High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures

Hatem Ltaief*, Piotr Luszczek† and Jack Dongarra†
*KAUST Supercomputing Laboratory, Thuwal, Saudi Arabia
Hatem.Ltaief@kaust.edu.sa
†Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996, USA
luszczek, dongarra@eecs.utk.edu

***Abstract*—This paper presents a new high performance bidiagonal reduction (BRD) on homogeneous multicore architectures. This paper is an extension of the high performance tridiagonal reduction implemented by the same authors (Luszczek et al., IPDPS 2011) to the BRD case. The BRD is the first step toward computing the singular value decomposition of a matrix which is one of the most important algorithms in numerical linear algebra due to its broad impact in computational science. The high performance of the BRD described in this paper comes from the combination of four important features: (1) tile algorithms with tile data layout which provide an efficient data representation in main memory, (2) a two-stage reduction approach which allows to cast most of the computation during the first stage (reduction to band form) into calls to Level 3 BLAS and reduces the memory traffic during the second stage (reduction from band to bidiagonal form) by using high performance kernels optimized for cache reuse, (3) a data dependence translation layer which maps the general algorithm with column-major data layout into the tile data layout and (4) a dynamic runtime system which efficiently schedules the newly implemented kernels across the processing units and ensures the data dependencies are not violated. A detailed analysis is provided to understand the critical impact of the tile size on the total execution time which also corresponds to the matrix bandwidth size after the reduction of the first stage. The observed performance results show a staggering improvement over currently established alternatives. The new high performance BRD achieves up to 30-fold speedup on a 16 core Intel Xeon machine with a 12000×12000 matrix size against the state-of-the-art open source and commercial numerical software packages, namely, LAPACK compiled with optimized and multithreaded BLAS from MKL as well as Intel MKL version 10.2.**

***Keywords*-Bidiagional Reduction; Tile Algorithms; Two-Stage Approach; Bulge Chasing; Data Translation Layer; High Performance Kernels; Dynamic Scheduling**

I. Introduction

The bidiagonal reduction (BRD) is an important first step when calculating the singular value decomposition (SVD) of any rectangular dense matrix [1]–[3].

$$A = U\Sigma V^T \quad A, \Sigma \in \mathbb{R}^{M\times N}, U \in \mathbb{R}^{M\times M}, V \in \mathbb{R}^{N\times N}.$$

The necessity of calculating SVDs emerges from various computational science areas, e.g., in statistics where it is directly related to the principal component analysis method [4], [5], in signal processing and pattern recognition as an essential filtering tool and in analysis control systems [6]. Following the decompositional approach to matrix computation [7], we transform the dense matrix A to an upper bidiagonal form B by applying successive distinct orthogonal transformations [8] from the left (X) as well as from the right (Y):

$$B = X^T AY \quad B, X, A, Y \in \mathbb{R}^{N\times N}.$$

This reduction step actually represents the most time consuming phase when computing the singular values. Figure 1 shows the time breakdown between the main phases of SVD calculations for various matrix sizes using multithreaded LAPACK implementation from Intel's Math Kernel Library (MKL) version 10.2 on a Intel Xeon core based on Core 2 architecture. The phases are: BRD (labelled as *Reduction*), obtaining the singular values and calculating the corresponding singular vectors from the reduced form using either the dqds algorithm [9] (this method is labelled as *dqds Iteration*) or using Cuppen's divide and conquer algorithm [10], [11] (labelled as *Divide and Conquer Back-transformation*.) Our primary focus is the BRD portion of the computation which can easily consume over 99% of the time needed to obtain the singular values and roughly 75% if singular vectors are additionally calculated. The QR iteration method for singular vectors [12], [13] takes longer by roughly 50% of total time. This method is now deprecated but is still included here for comparison in Figure 1 over the 100% mark.

With the emergence of multicore architectures, the state-of-the-art numerical libraries suffer tremendously from the new memory design characterized by small data caches associated to each core. However, in the relentless pursuit of adequate performance, this problem has already been addressed in the PLASMA library [14] in the context of the one-sided factorizations (LU, QR/LQ and Cholesky) for solving systems of linear equations [15] by redesigning the standard numerical methods using tile algorithms and providing a flexible dynamic runtime system to sustain the applications. More recently, the authors [16] implemented a very efficient two-stage tridiagonal reduction (TRD) ap-

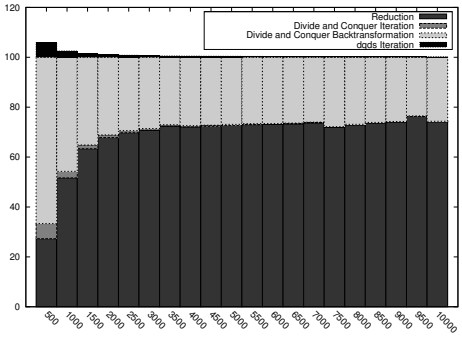


Figure 1. Breakdown of time between three stages of SVD computation: *Reduction* to bidiagonal form (BRD), various *Iteration* methods to obtain singular values, and *Backtransformation* to obtain both sets of singular vectors using LAPACK's routine implementation from MKL 10.2 on an Intel Xeon core based on Core 2 architecture.

proach for dense symmetric matrices using tile algorithms on multicore architectures.

This paper extends the authors' previous work (one-sided factorizations and TRD) to tackle the BRD case, which presents more challenges due to its increased algorithmic complexity. The standard BRD algorithm interleaves the QR and LQ factorizations and requires $^8/_3\ N^3$ floating-point operations for an N by N matrix: twice the cost of the TRD algorithm. Following a two-stage approach, the matrix is transformed into a band bidiagonal form with a bandwidth of size NB using compute intensive kernels, introduced by Ltaief et al. [17]. The band form is then further reduced to the required bidiagonal form using a bulge chasing procedure. This second stage requires the development of new *memory-aware* computational kernels, which reduce memory traffic and memory contention. A dependence translation layer (DTL) allows the mapping of the access pattern (column major) of the bulge chasing technique onto the tile layout, and helps to define the appropriate data dependencies. The dynamic runtime system called SMPSs [18], [19] enables scheduling and overlapping tasks generated from both stages while ensuring the data dependencies are not violated. Two-stage reduction algorithms for two-sided factorizations are not new approaches but have recently enjoyed rekindled interests in the community. For instance, it has been used by Bischof et al. [20] for TRD (SBR toolbox) and Kågström et al. [21] in the context of Hessenberg and Triangular reductions for the generalized eigenvalue problem for dense matrices. The tile bidiagonal reduction that was obtained in this way considerably outperforms the state-of-the-art open-source and commercial numerical libraries.

The bandwidth size b, which also corresponds to the tile size in our case, has a critical impact on the overall performance of the BRD algorithm. It has to be adequately chosen, possibly through an auto-tuning approach, so that the performance of either of the stages is not negatively affected.

