

Look-Ahead in the Two-Sided Reduction to Compact Band Forms for Symmetric Eigenvalue Problems and the SVD

Rafael Rodríguez-Sánchez* Sandra Catalán* José R. Herrero†
 Enrique S. Quintana-Ortí* Andrés E. Tomás*

November 8, 2018

Abstract

We address the reduction to compact band forms, via unitary similarity transformations, for the solution of symmetric eigenvalue problems and the computation of the singular value decomposition (SVD). Concretely, in the first case we revisit the reduction to symmetric band form while, for the second case, we propose a similar alternative, which transforms the original matrix to (unsymmetric) band form, replacing the conventional reduction method that produces a triangular–band output. In both cases, we describe algorithmic variants of the standard Level-3 BLAS-based procedures, enhanced with look-ahead, to overcome the performance bottleneck imposed by the panel factorization. Furthermore, our solutions employ an algorithmic block size that differs from the target bandwidth, illustrating the important performance benefits of this decision. Finally, we show that our alternative compact band form for the SVD is key to introduce an effective look-ahead strategy into the corresponding reduction procedure.

1 Introduction

The reduction to tridiagonal form is a crucial operation for the computation of the eigenvalues of a dense symmetric matrix when a significant part of the spectrum is required [15]. Similarly, the reduction to bidiagonal form is the preferred option to obtain (all) the singular values of a dense matrix via the singular value decomposition (SVD) [15]. The standard algorithms for these two reductions in the legacy implementation of LAPACK (*Linear Algebra PACKage*) [2] compute these reduced forms via two-sided, fine-grained unitary transformations. Unfortunately, these routines are rich in Level-2 BLAS (*Basic Linear Algebra Subroutines*) [13], which are memory-bounded kernels and, therefore, deliver only a small fraction of the peak (computational) performance of recent computer architectures.

An alternative approach replaces these low-performance standard routines with two-sided reduction (TSR) algorithms that consist of two stages [5]. The idea is to initially transform the dense matrix into a *compact band* form (first stage) to next operate on this by-product in order to yield the desired tridiagonal/bidiagonal form (second stage). For symmetric eigenvalue problems (SEVP), the compact by-product is a symmetric band matrix with upper and lower bandwidth w . For the SVD, the compact representation corresponds, by convention, to an upper triangular–band matrix with upper bandwidth w . The appealing property of the TSR algorithms is that

*Depto. Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain. {raro, catalans, quintana, tomasan}@icc.uji.es

†Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Spain. josepr@ac.upc.edu

the initial reduction mainly consists of high performance, compute-bounded Level-3 BLAS [12], which explains the renewed interest in developing two-stage TSR algorithms for multicore processors as well as manycore accelerators [4, 9, 18, 19]. To conclude this brief review of two-stage TSR algorithms, we note that in case w is small compared with the problem dimension, the computational cost of performing the reduction of the band matrix to tridiagonal/bidiagonal form is comparable with that of the initial reduction from the original dense matrix to the compact representation [16, 10]. Furthermore, except for a few special problems, computing the eigenvalues/singular values of the tridiagonal/bidiagonal matrices contributes a minor factor to the global cost of the procedure [11, 21, 14, 17]. In contrast, when the associated eigenvectors/singular vectors are to be computed, the cost of accumulating the unitary transformations during the second stage can be significant, even if the bandwidth is small compared with the problem dimension [4].

High-performance routines for the solution of linear systems (e.g., via the LU and QR factorizations [15]) as well as the initial stage of TSR algorithms usually implement a right-looking (RL) procedure that, at each iteration, factorizes the *current panel* of the matrix, and then applies the transformations that realized this reduction to update the *trailing submatrix*. On today’s multicore platforms, the factorization of the panel is a performance bottleneck because this operation is mostly memory-bounded and, moreover, exhibits a complex set of fine-grain data dependencies. Fortunately, there exist three (to a certain extent complementary) techniques to tackle the constraint imposed by the panel factorization:

- T1) exploit the fine-grain parallelism within the panel itself [7];
- T2) divide the factorization of the current panel into multiple operations whose execution can then be overlapped with certain parts of the trailing update, yielding the so-called algorithms-by-blocks or tile algorithms [16, 6, 22]; and
- T3) overlap the trailing update with the factorization of the “next” panel(s) [23].

Note the distinction between T2), which aims to exploit the parallelism among operations (tasks) in the same iteration of the RL algorithm; and T3), which exploits the parallelism among operations belonging to two (or more) consecutive iterations of the RL algorithm. Here it is worth pointing out that recent developments on the semi-automatic task-parallelization of dense linear algebra operations with the support of a “runtime” (such as SuperMatrix, Quark, OmpSs, StarPU, OpenMP, etc.) have partially blurred the frontier between T2) and T3). In particular, when this type of task-parallelization is applied to an algorithm-by-blocks for the solution of linear systems in order to realize T2), the result is that T3) is often obtained for free. For some TSR algorithms though, as we will discuss in the paper, this may be more difficult or even impossible.

In this paper we focus on T3), which is usually known as *look-ahead* [23]. Here, as we do not rely on a runtime to exploit “inter-iteration” parallelism, we can refer more precisely to this strategy as “static” look-ahead. While this technique has been long known and exploited for the solution of linear systems via the LU and QR factorizations¹, its application to TSR algorithms has not been fully discussed explicitly. In this paper we show that look-ahead can be introduced in the sophisticated TSR algorithms for SEVP and the SVD, delivering remarkable performance benefits. In particular, our paper makes the following contributions:

- We explore the integration of look-ahead into the reduction of symmetric matrices to band form for SEVP via two-sided unitary transformations. In this line, we propose two variants

¹Static look-ahead is for example the technique embedded in the implementation of these factorizations in Intel MKL.

of the reduction algorithm, enhanced with look-ahead, with distinct performance behaviour depending on the ratio between the algorithmic block size b (which dictates the number of columns in the panel,) and the target matrix bandwidth w . While LAPACK (version 3.7.1) and MAGMA (version 2.1.0) both include routines for SEVP to reduce the symmetric input matrix to band form, those implementations impose the restriction that the algorithmic block size must equal the bandwidth, limiting performance. Furthermore, the LAPACK routine for this reduction does not integrate look-ahead. The SBR (Successive Band Reduction) package [5] was a pioneer work that decoupled the bandwidth from the algorithmic block size in this type of reduction, but did not integrate look-ahead either.

- We extend our analysis of look-ahead to the reduction of general matrices to band form for the SVD via two-sided unitary transformations. Here we depart from the conventional TSR to band-triangular form, which imposes certain restrictions on the application of look-ahead, to advocate for the reduction to band form with equal lower and upper bandwidths. This change, in turn, yields two variants for the reduction algorithm for the SVD which are analogous to those identified for SEVP.
- We demonstrate the performance benefits of static-look ahead, using the reduction to band forms for SEVP and the SVD, on an Intel-based platform equipped with 8 Haswell cores. Our experimental analysis of the optimal block size clearly shows the importance of decoupling the algorithmic block size from the bandwidth, and the advantages of each variant.

The introduction of look-ahead paves the road to overlapping the panel factorization on a CPU with the execution of the (rich in Level-3 BLAS) trailing update on an accelerator (e.g., a GPU). Furthermore, on a multicore architecture, an algorithm that explicitly decomposes the TSR to expose look-ahead can apply this technique with variable depth, using the support of a runtime such as OpenMP, OmpSs or StarPU. In both cases, we can expect a notable increase of performance, as *i*) the panel factorization is potentially removed from the critical path of the algorithm; and *ii*) the algorithmic block size is decoupled from the bandwidth.

The rest of the paper is structured as follows. In Sections 2 and 3 we describe the introduction of look-ahead in the first stage of the TSR algorithms for SEVP and the SVD, respectively. In Section 4 we assess the benefits of a flexible implementation of this technique by experimentally demonstrating its effects for the TSR of dense matrices to the selected band forms for SEVP and the SVD. Finally, in Section 5 we close our paper with a few concluding remarks and a discussion of future work.

To close this introduction, we note that the mathematical equations, algorithms, and the evaluation in the remainder of the paper are all formulated for problems with real data entries, using orthogonal transformations, but their extension to the Hermitian case, involving unitary transformations, is straight-forward.

2 TSR for SEVP

2.1 Basic algorithm

Let us first describe the algorithm that reduces a dense symmetric matrix $A \in \mathbb{R}^{n \times n}$ to symmetric band form, with bandwidth w , via orthogonal similarity transformations. This procedure is numerically stable and, moreover, preserves the eigenvalues of the matrix [15]. Suppose that the first $k - 1$ rows/columns of A have been already reduced to band form; the algorithmic block size

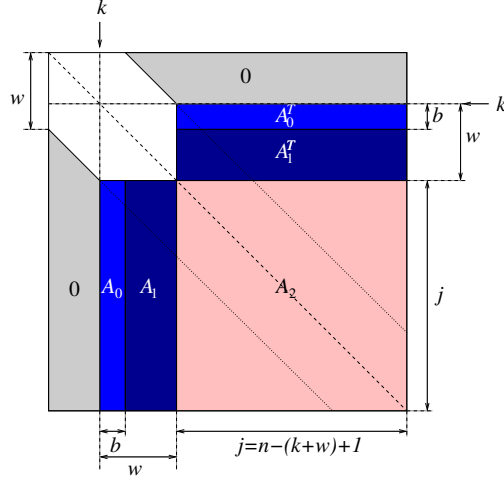


Figure 1: Partitioning of the matrix during one iteration of the reduction to symmetric band form for the solution of SEVP.

satisfies $b \leq w$; and assume for simplicity that $k + w + b - 1 \leq n$; see Figure 1. At this point we note the key roles of the bandwidth w and the block size b , and their interaction. The optimal bandwidth itself depends on the efficiency of the second stage of the reduction and, therefore, it cannot be chosen independently. To complicate things a bit further, the optimal bandwidth also depends on the problem dimensions and the selected block size. The take-away lesson of this short discussion is that the best combination of bandwidth and block size depends on several factors, some of which are external to the implementation of the first stage.

During the current iteration of the reduction procedure, b new rows/columns of the band matrix are computed as follows:

1. PANEL FACTORIZATION. Compute the QR factorization

$$A_0 = QR, \quad (1)$$

where $A_0 \in \mathbb{R}^{j \times b}$, with $j = n - (k + w) + 1$; $R \in \mathbb{R}^{j \times b}$ is upper triangular; and the orthogonal matrix Q is implicitly assembled using the WY representation [15] as $Q = I_j + WY^T$, where $W, Y \in \mathbb{R}^{j \times b}$ and I_j denotes the square identity matrix of order j .

