High Performance Polar Decomposition on Manycore Systems and its application to Symmetric Eigensolvers and the Singular Value Decomposition

Dissertation by
Dalal Sukkari

In Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy

King Abdullah University of Science and Technology
Thuwal, Kingdom of Saudi Arabia

April, 2019
The dissertation of Dalal Sukkari is approved by the examination committee

Committee Chairperson: Prof. David E. Keyes
Committee Members: Dr. Hatem Ltaief, Prof. Mohamed-Slim Alouini, Prof. Taous-Meriem Laleg, Prof. Daniel Kressner

King Abdullah University of Science and Technology
2019
ABSTRACT

High Performance Polar Decomposition on Manycore Systems and its application to Symmetric Eigensolvers and the Singular Value Decomposition

Dalal Sukkari

The Polar Decomposition (PD) of a dense matrix is an important operation in linear algebra, while being a building block for solving the Symmetric Eigenvalue Problem (SEP) and computing the Singular Value Decomposition (SVD). It can be directly calculated through the SVD itself, or iteratively using the $QR$ Dynamically-Weighted Halley (QDWH) algorithm. The former is difficult to parallelize due to the preponderant number of memory-bound operations during the bidiagonal reduction. The latter is an iterative method, which performs more floating-point operations than the SVD approach, but exposes at the same time more parallelism. Looking at the roadmap of the hardware technology scaling, algorithms performing floating-point operations on locally cached data should be favored over those requiring expensive horizontal data movement. In this context, this thesis investigates new high-performance algorithmic designs of QDWH algorithm to compute the PD. Originally introduced by Nakatsukasa et al. [1, 2], our algorithmic contributions include mixed precision techniques, task-based formulations, and parallel asynchronous executions. Moreover, by making the PD competitive, its application to the SEP and the SVD becomes practical. In particular, we introduce for the first time new algorithms for partial SVD decomposition using QDWH. By the same token, we extend the QDWH to support partial eigen decomposition for SEP. We present new high-performance implementations of the QDWH-based algorithms relying on fine-grained computations, which allows exploiting the sparsity
of the underlying data structure. To demonstrate performance efficiency, portability and scalability, we conduct benchmarking campaigns on some of the latest shared/distributed-memory systems. Our QDWH-based algorithm implementations outperform the state-of-the-art numerical libraries by up to 2.8x and 12x on shared and distributed-memory, respectively. The task-based QDWH has been integrated into the Chameleon library (https://gitlab.inria.fr/solverstack/chameleon) for support on shared-memory systems with hardware accelerators. It is also currently being used by astronomers from the Subaru telescope located at the summit of Mauna Kea, Hawaii, USA. The distributed-memory software library of QDWH and its SVD extension are freely available under modified-BSD license at https://github.com/ecrc/qdwh.git and https://github.com/ecrc/ksvd.git, respectively. Both software libraries have been integrated into the Cray Scientific numerical library LibSci v17.11.1 and v19.02.1.
ACKNOWLEDGEMENTS

I am incredibly grateful for those who stood by to support me during my Ph.D. journey. I would like to express my sincere gratitude to my advisor Prof. David Keyes for his encouragement, immense knowledge, guidance, and inspirations. His mesmerizing knowledge, wisdom and virtues cannot be put in words. His advice on both research as well as on my career have been invaluable. Heartly gratefulness is due to my main mentor Dr. Hatem Ltaief for his tremendous help, time, dedication, and patience. There are no words to express my sincere appreciation, most of my thesis ideas were suggested, supported by him. His enormous efforts bring this thesis to its success. Actually, the words can not express how grateful I am ...

I also want to thank my committee members for the brilliant comments and suggestions, and for letting my defense be an enjoyable moment.

I thank my colleagues at ECRC for insightful discussions. Special thanks to the members of HiCMA project for the collaborative and team-work. Many thanks to the members of the KSL lab for their readiness to support and help. Great thanks to Dr. Mustafa Abdul Jabbar for his enormous help, advice and encouragements. Also, sincere thanks to Dr. Mohammed Al Farhan for his support and help.

Highest appreciation goes to my father, for his love and support. His continuous prayers for me was what sustained me thus far. Equal thanks to my mother, may she rest in peace, whose memory and values gave me reasons to stay determined and achieve more.

Utmost gratefulness and recognition are for my husband and lifetime friend, Ayman, I am truly thankful for his love, support, and patience. My husband, along with my lovely sons (Wael, Wesam) have been my source of joy and for whom I present this work, hoping it would be an inspiration for them to pursue excellence in their studies.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examination Committee Page</td>
<td>2</td>
</tr>
<tr>
<td>Copyright</td>
<td>3</td>
</tr>
<tr>
<td>Abstract</td>
<td>4</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>6</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>10</td>
</tr>
<tr>
<td>List of Figures</td>
<td>11</td>
</tr>
<tr>
<td>List of Tables</td>
<td>14</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>15</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>15</td>
</tr>
<tr>
<td>1.2 Objectives and Contributions</td>
<td>16</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>19</td>
</tr>
<tr>
<td><strong>2 Dense Linear Algebra Algorithms</strong></td>
<td>20</td>
</tr>
<tr>
<td>2.1 Block Algorithms</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Tile Algorithms</td>
<td>23</td>
</tr>
<tr>
<td><strong>3 The Polar Decomposition Algorithms</strong></td>
<td>27</td>
</tr>
<tr>
<td>3.1 Background</td>
<td>27</td>
</tr>
<tr>
<td>3.2 The QDWH-based Polar Decomposition</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1 Deriving the formulas for ((a_k, b_k, c_k))</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2 Matrix Inversion Free QDWH</td>
<td>30</td>
</tr>
<tr>
<td>3.3 The ZOLO-PD Polar Decomposition</td>
<td>31</td>
</tr>
<tr>
<td>3.4 Algorithmic Complexity and Memory Footprint</td>
<td>33</td>
</tr>
<tr>
<td><strong>4 Application to SEP/SVD</strong></td>
<td>36</td>
</tr>
<tr>
<td>4.1 The Standard SEP/SVD</td>
<td>36</td>
</tr>
</tbody>
</table>
7 Massively Parallel ZOLO-PD on Distributed-Memory Systems

7.1 Numerical Accuracy

7.1.1 Environment Settings

7.2 Performance Comparisons of QDWH and ZOLO-PD

7.3 Performance Profiling

7.4 Performance Scalability

7.5 Discussion on Energy Efficiency

8 QDWHpartial-SVD as an Enabler for Extreme Adaptive Optics

8.1 Introduction

8.1.1 Adaptive Optics

8.1.2 Exoplanet Imaging, Search for Extraterrestrial Life and Extreme AO

8.1.3 Extreme Computing Needs

8.2 Extreme AO’s Extreme Computing Challenge

8.2.1 Real-time Computations

8.2.1.1 Predictive Control

8.2.1.2 Sensor Fusion

8.2.2 Inferring Control Matrices from Measurements

8.2.3 The SVD-Based Pseudo-Inverse Approach

8.3 The Pseudo-Inverse Algorithm

8.3.1 The Pseudo-Inverse Based on the Symmetric Eigenvalue Decomposition

8.3.2 The Pseudo-Inverse Based on the Standard SVD

8.3.3 The Pseudo-Inverse Based on the QDWH-based Partial SVD

8.3.4 Algorithmic Complexity

8.4 High Performance GPU-based Implementation

8.5 Experimental Results and Analysis

8.5.1 Environment Settings

8.5.2 Definitions of Test Matrices

8.5.3 Numerical Accuracy

8.5.4 Performance Results

9 Summary and Future Work

References
LIST OF ACRONYMS

**BLAS** Basic Linear Algebra Subroutines. 22

**DAG** Directed Acyclic Graph. 5, 18, 25

**DLA** Dense Linear Algebra. 25

**HeFT** Heterogeneous First Time. 27

**PD** Polar Decomposition. 15

**QDWH** QR Dynamically-Weighted Halley. 4, 16, 17

**SBR** Successive Band Reductions. 37

**SEP** Symmetric Eigenvalue Problem. 4, 16, 36, 132

**STF** Sequential Task Flow. 27

**SVD** Singular Value Decomposition. 4, 16, 36
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>LAPACK fork-join programming model.</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>Data layout translation.</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>QDWH-EIG: the matrix $A_{i,j}$ corresponds to the submatrix indexed $j$ at</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>the $i$th level of recursion.</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>A type (9, 8) rational function $y = r(x - s)$ that maps the interval</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>$[-1, 0]$ to $-1$. $s = 0.5$.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Semilog plot of $r(x) + 1$ for different choices of $s$, for Zolotarev</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>functions of type (9, 8) (left) and (15, 14).</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>A type (5, 5) rational function that maps the interval $[1/3, 1]$ to 1.</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Assessing the numerical accuracy/robustness of the task-based QDWH.</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Orthogonality of the left (a-c-e) and right (b-d-f) singular vectors.</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Accuracy of the singular values (a-c-e) and backward error of the overall</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>SVD (b-d-f).</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Accuracy of the mixed precision MAGMA-QDWH-SVD solver. SiDj means i</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>iterations are performed in single precision and the remaining</td>
<td></td>
</tr>
<tr>
<td></td>
<td>j iterations in double precision.</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Accuracy comparison of SVD solvers using 128x288 grid configuration:</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>(a-b-c) on Shaheen-2 and (d-e-f) on KNL system.</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Assessing the numerical accuracy/robustness using 16x36 grid topol-</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>ogy: (a-b-c) for SEP solvers and (d-e-f) for SVD solvers.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Assessing the performance of various incremental optimizations.</td>
<td>73</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparison of synchronous vs asynchronous execution traces of task-</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>based QDWH on the KNL system with a matrix size of 10240.</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Assessing the task-based QDWH scalability.</td>
<td>75</td>
</tr>
<tr>
<td>6.4</td>
<td>Assessing task-based QDWH performance against other QDWH variant</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>implementations on ill (left) and well (right) conditioned matrices.</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Task-based QDWH performance across various architectures.</td>
<td>78</td>
</tr>
</tbody>
</table>
6.6 Performance impact of square and rectangular grid topology \((P \times Q)\) for ill-conditioned matrix on 1152 nodes. 80
6.7 Performance impact of square and rectangular grid topology \((P \times Q)\) for well-conditioned matrix on 1152 nodes. 80
6.8 Performance scalability of QDWH in Tflop/s. 81
6.9 Performance scalability of QDWH in time (s). 82
6.10 Performance comparison of QDWH implementations for ill-conditioned matrix on different numbers of computational nodes. 85
6.11 Performance comparison of QDWH implementations for well-conditioned matrix on different numbers of computational nodes. 86
6.12 Ratios of the communication time between square and rectangular grid topologies for different matrix sizes. 87
6.13 Number of MPI\_Send calls and corresponding amount of bytes transferred per MPI rank as a function of matrix size \(n\). Data shown for square \((P = 192, Q = 192)\) and rectangular \((P = 128, Q = 288)\) communication layouts. For (a)-(d), the data is given for the four most time consuming ScALAPACK routines PDGEQRF, PDPOSV, PDORGQR, and PDORMQR calling MPI\_Send as well as for the sum of these four routines. The total number of MPI\_Send calls per rank for all routines is also plotted as a reference. Subfigures (e) and (f) show again the total number of MPI\_Send calls as stacked bars, where the bars are partitioned according to the number of calls with message sizes in certain ranges and are appropriately colored. 92
6.14 Number of MPI\_Reduce and MPI\_Bcast calls and corresponding amount of bytes transferred per MPI rank as a function of matrix size \(n\). Data shown for square \((P = 192, Q = 192)\) and rectangular \((P = 128, Q = 288)\) communication layouts. For (a)-(d), the data is given for the four most time consuming ScALAPACK routines PDGEQRF, PDPOSV, PDORGQR, and PDORQQR, where the number of calls to MPI\_Reduce and MPI\_Bcast are summed up, as well as for the sum of these four routines. The total number of MPI\_Reduce and MPI\_Bcast calls per rank for all routines is also plotted as a reference as well as the sum of these two values (total). Subfigures (e) and (f) show again the total number of MPI\_Reduce calls as stacked bars, where the bars are partitioned according to the number of calls with message sizes in certain ranges and are appropriately colored. 93
6.15 Performance of both symmetric eigensolvers. 95
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.16</td>
<td>Performance of various QDWH-SVD implementations</td>
<td>95</td>
</tr>
<tr>
<td>6.17</td>
<td>Performance comparisons of MAGMA-QDWH-SVD (GPU) against Intel MKL (CPU) for $\Sigma$ (a-c-e) and additionally $U\Sigma V^\top$ (b-d-f)</td>
<td>97</td>
</tr>
<tr>
<td>6.18</td>
<td>Performance comparisons of MAGMA SVD solvers (GPU) for $\Sigma$ (a-c-e) and additionally $U\Sigma V^\top$ (b-d-f)</td>
<td>98</td>
</tr>
<tr>
<td>6.19</td>
<td>Performance comparisons ((\Sigma) only) using mixed precision techniques. SiDj means i iterations are performed in single precision and the remaining j iterations in double precision.</td>
<td>99</td>
</tr>
<tr>
<td>6.20</td>
<td>Profiling the computational stages of MAGMA-QDWH-SVD</td>
<td>100</td>
</tr>
<tr>
<td>6.21</td>
<td>Performance scalability of MAGMA-QDWH-SVD on multiple GPUs</td>
<td>100</td>
</tr>
<tr>
<td>6.22</td>
<td>Performance comparison of SVD solvers on different process grid configurations</td>
<td>101</td>
</tr>
<tr>
<td>6.23</td>
<td>Performance comparison of SVD solvers on different process grid configurations</td>
<td>102</td>
</tr>
<tr>
<td>6.24</td>
<td>Performance comparison of symmetric EIG solvers on different process grid configurations (in time).</td>
<td>103</td>
</tr>
<tr>
<td>6.25</td>
<td>Performance comparison of symmetric EIG solvers on different process grid configurations (in Tflop/s).</td>
<td>104</td>
</tr>
<tr>
<td>6.26</td>
<td>Performance comparison of symmetric EIG solvers.</td>
<td>104</td>
</tr>
<tr>
<td>7.1</td>
<td>Assessing the numerical accuracy/robustness of QDWH and ZOLO-PD on Shaheen-2</td>
<td>107</td>
</tr>
<tr>
<td>7.2</td>
<td>Assessing the numerical accuracy/robustness of QDWH and ZOLO-PD on Crystal</td>
<td>107</td>
</tr>
<tr>
<td>7.3</td>
<td>Performance comparisons on Shaheen-2 (a-b-c) and Crystal (d-e-f)</td>
<td>109</td>
</tr>
<tr>
<td>7.4</td>
<td>Performance speedup ZOLO-PD versus QDWH on Shaheen-2.</td>
<td>109</td>
</tr>
<tr>
<td>7.5</td>
<td>Profiling QDWH and ZOLO-PD on Shaheen-2.</td>
<td>112</td>
</tr>
<tr>
<td>7.6</td>
<td>Performance scalability study of QDWH and ZOLO-PD on Shaheen-2.</td>
<td>112</td>
</tr>
<tr>
<td>8.2</td>
<td>Assessing the numerical accuracy/robustness of QDWHpartial-SVD on K80 using using synthetic ill-conditioned matrices (a, c, e) and real observational datasets (b, d, f)</td>
<td>132</td>
</tr>
<tr>
<td>8.3</td>
<td>Performance comparisons in Gflops/s of QDWHpartial-SVD using synthetic ill-conditioned matrices (a, b, c) and real observational datasets (d)</td>
<td>134</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Algorithmic complexity and memory footprint for various PD algorithms with $\kappa_2(A) = 10^{12}$.</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>Algorithmic complexity of the QDWH-based polar decomposition.</td>
<td>72</td>
</tr>
<tr>
<td>6.2</td>
<td>Ratios between communication and computation time for the QDWH experiments with square $(P = 192, Q = 192)$ and rectangular $(P = 128, Q = 288)$ grid topologies, extracted from Figures 6.6 and 6.7, for the largest matrix size $n = 122880$. Data is given for well and ill-conditioned matrices on both systems.</td>
<td>87</td>
</tr>
<tr>
<td>6.3</td>
<td>Sustained performance in Tflops/s and percentage of theoretical peak performance including and excluding the time spent for communication for the cases shown in Table 6.2.</td>
<td>88</td>
</tr>
<tr>
<td>8.1</td>
<td>Complexity for various pseudo-inverse algorithms.</td>
<td>128</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Semiconductor manufacturers are designing future manycore chips to be throughput-oriented and massively parallel, highly populated by floating-point units (e.g., x86 architectures with next generation AVX-512 instruction sets for Intel Xeon processors) at the price of increasingly limited bandwidth per core [3]. Therefore, reducing data movement is one of the most challenging tasks faced by application developers, due to the increasing hardware complexity and the lack of expressivity for data locality in the current programming models [4]. As a consequence, when it comes to high performance parallel implementations of numerical algorithms, flops are often no longer a sound proxy for execution time or energy expenditure, as long as data motion is confined. And as highlighted in the International Exascale Software Project [3], to exploit opportunities for energy-efficient performance, algorithms must be designed to boost concurrency and reduce data motion. Given this challenging mandate, we propose new novel algorithmic design and implementations of the Polar Decomposition (PD) for dense matrices, a fundamental matrix decomposition revealing the nearest orthogonal matrix [5, Ch. 8]. It is also used in several applications; in continuum mechanics to decompose stress tensors and to simulate the deformation of an object, in aerospace computations [6] 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 [7] to help understand properties
of electron pair (chemical bond) transferability, etc. Further applications are also reported by Higham in [8]. More recently, the PD appears as the building block for spectral divide-and-conquer algorithms to solve the Symmetric Eigenvalue Problem (SEP) for Hermitian matrices as well as to compute the Singular Value Decomposition (SVD) of a general non-symmetric matrix. These aforementioned matrix decompositions represent some of the main algorithmic workhorses required for a broad class of scientific applications [9, 10, 11]. Originally introduced by Nakatsukasa et al. [1, 2], the current PD employs the inverse-free, iterative QDWH algorithm. QDWH relies on conventional dense linear algebra operations such as $QR$/Cholesky factorizations and matrix-matrix operations, Level-3 BLAS compute-bound kernels, which are all widely available in vendors’ optimized numerical libraries. QDWH-based algorithms exhibit overheads in terms of the number of extra floating-point operations compared to their counterpart in standard algorithms, as implemented in LAPACK [12]. For instance, its SVD solver variant may perform up to three times more floating-point operations (flops) than the standard SVD solver based on reduction to bidiagonal form. However, QDWH-based algorithms are inherently designed for reducing data movement and for relying on computationally intensive numerical kernels. Therefore, these extra flops may be compensated by very efficient and high concurrency numerical kernel executions. All in all, these QDWH-based algorithms stress the floating-point units of the underlying hardware, while performing flops on locally cached data.

1.2 Objectives and Contributions

This thesis describes the design of novel algorithms for the QDWH iterations to efficiently compute the PD of dense matrices on manycore architectures. We then leverage the high performance QDWH implementation and use it a preprocessing toward solving the SEP (QDWH-EIG) and computing the SVD (QDWH-SVD), as introduced by Nakatsukasa et al. [2]. We also revisit a higher-order variant of QDWH [13] for
the PD, which uses the best rational approximation to the sign function (Zolotarev functions) derived by Zolotarev (ZOLO-PD) \([14]\). This 150 year-old algorithm has a convergence rate up to seventeen, more than five times the cubic convergence rate observed for QDWH. However, this comes at the cost of even higher arithmetic complexity and memory footprint than QDWH for the PD. These extra flops may however be processed in an embarrassingly parallel fashion. This naturally strong scaling algorithm is therefore well-suited for exploiting massively parallel systems.

Furthermore, we propose a novel algorithmic design to provide partial spectrum computations of SEP/SVD (QDWH\text{partial-SEP/SVD}) using a light-weight variant of QDWH. The QDWH\text{partial-SEP/SVD} algorithm translates the original matrix problem into a reduced problem size containing only the eigen/singular values of interest, cutting down the original prohibitive arithmetic complexity.

We accelerate the QDWH and its application to compute the SVD (QDWH-SVD) algorithms on shared-memory systems equipped with GPUs. We limit and hide data transfers between the host and the device in favor of \textit{in situ} GPU processing. We integrate a mixed precision technique into the original QDWH algorithm, to enable a tradeoff between accuracy and time complexity. The resulting high performance GPU-based QDWH-SVD implementation outperforms by up to four-fold, and up to three-fold for asymptotic random matrix sizes, the equivalent routines (\texttt{DGESVD}) from existing state-of-the-art commercial (Intel MKL \([15]\)) and open-source (MAGMA \([16]\)) libraries, respectively. To increase concurrency, we introduce a new task-based formulation of QDWH to mitigate the overhead of the bulk synchronous programming model. By relying on dynamic runtime systems, i.e., StarPU \([17]\), this new fine-grained formulation maximizes the hardware utilization, while improving the user productivity thanks to an oblivious data management. The fine-grained computation methodology gives the flexibility to exploit the sparse data structure of the matrix involved in the QDWH iterations. The task-based QDWH can be then rep-
resented as a Directed Acyclic Graph (DAG), where nodes represent computational tasks and edges define the inter-task data dependencies. StarPU is employed to orchestrate the scheduling of the various nodes of the DAG, to track the data dependencies and to asynchronously execute the computational tasks on the underlying hardware resources. StarPU takes advantage of look-ahead opportunities, which may lead to synchronization reducing situations during the task flow of execution. The task-based QDWH implementation for the PD outperforms by up to 2.8X existing state-of-the-art commercial (Intel MKL). To extend the QDWH-based algorithms on distributed-memory systems, we rely on the two-dimensional block cyclic data distribution to ensure load balancing as well as to reduce horizontal communication across the network interconnect. While most dense linear algebra algorithms on distributed-memory systems, including the High Performance Linpack benchmark [18], take advantage of this communication-reducing data layout, the processor grid topology remains an important factor to tune accordingly, depending on the considered workload type. We report performance using wide range of grid configurations on various large distributed-memory systems. In particular, the QDWH-SVD is able to achieve up to 8-fold speedup against its equivalent routine from ScaLAPACK [19].

Moreover, we develop the first high performance implementation of the QDWHpartial-SVD to support more economical partial spectrum requirements. It is noteworthy that there does not exist any support for the calculation of a subset of the singular values/vectors in the SVD solver software literature. Our QDWHpartial-SVD implementation achieves gains up to 6X against the standard distributed-memory SVD solver. Last but not least, we demonstrate the accuracy and effectiveness of the new QDWHpartial-SVD implementation in a real computational astronomy setting. Driven by real-time constraints to correct telescopic images from atmospheric aberrations in the search for exoplanets, the Subaru telescope [20] employs our fast QDWHpartial-SVD algorithm to cope with the changing pace of the atmospheric
1.3 Thesis Outline

Here we describe the organization of this document. In Chapter 2, we provide the necessary background about the necessary algorithmic paradigm shift for embracing manycore architectures. Chapter 3 presents the two incremental approaches to compute the PD, i.e., QDWH and ZOLO-PD, and compares their algorithmic complexities. Chapter 4 highlights the applications of the PD toward solving the SEP and computing the SVD. After reviewing the standard algorithms for the spectral decompositions and SVD, the chapter describes the divide-and-conquer symmetric eigensolver QDWH-EIG and QDWH-SVD. It also introduces the novel QDWH-based algorithm to calculate the economical partial spectrum (QDWHpartial-EIG/SVD). Chapter 5 introduces our high-performance implementations of the QDWH-based PD and its application to the SVD as well as partial spectrum computations. It describes our environment settings and presents the different shared/distributed-memory systems used to evaluate our various algorithmic designs and implementations, and presents the accuracy assessment of the different implementations against the state-of-the-art approaches and demonstrates its numerical robustness in practice.

