Sveučilište u Zagrebu

Faculty of Science
Department of Mathematics

Vedran Novaković

Parallel Jacobi-type algorithms for the singular and the generalized singular value decomposition

DOCTORAL THESIS

Zagreb, 2017

University of Zagreb

Faculty of Science
Department of Mathematics

Vedran Novaković

Parallel Jacobi-type algorithms for the singular and the generalized singular value decomposition

DOCTORAL THESIS

Supervisor: prof. Sanja Singer, PhD

Zagreb, 2017

Sveučilište u Zagrebu

Prirodoslovno–matematički fakultet
Matematički odsjek

Vedran Novaković

Paralelni algoritmi Jacobijeva tipa za singularnu i generaliziranu singularnu dekompoziciju

DOKTORSKI RAD

Mentor: prof. dr. sc. Sanja Singer

Zagreb, 2017.

A note on the supervisor and the committees (O mentoru i komisijama)

Sanja Singer was born in Zagreb on December 27th, 1963, where she earned BSc. (1986), MSc. (1993) and PhD. (1997) in mathematics at the University of Zagreb, Faculty of Science, Department of Mathematics.

From December 1987 to September 1989 she was employed as a junior assistant at the University of Zagreb, Faculty of Economics. Since September 1989 she has been affiliated with the University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture, first as a research assistant, then assistant professor (2001–2007), associate professor (2007–2013) and full professor (2013–).

She has co-authored 27 papers, 17 of them published in journals. Her research interests are the algorithms for the dense matrix factorizations, eigenvalues and singular values, and efficient parallelization of those algorithms.

Sanja Singer rođena je u Zagrebu 27. prosinca 1963., gdje je diplomirala (1986.), magistrirala (1993.) i doktorirala (1997.) na Matematičkom odsjeku Prirodoslovno–matematičkog fakulteta Sveučilišta u Zagrebu.

Od prosinca 1987. do rujna 1989. bila je zaposlena kao asistentica pripravnica na Ekonomskom fakultetu Sveučilišta u Zagrebu. Od rujna 1989. zaposlena je na Fakultetu strojarstva i brodogradnje Sveučilišta u Zagrebu, prvo kao znanstvena novakinja, a zatim kao docentica (2001.–2007.), izvanredna (2007.–2013.) i redovita profesorica (2013.–).

Koautorica je 27 znanstvenih radova, od kojih je 17 objavljeno u časopisima. Njezin znanstveni interes su algoritmi za računanje matričnih faktorizacija, svojstvenih i singularnih vrijednosti, te njihova efikasna paralelizacija.

The evaluation$^{\mathsf{E}}$ and defense$^{\mathsf{D}}$ committees (komisije za ocjenu$^{\mathsf{E}}$ i obranu$^{\mathsf{D}}$ rada):

1. $^{\mathsf{E,D}}$ Vjeran Hari, University of Zagreb (PMF-MO), full professor, chairman
2. $^{\mathsf{E,D}}$ Enrique S. Quintana Ortí, Universitat Jaume I, Spain, full professor
3. $^{\mathsf{E}}$ Zvonimir Bujanović, University of Zagreb (PMF-MO), assistant professor
4. $^{\mathsf{D}}$ Sanja Singer, University of Zagreb (FSB), full professor, supervisor

---

This thesis was submitted on September 4th, 2017, to the Council of Department of Mathematics, Faculty of Science, University of Zagreb. Following a positive review by the evaluation committee, approved by the Council on November 8th (at its 2nd session in academic year 2017/18), and after the submission of the thesis in its present (revised) form, the public defense was held on December 15th, 2017.

Acknowledgments

I would like to express my gratitude and dedicate this work to Sanja Singer, for trying and succeeding to be a proactive, but at the same time not too active supervisor,

For drawing many of the figures shown here with METAPOST from handmade sketches, helping when all hope seemed to be lost while debugging the code, sorting out the bureaucratic hurdles, financing herself the research efforts when there was no other way, and encouraging me for many years to finish this manuscript,

But also for stepping aside when it was best for me to discover, learn, and do things the hard way myself, knowing that such experience does not fade with time,

And for being a friend in times when friendship was scarce.

This work has been supported in part by Croatian Science Foundation under the project IP-2014-09-3670 ("Matrix Factorizations and Block Diagonalization Algorithms" – MFBDA); by grant 037–1193086–2771 ("Numerical methods in geophysical models") from Ministry of Science, Education and Sports, Republic of Croatia; by an NVIDIA's Academic Partnership Program hardware donation (arranged by Zlatko Drmač) to a research group from University of Zagreb, Croatia; and by the travel grants from Swiss National Science Foundation, Universitat Jaume I, Spain, and The University of Manchester, United Kingdom.

The completion of this work would not be possible without the author having generously been granted allocation of the computing resources and administrative support from Department of Mathematics, Faculty of Science, and from Faculty of Mechanical Engineering and Naval Architecture (by Hrvoje Jasak), University of Zagreb, Croatia; from Universitat Jaume I, Spain (by Enrique S. Quintana Ortí); and by the Hartree Centre and its staff at the Daresbury Laboratory of the Science and Technology Facilities Council, United Kingdom.

I am indebted to Saša Singer for fruitful advice, and for proofreading and correcting the manuscripts of the papers leading to this dissertation, alongside with Neven Krajina; and to Zvonimir Bujanović for GPU-related and Vjeran Hari for mathematical discussions, all of University of Zagreb. Norbert Juffa of NVIDIA has kindly provided an experimental CUDA implementation of the `rsqrt` routine.

The people of NLAFET Horizon 2020 project and others who happened to be in my vicinity should be thanked for bearing with me while finishing this manuscript.

"*Hope... is not the conviction that something will turn out well, but the certainty that something makes sense, regardless of how it turns out.*" — Václav Havel

Abstract

In this thesis, a hierarchically blocked one-sided Jacobi algorithm for the singular value decomposition (SVD) is presented. The algorithm targets both single and multiple graphics processing units (GPUs). The blocking structure reflects the levels of the GPU's memory hierarchy. To this end, a family of parallel pivot strategies on the GPU's shared address space has been developed, but the strategies are applicable to inter-node communication as well, with GPU nodes, CPU nodes, or, in general, any NUMA nodes. Unlike common hybrid approaches, the presented algorithm in a single-GPU setting needs a CPU for the controlling purposes only, while utilizing the GPU's resources to the fullest extent permitted by the hardware. When required by the problem size, the algorithm, in principle, scales to an arbitrary number of GPU nodes. The scalability is demonstrated by more than twofold speedup for sufficiently large matrices on a four-GPU system vs. a single GPU.

The subsequent part of the thesis describes how to modify the two-sided Hari–Zimmermann algorithm for computation of the generalized eigendecomposition of a symmetric matrix pair (A, B), where B is positive definite, to an implicit algorithm that computes the generalized singular value decomposition (GSVD) of a pair (F, G). In addition, blocking and parallelization techniques for accelerating both the CPU and the GPU computation are presented, with the GPU approach following the Jacobi SVD algorithm from the first part of the thesis.

For triangular matrix pairs of a moderate size, numerical tests show that the double precision sequential pointwise algorithm is several times faster than the established `DTGSJA` algorithm in LAPACK, while the accuracy is slightly better, especially for the small generalized singular values. Cache-aware blocking increases the performance even further. As with the one-sided Jacobi-type (G)SVD algorithms in general, the presented algorithm is almost perfectly parallelizable and scalable on the shared memory machines, where the speedup almost solely depends on the number of cores used. A distributed memory variant, intended for huge matrices that do not fit into a single NUMA node, as well as a GPU variant, are also sketched.

The thesis concludes with the affirmative answer to a question whether the one-sided Jacobi-type algorithms can be an efficient and scalable choice for computing the (G)SVD of dense matrices on the massively parallel CPU and GPU architectures.

Unless otherwise noted by the inline citations or implied by the context, this thesis is an overview of the original research results, most of which has already been published in [55, 58]. The author's contributions are the one-sided Jacobi-type GPU algorithms for the ordinary and the generalized SVD, of which the latter has not yet been published, as well as the parallelization technique and some implementation details of the one-sided Hari–Zimmermann CPU algorithm for the GSVD. The rest is joint work with Sanja and Saša Singer.

Key words: one-sided Jacobi-type algorithms, (generalized) singular value decomposition, parallel pivot strategies, graphics processing units, (generalized) eigenvalue problem, cache-aware blocking, (hybrid) parallelization.

Sažetak (prošireni)

Singularna dekompozicija, katkad zvana prema engleskom originalu i dekompozicija singularnih vrijednosti, ili kraće SVD, jedna je od najkorisnijih matričnih dekompozicija, kako za teorijske, tako i za praktične svrhe. Svaka matrica $G \in \mathbb{C}^{m\times n}$ (zbog jednostavnijeg zapisa, uobičajeno se smatra da je $m \geq n$; u protivnom, traži se SVD matrice G^*) može se rastaviti u produkt tri matrice

$$G = U\Sigma V^*,$$

gdje su $U \in \mathbb{C}^{m\times m}$ i $V \in \mathbb{C}^{n\times n}$ unitarne, a $\Sigma \in \mathbb{R}^{m\times n}$ je 'dijagonalna' s nenegativnim dijagonalnim elementima. Osim ovog oblika dekompozicije, koristi se i skraćeni oblik

$$G = U'\Sigma' V^*,$$

pri čemu je $U' \in \mathbb{C}^{m\times n}$ matrica s ortonormiranim stupcima, a $\Sigma' = \operatorname{diag}(\sigma_1, \ldots, \sigma_n)$, $\sigma_i \geq 0$ za $i = 0, \ldots, n$, je sada stvarno dijagonalna.

Izvan matematike, u 'stvarnom' životu, SVD se koristi u procesiranju slika (rekonstrukciji, sažimanju, izoštravanju) i signala, s primjenama u medicini (CT, tj. kompjuterizirana tomografija; MR, tj. magnetna rezonancija), geoznanostima, znanosti o materijalima, kristalografiji, sigurnosti (prepoznavanje lica), izvlačenja informacija iz velike količine podataka (na primjer, LSI, tj. latent semantic indexing), ali i drugdje. Većina primjena koristi svojstvo da se iz SVD-a lako čita najbolja aproksimacija dane matrice matricom fiksnog (niskog) ranga. Čini se da je lakše reći gdje se SVD ne koristi, nego gdje se koristi, stoga se SVD često naziva i "švicarskim nožićem matričnih dekompozicija".[1]

Prvi počeci razvoja SVD-a sežu u 19. stoljeće, kad su poznati matematičari Eugenio Beltrami, Camille Jordan, James Joseph Sylvester, Erhard Schmidt i Herman Weyl pokazali njezinu egzistenciju i osnovna svojstva (za detalje pogledati [74]).

Pioniri u numeričkom računanju SVD-a su Ervand George Kogbetliantz, te Gene Golub i William Kahan, koji su razvili algoritam za računanje (bidijagonalni QR), koji je dvadeset i pet godina vladao scenom numeričkog računanja SVD-a. U to vrijeme, sveučilište Stanford (gdje je Gene Golub radio) bilo je 'glavno sjedište' za razvoj primjena SVD-a.

Početkom devedesetih godina, 'sjedište SVD-a' preseljeno je u Europu, nakon objave članka [21] o relativnoj točnosti računanja svojstvenih vrijednosti simetričnih pozitivno definitnih matrica korištenjem Jacobijeve metode. Naime, problem računanja svojstvene dekompozicije pozitivno definitne matrice i problem računanja SVD-a usko su vezani. Ako je poznata dekompozicija singularnih vrijednosti matrice G punog stupčanog ranga, $G \in \mathbb{C}^{m\times n} = U\Sigma V^*$, pri čemu je G faktor matrice A, $A = G^*G$, onda je A simetrična i pozitivno definitna i vrijedi

$$A = G^*G = V\Sigma^T U^* U\Sigma V^* = V \operatorname{diag}(\sigma_1^2, \ldots, \sigma_m^2) V^*.$$

[1]Diane O'Leary, 2006.

Matrica V je matrica svojstvenih vektora, a svojstvene vrijednosti su kvadrati singularnih vrijednosti. Stoga se algoritmi za računanje svojstvenih vrijednosti, kod kojih se transformacija vrši dvostranim (i slijeva i zdesna) djelovanjem na matricu A, mogu napisati implicitno, tako da se transformacija vrši ili zdesna na faktor G ili slijeva na faktor G^*.

U svojoj doktorskoj disertaciji Drmač [24] je napravio daljnju analizu, ne samo singularne dekompozicije računate Jacobijevim algoritmom, nego i generalizirane singularne dekompozicije (GSVD). Temeljem tih istraživanja, SVD baziran na Jacobijevim rotacijama ušao je i u numeričku biblioteku LAPACK.

U međuvremenu, gotovo sva računala postala su višejezgrena, a moderni klasteri računala za znanstveno računanje sastoje se od nekoliko tisuća do nekoliko stotina tisuća višejezgrenih procesora[2], pa standardni sekvencijalni algoritmi nipošto više nisu primjereni za numeričko računanje. Stoga se ubrzano razvijaju paralelni algoritmi koji poštuju i hijerarhijsku memorijsku strukturu odgovarajućih računala, težeći iskoristiti brzu cache memoriju za procesiranje potproblema u blokovima, na koje je moguće primijeniti BLAS-3 operacije. Ideja blokiranja je u primjeni što više (tipično, kubično u dimenziji matrice) numeričkih operacija nad podacima u brzoj memoriji. Nadalje, pojavom grafičkih procesnih jedinica namijenjenih znanstvenom računanju, kao i drugih visokoparalelnih numeričkih akceleratora (npr. Intel Xeon Phi), otvorio se novi segment istraživanja, koji poštuje njihov masivni paralelizam, s pojedinačno slabašnom snagom svake dretve u odnosu na središnji procesor.

Generaliziranu singularnu dekompoziciju (GSVD) uveli su Van Loan [77], te Paige i Saunders [62]. Definicija GSVD-a nešto je manje poznata. Ako su zadane matrice $F \in \mathbb{C}^{m\times n}$ i $G \in \mathbb{C}^{p\times n}$, za koje vrijedi

$$K = \begin{bmatrix} F \\ G \end{bmatrix}, \quad k = \operatorname{rank}(K),$$

tad postoje unitarne matrice $U \in \mathbb{C}^{m\times m}$, $V \in \mathbb{C}^{p\times p}$, i matrica $X \in \mathbb{C}^{k\times n}$, takve da je

$$F = U\Sigma_F X, \quad G = V\Sigma_G X, \quad \Sigma_F \in \mathbb{R}^{m\times k}, \quad \Sigma_G \in \mathbb{R}^{p\times k}.$$

Elementi matrica Σ_F i Σ_G su nula, osim dijagonalnih elemenata, koji su realni i nenegativni. Nadalje, Σ_F i Σ_G zadovoljavaju

$$\Sigma_F^T\Sigma_F + \Sigma_G^T\Sigma_G = I.$$

Omjeri $(\Sigma_F)_{ii}/(\Sigma_G)_{ii}$ su generalizirane singularne vrijednosti para (F, G). Ako je G punog stupčanog ranga, tada je $\operatorname{rank}(K) = n$ i generalizirane singularne vrijednosti su konačni brojevi. Ako je par (F, G) realan, onda su realne sve matrice u dekompoziciji. Odavde nadalje, zbog jednostavnoti pretpostavlja se da je par realan.

Može se pokazati da, ako je $k = n$, tada se relacija između GSVD-a i reducirane forme CS (kosinus-sinus) dekompozicije (vidjeti, na primjer, [26]) može iskoristiti za njezino računanje (pogledati, na primjer članke Stewarta [72, 73] i Suttona [75]).

Slično kao i SVD, generalizirana singularna dekompozicija ima primjene u mnogim područjima, kao što je usporedna analiza podataka vezanih uz genome [1],

[2]https://www.top500.org

nepotpuna singularna metoda rubnih elemeneata [47], ionosferna tomografija [9], ali i mnogo drugih.

GSVD para matrica (F, G) blisko je vezana s hermitskim generaliziranim svojstvenim problemom za par $(A, B) := (F^*F, G^*G)$, tako da se metode za istovremenu dijagonalizaciju para (A, B) mogu modificirati za računanje GSVD-a para (F, G). U ovoj radnji razvijen je brzi i efikasan algoritam za računanje generalizirane singularne dekompozicije realnog para (F, G).

Metoda razvijena u radnji bazirana je na algoritmu za računanje generalizirane svojstvene dekompozicije,

$$Ax = \lambda Bx, \quad x \neq \mathbf{0}, \tag{1}$$

gdje su A i B simetrične matrice, a par je definitan, tj. postoji realna konstanta μ takva da je matrica $A - \mu B$ pozitivno definitna. Članke s metodom objavili su 1960. Falk i Langemeyer [31, 32] u slabo poznatom priručniku. Kad je paralelna verzija metode testirana, pokazalo se da pati zbog problema rastuće skale stupaca matrice tijekom procesa ortogonalizacije. Treba još primijetiti da pozitivna definitnost matrice B odmah znači da je definitan i par (A, B).

Gotovo desetljeće nakon Falka i Langemeyera, Katharina Zimmermann je u svojoj doktorskoj disertaciji [81] grubo skicirala metodu za rješavanje generaliziranog svojstvenog problema (1) ako je B pozitivno definitna. Gose [34] je predložio optimalnu ne-cikličku pivotnu strategiju i dokazao globalnu konvergenciju originalne metode. Hari je u svojoj disertaciji [37], potaknut Zimmermanninom skicom metode, izveo algoritam i pokazao njegovu globalnu i kvadratičnu konvergenciju uz cikličke pivotne strategije.

Kvadratičnu konvergenciju originalne Falk–Langemeyerove metode dokazao je 1988. Slapničar u svojem magisteriju, četiri godine nakon dokaza konvergencije Hari–Zimmermann metode. Hari je u [37] pokazao ključnu vezu između Hari–Zimmermannine i Falk–Langemeyerove varijante algoritma. Ako je matrica B obostrano skalirana dijagonalnom matricom D, tako da su joj dijagonalni elementi jednaki 1 prije svakog koraka poništavanja u Falk–Langemeyerovoj metodi, dobiva se Hari–Zimmermannina metoda. Dakle, nova metoda imala je ključno svojstvo normiranosti stupaca barem jedne matrice, što se pokazalo iznimno bitnim za uspjeh algoritma (izbjegavanje skaliranja matrica tijekom procesa ortogonalizacije).

Treba reći da se GSVD može računati i na druge načine. Drmač je u [26] izveo algoritam za računanje GSVD-a para (F, G), kad je G punog stupčanog ranga. Algoritam transformira problem na samo jednu matricu, a nakon toga primjenjuje jednostrani Jacobijev SVD algoritam. Taj algoritam računa generalizirane singularne vrijednosti s malom relativnom greškom. Algoritam svođenja na jednu matricu sastoji se od tri koraka: skaliranje stupaca matrica F i G, QR faktorizacije sa stupčanim pivotiranjem već skalirane matrice G, i konačno, rješavanjem trokutastog linearnog sustava s k desnih strana. Posljednja dva koraka su sekvencijalna i vrlo ih je teško paralelizirati.

Sama ideja korištenja implicitne (tj. jednostrane) Falk–Langemeyerove metode za GSVD para (F, G), s G punog stupčanog ranga, sreće se u disertaciji Annette Deichmöller [17], međutim, tamo se ne spominju usporedbe te metode s drugim metodama.

S druge strane, algoritam za računanje GSVD-a u biblioteci LAPACK (potprogram `xGGSVD`), je modificirani Kogbetliantzov algoritam (vidjeti Paige [61]) s obveznim pretprocesiranjem (vidjeti Bai i Demmel [5]). Algoritam pretprocesiranja [6] transformira zadani matrični par (F_0, G_0) u par (F, G), takav da su F i G gornjetrokutaste, a G je i nesingularna. Ako se unaprijed zna da je G punog stupčanog ranga, i implicitna Falk–Langemeyerova i implicitna Hari–Zimmermannina metoda će raditi i bez pretprocesiranja. Ako su F i G vitke (engl. "tall and skinny"), QR factorizacija obje matrice će ubrzati ortogonalizaciju. Ako G nije punog ranga, onda treba koristiti isto pretprocesiranje kao u LAPACK-u, budući da puni stupčani rang matrice G garantira pozitivnu definitnost matrice $B := G^T G$.

* * *

U ovoj radnji razvijen je i hijerarhijski, blokirani jednostrani algoritam za računanje SVD-a. Opisani algoritam može raditi na višeprocesorskom računalu, računalnim klasterima, jednoj ili više grafičkih procesnih jedinica. Princip rada algoritma na svim arhitekturama je sličan. Posebno je opisan algoritam koji radi na grafičkim procesnim jedinicama. Struktura blokiranja reflektira razine memorijske strukture grafičke procesne jedninice. Da bi se to postiglo, razvijene su familije paralelnih pivotnih strategija za dijeljenu (engl. shared) memoriju grafičkih procesnih jedinica. Uz dodatak rasporeda po procesima, strategije se mogu koristiti i kao strategije za komuniciranje među računalnim čvorovima (bili oni grafičke procesne jedinice, jezgre procesora ili tzv. NUMA čvorovi).

Razvijeni algoritam nije hibridni, tj. centralnu procesnu jedinicu koristi samo za kontrolne svrhe, a cjelokupno računanje odvija se na grafičkoj procesnoj jedinici. Kad je zbog veličine problema potrebno, algoritam se može rasprostrijeti (skalirati) na proizvoljan broj grafičkih procesnih jedinica. Na dovoljno velikim matricama, skalabilnost je pokazana ubrzanjem od preko dva puta na četiri grafičke procesne jedinice, obzirom na jednu.

U drugom dijelu radnje opisuje se jedan način modifikacije dvostranog Hari–Zimmermanninog algoritma za računanje generalizirane svojstvene dekompozicije matričnog para (A, B), gdje su obje matrice simetrične, a B je pozitivno definitna. Implicitni algoritam računa GSVD para (F, G), pri čemu je $(A, B) := (F^T F, G^T G)$. Nadalje, pokazuje se kako treba blokirati algoritam, te kako ga paralelizirati, i u slučaju standardnih, i u slučaju grafičkih procesora.

Za trokutaste matrične parove srednje velikih dimenzija (približno 5 000), pokazano je da je već sekvencijalni, neblokirani algoritam u dvostrukoj točnosti, predložen u radnji, nekoliko desetaka puta brži no što je to LAPACK potprogram `DTGSJA` i pritom ima nešto bolju točnost, posebno za male generalizirane singularne vrijednosti. Blokiranje algoritma koje odgovara *cacheima* znatno ubrzava algoritam. Pokazuje se da je i ovaj algoritam, slično kao jednostrani Jacobijev algoritam za SVD, gotovo idealno paralelizabilan i skalabilan na računalima s dijeljenom memorijom, te da njegovo ubrzanje gotovo isključivo ovisi o broju korištenih jezgara. U vrijeme testiranja, pokazalo se da je paralelizirani i blokirani Hari–Zimmermannin algoritam preko sto puta brži od LAPACK potprograma `DTGESJA` s višedretvenim BLAS potprogramima. Varijanta algoritma za razdijeljenu (engl. distributed) memoriju namijenjena je ogromnim matricama koje ne stanu u jedan NUMA čvor.

Također, skicirana je i GPU varijanta algoritma, koja je vrlo slična jednostranom Jacobijevom algoritmu za SVD.

Disertacija završava zaključkom da su ovi algoritmi Jacobijevog tipa efikasni i skalabilni i izvrstan su izbor za računanje (G)SVD-a punih matrica na masivno paralelnim standardnim arhitekturama i na grafičkim procesnim jedinicama.

Ova doktorska disertacija bazirana je na originalnim znanstvenim radovima [55, 58], te proširena nekim novim rezultatima. Autorov doprinos u ovoj disertaciji su novi paralelni algoritmi za (G)SVD za grafičke procesne jedinice, tehnike paralelizacije, te detalji implementacije jednostranog Hari–Zimmermannina algoritma. Ostatak je zajednički rad sa Sanjom Singer i Sašom Singerom.

Ključne riječi: jednostrani algoritmi Jacobijeva tipa, (generalizirana) dekompozicija singularnih vrijednosti, paralelne pivotne strategije, grafičke procesne jedinice, (generalizirani) problem svojstvenih vrijednosti, blokiranje obzirom na cache memoriju, (hibridna) paralelizacija.

Contents

1. Introduction

Why this thesis? Apart to fulfill one of the requirements for a doctoral degree, of course. . . ? Out of a conviction that the results it summarizes are useful to a diverse audience that tries to compute the **large** singular value decompositions (SVDs) and the generalized singular value decompositions (GSVDs), or many small, similar-sized ones, arising from a plethora of the real-life problems, as *fast* as possible, and/or as *accurately* as possible, on a wide range of dense matrices, on almost *any* modern high performance computing hardware.

The main part and contribution of the thesis is contained in chapters 2 and 3, in which the one-sided Jacobi-type SVD algorithm for the graphics processing units, and the implicit Hari–Zimmermann method for the GSVD, respectively, are developed. In the following, after a short overview of the singular value decomposition, those chapters are independently introduced, while each chapter on its own starts with a summary of its contents.

The thesis concludes with some directions for the future work on fully utilizing the vectorization capabilities of the CPUs for the Jacobi-type (G)SVD algorithms.

About the singular value decomposition

The SVD is one of the most useful matrix decompositions, for theoretical as well as practical considerations. Any matrix $G \in \mathbb{C}^{m\times n}$ (to simplify the notation, assume $m \geq n$; otheriwse, take the SVD of G^* instead) can be decomposed as a product of three matrices

$$G = U\Sigma V^*,$$

where $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{n\times n}$ are unitary, and $\Sigma \in \mathbb{R}^{m\times n}$ is a 'diagonal' matrix, with the non-negative diagonal elements. Apart from this form of the decomposition, a 'truncated' form

$$G = U'\Sigma' V^*$$

is also widely used, where $U' \in \mathbb{C}^{m\times n}$ is a matrix with the orthonormal columns, and $\Sigma' = \operatorname{diag}(\sigma_1, \ldots, \sigma_n)$, $\sigma_i \geq 0$ for $i = 0, \ldots, n$, is now truly diagonal.

It is often repeated that the "SVD is the Swiss Army knife of matrix decompositions".[1] And indeed, it would be easier to enumerate the applications of numerical linear algebra as such that do *not* benefit from the SVD either directly or indirectly, than to list those that do.

Outside of the realm of mathematics, in the 'real-life' domains, the SVD is used in image (reconstruction, compression, deblurring) and signal processing, with applications in medicine (CT, i.e., computed tomography; MRI, i.e., magnetic resonance imaging), geosciences, material science, crystallography, security applications (facial recognition), information retrieval from big datasets (e.g., LSI, i.e., latent semantic

[1] Diane O'Leary, 2006.

indexing), and elsewhere. Most applications rely on a property that the SVD provides an easy method of computing the best approximation of a given matrix by a matrix of the fixed (low) rank.

The first developments of a concept of the SVD date back to the 19th century, when the well-known mathematicians Eugenio Beltrami, Camille Jordan, James Joseph Sylvester, Erhard Schmidt, and Herman Weyl showed the decomposition's existence and some of its basic properties (see [74] for details).

Alongside Ervand George Kogbetliantz, the pioneers of the numerical computation of the SVD were Gene Golub and William Kahan, who had developed an effective algorithm (the bidiagonal QR), which dominated the computational scene for twenty five years. At that time, the Stanford University (where Gene Golub was teaching) was the 'headquarters' for development of the SVD's applications.

At the begining of 1990s, the 'SVD headquarters' moved to Europe, after the paper [21] had been published, concerning the relative accuracy of the computation of the eigenvalues of Hermitian positive definite matrices by the Jacobi method.

Computation of the eigenvalue decomposition (EVD) of a positive definite matrix and computation of the SVD are closely related. If the SVD of a matrix G of the full column rank is given, $G \in \mathbb{C}^{m\times n} = U\Sigma V^*$, where G is a factor of the matrix A, $A = G^*G$, then A is a Hermitian positive definite matrix, and it holds that

$$A = G^*G = V\Sigma^T U^* U\Sigma V^* = V \operatorname{diag}(\sigma_1^2, \ldots, \sigma_m^2)V^*.$$

The eigenvalues here are the squares of the singular values, and the eigenvectors are the columns of the matrix V. Therefore, the eigendecomposition algorithms, which transform the matrix A from both the left and the right side (the "two-sided" approach), can also be rewritten in an 'implicit' way (the "one-sided" approach), such that either the factor G (i.e., its columns) is transformed from the right side, or the factor G^* (i.e., its rows) is transformed from the left side, only.

In his doctoral thesis Drmač [24] has provided a more detailed analysis, not only of the SVD computed by the Jacobi algorithm, but also of the generalized singular value decomposition (GSVD). Following that research, the SVD based on the Jacobi rotations has been included in the LAPACK numerical subroutine library.

In the meantime, almost all processors have started to be designed as multicore ones, while the modern clusters for high performance computing contain from a couple of thousands up to several hundreds of thousands of multicore CPUs[2].

Therefore, the standard sequential algorithms are no longer adequate for numerical computing. Such a trend has spurred the development of the parallel algorithms that respect and follow the hierarchical structure of the computers' memory architecture, with an intent to utilize the fast cache memory for processing the subproblems in blocks, to which the BLAS-3 operations can be applied.

The blocking idea stems from the desire to apply as many (e.g., cubically in terms of the matrix dimension) floating-point operations over the data as possible, while it resides in the fast memory. Furthermore, the recent advent of the graphical processing units specifically designed for scientific computing, and other kinds of the highly parallel numerical accelerators (e.g., Intel Xeon Phi), a whole new segment

[2]https://www.top500.org

of research has been opened, that targets that massive parallelism while bearing in mind a relatively weak computational power of each thread, compared to a typical CPU.

The one-sided Jacobi-type SVD for the GPU(s)

Graphics processing units (GPUs) have become a widely accepted tool of parallel scientific computing, but many of the established algorithms still need to be redesigned with massive parallelism in mind. Instead of multiple CPU cores, which are fully capable of simultaneously processing different operations, GPUs are essentially limited to many concurrent instructions of the same kind—a paradigm known as SIMT (single-instruction, multiple-threads) parallelism.

The SIMT type of parallelism is not the only reason for the redesign. Modern CPU algorithms rely on (mostly automatic) multilevel cache management for speedup. GPUs instead offer a complex memory hierarchy, with different access speeds and patterns, and both automatically and programmatically managed caches. Even more so than in the CPU world, a (less) careful hardware-adapted blocking of a GPU algorithm is the key technique by which considerable speedups are gained (or lost).

The introductory paper [56] proposed a non-blocked (i.e., pointwise) single-GPU one-sided Jacobi SVD algorithm. In this thesis a family of the full block [40] and the block-oriented [39] one-sided Jacobi-type algorithm variants for the ordinary singular value decomposition (SVD) and the hyperbolic singular value decomposition (HSVD) of a matrix is presented, targeting both a single GPU and multiple GPUs. The blocking of the algorithm follows the levels of the GPU memory hierarchy; namely, the innermost level of blocking tries to maximize the amount of computation done inside the fastest (and smallest) memory of the registers and manual caches. The GPU's global RAM and caches are considered by the midlevel, while inter-GPU communication and synchronization are among the issues addressed by the outermost level of blocking.

At each blocking level an instance of either the block-oriented or the full block Jacobi (H)SVD is run, orthogonalizing pivot columns or block-columns by conceptually the same algorithm at the lower level. Thus, the overall structure of the algorithm is hierarchical (or recursive) in nature and ready to fit not only the current GPUs, but also various other memory and communication hierarchies, provided that efficient, hardware-tuned implementations at each level are available.

The Jacobi method is an easy and elegant way to find the eigenvalues and eigenvectors of a symmetric matrix. In 1958 Hestenes [41] developed the one-sided Jacobi SVD method: an implicit diagonalization is performed by orthogonalizing a factor of a symmetric positive definite matrix. But, after discovery of the QR algorithm in 1961–62 by Francis and Kublanovskaya, the Jacobi algorithm seemed to have no future, at least in the sequential processing world, due to its perceived slowness [30].

However, a new hope for the algorithm has been found in its amenability to parallelization (including vectorization, as a way of computing multiple transformations at once), in its proven high relative accuracy [21], and, finally, in the emergence of the fast Jacobi SVD implementation in LAPACK, due to Drmač and Veselić [28, 29].

In the beginning of the 1970s Sameh in [65] developed two strategies for parallel execution of the Jacobi method on Illiac IV. The first of those, the modulus strategy, is still in use, and it is one of the very rare parallel strategies for which a proof of convergence exists [50].

In the mid 1980s, Brent and Luk designed another parallel strategy [12], known by the names of its creators. The same authors, together with Van Loan [13], described several parallel one-sided Jacobi and Kogbetliantz (also known as "the two-sided Jacobi") algorithms. The parallel block Kogbetliantz method is developed in [78].

In 1987 Eberlein [30] proposed two strategies, the round-robin strategy, and another one that depends on the parity of a sweep. A new efficient recursive divide-exchange parallel strategy, specially designed for the hypercube topologies (and, consequently, matrices of order 2^n) is given in [33]. This strategy is later refined by Mantharam and Eberlein in [51] to the block-recursive (BR) strategy.

Two papers by Luk and Park [50, 49] published in 1989 established equivalence between numerous strategies, showing that if one of them is convergent, then all equivalent strategies are convergent. In the same year Shroff and Schreiber [66] showed convergence for a family of strategies called the wavefront ordering, and discussed the parallel orderings weakly equivalent to the wavefront ordering, and thus convergent.

One of the first attempts to implement a parallel SVD on a GPU was a hybrid one, by Lahabar and Narayanan [48]. It is based on the Golub–Reinsch algorithm, with bidiagonalization and updating of the singular vectors performed on a GPU, while the rest of the bidiagonal QR algorithm is computed on a CPU. In MAGMA[3], a GPU library of the LAPACK-style routines, `DGESVD` algorithm is also hybrid, with bidiagonalization (`DGEBRD`) parallelized on a GPU [76], while for the bidiagonal QR, LAPACK routine `DBDSQR` is used.

