Asynchronous Task-Based Polar Decomposition on Manycore Architectures

Dalal Sukkari, Hatem Ltaief, Mathieu Faverge, David Keyes

Asynchronous Task-Based Polar Decomposition on Manycore Architectures

Asynchronous Task-Based Polar Decomposition on Manycore Architectures

Dalal Sukkari, Hatem Ltaief and David Keyes
Extreme Computing Research Center
King Abdullah University of Science and Technology
Thuwal, Jeddah 23955
Email: Firstname.Lastname@kaust.edu.sa

Mathieu Faverge
Bordeaux INP, CNRS, INRIA et Université de Bordeaux
Talence, France
Email: Mathieu.Faverge@inria.fr

***Abstract*—This paper introduces the first asynchronous, task-based implementation of the polar decomposition on manycore architectures. Based on a new formulation of the iterative QR dynamically-weighted Halley algorithm (QDWH) for the calculation of the polar decomposition, the proposed implementation replaces the original and hostile LU factorization for the condition number estimator by the more adequate QR factorization to enable software portability across various architectures. Relying on fine-grained computations, the novel task-based implementation is also capable of taking advantage of the identity structure of the matrix involved during the QDWH iterations, which decreases the overall algorithmic complexity. Furthermore, the artifactual synchronization points have been severely weakened compared to previous implementations, unveiling look-ahead opportunities for better hardware occupancy. The overall QDWH-based polar decomposition can then be represented as a directed acyclic graph (DAG), where nodes represent computational tasks and edges define the inter-task data dependencies. The StarPU dynamic runtime system is employed to traverse the DAG, to track the various data dependencies and to asynchronously schedule the computational tasks on the underlying hardware resources, resulting in an out-of-order task scheduling. Benchmarking experiments show significant improvements against existing state-of-the-art high performance implementations (i.e., Intel MKL and Elemental) for the polar decomposition on latest shared-memory vendors' systems (i.e., Intel Haswell/Broadwell/Knights Landing, NVIDIA K80/P100 GPUs and IBM Power8), while maintaining high numerical accuracy.**

***Keywords*-Polar decomposition; Asynchronous execution; Dynamic runtime system; Fine-grained execution; Directed acyclic graph; High performance computing**

I. Introduction

Today's most powerful supercomputers are composed of fat computational nodes over-provisioned by floating-point units [1], which may distort the balance of characteristics systems with respect to other hardware resources (the *Kiviat* diagram), such as memory per core, aggregated bandwidth, I/O nodes, interconnect, etc. Although real scientific applications are often memory-bound with low arithmetic intensity kernels, and therefore limited by the bus bandwidth, we revisit the polar decomposition, an important dense linear algebra (DLA) algorithm, which can make an effective use of the predominant floating-point units provided by the current state-of-the-art hardware vendor chips (for instance, Intel Knights Landing and NVIDIA Pascal P100). Based on the QR-based dynamically weighted Halley (QDWH) iteration introduced by Nakatsukasa et. al [2], the polar decomposition is a key algorithm for various scientific applications, e.g., in continuum mechanics to decompose stress tensors and to simulate the deformation of an object, in aerospace computations [3] during strapdown inertial navigation and other aerospace systems to describe the rotation of one coordinate system relative to a reference coordinate system, and in chemistry [4] to help the understanding of properties in terms of electron pair (chemical bond) transferability, etc. Further applications are also reported by Higham in [5].

This paper describes the first asynchronous, task-based implementation of the QDWH-based polar decomposition on manycore architectures. The standard algorithm requires up to six iterations to converge and to calculate the polar factor, depending on the condition number of the input matrix, involving O(n^3) matrix operations at each operation. Its algorithmic complexity may, therefore, be prohibitive. Nevertheless, this challenge can be compensated for the high level of concurrency exposed at each iteration [6], [7].

This paper proposes to considerably improve previous works [2], [6], [7] from two distinct algorithmic and implementation perspectives. The former consists in replacing the *hostile* LU-based matrix condition number estimation by an *adequate* QR-based implementation for broader code portability across vendor architectures. The latter has twofold aspects: (1) it permits to take advantage and exploit the structure of the identity matrix involved at each QR-based QDWH iteration, which significantly reduces the algorithmic complexity, thanks to fine-grained computations associated with a dynamic asynchronous execution, and (2) the artifactual synchronization points are severely weakened, unveiling look-ahead opportunities for better hardware occupancy. The overall QDWH-based polar decomposition can then be represented as a directed acyclic graph (DAG), where nodes represent computational tasks and edges define the inter-task data dependencies. We employ the StarPU dynamic runtime system to unroll the DAG, to track the various data dependencies and to asynchronously schedule the computational tasks on the underlying hardware resources, resulting in an out-of-order task scheduling. StarPU increases user-productivity by establishing a separation of concerns consisting in hiding the

hardware complexity from library developers. This enables end-users to target various hardware architectures with a single source code. Extensive benchmarking experiments show significant improvements against existing state-of-the-art high performance implementations (i.e., MKL and Elemental) for the polar decomposition on latest shared-memory systems (i.e., Intel Haswell/Broadwell/Knights Landing, NVIDIA K80/P100 GPUs and IBM Power8), while maintaining high numerical accuracy for well and ill conditioned matrices.

The remainder of the paper is organized as follows. Section II presents related work. Section III highlights our research contributions. Section IV briefly recalls the polar decomposition and its main computational phases. Section V describes the fundamental design of current state-of-the-art DLA software libraries, as implemented in LAPACK [8], MAGMA and PLASMA [9]. The implementation details of the high performance task-based asynchronous QDWH are given in Section VI. Section VII provides new upper-bound for the QDWH algorithmic complexity. Numerical accuracy, implementations assessments and performance comparisons with existing state-of-the-art DLA softwares are given in Section VIII and we conclude in Section IX.

II. Related Work

The polar decomposition algorithm has been well studied in the last three decades in terms of complexity and numerical robustness/accuracy [10]–[16]. It consists in decomposing a dense matrix $A = U_pH$, where U_p is the orthogonal polar factor and H is the positive semi-definite Hermitian polar factor. Initially designed with Newton's method based on an explicit matrix inversion calculation, numerical instability was reported, especially in presence of ill-conditioned matrices. An algorithm based on Halley's iteration was then introduced with asymptotically cubic rate of convergence in obtaining the final polar factor. To solve the numerical accuracy issues due to the matrix inversion computation, an inverse-free QR-based dynamically weighted Halley (QDWH) has finally been proposed by Nakatsukasa et. al [2]. However, the polar decomposition has not been implemented in a high performance computing system's environment, most probably due to its excessive algorithmic complexity, which does not reflect a practical assessment of the method. More recently, Nakatsukasa and Higham [17] have shown that QDWH can be used as a building block for the dense symmetric eigensolver and singular value decomposition [18], [19], which has brought to the fore further research directions. Indeed, previous works from the authors have implemented QDWH-based singular value decomposition on hardware accelerators [6] and distributed-memory systems [7], where the calculation of the polar factor is the most-time consuming phase. The aforementioned implementations have somewhat demonstrated limited performance scalability on multiple GPUs and large clusters. This is mostly due to the low hardware resource occupancy achieved by the inherent bulk synchronous programming model (BSP), which both implementations rely on for parallel performance. By the same token, it is also noteworthy to mention that the high performance software library Elemental [20] provides a QDWH implementation for distributed-memory systems.