Chapter 6 shows the benchmarking campaigns on different hardware architectures to evaluate our aforementioned QDWH implementations. Chapter 7 presents an accuracy and performance study of ZOLO-PD against QDWH-based algorithm to compute the polar decomposition. Chapter 8 describes how the QDWHpartial-SVD implementation drives the real-time simulations of the ground-based Subaru telescope in the context of a computational astronomy application. The summary and future work are given in Chapter 9.
Chapter 2

Dense Linear Algebra Algorithms

Basic Linear Algebra Subroutines (BLAS) are categorized into three levels:

• Vector-vector operations, known as Level-1 BLAS, which are mostly of $O(n)$ complexity, operating on $O(n)$ data, and, thus, are memory-bound operations, having low arithmetic intensity.

• Matrix-vector operations, known as Level-2 BLAS, which are mostly of $O(n^2)$ complexity, operating on $O(n^2)$ data, and, thus, are also memory bound operations, having low arithmetic intensity.

• Matrix-matrix operations, known as Level-3 BLAS, which are mostly of $O(n^3)$ complexity, operating on $O(n^2)$ data, having high arithmetic intensity, thus, are mostly compute-bound operations.

For performance purposes, the high arithmetic intensity of Level-3 BLAS (low surface to volume ratio) makes it more favorable than Level 1 and 2. Therefore, parallelizing Level-3 BLAS with multiple threads in a threaded environment setting is more efficient than Level-1 and 2. Furthermore, the general matrix-matrix multiply (GEMM) is an embarrassingly parallel numerical kernel on various hardware architectures since each element of the product is independent. These BLAS routines are building blocks for the high-level Dense Linear Algebra (DLA) algorithms known as LAPACK routines [12].

LAPACK routines are designed so that the major portion of the computation be performed by calls to the BLAS routines, especially the Level-3 routines, to attain
high performance, rather than a naive serial implementation. Additionally, this provides a better memory access pattern of the multi-layered memory hierarchies of the multicore machines. The LAPACK routines implementation have witnessed an algorithmic paradigm shift in response to hardware evolution, moving from block to tile algorithms, the latter paradigm necessitating dynamic runtime system.

![Figure 2.1: LAPACK fork-join programming model.](image)

2.1 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 classic examples of the fork-join paradigm, as illustrated in Figure 2.1 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 parallel (BSP) paradigm corresponds to the backbone of many open-source and commercial state-of-the-art numerical libraries such as LAPACK [12], MAGMA [21] and ScaLAPACK [19] for shared-memory, accelerator-based and distributed-memory systems, respectively.

The MAGMA Library The Matrix Algebra on GPU and Multicore Architectures
(MAGMA) aims to provide a functionality similar to LAPACK but on hybrid multi-core and manycore architectures. However, the MAGMA library, which employs the BSP fork-join model, alleviates the global synchronization without changing the LAPACK numerical algorithms.

**The ScaLAPACK Library** ScaLAPACK relies on block algorithms, similar to LAPACK, but is designed for distributed-memory systems. ScaLAPACK also extracts its parallel performance during the update of the trailing submatrix by means of calls to Level-3 BLAS, as implemented in the Parallel BLAS (PBLAS) layer. PBLAS calls translate into on-node BLAS calls, usually supported by high performance implementations from vendors, followed by parallel reductions. Moreover, ScaLAPACK uses a two-dimensional (2D) block cyclic data distribution associated with a 2D processor grid topology to map the matrix data to the distributed-memory nodes, while ensuring load balancing. At the higher level of the library, ScaLAPACK employs the BSP model and uses the Message Passing Interface (MPI) to move data between remote processing units. In particular, the Basic Linear Algebra Communication Subprograms (BLACS) layer is in charge of setting up the MPI send/receive communication functions during the matrix computations. Furthermore, the processor grid topology needs to be appropriately dimensioned. The panel factorization is typically sequential and may not benefit from having many processors participating in this computational phase. However, one should properly calibrate the number of processors to carry on, in parallel, the update of the trailing submatrix.

As highlighted in the exascale software roadmap, which summarizes the HPC community consensus on an urgent call for sustainable software development for extreme scale, the BSP model may need to be reconsidered, especially in presence of millions of cores, already present in the leading supercomputers at the time of submission of this thesis.
2.2 Tile Algorithms

To answer this call for action and provide a solution for the challenge brought by the manycore era, the Dense Linear Algebra (DLA) community 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 column-major format, which is the standard for block algorithms, as shown in Figure 2.2. The various matrix operations can then be represented as a 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 opportunities for look-ahead, where subsequent tasks may already have their data dependencies satisfied and be ready for execution. In return, this can be exploited by dynamic runtime systems in keeping threads in a busy state throughout the entire execution. The performance gain of block versus tile algorithms has been thoroughly addressed in the literature \cite{23, 24, 25}, in the context of PLASMA \cite{26}, Chameleon \cite{27} and FLAME \cite{28} numerical software libraries.

The PLASMA Library The Parallel Linear Algebra Software for Multicore Architectures (PLASMA) is a numerical software library for solving problems in dense linear algebra on systems of multicore processors and multi-socket systems of mul-
ticore processors. PLASMA is designed to achieve performance greatly exceeding that of LAPACK, by bringing to the fore the parallelism from LAPACK currently residing only inside the BLAS function calls. It reformulates the standard column-major matrix computations so that it can operate on tiles, in which all elements are now contiguous in memory. Executing the tasks in PLAMSA occurs concurrently, while ensuring data dependencies are not violated. PLASMA provides three APIs for performing matrix operations: 1) the naive interface (\texttt{PLASMA\_xxxxx}), where the input and the output matrix are in column-major layout, the layout translation back and forth happens internally. However, this translation generates overheads and is not adequate when extensively calling matrix operations. 2) The tile interface (\texttt{PLASMA\_Tile\_xxxxx}), this interface necessitate the input matrix to be already in tile data layout, and therefore removes the data translations. However, an explicit synchronization or barrier after the completion of each matrix operation is required. 3) The asynchronous interface (\texttt{PLASMA\_Tile\_Async\_xxxxx}), is the most advanced interface. Where, the subsequent API calls can proceed concurrently, without waiting for the previous matrix operation completion, as long as data dependencies are not violated.

**The Chameleon Library** More recently, in a community effort to enhance user productivity by abstracting the hardware complexity, the Chameleon library \cite{27} 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 \cite{29}, OmpSs \cite{30,31,32}, QUARK \cite{33}, StarPU \cite{17,34}, PaRSEC \cite{35}, SuperMatrix \cite{36}) 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 hardware/runtime-oblivious software infrastructure has been already used with StarPU \cite{37}, and more recently with OmpSs \cite{38}, in the context of computational astronomy applications. Similar to the PLASMA library,
Chameleon library has three interfaces; naive, tile and async.

**The StarPU Dynamic Runtime System** Dynamic runtime systems are critical scheduling engines in supporting task-based programming models at large scale [39]. In particular, StarPU [17] is the *de facto* dynamic runtime system for Chameleon. StarPU deals with the execution of generic task graphs, 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 directions (i.e., read, write, read-write). StarPU is then in charge of dynamically scheduling the tasks while enforcing those dependencies. Although Chameleon supports other runtimes (e.g., PaRSEC [35], QUARK [33]), we decided to solely rely on the StarPU 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.

One of the main advantages of using the task-based implementation is to become oblivious to architecture. This improves user productivity, and it is even more realistic for runtimes such as StarPU, which are able to transparently handle single heterogeneous nodes, and eventually multiple heterogeneous nodes in case the StarPU-MPI [34] 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, etc. 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 (HeFT) [40] scheduling strategy used by StarPU, when accelerators are involved in the computations.

Further benefits to using such programming models are the capabilities offered to the programmer to submit simultaneously independent steps of an application. This
permits to raise the resource occupancy, and adds a single synchronization point when all steps are performed. The `CHAMELEON_xxxx_Tile_Async` interface of the Chameleon library offers this capability to interleave multiple dense linear algebra operations when it is possible. Conversely, the synchronous interface, `CHAMELEON_xxxx_Tile`, enforces a synchronization call at the end of the function to wait for the end of all submitted tasks.
Chapter 3

The Polar Decomposition Algorithms

3.1 Background

The polar decomposition is an important numerical algorithm for various applications, including aerospace computations [41] and factor analysis [42]. In this thesis, the polar decomposition is used as a first computational phase toward computing the SVD of a general dense matrix and EVD for symmetric matrix. Moreover, a light-weight version of the polar decomposition is used to compute the most significant singular values/vectors and the negative eigen values/vectors.

The Polar Decomposition (PD) consists in decomposing a dense matrix $A = U_p H$, where $U_p$ is the orthogonal polar factor and $H$ is the positive semi-definite Hermitian polar factor. The PD algorithm has been well studied in the last three decades in terms of complexity and numerical robustness/accuracy [43, 44, 45, 46, 47, 48, 49]. Initially designed with Newton’s method based on an explicit matrix inversion calculation, numerical instability has been reported, especially in the presence of ill-conditioned matrices. An algorithm based on Halley’s iteration has been introduced with an 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 QDWH has been finally proposed by Nakatsukasa et. al [50]. More recently, Nakatsukasa and Higham [2] have shown that QDWH can be used as a building block to solve the SVD and to compute the SVD [10, 11], which has brought to the fore further research directions.
Indeed, in our work, we have implemented QDWH-based singular value decomposition on hardware accelerators [51] and distributed-memory systems [52], where the calculation of the polar factor is the most-time consuming phase. The aforementioned implementations have demonstrated limited performance scalability on multiple GPUs and large clusters. This is mostly due to the low hardware resource occupancy achieved by the Bulk Synchronous Parallel (BSP) model, which both implementations rely on for parallel performance. By the same token, it is also noteworthy that the high performance software library Elemental [53] provides a QDWH implementation for distributed-memory systems. The benefits of applying an asynchronous task-based model to a multi accelerator-based system have been previously described in [31, 32].

The PD can alternatively be computed through an SVD as follows:

\[ A = U \Sigma V^T = UV^T V \Sigma V^T = U_p V \Sigma V^T = U_p H. \]

This strategy has shown some performance scalability issues, due to the slow convergence of the QR algorithm on the condensed bidiagonal form [52].

### 3.2 The QDWH-based Polar Decomposition

To find the polar decomposition, the original Dynamically-Weighted Halley (DWH) iteration can be derived as follows:

\[
U_0 = A/\alpha, \quad U_{k+1} = U_k (a_k I + b_k U_k^T U_k) (I + c_k U_k^T U_k)^{-1}, \tag{3.1}
\]

where \(\alpha = \|A\|_2\). The scalars \((a_k, b_k, c_k)\) are critical parameters since they drive the convergence speed at each iterate.
3.2.1 Deriving the formulas for \((a_k, b_k, c_k)\)

Assuming square matrices, let the SVD of \(U_k = U_k \Sigma_k V_k^\top\), such that \([\sigma_{\min}(U_k), \sigma_{\max}(U_k)] \subseteq [l_k, 1] \subset (0, 1]\). \(\sigma_{\min}(U_0) = \beta/\alpha = l_0\) and \(\beta = 1/\|A^{-1}\|_2\). Hence,

\[
U_{k+1} = U_k \Sigma_k V_k^\top(a_k I + b_k V_k \Sigma^2 V_k^\top)(I + c_k V_k \Sigma^2 V_k^\top)^{-1}
\]

\[
= U_k \Sigma_k(a_k I + b_k \Sigma^2)(I + c_k \Sigma^2)^{-1}V_k^\top
\]

\[
= U_k \Sigma_{k+1} V_k^\top.
\]

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

\[
\sigma_i(U_{k+1}) = g_k(\sigma_i(U_k)),
\]

\[
g_k(x) = x \frac{a_k + b_k x^2}{1 + c_k x^2}, \quad (3.2)
\]

Therefore, \([\sigma_{\min}(U_{k+1}), \sigma_{\max}(U_{k+1})] \subseteq \left[\min_{l_k \leq x \leq 1} g_k(x), \max_{l_k \leq x \leq 1} g_k(x)\right]\). The convergence of \(U_{k+1}\) to the polar factor can be measured by the maximum distance of its singular values and 1. A suboptimal choice of the parameters \((a_k, b_k, c_k)\) should make \(g_k\) bounded and satisfy the max-min \(\max_{a_k, b_k, c_k} \left\{ \min_{l_k \leq x \leq 1} g_k(x) \right\}\). The solution of this max-min problem can be found in [3], which gives the following formulas to calculate \((a_k, b_k, c_k)\):

\[
a_k = h(l_k), \quad b_k = (a_k - 1)^2/4, \quad c_k = a_k + b_k - 1,
\]

\[
l_0 = \frac{\beta}{\alpha}, \quad l_k = \frac{l_{k-1}(a_{k-1} + b_{k-1}l_{k-1}^2)}{1 + c_{k-1}l_{k-1}^2}, \quad k = 1, 2, \ldots, \quad (3.3)
\]

\[
h(l) = \sqrt{1 + d} + \frac{1}{2} \sqrt{8 - 4d + \frac{8(2 - l^2)}{l^2} \sqrt{1 + d}}\quad d = \sqrt{\frac{4(1 - l^2)}{l^4}}.
\]

and \(\alpha = \|A\|_2, \beta = 1/\|A^{-1}\|_2\)

**Convergence within 6 iterations**

The maximum number of DWH iterations for convergence can be calculated by determining the first \(k\) such that \(|1 - l_k| < \epsilon\) (machine epsilon). Assuming double
precision arithmetic with $\epsilon = 10^{-16}$, unrolling Equations 3.3 with $l_0 = 1/\kappa_2(U_0)$ and a fairly large condition number $\kappa_2(U_0) = 10^{16}$, the number of DWH iterations is six. Therefore, DWH converges within at most six iterations for any matrix with condition number $\kappa_2 \leq 10^{16}$.

### 3.2.2 Matrix Inversion Free QDWH

Equation 3.1 can be reformulated using the mathematically equivalent $QR$-based implementation [2]:

$$U_0 = A/\alpha, \quad \alpha = \|A\|_2$$

$$U_{k+1} = \frac{b_k}{c_k} U_k + \frac{1}{\sqrt{c_k}} \left( a_k - \frac{b_k}{c_k} \right) Q_1 Q_2^\top, \quad k \geq 0$$

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-step formula:

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

When $U_k$ becomes well-conditioned, it is possible to replace 3.5 with a Cholesky-based implementation as follows:

$$U_{k+1} = \frac{b_k}{c_k} U_k + \left( a_k - \frac{b_k}{c_k} \right) (U_k W_k^{-1}) W_k^{-\top},$$

$$W_k = \text{chol}(Z_k), \quad Z_k = I + c_k U_k^\top U_k.$$
3.3 The ZOLO-PD Polar Decomposition

The main idea behind the ZOLO-PD algorithm is to generalize the rational approximant underlying the QDWH iterations. Surprisingly, this also results in an opportunity to parallelize the overall PD procedure across iterations. ZOLO-PD was introduced by Nakatsukasa and Freund algorithms.

Once we view the QDWH iterates as a composition of type (3, 2) Zolotarev functions, a natural idea is to use Zolotarev functions of higher type \((2r + 1, 2r)\) for an integer \(r \geq 1\). One can notice that a more general approximation of \(R \in \mathcal{R}_{2r+1,2r}\) with \(r \geq 1\) for the sign function can be considered to map the singular value \(s\) of the orthogonal polar factor to 1. Indeed, since the sign function is an odd function, one can write \(R_{2r+1,2r} = x^{P(x^2)/Q(x^2)}\). Similar to QDWH, we can calculate \(P\) and \(Q\) by solving the following max-min problem:

\[
\max_{P,Q \in \mathbb{R}} \min_{x \in [-1,1]} x^{P(x^2)/Q(x^2)},
\]

where \(x^{P(x^2)/Q(x^2)} \leq 1\) on \([0,1]\). Solving (3.8) is nontrivial, however, an equivalent rational approximation problem was explicitly solved by Zolotarev back in 1877 in terms of Jacobi elliptic functions. To ensure the thesis is self-contained, we briefly recall from [13] how to apply the Zolotarev functions to obtain the solution of (3.8). Solving (3.8) is equivalent to solving the following min-max problem:

\[
\min_{R \in \mathcal{R}_{2r+1,2r}} \max_{x \in [-1,-l] \cup [l,1]} |\text{sign}(x) - R(x)|,
\]

where \(0 < l < 1\) and \(r \geq 0\). As derived by Zolotarev, the type \((2r + 1, 2r)\) Zolotarev function (the best rational approximant to the sign function) on \([-1,-l] \cup [l,1]\) is

\[
Z_{2r+1}(x; \ell) = M x \prod_{j=1}^{r} \frac{x^2 + c_{2j}}{x^2 + c_{2j-1}},
\]

where \(0 < l < 1\) and \(r \geq 0\). As derived by Zolotarev, the type \((2r + 1, 2r)\) Zolotarev function (the best rational approximant to the sign function) on \([-1,-l] \cup [l,1]\) is
where $M > 0$ and satisfies the following:

$$1 - Z_{2r+1}(1; \ell) = -(1 - Z_{2r+1}(l; \ell)).$$

The scalars $c_1, c_2, \ldots, c_{2r}$ can be computed using the Jacobi elliptic functions $sn(u; \ell'), cn(u; \ell')$. Evaluating $Z_{2r+1}(x; \ell)$ at a matrix argument $X$ to obtain $UZ_{2r+1}(\Sigma; \ell)V^*$ where $X = U\Sigma V^*$ is the SVD can be done by

$$Z_{2r+1}(X; \ell) = MX\prod_{j=1}^r (X^*X + c_{2j}I)(X^*X + c_{2j-1}I)^{-1}.$$

We rewrite this in partial fraction form, to improve the degree of parallelism: we can find $a_j \in \mathbb{R}$ such that

$$Z_{2r+1}(X; \ell) = M(X + \sum_{j=1}^r a_j X(X^*X + c_{2j-1}I)^{-1}). \quad (3.11)$$

Again, $Z_{2r+1}(X; \ell)$ in (3.11) can be computed stably as

$$Z_{2r+1}(X; \ell) = X + \sum_{j=1}^r \frac{a_j}{\sqrt{c_{2j-1}}} Q_{j1}Q_{j2}^*$.

This represents a number $r$ of embarrassingly parallel $QR$ factorizations and matrix-matrix multiplications $Q_{j1}Q_{j2}^*$. The upshot of ZOLO-PD is that by taking $r = 8$, we obtain $\|Z_{2r+1}(Z_{2r+1}(A; \ell_0); \ell_1) - U_p\| \leq 10^{-15}$ for any $A$ with $\kappa_2(A) \leq 10^{12}$, implying that convergence is attained in just two steps.

Similarly to the QDWH algorithm, the $QR$ iterations in (3.12) can be reformulated as Cholesky-based iterations with a lower arithmetic cost, as in (3.7), once $X_k$ is well-conditioned. This condition is already satisfied at the second iteration even for
ill-conditioned matrices. In fact, this new ZOLO-PD algorithm converges within two iterations in double precision, for matrices with condition number $\leq 10^{15}$, instead of the original six iterations, as in QDWH. The resulting ZOLO-PD convergence rate is up to seventeen, more than five times the cubic convergence rate observed for QDWH.

All in all, the ZOLO-PD algorithm can be seen as a generalization of QDWH into a series of independent QDWH subproblems with only two iterations per subproblem, instead of the original six iterations, as mentioned in Section 3.2.

### 3.4 Algorithmic Complexity and Memory Footprint

In this section, we compare the algorithmic complexity of the QDWH algorithm against the ZOLO-PD algorithmic variants with successive or independent PD iterations. We consider square matrices $A \in \mathbb{C}^{n \times n}$ for simplicity; the algorithms are directly applicable to rectangular matrices.

The condition number estimate $L_0$ can be calculated using the LU factorization, which requires $\frac{2}{3}n^3$, followed by a few triangular solvers, which cost $O(n^2)$ flops. As shown in (3.5) and (3.12) for QDWH and ZOLO-PD, respectively, the QR-based PD iteration requires the $QR$ factorization of $2n \times n$ matrix for a cost of $(3 + \frac{1}{3})n^3$ flops. Then, forming $[Q_1^* Q_2^*]^*$ explicitly, needs $(3 + \frac{1}{3})n^3$ flops. The product $Q_1Q_2^*$ additionally needs $2n^3$ flops. Therefore, the arithmetic cost of each QR-based iteration is $(8 + \frac{2}{3})n^3$ flops $^1$. For the Cholesky-based PD iteration in (3.7), 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 a Cholesky-based iteration (3.7) is $(3 + \frac{1}{3})n^3$ per iteration. Computing the positive semidefinite matrix $H = U_p^* A$ requires $2n^3$.

$^1$The flop counts here are different from [13] since the counting here does not exploit the symmetry and the identity structure in the bottom block.
Hence, the overall cost of QDWH is

$$\#flops = \frac{2}{3}n^3 + (8 + \frac{2}{3})n^3 \times \#it_{QR}$$
$$+ (3 + \frac{1}{3})n^3 \times \#it_{Chol}$$
$$+ 2n^3,$$

where $\#it_{QR}$ and $\#it_{Chol}$ correspond to the number of QR-based and Cholesky-based iterations, respectively.

The cost of ZOLO-PD is

$$\frac{2}{3}n^3 + (8 + \frac{2}{3})n^3 \times r + (3 + \frac{1}{3})n^3 \times r + 2n^3,$$

where the total number of iterations is 2 ($\#it_{QR}=1$ and $\#it_{Chol}=1$), and $r = 1, \ldots, 7, 8$ is the number of independent problems to be solved in an embarrassingly parallel fashion at each iteration, as we mentioned above. As shown in (3.12), ZOLO-PD solves $r$ embarrassing parallel factorizations, and along the critical path, the arithmetic cost of ZOLO-PD is

$$\frac{2}{3}n^3 + (8 + \frac{2}{3})n^3 + (3 + \frac{1}{3})n^3 + 2n^3.$$

For ill-conditioned matrices with condition number $\kappa = 10^{12}$, QDWH requires 2 QR-based iterations followed by 4 Cholesky-based iterations, and ZOLO-PD needs two successive iterations with $r = 8$. As far as memory footprint is concerned, there is a trade-off between degree of parallelism and memory allocation. Executing ZOLO-PD with independent problems obviously requires as many distinct data structures to operate on as the number of problems. The following table summarizes and compares the flop count and memory footprint of QDWH and ZOLO-PD for matrices with $\kappa = 10^{12}$. QDWH performs around 2.2 times more flops than the parallel ZOLO-PD version, but this assumes again that there are enough compute and memory resources.
Table 3.1: Algorithmic complexity and memory footprint for various PD algorithms with $\kappa_2(A) = 10^{12}$.

