Parallel Computing 74 (2018) 3–18

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Accelerating the SVD two stage bidiagonal reduction and divide and conquer using GPUs

Mark Gates[a,*], Stanimire Tomov[a], Jack Dongarra[a,b,c]

[a] Innovative Computing Laboratory, University of Tenneessee, Knoxville, TN, USA
[b] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[c] University of Manchester, Manchester, England, United Kingdom

ARTICLE INFO

Article history:
Received 29 December 2016
Revised 19 July 2017
Accepted 24 October 2017
Available online 2 November 2017

Keywords:
Singular value decomposition
SVD
Divide and conquer
GPU
Accelerator

ABSTRACT

The increasing gap between memory bandwidth and computation speed motivates the choice of algorithms to take full advantage of today's high performance computers. For dense matrices, the classic algorithm for the singular value decomposition (SVD) uses a one stage reduction to bidiagonal form, which is limited in performance by the memory bandwidth. To overcome this limitation, a two stage reduction to bidiagonal has been gaining popularity. It first reduces the matrix to band form using high performance Level 3 BLAS, then reduces the band matrix to bidiagonal form. As accelerators such as GPUs and co-processors are becoming increasingly widespread in high-performance computing, a question of great interest to many SVD users is how much the employment of a two stage reduction, as well as other current best practices in GPU computing, can accelerate this important routine. To fulfill this interest, we have developed an accelerated SVD employing a two stage reduction to bidiagonal and a number of other algorithms that are highly optimized for GPUs. Notably, we also parallelize and accelerate the divide and conquer algorithm used to solve the subsequent bidiagonal SVD. By accelerating all phases of the SVD algorithm, we provide a significant speedup compared to existing multi-core and GPU-based SVD implementations. In particular, using a P100 GPU, we illustrate a performance of up to 804 Gflop/s in double precision arithmetic to compute the full SVD of a $20k \times 20k$ matrix in 90 seconds, which is $8.9\times$ faster than MKL on two 10 core Intel Haswell E5-2650 v3 CPUs, $3.7\times$ over the multi-core PLASMA two stage version, and $2.6\times$ over the previously accelerated one stage MAGMA version.

1. Introduction

The increasing gap between memory bandwidth and computation speed motivates the choice of algorithms to take full advantage of today's high performance computers. This gap causes matrix-matrix multiply (gemm)[1] to be 30–40 times faster than matrix-vector multiply (gemv) on today's architectures. For dense matrices, the classic algorithm for the singular value decomposition (SVD) uses a one stage reduction to bidiagonal form that requires matrix-vector multiplies, hence its

* Corresponding author.
E-mail addresses: mgates3@icl.utk.edu (M. Gates), tomov@icl.utk.edu (S. Tomov), dongarra@icl.utk.edu (J. Dongarra).

[1] Throughout, we have annotated BLAS and LAPACK function names in parenthesis, such as (gemm), so that readers familiar with the nomenclature can readily identify the operations, without the text requiring such knowledge.

https://doi.org/10.1016/j.parco.2017.10.004
0167-8191/

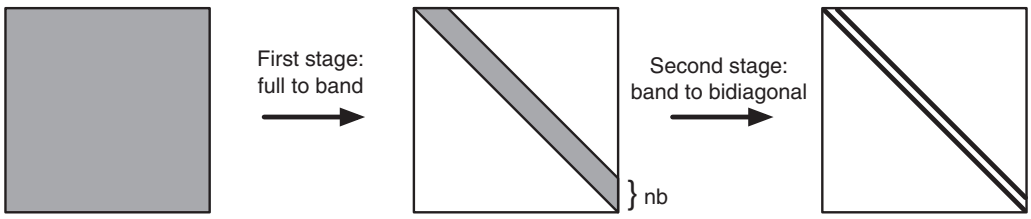


Fig. 1. Two stage reduction to bidiagonal form.

performance is limited by the memory bandwidth. To overcome this limitation, a two stage reduction to bidiagonal has been gaining popularity. It first reduces the matrix to band form using high performance Level 3 BLAS, then reduces the band matrix to bidiagonal form using optimized, cache-friendly kernels with dynamic scheduling. This removes the memory bandwidth limitation, decreases communication and synchronization overhead, and increases the computational intensity. As accelerators, such as GPUs and co-processors, are becoming increasingly widespread in high-performance computing, we have developed an accelerated SVD employing a two stage reduction to bidiagonal. We also parallelize and accelerate the divide and conquer algorithm used to solve the subsequent bidiagonal SVD.

The SVD of an $m \times n$ matrix A is given by

$$A = U \Sigma V^H,$$

where U and V are unitary and Σ is a real diagonal matrix with diagonal elements $\sigma_i \geq 0$. The σ_i are the *singular values* of A and the first $\min(m, n)$ columns of U and V are the *left* and *right singular vectors* of A, respectively. Without loss of generality, we assume $m \geq n$; the $m < n$ case is analogous but with operations transposed.

The classic Golub–Kahan–Reinsch algorithm [12,13] computes the SVD in three phases:

1. Reduce A to a bidiagonal matrix B via a unitary similarity transform, $A = U_1 B V_1^H$.
2. Compute the bidiagonal SVD as $B = U_2 \Sigma V_2^H$. Several algorithms exist for the bidiagonal SVD, the original being QR iteration.
3. If singular vectors are desired, back transform the singular vectors in U_2 and V_2 to yield $U = U_1 U_2$ and $V^H = V_2^H V_1^H$.

For modern machines with cache hierarchies, Dongarra et al. [10] improved the performance by blocking Householder reflectors together. This enabled half of the operations in the bidiagonal reduction to be performed in Level 3 BLAS [8] (matrix-matrix operations), which benefit from the *surface-to-volume* effect of having only $O(n^2)$ data to access for $O(n^3)$ operations. However, this blocking still leaves half of the operations of the bidiagonal reduction in memory-bandwidth-limited Level 2 BLAS [9] (matrix-vector operations), limiting its performance to twice the matrix-vector multiply (gemv) speed. Lahabar and Narayanan [24] had an early GPU-accelerated version of the one stage bidiagonal reduction and the subsequent bidiagonal SVD using QR iteration, while the current GPU-accelerated one stage implementation in MAGMA [20] is due to Tomov et al. [33]. These take advantage of the GPU's faster memory, but the algorithm remains memory bandwidth limited.

To remove this limitation, Großer and Lang introduced a two stage bidiagonal reduction, first reducing to band form, $A = U_a A_{\text{band}} V_a^H$ [14], then reducing to bidiagonal form, $A_{\text{band}} = U_b B V_b^H$ [25], as depicted in Fig. 1. While this incurs more operations than the one stage algorithm, most of the operations occur in the first stage, which uses high-performance Level 3 BLAS, making it much more efficient than the one stage bidiagonal reduction. Ltaief et al. implemented the first [26] and second stages [27] using tile algorithms with dynamic scheduling for multi-core CPUs in PLASMA [21], with later optimizations by Haidar et al. [16], 17]. Two stage algorithms have also been used for the reduction to tridiagonal form in the Hermitian eigenvalue problem [4], which have been accelerated with GPUs [3,18,19], and for the reduction to Hessenberg form in the non-symmetric eigenvalue problem [23]. In this paper, we accelerate the first stage using a GPU, while using the PLASMA CPU implementation for the second stage. For the bidiagonal reduction phase alone, this yields up to a 5.5 × improvement over the accelerated one stage bidiagonal reduction in MAGMA, and up to a 3.2 × improvement over the PLASMA two stage reduction, as described in Section 4.

A two stage reduction also requires using two corresponding back transformation steps, first multiplying $U_b U_2$ and $V_2^H V_b^H$, then multiplying $U_a(U_b U_2)$ and $(V_2^H V_b^H) V_a^H$. Having a second back transformation adds $4n^3$ operations, which fortunately are in Level 3 BLAS, but do reduce the potential speedup compared to a one stage algorithm when computing singular vectors. Section 5 describes our accelerated implementation of both stages of the back transformation.

In addition to improvements in the bidiagonal reduction, the subsequent bidiagonal SVD has also been addressed. The Golub–Reinsch algorithm used QR iteration. Bidiagonal divide and conquer (D&C) [15], based on Cuppen's divide and conquer algorithm [6] for the tridiagonal eigenvalue problem, and multiple relatively robust representations (MRRR) [34] were developed later. D&C improves performance when computing singular vectors in two ways. First, it reduces the bidiagonal SVD complexity to $\frac{8}{3}n^3$, or even $O(n^{2.3})$ or less [32], depending on deflation (see Section 7). This reduces the overall SVD operation count, including bidiagonal reduction and back-transformation of singular vectors, from $\approx 17n^3$ with QR iteration to $\approx 9n^3$ with D&C. Second, QR iteration performs $\approx 12n^3$ of its operations in Givens rotations, which are memory-bandwidth-limited Level 2 BLAS, while D&C performs most of its operations in Level 3 BLAS. MRRR improves performance by lowering

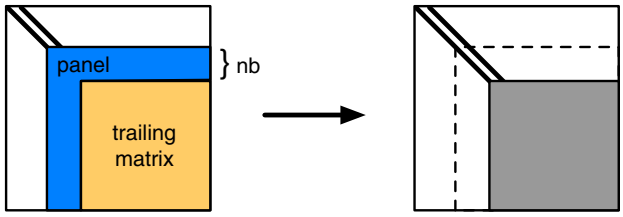


Fig. 2. One stage reduction to bidiagonal form brings each panel to bidiagonal form and updates the trailing matrix.

the bidiagonal SVD complexity to $O(n^2)$, reducing the overall SVD operation count to $\approx 7n^3$. Whether D&C or MRRR is faster depends on the distribution of singular values [34]. However, a stable version of MRRR for the SVD is not yet publicly available, e.g., in LAPACK. Therefore, we chose to examine D&C, which we first profiled to find both areas that can be parallelized on the CPU, and areas that can be accelerated with a GPU, as discussed in Section 7. For the D&C bidiagonal SVD phase alone, this yields a $2\times$ improvement using only the multi-core CPU, and a $3\times$ improvement when also using the GPU. The CPU-only improvements could be made available in LAPACK and PLASMA, without needing accelerators.