Last but not least, the polar decomposition can alternatively be computed through an SVD as follows: $A = U\Sigma V^\top = UV^\top V\Sigma V^\top = U_pV\Sigma V^\top = U_pH$. This strategy has shown some performance scalability issues, due to the slow convergence of the QR algorithm on the condensed bidiagonal form [7].

III. Contributions

The following contributions represent the crux of the paper: (1) improve the standard QDWH algorithm by replacing the LU-based condition estimator with QR, without increasing the overall algorithmic complexity, while enabling software portability across hardware architectures, (2) develop the first task-based QDWH implementation based on fine-grained computations, which enables to exploit the identity data structure during the QDWH iterations, reducing up to 20% the overall complexity, (3) rely on a dynamic runtime system (i.e., StarPU) to asynchronously schedule the computational tasks among available processing units in order to improve hardware occupancy, (4) provide a comprehensive performance assessment and comparisons on a myriad of high-end architectures.

IV. The Polar Decomposition

The paper focuses on the inverse-free QDWH-based iterative procedure to calculate the polar decomposition [2], [17] of a matrix $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$, such that $A = U_pH$. To ensure the paper is self-contained, we briefly recall the convergent sequence as follows, with A the initial matrix:

$$\begin{cases} U_0 & = A/\alpha,\ \alpha = \|A\|_2 \\ U_{k+1} & = \frac{b_k}{c_k}U_k + \frac{1}{\sqrt{c_k}}\left(a_k - \frac{b_k}{c_k}\right)Q_1Q_2^\top,\ k \geq 0 \end{cases} \tag{1}$$

where $U_p = \lim_{k\to\infty}(U_k)$, and $\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R = \begin{bmatrix} \sqrt{c_k}U_k \\ I \end{bmatrix}$. H can then be found with the two steps formula:

$$H = U_p^\top * A,\ H = \frac{1}{2}(H + H^\top) \tag{2}$$

The main goal consists in calculating the optimal parameters (a_k, b_k, c_k) so that cubical convergence can be attained during the QDWH iteration. The expression of the parameters (a_k, b_k, c_k) can be written as follows:

$$\begin{aligned} &a_k = h(l_k),\ b_k = (a_k - 1)^2/4,\ c_k = a_k + b_k - 1, \\ &l_0 = \frac{\beta}{\alpha},\ l_k = \frac{l_{k-1}(a_{k-1} + b_{k-1}l_{k-1}^2)}{1 + c_{k-1}l_{k-1}^2},\ k >= 1, \\ &h(l) = \sqrt{1+d} + \frac{1}{2}\sqrt{8 - 4d + \frac{8(2-l^2)}{l^2\sqrt{1+d}}}, d = \sqrt[3]{\frac{4(1-l^2)}{l^4}}, \end{aligned} \tag{3}$$

with $\beta = 1/\|A^{-1}\|_2$. For ill-conditioned matrices, the number of iterations k can vary up to six. We refer to [2] for further details on the theoretical proof. When U_k becomes

well-conditioned, it is possible to replace Equation 1 with a Cholesky-based implementation as follows:

$$U_{k+1} = \frac{b_k}{c_k} U_k + \left(a_k - \frac{b_k}{c_k} \right) (U_k W_k^{-1}) W_k^{-\top}, \\ W_k = \text{chol}(Z_k),\ Z_k = I + c_k U_k^{\top} U_k. \tag{4}$$

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 practice, this transition is monitored by setting a threshold for c_k. Once convergence is reached, the polar factor is $U_p = U_k$ and the positive semi-definite Hermitian polar factor corresponds to $H = U_p^{\top} A$. All in all, the number of floating-point operations depends on the number of iterations required to converge, which is dictated by the condition number of the original matrix problem. Typically, for ill-conditioned matrices, QDWH will perform three QR-based QDWH iterations (Equation 1), followed by three Cholesky-based QDWH iterations (Equation 4), besides executing other compute-intensive Level 3 BLAS operations, i.e., triangular solves, applications of Householder reflectors, matrix-matrix multiplications etc.

V. High Performance DLA Software Libraries

As discussed in Section IV, although the QDWH-based polar decomposition is a challenging and complex algorithm, it relies on conventional dense linear algebra operations, e.g., QR/Cholesky-based linear solvers, which are well-supported by several open-source and vendor-optimized numerical libraries. These libraries can be differentiated into the two following algorithmic categories.

A. Block Algorithms

Block algorithms rely on successive panel and update sequences to perform matrix computations. The panel phase is memory-bound and does not benefit from thread parallelism, while the phase of the trailing submatrix update is highly parallel, in which computations are applied by means of multithreaded Level 3 BLAS kernel executions. These sequences are characteristic of the fork-join paradigm, alternating sequential and parallel computational phases, and therefore, suffer from performance losses due to low hardware occupancy engendered by unnecessary in-between synchronization points. In fact, this bulk synchronous programming model corresponds to the backbone of many open-source and vendors' state-of-the-art numerical libraries such as LAPACK [8], MAGMA [21] and ScaLAPACK [22] for shared-memory, accelerator-based and distributed-memory systems, respectively. As highlighted in the exascale software roadmap [23], BSP models may need to be reconsidered, especially in presence of millions of cores, which already constitute today's supercomputers [1].

B. Tile Algorithms

To answer this call for action and provide a solution for the challenge brought by the manycore era, the DLA community has initiated a decade ago a profound redesign of matrix computation algorithms in order to benefit from the high level of concurrency. This translated into breaking down the dense matrix data structure into tiles following a tile data layout as opposed to the standard column-major format, which is the standard for block algorithms. The various matrix operations can then be represented as a directed acyclic graph (DAG), where nodes represent sequential computational tasks and edges define the inter-task data dependencies. The resulting fine-grained computations permit to weaken the artifactual synchronization points by bringing to the fore look-ahead opportunities, which in return, can be exploited by dynamic runtime systems in keeping threads in a busy state throughout the entire execution. The performance gain of tile versus block algorithms have been thoroughly addressed in the literature [24]–[26], in the context of PLASMA [27] and FLAME [28] numerical software libraries, using QUARK [29] and SuperMatrix [30] runtime systems, respectively. More recently, in a community effort to enhance user productivity by abstracting the hardware complexity, the Chameleon library [31] has been developed to target multiple hardware architectures with a single source code. This is achieved by standardizing existing dynamic runtime system APIs (e.g., OpenMP [32], OmpSs [33], QUARK [29], StarPU [34], PaRSEC [35]) through a thin layer of abstraction, making the user developer experience oblivious to the underneath runtime system and its corresponding hardware deployment. For instance, this oblivious software infrastructure has been already used in the context of computational astronomy using StarPU [36], and more recently with OmpSs [37].

The QDWH-based polar decomposition resurfaces during a possible period of convergence, where hardware/software co-design plays now a major role in designing future systems and numerical libraries for exascale.

