Implicit Hari–Zimmermann algorithm for the generalized SVD on the GPUs

Journal Title
XX(X):1–37

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Vedran Novaković[1] and Sanja Singer[2]

Abstract

A parallel, blocked, one-sided Hari–Zimmermann algorithm for the generalized singular value decomposition (GSVD) of a real or a complex matrix pair (F, G) is here proposed, where F and G have the same number of columns, and are both of the full column rank. The algorithm targets either a single graphics processing unit (GPU), or a cluster of those, performs all non-trivial computation exclusively on the GPUs, requires the minimal amount of memory to be reasonably expected, scales acceptably with the increase of the number of GPUs available, and guarantees the reproducible, bitwise identical output of the runs repeated over the same input and with the same number of GPUs.

Keywords

generalized singular value decomposition, generalized eigendecomposition, graphics processing units, implicit Hari–Zimmermann algorithm, hierarchical blocking

1 Introduction

The two-sided Hari–Zimmermann algorithm Hari (1984, 2018, 2019); Zimmermann (1969) is a Jacobi-type method for computing the generalized eigenvalue decomposition (GEVD) of a matrix pair (A, B), where both matrices are Hermitian of the same order and B is positive definite.

If A and B are instead given *implicitly* by their factors F and G (not necessarily square nor with the same number of rows), respectively, such that $(A, B) = (F^*F, G^*G)$, then the GEVD of (A, B) can be computed implicitly, i.e., without assembling A and B in entirety from the factors, by a modification of the Hari–Zimmermann algorithm Novaković et al. (2015). However, pivot submatrices of A and B of a certain, usually small order are formed explicitly throughout the computation.

The modified algorithm is a method that jointly orthogonalizes the pairs of columns of F and G by a sequence of transformations that are applied from the right side of the factors only. Such a one-sided algorithm computes U, Σ_F, V, Σ_G, and Z, where $FZ = U\Sigma_F$, $GZ = V\Sigma_G$, and $U^*U = V^*V = I$. The matrix Z is square and nonsingular, while Σ_F and Σ_G are non-negative, diagonal, and scaled such that $\Sigma_F^2 + \Sigma_G^2 = I$. The method thus implicitly computes the GEVD of (A, B), but *explicitly* the generalized singular value decomposition (GSVD; see, e.g., Paige and Saunders (1981); Van Loan (1976)) of (F, G), with the generalized singular values forming the diagonal of $\Sigma := \Sigma_G^{-1}\Sigma_F$ (all of them finite, since Σ_G has a positive diagonal). Furthermore, the generalized singular values can be considered to be sorted descendingly by a symmetric permutation, i.e., $\Sigma = P_0^T\Sigma' P_0$, and thus $U = U'P_0$, $V = V'P_0$, and $Z = Z'P_0$, where $FZ' = U'\Sigma'_F$, $GZ' = V'\Sigma'_G$, and $\Sigma' = \Sigma_G'^{-1}\Sigma'_F$ constitute a decomposition of (F, G) possessing all other aforementioned properties.

The GEVD of (A, B), if required, can be recovered by letting $\Lambda := \Sigma^2$ and noting that $AZ = BZ\Lambda$, i.e., the columns of Z are the generalized eigenvectors, and the diagonal of Λ contains the generalized eigenvalues of (A, B). However, the converse is *not* numerically sound, i.e., the GEVD should not, in general, be used for computing the GSVD. For a further clarification, see Appendix G.

The right generalized singular vectors $X := Z^{-1}$, if needed, can either be computed from Z, or can be obtained simultaneously with Z by accumulating the inverses of the transformations that have been multiplied to form Z Singer et al. (2020). With $\widetilde{\Theta}$ from subsection 2.4, if

$$Z = Z_0\widetilde{Z}\widetilde{\Theta} = Z_0 \cdot \widetilde{Z}_0 \cdot \widetilde{Z}_1 \cdots \widetilde{Z}_N \cdot \widetilde{\Theta},$$

[1]Completed a major part of this research while being affiliated to Universidad Jaime I, Av. Vicent Sos Baynat, 12071 Castellón de la Plana, Spain
[2]University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture, Ivana Lučića 5, 10000 Zagreb, Croatia

Corresponding author:
Sanja Singer, University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture, Ivana Lučića 5, 10000 Zagreb, Croatia
Email: ssinger@fsb.hr

when $N+1$ transformations have been applied, then

$$X = \widetilde{\Theta}^{-1} \cdot \widetilde{Z}_N^{-1} \cdot \widetilde{Z}_{N-1}^{-1} \cdots \widetilde{Z}_0^{-1} \cdot Z_0^{-1}.$$

The recent work Novaković et al. (2015) has shown that such method can be blocked and parallelized for the shared memory nodes and for the clusters of those, albeit only the real matrix pairs were considered therein. Even the sequential but blocked version outperformed the GSVD algorithm in LAPACK Anderson et al. (1999), and the parallel ones exhibited a decent scalability.

On the other hand, an efficient, parallel and blocked one-sided Jacobi-type algorithm for the "ordinary" and the hyperbolic SVD Novaković (2015, 2017) of a single real matrix has been developed for the GPUs, that utilizes the GPUs almost fully, with the CPU serving only the controlling purpose in the single-GPU case.

This work aims to merge the experience of those two approaches, and present a parallel and blocked one-sided (also called "implicit") Hari–Zimmermann algorithm for the GSVD on the GPU(s) as an extension of the latter, but for the complex matrix pairs as well as for the real ones.

Even though the research in parallelization of the GSVD has a long history Bai (1994); Luk (1985), three novel and major differences from the earlier, Kogbetliantz-based procedures aim to ensure both the high performance and the high relative accuracy of this one: using the implicit Hari–Zimmermann algorithm as the basic method, that is blocked to exploit the GPU memory hierarchy, and the massive parallelism of the GPUs that suits the algorithm (and vice versa) perfectly.

In the last twenty years, many applications of GSVD have been found in science and technology. To mention just a few applications, the GSVD is used for dimension reduction for clustered text data Howland et al. (2003) and for face recognition algorithms Howland et al. (2006), where in both cases the matrix pair is naturally given implicitly, i.e., in a factored form.

In Alter et al. (2003) the GSVD serves for comparison of two different organisms to find their biological similarities based on a genome-scale expression data sets. Also, the GSVD can be used in beamforming Senaratne and Tellambura (2013) and separation of partially overlapping data packets Zhou and van der Veen (2017) in communication systems, machine condition monitoring when looking for symptoms of wear Cempel (2009), and filtering of brain activities while preforming two different tasks Zhao et al. (2010). In the last case, matrices could be very large.

This paper continues with section 2, where the complex and the real one-sided Hari–Zimmermann algorithms are introduced, together with the general, architecturally agnostic principles of their blocking and parallelization. In section 3 the single-GPU implementation are described in detail, while in section 4 the most straightforward multi-GPU implementation approach is suggested. The numerical testing results are summarized in section 5, and the paper concludes with some directions for future research in section 6. In Appendix A a non-essential method for enhancing the accuracy of the real and the complex dot-products on the GPUs is proposed.

2 The complex and the real one-sided Hari–Zimmermann algorithms

In this section the complex and the real one-sided Hari–Zimmermann algorithms are briefly described. Please see Hari (1984, 2018, 2019) for a more thorough overview of the two-sided algorithms, and Novaković et al. (2015) for a detailed explanation of the real implicit Hari–Zimmermann algorithm. In this paper the terminology and the implementation decisions of Singer et al. (2020), where the complex generalized hyperbolic SVD based on the implicit Hari–Zimmermann approach has been introduced, are closely followed, but without the hyperbolic scalar products (i.e., the signature matrix J is taken to be identity here) and without forming the right generalized singular vectors X from Z.

Let the matrices F and G be of size $m_F \times n$ and $m_G \times n$, respectively, with $\min\{m_F, m_G\} \geq n$. Then, Z is square of order n, and assume that $n \geq 2$. Otherwise, for $n = 1$, the GSVD of (F, G) is obtained by taking

$$\begin{aligned} U &:= \|F\|_F^{-1} F, & \Sigma_F &:= \frac{\|F\|_F}{\sqrt{\|F\|_F^2 + \|G\|_F^2}}, \\ V &:= \|G\|_F^{-1} G, & & \\ Z &:= \frac{1}{\sqrt{\|F\|_F^2 + \|G\|_F^2}}, & \Sigma_G &:= \frac{\|G\|_F}{\sqrt{\|F\|_F^2 + \|G\|_F^2}}. \end{aligned}$$

Even though the algorithm works on the rectangular matrices, it might be beneficial performance-wise to avoid transforming very tall and skinny (block)columns by working on the square matrices instead. To shorten F and G, the problem is transformed by computing the QR factorization of F with the column pivoting, $FP_1 = Q_F R_F$, and then G, with its columns prepermuted by P_1, is shortened by the column-pivoted QR factorization, $(GP_1)P_2 = Q_G R_G$. The square matrices $F'' := R_F P_2$ and $G'' := R_G$, both of order n, take the place of F and G in the algorithm, respectively. With $\Sigma = \Sigma''$ in the decompositions of (F, G) and of (F'', G''), the matrix Z from the former, sought-for decomposition can be recovered by using $P'' := P_1 P_2$ and the computed Z'' from the latter as $Z := P'' Z''$.

It is assumed that $\mathrm{diag}(B) = I$, i.e., the column norms of G are unity. Should it not be the case, F and G are then prescaled by a nonsingular, diagonal matrix Z_0, where $(Z_0)_{jj} := 1/\sqrt{\|g_j\|_F}$, g_j is the jth column of G and $1 \leq$

$j \leq n$; otherwise, $Z_0 := I$. The iterative transformation phase starts with the matrix pair (F_0, G_0), where $F_0 := FZ_0$, and $G_0 := GZ_0$. Implicitly, A and B have been transformed by a congruence with Z_0 as $A_0 := F_0^* F_0$ and $B_0 := G_0^* G_0$.

2.1 Simultaneous diagonalization of a pair of pivot matrices

An iteration (or "step") $k \geq 0$ of the sequential non-blocked Hari–Zimmermann algorithm consists of selecting a pair of indices (i_k, j_k), $1 \leq i_k < j_k \leq n$, and thus two 2×2 pivot submatrices, one of $A_k := F_k^* F_k$,

$$\widehat{A}_k := \begin{bmatrix} a_{i_k i_k;k} & a_{i_k j_k;k} \\ \bar{a}_{i_k j_k;k} & a_{j_k j_k;k} \end{bmatrix} = \begin{bmatrix} f_{i_k;k}^* f_{i_k;k} & f_{i_k;k}^* f_{j_k;k} \\ f_{j_k;k}^* f_{i_k;k} & f_{j_k;k}^* f_{j_k;k} \end{bmatrix},$$

and one of $B_k := G_k^* G_k$,

$$\widehat{B}_k := \begin{bmatrix} 1 & b_{i_k j_k;k} \\ \bar{b}_{i_k j_k;k} & 1 \end{bmatrix} = \begin{bmatrix} 1 & g_{i_k;k}^* g_{j_k;k} \\ g_{j_k;k}^* g_{i_k;k} & 1 \end{bmatrix},$$

which are then jointly diagonalized by a congruence transformation with a nonsingular matrix $\widehat{Z}_k$, to be defined in subsections 2.1.1 and 2.1.2, as

$$\widehat{A}_{k+1} := \widehat{Z}_k^* \widehat{A}_k \widehat{Z}_k = \begin{bmatrix} a_{i_k i_k;k+1} & 0 \\ 0 & a_{j_k j_k;k+1} \end{bmatrix},$$
$$\widehat{B}_{k+1} := \widehat{Z}_k^* \widehat{B}_k \widehat{Z}_k = I_2.$$

If $\widehat{Z}_k$ is embedded into an $n \times n$ matrix $\widetilde{Z}_k$ such that $\widetilde{Z}_{i_k i_k;k} := \widehat{Z}_{11;k}$, $\widetilde{Z}_{i_k j_k;k} := \widehat{Z}_{12;k}$, $\widetilde{Z}_{j_k i_k;k} := \widehat{Z}_{21;k}$, $\widetilde{Z}_{j_k j_k;k} := \widehat{Z}_{22;k}$, while letting $\widetilde{Z}_k$ be the identity matrix elsewhere, then looking two-sidedly the congruence with $\widetilde{Z}_k$ transforms the pair (A_k, B_k) into a pair (A_{k+1}, B_{k+1}), where $A_{k+1} := \widetilde{Z}_k^* A_k \widetilde{Z}_k$ and $B_{k+1} := \widetilde{Z}_k^* B_k \widetilde{Z}_k$. One-sidedly, the transformation by $\widetilde{Z}_k$ orthogonalizes the i_kth and the j_kth pivot columns of F_k and G_k to obtain $F_{k+1} := F_k \widetilde{Z}_k$ and $G_{k+1} := G_k \widetilde{Z}_k$. Also, $\widetilde{Z}_k$ is accumulated into the product $Z_{k+1} := Z_k \widetilde{Z}_k$. In a one-sided sequential step only the i_kth and the j_kth columns of F_k, G_k, and Z_k are effectively transformed, in-place (i.e., overwritten), postmultiplying them by the 2×2 matrix $\widehat{Z}_k$, while the other columns of these matrices remain intact:

$$\begin{bmatrix} f_{i_k;k+1} & f_{j_k;k+1} \end{bmatrix} = \begin{bmatrix} f_{i_k;k} & f_{j_k;k} \end{bmatrix} \cdot \widehat{Z}_k,$$
$$\begin{bmatrix} g_{i_k;k+1} & g_{j_k;k+1} \end{bmatrix} = \begin{bmatrix} g_{i_k;k} & g_{j_k;k} \end{bmatrix} \cdot \widehat{Z}_k,$$
$$\begin{bmatrix} z_{i_k;k+1} & z_{j_k;k+1} \end{bmatrix} = \begin{bmatrix} z_{i_k;k} & z_{j_k;k} \end{bmatrix} \cdot \widehat{Z}_k.$$

As $\operatorname{diag}(\widehat{B}_{k+1}) = \operatorname{diag}(\widehat{B}_k) = I_2$, it follows that $\operatorname{diag}(B_{k+1}) = \operatorname{diag}(B_k) = I_n$. However, due to the floating-point rounding errors, these equations might not hold. To prevent $\operatorname{diag}(\widehat{B}_k)$ to drift too far away from $\operatorname{diag}(I_2)$ as the algorithm progresses, the squared Frobenius norms of $g_{i_k;k}$ and $g_{j_k;k}$ could be recomputed for each k as $b_{i_k i_k;k} = g_{i_k;k}^* g_{i_k;k}$ and $b_{j_k j_k;k} = g_{j_k;k}^* g_{j_k;k}$. Then, a rescaling of $\widehat{A}_k$ and $\widehat{B}_k$ as $\widehat{A}'_k := \widehat{D}_k^* \widehat{A}_k \widehat{D}_k$ and $\widehat{B}'_k := \widehat{D}_k^* \widehat{B}_k \widehat{D}_k$, by a diagonal matrix $\widehat{D}_k$ such that $\widehat{D}_{11;k} = 1/\sqrt{b_{i_k i_k;k}}$ and $\widehat{D}_{22;k} = 1/\sqrt{b_{j_k j_k;k}}$, should bring back $\operatorname{diag}(\widehat{B}_k)$ close to $\operatorname{diag}(I_2)$. From $\widehat{A}'_k$ and $\widehat{B}'_k$ it is then possible to compute $\widehat{Z}'_k$, with the final $\widehat{Z}_k := \widehat{D}_k \widehat{Z}'_k$. In this version of the algorithm it is not necessary to rescale the columns of F and G by $\widetilde{Z}_0$ at the start, since such rescaling happens at each step, so $\widetilde{Z}_0 := I$. If $\widehat{D}_k = I_2$, this version is equivalent to the standard (previously described) one, for which it can be formally set $\widehat{A}'_k := \widehat{A}_k$ and $\widehat{B}'_k := \widehat{B}_k$.

Suppose that $\widehat{Z}''_k$ has been computed (by either version) such that it diagonalizes $\widehat{A}_k$ and $\widehat{B}_k$, but $a_{i_k i_k;k+1} < a_{j_k j_k;k+1}$. To keep $\operatorname{diag}(\widehat{A}_k)$ sorted descendingly, swap the columns of $\widehat{Z}''_k$ by a permutation $\widehat{P}_k := \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ to obtain $\widehat{Z}_k := \widehat{Z}''_k \widehat{P}_k$. Such $\widehat{Z}_k$ will swap the i_kth and the j_kth columns of F_k and G_k as it orthogonalizes them. Sorting in each step is a heuristic that speeds up the algorithm notably in practice (see section 5), but it makes reasoning about the convergence harder and is not strictly necessary.

Computing $\widehat{Z}_k$ from $\widehat{A}_k$ and $\widehat{B}_k$ is more involved in the complex case than in the real one. However, in both cases, first it is established whether the i_kth and the j_kth columns of F_k and G_k are numerically relatively orthogonal,

$$|a'_{i_k j_k;k}| < \sqrt{a'_{i_k i_k;k}} \cdot \sqrt{a'_{j_k j_k;k}} \cdot \varepsilon \cdot \sqrt{n},$$
$$|b'_{i_k j_k;k}| < \varepsilon \cdot \sqrt{n},$$

where ε is the precision of the chosen floating-point datatype. The relation relies on the expected (as opposed to the worst case) rounding error for the dot-products Drmač (1997) that form the elements of $\widehat{A}'_k$ and $\widehat{B}'_k$, and while sensible in the real case, it is probably too tight in the complex case, where a more careful analysis of the complex dot-products might be employed in the future work and a handful of transformations subsequently might be skipped. If the aforesaid columns are relatively orthogonal, no non-trivial transformation is to take place, and $\widehat{Z}_k := \widehat{P}_k$, since still the column swap may be warranted. Rescaling by $\widehat{D}_k$ is thus not performed even for $\widehat{D}_k \neq I_2$, since it might perturb the columns sufficiently enough for them to cease to be numerically orthogonal.

2.1.1 The complex case The transformation matrix $\widehat{Z}'_k$ is sought in a form Hari (1984); Singer et al. (2020)

$$\widehat{Z}'_k := \frac{1}{t_k}\begin{bmatrix} \cos\varphi_k & e^{i\alpha_k}\sin\varphi_k \\ -e^{-i\beta_k}\sin\psi_k & \cos\psi_k \end{bmatrix}.$$

To that end, let $x_k := |b'_{i_kj_k;k}|$, $\zeta_k := \arg(b'_{i_kj_k;k})$, or $\zeta_k := 0$ if $b'_{i_kj_k;k} = 0$, $z_k := e^{-i\zeta_k}a'_{i_kj_k;k}$, and define $\operatorname{sign}(a,b)$ to be $|a|$ with the sign of b for a and b real. Then, let $t_k := \sqrt{1-x_k^2}$, set

$$\begin{aligned} u_k &:= \operatorname{Re}(z_k), \qquad & h_k &:= a'_{j_kj_k;k} - a'_{i_ki_k;k}, \\ v_k &:= \operatorname{Im}(z_k), & \tau_k &:= \operatorname{sign}(1, h_k), \end{aligned}$$

and, noting that $t_k > 0$ since $\widehat{B}'_k$ is positive definite, with these quantities compute

$$\begin{aligned} \tan(2\vartheta_k) &:= \tau_k \frac{2u_k - (a'_{i_ki_k;k} + a'_{j_kj_k;k})x_k}{t_k\sqrt{h_k^2 + 4v_k^2}}, \\ \tan\gamma_k &:= 2\frac{v_k}{h_k}, \end{aligned}$$

where $-\pi/4 < \vartheta_k \le \pi/4$ and $-\pi/2 < \gamma_k \le \pi/2$. In these ranges of the angles, for $\theta \in \{2\vartheta_k, \gamma_k\}$ the trigonometric identities $\cos\theta = 1/(1+\tan^2\theta)$ and $\sin\theta = \tan\theta\cos\theta$ hold when $\theta < \pi/2$. Otherwise, $\tan\theta = \infty$, $\cos\theta = 0$, and $\sin\theta = 1$. Then, compute $c_{2\vartheta} := \cos(2\vartheta_k)$, $s_{2\vartheta} := \sin(2\vartheta_k)$, $c_\gamma := \cos\gamma_k$, and $s_\gamma := \sin\gamma_k$, and with them finally obtain

$$\begin{aligned} \cos\varphi_k &:= \frac{1}{\sqrt{2}}\sqrt{1 + x_k s_{2\vartheta} + t_k c_\gamma c_{2\vartheta}}, \\ \cos\psi_k &:= \frac{1}{\sqrt{2}}\sqrt{1 - x_k s_{2\vartheta} + t_k c_\gamma c_{2\vartheta}}, \\ e^{i\alpha_k}\sin\varphi_k &:= e^{i\zeta_k}\frac{(s_{2\vartheta} - x_k) + it_k s_\gamma c_{2\vartheta}}{2\cos\psi_k}, \\ e^{-i\beta_k}\sin\psi_k &:= e^{-i\zeta_k}\frac{(s_{2\vartheta} + x_k) - it_k s_\gamma c_{2\vartheta}}{2\cos\varphi_k}, \end{aligned}$$

where $0 \le \varphi_k < \pi/2$ and $0 \le \psi_k < \pi/2$.

An exception If $v_k = h_k = 0$, i.e., if $\arg(b'_{i_kj_k;k}) = \arg(a'_{i_kj_k;k})$ and $a'_{i_ki_k;k} = a'_{j_kj_k;k}$, then $\tan\gamma_k$ is undefined, and $\tan(2\vartheta_k)$ might also be. In that case, it can be shown that $\widehat{A}'_k$ and $\widehat{B}'_k$ are diagonalized by

$$\widehat{Z}'_k := \frac{1}{\sqrt{2}}\begin{bmatrix} \frac{1}{\sqrt{1+x}} & \frac{-e^{i\zeta_k}}{\sqrt{1-x}} \\ \frac{e^{-i\zeta_k}}{\sqrt{1+x}} & \frac{1}{\sqrt{1-x}} \end{bmatrix}.$$

2.1.2 The real case The transformation matrix $\widehat{Z}'_k$ is sought in a form Hari (1984); Novaković et al. (2015)

$$\widehat{Z}'_k := \frac{1}{t_k}\begin{bmatrix} \cos\varphi_k & \sin\varphi_k \\ -\sin\psi_k & \cos\psi_k \end{bmatrix}.$$

To that end, let $x_k := b'_{i_kj_k;k}$ and $t_k := \sqrt{1-x_k^2} > 0$. Then, set

$$\begin{aligned} \xi_k &:= \frac{x_k}{\sqrt{1+x_k} + \sqrt{1-x_k}}, \\ \eta_k &:= \frac{x_k}{\left(1+\sqrt{1+x_k}\right)\left(1+\sqrt{1-x_k}\right)}, \end{aligned}$$

and compute

$$\cot(2\vartheta_k) := \frac{t_k(a'_{j_kj_k;k} - a'_{i_ki_k;k})}{2a'_{i_kj_k;k} - (a'_{i_ki_k;k} + a'_{j_kj_k;k})x_k},$$

where $-\pi/4 < \vartheta_k \le \pi/4$.

Note that $\cot(2\vartheta_k)$ and $\cot\vartheta_k$ (and the corresponding tangents) have the same sign in the range of ϑ_k. Assuming that the floating-point arithmetic unit does not trap on $\pm 1/0$ and $1/\infty$, obtain $\tan\vartheta_k$ as

$$\tan\vartheta_k := \frac{\operatorname{sign}(1, \cot(2\vartheta_k))}{|\cot(2\vartheta_k)| + \sqrt{1 + \cot^2(2\vartheta_k)}},$$

and from it $\cos\vartheta_k$ and $\sin\vartheta_k$ using the same trigonometric identities as in the complex case. Finally, compute

$$\begin{aligned} \cos\varphi_k &:= \cos\vartheta_k + \xi_k(\sin\vartheta_k - \eta_k\cos\vartheta_k), \\ \cos\psi_k &:= \cos\vartheta_k - \xi_k(\sin\vartheta_k + \eta_k\cos\vartheta_k), \\ \sin\varphi_k &:= \sin\vartheta_k - \xi_k(\cos\vartheta_k + \eta_k\sin\vartheta_k), \\ \sin\psi_k &:= \sin\vartheta_k + \xi_k(\cos\vartheta_k - \eta_k\sin\vartheta_k), \end{aligned}$$

where $0 \le \varphi_k < \pi/2$ and $0 \le \psi_k < \pi/2$.

An exception Since the real case is in fact a simplification of the complex case, when $\cot(2\vartheta_k)$ is undefined, being $0/0$, i.e., when $a'_{i_ki_k;k} = a'_{j_kj_k;k}$ and $a'_{i_kj_k;k} = a'_{i_ki_k;k}b'_{i_kj_k;k}$ (or, in other words, when $\widehat{A}'_k$ and $\widehat{B}'_k$ are proportional), define

$$\widehat{Z}'_k := \frac{1}{\sqrt{2}}\begin{bmatrix} \frac{1}{\sqrt{1+|x|}} & \frac{-1}{\sqrt{1-|x|}} \\ \frac{1}{\sqrt{1+|x|}} & \frac{1}{\sqrt{1-|x|}} \end{bmatrix}.$$

Figure 1 shows a schematic derivation of the two-sided Hari–Zimmermann transformations. Starting with a pair (A, B) of 2×2 symmetric matrices, where B is positive definite, they are jointly transformed in four steps, i.e., twice by a diagonal scaling followed by a Jacobi rotation, after which both matrices become diagonal. The above formulas for $\widehat{Z}'_k$ follow by combining the last three steps into a convenient computation without the intermediate matrices.

2.2 Parallelization of the one-sided algorithm

The sequential one-sided algorithm in each step chooses a single pivot index pair, according to some criterion that

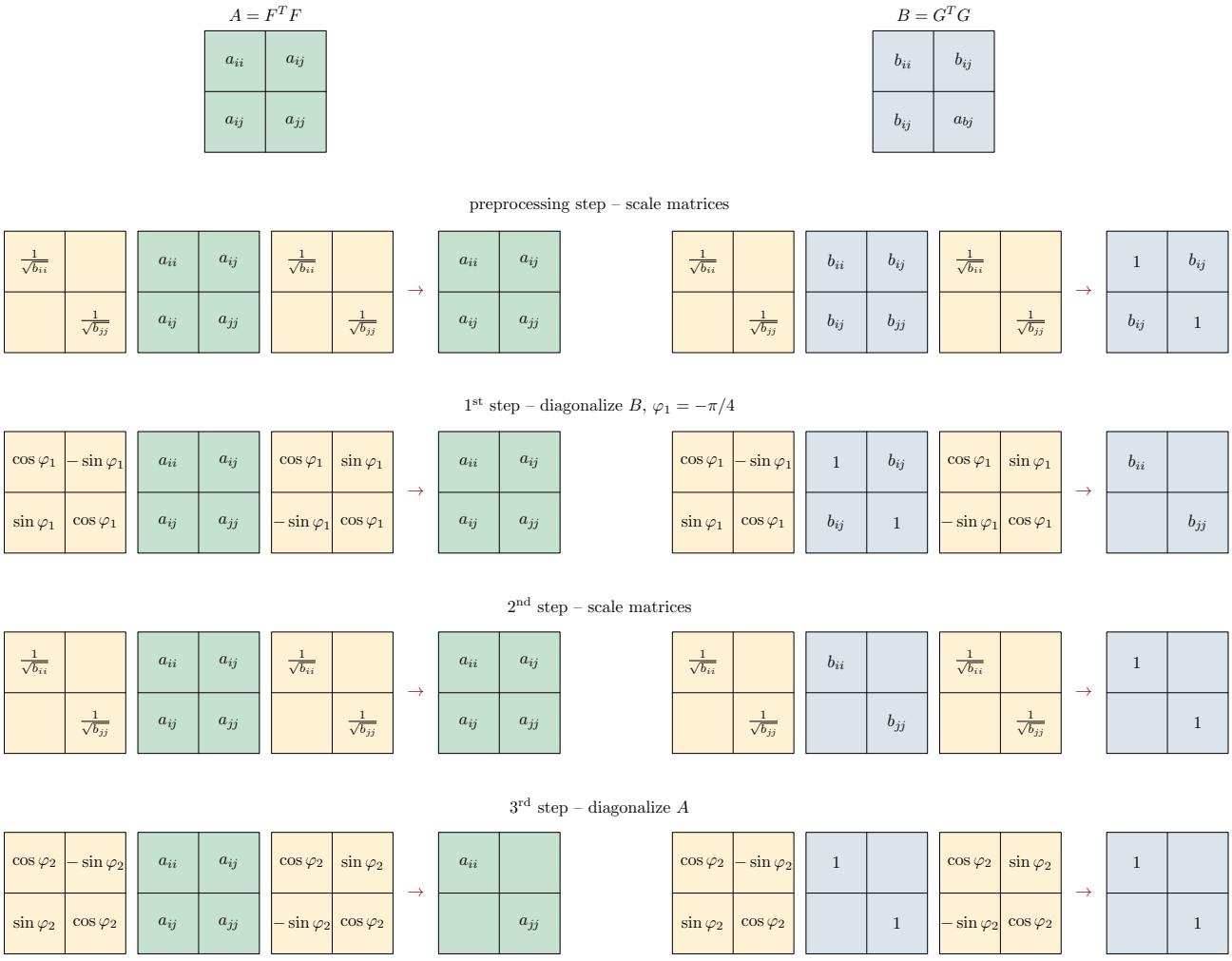

Figure 1. A decomposition of the two-sided Hari–Zimmermann joint diagonalization of a pair (A, B) of 2×2 symmetric matrices into four simple transformations. The second and the last are the orthogonal Jacobi rotations, but the first and the third (the diagonal scalings) are not orthogonal in general and also demonstrate why the positive definiteness of B is essential for the method.

is called a sequential Jacobi strategy. However, at most $\lfloor n/2 \rfloor$ pivot column pairs of each matrix can be transformed concurrently if the indices in all index pairs are distinct.