The remainder of this document is organized as follows: Section II recalls the block BRD algorithm as implemented in LAPACK [22] and explains its main deficiencies. Section III describes the implementation of the parallel tile BRD algorithm using a two-stage approach. Section IV outlines the dependence translation layer (DTL). Section V has an overview of the different code kernels that are both compute intensive and memory efficient. Section VI presents the performance results. A detailed analysis is provided to understand the critical impact of the bandwidth size on the overall algorithm. Comparison tests are run on shared-memory architectures against the state-of-the-art, high performance dense linear algebra software libraries, LAPACK [22] (open-source package) and Intel MKL 10.2 [23] (commercial package). Finally, Section VII summarizes the results of this paper and presents the ongoing work.

II. The LAPACK Block BRD Algorithm

This section recalls the notion of block algorithms in LAPACK and describes, in particular, the block bidiagonal reduction (BRD).

A. Block Algorithms

LAPACK implements block algorithms to solve linear systems of equations as well as eigenvalue problems and singular value decompositions. Block algorithms are characterized by two successive phases: panel factorization and update of the trailing submatrix. During the panel factorization, the transformations are only applied within the panel. The panel factorization is very rich in Level 2 BLAS operations because the transformations are singly applied. Once accumulated within the panel, those transformations are applied to the rest of the matrix (the trailing submatrix) in a blocking manner leading to Level 3 BLAS operations. While the update of the trailing submatrix is compute-bound and very efficient, the panel factorization is memory-bound and may appear to be a bottleneck for some numerical linear algebra algorithms. Last but not least, the parallelism within LAPACK occurs only at the level of the BLAS routines, which follows the expensive fork-join model. Basically, all processing units need to synchronize before and after each call to BLAS kernels.

B. LAPACK BRD Algorithm

The BRD algorithm with the TRD and the Hessenberg reduction (HRD) are the three two-sided factorizations. As opposed to one-sided factorizations (i.e., LU, Cholesky, QR/LQ), the computed transformations are applied from the left as well as from the right side of the matrix. In particular, Algorithm 1 and Figure 2 describe the LAPACK BRD algorithm for a square matrix of size N for simplicity purposes with a block size NB. The panel factorization (DLABRD) of the block BRD algorithm interleaves two

transformations, i.e. left and right Householder-based reductions. The corresponding left and right reflectors are saved in the original matrix A. Additionally, the accumulation of the left and right transformations (saved in two temporary storages X and Y) requires the memory access of the entire unreduced matrix prior to moving to the next computational phase of the reduction. The update of the trailing submatrix is then straightforward. Two matrix-matrix multiplications are needed, one to apply the accumulated transformations, X, using the left reflectors (V) and the other one to apply the accumulated transformations, Y, using the right reflectors (U). The final computed diagonal and upper or lower diagonal elements are stored in D and E, respectively. It is obvious that the panel factorization is the bottleneck phase for the BRD algorithm due to the accumulation of the left and right transformations which necessitates loading into memory the whole unreduced part of the matrix at each single reduction step. Moreover, this sequence of *Panel-Update* in LAPACK has clearly shown strong limitations on multicore architectures. Indeed, the LAPACK framework is not capable of performing any lookahead computations, where panel or update tasks from multiple steps can significantly overlap. Although, in practice, lookahead techniques would algorithmically be possible only for one-sided factorizations. For two-sided transformations, and the BRD algorithm in particular, the one-stage approach for the reduction to the bidiagonal form necessitates the panel computational step to be atomic because it requires access to the entire trailing submatrix. It is also noteworthy that the BRD algorithm is perhaps the most challenging two-sided transformation compared to HRD and TRD because of the large amount of Level 2 BLAS operations required during the panel factorization. The next section describes the concept of tile algorithms and explains how these new algorithms are able to supersede block algorithms, especially in the context of the BRD algorithm.

Algorithm 1 LAPACK Block Bidiagonal Reduction

```
for I = 1 to N step NB do
  → {Panel Factorization phase: Reduce rows and
  columns I:I+NB-1 to bidiagonal form and return the
  matrices X and Y, which are needed to update the
  unreduced part of the matrix}
  DLABRD(N-I+1, N-I+1, NB,
                A(I,I), LDA,
                D(I), E(I),
                X, LDX, Y, LDY)
  → {Update the trailing submatrix A(I+NB:N, I+NB:N)
  using an update of the form A := A − V ∗ Y' − X ∗ U'}
  DGEMM( 'NoTrans', 'Trans',
                N-I-NB+1, N-I-NB+1, NB, -ONE,
                A( I+NB, I ), LDA,
                Y, LDY, ONE,
                A( I+NB, I+NB ), LDA )
  DGEMM( 'NoTrans', 'NoTrans',
                N-I-NB+1, N-I-NB+1, NB, -ONE,
                X, LDX,
                A( I, I+NB ), LDA, ONE,
                A( I+NB, I+NB ), LDA )
end for
```

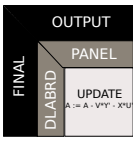


Figure 2. Panel-Update Sequence for the LAPACK BRD Algorithm

III. The Two-Stage Tile BRD Approach

This section recalls the general principles of tile algorithms as well as the idea behind two-stage approaches and describes how these core aspects lead to the tile two-stage BRD algorithm.

A. Tile Algorithms

Tile Algorithms [14] have already shown promising results for the one-sided factorizations as compared to LAPACK and vendor libraries on multicore architectures [15]. The general idea is to transform the original matrix to **tile data layout** (TDL) where each data tile is contiguous in memory as in Figure 3. This may demand a complete redesign of the standard numerical algorithm. The panel factorization as well as the update of the trailing submatrix are then decomposed into several fine-grained tasks which better fit the memory of the small core caches. The parallelism is no longer hidden inside the BLAS routines but rather is brought to the fore. The whole computation can then be represented with a directed acyclic graph (DAG), where nodes are computational tasks and edges represent the data dependencies among them. Next, it becomes critical to efficiently schedule the sequential fine-grained tasks across the processing units. A dynamic runtime environment system is used to distribute the tasks as soon as the data dependencies are satisfied.

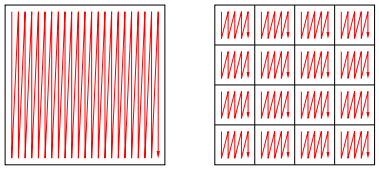


Figure 3. Translation from LAPACK Layout (column-major) to Tile Data Layout

Ltaief et al. [17] have previously attempted to apply the tile algorithm principles on the BRD algorithm. They have implemented new optimized kernels which dramatically

decrease the overhead of the panel factorization. Indeed, the standard BRD algorithm has been redesigned so that the panel factorization phases now involve only input/output data from the local corresponding tiles (and *not* from the entire unreduced matrix). Although high performance results were attained, the bidiagonal reduction was not complete and only a partial reduction to *band* bidiagonal was feasible, which is impractical since the full reduction needs to be achieved in order to calculate the singular value decomposition. More details can be found in Section IV C of [17].