VI. High Performance Implementations

In this section, we describe the task-based implementation of the QDWH algorithm and the novel optimizations introduced to increase hardware occupancy and overall performance, in the context of the Chameleon library [31].

A. The StarPU Runtime System

StarPU is the *de facto* dynamic runtime system for Chameleon. Although Chameleon supports other runtimes (e.g., PaRSEC [35], QUARK [29] and recently OmpSS [33]), we decided to solely rely on the StarPU [34] runtime system to implement this algorithm, since it is probably one of the most mature runtime systems when it comes to supporting various hardware architectures. StarPU is a runtime, which deals with the execution of generic task graphs. This task graph is given through the sequential task flow (STF) programming model where tasks are inserted to the runtime in a sequential manner with additional hints on the data usage: read, write, read-write. Then, the runtime automatically infers data dependencies from those hints while unrolling the sequential flow of task submissions, and is in charge of scheduling the

tasks while enforcing those dependencies. The PaRSEC [35] runtime is also used in dense linear algebra libraries but relies on a parameterized task graph (PTG) representation of the application. This model allows for compact and problem size independent representations of the graph to execute but is usually less intuitive to the non familiarized user.

One of the main advantages of using the task-based implementation is to become oblivious of the targeted architectures. This improves the user productivity, and it is even more realistic for runtimes as StarPU, which are able to transparently handle single heterogeneous nodes, and eventually multiple heterogeneous nodes in case the StarPU-MPI [38] extension is used. To enable such portability, StarPU tasks are associated to codelets which groups under the same name multiple implementations of the same task: CPU, CUDA, OpenCL, OpenMP, ... At runtime, StarPU will automatically decide which implementation of the task is better suited to achieve the highest performance based on cost models. These cost models are automatically generated by StarPU when executing the application and kept for subsequent executions. These models are especially important to the Heterogeneous First Time [39] (HeFT) scheduling strategy used by StarPU, when accelerators are involved in the computations. Another benefit from using such programming models is the capabilities offered to the programmer to submit simultaneously independent steps of an application to raise the resources occupancy, and add a single synchronization point when all steps are performed. The `MORSE_xxxx_Tile_Async` interface of the Chameleon library offers this capability to interleave multiple dense linear algebra operations when it is possible. Conversely, the non-asynchronous interface, `MORSE_xxxx_Tile`, enforces a synchronization call at the end of the function to wait for the end of all submitted tasks.

B. Task-Based QDWH Polar Decomposition Pseudo-Code

Algorithm 1 presents the pseudo-code of the task-based QDWH implementation on top of the Chameleon library. It is decomposed in three main code sections. The first one from row 1 to 6 evaluates the two-norm of the input matrix A that is required to start the iterative process. The specificity of this code section is the 2-norm estimator `genm2` that we introduced in the Chameleon library through an iterative computation in which we tried to minimized the number of synchronizations. The second section of the algorithm computes the initial condition number l_0 from row 7 to 19. The classical way consists in computing an LU factorization of the matrix A, and its one-norm. Then, it is possible to compute an estimator of the condition number with those two information (`dgecon`). The main challenge here resides in the LU factorization with partial pivoting, which is difficult to implement using task-based programming model, due to the global synchronization points needed during the panel factorization while looking for pivot candidates and the resulting row swapping step. Some solutions have been proposed on shared-memory systems [40] but there are no existing solutions that are oblivious of heterogeneous architectures. We thus propose

Algorithm 1 QDWH pseudo-code on top of Chameleon

```
1: /* Estimate the condition number */
2: dlacpy_Async( A, U )                                   ▷ U = A
3: dlacpy_Async( A, B )                                   ▷ B = A
4: Anorm = dlange_Async( A )                              ▷ ||A||_1
5: dgenm2( A, α )                                         ▷ α ≈ ||A||_2
6: RUNTIME_sequence_wait()
7: /* Compute U_0 and l_0*/
8: dlascl_Async( U, 1./α )                                ▷ U_0 = A/α
9: if lu then
10:    dgetrf_Async( B );                                 ▷ A = LU
11:    l_0 = dgecon( B, Anorm )                           ▷ l_0 ≈ 1/(||A^-1||_1 ||A||_1)
12: else
13:    dgeqrf_Async( B )                                  ▷ A = QR
14:    dtrtri_Async( B )                                  ▷ Compute R^-1
15:    Ainvnorm = dlantr_Async( B )                       ▷ ≈ ||A^-1||_1
16:    RUNTIME_sequence_wait()
17:    l_0 = 1./(Ainvnorm * Anorm)
18: end if
19: l_0 = (α/1.1) * l_0
20:
21: /* Compute the polar decomposition A = U_p H using QDWH */
22: k = 1, Li = β × α/1.1, conv = 100
23: while (conv ≥ ∛(5eps) || |Li − 1| ≥ 5eps) do
24:    L2 = Li^2, dd = ∛((4(1 − L2)/L2^2))
25:    sqd = √(1 + dd)
26:    a1 = sqd + √(8 − 4 × dd + 8(2 − L2)/(L2 × sqd))/2
27:    a = real(a1), b = (a − 1)^2/4, c = a + b − 1
28:    Li = Li(a + b × L2)/(1 + cL2)
29:    dlacpy_Async( U, U1 )                              ▷ Backup U_{k−1}
30:
31:    /* Compute U_k from U_{k−1} */
32:    if c > 100 then
33:       C = [C1; C2] = [√c U_{k−1}; I]
34:       dgeqrf_Async( C )                               ▷ C = QR = [Q1; Q2] R
35:       dorgqr_Async( C )                               ▷ C = Q = [Q1; Q2]
36:       dgemm_Async( Q1, Q2^⊤, U )                       ▷ U_k = 1/√c (a − b/c) Q1 Q2^⊤ + b/c U_{k−1}
37:    else
38:       dlaset_Async( Z, 0., 1. )                       ▷ Z = I
39:       dgemm_Async( U^⊤, U, Z)                         ▷ Z_k = I − c U_{k−1}^⊤ U_{−1} k
40:       dgeadd_Async( U, B )                            ▷ B = U_{k−1}^⊤
41:       dposv_Async( Z, B )                             ▷ Solve Z_k x = U_{k−1}^⊤
42:       dgeadd_Async( B, U )                            ▷
          U_k = b/c U_{k−1} + (a − b/c) (U_{k−1} W_{k−1}^-1) W_{k−1}
43:    end if
44:    dgeadd_Async( U, U1 )                              ▷ U_k − U_{k−1}
45:    dlange_Async( U1, conv )                           ▷ conv = ||U_k − U_{k−1}||_F
46:    RUNTIME_sequence_wait()
47:    k = k + 1
48: end while
49:
50: /* Compute H */
51: dgemm_Async( U_k, A, H )                              ▷ H = U_p^⊤ A
52: dlacpy_Async( H, B )                                  ▷ B = H
53: dgeadd_Async( B, H )                                  ▷ H = 1/2 (H + H^⊤)
54: RUNTIME_sequence_wait()
```

a QR-based solution which consists in estimating the norm of A^{-1} by computing the norm of R^{-1} with $A = QR$. This solution, which turns out to be less costly, alleviates the pivoting issue all together, uses only regular tile algorithms and allows code portability across various architectures, thanks to the underlying runtime system. The third section of the algorithm, rows 21 to 48, is the main loop of the algorithm, which iterates on U_k and converges to the polar factors. This section of the algorithm is straightforward and follows the mathematical description of the problem using either a QR or a Cholesky factorization to calculate the next U. Finally, the last section, rows 49 to 53, computes the Hermitian polar factor H from the polar factor computed out of the main loop.