In two previous papers [69, 68] the parallel one-sided Jacobi algorithms for the (H)SVD were discussed, with two and three levels of blocking, respectively. The outermost level is mapped to a ring of CPUs which communicate according to a slightly modified modulus strategy, while the inner two (in the three-level case) are sequential and correspond to the "fast" (L1) and "slow" (L2 and higher) cache levels.

At first glance a choice of the parallel strategy might seem like a technical detail, but the tests at the outermost level have shown that the modified modulus strategy can be two times faster than the round-robin strategy. That was a motivation to explore if and how even faster strategies could be constructed that preserve the accuracy of the algorithm. A class of parallel strategies is presented here, designed around a conceptually simple but computationally difficult notion of a metric on a set of strategies of the same order. These new strategies can be regarded as generalizations of the Mantharam–Eberlein BR strategy to all even matrix orders, outperforming the Brent and Luk and modified modulus strategies in the GPU algorithm.

However, a parallel strategy alone is not sufficient to achieve decent GPU performance. The standard routines that constitute a block Jacobi algorithm, like the

[3]Matrix Algebra on GPU and Multicore Architectures, http://icl.utk.edu/magma/

Gram matrix formation, the Cholesky (or the QR) factorization, and the pointwise one-sided Jacobi algorithm itself, have to be mapped to the fast, but in many ways limited, shared memory of a GPU, and to the peculiar way the computational threads are grouped and synchronized. Even the primitives that are usually taken for granted, like the numerically robust calculation of a vector's 2-norm, present a challenge on a SIMT architecture. Combined with the problems inherent in the block Jacobi algorithms, whether sequential or parallel, like the reliable convergence criterion, a successful design of the Jacobi-type GPU (H)SVD is far from trivial.

This thesis aims to show that such GPU-centric design is possible and that the Jacobi-type algorithms for a single GPU and multiple GPUs compare favorably to the present state of the art in the GPU-assisted computation of the (H)SVD. Since all computational work is offloaded to a GPU, there is no need for a significant amount of CPU $\leftrightarrow$ GPU communication, or for complex synchronization of their tasks. This facilitates scaling to a large number of GPUs, while keeping their load in balance and communication simple and predictable. While many questions remain open, the algorithms presented here might prove themselves to be a valuable choice to consider when computing the (H)SVD on the GPUs.

The implicit Hari–Zimmermann method for the GSVD

The singular value decomposition (SVD) is a widely used tool in many applications. Similarly, a generalization of the SVD for a matrix pair (F, G), the generalized SVD (GSVD), has applications in many areas, such as comparative analysis of the genome-scale expression data sets [1], incomplete singular boundary element method [47], ionospheric tomography [9], and many others.

The GSVD of a pair (F, G) is closely related to the Hermitian generalized eigenvalue problem (GEP) of a pair $(A, B) := (F^*F, G^*G)$, so the methods for simultaneous diagonalization of (A, B) can be modified to compute the GSVD of (F, G). The aim is to develop a fast and efficient parallel algorithm for the real pair (F, G).

The definition of the GSVD (see, for example, [62]), in its full generality, is as follows: for given matrices $F \in \mathbb{C}^{m\times n}$ and $G \in \mathbb{C}^{p\times n}$, where

$$K = \begin{bmatrix} F \\ G \end{bmatrix}, \quad k = \operatorname{rank}(K),$$

there exist unitary matrices $U \in \mathbb{C}^{m\times m}$, $V \in \mathbb{C}^{p\times p}$, and a matrix $X \in \mathbb{C}^{k\times n}$, such that

$$F = U\Sigma_F X, \quad G = V\Sigma_G X, \quad \Sigma_F \in \mathbb{R}^{m\times k}, \quad \Sigma_G \in \mathbb{R}^{p\times k}. \tag{1.1}$$

The elements of Σ_F and Σ_G are zeros, except for the diagonal entries, which are real and nonnegative. Furthermore, Σ_F and Σ_G satisfy

$$\Sigma_F^T\Sigma_F + \Sigma_G^T\Sigma_G = I.$$

The ratios $(\Sigma_F)_{ii}/(\Sigma_G)_{ii}$ are called the generalized singular values of the pair (F, G). If G is of full column rank, then $\operatorname{rank}(K) = n$, and the generalized singular values are finite numbers. If the pair (F, G) is real, then all matrices in (1.1) are real. From now on, it is assumed that all matrices are real.

In 1960, Falk and Langemeyer published two papers [31, 32] on the computation of the GEP,

$$Ax = \lambda Bx, \quad x \neq 0, \tag{1.2}$$

where A and B are symmetric matrices and B is positive definite. Their method was shown in 1991 by Slapničar and Hari [71] to work for definite matrix pairs. The pair (A, B) is definite if there exists a real constant μ such that the matrix $A - \mu B$ is positive definite.

Note that, if B is positive definite, then the pair (A, B) is definite. This can be proved easily by using the Weyl theorem (see, for example, [43, Theorem 4.3.1, page 181]). In this case,

$$\lambda_i(A + (-\mu)B) \geq \lambda_i(A) + \lambda_{\min}(-\mu B) = \lambda_i(A) - \mu\lambda_{\max}(B), \tag{1.3}$$

where $\lambda_i(\cdot)$ denotes the i-th smallest eigenvalue of a matrix. If A is positive definite, any $\mu \leq 0$ will make the pair definite. If A is indefinite, or negative definite, it suffices to choose $\mu < 0$ in (1.3) of such magnitude that $\lambda_{\min}(A) - \mu\lambda_{\max}(B) > 0$ holds.

Almost a decade after Falk and Langemeyer, Zimmermann in her Ph.D. thesis [81] briefly sketched a new method for the problem (1.2) if B is positive definite. Gose [34] proposed some optimal non-cyclic pivot strategies and proved the global convergence of the original method. Hari in his Ph.D. thesis [37] filled in the missing details of the method of Zimmermann, and proved its global and quadratic convergence under the cyclic pivot strategies.

The quadratic convergence of the original Falk–Langemeyer method was proved in 1988, by Slapničar in his MS thesis, four years after the proof of the convergence of the Hari–Zimmermann method, and the results were extended afterwards and published in [71]. In the same paper, Slapničar and Hari also showed the following connection between the Hari–Zimmermann and the Falk–Langemeyer variants of the method. If the matrix B is scaled (from both sides) so that its diagonal elements are equal to 1, before each annihilation step in the Falk–Langemeyer method, then the Hari–Zimmermann method is obtained.

The GSVD was introduced by Van Loan [77] and Paige and Saunders [62]. If $k = n$ in (1.1), then the relation between the GSVD and the reduced form of the CS (cosine-sine) decomposition (see, for example, [26]) could be used for the computation of the GSVD; see the papers by Stewart [72, 73] and Sutton [75].

Drmač in [26] derived an algorithm for the computation of the GSVD of a pair (F, G), with G of full column rank. This algorithm transforms the problem to a single matrix, and then applies the ordinary Jacobi SVD algorithm. The algorithm produces the generalized singular values with small relative errors. The part of the algorithm that reduces the problem to a single matrix consists of three steps: a scaling of the columns of both matrices F and G, the QR factorization with pivoting of the already scaled G, and, finally, a solution of a triangular linear system with k right-hand sides. The last two steps are inherently sequential and, therefore, hard to parallelize.

The idea of using an implicit (i.e., one-sided) version of the Falk–Langemeyer method for the GSVD of a pair (F, G), with G of full column rank, can be found in the Ph.D. thesis of Deichmöller [17], but there is no comment on how this method performs in comparison with the other methods.

On the other hand, the state of the art algorithm `xGGSVD` for computing the GSVD in LAPACK, is a Kogbetliantz-based variation of the Paige algorithm [61] by Bai and Demmel [5], with preprocessing. The preprocessing algorithm [6] transforms a given matrix pair (F_0, G_0) to a pair (F, G), such that F and G are upper triangular and G is nonsingular. If it is known in advance that G is of full column rank, both implicit methods, the Falk–Langemeyer and the Hari–Zimmermann method will work without preprocessing. But, if F and G are tall and skinny, the QR factorization of both matrices will speed up the orthogonalization. If G is not of full column rank, the same preprocessing technique (as in LAPACK) should be used by the implicit algorithm, since a full column rank G guarantees that $B := G^T G$ is positive definite.

Finally, a prototype of the implicit blocked Hari–Zimmermann method has been developed for the GPU(s), along the same lines (so much that many routines have simply been reused) as the one-sided Jacobi-type SVD algorithm from the previous chapter, and briefly sketched at the end of the main part of the thesis.

A comparison with the published work

The second chapter of the thesis is based on [55] and (unless stated by the citations otherwise or implied by the context) is a sole contribution of the candidate. The published material has been expanded, namely, by the additional considerations with regard to the parallel strategies, and by some implementation suggestions and new test results on the modern GPUs.

The third chapter is based on [58], where the published version has been expanded, most notably by an algorithm variant that targets the GPUs. That algorithm variant, as well as all software implementations (save for the modified modulus strategy and the first-level block-partitioning), the parallelization and column sorting techniques, and most of the numerical testing results are a sole contribution of the candidate, while the rest of the chapter's material has been prepared in collaboration with Sanja and Saša Singer.

2. The Jacobi-type multilevel (H)SVD algorithm for the GPU(s)

This chapter is organized as follows. In section 2.1. a brief summary of the one-sided Jacobi-type (H)SVD block algorithm variants is given. In section 2.2. new parallel Jacobi strategies are developed: nearest to row-cyclic and nearest to column-cyclic. The main part is contained in section 2.3., where a detailed implementation of a single-GPU Jacobi (H)SVD algorithm is described. In section 2.4., a proof-of-concept implementation on multiple GPUs is presented. In section 2.5., results of the numerical testing are given. Section 2.6. completes the chapter with a parallel, numerically stable procedure for computing the 2-norm of a vector.

2.1. Jacobi–type SVD algorithm

Suppose that a matrix $G \in \mathbb{F}^{m\times n}$, where $\mathbb{F}$ denotes the real ($\mathbb{R}$) or the complex ($\mathbb{C}$) field, is given. Without loss of generality, it may be assumed that $m \geq n$. If not, instead of G, the algorithm will transform G^*.

If $m \gg n$, or if the column rank of G is less than n, then the first step of the SVD is to preprocess G by the QR factorization with column pivoting [25] and, possibly, row pivoting or row presorting,

$$G = P_r Q R P_c = P_r Q \begin{bmatrix} R_0 \\ 0 \end{bmatrix} P_c, \tag{2.1}$$

where Q is unitary, $R_0 \in \mathbb{F}^{k\times n}$ is upper trapezoidal with the full row rank k, while P_r and P_c are permutations. If $k < n$, then R_0 should be factored by the LQ factorization,

$$R_0 = P_r' L Q' P_c' = P_r' \begin{bmatrix} L_0 & 0 \end{bmatrix} Q' P_c'. \tag{2.2}$$

Finally, $L_0 \in \mathbb{F}^{k\times k}$ is a lower triangular matrix of full rank. From the SVD of L_0, by (2.1) and (2.2), it is easy to compute the SVD of G. Thus, it can be assumed that the initial G is square and of full rank n, with $n \geq 2$.

The one-sided Jacobi SVD algorithm for G can be viewed as the implicit two-sided Jacobi algorithm which diagonalizes either G^*G or GG^*. Let, e.g., $H := G^*G$. Stepwise, a suitably chosen pair of pivot columns g_p and g_q of G is orthogonalized by postmultiplying the matrix $\begin{bmatrix} g_p & g_q \end{bmatrix}$ by a Jacobi plane rotation $\widehat{V}_{pq}$, which diagonalizes the 2×2 pivot matrix $\widehat{H}_{pq}$,

$$\widehat{H}_{pq} = \begin{bmatrix} h_{pp} & h_{pq} \\ h_{pq}^* & h_{qq} \end{bmatrix} = \begin{bmatrix} g_p^* g_p & g_p^* g_q \\ g_q^* g_p & g_q^* g_q \end{bmatrix} = \begin{bmatrix} g_p^* \\ g_q^* \end{bmatrix} \begin{bmatrix} g_p & g_q \end{bmatrix}, \tag{2.3}$$

such that

$$\widehat{V}_{pq}^* \widehat{H}_{pq} \widehat{V}_{pq} = \operatorname{diag}(\hat{\lambda}_p, \hat{\lambda}_q). \tag{2.4}$$

In case of convergence, after a number of steps, the product of transformation matrices will approach the set of eigenvector matrices. Let V be an eigenvector matrix of H. Then

$$\Lambda = V^* H V = (V^* G^*)(G V), \quad \Lambda = \operatorname{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n).$$

The resulting matrix GV has orthogonal columns and can be written as

$$GV = U\Sigma, \tag{2.5}$$

where U is unitary and $\Sigma = \Lambda^{1/2}$ is a diagonal matrix of the column norms of GV.

The matrix U of the left singular vectors results from scaling the columns of GV by $\Lambda^{-1/2}$, so only the right singular vectors V have to be obtained, either by accumulation of the Jacobi rotations applied to G, or by solving the linear system (2.5) for V, with the initial G preserved. The system (2.5) is usually triangular, since G is either preprocessed in such a form, or already given as a Cholesky factor. Solving (2.5) is therefore faster than accumulation of V, but it needs more memory and may be less accurate if G is not well-conditioned (see [27]).

The choice of pivot indices p, q in successive steps is essential for possible parallelization of the algorithm. Let the two pairs of indices, (p, q) and (p', q'), be called *disjoint*, or *non-colliding*, if $p \neq q$, $p' \neq q'$, and $\{p, q\} \cap \{p', q'\} = \emptyset$. Otherwise, the pairs are called *colliding*. These definitions are naturally extended to an arbitrary number of pairs. The pairs of indexed objects (e.g., the pairs of matrix columns) are disjoint or (non)colliding if such are the corresponding pairs of the objects' indices.

The one-sided Jacobi approach is better suited for parallelization than the two-sided one, since it can simultaneously process disjoint pairs of columns. This is still not enough to make a respectful parallel algorithm. In the presence of a memory hierarchy, the columns of G and V should be grouped together into block-columns,

$$G = \begin{bmatrix} G_1 & G_2 & \cdots & G_{\mathsf{b}} \end{bmatrix}, \quad V = \begin{bmatrix} V_1 & V_2 & \cdots & V_{\mathsf{b}} \end{bmatrix}. \tag{2.6}$$

In order to balance the workload, the block-columns should be (almost) equally sized.

Usually, a parallel task processes two block-columns G_{p} and G_{q}, indexed by a single pivot block-pair, either by forming the pivot block-matrix H_{pq} and its Cholesky factor R_{pq},

$$H_{\mathsf{pq}} = \begin{bmatrix} G_{\mathsf{p}}^* G_{\mathsf{p}} & G_{\mathsf{p}}^* G_{\mathsf{q}} \\ G_{\mathsf{q}}^* G_{\mathsf{p}} & G_{\mathsf{q}}^* G_{\mathsf{q}} \end{bmatrix} = \begin{bmatrix} G_{\mathsf{p}}^* \\ G_{\mathsf{q}}^* \end{bmatrix} \begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}, \quad P^* H_{\mathsf{pq}} P = R_{\mathsf{pq}}^* R_{\mathsf{pq}}, \tag{2.7}$$

or by shortening the block-columns $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$ directly, by the QR factorization,

$$\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix} P = Q_{\mathsf{pq}} \begin{bmatrix} R_{\mathsf{pq}} \\ 0 \end{bmatrix}. \tag{2.8}$$

When it is easy from the context to disambiguate a reference to the (block-)columns from a one to their (block-)indices, in what follows "the (block-)columns indexed by a pivot (block-)pair" expression will be shortened to "a (block-)pivot pair".

The diagonal pivoting in the Cholesky factorization, or analogously, the column pivoting in the QR factorization should be employed, if possible (see [69] for further discussion, involving also the HSVD case). However, the pivoting in factorizations (2.7) or (2.8) may be detrimental to performance of the parallel implementations of the respective factorizations, so the factorizations' nonpivoted counterparts have to be used in those cases (with $P = I$). Either way, a square pivot factor R_{pq} is obtained. Note that the unitary matrix Q_{pq} in the QR factorization is not needed for the rest of the Jacobi process, and it consequently does not have to be computed.

Further processing of R_{pq} is determined by a variant of the Jacobi algorithm. The following variants are advisable: *block-oriented* variant (see [39]), in which the communication (or memory access) overhead between the tasks is negligible compared to the computational costs, and *full block* variant (see [40]) otherwise.

In both variants, R_{pq} is processed by an inner one-sided Jacobi method. In the block-oriented variant, exactly one (quasi-)sweep of the inner (quasi-)cyclic[1] Jacobi method is allowed. Therefore, R_{pq} is transformed to $R'_{\mathsf{pq}} = R_{\mathsf{pq}}\widetilde{V}_{\mathsf{pq}}$, with $\widetilde{V}_{\mathsf{pq}}$ being a product of the rotations applied in the (quasi-)sweep. In the full block variant, the inner Jacobi method computes the SVD of R_{pq}, i.e., $R_{\mathsf{pq}}V_{\mathsf{pq}} = U_{\mathsf{pq}}\Sigma_{\mathsf{pq}}$. Let V'_{pq} denote the transformation matrix, either $\widetilde{V}_{\mathsf{pq}}$ from the former, or V_{pq} from the latter variant.

Especially for the full block variant, the width of the block-columns should be chosen such that R_{pq} and V'_{pq} jointly saturate, without being evicted from, the fast local memory (e.g., the private caches) of a processing unit to which the block-columns $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$ are assigned. This also allows efficient blocking of the matrix computations in (2.7) (or (2.8)) and (2.9), as illustrated in subsections 2.3.1. and 2.3.5.

Having computed V'_{pq}, the block-columns of G (and, optionally, V) are updated,

$$\begin{bmatrix} G'_{\mathsf{p}} & G'_{\mathsf{q}} \end{bmatrix} = \begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix} V'_{\mathsf{pq}}, \quad \begin{bmatrix} V'_{\mathsf{p}} & V'_{\mathsf{q}} \end{bmatrix} = \begin{bmatrix} V_{\mathsf{p}} & V_{\mathsf{q}} \end{bmatrix} V'_{\mathsf{pq}}. \tag{2.9}$$

The tasks processing disjoint pairs of block-columns may compute concurrently with respect to each other, up to the local completions of updates (2.9). A task then replaces (at least) one of its updated block-columns of G by (at least) one updated block-column of G from another task(s). Optionally, the same replacement pattern is repeated for the corresponding updated block-column(s) of V. The block-column replacements entail a synchronization of the tasks. The replacements are performed by communication or, on shared-memory systems, by assigning a new pivot block-pair to each of the tasks.

The inner Jacobi method of both variants may itself be blocked, i.e., may divide R_{pq} into block-columns of an appropriate width for the next (usually faster but smaller) memory hierarchy level. This recursive blocking principle terminates at the pointwise (nonblocked) Jacobi method, when no advantages in performance could be gained by further blocking. In that way a hierarchical (or multilevel) blocking algorithm is created, with each blocking level corresponding to a distinct communication or memory domain (see [68]).

[1]See section 2.2. for the relevant definitions.

For example, in the case of a multi-GPU system, access to the global memory (RAM) of a GPU could be identified as slow compared to the shared memory and register access, and data exchange with another GPU could be identified as slow compared to access to the local RAM. This suggests the two-level blocking for a single-GPU algorithm, and the three-level for a multi-GPU one.

The inner Jacobi method, whether blocked or not, may be sequential or parallel. Both a single-GPU and a multi-GPU algorithm are examples of a nested parallelism.

Similar ideas hold also for the HSVD. If $G \in \mathbb{F}^{m\times n}$, $m \geq n$, and $\operatorname{rank}(G) = \operatorname{rank}(GJG^*)$, where $J = \operatorname{diag}(\pm 1)$, with the number of positive signs in J already given (e.g., by the symmetric indefinite factorization with complete pivoting [70]), then the HSVD of G is (see [60, 80])

$$G = U \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^*, \quad \Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_n), \quad \sigma_1 \geq \sigma_2 \geq \cdots \sigma_n \geq 0. \tag{2.10}$$

Here, U is a unitary matrix of order m, while V is J-unitary (i.e., $V^*JV = J$) of order n. The HSVD in (2.10) can be computed by orthogonalization of either the of columns of G^* by trigonometric rotations [23], or the columns of G by hyperbolic rotations [79].

A diagonalization method for the symmetric definite (or indefinite) matrices requires only the partial SVD (or HSVD), i.e., the matrix V is not needed. With the former algorithm, the eigenvector matrix U should be accumulated, but with the latter, it is easily obtainable by scaling the columns of the final G. Thus, the hyperbolic algorithm is advantageous for the eigenproblem applications, as shown in [69]. From the HSVD of G, as in (2.10), it immediately follows the EVD of $A := GJG^*$, since (consider the case with $m = n$, for simplicity)

$$AU = (GJG^*)U = (U\Sigma V^* JV\Sigma U^*)U = U\Sigma^2 J(U^*U) = U\Sigma^2 J,$$

with the columns of U being the eigenvectors, and $\Sigma^2 J$ being the eigenvalues of A.

In the following it is assumed that $\mathbb{F} = \mathbb{R}$, but everything, save the computation of the Jacobi rotations and the hardware-imposed block sizes, is valid also for $\mathbb{F} = \mathbb{C}$.

2.2. Parallel pivot strategies

In each step of the classical, two-sided Jacobi (eigenvalue) algorithm, the pivot strategy seeks and annihilates an off-diagonal element h_{pq} with the largest magnitude. This approach has been generalized for the parallel two-sided block-Jacobi methods [7]. However, the one-sided Jacobi algorithms would suffer from a prohibitive overhead of forming and searching through the elements of $H = G^*G$. In the parallel algorithm there is an additional problem of finding $\lfloor n/2 \rfloor$ off-diagonal elements with large magnitudes, that can be simultaneously annihilated. Therefore, a cyclic pivot strategy—a repetitive, fixed order of annihilation of all off-diagonal elements of H—is more appropriate for the one-sided algorithms.

More precisely, let P_n be the set $\{(i,j) \mid 1 \leq i < j \leq n\}$ of all pivot pairs, i.e., pairs of indices of the elements in the strictly upper triangle of a matrix of order

n, and let $\tau = |\mathsf{P}_n|$ be the cardinality of P_n. Obviously, $\tau = n(n-1)/2$. A *pivot strategy* of order n is a function $\mathcal{P}_n\colon \mathbb{N} \to \mathsf{P}_n$ that associates with each step $k \geq 1$ a pivot pair $(p(k), q(k))$.

If $\mathcal{P}_n$ is a periodic function, with the fundamental period υ, then, for all $i \geq 1$, the pivot sequences $\mathsf{C}_i(\upsilon) = (\mathcal{P}_n(k) \mid (i-1)\upsilon+1 \leq k \leq i\upsilon)$, of length υ, are identical. Consider a case where such a sequence contains all the pivot pairs from P_n. Then, if $\upsilon = \tau$, $\mathcal{P}_n$ is called a *cyclic* strategy and $\mathsf{C}_i(\upsilon)$ is its i-th *sweep*. Otherwise, if $\upsilon > \tau$, $\mathcal{P}_n$ is called a *quasi-cyclic* strategy and $\mathsf{C}_i(\upsilon)$ is its i-th *quasi-sweep*. It follows that a (quasi-)cyclic strategy is completely defined by specifying its (quasi-)sweep as a pivot sequence (also called a *pivot ordering*, since it establishes a total order on P_n), the properties of which are described above. Therefore, a (quasi-)cyclic strategy (a function) can be identified with such a pivot ordering (a finite sequence).

A Jacobi method is called (quasi-)cyclic if its pivot strategy is (quasi-)cyclic. In the (quasi-)cyclic method the pivot pair therefore runs through all elements of P_n exactly (at least) once in a (quasi-)sweep, and repeats the same sequence until the convergence criterion is met.

The reader is referred to the standard terminology of equivalent, shift-equivalent and weakly equivalent strategies [66]. In what follows, a (quasi-)cyclic pivot strategy will be identified with its first (quasi-)sweep to facilitate applications of the existing results for finite sequences to the infinite but periodic ones.

A cyclic Jacobi strategy is *perfectly parallel* (p-strategy) if it allows simultaneous annihilation of as many elements of H as possible. More precisely, let

$$t = \left\lfloor \frac{n}{2} \right\rfloor, \quad s = \begin{cases} n-1, & n \text{ even}, \\ n, & n \text{ odd}, \end{cases} \tag{2.11}$$

and then exactly t disjoint pivot pairs can be simultaneously processed in each of the s parallel steps (p-steps). As the p-strategies for an even n admit more parallelism within a p-step, i.e., one parallel task more than the p-strategies for $n-1$, with the same number of p-steps in both cases, it is assumed that n is even.

A definition of a p-strategy *closest* to a given sequential strategy is now provided. The motivation was to explore whether a heuristic based on such a notion could prove valuable in producing fast p-strategies from the well-known row- and column-cyclic sequential strategies. The numerical testing (see section 2.5.) strongly supports an affirmative answer.

Let $\mathcal{O}$ be the pivot ordering[2] of a cyclic strategy of order n. Then, for each pivot pair $(i,j) \in \mathsf{P}_n$ there exists an integer k such that $(i,j) = (p(k), q(k))$, where $(p(k), q(k)) \in \mathcal{O}$. For any cyclic strategy $\mathcal{O}'$, and for each $(p'(k), q'(k)) \in \mathcal{O}'$, there is $(p(\ell(k)), q(\ell(k))) \in \mathcal{O}$, such that

$$(p'(k), q'(k)) = (p(\ell(k)), q(\ell(k))). \tag{2.12}$$

For $1 \leq k \leq \tau$, the values $\ell(k)$ are all distinct, and lie between 1 and τ, inclusive. For a fixed strategy $\mathcal{O}$, this induces a one-to-one mapping $I_{\mathcal{O}}$, from the set of all

[2]In what follows, when talking about the (quasi-)cyclic pivot strategies, their corresponding pivot orderings will be meant instead.

cyclic strategies on matrices of order n to the symmetric group $\mathsf{Sym}(\tau)$, as

$$I_{\mathcal{O}}(\mathcal{O}') = (\ell(1), \ell(2), \ldots, \ell(k), \ldots, \ell(\tau)) \in \mathsf{Sym}(\tau),$$

with $\ell(k)$ defined as in (2.12). For example, $I_{\mathcal{O}}(\mathcal{O}) = (1, 2, \ldots, k, \ldots, \tau)$, i.e., the identity permutation.

Definition 2.1. *For any two cyclic strategies, $\mathcal{O}_1$ and $\mathcal{O}_2$, let it be said that $\mathcal{O}_1$ is* closer *to $\mathcal{O}$ than $\mathcal{O}_2$, and let that be denoted by $\mathcal{O}_1 \preceq_{\mathcal{O}} \mathcal{O}_2$, if $I_{\mathcal{O}}(\mathcal{O}_1) \preceq I_{\mathcal{O}}(\mathcal{O}_2)$, where $\preceq$ stands for the lexicographic ordering of permutations.*

The relation "strictly closer to $\mathcal{O}$", denoted by $\prec_{\mathcal{O}}$, is defined similarly. Note that $\preceq_{\mathcal{O}}$ is a total order on the finite set of all cyclic strategies with a fixed n, and therefore, each non-empty subset (e.g., a subset of all p-strategies) has a least element. Now, take $\mathcal{O} \in \{\mathcal{R}_n, \mathcal{C}_n\}$, where $\mathcal{R}_n$ and $\mathcal{C}_n$ are the row-cyclic and the column-cyclic strategies, respectively. Then there exists a unique p-strategy $\mathcal{R}_n^{\|}$ (resp. $\mathcal{C}_n^{\|}$) that is closest to $\mathcal{R}_n$ (resp. $\mathcal{C}_n$).

Interpreted in the graph-theoretical setting, a task of finding the closest p-strategy amounts to a recursive application of an algorithm for generating all maximal independent sets (MIS) in lexicographic order (see, e.g., [45]). Let G be a simple graph with the vertices enumerated from 1 to τ, representing pivot pairs from a prescribed cyclic strategy $\mathcal{O}_n$, and the edges denoting that two pivot pairs collide (share an index). Note that $|\mathrm{MIS}(\mathsf{G})| \le t$, where t is defined by (2.11). Then a $\mathrm{MIS}(\mathsf{G})$ with t vertices is an admissible p-step, and vice versa. The same holds for the graph $\mathsf{G}' = \mathsf{G} \setminus S$, where S is any admissible p-step.

Since any permutation of pivot pairs in a p-step generates an equivalent (called step-equivalent) p-strategy, the vertices in each MIS can be assumed to be sorted in ascending order. With a routine `next_lex`, returning the lexicographically next MIS with t vertices (or $\emptyset$ if no such sets are left), Algorithm 1 always produces $\mathcal{O}_n^{\|}$, the p-strategy closest to $\mathcal{O}_n$. Note that, at the suitable recursion depths, `next_lex` could prepare further candidates in parallel with the rest of the search, and parallel searches could also be launched (or possibly canceled) on the waiting candidates.

Algorithm 1, however optimized, might still not be feasible even for the off-line strategy generation, with n sufficiently large. However, there are two remedies: first, no large sizes are needed due to the multi-level blocking, and, second, it will be shown in what follows that it might suffice to generate $\mathcal{R}_n^{\|}$ (or $\mathcal{C}_n^{\|}$) only for $n = 2o$, with o odd.

Lemma 2.2. *For all even n, the sequence of pivot pairs*

$$S_n^{(1)} = ((2k-1, 2k) \mid 1 \le k \le n/2)$$

is the first p-step of $\mathcal{R}_n^{\|}$ and $\mathcal{C}_n^{\|}$.

Proof. Note that $S_n^{(1)}$ is an admissible p-step, i.e., there exists a p-strategy having $S_n^{(1)}$ as one of its p-steps. For example, the Brent and Luk strategy starts with it.

The first pivot pair in $\mathcal{R}_n$ and $\mathcal{C}_n$ is $(1, 2)$, i.e., $(2k-1, 2k)$ for $k = 1$. If all pivot pairs in $\mathcal{R}_n$ or $\mathcal{C}_n$ containing indices 1 or 2 are removed, the first pivot pair in the

Algorithm 1: MIS-based generation of the p-strategy $\mathcal{O}_n^{\parallel}$ closest to $\mathcal{O}_n$.

```
Description: Input: the graph $\mathsf{G}$ induced by $\mathcal{O}_n$. Output: $\mathcal{O}_n^{\parallel}$ (initially $\emptyset$).
boolean gen_strat(in G);
begin
  if $\mathsf{G} = \emptyset$ then return true;            // no more pivot pairs (success)
  begin loop
    $S \leftarrow$ next_lex(G);        // take a lexicographically next MIS...
    if $S = \emptyset$ then return false;          // ...but there are none; fail
    append $S$ to $\mathcal{O}_n^{\parallel}$;           // ...else, $S$ is a new p-step candidate
    if gen_strat($\mathsf{G} \setminus S$) then return true;         // try recursively...
    remove $S$ from the back of $\mathcal{O}_n^{\parallel}$;         // ...and backtrack if failed
  end loop;
end
```

remaining sequence is $(3,4)$, i.e., $(2k-1,2k)$ for $k=2$. Inductively, after selecting the pivot pair $(2\ell-1,2\ell)$, with $\ell < n/2$, and removing all pivot pairs that contain $2k-1$ or $2k$, for all $1 \le k \le \ell$, the first remaining pivot pair is $(2\ell'-1,2\ell')$ for $\ell' = \ell+1$. $\square$

A matrix of order $2n$ can be regarded at the same time as a block matrix of order n with 2×2 blocks (see Figure 2.1). As a consequence of Lemma 2.2, after the first p-step of either $\mathcal{R}_{2n}^{\parallel}$ or $\mathcal{C}_{2n}^{\parallel}$ (i.e., $S_{2n}^{(1)}$), the diagonal 2×2 blocks are diagonalized, and the off-diagonal blocks are yet to be annihilated.

Once the diagonal blocks have been diagonalized, it is easy to construct the closest block p-strategy $\widetilde{\mathcal{O}}_{2n}^{\parallel}$ from $\mathcal{O}_n^{\parallel}$, since each pivot pair of $\mathcal{O}_n^{\parallel}$ corresponds uniquely to an off-diagonal 2×2 block. A p-step of $\mathcal{O}_n^{\parallel}$ is expanded to two successive p-steps of $\widetilde{\mathcal{O}}_{2n}^{\parallel}$. The expansion procedure is given by Algorithm 2, for $\mathcal{O}_n \in \{\mathcal{R}_n, \mathcal{C}_n\}$, and illustrated, for $n=6$ and $\mathcal{O}_n = \mathcal{R}_n$, with Figure 2.1. Note that a pivot pair of $\mathcal{O}_n^{\parallel}$ contributes two pairs, $(\textsc{nw},\textsc{se})$ and either $(\textsc{ne},\textsc{sw})$ or $(\textsc{sw},\textsc{ne})$, of non-colliding and locally closest pivot pairs in its corresponding block.

It is trivial to show that, with $\mathcal{O}_n^{\parallel}$ given, the p-strategy $\widetilde{\mathcal{O}}_{2n}^{\parallel}$ generated by Algorithm 2 is indeed the closest *block* p-strategy; any other such $\mathcal{S}_{2n}^{\parallel} \prec_{\mathcal{O}} \widetilde{\mathcal{O}}_{2n}^{\parallel}$ would induce, by the block-to-pivot correspondence, a strategy $\mathcal{S}_n^{\parallel} \prec_{\mathcal{O}} \mathcal{O}_n^{\parallel}$, which is impossible. Moreover, it has been verified that, for $n \le 18$ and both $\mathcal{R}_n$ and $\mathcal{C}_n$ strategies, $\widetilde{\mathcal{O}}_{2n}^{\parallel} = \mathcal{O}_{2n}^{\parallel}$, and although lacking a rigorous proof, it can be claimed that the same holds for all even n. Therefore, as a tentative corrolary, to construct $\mathcal{O}_m^{\parallel}$, for $\mathcal{O}_m \in \{\mathcal{R}_m, \mathcal{C}_m\}$ and $m = 2^k o$, with $k > 1$ and o odd, it would suffice to construct $\mathcal{O}_n^{\parallel}$, $n = 2o$, and apply, $k-1$ times, Algorithm 2.

