very convenient. In fact, the Q factor can be recycled for the first matrix iteration decreasing its overall cost from $7n^3$ to $6n^3$. The difference in flops count ($1n^3$) may seem low, but it corresponds to the most expensive flops involved in the most expensive iterations. However, in order to avoid relying on the assumption that $\|X_0^{-1}\|_1 \approx \|R^{-1}\|_1$, we will not consider (8) and derive our own estimate for ℓ_0.

In [9] the authors wonder why such crude estimate proved reliable and leave that matter to future investigations. First, we agree that (7) or (8) are reliable, however our experiments showed that they do not always minimize the number of iterations to reach convergence. Second, we feel that there is no need for such crude estimate. In fact, if we

Table 2: Estimated number of iterations ($\#QR + \#PO$) if ℓ_0 is computed as (7) (top) or (9) (bottom). We use matrices from Section 6.1 with $m = n = 4,000$. The difference between the actual and estimated number of iterations is displayed in parenthesis. The rows labeled *error* and *ortho* denote the Frobenius norms of the error in the factorization $A - U_pH$ (normalized by the Frobenius norm of A) and the distance of U_p to orthogonality $\|I - U_pU_p^*\|_\infty$ (normalized by $\|I\|_\infty = n$).

	$K_2(A)$	10^0	10^4	10^8	10^{12}	10^{16}
	#it	$1+3$ (-)	$2+3$ (-)	$2+4$ (+1)	$2+4$ (-)	$3+3$ (-)
(7)	ℓ_0	1.581e-02	3.609e-08	3.143e-12	3.143e-16	1.767e-20
	ortho	6.219e-40	1.110e-19	1.110e-19	1.110e-19	1.110e-19
	error	1.688e-31	4.649e-16	3.434e-16	5.201e-16	4.702e-16
	#it	$0+1$ (-)	$1+4$ (+1)	$2+3$ (-)	$2+4$ (+1)	$2+4$ (-)
(9)	ℓ_0	1.000e+00	1.002e-04	1.002e-08	1.002e-12	5.636e-17
	ortho	1.110e-19	1.110e-19	1.110e-19	1.110e-19	1.110e-19
	error	9.182e-17	3.953e-16	4.326e-16	3.535e-16	5.826e-16

are able to build X_0^{-1} then we can re-use our 2–norm estimator and consequently obtain an approximate $\|X_0^{-1}\|_2 = K_2(A)$. Hence, our estimate simply reads as

$$\ell_0 = 1/\|X_0^{-1}\|_2 \approx 1/\texttt{normest}(X_0^{-1}). \tag{9}$$

Following the suggestion of [14] we can apply the 2–norm estimator to R^{-1} resulting in an accurate estimate of $\|X_0^{-1}\|_2$. If $m > n$ then estimating $\|R^{-1}\|_2$ is cheaper than estimating $\|A\|_2$. Our approach seems safer as we have a way to verify the accuracy of power iterations. Not only did our method prove accurate but often resulted in less iterations than other estimates, see Table 2. More precisely, the distance of U_p to orthogonality remains fairly constant with respect to the condition number and the choice of ℓ_0. Additionally, for a similar number of iterations both estimates result in similar errors in the factorization. As expected, the higher the number of iterations the more accurate the factorization. In particular, the very small errors reported in the first column of the upper part of the table are due to an overestimation of the condition number and consequently an excessive number of iterations.

4.4 Complexity

Here we summarize the number of flops involved in `QDWH-PD` and `QDWH-SVD` with all the aforementioned optimizations.

Polar decomposition The number of flops needed by each QR iteration is $8mn^2{+}2/3n^3$ or $7mn^2$ if we exploit the identity structure. We can save another mn^2 if we recycle the Q factor from the parameter estimation. The number of flops needed by each PO iteration is $4mn^2 + 1/3n^3$. Given the number of iterations ($\#QR$ and #PO) required to reach convergence of the QDWH iterations that were established in the previous section we can deduce the total number of flops in Algorithm 2 by the following relation

$$\#flops(\texttt{QDWH}) = \#QR \times \left(7mn^2\right) + \#PO \times \left(4mn^2 + 1/3n^3\right) + 2mn^2$$

For well-conditioned matrices only 1 PO iteration is required, therefore the minimum number of flop required by QDWH is $6mn^2 + 1/3n^3$. For ill-conditioned matrices a maximum of 2 QR and 4 PO iterations are required, which translates to $(14+16+2)mn^2 + 4/4n^3 = 33mn^2 + 1/3n^3$ flops.

Polar-SVD The number of flops required by a symmetric eigensolver and a matrix multiplication are roughly $2mn^2 + 4n^3$. If we add it to the previous flop count we obtain the total number of flops required in `QDWH-SVD`, namely

$$\#flops(\texttt{QDWH}-\texttt{SVD}) = \#QR \times \left(7mn^2\right) + \#PO \times \left(4mn^2 + 1/3n^3\right) + 4mn^2 + 4n^3.$$

Consequently, the `QDWH-SVD` requires between $10mn^2+(4+1/3)n^3$ and $37mn^2+(4+1/3)n^3$. For square matrices ($m = n$), this translates to about $14n^3$ and $41n^3$, which is about 2 to 6 times the number of flops required by standard SVD. Finally, we noticed that `QDWH-PD` converge in 1 iteration for matrices A with $K_2(A) = 1$, whereas other relatively well–conditioned matrices may require much more iterations. In particular, for $K_2(A) = 100$ it takes 1 QR and 3 PO iterations to reach convergence in double precision. Hence, it seems honest to consider the latter case as a more realistic lower bound for the flops count as well as for the experimental time to solution.

5 Implementation in Shared and Distributed Memory

In this section we describe our implementation of the `QDWH-PD` and `QDWH-SVD` in a linear algebra library for multi- and many–core architectures (Section 5.1), and in libraries for distributed memory computing and accelerators (Section 5.2). We focus on describing how we calibrate each library to get best performance of the algorithms and some numerical limitations.

5.1 `QDWH-SVD` for Multi– and Many–core Architectures

A first step to assess the performance and accuracy is to consider computations on a single node. Therefore, the first task was for us to implement `QDWH-PD` and `QDWH-SVD` in a linear algebra library for multi– and many–core architectures. The PLASMA library gives an optimal framework for this type of algorithm as it provides optimized implementations of matrix multiplication (`gemm`), QR (`geqrf`) and Cholesky (`potrf`) decomposition, using bulk asynchronous task–based parallelism.