B. Two-Stage Approach

Two-stage approaches have recently proven to be an interesting solution in achieving high performance in the context of two-sided reductions [16], [20], [21]. The first stage consists into reducing the original matrix to band form. The overhead of the Level 2 BLAS operations dramatically decreases and most of the computation is performed in Level 3 BLAS, which makes this stage run closer to the theoretical peak of the machine. This stage is actually so compute-intensive that Bientinesi et al. [24] have proposed to completely offload it to GPU accelerators to further benefit from the underlying hardware. The second stage further reduces the band matrix to the corresponding compact form. A bulge chasing procedure using orthogonal transformations annihilates the off-diagonal elements column-wise and hunts down the fill-in elements to the bottom right corner of the matrix. Figure 4 depicts the execution breakdown of chasing the first column (black elements) on a band bidiagonal matrix of size N=16 and NB=4 with **column-major data layout** (CDL). The red and green rectangles show the left and right transformations, respectively. The dashed elements are the final elements of the bidiagonal structure of the matrix. The dark grey elements represent the fill-in elements left after this first sweep. They will eventually fade out thanks to the subsequent sweeps. It is noteworthy to mention that the introduced bulges are partially destroyed (actually only a single column/row per left/right transformations, respectively). Were the bulges destroyed in their entirety instead, the total number of operations this would increase and the subsequent sweeps would reintroduce them anew anyway. By only eliminating the necessary parts of the bulges within one sweep we allow the following sweeps to naturally chase down the leftover bulges. Finally, it may be readily observed that the whole matrix has to be traversed in order to annihilate a single column. Each sweep is very low in terms of floating-point operations and involves only small regions around the diagonal. Therefore, the standard bulge chasing procedure is completely memory-bound and suffers considerably from the lack of parallelism. Although successive sweeps could potentially be pipelined, it would seriously increase the memory bus traffic as each sweep would be working on different memory regions of the matrix and will not be able forward the data between the cores' caches.

C. Tile Two-Stage BRD Algorithm

The goal of this two-stage tile BRD algorithm presented in this paper is to incorporate the strengths of both tile algorithms and the two-stage approach in order to build an efficient framework for reducing a matrix to bidiagonal form.

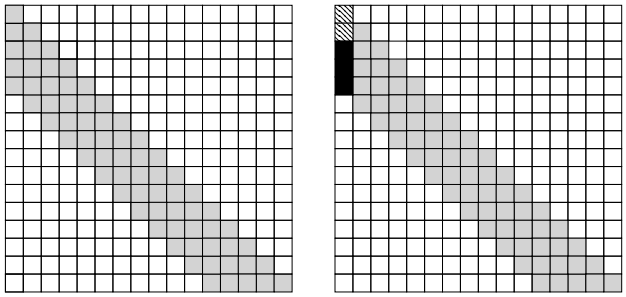
Implementing the first stage using tile algorithms has already been implemented in Ltaief et al. [17]. Figure 5 recalls how the band bidiagonal structure is obtained from a 4-by-4 tile matrix. The matrix is reduced to band form by interleaving QR (left transformations) and LQ (right transformations) factorizations. The light gray tiles correspond to transient data, which still need to be reduced. The black and dark gray tiles are being reduced and the dashed tiles are final data tiles. Four interleaved QR/LQ factorization steps are needed to achieve the band bidiagonal form. The authors refer to the paper for more detailed information.

In the second stage, the bulge chasing algorithm is far from trivial. The challenge resides in associating the different layouts i.e., TDL for tile algorithms and CDL for the bulge chasing. Figure 6 shows the dimension of the complexity. The bulge chasing procedure on the tile matrix creates bulges, which could span over multiple tiles, and therefore, they are not contiguous in memory anymore. Special computational kernels need obviously to be implemented to handle the various cases depending on the number of tiles involved in one particular task. Besides the development of new kernels, a layer of abstraction is required to map the bulge chasing algorithm running on top of CDL format into TDL format. This layer is a crucial component of the two-stage tile BRD algorithm as it homogenizes the layout format across both stages.

The next Section describes the data dependence translation layer in the context of the BRD algorithm.

IV. Dependence Translation Layer

To reiterate the premise explained in the previous sections: the first stage (band reduction) of the BRD reduction fits well with the tile data layout while the second stage (reduction from band to bidiagonal form) does not. The main reason for the mismatch is the misalignment of algorithmic tiles and storage tiles. The former operates at increments of a tile and thus can be easily made to match the storage tiles. The latter, on the other hand, works in one column increments, as each column is annihilated by a similarity transformation, and this results in algorithmic tiles spanning one, two or four storage tiles. The four-tile case is shown in Figure 7 where the misaligned tile spans four storage tiles. The translation layer (DTL), we have devised, provides a connection between the data access originating from the algorithmic formulation and the memory storage scheme. The abstraction provided by DTL affords the programmer the flexibility of working with a column-major layout while the data dependences



(a) Starting point: band bidiagonal matrix in CDL format.

(b) First column annihilation.

(c) Bulge creation: starting to partially chase it.

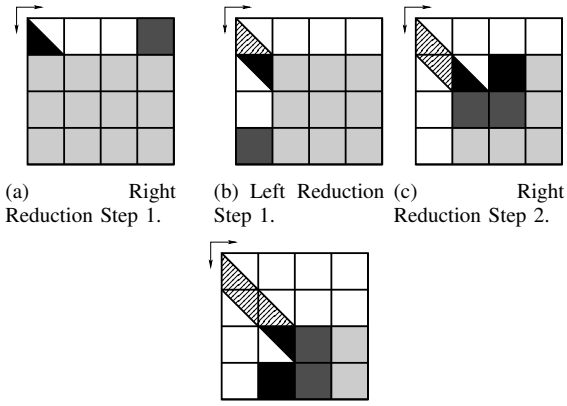
(d) Chasing it with the left and right Transformations.

(e) Chasing it further down.

(f) Reaching the bottom right corner of the matrix.

(g) Final Matrix after the first sweep: the original matrix has lost its band shape.

Figure 4. Execution breakdown of the bulge chasing procedure on a band bidiagonal matrix of size N=16 and NB=4 with **column-major data layout** (CDL) after the first column annihilation (black elements). The red and green rectangles show the left and right transformations, respectively. The dark grey elements represent the fill-in elements, which eventually need to be chased down to the bottom right corner of the matrix. The dashed elements are the final elements of the bidiagonal structure of the matrix.



(a) Right Reduction Step 1.

(b) Left Reduction Step 1.

(c) Right Reduction Step 2.

(d) Left Reduction Step 2.

Figure 5. First stage: reduction to band bidiagonal form applied on a 4x4 tile matrix.

between computational tasks are appropriately propagated to the dynamic scheduling module as if they were specified for tile storage.