<table>
<thead>
<tr>
<th></th>
<th>QDWH</th>
<th>Successive ZOLO-PD</th>
<th>Independent ZOLO-PD</th>
</tr>
</thead>
<tbody>
<tr>
<td># QR-based iterations</td>
<td>2</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td># Cholesky-based iterations</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Algorithmic complexity</td>
<td>$33n^3$</td>
<td>$100n^3$</td>
<td>$15n^3$</td>
</tr>
<tr>
<td>Memory footprint</td>
<td>$6n^2$</td>
<td>$6n^2$</td>
<td>$48n^2$</td>
</tr>
</tbody>
</table>

to simultaneously execute the independent problems on the targeted system.

Next chapter recalls the standard SEP and SVD algorithm and describes the QDWH-based application to compute the SEP and SVD.
Chapter 4

Application to SEP/SVD

4.1 The Standard SEP/SVD

The standard SEP/SVD algorithm first reduces a general dense matrix using a one-stage approach to condensed form tridiagonal/bidiagonal, respectively.

Then the symmetric tridiagonal eigenvalue problem can be solved using $QR$ \cite{54}, divide-and-conquer \cite{55} and bisection algorithm \cite{10}. The bidiagonal singular value problem can solved using $QR$ \cite{56,57} or divide-and-conquer \cite{58}, algorithms and, finally, accumulates the subsequent orthogonal transformations if eigen/singular vectors are also required. While the back transformation phase to calculate the corresponding eigen/singular vectors is rich in Level-3 BLAS operations, the reduction to condensed form is often seen as a major bottleneck for parallel performance due to the inefficient Level-2 BLAS kernels predominance.

The authors of the Successive Band Reductions (SBR) software package \cite{59,60} introduced an intermediary computational stage before getting to the final bidiagonal form. Originally applied to the symmetric eigensolver \cite{61,62}, the main ideas have been extended to SVD solver. The matrix is first reduced to band condensed form, where most of the original Level-2 BLAS operations are now cast into Level-3 BLAS, thus increasing the level of concurrency, as highlighted in \cite{63}. The second stage annihilates the extra off-diagonal elements using an efficient bulge chasing technique, until the final condensed structure is obtained. Although operations in the latter stage are memory-bound, data reuse at the high level of caches is possible, thanks to
some locality scheduling heuristics [64]. The standard column-major data format of the matrix has to be translated into a tile data layout. Indeed, the matrix is split into tiles, where elements of a given tile are now contiguous in memory. The algorithm of the EVD/SVD solver operates now on tiles and exposes fine-grained tasks to be scheduled by a dynamic runtime system [16]. The overall tile EVD/SVD solver also generates substantial extra flops [65], especially during the back transformation, in case of eigen/singular vectors are desired. All the aforementioned high performance EVD/SVD solver implementations based on a two-stage matrix reduction run on x86 architecture only (accelerators are currently not supported). There are mainly two reasons for this. Porting the two-stage SVD solver on such complex platforms requires new non-conventional computational kernels and increases software maintenance, while jeopardizing code sustainability across hardware generations. For solving large SVD or even eigenproblems on distributed-memory systems, it is not clear at this moment whether a two-stage reduction is the most efficient, due to the excessive number of communications.

Last but not least, a previous framework for computing the SVD via the polar decomposition and the eigendecomposition has already been presented in [66]. This algorithm requires three building blocks: matrix multiplication, matrix inversion, and solution of the Hermitian eigenproblem.

4.2 Computing the Full Spectrum based on QDWH

The QDWH-based symmetric eigensolver (QDWH-EIG) [2] is a spectral divide-and-conquer algorithm to solve the symmetric eigenvalue problem $A = V \Lambda V^\top$. It is built on the polar decomposition (see previous Chapter [3]) and operates by recursively decoupling the problem into independent subproblems through finding invariant subspaces.
Algorithm 1 outlines the overall QDWH-EIG algorithm. In order to provide a balanced division during the recursions, QDWH-EIG shifts the diagonal of the matrix by an estimate of the median of the eigenvalues. After computing the polar decomposition, the polar factor $U_p$ permits eventually to generate two orthogonal matrices $V_1$ and $V_2$ through a subspace iteration procedure. Applying these orthogonal matrices to the original matrix permits to discriminate among eigenvalues with respect to the shift $\sigma$ and consequently, to split the eigenvalue spectrum across subsequent subproblems. The algorithm proceeds recursively on each new subproblem until the submatrices become diagonalized. Figure 4.1 shows the recursive QDWH-EIG algorithm unrolled with four levels of recursion. The eigenvalues of the eight subproblems (from $A_{4,1}$ to $A_{4,8}$) correspond then to the eigenvalues of the original matrix $A_{1,1}$. Finally, all subsequent orthogonal transformations need to be accumulated in order to generate the eigenvectors corresponding to the eigenvalues.

**Algorithm 1 QDWH-EIG**

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

A framework to compute the SVD $A = U \Sigma V^T$ based on the polar decomposition and the eigensolver has been suggested by Higham and Papadimitriou [67, 68]. The polar decomposition $A = U_p H$ and the eigendecomposition $H = V \Sigma V^T$, then the SVD $A = (U_p V) \Sigma V^T$. The QDWH-based singular value decomposition (QDWH-SVD) [2] follows this framework using QDWH. for the polar decomposition and QDWH-EIG for the symmetric eigensolver. Algorithm 2 outlines the QDWH-SVD
Algorithm 2 QDWH-SVD

1: Compute the polar decomposition $A = U_p H$ by the QDWH algorithm.
2: Compute the eigenvalue decomposition of the positive semi-definite Hermitian polar factor $H = V \Sigma V^\top$.
3: Calculate the left singular vectors $U = U_p V$. The SVD $A = U \Sigma V^\top$

4.3 Computing a Partial Spectrum based on QDWH

We first recall the scheme of QDWH-partial-EIG, an algorithm for computing the negative eigenvalues and its corresponding eigenvectors (of course the algorithm can be modified trivially to find the eigenvalues/vectors smaller/larger than any specific number by shifting and scaling by $-1$). We first recall the mechanism of QDWH-EIG \cite{2} as shown in 4.2 on which QDWH-partial will be based. Let $A$ be an $n \times n$ symmetric matrix and $U_p H$ be the polar decomposition. Let $k$ be the number of negative eigenvalues, which we do not assume to be known. Computing the unitary polar factor $U_p$ means we have mapped all the eigenvalues to $1$ or $-1$. We partition $V = [V_+, V_-]$ conformably with $\Lambda$, and note that

$$\frac{1}{2}(I - U_p) = \frac{1}{2} \begin{pmatrix} I - [V_+ V_-] & \begin{bmatrix} I_{n-1} & 0 \\ 0 & -I_k \end{bmatrix} [V_+ V_-]^* \end{pmatrix} = [V_+ V_-] \begin{pmatrix} 0 \\ 0 & I_k \end{pmatrix} [V_+ V_-]^* = V_+ V_-^*,$$

so the symmetric matrix $C = \frac{1}{2}(I - U_p) = V_- V_-^*$ is an orthogonal projector onto $\text{Span}(V_-)$, the invariant subspace corresponding to the negative eigenvalues. We can then project the matrix $A$ (Rayleigh-Ritz process) to obtain the eigenvalues and eigenvectors: the eigenvalues of $V_-^* A V_-^*$ are equal to those of $\Lambda_-$, and denoting by $V_-^* A V_- = W \Lambda_- W^*$ the eigenvalue decomposition, the $V_- W$ is the matrix of eigenvectors. Analogously, we can obtain $V_+$ by finding the subspace spanned by $\frac{1}{2}(I + U_p)$.

The problem thus reduces to computing $U_p$. The mathematics underlying QDWH-
EIG is rational approximation: it finds a rational function $r$ that approximates the sign function, so that it maps the negative eigenvalues to $-1$, and positive eigenvalues to $1$. Thus $r(A)$ has eigenvalues $\pm 1$. Assuming that we only $V_-$ is needed, then the idea behind the partial computations is to avoid computing $U_p$ and skip computing $(V_-)^*$. Essentially, the negative eigenvalues have to map to $-1$; the positive eigenvalues are irrelevant. Another key observation: we can work with the shifted matrix $\tilde{A} := A - sI$, for $s > 0$. Where is the relevant part is the eigenvalues laying in the negative interval $[\lambda_{\min}(A) - s, -s]$. Note that $A - sI$ generally has more negative eigenvalues than $A$; these we are not interested in. The reason we introduce this shift $s$ is that it is much easier to map the interval $[\lambda_{\min}(A) - s, -s]$ to $-1$, rather than to map $[\lambda_{\min}(A), 0]$ (or $[\lambda_{\min}(A), -\epsilon]$ for $\epsilon = O(\mu)$) to $-1$. Specifically, the type of the rational function can be significantly lower for the former task. This can be done by approximating the sign function $r(x) \approx \text{sign}(x)$ on $[\lambda_{\min}(A) - s, -s] \cup [s, s - \lambda_{\min}(A)]$.

Equivalently, assume $a = \lambda_{\min}(A)$ (or its estimate; recall $a < 0$), and choose a scalar $b > |a|$ such that we take $r$ to approximate $\text{sign}(x - \frac{a+b}{2})$ on $[a, 0] \cup [b+a, b]$. This is equal to approximating $\text{sign}(x)$ on $[a - \frac{a+b}{2}, -\frac{a+b}{2}] \cup [\frac{a+b}{2}, b - a]$ (recall $a < 0$); that is, it corresponds to taking $s = \frac{a+b}{2}$ in the above description. Suppose that we construct a rational function $r$ that maps the interval $[\lambda_{\min}(A) - s, -s]$ to $-1$. Then the matrix function $r(\tilde{A})$ has $k$ (or more) eigenvalues at $-1$, and $n - k$ (or fewer) eigenvalues away from $-1$ whose precise values we do not care. For illustration, Figure 4.2 shows a typical plot of the rational functions $r(x - s)$ (a shifted Zolotarev function $Z(x; \ell)$ with $\ell = 1/3$; see [13] for details) that we employ. We suppose that the required eigenvalues lie in $(-1, 0)$, that is, $\lambda_{\min}(A) \geq -1$. We do require a lower bound for $\lambda_{\min}(A)$ to set the interval; many algorithms are available for this task; we use a few steps of the Lanczos iteration to estimate $\lambda_{\min}(A)$.

Figure 4.2 shows mapping the interval $[-1, 0]$ to $-1$. We also note that there is a nontrivial interval on the positive side (on $[0, \hat{s}]$ for $\hat{s} < s$) that is mapped close to
−1. This is a nature of the Zolotarev function, and can cause the null space of the matrix \( \frac{1}{2}(r(\tilde{A}) + I) \) to be larger than \( k \) (the number of negative eigenvalues). If we take \( s \) too large, that is, when all the eigenvalues of \( A \) are mapped close to \(-1\), then \( \frac{1}{2}(r(\tilde{A}) + I) \) converges to zero, and the process below leads to no efficiency gain (i.e., the whole space becomes the computed null space and no gain is obtained relative to doing a full eigendecomposition).

\[(4.1)\]

\[ r(\tilde{A}) + I = [Q_1, Q_2] \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \]

where the size \( \ell \times \ell \) of \( R_{22} \) is chosen so that it has “small” values; \( Q_2 \) is then the null space of \( A \) that we require. If \( (4.1) \) is a rank-revealing \( QR \) factorization (e.g. [69]), we have \( \|R_{22}\|_2 = c\sigma_{n-\ell+1} \) for a modest constant \( c \). If not, we can compute the \( QR \) factorization of \( (r(\tilde{A}) + I)\Omega \) for a random (Gaussian) matrix \( \Omega \)–it is known that such \( QR \) factorization is rank revealing with high probability. (We can alternatively use pivots, but this incurs substantial overhead in parallel computing).

Figure 4.2: A type (9, 8) rational function \( y = r(x - s) \) that maps the interval \([-1, 0]\) to \(-1\). \( s = 0.5 \).
Once the QR factorization is computed, we set a tolerance \( tol \) (well below 1 but well above machine precision; e.g. \( tol = 10^{-2} \)). We examine the first diagonal element \( R_{ii} \) of \( R \) that comes below \( tol \), and take \( i - 1 \) to be the size of \( R_1 \), that is, \( \ell = n - i + 1 \). The choice of \( tol \) is made to perform a conservative estimate of the null space. A perturbation analysis can be used to show that the computed null space contains the exact one up to \( O(u/tol) \), where \( u \) is the machine precision. If a higher precision is desired, then the \( tol \) should be large, at the cost of the projected size \( \ell \) being large.

Once \( Q_2 \) is computed, we then compute the the eigenvalues of \( Q_2^*A Q_2 \), whose negative eigenvalues should match those of \( \Lambda_- \). Denoting by \( Q_2^*A Q_2 = W \Lambda_- W^* \) the eigenvalue decomposition, the \( Q_2 W \) is the matrix of eigenvectors. To be precise, \( Q_2^*A Q_2 \) usually contains eigenvalues that are positive, and we discard those (since we are looking for extremal eigenvalues, Rayleigh-Ritz is a reliable means to extract the desired subspace); \( W \) is then an \( \ell \times k \) matrix corresponding to the negative eigenvalues.

For some matrices, such as matrices with HODLR structure \cite{2015arXiv151106317S}, the QR factorization may fail to be a rank-revealing QR. However, this issue can be solved either by calling the QR with pivoting, or destroying the structure of the matrix by multiplying the matrix by an arbitrary orthogonal matrix. Even the latter solution requires two extra \texttt{DGEMM}; it is favorable compared to the former one which is a bottleneck in terms of communications.

**Choosing \( s \)** Let us consider in more detail the choice of the shift parameter \( s \). The qualitative behavior has been explained already; taking \( s \) small results in the Zolotarev function being poor approximants to \( \text{sign}(x) \) in the interval \([-1, 0]\) that we care about, while a large \( s \) results in good approximation \( r(x) \approx -1 \) on \([-1, 0]\), but (undesirably) also on a significant positive interval, resulting in the projected size being large.

We illustrate this in Figure 4.3. For example, in the left plot, setting \( s = 0.875 \) gave \( O(u) \) approximation quality on \([-1, 0]\), but the region in which \( r(x) + 1 \) is close
to 0 extends far into the positive axis. Setting \( s = 0.1 \), on the other hand, solves that issue, but the approximation quality on \([-1, 0]\) is evidently worse. A similar behavior is seen on the right plot, where we use a Zolotarev function of higher type. Here the essence stays the same, but a much smaller \( s \) is enough to obtain \( O(u) \) accuracy on \([-1, 0]\), and the functions \( r(x) \) grow steeply until \( \approx 1 \) for \( x > 0 \). Since the goal of the QDWHpartial-EIG iterations is to map the eigenvalues in \([-1, 0]\) to 0 (to working precision \( O(u) \)) by the rational function \( r(x) + 1 \) while keeping the positive eigenvalues well separated from 0, the above observation leads to the following strategy for choosing \( s \):

1. Determine the type \((2m + 1, 2m)\) of rational function \( r(x) \) to be used.

2. Choose the smallest \( s \) so that \( |r(x) + 1| \leq O(u) \) on \([-1, 0]\).

In practice, the type \((2m + 1, 2m)\) is chosen depending on the computational budget, and we shall mainly focus on two values \( 2m + 1 = 3^2 \) and \( 2m + 1 = 3^3 \), as these lend to particularly efficient evaluation, by taking advantage of the optimality of Zolotarev functions under composition \([13]\). Specifically, they correspond to setting two \((2m + 1 = 3^2)\) and three \((2m + 1 = 3^3)\) QDWH iterations. For each choice, the value of \( s \) satisfying the second condition above is found (by simple experiments) to be

Figure 4.3: Semilog plot of \( r(x) + 1 \) for different choices of \( s \), for Zolotarev functions of type \((9, 8)\) (left) and \((15, 14)\).
1. Type (9, 8), two QDWH iterations: \( s = 0.875 \),

2. Type (27, 26), three QDWH iterations: \( s = 0.2 \).

These choices are shown in both plots of Figure 4.3. In most cases, three QDWH iterations is recommended as the overhead is not too much, while the benefit is significant, as can be seen clearly in Figure 4.3: the function \( r(x) + 1 \) of the corresponding cases \( (s = .875 \text{ and } s = 0.2) \) take \( O(1) \approx 2 \) values for \( x \geq 0.2 \) with three QDWH iterations, while with two QDWH iterations, we require \( 0.2 \leq x \leq 5.0 \). The implication is that the projected matrix size will be approximately equal to the number of eigenvalues in \([-1, 0.2]\) with three QDWH, but with two iterations, it captures eigenvalues in \([-1, 5]\), rendering QDWHpartial useless unless there is a significant portion of large and positive eigenvalues in \( A \). Algorithm 3 presents the main computational steps of QDWHpartial-EIG.

**Algorithm 3 QDWHpartial-EIG.**

1: Compute \( a, b \) using Lanczos iterations
2: Shift the matrix \( AA = (A - (a + b)/2 \cdot Id)/(b - a) \cdot 2 \)
3: Compute the polar decomposition \( AA = U_pH \) using QDWH
4: Calculate \( [Q R] = QR(U_p + Id) \)
5: Find the index \( \text{ind} = \min(\text{find}(\text{abs}(\text{diag}(R)) < \text{threshold})) \)
6: Extract \( Q = Q(:, \text{ind} : \text{end}) \)
7: Reduce the original matrix problem \( \tilde{A} = \tilde{Q}^\top A\tilde{Q} \)
8: Compute the SEP of the reduced matrix problem \( \tilde{A} = \tilde{V} \Sigma \tilde{V}^\top \)
9: Compute the right eigenvectors \( V = \tilde{Q}^\top \tilde{V} \)

Essentially the same idea can be applied for the SVD, to compute the singular values and singular vectors corresponding to the singular values above a given threshold. Here we cannot use shifts; the idea is simply to compute (assume w.l.o.g that \( \|A\|_2 = 1 \))

\[
r(A) := Ur(\Sigma)V^*
\]

where \( r \) is a rational function that maps the interval \([s, 1]\) to 1. As before, if \( s \) is well above \( \sigma_{\text{min}}(A) \), then \( r \) is allowed to be of much lower degree than would be needed for computing \( U_p \). See Figure 4.4 for an illustration.
Figure 4.4: A type (5, 5) rational function that maps the interval $[1/3, 1]$ to 1. $s = 1/3$.

Once $r(A)$ is computed, one can find the desired column space $U_1$ (leading columns of $U$) by finding the null space of $I - r(A)^*r(A)$, which we do as before using $QR$ (perhaps with randomization). Algorithm 12 describes the main steps to compute the most significant singular values with their corresponding singular vectors.

\begin{algorithm}
\caption{QDWHpartial-SVD.}
\begin{algorithmic}
\State 1: Compute the polar decomposition $A = U_p H$ using QDWH
\State 2: Calculate $[Q R] = QR(Id - U_p U_p)$
\State 3: Find the index $\text{ind} = \min(\text{find}(\text{abs(diag}(R)) < \text{threshold}))$
\State 4: Extract $\tilde{Q} = Q(:, \text{ind}: end)$
\State 5: Reduce the original matrix problem $\tilde{A} = A\tilde{Q}$
\State 6: Compute the SVD of the reduced matrix problem $\tilde{A} = U\Sigma\tilde{V}^T$
\State 7: Compute the eigen vectors $V = \tilde{Q}\tilde{V}$
\end{algorithmic}
\end{algorithm}

4.4 Arithmetic Complexity

In this section, we present the algorithmic complexity of the standard SEP/SVD and the QDWH-based algorithms. Without loss of generality, square matrices are assumed.

The standard SEP For finding the eigenvalues only, $\frac{4}{3}n^3$ flops are required to reduce the symmetric matrix into a band matrix (with the matrix bandwidth $nb \ll n$) and a lower-order amount for the subsequent bulge-chasing step (bandwidth reduction) and for solving the resulting tridiagonal eigenvalue problem. Solving the tridiagonal eigenvalue problem using divide and conquer algorithm D&C needs $\frac{4}{3}n^3$. 
If two-stage due to the two-stage tridiagonal reduction, recovering the original eigenvectors needs \(4n^3\). This complexity very often overestimates the computational costs of the divide-and-conquer due to significant deflation that is often observed.

**The standard SVD** The standard approach to compute the SVD of a dense matrix is to first reduce it to bidiagonal form \(A = U_1 BV_1^T\). This reduction can be done using Householder reflectors for a cost of \(\frac{8}{3}n^3\) and is a common step for both standard \texttt{DGESVD} and \texttt{DGESDD} functions, as implemented in LAPACK. If only singular values are needed, computing them from the bidiagonal form using the QR (\texttt{DGESVD}) or the divide-and-conquer (\texttt{DGESDD}) algorithms requires \(O(n^2)\) flops. If singular vectors are additionally required from \(B = U_2 \Sigma V_2\), the LAPACK subroutines \texttt{DBDSQR} and \texttt{DBDSDC} implement an iterative method and a recursive approach based on QR algorithm and divide-and-conquer, respectively. The subsequent left and right singular vectors are then accumulated during the back transformation phase, i.e., \(U = U_1 U_2\) 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 \(17n^3\) for either \texttt{DGESVD} or \texttt{DGESDD} [71].

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

The block algorithms (coarse-grained) used in the shared/distributed-memory QDWH-based framework will not permit taking into account the matrix structure of the identity during QDWH iterations and the upper triangular structure of the resulting \(Q_2\) (Equation 3.5).

**QDWH-EIG** The flop count of a single execution of QDWH-EIG includes the cost of the polar decomposition of a symmetric matrix \(A\), the subspace iteration and
forming the submatrices $A_1, A_2$. It is assumed, following [2], that during the QDWH-EIG execution, the shift is always taken so that $A_1$ and $A_2$ are both approximately of dimension $\frac{n}{2}$. Since one spectral division results in two submatrices of size $\approx \frac{n}{2}$ and the arithmetic cost scales cubically with the matrix size, the overall arithmetic cost is approximately $\sum_{i=0}^{\infty} (2\cdot 2^{-3})^i \beta = \frac{4}{3} \beta$, where $\beta$ is the number of flops needed for one run of QDWH-EIG for $n \times n$ matrix. The cost of the invariant subspace is $(3 + \frac{1}{2}) n^3$ for forming the full decomposition $X = [\hat{V}_1 \hat{V}_2] R$. Forming $A_1 = V_1^T A V_1$ and $A_2 = V_2^T A V_2$ requires $(1 + \frac{1}{4}) n^3$ flops for each. The final step of one recursion of QDWH-EIG is to update the eigenvectors $V_1 = V_1 V_{21}$ and $V_2 = V_2 V_{22}$, each requiring $\frac{1}{2} n^3$ flops.