For example, a three-level blocking algorithm for four GPUs and a matrix of order $15 \cdot 1024$ requires $\mathcal{O}_8^{\parallel}$, $\mathcal{O}_{240}^{\parallel}$, and $\mathcal{O}_{32}^{\parallel}$ strategies. To find $\mathcal{O}_{240}^{\parallel}$, it suffices to construct $\mathcal{O}_{30}^{\parallel}$, and expand (i.e., duplicate) it 3 times, since $240 = 2^3 \cdot (2\cdot 15)$. Thus, the $\mathcal{O}_m^{\parallel}$ strategies should be pretabulated once, for the small, computationally feasible orders m, and stored into a code library for future use. The expansion procedure

Algorithm 2: Expansion of $\mathcal{O}_n^{\|}$ to $\widetilde{\mathcal{O}}_{2n}^{\|}$ for $\mathcal{O}_n \in \{\mathcal{R}_n, \mathcal{C}_n\}$.

```
Description: Input: $\mathcal{O}_n^{\|}$, $\mathcal{O}_n \in \{\mathcal{R}_n, \mathcal{C}_n\}$. Output: $\widetilde{\mathcal{O}}_{2n}^{\|}$.
             $S_n^{(i)}$ is the $i$-th p-step of $\mathcal{O}_n^{\|}$, and $S_{2n}^{(i)}$ is the $i$-th p-step of $\widetilde{\mathcal{O}}_{2n}^{\|}$.
$S_{2n}^{(1)} \leftarrow ((2k-1, 2k) \mid 1 \le k \le n)$;
for $i \leftarrow 2$ to $2n-1$ do                                   // construct $S_{2n}^{(i)}$
    $S_{2n}^{(i)} = \emptyset$;
    foreach $(p,q) \in S_n^{(i \,\mathbf{div}\, 2)}$ do
        if even($i$) then
            $\mathrm{NW} = (2p-1, 2q-1)$;   $\mathrm{SE} = (2p, 2q)$;   append $(\mathrm{NW}, \mathrm{SE})$ to $S_{2n}^{(i)}$;
        else
            $\mathrm{NE} = (2p-1, 2q)$;   $\mathrm{SW} = (2p, 2q-1)$;
            if $\mathcal{O}_n = \mathcal{R}_n$ then append $(\mathrm{NE}, \mathrm{SW})$ to $S_{2n}^{(i)}$ else append $(\mathrm{SW}, \mathrm{NE})$ to $S_{2n}^{(i)}$;
        end if
    end foreach
end for
```

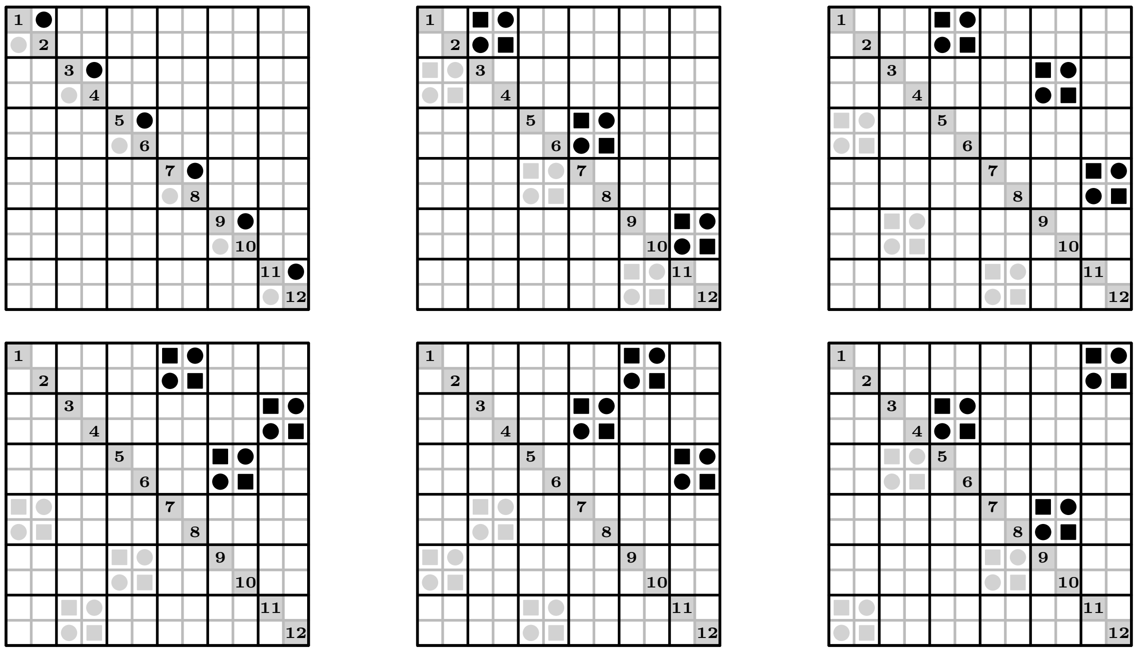


Figure 2.1: Expansion of $\mathcal{R}_6^{\|}$ to $\mathcal{R}_{12}^{\|}$, according to Algorithm 2. From left to right: the black disks represent the odd p-steps, while the black squares stand for the even p-steps.

can be performed at run-time, when the size of input is known.

The strategies just described progress from the diagonal of a matrix outwards. However, if the magnitudes of the off-diagonal elements in the final sweeps of the two-sided Jacobi method are depicted, a typical picture [29, page 1349] shows that the magnitudes rise towards the ridge on the diagonal. That was a motivation to explore whether a faster decay of the off-diagonal elements far away from the diagonal could be reached by annihilating them first, and the near-diagonal elements last. This change of annihilation order is easily done by reverting the order of pivot pairs in a sweep of $\widetilde{\mathcal{R}}_n^{\|}$ and $\widetilde{\mathcal{C}}_n^{\|}$. Formally, a *reverse* of the strategy $\mathcal{O}_n$ is the strategy[3] $\reflectbox{$\mathcal{O}$}_n$, given by

$$\reflectbox{$\mathcal{O}$}_n := ((p(\tau - k + 1), q(\tau - k + 1)) \mid 1 \leq k \leq \tau),$$

where $\mathcal{O}_n = ((p(k), q(k)) \mid 1 \leq k \leq \tau)$. Thus, $\reflectbox{$\mathcal{R}$}_n^{\|}$ and $\reflectbox{$\mathcal{C}$}_n^{\|}$ progress inwards, ending with $S_n^{(1)}$ reversed. The reverses of both $\mathcal{R}_n^{\|}$ and $\widetilde{\mathcal{R}}_n^{\|}$ (resp. $\mathcal{C}_n^{\|}$ and $\widetilde{\mathcal{C}}_n^{\|}$) are tentatively denoted by the same symbol.

For $m = 2^k$, both $\mathcal{R}_m^{\|}$ and $\mathcal{C}_m^{\|}$ can be generated efficiently by Algorithm 1, since (as an empirical, but not yet proven or well understood fact) no backtracking occurs. In this special case it holds that $\mathcal{R}_m^{\|} = \widetilde{\mathcal{R}}_m^{\|}$, $\mathcal{C}_m^{\|} = \widetilde{\mathcal{C}}_m^{\|}$, and $\mathcal{R}_m^{\|}$ is step-equivalent to $\mathcal{C}_m^{\|}$. The former claims are verified for $k \leq 14$.

The respective reverses, $\reflectbox{$\mathcal{R}$}_m^{\|}$ and $\reflectbox{$\mathcal{C}$}_m^{\|}$, operate in the same block-recursive fashion (preserved by Algorithm 2) of the Mantharam–Eberlein BR strategy [51], i.e., processing first the off-diagonal block, and then simultaneously the diagonal blocks of a matrix. It follows that all three strategies are step-equivalent. Thus, $\reflectbox{$\mathcal{R}$}_n^{\|}$ and $\reflectbox{$\mathcal{C}$}_n^{\|}$ can be regarded as the generalizations of the BR strategy to an arbitrary even order n, albeit lacking a simple communication pattern. Conversely, for the power-of-two orders, $\reflectbox{$\mathcal{R}$}_m^{\|}$ and $\reflectbox{$\mathcal{C}$}_m^{\|}$ might be replaced by the BR strategy with a hypercube-based communication.

2.2.1. Generating the Jacobi p-strategies

Finding a single new class of p-strategies is an extremely difficult task. It requires some ingenuity and a lot of luck, since there is no recipe for even starting to tackle the problem. Ideally, such a new class should be easy to generate, should preserve or enhance both the speed and the accuracy of the algorithms that employ it, and should be provably convergent.

The closest semblances of a recipe are contained in two papers [54, 66], by Nazareth and Shroff and Schreiber, respectively. The former one gives an effective algorithm for creating a family of the similar classes of the Jacobi strategies, but only the sequential ones. The latter offers a way to prove convergence of a parallel strategy, by reducing it to a wavefront ordering by a series of the equivalence relations, i.e., the convergence-preserving transformations of the Jacobi orderings. But both papers stop short of giving an effective procedure for generating a yet unseen p-strategy, or for proving convergence of a strategy that is not weakly equivalent to a wavefront one.

Let it first be shown that the Nazareth's algorithm [54, Procedure P] generates only the sequential strategies for a nontrivial matrix order.

Proposition 2.3. *For a matrix order $n > 3$, the procedure P from [54] generates only the sequential Jacobi strategies.*

[3]Here, the reverse is denoted by a mirror image of the strategy's symbol, but it is also common to denote the reverse of $\mathcal{O}$ by $\mathcal{O}^{\leftarrow}$ or $\mathcal{O}_{\leftarrow}$.

Proof. In the step B of the procedure P, the two nonempty sets of consecutive indices, $G_1 = \{k, k+1, \ldots, \ell\}$ and $G_2 = \{\ell+1, \ldots, m-1, m\}$, are obtained. Then, in the step C, a list of pairs L is formed by picking any member of G_1 and pairing it with every member of G_2 (taken in any order), or vice versa.

If either $\mu = |G_1| > 1$ or $\nu = |G_2| > 1$, the list L will contain a sublist, L', of the colliding consecutive pairs of the form (p, q_1), (p, q_2), $\ldots$, (p, q_ν) (or (p, q_μ)), which cannot be present in the same p-step. Note that $n > 3$ implies that $\mu > 1$ or $\nu > 1$ in the first level of a recursive call to P. Also, it is implied that either $\mu > 2$ or $\nu > 2$, since $\mu + \nu = n > 3$. Therefore, more than 2 elements will exist in L', which entails that L' cannot be split such that the first part of it belongs to one p-step, and the rest to the following p-step. $\square$

Now, a question could be made if the following principle might lead to a development of a "decent" parallel strategy: take a provably convergent, sequential one, and start applying all equivalence transformations known. After each transformation having been applied, check if the new strategy is perfectly parallel; if not, keep on trying.

But leaving the sheer combinatorial complexity of such an approach asside, for all but the very small matrix orders, a question on generalization still remains: given a p-strategy for order n, how to efficiently generate a p-strategy for order $m \neq n$, that operates in a fashion similar enough to the given one that they both may be considered to belong to the same class? Therefore, a complementary set of tools is needed; not only a number of equivalence transformations for an arbitrary, albeit fixed matrix order, but a set of prinicples that can "expand" or "contract" a given convergent (p-)strategy, and still obtain a convergent (p-)strategy as a result. The arsenal might include, e.g., the duplication principle from Algorithm 2, if it can be proven to posses such a property.

Note that the duplication principle will leave a lot of matrix orders uncovered. A way of "interpolating" between them, e.g., generating a p-strategy for even n, where $2^k < n < 2^{k+1}$, from the ones for orders 2^k and 2^{k+1}, would instantly generalize the Mantharam–Eberlein p-strategies to any even order.

For now, let those aims remain labeled as the future work, and let the generation of the "closest-to" p-strategies from the previous section be explained in more detail.

The MIS-based generation of $\mathcal{R}^{\parallel}$

For illustration of the graph-based approach, via maximal independent sets, to generate $\mathcal{R}^{\parallel}$ (similarly for $\mathcal{C}^{\parallel}$), take the first nontrivial even matrix order, $n = 4$ (for $n = 2$, there is only one Jacobi strategy, which is both $\mathcal{R}_2^{\parallel}$ and $\mathcal{C}_2^{\parallel}$).

Then, form a *collide graph* G_4 for $\mathcal{R}_4$, whose vertices are the pivot pairs taken from $\mathcal{R}_4$, enumerated in order of their appearance in the serial strategy. Note that there are $n(n-1)/2$ vertices for a cyclic strategy of order n. There exists an edge in G_4 between a vertex $(p, q) = v$ and a vertex $(p', q') = v' \neq v$ if and only if the pivot pairs v and v' collide (i.e., share an index). The first subfigure of Figure 2.2 shows G_4 with zero-based indexing of both the vertices and the matrix rows/columns.

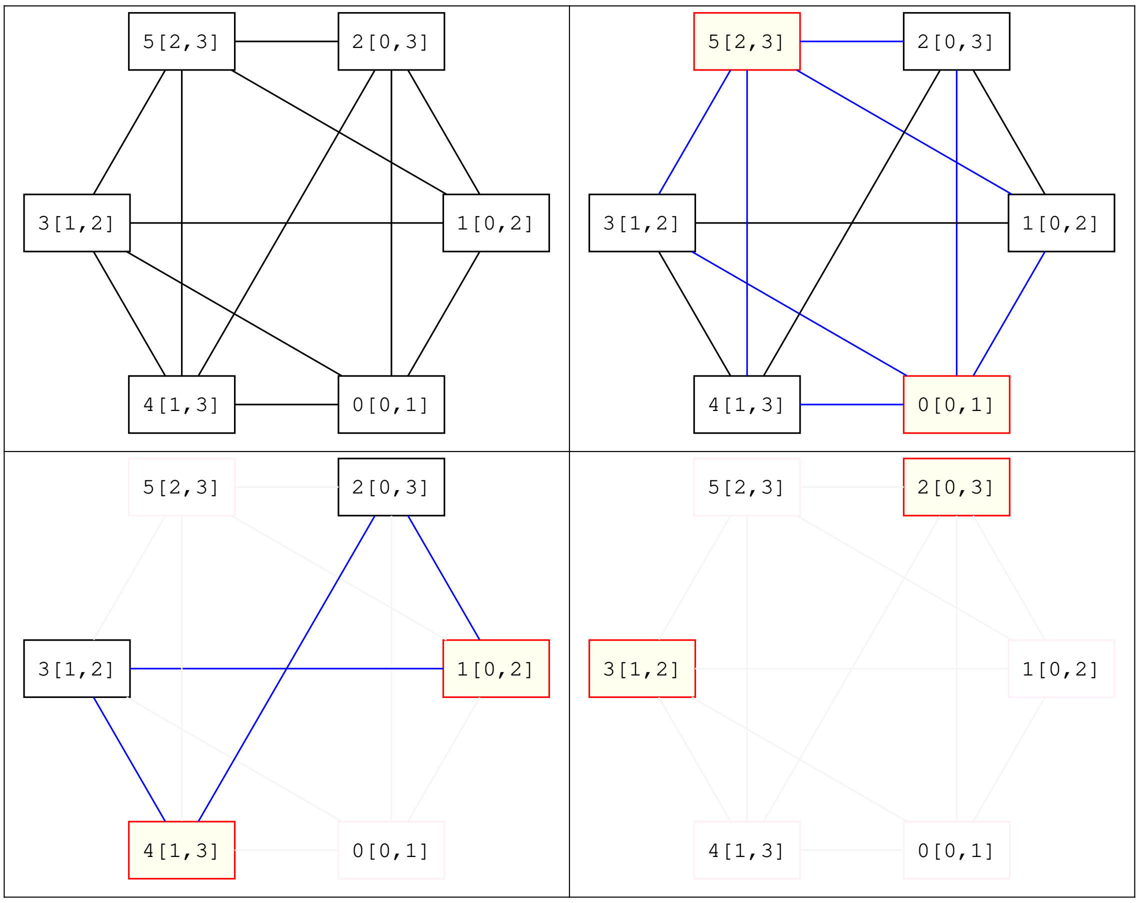


Figure 2.2: Generating the parallel steps of $\mathcal{R}_4^{\|}$; left to right, top to bottom.

In general, the graph density of G_n does not depend on the underlying cyclic Jacobi strategy. To show that, consider the following Proposition 2.4.

Proposition 2.4. *Given a matrix of order n, and a pivot pair (p, q), with $p < q$, the number of pivot pairs colliding with (and including) (p, q), denoted by $CP(n)$, does not depend on either p or q, and is equal to $2(n - 1) - 1$.*

Proof. Take a set of all the elements of the matrix that have either p or q or both as at least one of their indices. That set comprises of (and only of) the rows p and q, and the columns p and q. There are n elements in each row and in each column, which makes for $4n$ elements, but the ones that are in the intersections of the row p or q with the column p or q, i.e., (p, p), (p, q), (q, p), (q, q), are thus counted twice, so the total number of elements indexed by either p or q is $4n - 4$.

As those elements on the diagonal, i.e., (p, p) and (q, q), are of no interest here, there remain $(4n - 4) - 2$ elements, evenly distributed in the strictly upper and in the strictly lower triangle of the matrix. If only those elements in the strictly upper triangle are counted, that leaves us with $(4(n-1)-2)/2 = 2(n-1)-1$ pivot pairs, as claimed. $\square$

From that it immediately follows that all vertices of G_n have the same degree, that depends on n only, as shown in Corollary 2.5.

Corollary 2.5. *The degree of a vertex of* G_n *is equal to* $\delta_n := CP(n) - 1 = 2(n-2)$.

Proof. For an arbitrary vertex of G_n, take it out of the set of the pivot pairs colliding with it and apply Proposition 2.4. □

For example, $\delta_4 = 4$, as it can be seen on Figure 2.2; $\delta_6 = 8$, as on Figure 2.3; and $\delta_{10} = 16$, as on Figure 2.8. Corollary 2.6 ends these observations, and it will be used in the optimized generation procedure shown in Listing 1.

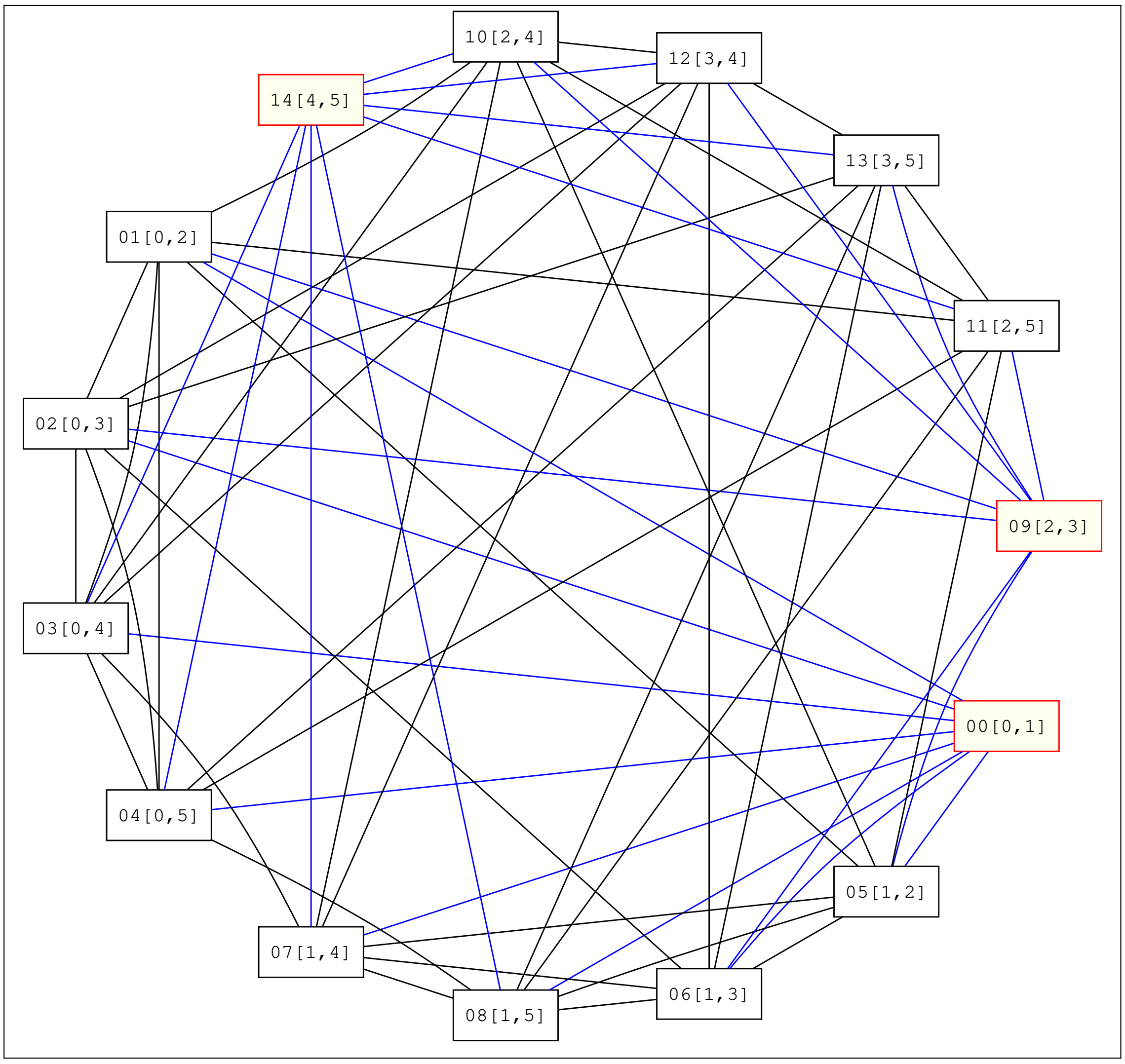


Figure 2.3: Generating the first parallel step of $\mathcal{R}_6^{\|}$.

Corollary 2.6. *Given a matrix of order* n, *and a pivot pair* (p, q), *with* $p < q$, *the number of pivot pairs non-colliding with* (p, q), *denoted by* $NCP(n)$, *is equal to* $n(n-1)/2 - (2(n-1) - 1)$.

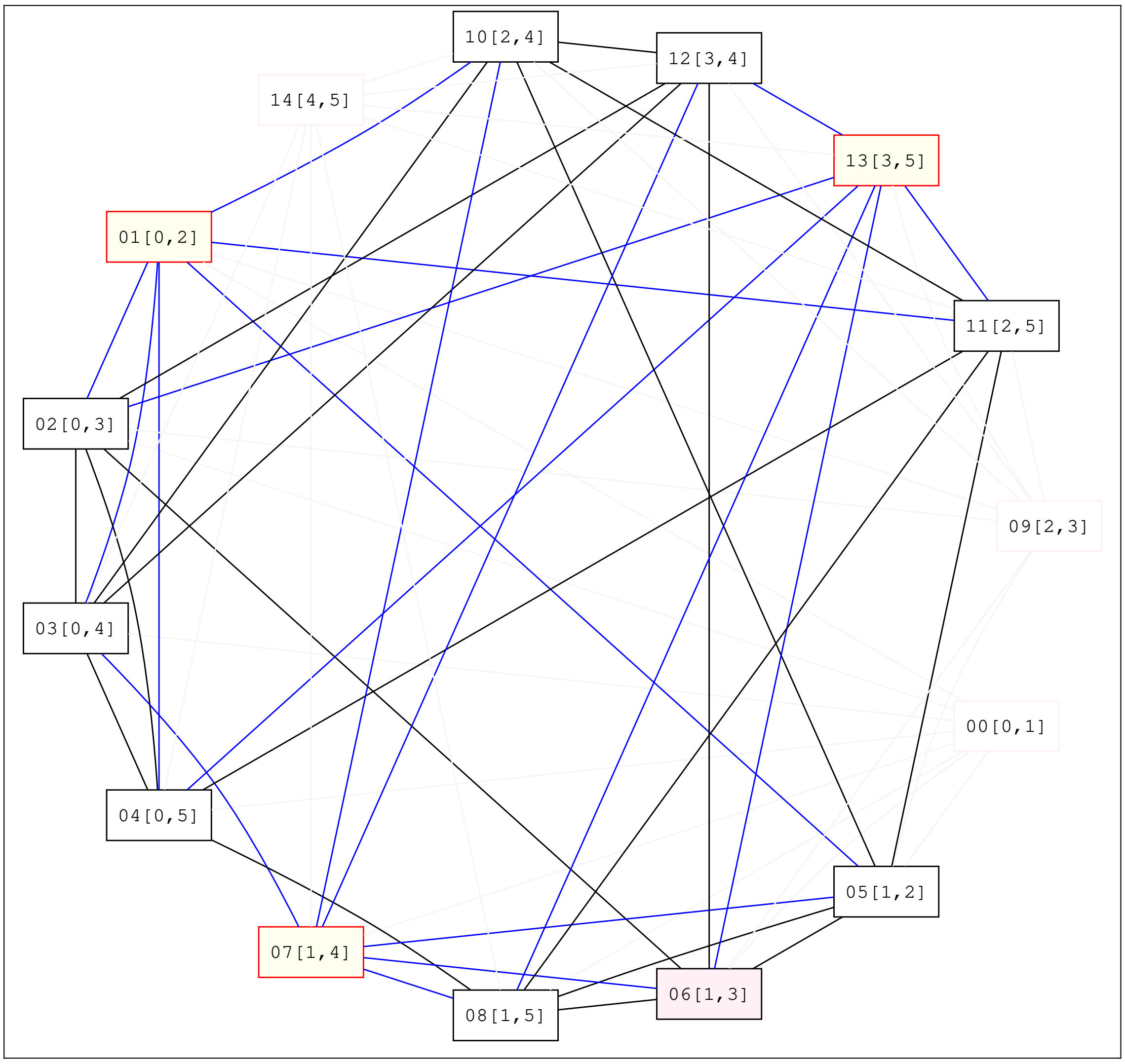


Figure 2.4: Generating the second parallel step of $\mathcal{R}_6^{\|}$.

Proof. The number of pivot pairs for a matrix of order n is a number of elements in the strictly upper triangle of the matrix, which is $n(n-1)/2$. Subtracting from that the number of pivot pairs colliding with (p,q), as per Proposition 2.4 above, concludes the proof. $\square$

The Figures 2.2, 2.3, 2.4, 2.5, 2.6, and 2.7 should be interpreted as follows. The vertices with a black border and a white background are “inactive” ones, that are out of consideration in a particular step of Algorithm 1. The vertices with a red border and a yellow background are the “active” ones, that are to form the next maximal independent set (i.e., a p-step). The edges incident to them are colored blue instead of black, and are to be removed, alongside the vertices, to progress to the next step. The vertices and edges of a very pale color are those already removed in some of the previous steps. Finally, the vertices with a pink background are the ones where the algorithm “backtracked”. They were considered to form a p-step,

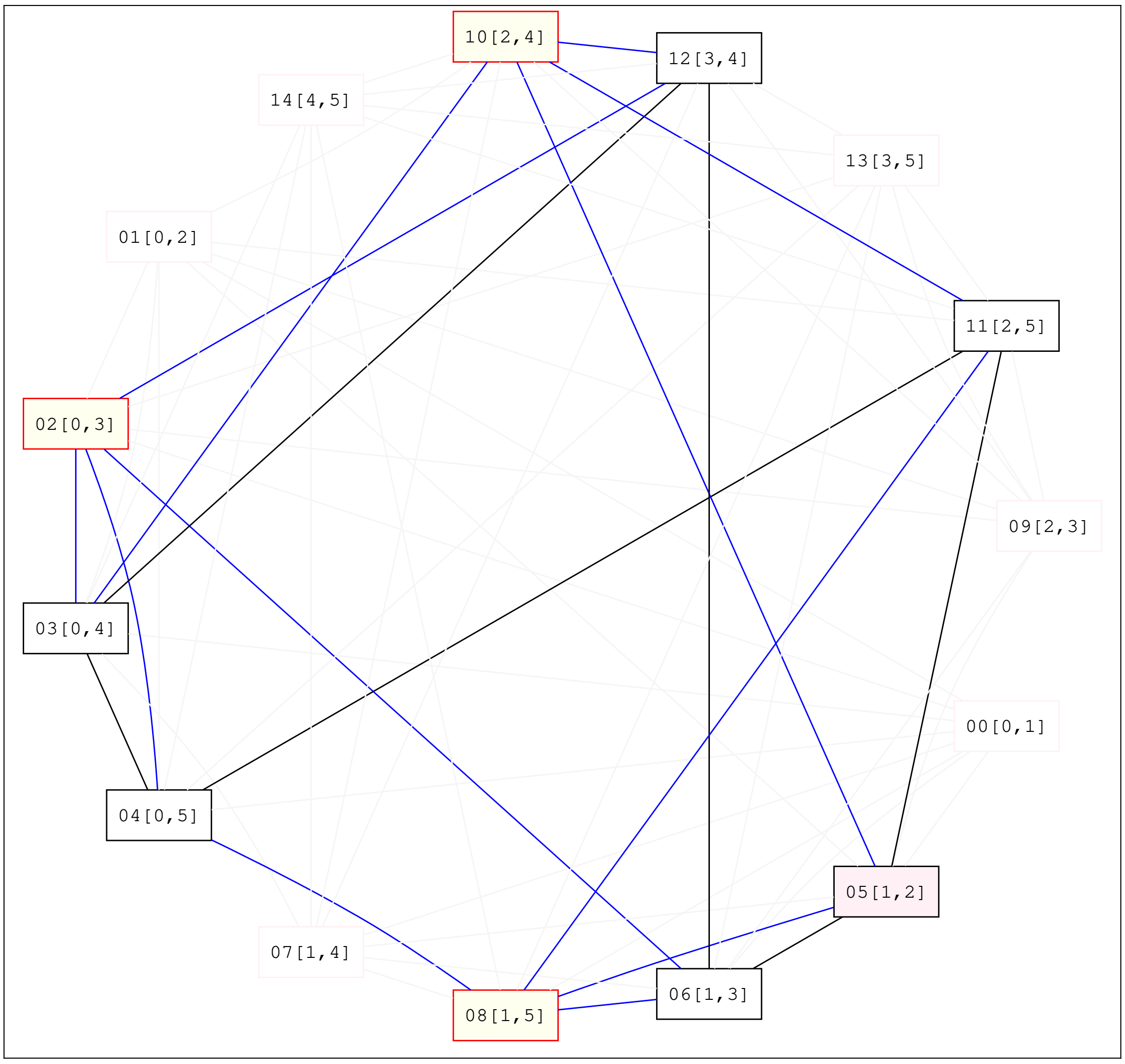


Figure 2.5: Generating the third parallel step of $\mathcal{R}_6^{\|}$.

but such a choice would lead to a MIS with less than $n/2$ pivot pairs, and therefore had to be discarded, as it will be detailed below.

For $n = 4$, Figure 2.2 shows that the vertices 0 and 5 form the first MIS. Starting from the vertex 0, i.e., from the pivot pair $(0, 1)$, the first independent vertex is 5, i.e., the first non-colliding pivot pair is $(2, 3)$. After the vertices 0 and 5 have been removed, from the reduced graph the vertex 1 is taken and paired with the first independent vertex 4. Finally, after the vertices 1 and 4 have also been removed, there are vertices 2 and 3 left, without any edges, so they can form the last p-step. Here, there is no backtracking involved, i.e., a "greedy" approach to choosing the vertices to complete a MIS works in every step. It seems that the greedy method also works for n being any power of 2, since no backtracking has been observed in all such instances tested.

A more difficult situation happens with $n = 6$, and all other non-power-of-2 instances tried. Generating the first p-step, as shown on Figure 2.3, proceeds as

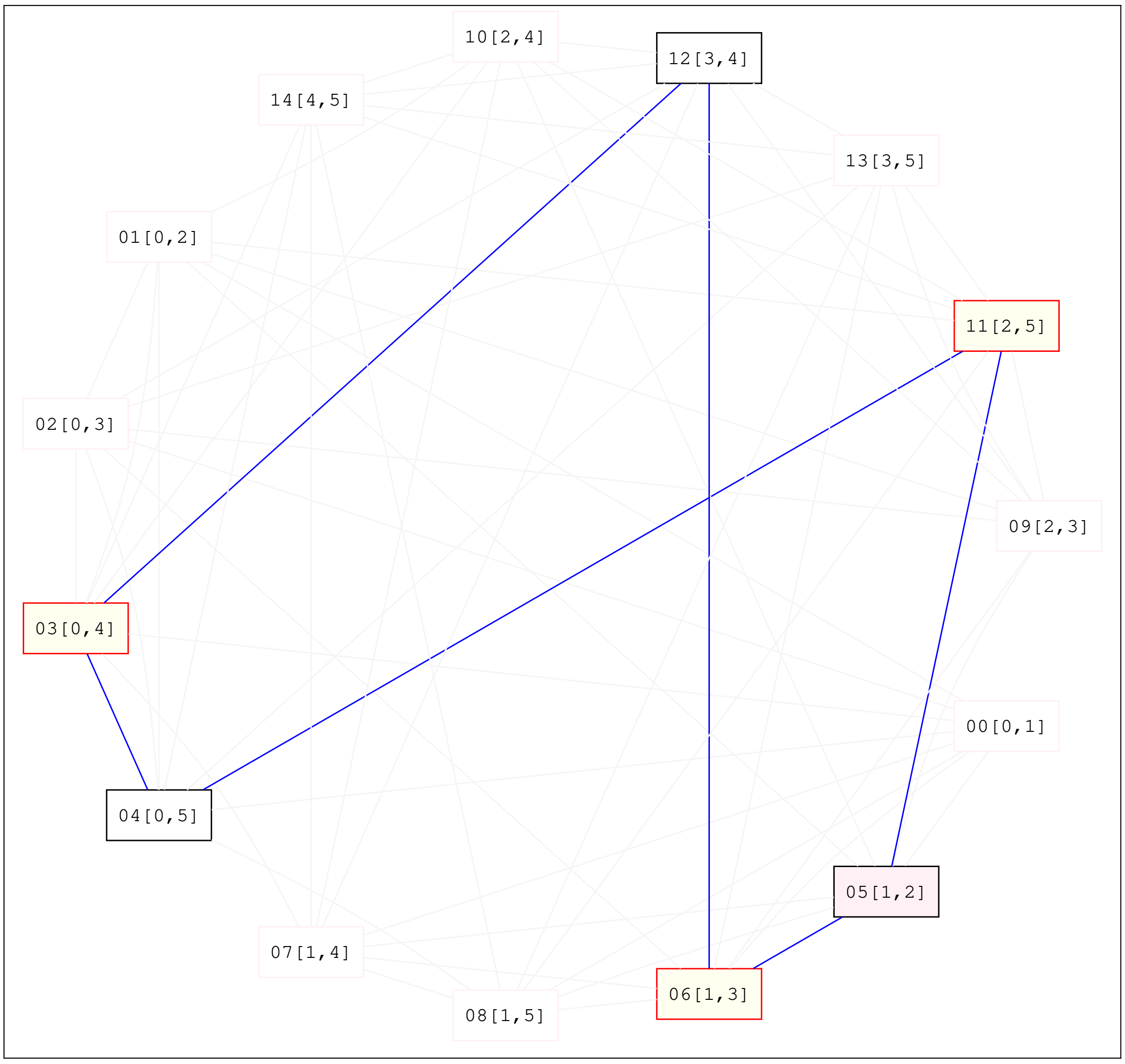


Figure 2.6: Generating the fourth parallel step of $\mathcal{R}_6^{\|}$.

explained, starting from the vertex 0 and greedily adding the independent vertices. But a backtrack is needed in the second step (see Figure 2.4). After taking the vertex 1, the next candidate should be the vertex 6. But if so, then all the indices 0, 1, 2, and 3 would be exhausted, and there would be no non-colliding pivot left to complete a MIS with three vertices. Therefore, the vertex 7 is chosen instead, and that suffices to proceed as usual.