2. TRAILING UPDATE. Apply the orthogonal matrix Q to $A_1 \in \mathbb{R}^{j \times w-b}$ from the left:

$$A_1 := Q^T A_1 = (I_j + WY^T)^T A_1 = A_1 + Y(W^T A_1); \quad (2)$$

and to $A_2 \in \mathbb{R}^{j \times j}$ from both left and right:

$$\begin{aligned} A_2 &:= Q^T A_2 Q = (I_j + WY^T)^T A_2 (I + WY^T) \\ &= A_2 + YW^T A_2 + A_2 WY^T + YW^T A_2 WY^T. \end{aligned} \quad (3)$$

During this last operation (only the lower or upper triangular part of) A_2 is updated, via the

following sequence of Level-3 BLAS operations:

$$X_1 := A_2 W, \tag{4}$$

$$X_2 := \frac{1}{2} X_1^T W, \tag{5}$$

$$X_3 := X_1 + Y X_2, \tag{6}$$

$$A_2 := A_2 + X_3 Y^T + Y X_3^T. \tag{7}$$

Provided b and w are both small compared with n , the global cost of the reduction of a full matrix to band form is $4n^3/3$ floating-point arithmetic operations (flops).² Furthermore, the bulk of the computation is performed in terms of the Level-3 BLAS operations in (4) and (7).

The problem with this basic algorithm is that the panel factorization in (2) is mainly memory-bounded (at least, for the usual values of b) as well as features some complex dependencies so that, as the number of cores performing the factorization is increased, the panel operation rapidly becomes a performance bottleneck. We next describe how to solve this problem via two algorithmic variants that implement a static look-ahead in order to overlap in time (i.e., run concurrently) the execution of the trailing update for the current iteration with the factorization of the next panel.

2.2 Introducing look-ahead

Consider the blocks A_0, A_1, A_2 involved in iteration k of the basic algorithm (see Figure 1); and let us refer to the panel that will be factorized in the subsequent iteration $\bar{k} = k + b - 1$ as \bar{A}_0 . The key to formulate a variant of the basic algorithm enhanced with look-ahead lies in:

1. identifying the parts of the trailing submatrix $[A_1, A_2]$ that will become \bar{A}_0 during the next iteration;
2. isolating the updates corresponding to application of the orthogonal transformations in (3) that affect \bar{A}_0 from those which modify those parts of $[A_1, A_2]$ that do not overlap with \bar{A}_0 ; and
3. during iteration k , overlapping the factorization of the subsequent panel \bar{A}_0 (look-ahead factorization) with the updates corresponding to this iteration.

At this point we distinguish two cases, leading to two variants of the TSR algorithm with look-ahead, depending on the relation between b and w :

- Variant V1: $2b \leq w$. In this case, \bar{A}_0 lies entirely within A_1 , as the number of columns in the latter satisfies $w - b \geq b$. We then define the following partitioning of the trailing submatrix:

$$\left[\begin{array}{c|c} A_1 & A_2 \end{array} \right] = \left[\begin{array}{c|c|c} A_1^L & A_1^R & A_2 \end{array} \right] = \left[\begin{array}{c|c} \frac{A_1^{TL}}{\bar{A}_0} & A_1^R \end{array} \middle| A_2 \right], \tag{8}$$

where A_1^L consists of b columns, A_1^{TL} is $b \times b$, and we (use the red color to) distinguish those blocks that overlap in the column range of \bar{A}_0 .

²Hereafter, lower order terms are neglected in the theoretical costs.

During iteration k , we can then perform the following three groups of operations (left, middle and right) concurrently:

Sequential panel factorization

$$\begin{aligned} A_1^L &:= Q^T A_1^L \\ \bar{A}_0 &= \bar{Q}\bar{R} \end{aligned}$$

$$A_1^R := Q^T A_1^R$$

Parallel remainder update

$$A_2 := Q^T A_2 Q$$

Note that, with this partitioning, \bar{A}_0 is generally small compared with A_2 . Therefore, we can expect that the factorization of the subsequent panel \bar{A}_0 can be overlapped with the update of A_2 on the right, eliminating the former from the critical path of the reduction. On a multicore architecture, we can achieve this by dedicating a few threads/cores to the panel factorization while the remaining ones compute the trailing update. On a CPU-GPU system, the CPU can take care of the panel factorization while the GPU updates the trailing submatrix. Hereafter, we will refer to these (two groups of) computational resources, few threads/CPU and many threads/GPU, as T_S (for sequential) and T_P (for parallel) respectively.

There exists a direct dependency between the two operations on the left-hand side group, that we can denote as $A_1^L \rightarrow \bar{A}_0$. Here, the update of A_1^L is a Level-3 BLAS operation that in general will offer low performance as the width of the panel is usually small. Due to the dependency and this low performance, the update of A_1^L will be performed by T_S . As for the update of A_1^R , in the middle “group”, in case this is also a narrow column panel (i.e., $w - 2b$ is small), we can expect low performance from it, so that it should join the group of “sequential” operations on the left (red group), to be performed by T_S . Otherwise, it can be merged with the “parallel” group on the right, to be computed by T_P .

- Variant V2: $2b > w$. In this case, \bar{A}_0 expands beyond the columns of A_1 to partially overlap with A_2 . Let us consider the following partitioning:

$$\left[A_1 \mid A_2 \right] = \left[A_1 \mid A_2^L \mid A_2^R \right] = \left[\begin{array}{c|c} A_1^T & A_2^{TL} \\ \hline & \bar{A}_0 \end{array} \mid A_2^R \right], \quad (9)$$

where $[A_1, A_2^L]$ consists of b columns and $[A_1^T, A_2^{TL}]$ is $b \times b$.

During iteration k , we initially compute the following operations:

$$A_1 := Q^T A_1, \quad (10)$$

$$X_1 := A_2 W, \quad (11)$$

$$X_2 := \frac{1}{2} X_1^T W, \quad (12)$$

$$X_3 := X_1 + Y X_2, \quad (13)$$

which correspond to the update of A_1 and part of the computations necessary for the update of A_2 ; see (4)–(7). After this is completed, we can concurrently perform the following two groups of operations:

Sequential panel factorization

$$\begin{aligned} A_2^L &:= A_2^L + X_3(Y^L)^T + Y(X_3^L)^T \\ \bar{A}_0 &= \bar{Q}\bar{R} \end{aligned}$$

Parallel remainder update

$$A_2^R := A_2^R + X_3(Y^R)^T + Y(X_3^R)^T$$

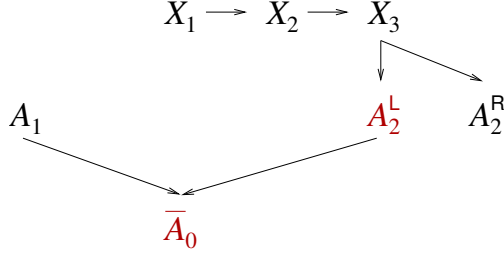


Figure 2: Dependencies among operations appearing in Variant V2 of the initial TSR to symmetric band form. For simplicity, each operation is identified by its output operand.

Here, $Y = [Y^L, Y^R]$ and $X_3 = [X_3^L, X_3^R]$ are partitionings conformal with $A_2 = [A_2^L, A_2^R]$. As in the previous variant (case), this pursues the goal of overlapping the factorization of the next panel \bar{A}_0 with a sufficiently-large Level-3 BLAS. In general, \bar{A}_0 is small compared with the trailing submatrix A_2^R so that we can expect this is the case.

To close the discussion of Variant V2, we note the collection of dependencies appearing among the operations identified in this case; see Figure 2. For these operations, the panel width determines whether they involve narrow panels and, therefore, can be considered memory-bounded low-performance kernels. Thus, together with the dependencies, this property will ultimately decide whether they are moved to the groups of either sequential or parallel kernels, to be tackled by T_S or T_P , respectively. For example, one possibility is to update A_1 on T_S , while X_1, X_2, X_3 are being computed by T_P ; when all these operations are completed, we can continue with the update of A_2^L and the factorization of \bar{A}_0 on T_S , while A_2^R is being updated by T_P .

At this point, it is fair to ask what is the value of explicitly exposing static look-ahead if the same effect could be obtained, in principle, with the combination of an algorithm-by-blocks and the support of a task-parallelizing runtime. Armed with the previous discussion of look-ahead, we can now offer several arguments in response to this question:

1. Exposing the look-ahead variant provides a better understanding of the algorithms.
2. Static look-ahead can be as efficient as or even outperform a runtime-assisted dynamic solution [8]. The reason is that, for regular dense linear algebra operations such as those in the Level-3 BLAS, dividing these kernels into fine-grain operations incurs into some packing/unpacking overheads. In addition, the use of a runtime promotes the exploitation of task-parallelism at the cost of a suboptimal use of the cache hierarchy.
3. As exposed in the next section, for the tile algorithm proposed for the reduction to triangular-band form, the application of a runtime may not allow *per se* the exploitation of task-parallelism among operations belonging to different iterations.
4. The look-ahead variants do not require the implementation of tuned kernels to factorize blocks with special structures as those that appear in the algorithm-by-blocks, and apply the corresponding transformations.³ Moreover, they do not incur the overhead due to the operation with these kernels and do not have an internal block size that needs to be tuned [6, 22].

³We recognize that this problem can be overcome by a careful reconstruction of the orthogonal factor, but this comes at the cost of an increase in the computational cost of the panel factorization [3].

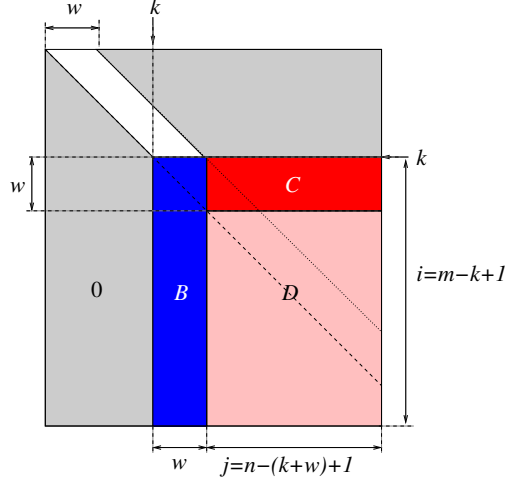


Figure 3: Partitioning of the matrix during one iteration of the reduction to triangular-band form for the SVD ($w = b$).