We have thus accelerated all phases of the SVD algorithm: bidiagonal reduction, bidiagonal SVD, and back transformation of singular vectors. In each stage, we chose an algorithm that emphasizes use of Level 3 BLAS to achieve high performance, and then modified the algorithm to use a GPU accelerator. When computing the complete SVD, including all three phases, we achieve speedups up to $8.9\times$ over the LAPACK implementation available in Intel MKL, $3.7\times$ over the multi-core PLASMA version, and $2.6\times$ over the previous accelerated one stage MAGMA version.

2. Test environment

All our tests are in double precision and use all 20 CPU cores. Test matrices have random entries that are uniform on (0,1), unless otherwise stated. Tests were run with `numactl --interleave=all` to distribute memory across the CPU sockets.

We use a machine with two 10 core Intel Haswell E5-2650 v3 CPUs at 2.3 GHz (turbo boost disabled), with 64 GiB memory and a peak speed of 736 Gflop/s. For matrix sizes up to $n = 20000$, the measured practical dgemm peak is 675 Gflop/s and dgemv peak is 19 Gflop/s (76GB/s). The STREAM triad benchmark [28] measured the memory bandwidth as 71GB/s. We use Intel MKL 11.3.0 [22] for vendor-optimized BLAS and LAPACK, and compile with Intel icc 16.0.

For an accelerator, we use an NVIDIA Pascal P100 GPU at 1.1 GHz, with 16 GiB memory, a peak speed of 4670 Gflop/s, and a peak memory bandwidth of 732GB/s. For matrix sizes up to $n = 20000$, the measured practical dgemm peak is 4572 Gflop/s and dgemv peak is 146 Gflop/s (584GB/s). We use NVIDIA cuBLAS 8.0 for GPU-accelerated BLAS, and the CUDA 8.0 compiler [29].

3. One stage reduction

As a comparison to motivate the two stage reduction, we first briefly sketch the one stage reduction, as implemented in the LAPACK routine gebrd. It iterates down the diagonal, at each step factoring a panel of width n_b that is a block column and block row, then updating the trailing matrix, as shown in Fig. 2. Within a panel, it uses a Householder reflector on the left to annihilate entries below the diagonal in each column, then another reflector on the right to annihilate entries right of the superdiagonal in each row, bringing that column and row to upper bidiagonal form. Data dependencies with the trailing matrix require doing a matrix-vector multiply (gemv) with the trailing matrix after factoring each column and each row, and using those results to update the next column and row prior to factoring. After factoring a panel, the trailing matrix is updated with two matrix-matrix multiplies (gemm). There are $\frac{4}{3}n^3$ operations in Level 2 BLAS gemv and $\frac{4}{3}n^3$ operations in Level 3 BLAS gemm. Thus, given that gemm performance is significantly faster than gemv performance, the potential performance is limited to twice the gemv speed. Algorithm 1 gives a high level overview; see Dongarra et al. [10] for further details.

4. Two stage reduction

4.1. First stage: General to band

The two stage bidiagonal reduction first reduces the matrix to upper band form, with a matrix upper bandwidth n_b, then reduces to upper bidiagonal form. The choice of n_b is discussed later in Section 6. In the first stage, reducing to band form eliminates data dependencies with the trailing matrix during the panel factorization, eliminating the matrix-vector operations during the panel that were a bottleneck. The first stage proceeds by doing a QR panel factorization of a block column to annihilate entries below the diagonal, updating the trailing matrix, then doing an LQ panel factorization of a block row to annihilate entries right of the matrix upper bandwidth, and updating the trailing matrix, as depicted in Fig. 3.

Algorithm 1 Blocked one stage bidiagonal reduction (gebrd). Householder reflectors use LAPACK representation $I - \tau v v^H$ with scalar τ and vector v.

for $i = 1$ to n by n_b
  $U = [\,]$; $V = [\,]$; $X = [\,]$; $Y = [\,]$; ▷ empty matrices
  for $j = 0$ to $n_b - 1$
    update column $A_{i+j:m,\ i+j}$ using Y and X
    generate reflector $(I - \tau_j v_j v_j^H)$ to eliminate entries in column $A_{i+j:m,\ i+j}$ below diagonal
    $y_j = A^H_{i+j:m,\ i:n} v_j$ (gemv)
    update row $A_{i+j,\ i+j+1:n}$ using Y and X
    generate reflector $(I - \pi_j u_j u_j^H)$ to eliminate entries in row $A_{i+j,\ i+j+1:n}$ right of superdiagonal
    $x_j = A_{i+j+1:m,\ i+j+1:n} u_j$ (gemv)
  end for
  update trailing matrix $A_{i+n_b:m,\ i+n_b:n} = A_{i+n_b:m,\ i+n_b:n} - VY^H - XU^H$ (gemms)
end for

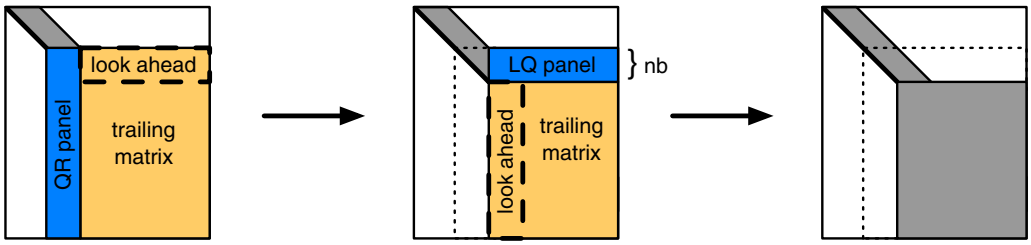


Fig. 3. One panel of first stage reduction to band form (gebrd_ge2gb). It does a QR factorization and trailing matrix update, then an LQ factorization and trailing matrix update, to bring each panel to upper band form.

Notice that the algorithmic block size (width/height of the QR/LQ panels) coincides with the matrix upper bandwidth, n_b. This stage costs $\frac{8}{3}n^3$ operations in Level 3 BLAS gemm.

The GPU accelerated version is outlined in Algorithm 2. As most of the operations occur in the trailing matrix update, the accelerated version stores the matrix on the GPU and performs this update on the GPU. At each step, the QR and LQ panels are copied to the CPU and factored on the CPU. The block Householder reflectors $Q_{(i)}$ and $P_{(i)}$ are represented in compact WY format [31] as $I - VTV^H$, where T is upper triangular and V is lower trapezoidal. To simplify the application of $Q_{(i)}$ and $P_{(i)}$ (larfb), zeros are stored explicitly in the upper triangle of the V matrices so that multiplying by V is a single matrix-matrix multiply (gemm), instead of a triangular matrix-matrix multiply (trmm) and gemm (as in LAPACK). This simplifies the code and makes it more efficient on the GPU for typical values of n_b. The panel, T, and V matrices are communicated between the CPU and GPU as needed.

Several optimizations can be made. An LQ factorization computes a Householder reflector of each row of the panel. As the matrices are stored in column-major order, this accesses a row of non-contiguous memory locations, reducing cache efficiency and causing the LQ panel factorization to be $1.5\times$ –$2.5\times$ slower than the QR panel factorization in our tests. To optimize this, we compute the LQ factorization by transposing the panel, performing a QR factorization, then transposing the result back.

In a one-sided factorization such as QR, where Q is applied on only the left side of A, a common optimization when updating the trailing matrix is to first update the next panel, called the *look-ahead panel*, then factor that panel in parallel with updating the remainder of the trailing matrix. In contrast, the SVD is a two-sided factorization, where Q is applied on the left and and P is applied on the right of A. This two-sided nature introduces data dependencies that restrict a look-ahead panel. In applying the trailing matrix update, $Q^H A = (I - VTV^H)^H A$, the routine larfb first computes $W^H = T^H(V^H A)$ using a matrix-matrix (gemm) and a triangular matrix (trmm) multiply. Because the next LQ panel is a block row, instead of a block column as in QR factorization, W^H must be computed for the entire trailing matrix before updating the look-ahead panel. It then updates $A = A - VW^H$ using another matrix-matrix multiply (gemm). For a look-ahead panel, we split this last multiply into two gemms by partitioning

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, \qquad V = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix},$$

where A_1 is the look-ahead panel as shown in Fig. 3, yielding $A_1 = A_1 - V_1 W^H$ and $A_2 = A_2 - V_2 W^H$. These correspond to gemm (1) and gemm (2) in Algorithm 2. The application of P on the right can be split similarly, corresponding to gemm (3) and gemm (4) in Algorithm 2. This look-ahead allows a partial overlap of the trailing matrix update (on the GPU) with the next QR or LQ panel factorization (on the CPU).

Algorithm 2 First stage: reduction to band form (gebrd_ge2gb). In block Householder representation, each V overwrites its panel in A; subscripts on V refer to location in A.