C. Code Optimizations

The Chameleon library provides two APIs to perform dense matrix computations. The first one, `MORSE_xxxx_Tile`, is a synchronous implementation of a linear algebra operation. This means that all the tasks required for the computations are submitted to the runtime, and then the library internally waits for the completion of all tasks before returning the control to the programmer. This is the first version we implemented, in black in the algorithm 1. To highlight the benefit of using task-based programming model (through tile algorithms) as opposed to the fork-join paradigm, as implemented in the LAPACK library, we have manually integrated synchronization points within the QR/Cholesky factorization kernel calls, at the end of each panel and update computations, to better emphasize on the performance discrepancy between both approaches. We refer to this reference implementation as *Sync*.

The second API, `MORSE_xxxx_Tile_Async`, ensures that all the tasks of an algorithm are submitted to the runtime, but their completion is not ensured when the function call returns. Thus, it is possible to simultaneously submit tasks of multiple operations. The runtime is in charge of keeping the data coherency of tasks, generated from different kernel calls, since these tasks may operate on the same data. Operations that are asynchronously submitted to the runtime are indicated in red in Algorithm 1. At some point of the algorithm, synchronization points are however required to guarantee the consistency of the results. This is made through a call to `RUNTIME_sequence_wait()`, which waits for the completion of all tasks. It is then possible to release synchronization in the three steps of the algorithm to ensure a better occupancy of the resources, especially on small to medium test cases, as presented in Section VIII. We refer to this implementation as *Async*. It is also noteworthy to mention that it is possible to estimate *offline* the minimal number of iterations performed in the main loop. In that case, the synchronization in line 45 can be safely removed for the first iterations and introduced only for the last iteration as a sanity check on the value $conv$ against the convergence threshold.

The last optimization is the possible acceleration of the QR-based Halley iterations. This optimization consists in exploiting the identity matrix structure of the C_2 matrix in the QR factorization (line 34 in Algorithm 1) and the corresponding Q generation (line 35 in Algorithm 1). Indeed, thanks to tile algorithms, it is possible to design a specific QR factorization algorithm in order to factorize a dense matrix on top of an identity matrix. This new QR factorization takes into account the identity matrix structure so that only non-zeros tiles are operated on during the factorization. By the same token, during the Q generation step, only the non-zeros tiles containing the Householder reflectors will be accessed. This optimization is important as it reduces the number of flops as well as data movement. We refer to this implementation as *OptId*.

VII. Arithmetic Complexity

In this Section, we present the algorithmic complexity (flops) of the polar decomposition using two variants based on the Halley iteration (QDWH) and the SVD. For simplicity purposes, we consider only square dense matrices, but QDWH works also for rectangular matrices [17].

A. The QDWH-based Polar Decomposition

The condition number estimation l_0 can be calculated using the LU factorization, which requires $\frac{2}{3}n^3$, followed by two triangular solvers $LX = Id$ and $UA^{-1} = X$, adding $2n^3$ flops. Alternatively, l_0 can be calculated using the QR factorization, $A = QR$ which needs $\frac{4}{3}n^3$, followed by inverting the upper triangular matrix R with $\frac{1}{3}n^3$. Calculating l_0 using the QR factorization needs less flops overall. Moreover, the resulting QR factors can be reused during the first iteration of QDWH, thanks to fine-grained computations.

As shown in Equation 1, 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 $Q_1 Q_2^\top$ requires $2n^3$ 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 4, 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 Hermitian polar factor $H = U_p^\top A$ 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 [17], the flop count of QDWH depends on l_0, which involves during the QDWH iteration. The total flop count of QDWH for dense matrices ranges then from $(10 + \frac{2}{3})n^3$ (for $l_0 \approx 1$ with $\#it_{Chol} = 2$) to $41n^3$ (for $l_0 \gg 1$, with typically $\#it_{QR} = 3$ and $\#it_{Chol} = 3$). Furthermore, taking advantage of the trailing identity matrix structure in the QR factorization (OptId) reduces the flop count of the iteration in Equation 1. Forming the upper triangular matrix R by applying the Householder reflectors with $n+1$ nonzero elements $\Pi_{k=1}^{n-1} H_k A = R$ to k vectors requires $4(n+1)k$ flops, therefore forming R needs $\Sigma_{k=1}^{n-1} 4(n-k)(n+1) = 2n^3$ flops. Accumulating the Householder reflectors to form $\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} = \Pi_{k=1}^{n-1} H_{n-k} \begin{bmatrix} I_n \\ 0 \end{bmatrix}$ requires $\Sigma_{k=1}^{n-1} 4nk = 2n^3$, as explained in [17]. Table I summarizes the total flop count of QDWH (including condition number estimation and Halley iteration) (1) when using LU to estimate l_0 (original implementation), (2) when using QR to estimate l_0 and reusing the QR factors in the first iteration of QDWH and (3) when additionally taking advantage of the identity matrix structure in QR-based iterations (Equation 1).

B. The SVD-based Polar Decomposition

The polar decomposition can be calculated via SVD as follows, $A = U\Sigma V^\top$, then, $U_p = UV$, $H = V^\top \Sigma V$. Therefore, the flop count of this approach includes the cost of an SVD decomposition, a matrix-matrix multiplication to compute the orthogonal polar factor U_p and a matrix-matrix

	well		ill
(1) QDWH+LU	$(13+\frac{1}{3})n^3 \leq$	$\cdots$	$\leq (43+\frac{2}{3})n^3$
(2) QDWH+QR	$(12+\frac{1}{3})n^3 \leq$	$\cdots$	$\leq (41+\frac{1}{3})n^3$
(3) QDWH+QR+OptId	$(12+\frac{1}{3})n^3 \leq$	$\cdots$	$\leq (33+\frac{1}{3})n^3$

TABLE I
ALGORITHMIC COMPLEXITY OF THE QDWH-BASED POLAR DECOMPOSITION.

multiplication to calculate the Hermitian polar factor H. The standard approach to compute the SVD of a dense matrix is to first reduce it to bidiagonal form $A = U_1 B V_1^\top$. The subsequent left and right singular vectors from the bidiagonal solver are then accumulated during the back transformation phase, i.e., $U = U_1 U_2$ and $V = V_2 V_1$, to calculate the singular vectors of the original matrix A. The final estimated flop count to calculate the SVD is $22n^3$, as implemented in the divide-and-conquer DGESDD [41]. Then, we need to add $2n^3$ to compute $U_p = UV$, and n^3 to compute $H = V^\top \Sigma V$ (symmetric rank-k update operation). The final estimated cost of the polar decomposition using SVD is, therefore, $25n^3$. Compared to the QDWH-based polar decomposition (3) in Table I, this is 30% less than in case of ill-conditioned matrices and almost twice the flops in case of well-conditioned matrices. In theory, it seems there is a clear advantage to use SVD-based for the polar decomposition in presence of ill-conditioned matrices. However, the SVD algorithm inherently suffers from lack of parallelism, due to a very expensive panel factorization phase and may not be as competitive as QDWH-based approaches.