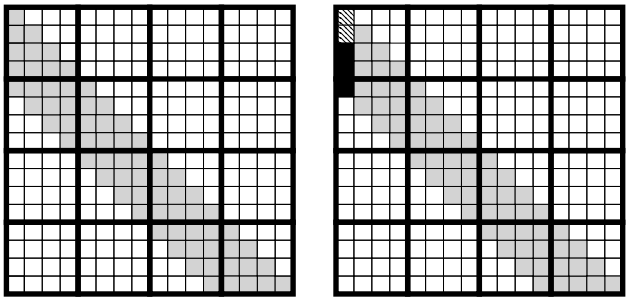
Furthermore, DTL is essential in that it provides the necessary information to the runtime system that allows for overlapping of tasks from both stages. The translation layer supplies the data dependences from the second stage in terms of data used in the first stage which allows the runtime to begin scheduling the second stage tasks as soon as a sufficient portion of the first stage work has been finished. This is readily visible in Figure 8 that shows a DAG for reduction of a 6 by 6 matrix with tile size 2. The second stage tasks (marked with gray) may already be scheduled when only half of the first stage is done in step 12.

The operation of DTL may be explained by perusing Figure 7 where the algorithmic tile spans four storage tiles. The flexibility of DTL allows specification of accesses to the matrix by using column-based storage. DTL intercepts these accesses and first it determines which tiles are affected. Depending on the number tiles, DTL then selects either the 1-tile kernel, 2-tile kernel, or a 4-tile kernel. Our formulation of bulge chasing guarantees that these are the only possible choices. Once the kernel is selected, it is then submitted to the runtime scheduling module for execution. To keep the data dependences satisfied, the submitted task requests exclusive access to the appropriate number of tiles: 4 tiles in the case of the scenario from Figure 7.

The next Section gives a detailed overview of the high performance computational kernels.

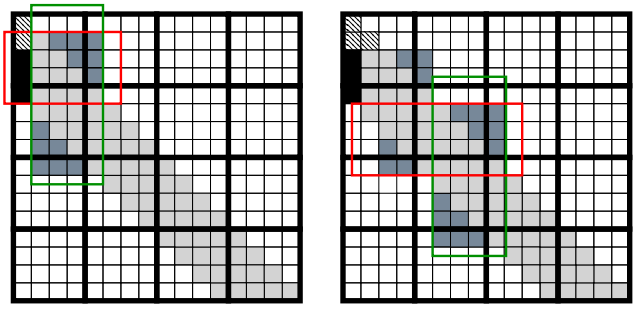
V. High Performance Kernel Descriptions

This Section recalls the computational kernels involved in the first stage and presents the newly developed kernels for the second stage in the context of the two-stage tile BRD algorithm.



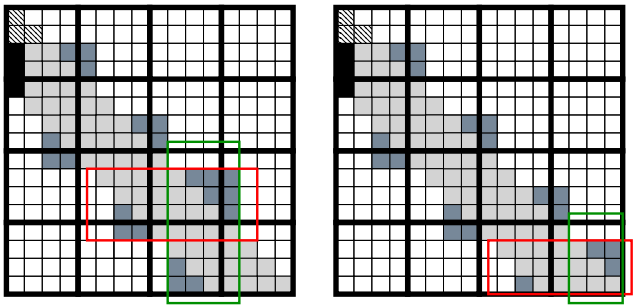
(a) Starting point: band bidiagonal matrix in TDL format.

(b) First column annihilation spanning across two tiles.



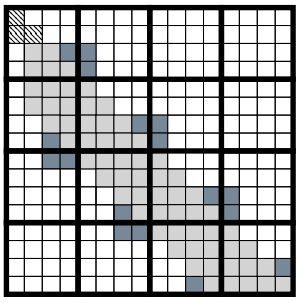
(c) Bulge creation: starting to partially chase it. The new bulges span over four tiles.

(d) Chasing it with left and right tile transformations.



(e) Chasing it further down.

(f) Reaching the bottom right corner of the matrix. The left transformation span over two tiles while the right transformation involves only a single tile.



(g) Final Tile Matrix after the first column annihilation: the tile band structure is restored.

Figure 6. Execution breakdown of the bulge chasing procedure on a band bidiagonal matrix of size N=16 and NB=4 with **tile data layout** (TDL) after the first column annihilation (black elements). The red and green rectangles show the left and right transformations, respectively. The dark grey elements represent the fill-in elements, which eventually need to be chased down to the bottom right corner of the matrix. The dashed elements are the final elements of the bidiagonal structure of the matrix.

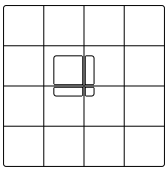


Figure 7. An access to a misaligned tile is broken down by DTL into four subtiles.

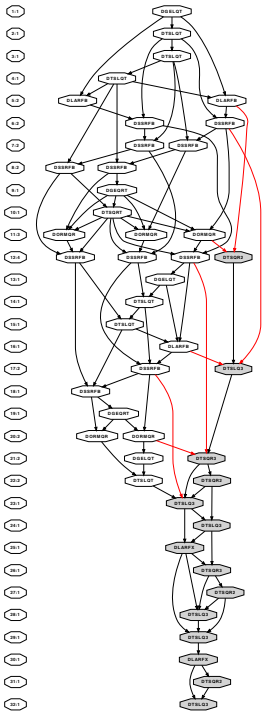


Figure 8. Optimally scheduled DAG (32 steps) for bidiagonal reduction with first stage node marked with dark gray and second – with light gray.

A. General Kernel Descriptions

All kernels are written in standard C and are composed of successive calls to BLAS routines. The kernels from the first stage are mostly Level 3 BLAS and those from the second stage are based on Level 1 and 2 BLAS. As implemented in LAPACK, these kernels rely on orthogonal transformations using Householder reflectors. Orthogonal transformations are an accepted technique that is commonly used for two-sided reductions because they guarantee numerical stability, as opposed to less computationally expensive elementary transformations similar to what is used in Gaussian elimination [3]. Also, as explained in Section III-A, the partitioning of the panel engenders the orthogonal transformation to be incremental rather than direct, as it would happen with block algorithms.

B. Compute-Bound Kernels from the First Stage

Those kernels have already been introduced and implemented in Section III A of [17]. Therefore, the purpose of this subsection is only to make the paper self-contained. There are six compute-intensive kernels overall. This stage basically interleaves the QR and LQ factorizations at each step.

- DGEQRT/DGELQT perform a QR and an LQ factorizations of a single tile, respectively.
- DTSQRT/DTSLQT compute a QR and an LQ factorizations by combining a triangular tile (upper if QR, lower if LQ) with a corresponding full square tile. DTSQRT and DTSLQT are shown in Figure 5(a) and Figure 5(b), respectively.
- DLARFB applies the orthogonal transformations computed from DGEQRT/DGELQT to the left/right side of the trailing submatrix.
- DSSRFB applies the orthogonal transformations computed from DTSQRT/DTSLQT to the left/right side of the trailing submatrix. The right and left applications from DSSRFB are laid out in Figure 5(c) and Figure 5(d) (the black and dark grey data tiles), respectively.