Input: Matrix A of size $m \times n$, with $m \geq n$, in CPU memory
Output: Vs representing Qs and Ps, and resulting band matrix A_{band} in CPU memory (overwriting A)

▷ W is workspace on accelerator of size $\max(m, n) \times n_b$

copy $A_{1:m,\ n_b:n}$ from CPU → accelerator
for $i = 1$ to n by n_b

▷ simultaneous with above copy or gemm (4) from previous iteration

QR factorization of block column on CPU: $Q_{(i)}R_{(i)} = A_{i:m,\ i:i+n_b-1}$ (geqrf)

if $i + n_b < n$ **then**
copy T and V that define $Q_{(i)}$ from CPU → accelerator

▷ Trailing matrix update $A := Q_{(i)}^H A = (I - VTV^H)^H A$ (larfb)

[on accelerator]: $W^H = T^H(V^H_{i:m,\ i:i+n_b-1} A_{i:m,\ i+n_b:n})$ (gemm, trmm)
[on accelerator]: $A_{i:i+n_b-1,\ i+n_b:n} \mathrel{-}= V_{i:i+n_b-1,\ i:i+n_b-1} W^H$ (gemm (1); look-ahead panel)
[on accelerator]: $A_{i+n_b:m,\ i+n_b:n} \mathrel{-}= V_{i+n_b:m,\ i:i+n_b-1} W^H$ (gemm (2))

▷ simultaneous with gemm (2) above

▷ LQ factorization done by transpose of panel, QR factorization, then transpose back

copy block row $A_{i:i+n_b-1,\ i+n_b:n}$ for LQ panel from accelerator → CPU
LQ factorization of block row on CPU: $L_{(i)}P_{(i)} = A_{i:i+n_b-1,\ i+n_b:n}$ (gelqf)
copy T and V that define $P_{(i)}$ from CPU → accelerator

▷ Trailing matrix update $A := AP_{(i)} = A(I - V^H TV)$

[on accelerator]: $W = (A_{i+n_b:m,\ i+n_b:n} V^H_{i:i+n_b-1,\ i+n_b:n})T$ (gemm, trmm)
[on accelerator]: $A_{i+n_b:m,\ i+n_b:i+2n_b-1} \mathrel{-}= WV_{i:i+n_b-1,\ i+n_b:i+2n_b-1}$ (gemm (3); look-ahead panel)
[on accelerator]: $A_{i+n_b:m,\ i+2n_b:n} \mathrel{-}= WV_{i:i+n_b-1,\ i+2n_b:n}$ (gemm (4))

▷ simultaneous with gemm (4) above

copy block column $A_{i+n_b:m,\ i+n_b:i+2n_b-1}$ for QR panel from accelerator → CPU
end if
end for

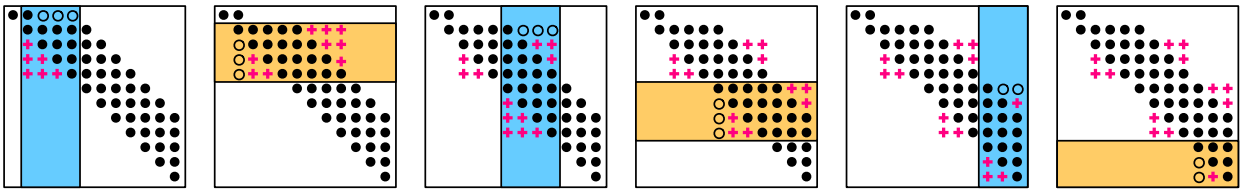


Fig. 4. One sweep of second stage: band to bidiagonal form (gebrd_gb2bd), with matrix upper bandwidth $n_b = 4$. Sweep i reduces row i to bidiagonal, then chases the resulting bulge down the matrix. 'o' indicates an annihilated entry, '+' indicates fill, and a shaded band indicates application of a reflector.

Alternatively to using look-ahead panels, if efficient native GPU-only QR and LQ factorizations are developed, the panels could be moved to the GPU. Otherwise, the panels are not wide enough to benefit from a hybrid CPU–GPU implementation. (For the current MAGMA hybrid QR factorization, at $n = 10000$ a panel of size $n_b \geq 1024$ would be required to be competitive with the CPU QR factorization, far larger than the optimal n_b found in Section 6.)

4.2. Second stage: Band to bidiagonal

The second stage reduces the upper band matrix to upper bidiagonal form. For this stage, we use the implementation by Haidar et al. [17], available in PLASMA 2.8 [21]. As this stage has limited parallelism, is close to memory bandwidth limited, and is already optimized for the CPU caches, it would not benefit much, if any, from an accelerator-based implementation.

It proceeds in $n - 2$ sweeps, where sweep i reduces row i to bidiagonal, as illustrated in Fig. 4. In a sweep, we first apply a Householder reflector on the right to annihilate elements right of the superdiagonal in row i, which also introduces fill below the diagonal in columns $i + 1$ to $i + n_b - 1$, known as a bulge (see Fig. 4). Applying a Householder reflector on the left annihilates entries below the diagonal in column $i + 1$, which in turn creates a bulge above the superdiagonal in rows $i + 1$ to $i + n_b - 1$. This pattern continues until the bulge disappears off the bottom-right of the matrix, hence the process is

termed "bulge chasing." The next sweep repeats this pattern, shifted down one row and right one column, to bring the next row to bidiagonal. The PLASMA implementation pays particular attention to doing this in parallel and keeping data cache resident for an efficient implementation.

This stage adds $O(n^2 n_b)$ more operations that did not occur in the original one stage reduction to bidiagonal. However, this is more than offset by the increased performance of the first stage compared to the one stage algorithm. Tuning is important, though, to balance the cost of the first and second stages. Section 6 will investigate tuning the matrix bandwidth.

5. Computation of singular vectors

As the back-transformation of singular vectors is closely related to the bidiagonal reduction, we will cover it first, and cover divide and conquer later in Section 7. The bidiagonal SVD computes the $n \times n$ matrices U_2 and V_2, which are the singular vectors of the bidiagonal matrix B. With the one stage bidiagonal reduction, the computation of singular vectors involves back transforming those to be singular vectors of A by multiplying with the unitary matrices U_1 and V_1 from the bidiagonal reduction:

$$\begin{aligned} A &= U_1 B V_1^H = U_1 (U_2 \Sigma V_2^H) V_1^H = U \Sigma V^H, \\ U &= U_1 U_2, \\ V^H &= V_2^H V_1^H. \end{aligned}$$

The U_1 and V_1 matrices are stored implicitly as a collection of vectors in compact WY format. Application of these is performed in $4n^3$ flops using the LAPACK routine ormbr.

The two stage bidiagonal reduction introduces an extra back transformation step:

$$\begin{aligned} A &= U_a A_{\text{band}} V_a^H = U_a (U_b B V_b^H) V_a^H = U_a U_b (U_2 \Sigma V_2^H) V_b^H V_a^H = U \Sigma V^H, \\ U &= U_a U_b U_2, \\ V^H &= V_2^H V_b^H V_a^H, \end{aligned}$$

requiring multiplication by both U_a and U_b for U, and by V_a and V_b for V, costing $8n^3$ operations total. We introduce two new routines: ormbr_ge2gb to multiply by U_a and V_a, the unitary matrices from the first stage; and ormbr_gb2bd to multiply by U_b and V_b, the unitary matrices from the second stage.

For the first stage back transformation, $U_a = Q_{(1)} \cdots Q_{(n/n_b)}$ and $V_a = P_{(1)} \cdots P_{(n/n_b)}$ with $Q_{(i)}$ and $P_{(i)}$ from Algorithm 2. Multiplying a matrix by U_a is exactly the same as multiplying by Q from a QR factorization—we simply call the existing routine ormqr. Multiplying by V_a is the same as multiplying by Q from an LQ factorization, except shifted to the right by n_b columns, so we call the existing routine ormlq with the appropriate submatrix. Both ormqr and ormlq have existing accelerated versions in MAGMA. They operate by looping over the $Q_{(i)}$ or $P_{(i)}$ and applying them as block Householder reflectors (larfb) on the GPU.

For the second stage back transformation, we update the scheme from Haidar et al. [19] for the two stage tridiagonal reduction, by extending it to the bidiagonal reduction, which requires applying V_b^H on the right. The updated scheme is given in Algorithm 3 . The Householder vectors that define U_b and V_b are shown in Fig. 5(a) and (e). For instance, the three Householder reflectors applied on the left in sweep 1 (see Fig. 4) are represented by vectors in column 1 of Fig. 5(a), while the three reflectors applied on the right in sweep 1 are in row 1 of Fig. 5(e). We block j_b vectors together into $V_{(k)}$ matrices defining block Householder reflectors, forming lozenge shapes shown in Fig. 5(b) and (f). Application of these lozenge shapes overlap—for instance, block reflectors 3 and 4 modify two overlapping rows in Fig. 5(c). This creates the dependencies shown in Fig. 5(b) and (f). The reflectors can be applied in any order that respects these dependencies. Currently, we loop over the block columns, from right to left, and then over the lozenges from top to bottom within a block column, shown by the numbering in Fig. 5(b) and (f). We modified the PLASMA code for the second stage reduction to store the lozenges in the order to be applied, rather than the order they are created, as shown in Fig. 5(d) and (h). This allows us to send a set of lozenges together to the GPU in one data transfer, then loop over them to apply them. We use double buffering to overlap data transfers and applications of the reflectors. As with the previous V matrices, the lozenges are stored with explicit zeros, to use a single matrix-matrix multiply (gemm) instead of two triangular matrix multiplies (trmm) and a matrix-matrix multiply (gemm).

6. Tuning

Selecting the optimal matrix bandwidth n_b for a particular hardware platform is an import aspect of achieving good performance with the two stage algorithm. The first stage reduction and both the first and second stage back transformations favor larger n_b, because the matrix multiplies become larger and more efficient. This is shown in Fig. 6(a) and (c) for the GPU accelerated version, and Fig. 7(a) and (c) for the PLASMA CPU version, where each line is a different problem size solved with varying n_b. In contrast, since the second stage (band to bidiagonal) must do $O(n^2 n_b)$ work, it favors a small n_b to reduce the amount of computation, as shown in Figs. 6(b) and 7(b). The optimal n_b is a compromise between these competing factors.