VIII. PERFORMANCE RESULTS AND ANALYSIS

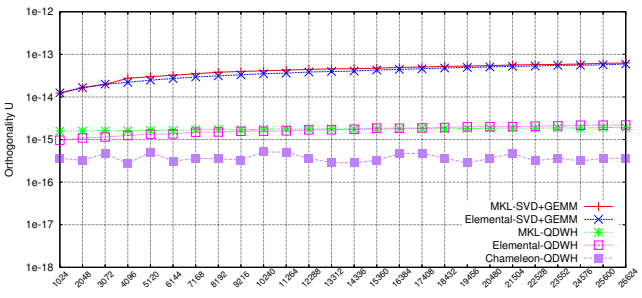
This Section provides a comprehensive performance analysis of the task-based QDWH implementation in the context of the Chameleon library with the dynamic runtime system StarPU on various architectures.

A. Environment Settings

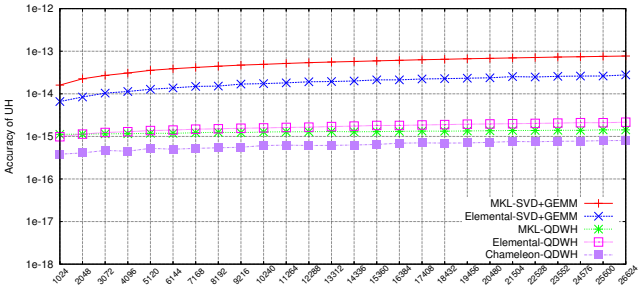
We have considered three different systems, which are representative of the current manycore-based hardware trends. The first system is composed of dual-socket 16-core Intel Haswell Intel Xeon CPU E5-2698 v3 running at 2.30GHz equipped with 8 NVIDIA K80s (2 GPUs per board). The second system hosts the latest Intel commodity chip with dual-socket 14-core Intel Broadwell Intel Xeon E5-2680 v4 running at 2.4GHz. The third system has the latest Intel manycore Knights Landing (KNL) 7210 chips with 64 cores. For simplicity purposes, each system is named after his chip codename. Our QDWH implementation has been compiled with Intel compiler 16 and linked against the Chameleon library v0.9.0 with hwloc v1.11.4, StarPU v1.2.0 and Intel MKL v11.3.1. We have mainly considered ill-conditioned randomly generated matrices, since this represents the worse case scenario, where QDWH performs a maximum of six iterations. In particular, in the subsequent experiments, our QDWH implementation switches from Equation 1 to Equation 4 if c_k is smaller than 100 (see Algorithm 1), which generates QR-based iterations for the first three followed by three Cholesky-based iterations.

B. Numerical Accuracy

We recall the polar decomposition of a given general matrix $A \in \mathbb{R}^{n \times n}$: $A = U_p H$. The norm $\| \cdot \|_F$ denotes the Frobenius norm. To highlight the numerical robustness of the method, we use the following two accuracy metrics: $\frac{\|I - U_p^\top U_p\|_F}{\|A\|_F}$ for the orthogonality of the polar factor U_p and $\frac{\|A - U_p H\|_F}{\|A\|_F}$ for the accuracy of the overall computed polar decomposition. Figure 1 presents the orthogonality of U_p and the accuracy of the polar decomposition $A = U_p H$ for ill-conditioned matrix on the KNL system (very similar numerical results on other systems). We can distinguish two clusters, i.e., QDWH-based and SVD-base polar decomposition, with up to two digits difference in the orthogonality and accuracy magnitudes. Although both mostly employ orthogonal transformations, the SVD variant of the polar decomposition necessitates the QR algorithm, which may show some convergence limitations with ill-conditioned matrices, as shown later in Section VIII-F.



(a) Orthogonality of U_p.



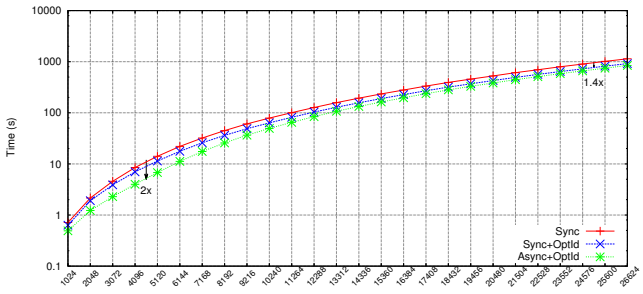
(b) Accuracy of $U_p H$.

Fig. 1. Assessing the numerical accuracy/robustness of the task-based QDWH.

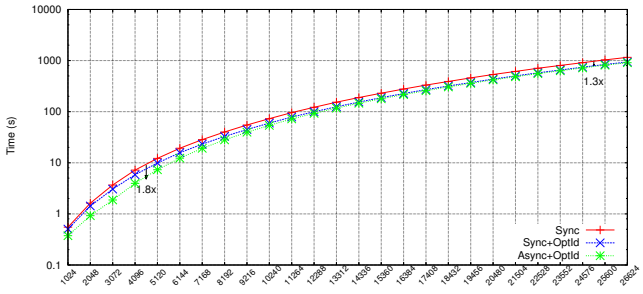
C. Incremental Optimizations

Figure 2 highlights the performance impact of various incremental optimizations on the task-based QDWH, as described in Section VI-C. Taking advantage of the identity matrix structure (OptId) engenders up to 20% performance improvements compared to the oblivious approach on all studied systems.

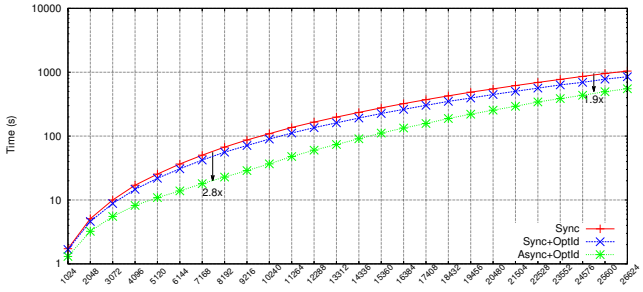
Running additionally in asynchronous mode (Async) further reduces time to solution (up to 2.8x), especially for medium range of matrix sizes, where processing units run out of work and look-ahead techniques jump right in to fill the performance gap. For asymptotic matrix sizes, although work is abundant, the asynchronous mode still provides additional performance. In particular, on KNL and Haswell+8x80 systems, data movement engendered by NUMA and PCIe channels is expensive and can be overlapped by computations, thanks to the Async optimization.



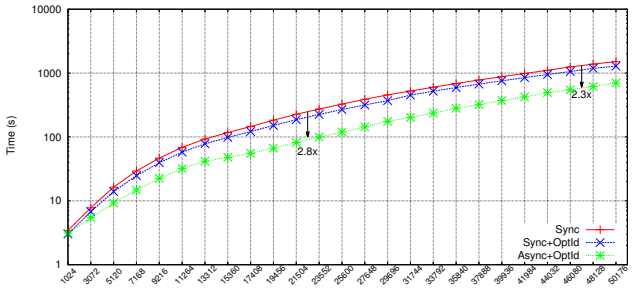
(a) Haswell.