Note that no extra storage is needed to save the Householder reflectors generated from the QR and LQ factorizations. An extra storage is only required to save the triangular factor T of the block reflectors computed from both factorizations, in order to apply them at once in the trailing submatrix.

C. Memory-Bound Kernels from the Second Stage

This second stage is clearly memory-bound and the new kernels need to take into account this delicate property. There are four kernels overall. Figure 9 shows the execution breakdown of the bulge chasing procedure for four complete sweeps on a 4-by-4 tile matrix. The figure represents the name of the consecutive tasks along with the reduction step (from 0 to 3) and the corresponding portions of the matrix accessed. Moreover, there are different cases to consider depending on the region characteristic of the tile matrix being updated (more precisely, the region can span over one, two or four tiles), and for each case, a particular instance of one of the three general kernels is required. In other words, the higher level kernel needs to handle the diverse case in a comprehensive manner. Below are some details about the new kernels:

- DTSQR2 (red in Figure 9) is used to annihilate a single column which can only fit on one or two tiles.
- DTSLQ3 (brown in Figure 9) applies the reflectors computed in DTSQR2 to a diagonal block from the left. Then, it reduces the first row of the introduced bulge and immediately applies the corresponding reflectors on the right of the rest of the diagonal tile. The block being affected can span over one, two, or four neighboring tiles, as shown in the execution breakdown in Figure 9.
- DTSQR3 (green in Figure 9) applies the reflectors calculated in DTSLQ3 from the right. It then annihilates the first column of the created bulge and applies those freshly created reflectors to the left within the block. Here, the block being affected can span over one, two, or four neighbored tiles.
- DLARFX (cyan in Figure 9) applies the reflectors computed from DTSLQ3 to the right and it *always* spans across two tiles.

The bulge chasing procedure necessitates an extra storage to save the generated Householder reflectors from the different bulges, especially if the calculation of the singular vectors is required. Furthermore, since those kernels are called extensively, the whole performance of the second stage relies heavily on an efficient implementation of those routines. All the functions called within the kernels have been inlined up to the level of the BLAS routines. Being memory-bound, this gives a certain flexibility to reorder and to reorganize the successive computational steps within those kernels in order to optimize for cache reuse and data locality. Last but not least, the tile bulge chasing procedure magnifies the DAG size and makes it much more complex with a number of nodes/tasks growing exponentially with the matrix size.

D. Algorithmic Complexity

Considering the original dense matrix square, the algorithmic complexity of the standard BRD is $^8\!/_3\, N^3$ with N the matrix size. The number of flops of the first stage in our tile two-stage BRD is $^8\!/_3 N \times (N-NB) \times (N-NB)$, since the reduction is only achieved up to the band form. The second stage chases the fill-in elements created by the annihilation of the extra entries during the N sweeps. Each sweep calls at most $2 \times {}^N\!/_{NB}$ kernels and $2 \times NB^2$ flops are performed for each kernel. After removing the lower order terms, the number of flops during the bulge chasing procedure is then equal to $4 \times N^2 \times NB$. Therefore, the overall algorithmic complexity of our tile two-stage BRD is roughly the same than the standard BRD.

The next Section presents some performance results of the overall two-stage tile BRD algorithm using the high performance kernels described above. The DTL framework works in association with the dynamic runtime system called SMPSs that is capable of scheduling tasks from both stages simultaneously across the cores of homogeneous multicore architectures as long as data dependences are not violated.

VI. Performance Analysis and Experiments

This Section highlights the parallel performance results achieved by the two-stage tile BRD algorithm. A detailed analysis on the impact of the tile size NB on the overall framework is also discussed.

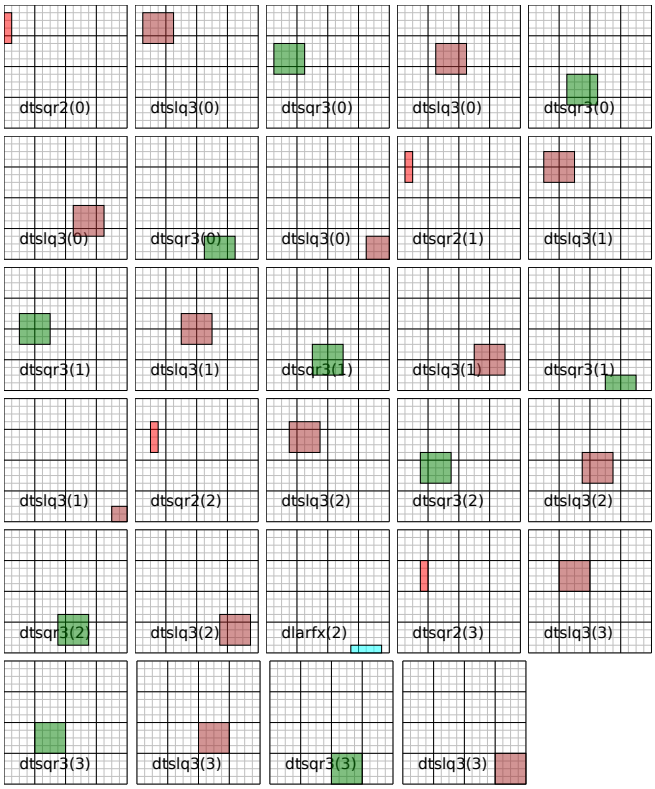


Figure 9. Second stage: graphical representation of portions of the matrix accessed by the consecutive tasks. The yellow lines represent division of the matrix into individual entries and the long black lines delineate matrix tiles in the first stage of the tridiagonal reduction and submatrices accessed in the second stage. The red tasks represent the DTSQR2 kernel, the brown tasks identify the DTSLQ3 kernel, the green tasks show the DTSQR3 kernels and finally, the blue tasks are the DLARFX kernels.

A. Experimental Environment

The experiments have been conducted on a 16-core machine based on an Intel Xeon EMT64 E7340 processor operating at 2.4 GHz. The theoretical peak is equal to 9.6 Gflop/s per core or 153.2 Gflop/s for the whole quad-core quad-socket board. There are two levels of cache. The level 1 cache, local to each core, is divided into 32 KiB of instruction cache and 32 KiB of data cache. Each quad-core processor is composed of two dual-core Core2 architectures, the level 2 cache has 2×4 MB per socket (each dual-core shares 4 MB). The effective bus speed is 1066 MHz per socket leading to a bandwidth of 8.5 GB/s (per socket). The machine is running Linux 2.6.25 and provides Intel Compilers 11.0 together with the Intel MKL 10.2 vendor library. All the experiments presented below focus on asymptotic performance and have been conducted on the maximum amount of cores available on the machine, i.e., 16 cores. The two-stage tile BRD algorithm is compared against the equivalent BRD function from the state-of-the-art open-source and commercial numerical libraries i.e., LAPACK 3.2 linked with optimized MKL BLAS and Intel MKL V10.2, respectively.