Algorithm 3 Second stage back transformation (ormbr_gb2bd).

```
                                                  ▷ Sends set of n_s V_(k) matrices at a time
s = 0                                             ▷ index of set
k = n_s + 1                                       ▷ index within set; force sending first set
for j = ⌈(n−2)/j_b − 1⌉ j_b + 1 down to 1 by j_b
    for i = j to n − 1 by n_b
        if k > n_s then
            for k = 1 to n_s
                Compute T_(sn_s+k) for V_(sn_s+k) (larft)
            end for
            copy set s of T_(sn_s+1:(s+1)n_s) from CPU → dK_(1:n_s) on accelerator
            copy set s of V_(sn_s+1:(s+1)n_s) from CPU → dV_(1:n_s) on accelerator
            k = 1
            s = s + 1
        end if
        if applying U_b then
            Apply reflectors defined by dT_(k) and dV_(k) to block row C_(i:i+n_b+j_b−1, 1:n) on accelerator (larfb)
        else
            Apply reflectors defined by dT_(k) and dV_(k) to block col C_(1:n, i:i+n_b+j_b−1) on accelerator (larfb)
        end if
        k = k + 1
    end for
end for
```

The overall SVD time for the singular values only (no vectors) case is shown in Figs. 6(d) and 7(d) for various problem sizes. This is dominated by the time for the bidiagonal reduction, since the bidiagonal SVD using QR iteration is of lesser order, $O(n^2)$, in this case. The large marker on each line indicates the optimal n_b for that problem size. As expected, smaller problem sizes tend to have smaller optimal n_b. For the accelerated version, $n_b = 32$ is optimal for $n \le 10000$, and $n_b = 64$ is optimal for larger sizes. Because PLASMA's first stage is more expensive, it has a larger optimal n_b: 96 for small problems, and up to 160 for $n \ge 8000$. Small $n_b \le 64$ are particularly slow for PLASMA due to its first stage reduction to band. In many cases, near optimal n_b are within 20% of the optimal time, as shown in Figs. 6(f) and 7(f), with a notable exception for PLASMA of $n = 2000$, which has a sharply defined optimum at $n_b = 96$.

When singular vectors are also computed, the first and second stage back transformations contribute. As they favor large n_b, the optimal n_b in Figs. 6(e) and 7(e) increases compared to the no-vectors case, for both the accelerated version and for PLASMA. The accelerated version has an optimal n_b of 96 for small problem sizes and 128 for large problems. PLASMA's optimal n_b ranges from 96 to 288, with 224 being a good all-around size. More so than in the no-vectors case, the time is not sensitive to the optimal n_b, with many sub-optimal n_b still within 5% or 10% of the optimal time, as shown in Figs. 6(g) and 7(g). For instance, with PLASMA at $n = 14000$, the times for $n_b = 160$ to 288 are all within 2.4% of the optimal time, explaining why its empirically optimal n_b is randomly larger than the optimal n_b for nearby values of n. However, the sensitivity of the optimal n_b is implementation and platform dependent, as is clear from the differences between tuning the accelerated and PLASMA versions.

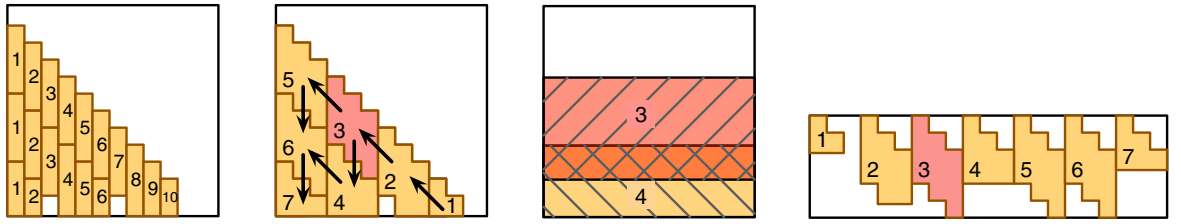
7. Divide and conquer

After reducing to bidiagonal form, the divide and conquer procedure computes the SVD of the bidiagonal matrix B. Gu and Eisenstat [15] derived a stable procedure; we will review the salient features following their derivation, except transposed to match the code and our assumption that $m \ge n$. Let B be an $n \times (n+1)$ upper bidiagonal matrix. To solve an $n \times n$ bidiagonal matrix, simply append a zero column. Partition the matrix B as

$$B = \begin{bmatrix} B_1 & 0 \\ \alpha e_k^T & \beta e_1^T \\ 0 & B_2 \end{bmatrix},$$

where B_1 and B_2 are upper bidiagonal matrices of size $(k-1) \times k$ and $(n-k) \times (n-k+1)$, respectively, typically with $k = \lfloor n/2 \rfloor$, and e_k is the kth column of an identity matrix. The SVDs of B_1 and B_2 are computed recursively as $B_i = W_i \begin{bmatrix} D_i & 0 \end{bmatrix} \begin{bmatrix} Q_i & q_i \end{bmatrix}^T$. For the base case, when B_i is small enough ($n \le 25$), its SVD is computed by QR iteration. To compute the SVD of B from that of B_1 and B_2, first, singular values that have already converged are separated to reduce the problem size, a process known as deflation, yielding a matrix of the form

$$B = \begin{bmatrix} \tilde{W} & W_d \end{bmatrix} \begin{bmatrix} M & 0 & 0 \\ 0 & \Omega_d & 0 \end{bmatrix} \begin{bmatrix} \tilde{Q} & Q_d & q \end{bmatrix}^T, \tag{1}$$

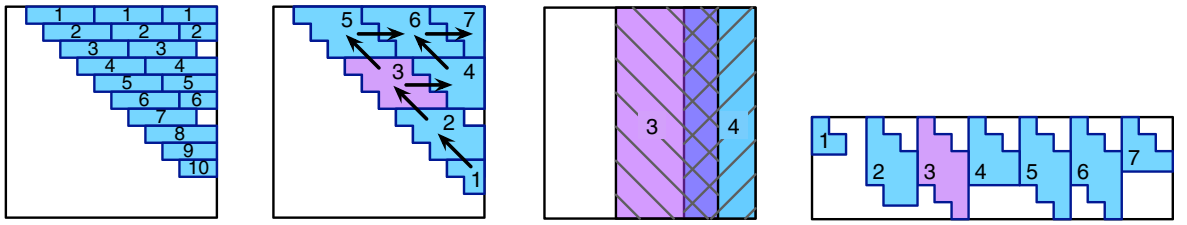


(a) U_b Householder vectors, numbered by sweep.

(b) Arrows indicate dependencies between blocks of U_b vectors, numbered by application order.

(c) Application of block reflectors 3 and 4 to U overlaps.

(d) Blocks of U_b vectors, stored in application order.



(e) V_b Householder vectors, numbered by sweep.

(f) Arrows indicate dependencies between blocks of V_b vectors, numbered by application order.

(g) Application of block reflectors 3 and 4 to V overlaps.

(h) Blocks of V_b vectors, stored transposed in application order.

Fig. 5. Second stage back transformation, with V block size $j_b = 3$ vectors. Block reflector 3 is highlighted to show overlap.

where

$$\tilde{W} = \begin{bmatrix} 0 & \tilde{W}_{0,1} & \tilde{W}_1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & \tilde{W}_{0,2} & 0 & \tilde{W}_2 \end{bmatrix}, \qquad \tilde{Q} = \begin{bmatrix} \tilde{Q}_{0,1} & \tilde{Q}_1 & 0 \\ \tilde{Q}_{0,2} & 0 & \tilde{Q}_2 \end{bmatrix}, \tag{2}$$

$\tilde{W}_i$, $\tilde{W}_{0,i}$, $\tilde{Q}_i$, and $\tilde{Q}_{0,i}$ are derived from W_i, Q_i, and q_i by the deflation process, Ω_d are deflated singular values, and W_d, Q_d, are deflated singular vectors; see Gu and Eisenstat [15] for details. Compute the SVD of $M = U\Omega V^T$, as described below in Section 7.1, and substitute into (1) to yield the SVD of B:

$$B = \begin{bmatrix} \tilde{W}U & W_d \end{bmatrix} \begin{bmatrix} \Omega & 0 & 0 \\ 0 & \Omega_d & 0 \end{bmatrix} \begin{bmatrix} \tilde{Q}V & Q_d & q \end{bmatrix}^T.$$

Taking advantage of the block structure in (2), compute the updated singular vectors $\tilde{W}U$ and $\tilde{Q}V$ with three matrix-matrix multiplies (gemm) each:

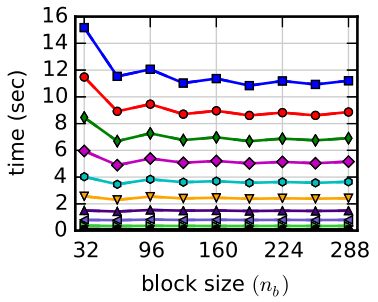
$$\tilde{W}U = \begin{bmatrix} \tilde{W}_{0,1}U_0 + \tilde{W}_1U_1 \\ u_0^T \\ \tilde{W}_{0,2}U_0 + \tilde{W}_2U_2 \end{bmatrix}, \text{ where } U = \begin{bmatrix} u_0^T \\ U_0 \\ U_1 \\ U_2 \end{bmatrix}; \tag{3}$$

$$\tilde{Q}V = \begin{bmatrix} \tilde{Q}_{0,1}V_0 + \tilde{Q}_1V_1 \\ \tilde{Q}_{0,2}V_0 + \tilde{Q}_2V_2 \end{bmatrix}, \text{ where } V = \begin{bmatrix} V_0 \\ V_1 \\ V_2 \end{bmatrix}. \tag{4}$$