Should no MIS have been completed in a particular step (not a case here), the whole step should be aborted by backtracking. Then, the previous MIS would have to be replaced by a lexicographically greater one, if possible. If not, the backtracking would propagate yet another step back, and so on. The procedure cannot ultimately fail, though, since the p-strategy sought for exists and will be found eventually.

The backtracking happens again in generation of the third p-step (see Figure 2.5). Here, starting from the vertex 2, the next candidate is the vertex 5, but it has to be discarded for reasons similar to above. However, the vertex 6 collides with the

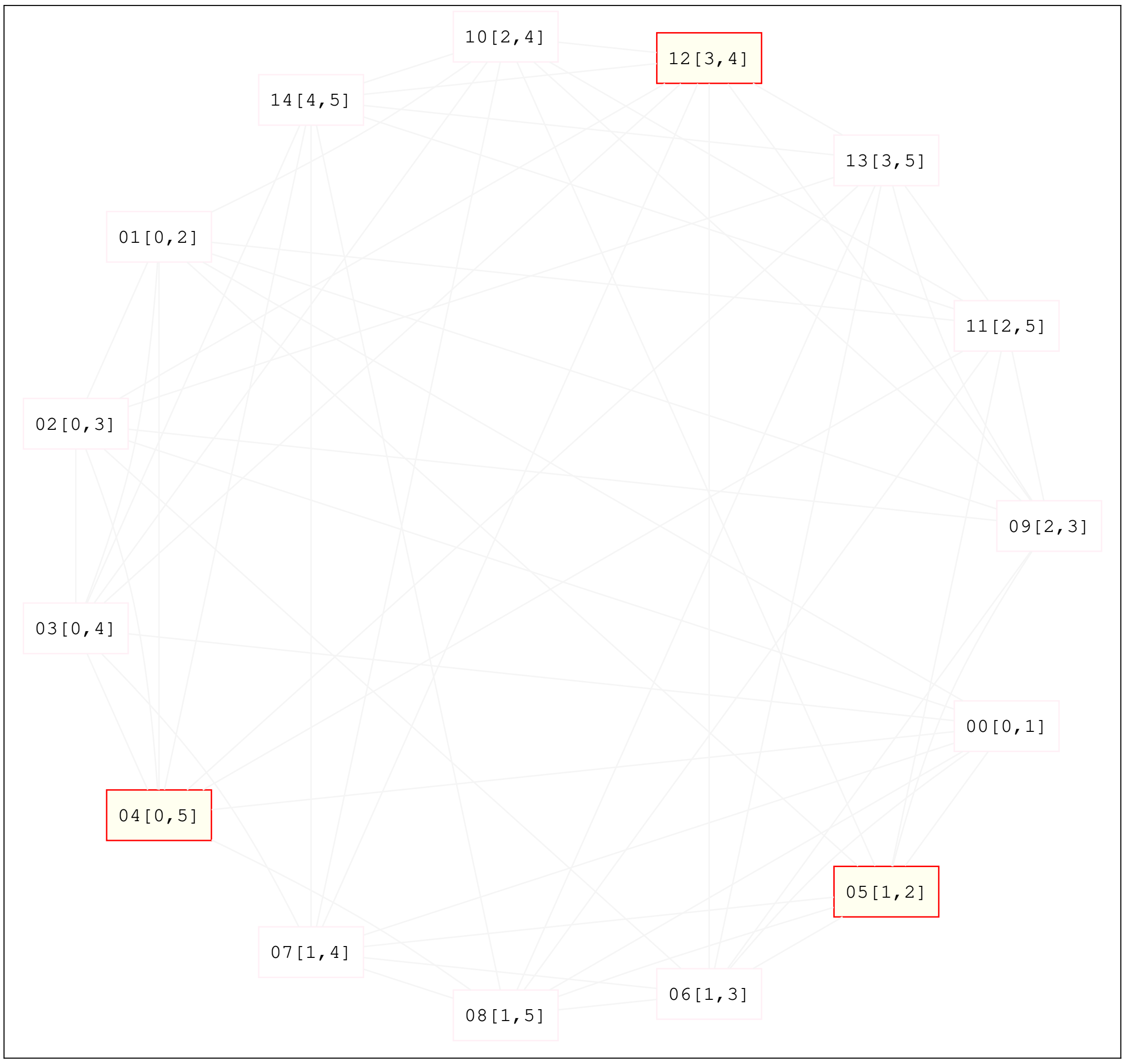


Figure 2.7: Generating the fifth (last) parallel step of $\mathcal{R}_6^{\|}$.

vertex 2, and the vertex 7 has already been removed, so the next candidate is the vertex 8, from where the usual procedure succeeds.

A similar situation occurs for the fourth p-step (see Figure 2.6). Here, starting from the vertex 3, the next candidate (the vertex 5) has to be discarded, but the vertex 6 can be chosen to proceed as usual.

Finally, the last p-step of $\mathcal{R}_6^{\|}$ is completed without backtracking, as shown in Figure 2.7. Even more complex situation would occur with $\mathcal{R}_{10}^{\|}$, so only the initial G_{10} is shown in Figure 2.8.

The optimized generation of $\mathcal{R}^{\|}$

Implementing Algorithm 1 as an efficient computer program is far from trivial, and has to resort to any and all optimizations possible to be able to generate all p-strategies for n up to 42 ($n = 46$ is out of reach, at least without a heavy parallelization and vectorization). A subset of such optimizations relies on the properties of the underlying row-cyclic strategy, so the implementation presented in Listing 1 is not as general as Algorithm 1 itself.

The code is thoroughly commented and should be self-explanatory. One may argue that the set operations (intersection, difference) could be implemented using bit sets (as they frequently are elsewhere), where bits of a vector register (or a couple of registers) indicate presence or absence of a particular element from the set, and the vector operations (bitwise `and`s and `andnot`s, respectively) on those registers.

Another important operation is computing the cardinalities of sets represented in such a way. On many modern machines there exists a "population count" instruction, that returns a number of bits set in a word, for one or more words at once.

Such code should then be manually tuned or rewritten for each CPU architecture. However, with 512-bit vector architectures already available, and the ones with even wider registers on the horizon, the vectorization should be the first thing to try for speeding up the code.

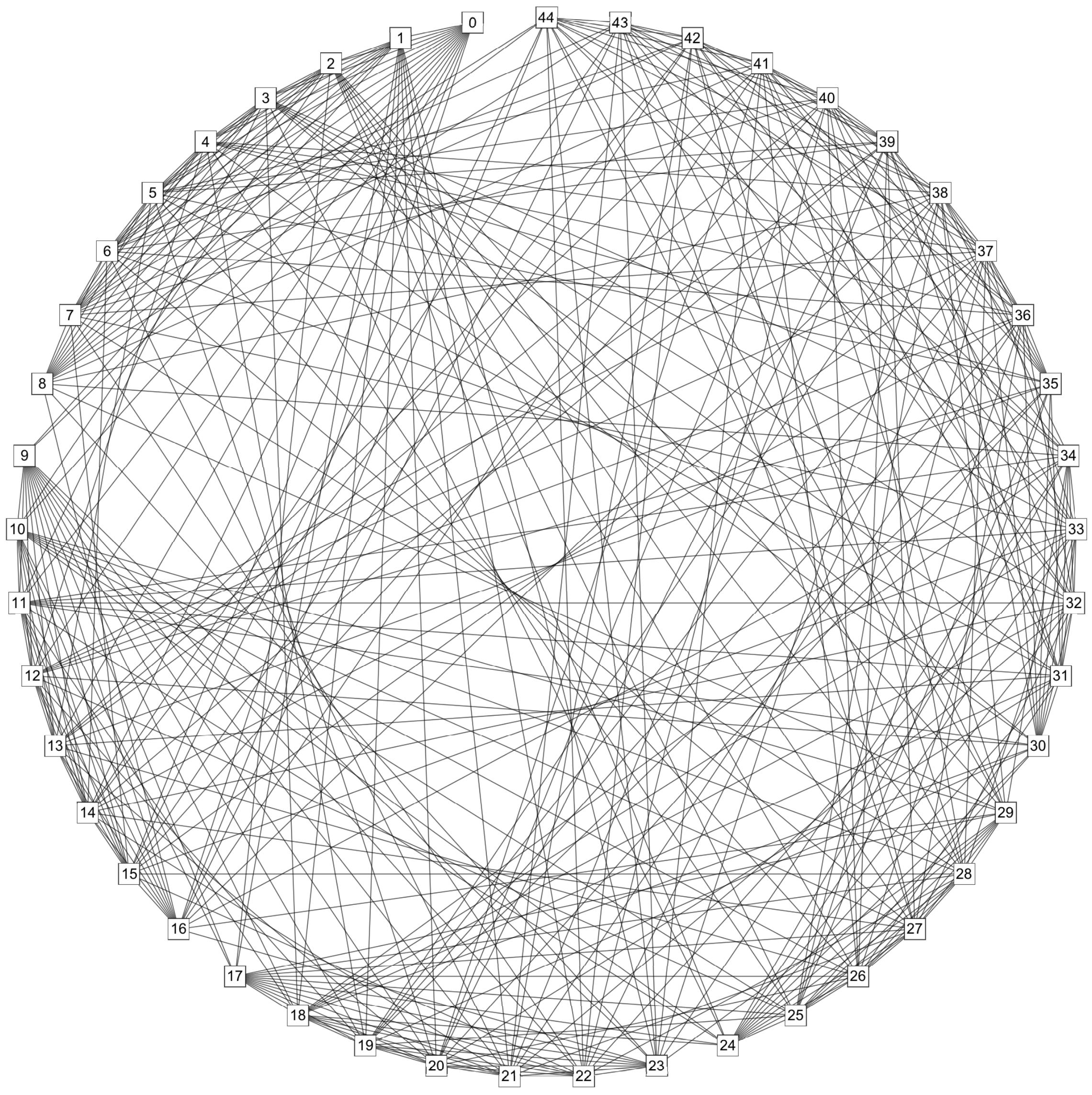


Figure 2.8: A collide graph for the row-cyclic strategy of order 10.

Listing 1: $\mathcal{R}^{\|}$ generation

```
#include <algorithm>
#include <cstdlib>
using namespace std;
// Define N s.t. 2 ≤ N ≤ 254 and N is even!
// # of pivots in a parallel step
#define P ((N) >> 1u)  // P = N/2
// # of parallel steps in a sweep
#define S ((N) - 1u)   // S = N - 1
// # of entries in the strictly upper triangle of an N × N matrix
#define E ((P) * (S))  // E = P · S
#define N_1 ((N) - 1u)
#define P_1 ((P) - 1u)
#define E_1 ((E) - 1u)
// # of the pivots not colliding with a given one
#define NCP ((E) - (((N_1) << 1u) - 1u))  // NCP = E - (2(N - 1) - 1)
typedef unsigned char uchar;
// Initial (here: row-cyclic) strategy is created in in_strat, as
// an array of (row,column), 0 ≤ row < column < N, pivot pairs.
static struct pivot { uchar r, c; } in_strat[E];
typedef unsigned short ushort;
// For a pivot pair from in_strat, there is a set of pivots that
// are independent from (i.e., non-colliding with) the chosen one.
// Only those pivots that come after the chosen one (let it be i)
// are stored in indep_set[i], while the size of such a set (at
// most NCP, but can be as low as 0) is recorded in indep_cnts[i].
static ushort indep_sets[E_1][NCP];  // I_j := indep_sets[j]
// Active set of a given pivot is a set from which the candidates
// for the pivot's successor in a step being generated are chosen.
// Active set of the first pivot in a step is the independent
// set of that pivot, without the pivots already put in used_set.
// Once a candidate j has been chosen from the active set of its
// predecessor i, the j's active set is formed from those pivots
// following j in the active set of i that are independent from j.
static ushort active_sets[E_1][NCP];  // A_j := active_sets[j]
static ushort indep_cnts[E_1];        // |I_j| := indep_cnts[j]
static ushort active_cnts[E_1];       // |A_j| := active_cnts[j]
// Keeps track of the pivots used so far to construct the partial
// parallel strategy.  Initially empty, the set's actual size is
// stored in used_cnt; it grows by speculatively appending, and
// shrinks by removing one pivot at the end of the set at a time.
static ushort used_set[E];            // U := used_set
// A buffer, when a sorted copy of used_set is needed, since the
// STL's set algorithms require the elements in ascending order.
static ushort tmp_set[E];             // U' := tmp_set
static ushort used_cnt;               // |U|, initially 0
static unsigned long long btrack;     // # of 'backtracks'
// .. EXECUTABLE CODE ..
```

```
void make_pairs_and_indep_sets()      // data initialization
{
  // Generating pivot pairs...
  ushort i = 0u;
  for (uchar r = 0u; r < N_1; ++r) {
    for (uchar c = r + 1u; c < N; ++c) {
      pivot &pvt = in_strat[i++];
      pvt.r = r; pvt.c = c;
    }
  }

  // Building independent sets...
 // Note that only indices of the pivots in in_strat are stored!
  for (i = 0u; i < E_1; ++i)
    for (ushort j = i + 1u; j < E; ++j)
      if ((in_strat[i].r != in_strat[j].r) &&
          (in_strat[i].r != in_strat[j].c) &&
          (in_strat[i].c != in_strat[j].r) &&
          (in_strat[i].c != in_strat[j].c))
        indep_sets[i][indep_cnts[i]++] = j;

  // Asserting monotonically non-increasing cardinalities...
 // Where it holds (e.g., for row-cyclic strategy), this property
 // shall be used to speed up the early pruning of the unfeasible
 // branches.  It may not hold for an arbitrary initial strategy!
  for (i = 1u; i < E_1; ++i)
    if (indep_cnts[i] > indep_cnts[i - 1u])
      exit(EXIT_FAILURE);
}

bool next_pivot()  // a recursive routine that generates $\mathcal{R}_N^{\parallel}$
{
  // Determine if a new parallel step should begin (rem = 0), or
  // if the current one should be expanded by another pivot...
  const div_t qr = div(int(used_cnt), int(P));
  if (qr.rem) { // expanding the current partial parallel step
    // take the index of the last pivot candidate from $U$
    const ushort prev_ix = used_set[used_cnt - 1u];
// How many more pivots are needed to complete this step, not
// counting the one that is about to be candidated in this call?
    const ushort needed = P_1 - ushort(qr.rem);
    if (needed) { // at least one more pivot is needed
// $P_{\mathrm{pix}}$ counts the number of indices NOT used so far in this step:
// to form rem pivots, rem distinct rows and the same number of
// distinct columns, different from rows, is needed, i.e., twice
// that in total, leaving $N - 2\cdot\mathrm{rem} = 2(P - \mathrm{rem})$ indices available.
      const ushort P_pix = (P - ushort(qr.rem)) << 1u;
      const ushort prev_cnt = active_cnts[prev_ix]; // $|A_{\mathrm{prev\_ix}}|$
    // (Over-)estimation of the length of a leading part of $A_{\text{prev\_ix}}$
    // that contains the viable candidates for the next pivot;
      // first estimate: $|A_{\text{prev\_ix}}|$, less the needed count.
      ushort j, prev_cnt_ = prev_cnt - needed;
      // refine (shorten) the length of a leading part of $A_{\text{prev\_ix}}$
      for (j = 0u; j < prev_cnt_; ++j) {
        // a potential candidate from $A_{\text{prev\_ix}}$
        const ushort my_ix = active_sets[prev_ix][j];
// If a candidate starts in row $r$, then it and its successors can
// only contain indices that are not less than $r$, since the column
// indices have to be greater than the corresponding row ones,
// and the row index of a pivot has to be greater than in any
// predecessor in the same step.  So, there are at most $N - r$
// indices available to generate the pivots; if that is less than
// $P_{\text{pix}}$, the candidate is not viable, since not all indices could
// be covered if it were chosen.  Stop iterating any further.
        if ((N - in_strat[my_ix].r) < P_pix)
          break;
        const ushort mync_cnt = indep_cnts[my_ix];
        // "Fast" break, but only if $|I_j|$ monotonically
        // non-increasing w.r.t. j.  Otherwise, continue.
        if (mync_cnt < needed)
          break;
      }
      if (j < prev_cnt_) // shorten on early exit
        prev_cnt_ = j;

      const ushort *const prev_end = // one-past-last of $A_{\text{prev\_ix}}$
        &(active_sets[prev_ix][prev_cnt]);

     // iterate over $A_{\text{prev\_ix}}$, testing the pivot candidates in turn
      for (ushort i = 0u; i < prev_cnt_; ++i) {
        // the candidate: my_ix := $A_{\text{prev\_ix}}[i]$
        const ushort my_ix = active_sets[prev_ix][i];
        // the candidate's independent-pivot-list size, $|I_{\text{my\_ix}}|$
        const ushort mync_cnt = indep_cnts[my_ix];
        const ushort *const prev_beg =     // start of the tail
          &(active_sets[prev_ix][i + 1u]); // of $A_{\text{prev\_ix}}$, after $i$
        const ushort *const mync_beg = // start of $I_{\text{my\_ix}}$
          &(indep_sets[my_ix][0u]);
        const ushort *const mync_end = // one-past-last of $I_{\text{my\_ix}}$
          &(indep_sets[my_ix][mync_cnt]);
        // about to generate $A_{\text{my\_ix}}$
        ushort *const my_dst = &(active_sets[my_ix][0u]);
        // $A_{\text{my\_ix}} = A_{\text{prev\_ix}}[i+1:] \cap I_{\text{my\_ix}}$
        active_cnts[my_ix] = ushort(set_intersection(
            prev_beg, prev_end, mync_beg, mync_end, my_dst)
          - my_dst);
```

```
  // If A_my_ix has enough pivots, there is a chance of completing
  // this step; otherwise, cycle and try with the next candidate.
        if (active_cnts[my_ix] >= needed) {
          used_set[used_cnt++] = my_ix;
          if (next_pivot())
            return true;
          --used_cnt; // discard the candidate
          ++btrack;   // backtrack
        }
        else // 'backtrack' (a kind of)
          ++btrack;
      }
    }
    else { // choose a candidate for the last pivot in the step
// So far, P-1 pivots have been chosen, i.e., P-1 distinct rows
// and P-1 distinct columns, different from rows. It means that
// 2(P-1)=N-2 indices have been used, so there is exactly one
// pivot candidate left (i.e., no need to iterate over A_prev_ix).
      // take the first (and only) pivot from A_prev_ix
      used_set[used_cnt++] = active_sets[prev_ix][0u];
      // try to progress into the next step, backtrack on failure
      if (next_pivot())
        return true;
      --used_cnt; // discard the candidate
      ++btrack;   // backtrack
    }
  }
  else if (used_cnt == E) // All available pivots used: SUCCESS!
    return true;          // propagates up to stop the recursion
  else {// a new parallel step's head (first pivot) is sought for
    // Which parallel step should be created?
    const ushort ix = ushort(qr.quot);
    if (ix) { // not the first one, so used_set is not empty
      // copy used_set to tmp_set
      copy(used_set, used_set + used_cnt, tmp_set);
      // sort tmp_set ascendingly
      sort(tmp_set, tmp_set + used_cnt);
    }

// The parallel steps are internally sorted, ascendingly w.r.t.
// the order that the pivots have in row-cyclic strategy.  The
// first row-cyclic pivot therefore has to be at a head of some
// parallel step, the second pivot of another (since the first
// pivot collides with the second), and so on.  The first paral-
// lel step has to begin with the first row-cyclic pivot; other-
// wise, by exchanging order of the steps (i.e., swapping the
// first step and the one which starts with the first pivot), a
// lexicographically closer parallel strategy would emerge.  The
// second parallel step has to start with the second row-cyclic
// pivot, the third step with the third pivot, and so on, follow-
// ing a similar argument. Therefore, step ix starts with pivot ix.

// If there are not enough non-colliding pivots available when the
// head is fixed, there is a global FAILURE (not just backtrack)!
    if (indep_cnts[ix] < P_1)
      return false;
    // $A_{\mathrm{ix}}$ consists of all the pivots that are non-colliding with
    // ix (i.e., of $I_{\mathrm{ix}}$), without those already used: $A_{\mathrm{ix}} = I_{\mathrm{ix}} \backslash U'$.
    // If $A_{\mathrm{ix}}$ does not contain enough pivot candidates to complete
    // this step, there is a global FAILURE (not just backtrack)!
    ushort *const dst = &(active_sets[ix][0u]);
    if ((active_cnts[ix] = ushort(set_difference(
            &(indep_sets[ix][0u]),
            &(indep_sets[ix][indep_cnts[ix]]),
            tmp_set, tmp_set + used_cnt, dst)
          - dst)
        ) < P_1)
      return false;
    // Fix the head to ix and try recursively to complete the
    // strategy.  Backtrack on failure to the previous step.
    used_set[used_cnt++] = ix;
    if (next_pivot())
      return true;
    --used_cnt; // discard the candidate
    ++btrack;   // backtrack
// Discard is needed even though ix is the correct choice here,
// but some of the previous candidates in $U$ have to get discarded
// as well.  On all future attempts of creating this step, ix will
// be chosen as its head again, but with a different state of $U$.
  }
  return false;
}

int main() // initialize the data and start the recursive search
{
  make_pairs_and_indep_sets();
  if (next_pivot()) {
    // Print out the parallel strategy recorded as a sequence of
    // pivot indices in used_set: in_strat[used_set[$i$]], $0 \le i < E$.
    return EXIT_SUCCESS;
  }
  else // Should never happen!
    return EXIT_FAILURE;
}
// specify N when compiling; e.g., `c++ -DN=42u this_code.cpp`
// tested with: g++ (GNU), clang++ (Apple), cl.exe (Microsoft)
```

On Figure 2.9 the performance of the code in Listing 1 is shown, alongside with the number of backtracks needed for a particular strategy order. Please bear in mind that the user time axis is in $\log_{10}$ scale! An older machine, with about 2/3 performance peak than the one used to get the timings, struggled for weeks to generate $\mathcal{R}^{\|}_{46}$ and $\mathcal{R}^{\|}_{50}$, and had to be shut down before completion.

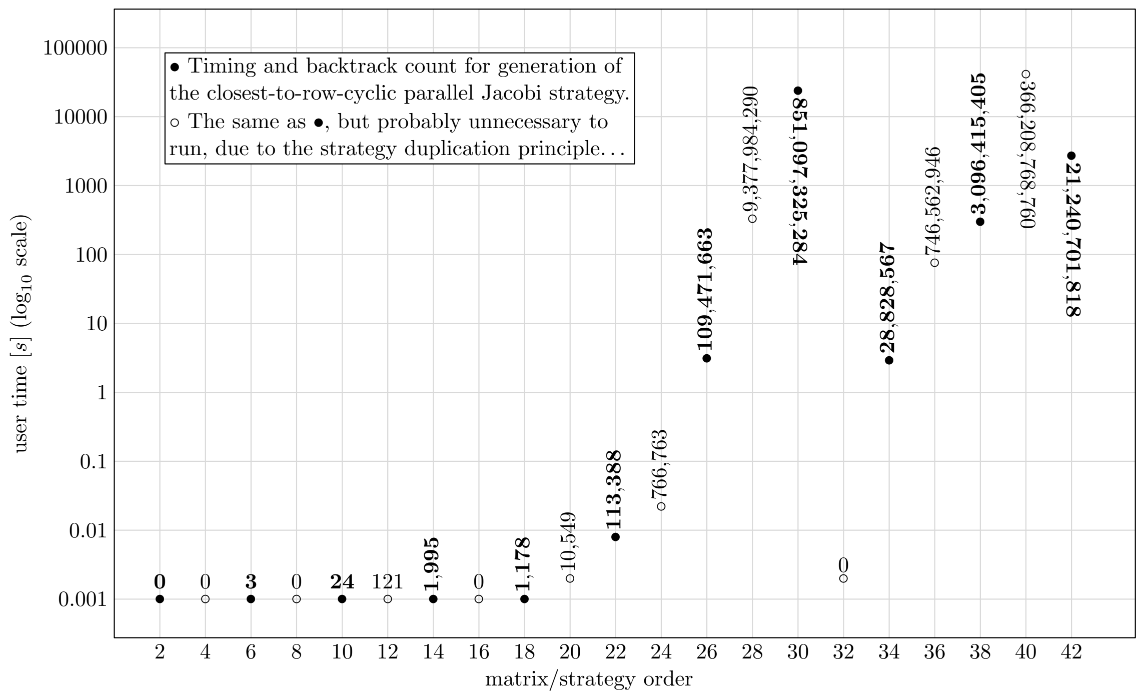


Figure 2.9: The user part of the execution time, as reported by `time` utility, of $\mathcal{R}^{\|}$ generation, with a number of pivot candidates discarded (i.e., the algorithm's "backtracks" counted as in Listing 1) per each strategy/matrix order, on an Intel Core i5-4570 CPU @ 3.20GHz with `clang-703.0.31` C++ compiler (macOS Sierra).

2.3. A single-GPU algorithm

In this section the two-level blocked Jacobi (H)SVD algorithm for a single GPU is described. The algorithm is designed and implemented with NVIDIA CUDA [59] technology, but is also applicable to OpenCL, and—at least conceptually—to the other massively parallel accelerator platforms.

It is assumed that the following CUDA operations are correctly rounded, as per IEEE standard 754-2008 [44]: $+$, $-$, $*$, $/$, $\sqrt{x}$, $\texttt{fma}(x, y, z) = x \cdot y + z$, and $\texttt{rcp}(x) = 1/x$. The round to nearest (tie to even) rounding is also assumed, unless stated otherwise. Under those assumptions, the algorithm may work in any floating-point precision available, but is tested in double precision only, on Fermi and Kepler GPU architectures.

The algorithm performs all computation on a GPU, and consists of three kernels:

1. `initV` – optional initialization of the matrix V to I_n, if the full (H)SVD is requested (for the HSVD, $V^{-T} = JVJ$ will be accumulated instead of V);
2. `pStep` – invoked once for each p-step in a block-sweep;
3. `Sigma` – a final singular value extraction ($\sigma_i = \|g'_i\|_2$).

The CPU is responsible only for the main control flow, i.e., kernel invocations and testing of the stopping criterion. Besides a simple statistics from each `pStep` call, there is no other CPU $\leftrightarrow$ GPU data transfer. It is assumed that the input factor G is preloaded onto a GPU. The matrix J is kept partitioned as $J = \operatorname{diag}(I, -I)$ and represent it with a parameter n_+, where n_+ denotes the number of positive signs in J. The output data remaining on the GPU are $G' = U\Sigma$ (overwrites G), Σ, and (optionally) V.

Data layout (i.e., array order) is column-major (as in Fortran), to be compatible with (cu)BLAS and other numerical libraries, like MAGMA. The one-based array indices will be written in parentheses, and zero-based ones in square brackets.

The matrices G and V are partitioned into $\mathsf{b} = n/16$ block-columns, as in (2.6). To simplify the algorithm, n must be divisible by 16 and b must be even. Otherwise, if n is not divisible by 32, G has to be bordered as in [56, eq. (4.1)]. The reason lies in the hardware constraints on the GPU shared memory configurations and a fixed warp size (32 threads), as explained in what follows.

The main focus will be on `pStep` kernel, since the other two are straightforward. An execution grid for `pStep` comprises $\mathsf{b}/2$ thread blocks, i.e., one thread block per a pivot block-pair. Each 2-dimensional thread block is assigned $32 \times 16 = 512$ threads, and 16 kB of the GPU shared memory. That imposes a theoretical limit of 3 thread blocks per multiprocessor, 100% occupancy on a Fermi, and 75% occupancy on a Kepler GPU.

A chosen block p-strategy $\mathcal{S}_\mathsf{b}$ is preloaded in the constant or global GPU memory in a form of an $O(1)$ lookup table S_b. A thread block t in a `pStep` invocation s during a block-sweep r obtains from S_b the indices p and q, $1 \le \mathsf{p} < \mathsf{q} \le \mathsf{b}$, of the block-columns t is about to process. In other words, a mapping $(\mathsf{r},\mathsf{s},\mathsf{t}) \mapsto (\mathsf{p},\mathsf{q}) \in \mathcal{S}_\mathsf{b}$ establishes a correspondence between the thread blocks and the pivot block-pairs.

A thread block behavior is uniquely determined by the block indices p and q, since the thread blocks in every `pStep` invocation are mutually independent. Computation in a thread block proceeds in the three major phases:

1. `factorize` – shortens the pivot block-pair $\begin{bmatrix} G_\mathsf{p} & G_\mathsf{q} \end{bmatrix}$, according to (2.7) or (2.8), into the triangular factor R_pq of order 32, and initializes $V'_\mathsf{pq} = I_{32}$;
2. `orthogonalize` – orthogonalizes R_pq by the inner pointwise Jacobi method, according to the block-oriented or the full block variant of the Jacobi (H)SVD algorithm (see section 2.1.), accumulating the applied rotations into V'_pq;
3. `postmultiply` – postmultiplies $\begin{bmatrix} G_\mathsf{p} & G_\mathsf{q} \end{bmatrix}$, and optionally $\begin{bmatrix} V_\mathsf{p} & V_\mathsf{q} \end{bmatrix}$, by V'_pq, according to (2.9).

The matrices R_pq and V'_pq reside in the shared memory and together occupy 16 kB. The entire allocated shared memory can also be regarded as a 64×32 double precision matrix, named G_pq, of which R_pq aliases the lower half and V'_pq the upper half.

There is no shared memory configuration that can hold two square double precision matrices of order that is a larger multiple of the warp size than 32. It is therefore optimal to use the smallest shared memory configuration (16 kB), leaving the highest amount (48 kB) of the L1 cache available. Also, since R_{pq} and V'_{pq} have to be preserved between the phases, all phases need to be completed in the same kernel invocation. That creates a heavy register pressure, which is the sole reason why only one thread block (instead of three) can be active on a multiprocessor.

In the complex case ($\mathbb{F} = \mathbb{C}$), the shared memory configuration would be 48 kB (suboptimal, 16 kB unutilized) or 32 kB, for a Fermi or a Kepler GPU, respectively.

The two approaches for `factorize` will be presented. The Cholesky factorization of the Gram matrix H_{pq}, as in (2.7), is described in subsection 2.3.1. in two subphases, and the QR factorization (2.8) is discussed in subsection 2.3.2.

2.3.1. The Cholesky factorization

The first subphase of `factorize` loads the successive 64×32 chunks of $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$ into G_{pq}. For a thread with the Cartesian indices $[x, y]$ in a particular thread block, x is its lane ID, and y its warp ID. Let $y' = y + 16$ onwards. After a chunk is loaded, the thread $[x, y]$ then updates $H_{\mathsf{pq}}[x, y]$ and $H_{\mathsf{pq}}[x, y']$ (kept in its registers and being initially 0),

$$H_{\mathsf{pq}}[x, y] \mathrel{+}= G_{\mathsf{pq}}[:, x]^T G_{\mathsf{pq}}[:, y],$$
$$H_{\mathsf{pq}}[x, y'] \mathrel{+}= G_{\mathsf{pq}}[:, x]^T G_{\mathsf{pq}}[:, y'].$$

Finally, when all the chunks are processed, H_{pq} is written into R_{pq}, and V'_{pq} is set to I_{32}. Note that data in the GPU RAM are accessed only once. No symmetrization is needed for H_{pq}, since only its lower triangle is taken as an input for the Cholesky factorization. For details of this subphase see Algorithm 3.

On a Kepler GPU, with 8-byte-wide shared memory banks, each thread in a warp can access a different bank. Due to column-major data layout, each of 32 consecutive (modulo 64) rows of G_{pq} belongs to a separate bank. Therefore, the modular row addressing of Algorithm 3 guarantees bank-conflict-free operation on a Kepler GPU, and generates two-way bank conflicts on a Fermi GPU, with 4-byte-wide banks.