From QUARK to OpenMP Starting from an implementation of the `QDWH-PD` provided by the authors of [16], that was based on the relatively old PLASMA-2.8[2], we provided an updated implementation in the latest version of the library, namely PLASMA-18.11[3]. The latter version improves its portability by switching from the QUARK runtime system to OpenMP task features [17]. Therefore, our main technical task was to convert `QDWH-PD` from the QUARK runtime system to OpenMP. A similar effort has been previously done for the 2–stage SVD and symmetric eigensolver.

[2]Can be downloaded from `bitbucket.org/icl/plasma/downloads/plasma-2.8.tar.gz`
[3]Can be downloaded from `bitbucket.org/icl/plasma`

PLASMA–QDWH Additionally, and maybe most importantly we needed to define a prototype for the polar decomposition but also a prototype for `Polar-SVD` that would be consistent with that of the existing SVD routines. We provide our prototype software as a fork of the latest version of PLASMA, called PLASMA-QDWH[4]. The fork contains a set of custom testers in the `drivers/` directory as well as standard testers in the `tests/` directory. Informations on how to run custom tests are provided in `drivers/README`. The names used for the routines and testers for `QDWH-PD` and `QDWH-SVD` are `xgepd_qdwh` and `xgesvd_qdwh` respectively, where the prefix `x` denotes the datatype and the suffixes stand for polar and singular value decomposition of a general matrix based on QDWH iterations.

5.2 `QDWH-SVD` in Distributed Memory

Here we describe our implementation of `QDWH-PD` and `QDWH-SVD` relying on libraries for numerical linear algebra in distributed memory.

ScaLAPACK We wish to verify the scalability of the `QDWH-PD` and `QDWH-SVD` algorithms and compare it to that of the standard approach to the SVD. Therefore, we provide a pure CPU implementation in distributed memory using the state-of-the-art library ScaLAPACK. The library is used in pure MPI mode, i.e., we use 1 MPI process per core. Rather than using existing implementations that may involve extra dependencies, we choose to provide our own implementation with a single dependency, namely Intel MKL's ScaLAPACK — the same library we use to perform standard 2–stage SVD. On the other hand, KAUST Extreme Computing Research Center's (ECRC) implementation of scalable algorithms for polar decomposition and SVD[5] relies on ELPA library for the symmetric eigensolver, see [15] for more details. Furthermore, our version implements our own approach to parameters estimation (Section 4.3). In order to maximize the performance of ScaLAPACK, a library exploiting a 2D block cyclic data distribution, we use MPI process grids that are as square as possible. Finally, the optimal block size for the `gemm`, `geqrf` and `potrf` in ScaLAPACK is usually $b = 64$ (or slightly higher), therefore we use the same value for `QDWH-PD` and `QDWH-SVD` on all hardware architectures.

SLATE SLATE is a successor to ScaLAPACK enabling computations on accelerators. Alternatives to SLATE for distributed memory and heterogeneous architectures include MAGMA, Chameleon or DPlasma. The case of MAGMA (synchronous) and Chameleon (asynchronous tasks scheduling based on StarPU) is considered in [14]. DPlasma is an alternative to Chameleon, that relies on the ParSEC runtime system instead of StarPU. We choose SLATE for its portability, its simplicity and the many convenient features it provides such as templatization, ScaLAPACK–like data distribution and user–friendly interface with ScaLAPACK. SLATE does not provide a symmetric eigensolver yet, however we were able to assemble a partially accelerated `QDWH-SVD` using the built-in ScaLAPACK interface. As shown by our preliminary experiment in Appendix 8.2, SLATE QR decomposition is working relatively well and reaches performance close to that of Cholesky decomposition. However, it is still under development and is not yet able to exploit the identity structure in (6).

Two different modes can be considered in SLATE:

[4]Can be downloaded from `bitbucket.org/piblanch/plasma-qdwh`
[5]Can be downloaded from `github.com/ecrc/ksvd`

- **1 proc per core (1ppc):** similar to ScaLAPACK, each MPI processus is mapped to a core.
- **1 proc per node (1ppn):** This mode corresponds to a configuration where the communication between nodes are handled by MPI, while communications within a node are handled by OpenMP.

Comparative results for the 3 building blocks of `QDWH-PD`, namely `geqrf`, `potrf` and `gemm`, are provided in Appendix 8.2 using both modes. ScaLAPACK can in theory be used in 1ppn mode, for instance by enabling multithreading in MKL's BLAS and LAPACK. However, it usually gives worse performance than 1ppc. In our case, using 1ppn provided better performance, therefore we decided to show the results in 1ppn mode. Optimal block sizes were determined empirically on recent Intel CPUs (see Appendix 8.2), we found that a good compromise for the polar decomposition is to use a block size of $b = 256$ on either Broadwell or Skylake nodes.

SLATE-POLAR Our prototype software for the `QDWH-PD` and `QDWH-SVD` in distributed memory and on accelerators is available as a fork of the SLATE library called SLATE-POLAR[6]. It implements both a ScaLAPACK only and a SLATE+ScaLAPACK version of the algorithms. The routines and testers for `QDWH-PD` and `QDWH-SVD` are denoted `gepd_qdwh` and `gesvd_qdwh` respectively.

5.3 `QDWH-PD` on Accelerators

Thanks to templatization our implementation of the polar decomposition in SLATE can be executed via the same source code on either CPUs and/or GPUs and in single or double precision. It is up to the user to provide a target type (`device` for GPU, `host` for CPU) and a data type (`s` for real single or `d` for real double) at run time. Additionally, computations on accelerators can only be done in the 1ppn mode, therefore all results associated with SLATE will involve 1 MPI process per node. Optimal block sizes were determined empirically on NVIDIA GPUs. We found that a good compromise for the polar decomposition is to use a block size of $b = 512$. Finally, because the `QDWH-SVD` cannot yet be fully accelerated we will only present numerical results on `QDWH-PD` with SLATE (see Section 6.4).

6 Numerical Experiments

In this section we describe a way to benchmark the polar decomposition based SVD in shared and distributed memory environments, then we provide some numerical results and discuss the relative performance of the `QDWH-SVD` and the 2–stage bidiagonal SVD. Here we strictly focus on performance for double precision real numbers, however Appendix 8.1 provides some important remarks on precision and accuracy.

6.1 Benchmarking the Polar–SVD

Test matrices Let us recall that for the sake of clarity we only consider real double precision square matrices. Since the number of iterations required by the polar decomposition algorithms depends on the conditioning of the matrix A we need to have control

[6] Can be downloaded from `bitbucket.org/piblanch/slate-polar`

over the value of $K_2(A)$. Thus, we generate input matrices with the `latms` function of LAPACK (equivalent to `gallery('randsvd')` in MATLAB) as it allows one to prescribe the singular values explicitly. We enforced singular values that decrease arithmetically ($mode = 4$) from 1 to $1/K_2(A)$. We consider at least 2 extreme cases, namely $K_2(A) = 1$ and $K_2(A) = 1/u = 10^{16}$. As discussed in 4.4, the intermediate case $K_2(A) = 100$ should be considered as a more realistic lower bound than $K_2(A) = 1$ for the experimental time to solution.