5. For CPU-GPU platforms, static look-ahead can be the only practical means to eliminate the panel factorization from the critical path of the algorithm.

3 TSR for the SVD

3.1 Triangular-band form

The conventional algorithm for the first stage of the TSR algorithm for the SVD computes an upper triangular matrix with upper bandwidth w . To describe this procedure, consider a matrix $A \in \mathbb{R}^{m \times n}$, where the first k rows/columns have already been reduced to the desired triangular-band form; and assume that $k + w + b - 1 \leq m, n$, with $m \geq n$. Furthermore, *let us consider initially the simpler case with $w = b$* ; see Figure 3. During the current iteration, the following computations thus advance the reduction by w additional rows/columns:

1. LEFT PANEL FACTORIZATION. Compute the QR factorization

$$B = UR, \quad (14)$$

where $B \in \mathbb{R}^{i \times w}$, with $i = m - k + 1$; $R \in \mathbb{R}^{i \times w}$ is upper triangular; and $U = I_i + W_U Y_U^T$ is orthogonal, with $W_U, Y_U \in \mathbb{R}^{i \times w}$.

2. LEFT TRAILING UPDATE. Apply U to $E = \begin{bmatrix} C \\ D \end{bmatrix} \in \mathbb{R}^{i \times j}$, with $j = n - (k + w) + 1$, from the left:

$$E := U^T E = (I_i + W_U Y_U^T)^T E = E + Y_U (W_U^T E). \quad (15)$$

3. RIGHT PANEL FACTORIZATION. Compute the LQ factorization [15]

$$C = LV^T, \quad (16)$$

where $C \in \mathbb{R}^{w \times j}$; $L \in \mathbb{R}^{w \times j}$ is lower triangular; and $V = I_j + W_V Y_V^T$ is orthogonal, with $W_V, Y_V \in \mathbb{R}^{j \times w}$.

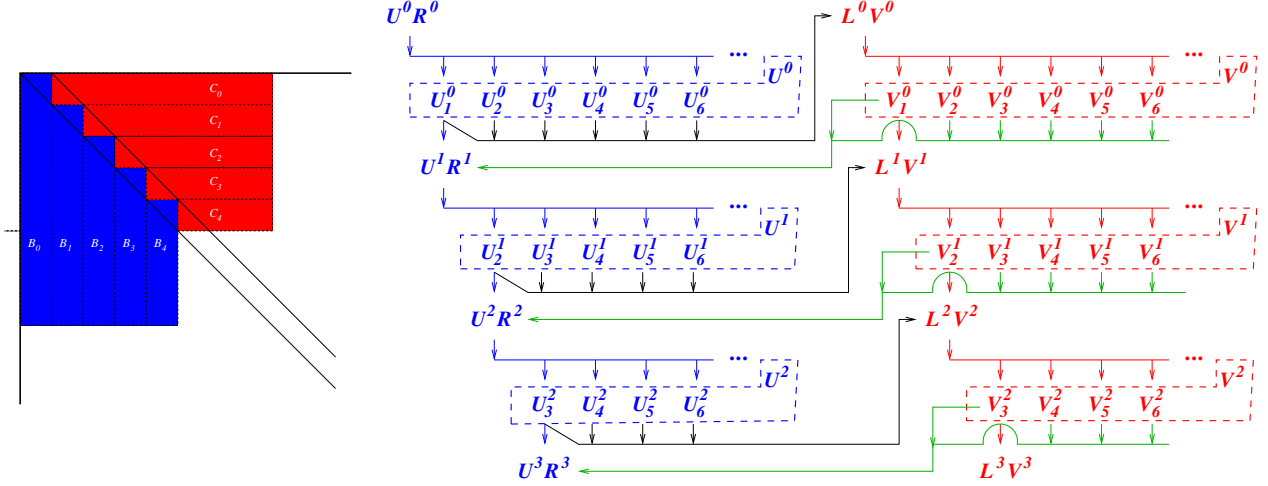


Figure 4: Matrix partitioning (left) and dependencies (right) for the reduction to triangular-band form for the SVD ($w = b$).

4. RIGHT TRAILING UPDATE. Apply V to $D \in \mathbb{R}^{(i-w) \times j}$, from the right:

$$D := DV = D(I_j + W_V Y_V^T) = D + (DW_V) Y_V^T. \quad (17)$$

Assuming $w \ll m, n$, this algorithm requires $4(mn^2 - n^3/3)$ flops and the major part of these operations are concentrated in the trailing updates (15), (17), which correspond to high performance Level-3 BLAS.

3.2 Triangular-band form and look-ahead

Unfortunately, the two panel factorizations in (14), (16) impose the same bottleneck as that discussed for the reduction to the symmetric band form. Furthermore, in the reduction to triangular-band form overcoming this problem via a look-ahead strategy enforces certain constraints on the relation between w and b that may impair performance. Let us explain this in detail via three cases, where the first one corresponds to the simple scenario with $w = b$, and the remaining two decouple the block size from the bandwidth so that $b \leq w$.

First case: $w = b$. Consider the scenario illustrated in Figure 4 where the superindices (as, e.g., in the factorization $B^0 = U^0 R^0$) indicate the iteration count (starting at 0), and the subindices specify the index of the block being updated by the corresponding transformations (either from the left, as in U_1^0 , or from the right, as in V_1^0). The arrows correspond to data dependencies and thus define a partial ordering for the execution of the operations. For simplicity, let us aggregate the updates $U_{k+1}^k, U_{k+2}^k, U_{k+3}^k, \dots$ into a single macro-update U^k and, similarly, $V_{k+1}^k, V_{k+2}^k, V_{k+3}^k, \dots$ into the macro-update V^k . Then, it is easy to verify that the existing dependencies enforce the strict ordering $U^0 R^0 \rightarrow U^0 \rightarrow L^0 V^0 \rightarrow V^0 \rightarrow U^1 R^1 \rightarrow \dots$, revealing that it is not possible to exploit look-ahead in this case.

Second case: $w = 2b$. The new situation is displayed in Figure 5, which will be leveraged to expose that the dependency problem identified in the previous case partially remains. In particular, let us now aggregate the updates $U_{k+2}^k, U_{k+3}^k, U_{k+4}^k, \dots$ into a single macro-update \bar{U}^k and

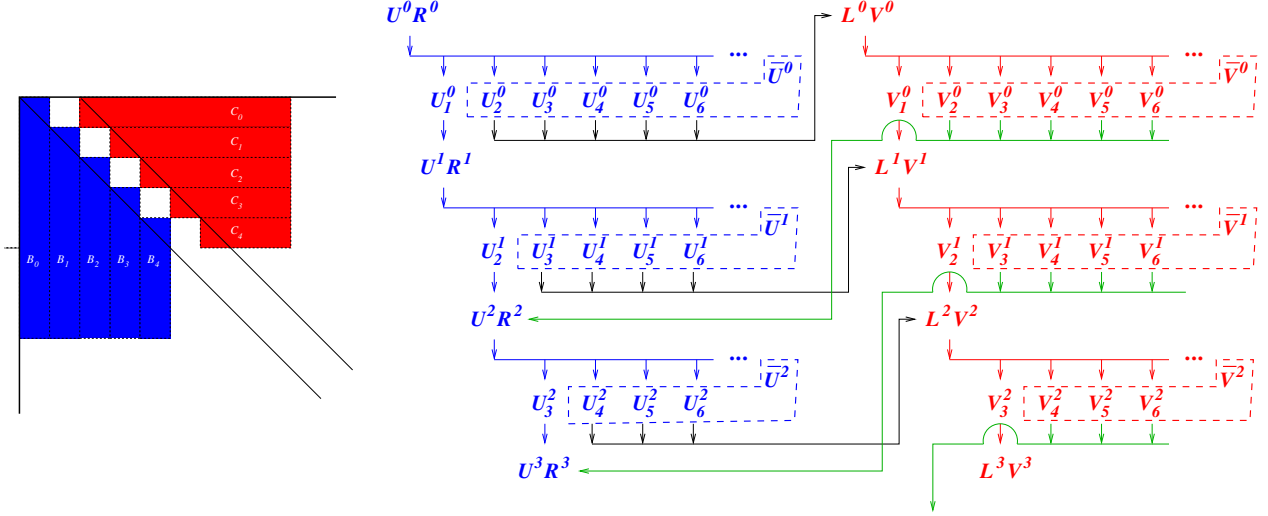


Figure 5: Matrix partitioning (left) and dependencies (right) for the reduction to triangular-band form for the SVD ($w = 2b$).

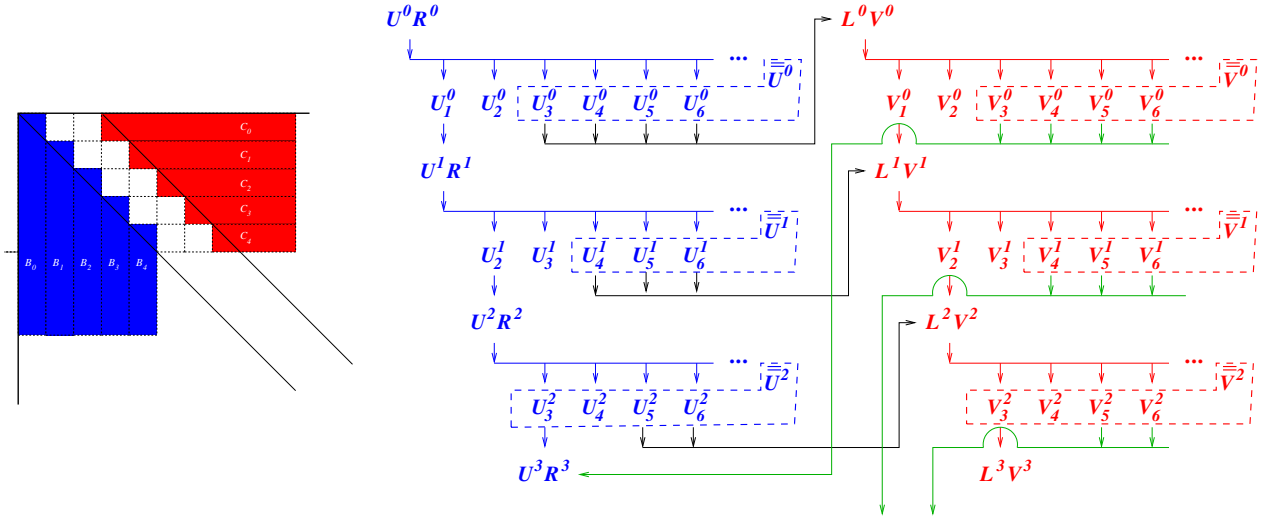


Figure 6: Matrix partitioning (left) and dependencies (right) for the reduction to triangular-band form for the SVD ($w = 3b$).