Hence, the total flop count of QDWH-EIG is

$$\frac{4}{3} \beta = \frac{4}{3} ((2 + \frac{2}{3}) + (8 + \frac{2}{3} - \frac{1}{2}) \times \#it_{QR} + (4 + \frac{1}{3} - \frac{1}{2}) \times \#it_{Chol} + (3 + \frac{1}{2}) + (1 + \frac{1}{4}) \times 2 + 1 \times i_{ev}) n^3$$

where, $i_{ev}$ is 1 if the eigenvectors are additionally needed. We note that for practical dense matrices of size sufficiently smaller than $10^5$, we get $l_0 > 10^5$. Hence, the total flop count is $[45 + \frac{7}{5}, (47 + \frac{1}{5}) n^3]$, if [eigenvalues only, eigenvectors additionally] are needed, respectively.

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

$$((2 + \frac{2}{3}) + (8 + \frac{2}{3}) \times \#it_{QR} + (4 + \frac{1}{3}) \times \#it_{Chol} + 2 + (37 + \frac{5}{3} + (1 + \frac{1}{3}) \times i_{rsu}) + 2 \times i_{lsu}) n^3,$$

where, $[i_{rsu}, i_{lsu}]$ correspond to right/left singular vectors.

**QDWHpartial-EIG** The algorithmic complexity of QDWHpartial-EIG depends on the number of QDWH Cholesky-based iterations (typically two or three), and the $QR$ required to find the reduced matrix size ($s$) followed by DGEMM. The actual SEP solver occurs now only on the reduced problem matrix of size $s$. Assuming $s << n$ and three iterations to get the polar factor from QDWH, the total number of operations
is \((4 + 1/3)Nn^3 \times \#it_{Chol} + 4/3n^3 + 2sn^2 + 2ns^2 + 4s^3\).

**QDWHpartial-SVD** Similarly, the flops count of QDWHpartial-SVD consists of the flops of QDWH Cholesky-based iterations (typically two or three), and the QR and DGEMM, and the SVD solver which occurs now only on the reduced problem matrix of size \(s\). The complexity is \((4 + 1/3)Nn^3 \times \#it_{Chol} + 4/3n^3 + 2sn^2 + 2ns^2 + 22s^3\). This is almost equal to \(14n^3\) which is about 66% of the the standard SVD.
Chapter 5

High-Performance Adaptations of QDWH-based Algorithms

Before focusing on the performance results, it is crucial to check on the numerical accuracy and stability of the various implemented algorithms. In this chapter, we describe the design criteria and implementations based on distinct numerical libraries. Moreover, we highlight the robustness of the different algorithmic improvements using variety of hardware architectures. Our codes are written in C programming language, and all computations are performed in double precision arithmetic. All experiments have been run five times and only the minimum time to solution is reported for each test case. Runtimes varied by less than 5%.

5.1 Polar Decomposition

We recall the polar decomposition of a given general matrix \( A \in \mathbb{R}^{n \times n} \): \( A = U_p H \). To assess the orthogonality of the polar factor \( U_p \) and the accuracy of the overall polar decomposition, we use the accuracy metrics defined in [72].

5.1.1 Shared-Memory

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 [27]. Algorithm 5 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 $A$, as in Eq. (3.4) from Section 3.2, that is required to start the iterative process. The two-norm estimator corresponds to the largest singular value of the matrix. It relies on the power iteration, which involves repeated multiplication by the matrix $A$ and $A^T$. The power iteration converges when the difference between two successive estimates fall within the specified relative tolerance. We have introduced \texttt{genm2} in the Chameleon library through an iterative procedure, in which we minimized the number of synchronizations, thanks to fine-grained computations and look-ahead techniques. The second section of the algorithm computes the initial condition number $l_0$ from row 7 to 19, as in Eq. (3.3) from Section 3.2. 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 \texttt{dgecon} by means of those two results. The main challenge here resides in the $LU$ factorization with partial pivoting, which is difficult to implement using task-based programming models. Indeed, searches for pivot candidates and row swapping generate many global synchronization points within the panel factorization and its resulting updates. Some solutions have been proposed on shared-memory systems [73] but there are no existing solutions that are oblivious of heterogeneous architectures. We thus propose 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 $U$, as in Eq. (3.5) or Eq. (3.7) from Section 3.2 respectively. Finally, the last section, rows 49 to 53, computes the Hermitian polar factor $H$ from the polar factor computed out of
the main loop.

**Environment Settings** We have considered three different single node systems, which are representative of the current manycore-based hardware trends. The first system is composed of dual-socket 16-core Intel Haswell Xeon CPU E5-2698 v3 running at 2.30GHz equipped with 8 K80 dual-boards with 16 effective GPUs. In the following, we call this system Haswell when no GPUs are used, and we add the suffix '+8xK80' whenever both CPUs and GPUs are exploited. The second system hosts the latest Intel commodity chip with dual-socket 14-core Intel Broadwell 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 its 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. Each dense synthetic matrix $A = QDQ^\top$ is generated by initially setting a diagonal matrix $D = \text{diag}(\Sigma)$ containing the singular values, with a specific condition number and from an orthogonal matrix $Q$ generated by calculating the $QR$ factorization of a random entries.

We have considered well and ill-conditioned randomly generated matrices, with the latter representing the worse case scenario, where QDWH performs a maximum of six iterations. In particular, in the subsequent experiments, our QDWH implementation switches Equations from (3.5) to (3.7) from Section 3.2 if $c_k$ is smaller than 100 (see Algorithm 5), which generates $QR$-based iterations for the first three followed by three Cholesky-based iterations.

Fig. 5.1 presents the orthogonality of $U_p$ and the accuracy of the polar decomposition $A = U_pH$ for ill-conditioned matrix on the KNL system. 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
Algorithm 5 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: dgemm2( A, α ) \{ α \approx ||A||_2 \}
6: RUNTIME_sequence_wait()
7: /* Compute U_0 and l_0 */
8: dlaset_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 \approx 1/(||A^{-1}||_1||A||_1) \}
12: else
13:  dgeqrf_Async( B ) \{ A = QR \}
14:  dttrri_Async( B ) \{ Compute R^{-1} \}
15:  Ainvnorm = dlantr_Async( B ) \{ \approx ||A^{-1}||_1 \}
16: RUNTIME_sequence_wait()
17: end if
18: l_0 = 1/(Ainvnorm * Anorm)
19: end for
20: l_0 = (α/1.1) * l_0
21: /* Compute the polar decomposition A = U_p H using QDWH */
22: k = 1, L_i = l_0, conv = 100
23: while (conv ≥ \sqrt{5eps} \{ ||L_i - I|| ≥ 5eps \}) do
24:  L2 = L_i^2, dd = \sqrt{4(1 - L2)/L2^2}
25:  sqd = \sqrt{1 + dd}
26:  a1 = sqd + \sqrt{8 - 4 \times dd + 8(2 - L2)/(L2 \times sqd)}/2
27:  a = real(a1); b = (a - 1)^2/4; c = a + b - 1
28:  L_i = L_i(a + b \times L2)/(1 + cL2)
29:  dlapc_Async( U, U1 ) \{ Backup U_{k-1} \}
30: end while
31: /* Compute U_k from U_{k-1} */
32: if c > 100 then
33:  C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \sqrt{\sqrt{2}U_{k-1}} \begin{bmatrix} 1 \\ I \end{bmatrix}
34:  dgeqrf_Async( C ) \{ C = QR = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R \}
35:  dorgqr_Async( C ) \{ C = Q = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} \}
36:  dgemm_Async( Q_1, Q_2^T, U ) \{ U_k = \frac{1}{\sqrt{2}} \left( \begin{array}{c} a - \frac{b}{2} \\ q_1 \end{array} \right) Q_1 Q_2^T + \frac{1}{2} U_{k-1} \}
37: else
38:  dlaset_Async( Z, 0., 1. ) \{ Z = I \}
39:  dgemm_Async( U^T, U, Z ) \{ Z_k = I - cU_k^T U_{k-1} \}
40:  dgeadd_Async( U, B ) \{ B = U_k^T \}
41:  dpvars_Async( Z, B ) \{ Solve Z_k x = U_k^T \}
42:  dgeadd_Async( B, U ) \{ U_k = \frac{1}{2} U_{k-1} + \left( a - \frac{b}{2} \right) (U_{k-1} W_{k-1}^T) W_{k-1} \}
43: end if
44: dgeadd_Async( U, U1 ) \{ U_k - U_{k-1} \}
45: dlange_Async( U, conv ) \{ conv = ||U_k - U_{k-1}||_F \}
46: RUNTIME_sequence_wait()
47: k = k + 1
48: end while
49: /* Compute H */
50: dgemm_Async( U_k, A, H ) \{ H = U_k^T A \}
51: dlapc_Async( H, B ) \{ B = H \}
52: dgeadd_Async( B, H ) \{ H = \frac{1}{2} (H + H^T) \}
53: RUNTIME_sequence_wait()
Figure 5.1: Assessing the numerical accuracy/robustness of the task-based QDWH.

employ orthogonal transformations, the SVD variant of the polar decomposition necessitates the DC algorithm.

5.1.2 Distributed-Memory

Algorithm 6 describes the pseudo-code of the distributed-memory QDWH based on ScaLAPACK [19]. We define the MPI process grid configuration $P \times Q$ so that the data mapping can occur following the 2D block-cyclic data distribution (2D-BCDD). Each data structure owns a handle or a descriptor, which is paramount for the computations as it describes which process has which chunk of the original matrix data. The size of the chunk is referred as $nb$ and is used to initialize the various data structures needed in the beginning of the code. The $nb$ parameter is also critical for performance as it trades-off concurrency with arithmetic intensity. We set $nb = 64$ as it seems to be the proper sweetspot for all our experiments.

**Environment Settings** Our experiments have been conducted on a Cray XC40 system codenamed Shaheen-2, installed at the KAUST Supercomputing Laboratory (KSL), with the Cray Aries network interconnect, which implements a Dragonfly network topology. It has 6174 compute nodes, each with two-socket Intel Haswell 16-cores running at 2.3GHz and 128GB of DDR3 main memory. We use the vendor ScaLAPACK library from the optimized Cray LibSci numerical library with an in-
Algorithm 6 Distributed-memory QDWH Pseudo-Code using ScaLAPACK.

1: (Set the block size)
2: \( nb = 64 \)
3: Cblacs_gridinit(-1, 0, ictxt_all);
4: Cblacs_gridinit( icxt_all, "R", nprow_all, npcol_all );
5: Cblacs_gridinfo( icxt_all, nprow_all, npcol_all, myrow, mycol );

▷ Initialize data structures using the 2D-BCDD descinit()

▷ Estimate the condition number
6: pdlacpy(A, descA, B, descB, icxt_all);
7: pdgetrf(B, descB, icxt_all);
8: pdgecon(B, descB, alpha, icxt_all);
9: pdlacpy(A, descA, X, descX, icxt_all);
10: pdlascl(X, descX, \( \alpha \), icxt_all), \( \alpha \approx \|A\|_2 \);
11: \( k = 1 \), \( L_i = \beta \times \alpha / 1.1 \), conv = 100;
12: while (conv \( \geq \sqrt{5}eps \) || \( |L_i - 1| \geq 5eps \)) do
13: \( L^2 = L_i^2 \), dd = \( \sqrt{(4(1 - L_i^2))/(L_i^2)} \);
14: \( sqd = \sqrt{1 + dd}, a_1 = sqd + \sqrt{8 - 4 \times dd + 8(2 - L_i^2)/(L_i^2 \times sqd)} \fed \);
15: \( a = real(a_1), b = (a - 1)^2/4, c = a + b - 1 \);
16: \( L_i = L_i(a + b \times L_i^2)/(1 + cL_i^2) \);
17: pdlacpy(X, descX, B, descB);
18: if \( c > 100 \) then
19: \( C = \begin{bmatrix} \sqrt{c} & \sqrt{c} \\ \sqrt{c} & I \end{bmatrix} \);
20: pdgeqrf (C, descC, tau, icxt_all);
21: pdorgqr (C, descC, tau, icxt_all);

▷ Compute \( X_k \) from \( X_{k-1} \)
22: pdgemm(C(1:m, :), descC, C(m:m+n, :), descC, X, descX, icxt_all);
23: else

▷ Cholesky-based iterations
24: pdlaset(C, descC, 0.0, 1.0, icxt_all);
25: pdgemm(B, descB, B, descB, C, descC, icxt_all);
26: pdposv(C, descC, B, descB, icxt_all);

▷ Compute \( X_k \) from \( X_{k-1} \)
27: pdgemm(B, descB, X, descX, icxt_all);
28: end if
29: conv ← \( \|X_k - X_{k-1}\|_F \)
30: \( k = k + 1 \)
31: end while
32: pdgemm(X, descX, A, descA, H, descH, icxt_all);
ternal block size of 64 and the Cray MPICH library. Moreover, we employ a Cray XC system with Aries network interconnect but featuring compute nodes with Intel Xeon Phi Knights Landing (KNL) processors to run more tests. Every KNL has a base frequency of 1.4GHz and is equipped with 192GB DDR4-2400 RAM and 16GB MCDRAM. The KNL system is operated in quadrant mode while the memory is in cache mode, where all MCDRAM is configured as direct-mapped cache. We use 64 cores out of 68 on every KNL for computation, while 4 cores were dedicated to system services. Executables for the KNL target have been generated with the Intel compiler v17.0.1.132. The work load manager is Moab/TORQUE+ALPS.

Our code is written in C programming language, is purely MPI, and is linked against sequential Intel Math Kernel Library for single core high performance. This MPI-only programming model turns out to be the best performing configuration for our ScaLAPACK-based code, as also seen in previous works [52, 74]. We have generated matrices using the pdmatgen ScaLAPACK routine. All computations are performed in double precision arithmetic. All experiments have been run five times and only the minimum time to solution is reported for each test case.

5.2 Full SVD based on QDWH

In this section we assess the shared/distributed-memory implementations of QDWH-SVD (Algorithm 2) to compute $A = U\Sigma V^T$. QDWH-SVD based on three successive computational stages: (1) the polar decomposition calculation of the original matrix using the QDWH algorithm, (2) the symmetric eigendecomposition of the resulting polar factor to obtain the singular values and the right singular vectors, and (3) the matrix-matrix multiplication to get the associated left singular vectors, the GPU accelerated QDWH-SVD framework. We test the accuracy of the singular values, the orthogonality of the singular vectors and the backward error of the overall SVD based on the accuracy metrics in [51].
5.2.1 Shared-Memory

Our QDWH-SVD implementation relies on high performance numerical libraries supporting these operations, such as Intel MKL or MAGMA for targeting CPU and GPU, respectively, and makes successive calls to the corresponding APIs to perform the required operations. Algorithm 7 gives the pseudo-code for the single/multiple GPU implementations of QDWH-SVD. $hA$ designates a pointer to a data structure located on the host (i.e., CPU) memory and $dA$ targets a data structure located on the device (i.e., GPU) memory.

**Algorithm 7** Pseudo-Code for MAGMA-QDWH-SVD on [Single || Multiple] GPU(s).

1: \[ \text{Computing the polar decomposition } A = U \Sigma H \]
2: \[ dX_0 \leftarrow \text{dscal}(dA, \alpha), \alpha \approx \|A\|_2 \]
3: \[ k = 1, L_i = \beta/\alpha, \beta \approx 1/\|A^{-1}\|_2, \text{ conv } = 100 \]
4: \[ \text{while } (\text{conv } \geq \frac{\text{eps}}{\text{eps}} || |L_i - 1| \geq \text{eps}) \text{ do} \]
5: \[ L_2 = L_2^2, dd = \sqrt{(4(1-L_2)/(L_2^2))} \]
6: \[ \text{sqd} = \sqrt{1 + dd}, a_1 = \text{sqd} + \sqrt{8 - 4 \times dd + 8(2 - L_2)/(L_2^2 \times dd)})/2 \]
7: \[ a = \text{real}(a_1), b = (a - 1)^2/4, c = a + b - 1 \]
8: \[ L_i = L_i(a + b \times L_2)/(1 + cL_2) \]
9: \[ dB \leftarrow \text{magma_dlacpy}(dX_k-1, dB) \]
10: \[ \text{if } c > 100 \text{ then} \]
11: \[ dC = \left[ \sqrt{\text{sqd} dB} \right] I \]
12: \[ dC \leftarrow \left[ \text{magma_dgeqrf}_gpp || \text{magma_dgeqrf}_mgpu(dC, tau, dB) \right] \]
13: \[ dC \leftarrow \left[ \text{magma_dorgqr}_gpp || \text{magma_dorgqr}_mgpu(dC, tau, dB) \right] \]
14: \[ dX_k \leftarrow \left[ \text{cublasDgemm}(dC(1: m, :) \text{, } dC(m : m + n, :) \text{, } dX_k-1) \right] || \]
\[ \text{klblas_dgemm}_mgpu(hC(1: m, :) \text{, } hC(m : m + n, :) \text{, } dX_k-1) \] \]
15: \[ \text{else} \]
16: \[ dC \leftarrow \left[ \text{magma_dlas routines || lapack_d77_dlasets}(hC) \right] \]
17: \[ \left[ \text{dC} \leftarrow \text{cublasDgemm}(dC \text{, } dB \text{, } dC) || hC \leftarrow \text{klblas_dgemm}_mgpu(dB \text{, } dB \text{, } hC) \right] \]
18: \[ \left[ \text{dC} \leftarrow \text{magma_dpotrf}_gpp(dC) || hC \leftarrow \text{magma_dpotrf}_mgpu(hC) \right] \]
19: \[ dB \leftarrow \left[ \text{cublasDtrsm}(dB \text{, } dB^{-1}) || \text{magma_dtrsm}_mgpu(hC \text{, } dB^T) \right] \]
20: \[ dB \leftarrow \left[ \text{cublasDtrsm}(dB \text{, } dB) || \text{magma_dtrsm}_mgpu(hC \text{, } dB) \right] \]
21: \[ dX_k \leftarrow \left[ \text{cublasDgeamm}(dB \text{, } dX_k-1) \right] \]
22: \[ \text{end if} \]
23: \[ \text{conv } \leftarrow \|dX_k - dX_{k-1}\|_F \]
24: \[ k = k + 1 \]
25: \[ \text{end while} \]
26: \[ dH \leftarrow \left[ \text{cublasDgemm}(dX_k \text{, } dA) || \text{klblas_dgemm}_mgpu(hX_k \text{, } hA) \right] \]
27: \[ \text{Computing the singular values and the right singular vectors } H = \Sigma \Sigma V^T \]
28: \[ \text{if Standard QDWH-SVD then} \]
29: \[ (\Sigma, dV) \leftarrow \text{magma - QDWHEig}(dH) \]
30: \[ \text{else} \]
31: \[ (\Sigma, hV) \leftarrow \left[ \text{magma_2stage_DSYEV}(hH) || \text{magma_2stage_DSYEV}_mgpu(hH) \right] \]
32: \[ \text{end if} \]
33: \[ \text{Computing the left singular vectors} \]
34: \[ dU \leftarrow \text{cublasDgemm}(dX_k \text{, } dV), hU \leftarrow \text{klblas_dgemm}_mgpu(X_k \text{, } hA) \]
**Environment Settings** The numerical accuracy assessments as well as the performance experiments have been run on a shared-memory multicore architecture composed of a dual-socket 10-core Ivy Bridge Intel(R) Xeon(R) CPU E5-2680 v2 (20 cores total), operating at 2.8 GHz. The system has 2.5 MB of L2 cache, 25 MB of L3 cache and 256 GB of DDR3 main memory. There are three NVIDIA Tesla K40 GPUs (ECC off) with 12 GB of main memory, each connected to the CPU through a PCIe bus 16x. The different implementations (all written in C) have been compiled using the Intel Compiler Suite v13.0.1 and NVIDIA CUDA compilation tools v6.0. The codes have been linked against the following numerical libraries: NVIDIA CUBLAS 6.0, Intel BLAS/LAPACK MKL and MAGMA v1.4.1 for CPU only and GPU only implementations, respectively.

This section compares the numerical accuracy of DGESVD and DGESDD, as implemented in MKL (CPU) and MAGMA (GPU) libraries, against MAGMA-QDWH-SVD (GPU). Since from an accuracy point of view MAGMA-QDWH-SVD with QDWH-EIG or two-stage-EIG gives a similar order of accuracy, only a single curve (MAGMA-QDWH-SVD with two-stage-EIG) is shown on the various graphs, for clarity purposes. Figure 5.2 shows the orthogonality of all singular vectors. In particular, Figures 5.2(a), 5.2(c), 5.2(e) and Figures 5.2(b), 5.2(d), 5.2(f) present the orthogonality of the left and right singular vectors, respectively, for an ill-conditioned matrix of type 4, a well-conditioned matrix, and a random matrix. All SVD solvers pass the orthogonality test and present comparable orders of accuracy. Figures 5.2(c) and 5.2(d) do not show the orthogonality of the left and right singular vectors obtained from DGESDD, due to its accuracy issues revealed in [51]. Figure 5.3 presents the accuracy of the singular values and the backward error of the overall SVD. In particular, Figures 5.3(a), 5.3(c), 5.3(e) and Figures 5.3(b), 5.3(d), 5.3(f) present the accuracy of the singular values and the backward error of the overall SVD solver, respectively, for an ill-conditioned matrix of type 4, a well-conditioned matrix, and a random matrix.
Figure 5.2: Orthogonality of the left (a-c-e) and right (b-d-f) singular vectors.
The accuracy curves for all SVD solvers show that all numerical accuracy tests pass, thanks to an order of accuracy close to the double precision floating-point arithmetic. Similarly, Figures 5.3(c) and 5.3(d) do not show the accuracy of the singular values and the backward error of the overall SVD solver obtained from DGESDD, due to its accuracy issues described in more details in [51].

Since the convergence speed of QDWH-SVD framework is remarkable, a mixed precision technique has been integrated into the original QDWH-SVD algorithm so that the first iterations are done in single precision and the subsequent ones in double precision in order to be able to recover some of the lost digits. This type of precision play, to maintain most of the value of a strictly high precision implementation while doing a significant fraction of the work in faster and cheaper arithmetic, is typical of algorithmic adaptations to extreme architectures to come. We provide empirical evidence that mixed-precision QDWH-SVD implementation can still recover some of the lost digits. Unfortunately, this novel contribution does not work when singular vectors are additionally needed, even though their orthogonality is preserved. The loss of digits in the singular values and their corresponding singular vectors directly affects the backward error of the overall SVD solver, which makes the mixed precision technique suitable only for calculating singular values. Figure 5.4 shows the impact of the mixed precision technique on the numerical accuracy of the MAGMA-QDWH-SVD singular values on an ill-conditioned matrix (type 4) and a random matrix only, since for a well-conditioned matrix, the QDWH-SVD framework rapidly converges after two iterations. The condition number for the random matrix is less than the ill-conditioned matrix and, therefore, the polar decomposition needs fewer iterations to converge, as shown in Figure 5.4(b). By increasing the number of iterations in single precision over double precision, the accuracy of the singular values of MAGMA-QDWH-SVD degrades and it loses two to three digits at most, compared to the other SVD solvers, including the full double precision MAGMA-QDWH-SVD-S0D6.
Figure 5.3: Accuracy of the singular values (a-c-e) and backward error of the overall SVD (b-d-f).
This can still be of interest for applications that show tolerance to a loss of few digits. Although we empirically observed this numerical behavior, formulating it mathematically remains an open problem.