(b) Broadwell.



(c) KNL.

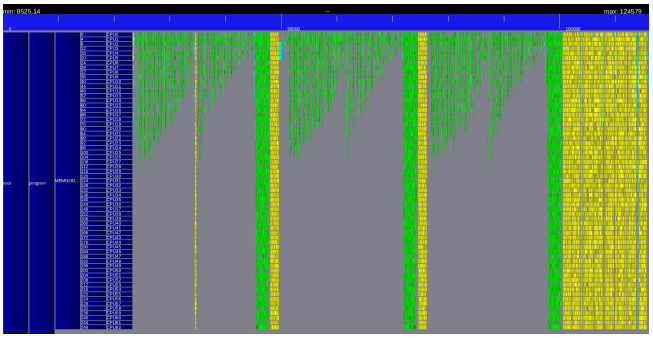


(d) Haswell+8xK80.

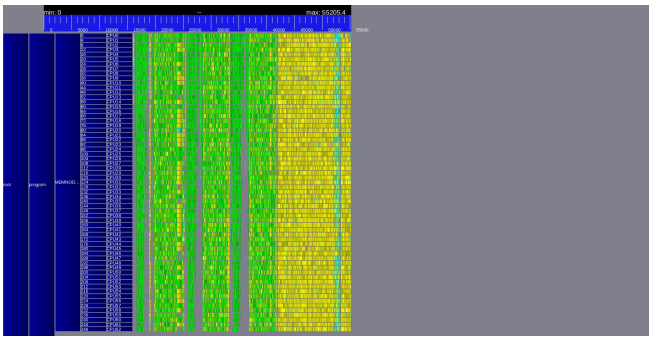
Fig. 2. Assessing the performance of various incremental optimizations.

D. Execution Traces

Figure 3 shows the execution traces when running in synchronous (_Tile API) and asynchronous (_Tile_Async API) modes. We have added additional synchronization points within the _Tile kernel API, after each panel/update computation, so that we can better capture the performance gain against coarse-grained computations engendered by block algorithms, as described in Section V. These traces have been obtained on the KNL system for a matrix size of 10K. Since the matrix is ill-conditioned, the task-based QDWH performs six iterations (three QR-based and three Cholesky-based). The green, blue and yellow blocks correspond to QR, Cholesky/Level 3 BLAS and Level 1/2 BLAS, respectively. We can clearly notice the idle time during the first three QR-based iterations when running with a synchronous mode (Figure 3(a)). The performance impact of synchronous execution for the next three Cholesky-based iterations is not as severe as QR-based iterations because the Cholesky panel factorization involves only the diagonal block (Figure 3(b)). For the subsequent



(a) Synchronous task-based QDWH.



(b) Asynchronous task-based QDWH.

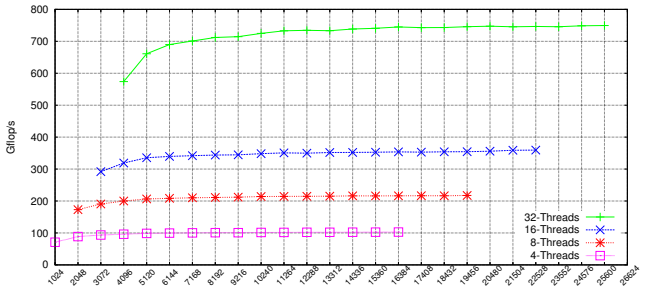
Fig. 3. Assessing synchronous Vs asynchronous execution traces of task-based QDWH on the KNL system with a matrix size of 10K.

graphs, the performance curves of the task-based QDWH correspond to performance when all optimizations are enabled (i.e., Async and OptId).

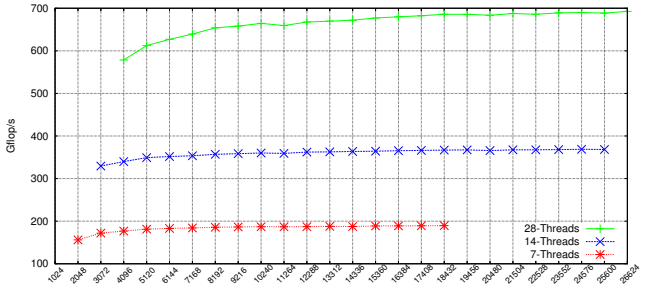
E. Performance Scalability

Figure 4 demonstrates the performance scalability of the task-based QDWH implementation. The scalability is almost linear for the commodity CPU systems (i.e, Haswell/Broadwell). On KNL and Haswell+8xK80, although

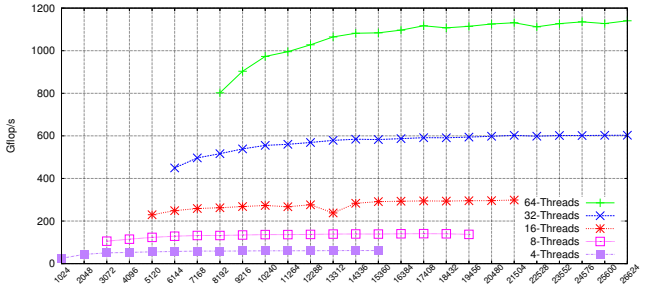
the overhead of moving data on these systems is higher than the commodity CPU platforms, StarPU is able to cope with these communication overheads and the overall scalability is still decent.



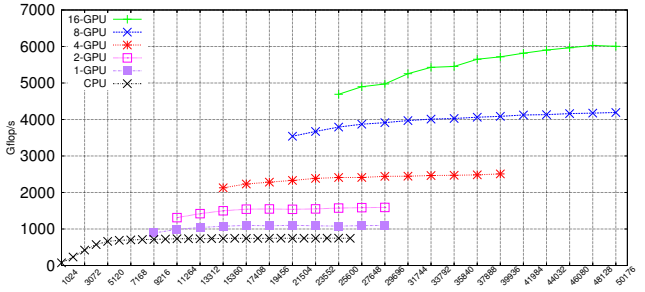
(a) Haswell.



(b) Broadwell.



(c) KNL.



(d) Haswell+8xK80.

Fig. 4. Assessing the task-based QDWH scalability.