In a parallel step $k \geq 0$ a sequence $(i_k^{(\ell)}, j_k^{(\ell)})$ of pivot index pairs, where $1 \leq \ell \leq \lfloor n/2 \rfloor$, such that each index in the range from 1 to n appears at most (and for even n, exactly) once in the sequence, addresses $\lfloor n/2 \rfloor$ pivot column pairs of A_k and B_k to be transformed—each pair by a separate, concurrent task. All permutations of a given $(i_k^{(\ell)}, j_k^{(\ell)})$ are equivalent from the numerical point of view, since the resulting A_{k+1} and B_{k+1} are the same for every reordering of the sequence, and therefore any reordering represents the entire equivalence class.

For simplicity, a barrier is assumed between the successive parallel steps, i.e., all tasks of a step have to be completed before those of the following step are started.

A criterion to choose a pivot index pair sequence for each parallel step is called a parallel Jacobi strategy. Among the strategies that are simplest to compute are the ones that prescribe a pivot sequence for each step, until all $n(n-1)/2$ index pairs (i, j) are selected at least once. The choice of the steps is then periodically repeated. Let s be the shortest such period. The first s steps constitute the first sweep, the following s steps the second sweep, and so on.

If in any sweep exactly $n(n-1)/2$ different index pairs are chosen, such a strategy is called cyclic; otherwise, some index pairs are repeated in a sweep, and the strategy is called quasi-cyclic. For even n, $s \geq n-1$, and the equality holds if and only if the strategy is cyclic.

A strategy is defined for a fixed n; however, by a slight abuse of the usual terminology, a single principle by which the particular strategies are generated for some given matrix

orders will simply be called a strategy kind, or even a strategy for short.

Based on the previous experience with the one-sided Jacobi-like algorithms, two parallel Jacobi strategy kinds have been selected for testing: the modified modulus (MM; see, e.g., Novaković and Singer (2011); Novaković et al. (2015)), quasi-cyclic with $s = n$, and the generalized Mantharam–Eberlein (ME; see Mantharam and Eberlein (1993); Novaković (2015)) cyclic one. Please see Figures 1 and 2 in the supplementary material, where a sweep of ME and of MM, respectively, is shown two-sidedly on a matrix of order 32.

2.3 Blocking of the one-sided algorithm

Parallelization alone is not sufficient for achieving a decent performance of the algorithm on the modern architectures with multiple levels of the memory hierarchy.

The pointwise algorithm just described is therefore modified to work on the block columns of the matrices, instead of the columns proper. Each block column comprises an arbitrary but fixed number $\mathtt{w}$, $1 < \mathtt{w} < \lfloor n/2 \rfloor$, of consecutive matrix columns. Instead of 2×2 pivot submatrices of A_k and B_k, in the blocked algorithm $2\mathtt{w} \times 2\mathtt{w}$ pivot submatrices $\widehat{\mathsf{A}}_k^{(\ell)}$ and $\widehat{\mathsf{B}}_k^{(\ell)}$ are formed in the kth (parallel or sequential) step by matrix multiplications,

$$\begin{aligned}
\widehat{\mathsf{A}}_k^{(\ell)} &:= \begin{bmatrix} A_{i_k^{(\ell)} i_k^{(\ell)};k} & A_{i_k^{(\ell)} j_k^{(\ell)};k} \\ A^*_{i_k^{(\ell)} j_k^{(\ell)};k} & A_{j_k^{(\ell)} j_k^{(\ell)};k} \end{bmatrix} \\
&= \begin{bmatrix} F^*_{i_k^{(\ell)};k} F_{i_k^{(\ell)};k} & F^*_{i_k^{(\ell)};k} F_{j_k^{(\ell)};k} \\ F^*_{j_k^{(\ell)};k} F_{i_k^{(\ell)};k} & F^*_{j_k^{(\ell)};k} F_{j_k^{(\ell)};k} \end{bmatrix}, \\
\widehat{\mathsf{B}}_k^{(\ell)} &:= \begin{bmatrix} B_{i_k^{(\ell)} i_k^{(\ell)};k} & B_{i_k^{(\ell)} j_k^{(\ell)};k} \\ B^*_{i_k^{(\ell)} j_k^{(\ell)};k} & B_{j_k^{(\ell)} j_k^{(\ell)};k} \end{bmatrix} \\
&= \begin{bmatrix} G^*_{i_k^{(\ell)};k} G_{i_k^{(\ell)};k} & G^*_{i_k^{(\ell)};k} G_{j_k^{(\ell)};k} \\ G^*_{j_k^{(\ell)};k} G_{i_k^{(\ell)};k} & G^*_{j_k^{(\ell)};k} G_{j_k^{(\ell)};k} \end{bmatrix},
\end{aligned}$$

where $F_{i_k^{(\ell)};k}$, $F_{j_k^{(\ell)};k}$, $G_{i_k^{(\ell)};k}$, $G_{j_k^{(\ell)};k}$, $Z_{i_k^{(\ell)};k}$, and $Z_{j_k^{(\ell)};k}$ are the $i_k^{(\ell)}$th and $j_k^{(\ell)}$th block columns of F_k, G_k, and Z_k of width $\mathtt{w}$.

Now, $\widehat{\mathsf{A}}_k^{(\ell)}$ and $\widehat{\mathsf{B}}_k^{(\ell)}$ can either be jointly diagonalized by a matrix $\widehat{\mathsf{Z}}_k^{(\ell)}$, which leads to the full block (FB) algorithm Hari et al. (2014), as called in the context of the Jacobi methods, or their off-diagonal norms can be reduced by a sequence of congruences accumulated into $\widehat{\mathsf{Z}}_k^{(\ell)}$, which is called the block-oriented (BO) algorithm Hari et al. (2010). The idea behind blocking is that $\widehat{\mathsf{A}}_k^{(\ell)}$, $\widehat{\mathsf{B}}_k^{(\ell)}$, and $\widehat{\mathsf{Z}}_k^{(\ell)}$ fit, by choosing $\mathtt{w}$, into the small but fast cache memory (e.g., the shared memory of a GPU), to speed up the computation with them, as well as employing BLAS 3 (matrix multiplies) operations for the block column updates by $\widehat{\mathsf{Z}}_k^{(\ell)}$ afterwards:

$$\begin{aligned}
\begin{bmatrix} F_{i_k^{(\ell)};k+1} & F_{j_k^{(\ell)};k+1} \end{bmatrix} &= \begin{bmatrix} F_{i_k^{(\ell)};k} & F_{j_k^{(\ell)};k} \end{bmatrix} \cdot \widehat{\mathsf{Z}}_k^{(\ell)}, \\
\begin{bmatrix} G_{i_k^{(\ell)};k+1} & G_{j_k^{(\ell)};k+1} \end{bmatrix} &= \begin{bmatrix} G_{i_k^{(\ell)};k} & G_{j_k^{(\ell)};k} \end{bmatrix} \cdot \widehat{\mathsf{Z}}_k^{(\ell)}, \\
\begin{bmatrix} Z_{i_k^{(\ell)};k+1} & Z_{j_k^{(\ell)};k+1} \end{bmatrix} &= \begin{bmatrix} Z_{i_k^{(\ell)};k} & Z_{j_k^{(\ell)};k} \end{bmatrix} \cdot \widehat{\mathsf{Z}}_k^{(\ell)}.
\end{aligned}$$

The computation of $\widehat{\mathsf{Z}}_k^{(\ell)}$ in either FB or BO can be done by any convergent method; a two-sided method can be applied straightforwardly, but for the one-sided approach $\widehat{\mathsf{A}}_k^{(\ell)}$ and $\widehat{\mathsf{B}}_k^{(\ell)}$ have to be factorized first by, e.g., the Cholesky factorization

$$\widehat{\mathsf{A}}_k^{(\ell)} = \widehat{\mathsf{F}}_k^{(\ell)*} \widehat{\mathsf{F}}_k^{(\ell)}, \quad \widehat{\mathsf{B}}_k^{(\ell)} = \widehat{\mathsf{G}}_k^{(\ell)*} \widehat{\mathsf{G}}_k^{(\ell)},$$

and then the same implicit Hari–Zimmermann method, pointwise or blocked, and in both cases, either parallel or sequential, can be recursively applied to $\widehat{\mathsf{F}}_k^{(\ell)}$ and $\widehat{\mathsf{G}}_k^{(\ell)}$.

In the single-GPU algorithm, there is only one level of such a recursion, i.e., one level of blocking. The block, outer level of the algorithm and the pointwise, inner level do not need to employ the same strategy kind. Both levels, however, are parallel. The sweeps of the outer level are called the block (or outer) sweeps, and those of the inner level are called the pointwise (or inner) sweeps, which for FB are limited to 30 ($\widehat{\mathsf{A}}_k^{(\ell)}$ and $\widehat{\mathsf{B}}_k^{(\ell)}$ are usually fully diagonalized in less than that number of sweeps), and for BO are limited to only one inner sweep. Apart from that, there is no other substantial difference between FB and BO.

The Cholesky factorization is not the only way to form $\widehat{\mathsf{F}}_k^{(\ell)}$ and $\widehat{\mathsf{G}}_k^{(\ell)}$. One numerical stability improvement would be to use a diagonally pivoted version of the factorization instead Singer et al. (2012),

$$\widehat{\mathsf{A}}_k^{(\ell)} = P_{F;k}^{(\ell)} \widetilde{\mathsf{F}}_k^{(\ell)*} \widetilde{\mathsf{F}}_k^{(\ell)} P_{F;k}^{(\ell)T}, \quad \widehat{\mathsf{B}}_k^{(\ell)} = P_{G;k}^{(\ell)} \widetilde{\mathsf{G}}_k^{(\ell)*} \widetilde{\mathsf{G}}_k^{(\ell)} P_{G;k}^{(\ell)T}.$$

Another one would be to skip forming $\widehat{\mathsf{A}}_k^{(\ell)}$ and $\widehat{\mathsf{B}}_k^{(\ell)}$ explicitly by shortening the pivot block columns by the column-pivoted QR factorization directly Singer et al. (2020),

$$\begin{aligned}
\widetilde{\mathsf{F}}_k^{(\ell)} &:= \begin{bmatrix} \widetilde{F}_{i_k^{(\ell)};k} & \widetilde{F}_{j_k^{(\ell)};k} \end{bmatrix} \\
&= Q_{F;k}^{(\ell)*} \cdot \begin{bmatrix} F_{i_k^{(\ell)};k} & F_{j_k^{(\ell)};k} \end{bmatrix} \cdot P_{F;k}^{(\ell)}, \\
\widetilde{\mathsf{G}}_k^{(\ell)} &:= \begin{bmatrix} \widetilde{G}_{i_k^{(\ell)};k+1} & \widetilde{G}_{j_k^{(\ell)};k} \end{bmatrix} \\
&= Q_{G;k}^{(\ell)*} \cdot \begin{bmatrix} G_{i_k^{(\ell)};k} & G_{j_k^{(\ell)};k} \end{bmatrix} \cdot P_{G;k}^{(\ell)}.
\end{aligned}$$

In both cases, let

$$\widehat{\mathsf{F}}_k^{(\ell)} := \widetilde{\mathsf{F}}_k^{(\ell)} P_{F;k}^{(\ell)T}, \quad \widehat{\mathsf{G}}_k^{(\ell)} := \widetilde{\mathsf{G}}_k^{(\ell)} P_{G;k}^{(\ell)T},$$

where $P_{F;k}^{(\ell)}$ and $P_{G;k}^{(\ell)}$ are permutation matrices, while $Q_{F;k}^{(\ell)}$ and $Q_{G;k}^{(\ell)}$ are unitary and are not required to be stored, implicitly or explicitly, for any further computation.

However, the QR factorization (even without the column pivoting) of a pair of the tall and skinny block columns comes with a significant performance penalty on a GPU compared to the Cholesky factorization of a small, square pivot submatrix Novaković (2015), and the pivoted Cholesky factorization does not avoid a possibility of getting a severely ill-conditioned $\widehat{\mathsf{A}}_k^{(\ell)}$ or $\widehat{\mathsf{B}}_k^{(\ell)}$ by multiplying an ill-conditioned pair of block columns by itself. Both of these enhancements are therefore only mentioned here, with a performance comparison of the in-kernel QR factorizations versus the formation of the Grammian matrices and their Cholesky factorizations available in Appendix E. If the batched tall-and-skinny QR factorizations prove indispensable for a particularly ill-conditioned problem, `cublasXgetrfBatched` routine (with $\mathtt{X} \in \{\mathtt{D}, \mathtt{Z}\}$) and Boukaram et al. (2018) could also be considered.

In the following, the blocked algorithm is assumed to form the pivot submatrices as

$$\widehat{\mathsf{A}}_k^{(\ell)} := \begin{bmatrix} F_{i_k^{(\ell)};k} & F_{j_k^{(\ell)};k} \end{bmatrix}^* \cdot \begin{bmatrix} F_{i_k^{(\ell)};k} & F_{j_k^{(\ell)};k} \end{bmatrix},$$
$$\widehat{\mathsf{B}}_k^{(\ell)} := \begin{bmatrix} G_{i_k^{(\ell)};k} & G_{j_k^{(\ell)};k} \end{bmatrix}^* \cdot \begin{bmatrix} G_{i_k^{(\ell)};k} & G_{j_k^{(\ell)};k} \end{bmatrix},$$

i.e., each one by a `ZHERK` (`DSYRK` in the real case) like operation in the BLAS terminology, and the non-pivoted Cholesky factorization is then used to obtain $\widehat{\mathsf{F}}_k^{(\ell)}$ and $\widehat{\mathsf{G}}_k^{(\ell)}$, as demonstrated in Figure 2, where eight block columns of F are depicted. The same illustration holds if F is replaced by G. The block columns of the same hue are paired together, according to the first step of the ME strategy, giving four square blocks to be formed and factorized.

Figure 3 shows how each pair of the factors $\widehat{\mathsf{F}}_k$ and $\widehat{\mathsf{G}}_k$ is processed by the pointwise Hari–Zimmermann GSVD, leaving two matrices of the scaled left generalized singular vectors that are not used further, and a single matrix (rescaled, as noted in the following subsection 2.4) $\widetilde{\mathsf{Z}}_k$ of the accumulated transformations. The block column pairs of F, G, and Z, with the physically disjoint but logically contiguous block columns, are then postmultiplied, each from the right by the corresponding $\widetilde{\mathsf{Z}}_k$, and replaced by the result.

2.4 Rescalings

Observe that $\widehat{\mathsf{Z}}_k^{(\ell)}$ is a product of several non-unitary matrices, elements of which can be larger than 1 by magnitude, so the norm of $\widehat{\mathsf{Z}}_k^{(\ell)}$ can build up significantly by such accumulation of the transformations. Also, if $\widehat{\mathsf{Z}}_k^{(\ell)}$ diagonalizes $\widehat{\mathsf{A}}_k^{(\ell)}$ and $\widehat{\mathsf{B}}_k^{(\ell)}$, or reduces their off-diagonal norms, so does any matrix $\widehat{\mathsf{Z}}_k^{(\ell)}\widehat{\Theta}_k^{(\ell)}$, where $\widehat{\Theta}_k^{(\ell)}$ is a real, diagonal matrix with its diagonal elements positive and smaller than 1.

Let $\widetilde{\Theta}_k^{(\ell)}$ be such a matrix that reduces the norm of $\widehat{\mathsf{Z}}_k^{(\ell)}$,

$$\left(\widetilde{\Theta}_k^{(\ell)}\right)_{jj} := \left(\left\| \left(\widehat{\mathsf{F}}_k^{(\ell)}\right)_j' \right\|_F^2 + \left\| \left(\widehat{\mathsf{G}}_k^{(\ell)}\right)_j' \right\|_F^2 \right)^{-1/2},$$

where $\left(\widehat{\mathsf{F}}_k^{(\ell)}\right)_j'$ and $\left(\widehat{\mathsf{G}}_k^{(\ell)}\right)_j'$ stand for the jth column of the final, transformed $\widehat{\mathsf{F}}_k^{(\ell)}$ and $\widehat{\mathsf{G}}_k^{(\ell)}$, respectively, of which the latter has unit norm, and thus $\max_j \left(\widetilde{\Theta}_k^{(\ell)}\right)_{jj} < 1$.

This is exactly the same scaling as it would be performed in the last, post-iterative phase of the algorithm,

$$(\widetilde{\Sigma}_F)_{jj} := \|(F_N)_j\|_F, \quad (\widetilde{\Sigma}_G)_{jj} := \|(G_N)_j\|_F,$$

$$\widetilde{\Theta}_{jj} := ((\widetilde{\Sigma}_F)_{jj}^2 + (\widetilde{\Sigma}_G)_{jj}^2)^{-1/2},$$

$$\Sigma_F := \widetilde{\Sigma}_F\widetilde{\Theta}, \qquad \Sigma_G := \widetilde{\Sigma}_G\widetilde{\Theta}, \qquad \Sigma := \Sigma_G^{-1}\Sigma_F,$$
$$U := F_N\widetilde{\Sigma}_F^{-1}, \qquad V := G_N\widetilde{\Sigma}_G^{-1}, \quad Z := Z_N\widetilde{\Theta},$$

except that $\widehat{\mathsf{F}}_k^{(\ell)\prime}$ and $\widehat{\mathsf{G}}_k^{(\ell)\prime}$ do not have to be rescaled and the norms of their columns do not have to be kept as they are all discarded immediately after $\widetilde{\Theta}_k^{(\ell)}$ has been computed.

Then, $\widetilde{\mathsf{Z}}_k^{(\ell)} := \widehat{\mathsf{Z}}_k^{(\ell)}\widetilde{\Theta}_k^{(\ell)}$, is applied to the pivot block column pair of F_k, G_k, and Z_k instead of $\widehat{\mathsf{Z}}_k^{(\ell)}$, and is considered embedded into $\widetilde{Z}_k$ in a similar way as $\widehat{Z}_k^{(\ell)}$ would be in the pointwise case, i.e., starting from $\widetilde{Z}_k$ being I_n, for each ℓ let

$$\begin{aligned}
&\widetilde{Z}_k((i_k^{(\ell)}-1)\cdot\mathtt{w}+1 : i_k^{(\ell)}\cdot\mathtt{w},\ (i_k^{(\ell)}-1)\cdot\mathtt{w}+1 : i_k^{(\ell)}\cdot\mathtt{w}) \\
&\quad := \widetilde{\mathsf{Z}}_k^{(\ell)}(1:\mathtt{w},\ 1:\mathtt{w}) \\
&\widetilde{Z}_k((i_k^{(\ell)}-1)\cdot\mathtt{w}+1 : i_k^{(\ell)}\cdot\mathtt{w},\ (j_k^{(\ell)}-1)\cdot\mathtt{w}+1 : j_k^{(\ell)}\cdot\mathtt{w}) \\
&\quad := \widetilde{\mathsf{Z}}_k^{(\ell)}(1:\mathtt{w},\ \mathtt{w}+1:2\cdot\mathtt{w}) \\
&\widetilde{Z}_k((j_k^{(\ell)}-1)\cdot\mathtt{w}+1 : j_k^{(\ell)}\cdot\mathtt{w},\ (i_k^{(\ell)}-1)\cdot\mathtt{w}+1 : i_k^{(\ell)}\cdot\mathtt{w}) \\
&\quad := \widetilde{\mathsf{Z}}_k^{(\ell)}(\mathtt{w}+1:2\cdot\mathtt{w},\ 1:\mathtt{w}) \\
&\widetilde{Z}_k((j_k^{(\ell)}-1)\cdot\mathtt{w}+1 : j_k^{(\ell)}\cdot\mathtt{w},\ (j_k^{(\ell)}-1)\cdot\mathtt{w}+1 : j_k^{(\ell)}\cdot\mathtt{w}) \\
&\quad := \widetilde{\mathsf{Z}}_k^{(\ell)}(\mathtt{w}+1:2\cdot\mathtt{w},\ \mathtt{w}+1:2\cdot\mathtt{w}),
\end{aligned}$$

where the subscripting is to be interpreted as in Fortran.

To reduce the norm of the entire Z_k, a similar rescaling can be applied on Z_k, using the column norms of F_k and G_k, after each but the last block sweep. After the last block sweep, a rescaling of all three matrices (F_N, G_N, and Z_N) is performed to obtain U, V, and Z, with the extraction of Σ_F, Σ_G, and Σ.

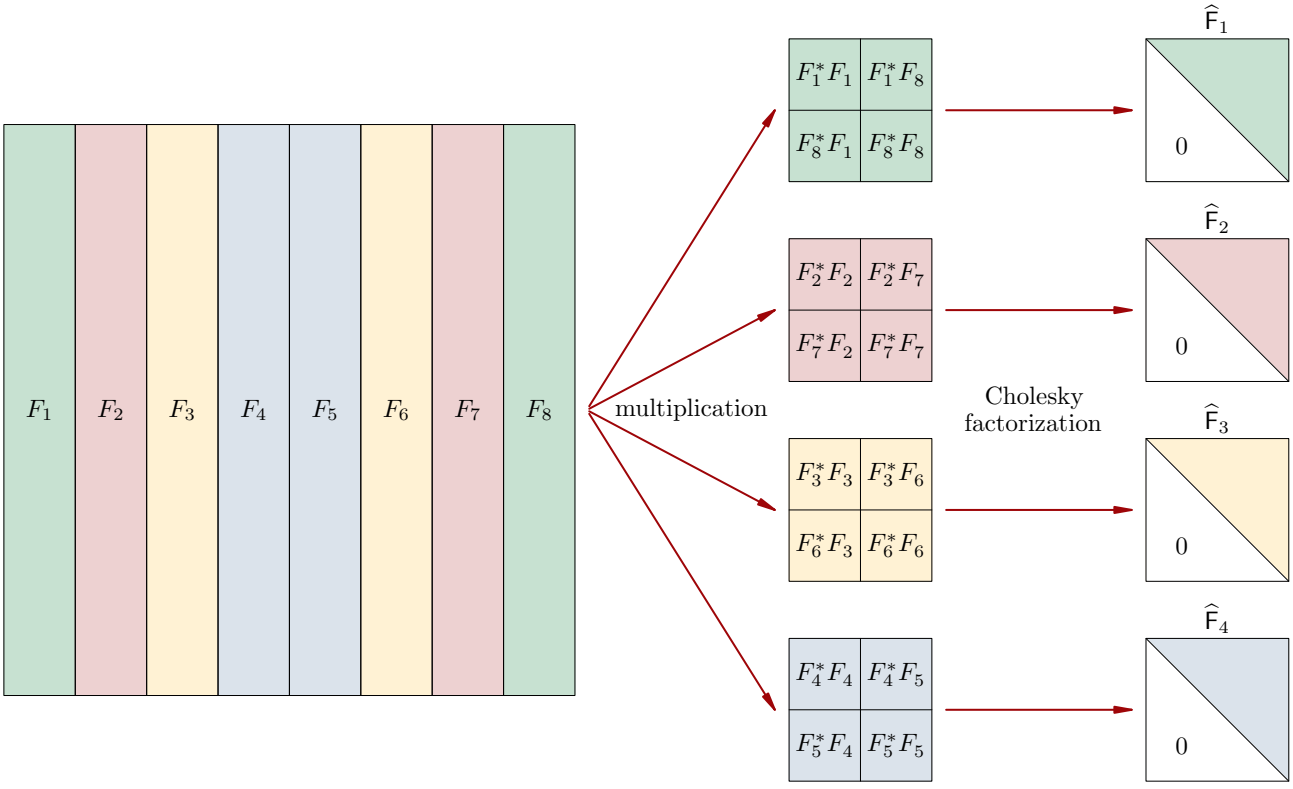

Figure 2. Formation of the Grammian matrices from the pairs of block columns of F and their subsequent Cholesky factorizations. Each pair is indicated by a different hue, and varies with a block step. The same process is repeated for G, to obtain the factors $\widehat{\mathsf{G}}_k$.

2.5 Convergence

The inner level of the algorithm stops when there were no transformations, apart from the sorting permutations, applied in a sweep, or when the prescribed maximal number of sweeps has been reached. Then, the pivot block column pairs of F_k, G_k, and Z_k are updated concurrently for all ℓ by $\widetilde{\mathsf{Z}}_k^{(\ell)}$, which can be skipped for those ℓ where $\widetilde{\mathsf{Z}}_k^{(\ell)} = I_{2\mathtt{w}}$.

The same criterion could be used for the outer level, where the count of transformations applied in an outer sweep is a sum of all transformations applied in the inner level in all steps of the outer sweep. However, this criterion has to be relaxed Novaković (2015); Novaković et al. (2015), since the rounding errors in forming and factorizing the block pivot submatrices could spoil the attained numerical orthogonality of the original columns, and introduce a small number of unwarranted transformations that prevent the algorithm from detecting convergence even if it has in fact been reached.

Therefore, the transformations are divided in two classes: "big" and "small". The latter are all $\widehat{Z}_k^{(\ell)}$ where either:

C1. $(\cos\varphi_k)/t_k = (\cos\psi_k)/t_k = 1$, or
C2. $\cos\varphi_k = \cos\psi_k = 1$,

i.e., where $\widehat{Z}_k^{(\ell)}$ is close to (a multiple of) identity, and the former are all other transformations. Note that neither definition of the small transformations implies that $\sin\varphi_k$ or $\sin\psi_k$ are *numerically* equal to zero (and are usually not). Also, since x_k tends to zero and therefore t_k to one in the last sweeps of the algorithm, the first and the second definition should not differ significantly.

There are separate counters of the big transformations, and of all transformations applied in the inner level of the algorithm. The inner level still halts when there were no transformations of any class in a sweep, but the outer level stops when there were no big transformations applied in an outer sweeps (i.e., small transformations are allowed to occur but do not spoil the overall convergence). Such a heuristic criterion prevents in practice a long sequence of outer sweeps at the end of the algorithm, with only a few transformations close to identity in each.