B. The Dynamic Runtime System

SMP Superscalar (SMPSs) [18], [19] is a parallel programming framework developed at the Barcelona Supercomputer Center (Centro Nacional de Supercomputación). SMPSs is aimed at "standard" (x86 and like) multicore processors and symmetric multiprocessor systems. The programmer is responsible for identifying parallel tasks, which have to be side-effect-free (atomic) functions. Additionally, the programmer needs to specify the directionality of each parameter (input, output, inout). However, the programmer is not responsible for exposing the structure of the task graph. The task graph is built automatically, based on the information of task parameters and their directionality. The programming environment consists of a source-to-source compiler and a supporting runtime library. The compiler translates C code with pragma annotations to standard C99 code with calls to the supporting runtime library and compiles it using the platform native compiler (Fortran codes are also supported). At runtime the main thread creates worker threads, as many as necessary to fully utilize the system, and starts constructing the task graph (populating its ready list). Furthermore, the SMPSs scheduler attempts to exploit locality by scheduling dependent tasks to the same thread, such that output data is reused immediately.

C. Modeling Performance with Respect to Tile Size

In order to better understand the behavior of our implementation and how it changes with various matrix and tile sizes, we created a performance model. There are two components in the model:

1) *Computation time* (t_x) which encompasses the floating-point operations performed within each task inside the computation kernel routines, and
2) *Communication time* (t_c) which covers the latency of fetching the first cache line of a matrix tile and the time to communicate the rest of it from the main memory to the cache close to the computing core.

For a N by N matrix with a tile size NB, time to completion $t(\mathsf{N},\mathsf{NB})$ for both stages of the reduction is

$$t(\mathsf{N},\mathsf{NB}) = t_x(\mathsf{N},\mathsf{NB}) + t_c(\mathsf{N},\mathsf{NB})$$

In the first stage the individual components of running time are (constant factors ommitted for simplicity of exposition):

$$t_x(\mathsf{N},\mathsf{NB}) = \underbrace{\left(\frac{\mathsf{N}}{\mathsf{NB}}\right)^3}_{\text{number of tasks}} \cdot \overbrace{\mathsf{NB}^3}^{\text{number of flops per task}} = \mathsf{N}^3$$

and

$$t_c(\mathsf{N},\mathsf{NB}) = \overbrace{\left(\frac{\mathsf{N}}{\mathsf{NB}}\right)^3}^{\text{number of tasks}} \cdot \left(\underbrace{\mathsf{NB}^2}_{\text{items to transfer}} + \overbrace{\mathbf{1}}^{\text{latency}} \right)$$
$$= \frac{\mathsf{N}^3}{\mathsf{NB}^2} \cdot \left(1 + \frac{1}{\mathsf{NB}}\right)$$

Similarly for the second stage:

$$t_x(\mathsf{N},\mathsf{NB}) = \underbrace{\mathsf{N}}_{\text{No. of columns}} \cdot \overbrace{\frac{\mathsf{N}}{\mathsf{NB}}}^{\text{number of bulges}} \cdot \underbrace{\mathsf{NB}^2}_{\text{number of flops}} = \mathsf{N}^2 \cdot \mathsf{NB}$$

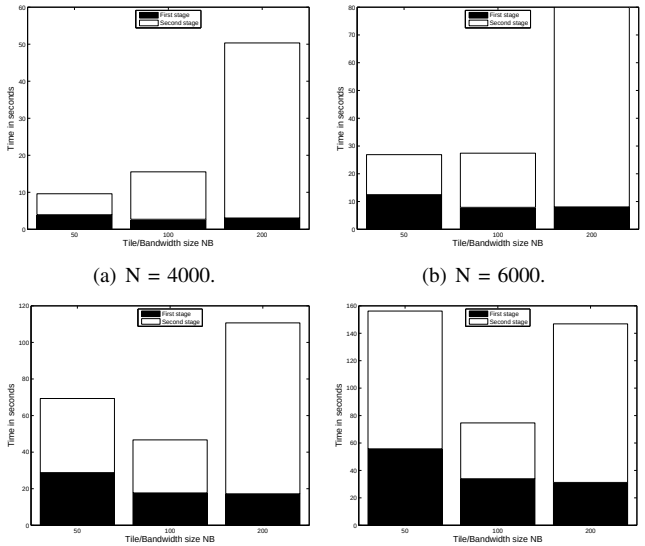
and

$$t_c(\mathsf{N},\mathsf{NB}) = \underbrace{\mathsf{N}}_{\text{columns}} \cdot \overbrace{\frac{\mathsf{N}}{\mathsf{NB}}}^{\text{number of bulges}} \cdot \left(\underbrace{\mathsf{NB}^2}_{\text{items to transfer}} + \overbrace{\mathbf{1}}^{\text{latency}} \right)$$
$$= \mathsf{N}^2 \cdot \left(\mathsf{NB} + \frac{1}{\mathsf{NB}}\right)$$

The model clearly indicates that we should expect a drastically different behavior for the first and second stages of the reduction. The first stage benefits from larger tile sizes because the time is inversely proportional to the tile size NB. The second stage, on the other hand, needs a particular tile size to achieve an optimal behavior because the communication component of the second stage is a rational function of the form $x+1/x$. In the next section, we turn to experiment to investigate this phenomenon further.

D. Tuning the Tile Size Experimentally

Out of the many tunable parameters available for tuning in the TRD code, the tile size NB stands out as the most critical for achieving optimal performance. It determines both the number of tasks and their granularity and is difficult to tune optimally even for one-sided matrix factorizations [15]. In TRD, a two-sided factorization, with the two-stage approach that we employ, there exists a natural tension between the stages that affects the choice of NB. The computational kernels from the first stage benefit greatly from coarse task granularity which allows them to run closer to their sequential kernel peak performance. This follows from the compute-intensive nature of the kernels. On the contrary, the kernels of the second stage are mostly memory-bound and rely on data locality to achieve acceptable performance. Therefore, these kernels depend on data reuse and minimization of data being loaded from memory. The former is most commonly achieved by proper arrangement of data access patterns which in our case can be achieved by memory-friendly scheduling of tasks and having a small NB so that all of the tile data can be retained in the highest level of cache. The latter may simply be achieved by choosing a small NB which results in small band.



(a) N = 4000. (b) N = 6000.

(c) N = 8000. (d) N = 10000.

Figure 10. Impact of NB on the elapsed time (in seconds) of the two-stage BRD for different matrix sizes.