Finally, because it requires about 5 dense n-by-n matrices, the memory footprint of the `QDWH-PD` is relatively high and very quickly exceeds the size of the low-bandwidth memory on KNLs (16GB) or even the memory available on a GPU (12GB to 24GB in NVIDIA K80). Therefore we will only consider matrix sizes below $m = n = 20,000$. Moreover, generating larger matrices with `latms` often requires several times the time required to factorize them.

Hardware Architectures In this report we are first concerned with modern multi–core and many–core architectures. Our experiments will be performed on 2-socket (NUMA islands) nodes equipped with relatively recent Intel CPUs. We will use the CPUs listed below and sorted by increasing throughput (GFlops/s):

- Haswell nodes[1] are composed of 2x Intel Xeon E5-2650 v3 @ 2.30GHz.
  20 cores, 64GB RAM. Available Instructions: AVX2.
  Theoretical peak near 736 GFlops/s (double precision).

- Broadwell nodes [2] are composed of 2x Intel Xeon E5-2690v4 @ 2.60GHz.
  28 cores, 128GB RAM. Available Instructions: AVX2 & FMA3.
  Theoretical peak near 1160 GFlops/s (double precision).

- Skylake nodes[2] are composed of 2x Intel Xeon Gold 6132 @ 2.60GHz.
  28 cores, 192GB RAM. Available Instructions: SSE4.2, AVX, AVX2, AVX-512.
  Theoretical peak near 2330 GFlops/s (double precision).

- KNL nodes[2] are composed of 1x Xeon Phi 7250 @ 1.40GHz (only one socket)
  68 cores, 192GB RAM + 16 GB MCDRAM. Instructions: AVX-512.
  Theoretical peak near 3050 GFlops/s (double precision).

Note that the difference in flop rates between Haswell and Broadwell or Skylake and KNL is mainly due to the respective numbers of cores, while the type of vectorization (length of SIMD instructions) explains the difference between Broadwell and Skylake. Of course, the memory hierarchy plays a crucial role too, especially on KNLs and accelerators.

Tuning Parameters We also chose to study the square case because it is relatively easy to set up in terms of optimal tile size and MPI grid size. Hence, it is easier to see how `Polar-SVD` can be competitive with state–of–the–art approaches. The cost of the `Polar-SVD` algorithm studied in this report is governed by QR decomposition (`geqrf`), Cholesky decomposition (`potrf`), and matrix multiplication (`gemm`). Therefore, the tuning parameters should be picked to provide optimal performance of those kernels with stronger emphasis on QR, as it dominates the overall cost.

[1]Saturn: ICL's experimental platform located at the University of Tennessee in Knoxville, TN.
[2]Kebnekaise: Swedish national supercomputer located at the University of Umeå, Sweden.

In PLASMA-18.1, the Cholesky factorization typically gives best performance around 550 to 600 GFlops/s on Haswell with a tile size of $b = 224$ (see [17]). Similar to matrix multiplication, factorization in QR form delivers best performance for slightly larger b, namely $b = 336$ on Haswell and Broadwell. We use a standard setup for the value of the inner block size in the QR routine, namely $i = b/4$, and we prefer the Householder variant over the one based on a flat tree. The optimal tile size increases slightly for `QDWH-SVD` as the number of cores (and flop rate) of the hardware grows. Hence, for `QDWH-SVD`on Skylake we set $b = 384$ and on KNLs we set $b = 448$. For the `2-stage SVD` the optimal tile size remains relatively constant, namely around $b = 160$ for all architectures.

Reference implementations Our aim is to compare our prototype for the `Polar-SVD` to the standard approach, namely SVD based on bidiagonal reduction (with either the 1– or 2–stage variant). A state–of–the–art implementation is provided in the Intel Math Kernel Library (MKL). Because it is the most widely used in the scientific computing community it will serve as a reference for our discussions on the relative performance and accuracy. It is a fair comparison as MKL also serves as a BLAS and LAPACK core engine in our experiments with PLASMA. However, MKL is relatively opaque and it is not obvious what algorithm is actually implemented for the SVD and symmetric eigensolvers. While `2-stage SVD` seems to have been implemented in MKL since at least 2018, 2–stage eigensolvers were only added for certain variants in version 2018.2+ [7].

6.2 Experiments on a Single Node

QUARK vs OpenMP Figure 2 shows the relative time to solution and performance of the QUARK and OpenMP implementations with or without optimizations (i.e., exploiting identity structure and recycling). As expected no significant difference is observed between the 2 runtime systems in this range of sizes. Looking closer at the performance of the OpenMP version, we observe that we get experimental flop rates around 75% of the peak performance of a single Haswell node (including turbo boost). Optimizations do not change the nature of the flops or the number of iterations, hence they do not significantly impact the performance but rather decrease the time to solution (and the flop count) by a small factor. The speedup associated with optimization is relatively small but still much larger than that associated with the choice of runtime system (close to zero).

Bidiagonal–SVD vs QDWH–SVD Figure 3 (left) shows clearly that `QDWH-SVD` is much slower than standard approaches to compute the singular values only. On the other hand, Figure 3 (right) shows that `QDWH-SVD` is faster than the QR variant (`gesvd`) but slower than the D&C variant of the `2-stage SVD` (`gesdd`) in the ill–conditioned case. In the following we will always take 2–stage D&C SVD as the reference and only consider the computation of all singular values and vectors.