2.6 Variants of the algorithm

To summarize the variants of the algorithm, see Table 1. The first column, ID, sets a shorthand for the corresponding variant. The second column specifies a convergence criterion used. The third column distinguished between assuming the column norms of the second matrix to

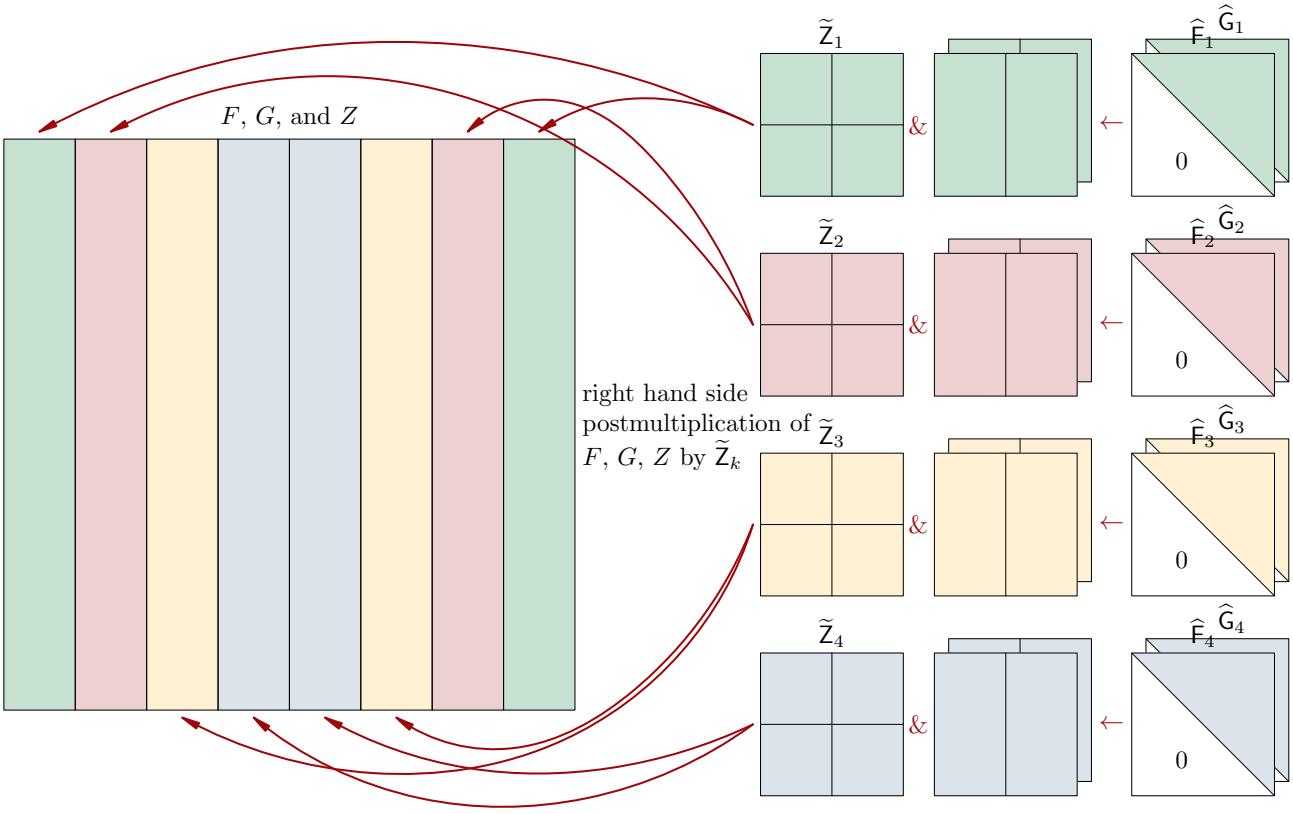

Figure 3. The pointwise Hari–Zimmermann GSVD of four $(\widehat{\mathsf{F}}_k, \widehat{\mathsf{G}}_k)$ pairs results in two unused scaled left generalized singular vector matrices per pair, and a single accumulated and rescaled transformation matrix $\widetilde{\mathsf{Z}}_k$. Each of the four original block column pairs of F, G, and Z is then updated by multiplying it from the right by the corresponding $\widetilde{\mathsf{Z}}_k$ (indicated by a pair of arrows).

Table 1. Variants of the implicit Hari–Zimmermann algorithm.

ID	convergence	transformations	dot-products
0	criterion C1	$\widehat{Z}_k$ (F_k, G_k prescaled)	ordinary
1	criterion C1	$\widehat{Z}_k$ (F_k, G_k prescaled)	enhanced
2	criterion C1	$\widehat{Z}'_k$ (F_k, G_k not scaled)	ordinary
3	criterion C1	$\widehat{Z}'_k$ (F_k, G_k not scaled)	enhanced
4	criterion C2	$\widehat{Z}_k$ (F_k, G_k prescaled)	ordinary
5	criterion C2	$\widehat{Z}_k$ (F_k, G_k prescaled)	enhanced
6	criterion C2	$\widehat{Z}'_k$ (F_k, G_k not scaled)	ordinary
7	criterion C2	$\widehat{Z}'_k$ (F_k, G_k not scaled)	enhanced

be unity, and rescaling of both matrices with each transformation. The fourth column relates to computing the dot-products the usual way, or by an enhanced, possibly more accurate procedure from Appendix A. Unless specified otherwise, the column sorting is employed in all cases. Thus, e.g., DHZ3-(MM-FB-ME) refers to the double-precision real implicit blocked Hari–Zimmermann algorithm with ID equal to 3, using MM at the outer and ME at the inner level of blocking of type FB. Similarly, ZHZ0-(ME-BO-ME) stands for the double-precision complex implicit blocked Hari–Zimmerman algorithm with ID equal to 0 (the "standard" variant), using ME at both levels of blocking of type BO.

From now on, when a numeric variant ID is mentioned in the text, it is assumed that it should be looked up in Table 1.

3 The single-GPU implementation

In this section the single-GPU implementation of the complex and the real one-sided Hari–Zimmermann algorithms are described. The focus is on the complex algorithm, and the real one is commented on when substantially different. The target framework is CUDA C++ NVIDIA Corp. (2019) for the NVIDIA GPUs (Kepler series and newer), but also another general-purpose GPU programming environment with the analogous concepts and constructs could probably be used.

3.1 Data layout and transfer

Due to blocking employed by the algorithm, each matrix is viewed as column-striped, with the block columns

containing $\mathtt{w} = 16$ consecutive columns each. To simplify the implementation, assume that n is a multiple of 32, and let $\mathtt{n} := n/\mathtt{w}$ (so $\mathtt{n}$ is even). If the assumption does not hold for the input, the matrices should then be bordered by appending $32 - (n \bmod 32)$ columns to the right, and as many rows to the bottom. The elements $(m_Y + 1, n + 1)$, $(m_Y + 2, n + 2)$, ..., in the columns newly added to the matrix $Y \in \{F, G\}$ should be set to unity, to avoid introducing zero columns, since it is essential for Y to be of full column rank. Other bordering elements should be set to zero, to prevent any transformation not implied by the original matrices from happening (see a bordering example in Novaković and Singer (2011)).

Another assumption, to simplify the loop unrolling in various parts of the code, is to have m_F and m_G as a multiple of 64. If it is not the case with the input, then, after a possible bordering as described above, $64 - (m_Y \bmod 64)$ rows of zeros should be appended to the bottom of the matrix $Y \in \{F, G\}$.

3.1.1 The CPU and the GPU RAM layout and transfer Data is laid out in the GPU RAM (also called "global memory" in the GPU context) in the following sequence:

$$\mathrm{Re}(F), \mathrm{Im}(F); \quad \mathrm{Re}(G), \mathrm{Im}(G); \quad \mathrm{Re}(Z), \mathrm{Im}(Z),$$

after which follow the output-only vectors Σ, Σ_F, Σ_G (with double-precision elements, each of length n), and C (holding unsigned 8-byte integers, of length $\mathtt{n}$). The rest of data is used both for input and output, i.e., the six double-precision matrices are constantly being read and overwritten within the GPU as the algorithm progresses. The matrices are loaded to the GPU at the beginning of the algorithm's execution, if they are not already in place as a result of another computation, and optionally copied to the CPU at its end, as well as Σ, Σ_F, and Σ_G.

In the pre- and post-processing stages on the CPU, input (F, G) and output data (U in place of F; V in place of G; and Z), respectively, is repacked from (or to) the standard representation of complex matrices, in which the successive elements are complex numbers $z = (\mathrm{Re}(z), \mathrm{Im}(z))$. Each double-precision matrix can therefore be loaded to, or copied from, the GPU with a single CUDA call.

This decision to keep all data in real-typed variables by splitting the real and the imaginary matrix parts and to perform the complex arithmetic manually is a design choice, not a necessity, since an implementation of the algorithm with the real and the imaginary parts interleaved in the customary way is also possible. There is no direct support for the standard C (with `_Complex` types) or C++ (with `std::complex` types) complex arithmetic in CUDA, so some non-standard approach has to be used anyway; e.g., the datatypes and the routines from the `cuComplex.h` header file, or those from the `thrust` library, or a custom implementation—possibly with a different memory layout—of complex numbers and the operations with them. The chosen, custom approach with the split data layout makes reading or writing only one (real or imaginary) component of the successive matrix elements straightforward, and such memory accesses can be contiguous.

In the auxiliary vector C there are two counters, $C_\ell^{(0)}$ and $C_\ell^{(1)}$, where ℓ is the index of a thread block in a grid of the main computational kernel. In $C_\ell^{(0)}$ the count of the "big" transformations, and in $C_\ell^{(1)}$ the count of all transformations applied in all kernel launches within a single block sweep are accumulated. At the beginning of each block sweep C is zeroed out on the GPU, and is copied to a CPU vector $\widetilde{C}$ at the end of the sweep.

3.1.2 The shared memory layout For each thread block, the non-constant, non-register data (comprising three complex matrices: F, G, and Z) for the main computational kernel is laid out in the shared memory as:

$$\mathrm{Re}(\mathsf{F}), \mathrm{Re}(\mathsf{G}), \mathrm{Re}(\mathsf{Z}); \quad \mathrm{Im}(\mathsf{F}), \mathrm{Im}(\mathsf{G}), \mathrm{Im}(\mathsf{Z}).$$

Each double-precision matrix is square, of order 32, with the elements stored in Fortran array order, for a total shared memory requirement of $(3 \times 2) \times (32 \times 32) \times 8\,\mathrm{B} = 48\,\mathrm{kiB}$ for a thread block. The shared memory is configured with 8-byte-wide banks. No other kernel requires any shared memory.

Let $\mathrm{Re}(\mathsf{F}_{64\times32})$ stand for the contiguous memory space occupied by $\mathrm{Re}(\mathsf{F})$ and $\mathrm{Re}(\mathsf{G})$; $\mathrm{Im}(\mathsf{F}_{64\times32})$ for $\mathrm{Im}(\mathsf{F})$ and $\mathrm{Im}(\mathsf{G})$; $\mathrm{Re}(\mathsf{G}_{64\times32})$ for $\mathrm{Re}(\mathsf{G})$ and $\mathrm{Re}(\mathsf{Z})$; and $\mathrm{Im}(\mathsf{G}_{64\times32})$ for $\mathrm{Im}(\mathsf{G})$ and $\mathrm{Im}(\mathsf{Z})$, as the real and the imaginary parts of $\mathsf{F}_{64\times32}$ and $\mathsf{G}_{64\times32}$ matrices that share the same storage with F, G, and Z. Such overlapping of data is necessary for the formation of F and G from the block columns of F and G, respectively, as described below. Also, let $\mathrm{Re}(\mathsf{F}_{96\times32})$ stand for $\mathrm{Re}(\mathsf{F})$, $\mathrm{Re}(\mathsf{G})$, $\mathrm{Re}(\mathsf{Z})$; and $\mathrm{Im}(\mathsf{F}_{96\times32})$ for $\mathrm{Im}(\mathsf{F})$, $\mathrm{Im}(\mathsf{G})$, $\mathrm{Im}(\mathsf{Z})$.

The real case In the real Hari–Zimmermann algorithm $\mathrm{Im}(\cdot)$ matrices do not exist, so repacking of the input and the output data does not happen. The other properties of two data layouts still hold. The shared memory requirements are half of those for the complex algorithm, i.e., $24\,\mathrm{kiB}$.

3.1.3 The constant memory layout The constant memory on the GPU holds the pointers to the matrices and the vectors described above, with their dimensions, to avoid sending them as parameters in each kernel call. The Jacobi strategy table for the first, pointwise level of the algorithm is also stored in the constant memory, since it does not depend on the actual input data.

The strategy table contains 31 (or 32) rows, same as the number of steps of a chosen (quasi-)cyclic parallel strategy. Each row is an array of 16 index pairs (p, q), with $\mathsf{p} < \mathsf{q}$,

where no two indices in a row are the same. A pair of such indices addresses a pair of columns of the matrices F and G to be transformed concurrently with all other column pairs in the step.

3.1.4 Constants in the global memory The Jacobi strategy table for the second, block level of the algorithm might not fit in the constant memory for the large n, so it has to be stored in the global memory in such a case. It is similarly formatted as the table for the pointwise level, but with $\mathtt{n}-1$ (or n) rows, each with $\mathtt{n}/2$ index pairs. Here, a pair $(\mathtt{p},\mathtt{q})$, with $\mathtt{p}<\mathtt{q}$, addresses a pair of block columns of the matrices F and G. No two indices in a row are the same, i.e., every integer between 0 and $\mathtt{n}-1$ appears exactly once in a row. Each row encodes a step of the chosen block level (quasi-)cyclic parallel strategy, which does not have to be of the same kind as the one chosen for the pointwise level.

Both tables are precomputed on and preloaded from the CPU Novaković (2017); Singer et al. (2020) before any computation starts on the GPU.

3.2 Arithmetic operations

Since the data is held in the real-valued arrays only, the complex arithmetic is performed manually, computing the real and the imaginary parts of the result separately, rather than assembling the complex operands in the CUDA format each time an operation has to be performed, and disassembling the result when it has to be stored back in memory.

3.2.1 Complex arithmetic The arithmetic operations on complex numbers needed by the algorithm are addition, subtraction, negation, complex conjugation, multiplication by a complex or a real number (or an inverse of the latter), and taking the absolute value. Only $|z|$, $a\cdot b$, and an FMA-like operation $a\cdot b+c$ (a complex multiplication and an addition fused) require special attention, while the rest are trivial to express by the real arithmetic directly in the code.

The absolute value is obtained as $|z|:=\mathtt{hypot}(\mathrm{Re}(z),\mathrm{Im}(z))$, without undue overflow. Still, it is possible that $|z|$ overflows when at least one component of z is close enough by magnitude to the largest representable finite double-precision number, but such a problem can be mitigated by a joint downscaling of two matrices under transformation. For example, a scaling by $1/2$ would suffice, and would also keep the significand intact for all normalized (i.e., finite non-subnormal) numbers. Such rescaling has not been implemented, though it would not be overwhelmingly hard to apply the rescaling and restart the computation if any thread detects that its $|z|$ operation has overflowed, and makes that known to other threads in a block by a subsequent __syncthreads_count CUDA primitive invoked with a Boolean value indicating the presence of an overflow.

For multiplication, an inlineable routine (zmul) computes $z:=a\cdot b$ and returns the result via two output-only arguments, referring to $\mathrm{Re}(z)$ and $\mathrm{Im}(z)$. With the CUDA FMA intrinsic __fma_rn it holds

$$\mathrm{Re}(z)=\mathtt{\_\_fma\_rn}(\mathrm{Re}(a),\mathrm{Re}(b),-\mathrm{Im}(a)\cdot\mathrm{Im}(b)),$$

computed in a way that requires three floating-point operations but two roundings only. Note that the operations are ordered arbitrarily, thus zmul could also be realized by multiplying the real parts of the factors first. $\mathrm{Im}(z)$ is obtained by

$$\mathrm{Im}(z)=\mathtt{\_\_fma\_rn}(\mathrm{Re}(a),\mathrm{Im}(b),\mathrm{Im}(a)\cdot\mathrm{Re}(b)),$$

where only two instead of three floating-point operations are required, with two roundings, and the choice of the real product arguments is arbitrary. In total, five operations (of which the negation is trivial) instead of six are needed.

The FMA-like operation is modeled after the CUDA one in the cuComplex.h header. Let $z:=a\cdot b+c$. Then, zfma routine requires 3 operations with 2 roundings for

$$\begin{aligned}d&:=\mathtt{\_\_fma\_rn}(-\mathrm{Im}(a),\mathrm{Im}(b),\mathrm{Re}(c)),\\ \mathrm{Re}(z)&=\mathtt{\_\_fma\_rn}(\mathrm{Re}(a),\mathrm{Re}(b),d),\end{aligned}$$

and 2 operations with 2 roundings for

$$\begin{aligned}d&:=\mathtt{\_\_fma\_rn}(\mathrm{Im}(a),\mathrm{Re}(b),\mathrm{Im}(c)),\\ \mathrm{Im}(z)&=\mathtt{\_\_fma\_rn}(\mathrm{Re}(a),\mathrm{Im}(b),d).\end{aligned}$$

It holds $\mathtt{zfma}(a,b,0)=\mathtt{zmul}(a,b)$ for all a and b.

3.2.2 Real arithmetic The real arithmetic uses operations with the accuracy guarantees mandated by the IEEE 754 standard for floating-point arithmetic in rounding to nearest (ties to even) mode, except in the optional enhanced dot-product computation, where rounding to $-\infty$ is also employed, as described in Appendix A.

A correctly rounded (i.e., with the relative error of no more than half ulp) double-precision $\mathtt{rsqrt}(x):=1/\sqrt{x}$ device function, provided by Norbert Juffa in private communication, that improves the accuracy of the CUDA math library routine of the same name (let it be referred to by rsqrt_rn when a need arises to disambiguate between the two, and by rsqrt when either is acceptable), is called wherever such an expression has to be computed.

3.2.3 Reproducibility In both the real and the complex code rsqrt_rn function is expected, but not extensively verified, to be correctly rounded and thus reproducible. Reproducibility of the results is guaranteed for the complex code as long as it is for the hypot function in all CUDA versions and on all GPUs under consideration. All other floating-point arithmetic operations with rounding (i.e., not including the comparisons and the negations) are expressed in the terms of the seven double precision CUDA intrinsics.

3.2.4 Integer arithmetic To keep the memory requirements low, the pointwise level indices in the strategy table are stored as unsigned 1-byte integers, while the block level indices occupy 2 bytes each (i.e., $\mathtt{n} \leq 65536$, what is enough to exceed the RAM sizes of the present-day GPUs).

For dimensioning and indexing purposes the unsigned 4-byte integers (after a possible promotion) are used, since their range allows for addressing up to 32 GiB of double-precision floating-point data, which is twice the quantity of GPU RAM available on the testing hardware. However, 8-byte integers should be used instead if the future GPUs provide more memory than this limit.

Although Fortran array order is assumed throughout the paper and the code, the indices on a GPU are zero-based. The CUDA thread (block) indices `blockIdx.x`, `threadIdx.x`, and `threadIdx.y` are shortened as $\mathtt{b_x}$, $\mathtt{t_x}$, and $\mathtt{t_y}$, respectively.

3.3 Initialization of Z with optional rescaling of F and G

Here, `initFGZ`, the first of three computational kernels, is described. Its purpose is to initialize the matrix Z, having been zeroed out after allocation, to Z_0, a diagonal matrix such that $(Z_0)_{jj} := 1/\|g_j\|_F$, and to rescale F and G to F_0 and G_0, by multiplying the elements of each column j of the matrices by $(Z_0)_{jj}$ in the variants 0, 1, 4, and 5. Else, in other variants, $Z_0 = I_n$.

The kernel is launched once, before the iterative phase of the algorithm, with a one-dimensional grid of $n/2$ thread blocks, each of which is also one-dimensional, with 64 threads (two warps of 32 consecutive-numbered threads).

A warp is in charge of one column of F, G, and Z, i.e., its threads access only the elements i of that column j, where

$$j := \mathtt{b_x} \cdot 2 + \lfloor \mathtt{t_x}/32 \rfloor, \quad i \bmod 32 = \mathtt{t_x} \bmod 32.$$

A warp reads 32 consecutive elements of $\mathrm{Re}(G)_j$ and $\mathrm{Im}(G)_j$ at a time. Each of its threads updates its register-stored partial sums

$$\hat{c}'_{\mathrm{r}}[\mathtt{t_x}] := \hat{c}_{\mathrm{r}}[\mathtt{t_x}] + \mathrm{Re}(G)^2_{ij}, \quad \hat{c}'_{\mathrm{i}}[\mathtt{t_x}] := \hat{c}_{\mathrm{i}}[\mathtt{t_x}] + \mathrm{Im}(G)^2_{ij},$$

using one FMA operation for each update, and this is repeated by going to rows $i := i + 32$ until $i \geq m_G$. Initially, $i = \mathtt{t_x} \bmod 32$ and $\hat{c}_{\mathrm{r}}[\mathtt{t_x}] = \hat{c}_{\mathrm{i}}[\mathtt{t_x}] = 0$. After passing through the entire column, those partial sums are added to obtain $\hat{s}[\mathtt{t_x}] := \hat{c}_{\mathrm{r}}[\mathtt{t_x}] + \hat{c}_{\mathrm{i}}[\mathtt{t_x}]$. Then, $\hat{s}[\mathtt{t_x}]$ are summed and the result is distributed across the warp by a warp-shuffling NVIDIA Corp. (2019) sum-reduction, described in Appendix C, yielding the sum of squares of the magnitudes of the elements in the column, i.e., $\|g_j\|^2_F$.

Such a computation occurs in the variants 0 and 4, while in the variants 1 and 5 the enhanced dot-product computation as in Appendix A updates the per-thread, register-stored partial sums $c_{\mathrm{r}}[\mathtt{t_x}]$, $c_{\mathrm{i}}[\mathtt{t_x}]$, $d_{\mathrm{r}}[\mathtt{t_x}]$, $d_{\mathrm{i}}[\mathtt{t_x}]$. After a pass over the column completes, $s[\mathtt{t_x}]$ are formed according to the rules of Appendix A and summed as above.

Either way, $z_j[\mathtt{t_x}] := 1/\sqrt{\|g_j\|^2_F}$ is then computed, and the jth columns of F and G are scaled by $z_j[\mathtt{t_x}]$ in a loop similar to the one described above, i.e., for i in steps of 32 while $i < m_F$,

$$\mathrm{Re}(F)'_{ij} := \mathrm{Re}(F)_{ij} \cdot z_j[\mathtt{t_x}],$$
$$\mathrm{Im}(F)'_{ij} := \mathrm{Im}(F)_{ij} \cdot z_j[\mathtt{t_x}],$$

and then the same scaling is performed on G, with $i < m_G$.

Finally, $z_j[\mathtt{t_x}]$ is written to $\mathrm{Re}(Z)_{lj}$ by the lowest-numbered thread in a warp, i.e., $\mathrm{t_x} \equiv 0 \pmod{32}$, where l is an index making a physical column j treated as a logical column l. In the single-GPU case, $l = j$. In the variants 2, 3, 6, and 7, $\mathrm{Re}(Z)_{lj}$ is set to 1 and no other processing occurs.

This and any other computation of the Frobenius norm of a vector via the sum of squares of its elements could overflow even if the result itself would not. See (Novaković 2015, Appendix A) for one of several possible remedies.

3.4 Rescaling of Z and extraction of U, Σ_F, V, Σ_G, and Σ

After each block sweep, another kernel, `rescale`, is called, with a Boolean flag `f` indicating whether it is the last sweep.

If `f` is `false`, only Z is rescaled according to the rules of subsection 2.4, and otherwise the full results of the GSVD computation (U, Σ_F, V, Σ_G, and Σ) are produced.

The kernel's grid is identical, and the operation very similar to `initFGZ`. First, $\|f_j\|^2_F$ is computed, and if non-unity and `f`, f_j is scaled by $1/\sqrt{\|f_j\|^2_F}$. If `f`, $\Sigma'_j[\mathtt{t_x}] := \sqrt{\|f_j\|^2_F}$. Then, $\|g_j\|^2_F$ is computed, and if non-unity and `f`, g_j is scaled by $1/\sqrt{\|g_j\|^2_F}$, as well as $\Sigma'_j[\mathtt{t_x}]$ to obtain $\Sigma_j[\mathtt{t_x}]$; else, if `f`, $\Sigma_j[\mathtt{t_x}] := \Sigma'_j[\mathtt{t_x}]$.

Then $\Sigma'_{F;j}[\mathtt{t_x}] := \sqrt{\|f_j\|^2_F}$, $\Sigma'_{G;j}[\mathtt{t_x}] := \sqrt{\|g_j\|^2_F}$, and $\theta[\mathtt{t_x}] := 1/\sqrt{\|f_j\|^2_F + \|g_j\|^2_F}$. If $\theta[\mathtt{t_x}] \neq 1$, z_j is scaled by $\theta[\mathtt{t_x}]$, as well as $\Sigma'_{F;j}[\mathtt{t_x}]$ and $\Sigma'_{G;j}[\mathtt{t_x}]$ to obtain $\Sigma_{F;j}[\mathtt{t_x}]$ and $\Sigma_{G;j}[\mathtt{t_x}]$; else, $\Sigma_{F;j}[\mathtt{t_x}] := \Sigma'_{F;j}[\mathtt{t_x}]$ and $\Sigma_{G;j}[\mathtt{t_x}] := \Sigma'_{G;j}[\mathtt{t_x}]$.

Finally, if `f`, Σ_j, $\Sigma_{F;j}$, and $\Sigma_{G;j}$ are written to the GPU RAM by a thread $\mathrm{t_x} \equiv 0 \pmod{32}$. All variables indexed by $\mathtt{t_x}$ above are per-thread and register-stored, unless a register spill occurs.

3.5 The main computational kernel

The main kernel comes in `bstep1s` and `bstep1n` versions, where the former is the default one, with the column sorting, while the latter is a non-sorting version.

The kernel is called once per a block step. Each such call constitutes the entire block step, and it cannot run concurrently with any other GPU part of the algorithm since it can update almost the whole allocated GPU memory.

The kernel's grid is one-dimensional, with $\mathtt{n}/2$ two-dimensional thread blocks, each of them having $32 \times \mathtt{w} = 512$ threads. A thread block $\ell := \mathtt{b_x}$ in the block step $\mathtt{k} := k \bmod \mathtt{n}'$ is in charge of one pivot block column pair, $(\mathtt{p}_\mathtt{k}^{(\ell)}, \mathtt{q}_\mathtt{k}^{(\ell)})$, of F, G, and Z, where $\mathtt{n}'$ is $\mathtt{n}-1$ for the ME or $\mathtt{n}$ for the MM strategy kind.

The computational subphases of `bstep1(s/n)`(k),

1. formation of $\widehat{\mathsf{A}}_k^{(\ell)}$ and $\widehat{\mathsf{B}}_k^{(\ell)}$ in the shared memory,
2. the Cholesky factorizations of $\widehat{\mathsf{A}}_k^{(\ell)}$ and $\widehat{\mathsf{B}}_k^{(\ell)}$ as $\widehat{\mathsf{F}}_k^{(\ell)*}\widehat{\mathsf{F}}_k^{(\ell)}$ and $\widehat{\mathsf{G}}_k^{(\ell)*}\widehat{\mathsf{G}}_k^{(\ell)}$, respectively,
3. the pointwise implicit Hari–Zimmermann algorithm on the matrix pair $(\widehat{\mathsf{F}}_k^{(\ell)}, \widehat{\mathsf{G}}_k^{(\ell)})$, yielding $\widetilde{\mathsf{Z}}_k^{(\ell)}$,
4. postmultiplication of the pair ℓ of pivot block columns of F, G, and Z by $\widetilde{\mathsf{Z}}_k^{(\ell)}$,

are all fused into a single kernel to effortlessly preserve the contents of the shared memory between them.

All the required matrix algebra routines have been written as device functions with the semantics similar to, but different from the standard BLAS, due to the data distribution and the memory constraints. For example, a single call of the BLAS-compatible `ZHERK` (or `DSYRK` in the real case) operation for the subphase **1** is not possible, since the two pivot block columns do not have to be adjacent in the global memory. The subphase **3** cannot use a single standard `ZGEMM` (or `DGEMM`) call for the same reason, but also because the block columns have to be overwritten in-place to avoid introducing any work arrays.

Since no two pivot block index pairs share an index, all thread blocks can be executed concurrently without any interdependencies or data races. Due to the shared memory requirement and a high thread count, it is not possible that more than two (or, in the real case, four) thread blocks could share a single GPU multiprocessor (an SM for short, which cannot have more than 2048 threads resident at present). On a Maxwell GPU, the profiler reports occupancy of 25% for the real and the complex `bstep1s`, i.e., at most one thread block is active on an SM at any time. That can be attributed to a huge register pressure, since 128 registers per thread are used for the main kernel (in the variant 0), with a significant amount of spillage, thus completely exhausting the SM's register file. Should more than 2^{16} registers be available per SM, it might be possible to achieve a higher occupancy.

Therefore, for the matrices large enough, only a fraction of all thread blocks in the grid can execute at the same time on a GPU. It is a presumption (but not a requirement) that the CUDA runtime shall schedule a thread block for execution at an early opportunity after a running one terminates, thereby keeping the GPU busy despite of the possible execution time variations (i.e., the number of the inner sweeps and the transformations required) among the thread blocks, especially in the FB case.

Note that $\mathtt{t_y}$ addresses a warp, $0 \le \mathtt{t_y} < \mathtt{w}$, and $\mathtt{t_x}$, $0 \le \mathtt{t_x} < 32$, denotes a lane (a thread) within the warp. Throughout a thread block, each warp is in charge of two "ordinary" (i.e., not block) columns, in the global or in the shared memory, but of which two varies between and within the subphases.

*3.5.1 Subphase **1** (two `ZHERK` or `DSYRK` like operations)* The task of this subphase is to form $\widehat{\mathsf{A}}_k^{(\ell)}$ and then $\widehat{\mathsf{B}}_k^{(\ell)}$ in the shared memory, occupying $\mathrm{Re}(\mathsf{F})$ (and $\mathrm{Im}(\mathsf{F})$), and $\mathrm{Re}(\mathsf{G})$ (and $\mathrm{Im}(\mathsf{G})$), respectively, by a single pass through the pivot block columns of F_k and G_k. The resulting matrices are Hermitian in theory, but unlike in BLAS, both the strictly lower and the strictly upper triangle of each matrix are explicitly computed, even though only the lower triangle is read in the subphase **2**, thus avoiding a possible issue with one triangle not being the exact transpose-conjugate of the other numerically.