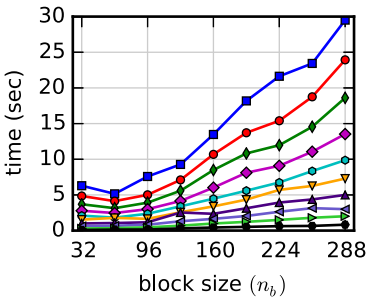
7.1. SVD of M

The matrix M has the special structure:

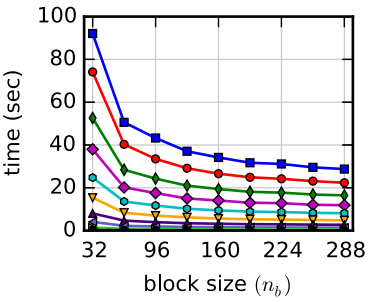
$$M = \begin{bmatrix} z_1 & z_2 & \dots & z_m \\ & d_2 & & \\ & & \ddots & \\ & & & d_m \end{bmatrix},$$



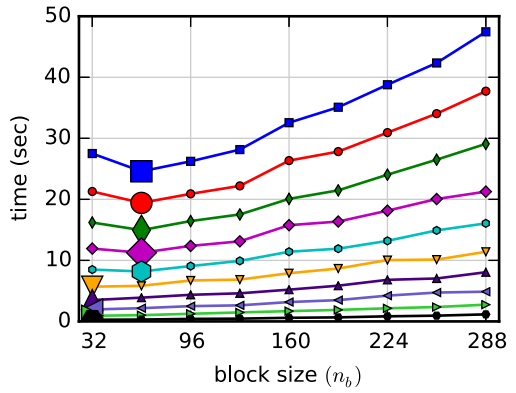
(a) 1st stage: full to band



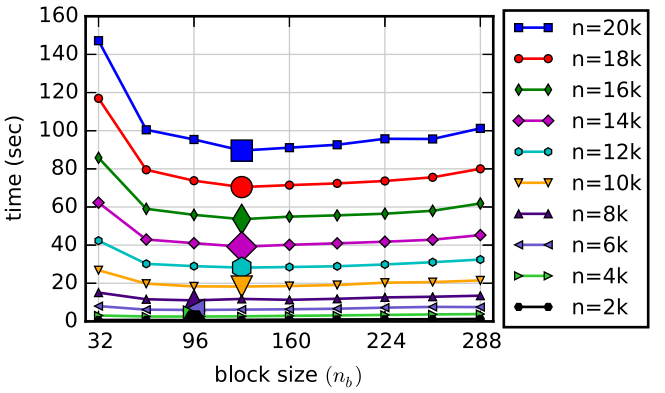
(b) 2nd stage: band to bidiag



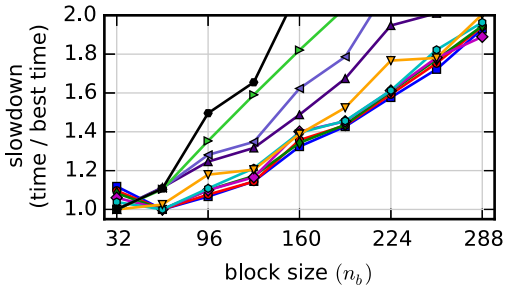
(c) 1st & 2nd stage back transform



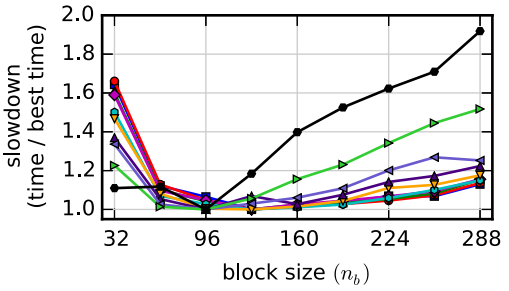
(d) SVD, no vectors. Includes time from (a,b), plus bidiagonal SVD solve (no vectors).



(e) SVD, with singular vectors. Includes time from (a,b,c), plus bidiagonal SVD solve (D&C with vectors).



(f) Time from (d) divided by best time for each size.



(g) Time from (e) divided by best time for each size.

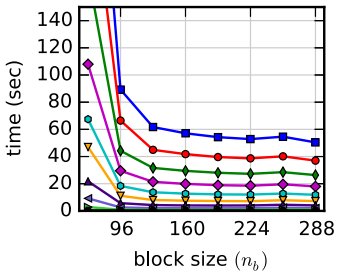
Fig. 6. Tuning of GPU-accelerated 2-stage algorithm with varying n_b. Each line is a different problem size, with the optimal time highlighted by large marker. The bidiagonal SVD solve is independent of n_b.

where m is the number of non-deflated singular values. The singular values ω_i of M are the roots of the secular equation,

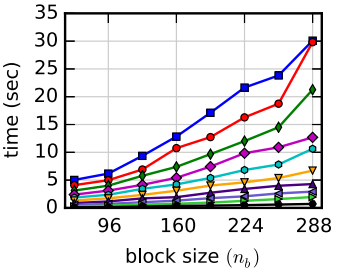
$$f(\omega_i) = 1 + \sum_{k=1}^{m} \frac{z_k^2}{d_k^2 - \omega_i^2} = 0. \tag{5}$$

While the computed singular values, $\tilde{\omega}_i$, have high relative accuracy, the small approximation error incurred would cause the computed singular vectors to lose orthogonality. Instead, to ensure stability and orthogonality of the singular vectors, Gu and Eisenstat [15] compute a new matrix $\tilde{M}$ in the same form as M, for which the computed $\tilde{\omega}_i$ are the *exact* singular values, with

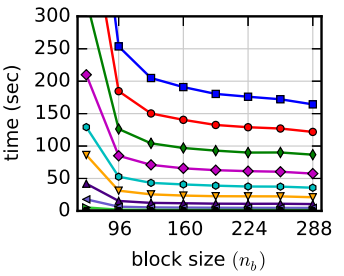
$$|\tilde{z}_i| = \sqrt{(\tilde{\omega}_m^2 - d_i^2) \prod_{j=1}^{i-1} \frac{\tilde{\omega}_j^2 - d_i^2}{d_j^2 - d_i^2} \prod_{j=i}^{m-1} \frac{\tilde{\omega}_j^2 - d_i^2}{d_{j+1}^2 - d_i^2}}. \tag{6}$$



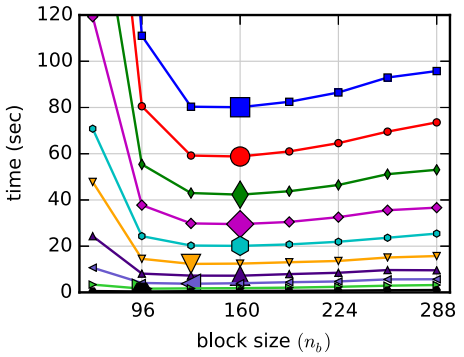
(a) 1st stage: full to band



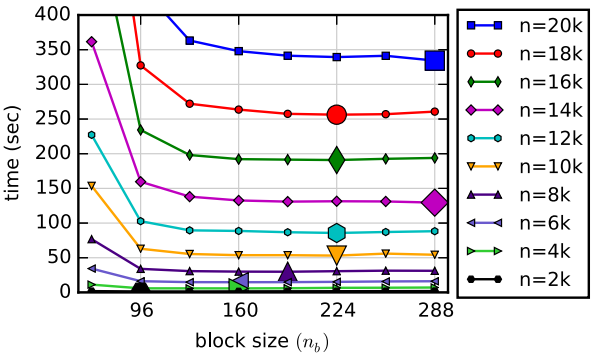
(b) 2nd stage: band to bidiag



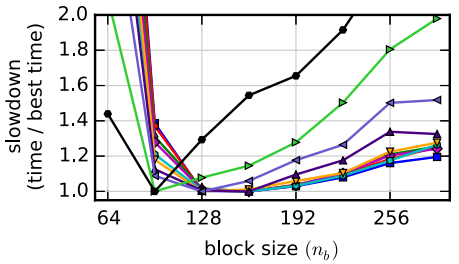
(c) 1st & 2nd stage back transform



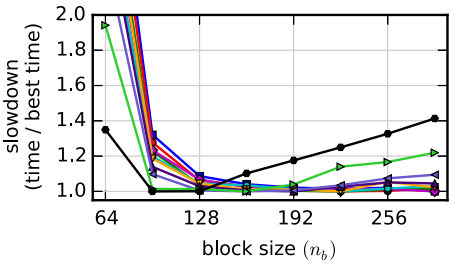
(d) SVD, no vectors. Includes time from (a,b), plus bidiagonal SVD solve (no vectors).



(e) SVD, with singular vectors. Includes time from (a,b,c), plus bidiagonal SVD solve (D&C with vectors).



(f) Time from (d) divided by best time for each size.



(g) Time from (e) divided by best time for each size.

Fig. 7. Tuning of PLASMA multi-core implementation of 2-stage algorithm. Tests with $n_b = 32$ were very slow and clearly not optimal, so were excluded.

The left and right singular vectors of $\tilde{M}$ are then computed as

$$u_i = \left[-1, \frac{d_2 \tilde{z}_2}{d_2^2 - \tilde{\omega}_i^2}, \ldots, \frac{d_m \tilde{z}_m}{d_m^2 - \tilde{\omega}_i^2} \right]^T, \tag{7}$$

$$v_i = \left[\frac{\tilde{z}_1}{d_1^2 - \tilde{\omega}_i^2}, \ldots, \frac{\tilde{z}_m}{d_m^2 - \tilde{\omega}_i^2} \right]^T, \tag{8}$$

and normalized so $\|u_i\|_2 = 1$ and $\|v_i\|_2 = 1$.