F. Performance Comparisons of QDWH Variants

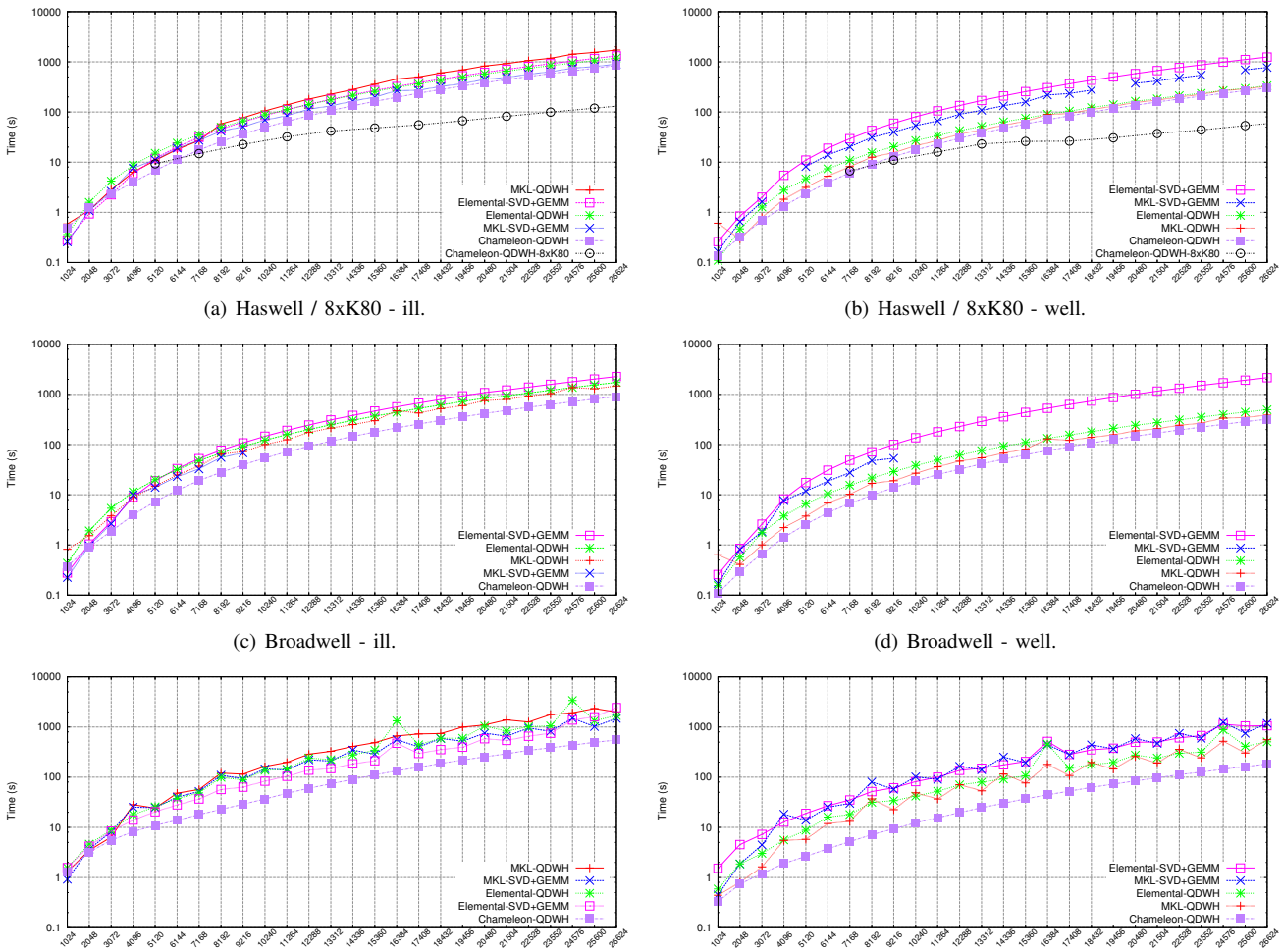
Figure 5 reports task-based QDWH performance against other various existing QDWH implementations on ill (left) and well (right) conditioned matrices, across the three systems. Missing data points correspond to runs, which did not achieve the proper accuracy, as defined in Section VIII-B. For well-conditioned matrices, time to solution is much more shortened for the QDWH implementation variants, thanks to less iterations for convergence. The SVD variants of the polar decomposition do not seem to take advantage of such matrices since the bidiagonal reduction and the matrix-matrix multiplication have still to be performed in the same manner, regardless of the matrix condition number. All in all, the task-based QDWH achieves gains up to [6%, 8%] on Haswell and [39%, 17%] on Broadwell, [85%, 82%] on Haswell+8xK80, and [63%, 67%] on KNL against the best (non task-based) implementation for [ill, well]-conditioned matrices, respectively. Highest performance are achieved on systems where data movement are most expensive (e.g., NUMA for KNL and PCIe for Haswell+8xK80) since the asynchronous mode can mitigate the overhead of data transfers by overlapping communications with task computations. Also, compared to MAGMA_QDWH [6], the task-based QDWH achieves gains up to [71%, 22%] on Haswell+4xK80 for [ill, well]-conditioned matrices, respectively.

G. Performance Comparisons Across Architectures

We have additionally considered two more recent architectures, i.e., a dual-socket 10-cores IBM Power8 (3.69GHz) and a dual-socket 16-cores Intel Haswell equipped with four NVIDIA Pascal P100 GPUs. Figure 6 presents the performance of the task-based QDWH across all systems investigated in the paper. The main idea is not to cross-compare the performance delivered by each system but rather to show that the task-based QDWH can support various architectures with a decent sustained peak (up to 90% and up to 60% of the sustained Chameleon DGEMM peak for CPU only systems and for KNL/GPUs platforms, respectively).

IX. Conclusion and Future Work

We have presented a comprehensive performance analysis of a novel asynchronous task-based QDWH algorithm for the polar decomposition of a dense matrix. Thanks to fine-grained computations, we have reduced by 20% the overall complexity by taking advantage of the identity structure of the matrix during the iterations, while exposing look-ahead opportunities to increase hardware occupancy. Furthermore, the Chameleon library and its dynamic runtime system StarPU abstracts the hardware complexity from end-users and is capable of asynchronously scheduling computational tasks on the underlying processing units. Thanks to its wide hardware range support, we demonstrated that StarPU can port a single sequential source code to a myriad of hardware systems. Experimental results of the asynchronous task-based QDWH show significant performance improvement (up to an order of magnitude) against state-of-the-art implementations on ill and well-conditioned matrices across various hardware technologies, which are paving the road to future petascale/exascale systems. Future works include using the task-based QDWH as a building block for the dense symmetric eigensolver and SVD on shared and distributed-memory systems. We would like also



(a) Haswell / 8xK80 - ill.

(b) Haswell / 8xK80 - well.

(c) Broadwell - ill.

(d) Broadwell - well.

(e) KNL - ill.

(f) KNL - well.

Fig. 5. Assessing task-based QDWH performance against other QDWH variant implementations on ill (left) and well (right) conditioned matrices.

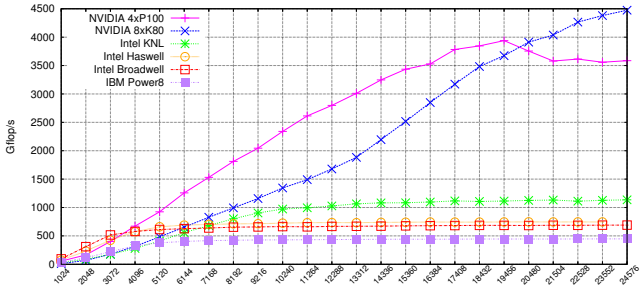


Fig. 6. Task-based QDWH performance across various architectures.

to investigate other QDWH variants, which may require more flops but entails an even higher level of concurrency.