A warp indexed by $\mathtt{t_y}$ is assigned two column indices, $p_{\mathtt{y};\mathtt{k}}^{(\ell)}$ and $q_{\mathtt{y};\mathtt{k}}^{(\ell)}$, in the range of the first and the second pivot block column, respectively, as

$$p_{\mathtt{y};\mathtt{k}}^{(\ell)} := \mathtt{p}_\mathtt{k}^{(\ell)} \cdot \mathtt{w} + \mathtt{t_y}, \quad q_{\mathtt{y};\mathtt{k}}^{(\ell)} := \mathtt{q}_\mathtt{k}^{(\ell)} \cdot \mathtt{w} + \mathtt{t_y}.$$

Each thread holds four register-stored variables,

$$\mathtt{r}[\mathtt{t_x}, \mathtt{t_y}], \quad \mathtt{r}[\mathtt{t_x}, \mathtt{t}'_\mathtt{y}], \quad \mathtt{i}[\mathtt{t_x}, \mathtt{t_y}], \quad \mathtt{i}[\mathtt{t_x}, \mathtt{t}'_\mathtt{y}],$$

initially set to zero, that hold the real (first two) and the imaginary (last two) parts of two (partial) dot-products of the columns of F_k and, in the second instance, of G_k, where $\mathtt{t}'_\mathtt{y} := \mathtt{t_y} + \mathtt{w}$.

In a loop over i, starting from $i := \mathtt{t_x}$ and terminating when $i \ge m_F$, with $i := i + 64$, in each step two consecutive chunks of 32 rows (i.e., 64 rows) of the columns $p_{\mathtt{y};\mathtt{k}}^{(\ell)}$ and $q_{\mathtt{y};\mathtt{k}}^{(\ell)}$ are read from $\mathrm{Re}(F_k)$ and $\mathrm{Im}(F_k)$ into $\mathrm{Re}(\mathsf{F}_{64\times32})$ and $\mathrm{Im}(\mathsf{F}_{64\times32})$. Each lane reads an element from the global memory and writes it into the shared memory, both in the coalesced manner, four times per chunk. The elements of the column $p_{\mathtt{y};\mathtt{k}}^{(\ell)}$ are stored into the $\mathtt{t_y}$th column, and those of the column $q_{\mathtt{y};\mathtt{k}}^{(\ell)}$ are stored into the $\mathtt{t}'_\mathtt{y}$th column of the shared memory buffer. The elements of the first chunk are stored into the $\mathtt{t_x}$th row, and of the second chunk into the $(\mathtt{t_x}+32)$th row of the buffer. The thread block is then synchronized, to complete filling the buffer by all warps.

An unrolled inner loop over j, $0 \le j < 64$, followed by a synchronization call, updates the local partial dot-products.

For each j, let $\mathtt{t}'_\mathtt{x} := (\mathtt{t_x} + j) \bmod 64$, and

$$z_\mathtt{y} := (\mathtt{r}[\mathtt{t_x}, \mathtt{t_y}], \mathtt{i}[\mathtt{t_x}, \mathtt{t_y}]), \quad z'_\mathtt{y} := (\mathtt{r}[\mathtt{t_x}, \mathtt{t}'_\mathtt{y}], \mathtt{i}[\mathtt{t_x}, \mathtt{t}'_\mathtt{y}]),$$

$$\mathsf{z}^*_{\mathtt{x}} := (\mathrm{Re}(\mathsf{F}_{64\times 32})[\mathtt{t}'_{\mathtt{x}}, \mathtt{t}_{\mathtt{x}}], -\mathrm{Im}(\mathsf{F}_{64\times 32})[\mathtt{t}'_{\mathtt{x}}, \mathtt{t}_{\mathtt{x}}]),$$
$$\mathsf{z}_{\mathtt{y}} := (\mathrm{Re}(\mathsf{F}_{64\times 32})[\mathtt{t}'_{\mathtt{x}}, \mathtt{t}_{\mathtt{y}}], \mathrm{Im}(\mathsf{F}_{64\times 32})[\mathtt{t}'_{\mathtt{x}}, \mathtt{t}_{\mathtt{y}}]),$$
$$\mathsf{z}'_{\mathtt{y}} := (\mathrm{Re}(\mathsf{F}_{64\times 32})[\mathtt{t}'_{\mathtt{x}}, \mathtt{t}'_{\mathtt{y}}], \mathrm{Im}(\mathsf{F}_{64\times 32})[\mathtt{t}'_{\mathtt{x}}, \mathtt{t}'_{\mathtt{y}}]).$$

Two fused multiply-add operations perform the updates

$$z_{\mathtt{y}} := \mathtt{zfma}(\mathsf{z}^*_{\mathtt{x}}, \mathsf{z}_{\mathtt{y}}, z_{\mathtt{y}}), \quad z'_{\mathtt{y}} := \mathtt{zfma}(\mathsf{z}^*_{\mathtt{x}}, \mathsf{z}'_{\mathtt{y}}, z'_{\mathtt{y}}).$$

The first updates constitute a computation of the dot-product of $\mathtt{t}_{\mathtt{x}}$th and $\mathtt{t}_{\mathtt{y}}$th column of $\mathsf{F}_{64\times 32}$ and updating the partial sum $z_{\mathtt{y}}$ with it, while the second ones form the dot-product of the $\mathtt{t}_{\mathtt{x}}$th and $\mathtt{t}'_{\mathtt{y}}$th column and update $z'_{\mathtt{y}}$ with it. Note that all the rows of the buffer are read exactly once, albeit in the modular (circular) fashion throughout the loop, with the different starting offsets in each column to minimize the shared memory bank conflicts.

When the outer loop over i terminates, $z_{\mathtt{y}}$ and $z'_{\mathtt{y}}$ are stored into $\mathsf{F}_{64\times 32}$ at the corresponding indices, and a synchronization barrier is reached, thus finalizing the formation of $\widehat{\mathsf{A}}^{(\ell)}_k$. The same procedure is repeated with G_k instead of F_k to obtain $\widehat{\mathsf{B}}^{(\ell)}_k$, substituting G and G for F and F, respectively, in the procedure described above. Note that $\mathsf{F}_{96\times 32}$ could (however, unclear if it should) be used instead of $\mathsf{F}_{64\times 32}$, i.e., three chunks instead of two would be read into the buffer and the dot-products of the columns of length 96 instead of 64 would be computed. That would not be possible, though, for G, since $\widehat{\mathsf{A}}^{(\ell)}_k$, once formed, must not be overwritten until the next subphase.

In Figure 4 the arguments `A0D`, `A0J`, `A1D`, `A1J`, `AD`, and `AJ` stand for the real and the imaginary planes of the $p^{(\ell)}_{\mathtt{y};\mathtt{k}}$th and the $q^{(\ell)}_{\mathtt{y};\mathtt{k}}$th columns of F_k, and for $\mathrm{Re}(\mathsf{F}_{64\times 32})$ and $\mathrm{Im}(\mathsf{F}_{64\times 32})$, respectively, in the first call of the device function. The same holds for G_k and $\mathsf{G}_{64\times 32}$ in the second call. The indices `x`, `y0`, and `y1` correspond to $\mathtt{t}_{\mathtt{x}}$, $\mathtt{t}_{\mathtt{y}}$, and $\mathtt{t}'_{\mathtt{y}}$, respectively, while `m` is the number of rows of F or G.

*3.5.2 Subphase **2** (two `ZPOTRF` or `DPOTRF` like operations)* The Cholesky factorization of $\mathsf{A} := \widehat{\mathsf{A}}^{(\ell)}_k$ or $\mathsf{B} := \widehat{\mathsf{B}}^{(\ell)}_k$ consists of two similar, unrolled loops over j. The matrix (in Fortran array order) is accessed and transformed columnwise to avoid the shared memory bank conflicts, but then a transpose-conjugate operation must follow on the computed lower triangular factor to obtain the corresponding upper triangular one. Along with the transposition-conjugation, the strictly lower triangle is zeroed-out, since the following subphase makes no assumptions about the triangularity of the initial matrices.

The first loop iterates over $0 \le j < \mathtt{w}$. First, the jth diagonal element of $\mathrm{Re}(\mathsf{A})$, a_{jj}, is read (the imaginary part is assumed to be zero) if $\mathtt{t}_{\mathtt{y}} = j$ and $\mathtt{t}_{\mathtt{x}} \ge j$ (i.e., in the threads of the jth warp which correspond to the lower triangle, called the "active" threads), and the thread block is then synchronized.

```
// F??(A, i, j) = A[?? * j + i] (??=32|64)
// cuD: real, cuJ: imaginary part (double)
__device__ __forceinline__ void zAhA
(const cuD *const __restrict__ A0D,
 const cuJ *const __restrict__ A0J,
 const cuD *const __restrict__ A1D,
 const cuJ *const __restrict__ A1J,
 volatile cuD *const __restrict__ AD,
 volatile cuJ *const __restrict__ AJ,
 const unsigned m, const unsigned x,
 const unsigned y0, const unsigned y1)
{
  cuD y0xD = 0.0, y1xD = 0.0;
  cuJ y0xJ = 0.0, y1xJ = 0.0;
  const unsigned x32 = x + 32u;

  for (unsigned i = x; i < m; i += 32u) {
    // read the 1st 32 x 32 chunk from RAM
    F64(AD, x, y0) = A0D[i];
    F64(AJ, x, y0) = A0J[i];
    F64(AD, x, y1) = A1D[i];
    F64(AJ, x, y1) = A1J[i];

    i += 32u;
    // read the 2nd 32 x 32 chunk from RAM
    F64(AD, x32, y0) = A0D[i];
    F64(AJ, x32, y0) = A0J[i];
    F64(AD, x32, y1) = A1D[i];
    F64(AJ, x32, y1) = A1J[i];
    __syncthreads();

    #pragma unroll
    for (unsigned j = 0u; j < 64u; ++j) {
      const unsigned x_64 =
        (x + j) & 0x3Fu; // (x + j) % 64u
      const cuD _x_hD =  F64(AD, x_64, x);
      const cuJ _x_hJ = -F64(AJ, x_64, x);
      const cuD _y0_D =  F64(AD, x_64, y0);
      const cuJ _y0_J =  F64(AJ, x_64, y0);
      const cuD _y1_D =  F64(AD, x_64, y1);
      const cuJ _y1_J =  F64(AJ, x_64, y1);
      // [complex] y0x = _x_h * _y0_ + y0x
      Zfma(y0xD, y0xJ, _x_hD, _x_hJ,
        _y0_D, _y0_J, y0xD, y0xJ);
      // [complex] y1x = _x_h * _y1_ + y1x
      Zfma(y1xD, y1xJ, _x_hD, _x_hJ,
        _y1_D, _y1_J, y1xD, y1xJ);
    }
    __syncthreads();
  }

  // A^H * A stored into the shared memory
  F32(AD, x, y0) = y0xD;
  F32(AJ, x, y0) = y0xJ;
  F32(AD, x, y1) = y1xD;
  F32(AJ, x, y1) = y1xJ;
  __syncthreads();
}
```

Figure 4. A CUDA implementation of the subphase **1** ($\mathbb{C}$).

The active threads then scale the jth column below the diagonal, each thread the real and the imaginary part of its element in the $\mathtt{t_x}$th row, by $1/\sqrt{a_{jj}}$, while the diagonal is set to $(\sqrt{a_{jj}}, 0)$, and the thread block is then synchronized.

Next, the columns to the right of the jth have to be updated, with all warps (but not all their threads) participating in the update. Let $j' := (j+1) + \mathtt{t_y}$. Then, if $\mathtt{t_x} \geq j'$,

$$\mathsf{A}[\mathtt{t_x}, j'] := \mathtt{zfma}(-\mathsf{A}[\mathtt{t_x}, j], \overline{\mathsf{A}[j', j]}, \mathsf{A}[\mathtt{t_x}, j']),$$

and the thread block is synchronized. However, this only updates the columns from $j+1$ to $j+\mathtt{w}$. The same update has to be performed with $j'' := j' + \mathtt{w}$ instead of j', i.e., if $\mathtt{t_x} \geq j'$ (which also ensures that $j'' < 32$),

$$\mathsf{A}[\mathtt{t_x}, j''] := \mathtt{zfma}(-\mathsf{A}[\mathtt{t_x}, j], \overline{\mathsf{A}[j'', j]}, \mathsf{A}[\mathtt{t_x}, j'']),$$

and another thread synchronization occurs.

The second loop over $\mathtt{w} \leq j < 32$ is identical to the first one, except that $\mathtt{t}'_\mathtt{y}$ is used instead of $\mathtt{t_y}$ and the second updates (of the j''th columns) are not needed since $j'' \geq 32$.

The ensuing transpose-conjugate with zeroing-out of the strictly lower triangle is performed by reading $\mathsf{A}[\mathtt{t_x}, \mathtt{t_y}]$ and $\mathsf{A}[\mathtt{t_x}, \mathtt{t}'_\mathtt{y}]$ into the register of the $[\mathtt{t_x}, \mathtt{t_y}]$th thread if $\mathtt{t_x} \geq \mathtt{t_y}$ and $\mathtt{t_x} \geq \mathtt{t}'_\mathtt{y}$, respectively (i.e., the indices belong to the lower triangle of A). Otherwise, those registers are set to 0. After negating the imaginary parts in the former case, the values are written to $\mathsf{A}[\mathtt{t_y}, \mathtt{t_x}]$ and $\mathsf{A}[\mathtt{t}'_\mathtt{y}, \mathtt{t_x}]$, respectively, unfortunately requiring the shared memory bank conflicts, and the thread block is synchronized, yielding $\mathsf{F} := \widehat{\mathsf{F}}_k^{(\ell)}$. The same procedure is then repeated with B instead of A, yielding $\mathsf{G} := \widehat{\mathsf{G}}_k^{(\ell)}$.

3.5.3 Subphase ***3*** *(the pointwise one-sided algorithm)*

The pointwise implicit Hari–Zimmermann algorithm, described in section 2, subsections 2.1, 2.2, and the relevant parts of subsections 2.4 and 2.5, is implemented as follows.

The $\mathtt{t_y}$th warp transforms the pairs of columns of F, G, and Z in each inner step $l' \geq 0$. Let $l := l' \bmod 31$, since the ME strategy is used exclusively at the inner level in the tests. Each of the three pivot pairs comprise the columns indexed by $\mathtt{p}_{\mathtt{y};l}$ and $\mathtt{q}_{\mathtt{y};l}$, where the indices are read from the lth row of the inner strategy table at the position $\mathtt{t_y}$. Within a warp, the $\mathtt{t_x}$th thread is responsible for the elements in the $\mathtt{t_x}$th row of those columns.

First, Z is initialized similarly to the procedure described in subsection 3.3, but on the shared memory level. In the variants 2, 3, 6, and 7, the diagonal of $\mathrm{Re}(\mathsf{Z})$ is set to unity, and the rest to zero, by the threads in charge of those elements. In the variants 0 and 4, the sum of squares of the magnitudes of the elements of the columns g_j, i.e., $\|\mathsf{g}_j\|_F^2$, where $j \in \{\mathtt{p}_{\mathtt{y};l}, \mathtt{q}_{\mathtt{y};l}\}$ and $l = 0$, is computed by a sum-reduction as in Appendix C. The thread block is then synchronized. For each of the two indices j, $\mathrm{Im}(\mathsf{Z})[\mathtt{t_x}, j]$ is set to zero, as well as $\mathrm{Re}(\mathsf{Z})[\mathtt{t_x}, j]$, except when $\mathtt{t_x} = j$, where $\mathrm{Re}(\mathsf{Z})[j,j] := 1/\sqrt{\|\mathsf{g}_j\|_F^2}$ if $\|\mathsf{g}_j\|_F^2 \neq 1$, and one otherwise. The columns f_j and g_j are scaled by $\mathrm{Re}(\mathsf{Z})[j,j]$ if it is not unity, and the thread block is synchronized. The similar procedure is applied in the variants 1 and 5, except that the partial sums of squares are computed as in Appendix A (see subsection 3.3), and summed by a routine from Appendix C.

Having thus obtained F_0, G_0, and Z_0, the iterative part of the algorithm starts, with at most 30 (FB) or 1 (BO) inner sweeps. At the start of each sweep two per-sweep counters, of the "big" ($\mathfrak{b}$) and of all ($\mathfrak{s}$) transformations applied, are reset to zero. The counters are kept in each thread, but their values are synchronized across all threads in a thread block.

In the step l and the warp $\mathtt{t_y}$, let $i := \mathtt{p}_{\mathtt{y};l}$ and $j := \mathtt{q}_{\mathtt{y};l}$. The elements of the three pivot column pairs are loaded into the registers by each thread reading its row from the shared memory, after which the thread block is synchronized. For each original element, there are two variables for its real and imaginary parts, and two more variables to hold the value of the new element after transformation, since the old value is used twice in computing the new one and thus cannot be overwritten. For example, $\mathrm{Im}(\mathsf{F})[\mathtt{t_x}, i]$ has $\mathrm{Im}(\mathsf{F}')[\mathtt{t_x}, i]$ as its counterpart.

The 2×2 pivot submatrices $\widehat{A}'_l$ and $\widehat{B}'_l$ are then formed. The diagonal elements are obtained by computing the squares of the column norms as above, and the off-diagonal ones are given by the dot-products, either ordinary (i.e., by sum-reducing the real and the imaginary parts of the products of an element of the ith column conjugated and the corresponding element of the jth column) or enhanced (as in Appendix A) ones.

However, $\widehat{A}_l$ and $\widehat{B}_l$ thus obtained have to be multiplied by $\widehat{D}_l$ from the left and right in the variants 2, 3, 6, and 7 to get $\widehat{A}'_l$ and $\widehat{B}'_l$. If $\widehat{B}_{11;l} \neq 1$, then $\widehat{A}'_{11;l} := \widehat{A}_{11;l}/\widehat{B}_{11;l}$, $\widehat{D}_{11;l} := 1/\sqrt{\widehat{B}_{11;l}}$, and $\widehat{A}_{12;l}$, $\widehat{B}_{12;l}$ are scaled by $\widehat{D}_{11;l}$; otherwise, $\widehat{D}_{11;l} = 1$, as it is in the variants 0, 1, 4, and 5. If $\widehat{B}_{22;l} \neq 1$, then $\widehat{A}'_{22;l} := \widehat{A}_{22;l}/\widehat{B}_{22;l}$, $\widehat{D}_{22;l} := 1/\sqrt{\widehat{B}_{22;l}}$, and $\widehat{A}_{12;l}$, $\widehat{B}_{12;l}$ are scaled by $\widehat{D}_{22;l}$; otherwise, $\widehat{D}_{22;l} = 1$.

All threads in a warp now have the elements of the pivot submatrices held in their register-stored variables, and the elements' values are identical across the warp. Therefore, the subsequent computation of $\widehat{Z}_l$ on a per-thread basis also has to produce the same transformation across the warp.

First it has to be established whether a transformation is warranted. If the relative orthogonality criterion is satisfied, $\hat{\mathfrak{s}}$ is set to zero, else to one. All threads in a thread block agree if there is some computational work (apart from merely the optional column sorting) to be done in the

current step by uniformly incrementing $\mathfrak{s}$,

$$\mathfrak{s} := \mathfrak{s} + \texttt{__syncthreads\_count}(\hat{\mathfrak{s}})/32,$$

by the number of the thread block's warps with the non-trivial transformations to be applied.

If $\hat{\mathfrak{s}} = 0$ and $\widehat{A}'_{11;l} < \widehat{A}'_{22;l}$, then $\mathrm{V}(\mathsf{Y}')[\mathtt{t_x}, i] := \mathrm{V}(\mathsf{Y})[\mathtt{t_x}, j]$ and $\mathrm{V}(\mathsf{Y}')[\mathtt{t_x}, j] := \mathrm{V}(\mathsf{Y})[\mathtt{t_x}, i]$, where $\mathrm{V} \in \{\mathrm{Re}, \mathrm{Im}\}$ and $\mathsf{Y} \in \{\mathsf{F}, \mathsf{G}, \mathsf{Z}\}$ in `bstep1s`. Then, the values of the new variables are stored in the shared memory. When $\hat{\mathfrak{s}} = 0$ in `bstep1n`, or in `bstep1s` and $\widehat{A}'_{11;l} \geq \widehat{A}'_{22;l}$, the new variables take the value of the corresponding old ones, i.e., no column swapping occurs.

Otherwise, for $\hat{\mathfrak{s}} = 1$, $\widehat{Z}'_l$ is computed according to a procedure described either in subsection 2.1.1 for the complex, or in subsection 2.1.2 for the real case. Then, it is established whether the criterion C1 (for the variants 0, 1, 2, and 3) or the criterion C2 (for the variants 4, 5, 6, and 7) indicates that the transformation is "small". If so, $\hat{\mathfrak{b}} := 0$; else, $\hat{\mathfrak{b}} := 1$.

If $\widehat{D}_{11;l} \neq 1$, the first row of $\widehat{Z}'_l$ is scaled by $\widehat{D}_{11;l}$. If $\widehat{D}_{22;l} \neq 1$, the second row of $\widehat{Z}'_l$ is scaled by $\widehat{D}_{22;l}$. Now the completed transformation $\widehat{Z}_l$ has to be applied to the pivot columns:

$$\mathsf{Y}'[\mathtt{t_x}, i] := \texttt{zfma}(\mathsf{Y}[\mathtt{t_x}, j], \widehat{Z}_{21;l}, \mathsf{Y}[\mathtt{t_x}, i] \cdot \mathrm{Re}(\widehat{Z}_{11;l})),$$
$$\mathsf{Y}'[\mathtt{t_x}, j] := \texttt{zfma}(\mathsf{Y}[\mathtt{t_x}, i], \widehat{Z}_{12;l}, \mathsf{Y}[\mathtt{t_x}, j] \cdot \mathrm{Re}(\widehat{Z}_{22;l})),$$

where $\mathsf{Y} \in \{\mathsf{F}, \mathsf{G}, \mathsf{Z}\}$. If one or both scaled cosines lying on the diagonal of $\widehat{Z}_l$ are equal to one, the transformation can be (and is) simplified by removing the corresponding multiplications without numerically affecting the result.

In `bstep1s`, to determine if the column swap is required, the squares of the norms of the transformed columns of F are computed as the sum-reduced sums of squares of the magnitudes of the new (F') elements, depending on the variant. Those two values are however not stored for the next step, because that would require an additional shared memory workspace that might not be available on all supported architectures.

In the real case it is easy to compute instead the transformed diagonal elements of the first pivot submatrix directly Novaković et al. (2015):

$$a''_{11} := \widehat{Z}^2_{11;l}\widehat{A}'_{11;l} + 2\widehat{Z}_{11;l}\widehat{Z}_{21;l}\widehat{A}'_{12;l} + \widehat{Z}^2_{21;l}\widehat{A}'_{22;l},$$
$$a''_{22} := \widehat{Z}^2_{12;l}\widehat{A}'_{11;l} + 2\widehat{Z}_{22;l}\widehat{Z}_{12;l}\widehat{A}'_{12;l} + \widehat{Z}^2_{22;l}\widehat{A}'_{22;l},$$

and to swap the ith and the jth column when $a''_{11} < a''_{22}$.

If the norm of the ith column is smaller than the norm of the jth column, then the values of $\mathsf{Y}'[\mathtt{t_x}, i]$ and $\mathsf{Y}'[\mathtt{t_x}, j]$ are swapped via an intermediary variable. Else, or in `bstep1n`, no swaps occur. The values of the new variables are then stored in the shared memory, and $\mathfrak{b}$ is uniformly incremented across the thread block,

$$\mathfrak{b} := \mathfrak{b} + \texttt{__syncthreads\_count}(\hat{\mathfrak{b}})/32.$$

The lth step is now complete.

At the end of a sweep, if $\hat{s} = 0$, the loop is terminated. Else, the counters $\mathfrak{S}$ and $\mathfrak{B}$, set at the start of this subphase to zero, are incremented by $\mathfrak{s}$ and $\mathfrak{b}$, respectively.

The same rescaling as in subsection 3.4 with `f=false`, but performed on the shared memory, yields $\widetilde{\mathsf{Z}}^{(\ell)}_k$. Using the last values of $\mathsf{F}'[\mathtt{t_x}, i]$ and $\mathsf{G}'[\mathtt{t_x}, i]$, the squares of the norms of the ith column of F' and G', respectively, are computed. Then, $\mathsf{Z}'[\mathtt{t_x}, i]$ is read (or its last value is used), scaled by $1/\sqrt{\|\mathsf{f}'_i\|^2_F + \|\mathsf{g}'_i\|^2_F}$, stored, and the thread block is synchronized. The same procedure is repeated with j instead of i, giving $\widetilde{\mathsf{Z}}^{(\ell)}_k := \mathsf{Z}'$.

A thread with $\mathtt{t_x} = \mathtt{t_y} = 0$ stores $\mathfrak{S}$ and $\mathfrak{B}$ into C as

$$C[2 \cdot \mathtt{b_x}] := \mathfrak{S}, \quad C[2 \cdot \mathtt{b_x} + 1] := \mathfrak{B},$$

and finally the thread block is synchronized.

*3.5.4 Subphase **4** (three postmultiplications)* In this subphase the pivot block columns of F_k, G_k, and Z_k are multiplied by $\widetilde{\mathsf{Z}}^{(\ell)}_k$ and overwritten by the respective results.

Each multiplication of a pair of pivot block columns (residing in the global memory) by $\widetilde{\mathsf{Z}}^{(\ell)}_k$ (residing in the shared memory in Z) and the following update are performed by a single pass over (i.e., a single read from and a single write to) the block columns, using the Cannon-like algorithm Cannon (1969) for parallel multiplication of two square matrices.

Reading the chunks of a block column pair from the global memory is identical to the one from the subphase **1** in subsection 3.5.1, except that in each iteration of the outer loop (over i) only one chunk is read to Y, instead of two (which would also be a possibility). The number of loop iterations (in parenthesis) depends on the number of rows of F_k ($m_F/32$), G_k ($m_G/32$), and Z_k ($n/32$). Here, Y is F when updating F_k and Z_k, and G when updating G_k. The thread block is then synchronized.

The per-thread variables to hold the product of the current chunk with Z are set to zero. Each thread is in charge of forming the elements with indices $[\mathtt{t_x}, \mathtt{t_y}]$ and $[\mathtt{t_x}, \mathtt{t'_y}]$ of the product Π.

The initial skews are defined as $\imath := (\mathtt{t_y} + \mathtt{t_x}) \bmod 32$ and $\imath' := (\mathtt{t'_y} + \mathtt{t_x}) \bmod 32$. Then, in each iteration of the unrolled inner loop over $0 \leq j < 32$ the local elements of Π are updated,

$$\Pi[\mathtt{t_x}, \mathtt{t_y}] := \texttt{zfma}(\mathsf{Y}[\mathtt{t_x}, \imath], \mathsf{Z}[\imath, \mathtt{t_y}], \Pi[\mathtt{t_x}, \mathtt{t_y}]),$$
$$\Pi[\mathtt{t_x}, \mathtt{t'_y}] := \texttt{zfma}(\mathsf{Y}[\mathtt{t_x}, \imath'], \mathsf{Z}[\imath', \mathtt{t'_y}], \Pi[\mathtt{t_x}, \mathtt{t'_y}]),$$

and $\imath$ and $\imath'$ are cyclically shifted as $\imath := (\imath + 1) \bmod 32$ and $\imath' := (\imath' + 1) \bmod 32$. When the inner loop terminates, the thread block is synchronized.

The local values of Π now have to be written back to the global memory, where $\Pi[\mathtt{t_x}, \mathtt{t_y}]$ overwrites $Y_k[i, p^{(\ell)}_{\mathtt{y;k}}]$, while $\Pi[\mathtt{t_x}, \mathtt{t'_y}]$ overwrites $Y_k[i, q^{(\ell)}_{\mathtt{y;k}}]$, for Y being one of $\{F, G, Z\}$. The thread block is then synchronized and the next outer iteration, if any are left, follows.

This procedure is called thrice to update F_k, G_k and Z_k, after which the kernel execution (i.e., the kth outer step) terminates and the control returns to the CPU.

Figure 5 shows the postmultiplication device function, where the arguments `A0D`, `A0J`, `A1D`, and `A1J` have the same meaning as in Figure 4, but the columns of Z_k are also expected. A shared memory buffer, in which the 32×32 chunks of a block column pair are loaded and packed, is pointed to by `AD` and `AJ`, while `BD` and `BJ` point to the accumulated transformation matrix from the subphase **3**, by which the postmultiplication has to take place. The product of matrices `A` and `B` overwrites the respective chunk of the original block columns before another chunk is loaded.