Since our algorithm is compute–bound, it suffices to use CPUs with larger flops capabilities to speed it up. In fact, Figure 4 shows that for $n = 14,000$ `QDWH-SVD` completes in 3 to 8 minutes on Haswell (left) against about 1 to 2 minutes on KNLs, which means a speedup of 4 in the ill –conditioned case. The latter provide a significant acceleration of the `QDWH-SVD` while the `2-stage SVD` (still memory bandwidth bound) is slightly less accelerated, namely we measured a speedup of 3 on KNL for the divide and conquer

[7]More information on symmetric eigensolvers in MKL at `https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-2018-update-2-scalapack-symmetric-eigensolver`

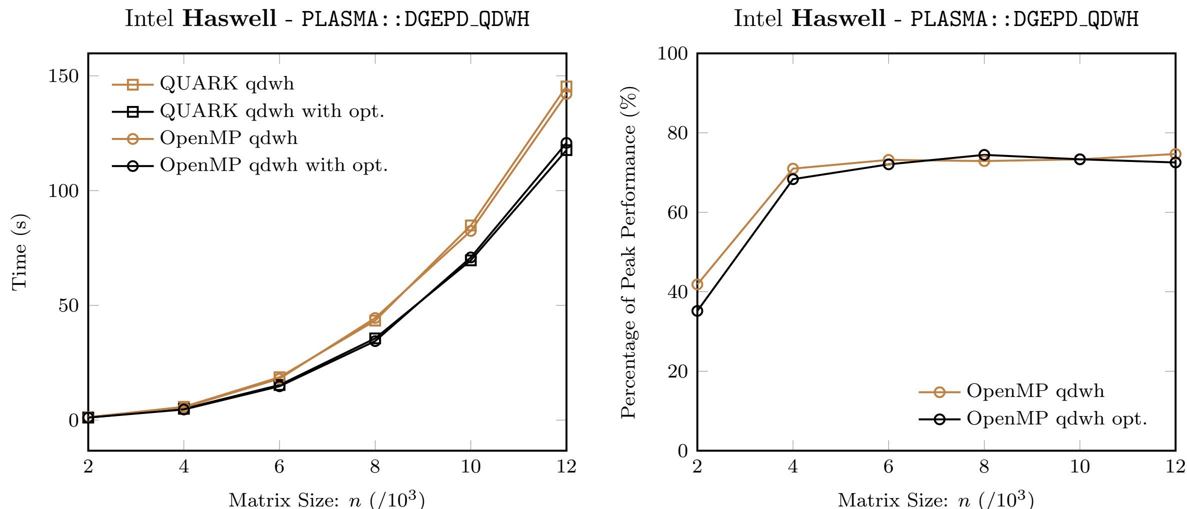

Figure 2: Time to PD (left) and percentage of the peak performance including turbo boost for `QDWH-PD` (right) for different task schedulers, namely QUARK (squares) and OpenMP (circles). The input matrix is ill–conditioned ($K_2(A) = 10^{16}$) and factorized using a dense `QDWH-PD` in brown or a fully optimized `QDWH-PD` (identity structure + recycling) in black.

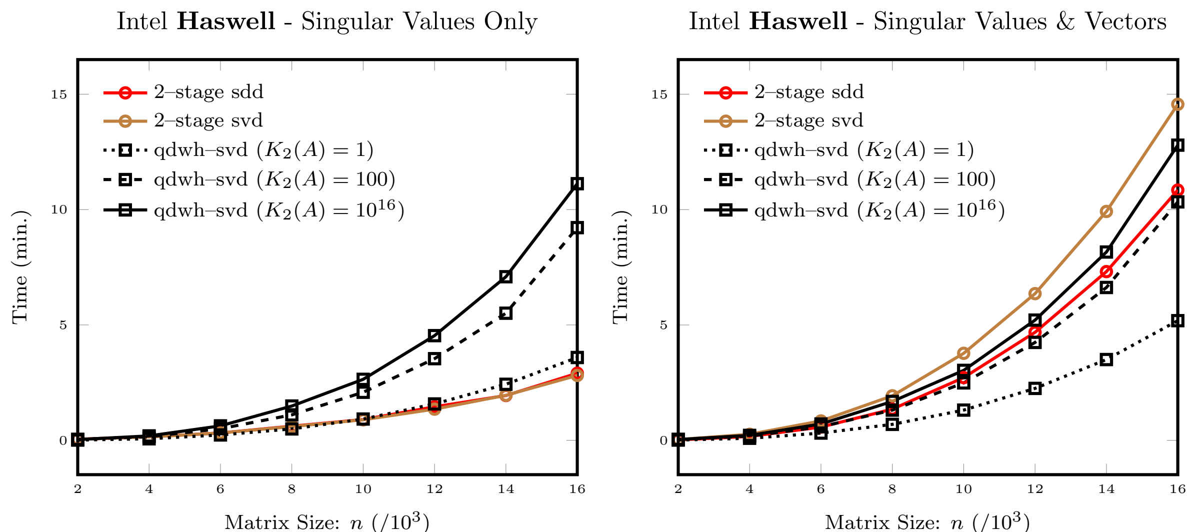

Figure 3: Time to compute the singular values only (left) or the singular values and vectors (right) using PLASMA's 2–stage bidiagonal SVD based on QR (red circles) or D&C (brown circles) or QDWH–SVD (black squares) for different values of the condition numbers $K_2(A)$: 1 (dotted lines), 100 (dashed lines) and 10^{16} (solid lines).

approach and a speedup of 2 for the QR approach. The larger speed up obtained for `QDWH-SVD` is mainly due to its much higher level of concurrency. As a result, on Intel KNLs the `QDWH-SVD` becomes slightly faster than `2-stage SVD` for ill–conditioned matrices and twice as fast for well–conditioned matrices. Finally, because of its relatively large memory footprint it may not be very convenient in general to compute `QDWH-SVD` on a single node. Moreover, if the memory footprint exceeds the size of the MCDRAM (high memory bandwidth) on Intel KNLs then we observe a large performance drop and `QDWH-SVD` becomes slower than `2-stage SVD`. Therefore, and also because in theory it must have better scalability, `QDWH-SVD` should be an even better alternative to 2–stage in distributed memory.

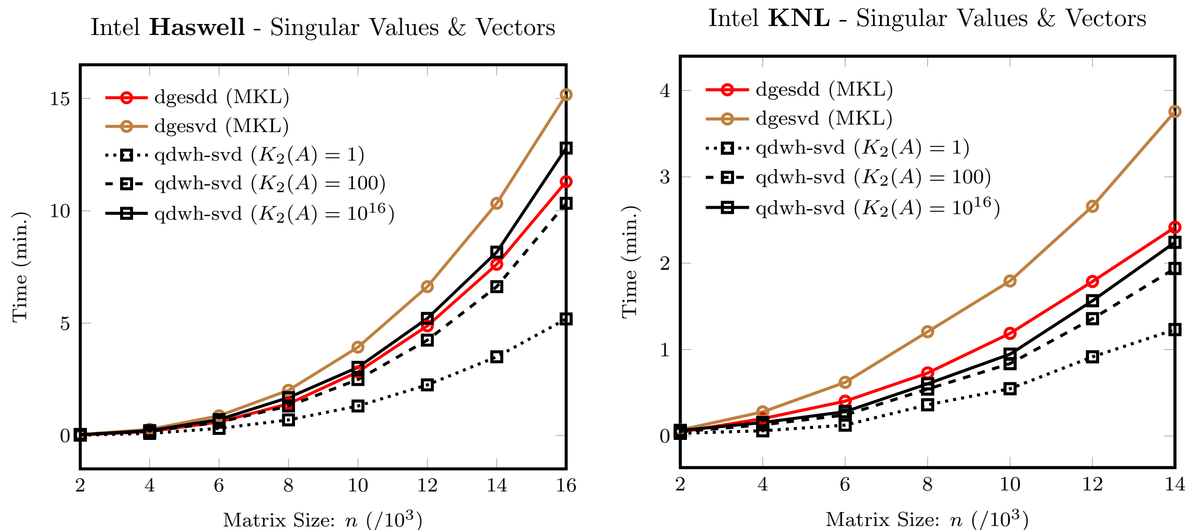

Figure 4: Time to compute SVD (min.) using MKL's 2–stage bidiagonal SVD based on D&C (red), based on QR (brown) or PLASMA's QDWH–SVD (black) for different values of the condition numbers $K_2(A)$: 1 (dotted lines), 100 (dashed lines) and 10^{16} (solid lines).