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

(b) Random matrix.

Figure 5.4: Accuracy of the mixed precision MAGMA-QDWH-SVD solver. SiDj means i iterations are performed in single precision and the remaining j iterations in double precision.

5.2.2 Distributed-Memory

Algorithm 8 describes the three computational stages implementation of the QDWH-SVD. Once the QDWH-based polar decomposition (stage 1) has been calculated, we can plug in a symmetric dense eigensolver of choice (stage 2) (e.g., PDSYEVX from ScaLAPACK based on the MRRR eigensolver [75] or ELPA-EIG [76] which combines a two-stage reduction with a divide-and-conquer eigensolver) to compute the singular values and the right singular vectors, and finally, perform a matrix-matrix multiplication to get the left singular vectors. This section presents accuracy assessment of the distributed-memory implementation of QDWH-SVD 8. Figure 5.5 presents the accuracy of the singular values ($\Sigma$), the orthogonality of the left and right singular vectors ($U, V$), and the backward error of the overall SVD ($U\Sigma V^\top$).

Our accuracy as well as performance results for well and ill-conditioned matrices are reported using the systems as in 5.1.2. A single grid configuration 128x288 is used
Algorithm 8 Distributed-memory QDWH-SVD Pseudo-Code using ScaLAPACK.

1: \{Set the block size\}
2: \texttt{nb} = 64
3: Cblacs\_get(-1, 0, ictxt\_all);
4: Cblacs\_gridinit(ictxt\_all, "R", nprow\_all, npcol\_all);
5: Cblacs\_gridinfo(ictxt\_all, nprow\_all, npcol\_all, myrow, mycol);

\textbf{▷ Initialize data structures using the 2D-BCDD descinit()}

\textbf{▷ Compute the polar decomposition} $A = U_p H$ using QDWH:
6: PDGEQDWH(A, $U_p$, H);

\textbf{▷ Compute the singular values and the right singular vectors} $H = V \Sigma V^T$
7: if ScaLAPACK PDSYEVR then
8: pdsyevr(H, descH, $\Sigma$, V, descV)
9: else
10: ELPA\_DSYEVD(H, descH, $\Sigma$, V, descV)
11: end if
12: \{Compute the left singular vectors $U_p \times V$\}
13: pdgemm($U_p$, desc$U_p$, V, descV, U, descU)

to illustrate the numerical experiments. Similar accuracy results can be reported for well and ill-conditioned matrices using the other grid configurations. Herein, there are three variants for the SVD solvers to distinguish: (1) the standard \texttt{PDGESVD} routine from ScaLAPACK, (2) the one-stage divide-and-conquer symmetric eigensolver \texttt{PDSYEVD} from ScaLAPACK, and (3) our QDWH-based SVD solver using the two-stage symmetric eigensolver from the ELPA library \texttt{[76] v2015.11.001}. The QDWH-SVD implementations provide satisfactory accuracy up to the machine precision in double precision arithmetics, across all matrix sizes, for well and ill-conditioned matrices.

5.3 Partial SEP/SVD based on QDWH

This section shows the numerical robustness of the QDWH\_partial-EIG/SVD using up to 36864 processes on \textit{Shaheen-2} System as described in 5.1.2.

5.3.1 Distributed-Memory

Algorithm 9 presents the pseudo-inverse of the QDWH\_partial-EIG/SVD. The iterative QDWH procedure used in the partial SVD calculation is a lightweight variant of the original QDWH method, i.e., performing fewer iterations and without estimating the matrix condition number and second norm. In fact, QDWH enables to isolate the
Figure 5.5: Accuracy comparison of SVD solvers using 128x288 grid configuration: (a-b-c) on Shaheen-2 and (d-e-f) on KNL system.
subspectrum of interest from the overall spectrum. This is done through a tunable numerical user-defined threshold. The threshold is an application-dependent parameter, which physically corresponds to the lower bound of the wanted eigen/singular values. In fact, it is a tunable parameter in the sense that it infers the actual number of wanted eigen/singular values. Therefore, there is a direct link between the percentage of the wanted eigen/singular values and the threshold \( L_i \). A \( QR \) factorization is then applied on \( U_p + Id \) to reveal the index of the first diagonal elements of \( R \) in absolute value, which is below the threshold, also referred to as the null space. This index then determines the orthogonal vectors to extract from \( Q \), from which the projected size of the reduced matrix problem is calculated. Once the smaller matrix problem is generated, a standard SEP/SVD solver can be applied to get the full set of eigen/singular values/vectors. This full set of the spectrum of the reduced matrix problem relates to the subset of desired eigen/singular values/vectors from the original matrix problem. Under extremely ill-conditioned matrices, the method may however raise numerical issues, in case a larger spectrum of the eigen/singular values/vectors are needed. These issues may be fixed by enforcing QR-based QDWH iterations (instead of Cholesky), at the expense of increasing the algorithmic complexity, and therefore, the overall elapsed time.

**Synthetic Matrices** The dense synthetic matrices \( A \in \mathbb{R}^{N \times N} \) are generated using the ScaLAPACK routine \( \text{PDLATMS} \) \( A = Q_1 D Q_2^\top \) with setting mode = 0. For the SEP solvers testing, the matrices are generated with an equispaced eigenvalues \((\lambda)\) as follows:

\[
D = \begin{cases} 
  D[i] = -k \times \text{randn}[i], & i \leq k \\
  D[i] = N - k \times \text{randn}[i], & i \geq k
\end{cases}, \text{ where } k \text{ is the number of negative } \lambda.
\]

For testing the SVD solvers, the distribution of the singular values of the generated matrices follows a geometrical series: \((D[i] = (0.5)^{\frac{i}{N}+100})\) where, \( Q_1, Q_2 \) are ortho-
Algorithm 9: Pseudo-code of the QDWH partial-EIG/SVD using ScALAPACK.

/* Set block size and initiate CBLACS context */
1: Cblacs_get(0, 0, ipctxt)
2: Cblacs_gridmap(ipctxt, imap, nprow, npcol)
3: Cblacs_gridinfo(ipctxt, npcol, myrow, mycol)
4: /* Initialize data structures using the 2D-BCDD descinit() */
5: descinit(nb, nb, A, descA); Fill_in(A, descA)
6: /* Computing the polar factor */
7: if EIG then
8: /* Compute (a, b) and shift the matrix A */
9: Compute (a, b) using Lanczos iterations
10: pdgeadd(A, ((a + b)/2)I) { A - ((a + b)/2)I}
11: end if
12: if a > 0 then
13: Compute the EIG/SVD on the reduced problem
14: end if
15: end while
16: L2 = Li, sqd = \sqrt{(4(1 - L2)/L2^2)}
17: a1 = sqd + \sqrt{8 - 4 \times dd + 8(2 - L2)/((L2 \times sqd)/2}
18: if a = real(a1); b = (a - 1)^2 / 4; c = a + b - 1
19: Li = Li(a + b \times L2)/(1 + cL2)
20: else
21: pdlacl(0, U1) { Backup Uk-1 }
22: end if
23: if c > 100 then
24: C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} \sqrt{c}U_{k-1} \\ I \end{bmatrix}
25: end if
26: /* Compute Uk from Uk-1 */
27: pdgeqr(C) { C = QR = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R }
28: pdorgqr(C) { C = Q = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} }
29: pdgemm(Q_1, Q_2^T, U_1) { U_k = \frac{1}{\sqrt{c}} \begin{bmatrix} (a - \frac{b}{2})Q_1Q_2^T + \frac{b}{2}U_{k-1} \end{bmatrix}
30: else
31: pdlacl(Z, 0, 1) \{ Z = I \}
32: pdgemm(U_1^T, U, Z) { Z_k = I - cU_{k-1}^TU_{k-1} }
33: pdgeadd(U, B) \{ B = U_{k-1}^T \}
34: pdpose( Z, B ) \{ Solve Z_kx = U_{k-1}^T \}
35: pdgeadd(U, B) \{ U_k = \frac{1}{\sqrt{c}}U_{k-1} + \begin{bmatrix} a - \frac{b}{2} \end{bmatrix}(U_{k-1}W_{k-1}^{-1})W_{k-1} \}
36: end if
37: pdsave(U, U1) \{ U_k - U_{k-1} \}
38: pdlacl(U1, conv) \{ conv = ||U_k - U_{k-1}||_F \}
39: k = k + 1
40: end while
/* Uk contains the isolated subspectrum of interests */
41: pdlacl(0.0, 1.0, B)
42: if EIG then
43: pdgeadd(U_k, B)
44: else if SVD then
45: pdgemm(U_k^T, U_k, B)
46: end if
47: pdgeqr(B, tau)
48: ind = min(find(abs(diag(B)) < threshold))
49: size(\tilde{A}) = N - ind
50: pdorgqr(B, tau, Q)
51: \tilde{Q} = Q(c, indend)
52: pdgemm(A_{sort}, \tilde{Q}, \tilde{A})
/* Calculate the EIG/SVD on the reduced problem */
53: if EIG then
54: \tilde{V}\Sigma\tilde{V}^T \leftarrow DSYEVD(\tilde{A})
55: else
56: \tilde{U}\Sigma\tilde{V}^T \leftarrow DGESVD(\tilde{A})
57: end if
58: V \leftarrow dgemm(\tilde{V}, \tilde{Q}^T, V)
onal matrices can be generated by computing the $QR$ factorization of an arbitrary matrices, where $Q_1 = Q_2$ for SEP solvers testing. Figure 5.6 (a, b, c) show the numerical accuracy of three eigensolver variants to compute 10% of the negative eigenvalues: ELPA DC, ScaLAPACK PDSYEVD and QDWHpartial-EIG. The ELPA DC and QDWHpartial-EIG extract the negative eigenspectrum as needed. Figure 5.6 (d, e, f) compare the accuracy of QDWHpartial-SVD (setting different threshold $(s)$) and ScaLAPACK PDGESVD. The QDWHpartial-SVD is capable to extract only the singular values/vectors of interests within the user-defined threshold $(s)$. The ScaLAPACK PDGESVD cannot compute subset of the spectrum. The threshold $(s)$ can be tuned with a priori knowledge on the singular value distribution and can directly influence the number of the computed singular values/vectors (i.e., for $s = 0.1, 0.01, 0.001, 0.0001$, the percentage of the computed singular values/vectors 3%, 7%, 10%, 13%, respectively) and as a results affect the performance of QDWHpartial-SVD. These extensive numerical tests demonstrate the numerical robustness of QDWHpartial-EIG/SVD to provide satisfactory accuracy up to the machine precision for DP computations across all matrix sizes.
Figure 5.6: Assessing the numerical accuracy/robustness using 16x36 grid topology: (a-b-c) for SEP solvers and (d-e-f) for SVD solvers.
Chapter 6

Performance Benchmark of Optimized QDWH-based Implementations on HPC Systems

After the extensive accuracy assessment of the different algorithmic design and improvements shown in Chapter 5, we present comprehensive performance study on shared/distributed-memory systems.

6.1 Polar Decomposition

In this section, we provide performance analysis of the QDWH algorithm 3.2. We highlight the different optimization techniques in the context of Chameleon library. We study the performance of QDWH as well as ZOLO-PD on distributed-memory system.

6.1.1 Shared-Memory

This section presents comprehensive performance assessment and comparisons on a myriad of high-end architectures of the task-based QDWH implementation introduced in 5.1.1. The task-based QDWH implementation in the context of the Chameleon library based on fine-grained computations enables to exploit the identity data structure during the QDWH iterations 3.5. We rely on a dynamic runtime system (i.e., StarPU) 2 to asynchronously schedule the computational tasks among available processing units in order to improve hardware occupancy.
6.1.1.1 Code Optimizations

The Chameleon library provides two APIs to perform dense matrix computations. The first one, CHAMELEON.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 the Algorithm 5. To highlight the benefit of using a 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 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 5) and the corresponding $Q$ generation (line 35 in Algorithm 5). 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-zero tiles are operated on during the factorization. By the same token, during the $Q$ generation step, only the non-zero 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.

The last optimization, CHAMELEON.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. This may unveil look-ahead opportunities at runtime, once
the data dependencies are satisfied, and may engender out-of-order task execution. Indeed, 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 bold font in Algorithm 5. 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 \texttt{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 6.1.1.2. We refer to this implementation as \textit{Async}. It is also noteworthy that it is possible to estimate \textit{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 \textit{conv} against the convergence threshold.

These three code optimizations (i.e., \textit{Sync}, \textit{OptId} and \textit{Async}) can be combined for further performance. While \textit{Sync} and \textit{Async} have a direct impact on task scheduling, \textit{OptId} actually changes the algorithm and reduces the algorithmic complexity.

We first 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 [2].

The condition number estimation $l_0$ can be calculated using the LU factorization, which requires $\frac{2}{3}n^3$ operations, 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 $\frac{4}{3}n^3$ operations, followed by inverting the upper triangular matrix $R$ with $\frac{1}{3}n^3$ operations. 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 3.5, 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 $Q_1 Q_2^T$ 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 3.7, matrix-matrix multiplication involves $2n^3$ operations, the Cholesky factorization needs $\frac{1}{3}n^3$, and solving two linear systems requires $2n^3$ operations. Therefore, the arithmetic cost of Cholesky-based iteration is $(4 + \frac{1}{3})n^3$ operations. Computing the Hermitian polar factor $H = U_p^T A$ requires $2n^3$ operations. 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 [2], 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 3.5. 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 $\sum_{k=1}^{n-1} 4(n - k)(n + 1) = 2n^3$ 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 $\sum_{k=1}^{n-1} 4nk = 2n^3$, as explained in [2]. Table 6.1 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 (3.5).

The standard approach to compute the SVD of a dense matrix is to first reduce it
Table 6.1: Algorithmic complexity of the QDWH-based polar decomposition.

<table>
<thead>
<tr>
<th></th>
<th>well</th>
<th>ill</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) QDWH+LU</td>
<td>$(13 + \frac{1}{3})n^3 \leq \cdots \leq (43 + \frac{3}{3})n^3$</td>
<td></td>
</tr>
<tr>
<td>(2) QDWH+QR</td>
<td>$(12 + \frac{1}{3})n^3 \leq \cdots \leq (41 + \frac{1}{3})n^3$</td>
<td></td>
</tr>
<tr>
<td>(3) QDWH+QR+OptId</td>
<td>$(12 + \frac{1}{3})n^3 \leq \cdots \leq (33 + \frac{1}{3})n^3$</td>
<td></td>
</tr>
</tbody>
</table>

to bidiagonal form $A = U_1BV_1^\top$. The subsequent left and right singular vectors from the bidiagonal solver are then accumulated during the back transformation phase, i.e., $U = U_1U_2$ and $V = V_2V_1$, 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 [71]. Then, we need to add $2n^3$ operations 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 6.1, 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.

6.1.1.2 Performance Results and Analysis

Figure 6.1 highlights the performance impact of various incremental optimizations on the task-based QDWH, as described in Section 6.1.1.1. Taking advantage of the identity matrix structure (OptId), by only operating on non-zero tiles, engenders up to 20% performance improvements compared to the oblivious approach on all studied
systems.

Thanks to fine-grained computations, look-ahead opportunities are unveiled, which directly translate into asynchronous out-of-order task execution at runtime. In fact, this has been previously reported in the literature [77, 36, 38] for many of the dense linear algebra matrix operations, which compose our task-based QDWH implementation (e.g., Cholesky, QR, etc). Therefore, running additionally in asynchronous mode (Async) further reduces time to solution (up to 2.8x). This is true, 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+8xK80 systems, data movement engendered by NUMA and PCIe channels is expensive and can be overlapped by computations, thanks to the Async optimization. Figure 6.2 shows the execution traces

![Figure 6.1: Assessing the performance of various incremental optimizations.](image)

when running in synchronous (Sync API) and asynchronous (Async API) modes. We have added additional synchronization points within the Sync API, after each pan-
el/update computation, so that we can better capture the performance gain against coarse-grained computations engendered by block algorithms, as described in Section 2. 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, yellow and blue 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 6.2(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 6.2(b)).

![Figure 6.2](image)

(a) Synchronous QDWH. (b) Asynchronous QDWH.

Figure 6.2: Comparison of synchronous vs asynchronous execution traces of task-based QDWH on the KNL system with a matrix size of 10240.

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

Figure 6.21 demonstrates the performance scalability of the task-based QDWH implementation. The scalability is almost linear for the commodity CPU systems. For instance, 7.5/3.7-fold speedups are achieved using 32/28 threads from the reference points of 4/7 threads on the Haswell/Broadwell systems, respectively. On the KNL system, the task-based QDWH implementation obtains a 1.9-fold speedup on 64 threads, compared to 32 threads.
Figure 6.3: Assessing the task-based QDWH scalability.

On the densely GPU populated Haswell+8xK80 system, with a total of 16 GPUs, moving data between host and device memory turns out to be challenging. The performance bottleneck has been reported in a recent study [78]. By using the StarPU framework, the data movement overhead of moving data through the thin PCIe bus is partially hidden. StarPU is able to cope with some of these communication overheads by mitigating and adjusting to the memory congestion, thanks to its asynchronous mode of execution. The obtained speedup is 1.4-fold on 16 GPUs, compared to 8 GPUs. Further optimizations are possible by integrating into StarPU some of the performance models discussed in [78].

6.1.1.3 Performance Comparisons of QDWH Variants

Figure 6.4 reports task-based QDWH performance against other various existing QDWH implementations and SVD-based polar decomposition on ill (left) and well (right) conditioned matrices, across the three systems. The missing data points for the polar decomposition variant based on the SVD from MKL correspond to runs,
Figure 6.4: Assessing task-based QDWH performance against other QDWH variant implementations on ill (left) and well (right) conditioned matrices.
which did not achieve the proper accuracy, as defined in Section 5.1.1, probably due to
the convergence failure encountered by the algorithm in the SVD. The corresponding
variant with Elemental does not face this problem because it uses a different imple-
mentation of the QR algorithm in the SVD instead. 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 decomposi-
tion 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 may still mitigate the overhead of data transfers by overlapping
communications with task computations. But recent work [78] may further enhance
the reported work.

It is noteworthy that our task-based QDWH implementation using Chameleon on
the 16 GPUs does not perform well on small matrix sizes. At that matrix scale, the
workloads are too small to saturate all the devices’ floating-point units, and therefore,
performance is ultimately limited by the overhead of off-loading data back and forth.

Also, compared to MAGMA QDWH [51], the task-based QDWH achieves gains
up to [71%, 22%] on Haswell+4xK80 for [ill, well]-conditioned matrices, respectively.

6.1.1.4 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 6.5 presents the performance of the task-based QDWH across all systems investigated in the chapter. 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 not only various architectures but also can achieve 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).

![Figure 6.5: Task-based QDWH performance across various architectures.](image)

6.1.2 Distributed-Memory

In this section, we execute extensive performance tests of the QDWH framework described in 5.1.2 on distributed-memory systems. We investigate the impact of the processor grid topology (square versus rectangular) on the overall performance, and we study the scalability of QDWH using different number of nodes and compare against other methods to compute the polar decomposition. Moreover, we study the performance of QDWH 3.2 against ZOLO-PD 3.3.
6.1.2.1 Impact of the Grid Topology on Performance

The square grid used in [52] may not be adequate for dense linear algebra algorithms involved in the QDWH-SVD or even in the standard bidiagonal SVD algorithm (i.e., the PDGESVD routine from ScaLAPACK), due to excessive data movement in the panel factorization and limited parallelism in the update of the trailing submatrix. Therefore, the grid topology needs to be reconciled with the dense matrix computations, while keeping the same two-dimensional block cyclic data distribution inherited from ScaLAPACK. The processor grid topology needs to be appropriately dimensioned. The panel factorization is typically sequential and may not benefit from having many processors participating in this computational phase. However, one should properly calibrate the number of processors to carry on, in parallel, the update of the trailing submatrix. The performance impact of the processor grid topology is, therefore, critical and is one of the main subjects addressed in the performance results section. This aspect is particularly challenging for QDWH, since it corresponds to a collection of successive calls to several ScaLAPACK routine, see Algorithm 1 in [52], e.g., PDGETRF / PDGECON for the matrix condition estimate, PDGEQRF / PDORMQR for the QR-based QDWH iterations and PDPOSV / PDGEMM for the Cholesky-based QDWH iterations.

The execution time (seconds) and performance (Tflop/s) of the QDWH algorithm for a square ($P = 192, q = 192$) and rectangular ($P = 128, q = 288$) grid topology are shown in Figures 6.6 and 6.7 for ill-conditioned ($\text{cond} = 1\text{e16}$) and well-conditioned ($\text{cond} = 1$) synthetic matrices, respectively, of various sizes and on both platforms. Showing graphs with both metrics, i.e., time and performance, is equally important. The former tells how fast the algorithms is compared to other approaches, while the latter demonstrates how well the underlying hardware is being utilized.

Furthermore, QDWH may perform more iterations with well-conditioned matrices higher than $\text{cond} = 1$. This may translate into a moderate increase of the elapsed time for all variants of QDWH, while keeping a similar trend observed for the impact
Figure 6.6: Performance impact of square and rectangular grid topology \((P \times Q)\) for ill-conditioned matrix on 1152 nodes.

Figure 6.7: Performance impact of square and rectangular grid topology \((P \times Q)\) for well-conditioned matrix on 1152 nodes.
of the grid topology.

The rectangular topology \((P < Q)\) yields higher performance in all cases, up to 31% / 33% and up to 40% / 45% on \textit{Shaheen-2} / \textit{KNL system} for well and ill-conditioned matrices, respectively. This performance gain for the rectangular processor grid configuration may be attributed to less data movement during the panel factorization (thanks to a lower \(P\) number of processors) and improved parallelism in the update of the trailing submatrix (thanks to a higher \(Q\) number of processors) of the various matrix operations involved in the QDWH algorithm. A deeper analysis, based on profiling results obtained from the ratios between computation and communication time, is provided in Section 6.1.2.4 with a focus on the largest matrix size.

### 6.1.2.2 QDWH Scalability for the Polar Decomposition

![Graphs showing performance scalability of QDWH across various matrix sizes and number of MPI processes on both platforms.](image)

(a) Ill-conditioned matrix on \textit{Shaheen-2}.  
(b) Well-conditioned matrix on \textit{Shaheen-2}.  
(c) Ill-conditioned matrix on \textit{KNL system}.  
(d) Well-conditioned matrix on \textit{KNL system}.

Figure 6.8: Performance scalability of QDWH in Tflop/s.