The next subphase consists of the in-place, forward-looking Cholesky factorization of H_{pq} without pivoting, i.e., $H_{\mathsf{pq}} = L_{\mathsf{pq}} L_{\mathsf{pq}}^T$. The factorization proceeds columnwise to avoid bank conflicts. After the factorization, the upper triangle of R_{pq} is set to L_{pq}^T, and the strict lower triangle to zero. This transposition is the only part of the entire algorithm that necessarily incurs the bank conflicts. The factorization has 32 steps. The step k, for $0 \le k < 32$, transforms $H_{\mathsf{pq}}[k{:}, k{:}]$ (where this notation indicates a range of all rows from—and including—the k-th, and of all columns from—and including—the k-th, onwards) in two or three stages as follows:

(a) Compute $L_{\mathsf{pq}}[k{:}, k]$, overwriting $H_{\mathsf{pq}}[k{:}, k]$ (see Figure 2.10(a)). Only one warp is active. The thread $[x, y]$ performs the following operations:
   - If $x = y = k$, then $L_{\mathsf{pq}}[k, k] = \sqrt{H_{\mathsf{pq}}[k, k]}$;

Algorithm 3: Device function that computes the Gram matrix H_{pq}.

```
Description: Input: $[G_{\mathsf{p}}\ G_{\mathsf{q}}]$. Output: $H_{\mathsf{pq}}$. Thread ID: $[x, y]$.
$x' = x + 32$; $y' = y + 16$; $h_{xy} = h_{xy'} = 0$; //$H_{\mathsf{pq}}$ elements kept in registers
for ($i = x$; $i < n$; $i$ += 64)        // process the next chunk of $[G_{\mathsf{p}}\ G_{\mathsf{q}}]$
    $G_{\mathsf{pq}}[x, y] = G_{\mathsf{p}}[i, y]$; $G_{\mathsf{pq}}[x, y'] = G_{\mathsf{q}}[i, y]$; //load the first 32 chunk rows
    if ($(i' = i + 32) < n$) then        // load the remaining chunk rows
        $G_{\mathsf{pq}}[x', y] = G_{\mathsf{p}}[i', y]$; $G_{\mathsf{pq}}[x', y'] = G_{\mathsf{q}}[i', y]$;
    else                                  // border $G_{\mathsf{pq}}$ with zeros
        $G_{\mathsf{pq}}[x', y] = 0$; $G_{\mathsf{pq}}[x', y'] = 0$;
    end if
    __syncthreads();      // the shared memory writes must take effect
    unrolled for ($j = 0$; $j < 64$; ++$j$) // compute the partial dot-products
        $j' = (x + j) \bmod 64$; // modular row indexing, avoid bank conflicts
        $g_{j'x} = G_{\mathsf{pq}}[j', x]$; $g_{j'y} = G_{\mathsf{pq}}[j', y]$; $g_{j'y'} = G_{\mathsf{pq}}[j', y']$;
        $h_{xy}$ = fma($g_{j'x}, g_{j'y}, h_{xy}$);          // update $H_{\mathsf{pq}}[x, y]$
        $h_{xy'}$ = fma($g_{j'x}, g_{j'y'}, h_{xy'}$);       // update $H_{\mathsf{pq}}[x, y']$
    endfor
    __syncthreads();      // ensure that $G_{\mathsf{pq}}$ is free to be overwritten
endfor
$R_{\mathsf{pq}}[x, y] = h_{xy}$; $R_{\mathsf{pq}}[x, y'] = h_{xy'}$;   // store (unsymmetrized) $H_{\mathsf{pq}}$ to $R_{\mathsf{pq}}$
__syncthreads(); // ensure the shared memory writes have taken effect
```

- else, if $x > y = k$, then $L_{\mathsf{pq}}[x, k] = H_{\mathsf{pq}}[x, k]/\sqrt{H_{\mathsf{pq}}[k, k]}$;[4]
- else, the thread is dormant, i.e., does nothing.

(b) Update at most 16 subsequent columns of H_{pq}. Let $j = (k + 1) + y$. If $x \geq j$ and $j < 32$, then $H_{\mathsf{pq}}[x, j] = \texttt{fma}(-L_{\mathsf{pq}}[x, k], L_{\mathsf{pq}}[j, k], H_{\mathsf{pq}}[x, j])$, else do nothing (see Figure 2.10(b)).

(c) If there are more columns remaining, let $j' = (k + 1) + y'$. If $x \geq j'$ and $j' < 32$, then $H_{\mathsf{pq}}[x, j'] = \texttt{fma}(-L_{\mathsf{pq}}[x, k], L_{\mathsf{pq}}[j', k], H_{\mathsf{pq}}[x, j'])$, else do nothing (see Figure 2.10(c)).

After each stage, a thread-block-wide synchronization (`__syncthreads`) is necessary.

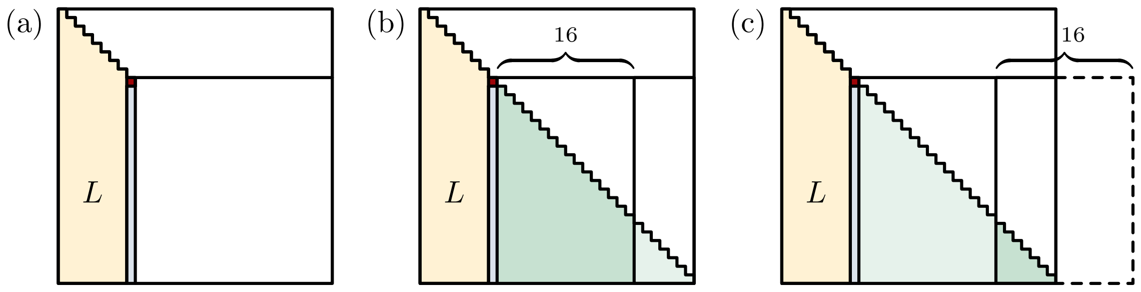


Figure 2.10: The forward-looking Cholesky factorization $H_{\mathsf{pq}} = L_{\mathsf{pq}} L_{\mathsf{pq}}^T$.

[4]Could possibly be faster if implemented as $L_{\mathsf{pq}}[x, k] = H_{\mathsf{pq}}[x, k] * \texttt{rsqrt}(H_{\mathsf{pq}}[k, k])$.

2.3.2. The QR factorization

When the input matrix G is badly scaled, the QR factorization (2.8) is required at all blocking levels, since the input columns of too large (resp. too small) norm could cause overflow (resp. underflow) while forming the Gram matrices. If the QR factorization employs the Householder reflectors, the column norm computations should be carried out carefully, as detailed in section 2.6.

The tall-and-skinny in-GPU QR factorization is described in [3]. It is applicable when a single QR factorization per p-step is to be performed on a GPU, e.g., in the shortening phase of a multi-GPU algorithm. On the shared memory blocking level, each thread block has to perform its own QR factorization. Therefore, an algorithm for the batched tall-and-skinny QRs is needed in this case.

Ideally, such an algorithm should access the GPU RAM as few times as possible and be comparable in speed to the Cholesky factorization approach. The algorithm can be made to access $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$ exactly once, as it will be shown, but the latter remains difficult to accomplish.

Let $A_0^{(i)}$ and $A_1^{(i)}$ be the 32×32 matrices that alias the lower half and the upper half of G_{pq}, i.e., R_{pq} and V'_{pq}, respectively. The first 32×32 chunk of $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$ is loaded into $A_0^{(0)}$ and factorized as $A_0^{(0)} = Q_0^{(0)} R_0^{(0)}$. The factorization is performed by 31 successive applications of the Householder reflectors, in a pattern similar to Figure 2.10(b),(c). A reflector is simultaneously computed in all active warps before the update, but is not preserved, and $Q_0^{(0)}$ is not (explicitly or implicitly) formed.

More precisely, for $0 \leq k < 31$, let H_k be the reflector annihilating the subdiagonal of the k-th column,

$$H_k = I_{32} - \tau_k w_k w_k^T,$$

where $w_k = \begin{bmatrix} \mathbf{0} & 1 & v_k \end{bmatrix}^T$ ($\mathbf{0}$ is a vector of k zeros). In a thread with the row index x, H_k is represented by τ_k and $w_k[x]$. When the reflector is found, the warp y transforms $A_0^{(0)}[\ell{:}, \ell]$, where $\ell = k + y$. Let z_ℓ be the scalar product $z_\ell = w_\ell^T A_0^{(0)}[:, \ell]$, computed by warp-level shared memory reduction (on Fermi), or by warp shuffle reduction (on Kepler, see Listing 2). Then, the update by H_k is

$$A_0^{(0)}[x, \ell]' = \texttt{fma}(-\tau_\ell \cdot z_\ell, w_\ell[x], A_0^{(0)}[x, \ell]).$$

The transformation is then repeated for $\ell' = k + y'$ (see Listing 3 for the implementation, up to the choice of the norm computation algorithm `dNRM2_32`).

After $R_0^{(0)}$ is formed, the second chunk of $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$ is loaded into $A_1^{(0)}$ and similarly factored as $A_1^{(0)} = Q_1^{(0)} R_1^{(0)}$.

The factors $R_0^{(0)}$ and $R_1^{(0)}$ are combined into $R_0^{(1)}$ by a "peel-off" procedure, illustrated in Figure 2.11. The procedure peels off one by one (super)diagonal of $R_1^{(0)}$ by the independent Givens rotations, until (after 32 stages) $R_1^{(0)}$ is reduced to a zero matrix. In the stage k, the row x of $R_0^{(0)}$ and the row $x - k$ of $R_1^{(0)}$ are transformed by a rotation determined from $R_0^{(0)}[x, x]$ and $R_1^{(0)}[x - k, x]$ to annihilate the latter. This is the main conceptual difference from the tall-and-skinny QR, described, e.g., in [18, 19], where the combining is performed by the structure-aware Householder

Listing 2: Sum-reduction across a warp by the warp shuffle Kepler instructions

```
// Kepler warp shuffle +-reduction of x
__device__ double dSum32(const double x) {
  // my lane's value, broken into 2 x 4 bytes
  int lo_my, hi_my;
  // the other lane's value, broken into 2 x 4 bytes
  int lo_his, hi_his;
  // my lane's and the other lane's values
  double x_my = x, x_his;

  // the following loop is unrolled log2 32 = 5 times
  for (int i = 16; i > 0; i >>= 1) {
    // x[my_id] += x[my_id XOR i]
    lo_my  = __double2loint(x_my);
    hi_my  = __double2hiint(x_my);
    lo_his = __shfl_xor(lo_my, i);
    hi_his = __shfl_xor(hi_my, i);
    x_his  = __hiloint2double(hi_his, lo_his);
    x_my  += x_his;
  }

  // get the value of x[0] to x_his
  lo_my  = __double2loint(x_my);
  hi_my  = __double2hiint(x_my);
  lo_his = __shfl(lo_my, 0);
  hi_his = __shfl(hi_my, 0);
  x_his  = __hiloint2double(hi_his, lo_his);

  return x_his;
}
```

reflectors. The Givens rotations are chosen to avoid the expensive column norm computations.

Each remaining chunk of $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$ is loaded into $A_1^{(i)}$, factored as $A_1^{(i)} = Q_1^{(i)} R_1^{(i)}$, and combined with $R_0^{(i)}$ to obtain $R_0^{(i+1)}$. After the final $R_{\mathsf{pq}} = R_0^{(n/32-1)}$ is formed, V'_{pq} is set to I_{32}.

Unfortunately, this approach is not recommendable when efficiency matters. For example, on matrices of order 3072, the QR factorization is 12–15 times slower than the Cholesky factorization, depending on the column norm computation algorithm.

2.3.3. The orthogonalization

In this phase, the inner pointwise Jacobi (H)SVD method is run on R_{pq}. A constant memory parameter M_{s}, representing the maximal number of inner sweeps, decides whether the block-oriented ($M_{\mathrm{s}} = 1$) or the full block variant ($M_{\mathrm{s}} > 1$, usually $M_{\mathrm{s}} = 30$) should be performed.

Listing 3: The 32×32 QR factorization as a CUDA device function

```
__device__ void dQR32( // compute R = qr(A), with 32x32 double A
 volatile double *const A, // shared mem, to be overwritten by R
 // in the upper triangle, and 0s in the strictly lower triangle
 const unsigned x ,    // lane ID      [ 0..31], row index
 const unsigned y0,    // warp ID      [ 0..15], column index
 const unsigned y1) {  // warp ID + 16 [16..31], column index
  for (unsigned k = 0u; k < 31u; ++k) { // unrolled loop
    double nrm2 = 0.0, Axk, Axk_, beta, alpha_beta, tau;
    // compute the Househoher reflector, see LAPACK's DLARFG
    const unsigned l0 = k + y0;
    if (l0 < 32u) { // A[(row,col)] := A[32 * col + row]
      Axk  = A[(x,k)];  Axk_ = ((x >= k) ? Axk : 0.0);
      // compute $||A[k:,k]||_2$ using the warp shuffles
      nrm2 = dNRM2_32(Axk_); // $\ell_2$-norm computation
      if (nrm2 > 0.0) { // __ddiv_rn(x,y): intrinsic for $x \oslash_{\mathrm{RN}} y$
        const double alpha = A[(k,k)];
        beta = copysign(nrm2, alpha);
        alpha_beta = alpha + beta;
        tau = __ddiv_rn(alpha_beta, beta);
      } // end if nrm2...
    } // end if l0...
    __syncthreads();

    if (nrm2 > 0.0) { // apply the Householder reflector
      if (l0 == k) // set $A[k,k] = -\beta$ and annihilate $A[k+1:,k]$
        A[(x,k)] = ((x == k) ? -beta : ((x > k) ? 0.0 : Axk));
      else { // __fma_rn(x,y,z): Fused-Multiply-Add $(x * y + z)_{\mathrm{RN}}$
        const double Axy = A[(x,l0)];
        const double Wxk = ((x == k) ? 1.0 :
          ((x > k) ? __ddiv_rn(Axk_, alpha_beta) : 0.0));
        const double zl0 = dSum32(Wxk * Axy);
        const double _tz = -(tau * zl0);
        A[(x,l0)] = __fma_rn(_tz, Wxk, Axy);
      } // end else
      const unsigned l1 = k + y1;
      if (l1 < 32u) { // _rn: round-to-nearest(even) in $\oslash$ and FMA
        const double Axy = A[(x,l1)];
        const double Wxk = ((x == k) ? 1.0 :
          ((x > k) ? __ddiv_rn(Axk_, alpha_beta) : 0.0));
        const double zl1 = dSum32(Wxk * Axy);
        const double _tz = -(tau * zl1);
        A[(x,l1)] = __fma_rn(_tz, Wxk, Axy);
      } // end if l1...
    } // end if nrm2...
    __syncthreads();
  } // end for
} // end dQR32
```

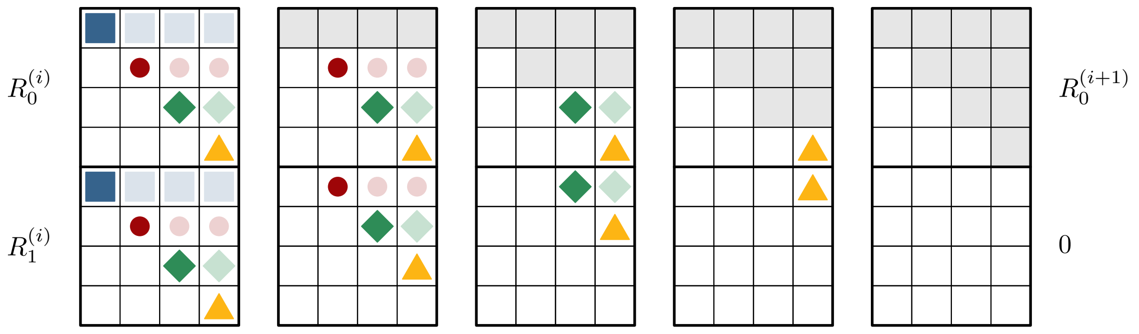


Figure 2.11: A parallel “peel-off” procedure on 4×4 matrices, in 4 stages. The rows with the same symbol are transformed in a stage independently of each other by the Givens rotations computed from the dark-colored diagonal elements. The final elements of $R_0^{(i+1)}$ are fully shaded.

The inner p-strategy $\mathcal{S}'_{32}$ is encoded into a constant memory lookup table S'_{32}. In principle, $\mathcal{S}'_{32}$ need not be of the same type of p-strategies as $\mathcal{S}_{\mathsf{b}}$. For example, $\mathcal{S}'_{32}$ may be of the Brent and Luk type, and $\mathcal{S}_{\mathsf{b}}$ may be of the Matharam–Eberlein type, but usually a choice of the p-strategies is uniform, with only a single type for all levels.

In each p-step s of an inner sweep r, the warp y is assigned the pivot pair $(p, q) = S'_{32}(r, s)[y]$, i.e., the pair of columns $\begin{bmatrix} g_p & g_q \end{bmatrix}$ of R_{pq}, and the pair of columns $\begin{bmatrix} v_p & v_q \end{bmatrix}$ of V'_{pq}. Then, the following subphases are executed:

1. The 2×2 pivot matrix $\widehat{H}_{pq}$ from (2.3) is computed. As explained in subsection 2.3.4., three dot products (for h_{pq}, h_{pp}, and h_{qq}) are needed when the rotation formulas from [25] are not used. The elements $g_p[x]$, $g_q[x]$, $v_p[x]$, and $v_q[x]$ are preloaded into registers of a thread with lane ID x, so, e.g., V'_{pq} may be overwritten as a scratch space for warp-level reductions on Fermi GPUs.
2. The relative orthogonality criterion for g_p and g_q is fulfilled if
$$|h_{pq}| < c(\varepsilon)\sqrt{h_{pp}}\sqrt{h_{qq}} = c(\varepsilon)\|g_p\|_2\|g_q\|_2,$$
where $c(\varepsilon) = \varepsilon\sqrt{\hat{n}}$, and $\hat{n}$ is the matrix order (here, $\hat{n} = 32$). If g_p and g_q are relatively orthogonal, then set an indicator ρ_s that determines whether a rotation should be applied, to $\rho_s = 0$, else set $\rho_s = 1$. Note that ρ_s is a per-thread variable, but has the same value across a warp.
3. Let a_s be a thread-block-wide number of warps about to perform the rotations. A warp has 32 threads, so $a_s = (\Sigma\rho_s)/32$, where the sum ranges over all threads in the thread block. Compute a_s as `__syncthreads_count`$(\rho_s)/32$. Since the pointwise Jacobi process stops if no rotations occurred in a sweep, a per-sweep counter of rotations, A_r, has to be increased by a_s. The counters a_s and A_r are kept per thread, but have the same value in the thread block.
4. Let the pivot indices p and q correspond to the columns k and ℓ, respectively, of the input factor G. If $k \leq n_+ < \ell$, then compute the transformation $\widehat{V}_{pq}$ from (2.4) as a hyperbolic rotation (2.14), else as a trigonometric one (2.13),

according to subsection 2.3.4. If $\operatorname{cs}\varphi \neq 1$, then set $\rho_s' = 1$ (a proper rotation), else leave $\rho_s' = 0$ to indicate that the rotation is nearly identity. If ρ_s was 0, just determine if the rotation would be a trigonometric or a hyperbolic one, instead of computing it.

5. If the rotation is trigonometric, find the new diagonal elements, h_{pp}' and h_{qq}',
$$h_{pp}' = \mathtt{fma}(\tan\varphi, h_{pq}, h_{pp}), \quad h_{qq}' = \mathtt{fma}(-\tan\varphi, h_{pq}, h_{qq}).$$
If $\rho_s = 0$ (i.e., $\widehat{V}_{pq}$ is the identity), take $h_{pp}' = h_{pp}$ and $h_{qq}' = h_{qq}$. To keep the eigenvalues sorted nonincreasingly [40], if $h_{pp}' < h_{qq}'$ when $\ell \leq n_+$, or $h_{pp}' > h_{qq}'$ when $k > n_+$, set $P_2 = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, else $P_2 = I_2$. Define
$$\widehat{V}_{pq}' = \widehat{V}_{pq} P_2.$$
The eigenvalue order tends to stabilize eventually; thus no swapping is usually needed in the last few sweeps [53, 56]. If the rotation is hyperbolic, to keep J partitioned, set $\widehat{V}_{pq}' = \widehat{V}_{pq}$ (there is no sorting, and the new diagonal elements are not needed). An unpartitioned J could lead to a slower convergence [67].
6. Apply, per thread, $\widehat{V}_{pq}'$ to $\begin{bmatrix} g_p[x] & g_q[x] \end{bmatrix}$ and $\begin{bmatrix} v_p[x] & v_q[x] \end{bmatrix}$ from the right, and store the new values into shared memory.
7. Compute $b_s = (\Sigma\rho_s')/32$, similarly to subphase 3, and increase a per-sweep counter of proper rotations B_r by b_s. This concludes the actions of one p-step.

After the sweep r finishes, update B, the total number of proper rotations in the thread block, by B_r. If no more sweeps follow, i.e., if $A_r = 0$ (no rotations have been performed in the last sweep), or $r = M_{\mathrm{s}}$ (the maximal number of sweeps is reached), the thread $[0, 0]$ atomically adds B to the global rotation counter B, mapped from the CPU memory.

In B the number of rotations from all thread blocks is accumulated. Note that B is accessible from both the CPU and the GPU and is the sole quantum of information needed to stop the global Jacobi process. More details about a GPU-wide stopping criterion can be found in subsection 2.3.6. This ends the `orthogonalize` phase.

The dot-products. Relying on a trick from [35], the dot-products could possibly be computed more accurately by using one `fma` and one negation per vector element in addition to one multiplication, and two +-reductions instead of one.

Let $c = a \odot_{\mathrm{RD}} b$, and $d = \mathtt{fma}(a, b, -c)$, where RD stands for rounding towards $-\infty$. Then it is easy to show, by looking separately at the both possible signs of c, that for the rounding error of the multiplication extracted by the `fma` holds $d \geq 0$. By +-reducing d across a warp, no cancellation can occur. That value may be added to the +-reduction result on c, to form the final dot-product.

This approach has been very briefly tested for the generalized SVD version of the code, without a dramatic slowdown but also without any noticeable effect on accuracy of the result. However, much more testing is needed in the future.

2.3.4. The Jacobi rotations

The numerically stable, state-of-the-art procedure of computing the trigonometric Jacobi rotations is described in [25]. The procedure relies on computing the column norms reliably, as described in section 2.6.

The rest of the computation from [25] is straightforward to implement. Since the entire shared memory per thread block is occupied, storing and updating the column scales, as in [2], is not possible without changing the shared memory configuration and reducing the L1 cache. The memory traffic that would thus be incurred overweights the two additional multiplications by a cosine per GPU thread. Therefore, rotations in the following form (`fma` followed by a multiplication, if $\cos\varphi \neq 1$) are chosen,

$$\begin{bmatrix} g_p' & g_q' \end{bmatrix} = \cos\varphi \begin{bmatrix} g_p & g_q \end{bmatrix} \begin{bmatrix} 1 & \tan\varphi \\ -\tan\varphi & 1 \end{bmatrix}. \tag{2.13}$$

The hyperbolic rotations may be computed similarly to the trigonometric ones, by adapting the ideas from [25], in the form

$$\begin{bmatrix} g_p' & g_q' \end{bmatrix} = \cosh\varphi \begin{bmatrix} g_p & g_q \end{bmatrix} \begin{bmatrix} 1 & \tanh\varphi \\ \tanh\varphi & 1 \end{bmatrix}. \tag{2.14}$$

Let `DDRJAC` be a procedure that computes, as in [25], the trigonometric rotations in form (2.13) from the column norms it obtains by calling `DRDSSQ` (see section 2.6.). On a Fermi GPU (matrix order 6144), `DDRJAC` is only 14% slower than a simple procedure discusses in the following. However, the protection from the input columns of too large (or too small) a norm that `DDRJAC` offers has to be complemented by the QR factorization at all blocking levels, which is extremely expensive.

Assume instead that the Gram matrix formation, the ordinary scalar products, and the induced norm computations never overflow. By using only correctly rounded arithmetic, $\cos\varphi$ and $\tan\varphi$ of (2.13), or $\cosh\varphi$ and $\tanh\varphi$ of (2.14), may be computed as in (2.15)–(2.18) (an adapted version of the Rutishauser's formulas [64]):

$$h = h_{qq} - \mathtt{t} \cdot h_{pp}; \quad \operatorname{ct} 2\varphi = \mathtt{t} \cdot \frac{h}{2h_{pq}}; \tag{2.15}$$

$$|\operatorname{ct}\varphi| = |\operatorname{ct} 2\varphi| + \sqrt{\mathtt{fma}(\operatorname{ct} 2\varphi, \operatorname{ct} 2\varphi, \mathtt{t})}; \tag{2.16}$$

$$\operatorname{tn}\varphi = \operatorname{sign}(\operatorname{ct} 2\varphi) \cdot \mathtt{rcp}(|\operatorname{ct}\varphi|); \tag{2.17}$$

$$\operatorname{cs}_1\varphi = \mathtt{rcp}(\sqrt{\mathtt{fma}(\mathtt{t}\cdot\operatorname{tn}\varphi, \operatorname{tn}\varphi, 1)}); \quad \operatorname{cs}_2\varphi = \frac{|\operatorname{ct}\varphi|}{\sqrt{\mathtt{fma}(|\operatorname{ct}\varphi|, |\operatorname{ct}\varphi|, \mathtt{t})}}. \tag{2.18}$$

Formulas (2.15)–(2.18), for $\mathtt{t} = 1$, produce the parameters of a trigonometric rotation (ct = cot, tn = tan, cs = cos), and for $\mathtt{t} = -1$, of a hyperbolic rotation (ct = coth, tn = tanh, cs = cosh). If, numerically, $|\coth 2\varphi| = 1$, it is substituted by $5/4$ (see [79]). In all other hyperbolic cases $|\tanh\varphi| < 1$ holds.

If $|\cot 2\varphi| < \sqrt{\varepsilon}$, then $\sqrt{\mathtt{fma}(\cot 2\varphi, \cot 2\varphi, 1)} = 1$, and (2.16) in the trigonometric case simplifies to $|\cot\varphi| = |\cot 2\varphi| + 1$. If $|\operatorname{ct} 2\varphi| \geq \sqrt{2/\varepsilon}$, then (barring an overflow) $\sqrt{\mathtt{fma}(\operatorname{ct} 2\varphi, \operatorname{ct} 2\varphi, \mathtt{t})} = |\operatorname{ct} 2\varphi|$, with (2.16) and (2.18) simplifying to

$|\operatorname{ct}\varphi| = 2 \cdot |\operatorname{ct} 2\varphi|$ and $\operatorname{cs}\varphi = 1$, respectively. These simplifications avoid taking square roots and a possible overflow of $\operatorname{ct}^2 2\varphi$, at a price of at most three floating-point comparisons. If $|\operatorname{ct} 2\varphi| \le \nu/4$, with ν being the largest positive normalized floating-point number, then tn is normalized.

If $\texttt{rsqrt}(x) = 1/\sqrt{x}$ were correctly rounded in CUDA, $\operatorname{cs}_1\varphi$ could be written as

$$\operatorname{cs}_1'\varphi = \texttt{rsqrt}(\texttt{fma}(\texttt{t} \cdot \operatorname{tn}\varphi, \operatorname{tn}\varphi, 1)). \tag{2.19}$$

Although (2.19) has only one iterative operation (`rsqrt`) instead of two (`rcp` and $\sqrt{x}$), and thus has a potential to be faster than (2.18), while also improving the orthogonality of V, (2.19) has been omitted from the final testing due to a slowdown of about 1%, using a correctly rounded-to-nearest `rsqrt` prototype implementation[5], that could be expected to vanish with the subsequent implementations of `rsqrt`.

In (2.18) and (2.19) there are three mathematically (but not numerically) equivalent expressions, $\operatorname{cs}_1\varphi$, $\operatorname{cs}_1'\varphi$, and $\operatorname{cs}_2\varphi$, which compute the cosine. By a similar analysis as above, $|\operatorname{ct}\varphi| \ge \sqrt{2/\varepsilon}$ implies $\operatorname{cs}_2\varphi = 1$, $\operatorname{tn}\varphi \le \sqrt{\varepsilon/2}$, and therefore $\operatorname{cs}_1\varphi = 1$ and $\operatorname{cs}_1'\varphi = 1$. Testing that condition also avoids an overflow of $\operatorname{ct}^2\varphi$.

A choice of the rotation formulas

In the block Jacobi algorithms, it is vital to preserve (J-)orthogonality of the accumulated V. In the hyperbolic case, the perturbation of the hyperbolic singular values also depends on the condition number of V [40, Proposition 4.4]. A simple attempt would be to try to compute each rotation as (J-)orthogonal as possible, without sacrificing performance.

Departure from a single rotation's (J-)orthogonality should be checked in a sufficiently high (e.g., 128-bit quadruple) precision, as $d_t = |(\cos^2\varphi + \sin^2\varphi) - 1|$, or as $d_h = |(\cosh\varphi - \sinh\varphi)(\cosh\varphi + \sinh\varphi) - 1|$, with $\sin\varphi = \cos\varphi * \tan\varphi$, or $\sinh\varphi = \cosh\varphi * \tanh\varphi$. For each binary exponent $-53 \le \texttt{e} \le 53$, 2^{24} uniformly distributed pseudorandom 52-bit integers $\texttt{m}_i$ had been generated on a CPU, to form $|\operatorname{ct} 2\varphi|_i$ with the exponent $\texttt{e}$ and the non-implied bits of the significand equal to $\texttt{m}_i$. From $|\operatorname{ct} 2\varphi|_i$ and (2.16)–(2.18), $(\operatorname{tn}\varphi)_i$, $(\operatorname{cs}_1\varphi)_i$, and $(\operatorname{cs}_2\varphi)_i$ had been computed in double precision. In the Fortran's quadruple arithmetic the corresponding d_t and d_h were then found and averaged, over all tested exponents. The results are summarized in Table 2.1.

Table 2.1: The average departures from (J-)orthogonality of the rotations given by (2.16)–(2.18).

trigonometric rotations		hyperbolic rotations	
d_t with $\cos_1\varphi$	d_t with $\cos_2\varphi$	d_h with $\cosh_1\varphi$	d_h with $\cosh_2\varphi$
$8.270887 \cdot 10^{-17}$	$8.335956 \cdot 10^{-17}$	$7.575893 \cdot 10^{-17}$	$6.586691 \cdot 10^{-17}$

[5]Courtesy of Norbert Juffa of NVIDIA.

Table 2.1 indicates that $\cosh_2\varphi$ produces, on average, more J-orthogonal hyperbolic rotations than $\cosh_1\varphi$. In the trigonometric case it is the opposite, but with a far smaller difference. Orthogonality of the final V was comparable in the tests for both trigonometric versions, often slightly better (by a fraction of the order of magnitude) using $\cos_2\varphi$. Therefore, $\mathrm{cs}_2\,\varphi$ formulas were chosen for a full-scale testing.

It is still far from conclusive which formulas from (2.18) or (2.19), and for which ranges of $\mathrm{ct}\,2\varphi$, should be used. However, $\mathrm{cs}_2\,\varphi$ or $\mathrm{cs}_1'\,\varphi$ formulas might be an alternative to the established $\mathrm{cs}_1\,\varphi$ ones. A deeper analysis is left for future work.

2.3.5. The postmultiplication

This phase postmultiplies $\begin{bmatrix}G_\mathsf{p} & G_\mathsf{q}\end{bmatrix}$ and, optionally, $\begin{bmatrix}V_\mathsf{p} & V_\mathsf{q}\end{bmatrix}$ by V_pq' only if the rotation counter from `orthogonalize` is nonzero, i.e., if $V_\mathsf{pq}' \neq I_{32}$. Let A_pq alias the first half of G_pq (i.e., R_pq). The procedure, detailed in Algorithm 4, is based on the Cannon parallel matrix multiplication algorithm [15].

Algorithm 4: Pseudocode for Cannon-like postmultiplication by V_pq'.

```
Description: Input: $\begin{bmatrix}A_\mathsf{p} & A_\mathsf{q}\end{bmatrix}, A \in \{G, V\}$. Output: $\begin{bmatrix}A_\mathsf{p}' & A_\mathsf{q}'\end{bmatrix} = \begin{bmatrix}A_\mathsf{p} & A_\mathsf{q}\end{bmatrix} V_\mathsf{pq}'$.
for ($i = x$; $i < n$; $i \mathrel{+}= 32$)   // multiply the next chunk of $\begin{bmatrix}A_\mathsf{p} & A_\mathsf{q}\end{bmatrix}$ by $V_\mathsf{pq}'$
    $A_\mathsf{pq}[x,y] = A_\mathsf{p}[i,y]$;  $A_\mathsf{pq}[x,y'] = A_\mathsf{q}[i,y]$;  // load the RAM chunk into $A_\mathsf{pq}$
    __syncthreads();       // the shared memory writes must take effect
    $a_{xy} = a_{xy'} = 0$;                 // $\begin{bmatrix}A_\mathsf{p}' & A_\mathsf{q}'\end{bmatrix}$ elements kept in registers
    $j = (y + x) \bmod 32$;   $j' = (y' + x) \bmod 32$;     // initial skew modulo 32
    unrolled for ($k = 0$; $k < 32$; $++k$)          // multiply-and-cyclic-shift
        $a_{xy} = \mathtt{fma}(A_\mathsf{pq}[x,j], V_\mathsf{pq}'[j,y], a_{xy})$;            // update $A_\mathsf{p}'[i,y]$
        $a_{xy'} = \mathtt{fma}(A_\mathsf{pq}[x,j'], V_\mathsf{pq}'[j',y'], a_{xy'})$;          // update $A_\mathsf{q}'[i,y]$
        $j = (j + 1) \bmod 32$;   $j' = (j' + 1) \bmod 32$;  // cyclic shift modulo 32
    endfor
    __syncthreads();       // ensure that $A_\mathsf{pq}$ is free to be overwritten
    $A_\mathsf{p}'[i,y] = a_{xy}$;   $A_\mathsf{q}'[i,y] = a_{xy'}$;   // store the product in the RAM chunk
endfor
```

Finally, Figure 2.12 summarizes the entire `pStep` kernel, from the perspective of the shared memory state transitions per thread block. The GPU RAM is accessed by one read (when no rotations occur), or by two reads and one write per element of G (and, optionally, at most one read and write per element of V), with all operations fully coalesced. The only additional global memory traffic are the atomic reductions necessary for the convergence criterion in `orthogonalize`.

2.3.6. A GPU-wide convergence criterion

Contrary to the pointwise Jacobi algorithm, which is considered to converge when no rotations have been performed in a sweep, stopping of the block algorithms for large inputs is more complicated than observing no rotations in a block-sweep.