3.5.5 Dataflow and the shared memory perspective

In Figure 6 the subphases in the simpler, real case are summarized from a perspective of the data in the shared memory and the transformations that it undergoes. The subfigures (a–b) show the chunks of data coming from the global memory to form A and B. Two Cholesky factorizations are shown in the subfigures (c–d), after which the subfigures (e–h) correspond to the subphase **3**. The subfigure [f] shows the optional (depending on the variant) prescaling of F and G. The final three subfigures, (i–k), show the three postmultiplications taking place, with a chunk of data read from and written to the global memory. Two different hues of the first and the middle parts of the shared memory depiction indicate that the chunks with 64 instead of 32 rows might be used there.

3.6 The CPU part of the algorithm

Algorithm 1 summarizes the CPU-side actions with a single GPU. The same routine is called within the multi-GPU algorithm, except for allowing S to be a parameter (not the constant 30 as it is assumed here), a variation of the final rescaling of Z, and some differences regarding the `initFGZ` call, described in subsections 3.3 and 4.2. The copy-ins and copy-outs of the majority of data, as well as the initialization of the constant memory, are left out from Algorithm 1 but are included in the single-GPU timing in subsection 5.

Apart from the copy-ins, copy-outs, and zeroings of data, there is no scope for using more than one CUDA stream. All GPU operations can be performed in any predefined stream `s` (e.g., in the default one if no other has been explicitly chosen). Also, as no GPU computation, except the fast `rescale`, can be overlapped with any CPU task, the execution time of the algorithm depends almost solely on the GPU performance and the time required to set up a kernel call.

Each kernel invocation and each sequence of memory copy/set operations is followed in the testing code by a (generally redundant, except where noted in Algorithm 1) `cudaStreamSynchronize` call on the chosen stream `s`, to keep the CPU-side timing consistent (but maybe higher than it is necessary).

```
__device__ __forceinline__ void zPostMult
(cuD *const __restrict__ A0D,
 cuJ *const __restrict__ A0J,
 cuD *const __restrict__ A1D,
 cuJ *const __restrict__ A1J,
 volatile cuD *const __restrict__ AD,
 volatile cuJ *const __restrict__ AJ,
 volatile const cuD *const __restrict__ BD,
 volatile const cuJ *const __restrict__ BJ,
 const unsigned x, const unsigned y0,
 const unsigned y1, const unsigned m)
{
  // Cannon-like C = A * B
  for (unsigned i = x; i < m; i += 32u) {
    F32(AD, x, y0) = A0D[i];
    F32(AJ, x, y0) = A0J[i];
    F32(AD, x, y1) = A1D[i];
    F32(AJ, x, y1) = A1J[i];
    __syncthreads();

    cuD Cxy0D = 0.0, Cxy1D = 0.0;
    cuJ Cxy0J = 0.0, Cxy1J = 0.0;
    unsigned // skew (mod 32)
      p0 = ((y0 + x) & 0x1Fu),
      p1 = ((y1 + x) & 0x1Fu);

    // multiply and cyclic shift (mod 32)
    #pragma unroll
    for (unsigned k = 0u; k < 32u; ++k) {
      Zfma(Cxy0D, Cxy0J,
        F32(AD, x, p0), F32(AJ, x, p0),
        F32(BD, p0, y0), F32(BJ, p0, y0),
        Cxy0D, Cxy0J);
      Zfma(Cxy1D, Cxy1J,
        F32(AD, x, p1), F32(AJ, x, p1),
        F32(BD, p1, y1), F32(BJ, p1, y1),
        Cxy1D, Cxy1J);
      p0 = (p0 + 1u) & 0x1Fu;
      p1 = (p1 + 1u) & 0x1Fu;
    }
    __syncthreads();

    A0D[i] = Cxy0D; A0J[i] = Cxy0J;
    A1D[i] = Cxy1D; A1J[i] = Cxy1J;
    __syncthreads();
  }
}
```

Figure 5. A CUDA C implementation of the subphase **4** ($\mathbb{C}$).

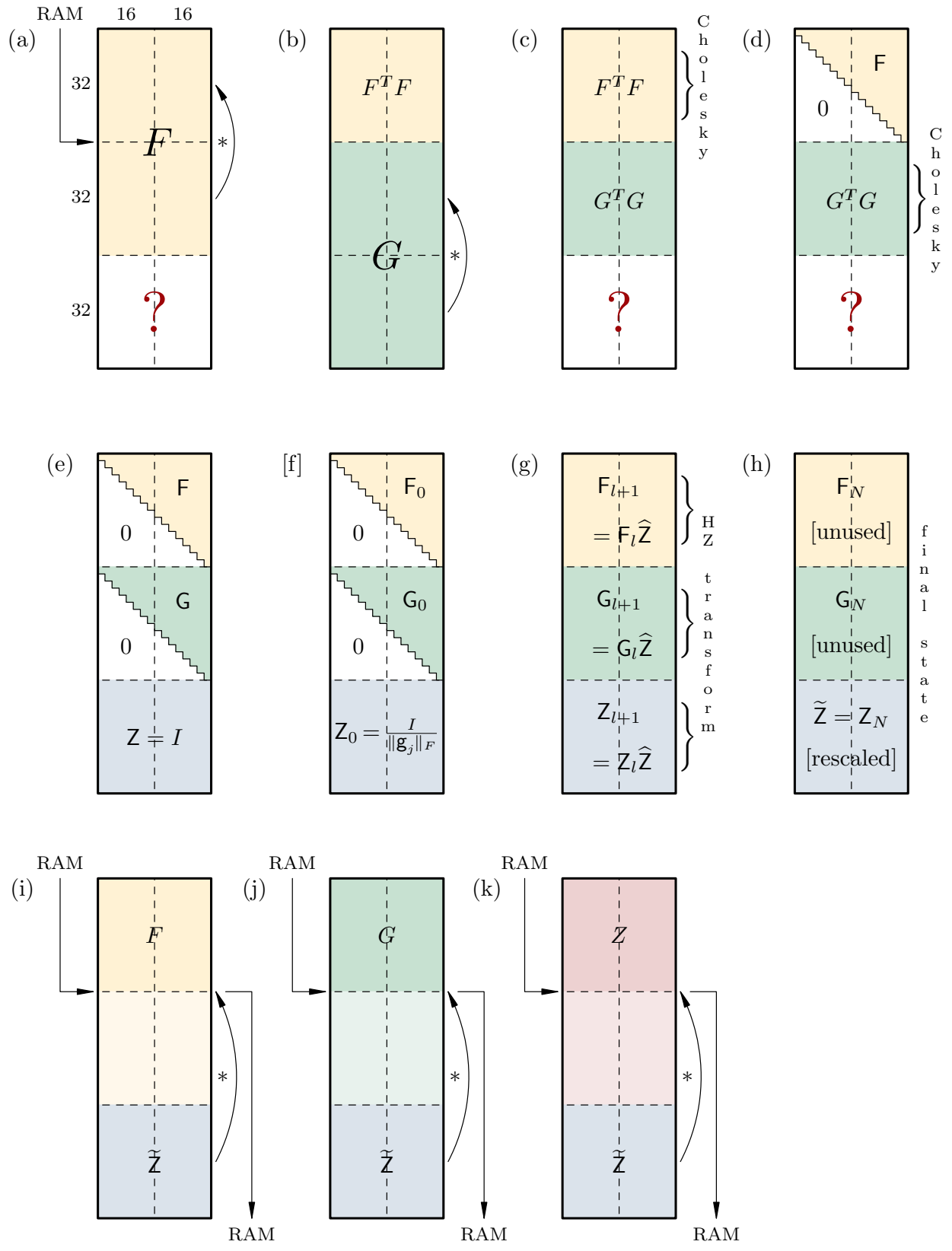

Figure 6. The sequence of operations performed on the shared memory of a GPU's multiprocessor by a thread block of the main computational kernel.

Algorithm 1: The CPU part of the single-GPU implicit Hari–Zimmermann algorithm with data in place.

```
initFGZ();    // execute once on a single GPU in the stream s to get F_0, G_0, Z_0
S̃ := 0;  B̃ := 0;                    // initialize the global convergence statistics
for 0 ≤ c < S do                                                    // outer sweep c
    cudaMemsetAsync(C, 0, n · sizeof*C, s);                          // zero-out C
    for 0 ≤ k < n′ do                                  // a loop over the outer steps
        bstep1(s/n)(k); // each main kernel call in s transforms n/2 block pivots
    end for
    cudaMemcpyAsync(C̃, C, n · sizeof*C, cudaMemcpyDeviceToHost, s);     // retrieve C
    cudaStreamSynchronize(s);                                          // synchronize s
    s̃ := 0;  b̃ := 0;                    // initialize the sweep convergence statistics
    for 0 ≤ i < n/2 do                     // gather the sweep convergence statistics
        s̃ := s̃ + C̃[2 · i];  b̃ := b̃ + C̃[2 · i + 1];
    end for
    S̃ := S̃ + s̃;  B̃ := B̃ + b̃;              // update the global convergence statistics
    if b̃ = 0 then break;                // no big transformations performed in the sweep
    rescale(false);                                  // a fast rescaling of Z in s
end for
rescale(f);// queued in s with f=true on one GPU and f=false on multiple GPUs
```

Except the reductions of $\tilde{\mathfrak{s}}$ and $\tilde{\mathfrak{b}}$ for very large matrices, no other part of the algorithm might benefit from being executed multi-threadedly on the CPU. From the CPU perspective, the algorithm is therefore purely sequential.

Note that the algorithm stops when $\tilde{\mathfrak{b}} = 0$, i.e., when no big transformations occurred in an outer sweep. The "global" counts $\widetilde{\mathfrak{S}}$ and $\widetilde{\mathfrak{B}}$ of all and of big transformations applied during the execution of Algorithm 1 are only informative here, but they are consulted in the multi-GPU algorithm's stopping criterion.

4 The multi-GPU implementation

When the input data is larger than the available RAM on a single GPU, it is necessary to either split the workload among the several GPUs, or resort to some out-of-core technique for swapping the parts of data in and out of the GPU as the computation progresses. Here, only the former approach is followed, since it is simpler, more efficient and widely applicable now when the multi-GPU installations are becoming ubiquitous. In the case when not enough GPUs are available for the input data to be distributed across them, see an outline of a possible out-of-core single-GPU algorithm in Appendix B.

There is no single, best and straightforward way of generalizing a single-GPU algorithm to multiple GPUs. For the (ordinary and hyperbolic) SVD, the approach in Novaković (2015) was to distribute the matrix over the GPUs, shorten the assigned part of the matrix (the Grammian formation being done by cuBLAS, and the ensuing Cholesky factorization by MAGMA Tomov et al. (2010)), run the single-GPU algorithm on the shortened part, update the original (non-shortened) columns, and redistribute the parts. Despite its decent performance, such a three-level algorithm suffered from the increased memory usage and some numerical difficulties, both with the stopping criteria and with the relative accuracy obtained.

A different approach is taken here, to achieve the optimal GPU *and* CPU memory usage (without any work arrays) and a better accuracy, but with a possible performance penalty induced by transforming the tall and skinny parts of the matrices directly, without any shortening. As no floating-point computation is performed on the CPU, while the GPU computation still does not rely on any numerical libraries, it is guaranteed that the results stay bitwise identical in the repeated runs over the same input data with the same parameters and the same number of GPUs.

4.1 Algorithm setup

In this subsection the multi-GPU computational environment, the input and the output data distribution across the CPUs and the GPUs, the communication-aware Jacobi strategies, and the algorithm's initialization are explained.

4.1.1 MPI environment Unlike in Novaković (2015), where the multiple GPUs were assumed to belong to the same node, and thus a separate CPU thread of a single process could be dedicated for driving the computation on each GPU, here a more flexible solution has been chosen, by assigning to a GPU a single-threaded MPI Message Passing Interface Forum (2015) process. The GPUs can thus be selected from any number of nodes, with a different

number of them on each node. Also, the GPUs are not required to be architecturally identical or even similar across the processes in an MPI job, as long as they all have enough RAM available.

The count of GPUs, and thus the governing MPI processes ($\mathbf{s}$), for the multi-GPU algorithm is not constrained in principle, save for being at least two (otherwise, the single-GPU algorithm is sufficient), and small enough so that at least two (but for the reasons of performance, a multiple of 32) columns of each matrix are available to each process, when the matrices are divided among them columnwise, as described below. The upper bound on the number of GPUs is a tunable parameter in the code, while the MPI implementation might have its own limit on the number of processes in a job.

The MPI processes need not be arranged in any special topology. Only the predefined `MPI_COMM_WORLD` communicator is used. A GPU and its governing process are jointly referred to by the process' rank ($\mathbf{r}$) in that communicator.

It is assumed that the MPI distribution is CUDA-aware in a sense that sending data from the GPU RAM of one process and receiving it in the CPU RAM of another (including the same) process is possible with the standard MPI point-to-point communication routines (i.e., no manual CPU buffering of the GPU data is necessary).

Also, the number of elements of each local submatrix has to be at most `INT_MAX`, which at present is the upper limit on the count of elements that can be transferred in a single MPI operation Hammond et al. (2014). That limit is easily circumvented by transferring the data in several smaller chunks, but such chunking has not been implemented since it was not needed for the amount of RAM (16 GiB) of the GPUs used for testing. That issue will have to be addressed for the future GPUs.

4.1.2 Data distribution The matrices F, G, and Z, and the vectors Σ_F, Σ_G, and Σ, are assumed to always stay distributed among the MPI processes, i.e., at no moment they are required to be present in entirety in any subset of the processes. The amount of the CPU and the GPU memory required is identical (i.e., not depending on $\mathbf{r}$), constant throughout the computation, and derivable in advance from $m_F, m_G, m_Z := n$, and $\mathbf{s}$ for all processes.

If $n \bmod \mathbf{s} \neq 0$ or $(n/\mathbf{s}) \bmod 32 \neq 0$, the matrices F, G, and Z are bordered as described in subsection 3.1, but requiring that the enlarged n satisfy $n \bmod (32 \cdot \mathbf{s}) = 0$. Similarly, the bordering is required if $m_F \bmod 64 \neq 0$ or $m_G \bmod 64 \neq 0$.

The columns of the bordered matrices can be distributed evenly among the processes, such that each process is assigned $\mathbf{n} := n/\mathbf{s}$ columns. Let $\mathbf{w} := \mathbf{n}/2$ consecutive columns of an entire matrix be called a stripe, to avoid reusing the term "block column". Then, a process holds two, not necessarily consecutive, stripes of each matrix, logically separate but with their real parts physically joined in the same memory allocation, as well as their imaginary parts. The dimensions of two joined stripes, one following the other in the Fortran array order, of $\mathrm{Re}(F)/\mathrm{Im}(F)$ ($m_F \times \mathbf{n}$), $\mathrm{Re}(G)/\mathrm{Im}(G)$ ($m_G \times \mathbf{n}$), and $\mathrm{Re}(Z)/\mathrm{Im}(Z)$ ($n \times \mathbf{n}$), fit the requirements for the input data of the single-GPU algorithm.

The CPU RAM of the $\mathbf{r}$th process thus holds two joined stripes in $\mathrm{Re}(F^{(\mathbf{r})})$, $\mathrm{Im}(F^{(\mathbf{r})})$, $\mathrm{Re}(G^{(\mathbf{r})})$, $\mathrm{Im}(G^{(\mathbf{r})})$, $\mathrm{Re}(Z^{(\mathbf{r})})$, and $\mathrm{Im}(Z^{(\mathbf{r})})$ allocations. The same memory arrangement is present in the GPU RAM, which holds the same stripes undergoing the transformations and joined in the allocations $\mathrm{Re}(F^{[\mathbf{r}]})$, $\mathrm{Im}(F^{[\mathbf{r}]})$, $\mathrm{Re}(G^{[\mathbf{r}]})$, $\mathrm{Im}(G^{[\mathbf{r}]})$, $\mathrm{Re}(Z^{[\mathbf{r}]})$, and $\mathrm{Im}(Z^{[\mathbf{r}]})$. The first stripe within an allocation is denoted by the index $\mathbf{0}$, and the second one by the index $\mathbf{1}$; e.g., $\mathrm{Im}(G^{[\mathbf{r}]}_{\mathbf{1}})$ is the second stripe in $\mathrm{Im}(G^{[\mathbf{r}]})$.

A similar distribution is in place for Σ_F, Σ_G, and Σ, where each process holds $\Sigma_F^{(\mathbf{r})}$, $\Sigma_G^{(\mathbf{r})}$, and $\Sigma^{(\mathbf{r})}$ in the CPU RAM, and $\Sigma_F^{[\mathbf{r}]}$, $\Sigma_G^{[\mathbf{r}]}$, and $\Sigma^{[\mathbf{r}]}$ in the GPU RAM, where each allocation is of length $\mathbf{n}$ and is unused in the algorithm until after the last step. Each process also has its convergence vectors $C^{(\mathbf{r})}$ and $C^{[\mathbf{r}]}$, of length $\mathbf{n}/\mathtt{w}$, in the CPU and in the GPU RAM, respectively.

4.1.3 Communication-aware Jacobi strategies The parallel Jacobi strategies, as defined in subsection 2.2, do not contain any explicit information on how to progress from one step to another by exchanging the pivot (block) columns among the tasks in a distributed memory environment. However, such a communication pattern can be easily retrieved by looking at each two successive steps, k and $k' := (k+1) \bmod s$, and for each task ℓ in the kth step finding the tasks ℓ' and ℓ'' in the k'th step that are to hold either $i_k^{(\ell)}$th or $j_k^{(\ell)}$th (block) column.

Therefore, given either ME or MM strategy table for the order $\mathfrak{n} := n/\mathbf{w}$ (with the stripes seen as the block columns), each process independently computes and encodes the strategy's communication pattern before the start of the algorithm. Such a computation requires $O(\mathfrak{n}^3)$ comparisons, but since $\mathfrak{n}$ is a small number an unacceptable overhead is not incurred. The computation can be (but it has not been) parallelized on a CPU, e.g., by turning the outer loop over k into a parallel one. The multi-GPU algorithm then references the following encoded mapping when progressing from one step to the next.

After each outermost (multi-GPU) step $\mathbf{k}$ of the algorithm, the first of each two joined stripes on the $\mathbf{r}$th GPU has to be transferred to either the first or the second stripe on the $\mathbf{t}^{\{\mathbf{0}\}}$th CPU, for some $\mathbf{t}^{\{\mathbf{0}\}}$. Similarly, the second stripe on the $\mathbf{r}$th GPU has to be transferred to either the first or the second stripe on the $\mathbf{t}^{\{\mathbf{1}\}}$th CPU, for some $\mathbf{t}^{\{\mathbf{1}\}}$, establishing a mapping

$$(\mathbf{k}, \mathbf{r}) \mapsto (\mathbf{p}_{\mathbf{k},\mathbf{r}}, \mathbf{q}_{\mathbf{k},\mathbf{r}}, \mathbf{t}^{\{\mathbf{0}\}}_{\mathbf{k},\mathbf{r}}, \mathbf{t}^{\{\mathbf{1}\}}_{\mathbf{k},\mathbf{r}}),$$

where $\mathbf{t}^{\{\mathbf{0}\}}_{\mathbf{k},\mathbf{r}} := \pm(\mathbf{t}^{\{\mathbf{0}\}}_{\mathbf{k},\mathbf{r}} + 1)$, $\mathbf{t}^{\{\mathbf{1}\}}_{\mathbf{k},\mathbf{r}} := \pm(\mathbf{t}^{\{\mathbf{1}\}}_{\mathbf{k},\mathbf{r}} + 1)$, while $\mathbf{p}_{\mathbf{k},\mathbf{r}}$ and $\mathbf{q}_{\mathbf{k},\mathbf{r}}$ are indices of the first and the second stripe in the entire matrix, $0 \leq \mathbf{p}_{\mathbf{k},\mathbf{r}} < \mathbf{q}_{\mathbf{k},\mathbf{r}} < \mathfrak{n}$, and $\mathbf{t}^{\{\mathbf{0}\}}_{\mathbf{k},\mathbf{r}} \neq \mathbf{t}^{\{\mathbf{1}\}}_{\mathbf{k},\mathbf{r}}$ are the MPI ranks. If the $\mathbf{p}_{\mathbf{k},\mathbf{r}}$th stripe globally (i.e., the first locally) has to be transferred to the first stripe in the $\mathbf{t}^{\{\mathbf{0}\}}_{\mathbf{k},\mathbf{r}}$th process, that is encoded as $-(\mathbf{t}^{\{\mathbf{0}\}}_{\mathbf{k},\mathbf{r}} + 1)$, else the second stripe of the target is encoded as $(\mathbf{t}^{\{\mathbf{0}\}}_{\mathbf{k},\mathbf{r}} + 1)$. Similarly, if the $\mathbf{q}_{\mathbf{k},\mathbf{r}}$th stripe globally (i.e., the second locally) has to be transferred to the first stripe in the $\mathbf{t}^{\{\mathbf{1}\}}_{\mathbf{k},\mathbf{r}}$th process, that is encoded as $-(\mathbf{t}^{\{\mathbf{1}\}}_{\mathbf{k},\mathbf{r}} + 1)$, else the second stripe of the target is encoded as $(\mathbf{t}^{\{\mathbf{1}\}}_{\mathbf{k},\mathbf{r}} + 1)$. Adding 1 to the rank ensures that the rank 0 can be encoded as either 1 or -1.

The number of steps in a sweep, denoted by $\mathfrak{n}'$, is $\mathfrak{n} - 1$ for ME, and $\mathfrak{n}$ for MM. The strategy mapping, once computed, can be reused for multiple runs of the algorithm, as long as the strategy kind, n (after bordering), and $\mathbf{s}$ do not change between the runs.

4.1.4 Algorithm initialization First, the CPU memory is allocated in each process, and the data is loaded (e.g., from a file), assuming $\mathbf{k} = 0$, i.e., the $\mathbf{r}$th process contains the $\mathbf{p}_{\mathbf{0},\mathbf{r}}$th and $\mathbf{q}_{\mathbf{0},\mathbf{r}}$th stripes of F, G, and Z.

Then, the device memory is allocated, if it is not already available, and an MPI barrier is reached. Timing of the algorithm includes everything that occurs from this barrier on, except the optional deallocation of the device memory.

The constant memory on each GPU is set up, and the stripes are copied to the device (global) memory, all of which could be done asynchronously. The involved stream(s) are then synchronized, depending on the way the copies have been performed.

It remains to be decided how many sweeps S in Algorithm 1 to allow. As with the pointwise level, there are two obvious choices: either some reasonably large number, e.g., 30 (as in FB), or 1 (as in BO). Now a variant of the multi-GPU algorithm is specified by the selected variant of the single-GPU algorithm, with the outermost strategy and the choice of S added; e.g., ZHZ0-(ME-BO, ME-FB-ME) for ME and $S = 1$, respectively, using ZHZ0-(ME-FB-ME) at the single-GPU level.

As shown in subsection 5.3.2, the imbalance of the computational time each GPU requires with FB (i.e., one GPU may need more sweeps in Algorithm 1 than another to reach convergence within an outermost step) is significantly detrimental to the overall performance—contrary to the single-GPU case (see subsection 3.5). Unlike there, where such imbalance between the thread blocks' sweep counts is offset by a large number of thread blocks to be scheduled on a small number of multiprocessors, here in the multi-GPU case there is a one-to-one correspondence between the number of tasks to perform and the number of execution units (GPUs) to perform them, so the time required for an outermost step depends on the slowest run of Algorithm 1 within it. Thus, BO is recommended here instead.

4.2 The main part of the algorithm

In the pre-iterative part of the algorithm, `initFGZ` kernel is called (see subsection 3.3), once in each process, in the chosen stream $\mathrm{S}_{\mathbf{r}}$. It is *not* called again in the context of Algorithm 1. Here, the row offset l in `initFGZ` is calculated according to the logical (not physical) index of a column, i.e., $l := j + \mathbf{p}_{\mathbf{0},\mathbf{r}} \cdot \mathbf{w}$ if $j < \mathbf{w}$, and $l := j - \mathbf{w} + \mathbf{q}_{\mathbf{0},\mathbf{r}} \cdot \mathbf{w}$ otherwise, with $\mathbf{p}_{\mathbf{0},\mathbf{r}}$ and $\mathbf{q}_{\mathbf{0},\mathbf{r}}$ sent to the kernel as parameters. The stripes of F_0, G_0, and Z_0 are then ready on each GPU (copying them to the CPU is not needed) for the iterative part of the algorithm.

4.2.1 Point-to-point communication and reductions Except for a single collective `MPI_Allreduce` operation required per an outermost sweep, all other communication in the algorithm is of the non-blocking, point-to-point kind, occurring in every outermost step. The communication parts of the algorithm, from a given process' perspective, are formalized in Algorithms 2, 3, 4, and 5, and put together in Algorithm 6.

The first guiding principle for such a design of the communication is to keep it as general as possible. Any process topology (including no topology in particular), suggested by the communication pattern of the chosen Jacobi strategy can be accommodated with equal ease.

The second principle is to facilitate hiding the communication overhead behind the GPU computation. Before a call of Algorithm 1 occurs within an outermost step of a given process, the non-blocking receives to the CPU stripes are started in anticipation of an early finish of the GPU work of the step in the processes that are to send their transformed stripes to the process in question. That way, while the given GPU still computes, its CPU can in theory start or even complete receiving one or both transformed stripes required in the following step. There remains an issue with several slowly progressing processes that might keep the rest of them idle, but at least the point-to-point data transfers can happen soon after the data is ready, not waiting for a massive data exchange with all processes communicating at the same time.

The third principle is to minimize the memory requirements of both the CPUs and the GPUs by sending the transformed data from the GPU RAM of one process to the CPU RAM of another two. That way, no separate, "shadow" GPU buffers are required to receive the data. The CPU stripes have to be present anyhow, to load the inputs and to collect the outputs, so they are reused as the communication buffers, with a penalty of the additional CPU-to-GPU data transfers after the main data exchange.

Matching a stripe to be sent from one process to a stripe that has to be received in another process is accomplished

Algorithm 2: The non-blocking receives in the **k**th step of the **r**th process.

```
tag := 1;  i := 0;      // tag tells which stripe from a sender has to be received
foreach o ∈ {0, 1} do      // o indexes the first or the second destination's stripe
    foreach Y ∈ {F_o^(r), G_o^(r), Z_o^(r)} do          // Y denotes the destination's host matrix
        foreach V ∈ {Re, Im} do          // V refers to the real or the imaginary part
            MPI_Irecv(V(Y), m_Y · w, MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, r[i]);
            tag := tag + 1;  i := i + 1;  // increment tag and i, which indexes requests
        end foreach
    end foreach
end foreach
```

Algorithm 3: The non-blocking sends in the **k**th step of the **r**th process.

```
// variable i is assumed to hold the last value assigned to it in Algorithm 2 in the k th step
foreach o ∈ {0, 1} do          // o indexes the first or the second source's stripe
    j := 1;                                                          // j is a base tag
    foreach Y ∈ {F_o^[r], G_o^[r], Z_o^[r]} do              // Y denotes the source's device matrix
        foreach V ∈ {Re, Im} do          // V refers to the real or the imaginary part
            // tag: base + offset 0 or 6 (first or second stripe at destination)
            tag := j + (sign(t_{k,r}^{o}) + 1) · 3;
            // send to destination t_{k,r}^{o}
            MPI_Isend(V(Y), m_Y · w, MPI_DOUBLE, t_{k,r}^{o}, tag, MPI_COMM_WORLD, r[i]);
            j := j + 1;  i := i + 1;                                 // increment j and i
        end foreach
    end foreach
end foreach
```

Algorithm 4: Completion of the communication and the host-to-device transfers in the **k**th step of the **r**th process.

```
// variable i is assumed to hold the last value assigned to it in Algorithm 3 in the k th step
MPI_Waitall(i, r, statuses);     // wait for all pending MPI requests to complete
foreach (W, Y) ∈ {(F^(r), F^[r]), (G^(r), G^[r]), (Z^(r), Z^[r])} do // (W, Y) are (host,device) matrices
    foreach V ∈ {Re, Im} do          // V refers to the real or the imaginary part
        copy V(W) to V(Y) using cudaMemcpy2DAsync;     // in the appropriate stream(s)
    end foreach
end foreach
synchronize the stream(s) used for copying and call MPI_Barrier(MPI_COMM_WORLD);
```

Algorithm 5: Convergence criterion checking in the **c**th sweep of the **r**th process.

```
MPI_Allreduce({S̃_c, B̃_c}, {Σ_s S̃_c, Σ_s B̃_c}, 2, MPI_UNSIGNED_LONG, MPI_SUM, MPI_COMM_WORLD);
// Is the sum of all per-process, per-sweep big transformation counters 0?
if Σ_s B̃_c = 0 then break;
```

by MPI tags annotating the messages. In the complex case there are twelve stripes in total (six in the real case, without the imaginary stripes) to be received by a process in a single outermost step (see Algorithm 2 for their tag numbers).