Figure 6.8 shows the parallel performance of QDWH in Tflop/s across various matrix sizes and number of MPI processes on both platforms. QDWH achieves around 200 Tflop/s on 1152 nodes of \textit{Shaheen-2} and 120 Tflop/s on 576 nodes of \textit{KNL sys-
tem for well and ill-conditioned matrices, respectively. The obtained sustained peak performance numbers of the QDWH-based polar decomposition are almost identical for a given system, regardless of the matrix condition number. This can be explained by a similar arithmetic intensity achieved by the matrix computation kernels involved during the QDWH iterations. Figure 6.9 reports the elapsed time in seconds using the same configurations introduced in Figure 6.8. For rather small matrix sizes, the strong scalability of QDWH is limited due to a small workload on each processor. This indicates a low computation / communication ratio and affects the overall performance. However, the scalability decently improves as the matrix sizes increases and this trend can be observed on both platforms. By running on the same number of cores on both systems, we may be able to compare the sustained performance obtained across systems. Considering that both systems rely on the Aries network interconnect, the main difference is the clock frequency of a single core. Shaheen-2’s single core runs at a 60% frequency higher than KNL system, which roughly corresponds to the QDWH performance discrepancy noticed between both systems.

Figure 6.9: Performance scalability of QDWH in time (s).
6.1.2.3 Performance Comparisons of the Polar Decomposition

Figures 6.10 and 6.11 compare the implementation of our QDWH-based polar decomposition with two other state-of-the-art implementations of the polar decomposition, on ill and well-conditioned matrices, respectively, using various numbers of computational nodes. The first one uses an SVD-based polar decomposition, as described in [52], and relies on ScaLAPACK PDGESVD from the Cray LibSci numerical library. The second one uses also a QDWH-based polar decomposition, as implemented in the high performance library Elemental [53]. Unlike ScaLAPACK, which links the algorithmic and distribution block sizes, Elemental uses instead an elemental distribution, which fixes the distribution block sizes to one, and may provide more flexibility for computation/communication optimizations. The range of matrix sizes are adjusted for each subfigure, depending on the number of computational nodes. Our QDWH-based implementation for the polar decomposition outperforms the SVD-based implementation up to 5x and 12x on ill-conditioned matrices on Shaheen-2 and KNL system, respectively. These high speedup numbers can be explained by the slow convergence of the QR iteration for the SVD solver within PDGESVD. Compared to the SVD-based approach, our implementation seems also more tolerant to bandwidth starvation, as seen for manycore KNL architectures, having less bandwidth per core than Haswell architectures. On well-conditioned matrices, although PDGESVD performs better than for ill-conditioned matrices, our QDWH-based implementation for the polar decomposition achieves up to 3x and up to 4x on Shaheen-2 and KNL system, respectively. Furthermore, our QDWH-based implementation for the polar decomposition outperforms the QDWH-based Elemental implementation up to 4x on ill-conditioned matrices on both systems. On well-conditioned matrices, our QDWH-based implementation for the polar decomposition achieves up to 2x and up to 5x on Shaheen-2 and KNL system, respectively. Figures 6.10 and 6.11
permit also to investigate the scalability of the other implementations for the polar
decomposition. Given the high number of processors, the matrix sizes being stud-
ied may engender small workload per processor, and therefore, highlight the impact
of data motion on the overall performance, i.e., on-node communications through
QuickPath/OmniPath network interconnects for Shaheen-2 and KNL system, respec-
tively, or inter-node communications through the Aries network interconnect for both
systems. Our QDWH-based implementation for the polar decomposition is able to
better cope with challenging mode of operations, such as strong scaling.

6.1.2.4 Profiling the Impact of Grid Topology

Possible reasons for the superiority of the rectangular grid topology over the square
topology have been mentioned in Section 6.1.2.1. This section provides a simplified
yet comprehensive profiling view by splitting the execution time into computation
and communication. Table 6.2 shows the ratios between communication and com-
putation time for the QDWH experiments, depicted in Figures 6.6 and 6.7 for the
largest matrix size \( n = 122880 \), on Shaheen-2 and KNL system. The communication
time is measured by intercepting the MPI calls at runtime and by accumulating the
time spent in these calls using an appropriately instrumented and linked executable.
The difference between total time and computation yields the communication time.
The impact of intercepting the MPI calls, in order to calculate the ratio between
communication and computation, is assumed to be negligible, although this overhead
may increase the communication part in the ratio. However, the total time in col-
umn 4 has been measured with a fully optimized executable, as used for Figures 6.6
and 6.7. For both rectangular and square processor layouts, the communication is
predominant over computation. This is typically the case when the work per rank
diminishes, such as in a strong scaling limit context. The rectangular layout shows
a slightly larger computation component, which could be attributed to the reduc-
Figure 6.10: Performance comparison of QDWH implementations for ill-conditioned matrix on different numbers of computational nodes.
Figure 6.11: Performance comparison of QDWH implementations for well-conditioned matrix on different numbers of computational nodes.
<table>
<thead>
<tr>
<th>System</th>
<th>Condition</th>
<th>Grid</th>
<th>Time[s]</th>
<th>Comp.[%]</th>
<th>Comm.[%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaheen-2</td>
<td>Well</td>
<td>Square</td>
<td>179.37</td>
<td>12</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>101.38</td>
<td>21</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>Ill</td>
<td>Square</td>
<td>541.99</td>
<td>14</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>377.87</td>
<td>19</td>
<td>81</td>
</tr>
<tr>
<td>KNL system</td>
<td>Well</td>
<td>Square</td>
<td>286.45</td>
<td>10</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>170.84</td>
<td>16</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>Ill</td>
<td>Square</td>
<td>899.39</td>
<td>11</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rec</td>
<td>615.10</td>
<td>15</td>
<td>85</td>
</tr>
</tbody>
</table>

Table 6.2: Ratios between communication and computation time for the QDWH experiments with square \((P = 192, Q = 192)\) and rectangular \((P = 128, Q = 288)\) grid topologies, extracted from Figures 6.6 and 6.7 for the largest matrix size \(n = 122880\). Data is given for well and ill-conditioned matrices on both systems.

The observation of excessive data movement in the panel factorization compared to the square case. Indeed, Figure 6.12 highlights the ratios of the communication times between square and rectangular grid topologies for ill-conditioned matrices of various sizes on Shaheen-2. The square grid topology performs up to 1.5x more communication than the rectangular grid topology. Another reason may be the improved degree of parallelism during the update of the trailing submatrix of the underlying methods in the rectangular case. However, a deeper investigation of communication as given in Sec. 6.1.2.6 is required to better understand the impact of the grid topology. The rank reordering study of the processors can be found in details in \[79\].

![Figure 6.12: Ratios of the communication time between square and rectangular grid topologies for different matrix sizes.](image)

parallelism during the update of the trailing submatrix of the underlying methods in the rectangular case. However, a deeper investigation of communication as given in Sec. 6.1.2.6 is required to better understand the impact of the grid topology. The rank reordering study of the processors can be found in details in \[79\].
Table 6.3: Sustained performance in Tflops/s and percentage of theoretical peak performance including and excluding the time spent for communication for the cases shown in Table 6.2.

### 6.1.2.5 Percentages of Theoretical Peak Performance

Although our QDWH-based polar decomposition starts to be communication-bound when reaching the limit of strong scaling, we here assess the compute-bound part of our implementation. As described in Section 5.1.2, the compute nodes on Shaheen-2 and KNL system have a theoretical peak performance of approximately 1.18 Tflops/s and 2.6 Tflops/s, respectively, when AVX2 and AVX512 vectorizations are taken into account. Using the expressions for the algorithmic complexity of QDWH, i.e., $(13+1/3)n^3$ for well-conditioned and $(43+2/3)n^3$ for ill-conditioned matrices, as well as the timings reported in Table 6.2, one can compute the number of floating point operations per second and express these values relative to the peak performance of the various platforms, as shown case-by-case in Table 6.3.

For a matrix size of $n = 122880$, 576 compute nodes were used on KNL system and 1152 on Shaheen-2. The ratios from Table 6.2 also allow to exclude the communication time from the total time and thus to estimate the performance of the pure calculation, which is also reported in Table 6.3. Typical fractions of peak performance reached on a single node of Shaheen-2 and KNL system using a simple matrix multiplication (DGEMM) benchmark are 82% and 59%, respectively, which shows that the part of the
algorithm involved with the computation in QDWH is able to reach a satisfactory fraction of the peak performance. In some cases, this performance is even slightly larger than the sustained DGEMM performance on single nodes, since the computation part may be underestimated due to the overhead of the MPI communication profiling tool necessary to calculate the ratios, as already mentioned in Section 6.1.2.4.

6.1.2.6 Towards Performance Modeling

The MPI communication plays a dominant role for the QDWH experiments carried out on 1152 compute nodes of Shaheen-2, as can be seen in Table 6.2. Detailed profiles have been generated with the Cray Perftools suite on Shaheen-2 only, for ill-conditioned matrices on square \((P = 192, Q = 192)\) and rectangular \((P = 128, Q = 288)\) grid topologies, to further characterize the complexity of communication as a function of matrix size for this specific node count.

Information, such as the number of calls of point-to-point and collective MPI communication routines as well as the corresponding amount of bytes transferred and message sizes, have been extracted from these profiles. Without considering the details of the QDWH algorithm, least squares polynomial fits have been used to model the quantities as a function of matrix size using an empirical approach. For a given data set, the polynomial degree was lowered from 4 to 0 and the degree yielding the smallest relative error between data and fit has been chosen.

In general, the execution time of an application instrumented for profiling may be larger than the time spent by a fully optimized executable to process the same problem because instrumentation typically affects optimizations such as inlining. Furthermore, data collection for profiling at runtime may engender overheads. However, it is assumed that the bytes transferred and number of messages are not noticeably affected by profiling.

The profiles show that blocking point-to-point communication predominates, com-
pared to collective communication for all matrix sizes and both grid topologies, as shown in Figure 6.13 and Figure 6.14, respectively. In particular, Figure 6.13 shows the number of calls, total bytes, and message sizes sent over MPI_Send, where (a), (c), and (e) relate to the square grid topology and (b), (d), and (f) to the rectangular one, as a function of matrix size $n$. The data is always an average over the total number of MPI ranks. From the profiles, four ScaLAPACK computational routines emerge to be predominantly calling MPI_Send with QDWH, i.e., PDGEQRF, PDPOSV, PDORGQR, and PDORGQR (see also the QDWH pseudo-code in [52]). This can be seen in (a)-(d) by summing up the individual data for these four routines and compare it to the total sum for all routines in the program. The polynomial fits show that the number of calls scales like $O(n)$ while the number of bytes sent scales like $O(n^2)$, where $n$ is the matrix size. This is valid for both square and rectangular grid topologies.

It is worth noting that the number of bytes transferred during point-to-point communication in the square topology is smaller than in the rectangular case, as reported in Figures 6.13(c) and 6.13(d), respectively. This seems to contradict the superior performance of the rectangular layout at first sight. Indeed, the update of the trailing submatrix is the richest phase in terms of point-to-point communication. Since more processors are involved in that phase when using a rectangular instead of a square topology, the number of transferred bytes is then higher when dealing with rectangular topology. The same holds for the total number of messages. However, the effective time spent in communication is larger for the square topology, as shown by the ratios in Figure 6.12, which means that more time is spent in blocking during point-to-point communication and collective synchronization compared to the rectangular case. The better load balance for the rectangular topology can be attributed to a higher degree of parallelism during the update of the trailing submatrix.

Figures 6.14(c) and 6.14(d) demonstrate an opposite trend, when it comes to assessing collective communication. The panel factorization is the richest phase in terms
of collective communication, especially for the QR-related routines during/after the calculation of the *Householder* reflectors. Therefore, the number of transferred bytes is higher when using a square compared to a rectangular topology, since more processors participate in the panel factorization in the former than the latter configuration. Furthermore, for the square topology scenario, since the parallelism is limited in the panel factorization, mainly characterized by memory-bound operations, as well as during the update of trailing submatrix, with a fewer number of processors, the overall obtained performance is, therefore, lower than the rectangular topology scenario.

The messages are grouped according to their sizes in the ranges \([0B, 16B]\), \([16B, 256B]\), \([256B, 4KiB]\), \([4KiB, 64KiB]\), and \([64KiB, 1MiB]\) and plotted in Figures 6.13(e) and 6.13(f), as a function of \(n\) for square and rectangular grid topology, respectively. Most of the messages have sizes in the range \([64KiB, 1MiB]\) and this set grows like \(O(n)\). This finding is consistent with the linear scaling of the total execution time shown in Figures 6.10(g) and (h), considering that the simulations are communication-bound. The \(O(n^2)\) growth of the total number of bytes transferred is apparently not yet strong enough to affect the linear scaling of the messages in \([64KiB, 1MiB]\), and therefore, the total execution time.

Figures 6.14(e) and 6.14(f) present a similar analysis, as in Figures 6.13(e) and 6.13(f), but for collective communication. In particular, *MPI_Reduce* and *MPI_Bcast* are considered, where the latter is less dominant. Again the number of calls and bytes transferred scale like \(O(n)\) and \(O(n^2)\), respectively, and the majority of messages for *MPI_Reduce* are contained in \([64KiB, 1MiB]\).

All in all, in the strong scaling context, the message counts linearly grow like \(O(n)\) over the problem sizes and this translates into the observed linear scaling of the total execution time in performance figures, as reported in this section.
Figure 6.13: Number of MPI Send calls and corresponding amount of bytes transferred per MPI rank as a function of matrix size $n$. Data shown for square ($P = 192, Q = 192$) and rectangular ($P = 128, Q = 288$) communication layouts. For (a)-(d), the data is given for the four most time consuming ScaLAPACK routines PDGEQRF, PDPOSV, PDORQQR, and PDORQQR calling MPI Send as well as for the sum of these four routines. The total number of MPI Send calls per rank for all routines is also plotted as a reference. Subfigures (e) and (f) show again the total number of MPI Send calls as stacked bars, where the bars are partitioned according to the number of calls with message sizes in certain ranges and are appropriately colored.
Figure 6.14: Number of MPI Reduce and MPI Bcast calls and corresponding amount of bytes transferred per MPI rank as a function of matrix size $n$. Data shown for square ($P = 192, Q = 192$) and rectangular ($P = 128, Q = 288$) communication layouts. For (a)-(d), the data is given for the four most time consuming ScALAPACK routines PDGEQRF, PDPOSV, PDORGQR, and PDORGQR, where the number of calls to MPI Reduce and MPI Bcast are summed up, as well as for the sum of these four routines. The total number of MPI Reduce and MPI Bcast calls per rank for all routines is also plotted as a reference as well as the sum of these two values (total). Subfigures (e) and (f) show again the total number of MPI Reduce calls as stacked bars, where the bars are partitioned according to the number of calls with message sizes in certain ranges and are appropriately colored.
6.2 Full SVD based on QDWH

This section features the performance results of QDWH-SVD using the block algorithm on shared-memory system enhanced with GPUs and on distributed-memory systems. The basic blocks of QDWH-SVD correspond to high-level dense linear algebra operations (QR/Cholesky factorization, symmetric eigensolver, matrix multiplication, etc.).

6.2.1 Shared-Memory

As shown in Algorithm 7, the iterative QDWH-SVD procedure relies primarily upon communication friendly and compute-intensive matrix operations, such as the QR/-Cholesky factorization and the matrix-matrix multiplication, which are very often further optimized by the vendors (Intel MKL, AMD ACML, IBM ESSL, etc.). These types of operations also exhibit a high degree of parallelism. However, although the algorithm performs at most six iterations from the outer loop of the QDWH-SVD framework, it is still important to assess the algorithmic complexity of the overall QDWH-SVD and to compare it against the standard SVD approach. The following performance results are based on the environment setting used to assess the numerical accuracy in 5.2.1.

6.2.1.1 Performance Comparisons on Single GPU

Figure 6.15 shows the performance of both symmetric eigensolvers (MAGMA-QDWH-EIG and MAGMA-two-stage-EIG) on random matrices, when calculating all singular vectors. Because MAGMA-QDWH-EIG performs around 6.5x more flops than MAGMA-two-stage-EIG, there is almost an order of magnitude difference in terms of elapsed time between them, when looking at asymptotic matrix sizes. Figure 6.16 highlights the overall performance of the two QDWH-SVD variants, after plugging in QDWH-EIG or two-stage-EIG, and shows up an overall twofold speedup when
selecting two-stage-EIG over QDWH-EIG as the symmetric eigensolver. For the subsequent performance graphs, MAGMA-QDWH-SVD will always refer to QDWH-SVD with the symmetric eigensolver two-stage-EIG. The authors did not include the performance graph of QDWH-SVD solely based on Intel MKL library \[15\]. Although this variant stands as a natural reference implementation, the extra flops required by QDWH-SVD prohibit getting a competitive CPU implementation, even against other standard SVD solvers on CPUs. This reinforces the need for GPU high computational power in order to overcome these extra flops.

Figures 6.17 and 6.18 present the performance comparisons of all SVD solvers (i.e., DGESVD, DGESDD and QDWH-SVD) for an ill-conditioned matrix of type 4, a well-conditioned matrix, and a random matrix on x86 (using Intel MKL) and GPU, respectively. In particular, Figures 6.17(a), 6.17(c), 6.17(e), 6.18(a), 6.18(c) and 6.18(e) and Figures 6.17(b), 6.17(d), 6.17(f), 6.18(b), 6.18(d) and 6.18(f) depict the performance comparisons when computing the singular values and additionally all singular vectors, respectively. Figures 6.17(c), 6.17(d), 6.18(c) and 6.18(d) do not draw the performance curves of MKL/MAGMA-DGESDD, because of the accuracy issue mentioned in [51]. When calculating only the singular values, the MKL variants (CPU) of DGESVD and DGESDD outperform all MAGMA-DGESVD/DGESDD/QDWH-SVD versions on small ma-
trix sizes, due to the low arithmetic complexity which unveils the overhead of moving data to the device. The same analysis also applies when singular vectors are required additionally for MKL-DGESVD (CPU), only for small well-conditioned matrix sizes as in Figure 6.17(d), thanks to a faster convergence of the SVD solvers. For large matrix sizes, when only singular values are calculated, MAGMA-QDWH-SVD shows competitive performance compared to other SVD solvers on single GPU for ill-conditioned and random matrices, although MAGMA-QDWH-SVD performs more flops than other SVD solvers. It even achieves a twofold and 3.5x speedups against MAGMA-DGESVD (GPU) and MKL-DGESVD (CPU) for well-conditioned matrices, thanks to a reduced number of flops. When singular vectors are required additionally, MAGMA-QDWH-SVD outperforms the SVD solvers on all matrix types, with up to 18% and 30% compared to MAGMA-DGESDD, for ill-conditioned and random matrices, respectively. In fact, the maximum speedup for MAGMA-QDWH-SVD has been achieved against a matrix type, which has similar condition number to the random matrix: it outperforms by up to four-fold, and up to three-fold for asymptotic random matrix sizes, the equivalent MKL-DGESVD and MAGMA-DGESVD routines, respectively, when computing all singular values and vectors.

Figure 6.19 assesses the performance for computing singular values using the mixed precision technique. The condition number for the random matrix is less than the ill-conditioned matrix (type 4) and, therefore, the polar decomposition needs less iteration to converge, as shown in Figure 6.19(b). As expected, time to solution of MAGMA-QDWH-SVD decreases when executing more QDWH iterations in single precision floating-point arithmetic. The mixed precision technique allows to further reduce time to solution and brings an additional 40% performance improvement compared to the full double precision floating-point arithmetic, at the expense of losing two to three digits of accuracy at most.

Finally, Figure 6.20 shows the time breakdown between the main phases of MAGMA-
Figure 6.17: Performance comparisons of MAGMA-QDWH-SVD (GPU) against Intel MKL (CPU) for \( \Sigma \) (a-c-e) and additionally \( U \Sigma V^T \) (b-d-f).
Figure 6.18: Performance comparisons of MAGMA SVD solvers (GPU) for $\Sigma$ (a-c-e) and additionally $U\Sigma V^\top$ (b-d-f).
Figure 6.19: Performance comparisons (Σ only) using mixed precision techniques. SiDj means i iterations are performed in single precision and the remaining j iterations in double precision.

QDWH-SVD for random matrices. It is clear from this profiling Figure that the polar decomposition (QDWH) is the most time consuming stage and its elapsed time increases substantially as the matrix size gets larger.

6.2.1.2 Scalability on Multiple GPUs

Figure 6.21 shows the performance scalability of the multiple GPU implementation of MAGMA-QDWH-SVD. In fact, this corresponds to the first multiple GPU implementation of an SVD solver (to our knowledge). The curves show the speedup of MAGMA-QDWH-SVD on up to three K40 GPUs. This first attempt presents a speedup of roughly 1.7x and 2.1x on two and three GPUs, respectively. The scalability is somewhat limited mainly because of the overhead of moving data between the host and the three GPUs. There is obviously still room for improvement by doing more GPU computations and further hiding the communication overhead by computations, which are doable thanks to the nature of operations (compute-bound) occurring in the polar decomposition step.
6.2.2 Distributed-Memory

The numerical accuracy of QDWH-SVD using different dense symmetric eigensolvers in the backend has been verified in 5.2.2. Figure 6.22 presents the elapsed time of various SVD solvers for well and ill-conditioned matrices, using various processor configurations on Shaheen-2 and KNL systems described in 5.1.2. Although our two variants of QDWH-based SVD solvers perform more flops than PDGESVD, they outperform it by up to 3.6x / 1.7x speedups for ill and well-conditioned matrices, respectively, on Shaheen-2 and by up to 9x / 2x speedups for ill and well-conditioned matrices, respectively, on KNL system. Last but not least, although one can plug in any dense symmetric eigensolvers in the backend, the variant (3) is currently preferred since it relies on the standard and widely-used ScaLAPACK library. In fact, ScaLAPACK is being supported by many high performance vendor numerical libraries, including the Cray LibSci.

6.3 Partial SEP/SVD based on QDWH

In this section, we present the performance results of QDWHpartial-EIG/SVD to compute eigen/singular values of interest.
Figure 6.22: Performance comparison of SVD solvers on different process grid configurations.
6.3.1 Distributed-Memory

Figure 6.23: Performance comparison of SVD solvers on different process grid configurations.

The accuracy assessment of computing the negative eigenspectrum and the most significant singular values/vectors based on the QDWH is presented in Section 5.3.1. Figure 6.23 reports the performance benchmarking campaign of our QDWHpartial-SVD against two other SVD solvers; ScaLAPACK PDGESVD and KSVD [74] across various matrix sizes and number of MPI processes. QDWHpartial-SVD has the flexibility to compute fraction of the singular values/vectors based on a user-defined threshold (s). This threshold (s) can be tuned with a priori knowledge on the singular value distribution and can directly influence the number of the computed singular values/vectors, i.e., for $s = 0.1, 0.01, 0.001, 0.0001$, the percentage of the computed singular values/vectors $3\%, 7\%, 10\%, 13\%$, respectively, and as a results affect the performance of QDWHpartial-SVD. On $16 \times 36$ grid configuration, the QDWHpartial-SVD with setting $s = 0.0001$, is up to 30\% faster than the QDWHpartial-SVD with using other values of $s$. QDWHpartial-SVD achieves gains up to $[6.3x, 2x]$ on $16 \times 36$, $[6x, 2x]$ on $32 \times 72$, $[3.3x, 2.3x]$ on $64 \times 144$ and $[4x, 1.8x]$ on $128 \times 288$ grid topology.
against \[ \text{PDGESVD, KSVD} \] from ScaLAPACK, respectively.