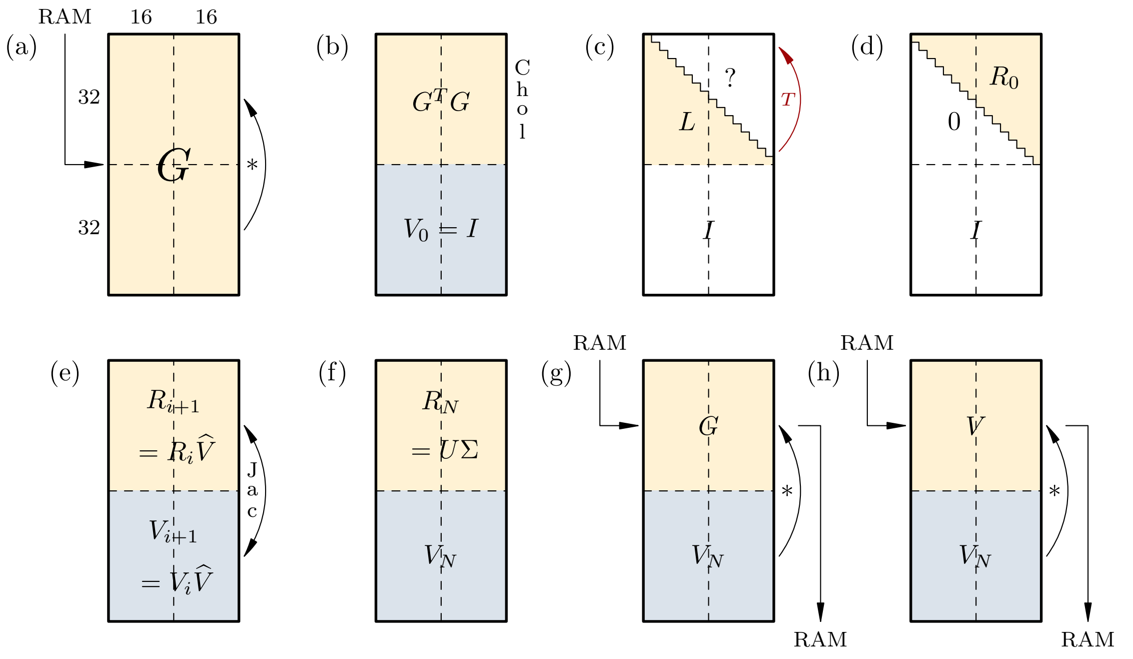


Figure 2.12: An overview of `pStep` kernel per thread block, the full block variant. Each subfigure depicts the state of the shared memory, the computation subphase performed, and the data transfer in or out of the GPU RAM. Subfigures (a)–(d) illustrate `factorize` with the Cholesky factorization, (e) and (f) belong to `orthogonalize`, and (g) and (h) to `postmultiply` phase.

There has to be an additional, more relaxed stopping criterion at the block level (motivated in what follows), while keeping the one at the inner (32×32) level unchanged.

The columns addressed by a pivot block-pair should be relatively orthogonal after completion of the `pStep` call in the full-block variant, but instead they may have departed from orthogonality, because (see [40, 69])

1. the accumulated rotations are not perfectly (J-)orthogonal, and
2. the postmultiplication introduces rounding errors.

Independently from that, in all variants, even the numerically orthogonal columns, when subjected to `factorize` (and its rounding errors), may result in the shortened ones that fail the relative orthogonality criterion.

If an orthogonality failure results from the first two causes, the ensuing rotations might be justified. However, if the failure is caused only by the rounding errors of the factorization process, the spurious rotations needlessly spoil the overall convergence.

To overcome this problem, a simple heuristics has been devised to avoid excessive block-sweeps with just a few rotations. These rotations are expected not to be proper, i.e., to have very small angles. Let `B` be a counter in the CPU RAM, mapped to the GPU RAM. The counter is reset at the beginning of each block-sweep and is updated from the `pStep` calls as described in subsection 2.3.3. At the end of a block-sweep, `B` contains the total number of proper rotations in that sweep. If $\mathsf{B} = 0$, or the maximal number of block-sweeps has been reached without convergence, the process stops.

This heuristics may skip over a relatively small number of legitimate rotations, but nevertheless produces reasonable relative errors in the computed singular values (see section 2.5.). A reliable way of telling (or avoiding) the exact cause of the orthogonality failures is strongly needed in that respect.

An alternative stopping criterion

The same software infrastructure (i.e., the allocated counters and their handling) herein described can also be used for another kind of convergence criterion. In essence, the new criterion substitutes a counter of transformations with nontrivial effects for a counter of “proper” (in the sense of having $\cos\varphi \neq 1$) rotations.

More formally, let V' be a union of V'_{pq} over all block pairs (p, q), let $v_p[x]$ be an element of V' to be transformed by a rotation, and let $v'_p[x]$ be the transformed value in that inner step. Note that both values can be simultaneously held in registers on an architecture with a large register file (e.g., Pascal). Then, compare $v_p[x]$ and $v'_p[x]$; if they are not equal (i.e., a nontrivial change has occured), then set the counter variable to 1, else keep the counter unchanged. Repeat the same for $v_q[x]$ and $v'_q[x]$. If only swapping of $v_p[x]$ and $v_q[x]$ has occured, treat that case as a trivial change. At the start of each block-sweep, reset the counter to 0. In all other aspects, treat the counter in the same way as it would be handled for the previously described criterion, where the counter is set to 1 if $\cos\varphi \neq 1$, and to 0 otherwise.

The motivation behind such a criterion is an observation that V', once it does not change in a particular block-sweep, i.e., does not accumulate any rotations, be they proper or not, it is hardly conceivable that it should start changing at some later block-sweep, at least not significantly enough to warrant a few extra block-sweeps at the end of the process. Note that such heuristics does not imply anything about (non)orthogonality of the columns of G'.

Complementary to looking at V', the “standard” stopping criterion in an inner sweep, based on relative orthogonality of the columns of R_{pq}, can be augmented by checking whether the elements $g_p[x]$ and $g'_p[x]$ (resp. $g_q[x]$ and $g'_q[x]$) differ after a transformation has been applied. The transformation still shall not be applied if the relative orthogonality criterion has been satisfied. However, if and when the elements are transformed, even by a proper rotation, there might be a chance (purely hypothetical, perhaps) that they have not actually changed. If the columns have not changed at all during the entire inner sweep, the standard criterion might not be satisfied, but then it will be impossible ever to reach convergence, and the only reasonable option would be to stop the inner process at that point. Such a situation does not preclude a possibility that the whole process will eventually converge; however, if there are no changes to G' during the entire block-sweep, what can be deduced from aggregating (i.e., +-reducing) the inner sweep change counters, the whole process should be terminated at once.

It is important to note that no transformations are ever skipped in any inner sweep due to this criterion in place that would not have been skipped otherwise, or that would have had no effect. Only the number of block-sweeps required to reach overall convergence may change, and thus, indirectly, some transformations at the end of the process, that would have been applied under the previously proposed criterion's regime, might never be performed.

Carefully implementing and testing the new criterion is left for future work.

2.3.7. An illustration of the algorithm's execution

As an illustration of the execution of the block-oriented algorithm's variant under $\mathfrak{R}^{\|}$ strategies, Figures 2.13, 2.14, 2.15, 2.16, and 2.17 depict, for an input matrix G of order 1024, the initial range of values of a non-negative symmetric matrix $|H|$, where $H = G^T G$, and the subsequent values of $|H|$, computed at the end of each block-sweep from the matrix G, by forming H and taking $\log_2 |H_{ij}|$ whenever the logarithm's argument is a positive normalized number, or $-\infty$ otherwise (which has never happend with the particular input). The colorbar at the right side of Figure 2.17 shows the ranges of the actual values of $|H|$ covered by the corresponding ranges of colors.

It can be seen from the Figures that the largest drop in the magnitudes of the off-diagonal elements happens in the block-sweeps 3 to 7. After that, H is almost diagonal, but still needs two more block-sweeps to clean the remaining unconverged regions close to the diagonal, while the majority of the pivot block-pairs (corresponding to the regions far away from the diagonal) require no transformations at all. That is the major drawback of any one-sided Jacobi-type SVD which blindly cycles through all pivot (block-)pairs, instead of trying to reduce the bandwidth around the diagonal from which the (block-)pivots for the last (block-)sweeps are taken when the quadratic convergence regime has been in effect, and detected (e.g., by a quadratical drop in the number of rotations performed in a (block-)sweep, compared to the previous ones).

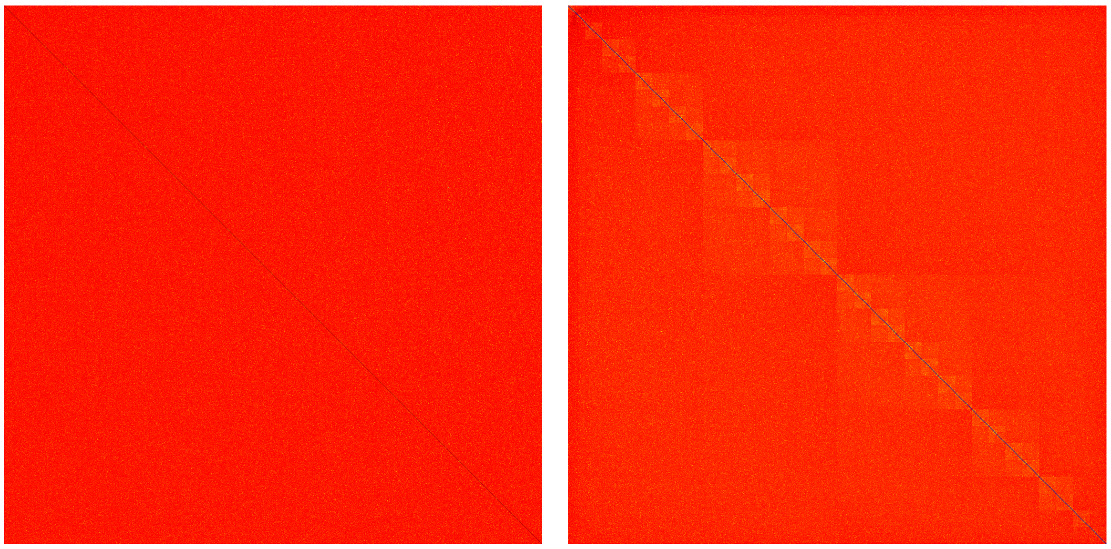


Figure 2.13: The initial $|G^T G|$, and $|G^T G|$ after block-sweep 1.

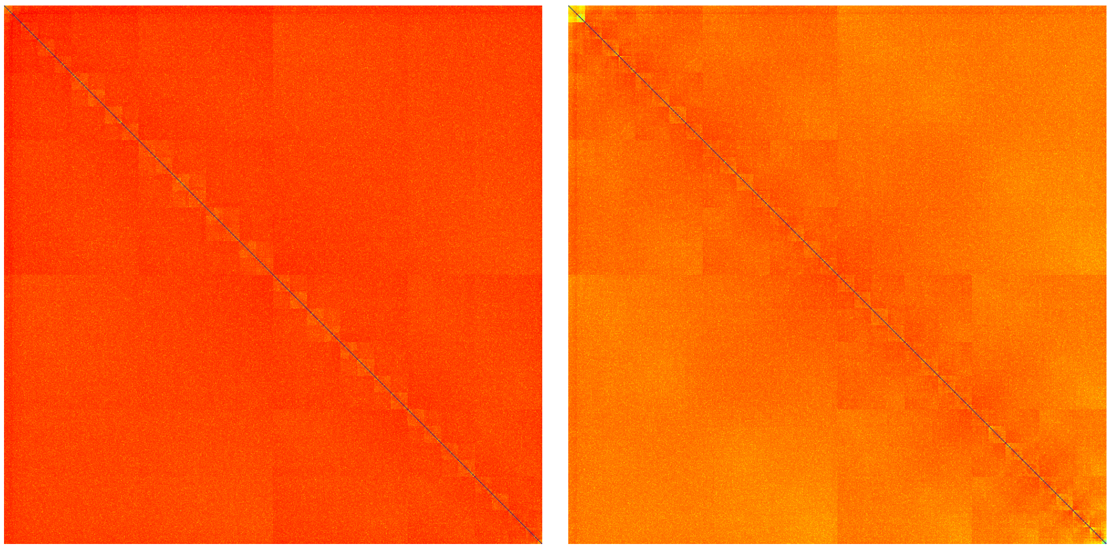


Figure 2.14: $|G^T G|$ after block-sweeps 2 and 3.

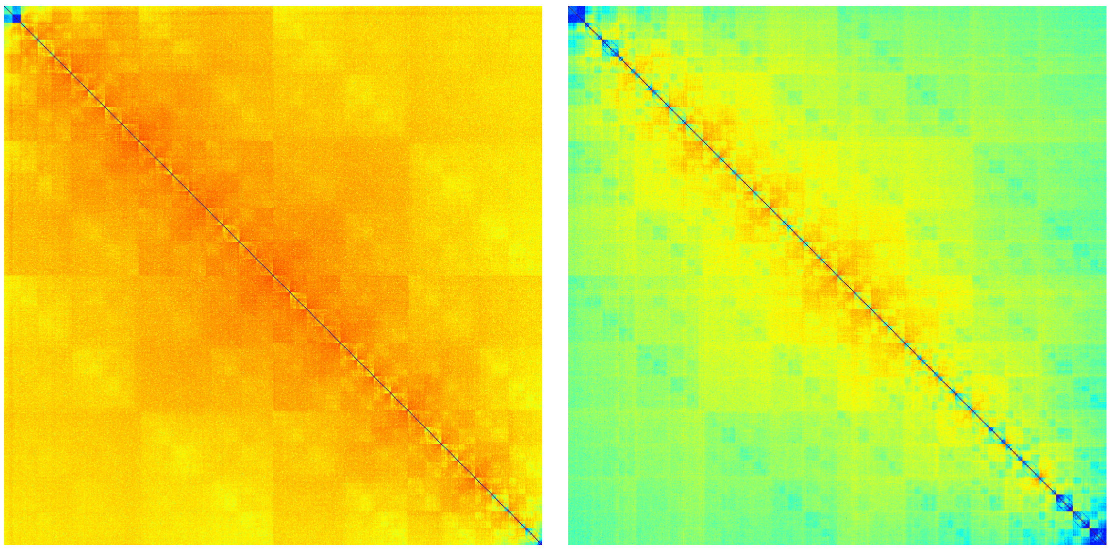


Figure 2.15: $|G^T G|$ after block-sweeps 4 and 5.

2.4. A multi-GPU algorithm

In this section the same blocking principles are applied one level up the hierarchy, to the case of multiple GPUs. As a proof-of-concept, the algorithm is developed on a 4-GPU Tesla S2050 system, and implemented as a single CPU process with 4 threads, where the thread $0, \ldots, 3$ controls the same-numbered GPU. Were the GPUs connected to multiple machines, on each machine the setup could be similar, with a CPU process and an adequate number of threads. Multiple processes on different machines could communicate via the CUDA-optimized message passing interface (MPI) subsystem. Except replacing the inter-GPU communication

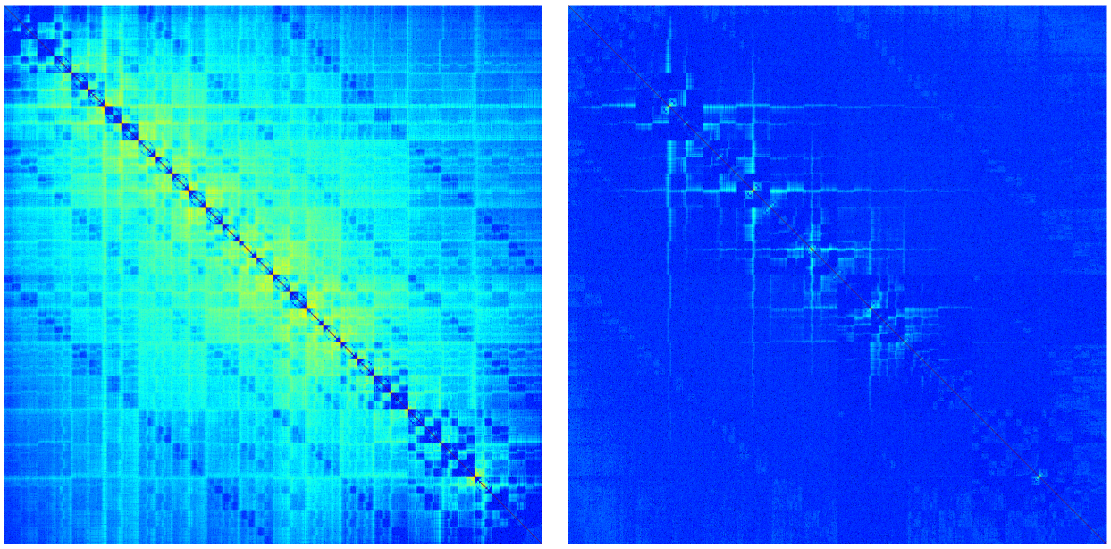


Figure 2.16: $|G^T G|$ after block-sweeps 6 and 7.

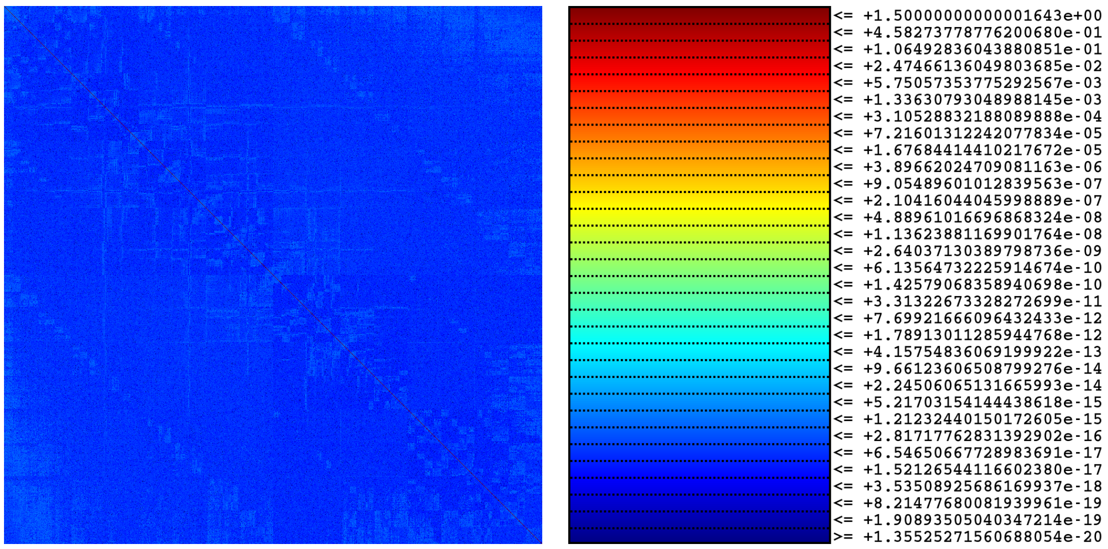


Figure 2.17: $|G^T G|$ after block-sweep 8. The heatmap corresponding to $|G^T G|$ after the last (9th) block-sweep is omitted because it is hardly distinguishable from the one for the 8th block-sweep. A colorbar, with the value ranges for $|G^T G|$ in $\log_2$ scale, is shown instead.

application programming interfaces (APIs), the algorithm would stay the same.

Each of g GPUs holds two block-columns addressed by a pivot block-pair with a total of $\mathsf{n} = n/\mathsf{g}$ columns. For simplicity, it is assumed that $n \bmod \mathsf{g} = 0$. After an outer block step, a single block-column on a GPU i is sent to a GPU j, and replaced by the one received from a GPU k, where j may or may not be equal to k, according to a block-column mapping $S''_{2\mathsf{g}}$. For the outer p-strategy $\mathcal{S}''_{2\mathsf{g}}$, a block-column mapping has to be chosen such that the communication rules implied by the mapping are (nearly) optimal for the given network topology.

For example, in a test system with $\mathtt{g} = 4$, a GPU i communicates with a GPU j, $j = i \operatorname{xor} 1$, faster than with the others. The amount of fast exchanges has been maximized within an outer block-sweep for $\mathcal{R}_8^{\|}$ (equivalent to Mantharam–Eberlein BR on a 2-dimensional hypercube) to 3 by choosing which pivot block-pair is assigned to which GPU in each block step. The result is a block-column mapping shown in Figure 2.18.

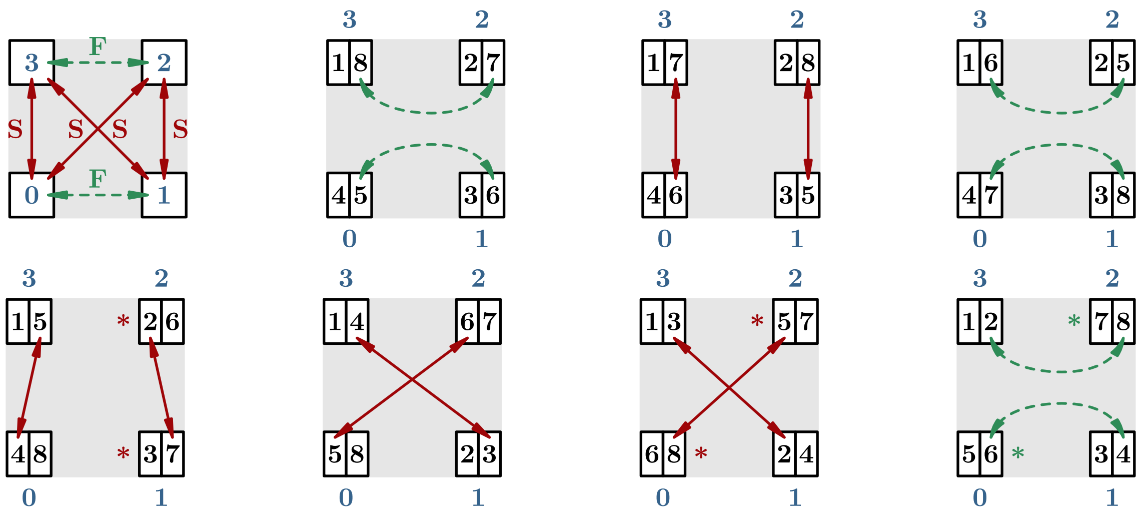


Figure 2.18: The block-column mapping in a single block-sweep of a p-strategy equivalent to Mantharam–Eberlein BR to GPUs $0, \ldots, 3$. The fast (F) communications for column exchanges between GPU peers are denoted by dashed curves, and the slow (S) exchanges by solid lines. Two-speed communication is defined on the top left subfigure. The (logical) column swaps needed to maintain $p < q$ are shown by an asterisk.

Besides the two outer block-columns of G (and, optionally, V), stored in G_A and V_A regions of the GPU RAM, respectively, an additional buffer space of the same size, G_B and V_B, has to be allocated to facilitate the BLAS-3-style matrix multiplications and the full-duplex asynchronous communication between GPUs. Also, for the shortening and the single-GPU Jacobi phases, two $\mathtt{n} \times \mathtt{n}$ matrices, X and Y, are needed. With a small auxiliary space AUX for the final singular value extraction, the total memory requirements are at most $m \times 5\mathtt{n}$ `double` elements per a GPU.

In an outer block step the following operations are performed (see Figure 2.19):

(0) form the Gram matrix $G_A^T G_A$ in X by `cublasDsyrk`;

(1) factorize $G_A^T G_A = R^T R$ by the Cholesky factorization (the hybrid MAGMA's `dpotrf_gpu` has been chosen, and this is the *only* place where a CPU is used for computation, which may be circumvented by a GPU-only implementation);

(2a) case (acc.): if accumulation of the product $\widehat{V}$ of the Jacobi rotations is desired, call a full SVD single-GPU Jacobi variant (full block, block-oriented, or hybrid) from section 2.3. on X, storing $\widehat{V}$ in Y; else

(2b) case (solve): copy R from X to Y, call a partial SVD single-GPU Jacobi variant on Y, and solve the triangular linear system $R\widehat{V} = \widehat{U}\widehat{\Sigma}$ for $\widehat{V}$ by `cublasDtrsm`, with the original R in X and $\widehat{V}$ overwriting $\widehat{U}\widehat{\Sigma}$ in Y;

(3) postmultiply G_A and V_A by $\widehat{V}$, using two `cublasDgemm` calls running in their own CUDA streams, and store the updated block-columns in G_B and V_B;

(4) ensure that all GPUs have completed the local updates by a device-wide synchronization (`cudaDeviceSynchronize`), followed by a process-wide thread synchronization (wait on a common barrier), and a suitable MPI collective operation (e.g., `MPI_Barrier`) in the multiprocess case;

(5) start, via CUDA streams, the asynchronous sends of one block-column from G_B and the corresponding one from V_B to another GPU, and start the asynchronous copies of the other column of G_B and the corresponding one of V_B to either the first or the second block-column of G_A and V_A, according to the block-column mapping rules for transition to the subsequent block step;

(6) wait for the outstanding asynchronous operations to finish by the same synchronization procedure as in (4), after which a block step is completed.

At the end of an outer block-sweep, the threads (and processes, where applicable) $+$-reduce their local counters B_i of proper rotations (cf. subsection 2.3.6.) to the system-wide number $\sum_i \mathsf{B}_i$ of proper rotations performed in all block steps in that sweep. If the result is 0, or the limit on the number block-sweeps has been reached, the iteration stops and the final singular values are extracted.

The full block variant of phases (2a) and (2b) usually has a 30-sweep limit for both the inner blocking and the pointwise, shared-memory Jacobi level. The block-oriented variant has both limits set to 1. Between them many hybrid variants may be interpolated.

Observe that phase (4) forces all GPUs to wait for the slowest one, in terms of the execution of phase (2a) or (2b). The full block variant exhibits the largest differences in running times between GPUs, depending on how orthogonal the block-columns are in the current block step. Although the full block variant is the fastest choice for a single-GPU algorithm (see section 2.5.), it may be up to 35% slower in a multi-GPU algorithm than the block-oriented variant, which has a predictable, balanced running time on all GPUs.

A reasonable hybrid variant might try to keep the running times balanced. A CPU thread that first completes the full block variant of (2a) or (2b) immediately informs other threads before proceeding to phase (4). The other threads then stop their inner block-sweeps loops in (2a) or (2b) when the running iteration is finished.

The wall-clock execution times of such an approach may be even lower than the block-oriented variant, but on the average are 10% higher. Moreover, both the full block and its hybrid variant induce the larger relative errors in Σ than the block-oriented variant. Such effect may be partially explained, as in section 2.5., by the same reasons valid for the single-GPU case (more rotations applied), but the larger differences in the multi-GPU case require further attention. Therefore, the numerical tests are presented for the block-oriented multi-GPU variant only.

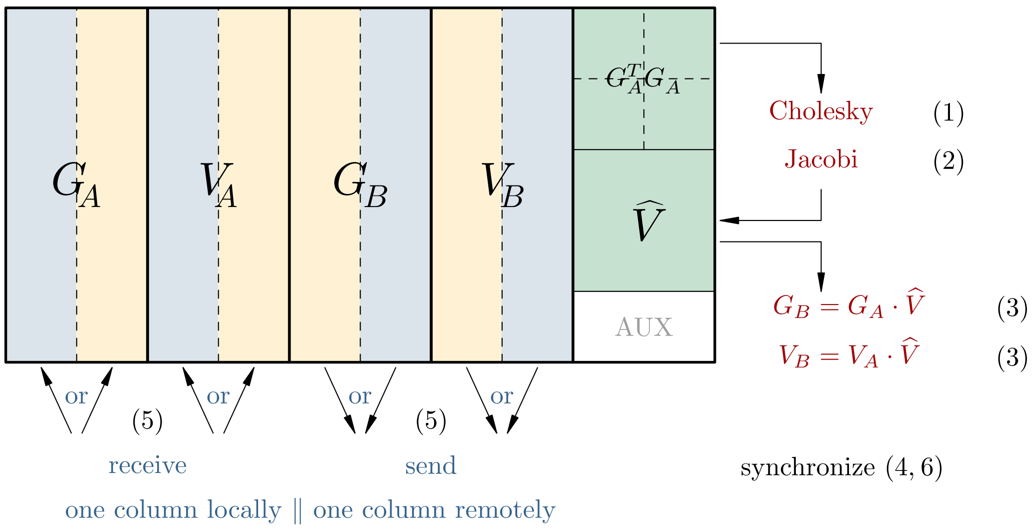

G_A
V_A
G_B
V_B
$G_A^T G_A$
$\widehat{V}$
AUX
Cholesky (1)
Jacobi (2)
$G_B = G_A \cdot \widehat{V}$ (3)
$V_B = V_A \cdot \widehat{V}$ (3)
or
(5)
receive
send
synchronize (4, 6)
one column locally ‖ one column remotely

Figure 2.19: Schematics of a GPU's memory organization and the outer block step phases. Operations with the same number may be performed concurrently, in streams, if the hardware allows that.

2.5. Numerical testing

In this section the testing data is defined, the hardware is described, and the speed and accuracy results for both the single-GPU and the multi-GPU implementations are presented. By these results the p-strategies ($\mathcal{R}^{\|}$ and $\mathcal{O}^{\|}$)[6] from section 2.2. are also confirmed as a reliable choice for implementing the fast Jacobi algorithms on the various parallel architectures.

Let `norm` and `unif` be the double precision pseudorandom number generators, such that $\texttt{norm}(\mu, \sigma)$ returns the nonzero samples from normal distribution $\mathcal{N}(\mu, \sigma)$, and $\texttt{unif}(S)$ returns the samples from the continuous uniform distribution $\mathcal{U}$ over S. The following pseudorandom spectra, for $k = 1, \ldots, 16$, have been generated:

1. $\Lambda_k^{(1)}(1{:}16) = 0.5;\quad \Lambda_k^{(1)}(17{:}1024k) = \texttt{norm}(0, 0.1)$,
2. $\Lambda_k^{(2)} = 1 + \Lambda_k^{(1)}$ (verified to be greater than zero),
3. $\Lambda_k^{(3)}(1{:}1024k) = \pm\,\texttt{unif}(\langle 10^{-7}, 10k\rangle)$, where a positive or a negative sign for each $\Lambda_k^{(3)}(i)$ is chosen independently for $1 \le i \le 1024k$ with equal probability,
4. $\Lambda_k^{(4)}(1{:}1024k) = \texttt{unif}(\langle 10^{-7}, 10k\rangle)$.

These arrays have been cast to the Fortran's quadruple precision type and denoted by $\mathbf{\Lambda}_k^{(1)}$ to $\mathbf{\Lambda}_k^{(4)}$. By a modified LAPACK `xLAGSY` routine, working in quadruple precision, a set of symmetric matrices $\mathbf{A}_k^{(j)} = \mathbf{U}_k^{(j)} \mathbf{\Lambda}_k^{(j)} [\mathbf{U}_k^{(j)}]^T$ has been generated, for $j = 1, \ldots, 4$, by pre- and postmultiplying $\mathbf{\Lambda}_k^{(j)}$ with a product $\mathbf{U}_k^{(j)}$ of the random Householder reflectors. The matrices $\mathbf{A}_k^{(j)}$ have then been factored by the symmetric

[6]Throughout this section, the subscripts indicating the matrix order on the p-strategies' symbols are omitted. For each occurrence of a particular symbol, the matrix order is implied by the context.

indefinite factorization with the complete pivoting [70] in quadruple precision:

$$P_k^{(j)}\mathbf{A}_k^{(j)}[P_k^{(j)}]^T = \widehat{\mathbf{G}}_k^{(j)}\widetilde{P}_k^{(j)}[\widetilde{P}_k^{(j)}]^T\widehat{J}_k^{(j)}\widetilde{P}_k^{(j)}[\widetilde{P}_k^{(j)}]^T[\widehat{\mathbf{G}}_k^{(j)}]^T = \mathbf{G}_k^{(j)}J_k^{(j)}[\mathbf{G}_k^{(j)}]^T.$$

The inner permutation $\widetilde{P}_k^{(j)}$ brings $\widehat{J}_k^{(j)}$ into $J_k^{(j)} = \operatorname{diag}(I, -I)$ form. For $j \in \{2, 4\}$ the symmetric indefinite factorization is equivalent to the Cholesky factorization with diagonal pivoting ($J_k^{(j)} = I$). Finally, $\mathbf{G}_k^{(j)}$ have been rounded back to double precision, and stored as the input factors $G_k^{(j)}$, along with $\Lambda_k^{(j)}$ and $J_k^{(j)}$.[7]

Since one of the important applications of the (H)SVD is the eigensystem computation of the symmetric (in)definite matrices, the procedure just described has been designed to minimize, as much as is computationally feasible, the effect of the rounding errors in the factorization part. Therefore, the relative errors of the computed $\Sigma^2 J$ (from $G = U\Sigma V^T$, and let $j_i = J_{ii}$) have been measured versus the given $\Lambda_k^{(j)}$ (with the elements denoted λ_i, for $1 \le i \le n$), i.e.,

$$\max_{i=1,\ldots,n} \frac{|\,f\ell(\sigma_i)^2 j_i - \lambda_i|}{|\lambda_i|}. \tag{2.20}$$

It may be more natural and reliable to compute the (H)SVD of $G_k^{(j)}$ in quadruple precision and compare the obtained singular values with the ones produced by the double precision algorithms. For an extremely badly conditioned $\mathbf{A}$, $\sqrt{|\Lambda|}$ may not approximate the singular values of G well; e.g., if, by the same procedure as above, $\mathbf{A}$ (definite or indefinite) is generated, with $n = 4096$ and $\kappa_2(\mathbf{A}) \ge 10^{24}$, the resulting G may have the singular values (found by the quadruple Jacobi algorithm) differing in at least 4–5 least significant double precision digits from the prescribed singular values $\sqrt{|\Lambda|}$. However, for a larger n, the quadruple precision SVD computation is infeasible. Therefore, accuracy of the algorithms has been verified as in (2.20), but with the modestly conditioned test matrices generated, to avoid the problems described.

The NVIDIA graphics testing hardware, with accompanying CPUs, consists of the following:

A. Tesla C2070 (Fermi) GPU and Intel Core i7–950 CPU (4 cores),
B. Tesla K20c (Kepler) GPU and Intel Core i7–4820K CPU (4 cores),
C. Tesla S2050 (Fermi) 4 GPUs and two Intel Xeon E5620 CPUs (2×4 cores).