When a message comes to a process, from any sender, it is only accepted if it bears a valid tag (between 1 and 12, inclusive) and the message data is stored in the corresponding stripe, as in Algorithm 2. Likewise, when a stripe has to be sent, the strategy mapping is consulted

to get the destination process' rank, and decide if the stripe should become the first or the second one at the destination. According to that information, the message's tag is calculated as in Algorithm 3.

4.2.2 The CPU part of the algorithm The pre-iterative, iterative, and post-iterative parts of the algorithm are shown in Algorithm 6. The final full rescaling with the extraction of the generalized singular values happens only once (i.e., not in the context of Algorithm 1). As the convergence criterion relies on sum-reducing the per-sweep counters of the big transformations applied in all processes, an implicit synchronization point at the end of a sweep is introduced.

5 Numerical testing

The purpose of the numerical testing of the single-GPU and the multi-GPU algorithms is twofold. First, it has been meant to compare the variants of the algorithms in terms of performance and accuracy and discover which (if any) variant stands out as the best one in either aspect. Second, it should inform the potential users about the algorithms' behavior on two sets of realistic, small and medium-to-large sized problems.

By performance it is meant the wall execution time. Counting FLOPS (floating-point operations per second) rate makes less sense here than in the algorithms (such as the matrix multiplication) that solely depend on a subset of the arithmetic operations of a similar execution complexity, such as additions, subtractions, multiplications, and FMAs. Instead, the algorithms presented here necessarily involve a substantial amount of divisions and (reciprocal) square roots. Moreover, the majority of performance gains compared to a simple, pointwise algorithm come from a careful usage of the fast shared memory and the GPU registers, as it is also shown in Novaković (2015), and not from tweaking the arithmetic intensity. The wall time should therefore be more informative than FLOPS about the expected behavior of the algorithm on present-day hardware, and about the differences in the algorithm's variants, since the future performance is very hard to predict without a complex model that takes into account all levels of the memory hierarchy, not only the arithmetic operations and the amount of parallelism available.

Accuracy of the algorithm can be assessed in several ways. In both the real and the complex case the relative normwise errors of the decompositions of F and G,

$$\|F - U\Sigma_F X\|_F / \|F\|_F, \quad \|G - V\Sigma_G X\|_F / \|G\|_F,$$

were computed the same way as in Singer et al. (2020). Namely, X had to be explicitly obtained as Z^{-1} by solving the linear system $ZX = I$. First, the LU factorization of Z with complete pivoting was performed by the LAPACK routine `DGETC2` (or `ZGETC2`), followed by the system solving using the routine `DGESC2` (or `ZGESC2`).

The ensuing matrix multiplications and the Frobenius norm computations were using Intel 80-bit hardware-supported extended precision (`REAL(KIND=10)` in GNU Fortran), to reduce the effects of the rounding errors on the final result while avoiding the expensive, emulated quadruple (128-bit) precision.

The numerical orthogonality of the left generalized singular vectors U and V was computed in the extended precision as $\|U^*U - I\|_F$ and $\|V^*V - I\|_F$, respectively.

When the (almost) exact generalized singular values Σ are known in advance, as is the case with the small real dataset (see subsection 5.1.2), the maximal relative error in the computed $\widehat{\Sigma}$ can be obtained as

$$\max_{1 \le i \le n} |(\sigma_i - \hat{\sigma}_i)/\sigma_i|.$$

5.1 Testing environment and data

The testing environment was the same for all tests, as described in subsection 5.1.1. Apart from the GPU compute architecture 7.0, some tests have been repeated on a Maxwell GPU (GeForce GTX TITAN X, architecture 5.2) and a Kepler GPU (GeForce GT 730, architecture 3.5), to verify the portability of the code and the numerical reproducibility of the results. Also, a few sample runs of the multi-GPU algorithm on a small matrix have been tried on a combination of those two GPUs, with the code built for both architectures, to ensure that the algorithm functions correctly in such a heterogeneous environment.

The testing data is synthetic (not from any application domain) and is described in subsection 5.1.2. Please see the supplementary material for its availability.

5.1.1 Testing environment The testing environment comprises two Intel Xeon Silver 4114 CPUs, 384 GiB of RAM, and four NVIDIA Tesla V100-SXM2-16GB (Volta) GPUs per node, with a 64-bit Linux (CentOS 7.5.1804), the GCC 4.8.5 C++ compiler, CUDA 10.0, and a build of Open MPI 3.0.0 distribution with the CUDA support.

5.1.2 Testing data Two datasets have been generated: a "small" and a "large" one, with their names referring to the orders of the square matrices forming the pairs contained in them. The small dataset contains both the real and the complex matrix pairs, with each matrix stored in (and then read from) its unformatted binary file, while the large dataset contains only the complex matrix pairs.

The small dataset has 19 matrix pairs for each datatype, with the orders of the matrices ranging from 512 to 9728 in steps of 512. The large dataset has 3 matrix pairs, with the orders of the matrices being $18 \cdot 1024 = 18432$, $24 \cdot 1024 = 24576$, and $36 \cdot 1024 = 36864$, so that the GPU RAM requirements do not exceed the memory provided by

Algorithm 6: The CPU part of the multi-GPU implicit Hari–Zimmermann algorithm (for the $\mathbf{r}$th process).

```
initFGZ(p_{0,r}, q_{0,r}); // compute F_0^[r], G_0^[r], Z_0^[r] in the stream s_r
for 0 ≤ c < 30 do // outermost sweep c
    {Ŝ_c, B̂_c} := {0, 0}; // reset the per-process, per-sweep transformation counters
    for 0 ≤ k < n' do // outermost step k
        start receiving into F^(r), G^(r), Z^(r) as in Algorithm 2;
        call the single-GPU Algorithm 1 with s = s_r on F^[r], G^[r], Z^[r] with the chosen S;
        // increment the transformation counters by those from Algorithm 1
        Ŝ_c := Ŝ_c + S̃;   B̂_c := B̂_c + B̃;
        start sending the transformed F^[r], G^[r], Z^[r] as in Algorithm 3;
        complete the communication and copy the received F^(r), G^(r), Z^(r) to F^[r], G^[r], Z^[r], as in Algorithm 4;
    end for
    reduce the transformation counters across the communicator;
    break if the convergence has been reached, as in Algorithm 5;
end for
rescale(true); cudaStreamSynchronize(s_r); // full rescaling of Z^[r] in s_r
optionally, copy F^[r], G^[r], Z^[r], Σ_F^[r], Σ_G^[r], Σ^[r] to F^(r), G^(r), Z^(r), Σ_F^(r), Σ_G^(r), Σ^(r) and synchronize the stream(s);
MPI_Barrier(MPI_COMM_WORLD); // completion of the algorithm and its timing
```

one, two, and four GPUs, respectively. No matrices in either dataset require bordering.

The real matrix pairs in the small dataset were generated in quadruple datatype (`REAL(KIND=16)` in Intel Fortran) and rounded to double precision datatype. The same test generation method was employed as in Novaković et al. (2015). The required BLAS and LAPACK routines had been adapted as required. The core of the generation are two quadruple-adapted LAPACK testing routines: `xLAGSY`, that generates a pseudorandom symmetric matrix, here of the full bandwidth, from a given diagonal prescribing the eigenvalues of the matrix; and `xLAROR`, that here multiplies a given matrix from the left by a pseudorandom orthogonal matrix. The diagonals of Σ_F, Σ_G, and Λ_X were generated by `DLARND`, a standard LAPACK's pseudorandom number generator, here with the uniform probability distribution on $(0,1)$, such that only those values returned by it that had been greater than 10^{-10} were accepted. Then, $U\Sigma_F$ was generated from Σ_F, and $V\Sigma_G$ from Σ_G, both using `xLAROR`, while X was obtained from Λ_X using `xLAGSY`. After that, $F := U\Sigma_F \cdot X$ and $G := V\Sigma_G \cdot X$ (in quadruple). The generator finishes with a call to the preprocessing part, `DGGSVP3`, of the LAPACK's GSVD method (see Anderson et al. (1999) and the routine's comments), to make the data usable for comparison with `DTGSJA` Kogbetliantz-type GSVD routine from LAPACK. On the small dataset the relative errors in the generalized singular values computed on the CPU by LAPACK and on a single GPU by the proposed algorithm were compared.

The complex matrix pairs in both datasets were generated by a much simpler procedure, described in Singer et al. (2020). Namely, each matrix in a pair was generated by a call to `ZLATMS` LAPACK testing routine as Hermitian and positive definite, with its pseudorandom eigenvalues uniformly distributed in $(0,1)$.

5.2 Results with the single-GPU algorithm

When presenting the performance of several variants, a decision has been made to show the execution time plots and tables for the fastest variant on the largest matrix pair in the small dataset, separately in the real and in the complex case. To compare those results with the ones from other variants, a useful measure is the relative slowdown on a given matrix order. Fixing the reference variant r, as suggested by the results in the plots, for another variant v and a matrix order n let $T_n^{(r)}$ and $T_n^{(v)}$ be the execution times of r and v on a matrix pairs with the matrices of order n, respectively. The relative slowdown $S_n^{(v:r)}$ of v compared to r on n, given in percentages of the execution time of r, is

$$S_n^{(v:r)} := \frac{T_n^{(v)} - T_n^{(r)}}{T_n^{(r)}} \cdot 100.$$

The average relative slowdown across the entire dataset (with 19 matrix pairs) is given as $S_{\text{avg}}^{(v:r)} := \sum_n S_n^{(v:r)}/19$.

5.2.1 Performance in the real case In the left subfigure of Figure 7 the wall execution time of four subvariants of DHZ0 (the fastest of eight variants from Table 1 on the largest matrix pair) and the number of outer sweeps for two fastest subvariants on the small real dataset are shown. Table 21 in Appendix E contains the wall time for

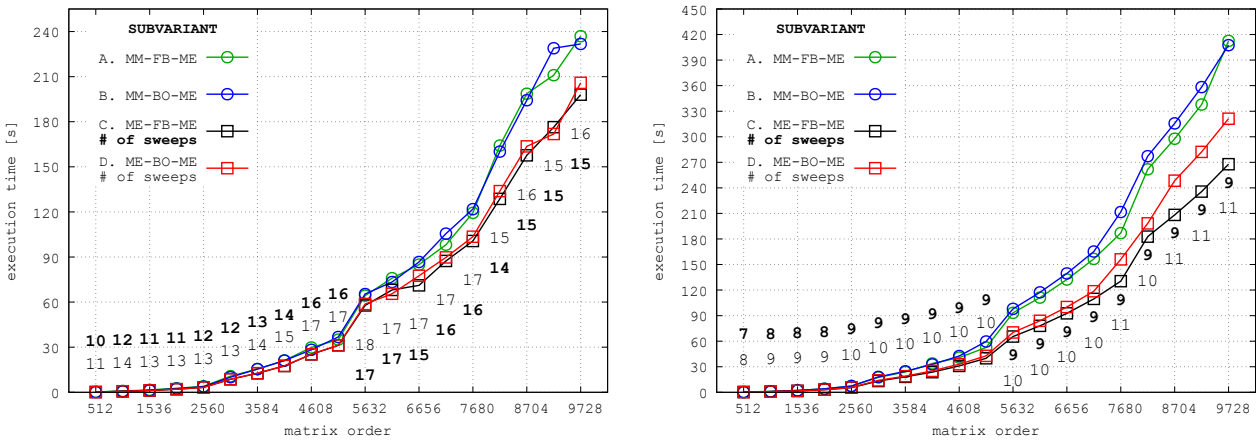

Figure 7. The wall execution time of four subvariants of DHZ0 (left) and ZHZ4 (right) and the number of outer sweeps for two fastest subvariants on the small real (left) and complex (right) datasets.

DHZ0-(ME-FB-ME), which is generally the fastest of the four subvariants.

In Table 2 the intervals of relative slowdown of other real single-GPU (ME-FB-ME) variants compared to DHZ0-(ME-FB-ME) (the reference variant) are given.

Table 2. The intervals of relative slowdown of other real single-GPU variants compared to DHZ0, all of (ME-FB-ME) subvariant. A negative slowdown is a speedup.

ID	maximal relative slowdown [%]	minimal relative slowdown [%]	average relative slowdown [%]
1	11.107853	0.984035	3.093490
2	17.137007	1.722547	4.870879
3	32.907217	3.144621	9.204879
4	0.328320	-0.678588	0.139650
5	11.353652	1.395604	3.373963
6	17.962363	2.065662	5.199807
7	32.290521	3.511386	9.413653

5.2.2 Performance in the complex case In the right subfigure of Figure 7 the wall execution time of four subvariants of ZHZ4 (the fastest of eight variants from Table 1 on the largest matrix pair) and the number of outer sweeps for two fastest subvariants on the small complex dataset are shown.

In Table 3 the intervals of relative slowdown of other complex single-GPU (ME-FB-ME) variants compared to ZHZ4-(ME-FB-ME) (the reference variant) are given. Since ZHZ0-(ME-FB-ME) is the fastest variant *on average*, differing from ZHZ4 only subtly (in the convergence criterion), ZHZ0 is used instead of ZHZ4 in the multi-GPU algorithm and in subsection 5.2.6. Table 22 in Appendix F shows the wall time for ZHZ0-(ME-FB-ME).

Table 3. The intervals of relative slowdown of other complex single-GPU variants compared to ZHZ4, all of (ME-FB-ME) subvariant. A negative slowdown is a speedup; e.g., ZHZ0 is faster than ZHZ4 on average, even if it is slower sometimes.

ID	maximal relative slowdown [%]	minimal relative slowdown [%]	average relative slowdown [%]
0	0.488301	-6.403801	-0.493666
1	2.309416	-0.241201	0.841493
2	7.135216	-1.206508	1.756809
3	12.632933	0.866139	3.167284
5	1.366689	-0.442658	0.312471
6	8.423992	0.720393	2.287251
7	11.257148	1.092003	3.240513

5.2.3 Detailed timings of the main kernel's subphases Tables 4 and 5 show the percentages of time each subphase of the main kernel takes on a Volta GPU for the selected subvariants of the real and the complex single-GPU algorithm, respectively. As the problem size increases, the first and the fourth subphases (i.e., the matrix multiplications) start to dominate over the others. With a suitable modification for the recent GPU architectures like Ampere, the Hari–Zimmermann algorithm could therefore benefit from their dedicated hardware and instructions for speeding up the GEMM-like operations.

5.2.4 Intensities of the floating-point arithmetic operations Tables 6 and 7 contain the relative intensity of floating-point operations with rounding across all invocations of all kernels (not only the main one, though other kernels' contributions are negligible) on a Maxwell GPU. The columns names correspond to the double precision CUDA arithmetic intrinsics, while ◇ represents the calls

Table 4. Percentage of time (rounded to the nearest per mil) spent in the subphases **1** to **4** in all invocations of the main kernel for the subvariants $\mathtt{C}_0^{\mathbb{R}}$, i.e., DHZ0-(ME-FB-ME) and $\mathtt{D}_0^{\mathbb{R}}$, i.e., DHZ0-(ME-BO-ME), on the small real dataset.

n	subphases of $\mathtt{C}_0^{\mathbb{R}}$ [%]				subphases of $\mathtt{D}_0^{\mathbb{R}}$ [%]			
	1	**2**	**3**	**4**	**1**	**2**	**3**	**4**
512	12.3	10.0	53.8	23.9	20.9	17.0	21.3	40.7
1024	17.5	7.1	41.1	34.4	25.7	10.4	13.1	50.8
1536	20.6	5.3	34.1	40.0	28.2	7.3	9.3	55.2
2048	22.8	4.4	28.2	44.6	29.4	5.7	7.3	57.6
2560	24.9	3.7	23.3	48.0	30.6	4.6	5.9	59.0
3072	26.1	3.3	20.5	50.1	31.1	3.9	5.0	59.9
3584	26.5	2.9	18.7	51.8	31.2	3.4	4.4	61.0
4096	27.3	2.6	17.1	53.0	31.6	3.0	3.9	61.4
4608	27.8	2.4	15.4	54.4	31.7	2.7	3.5	62.1
5120	28.2	2.2	14.5	55.2	32.0	2.5	3.2	62.4
5632	28.8	2.0	13.2	56.0	32.2	2.2	2.9	62.7
6144	29.2	1.9	12.0	56.9	32.2	2.1	2.7	63.0
6656	29.4	1.8	11.7	57.2	32.4	1.9	2.5	63.2
7168	29.7	1.7	10.8	57.9	32.5	1.8	2.3	63.4
7680	30.0	1.7	10.1	58.3	32.5	1.8	2.1	63.6
8192	29.5	1.6	9.9	59.0	32.0	1.6	2.1	64.3
8704	30.3	1.5	8.9	59.2	32.6	1.7	1.9	63.8
9216	30.4	1.4	8.7	59.5	32.7	1.5	1.9	64.0
9728	30.6	1.4	8.2	59.9	32.7	1.5	1.7	64.1

Table 5. Percentage of time (rounded to the nearest per mil) spent in the subphases **1** to **4** in all invocations of the main kernel for the subvariants $\mathtt{C}_0^{\mathbb{C}}$, i.e., ZHZ0-(ME-FB-ME) and $\mathtt{D}_0^{\mathbb{C}}$, i.e., ZHZ0-(ME-BO-ME), on the small complex dataset.

n	subphases of $\mathtt{C}_0^{\mathbb{C}}$ [%]				subphases of $\mathtt{D}_0^{\mathbb{C}}$ [%]			
	1	**2**	**3**	**4**	**1**	**2**	**3**	**4**
512	13.9	6.5	51.6	28.1	23.2	10.9	18.7	47.2
1024	19.6	4.7	35.8	39.9	27.2	6.5	10.8	55.5
1536	21.9	3.5	29.3	45.4	28.5	4.6	7.7	59.3
2048	24.7	2.8	23.3	49.2	30.2	3.5	6.0	60.3
2560	27.3	2.4	19.2	51.1	32.1	2.8	4.6	60.5
3072	28.4	2.1	16.3	53.2	32.5	2.4	3.9	61.2
3584	29.1	1.9	14.3	54.7	32.8	2.1	3.4	61.8
4096	29.5	1.7	12.9	55.9	32.9	1.8	3.0	62.3
4608	29.9	1.5	12.0	56.5	33.0	1.7	2.7	62.6
5120	30.1	1.4	11.2	57.3	33.1	1.5	2.4	63.0
5632	30.6	1.3	10.5	57.5	33.5	1.4	2.2	62.9
6144	30.8	1.2	9.7	58.3	33.4	1.3	2.1	63.2
6656	31.1	1.1	9.2	58.6	33.6	1.2	1.9	63.3
7168	31.3	1.1	8.5	59.2	33.6	1.1	1.8	63.5
7680	31.3	1.0	8.0	59.6	33.5	1.0	1.6	63.8
8192	32.2	1.0	7.4	59.4	34.2	1.0	1.6	63.3
8704	31.7	0.9	7.3	60.1	33.7	1.0	1.5	63.9
9216	31.7	0.9	7.1	60.3	33.7	0.9	1.4	64.0
9728	32.0	0.8	6.7	60.5	33.8	0.9	1.3	64.0

to `hypot` and `rsqrt_rn` functions (their constituent operations were not counted separately). The results strongly correlate with those from Tables 4 and 5; namely, the intensity of `fma` increases as the matrix multiplications take the larger portions of the overall time. For the smaller inputs the combined amount of divisions/reciprocals, square roots, and the $\diamond$ function calls is in single percents, but according to (Arafa et al. 2019, Table IV) even such an amount has a considerable influence on the execution time; e.g., a division has on average close to 20 times the latency of a simple instruction (like `fma`) on the Volta GPUs.

5.2.5 Accuracy in the real case In the left subfigure of Figure 8 the normwise relative errors of four subvariants of DHZ0 are shown in a logarithmic scale, while in Table 8 the spectral condition numbers $\kappa_2(F)$ in the small real dataset are given, offering a justification for the non-monotonicity of the error graphs. In Table 9 the spectral condition numbers $\kappa_2(G)$ in the small real dataset are given, without a corresponding error figure, which is almost indistinguishable from the left subfigure of Figure 8.

Figure 9 shows the numerical orthogonality of the left generalized singular vectors U (left subfigure) and V (right subfigure) across the small real dataset achieved by DHZ0.

Tables 10 and 11 show the maximal relative errors in the generalized singular values computed by DHZ0-(ME-FB-ME) and `DTGSJA` from the Intel Math Kernel Library

Table 6. Percentage of the total number of the floating-point operations with rounding performed in the invocations of all kernels (rounded to the nearest per myriad) for the subvariant DHZ0-(ME-FB-ME) on the small real dataset.

n	$\diamond$	`add`	`sub`	`mul`	`fma`	`div`	`rcp`	`sqrt`
512	0.83	22.68	1.32	16.33	54.20	1.71	0.57	2.35
1024	0.55	15.86	0.91	11.18	68.29	1.17	0.39	1.64
1536	0.43	12.34	0.71	8.70	75.33	0.91	0.30	1.27
2048	0.34	9.99	0.57	7.01	80.06	0.74	0.25	1.03
2560	0.27	8.00	0.45	5.53	84.16	0.58	0.19	0.82
3072	0.23	6.86	0.38	4.72	86.44	0.49	0.16	0.71
3584	0.20	6.16	0.35	4.25	87.80	0.45	0.15	0.63
4096	0.17	5.43	0.31	3.71	89.31	0.39	0.13	0.56
4608	0.16	4.89	0.28	3.34	90.36	0.35	0.12	0.50
5120	0.14	4.51	0.26	3.09	91.10	0.33	0.11	0.46
5632	0.12	3.94	0.22	2.67	92.27	0.28	0.09	0.40
6144	0.11	3.62	0.20	2.45	92.89	0.26	0.09	0.37
6656	0.11	3.52	0.20	2.41	93.06	0.26	0.09	0.36
7168	0.10	3.13	0.18	2.13	93.85	0.22	0.07	0.32
7680	0.09	2.91	0.16	1.96	94.30	0.21	0.07	0.30
8192	0.09	2.85	0.16	1.94	94.39	0.21	0.07	0.29
8704	0.08	2.59	0.15	1.75	94.92	0.18	0.06	0.27
9216	0.08	2.48	0.14	1.68	95.13	0.18	0.06	0.25
9728	0.07	2.34	0.13	1.58	95.41	0.17	0.06	0.24

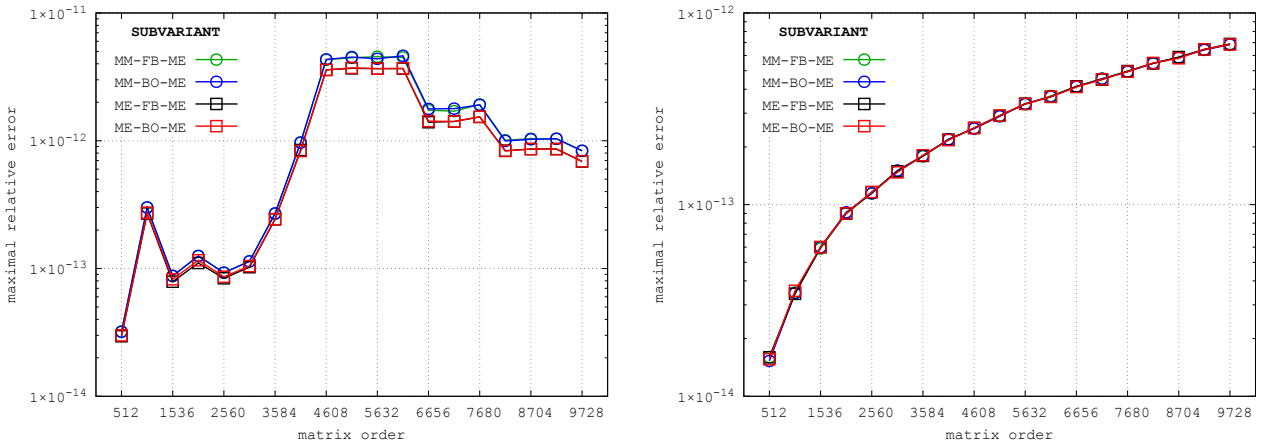

Figure 8. The relative normwise errors, $\|F - U\Sigma_F X\|_F / \|F\|_F$, of four subvariants of DHZ0 (left) and ZHZ0 (right) on the small real (left) and complex (right) datasets.

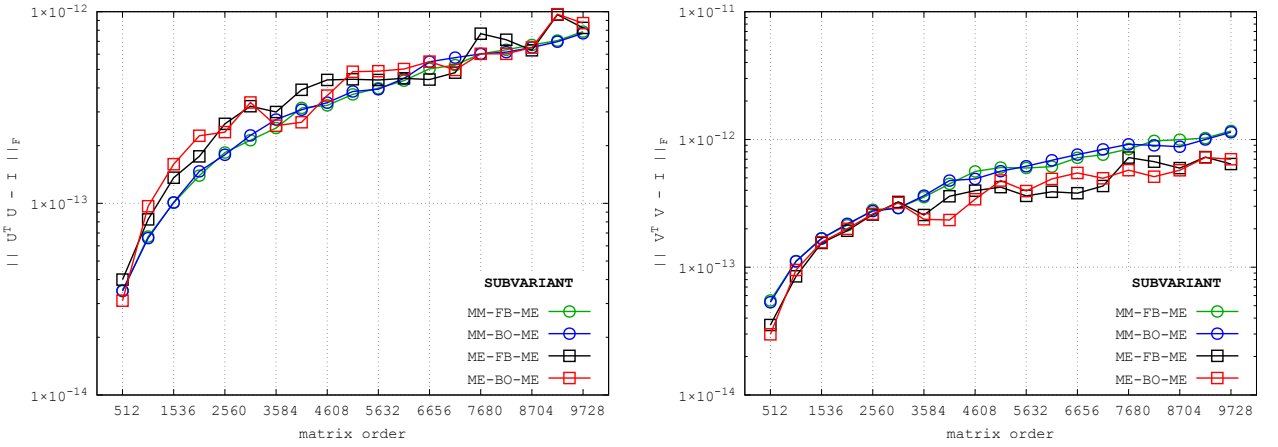

Figure 9. The numerical orthogonality $\|U^T U - I\|_F$ (left) and $\|V^T V - I\|_F$ (right) of the left generalized singular vectors U and V, respectively, achieved by four subvariants of DHZ0 on the small real dataset.

(version 2020.1.217 on an Intel Xeon Phi 7210 CPU), respectively, on the small real dataset.

The average relative errors in the computed generalized singular values in both the GPU and the CPU case are two to three orders of magnitude smaller than the maximal ones, as can be seen for the former in the supplementary material. By comparing Tables 10 and 11 it can be concluded that, in this sense and instance, the proposed GPU algorithm exhibits accuracy similar to the LAPACK's `DTGSJA`.

5.2.6 Accuracy in the complex case In the right subfigure of Figure 8 the normwise relative errors of four subvariants of ZHZ0 are shown in a logarithmic scale, while in Table 12 the spectral condition numbers $\kappa_2(F)$ in the small complex dataset are presented, offering a justification for a smooth shape of the error graphs. In Table 13 the spectral condition numbers $\kappa_2(G)$ in the small complex dataset are given. A corresponding error figure would be almost identical to the right subfigure of Figure 8.

Figure 10 shows the numerical orthogonality of the left generalized singular vectors U (left subfigure) and V (right subfigure) on the small complex dataset achieved by ZHZ0.

5.2.7 Conclusions From Figure 7 it is obvious that MM gives a significantly slower execution than ME at the outer level, and that FB is slightly faster than BO. From Tables 2 and 3 it is clear that the execution times across the variants do not widely differ, and that the enhanced dot-products from Appendix A add from a few percent to something more than 15% to the wall time.

In both the real and the complex case the variant 0 with (ME-FB-ME) is a reasonable choice performance-wise.