$V_{k+2}^k, V_{k+3}^k, V_{k+4}^k, \dots$ into \bar{V}^k . Moreover, for simplicity let us consider that the updates of the form U_{k+1}^k and V_{k+1}^k respectively occur inside the factorizations $U^{k+1}R^{k+1}$ and $L^{k+1}V^{k+1}$. Then, we can initially compute U^0R^0 ; followed by the overlapped execution of the factorization U^1R^1 with the macro-update \bar{U}^0 ; and the factorization L^1V^1 next. At this point, we would like to overlap U^2R^2 with \bar{U}^1 and L^1V^1 with \bar{V}^0 . However, because of the dependencies, we can exploit one of the overlappings, but not both. To see this, assume our goal is to encode the first overlapping. Then, \bar{V}^0 must be available (green lines), but L^1V^1 cannot be computed yet (black lines). Therefore, the second overlapping is not possible. Conversely, assume that we intend to encode the second overlapping. Then, \bar{U}^1 must be available (black lines), but L^1V^1 cannot be computed yet (green lines), and the first overlapping is not possible.

Third case: $w = 3b$. Consider next the scenario in Figure 6. Let us use the macro-update \bar{U}^k to stand now for $U_{k+3}^k, U_{k+4}^k, U_{k+5}^k, \dots$; and \bar{V}^k for $V_{k+3}^k, V_{k+4}^k, V_{k+5}^k, \dots$. Also, assume for simplicity that the updates of the form U_{k+1}^k, U_{k+2}^k and V_{k+1}^k, V_{k+2}^k respectively occur as part of the factorizations $U^{k+1}R^{k+1}$ and $L^{k+1}V^{k+1}$. As in the previous case, we can initially compute U^0R^0 ; followed by the overlapped execution of the factorization U^1R^1 with the macro-update \bar{U}^0 ; and the factorization L^1V^1 next. However, because of the distinct dependencies that are present in this third case, nothing prevents us in the following steps from overlapping U^2R^2 with \bar{U}^1 ; L^1V^1 with \bar{V}^0 ; U^3R^3 with \bar{U}^2 ; and so on.

The conclusion from this study is that, in the reduction to triangular–band form, applying look-ahead for both the left and right panel factorizations requires $w \geq 3b$. While this is doable, it has some practical implications on the relation between the practical values of w and b . On one hand, w should be kept small to moderate because the selection of a large value delays much of the computational cost into the second stage (reduction from triangular–band to bidiagonal form), which is realized via slower Level-2 BLAS. On the other hand, b needs to be set to a large value as otherwise the updates will not fully benefit from the performance of Level-3 BLAS. The practical consequence is that the constraint that $w \geq 3b$ in this type of decomposition can exert a strong negative impact on performance.

Pipelining the factorizations. Consider again the simple case $w = b$ and the operations (14)–(17) to be computed in a given iteration. A different (but related) possibility to attain an overlapped execution is to decompose the left panel factorization in this iteration into several column micro-panels, of width $b_l < w$, and then overlap the factorization of the micro-panels with their application to the remaining part(s) of the matrix (i.e., within the same micro-panel within B as well as to E). When this is completed, the algorithm proceeds to the right panel factorization in the same iteration, and basically applies the same idea using row micro-panels of height b_r . With this strategy we can choose a value for the micro-panels that simply satisfies $w = 2b_l = 2b_r$. However, note that with this approach, the first micro-panel for both the left and right panel factorizations (at each iteration) cannot be overlapped, with a strong negative impact on performance. This will enforce us to select $w \geq 3b_l, 3b_r$, with the same consequences as those discussed in the previous paragraph. Even worse, with this approach, the first left and right panel factorizations of each iteration cannot be overlapped.

Other implementations. The discussion of the reduction to triangular–band reveals a strong limitation when aiming to exploit task-parallelism among operations belonging to different iterations. We note that this algorithm is precisely the selection that was made for the message-passing implementation of TSR to triangular–band form in [16]. It is also the choice for the tile algorithms in PLASMA that perform this reduction on multicore platforms in [18, 19]. In addition, all of these implementations couple the algorithmic block size to the bandwidth, so that $b = w$, with a potential negative effect on performance.

3.3 Band form and look-ahead

In [16] the authors explored the triangular–band reduction as well as an alternative algorithm that reduces the dense matrix to a band form with the same upper and lower bandwidth. However, the latter algorithm was abandoned in that work as it did not offer any special advantage. In

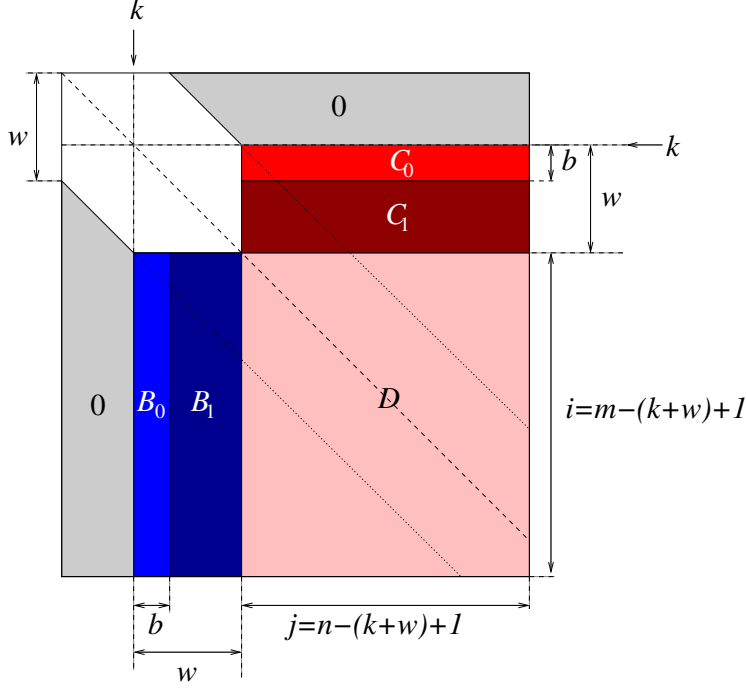


Figure 7: Partitioning of the matrix during one iteration of the reduction to band form for the SVD.

this subsection we show that this approach is actually the key to obtaining two variants for the reduction to band form for the SVD, enhanced with look-ahead, which are analogous to those already presented for SEVP in subsection 2.2. Importantly, this approach does not enforce that $w \geq 3b$, as was the case of the reduction to triangular–band form. Before we review this algorithm, we note that the cost of applying this procedure to reduce a dense matrix to band form, with upper and lower bandwidth $w = w'/2$, is about the same as that of reducing the matrix to triangular–band form with bandwidth w' .

The basic algorithm (i.e., without look-ahead) is very similar to the reduction to symmetric band form, with the differences stemming from the fact that A is now an unsymmetric matrix, which requires separate left and right factorizations. As usual, consider that the first $k - 1$ rows/columns of A have been already reduced to band form; select $b \leq w$; and assume for simplicity that $k + w + b - 1 \leq m, n$; see Figure 7.

During the current iteration of the reduction procedure, b new rows/columns of the band matrix are computed as follows (for brevity, we do not state explicitly the dimensions and properties of the matrix blocks/factors in the following, as they can be easily derived from the context and Figure 7):

1. LEFT PANEL FACTORIZATION. Compute the QR factorization

$$B_0 = UR, \quad \text{with } U = I_i + W_U Y_U^T. \quad (18)$$

2. LEFT TRAILING UPDATE. Apply U to the trailing submatrix:

$$B_1 := U^T B_1 = B_1 + Y_U (W_U^T B_1); \quad (19)$$

$$D := U^T D = D + Y_U (W_U^T D); \quad (20)$$

3. RIGHT PANEL FACTORIZATION. Compute the LQ factorization

$$C_0 = LV^T, \quad \text{with } V = I_j + W_V Y_V^T. \quad (21)$$

4. RIGHT TRAILING UPDATE. Apply V to the trailing submatrix:

$$C_1 := C_1 V = A_1 + (A_1 W_V) Y_V^T; \quad (22)$$

$$D := D V = D + (D W_V) Y_V^T. \quad (23)$$

From these expressions, let us now re-consider the two cases leading to Variants V1 and V2 of the look-ahead strategy:

- Variant V1: $2b \leq w$. The next panels \bar{B}_0 and \bar{C}_0 lie entirely within B_1 and C_1 , respectively. Therefore, the update and factorization of these panels can be overlapped with the updates performed on D from the left and right, respectively.
- Variant V2: $2b > w$. Now both \bar{B}_0 and \bar{C}_0 extend to overlap with D . The key to introduce look-ahead is that the left and right updates of D can be performed “simultaneously” as follows [16]:

$$Z_L := D^T W_U, \quad (24)$$

$$Z_R := D W_V, \quad (25)$$

$$X := Z_R + Y_U (Z_L^T W_V), \quad (26)$$

$$\begin{aligned} D &:= U^T D V \\ &= D + Y_U W_U^T D + D W_V Y_V^T + Y_U W_U^T D W_V Y_V^T \\ &= D + [X, Y_U][Y_V, Z_L]^T. \end{aligned} \quad (27)$$

Therefore, we can initially perform the updates of B_1 , C_1 and compute Z_L , Z_R , X . Next, we partition the update of D to expose those parts of the result that overlap with \bar{B}_0 and/or \bar{C}_0 :

$$D = \left[\begin{array}{c|c} D_{11} & D_{12} \\ \hline D_{21} & D_{22} \end{array} \right], \quad (28)$$

where $D_{11} \in \mathbb{R}^{(2b-w) \times (2b-w)}$. Finally, by partitioning the operands $[X, Y_U]$, $[Y_V, Z_L]^T$ in (27) conformally with D in (28), we can overlap the updates of D_{11} , D_{21} , D_{12} , and the small left and right panel factorizations in T_S with the update of the larger D_{22} in T_P .