The software used is CUDA 5.5 (nvcc and cuBLAS) under 64-bit Windows and Linux, and MAGMA 1.3.0 (with sequential and parallel Intel MKL 11.1) under 64-bit Linux.

As shown in Table 2.2, the sequential Jacobi algorithm `DGESVJ`, with the parallel MKL BLAS-1 operations, on machine C runs approximately 16 times slower than a single-GPU (Fermi) algorithm for the large enough inputs.

In Table 2.3 the differences in the execution times of the $\mathfrak{R}^{\|}$ p-strategy on Fermi and Kepler are given. There are the three main reasons, outlined in section 2.3., why the Kepler implementation is much faster than the Fermi one. In order of importance:

[7]The test matrices are available at http://euridika.math.hr:1846/Jacobi/ for download.

Table 2.2: The ratio of the wall-clock running times of `DGESVJ` on machine C versus a single GPU (Fermi) for the Cholesky factors of the matrices of order $n = 1024k$ with spectra $\Lambda_k^{(2)}$.

k	`DGESVJ`/$\mathfrak{R}^{\parallel}$	k	`DGESVJ`/$\mathfrak{R}^{\parallel}$	k	`DGESVJ`/$\mathfrak{R}^{\parallel}$	k	`DGESVJ`/$\mathfrak{R}^{\parallel}$
1	5.57	5	12.34	9	14.89	13	16.49
2	8.61	6	13.47	10	15.45	14	16.46
3	11.75	7	13.62	11	15.62	15	16.19
4	11.83	8	13.58	12	16.14	16	16.00

(i) 8-byte-wide shared memory banks on Kepler versus 4-byte-wide on Fermi—the profiler reports 99.8% shared memory efficiency on Kepler versus 49.8% on Fermi;

(ii) warp shuffle reductions on Kepler (the warp-level reductions do not need the shared memory workspace); and

(iii) no register spillage on Kepler, due to the larger register file.

The other profiler metrics are also encouraging: the global memory loads and stores are more than 99% efficient, and the warp execution efficiency on Fermi is about 96.5%, which confirms that the presented algorithms are almost perfectly parallel.

Table 2.3: The wall-clock running times (in seconds) of a Fermi (F) versus a Kepler (K) GPU for the full SVD of the Cholesky factors of the matrices of order $n = 1024k$ with spectra $\Lambda_k^{(2)}$, the full block variant.

k	Kepler [s]	Fermi [s]	K/F[%]	k	Kepler [s]	Fermi [s]	K/F[%]
1	1.413099	2.376498	59.5	9	506.365598	850.279539	59.6
2	7.206334	12.438532	57.9	10	682.577101	1153.337956	59.2
3	22.980686	35.783290	64.2	11	904.212224	1545.451594	58.5
4	46.357804	84.466500	54.9	12	1148.881987	1970.591570	58.3
5	95.828870	160.382859	59.8	13	1439.391787	2500.931105	57.6
6	154.643361	261.917934	59.0	14	1809.888207	3158.116986	57.3
7	246.114488	403.150779	61.0	15	2196.755474	3820.551746	57.5
8	346.689433	621.341377	55.8	16	2625.642659	4662.748709	56.3

Even though the instruction and thread block execution partial orders may vary across the hardware architectures, the presented algorithms are observably deterministic. Combined with a strong IEEE floating-point standard adherence of both the Fermi and the Kepler GPUs, that ensures the numerical results on one architecture are bitwise identical to the results on the other. This numerical reproducibility property should likewise be preserved on any future, standards-compliant hardware.

In the following, it is shown that $\mathcal{K}^{\parallel}$ and $\mathcal{J}^{\parallel}$ p-strategies are superior in terms of speed to $\mathcal{R}^{\parallel}$, $\mathcal{C}^{\parallel}$, the Brent and Luk ($\mathcal{B}$), and modified modulus ($\mathcal{M}$) strategies, in both the definite and the indefinite case. By abuse of notation, let $\mathcal{K}_b^{\parallel}$ stand for the block-oriented variant (otherwise, the full block variant is measured), and let $\mathcal{K}_{4b}^{\parallel}$ be used for its 4-GPU implementation. Figures 2.20 and 2.21 depict the wall-clock time ratios of the other strategies versus $\mathcal{K}^{\parallel}$. Except $\mathcal{B}$ and $\mathcal{J}^{\parallel}$, the other strategies are consistently about 14–21% slower than $\mathcal{K}^{\parallel}$, while $\mathcal{J}^{\parallel}$ is almost equally fast as $\mathcal{K}^{\parallel}$. Therefore, in what follows only $\mathcal{K}^{\parallel}$ has been timed.

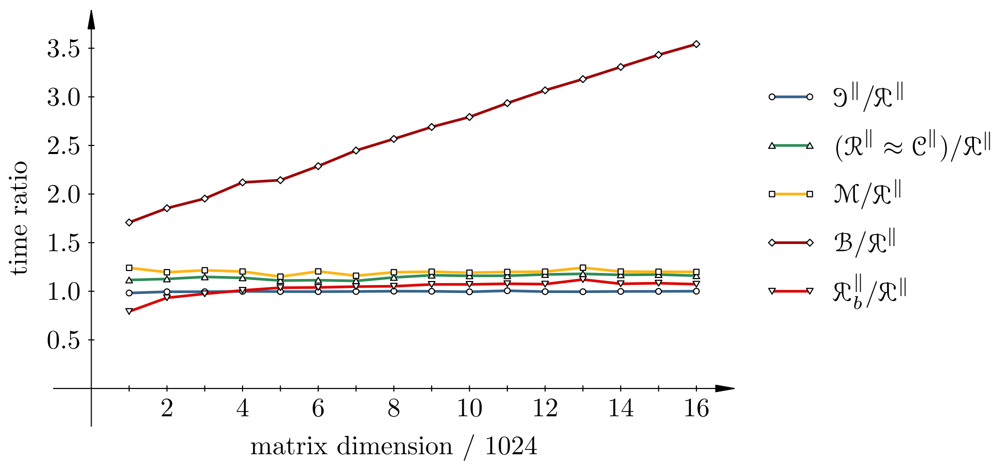


Figure 2.20: The wall-clock time ratio of the various parallel strategies on a single GPU (Fermi). The test spectra are $\Lambda_k^{(2)}$.

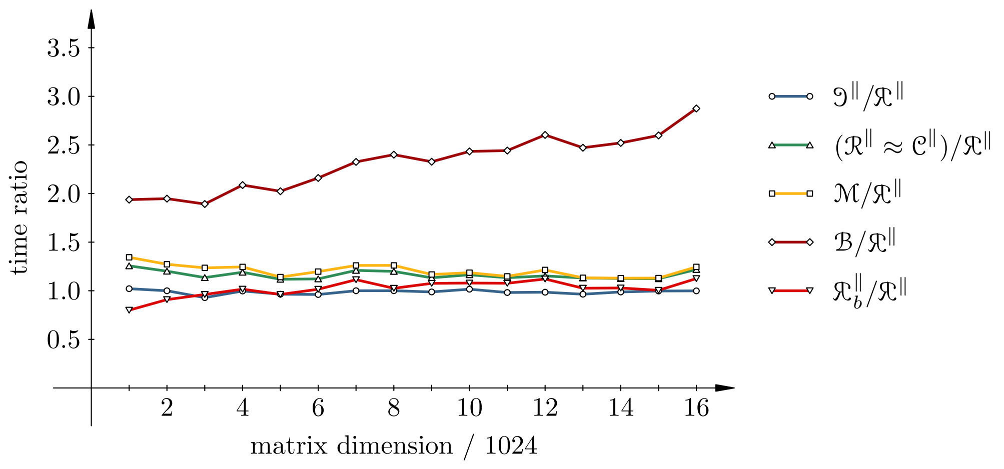


Figure 2.21: The wall-clock time ratio of the various parallel strategies on a single GPU (Fermi). The test spectra are $\Lambda_k^{(1)}$.

The standard counter-example that shows nonconvergence of the Jacobi method under the Brent and Luk strategy for matrices of even orders (first constructed by Hansen in [36], and later used in [50]), is actually not a counter-example in the usual diagonalization procedure which skips the rotations with the very small angles, because there is no need for diagonalization of an already diagonal matrix of order 2. On the contrary, the standard algorithm will diagonalize this matrix in only one (second) step of the first sweep.

However, this still does not mean that no serious issues exist regarding convergence of the Jacobi method under $\mathcal{B}$. Figures 2.20 and 2.21 indicate that a further investigation into the causes of the extremely slow convergence (approaching 30 block-sweeps) under $\mathcal{B}$ may be justified.

The block-oriented variant has more block-sweeps and, while slightly faster for the smaller matrices, is about 7% slower for the larger matrices than the full block variant. It may be more accurate in certain cases (see Figures 2.22 and 2.23), due to the considerably smaller total number of the rotations performed, as shown in Table 2.4. The strategies $\mathcal{M}$ and $\mathcal{B}$ are far less accurate than the new p-strategies.

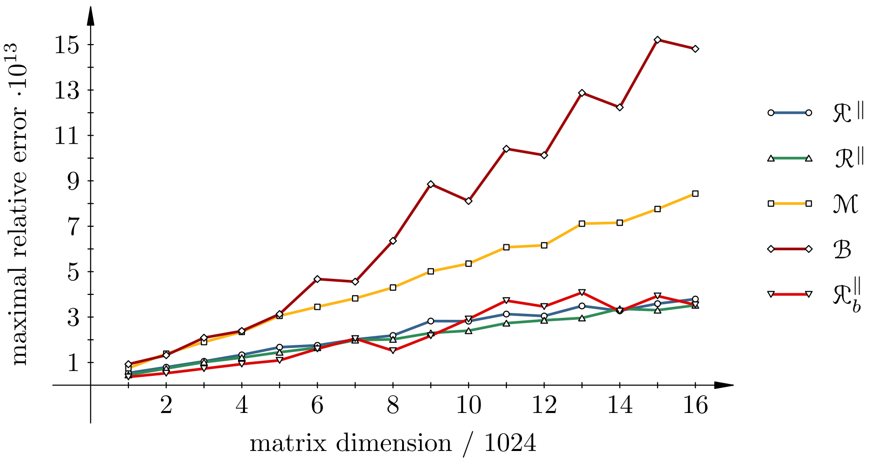


Figure 2.22: The relative accuracy of the various parallel strategies on a single GPU. The test spectra are $\Lambda_k^{(2)}$.

Table 2.4: The number of block-sweeps and the average ratios (with small variance) of the total number of rotations of the full block versus the block-oriented variant, per four spectrum types on a single GPU.

Spectrum type	1	2	3	4
average ratio of the number of rotations $\mathfrak{R}^{\|}/\mathfrak{R}_b^{\|}$	2.29	2.10	2.30	2.08
range of the number of block-sweeps $\mathfrak{R}^{\|}$	8–12	8–9	8–11	7–9
range of the number of block-sweeps $\mathfrak{R}_b^{\|}$	10–14	9–12	9–14	9–12

MAGMA's `dgesvd` routine has been tested with the sequential (seq.) and the parallel (par.) (four threads) MKL library on machine A. The relative accuracy is identical in both cases. Compared with the single-GPU Fermi algorithm, MAGMA

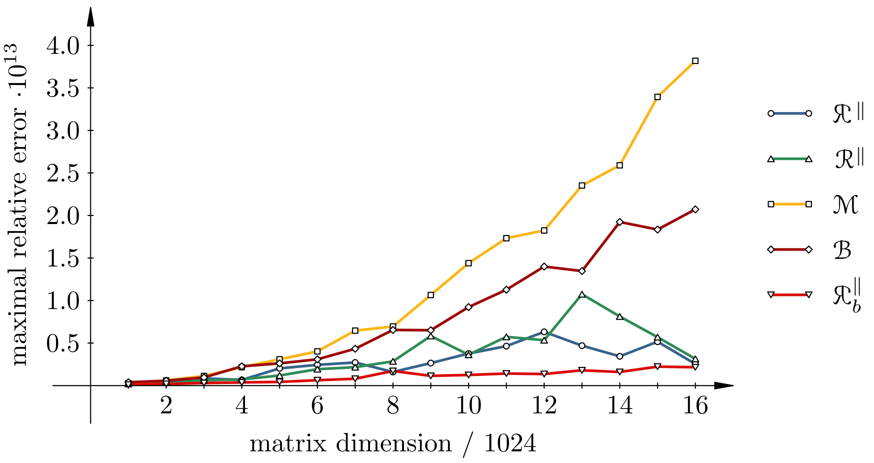


Figure 2.23: The relative accuracy of the various parallel strategies on a single GPU. The test spectra are $\Lambda_k^{(1)}$.

(seq.) is 1.5–3 times slower, and MAGMA (par.) is up to 2 times faster. On the other hand, MAGMA (par.) is, on average, 30% slower, and for the larger matrix sizes, more than 45% slower than the block-oriented 4-GPU Fermi (solve) implementation. The (acc.) implementation is about 35% slower than (solve) (see Figures 2.24 and 2.25), and only marginally more accurate (see Figures 2.26 and 2.27). For the matrix orders of at least 4096, the fastest Jacobi implementation on 4 GPUs is about 2.7 times faster than the fastest one on 1 GPU.

MAGMA's accuracy is comparable to a single-GPU algorithm for the well-conditioned test matrices, and better than a multi-GPU algorithm, but in the (separately tested) case of matrices with badly scaled columns ($\kappa_2 \approx 10^{12}$), the relative errors of MAGMA could be more than 20 times worse than the Jacobi ones.

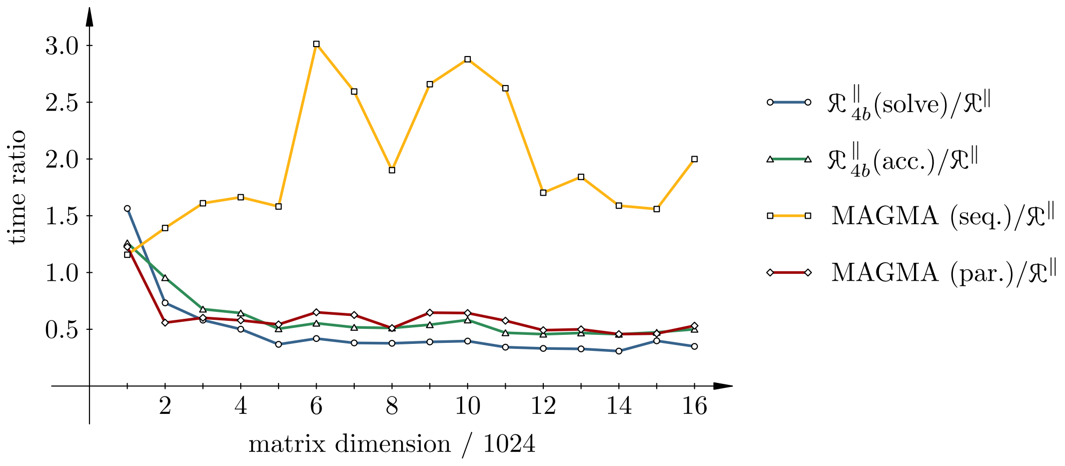


Figure 2.24: The wall-clock time ratio of the block-oriented 4-GPU Fermi implementations and MAGMA versus a single GPU. The test spectra are $\Lambda_k^{(2)}$.

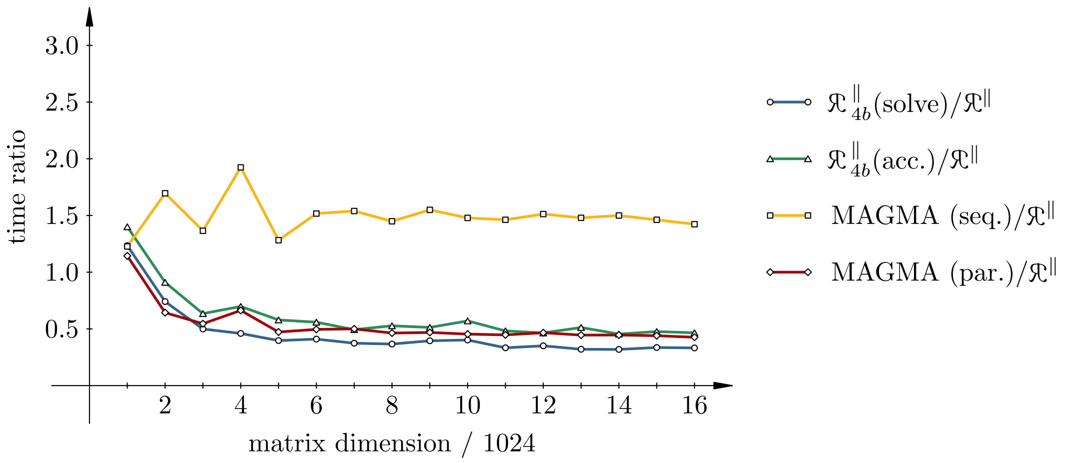


Figure 2.25: The wall-clock time ratio of the block-oriented 4-GPU Fermi implementations and MAGMA versus a single GPU. The test spectra are $\Lambda_k^{(4)}$.

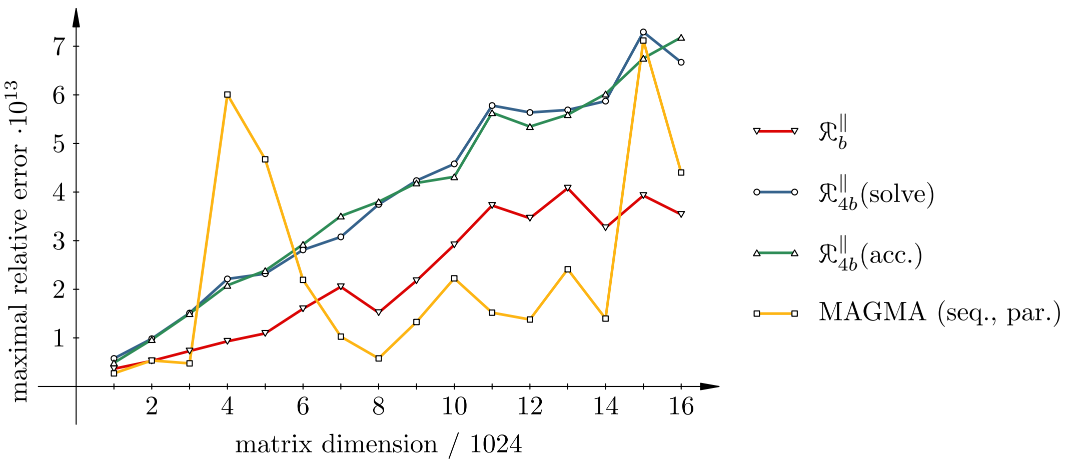


Figure 2.26: The relative accuracy of the block-oriented 1- and 4-GPU Fermi implementations and MAGMA. The test spectra are $\Lambda_k^{(2)}$.

Unlike MAGMA, the Jacobi GPU algorithms are perfectly scalable to an arbitrary number of GPUs, when the matrix order is a growing function of the number of assigned GPUs. That makes the Jacobi-type algorithms readily applicable on the contemporary large-scale parallel computing machinery, which needs to leverage the potential of a substantial amount of numerical accelerators.

2.5.1. Modern CUDA on modern hardware

While the GPU hardware and the CUDA software constantly evolve, the previously shown results do not reflect the latest performance gains, made possible by having the GPUs with more registers available per thread, and with faster clocks.

The source code required virtually no change to be compiled with CUDA 8.0,

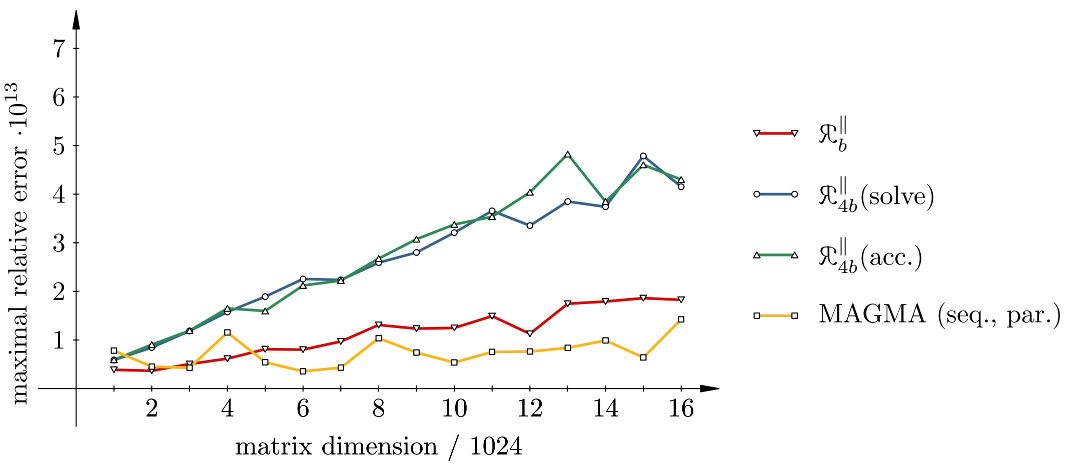


Figure 2.27: The relative accuracy of the block-oriented 1- and 4-GPU Fermi implementations and MAGMA. The test spectra are $\Lambda_k^{(4)}$.

and only some syntactical changes for the warp shuffles with (as of now not officially released) CUDA 9.0.

Though the latest NVIDA GPUs released are based on the Volta architecture, only the ones based on the penultimate architecture, Pascal, have been available for testing, in an IBM POWER8NVL ("Minsky") system with two CPUs, two pairs of NVLink-connected P100 GPUs, and CUDA 8.0 software stack.

Some numerical results on the Pascal GPU architecture with CUDA 8

Table 2.5 contains the wall-clock time results on a P100 GPU. The ratio B/F, also shown, is given normalized per block-sweep, i.e.,

$$\frac{B}{F} = \frac{\text{(wall-clock time of B.O.)/(number of block-sweeps)}}{\text{(wall-clock time of F.B.)/(number of block-sweeps)}} \cdot 100\%,$$

in order to demonstrate the time needed to prepare and execute one inner sweep (the B.O., i.e., block-oriented variant), versus preparing and executing a number of inner sweeps until convergence (the F.B., i.e., full block variant). The results have been obtained by using the correctly rounded `rsqrt` prototype code.

Such a huge performance gain compared to the Kepler results might be explained by having no register spillage at all (the compiler managed to fit all per-thread data in at most 127 registers), and by having all the hardware elements running faster than in the Kepler era, as well.

The Kogbetliantz-type (two-sided) SVD in CUDA 9

CUDA 9.0 has not yet been publicly released at the time of completing this thesis, but it should be noted that the cuSOLVER library might offer a GPU SVD algorithm that seems to be an implementation of the two-sided Kogbetliantz procedure [46], even though it bears Jacobi's name in the documentation. In its "early access"

Table 2.5: The wall-clock running times (in seconds) of the full block (F) versus the block-oriented (B) variant on a single P100 Pascal GPU with CUDA 8 for the full SVD of the Cholesky factors of order $n = 1024k$, with spectra $\Lambda_k^{(2)}$.

k	F.B. [s]	B.O. [s]	B/F [%]
1	0.339778	0.301851	78.97
2	1.850578	1.930034	83.43
3	4.152033	4.460573	85.94
4	10.162217	10.928215	86.03
5	17.116205	18.860442	90.16
6	30.998928	33.845042	89.33
7	43.047102	45.027426	94.14
8	67.491073	73.790574	89.45
9	100.458574	110.667394	90.13
10	127.070338	140.527818	90.48
11	174.411275	192.535960	90.32
12	209.581525	240.855742	86.19
13	277.494765	307.191926	90.57
14	326.804596	360.364868	90.22
15	414.276624	458.707541	90.59
16	514.069278	561.758535	89.41

version, the routine has shown promising performance on small matrices (from 1024 to 3072) on a consumer Kepler GPU, albeit lower than the algorithm proposed here.

2.6. Parallel norm computation

An essential prerequisite for computing the Householder reflectors and the Jacobi rotations [25] is obtaining the column norms (effectively, the sums of squares) reliably, avoiding the possible underflows and overflows of an ordinary scalar product. However, a strictly sequential nature of LAPACK's `DLASSQ` is unsuitable for parallel processing. Therefore, an alternative procedure, `DRDSSQ`, is proposed here, based on the parallel reduction concept.

The proposed procedure is one of the many approaches to compute a sum of squares of the floating-point numbers, but those approaches differ in their assumptions and in their ultimate goals. An assumption might be that the elements of a vector should be accessed only once (which will not hold for `DRDSSQ`), or that the floating-point arithmetic does not necessarily adhere to the standard [44] (here, such adherence is required).

The goal may be only to avoid the overflows and the underflows by scaling (as in [10]), or to compute the result as accurately as possible (as in [35]), with the overflow and underflow protection and/or vectorization as a bonus, or to seek reproducibility under various assumptions [20], etc.

Here, the goal is *not* the best possible accuracy or reproducibility as such (even though it is achieved on all current CUDA architectures as a consequence of the algorithm's design), but a high degree of paralellism under the SIMT computing model, the certainity of the overflows and the underflows being avoided, and an efficient way of utilizing the reductions, on the warp level and beyond.

Let μ be the smallest and ν the largest positive normalized floating-point number, ε the maximal relative roundoff error ($\varepsilon = 2^{-53}$ for `double` with rounding to nearest),

$\gamma = 1-\varepsilon$, $\delta = 1+\varepsilon$, and x a vector of length n, with no special values ($\pm\infty$, `NaN`s) for its components. A floating-point approximation of an exact quantity ξ is denoted by $\mathrm{rn}(\xi)$, $\mathrm{ru}(\xi)$, or $\mathrm{rz}(\xi)$, for rounding to nearest, to $+\infty$, or to 0, respectively.

Find $M := \max_i |x_i|$. If $M = 0$, x is a zero vector. Else, there exists the smallest nonzero $|x_i|$, which can be computed as $m := \min_i |x_i'|$, where $x_i' = x_i$ for $|x_i| > 0$, and $x_i' = \nu$ otherwise. Provided enough workspace for holding, or another technique for exchanging the partial results (such as the warp shuffle primitives of the Kepler GPU architecture), M and m could be found by a parallel min/max-reduction of x.

If the floating-point subnormals and infinity are supported, inexpensive, and safe to compute with (i.e., no exceptions are raised, or the non-stop exception handling is in effect), a sum of x_i^2 might also be computed. If the sum does not overflow, and a satisfactory accuracy is found to be maintained (e.g., the underflows could not have happened if $\mathrm{rn}(m^2) \geq \mu$), `DRDSSQ` stops here.

Otherwise, note that the depth of a reduction tree for the summation of x_i^2 is $\lceil \lg n \rceil$, and at each tree level (the first one being level 0) at most δ relative error is accumulated. Inductively, it follows that if, for some $s = 2^\ell$,

$$2^{\lceil \lg n \rceil}(sM)^2 \delta^{(1+\lceil \lg n \rceil)} \leq \nu,$$

then the sum of $(sx_i)^2$ cannot overflow. Also, if $(sm)^2\gamma \geq \mu$, for some $s = 2^k$, then no $(sx_i)^2$ can underflow. If some j could be substituted for both k and ℓ, it would define a scaling factor that simultaneously protects from the potential overflows and underflows. When the range of values of x does not permit a single scaling, the independent scalings of too large and too small values of x should be performed.

Such a scaling of x_i by s in the binary floating-point arithmetic introduces no rounding errors and amounts to a fast integer addition of the exponents of x_i and s. Instead of the scale factors themselves, only their exponents need to be stored and manipulated as machine integers. For clarity, the scales remain written herein as the integer powers of 2. A pair (s, y) thus represents a number with the same precision as y, but with the exponent equal to a sum of the exponents of s and y.

As motivated above, define the safe, inclusive bounds $\tilde{\mu}$ (lower) and $\hat{\nu}$ (upper) for the values of x for which no overflow nor underflow can happen, as

$$\tilde{\mu} = \sqrt{\mu/\gamma}, \quad \delta_n = 2^{\lceil \lg n \rceil}\delta^{(1+\lceil \lg n \rceil)}, \quad \hat{\nu} = \sqrt{\nu/\delta_n}.$$

Consider the following computations over a partition of the set of values of x:

- if $[m, M] \cap [\tilde{\mu}, \hat{\nu}] \neq \emptyset$, set $s_1 = 1 = 2^0$ (no scaling needed), and compute

$$\sigma_1^2 = \sum_{i=1}^{n} \bar{x}_i^2, \quad \bar{x}_i = \begin{cases} x_i, & \tilde{\mu} \leq |x_i| \leq \hat{\nu}, \\ 0, & \text{otherwise,} \end{cases}$$

- if $M > \hat{\nu}$, take the largest s such that $sM \leq \hat{\nu}$, denote it by s_2, and compute

$$\sigma_2^2 = \sum_{i=1}^{n} (s_2\hat{x}_i)^2, \quad \hat{x}_i = \begin{cases} x_i, & |x_i| > \hat{\nu}, \\ 0, & \text{otherwise,} \end{cases}$$

- if $m < \tilde{\mu}$, take the smallest s such that $sm \geq \tilde{\mu}$, denote it by s_0, and compute

$$\sigma_0^2 = \sum_{i=1}^{n} (s_0 \tilde{x}_i)^2, \quad \tilde{x}_i = \begin{cases} x_i, & |x_i| < \tilde{\mu}, \\ 0, & \text{otherwise.} \end{cases}$$

From m, M, $\tilde{\mu}$, $\hat{\nu}$ it is known in advance which partial sums are necessarily 0, and the procedure should be simplified accordingly. If, e.g., $m \geq \tilde{\mu}$, then $\sigma_0^2 = 0$.

A C/C++ implementation of finding $s_0 = 2^k$ or $s_2 = 2^\ell$ is remarkably simple. An expression `y = frexp(x, &e)` breaks `x` into $0.5 \leq$ `y` < 1 and `e` such that $2^{\mathtt{e}}\mathtt{y} = \mathtt{x}$. Let `f` $= m$, `t` $= \mathrm{ru}(\tilde{\mu})$, `j` $= k$ for s_0, or `f` $= M$, `t` $= \mathrm{rz}(\hat{\nu})$, `j` $= \ell$ for s_2. Also, let `fy = frexp(f, &fe)` and `ty = frexp(t, &te)`. Then `j` is returned by a code fragment:

```
j = (f <= t) ? (te - fe) + (fy < ty) : (te - fe) - (fy > ty).
```

If there is more than one nonzero partial sum of squares, such (s_i^{-2}, σ_i^2) are expressed in a "common form", $(\breve{s}_i^{-2}, \breve{\sigma}_i^2)$, where $0.5 \leq \breve{\sigma}_i^2 < 2$, and the scales' exponents remain even. Let $(s_i^{-2}, \sigma_i^2) = (2^j, 2^m y)$, where y is a significand of σ_i^2. Since σ_i^2 is normalized by construction, $1 \leq y < 2$. Define $m' = -(m \bmod 2)$ and $j' = j + m - m'$. Then $m' \in \{-1, 0\}$, j' remains even, and $(\breve{s}_i^{-2}, \breve{\sigma}_i^2) = (2^{j'}, 2^{m'} y)$.

The common form makes ordering the pairs by their magnitudes equivalent to ordering them lexicographically. First, find the two (out of at most three) partial sums which are the smallest by magnitude. Then add these partial sums together, such that the addend smaller by magnitude is rescaled to match the scale of the larger one. Let $(s_+^{-2}, \sigma_+^2) = (\breve{s}_\leq^{-2}, \breve{\sigma}_\leq^2) + (\breve{s}_>^{-2}, \breve{\sigma}_>^2)$, with $(\breve{s}_\leq^{-2}, \breve{\sigma}_\leq^2) \leq (\breve{s}_>^{-2}, \breve{\sigma}_>^2)$. Then $s_+^{-2} = \breve{s}_>^{-2}$, $s_-^{-2} = \breve{s}_\leq^{-2} / \breve{s}_>^{-2}$, and $\sigma_+^2 = s_-^{-2} \breve{\sigma}_\leq^2 + \breve{\sigma}_>^2$.

If one more addition is needed, (s_+^{-2}, σ_+^2) has to be brought into the common form $(\breve{s}_+^{-2}, \breve{\sigma}_+^2)$, and summed with the remaining addend by the above procedure. However, both (s_2^{-2}, σ_2^2) and (s_0^{-2}, σ_0^2) have to be computed only when $n\tilde{\mu}^2 \approx \varepsilon \hat{\nu}^2$. Such large n seldom occurs. In either case, accuracy of the final result is maintained by accumulating the partial sums in the nondecreasing order of their magnitudes.

The result of `DRDSSQ` is (s^{-2}, σ^2), and the norm of x is $\|x\|_2 = \sqrt{\sigma^2}/s$. If $\|x\|_2$ overflows or underflows for x a column of G, the input factor should be initially rescaled (if possible). A procedure similar to `DRDSSQ` is implementable wherever the parallel reduction is a choice (e.g., with `MPI_Allreduce` operation).

By itself, `DRDSSQ` does not guarantee numerical reproducibility, if the underlying parallel reductions do not possess such guarantees. The ideas from [20] might be useful in that respect.

3. The implicit Hari–Zimmermann algorithm for the GSVD

This chapter is organized as follows. Section 3.1. contains a description of two pointwise Jacobi-type algorithms for the GEP: the Falk–Langemeyer and the Hari–Zimmermann algorithm. The one-sided analogue of the Hari–Zimmermann algorithm for the GSVD is presented at the beginning of section 3.2. Then the algorithms are described that orthogonalize a matrix pair block-by-block, to maximize the efficiency on hierarchical memory architectures. The parallel algorithms on the shared and distributed memory architectures are detailed in section 3.3. The results of numerical testing are presented and discussed in section 3.4., and a GPU implementation of the implicit Hari–Zimermann is sketched in section 3.5.