6.3 Experiments in Distributed Memory

We expect the `QDWH-SVD` to be even more competitive in distributed memory as it is very light in communications and in theory more scalable than `2-stage SVD`. We verify that assumption using our ScaLAPACK implementation.

Scalability Figure 5 shows the cumulative time (i.e., time multiplied by number of nodes) for `2-stage SVD` and `QDWH-SVD`. For a perfectly scalable algorithm and sufficiently large matrices, the cumulative time should be constant with respect to the number of nodes, i.e., for a given method all curves should coincide. First, the `2-stage SVD` clearly shows a lack of scalability that is characterized by the large gap between each red curve. Second, the `QDWH-SVD` shows much better scalability (smaller gaps) for both ill– and well–conditioned matrices. This is explained by the better scalability of `gemm`, `geqrf` and `potrf` compared to `2-stage SVD`.

Detailed timing Figure 6 shows the detailed timing associated with the `QDWH-SVD`. Top graphs illustrate the benefit of using a 2–stage eigensolver (MKL 2019, on the right) instead of the regular 1–stage (MKL 2018, on the left). The bottom graph shows the

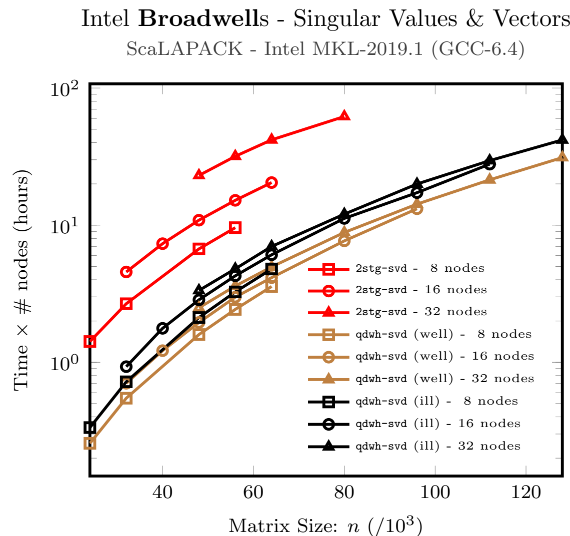

Figure 5: Time to SVD times the number of nodes (min.) using MKL's `2-stage SVD` (red) or PLASMA's `QDWH-SVD` for well– (brown) or ill–conditioned matrices (black). The smaller the gab between curves of a given color the higher the scalability of the underlying implementation/algorithm.

benefit of using a CPU with higher throughput, namely Skylake. The main acceleration is observed for the `QDWH-PD` (about 50% speedup compared to Broadwell). The `2-stage SVD` is also slightly faster on Skylake as it involves a significant part of level–3 BLAS operations. However, the final overall gain is still slightly better for Skylake than Broadwell. Hence, not only is `QDWH-SVD` more scalable but it is roughly 2 to 5 times faster than `2-stage SVD` on 8 nodes.

6.4 Experiments with Accelerators

We expect the speedup to be much larger when performing the polar decomposition on accelerators such as NVIDIA GPUs. We provide some preliminary results using SLATE in order to validate its performance.

Figure 7 shows the relative performance on a single CPU versus a single GPU as well as a single GPU versus multiple GPUs. The GPUs used in our experiments are NVIDIA V100 or K80, more precisely each node hosts 2 Broadwell CPUs plus 2 GPUs. Results show that acceleration is working effectively, however scalability is not yet fully ensured because some minor operations are still performed on CPUs and thus unnecessary data movement between CPUs and GPUs still occurs.

Those preliminary results are encouraging and provide a solid basis for upcoming implementations of the SVD in SLATE as well as other standard matrix factorizations powered by QDWH iterations.

7 Conclusions

We showed that algorithms for the polar decomposition can be straightforwardly implemented and used to speed–up the SVD of large matrices. The resulting `QDWH-SVD` uses computing resources with an efficiency close to that of matrix multiplication, namely about 75% of the peak performance on a single node. Consequently, it can run faster