4 Experimental Evaluation

In this section, we analyze in detail the performance benefits obtained by introducing the look-ahead strategies formulated in this paper as well as the decoupling of the algorithmic block size from the bandwidth in the TSR algorithms for SEVP and the SVD. All experiments were performed using IEEE double-precision arithmetic on an Intel Xeon E5-2630 v3 processor (8 cores running at a nominal frequency of 2.4 GHz). The implementations were linked with BLIS (version 0.1.8) [24].

In the experiments, we employed square symmetric matrices for SEVP, and both square and rectangular matrices for the SVD, with random entries uniformly distributed in $(0, 1)$, and dimensions of up to 10000 in steps of 500. We reiterate that the optimal bandwidth w depends not only on

the implementation of the first stage, but also on that of the second stage, for which there exist multiple algorithms and tuned implementations, depending on the target architecture [9, 18, 19, 10], the problem size, etc. For this reason, we decided to test the algorithms using six bandwidths: $w = \{32, 64, 96, 128, 192, 256\}$. For these cases, the block size b was then tuned using values ranging from 16 up to $w/2$ for Variant V1 and up to w for V2, in steps of 16. We employed one thread per core in all executions. For the look-ahead versions, we set T_S with 1 core and T_P with the remaining 7 cores; for the reference implementations without look-ahead, there is no separation of the threads into groups so that all of them participate in the execution of each BLAS.

In all cases, we use the nominal flop count to compute the GFLOPS (billions of flops/sec.). For example, for the reduction in the SEVP, we employ $4n^3/3$ flops independently of the target bandwidth w . Since the comparison between algorithms/variants is performed in an scenario with fixed w , this is a reasonable approach to obtain a scaled version of the execution time, with the differences being more visible for smaller problem sizes than those that could have been exposed using the execution time itself.

4.1 Reference implementation

Our implementation of (the first stage of) the TSR algorithms for SEVP is based on the codes in the SBR package [5]. On the original version of these codes, we performed two relevant optimizations:

- We replaced the routine for the panel factorization in the SBR package, based on Level-2 BLAS, for an alternative that factorizes this panel using a blocked left-looking (LL) algorithmic variant that, furthermore, relies on Level-3 BLAS. The inner block size for this routine was set to 16, with this value being determined (i.e., tuned) in an independent experiment. The LL variant was selected because it offered higher performance than its RL counterpart in our experiments.
- The routine that builds the matrices W, Y in SBR, which define the orthogonal factors, was modified to assemble W as the product of Y and the triangular $b \times b$ matrix T that is obtained from the alternative *compact WY representation* [15]. This modification considerably reduced the cost of building W as this can then be based entirely on Level-3 BLAS.

The implementations of the TSR algorithms for the SVD re-utilized as much as possible of these building blocks, including the ideas underlying the previous two code optimizations. The legacy LAPACK (version 3.7.1) comprises routine `dsytrd_sy2sb` for the reduction of a symmetric matrix to symmetric band form which also features these optimizations (except for the use of the left-looking factorization). However, the LAPACK routine does not include look-ahead and, furthermore, it imposes the restriction that $b = w$. As our experimental evaluation of the optimal block size will show, this limitation severely impairs performance.

4.2 The role of the block size

In practice, the algorithmic block size b has an important impact on performance. For the particular case (of the first stage) of both TSR algorithms, the block size should balance two criteria:

1. Deliver high performance for the BLAS-3 kernels that compose the trailing update. Small values of b turn W, Y, W_U, Y_U, W_V, Y_V into narrow column panels, affecting the performance of the Level-3 BLAS, since the amount of data reuse is greatly reduced so that, eventually, the kernels become memory-bounded.

2. Reduce the amount of flops performed in the panel factorization. Small values of b reduce the number of operations performed in these intrinsically-sequential operations.

Figure 8 illustrates the interplay between the block size and the bandwidth, using the reduction of symmetric matrices to a banded form (first stage of SEVP). The figure includes the reference implementation (hereafter, labeled as “Reference SEVP”) as well as the two look-ahead variants introduced in this work.

This experiment reveals that, for the small problem (two top plots in Figure 8), the optimal performance is attained for small values of the block size, since they balance the execution times of the panel factorization and the trailing update. In rough detail, for the small problem dimensions, if the block size is too large, the panel factorization will require more time to complete than the trailing update. As a consequence, many computing resources (i.e., threads/cores) will become idle during the iteration, degrading performance. At this point, we note that the degree of resource concurrency also exerts its impact on the workload balance, as this factor can change the problem size threshold from which the trailing update is more expensive than the panel factorization.

In addition, the plots show that, regardless of the implementation, the lowest performance for the large problem (two bottom plots in Figure 8) is observed for the smallest and the largest block sizes, with the optimal choice residing in the middle range. For the smallest block size, the BLAS-3 kernels invoked from the trailing update cannot efficiently utilize all the computational potential of the platform. For largest block sizes too many flops are devoted to the panel factorization.

The previous experimental analysis conforms that in [8] for the LU factorization, and exposes that choosing the optimal block size is a non-trivial task since this parameter depends on many other factors such the problem dimension n , the bandwidth w , and the degree of parallelism. To further complicate the selection of the optimal block size, as the reduction progresses, the operation is decomposed into sub-problems of dimension $n - b, n - 2b, \dots$. This implies that the optimal block size for the initial sub-problem(s) may not be the optimal for the subsequent ones. To partially compensate for this, the computational cost of the sub-problems rapidly decreases with their dimensions.

These experiments clearly show that coupling the block size to the bandwidth, so that $b = w$, in general results in suboptimal performance. Taking into consideration the elaboration and results in this subsection, in the remainder of this paper we will perform an extensive tuning of the block size, for each problem dimension and bandwidth.

4.3 Performance of TSR to symmetric band form for SEVP

In this section, we analyze in detail the performance behavior of the multi-threaded variants with look-ahead aimed to enhance the computational throughput of the algorithms for the first stage of SEVP. Specifically, the following implementations are compared:

- **Reference SEVP:** Reference implementation from SBR with the optimizations described in subsection 4.1.
- **Variante V1:** Look-ahead variant for problems with $2b \leq w$. Two different mappings of the update of A_1^R to the threads/cores were considered, depending on whether this operation is performed by either T_S or T_P . In both cases, the factorization of \bar{A}_0 and the update of A_1^L are performed by T_S ; and the update of A_2 is done by T_P . Our experiments with these mappings demonstrated that the first option, which updates the full A_1 using T_S , always delivers equal or lower performance than the alternative mapping for this variant. (The reason is that, for

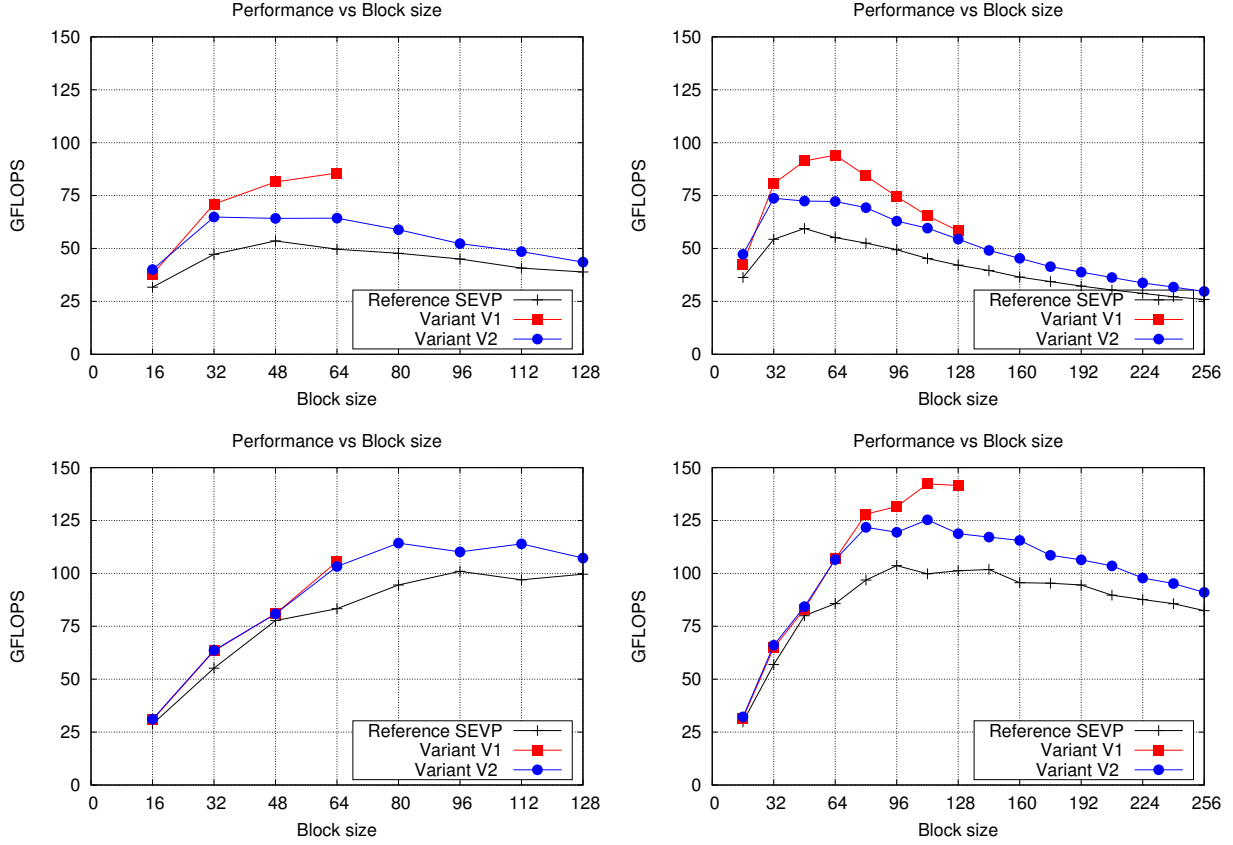


Figure 8: Performance vs block size of the SEVP implementations; The bandwidth is set to $w = 128$ (left) and $w = 256$ (right), and the problem dimension to $n = 2500$ (top) and $n = 10000$ (bottom).

small bandwidths, A_1^R is consequently small, and its execution time does not affect the overall execution time of the reduction; for large bandwidths, the multi-threaded execution of A_1^R is the preferred choice.) Therefore, for clarity, we removed the first mapping from the following plots.