7.2. Accelerated version

There are several potential sources of parallelism in the D&C algorithm. For instance, in the recursion tree, each subproblem is independent. We profiled it in Fig. 8 to identify areas for optimization. Most of the time is spent in the merge step, in particular, in the matrix multiplies $\tilde{W}U$ and $\tilde{Q}V$. Further, most time is spent in the top couple levels of the recursion tree, near the root, rather than in the leaf nodes. Therefore, we focus on the merge step, which is performed in the LAPACK routine lasd3.

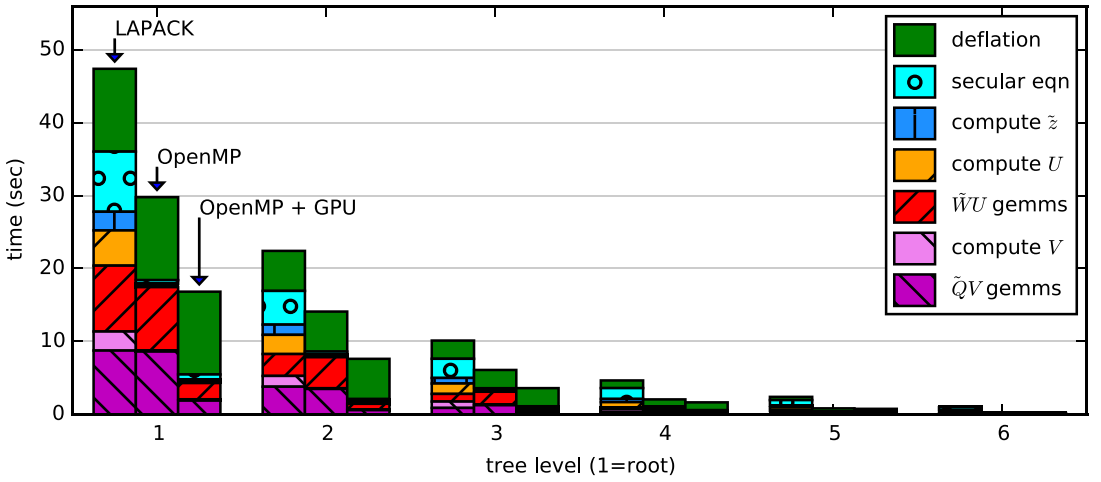


Fig. 8. Profile of divide and conquer for $n = 20000$, showing time spent at each level of the recursion tree, for three different implementations.

Algorithm 4 Merge step of divide and conquer algorithm (lasd3).

[on accelerator]: copy Q from CPU → accelerator
[on accelerator]: copy W from CPU → accelerator
▷ simultaneous with above copies

parallel for $i = 1$ to m
  compute ω_i by solving (5)
end for
parallel for $i = 1$ to m
  compute z_i by (6)
end for
parallel for $i = 1$ to m
  compute v_i by (8)
  compute u_i from v_i by (7)
  normalize u_i
end for
[on accelerator]: copy U from CPU → accelerator
[on accelerator]: $U = WU$ as 3 gemms by (3)
[on accelerator]: copy U from accelerator → CPU
▷ simultaneous with above accelerator gemms

parallel for $i = 1$ to m
  normalize v_i
end for

[on accelerator]: copy V from CPU → accelerator
[on accelerator]: $V = QV$ as 3 gemms by (4)
[on accelerator]: copy V from accelerator → CPU

For our implementation, given in Algorithm 4, it is noted that Eqs. (5)–(8) are each loops over m independent values, which can thus be implemented as parallel for loops with OpenMP. Synchronizations are needed after (5) and (6). For $j > 1$, $u_{ij} = d_j v_{ij}$, so the loops to compute u_i and v_i are merged. After computing U by (7), we finish normalizing V on the CPU, in parallel with multiplying $\tilde{W}U$ (3) on the accelerator, then multiply $\tilde{Q}V$ (4) on the accelerator. Care is taken to overlap CPU computation with accelerator–CPU communication and accelerator computation where possible.

The profile in Fig. 8 shows that the OpenMP version shrinks the parallel loops noticeably, leaving just the deflation and gemms as major contributors. The accelerated version improves the speed of the gemms, leaving the deflation process as the dominant factor for further optimization. The D&C performance is shown in Fig. 9. Here we assume D&C takes $\frac{8}{3}n^3$ operations; the actual operation count may be lower due to deflation. D&C is 2–3 times faster than the QR iteration algorithm for sizes $n \geq 5000$. Using OpenMP to parallelize Eqs. (5)–(8) almost doubles the D&C performance, from 220 Gflop/s to 426 Gflop/s at $n = 20000$. These OpenMP improvements of course do not depend on an accelerator, so could be incorporated into existing CPU libraries such as LAPACK. Performing the $\tilde{W}U$ and $\tilde{Q}V$ products on the accelerator yields an additional 1.6 ×

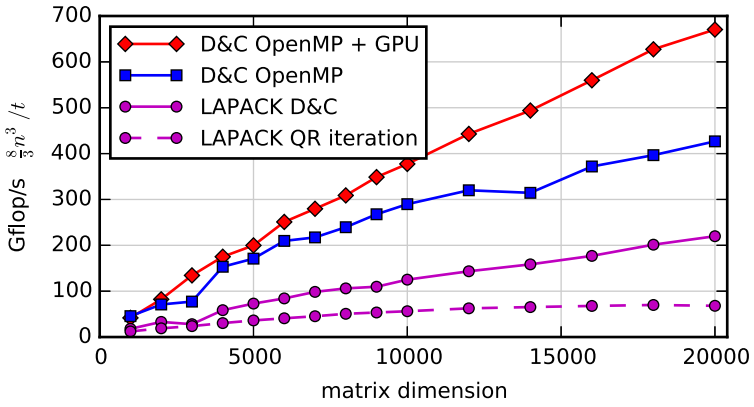


Fig. 9. Divide and conquer performance, using $\frac{8}{3}n^3$ estimate for operation count in all cases.

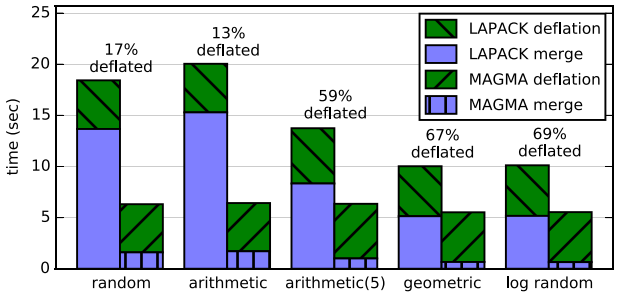


Fig. 10. Divide and conquer performance for various test matrices of size $n = 10000$. Percentage of singular values that were deflated at the root level is annotated above each set of bars.

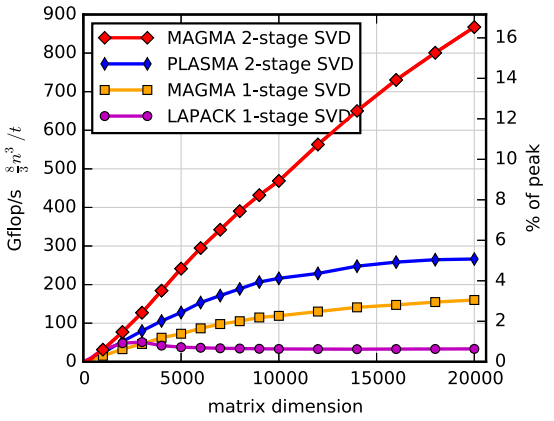
speedup of D&C, up to 670 Gflop/s. For sizes $n \leq 5000$, the speedup of the GPU version over the OpenMP version is generally less, about 15%, while it gradually increases for larger sizes.

7.3. Additional test matrices

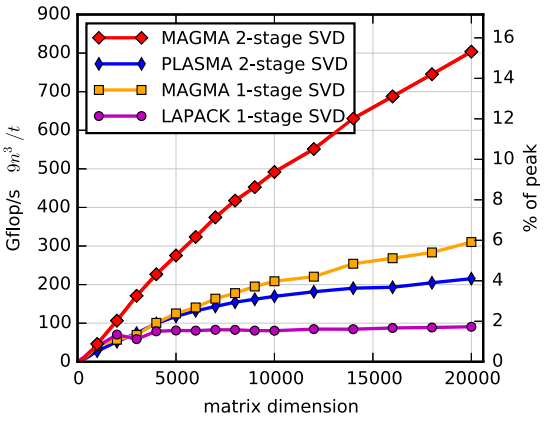
The performance of divide and conquer depends on the amount of deflation that occurs, which increases with clustered singular values. We tested five matrices for differences in performance:

random: matrix entries are random uniform on (0, 1); default test case in this paper.
arithmetic: singular values are arithmetically distributed: $\sigma_i = 1 - \frac{i-1}{n-1}(1-\epsilon)$.
arithmetic(5): like arithmetic, but repeats each value 5 times: $\sigma_i = 1 - \frac{\lfloor (i-1)/5 \rfloor}{\lfloor (n-1)/5 \rfloor}(1-\epsilon)$.
geometric: singular values are geometrically distributed: $\sigma_i = \left(\frac{1}{\epsilon}\right)^{-(i-1)/(n-1)}$.
log random: singular values are random in $(\epsilon, 1)$ such that their logarithms are random uniform.