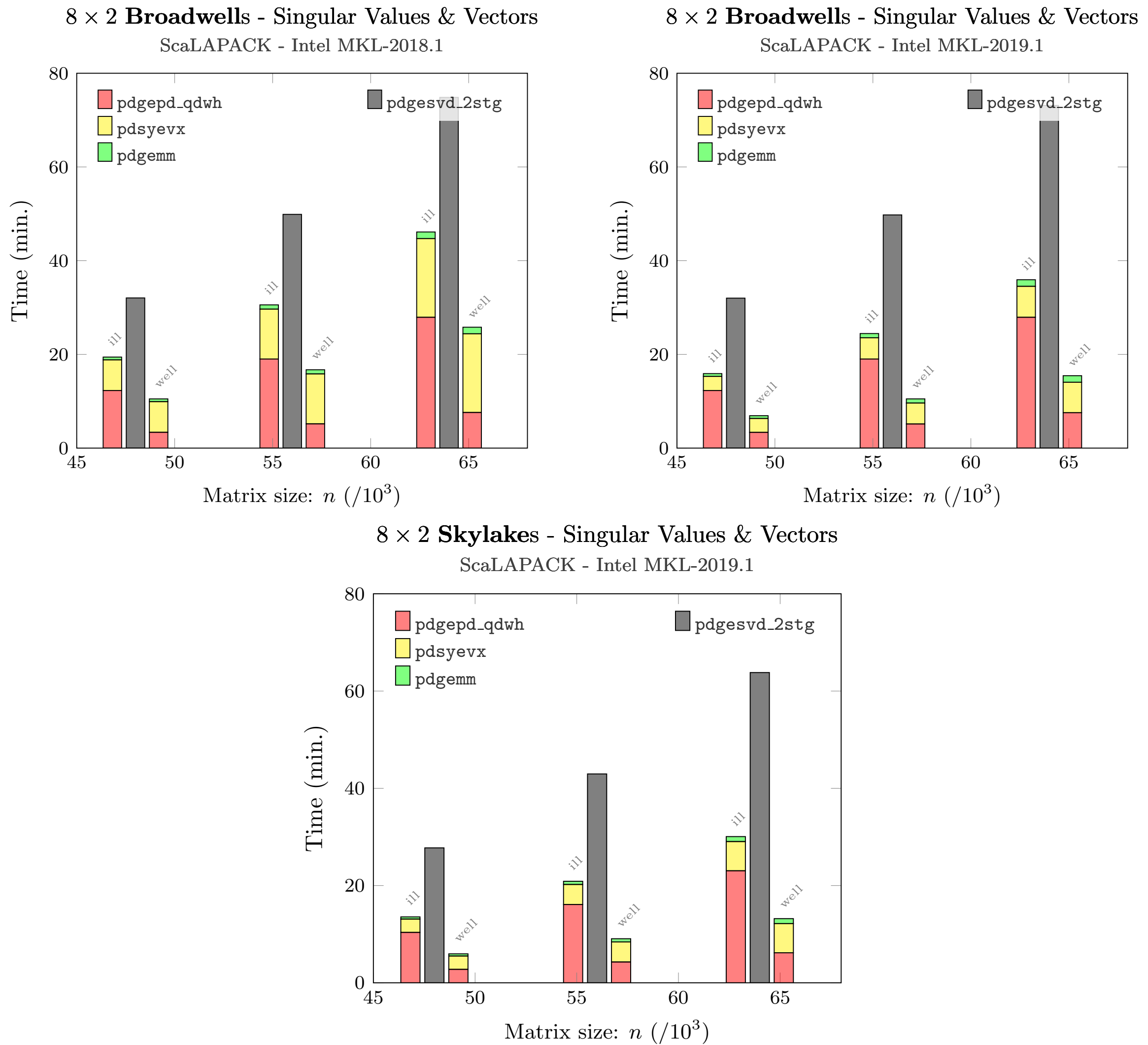

Figure 6: Time to SVD (min.) using ScaLAPACK's `2-stage SVD` (gray) or QDWH–SVD (colors) on Broadwell with MKL 2018 (top left), Broadwell with MKL 2019 (top right) and Skylake with MKL 2019 (bottom). We use 8 times 2–socket Broadwell nodes. The labels *ill* and *well* indicate that the associated matrices have condition number 10^{16} and 1 respectively.

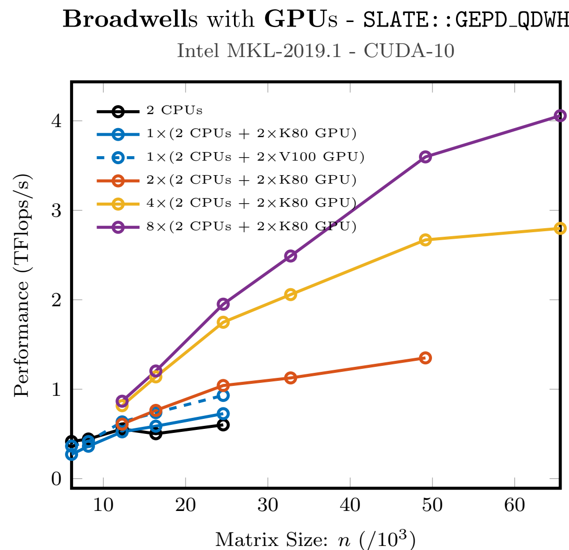

Figure 7: Performance (TFlops/s) of the GPU accelerated `QDWH-PD`. Each node as 2 Broadwell CPUs and 2 NVIDIA GPUs. The input matrix A has an intermediate condition number of $K_2(A) = 100$.

than the state–of–the–art SVD based on 2–stage bidiagonal reduction, even though it involves much more flops.

Redesigning the SVD algorithm to use matrix iterations and thus be predominantly compute–bound allowed us to fully benefit from recent CPU architectures that have augmented flop capacities. Hence, we showed that thanks to its very high level of concurrency the `QDWH-SVD` can fully benefit from the large flop rate of recent CPUs, co–processors and accelerators.

Additionally, numerical experiments in distributed memory showed that `QDWH-SVD` scales much better than 2–stage SVD. As a result, the measured speed–up was even more significant at large scale. Speed–up factors of 2 to 5 were observed for matrices of intermediate sizes and experiments running on 8 Broadwell nodes. Finally, experiments highlighted that `QDWH-SVD` can benefit greatly from a high performance symmetric eigensolver, like the one based on 2–stage tridiagonal reduction.

An interesting feature of this algorithm is that it runs more than 2 times faster than double precision when executed in single precision, see Appendix 8.1. The main reason for that is the slightly lower number of iterations required to get convergence. This feature should be exploited in future research to further increase the performance of Polar Decomposition based SVD algorithms.

Finally, there exists a much more involved version of the QDWH iterations having arbitrary high orders of convergence and known has Zolotarev iterations. This variant is very similar to the QDWH iterations in terms of algorithm and parallel performances but involves much more flops, namely about r times if $2r + 1$ is the order of convergence ($r = 8$ in practice). Fortunately, all extra flops are embarrassingly parallel. As a result, Zolotarev–based SVD can run up to 3 times faster than `QDWH-SVD` if enough computing resources are available. For more details on Zolotarev iterations please refer to [10, 7, 8]. This type of alternatives should be considered at very large scales if enough resources are available.

Acknowledgments

We thank the High Performance Computing Center North (HPC2N) at Umeå University, which is part of the Swedish National Infrastructure for Computing (SNIC), for providing computational resources and valuable support.

References

[1] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. The Singular Value Decomposition: Anatomy of optimizing an algorithm for extreme scale. *SIAM Review*, 60(4):808–865, 2018.

[2] Jack J. Dongarra, Danny C. Sorensen, and Sven J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215–227, 1989.

[3] Benedikt Großer and Bruno Lang. Efficient parallel reduction to bidiagonal form. *Parallel Computing*, 25(8):969–986, 1999.

[4] Nicholas J. Higham. Matrix nearness problems and applications. In M. J. C. Gover and S. Barnett, editors, *Applications of Matrix Theory*, pages 1–27. Oxford University Press, 1989.

[5] Nicholas J. Higham. *Functions of Matrices: Theory and Computation.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[6] Bruno Lang. Parallel reduction of banded matrices to bidiagonal form. *Parallel Computing*, 22(1):1–18, 1996.