- **Variant V2:** Look-ahead variant for problems with $2b > w$. Two different mappings are possible, depending on which threads update A_1, X_1, X_2, X_3 .
 - A_1 on T_S (while X_1, X_2 , and X_3 are computed concurrently on T_P).
 - A_1 on $T_S + T_P$ (after which, X_1, X_2, X_3 are updated by the same $T_S + T_P$ threads).

The plots in the left-hand side of Figure 9 and Figure 10 report the GFLOPS rates attained by the configurations (namely algorithms/variants/mappings) for several bandwidths in the range 32–256. The right-hand side plots in both figures illustrate the optimal block size for each problem dimension and configuration, showing the crucial role of this parameter.

The first conclusion that can be extracted from the plots is that the performance of the two Variants V1 and V2 enhanced with look-ahead depends on the ratio between the algorithmic block size b (equivalent to the number of columns in the panel), the target matrix bandwidth w , and the problem dimension n .

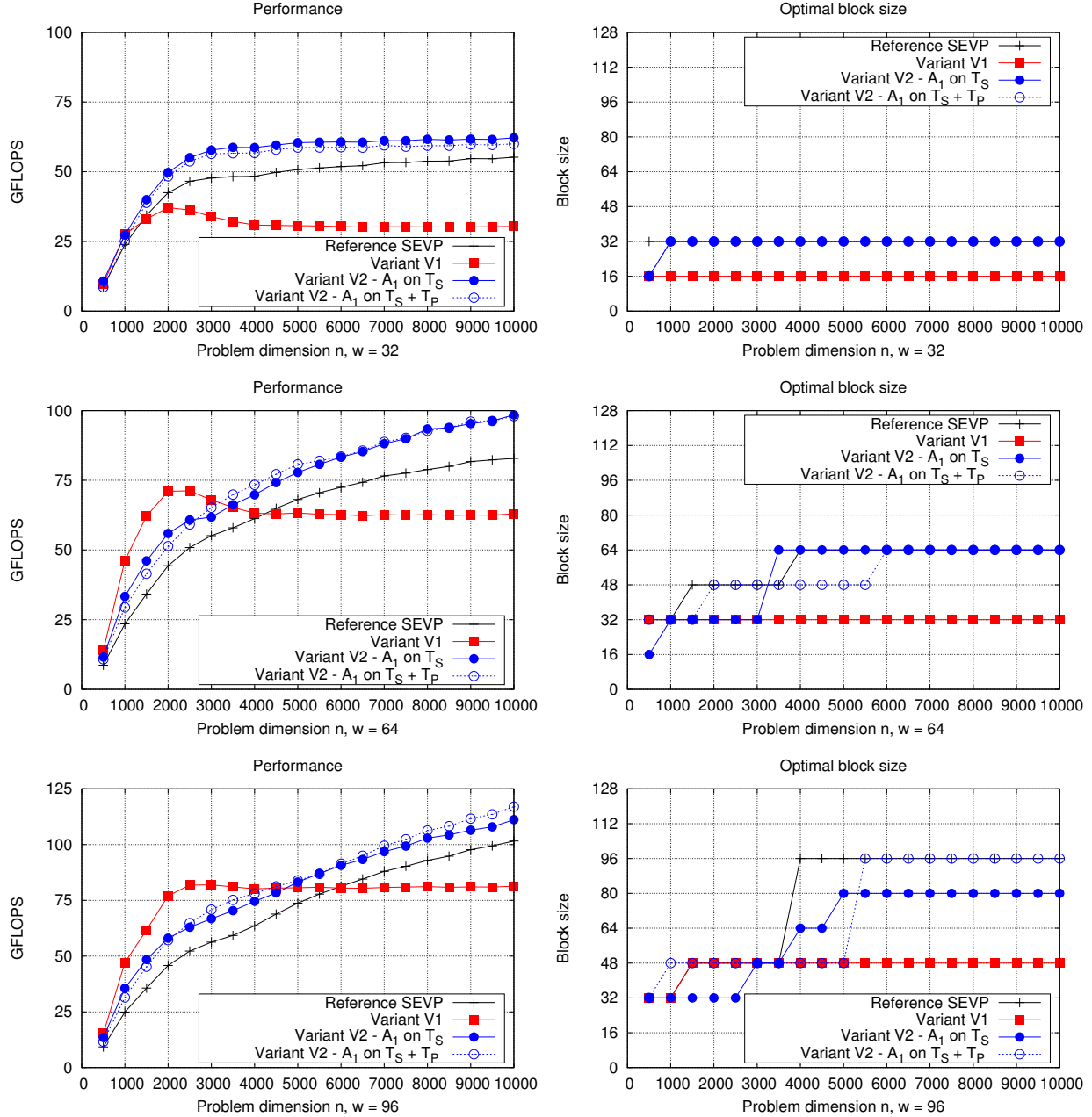


Figure 9: Performance vs problem dimension (left) of the SEVP implementations with $w = 32, 64$ and 96 ; and optimal block size vs problem dimension (right).

Focusing on Variant V1, we identify a drawback due to the limitation imposed on the block size by the condition $2b \leq w$. This implies that, for small bandwidths, Variant V1 can only employ very reduced block sizes. In consequence, the invocation to the Level-3 BLAS to perform the trailing update cannot efficiently exploit all resources of the processor. To illustrate this behavior, let us focus on the experiments with $w = 64$ in Figure 9. The right-hand side plot there shows that, for

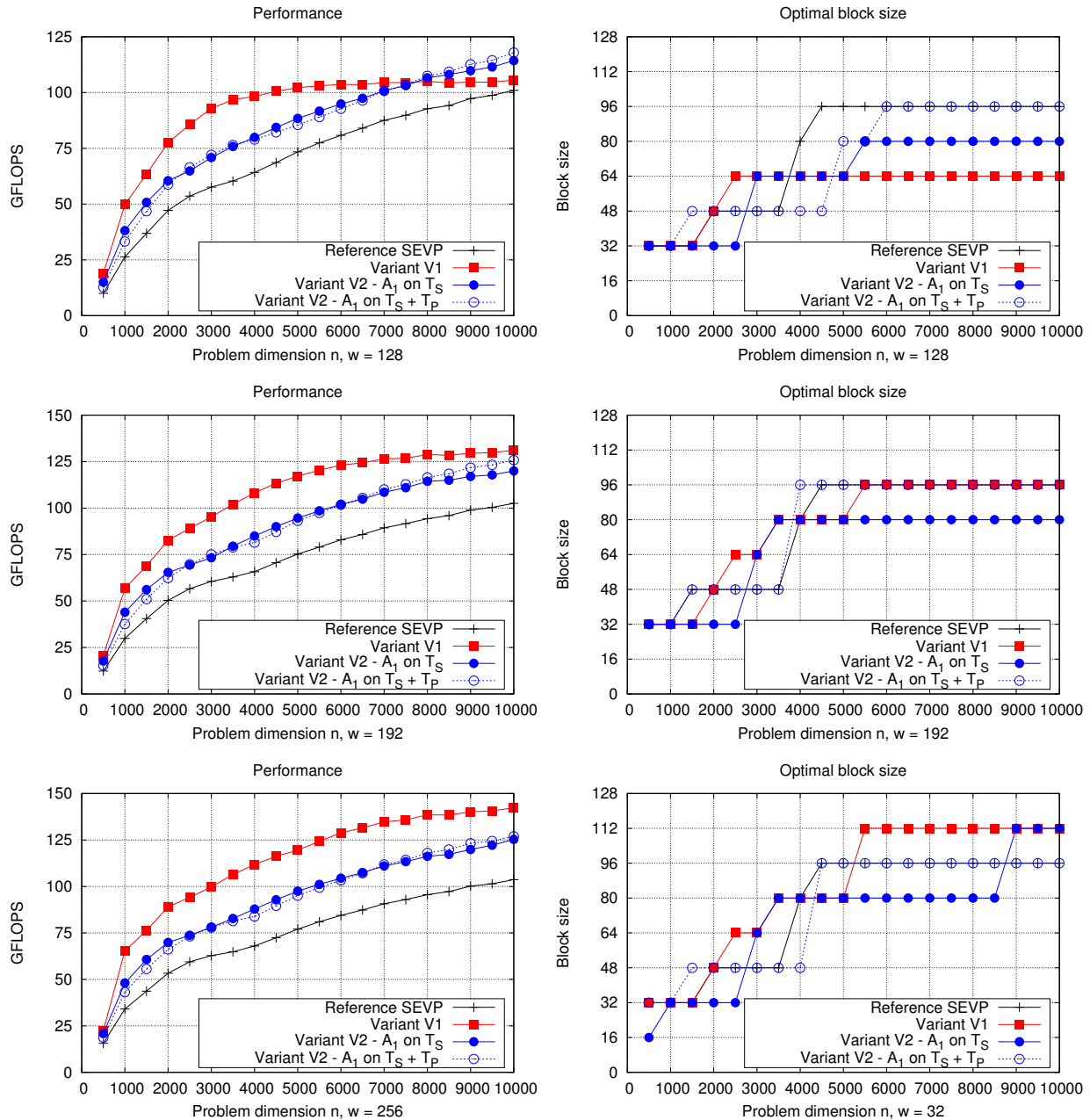


Figure 10: Performance vs problem dimension (left) of the SEVP implementations with $w = 128, 192$ and 256 ; and optimal block size vs problem dimension (right).

Variant V1, the block size is always 32, which is smaller than those selected for Variant V2 and the reference implementation. In addition, the left-hand side plot shows that, despite Variant V1 integrates look-ahead, its performance is inferior to that of the reference implementation for large problem dimensions as the overall execution time in those cases is dominated by the trailing update. In contrast, in the plots using larger bandwidths (i. e. $w = 128$ and 256) we observe that this fact

enables the selection of larger block sizes, considerably improving the throughput of Variant V1, which now outperforms all other configurations.

Our implementations of Variant V2 always improve the performance of the reference routine; and also that of Variant V1 for small bandwidths, and for medium-size bandwidths combined with small problem dimensions. Indeed, as there are no strict restrictions on the block size for Variant V2 (other than $b \leq w$), large values can be selected for b , and the Level-3 BLAS for the trailing update tend to deliver a good fraction of the peak performance even for small bandwidths. However, the drawback of Variant V2 lies in the parts of the algorithm that may be overlapped as part of the look-ahead strategy. In particular, as $2b$ can be larger than w , the factorization of \bar{A}_0 cannot start till the updates of A_1 and X_3 are completed (requiring a synchronization point after them). This implies that only the factorization of \bar{A}_0 and the update of A_2^L can be overlapped with the update of A_2^R . In contrast, for Variant V1, the factorization of \bar{A}_0 only requires that the update of A_1^L is completed, therefore removing this synchronization point; in consequence, the update of A_1^L and the factorization of \bar{A}_0 can be overlapped with the update of A_2 .