Figure 10 shows the impact of NB on the overall performance of the two-stage BRD with various matrix sizes. For small matrix sizes, i.e., 4000 and 6000, the elapsed time increases with the tile size NB. However, for larger matrix sizes, i.e., 8000 and 10000, the results are not this straightforward. The elapsed time of the second stage is substantially shorter for tile size NB$= 50$ than for NB$= 100$ and even for NB$= 200$ when the matrix size is 10000. Therefore, our simple assumption made above that a small tile size will benefit the second stage is incorrect and it has to be closely examined for a given matrix size. Intuitively, a large matrix size N and a small tile size NB result in an increased number of data requests to the main memory. On the other hand, performance of the first stage deteriorates, as expected, for small tile size $NB = 50$ and is virtually constant for NB$= 100$ and NB$= 200$. These simple analysis and experiments lead us to believe that a good default value for tile size is 100 and this is what we chose for the large scale experiments. We also backed this choice with a series of performance analysis experiments as shown below.

Figure 11 shows a detailed study of how the tile size influences the time to run the first and second stages of TRD as well as both stages simultaneously. The matrix size was set to 5040 because this allows us to have over 30 different NB values smaller than 200 (number 5040 has over 30 divisors due to its set of prime factors). The figure clearly indicates the predicted behavior for the first stage as the performance depends proportionally on the tile size. The larger the tile size, the better the performance of the computational kernel. However, the performance of the second stage exhibits a less obvious trend of having

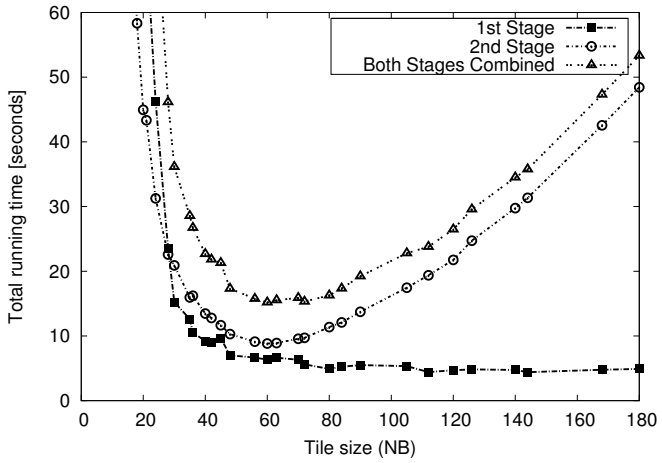


Figure 11. Running time for first, second, and both stages for various tile sizes NB for matrix size 5040.

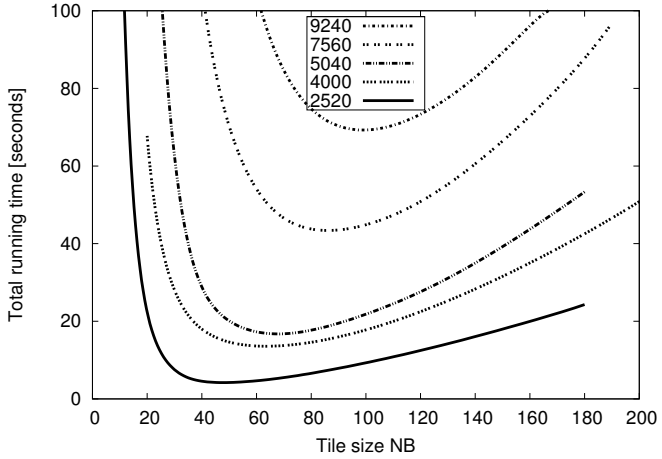


Figure 12. Combined running time for various tile sizes NB for various matrix sizes.

a local minimum at 60. Departure from this value causes deterioration in performance. Accordingly, the cumulative performance of both stages has a similar property. To investigate this further, we chose additional matrix sizes that allow for a wide range of tile sizes and noted the combined performance for both stages. Figure 12 summarizes the results. Two observations are in order. First, for all matrix sizes there exists a locally optimal tile size. And second, an optimal tile size for one matrix size is not optimal for a different matrix size. The former observation makes a case for an autotunig method to be used to choose the optimal tile size [25]. The latter observation raises a question of whether choosing an optimal tile size for each stage independently would benefit performance. Table I attempts to answer this question. The tabulated data shows stage-independent minimums (in the column marked as "Sum") and the minimum of the total time. The stage-independent numbers are purely theoretical as both stages have to share the same tile size. But, in our opinion, it is still instructive to perform this "what-if" experiment. The column labelled "Improvement" shows the potential improvement in running time. It turns out that the improvement is not large (at most 17%) and decreases with the matrix size.

Table I
TIME BREAKDOWN AMONG THE REDUCTION STAGES FOR VARIOUS MATRIX SIZES AND POTENTIAL IMPROVEMENT IF THE TILE SIZE WAS CHOSEN INDEPENDENTLY.

	Shortest running time				
N	1st stage	2nd stage	Sum	Actual	Improvement
	[seconds]			[seconds]	[%]
2520	0.9	2.3	3.2	3.6	10.8%
4000	2.6	5.7	8.3	10.0	16.9%
5040	4.4	8.8	13.2	15.2	12.8%
7560	13.7	21.4	35.0	38.0	7.7%
9240	23.6	32.9	56.5	61.3	8.0%

Choosing the minimum time from both stages is at most 16% faster than choosing the sum.

E. Analysis of Algorithmic Variants

The two most common renditions for numerical linear algebra kernels are right- and left-looking [26]. The former is a preferred option when the amount of available parallelism is the limiting factor [27], [28]. The latter, on the other hand, has much better locality characteristics especially with respect to write operations and is the preferred option for out-of-core codes [29]. One of the consequences of using dynamic DAG scheduling is the loss of fine-grain control of the exact ordering of computations [30]. Despite this loss, we still attempted to investigate the influence of the algorithmic formulation by changing the order in which tasks are submitted to the runtime DAG scheduler. This is a crude approximation of either of the two popular variants but the performance results still give us an indication of which is the more important trait in our code: parallelism or locality. It turned out, that the former is more desirable as the right looking variant consistently outperformed the latter one, albeit by a small margin.

F. Experimental Results

This Section presents the performance results of the overall two-stage BRD algorithm. Figure 13 compares our algorithm with the state-of-the-art open-source and commercial numerical libraries i.e., multithreaded LAPACK compiled with optimized MKL BLAS and Intel MKL version 10.2, respectively. It is surprising to see the same curve behaviors for both packages. The performance of both libraries goes up for small matrix sizes but then it just dies off considerably and does not scale while the matrix size increases. Our two-stage BRD approach starts to go beyond both numerical packages at the crossover point $N = 1500$ and outperforms

them by far for large matrix sizes reaching up to 30-fold speed up on a 12000×12000 matrix size.

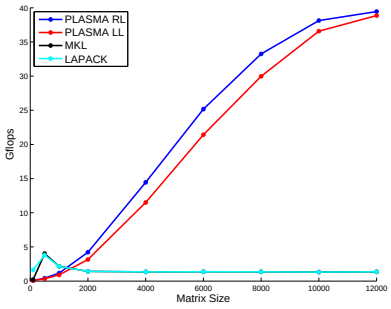


Figure 13. Performance Comparison of the Tile Two-Stage BRD Algorithm against Intel MKL version 10.2 and LAPACK 3.2 xGEBRD.