[7] Shengguo Li, Jie Liu, and Yunfei Du. A new high performance and scalable SVD algorithm on distributed memory systems. *arXiv preprint*, 2018.

[8] Hatem Ltaief, Dalal Sukkari, Aniello Esposito, and David Keyes. Massively parallel polar decomposition on distributed-memory systems. *preprint*, 2018.

[9] Yuji Nakatsukasa, Zhaojun Bai, and François Gygi. Optimizing Halley's iteration for computing the matrix polar decomposition. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2700–2720, 2010.

[10] Yuji Nakatsukasa and Roland W. Freund. Computing fundamental matrix decompositions accurately via the matrix sign function in two iterations: The power of Zolotarev's functions. *SIAM Review*, 58(3):461–493, 2016.

[11] Yuji Nakatsukasa and Nicholas J. Higham. Backward stability of iterations for computing the polar decomposition. *SIAM J. Matrix Anal. Appl.*, 33(2):460–479, 2012.

[12] Yuji Nakatsukasa and Nicholas J. Higham. Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD. *SIAM J. Sci. Comput.*, 35(3):A1325–A1349, 2013.

[13] Takeshi Ogita and Kensuke Aishima. Iterative refinement for symmetric eigenvalue decomposition. *Japan Journal of Industrial and Applied Mathematics*, 35(3):1007–1035, nov 2018.

[14] Dalal Sukkari, Hatem Ltaief, Mathieu Faverge, and David Keyes. Asynchronous task-based polar decomposition on single node manycore architectures. *IEEE Trans. Parallel Distrib. Syst.*, 29(2):312–323, 2018.

[15] Dalal Sukkari, Hatem Ltaief, and David Keyes. High performance polar decomposition on distributed memory systems. In *Euro-Par*, pages 605–616, 2016.

[16] Dalal E Sukkari, Hatem Ltaief, and David E Keyes. A high performance QDWH-SVD solver using hardware accelerators. *ACM Trans. Math. Softw.*, 43(1):6:1–6:25, 2016.

[17] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the PLASMA numerical library to the OpenMP standard. *International Journal of Parallel Programming*, 45(3):612–633, 2017.

8 Appendix

8.1 Remarks on Accuracy and Precision

Accuracy Comparing the relative accuracy of QDWH–based SVD and standard algorithms for SVD can be tricky. However, we believe this is a fundamental aspect of novel SVD algorithms that needs to be addressed with a lot of care. In fact, in most cases both approaches yields similar accuracy, i.e., have the same order of magnitude and differ only by a small constant. However, divide and conquer approaches may not converge in case of clustered eigenvalues. For illustration of the latter phenomenon as well as a more detailed comparison of the accuracy of `QDWH-SVD` and standard approaches please refer to [16].

Figure 8 shows the error in factorization $\|A - U\Sigma V^T\|_F$ and the distance of U to orthogonality in infinity norm (results for V are similar) for the matrices studied in Section 6.2. The measured accuracy for the QR based approach is about 3 times worse than that of the divide and conquer approach. The distance to orthogonality for the `QDWH-SVD` lies somewhere inbetween (closer to the QR approach for large n), while the error in factorization almost coincides with that of the divide and conquer approach. On the other hand, due to its iterative nature the output accuracy of the PD— and consequently the SVD —can easily be adjusted. For instance, allowing one more matrix iterations can be relatively harmless to the cost of the overall algorithm, while providing a significant improvement of the accuracy, see Table 2.

A standard method suggested in [12] in order to improve the accuracy of the `QDWH-SVD` and `QDWH-PD` a posteriori and at a low cost is to use 1 or 2 Newton–Schultz iterations.

Finally, the choice of symmetric eigensolver does matter too. In particular, divide and conquer approaches may also encounter convergence issues in case of clustered eigenvalues.

Precision This report focuses on computations in double precision floating point arithmetic. Here we provide some results using single precision. It would be natural to consider 10^8 as the maximum condition number, however we want to run the algorithm on the same

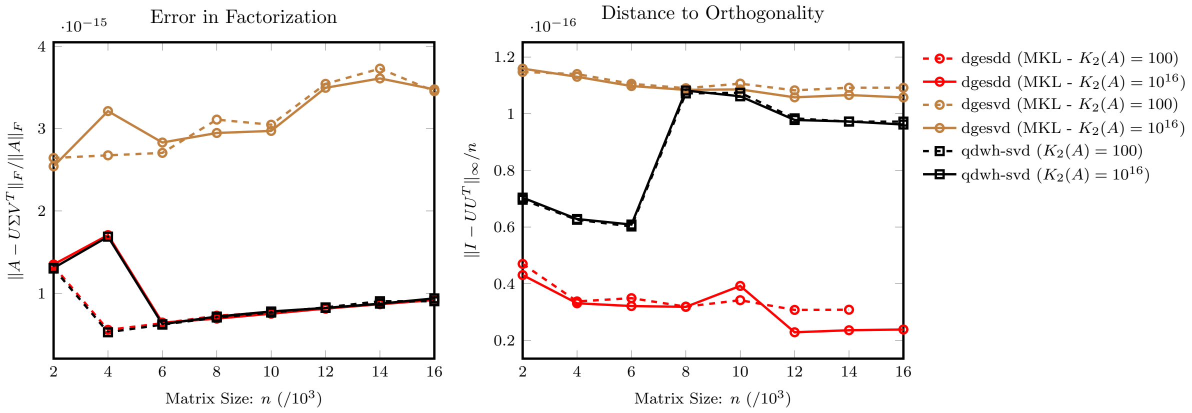

Figure 8: Accuracy of `QDWH-SVD` and 2–stage approaches with respect to the matrix size. The error in factorization (left) and the distance of U to orthogonality are represented.

matrices in order to compute a speedup between double and single precision. First, because our stopping criterion depends on the unit round–off it is possible that the number of iterations needed to reach convergence will be lower than in double precision. Second, the building blocks of `QDWH-PD` usually run slightly more than twice faster in single than in double precision. Figure 9 shows the experimental speedup on one two–socket Intel Haswell node for `QDWH-SVD` as well as for `2-stage SVD` based on QR and on divide and conquer. The QR approach fails to provide any speedup except for large matrices while all other approaches provide a speedup larger than 2. Moreover, `QDWH-SVD` almost always shows a larger speedup than divide and conquer. In particular, when the number of iterations increases ($K_2 \geq 100$), single precision is about 2.5 times faster than double precision in average.

As mentioned in our paragraph on accuracy, since it is sufficient to perform a few more iterations to improve the accuracy, matrix iterations inherently provide a straightforward way to improve the accuracy of the factorization if ever required. Additionally, recent papers [13] have derived cheap ways of refining eigenvalue decompositions and SVD using combinations of Newton–Schultz iterations and Davies–Modi's approach.