Regarding the normwise relative errors on the matrices of moderate spectral conditions, as it is in the real case,

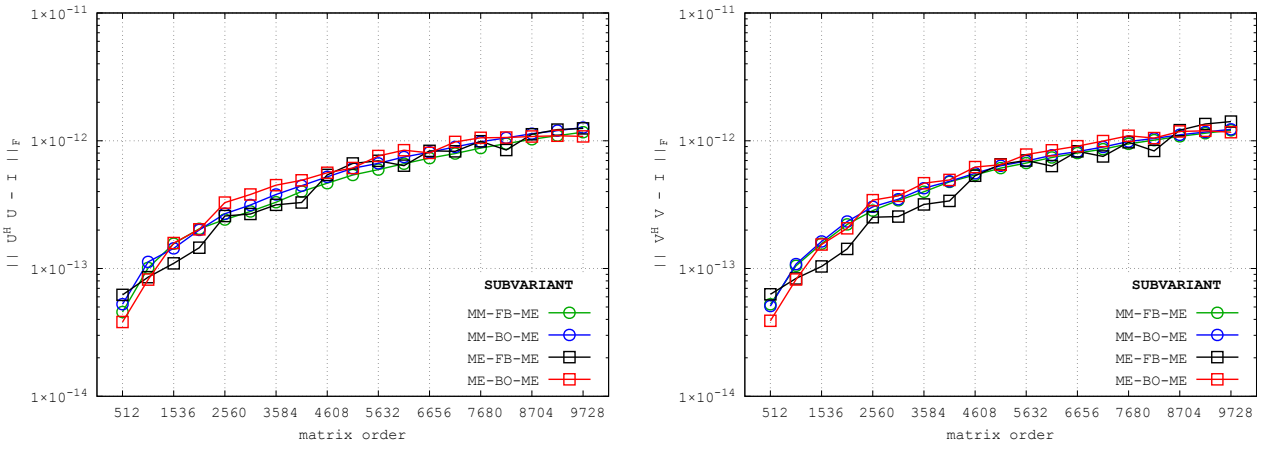

Figure 10. The numerical orthogonality $\|U^*U - I\|_F$ (left) and $\|V^*V - I\|_F$ (right) of the left generalized singular vectors U and V, respectively, achieved by four subvariants of ZHZ0 on the small complex dataset.

Table 7. Percentage of the total number of the floating-point operations with rounding performed in the invocations of all kernels (rounded to the nearest per myriad) for the subvariant ZHZ0-(ME-FB-ME) on the small complex dataset.

n	◇	add	sub	mul	fma	div	rcp	sqrt
512	0.92	12.81	0.42	11.63	71.86	0.42	0.83	1.12
1024	0.53	7.46	0.24	6.63	83.78	0.24	0.47	0.65
1536	0.39	5.42	0.17	4.83	88.21	0.17	0.34	0.47
2048	0.30	4.16	0.13	3.68	90.98	0.13	0.26	0.36
2560	0.23	3.27	0.10	2.84	92.97	0.10	0.20	0.28
3072	0.19	2.70	0.08	2.33	94.21	0.08	0.16	0.23
3584	0.17	2.34	0.07	2.02	94.98	0.07	0.14	0.20
4096	0.15	2.03	0.06	1.74	95.66	0.06	0.12	0.17
4608	0.13	1.88	0.06	1.61	95.99	0.06	0.11	0.16
5120	0.12	1.71	0.05	1.46	96.35	0.05	0.10	0.15
5632	0.11	1.55	0.05	1.32	96.70	0.05	0.09	0.13
6144	0.10	1.40	0.04	1.19	97.02	0.04	0.08	0.12
6656	0.09	1.32	0.04	1.12	97.20	0.04	0.08	0.11
7168	0.09	1.20	0.04	1.03	97.44	0.04	0.07	0.10
7680	0.08	1.13	0.03	0.96	97.60	0.03	0.07	0.10
8192	0.07	1.03	0.03	0.88	97.81	0.03	0.06	0.09
8704	0.07	1.01	0.03	0.86	97.86	0.03	0.06	0.09
9216	0.07	0.96	0.03	0.81	97.96	0.03	0.06	0.08
9728	0.07	0.91	0.03	0.78	98.06	0.03	0.05	0.08

Table 8. The spectral condition numbers $\kappa_2(F)$ in the small real dataset.

n	$\kappa_2(F)$	n	$\kappa_2(F)$
512	$7.19081 \cdot 10^4$	5632	$2.33003 \cdot 10^6$
1024	$1.29837 \cdot 10^6$	6144	$3.69195 \cdot 10^6$
1536	$2.35302 \cdot 10^5$	6656	$1.90070 \cdot 10^6$
2048	$1.48022 \cdot 10^5$	7168	$7.72806 \cdot 10^5$
2560	$1.69855 \cdot 10^5$	7680	$5.33036 \cdot 10^5$
3072	$7.28415 \cdot 10^4$	8192	$3.59719 \cdot 10^5$
3584	$2.78307 \cdot 10^5$	8704	$3.54038 \cdot 10^5$
4096	$1.43141 \cdot 10^6$	9216	$1.68270 \cdot 10^6$
4608	$1.35132 \cdot 10^6$	9728	$3.90607 \cdot 10^5$
5120	$2.33209 \cdot 10^6$	–	–

Table 9. The spectral condition numbers $\kappa_2(G)$ in the small real dataset.

n	$\kappa_2(G)$	n	$\kappa_2(G)$
512	$7.79269 \cdot 10^3$	5632	$1.94774 \cdot 10^6$
1024	$5.17836 \cdot 10^4$	6144	$1.14571 \cdot 10^7$
1536	$2.41272 \cdot 10^4$	6656	$3.21194 \cdot 10^7$
2048	$1.91692 \cdot 10^4$	7168	$6.84297 \cdot 10^6$
2560	$3.30063 \cdot 10^4$	7680	$2.24983 \cdot 10^7$
3072	$1.90896 \cdot 10^4$	8192	$2.89085 \cdot 10^6$
3584	$6.56610 \cdot 10^4$	8704	$1.20004 \cdot 10^6$
4096	$2.08194 \cdot 10^5$	9216	$3.35081 \cdot 10^6$
4608	$8.83907 \cdot 10^5$	9728	$3.58109 \cdot 10^6$
5120	$1.53531 \cdot 10^6$	–	–

the MM subvariants are almost indistinguishable, as are the ME subvariants, with the latter being slightly more accurate than the former, as shown in the left subfigure of Figure 8. With the matrices of small spectral condition, as it is in the complex case, all subvariants are almost indistinguishable, as shown in the right subfigure of Figure 8. The spectral condition numbers were computed by Matlab R2019a.

In Table 14 the maximal relative normwise errors with respect to F and G in the real and the complex case on the small dataset for all variants of the single-GPU algorithm with (ME-FB-ME) are given. Looking at the minimal value in each data column, it is evident that, except for G in the real case, the enhanced dot-products offer a small advantage

Table 10. The relative errors in $\widehat{\Sigma}$, the generalized singular values computed by DHZ0-(ME-FB-ME) on the small real dataset.

n	$\max_i \lvert(\sigma_i - \hat{\sigma}_i)/\sigma_i\rvert$	n	$\max_i \lvert(\sigma_i - \hat{\sigma}_i)/\sigma_i\rvert$
512	$4.20077 \cdot 10^{-13}$	5632	$6.66337 \cdot 10^{-11}$
1024	$2.00541 \cdot 10^{-11}$	6144	$6.85154 \cdot 10^{-11}$
1536	$7.29365 \cdot 10^{-12}$	6656	$3.39017 \cdot 10^{-10}$
2048	$8.80861 \cdot 10^{-13}$	7168	$5.65600 \cdot 10^{-11}$
2560	$1.08576 \cdot 10^{-12}$	7680	$8.18645 \cdot 10^{-11}$
3072	$1.12990 \cdot 10^{-12}$	8192	$1.51962 \cdot 10^{-11}$
3584	$6.02566 \cdot 10^{-12}$	8704	$8.99801 \cdot 10^{-12}$
4096	$1.64618 \cdot 10^{-11}$	9216	$1.81947 \cdot 10^{-11}$
4608	$2.01553 \cdot 10^{-11}$	9728	$4.48431 \cdot 10^{-11}$
5120	$4.85422 \cdot 10^{-11}$	–	–

Table 11. The relative errors in $\widetilde{\Sigma}$, the generalized singular values computed by LAPACK's DTGSJA on the small real dataset.

n	$\max_i \lvert(\sigma_i - \tilde{\sigma}_i)/\sigma_i\rvert$	n	$\max_i \lvert(\sigma_i - \tilde{\sigma}_i)/\sigma_i\rvert$
512	$4.24173 \cdot 10^{-13}$	5632	$6.65480 \cdot 10^{-11}$
1024	$2.00948 \cdot 10^{-11}$	6144	$6.85305 \cdot 10^{-11}$
1536	$7.28953 \cdot 10^{-12}$	6656	$3.38985 \cdot 10^{-10}$
2048	$9.06332 \cdot 10^{-13}$	7168	$5.65602 \cdot 10^{-11}$
2560	$1.04699 \cdot 10^{-12}$	7680	$8.18993 \cdot 10^{-11}$
3072	$1.10638 \cdot 10^{-12}$	8192	$1.51537 \cdot 10^{-11}$
3584	$6.00819 \cdot 10^{-12}$	8704	$9.00031 \cdot 10^{-12}$
4096	$1.64629 \cdot 10^{-11}$	9216	$1.81785 \cdot 10^{-11}$
4608	$2.03583 \cdot 10^{-11}$	9728	$4.48437 \cdot 10^{-11}$
5120	$4.84123 \cdot 10^{-11}$	–	–

Table 12. The spectral condition numbers $\kappa_2(F)$ in the small complex dataset.

n	$\kappa_2(F)$	n	$\kappa_2(F)$
512	$2.30011 \cdot 10^3$	5632	$8.81464 \cdot 10^3$
1024	$2.30011 \cdot 10^3$	6144	$8.81464 \cdot 10^3$
1536	$2.30083 \cdot 10^3$	6656	$8.81464 \cdot 10^3$
2048	$2.30083 \cdot 10^3$	7168	$8.81464 \cdot 10^3$
2560	$4.92822 \cdot 10^3$	7680	$8.81464 \cdot 10^3$
3072	$4.92822 \cdot 10^3$	8192	$8.81464 \cdot 10^3$
3584	$4.92822 \cdot 10^3$	8704	$8.81464 \cdot 10^3$
4096	$4.92822 \cdot 10^3$	9216	$8.81464 \cdot 10^3$
4608	$7.69922 \cdot 10^3$	9728	$8.81464 \cdot 10^3$
5120	$8.81464 \cdot 10^3$	–	–

in accuracy, but not so significant that it would not be offset by a drop in performance when the focus is on the latter.

Despite the low occupancy, Table 15 shows that all warps that can occupy a multiprocessor execute with almost full efficiency. It is therefore expected that, should the present bottleneck on multiprocessors be removed in the future by increasing the register file (see subsection 3.5) as it has recently been done with the shared memory, the occupancy might increase as well, and with it the overall performance, while the efficiency would stay at the same high level.

Table 13. The spectral condition numbers $\kappa_2(G)$ in the small complex dataset.

n	$\kappa_2(G)$	n	$\kappa_2(G)$
512	$5.62075 \cdot 10^2$	5632	$7.79104 \cdot 10^3$
1024	$1.54312 \cdot 10^3$	6144	$4.69173 \cdot 10^3$
1536	$8.58280 \cdot 10^2$	6656	$1.79255 \cdot 10^4$
2048	$1.69892 \cdot 10^3$	7168	$7.57668 \cdot 10^3$
2560	$2.66973 \cdot 10^5$	7680	$1.12171 \cdot 10^5$
3072	$4.12692 \cdot 10^4$	8192	$1.38621 \cdot 10^4$
3584	$9.24332 \cdot 10^3$	8704	$9.11971 \cdot 10^3$
4096	$1.67599 \cdot 10^4$	9216	$8.54830 \cdot 10^3$
4608	$6.50253 \cdot 10^3$	9728	$5.97562 \cdot 10^3$
5120	$8.40210 \cdot 10^3$	–	–

5.3 Results with the multi-GPU algorithm

Due to a limited availability of GPUs in the testing environment, only the complex case in the variant 0 was tested on the large dataset and compared with a single-GPU baseline.

Here, the full accuracy testing was skipped, due to the huge computational demands of the matrix inversions and of the error calculation in extended precision. For $n = 18432$, the relative errors in the corresponding outputs with one and with two GPUs (in both cases in all variants that were timed) were compared and all of them, for both F and G, were found to be less than $1.7 \cdot 10^{-12}$. Also, for a given matrix, the relative errors in all cases differed less than 10^{-13}, indicating that the multi-GPU algorithm did not introduce any instability in the computation.

5.3.1 A single-GPU baseline In Table 16 the wall time in seconds and the number of the outermost sweeps are shown for the ME-FB-ME and the ME-BO-ME subvariants, with and without the column sorting, of the ZHZ0 single-GPU algorithm, as a baseline for the comparison with the multi-GPU algorithm. As the similar benefits of the column sorting were obvious in other trial tests runs, the non-sorting version was not considered for the full testing.

5.3.2 The multi-GPU performance In Tables 17, 18, 19, and 20 the wall time in seconds and the outermost sweep count are shown for the multi-GPU variants

$$\mathcal{A} := \text{ZHZ0-(ME-FB, ME-FB-ME)},$$
$$\mathcal{B} := \text{ZHZ0-(ME-BO, ME-FB-ME)},$$
$$\mathcal{C} := \text{ZHZ0-(ME-FB, ME-BO-ME)},$$
$$\mathcal{D} := \text{ZHZ0-(ME-BO, ME-BO-ME)},$$

respectively, run on two, four, and eight GPUs. The tests on two and four GPUs required a single node, and those on eight GPUs required two InfiniBand-connected nodes.

Table 14. The maximal relative normwise errors with respect to F and G in the real and the complex case on the small dataset for all variants of the single-GPU algorithm with (ME-FB-ME). The **minimal** value in each column is shown in bold.

ID	real case		complex case	
	max. relative error w.r.t. F	max. relative error w.r.t. G	max. relative error w.r.t. F	max. relative error w.r.t. G
0	$3.68432 \cdot 10^{-12}$	$3.70732 \cdot 10^{-12}$	$6.89432 \cdot 10^{-13}$	$6.89366 \cdot 10^{-13}$
1	$3.68346 \cdot 10^{-12}$	$3.70057 \cdot 10^{-12}$	$6.89297 \cdot 10^{-13}$	$6.89204 \cdot 10^{-13}$
2	$3.68803 \cdot 10^{-12}$	$3.69833 \cdot 10^{-12}$	$6.89300 \cdot 10^{-13}$	$6.89375 \cdot 10^{-13}$
3	$3.68659 \cdot 10^{-12}$	$3.70446 \cdot 10^{-12}$	$\mathbf{6.86220 \cdot 10^{-13}}$	$6.90340 \cdot 10^{-13}$
4	$3.68606 \cdot 10^{-12}$	$3.72483 \cdot 10^{-12}$	$6.89432 \cdot 10^{-13}$	$6.89366 \cdot 10^{-13}$
5	$\mathbf{3.67729 \cdot 10^{-12}}$	$3.71555 \cdot 10^{-12}$	$6.89297 \cdot 10^{-13}$	$\mathbf{6.89204 \cdot 10^{-13}}$
6	$3.68803 \cdot 10^{-12}$	$\mathbf{3.69833 \cdot 10^{-12}}$	$6.89300 \cdot 10^{-13}$	$6.89375 \cdot 10^{-13}$
7	$3.68659 \cdot 10^{-12}$	$3.70446 \cdot 10^{-12}$	$6.86220 \cdot 10^{-13}$	$6.90340 \cdot 10^{-13}$

Table 15. Aggregate minimal and maximal values of several profiler metrics for `bstep1s` kernel invocations. Min/max was again taken over the results in four contexts: DHZ0 and ZHZ0, both in (ME-FB-ME) and (ME-BO-ME) subvariants; $n = 9728$.

`nvprof` metric `sm_52` architecture	minimal value [%]	maximal value [%]
achieved_occupancy	25.00	25.00
branch_efficiency	99.75	99.94
sm_efficiency	96.54	97.64
warp_execution_efficiency	99.92	99.96
warp_nonpred_execution_efficiency	98.57	99.90

Table 16. Wall time in seconds and the sweep count for the ME-FB-ME and the ME-BO-ME subvariants of the ZHZ0 single-GPU algorithm, with and without the column sorting, on a pair of matrices of order $n = 18432$.

ME-FB-ME	ME-BO-ME	column sort
2063.19 s; 10	2248.48 s; 11	yes
3509.21 s; 17	3482.66 s; 17	no

When a particular test was not possible to be run due to an insufficient amount of the GPU RAM, "n/a" is shown in the test's table cell.

Table 17. Wall time in seconds and the outermost sweep count for the variant $\mathcal{A}$.

n	2 GPUs	4 GPUs	8 GPUs
18432	3796.78 s; 5	2594.38 s; 6	1638.43 s; 7
24576	8869.04 s; 5	6718.15 s; 6	5134.84 s; 7
36864	n/a	21271.83 s; 6	12560.23 s; 7

5.3.3 Conclusions It is clear from Tables 17, 18, 19, and 20, that the multi-GPU variant $\mathcal{B}$ is to be recommended when performance matters.

Table 18. Wall time in seconds and the outermost sweep count for the variant $\mathcal{B}$.

n	2 GPUs	4 GPUs	8 GPUs
18432	1606.17 s; 9	1085.34 s; 9	774.97 s; 9
24576	3536.52 s; 9	2568.67 s; 9	2004.23 s; 9
36864	n/a	7643.07 s; 9	4870.56 s; 9

Table 19. Wall time in seconds and the outermost sweep count for the variant $\mathcal{C}$.

n	2 GPUs	4 GPUs	8 GPUs
18432	4295.87 s; 5	2903.28 s; 6	1806.63 s; 7
24576	10098.11 s; 5	7693.07 s; 6	5825.82 s; 7
36864	n/a	23611.93 s; 6	14225.79 s; 7

Table 20. Wall time in seconds and the outermost sweep count for the variant $\mathcal{D}$.

n	2 GPUs	4 GPUs	8 GPUs
18432	1759.48 s; 10	1201.17 s; 10	849.58 s; 10
24576	3926.69 s; 10	2863.27 s; 10	2224.73 s; 10
36864	n/a	9248.23 s; 11	5877.18 s; 11

Dividing the shortest single-GPU baseline wall time from Table 16 for $n = 18432$ with the wall times from Table 18 for the same n but with a different number of GPUs, it can be derived that the speedup with two GPUs is $1.28\times$, with four GPUs is $1.90\times$, and with eight GPUs is $2.66\times$. These speedups could be even lower on a slower network or if there were fewer than four GPUs present per node.

It would be interesting to see for what number of GPUs, depending on the input sizes, the speedup peaks and starts falling, but that is beyond reach of the testing environment. In the absence of such information, a safe rule of thumb would be to use a modest number of GPUs on a fast interconnect for a given problem, such that they are fully utilized in the terms of multiprocessors and memory.

6 Conclusions and future work

The proposed algorithms compute the generalized SVD efficiently, accurately, and almost entirely on the GPU(s). The single-GPU algorithm requires a CPU only for the controlling purposes. The multi-GPU algorithm involves a substantial amount of unavoidable communication, but scales acceptably as long as each GPU if kept fully utilized.

Several generalizations of the algorithms' design are possible for the other implicit Jacobi-type methods that are to be ported to the GPUs. One such method is a computation of the generalized hyperbolic SVD (GHSVD) Bojanczyk (2003) by a modification of the implicit Hari–Zimmermann algorithm, as described in Singer et al. (2020).

Acknowledgements

This research was performed using the resources of computer cluster Isabella based in SRCE - University of Zagreb University Computing Centre.

We would like to thank the associate editor and the anonymous referees for their recommendations for improving this manuscript.

Declaration of conflicting interests

The Authors declare that there is no conflict of interest.

Funding

This work has been supported in part by Croatian Science Foundation under the project IP–2014–09–3670.

Supplemental material

Details of the chosen Jacobi strategies and the additional testing results are provided as the supplementary material at `https://arxiv.org/src/1909.00101v3/anc/sm.pdf` URL.

The source code is available in `https://github.com/venovako/GPUHZGSVD` repository. For comparing with the accuracy results a tag `https://github.com/venovako/GPUHZGSVD/tree/rev0` should be used. Note that the actual names of the kernels, variables, etc. in the code are different. The names in the paper are chosen for the simplicity of referencing.

References

Alter O, Brown PO and Botstein D (2003) Generalized singular value decomposition for comparative analysis of genome-scale expression data sets of two different organisms. *P. Natl. Acad. Sci. USA* 100(6): 3351–3356. DOI:10.1073/pnas.0530258100.

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A and Sorensen D (1999) *LAPACK Users' Guide*. 3rd edition. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics. ISBN 0-89871-447-8 (paperback).

Arafa Y, Badawy AA, Chennupati G, Santhi N and Eidenbenz S (2019) Low overhead instruction latency characterization for NVIDIA GPGPUs. In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. pp. 1–8. DOI: 10.1109/HPEC.2019.8916466.

Bai Z (1994) A parallel algorithm for computing the generalized singular value decomposition. *J. Parallel Distrib. Comput.* 20(3): 280–288. DOI:10.1006/jpdc.1994.1027.

Bojanczyk AW (2003) An implicit Jacobi-like method for computing generalized hyperbolic SVD. *Linear Algebra Appl.* 358(1): 293–307. DOI:10.1016/S0024-3795(02)00394-4.

Boukaram WH, Turkiyyah G, Ltaief H and Keyes DE (2018) Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression. *Parallel Comput.* 74: 19–33. DOI:10.1016/j.parco.2017.09.001.

Cannon LE (1969) *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD Thesis, Montana State University, Bozeman, MT, USA.

Cempel C (2009) Generalized singular value decomposition in multidimensional condition monitoring of machines–A proposal of comparative diagnostics. *Mech. Syst. Signal Pr.* 23(3): 701–711. DOI:10.1016/j.ymssp.2008.07.004.

Drmač Z (1997) Implementation of Jacobi rotations for accurate singular value computation in floating point arithmetic. *SIAM J. Sci. Comput.* 18(4): 1200–1222. DOI:10.1137/S1064827594265095.

Graillat S, Lauter C, Tang PTP, Yamanaka N and Oishi S (2015) Efficient calculations of faithfully rounded l_2-norms of n-vectors. *ACM Trans. Math. Software* 41(4): art. no. 24. DOI: 10.1145/2699469.

Hammond JR, Schäfer A and Latham R (2014) To `INT_MAX`... and beyond! Exploring large-count support in MPI. In: *2014 Workshop on Exascale MPI at Supercomputing Conference*. pp. 1–8. DOI:10.1109/ExaMPI.2014.5.

Hari V (1984) *On Cyclic Jacobi Methods for the Positive Definite Generalized Eigenvalue Problem*. PhD Thesis, FernUniversität–Gesamthochschule, Hagen, Germany.

Hari V (2018) Globally convergent Jacobi methods for positive definite matrix pairs. *Numer. Algorithms* 79(1): 221–249. DOI:10.1007/s11075-017-0435-5.

Hari V (2019) On the global convergence of the complex HZ method. *SIAM J. Matrix Anal. Appl.* 40(4): 1291–1310. DOI: 10.1137/19M1265594.

Hari V, Singer S and Singer S (2010) Block-oriented J-Jacobi methods for Hermitian matrices. *Linear Algebra Appl.* 433(8–10): 1491–1512. DOI:10.1016/j.laa.2010.06.032.

Hari V, Singer S and Singer S (2014) Full block J-Jacobi method for Hermitian matrices. *Linear Algebra Appl.* 444: 1–27. DOI:10.1016/j.laa.2013.11.028.

Howland P, Jeon M and Park H (2003) Structure preserving dimension reduction for clustered text data based on

the generalized singular value decomposition. *SIAM J. Matrix Anal. Appl.* 25(1): 165–179. DOI:10.1137/S0895479801393666.

Howland P, Wang J and Park H (2006) Solving the small sample size problem in face recognition using generalized discriminant analysis. *Pattern Recogn.* 39(2): 277–287. DOI:10.1016/j.patcog.2005.06.013.

Luk FT (1985) A parallel method for computing the generalized singular value decomposition. *J. Parallel Distrib. Comput.* 2(3): 250–260. DOI:10.1016/0743-7315(85)90027-9.

Mantharam M and Eberlein PJ (1993) Block recursive algorithm to generate Jacobi–sets. *Parallel Comput.* 19: 481–496. DOI:10.1016/0167-8191(93)90001-2.

Mary T, Yamazaki I, Kurzak J, Luszczek P, Tomov S and Dongarra J (2015) Performance of random sampling for computing low-rank approximations of a dense matrix on GPUs. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. pp. 1–11 (art. no. 60). DOI:10.1145/2807591.2807613.

Message Passing Interface Forum (2015) *MPI: A Message-Passing Interface Standard, Version 3.1*. High Performance Computing Center Stuttgart (HLRS).

Novaković V (2015) A hierarchically blocked Jacobi SVD algorithm for single and multiple graphics processing units. *SIAM J. Sci. Comput.* 37(1): C1–C30. DOI:10.1137/140952429.

Novaković V (2017) *Parallel Jacobi-type algorithms for the singular and the generalized singular value decomposition*. PhD Thesis, University of Zagreb, Croatia. URL `https://urn.nsk.hr/urn:nbn:hr:217:515320`.

Novaković V and Singer S (2011) A GPU-based hyperbolic SVD algorithm. *BIT* 51(4): 1009–1030. DOI:10.1007/s10543-011-0333-5.

Novaković V, Singer S and Singer S (2015) Blocking and parallelization of the Hari–Zimmermann variant of the Falk–Langemeyer algorithm for the generalized SVD. *Parallel Comput.* 49: 136–152. DOI:10.1016/j.parco.2015.06.004.

NVIDIA Corp (2019) *CUDA C Programming Guide v10.1.243*. URL `https://docs.nvidia.com/cuda/archive/10.1/cuda-c-programming-guide/`.

Paige CC and Saunders MA (1981) Towards a generalized singular value decomposition. *SIAM J. Numer. Anal.* 18(3): 398–405. DOI:10.1137/0718026.

Quintana-Ortí G, Sun X and Bischof C (1998) A BLAS-3 version of the QR factorization with column pivoting. *SIAM J. Sci. Comp.* 19(5): 1486–1494. DOI:10.1137/S1064827595296732.

Senaratne D and Tellambura C (2013) GSVD beamforming for two-user MIMO downlink channel. *IEEE T. Veh. Technol.* 62(6): 2596–2606. DOI:10.1109/TVT.2013.2241091.

Singer S, Di Napoli E, Novaković V and Čaklović G (2020) The LAPW method with eigendecomposition based on the Hari–Zimmermann generalized hyperbolic SVD. *SIAM J. Sci. Comput.* 42(5): C265–C293. DOI:10.1137/19M1277813.

Singer S, Singer S, Novaković V, Uščumlić A and Dunjko V (2012) Novel modifications of parallel Jacobi algorithms. *Numer. Algorithms* 59: 1–27. DOI:10.1007/s11075-011-9473-6.

Slapničar I (1998) Componentwise analysis of direct factorization of real symmetric and Hermitian matrices. *Linear Algebra Appl.* 272: 227–275. DOI:10.1016/S0024-3795(97)00334-0.

Tomov S, Dongarra J and Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.* 36(5–6): 232–240. DOI:10.1016/j.parco.2009.12.005.

Van Loan CF (1976) Generalizing the singular value decomposition. *SIAM J. Numer. Anal.* 13(1): 76–83. DOI:10.1137/0713009.

Zhao Q, Rutkowski TM, Zhang L and Cichocki A (2010) Generalized optimal spatial filtering using a kernel approach with application to EEG classification. *Cogn. Neurodyn.* 4(4): 355–358. DOI:10.1007/s11571-010-9125-x.

Zhou M and van der Veen AJ (2017) Blind separation of partially overlapping data packets. *Digit. Signal Process.* 68: 154–166. DOI:10.1016/j.dsp.2017.06.009.

Zimmermann K (1969) *Zur Konvergenz eines Jacobiverfahren für gewönliche und verallgemeinerte Eigenwertprobleme*. Dissertation no. 4305, Eidgenössische Technische Hochschule, Zürich, Switzerland.

A Enhanced dot-product computation

In CUDA, the rounding mode can be specified explicitly for each arithmetic operation using the intrinsic functions. That makes an ideal setting for employing a trick from Graillat et al. (2015) to cheaply compute possibly more accurate real and complex dot-products.

For two real vectors $\mathbf{a}$ and $\mathbf{b}$ of the same length, their enhanced dot-product would require one FMA and one negation per a pair of vector elements, in addition to one multiplication. Also, adding the partial sums together needs two sum-reductions instead of one, as follows.

Let a and b be the elements of $\mathbf{a}$ and $\mathbf{b}$ at the same, arbitrary index, and let `__dmul_rd` stand for a CUDA intrinsic performing a multiplication with rounding towards $-\infty$. Take

$$c := \texttt{\_\_dmul\_rd}(a, b), \quad d := \texttt{\_\_fma\_rn}(a, b, -c).$$

Then, by looking separately at the both possible signs of c, it can be shown that for the rounding error of the multiplication extracted by the FMA holds $d \geq 0$. By sum-reducing d, no cancellation can occur. That value may be

added to the sum-reduction result on c, to form the final dot-product.