VII. Summary

This paper focusses on a new high performance two-stage tile bidiagonal reduction (BRD) on homogeneous multicore architectures. Using a two-stage approach on top of tile data layout, the original matrix is first reduced to band form using high performance compute intensive kernels and then further reduced to the final condensed form with efficient memory-optimized kernels. A data dependence translation layer allows us to merge the directed acyclic graphs of tasks from both stages and removes the unnecessary in-between synchronization step. The dynamic runtime system SMPSs can then safely schedule the different computational tasks across the processing units and ensure that the data dependences are not violated. Tuning the tile size is obviously paramount to get the best performance out of the two stage. A brute force mechanism allows the retrieval of an optimal tile size NB depending on the problem size N. We achieved performance results that by far exceed what is available from any alternative implementation we know. The new high performance two-stage tile BRD achieves up to 30-fold speed up on a 16 core Intel Xeon machine with a 12000×12000 matrix size against the state-of-the-art open source and commercial numerical softwares i.e., multithreaded LAPACK compiled with optimized MKL BLAS and Intel MKL V10.2 (2.5 Gflop/s for both), respectively. Last but not least, it is noteworthy to mention that the overall performance of the two-stage tile BRD algorithm 40 Gflop/s represents only a small portion of the theoretical peak of the machine, roughly 25%. But considering the memory-bound nature of the second stage, the performance obtained is actually very encouraging and exceeds the expectation for such a type of algorithm.

One of the future projects in this direction will be the calculation of the singular vectors. For that, the orthogonal transformations from both stages need to be accumulated into U (left transformations) and V (right transformations). While the accumulation of the reflectors from the first stage is straightforward and can be implemented very efficiently, the second stage produces a tremendous number of small transformations which would add an $O(n^3)$ term in the overall complexity of the algorithm. This is still an open research problem and the authors are currently looking into removing this bottleneck. Finally, the authors are also investigating how this work can be extended to distributed environment systems within the DPLASMA framework [31].

References

[1] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," *Numer. Math.*, vol. 14, pp. 403–420, 1970.

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

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

[4] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *J. Educ. Psych.*, vol. 24, pp. 417–441, 498–520, 1933.

[5] ——, "Simplified calculation of principal components," *Psychometrica*, vol. 1, pp. 27–35, 1935.

[6] B. C. Moore, "Principal component analysis in linear systems: Controllability, observability, and model reduction," *IEEE Transactions on Automatic Control*, vol. AC-26, no. 1, February 1981.

[7] G. W. Stewart, "The decompositional approach to matrix computation," *Computing in Science & Engineering*, vol. 2, no. 1, pp. 50–59, Jan/Feb 2000, iSSN: 1521-9615; DOI 10.1109/5992.814658.

[8] A. S. Householder, "Unitary triangularization of a nonsymmetric matrix," *Journal of the ACM (JACM)*, vol. 5, no. 4, October 1958, dOI 10.1145/320941.320947.

[9] V. Fernando and B. Parlett, "Accurate singular values and differential qd algorithms," *Numerisch Math.*, vol. 67, pp. 191–229, 1994.

[10] E. Jessup and D. Sorensen, "A parallel algorithm for computing the singular value decomposition of a matrix," Argonne National Laboratory, Argonne, IL, Mathematics and Computer Science Division Report ANL/MCS-TM-102, December 1987.

[11] M. Gu and S. Eisenstat, "A divide-and-conquer algorithm for the bidiagonal SVD," *SIAM J. Mat. Anal. Appl.*, vol. 16, pp. 79–92, 1995.

[12] J. W. Demmel and W. Kahan, "Accurate singular values of bidiagonal matrices," *SIAM J. Sci. Stat. Comput.*, vol. 11, no. 5, pp. 873–912, September 1990, (Also LAPACK Working Note #3).

[13] P. Deift, J. W. Demmel, L.-C. Li, and C. Tomei, "The bidiagonal singular value decomposition and Hamiltonian mechanics," *SIAM J. Numer. Anal.*, vol. 28, no. 5, pp. 1463–1516, October 1991, (LAPACK Working Note #11).

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

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

[16] P. Luszczek, H. Ltaief, and J. Dongarra, "Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures," in *Proceedings of IPDPS 2011*. Anchorage, AK USA: ACM, 2011.

[17] H. Ltaief, J. Kurzak, and J. Dongarra, "Parallel two-sided matrix reduction to band bidiagonal form on multicore architectures." *IEEE Trans. Parallel Distrib. Syst.*, pp. 417–423, 2010.

[18] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE International Conference on*. Tsukuba International Congress Center, EPOCHAL TSUKUBA: IEEE, 29 2008-oct. 1 2008, pp. 142 –151.

[19] "SMP superscalar (SMPSs) user's manual, version 2.3," September 2008. [Online]. Available: http://www.bsc.es/media/3833.pdf

[20] C. H. Bischof, B. Lang, and X. Sun, "Algorithm 807: The sbr toolbox—software for successive band reduction," *ACM Trans. Math. Softw.*, vol. 26, no. 4, pp. 602–616, 2000.

[21] B. Kågström, D. Kressner, E. Quintana-Ortí, and G. Quintana-Ortí, "Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited," *BIT Numerical Mathematics*, vol. 48, pp. 563–584, 2008.

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

[23] "Intel, Math Kernel Library (MKL)," http://www.intel.com/software/products/mkl/, 2011, version 10.2.

[24] P. Bientinesi, F. Igual, D. Kressner, and E. Quintana-Ortí, "Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures," *Parallel Processing and Applied Mathematics*, vol. 6067, pp. 387–395, 2010.

[25] E. Agullo, J. Dongarra, R. Nath, and S. Tomov, "Autotuned dense QR factorization for multicore architectures," Institut National de Recherche en Informatique et en Automatique (INRIA), Tech. Rep. RR-7526, 2010, arXiv:1102.5328.

[26] Q. Yi, K. Kennedy, H. You, K. Seymour, and J. Dongarra, "Automatic blocking of qr and lu factorizations for locality," in *2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*. Washington, DC: ACM, 2004.

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

[28] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley, "The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines," *Scientific Programming*, vol. 5, pp. 173–184, 1996.

[29] E. D'Azevedo and P. Luszczek, "A framework for checkpointed fault-tolerant out-of-core linear algebra," in *SIAM Conference on Computational Science and Engineering (CSE03)*. Hyatt Regency Islandia Hotel and Marina, San Diego, CA: SIAM, February 10-13 2003.

[30] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra, "Analysis of Dynamically Scheduled Tile Algorithms for Dense Linear Algebra on Multicore Architectures," Innovative Computing Laboratory, University of Tennessee, Tech. Rep. ut-cs-11-666, 2011, submitted to Concurrency and Computations: Practice and Experience.

[31] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, J. K. Thomas Herault, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA," in *Accepted at the 12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-11)*. Anchorage, AK, USA: ACM, May 2011.