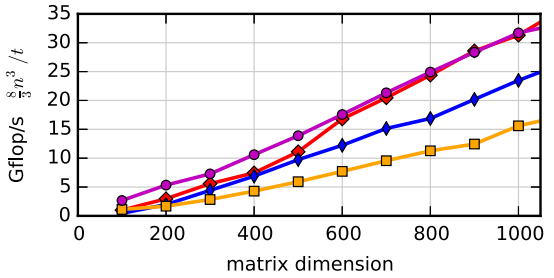
The random matrix is generated by the LAPACK routine larnv; its singular values and condition number are unknown *a priori*. The other matrices are generated by the LAPACK testing routine latms, which multiplies a prescribed set of singular values by random orthogonal matrices on the left and right to generate a test matrix. The condition number is set to $\frac{1}{\epsilon}$, where $\epsilon = 2.22 \times 10^{-16}$ is machine precision. The D&C times for these five matrices are shown in Fig. 10, with the percentage of deflated singular values annotated. For the LAPACK results, we see that the arithmetic test has slightly less deflation and is slightly slower (5%) than the random test. Clusters of repeated singular values in the arithmetic(5) test causes significant deflation and is 26% faster than the random test, while the geometric and log random tests had even more deflation and are 47% and 50% faster, respectively, than the random test. Compared to LAPACK, the accelerated MAGMA version achieves speedups in all cases. However, it benefits much less from deflation, with the random, arithmetic, and arithmetic(5) cases being nearly identical, and the geometric and log random cases being 11% faster than the random test. While the merge step (Algorithm 4 and bottom blue tier in Fig. 10) does benefit from deflation, the deflation step itself now dominates the time, so future work should focus on optimizing that step. Despite deflation, the arithmetic(5) case is about the same time as the random case because, while arithmetic(5) has a shorter merge step, it has a longer deflation step, so there is no overall savings compared to the random test.



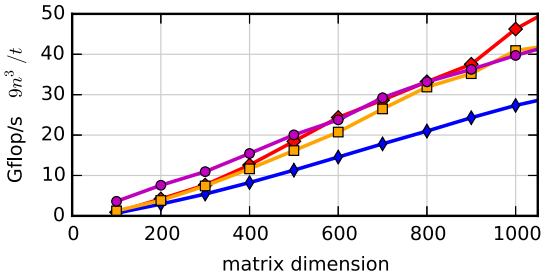
(a) Singular values only (no vectors), using $\frac{8}{3}n^3$ for operation count in all cases. Compares to CPU + GPU gemm peak of 5247 Gflop/s.



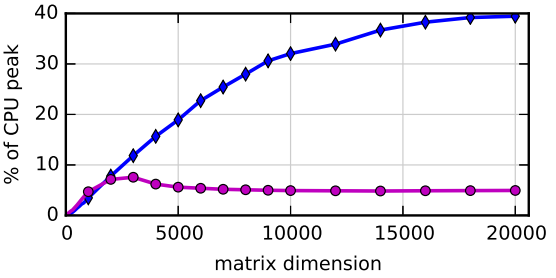
(b) With singular vectors, using $9n^3$ for operation count in all cases.



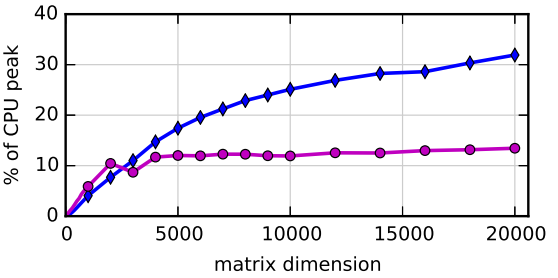
(c) Zoom of (a) to show crossovers at small sizes.



(d) Zoom of (b) to show crossovers at small sizes.



(e) % of CPU gemm peak (675 Gflop/s) for PLASMA and LAPACK from (a).



(f) % of CPU peak for PLASMA and LAPACK from (b).

Fig. 11. SVD performance using optimal n_b from Section 6 for each problem size.

8. Combined results

We have now accelerated all three phases of the SVD algorithm:

1. reduction to bidiagonal, using a two stage algorithm;
2. bidiagonal SVD, using divide and conquer;
3. back transformation of singular vectors.

In Fig. 11 we compare the performance of the one stage algorithm in LAPACK, the previously accelerated one stage algorithm in MAGMA, the two stage algorithm in PLASMA, and the newly developed accelerated two stage algorithm, which will be released in an upcoming version of MAGMA. The PLASMA and MAGMA two stage algorithms use the optimal n_b determined in Section 6 for each size. Fig. 11(a) is the singular values only (no vectors) case, which is predominantly the bidiagonal reduction. To compute Gflop/s, we use $\frac{8}{3}n^3$ operations for all cases, though the two stage algorithms perform an additional $O(n^2 n_b)$ operations. The one stage MAGMA has a speedup over LAPACK up to 4.8 ×, reflecting the increased

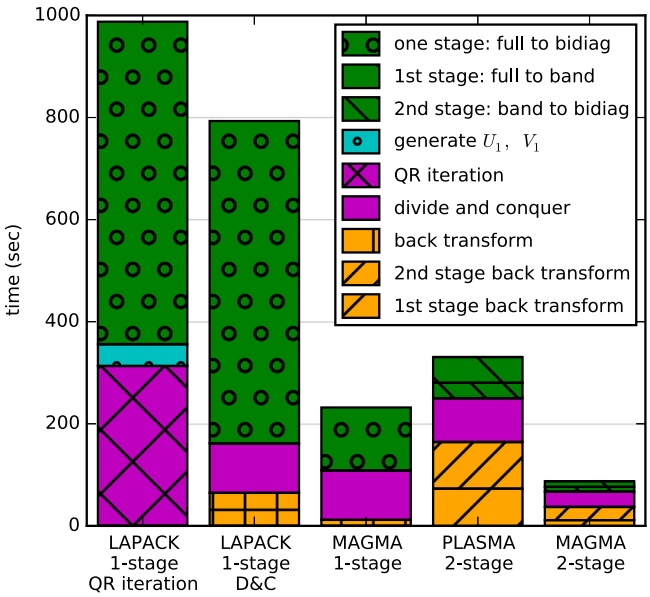


Fig. 12. Profile of SVD implementations showing each phase, for $n = 20000$. With QR iteration, it first explicitly generates U_1 and V_1, then multiplies them by U_2 and V_2 during QR iteration, whereas D&C explicitly generates U_2 and V_2, and subsequently multiplies them by implicit U_1 and V_1 in back transformation.

memory bandwidth of the GPU. PLASMA has a speedup over LAPACK up to $8\times$, showing the superiority the two stage algorithm gains by using Level 3 BLAS. The new two stage MAGMA implementation has a speedup up to $26\times$ over LAPACK, $3.2\times$ over PLASMA's performance, showing the advantage we gain by using an accelerator for the first stage, and $5.4\times$ over the one stage MAGMA version.

Performance for the complete SVD, including singular vectors, is shown in Fig. 11(b), which combines the two stage reduction, divide and conquer, and back transformation algorithms. We use $9n^3$ operations to compute Gflop/s in all cases for comparison; in actuality, the two stage algorithms perform an additional $4n^3$ operations for the second back transformation. Here, the one stage MAGMA has a speedup over LAPACK up to $3.4\times$. Because PLASMA must do a second back transformation, its speedup is significantly less than in the no-vectors case, but it still achieves up to $2.3\times$, showing that the fast two stage reduction can compensate for the extra cost of the back transformation. The two stage MAGMA implementation is much more efficient at this extra back transformation, attaining a speedup of up to $8.9\times$ over LAPACK, $3.7\times$ over PLASMA's performance, and $2.6\times$ that of the one stage MAGMA.

Fig. 11(c) and (d) zoom in to show performance for small matrices. LAPACK is faster for matrices with $n \leq 800$, which easily fit in the L2 cache and so run at much higher speeds. The release version of MAGMA will have a threshold to call LAPACK for these small sizes.

To see the proportion of time spent in each phase, we show a profile of the SVD time for $n = 20000$ in Fig. 12. For the LAPACK algorithms, the bidiagonal reduction dominates the time. LAPACK with D&C is only 25% faster than with QR iteration, but clearly if QR iteration were used instead of D&C with MAGMA or PLASMA, its time would dominate. The one stage MAGMA algorithm improved both the bidiagonal reduction and the back transformation. PLASMA's two stage reduction is even smaller, but it more than doubles the back transformation time, due to the extra $4n^3$ operations and that the small gemms in PLASMA's back transformations are not as efficient as the large gemms in LAPACK's back transformation. Similarly, the MAGMA two stage back transformation is larger than the MAGMA one stage back transformation. Note that the same second stage routine is faster in the context of the MAGMA two stage version than for PLASMA, because the optimal n_b is smaller for MAGMA (see Section 6). The new two stage MAGMA shows improvements across all three phases, yielding a significant overall improvement. Failure to accelerate any one of the stages would substantially reduce the performance.

9. Conclusions

In accelerating the SVD, we have seen that choosing the right algorithm is important to achieving the best performance. While traditional performance analysis has focused on minimizing floating point operations, more important on today's architectures is minimizing memory accesses, even at the expense of more operations. The two stage bidiagonal reduction accomplishes this by shifting operations from Level 2 to Level 3 BLAS, adding a small number of operations in the second stage, and adding a significant $4n^3$ number of operations in the back transformation. Even with the extra operations, it achieves significant speedups compared to the one stage version. The divide and conquer algorithm both reduces the num-

ber of floating point operations and uses Level 3 BLAS instead of the Level 2 BLAS used in QR iteration, making it 2–3 times faster than QR iteration.

With an appropriate algorithm, rich in Level 3 BLAS, we can then develop a hybrid version that best utilizes the high performance available in today's accelerators. To develop an efficient accelerated version, we concentrate on the strengths of the CPU and accelerator by moving compute intensive Level 3 BLAS operations to the accelerator, leaving on the CPU portions with less parallelism and more complex control flow, such as the second stage band to bidiagonal reduction. Though data dependencies in the SVD often prevent overlapping operations, we overlap CPU computation, accelerator computation, and CPU–accelerator communication where possible. Overall, we achieve 2.6–5.4 times speedup over existing implementations, both the two stage multi-core implementation in PLASMA and the accelerated one stage implementation in MAGMA.