For two complex vectors $\mathbf{a}$ and $\mathbf{b}$ of the same length, defining an enhanced dot-product is not so unambiguous. A special case of computing $\|\mathbf{a}\|_2^2 = \mathbf{a}^*\mathbf{a}$ can be handled as follows. For an element a of $\mathbf{a}$, take

$$c_\mathrm{r} := \texttt{__dmul\_rd}(\mathrm{Re}(a), \mathrm{Re}(a)),$$
$$d_\mathrm{r} := \texttt{__fma\_rn}(\mathrm{Re}(a), \mathrm{Re}(a), -c_\mathrm{r}),$$

considering the real part of a, and

$$c_\mathrm{i} := \texttt{__dmul\_rd}(\mathrm{Im}(a), \mathrm{Im}(a)),$$
$$d_\mathrm{i} := \texttt{__fma\_rn}(\mathrm{Im}(a), \mathrm{Im}(a), -c_\mathrm{i}),$$

considering the imaginary part of a. Let $\Sigma\, c_\mathrm{r}$, $\Sigma\, c_\mathrm{i}$, $\Sigma\, d_\mathrm{r}$, and $\Sigma\, d_\mathrm{i}$ be the sum-reductions of all c_r, c_i, d_r, and d_i, respectively. Then, return $((\sigma_1 + \sigma_2) + \sigma_3) + \sigma_4$, where the σ_j quantities are the four Σ-sums above, ordered increasingly.

However, such an approach requires four reductions. To simplify it by trading off accuracy for speed, let $e = d_\mathrm{r} + d_\mathrm{i}$ be the sum of the rounding errors. Then, if $c_\mathrm{r} \le c_\mathrm{i}$, take $s = (e + c_\mathrm{r}) + c_\mathrm{i}$, else let $s = (e + c_\mathrm{i}) + c_\mathrm{r}$, and return the sum-reduction of s as an approximation of $\Sigma(\bar{a} \cdot a)$.

The real dot-product and the simpler of the two procedures for computing of the square of the Euclidean norm of a complex vector have been incorporated in the special variants of the algorithm and tested, both without a huge slowdown but also without a significant effect on accuracy of the result, as explained in section 5.

B A single-GPU out-of-core algorithm

Algorithm 7 gives an overview of a single-GPU "out-of-core" algorithm, when the whole data does not fit into the GPU RAM, but at least one block pair of F, G, and Z does.

This algorithm has not been implemented, but essentially the `bstep1(s/n)` kernel would be converted to a kernel `OoCstep1(s/n)` that operates on a suitably sized subset of all block pivots of an outer step. The subset is loaded into the GPU RAM beforehand, and is brought back, transformed, to the CPU. That process is repeated until all block pivots in the outer step have been transformed, exactly as they would be by a single call to `bstep1(s/n)`. Apart from being inefficient, such an approach cannot handle the case when not even a single block pair can fit into a GPU. For the GPU RAM being 16 GiB and the matrices complex and square, n would then have to be larger than 11 million, in which case the multi-GPU algorithm is a must anyway.

C Warp-shuffle +-reduction of 32 doubles

In Figure 11 a warp-shuffle sum-reduction of 32 double-precision values is shown, where each thread in a warp holds one value at the start, and all threads get the sum at the end. The loop over `i` is manually unrolled in the code.

Algorithm 7: A split of the implicit ℓ-loop of a single outer step of Algorithm 1 for the case when the whole data does not fit into the GPU RAM.

for $0 \le k < \mathtt{n}'$ **do** `// for all outer steps`
  `/* Let` $\mathbf{o}$ `be the number of block column pairs of` F, G, `and` Z `that fit into the GPU RAM. */`
  for $0 \le o < (\mathtt{n}/2)/\mathbf{o}$ **do** `//` $\mathbf{o} \bmod (\mathtt{n}/2) = 0$
    for $o \cdot \mathbf{o} \le \ell < (o+1) \cdot \mathbf{o}$ **do** `// async`
      copy the ℓth block pivot pairs of F, G, and Z from the CPU to the GPU;
    end for
    `OoCstep1(s/n)(`o, k`); // transform`
    for $o \cdot \mathbf{o} \le \ell < (o+1) \cdot \mathbf{o}$ **do** `// async`
      copy the ℓth block pivot pairs of F, G, Z, transformed, from the GPU to the CPU;
    end for
  end for
end for

```
__device__ __forceinline__
double dSum32(const double x) {
  int lo0, hi0, lo1, hi1;
  double x0 = x, x1;
  for (int i = 16; i; i >>= 1) {
    lo0 = __double2loint(x0);
    hi0 = __double2hiint(x0);
    lo1 = __shfl_xor_sync(~0u, lo0, i);
    hi1 = __shfl_xor_sync(~0u, hi0, i);
    x1 = __hiloint2double(hi1, lo1);
    x0 = __dadd_rn(x0, x1);
  }
  lo0 = __double2loint(x0);
  hi0 = __double2hiint(x0);
  lo1 = __shfl_sync(~0u, lo0, 0);
  hi1 = __shfl_sync(~0u, hi0, 0);
  return __hiloint2double(hi1, lo1);
}
```

Figure 11. Sum-reduction of 32 doubles across a warp.

D An implementation of the main kernel's subphases 1 and 4 with the batched matrix multiplication from cuBLAS

The subphases **1** and **4** of the main kernel (see subsection 3.5) could have also been decoupled and implemented separately, by the batched matrix multiplication routines from the cuBLAS library. For that, more than twice the GPU RAM is required, to maintain two copies of the data.

The first copy, called "actual", holds the current state of the computation at the start of each block step. The second one, called "shadow", is used as a temporary buffer, in which the new state is assembled at the end of the step. These copies then change the roles by a simple swap of the pointers to them. Schematically, this layout can be represented as

$$\text{actual} \rightarrow \boxed{\boxed{F^{!}}\boxed{G^{!}}\boxed{Z^{!}}}, \quad \text{shadow} \rightarrow \boxed{\boxed{F^{?}}\boxed{G^{?}}\boxed{Z^{?}}}.$$

Let $\mathbb{F} \in \{\mathbb{R}, \mathbb{C}\}$ be the field in which the computations take place. For the complex algorithm, a data layout with the real and the imaginary components kept separately is no longer viable because cuBLAS does not support it.

Furthermore, let $\mathtt{b} = \mathtt{n}/2$ be the number of block column pairs processed at each block step, i.e., the number of thread blocks launched. An additional workspace of $\mathtt{b}$ square matrices Z_l of order 32 is required, where $1 \le l \le \mathtt{b}$, as

$$\text{work} \rightarrow \boxed{\boxed{\mathsf{Z}_0}\cdots\boxed{\mathsf{Z}_{\mathtt{b}-1}}}.$$

In this GPU RAM workspace the transformation matrices Z_l accumulated in the subphase **3** will be stored from the shared memory of each thread block.

For each l there is an associated pair of block indices $(\mathtt{p}_l, \mathtt{q}_l)$ drawn from the chosen block strategy for the current block step, with the subscripts l omitted when they are implied by the context. Then, the block columns of the actual copy can be logically grouped and denoted as

$$F^{!}_l = \begin{bmatrix} F^{!}_{\mathtt{p}} & F^{!}_{\mathtt{q}} \end{bmatrix}, \quad G^{!}_l = \begin{bmatrix} G^{!}_{\mathtt{p}} & G^{!}_{\mathtt{q}} \end{bmatrix}, \quad Z^{!}_l = \begin{bmatrix} Z^{!}_{\mathtt{p}} & Z^{!}_{\mathtt{q}} \end{bmatrix},$$

and a similar notation is used for the shadow copy as well.

Replacing the subphase **1** involves forming the lower block-triangle of the Grammian matrices $\widehat{\mathsf{A}}_l$ and $\widehat{\mathsf{B}}_l$ as

$$\widehat{\mathsf{A}}_l := \begin{bmatrix} F^{!*}_{\mathtt{p}} \cdot F^{!}_{\mathtt{p}} & ??? \\ F^{!*}_{\mathtt{q}} \cdot F^{!}_{\mathtt{p}} & F^{!*}_{\mathtt{q}} \cdot F^{!}_{\mathtt{q}} \end{bmatrix}, \quad \widehat{\mathsf{B}}_l := \begin{bmatrix} G^{!*}_{\mathtt{p}} \cdot G^{!}_{\mathtt{p}} & ??? \\ G^{!*}_{\mathtt{q}} \cdot G^{!}_{\mathtt{p}} & G^{!*}_{\mathtt{q}} \cdot G^{!}_{\mathtt{q}} \end{bmatrix},$$

by two batched `GEMM` calls: one for $\widehat{\mathsf{A}}_l$, and another for $\widehat{\mathsf{B}}_l$.

Each batched `GEMM` call forms all $(\mathtt{p},\mathtt{p})$, $(\mathtt{q},\mathtt{p})$, and $(\mathtt{q},\mathtt{q})$ blocks, since the inputs are all of the same dimensions, as well as the outputs. Forming the remaining, upper-right blocks is not necessary, since the subphase **2** reads only the lower triangle as the Grammian matrices are Hermitian. The diagonal blocks should be formed by the batched `SYRK` or `HERK` operations instead, but at present they are not implemented in cuBLAS. Each $\widehat{\mathsf{A}}_l$ is stored in the first 32 rows of the shadow copy $F^{?}$, starting from its $(32 \cdot l)$th column, and similarly for $\widehat{\mathsf{B}}_l$ and $G^{?}$, from where they are loaded by the modified subphase **2** into the shared memory and their Cholesky factorizations are then computed as before. Note that this approach involves storing $\widehat{\mathsf{A}}_l$ and $\widehat{\mathsf{B}}_l$ in the GPU RAM, only to be read afterwards into the shared memory, what is a redundant memory traffic avoided by the original implementation.

The modified main kernel, after completing the subphase **2**, computes the transformation matrices Z_l in the subphase **3**, and stores them from the shared memory to the workspace (another redundant memory traffic). For the postmultiplications, i.e., the block column updates, consider each Z_l to be logically partitioned as

$$\mathsf{Z}_l = \begin{bmatrix} \mathsf{Z}_{11} & \mathsf{Z}_{12} \\ \mathsf{Z}_{21} & \mathsf{Z}_{22} \end{bmatrix}, \quad \mathsf{Z}_{\mathtt{ij}} \in \mathbb{F}^{16\times 16}, \quad 1 \le \mathtt{i},\mathtt{j} \le 2.$$

Now, compute a partial update of $F^{!}$ by a single batched `GEMM` call (and similarly for $G^{!}$ and $Z^{!}$), where for each l,

$$F^{?}_{\mathtt{p}} := F^{!}_{\mathtt{p}} \cdot \mathsf{Z}_{11}, \quad F^{?}_{\mathtt{q}} := F^{!}_{\mathtt{p}} \cdot \mathsf{Z}_{12},$$

and complete the update by another batched `GEMM` call as

$$F^{?}_{\mathtt{p}} := F^{!}_{\mathtt{q}} \cdot \mathsf{Z}_{21} + F^{?}_{\mathtt{p}}, \quad F^{?}_{\mathtt{q}} := F^{!}_{\mathtt{q}} \cdot \mathsf{Z}_{22} + F^{?}_{\mathtt{q}}.$$

A swap of the pointers to the copies, $\text{actual} \leftrightarrows \text{shadow}$, completes the block step. See the `var` subdirectory of the code repository for a double precision implementation.

A benefit of this alternative is that it significantly reduces the register pressure in the main kernel, e.g., close to halving the required number of registers on a Volta GPU. However, that is offset by allocating more than twice the GPU RAM, which is still a scarce resource, and consequently by reducing the maximal input sizes per GPU. Also, setting up a huge number of pointers for each batched `GEMM` call, as well as the redundant stores to and loads from the GPU RAM of the contents of the shared memory makes the prototype implementation about 20 times slower on average than the original one. Therefore, this version of the algorithm is not recommendable on the present hardware.

E The QR factorizations as an alternative to the main kernel's subphases 1 and 2 for the ill-conditioned input matrices

As mentioned in subsection 2.3, for a highly ill-conditioned input matrix the formation of the Grammians of its block-column pairs by the main kernel's subphase **1** might result in several numerically singular or indefinite blocks, on which the Cholesky factorizations in the subphase **2** are bound to fail. In such cases the affected (or, for simplicity, all) block-column pairs have to be shortened by the QR factorizations instead. Ideally, the factorizations would employ column pivoting (see, e.g., Quintana-Ortí et al. (1998) and the LAPACK's `XGEQP3` routines, $\mathtt{X} \in \{\mathtt{D},\mathtt{Z}\}$), but it can be inefficient on GPUs Mary et al. (2015).

The batched non-pivoted QR factorization routine from cuBLAS could be used with the memory layout proposed for Appendix D, where the τ vectors would be stored in

the work buffer. Replacing the subphases **1** and **2** that way would require to pack the actual block columns, e.g., $F_{\mathrm{p}}^{!}$ and $F_{\mathrm{q}}^{!}$, consecutively in the shadow copy $F^{?}$ for each l (similarly for $G^{!}$). The packed block-column pairs in the shadow copy would then be subject to the QR factorizations, and the upper triangular 32×32 factors would be extracted from it by the modified subphase **3**. The post-multiplication subphase would remain as in Appendix D. However, this approach would also suffer from the redundant traffic between the shared and the global memory.

Another approach is to keep the original memory layout and have two "streaming" QR factorizations (one for the assigned block-column pair of F, and another for the same-indexed block-column pair of G) embedded into the main kernel. Each factorization reads from the global memory only once, in chunks, never writes to it (since the matrix Q does not have to be applied again or restored), and leaves the upper triangular factor in the shared memory where the subphase **3** expects it to be. The full details of this in-kernel QR factorization can be found in (Novaković 2015, subsection 4.2) for double precision, while a complex version would be implemented similarly.

Table 21 demonstrates that close to eightfold slowdown can be expected on a Volta GPU with the in-kernel QR factorizations on larger matrices, even when using the simplest, unsafe Frobenius norm computations inside them.

Table 21. The wall times and their ratios of DHZ0-(ME-FB-ME) with the Cholesky factorizations of the Grammian matrices (■) and with the QR factorizations of the block-column pairs (□), both in-kernel, on the small real dataset.

n	□ [s]	■ [s]	□/■	n	□ [s]	■ [s]	□/■
512	0.83	0.20	4.1	5632	437.48	57.81	7.6
1024	3.57	0.69	5.1	6144	522.05	68.10	7.7
1536	7.30	1.24	5.9	6656	544.53	71.13	7.7
2048	12.87	2.02	6.4	7168	674.35	87.38	7.7
2560	21.86	3.31	6.6	7680	775.09	100.64	7.7
3072	61.46	8.69	7.1	8192	1015.82	128.65	7.9
3584	90.87	12.66	7.2	8704	1233.91	157.62	7.8
4096	128.73	17.52	7.3	9216	1389.86	176.37	7.9
4608	186.27	25.22	7.4	9728	1551.47	198.03	7.8
5120	230.32	31.26	7.4	–	–	–	–

F Performance-wise comparison of a CPU and a single-GPU implementation of the complex Hari–Zimmermann GSVD

Table 22 shows the speedup of the complex single-GPU algorithm on the small dataset versus the CPU GHSVD algorithm from Singer et al. (2020), with the signature matrix for the latter being $J = I$. The GHSVD's implementation was not tuned for the simpler GSVD computation. If it were, it could have been at least twice as fast on a more modern CPU, but nevertheless noticeably slower than the GPU algorithm on the Volta architecture.

Table 22. The wall times and their ratios of ZHZ0-(ME-FB-ME) (•) and a close CPU analogue of ZHZ6-(ME-BO-ME) with the variable block sizes and 64 threads (one per core) of an Intel Xeon Phi 7210 (○), on the small complex dataset.

n	○ [s]	• [s]	○/•	n	○ [s]	• [s]	○/•
512	1.22	0.26	4.7	5632	422.37	65.27	6.5
1024	9.52	0.81	11.8	6144	671.15	77.85	8.6
1536	21.52	1.68	12.8	6656	659.89	92.55	7.1
2048	45.40	2.98	15.3	7168	756.02	109.74	6.9
2560	70.81	5.46	13.0	7680	971.17	131.07	7.4
3072	114.68	13.30	8.6	8192	1931.45	182.53	10.6
3584	164.41	17.96	9.2	8704	1258.28	208.32	6.0
4096	397.57	23.63	16.8	9216	1520.19	236.45	6.4
4608	270.74	30.77	8.8	9728	1646.25	269.04	6.1
5120	360.70	40.11	9.0	–	–	–	–

G Computational pitfalls of the GEVD of (F^*F, G^*G) versus the GSVD of (F, G)

This small example shows what happens with the GEVD of (F^*F, G^*G) versus the GSVD of (F, G) when one of the matrices (here, G) is even mildly ill-conditioned. All results were obtained by Matlab R2020a Update 3.

Let F and G, with $0 < g_{11} \ll 1$, be given as

$$F = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad G = \begin{bmatrix} g_{11} & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Clearly, F and G are non-singular, so $A := F^T F$ and $B := G^T G$ are symmetric positive definite. If, e.g., $g_{11} = 10^{-10}$ (rounded to double precision) is taken, the generalized eigenvalues of (A, B), also rounded to double precision, are

$$\Lambda(\mathtt{A}, \mathtt{B}) = \begin{bmatrix} 5.000000000500000 \cdot 10^{-1} \\ 1.000000000000000 \cdot 10^{0} \\ 1.000000000000000 \cdot 10^{0} \\ 1.999999999800000 \cdot 10^{20} \end{bmatrix},$$

where A and B are the Grammians of the rational (symbolic) representations of F and G, respectively, while $\Lambda(\mathtt{A}, \mathtt{B})$ is a sorted representation of the symbolic output of $\mathtt{eig}(\mathtt{A}, \mathtt{B})$.

When computing $\Lambda(A, B)$ in double precision, two routes can be taken. The first one finds the generalized

singular values of F and G, i.e., $\Sigma(F,G) := \texttt{gsvd}(F,G)$,

$$\Sigma(F,G) = \begin{bmatrix} 7.071067812219032 \cdot 10^{-1} \\ 9.999999999999997 \cdot 10^{-1} \\ 9.999999999999997 \cdot 10^{-1} \\ 1.414213562302384 \cdot 10^{10} \end{bmatrix},$$

and squares them to get the generalized eigenvalues $\widehat{\Lambda}$,

$$\widehat{\Lambda}(A,B) = \begin{bmatrix} 5.000000000500004 \cdot 10^{-1} \\ 9.999999999999993 \cdot 10^{-1} \\ 9.999999999999993 \cdot 10^{-1} \\ 1.999999999800000 \cdot 10^{20} \end{bmatrix},$$

which are close to $\Lambda(A,B)$, i.e., the exact ones.

The second route computes $\widetilde{\Lambda}(A,B) := \texttt{eig}(A,B)$ from A and B by a "proper" GEVD routine, getting

$$\widetilde{\Lambda}(A,B) = \begin{bmatrix} 0.000000000000000 \cdot 10^{0} \\ 0.000000000000000 \cdot 10^{0} \\ 0.000000000000000 \cdot 10^{0} \\ 1.999999999800000 \cdot 10^{20} \end{bmatrix},$$

with the lower three generalized eigenvalues vanishing.

The spectral condition numbers of the matrices are:

$$\begin{aligned} \kappa_2(F) &\approx 5.411474127809772 \cdot 10^{0} \\ \kappa_2(G) &\approx 3.972825427374361 \cdot 10^{10} \\ \kappa_2(A) &\approx 2.928405223595454 \cdot 10^{1} \quad = \kappa_2^2(F) \\ \kappa_2(B) &\approx 1.578344820257155 \cdot 10^{21} \quad = \kappa_2^2(G) \end{aligned},$$

i.e., the condition numbers of A and B are the squares of those of F and G, respectively, which is easily shown by taking the SVD of F or G and expressing A or B in terms of it. Therefore, a mildly ill-conditioned matrix G causes its Grammian B to be severely ill-conditioned and the GEVD fails, so the matrix multiplications have to be avoided in such a case by computing the GSVD of (F,G) instead.

Motivated by the previous example, 16 pairs (F,G_j) of square real matrices of order 512 were generated in quadruple precision by the adapted symmetric indefinite factorization with complete pivoting Slapničar (1998), and then rounded to double precision, such that $A = F^TF$, with $\kappa_2(A) = 10$, was fixed, while $B_j = G_j^TG_j$ varied, with the prescribed values for $\kappa_{B_j} = \kappa_2(B_j) = 10^j$, for $1 \le j \le 16$. The matrices A and B_j were generated from the given positive eigenvalues by the LAPACK's testing routine `DLAROR` adapted to quadruple precision, ensuring that their condition numbers are as close to the prescribed ones as possible, and that their ensuing factorizations were identical in effect to the Cholesky factorizations with diagonal pivoting. Then, the generalized singular values $\widehat{\Sigma}_j$ of (F,G_j), i.e., the singular values of FG_j^{-1}, were computed in high precision with 32 significant `digits` as

$$\widehat{\Sigma}_j = \texttt{double}(\texttt{svd}(\texttt{vpa}(F)/\,\texttt{vpa}(G_j))),$$

by Matlab R2020b, to be compared against in two ways.

The generalized singular values Σ_j of (F,G_j) were recomputed by the DHZ0-(ME-FB-ME) variant, and the generalized eigenvalues Λ_j of (A,B_j) were obtained by the `cusolverDnDsygvd` routine from the cuSOLVER library, to assess how well they approximate squares of the generalized singular values. The maximal relative errors in the generalized singular values computed by the last two approaches were then found, with $1 \le i \le 512$, as

$$\begin{aligned} \mathrm{mre}(\Sigma_j) &= \max_i \frac{|\,\texttt{ext}(\sigma_{j,i}) - \texttt{ext}(\hat{\sigma}_{j,i})|}{\texttt{ext}(\hat{\sigma}_{j,i})}, \\ \mathrm{mre}(\Lambda_j) &= \max_i \frac{|\sqrt{\texttt{ext}(\lambda_{j,i})} - \texttt{ext}(\hat{\sigma}_{j,i})|}{\texttt{ext}(\hat{\sigma}_{j,i})}, \end{aligned}$$

where $\widehat{\Sigma}_j$, Σ_j, and Λ_j were sorted descendingly. Here, `ext` denotes a conversion to the Intel's extended 80-bit datatype.

As it can be seen in Table 23, the generalized singular values computed by taking square roots of the generalized eigenvalues rapidly lose accuracy when the condition of B_j increases beyond about $1/\sqrt{\varepsilon}$, where ε is the machine precision. Entirely invalid results are possible, indicated by $\mathrm{mre}(\Lambda_{16}) = \texttt{NaN}$, since the smallest six generalized eigenvalues of (A,B_{16}) were computed as negative and the square roots could not have been taken. The generalized singular values computed by a "proper" GSVD (the HZ algorithm) remained relatively accurate in all cases, though.

Table 23. Relative accuracy of the generalized singular values of (F,G_j) computed by the GSVD and the GEVD algorithms.

κ_B	$\mathrm{mre}(\Sigma)$	$\mathrm{mre}(\Lambda)$	κ_B	$\mathrm{mre}(\Sigma)$	$\mathrm{mre}(\Lambda)$
10^1	$8.9 \cdot 10^{-16}$	$8.1 \cdot 10^{-16}$	10^9	$1.4 \cdot 10^{-15}$	$2.9 \cdot 10^{-9}$
10^2	$9.3 \cdot 10^{-16}$	$5.8 \cdot 10^{-15}$	10^{10}	$1.2 \cdot 10^{-15}$	$1.2 \cdot 10^{-7}$
10^3	$1.2 \cdot 10^{-15}$	$9.1 \cdot 10^{-15}$	10^{11}	$1.1 \cdot 10^{-15}$	$1.3 \cdot 10^{-6}$
10^4	$2.2 \cdot 10^{-15}$	$7.1 \cdot 10^{-14}$	10^{12}	$2.2 \cdot 10^{-15}$	$3.4 \cdot 10^{-4}$
10^5	$1.2 \cdot 10^{-15}$	$7.6 \cdot 10^{-13}$	10^{13}	$1.1 \cdot 10^{-15}$	$3.7 \cdot 10^{-3}$
10^6	$2.2 \cdot 10^{-15}$	$1.1 \cdot 10^{-11}$	10^{14}	$1.4 \cdot 10^{-15}$	$2.3 \cdot 10^{-2}$
10^7	$2.3 \cdot 10^{-15}$	$8.4 \cdot 10^{-11}$	10^{15}	$1.1 \cdot 10^{-15}$	$3.5 \cdot 10^{-2}$
10^8	$1.3 \cdot 10^{-15}$	$1.5 \cdot 10^{-9}$	10^{16}	$1.6 \cdot 10^{-15}$	NaN

However, if it is known in advance that a particular GSVD problem on (F,G) is extremely well conditioned, solving it as the GEVD of (A,B) might be significantly faster. From $AZ = BZ\Lambda$, $FZ = U\Sigma_F$, and $GZ = V\Sigma_G$, it follows that finding Z (the generalized eigenvalues) and Λ (the generalized eigenvectors) suffices to compute $U\Sigma_F$, $V\Sigma_G$, and $\Sigma = \Lambda^{1/2}$. Normalizing the columns of $U\Sigma_F$ both U and Σ_F (with the extracted norms on its diagonal)

are obtained, and similarly for V and Σ_G, but U and V might not be as orthogonal as those that come from a "proper" GSVD. For the left generalized singular vectors to be computed, the original matrices F and G have to be preserved, which, along with a non-trivial amount of workspace needed for the GEVD routine call, more than doubles the memory requirements compared to those of the single-GPU Hari–Zimmermann GSVD algorithm.

The matrices A and B are formed on a GPU, each by a single call of the cuBLAS' routine `cublasDsyrk` (or `cublasZherk` in the complex case) from F and G, respectively. Then, the cuSOLVER's (legacy) routine `cusolverDnDsygvd` (or `cusolverDnZhegvd` in the complex case) solves the GEVD problem on (A, B). For FZ and GZ multiplications the cuBLAS' `cublasDgemm` (or `cublasZgemm` in the complex case) routine can be used. Finally, the normalizing, as described above, can be performed by a custom kernel similar to `rescale`.

The computation as described above, but skipping the normalization, was compared performance-wise to the DHZ0-(ME-FB-ME) and the ZHZ0-(ME-FB-ME) variants. Let `D` stand for the ratio of the wall times of the "proper" GSVD and the GSVD-using-GEVD approaches in the real, and `Z` in the complex case, on the respective small datasets. Then, `D` and `Z`, as shown in Table 24, are relaxed upper bounds on the expected speedup by the GEVD approach (the actual speedup should be somewhat smaller) on the GPUs. For comparison, about 15–$35\times$ speedup is expected on the modern CPUs in the complex case Singer et al. (2020) on a set of small-to-medium sized matrices. But it should never be understated that there are hard limits to the applicability of the GEVD approach to the GSVD when the condition of the problem is high or unknown in advance.

Table 24. The wall time ratios of the "proper" (HZ) GSVD and the partial GSVD-using-GEVD on a single Volta GPU in the real (`D`) and the complex (`Z`) cases, on the small datasets.

n	`D` [×]	`Z` [×]	n	`D` [×]	`Z` [×]
512	6.57	8.28	5632	49.92	26.63
1024	9.39	10.49	6144	49.25	25.30
1536	10.76	11.75	6656	41.63	24.02
2048	11.92	12.53	7168	42.38	23.60
2560	13.88	15.05	7680	41.82	23.45
3072	26.32	24.72	8192	45.45	26.95
3584	27.76	22.83	8704	46.42	25.92
4096	30.06	21.93	9216	44.71	25.07
4608	34.12	21.35	9728	43.41	24.50
5120	34.15	21.53	–	–	–

Author Biographies

Vedran Novaković received his Ph.D. in Mathematics in 2017 from University of Zagreb, Croatia, where he started his career as a teaching assistant. He also worked at STFC Daresbury Laboratory, UK, as a computational scientist and at Universidad Jaime I, Spain. He is interested in parallel algorithms of numerical linear algebra, especially eigenvalue and singular value algorithms.

https://orcid.org/0000-0003-2964-9674

Sanja Singer is a tenured professor of Mathematics at the Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, Croatia. She received her Ph.D. in Mathematics in 1997 from the same University. Her interests lie in accurate and high-performance algorithms of numerical linear algebra, especially matrix factorization algorithms, eigenvalue and singular value algorithms.

https://orcid.org/0000-0002-4358-1840