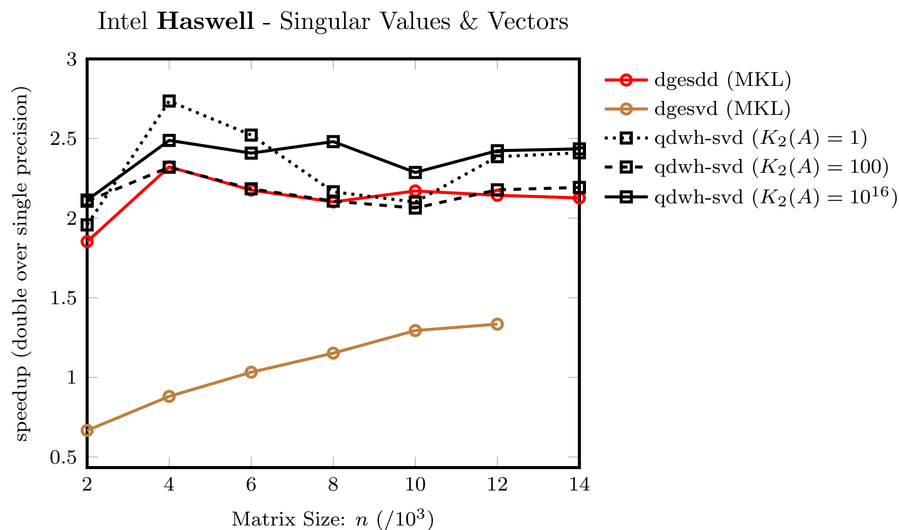

Figure 9: Ratio between the time to solution in double and single precision. A ratio higher than 2 means that single precision is more than twice faster than double precision.

8.2 Benchmarking SLATE

Figure 10 provides performance results for the matrix multiplication (`gemm`), QR decomposition (`geqrf`) and Cholesky decomposition (`potrf`) in SLATE using either the 1ppn mode (solid lines) or the 1ppc mode (dashed lines) for a fixed value of the tile size, namely $b = 256$. Hence, we do not expect to obtain best performance of each routine but rather a good compromise for the performance of `QDWH-PD`. Results show experimental flop rates around 75% of the theoretical peak performance for `gemm` and 50% for `geqrf` and `potrf`. It also shows relatively good scalability as long as saturation is reached. The performance are better in the 1ppn mode, except for the QR factorization that is not yet fully optimized. The performance of SLATE's `QDWH-PD` depends mainly on these results.

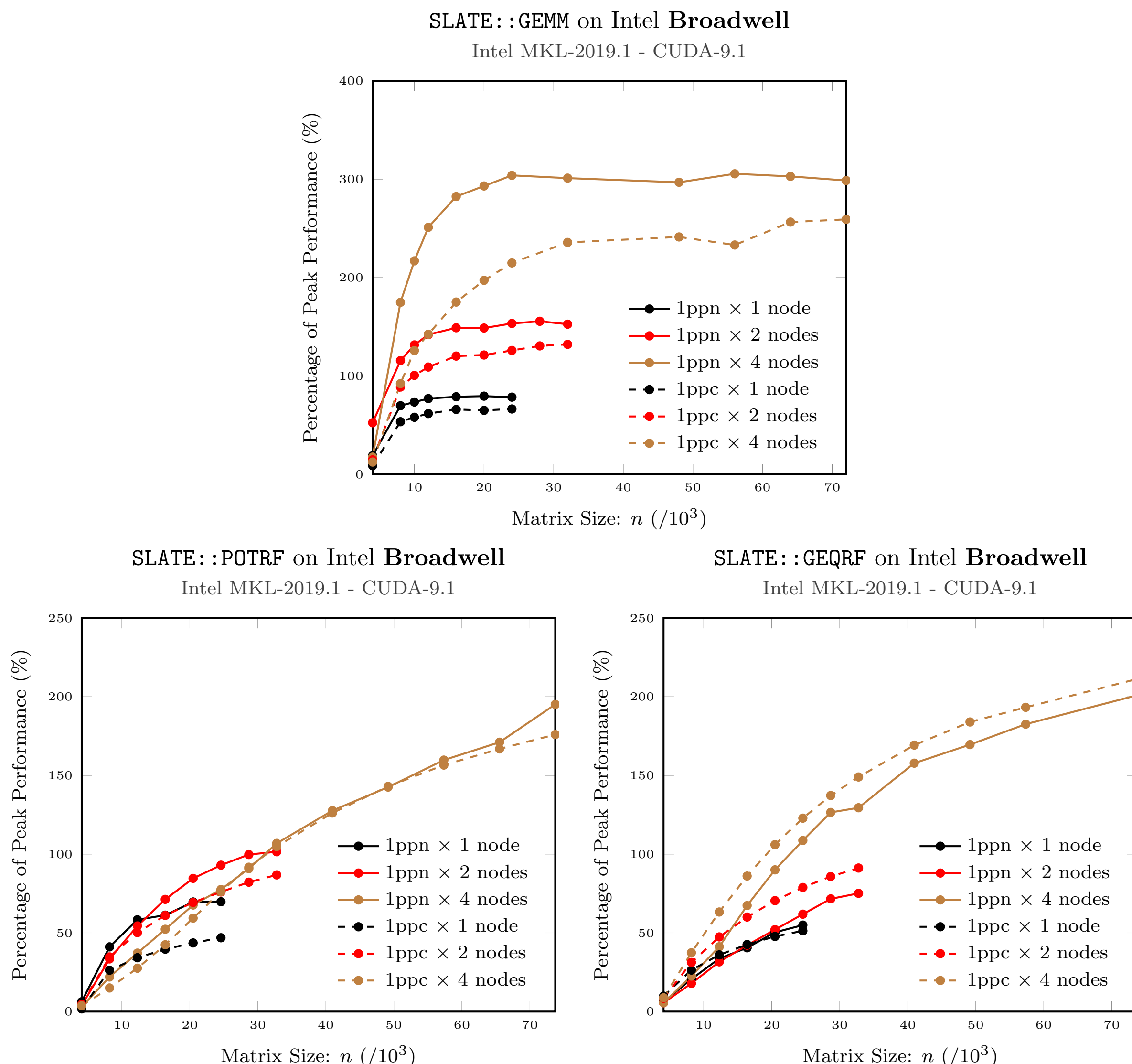

Figure 10: Percentage of the peak performance including turbo boost for matrix multiplication (top), Cholesky (bottom left) and QR (bottom right) using SLATE in 1ppn and 1ppc modes on Intel Broadwell nodes (max. 1.16 TFlops/s per node).