So far, we have focused on computing all singular values and optionally all singular vectors of a square (or nearly square) matrix. For a tall matrix with $m \gg n$, a QR factorization of A is used to reduce the problem to an SVD of the square R matrix, an optimization analyzed by Chan [5]. (A wide rectangular matrix is handled analogously by the transpose operations.) This QR factorization and subsequent additional back-transformation by Q are also accelerated in both the one and two stage MAGMA SVD implementations. Many applications such as least squares need only $\min(m, n)$ columns of U and V, the so-called "economy size" SVD or "some vectors" case in LAPACK. The code differences between the some vectors and all vectors cases are minor modifications to whether a portion or all of Q is generated; the difference in computational complexity is significant: $O(mn^2)$ for some vectors vs. $O(m^2n)$ for all vectors, assuming $m \geq n$. If a subset of the $\min(m, n)$ vectors is needed, the computation could be optimized by including only the desired vectors in the root level of D&C and in the back-transformation, as done for the symmetric eigenvalue problem [19]. If only a few vectors are needed, using inverse iteration such as in the LAPACK routine gesvdx is probably more efficient than using D&C. This would still benefit from our accelerated two stage bidiagonal reduction and back-transformation. Alternatively, a randomized SVD algorithm [30] may be appropriate.

As a concluding thought, we also note the existence of the Jacobi method as an alternative to the bidiagonalization methods. Jacobi iteratively reduces the matrix directly from full to diagonal, without ever reducing to bidiagonal. While the basic Jacobi algorithm is slow, its advantages are being very parallel [1] and attaining higher accuracy than bidiagonalization methods [7]. Blocking and preprocessing can improve the performance to be competitive with other methods [2,11]. Due to its inherent parallelism, Jacobi may be another good candidate for acceleration.

Acknowledgments

We thank the anonymous reviewers for their questions and suggestions that helped us to improve the algorithm and its presentation. This work was supported by the National Science Foundation under grant 1339822, MathWorks, and NVIDIA.

References

[1] M. Bečka, G. Okša, M. Vajteršic, Parallel block-Jacobi SVD methods, in: High-Performance Scientific Computing, Springer, 2012, pp. 185–197. URL https://doi.org/10.1007/978-1-4471-2437-5_9.
[2] M. Bečka, G. Okša, M. Vajteršic, L. Grigori, On iterative QR pre-processing in the parallel block-Jacobi SVD algorithm, Parallel Comput. 36 (5) (2010) 297–307. URL https://doi.org/10.1016/j.parco.2009.12.013.
[3] P. Bientinesi, F.D. Igual, D. Kressner, E.S. Quintana-Ortí, Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2009, pp. 387–395. URL https://doi.org/10.1007/978-3-642-14390-8_40.
[4] C.H. Bischof, B. Lang, X. Sun, Algorithm 807: The SBR Toolbox – software for successive band reduction, ACM Trans. Math. Softw. 26 (4) (2000) 602–616. URL https://doi.org/10.1145/365723.365736.
[5] T.F. Chan, An improved algorithm for computing the singular value decomposition, ACM Trans. Math. Softw. 8 (1) (1982) 72–83. URL https://doi.org/10.1145/355984.355990.
[6] J.J.M. Cuppen, A divide and conquer method for the symmetric tridiagonal eigenproblem, Numer. Math. 36 (2) (1980) 177–195. URL https://doi.org/10.1007/BF01396757.
[7] J. Demmel, K. Veselic, Jacobi'S method is more accurate than QR, SIAM J. Matrix Anal. Appl. 13 (4) (1992) 1204–1245. URL https://doi.org/10.1137/0613074.
[8] J.J. Dongarra, J. Du Croz, S. Hammarling, I.S. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Softw. 16 (1) (1990) 1–17. URL https://doi.org/10.1145/77626.79170.
[9] J.J. Dongarra, J. Du Croz, S. Hammarling, R.J. Hanson, An extended set of FORTRAN basic linear algebra subprograms, ACM Trans. Math. Softw. 14 (1) (1988) 1–17. URL https://doi.org/10.1145/42288.42291.
[10] J.J. Dongarra, D.C. Sorensen, S.J. Hammarling, Block reduction of matrices to condensed forms for eigenvalue computations, J. Comput. Appl. Math. 27 (1) (1989) 215–227. Special Issue on Parallel Algorithms for Numerical Linear Algebra. URL https://doi.org/10.1016/0377-0427(89)90367-1.
[11] Z. Drmac, K. Veselic, New fast and accurate Jacobi SVD algorithm, I, SIAM J. Matrix Anal. Appl. 29 (4) (2008) 1322–1342. URL https://doi.org/10.1137/050639193.
[12] G. Golub, W. Kahan, Calculating the singular values and pseudo-inverse of a matrix, SIAM J. Numer. Anal. 2 (2) (1965) 205–224. URL https://doi.org/10.1137/0702016.
[13] G.H. Golub, C. Reinsch, Singular value decomposition and least squares solutions, Numer. Math. 14 (5) (1970) 403–420. URL https://doi.org/10.1007/BF02163027.
[14] B. Großer, B. Lang, Efficient parallel reduction to bidiagonal form, Parallel Comput. 25 (8) (1999) 969–986. URL https://doi.org/10.1016/S0167-8191(99)00041-1.
[15] M. Gu, S.C. Eisenstat, A divide-and-conquer algorithm for the bidiagonal SVD, SIAM J. Matrix Anal. Appl. 16 (1) (1995) 79–92. URL https://doi.org/10.1137/S0895479892242232.
[16] A. Haidar, J. Kurzak, P. Luszczek, An improved parallel singular value algorithm and its implementation for multicore hardware, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13), ACM, 2013a, p. 90. URL https://doi.org/10.1145/2503210.2503292.

[17] A. Haidar, H. Ltaief, P. Luszczek, J. Dongarra, A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2012, pp. 25–35. URL https://doi.org/10.1109/IPDPS.2012.13.
[18] A. Haidar, R. Solcà, M. Gates, S. Tomov, T. Schulthess, J. Dongarra, Leading edge hybrid multi-GPU algorithms for generalized eigenproblems in electronic structure calculations, in: International Supercomputing Conference, Springer, 2013b, pp. 67–80. URL https://doi.org/10.1007/978-3-642-38750-0.
[19] A. Haidar, S. Tomov, J. Dongarra, R. Solca, T. Schulthess, A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine-grained memory aware tasks, Int. J. High Perform. Comput. Appl. 28 (2) (2014) 196–209. URL https://doi.org/10.1177/1094342013502097.
[20] MAGMA 2.2.0, Innovative Computing Laboratory, 2016. URL http://icl.utk.edu/magma/.
[21] PLASMA 2.8.0, Innovative Computing Laboratory, 2016. URL http://icl.utk.edu/plasma/.
[22] User's Guide for Intel Math Kernel Library for Linux OS, Intel Corporation, 2015. URL http://software.intel.com/en-us/mkl-for-linux-userguide.
[23] L. Karlsson, B. Kågström, Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures, Parallel Comput. 37 (12) (2011) 771–782. URL https://doi.org/10.1016/j.parco.2011.05.001.
[24] S. Lahabar, P. Narayanan, Singular value decomposition on GPU using CUDA, in: IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009., IEEE, 2009, pp. 1–10. URL https://doi.org/10.1109/IPDPS.2009.5161058.
[25] B. Lang, Parallel reduction of banded matrices to bidiagonal form, Parallel Comput. 22 (1) (1996) 1–18. URL https://doi.org/10.1016/0167-8191(95)00064-X.
[26] H. Ltaief, J. Kurzak, J. Dongarra, Parallel two-sided matrix reduction to band bidiagonal form on multicore architectures, IEEE Trans. Parallel Distrib. Syst. 21 (4) (2010) 417–423. URL https://doi.org/10.1109/TPDS.2009.79.
[27] H. Ltaief, P. Luszczek, J. Dongarra, High-performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures, ACM Trans. Math. Softw. 39 (3) (2013) 16:1–16:22. URL https://doi.org/10.1145/2450153.2450154.
[28] J.D. McCalpin, A survey of memory bandwidth and machine balance in current high performance computers, IEEE Comput. Soc. Tech. Committee Comput. Archit. Newslett. (1995) 19–25. URL http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps.
[29] CUDA Toolkit v8.0, NVIDIA Corporation, 2016.
[30] V. Rokhlin, A. Szlam, M. Tygert, A randomized algorithm for principal component analysis, SIAM J. Matrix Anal. Appl. 31 (3) (2009) 1100–1124. URL https://doi.org/10.1137/080736417.
[31] R. Schreiber, C. Van Loan, A storage-efficient WY representation for products of Householder transformations, SIAM J. Sci. Stat. Comput. 10 (1) (1989) 53–57. URL https://doi.org/10.1137/0910005.
[32] F. Tisseur, J. Dongarra, A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures, SIAM J. Sci. Comput. 20 (6) (1999) 2223–2236. URL https://doi.org/10.1137/S1064827598336951.
[33] S. Tomov, R. Nath, J. Dongarra, Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing, Parallel Comput. 36 (12) (2010) 645–654. URL https://doi.org/10.1016/j.parco.2010.06.001.
[34] P.R. Willems, B. Lang, C. Vömel, Computing the bidiagonal SVD using multiple relatively robust representations, SIAM J. Matrix Anal. Appl. 28 (4) (2006) 907–926. URL https://doi.org/10.1137/050628301.