To close the experiments in this subsection, we evaluate the impact that the TSR algorithms for SEVP make on the overall computation of the eigenvalue decomposition. We remind that, in the two-stage reduction to tridiagonal form, the matrix is reduced to a symmetric band form employing one of the TSR algorithms presented earlier in this subsection (the SBR-based Reference, Variant V1 or Variant V2) and this banded matrix is next reduced to tridiagonal form. For the second stage, in our evaluation we will employ routine SBRDT from the SBR package. After that, the eigenvalues are obtained using routine DSTERF from LAPACK. Alternatively, when using the traditional solver for SEVP in LAPACK, the input matrix is directly reduced to tridiagonal form, using routine DSYTRD routine from LAPACK, after which routine DSTERF is applied to obtain the eigenvalues. Routines SBRDT, DSTERF and DSYTRD are mostly composed of calls to kernels in the Level-1 and Level-2 of BLAS. Due to the sequential implementation and lack of optimization of these kernels in BLIS, in our experiments we linked these routines to Intel MKL. (The initial reduction to band form via the TSR algorithms was still performed using the kernels from BLIS.)

Table 1 reports the execution time of the different stages that are present in the solution of SEVP as well as the acceleration with respect to the single-stage reduction approach to tridiagonal form for three problem sizes and several bandwidth dimensions. These results show that, for the smallest problem, as the symmetric matrix fits in the L3 cache on chip, the best option is to employ the conventional solver in LAPACK, based on routines DSYTRD+DSTERF. On the other hand, for the larger two problems, the best option corresponds to the two-stage reduction to tridiagonal form, using Variant V2 with a narrow bandwidth $w = 64$ in both cases. In particular, the speed-ups with respect to LAPACK's solver with a single-stage reduction are 1.96 and 2.78 when the complete process is considered. Focussing on the two-stage approach, the results also expose the need to limit the bandwidth of the compact form as the cost of routine SBRDT rapidly grows with w . Finally, the table reveals that the speed-ups observed for Variant V2 with respect to the reference implementation vary between 1.16 and 1.19 for the two largest problem sizes and $w = 64$. At this point, we note that the contribution of the new TSR to band form to the total cost of the eigenvalue computation is largely dependent on the implementation and efficiency of the subsequent stages. (Thus, for example, the results can be significantly different if one employs a solver that directly obtains the eigenvalues from the band form, without requiring the reduction to tridiagonal form [20], or just applies an iterative solver on the band matrix to obtain a few selected eigenvalues [1].) However, the acceleration factors observed for Variants V1 and V2 with respect to the reference implementation in the first stage will remain constant.

n	w	Variants			SBRDT	DSTERF	Speed-up vs		
		TSR to band form					DSYTRD + DSTERF		
		Ref	V1	V2			Ref	V1	V2
2000	32	0.25	0.30	0.22	0.20	0.09	0.44	0.41	0.47
	64	0.24	0.16	0.19	0.33	0.09	0.36	0.41	0.39
	96	0.24	0.14	0.19	0.43	0.09	0.32	0.36	0.34
	128	0.23	0.14	0.18	0.93	0.11	0.19	0.20	0.20
	192	0.22	0.13	0.17	1.21	0.11	0.15	0.16	0.16
	256	0.20	0.12	0.16	1.27	0.11	0.15	0.16	0.16
6000	32	5.55	9.50	4.76	1.51	0.74	1.69	1.12	1.89
	64	3.97	4.62	3.42	2.58	0.74	1.81	1.66	1.96
	96	3.54	3.57	3.18	3.71	0.74	1.64	1.65	1.73
	128	3.55	2.78	3.05	8.98	0.78	0.99	1.05	1.03
	192	3.48	2.34	2.82	12.08	0.78	0.81	0.87	0.84
	256	3.41	2.23	2.77	14.40	0.78	0.71	0.76	0.74
10000	32	24.15	43.85	21.52	4.18	2.02	2.13	1.29	2.33
	64	16.11	21.26	13.50	7.78	2.02	2.50	2.08	2.78
	96	13.14	16.41	12.01	10.68	2.02	2.50	2.22	2.62
	128	13.15	12.59	11.70	25.72	2.05	1.58	1.60	1.64
	192	12.99	10.15	11.09	35.14	2.05	1.29	1.37	1.34
	256	12.86	9.28	10.65	41.88	2.06	1.14	1.22	1.19

Table 1: Execution time (in sec.) of the different stages for the solution of the SEVP (computation of the eigenvalues only) via the TSR algorithms and speed-up with respect to the conventional approach in LAPACK (DSYTRD+DSTERF).

4.4 Performance of TSR to band form for the SVD.

We next analyze the performance behavior of the multi-threaded variants with look-ahead aimed to enhance the computational throughput of the TSR algorithms for the first stage of the SVD. Specifically, the following implementations are compared:

- **Reference implementations:** These routines depart from the one presented for SEVP in that the matrix is not symmetric and, therefore, distinct left and right panel factorizations are required. In addition, two different implementations are possible for the SVD, depending on how the trailing update is performed:
 - Reference SVD. This case adheres to the formulation in equations (18)–(23). At each iteration this implementation first computes the QR factorization; then applies the resulting orthogonal matrix U to the trailing submatrix; next computes the subsequent LQ factorization; and finally applies the resulting V to the trailing submatrix.
 - Reference SVD Simultaneous. This implementation performs the update of the trailing submatrix as in (27). That is, at each iteration of the algorithm, the QR and LQ factorizations are performed first; and then the updates of the trailing submatrix with U and V are fused into a single “step”.
- **Variant V1:** Look-ahead variant of the “Reference SVD” implementation for problems with $2b \leq w$. Similarly to Variant V1 for SEVP, two different mappings of the updates of B_1 and C_1 are possible, where these two blocks are split into two independent sub-blocks (B_1^L and

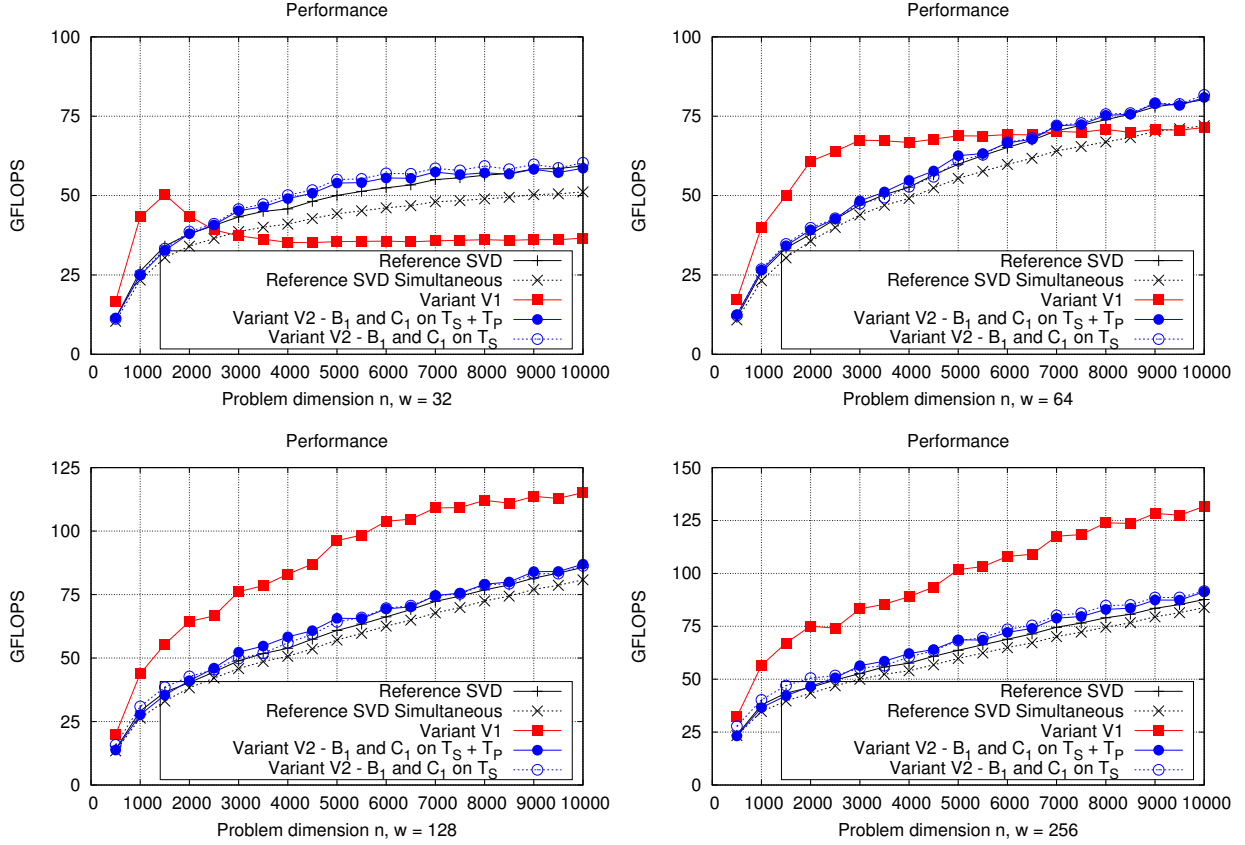


Figure 11: Performance vs problem dimension of the SVD implementations with $w = 32, 64, 128,$ and 256 .

$B_1^R; C_1^T$ and B_1^B). However, for similar reasons, in the experiments we only consider the case with B_1^R and C_1^B updated by T_P .

- **Variant V2:** Look-ahead variant of the “Reference SVD Simultaneous” implementation for problems with $2b > w$. Two different mappings are possible, depending on which threads update B_1, C_1, Z_L, Z_R, X .
 - B_1 and C_1 on T_S (while $Z_L, Z_R,$ and X are computed concurrently on T_P).
 - B_1 and C_1 on $T_S + T_P$ (after which, $Z_L, Z_R,$ and X are updated by the same $T_S + T_P$ threads).

Following the optimizations presented earlier for the SBR routine, all routines for the SVD perform the factorization of the panels via Level-3 BLAS procedures, and compute the matrices W_U, W_V from the product of the corresponding compact WY factors T_U, T_V and Y_U, Y_V .