ACKNOWLEDGMENT

The authors would like to thank Samuel Thibault from Inria for his support with StarPU, Jack Poulson from Google Inc. for his help in tuning Elemental and the vendors Cray/IBM/Intel/NVIDIA for their hardware donations and/or systems' remote accesses in the context of the Cray Center of Excellence, the Intel Parallel Computing Center and the NVIDIA GPU Research Center awarded to the Extreme Computing Research Center at KAUST.

REFERENCES

[1] "The Top500 List," http://www.top500.org/.

[2] Y. Nakatsukasa, Z. Bai, and F. Gygi, "Optimizing Halley's Iteration for Computing the Matrix Polar Decomposition," *SIAM Journal on Matrix Analysis and Applications*, pp. 2700–2720, 2010.

[3] J. Meyer and I. Y. Bar-itzhack, "Practical Comparison of Iterative Matrix Orthogonalization Algorithms," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-13, no. 3, pp. 230–235, May 1977.

[4] J. A. Goldstein and M. Levy, "Linear algebra and quantum chemistry," *Am. Math. Monthly*, vol. 98, no. 10, pp. 710–718, Oct. 1991. [Online]. Available: http://dx.doi.org/10.2307/2324422

[5] N. J. Higham, "Computing the Polar Decompositionwith Applications," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 4, pp. 1160–1174, 1986. [Online]. Available: http://dx.doi.org/10.1137/0907079

[6] D. Sukkari, H. Ltaief, and D. Keyes, "A High Performance QDWH-SVD Solver Using Hardware Accelerators," *ACM Trans. Math. Softw.*, vol. 43, no. 1, pp. 6:1–6:25, Aug 2016. [Online]. Available: http://doi.acm.org/10.1145/2894747

[7] ——, "High Performance Polar Decomposition on Distributed Memory Systems," in *Best Papers, Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing,*

Grenoble, France, August 24-26, 2016, Proceedings, P.-F. Dutot and D. Trystram, Eds. Cham: Springer International Publishing, 2016, pp. 605–616. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-43659-3_44

[8] E. Anderson, Z. Bai, C. H. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK User's Guide*, 3rd ed. Philadelphia: Society for Industrial and Applied Mathematics, 1999.

[9] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects," in *Journal of Physics: Conference Series*, vol. 180, 2009.

[10] W. Gander, "On Halley's iteration method," *American Mathematical Monthly*, vol. 92, no. 2, pp. 131–134, ???? 1985.

[11] ——, "Algorithms for the polar decomposition," *SIAM J. Scientific Computing*, vol. 11, no. 6, pp. 1102–1115, 1990. [Online]. Available: http://dx.doi.org/10.1137/0911062

[12] C. S. Kenney and A. J. Laub, "On scaling Newton's method for polar decomposition and the matrix sign function," *SIAM J. Matr. Anal. Appl.*, vol. 13, pp. 688–706, 1992, cited in a personal communication by Alan Laub.

[13] N. J. Higham and P. Papadimitriou, "A parallel algorithm for computing the polar decomposition," *Parallel Computing*, vol. 20, no. 8, pp. 1161–1173, Aug. 1994.

[14] A. Kielbasinski and K. Zietak, "Numerical behaviour of higham's scaled method for polar decomposition," *Numerical Algorithms*, vol. 32, no. 2-4, pp. 105–140, 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1024098014869

[15] R. Byers and H. Xu, "A new scaling for newton's iteration for the polar decomposition and its backward stability," *SIAM J. Matrix Analysis Applications*, vol. 30, no. 2, pp. 822–843, 2008. [Online]. Available: http://dx.doi.org/10.1137/070699895

[16] B. Laszkiewicz and K. Zietak, "Approximation of matrices and a family of gander methods for polar decomposition," *BIT Numerical Mathematics*, vol. 46, no. 2, pp. 345–366, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10543-006-0053-4

[17] Y. Nakatsukasa and N. J. Higham, "Stable and Efficient Spectral Divide and Conquer Algorithms for the Symmetric Eigenvalue Decomposition and the SVD," *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. A1325–A1349, 2013. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/120876605

[18] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., ser. John Hopkins Studies in the Mathematical Sciences. Baltimore, Maryland: Johns Hopkins University Press, 1996.

[19] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. Philadelphia, PA: SIAM, 1997. [Online]. Available: http://www.siam.org/books/OT50/Index.htm

[20] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Softw*, vol. 39, no. 2, p. 13, 2013. [Online]. Available: http://doi.acm.org/10.1145/2427023.2427030

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

[22] L. S. Blackford, J. Choi, A. Cleary, E. F. D'Azevedo, J. W. Demmel, I. S. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia: Society for Industrial and Applied Mathematics, 1997.

[23] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The International Exascale Software Project Roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342010391989

[24] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parellel Comput. Syst. Appl.*, vol. 35, pp. 38–53, 2009, http://dx.doi.org/10.1016/j.parco.2008.10.002 DOI: 10.1016/j.parco.2008.10.002.

[25] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. V. D. Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.*, vol. 36, pp. 14:1–14:26, July 2009. [Online]. Available: http://doi.acm.org/10.1145/1527286.1527288

[26] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra, "Comparative study of one-sided factorizations with multiple software packages on multi-core hardware," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[27] *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3*, University of Tennessee, November 2010.

[28] "The FLAME project," April 2010, http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage.

[29] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.

[30] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 116–125.

[31] "The Chameleon project," January 2016, https://project.inria.fr/chameleon/.

[32] A. OpenMP, "Openmp application program interface version 4.0," 2013.

[33] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta, "A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks," *International Journal of Parallel Programming*, vol. 37, no. 3, pp. 292–305, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10766-009-0101-1

[34] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency Computat. Pract. Exper.*, vol. 23, pp. 187–198, 2011, (to appear).

[35] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for High Performance Computing," *Parallel Computing*, vol. 38, no. 1–2, pp. 37–51, 2012, extensions for Next-Generation Parallel Programming Models. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001347

[36] A. Charara, H. Ltaief, D. Gratadour, D. E. Keyes, A. Sevin, A. Abdelfattah, E. Gendron, C. Morel, and F. Vidal, "Pipelining Computational Stages of the Tomographic Reconstructor for Multi-Object Adaptive Optics on a Multi-GPU System," in *SC'14*. IEEE, 2014, pp. 262–273. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7012142

[37] H. Ltaief, D. Gratadour, A. Charara, and E. Gendron, "Adaptive Optics Simulation for the World's Largest Telescope on Multicore Architectures with Multiple GPUs," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '16. New York, NY, USA: ACM, 2016, pp. 9:1–9:12. [Online]. Available: http://doi.acm.org/10.1145/2929908.2929920

[38] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," Inria Bordeaux Sud-Ouest ; Bordeaux INP ; CNRS ; Université de Bordeaux ; CEA, Research Report RR-8927, Jun. 2016. [Online]. Available: https://hal.inria.fr/hal-01332774

[39] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[40] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, "Exploiting fine-grain parallelism in recursive lu factorization." in *PARCO*, 2011, pp. 429–436.

[41] P. C. Hansen, *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*, ser. Mathematical Modeling and Computation. Philadelphia: Society for Industrial and Applied Mathematics, 1998. [Online]. Available: http://books.google.com.sa/books?id=A5XWG_PFFdcC