![Graphs showing performance comparison of symmetric EIG solvers on different process grid configurations.](image)

Figure 6.24: Performance comparison of symmetric EIG solvers on different process grid configurations (in time).

Figure 6.24 depicts performance comparison (in time) of QDWHpartial-EIG, ScaLAPACK PDSYEVD and ELPA 2-stage eigensolver across a range of MPI process grid configurations. QDWHpartial-EIG achieves up to 4x speedup against ScaLAPACK PDSYEVD. However, due to the algorithmic complexity overhead of QDWHpartial-EIG \( (14n^3) \) compared to ELPA 2-stage eigensolver \( (6 + 2/3)n^3) \), QDWHpartial-EIG is around 5x slower than ELPA eigensolver. One can also notice that the performance gap decreases as the number of node increases, thanks to a better exploitation of the resources available at hand, but still suffers from the lower number of computing resources. Figure 6.25 shows the performance of the eigensolver variants in Tflop/s metric which is important in the sense that it indicates how well the underlying architecture is used. QDWHpartial-EIG and ELPA eigensolver achieve around 180 Tflop/s and 90 Tflop/s on 36864 MPI processes, respectively. This show the high level of concurrency that QDWHpartial-EIG algorithm is able to expose.
Figure 6.25: Performance comparison of symmetric EIG solvers on different process grid configurations (in Tflop/s).

The ScaLAPACK implementation of QDWHpartial-EIG based on the block algorithm. As discussed in 2.1 the block algorithm suffers from performance losses due to low hardware occupancy engendered by unnecessary in-between synchronization points. Therefore, we would like also to investigate task-based programming model to break the ScaLAPACK bulk synchronous programming model in the context of Chameleon library using dynamic runtime systems for asynchronous task scheduling. Figure 6.26 reports the task-based QDWHpartial-EIG against the task-based 2-stage PLASMA DSYEVD eigensolver.

Because QDWHpartial-EIG performs around 2x more flops than 2-stage PLASMA eigensolver, there is almost 2x difference in terms of the elapsed time. However, adding
GPUs provides QDWHpartial-EIG with the needed computational power to compensate for the arithmetic complexity overhead. Enabling single GPU, QDWHpartial-EIG performs as fast as 2-stage PLASMA eigensolver, using 2 GPUS, QDWHpartial-EIG is up to 2x faster than the 2-stage PLASMA eigensolver.

To enhance the performance of the polar decomposition and its application to computing the full/partial SVD and symmetric eigensolver, next chapter provides a faster implementation of the polar decomposition based on ZOLO-PD algorithm.
Chapter 7

Massively Parallel ZOLO-PD on Distributed-Memory Systems

This chapter presents a high performance implementation of the ZOLO-PD\textsuperscript{3.3} to compute the polar decomposition, which is projected to further improve the parallel performance, thanks to the high concurrency exposed by the ZOLO-PD algorithm. The ZOLO-PD algorithm converges within two iterations even for ill-conditioned matrices, instead of the original six iterations needed for QDWH. The resulting ZOLO-PD has a convergence rate up to seventeen, in contrast to the cubic convergence rate for QDWH. This comes at the price of higher arithmetic costs and memory footprint as shown in\textsuperscript{3.4}. These extra floating-point operations can, however, be processed in an embarrassingly parallel fashion.

7.1 Numerical Accuracy

This section highlights the comprehensive experimental results and reports numerical accuracy of ZOLO-PD against QDWH-based to compute the polar decomposition.

7.1.1 Environment Settings

Our experiments have been conducted on two systems, the first one is Shaheen-2 system described in\textsuperscript{5.1.2} and the second test system codenamed Crystal, is still a Cray XC but now featuring compute nodes with two-socket Intel Broadwell processors each. The core counts of the Broadwell processors range from 18 to 22, i.e., from 36
Figure 7.1: Assessing the numerical accuracy/robustness of QDWH and ZOLO-PD on Shaheen-2.

Figure 7.2: Assessing the numerical accuracy/robustness of QDWH and ZOLO-PD on Crystal.
to 44 per node, with the majority of processors having 18 cores. The base frequency is 2.1GHz and both processors on a node share 128GB of DDR4 memory. We use only 32 cores per compute node, which are evenly distributed among the two sockets in order to properly compare the experiments between both systems. Similarly, the Cray LibSci and MPICH libraries are used. The work load managers on Crystal is Moab/TORQUE+ALPS. Hugepages are employed on Crystal system to improve memory accesses and communication. While Shaheen-2 is a shared resource with many users during the experiments, Crystal is used exclusively for this purpose.

Figures 7.1 and 7.2 show the numerical assessment by checking on the orthogonality of the polar factor as well as the backward error using 200, 400 and 800 nodes, on Shaheen-2 and Crystal, respectively. The orders of the orthogonality and backward errors are similar to QDWH and stand around machine precision, i.e., $1e-15$, which demonstrates the numerical robustness of the algorithm. QDWH is indeed proven to be stable [80], while a proof for ZOLO-PD is currently unavailable; it is conjectured to be stable.

### 7.2 Performance Comparisons of QDWH and ZOLO-PD

Figure 7.3 presents the performance comparisons of ZOLO-PD against QDWH using 200, 400 and 800 nodes on Shaheen-2 and Crystal systems described in 5.1.2. On 200 nodes for both systems, QDWH outperforms ZOLO-PD across all matrix sizes, especially for large matrix sizes. Indeed, ZOLO-PD performs much more flops than QDWH and is not capable of compensating them by executing the independent problems in parallel, due to the lack of resources. To better understand this phenomenon, one should recall how the PD iteration works for QDWH as opposed to ZOLO-PD (see Section 5.1.2). In QDWH, although the PD iterations are done successively, and therefore, all processes work together in computing the QR and Cholesky-based iterations (up to six). In ZOLO-PD, although the PD iterations are performed in parallel,
the overall number of processes is split in process subgroups to work independently on each iteration. As a consequence, there are less processing units per subproblem, which is of similar size than the single QDWH problem. Therefore, for the same matrix size and number of processes, we can highlight the fundamental performance trade-off between QDWH and ZOLO-PD: successive versus independent PD iteration and all processes versus process subgroup per PD iteration. This trade-off stands like a tuning recipe, provides a great flexibility and makes ZOLO-PD amenable to various hardware configurations.

Figure 7.3: Performance comparisons on Shaheen-2 (a-b-c) and Crystal (d-e-f).

Figure 7.4: Performance speedup ZOLO-PD versus QDWH on Shaheen-2.
On 400 nodes, we notice a quite similar performance pattern between QDWH and ZOLO-PD for the small matrix sizes. For the large matrix sizes, the performance gap between both implementations shrinks: as the matrix size increases, the highly parallel ZOLO-PD code starts getting closer to the performance of QDWH, thanks to a better exploitation of the resources available at hand, but still suffers from the lower number of computing resources per independent PD iteration.

On 800 nodes, this configuration actually provides the necessary computational power for ZOLO-PD to outperform QDWH. The performance curves are now inverted. The crossover point occurs directly at the very first small matrix size. While ZOLO-PD exposes plenty of concurrent workloads, QDWH runs out of work and hits the limits of strong scaling by being mostly communication-bound. ZOLO-PD takes better advantage of the underlying hardware than QDWH and demonstrates a clear performance advantage across all matrix sizes. Furthermore, the performance reported on Shaheen-2 and Crystal are very similar, although one would have expected that Shaheen-2 would have been faster due to a higher clock frequency, as described in Section 5.1.2. But since Shaheen-2 resources are shared (e.g., the network interconnect) and not dedicated like Crystal, performance on the former may be close to Crystal or slightly slower, as shown in Figure 7.3.

For the subsequent graphs, we decide to only focus on Shaheen results, since similar benchmarking numbers have been obtained for Crystal.

To further highlight the performance gain, Figure 7.4 reports the speedup between QDWH and ZOLO-PD. The trend is even clearer: the speedup improves significantly, as the number of node increases. All in all, ZOLO-PD outperforms QDWH by achieving up to 2.3x speedup on up to 102,400 cores (i.e., 800 nodes), especially for small matrix sizes, when running in challenging situations, such as strong scaling mode of operation.

One can also notice that the performance speedup decreases as the matrix size
increases, for a given node configuration. This shows QDWH regaining its compute-bound regime of operations, while ZOLO-PD performance starting to cripple due to the algorithmic complexity overhead.

Recalling previous performance comparisons on ill-conditioned matrices obtained in [74], QDWH outperforms its two counterparts, from Elemental [53] and from the SVD-based ScaLAPACK implementation, by up to 4x and 5x, respectively, on Shaheen-2. This makes our ZOLO-PD implementation, which is the crux of this chapter, outperforming by an order of magnitude the current state-of-the-art software libraries for the polar decomposition.

### 7.3 Performance Profiling

To better put the performance results from Section 7.2 in perspective, we present in this section some profiling results where we break down the time to solution into the computational phases. Figure 7.5 shows the time breakdown for QDWH and ZOLO-PD on 200, 400 and 800 nodes. For QDWH, we can see that the PD iterations (i.e., including PO/QR iterations) take up to 85% of the overall execution time. In particular, the QR-based iterations are more time consuming than Cholesky-based iterations. We can see a nice stair-step shape for QDWH, as the matrix size increases for the configuration of 200 nodes. This shape gets attenuated for larger node counts, due to the predominance of communication overheads in performing small dense linear algebra workloads on rather large number of processing units. In particular, for the 800 node case, QDWH experiences a slowdown, since the implementation is mostly communication-bound driven and performs only low arithmetic intensity kernel computation at that scale.

For ZOLO-PD, we observe a nice stair-step shape for all node configurations, as we increase the matrix sizes. The time taken by the independent subproblems to calculate the two PD iterations (one QR-based and one Cholesky-based iterations) is also the
predominant computational part of the overall execution time. When the number of nodes increases, the PD iterations keep scaling nicely, since they can be launched in an embarrassingly parallel fashion. The remaining necessary housekeeping operations for ZOLO-PD (i.e., Gather and Scatter operations) are not critical and do not impede the overall parallel performance. All in all, ZOLO-PD benefits from the concurrency exposure and extracts performance from the available processing units.

Figure 7.5: Profiling QDWH and ZOLO-PD on Shaheen-2.

Figure 7.6: Performance scalability study of QDWH and ZOLO-PD on Shaheen-2.
7.4 Performance Scalability

Figure 7.6 shows the strong scalability for each PD implementations. This figure does not compare QDWH against ZOLO-PD but rather looks separately at their own strong scalability on 400 and 800 nodes, using the corresponding elapsed time of the 200 nodes configuration as the reference. QDWH has major issues in scaling for all matrix sizes studied in this chapter, which are representative of strong scaling scenarios. The 800 node case slows down the overall application and does not leverage performance compared to 400 nodes, let alone the 200 nodes case. On the opposite, ZOLO-PD decently scales up to 800 nodes. There are however some room for further performance improvements. For instance, process placements have not been studied in this chapter and this is an important tuning parameter in order to mitigate the data movement overheads in favor of locally cached data within a single node and/or closer physical inter-node communication operations, especially when the hardware occupancy is low [4).

7.5 Discussion on Energy Efficiency

The strong scaling mode of operation requires massive computational resources to work together in effectively solving a given problem. As seen in previous sections, ZOLO-PD has a better hardware utilization than QDWH at the cost of performing more flops. However, QDWH is slowed down by the excessive data movement occurring during this regime of operation. When looking at today’s hardware trends and memory technologies, the energy consumption of moving a 64-bit word across the network interconnect costs around two orders of magnitude more than the energy consumed to perform a single double precision flop on registers [3]. Moving forward with upcoming exascale systems, this gap will further widen, which makes our ZOLO-PD algorithm standing on the right side, not only in terms of performance, but also
in terms of energy efficiency. A thorough energy consumption comparative study may provide more insights. This may be considered for future work, since the energy assessment of both algorithms is beyond the scope of this thesis.
Chapter 8

QDWHpartial-SVD as an Enabler for Extreme Adaptive Optics

The real-time correction of telescopic images in the search for exoplanets is highly sensitive to atmospheric aberrations. The pseudo-inverse algorithm is an efficient mathematical method to filter out these turbulences. We introduce a new partial singular value decomposition (SVD) algorithm based on QR-based Diagonally Weighted Halley (QDWH) iteration for the pseudo-inverse method of adaptive optics. The QDWHpartial-SVD algorithm selectively calculates the most significant singular values and their corresponding singular vectors. We develop a high performance implementation and demonstrate the numerical robustness of the QDWH-based partial SVD method. We also perform a benchmarking campaign on various generations of GPU hardware accelerators and compare against the state-of-the-art SVD implementation SGESDD from the MAGMA library. Numerical accuracy and performance results are reported using synthetic and real observational datasets from the Subaru telescope. Our implementation outperforms SGESDD by up to five-fold and four-fold performance speedups on ill-conditioned synthetic matrices and real observational datasets, respectively. The pseudo-inverse simulation code will be deployed on-sky for the Subaru telescope during observation nights scheduled early 2018.
8.1 Introduction

Over the last decade, astronomers have identified thousands of exoplanets - planets orbiting stars other than our Sun. There is now solid evidence that habitable planets (rocky planets with temperate surface temperature, able to hold liquid water) are abundant. Most of the 200 billion stars in our galaxy may host such planets. While current telescopes do not have the sensitivity to find evidence of life on nearby habitable planets, the next generation of large (∼30m diameter) telescopes is entering construction and will enter operation during the 2020s decade. These larger telescopes will be capable of detecting water, oxygen, and other gases indicative of biological activity. Yet, a significant challenge remains to be solved: Earth’s atmosphere blurs the image, rendering exoplanet imaging extremely difficult.

8.1.1 Adaptive Optics

To deliver sharp images of the sky, ground-based astronomical telescopes must overcome optical disturbances introduced by Earth’s atmosphere. Without active correction of such defects, images would be blurred to approximately one arcsecond angle (1/3600 of a degree), while in the absence of optical aberrations, a telescope of diameter $D$ imaging at wavelength $\lambda$ should provide $\lambda/D$ angular resolution ($\approx 0.04$ arcsecond in the near-infrared with current 10-m class telescopes). To recover this approximately 25-fold loss of angular resolution, adaptive optics (AO) systems measure and correct for atmospheric turbulence. Any AO system must include three key elements:

- One or several wavefront sensor(s) (WFS) measure optical aberrations in the incoming light beam. The light source for the WFS must be located near the scene to be imaged, and can be a natural star or an artificial laser guide star if no bright star is near the area of interest. Multiple WFSs can participate to
the measurement to provide correction over a wider field of view.

- One or several **deformable mirror(s)** (DM) perform the optical correction. The DM(s) induce optical aberrations that are opposite to the ones induced by atmospheric turbulence.

- **A real-time controller** (RTC) processes the WFS signals to compute DM commands. The RTC must reliably compute commands at 1-5 kHz frequency so that the AO system can keep up with fast changes in atmospheric turbulence.

AO systems are in use on most current large telescopes, routinely delivering images that are as sharp as allowed by the telescope diffraction limit $\lambda/D$.

### 8.1.2 Exoplanet Imaging, Search for Extraterrestrial Life and Extreme AO

Astronomers have long suspected that exoplanets orbit many of the stars visible in the night sky, by analogy to our solar system. The term **exoplanet** is used here to distinguish them from the planets in our solar system. While exoplanets are challenging to observe due to their small mass and size in comparison with the stars they orbit, technological advances have recently enabled their detection in large numbers. Several thousand exoplanets have now been identified [86], and a significant fraction of stars (from 20% to 50%) is believed to host Earth-size rocky planets with temperate surface where liquid water may exist; such planets are referred to as “habitable planets”. Despite the large and growing number of confirmed exoplanets, little is known about their potential to harbor life: habitable planets can currently only be detected indirectly from the gravitational pull they exert on their host star [87], or from the transit events when the planet passes in front of the star [88]. Characterization of exoplanet atmospheres and search for signs of biological activity will require their light to be isolated and analyzed in a spectrograph so that molecular species
such as oxygen and water can be identified. Exoplanet imaging and spectroscopy are the most challenging application for AO systems: exoplanets are between five to ten orders of magnitude fainter than the stars they orbit, and the angular separation between the two objects is usually well below one arcsecond. AO systems optimized for this task are referred to as “Extreme AO” systems (ExAO) due to the extreme level of wavefront correction required. While conventional AO systems aimed at reaching the telescope diffraction limit must correct optical aberrations to a \( \approx 1 \) radian phase residual error (corresponding to \( \approx 250 \)nm in the near-infrared), ExAO systems require nm-level exquisite wavefront correction of low and mid-spatial frequencies. The brightest, largest and nearest exoplanets can be imaged with current large telescopes and high performance AO systems \cite{89, 90, 91}: these gas giant planets are several times more massive than Jupiter (which itself is \( 317 \times \) Earth mass) and are seen in near-IR light owing to their strong internal heat. While habitable exoplanet imaging is widely considered out of reach of current telescopes, the next generation of 30m class telescopes, when equipped with high performance ExAO systems, will have the sensitivity to image and study habitable planets around dozens of nearby stars.

Due to the exquisite wavefront correction requirements of ExAO systems and their large number of sensing and correcting elements, the RTC is a significant challenge: every few hundred microseconds, new input measurements must be processed to compute DM(s) corrections. The computational requirements are significant due to the size of the input and output spaces (each several tens of thousands of variables), high speed (1-10 kHz) and the need for an accurate solution to be produced.

### 8.1.3 Extreme Computing Needs

One of the major components of the AO framework consists in computing the pseudo-inverse of the response matrix (RM), a linear calibration generated during the observational data acquisition. The pseudo-inverse algorithm is composed of dense
linear algebra operations based on compute-intensive numerical kernels. This class of algorithms usually maps well onto manycore architectures, such as GPU hardware accelerators. The main idea consists in extracting the most significant singular values with their associated singular vectors and calculating the explicit pseudo-inverse of the matrix from this subspectrum. The challenges for the pseudo-inverse are twofold: numerical and computational. The numerical challenge resides in dealing with the rectangular shape of the control matrix (CM), which may engender numerical instabilities if a spectral decomposition, i.e., symmetric eigensolver, is employed. On the other hand, the computational challenge lies in the high algorithmic complexity of the pseudo-inverse: although being not real-time, it should still be able to keep up with the overall throughput of the AO framework.

We propose a novel high performance implementation of the partial SVD decomposition using an extension of the QR-based dynamically weighted Halley algorithm (QDWH) [2]. This new algorithm translates the original matrix problem into a reduced problem size containing only the singular values/vectors of interest. The QDWH partial-SVD relies on GPU-friendly numerical kernels, which expose more parallelism than the standard bidiagonal-based SVD solver [10]. We benchmark our new pseudo-inverse implementation against the SVD routine with divide and conquer solver (i.e., SGESDD) from the MAGMA library [92] on various generations of GPU hardware accelerators using synthetic (randomly generated matrices) as well as matrices coming from real datasets generated from the Subaru Telescope.

8.2 Extreme AO’s Extreme Computing Challenge

The ExAO system must perform real-time computation of a DM solution from WFS input data. As described in §8.2.1, this consists of a matrix-vector multiplication (MVM), which is well within the computing capabilities of modern manycore hardware. Derivation of the control matrix to perform this MVM operation, described in
§8.2.2 is considerably more challenging: this is the challenge we are addressing in this thesis. As shown in §8.2.3, it relies on computing the pseudo-inverse of large matrices constructed from WFS measurements. Current (2019) implementations of SGESDD deployed on off-the-shelf GPU hardware fall approximately one order of magnitude short of full-scale derivation of predictive control and sensor fusion control laws on ExAO systems deployed on current large telescope. The controller optimization scope must therefore be reduced by, for example, selecting a subset of the control modes, processing a subset of the accumulated WFS data, or selecting a subset of the available sensors. The performance gap is yet another of magnitude wider for 30-m class telescopes.

8.2.1 Real-time Computations

The conventional AO control scheme is to derive from each WFS measurement the corresponding DM solution. The relationship between WFS and DM is generally assumed to be linear, so this step is a MVM. The input vector (WFS measurement) is multiplied by the CM to estimate the output DM vector that would generate the measured WFS signal. By sending a DM command opposite to this output vector, the optical aberrations measured by the WFS are canceled. Figure 8.1 shows two more advanced control schemes, which are essential to ExAO systems performance.

8.2.1.1 Predictive Control.

The conventional AO control scheme suffers from time lag: corrections applied to the DM are slightly outdated due to hardware and software delays, including WFS camera exposure, readout and data transfer time, computing time, and DM response time. Together, these delays typically add to one millisecond. In addition to hardware and software lag, conventional AO controllers perform some time-averaging of WFS measurements to reduce measurement noise due to shot noise and WFS readout
Figure 8.1: Three possible AO control loop real-time implementations: conventional AO control (top), predictive control (center) and sensor fusion + predictive control (bottom). Vector and matrix sizes are shown below names.

noise. The combined effect is a 1 to 5 ms effective time delay in the correction, and a corresponding error in the AO correction. In ExAO imaging, this error term needs to be reduced by use of “predictive control”: the last \( N \) measurements should be optimally used to estimate the optical aberrations at the time of correction. As shown in Figure 8.1, predictive control can be implemented by adopting the last \( N \) WFS measurements as the input to the correction. The corresponding control matrix is then \( N \) times larger than in conventional AO control.

8.2.1.2 Sensor Fusion.

ExAO’s need for high accuracy measurement of optical aberrations requires multiple sensors to be deployed, and their signals to be optimally combined. The multi-sensor
approach can measure small wavelength-dependent changes in optical aberrations, and reduces the null measurement space to the intersection of individual WFSs null spaces. Figure [8.1] shows a control scheme combining prediction and sensor fusion, assuming that each of the $K$ sensors has the same number of elements $n$. The control matrix size can be significantly larger than in the two previous examples.

### 8.2.2 Inferring Control Matrices from Measurements

In each of the cases presented, the CM computation is a non-real-time task that must be performed before operating the AO system, and may need to be updated while the system is running. In conventional AO control, the CM is computed as the pseudo-inverse of the system response matrix (RM) describing the linear response between DM commands and WFS measurements. The RM is acquired by issuing commands to the DM and measuring the corresponding WFS signal changes. The pseudo-inverse is most often computed using the SVD to allow control modes selection, and reject modes for which the WFS response is weak. In predictive control and sensor fusion, the linear relationships between WFS measurements are usually poorly constrained, and cannot be measured directly. For example, the linear relationship between past and present optical aberrations is a function of wind speed at multiple altitudes above the telescope and the contribution of each atmosphere layer to the overall measured wavefront. This information is generally not accessible in real-time, so the predictive control matrix must be inferred from the WFS measurements. The problem can be translated into a “supervised machine learning” challenge operating in a continuum multi-dimensional space where input vectors are WFS measurements (including history for predictive control and spanning multiple WFSs for sensor fusion) and output vectors are the optical aberrations without time lag. Both input and output vectors of the training set are acquired by the WFS(s). More details can be found in [93].
8.2.3 The SVD-Based Pseudo-Inverse Approach

Thanks to the high measurement cadence, a large training set can be acquired on a timescale shorter than the temporal evolution of underlying linear dependencies (e.g., a change in wind speed and direction): at 2 kHz measurement cadence, \( l = 1.2 \) million samples are collected in 10 min. In a typical ExAO system, upward of \( n = 1,000 \) modes are measured by the main WFS. The predictive control challenge is a supervised machine learning problem, aimed at finding the \( n \)-by-\( Nn \) prediction matrix \( X \) that maps input vectors \( a \) containing the \( N \) last WFS measurements to the \( m \)-sized output space consisting of future value of the WFS coefficients. The input vector \( a \) and output vector \( b \) are therefore of sizes \( Nn \) and \( m = n \) respectively. The optimal prediction matrix \( X \) minimizes the Euclidian distance \( ||Xa - b|| \) across the training set consisting of \( l \) pairs \((a, b)\). By writing the \( l \) training pairs \((a, b)\) as a \( Nn \)-by-\( l \) data matrix \( A \), and a \( n \)-by-\( l \) output matrix \( B \), the problem requires solving a linear system \( XA = B \) where the solution \( X \), i.e., the prediction matrix, is a \((n \approx 1000)\)-by-\((Nn \approx 20,000)\) matrix. The linear system must be solved in a few minutes to keep up with incoming data rates. Computation requirements become even larger with sensor fusion, as described in Section 8.2.1.2 the dimension of the input space grows linearly with the number \( K \) of WFSs.