Figure 11 reports the GFLOPS rates attained by the configurations for bandwidths ranging from 32 to 256 and square matrices. For brevity, the analysis of the optimal block size is not presented as it revealed a similar behavior as that observed for SEVP. Let us focus first on the implementations without look-ahead. From the plots, it is clear that the “Reference SVD” implementation

outperforms its “Reference SVD Simultaneous” counterpart. At this point we would remark that the second implementation underlies variant V2 of the SVD algorithm with look-ahead.

Focusing on Variant V1, we detect the same drawback as that identified in Variant V1 for SEVP in that, for small bandwidths, the block size is strongly constrained. In contrast, large performance improvements are reported, compared with all other implementations, for medium and large bandwidths.

A less pleasant case is encountered for Variant V2, which is not able to improve significantly the performance results of the “Reference SVD” implementation for any bandwidth nor problem dimension though it outperforms its baseline “Reference SVD Simultaneous” implementation. The reason for this result is that, for this implementation, the update of D_{22} cannot be fully overlapped with the execution time of the next panel factorizations (both left and right). For Variant V1, the execution of the panel factorizations is overlapped with the updates of B_1^R , C_1^B , and D ; but due to the data dependencies in Variant V2, we can only overlap the execution of the panel factorizations with the update of D_{22} , which exhibits a considerably more reduced number of flops.

Figure 12 displays the GFLOPS rates observed for bandwidths ranging from 32 to 256 and non-square matrices. In the plots, the m -dimension on the matrices is fixed to 10000, while the n -dimension is varied in the range 500–10000 in steps of 500. The plots reveal performance numbers that are very close to those observed for the reductions of square matrices, showing that the new variants for TSR are not sensitive to the matrices shape.

5 Concluding Remarks

We have analyzed in detail the impact that static look-ahead exerts on the performance of two-sided routines that perform the reduction to compact band forms for SEVP and the SVD. Our study shows that a correct selection of the look-ahead variant as well as an appropriate mapping of tasks to cores are key to optimize performance. Even more importantly, the block size plays a crucial role in the computational throughput of these reduction routines. Decoupling this parameter from the target bandwidth is a must, and therefore we have to depart from the solution adopted in the corresponding routines included in the current versions of LAPACK, PLASMA and MAGMA, which simply set the block size to equal the bandwidth.

For the SVD, our analysis also advocates for an alternative option that reduces the original dense matrix to a band form with the same upper and lower bandwidths, allowing an efficient exploitation of the look-ahead strategy. This choice thus overcomes some of the difficulties of the traditional reduction to band-triangular from that is adopted in LAPACK and MAGMA.

Acknowledgements

This research was partially sponsored by projects TIN2014-53495-R and TIN2015-65316-P of the Spanish *Ministerio de Economía y Competitividad*, project 2014-SGR-1051 from the Generalitat de Catalunya, and the EU H2020 project 732631 OPRECOMP.

References

- [1] Jos I. Aliaga, Pedro Alonso, Jos M. Bada, Pablo Chacn, Davor Davidovi, Jos R. Lpez-Blanco, and Enrique S. Quintana-Ort. A fast bandkrylov eigensolver for macromolecular functional

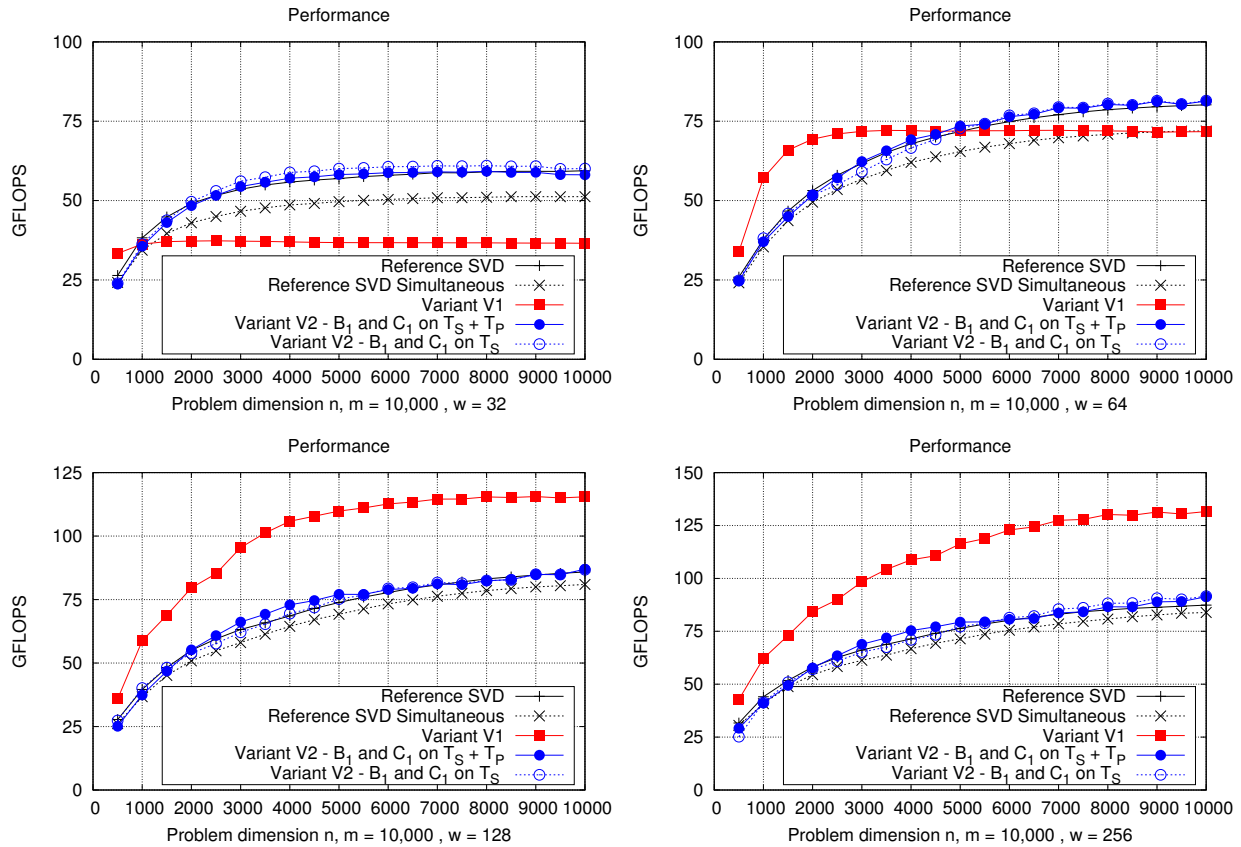


Figure 12: Performance vs problem dimension of the SVD implementations for rectangular matrices with $w = 32, 64, 128,$ and 256 .

- motion simulation on multicore architectures and graphics processors. *Journal of Computational Physics*, 309(Supplement C):314 – 323, 2016.
- [2] Edward Anderson, Zhaojun Bai, L. Susan Blackford, James Demmel, Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, Anne Greenbaum, Alan McKenney, and Danny C. Sorensen. *LAPACK Users' guide*. SIAM, 3rd edition, 1999.
 - [3] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, N. Knight, and H. D. Nguyen. Reconstructing householder vectors from tall-skinny QR. *J. Parallel & Distributed Comp.*, 85:3–31, 2015.
 - [4] Paolo Bientinesi, Francisco D. Igual, Daniel Kressner, Matthias Petschow, and Enrique S. Quintana-Ortí. Condensed forms for the symmetric eigenvalue problem on multi-threaded architectures. *Concurrency & Comp.: Practice & Exp.*, 23(7):694–707, 2011.
 - [5] C. H. Bischof, B. Lang, and X. Sun. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM Trans. Math. Soft.*, 26(4):602–616, 2000.
 - [6] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

- [7] Anthony M. Castaldo, R. Clint Whaley, and Siju Samuel. Scaling LAPACK panel operations using parallel cache assignment. *ACM Trans. Math. Softw.*, 39(4):22:1–22:30, July 2013.
- [8] Sandra Catalán, José R. Herrero, Enrique S. Quintana-Ortí, Rafael Rodríguez-Sánchez, and Robert A. van de Geijn. A case for malleable thread-level linear algebra libraries: The LU factorization with partial pivoting. *CoRR*, abs/1611.06365, 2016.
- [9] D. Davidović and E. S. Quintana-Ortí. Applying OOC techniques in the reduction to condensed form for very large symmetric eigenproblems on GPUs. In *Proceedings of the 20th Euromicro Conference on Parallel, Distributed and Network based Processing – PDP 2012*, pages 442–449, 2012.
- [10] T. A. Davis and S. Rajamanickam. Algorithm 8xx: PIRO BAND, pipelined plane rotations for band reduction. *ACM Trans. Math. Softw.* Submitted.
- [11] Inderjit S. Dhillon, Beresford N. Parlett, and Christof Vömel. The design and implementation of the MRRR algorithm. *ACM Trans. Math. Softw.*, 32(4):533–560, Dec. 2006.
- [12] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.
- [13] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, March 1988.
- [14] K. Vince Fernando and Beresford N. Parlett. Accurate singular values and differential QD algorithms. *Numer. Mathematik*, 67(2):191–229, 1994.
- [15] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.
- [16] Benedikt Grosser and Bruno Lang. Efficient parallel reduction to bidiagonal form. *Parallel Computing*, 25(8):969 – 986, 1999.
- [17] Ming Gu and Stanley C. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Matrix Analysis & Appl.*, 16(1):79–92, 1995.
- [18] A. Haidar, H. Ltaief, and J. Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, Nov 2011.
- [19] Azzam Haidar, Jakub Kurzak, and Piotr 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*, pages 90:1–90:12, New York, NY, USA, 2013. ACM.
- [20] Michael Moldaschl and Wilfried N. Gansterer. Comparison of eigensolvers for symmetric band matrices. *Sci. Comput. Program.*, 90(PA):55–66, Sept. 2014.
- [21] Matthias Petschow, Elmar Peise, and Paolo Bientinesi. High-performance solvers for dense Hermitian eigenproblems. *SIAM J. Scientific Comp.*, 35(1):C1–C22, Jan. 2013.

- [22] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, 2009.
- [23] Peter Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Technical Report TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.
- [24] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. The BLIS framework: Experiments in portability. *ACM Trans. Math. Softw.*, 42(2):12:1–12:19, June 2016.