Computing the solution \( X = BA^+ \) with a SVD-based pseudo-inverse allows to filter out the noise from the empirical measurement data structures \( A \), by extracting the most significant singular values and their corresponding singular vectors.

8.3 The Pseudo-Inverse Algorithm

This section describes the various approaches for calculating the pseudo-inverse of a general dense matrix and presents their algorithmic complexities.
8.3.1 The Pseudo-Inverse Based on the Symmetric Eigenvalue Decomposition

The pseudo-inverse of a general dense matrix $A$ using the SEVD can be described as in Algorithm 10. The general matrix is first symmetrized by multiplying it by its transpose. The eigenvalue decomposition of the resulting symmetric matrix is then performed, from which only the eigenpairs above a certain threshold are saved. The pseudo-inverse can then be computed by forming the diagonal and the orthogonal matrices containing the selected eigenvalues and eigenvectors, respectively. The pseudo-inverse is finally obtained by effectively multiplying both matrices, while taking advantage of their nice numerical properties, e.g., the inverse of an orthogonal matrix is equal to its transpose.

Algorithm 10 Pseudo-Inverse using the spectral decomposition.

1: Compute $S = A^T A$
2: Solve $Sx = \lambda x$
3: Extract the eigenpairs for which $\lambda > \text{threshold}$
4: Calculate the pseudo-inverse $S^+ = X^T \Lambda^{-1} X$

The main challenge with this approach is its numerical robustness. Indeed, the condition number of the symmetric matrix $S$ is equal to the square of the condition number of the original matrix $A$. Although some applications may be resilient to it, this approach is usually not pursued as numerical instabilities may emerge and eventually disturb the eigensolver employed during the diagonalization procedure. From an algorithmic complexity perspective, the bulk of the computation occurs at the symmetric eigenvalue decomposition. As indicated in Section 4.1, one-stage or two-stage tridiagonal reductions correspond to the first computational phase for the symmetric eigenvalue decomposition, regardless if a single or a subset of eigenpairs is needed. This reduction phase may typically account up to 50% of the total elapsed time, when all eigenpairs are needed. Once the condensed form has been achieved, there exist then symmetric eigensolvers (e.g., based on the multiple relatively ro-
bust representations), which are capable of extracting only the eigenpairs of interests within a user-defined range or interval. The back transformation has still to be carried on to retrieve the spectral decomposition of the original dense matrix.

8.3.2 The Pseudo-Inverse Based on the Standard SVD

The pseudo-inverse of a general dense matrix $A$ using the SVD can be described as in Algorithm 11. This approach does not suffer from numerical sensitivity but may be time-consuming, due to the high number of operations. In case the matrix is rectangular, an initial QR or LQ factorization is applied to annihilate the extra rows and columns, respectively. The first computational phase, which reduces the dense matrix into bidiagonal form, may cost twice as much as the tridiagonal reduction. Moreover, similarly to the symmetric eigensolver for the pseudo-inverse in Section 8.3.1, this reduction to condensed form is required, regardless of how many singular values/vectors are needed. In fact, the SVD solver based on the QR algorithm or the divide-and-conquer method cannot calculate a subset of the singular values/vectors. Therefore, the whole spectrum of the singular values/vectors has first to be determined and later filtered out using the threshold parameter, set by the application. The back transformation is then necessary to compute the singular values/vectors of the original matrix $A$, and here again, this expensive computational stage is mandatory, regardless if a single or several singular values/vectors have been requested.

**Algorithm 11** Pseudo-Inverse using the standard SVD.

1: Compute $A = U \Sigma V^T$
2: Extract the singular values/vectors for which $\sigma > \text{threshold}$
3: Calculate the pseudo-inverse $A^+ = V \Sigma^{-1} \tilde{U}^T$
8.3.3 The Pseudo-Inverse Based on the QDWH-based Partial SVD

We herein present an approach, which combines the numerical robustness and the low arithmetic complexity to calculate the pseudo-inverse of a dense matrix $A$. By extending the QR-based dynamically weighted Halley algorithm (QDWH) [2] for the SVD, this new SVD-based pseudo-inverse inherently provides the flexibility to calculate only a subset of the singular values/vectors spectrum, while ensuring backward numerical stability. To our knowledge, this is perhaps the only dense SVD solver that provides this versatility toward computing only a partial SVD within a user-defined range or interval. This QDWHpartial-SVD algorithm permits to project the original matrix spectrum into a reduced matrix problem containing only the desired singular values/vectors. Therefore, it allows to remove altogether the incompressible costs required by the two aforementioned methods during the reduction phase to condensed form. In case the original matrix problem is rectangular, similarly to the pseudo-inverse based on the standard SVD, an initial QR or LQ factorization is applied to annihilate the extra rows and columns, respectively.

Algorithm 12 presents the pseudo-inverse method for a general dense matrix $A$ using the QDWHpartial-SVD. The iterative QDWH procedure used in the partial SVD calculation is a lightweight variant of the original QDWH method, i.e., performing less iterations and without estimations of the matrix condition number and second norm, to calculate the polar decomposition $A = U_p H$. It requires only a few Cholesky or QR-based iterations to get the polar factor $U_p$ containing the desired singular values/vectors. In fact, QDWH enables to isolate the subspectrum of interest from the overall spectrum. This is done through a tunable numerical user-defined threshold. The threshold is an application-dependent parameter, which physically corresponds to the lower bound of the wanted singular values. In fact, it is a tunable parameter in the sense that it infers the actual number of wanted singular values. Therefore, there
is a direct link between the percentage of the wanted singular values and the threshold \( L_i \). A QR factorization is then applied on \( U_p + Id \) to reveal the index of the first diagonal elements of \( R \) in absolute value, which is below the threshold, also referred to as the null space. This index then determines the orthogonal vectors to extract from \( Q \), from which the projected size of the reduced matrix problem is calculated. Once the smaller matrix problem is generated, a standard SVD solver can be applied to get the full set of singular values/vectors. This full set of singular values/vectors of the reduced matrix problem relates to the subset of desired singular values/vectors from the original matrix problem. The desired singular values and left singular vectors are identical for both matrix problems, while the desired right singular vectors of the original matrix problem needs to accumulate \( \tilde{Q} \) with the right singular vectors of the reduced matrix problem. This pseudo-inverse approach presents many advantages. It relies on one-sided transformations, i.e., Cholesky and QR factorizations, which expose much more parallelism than the previous two pseudo-inverse methods based on two-sided transformations. It also focusses the computational power only on the desired spectrum, without wasting resources by over-solving for the whole spectrum. Under extremely ill-conditioned matrices, the method may however raise numerical issues, in case a larger spectrum of the singular values/vectors are needed. These issues may be fixed by enforcing QR-based QDWH iterations (instead of Cholesky), at the expense of increasing the algorithmic complexity, and therefore, the overall elapsed time.

**Algorithm 12** Pseudo-Inverse using the QDWHpartial-SVD.

1: Compute the polar decomposition \( A = U_p H \) using QDWH
2: Calculate \([Q, R] = QR(U_p + Id)\)
3: Find the index \( \text{ind} = \min(\text{find}(\text{abs(diag}(R)) < \text{threshold})) \)
4: Extract \( \tilde{Q} = Q(:, \text{ind} : \text{end}) \)
5: Reduce the original matrix problem \( \tilde{A} = A \times \tilde{Q} \)
6: Compute the SVD of the reduced matrix problem \( \tilde{A} = U \Sigma \tilde{V}^T \)
7: Compute the right singular vectors \( V = \tilde{Q}^T \times \tilde{V} \)
8: Calculate the pseudo-inverse \( A^+ = V \Sigma^{-1} U^T \)
8.3.4 Algorithmic Complexity

Without loss of generality, Table 8.1 reports the algorithmic complexity for square matrices of size \( Nn \) (i.e., the number of predictive filter steps times the number of modes measured by the WFS) based on the three pseudo-inverse approaches discussed in this section. The pseudo-inverse based on the SEVD is cheaper than the SVD-based but may encounter numerical issues, especially for ill-conditioned matrices. The algorithmic complexity of the pseudo-inverse based on the QDWH partial-SVD depends on the number of QDWH Cholesky-based iterations (typically two or three) and the size \( s \) of selected singular values/vectors. Assuming \( s \ll Nn \) and three iterations to get the polar factor from QDWH, the total number of operations is \( 14Nn^3 \), about 66% of the pseudo-inverse based on the standard SVD. The actual SVD solver occurs now only on the reduced problem matrix of size \( s \).

<table>
<thead>
<tr>
<th>Algorithmic complexity</th>
<th>SEVD-based ( 9Nn^3 )</th>
<th>Standard SVD-based ( 22Nn^3 )</th>
<th>QDWH-based partial SVD ( (4+1/3)Nn^3 \times #_{\text{Chol}} )</th>
</tr>
</thead>
</table>

Table 8.1: Complexity for various pseudo-inverse algorithms.

8.4 High Performance GPU-based Implementation

We describe the high performance GPU-based implementation of the pseudo-inverse of a dense matrix \( A \) of size \( n \) using the QDWH partial-SVD algorithm, described in Section 8.3.3. The pseudo-code is highlighted in Algorithm 13 and relies on the MAGMA [92] and NVIDIA cuBLAS [94] libraries. The code is mostly composed of dense linear algebra kernels rich in compute-intensive Level-3 BLAS operations, which makes it GPU-friendly. It can therefore achieve a decent percentage of the system’s theoretical peak performance. Our implementation is GPU-resident, i.e., the memory footprint of the code fits on the GPU’s main memory. MAGMA provides
some support for out-of-core algorithms, in case the GPU’s main memory is limited or simply the problem is too large to fit on the device. Data movement will have then to be carefully handled, since the GPU interconnect bandwidth (i.e., PCIe bus) is typically more than an order of magnitude slower than the GPU onboard memory bandwidth. Last but not least, although the pseudo-code is written in single precision arithmetics to satisfy the computational requirements of the AO framework, it can be easily extended to other precisions for a broader application impact.

Algorithm 13 Pseudo-code of the pseudo-inverse matrix calculation based on the QDWHpartial-SVD using MAGMA/cuBLAS.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$k = 1$, $L_i = \text{threshold}$, $\text{conv} = 100$ (conv $\geq \sqrt{\text{eps}}$ &amp; $</td>
</tr>
<tr>
<td>2.</td>
<td>$L_2 = L_i^2$, $dd = \sqrt{(1 - L_2)L_2}$</td>
</tr>
<tr>
<td>3.</td>
<td>$sqd = \sqrt{1 + dd}$</td>
</tr>
<tr>
<td>4.</td>
<td>$a_1 = sqd + \sqrt{8 - 4 \times dd + 8 \times (2 - L_2)/(L_2 \times sqd)}$</td>
</tr>
<tr>
<td>5.</td>
<td>$a = \text{real}(a_1)$, $b = (a - 1)^2/4$, $c = a + b - 1$</td>
</tr>
<tr>
<td>6.</td>
<td>$L_i = L_i \times (a + b \times L_2)/(1 + c \times L_2)$</td>
</tr>
<tr>
<td>7.</td>
<td>$B \leftarrow \text{magma_slaset}(\text{MagmaFull}, 0.0, 1.0, B)$</td>
</tr>
<tr>
<td>8.</td>
<td>$B \leftarrow \text{cublasSgemm}(A^T, A, B)$</td>
</tr>
<tr>
<td>9.</td>
<td>$C \leftarrow \text{magma_stranspose}(A, C)$</td>
</tr>
<tr>
<td>10.</td>
<td>$C \leftarrow \text{magma_sposv}(\text{MagmaUpper}, B, C)$</td>
</tr>
<tr>
<td>11.</td>
<td>$U^k_p \leftarrow \text{magma_slacpy}(\text{MagmaFull}, A, U^k_p)$</td>
</tr>
<tr>
<td>12.</td>
<td>$A \leftarrow \text{cublasSgemm}(C, U^k_p, A)$</td>
</tr>
<tr>
<td>13.</td>
<td>$U^{k+1}_p \leftarrow \text{cublasSgemm}(A, U^k_p, U^k_p)$</td>
</tr>
<tr>
<td>14.</td>
<td>$\text{conv} \leftarrow |U^{k+1}_p - U^k_p|_F$</td>
</tr>
<tr>
<td>15.</td>
<td>$k = k + 1$</td>
</tr>
</tbody>
</table>

8.5 Experimental Results and Analysis

This section describes the experimental results and presents performance analysis of the GPU-based implementation for the QDWHpartial-SVD used in the pseudo-
inverse, using synthetic matrices and real observational datasets.

8.5.1 Environment Settings

We perform our experiments on three GPU systems, each equipped with a different hardware accelerator generation: K80, P100 and V100 with 12GB, 16GB and 16GB of memory, respectively. The GPU K80 host is a two-socket 14-core Intel Broadwell system with 128GB of main memory. The GPU P100 and V100 hosts are two-socket 16-core Intel Haswell systems, each with 128GB of main memory. We use MAGMA v2.3 and CUDA v9.0 (including cuBLAS) with GCC compilers. Our ultimate goal is to use the QDWHpartial-SVD algorithm to solve the machine learning predictive control challenge, described in §8.2.1.1 for the Subaru Coronagraphic Extreme AO (SCExAO) system [95], deployed on the 8.3m diameter Subaru Telescope. We compare our QDWHpartial-SVD solver against the standard SVD based on the divide-and-conquer solver, since the symmetric eigendecomposition (SEVD) may raise numerical issues, as described in Section 8.3.1. All computations have been performed in single precision (SP) arithmetic. Performance results have been averaged over three successive executions. The order of the numerical accuracy has been recorded for the first run, since the orders of the subsequent runs are similar.

8.5.2 Definitions of Test Matrices

We run the QDWHpartial-SVD against ill-conditioned synthetic matrices as well as matrices generated from real observational datasets. Since the QDWHpartial-SVD performance is sensitive to the matrix condition number due to more iterations required before convergence, ill-conditioned matrices represent the most challenging testcases for our implementation and are, therefore, those of interest in our experimental study. These matrices have been generated using the testing matrix generator SLATMS from LAPACK [96] from geometrically distributed singular values. For the
testing matrices generated from real observational datasets, \(x\) and \(y\) were constructed from WFS measurements acquired on-sky on Sept 12, 2017 (UT). The WFS speed was set at \(f = 2\) kHz, and \(n = 1161\) modes were measured. Four data matrices \(A\), representative of predictive control requirements, were constructed for computing \(N = [1, 5, 10, 15]\) predictive filter steps from \(t = 15\) second of WFS measurements. The ultimate matrix sizes are \(Nn = [1161, 5805, 11610, 17415]\) by 29990, respectively.

It will be also interesting to investigate advanced configurations with higher WFS frequencies and longer period of WFS measurements in the context of future Extremely Large Telescopes (ELT), which will require solving even larger problem sizes. Since GPU’s main memory is a scarce resource, out-of-core algorithms (using CPU’s main memory) and multiple GPUs may enable solving such large problems with additional data movements. These communications can be hidden by the large amount of available computations. Simulating envisioned hardware specifications of ELTs and software deployment on today’s hardware are critical to identify algorithmic and implementation performance bottlenecks early on.

8.5.3 Numerical Accuracy

For the numerical accuracy, we look at three assessment metrics, which are typical for the SVD: the accuracy of the singular values, the orthogonality of the singular vectors and the backward stability [10], as described in Section 7.3 of [51]. Figures 8.2 (a, c, e) show the numerical accuracy/robustness results of QDWHpartial-SVD, performed on the K80 system, using synthetic ill-conditioned matrices. The threshold \(L_t\), introduced used in line 1 of Algorithm 13, is a tunable numerical parameter, which directly influences the number of requested singular values/vectors (i.e., 3%-7%-10%-13%) as well as the obtained accuracy/performance. Numerical results demonstrate the robustness of the QDWHpartial-SVD method, by getting accuracy results around the machine precision for SP computations. The accuracy starts to slightly deteriorate
Figure 8.2: Assessing the numerical accuracy/robustness of QDWHpartial-SVD on K80 using synthetic ill-conditioned matrices (a, c, e) and real observational datasets (b, d, f).
at 13%. As explained at the end of Section 8.3.3, the lost digits can be recovered by enforcing the first iteration of QDWH to rely on the QR factorization, which is much more robust numerically than the Cholesky factorization, as seen in the curves labelled “13% SVD, QR+PO” of Figures 8.2. If the threshold \( Li \) is even smaller so that a high percentage of singular values can be eventually selected, the accuracy may further deteriorate without possible recovery, due to the increase of the condition number. The QDWH\textsubscript{partial-SVD} method is typically robust when extracting less than 15%-20% of the overall spectrum. This threshold \( Li \) can be tuned with a priori knowledge on the singular value distribution. Figures 8.2 (b, d, f) provide the numerical assessment using the same metrics on matrices generated from observational data. We set the threshold so that around 10% of the most significant singular values/vectors are retained in order to fulfill the SCExAO specifications. The obtained accuracy results are at the level of machine precision for SP arithmetics, which highlight the robustness of QDWH\textsubscript{partial-SVD}.

### 8.5.4 Performance Results

Figures 8.3 highlight the performance comparisons of QDWH\textsubscript{partial-SVD} against \texttt{SGESDD} using two performance metrics: time to solution (in seconds) and floating-point operations per second (Gflop/s), respectively. While the former is commonly reported, the latter allows to assess the performance of the actual implementation on a given hardware. In particular, Figures 8.3 (a, b, c) show the elapsed time in seconds using synthetic ill-conditioned matrices on the K80, P100 and V100 systems, respectively. Our QDWH\textsubscript{partial-SVD} is much faster than \texttt{SGESDD}, achieving up to threefold, fourfold and fivefold performance speedups on aforementioned systems. When the required spectrum increases, QR-based QDWH iterations are favored to ensure proper numerical accuracy and engender time performance penalties, but still remain faster than \texttt{SGESDD}. Figure 8.3 (d) draws the elapsed time on real observational datasets.
Figure 8.3: Performance comparisons in Gflops/s of QDWHpartial-SVD using synthetic ill-conditioned matrices (a, b, c) and real observational datasets (d).
for the matrix sizes described in Section 8.5.2. Our QDWHpartial-SVD outperforms SGESDD by more than fourfold performance speedup. Furthermore, Figures 8.3 (e, f, g) show up to 1.8 Tflop/s, 7 Tflop/s and 9 Tflop/s performance, which corresponds to 45%, 75%, and 65% of the theoretical performance peak on the K80, P100 and V100 systems, respectively. Although the sustained peak performance is usually a more representative metric, the current performance assessment shows that there is still room for improvement, although our implementation is able to take advantage of the underlying hardware resources much more than SGESDD. In fact, the performance gap between QDWHpartial-SVD and SGESDD widens, as the hardware technology scaling improves. This indicates that our algorithm is capable of extracting performance, and thanks to its inherent compute-bound kernels, benefits from hardware with over-provisioned floating-point units, which will populate future supercomputers.
Chapter 9

Summary and Future Work

In this thesis, we have demonstrated how QDWH-based algorithms may become competitive alternatives for calculating the Polar Decomposition (PD), for solving the SEP, and for computing the SVD. The comprehensive numerical testing have confirmed the robustness of our distinct QDWH-based implementations. Although they present higher algorithmic complexities than their state-of-the-art counterparts, these QDWH-based algorithms are capable of extracting not only performance from the targeted hardware but also exposing concurrency. As proclaimed in the prominent 2005 article by Herb Sutter \textsuperscript{97} entitled “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”, it is critical to design numerical algorithms that can exploit massively parallel hardware systems with the advent of exascale computing.

In particular, various algorithmic strategies have been deployed to make QDWH-based algorithms attractive, such as mixed precision techniques, asynchronous task-based programming model along with high performance implementations on shared-memory with hardware accelerators as well as large scale distributed-memory systems. The resulting QDWH-based PD, QDWH-SVD and QDWHpartial-EIG/SVD implementations outperform the existing high performance state-of-the-art numerical libraries by up to 2.8x and 12x on shared and distributed-memory, respectively.

The task-based QDWH has been integrated into the Chameleon library (\url{https://gitlab.inria.fr/solverstack/chameleon}) for support on shared-memory systems with hardware accelerators. It is also currently being used by astronomers from the
Subaru telescope located at the summit of Mauna Kea, Hawaii, USA. The distributed-memory software library of QDWH and its SVD extension are freely available under modified-BSD license at [https://github.com/ecrc/qdwh.git](https://github.com/ecrc/qdwh.git) and [https://github.com/ecrc/ksvd.git](https://github.com/ecrc/ksvd.git), respectively. Both software libraries have been integrated into the Cray Scientific numerical library LibSci v17.11.1 and v19.02.1.

Moving forward, task-centric numerical algorithms are paramount to further expose parallelism, while maximizing hardware resources. Since distributed-memory systems equipped with hardware accelerators are becoming ubiquitous in scientific computing, fine-grained task model may help in mitigating the overhead of horizontal (across nodes) and vertical (e.g., across PCIe) data movement by overlapping the expensive data traffic with useful computations. It is important, however, to improve the support of dynamic runtime systems to better orchestrate task scheduling on heterogeneous systems. Last but not least, while solving the Schrödinger equation, material science applications may necessitate to solve SEP for which the matrix operators may present a data-sparse or hierarchically low-rank structure. The main idea will consist in exploiting the structure of the matrix operators to further reduce the algorithmic complexity [98].

Furthermore, ZOLO-PD opens up new research opportunities in investigating the direct performance impact it may have on computing the symmetric eigendecomposition and the SVD, since both traditional linear algebra algorithms suffer from data movement overheads during the panel factorization and the resulting poor scalability.
REFERENCES


[64] A. Haidar, J. Kurzak, and P. Luszczek, “An Improved Parallel Singular Value Algorithm and Its Implementation for Multicore Hardware,” *Proceedings of


List of Publications


6. Hatem Ltaief, Dalal Sukkari, Aniello Esposito, Yuji Nakatsukasa, and David