3.1. The Jacobi–type sequential algorithms for GEP

3.1.1. The Falk–Langemeyer algorithm

The Falk–Langemeyer method solves the GEP (1.2) for a symmetric and definite matrix pair (A, B). The method constructs a sequence of matrix pairs $(A^{(\ell)}, B^{(\ell)})$,

$$A^{(\ell+1)} = C_\ell^T A^{(\ell)} C_\ell, \quad B^{(\ell+1)} = C_\ell^T B^{(\ell)} C_\ell, \quad \ell \in \mathbb{N}, \tag{3.1}$$

where $A^{(1)} := A$, and $B^{(1)} := B$, according to some pivot strategy that selects the order in which the off-diagonal elements are annihilated. If the transformation matrix C_ℓ is nonsingular, the two consecutive pairs are congruent, so their eigenvalues are equal [63]. A matrix that diagonalizes the pair is computed by accumulating the transformations

$$C^{(1)} = I, \qquad C^{(\ell+1)} = C^{(\ell)} C_\ell, \quad \ell = 1, 2, \ldots. \tag{3.2}$$

The matrix C_ℓ resembles a scaled plane rotation: it is the identity matrix, except for its (i, j)-restriction $\widehat{C}_\ell$, where it has two parameters, α_ℓ and β_ℓ

$$\widehat{C}_\ell = \begin{bmatrix} 1 & \alpha_\ell \\ -\beta_\ell & 1 \end{bmatrix}. \tag{3.3}$$

The parameters α_ℓ and β_ℓ in (3.3) are determined so that the transformations in (3.1) diagonalize the pivot submatrices

$$\widehat{A}^{(\ell)} = \begin{bmatrix} a_{ii}^{(\ell)} & a_{ij}^{(\ell)} \\ a_{ij}^{(\ell)} & a_{jj}^{(\ell)} \end{bmatrix}, \qquad \widehat{B}^{(\ell)} = \begin{bmatrix} b_{ii}^{(\ell)} & b_{ij}^{(\ell)} \\ b_{ij}^{(\ell)} & b_{jj}^{(\ell)} \end{bmatrix}. \tag{3.4}$$

The annihilation equations are

$$\begin{aligned}
\begin{bmatrix} 1 & -\beta_\ell \\ \alpha_\ell & 1 \end{bmatrix} \begin{bmatrix} a_{ii}^{(\ell)} & a_{ij}^{(\ell)} \\ a_{ij}^{(\ell)} & a_{jj}^{(\ell)} \end{bmatrix} \begin{bmatrix} 1 & \alpha_\ell \\ -\beta_\ell & 1 \end{bmatrix} &= \begin{bmatrix} a_{ii}^{(\ell+1)} & 0 \\ 0 & a_{jj}^{(\ell+1)} \end{bmatrix}, \\
\begin{bmatrix} 1 & -\beta_\ell \\ \alpha_\ell & 1 \end{bmatrix} \begin{bmatrix} b_{ii}^{(\ell)} & b_{ij}^{(\ell)} \\ b_{ij}^{(\ell)} & b_{jj}^{(\ell)} \end{bmatrix} \begin{bmatrix} 1 & \alpha_\ell \\ -\beta_\ell & 1 \end{bmatrix} &= \begin{bmatrix} b_{ii}^{(\ell+1)} & 0 \\ 0 & b_{jj}^{(\ell+1)} \end{bmatrix}.
\end{aligned} \tag{3.5}$$

In terms of α_ℓ and β_ℓ, the equations (3.5) reduce to

$$\begin{aligned}
\alpha_\ell a_{ii}^{(\ell)} + (1 - \alpha_\ell\beta_\ell) a_{ij}^{(\ell)} - \beta_\ell a_{jj}^{(\ell)} &= 0, \\
\alpha_\ell b_{ii}^{(\ell)} + (1 - \alpha_\ell\beta_\ell) b_{ij}^{(\ell)} - \beta_\ell b_{jj}^{(\ell)} &= 0,
\end{aligned}$$

and the solution can be written as

$$\alpha_\ell = \frac{I_j^{(\ell)}}{\nu_\ell}, \quad \beta_\ell = \frac{I_i^{(\ell)}}{\nu_\ell},$$

where

$$\begin{aligned}
I_i^{(\ell)} &= a_{ii}^{(\ell)} b_{ij}^{(\ell)} - a_{ij}^{(\ell)} b_{ii}^{(\ell)}, \qquad & I_{ij}^{(\ell)} &= a_{ii}^{(\ell)} b_{jj}^{(\ell)} - a_{jj}^{(\ell)} b_{ii}^{(\ell)}, \\
I_j^{(\ell)} &= a_{jj}^{(\ell)} b_{ij}^{(\ell)} - a_{ij}^{(\ell)} b_{jj}^{(\ell)}, \qquad & I^{(\ell)} &= (I_{ij}^{(\ell)})^2 + 4 I_i^{(\ell)} I_j^{(\ell)},
\end{aligned}$$
$$\nu_\ell = \frac{1}{2} \operatorname{sign}(I_{ij}^{(\ell)}) \Big(|I_{ij}^{(\ell)}| + \sqrt{I^{(\ell)}} \Big).$$

If $I^{(\ell)} = 0$, then a special set of formulas is used (for details, see [71, Algorithm 4]).

Now suppose that the matrices in (3.1) are computed according to the IEEE 754–2008 standard [44] for floating–point arithmetic. Unfortunately, for large matrices, after a certain number of transformations, the elements of both $A^{(\ell)}$ and $B^{(\ell)}$ can become huge in magnitude (represented as floating-point infinites). A solution to this problem is an occasional rescaling, but how often that needs to be done, depends on the dimension of the pair, the size of its elements, and the chosen pivot strategy.

3.1.2. The Hari–Zimmermann algorithm

If B is positive definite, the initial pair (A, B) can be scaled so that the diagonal elements of B are all equal to one. By taking

$$D = \operatorname{diag}\left(\frac{1}{\sqrt{b_{11}}}, \frac{1}{\sqrt{b_{22}}}, \ldots, \frac{1}{\sqrt{b_{kk}}} \right),$$

and making the congruence transformations

$$A^{(1)} := DAD, \quad B^{(1)} := DBD,$$

a new pair $(A^{(1)}, B^{(1)})$ is obtained, which is congruent to the original pair.

The idea behind this modification of the Falk–Langemeyer algorithm is: when B is the identity matrix, the transformations are the ordinary Jacobi rotations. The method constructs a sequence of matrix pairs

$$A^{(\ell+1)} = Z_\ell^T A^{(\ell)} Z_\ell, \quad B^{(\ell+1)} = Z_\ell^T B^{(\ell)} Z_\ell, \quad \ell \in \mathbb{N}, \tag{3.6}$$

such that the diagonal elements of $B^{(\ell)}$ remain ones after each transformation. The matrices Z_ℓ in (3.6) are chosen to annihilate the elements at positions (i,j) (and (j,i)), and to keep ones as the diagonal elements of $B^{(\ell)}$. If the matrix that diagonalizes the pair is needed, the accumulation procedure is given by (3.2), with C replaced by Z.

The matrix Z_ℓ is the identity matrix, except in the plane (i,j), where its restriction $\widehat{Z}_\ell$ is equal to

$$\widehat{Z}_\ell = \frac{1}{\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}} \begin{bmatrix} \cos\varphi_\ell & \sin\varphi_\ell \\ -\sin\psi_\ell & \cos\psi_\ell \end{bmatrix}, \tag{3.7}$$

with

$$\begin{aligned}
\cos\varphi_\ell &= \cos\vartheta_\ell + \xi_\ell(\sin\vartheta_\ell - \eta_\ell\cos\vartheta_\ell), \\
\sin\varphi_\ell &= \sin\vartheta_\ell - \xi_\ell(\cos\vartheta_\ell + \eta_\ell\sin\vartheta_\ell), \\
\cos\psi_\ell &= \cos\vartheta_\ell - \xi_\ell(\sin\vartheta_\ell + \eta_\ell\cos\vartheta_\ell), \\
\sin\psi_\ell &= \sin\vartheta_\ell + \xi_\ell(\cos\vartheta_\ell - \eta_\ell\sin\vartheta_\ell),
\end{aligned}$$

$$\begin{aligned}
\xi_\ell &= \frac{b_{ij}^{(\ell)}}{\sqrt{1+b_{ij}^{(\ell)}} + \sqrt{1-b_{ij}^{(\ell)}}}, \\
\eta_\ell &= \frac{b_{ij}^{(\ell)}}{\Big(1+\sqrt{1+b_{ij}^{(\ell)}}\Big)\Big(1+\sqrt{1-b_{ij}^{(\ell)}}\Big)},
\end{aligned} \tag{3.8}$$

$$\tan(2\vartheta_\ell) = \frac{2a_{ij}^{(\ell)} - \big(a_{ii}^{(\ell)} + a_{jj}^{(\ell)}\big)b_{ij}^{(\ell)}}{\big(a_{jj}^{(\ell)} - a_{ii}^{(\ell)}\big)\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}}, \qquad -\frac{\pi}{4} < \vartheta_\ell \le \frac{\pi}{4}.$$

If $a_{ij}^{(\ell)} = b_{ij}^{(\ell)} = 0$, set $\vartheta_\ell = 0$. Else, if $a_{ii}^{(\ell)} = a_{jj}^{(\ell)}$, and $2a_{ij}^{(\ell)} = \big(a_{ii}^{(\ell)} + a_{jj}^{(\ell)}\big)b_{ij}^{(\ell)}$, then the matrices $\widehat{A}^{(\ell)}$ and $\widehat{B}^{(\ell)}$ are proportional. Hence, set $\vartheta_\ell = \frac{\pi}{4}$.

To justify the notation, in the following it is proven that the quantities defined in (3.8) are indeed trigonometric functions of certain angles. Note that ϑ_ℓ in the specified range is always uniquely defined by $\tan(2\vartheta_\ell)$.

Proposition 3.1. *The quantities* $\cos\varphi_\ell$*,* $\sin\varphi_\ell$*,* $\cos\psi_\ell$*, and* $\sin\psi_\ell$ *represent the stated trigonometric functions of certain angles* φ_ℓ*, and* ψ_ℓ*, respectively.*

Proof. It is sufficient to prove that $\cos^2\varphi_\ell + \sin^2\varphi_\ell = 1$, and $\cos^2\psi_\ell + \sin^2\psi_\ell = 1$. From the first two lines of (3.8) it follows that

$$\cos^2\varphi_\ell + \sin^2\varphi_\ell = \cos^2\psi_\ell + \sin^2\psi_\ell = (1-\xi_\ell\eta_\ell)^2 + \xi_\ell^2 = 1 + \xi_\ell(-2\eta_\ell + \xi_\ell\eta_\ell^2 + \xi_\ell).$$

To prove the claim, it is sufficient to show that

$$\xi_\ell(-2\eta_\ell + \xi_\ell\eta_\ell^2 + \xi_\ell) = 0.$$

If $\xi_\ell = 0$, the conclusion is obvious. Otherwise, the result is obtained by substitution of ξ_ℓ and η_ℓ from (3.8) as the functions of $b_{ij}^{(\ell)}$, followed by simplification of this expression. □

Hari in his Ph.D. thesis proved [37, Proposition 2.4] that

$$\min\{\cos\varphi_\ell, \cos\psi_\ell\} > 0,$$

and

$$-1 < \tan\varphi_\ell \tan\psi_\ell \leq 1,$$

with $\tan\varphi_\ell \tan\psi_\ell = 1$, if and only if $\vartheta_\ell = \frac{\pi}{4}$. Hence, $\widehat{Z}_\ell$ is nonsingular. More precisely,

$$\begin{aligned}\det(\widehat{Z}_\ell) &= \frac{1}{1-\big(b_{ij}^{(\ell)}\big)^2}(\cos\varphi_\ell\cos\psi_\ell + \sin\varphi_\ell\sin\psi_\ell) = \frac{1}{1-\big(b_{ij}^{(\ell)}\big)^2}(1-2\xi_\ell^2)\\ &= \frac{1}{1-\big(b_{ij}^{(\ell)}\big)^2}\left(1-\frac{\big(b_{ij}^{(\ell)}\big)^2}{1+\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}}\right) = \frac{1}{\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}}.\end{aligned}$$

The transformations applied on B are congruences, and consequently, they preserve the inertia of B. Therefore, all matrices $B^{(\ell)}$ are positive definite. Since their diagonal elements are ones, the absolute value of all the other elements is smaller than one, i.e., the elements of $B^{(\ell)}$ cannot overflow. For the elements of $A^{(\ell)}$, the situation is slightly more complicated.

Proposition 3.2. *The elements of the matrix $A^{(\ell+1)} = Z_\ell^T A^{(\ell)} Z_\ell$ are bounded in terms of the elements of $A^{(\ell)}$, and the pivot element $b_{ij}^{(\ell)}$ of $B^{(\ell)}$. They are equal to the elements of $A^{(\ell)}$, except in the columns (and, due to symmetry, rows) with indices i and j, where $a_{ij}^{(\ell+1)} = 0$, and*

$$\max\Big\{|a_{ii}^{(\ell+1)}|, |a_{jj}^{(\ell+1)}|\Big\} \leq \frac{4\max\Big\{|a_{ii}^{(\ell)}|, |a_{ij}^{(\ell)}|, |a_{jj}^{(\ell)}|\Big\}}{1-\big(b_{ij}^{(\ell)}\big)^2},$$

$$\max\Big\{|a_{pi}^{(\ell+1)}|, |a_{pj}^{(\ell+1)}|\Big\} \leq \frac{2\max\Big\{|a_{pi}^{(\ell)}|, |a_{pj}^{(\ell)}|\Big\}}{\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}}, \quad p = 1,\dots,k.$$

Moreover, if A is positive definite, the bound for the diagonal elements of $A^{(\ell+1)}$ can be written as

$$\max\Big\{a_{ii}^{(\ell+1)}, a_{jj}^{(\ell+1)}\Big\} \leq \frac{4\max\Big\{a_{ii}^{(\ell)}, a_{jj}^{(\ell)}\Big\}}{1-\big(b_{ij}^{(\ell)}\big)^2}.$$

Proof. From the structure of Z_ℓ it follows that the elements of $A^{(\ell+1)}$ are equal to the elements of $A^{(\ell)}$, except in the pivot rows and columns, i.e., the rows and columns with indices i and j. Therefore,

$$\begin{aligned}
a_{ii}^{(\ell+1)} &= \frac{\cos^2\varphi_\ell a_{ii}^{(\ell)} - 2\cos\varphi_\ell \sin\psi_\ell a_{ij}^{(\ell)} + \sin^2\psi_\ell a_{jj}^{(\ell)}}{1-\big(b_{ij}^{(\ell)}\big)^2}, \\
a_{pi}^{(\ell+1)} = a_{ip}^{(\ell+1)} &= \frac{a_{pi}^{(\ell)}\cos\varphi_\ell - a_{pj}^{(\ell)}\sin\psi_\ell}{\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}}, \quad p \neq i,j, \\
a_{pj}^{(\ell+1)} = a_{jp}^{(\ell+1)} &= \frac{a_{pi}^{(\ell)}\sin\varphi_\ell + a_{pj}^{(\ell)}\cos\psi_\ell}{\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}}, \quad p \neq i,j, \\
a_{jj}^{(\ell+1)} &= \frac{\sin^2\varphi_\ell a_{ii}^{(\ell)} + 2\sin\varphi_\ell \cos\psi_\ell a_{ij}^{(\ell)} + \cos^2\psi_\ell a_{jj}^{(\ell)}}{1-\big(b_{ij}^{(\ell)}\big)^2},
\end{aligned} \tag{3.9}$$

and $a_{ij}^{(\ell+1)} = a_{ji}^{(\ell+1)} = 0$. From (3.9), for $s = i, j$ and $p \neq s$, it follows

$$|a_{ps}^{(\ell+1)}| = |a_{sp}^{(\ell+1)}| \leq \frac{|a_{pi}^{(\ell)}| + |a_{pj}^{(\ell)}|}{\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}} \leq \frac{2\max\Big\{|a_{pi}^{(\ell)}|, |a_{pj}^{(\ell)}|\Big\}}{\sqrt{1-\big(b_{ij}^{(\ell)}\big)^2}},$$

and

$$|a_{ss}^{(\ell+1)}| \leq \frac{|a_{ii}^{(\ell)}| + 2|a_{ij}^{(\ell)}| + |a_{jj}^{(\ell)}|}{1-\big(b_{ij}^{(\ell)}\big)^2} \leq \frac{4\max\Big\{|a_{ii}^{(\ell)}|, |a_{ij}^{(\ell)}|, |a_{jj}^{(\ell)}|\Big\}}{1-\big(b_{ij}^{(\ell)}\big)^2}.$$

The bound for positive definite matrices A follows from the fact that all matrices $A^{(\ell)}$ are positive definite, and all principal minors of order 2 in $A^{(\ell)}$ are positive, i.e., $|a_{ij}^{(\ell)}| \leq \sqrt{a_{ii}^{(\ell)} a_{jj}^{(\ell)}} \leq \max\{a_{ii}^{(\ell)}, a_{jj}^{(\ell)}\}$. □

Note that if $B = I$, then $\xi_\ell = \eta_\ell = 0$ for all ℓ in (3.8), and ϑ_ℓ is the angle from the ordinary Jacobi method for a single symmetric matrix A. In this case, $\widehat{Z}_\ell$ in (3.7) is the ordinary plane rotation for the angle ϑ_ℓ. A similar situation occurs near the end of the diagonalization process, when B is close to I, and the matrices $\widehat{Z}_\ell$ defined by (3.7) tend to the ordinary rotations. This reasoning is justified by the following convergence theorem, proved in [37].

Theorem 3.3 (Hari). *If B is positive definite, the row- and the column-cyclic Hari–Zimmermann method is globally convergent. There exists a permutation π of the generalized eigenvalues $\lambda_1 \leq \cdots \leq \lambda_k$ of the pair (A, B), such that*

$$\lim_{\ell\to\infty} A^{(\ell)} = \operatorname{diag}(\lambda_{\pi(k)}, \ldots, \lambda_{\pi(1)}), \qquad \lim_{\ell\to\infty} B^{(\ell)} = I.$$

Since the elements of $A^{(\ell+1)}$ are bounded, and the algorithm is convergent, it can be expected that the elements of $A^{(\ell)}$, for all ℓ, will not overflow, provided that the initial matrix B is not severely ill-conditioned.

3.2. The Jacobi–type algorithms for GSVD

3.2.1. The implicit Hari–Zimmermann algorithm

In the GSVD problem, two matrices $F_0 \in \mathbb{R}^{m\times n}$ and $G_0 \in \mathbb{R}^{p\times n}$ are given. If G_0 is not of full column rank, then after the preprocessing step from [6], a matrix pair of two square matrices (F, G) is obtained, with G of full rank k.

For such F and G, since G^TG is a positive definite matrix, the pair (F^TF, G^TG) in the corresponding GEP is symmetric and definite. There exist many nonsingular matrices Z that simultaneously diagonalize the pair (F^TF, G^TG) by congruences [63, Theorem 15.3.2, p. 344],

$$Z^TF^TFZ = \Lambda_F, \quad Z^TG^TGZ = \Lambda_G, \tag{3.10}$$

where Λ_F and Λ_G are diagonal matrices such that $(\Lambda_F)_{ii} \geq 0$ and $(\Lambda_G)_{ii} > 0$, for $i = 1, \dots, k$. Since $Z^TG^TGZ\Lambda_G^{-1} = I$, the problem (3.10) can be rewritten as

$$(F^TF)Z = Z^{-T} \cdot I \cdot \Lambda_F = Z^{-T}(Z^TG^TGZ\Lambda_G^{-1})\Lambda_F = (G^TG)Z(\Lambda_G^{-1}\Lambda_F),$$

and the eigenvalues of the pair (F^TF, G^TG) are the diagonal elements of the matrix $\Lambda_G^{-1}\Lambda_F$, while the generalized eigenvectors are the columns of the matrix Z.

First note that the congruence transformations by Z in (3.10) can be written as one-sided transformations from the right-hand side on F and G, with transformation parameters computed from F^TF and G^TG. Moreover, the final matrices Λ_F and Λ_G are diagonal, so the columns of FZ and GZ are orthogonal (not orthonormal), with the column norms equal to the square roots of the diagonal elements of Λ_F and Λ_G, respectively. Hence, (3.10) can be written as

$$FZ = U\Lambda_F^{1/2}, \quad GZ = V\Lambda_G^{1/2}, \tag{3.11}$$

where U and V are orthogonal matrices, and $(\;)^{1/2}$ denotes the principal square root of a matrix (see [42, Section 1.7]). Finally, if $\Lambda_F + \Lambda_G \neq I$, then define the diagonal scaling

$$S = (\Lambda_F + \Lambda_G)^{1/2}. \tag{3.12}$$

By setting

$$X := SZ^{-1}, \qquad \Sigma_F := \Lambda_F^{1/2}S^{-1}, \qquad \Sigma_G := \Lambda_G^{1/2}S^{-1}, \tag{3.13}$$

the relation (3.11) becomes (1.1), i.e., the GSVD of the pair (F, G). If only the generalized singular values are sought for, this rescaling is not necessary, and the generalized singular values are simply $\sigma_i = (\Lambda_G^{-1/2}\Lambda_F^{1/2})_{ii}$, for $i = 1, \dots, k$.

Now it is easy to establish a connection between the two-sided GEP algorithm and the one-sided GSVD algorithm. Suppose that the algorithm works in sweeps. Each sweep annihilates all pivot pairs (i, j), for $1 \leq i < j \leq k$, only once, in some prescribed cyclic order (for example, row- or column-cyclic). To determine the annihilation parameters from (3.7)–(3.8), the elements of the matrices $\widehat{A}^{(\ell)}$ and $\widehat{B}^{(\ell)}$ from (3.4) should be computed. If the pivot columns of $F^{(\ell)}$ and $G^{(\ell)}$ are denoted by $f_i^{(\ell)}$, $f_j^{(\ell)}$, $g_i^{(\ell)}$, and $g_j^{(\ell)}$, then the elements of $\widehat{A}^{(\ell)}$ and $\widehat{B}^{(\ell)}$ are the inner products of pivot columns of $F^{(\ell)}$ and $G^{(\ell)}$, respectively,

$$\begin{aligned} a_{ii}^{(\ell)} &= (f_i^{(\ell)})^T f_i^{(\ell)}, & a_{ij}^{(\ell)} &= (f_i^{(\ell)})^T f_j^{(\ell)}, & a_{jj}^{(\ell)} &= (f_j^{(\ell)})^T f_j^{(\ell)}, \\ b_{ii}^{(\ell)} &= (g_i^{(\ell)})^T g_i^{(\ell)}, & b_{ij}^{(\ell)} &= (g_i^{(\ell)})^T g_j^{(\ell)}, & b_{jj}^{(\ell)} &= (g_j^{(\ell)})^T g_j^{(\ell)}. \end{aligned} \tag{3.14}$$

The rest of the one-sided or the implicit algorithm consists of computing the elements of the transformation matrix from (3.8), and updating of the corresponding columns in $F^{(\ell)}$ and $G^{(\ell)}$, as described in Algorithm 5.

Note that Algorithm `HZ_Orthog` works in sweeps, where *max_sw* is the maximal number of allowed sweeps. The step index ℓ is omitted, and the pivot submatrices are denoted by hat. After completion of the algorithm, `Finalize` computes all the matrices in the GSVD of the pair (F, G). The matrices U and V are stored in F and G, respectively. If *acc* is set to true in `HZ_GSVD`, the transformation matrix $Z = X^{-1}S$ is accumulated, and X is obtained by solving the linear system

$$Z^T X^T = \operatorname{diag}(S_{11}, \ldots, S_{kk}).$$

The maximal value of 50 sweeps in `HZ_GSVD` is set provisionally, because numerical testing shows that `HZ_Orthog` terminates much sooner than that. No transformations in this sweep means that all the transformations have been computed as the identity matrices.

In principle, the GSVD of a given pair (F, G) can be computed by solving the GEP for the pair $(F^T F, G^T G)$. The first step is to compute the products $A = F^T F$ and $B = G^T G$. Unfortunately, the conditions of A and B are squares of the original conditions of F and G. For example, since A is symmetric and positive definite,

$$\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} = \frac{\sigma_{\max}^2(F)}{\sigma_{\min}^2(F)} = \kappa^2(F),$$

and, similarly, for B and G. As a consequence, the computed generalized eigenvalues may suffer a significant loss of accuracy with respect to the computed solution of the original GSVD problem. In addition, the GEP algorithm uses two-sided transformations, which are not cache-aware and are hard to parallelize. The one-sided transformations do not suffer from any of these drawbacks, so the GSVD algorithm turns out to be much faster and more stable.

From the practical point of view, it is better to solve the GEP by reducing it to the GSVD, than the other way around. The original two-sided Hari–Zimmermann algorithm for the GEP can be replaced by a two-step algorithm. When A is positive definite, the first step is the Cholesky factorization (preferably, with diagonal pivoting [69]) of A,

$$A = P^T F^T F P, \quad \text{i.e.,} \quad PAP^T = F^T F,$$

followed by the Cholesky factorization of PBP^T (with P as above), i.e.,

$$PBP^T = G^T G.$$

Algorithm 5: Implicit cyclic Hari–Zimmermann algorithm for the GSVD.

```
HZ_Orthog($inout::F,\ G,\ Z,\ \ in::k,\ acc,\ max\_sw$);
begin
    if $acc$ then $Z = I$;
    $it = 0$;
    repeat  // sweep loop
        $it = it + 1$;
        for all pairs $(i, j)$, $1 \le i < j \le k$ do
            compute $\widehat{A}$ and $\widehat{B}$ from (3.14);
            compute the elements of $\widehat{Z}$ by using (3.8);
            // transform $F$ and $G$
            $[f_i, f_j] = [f_i, f_j] \cdot \widehat{Z}$;
            $[g_i, g_j] = [g_i, g_j] \cdot \widehat{Z}$;
            // if needed, accumulate $Z$
            if $acc$ then $[z_i, z_j] = [z_i, z_j] \cdot \widehat{Z}$;
        end for
    until (no transf. in this sweep) or ($it \ge max\_sw$);
end

Finalize($inout::F,\ G,\ \ in::Z,\ k,\ acc,\ \ out::\Sigma_F,\ \Sigma_G,\ X$);
begin
    for $i = 1$ to $k$ do
        $S_{ii} = \sqrt{\|f_i\|_2^2 + \|g_i\|_2^2}$;
        $(\Sigma_F)_{ii} = \|f_i\|_2 / S_{ii}$;
        $(\Sigma_G)_{ii} = \|g_i\|_2 / S_{ii}$;
        $f_i = f_i / \|f_i\|_2$;
        $g_i = g_i / \|g_i\|_2$;
    end for
    if $acc$ then
        solve the linear system $Z^T X^T = S$ to obtain $X$;
    end if
end

HZ_GSVD($inout::F,\ G,\ \ in::k,\ \ out::Z,\ \Sigma_F,\ \Sigma_G,\ X$);
begin
    set $acc$;    $max\_sw = 50$;
    call HZ_Orthog($F,\ G,\ Z,\ k,\ acc,\ max\_sw$);
    call Finalize($F,\ G,\ Z,\ k,\ acc,\ \Sigma_F,\ \Sigma_G,\ X$);
end
```

If no pivoting is employed (i.e., if $P = I$), then the Cholesky factorizations of both matrices can be performed concurrently.

In either case, in the second step the implicit Hari–Zimmermann method is applied to the two triangular matrices, F and G, resulting from the previous step.

Since the iterative part of the algorithm is one-sided, only the transformations of columns are performed, which are considerably faster (when the data is stored and accessed in the column-major array order, as in Fortran) than the two-sided transformations in the original algorithm.

When A is symmetric and indefinite, and B is symmetric positive definite, an algorithm similar to Algorithm 5 can also be used as the second step in solving the GEP for the pair (A, B).

In such an algorithm, the first step should be a slightly modified symmetric indefinite factorization [70] of A (preferably, with complete pivoting [14]), that produces the signs of the eigenvalues,

$$A = P^T F^T J F P, \quad \text{i.e.,} \quad P A P^T = F^T J F,$$

where P is a permutation, F is block upper triangular with diagonal blocks of order 1 or 2, and J is a diagonal matrix with ± 1 on its diagonal.

For easier computation, J should be partitioned in a way that all positive signs precede all negative ones along its diagonal. If it is not already so, an additional symmetric permutation Q is needed to establish that partitioning,

$$A = P^T F^T Q^T J Q F P, \quad \text{i.e.,} \quad P^T A P = F_Q^T J F_Q,$$

where $F_Q := QF$. In what follows, substitute F_Q for F in such a case, and assume that J is partitioned as above.

Then, G is computed from the Cholesky factorization of the matrix

$$P B P^T = G^T G.$$

However, the second step should employ a modification of Algorithm 5, as follows.

First, note that now it is the pair $(F^T J F, G^T G)$ that is being simultaneously diagonalized,

$$Z^T F^T J F Z = \Lambda_F, \quad Z^T G^T G Z = \Lambda_G,$$

where Λ_F has some negative diagonal elements. Therefore, (3.14) has to be modified to take into account the signature matrix J,

$$a_{ii}^{(\ell)} = (f_i^{(\ell)})^T J f_i^{(\ell)}, \qquad a_{ij}^{(\ell)} = (f_i^{(\ell)})^T J f_j^{(\ell)}, \qquad a_{jj}^{(\ell)} = (f_j^{(\ell)})^T J f_j^{(\ell)}.$$

Also, no postprocessing with `Finalize` in Algorithm 5 is needed for the GEP, regardless of whether A is positive definite or indefinite. In both cases it holds that

$$P A P^T Z = P B P^T Z (\Lambda_G^{-1} \Lambda_F). \tag{3.15}$$

Multiplying (3.15) by P^T from the left, and denoting the generalized eigenvalues $\Lambda_G^{-1} \Lambda_F$ by Λ, it follows that

$$A P^T Z = B P^T Z \Lambda.$$

The generalized eigenvectors of the pair (A, B) are the columns of the matrix $P^T Z$.

After establishing the connection between one-sided and two-sided Hari–Zimmermann algorithms, the global convergence of the one-sided algorithm follows directly from Theorem 3.3.

Theorem 3.4. *If G is of full column rank, the row- and the column-cyclic implicit Hari–Zimmermann method for the GSVD of the pair (F, G) is globally convergent.*

Proof. Let $A = F^T F$ and $B = G^T G$. Since G is of full column rank, the matrix B is positive definite. According to Theorem 3.3, the two-sided Hari–Zimmermann algorithm for the GEP of the pair (A, B) is globally convergent. There exists a permutation π of the generalized eigenvalues $\lambda_1 \leq \cdots \leq \lambda_k$ of the pair (A, B), i.e., a permutation of the generalized singular values $\sigma_i = \sqrt{\lambda_i}$, $i = 1, \ldots, k$, of the pair (F, G), such that

$$\lim_{\ell \to \infty} A^{(\ell)} = \operatorname{diag}(\lambda_{\pi(k)}, \ldots, \lambda_{\pi(1)}) = \operatorname{diag}(\sigma^2_{\pi(k)}, \ldots, \sigma^2_{\pi(1)}),$$
$$\lim_{\ell \to \infty} B^{(\ell)} = \operatorname{diag}(1, \ldots, 1) = I.$$

The relationship (3.14) between the elements of A and B, and the columns of F and G, shows that in the limit, when $\ell \to \infty$, the matrices $F^{(\ell)}$ tend to a matrix with orthogonal (but not orthonormal) columns, while the matrices $G^{(\ell)}$ tend to an orthogonal matrix. Finally, the postprocessing by the one-sided scaling S^{-1} in (3.12)–(3.13) has no influence on the convergence. □

3.2.2. Blocking in the one-sided algorithm

The main motivation for blocking of the algorithm is an efficient utilization of the cache memory. To this end, the columns of F and G are grouped into nb block-columns, denoted by F_i and G_i, respectively, with an almost equal number of columns $k_i \approx k/nb$ in each block

$$F = [F_1, F_2, \ldots, F_{nb}], \quad G = [G_1, G_2, \ldots, G_{nb}]. \tag{3.16}$$

A blocked algorithm works with the block-columns in a similar way as the non-blocked algorithm works with the individual columns. The notation remains the same, but now it is used at the level of blocks, instead of columns.

A chosen block-pivot strategy in block-step ℓ selects a pair of block-columns (i, j), where $1 \leq i < j \leq nb$, as the pivot pair. That generates the following pair of pivot submatrices

$$(\widehat{A}^{(\ell)}_{ij}, \widehat{B}^{(\ell)}_{ij}) := \big((F^{(\ell)}_{ij})^T F^{(\ell)}_{ij}, (G^{(\ell)}_{ij})^T G^{(\ell)}_{ij}\big),$$

where

$$F^{(\ell)}_{ij} = [F^{(\ell)}_i, F^{(\ell)}_j], \quad G^{(\ell)}_{ij} = [G^{(\ell)}_i, G^{(\ell)}_j], \tag{3.17}$$

and

$$\begin{aligned}
\widehat{A}^{(\ell)}_{ij} &:= (F^{(\ell)}_{ij})^T F^{(\ell)}_{ij} = \begin{bmatrix} (F^{(\ell)}_i)^T F^{(\ell)}_i & (F^{(\ell)}_i)^T F^{(\ell)}_j \\ (F^{(\ell)}_j)^T F^{(\ell)}_i & (F^{(\ell)}_j)^T F^{(\ell)}_j \end{bmatrix}, \\
\widehat{B}^{(\ell)}_{ij} &:= (G^{(\ell)}_{ij})^T G^{(\ell)}_{ij} = \begin{bmatrix} (G^{(\ell)}_i)^T G^{(\ell)}_i & (G^{(\ell)}_i)^T G^{(\ell)}_j \\ (G^{(\ell)}_j)^T G^{(\ell)}_i & (G^{(\ell)}_j)^T G^{(\ell)}_j \end{bmatrix}.
\end{aligned} \tag{3.18}$$

The order of pivot matrices in (3.18), i.e., the number of columns in the so-called pivot blocks $F_{ij}^{(\ell)}$ and $G_{ij}^{(\ell)}$ in (3.17), is $k_i + k_j$. The blocked algorithm then performs a one-sided transformation of both pivot blocks to obtain a new pair of blocks

$$F_{ij}^{(\ell+1)} = F_{ij}^{(\ell)} \widehat{Z}_{ij}^{(\ell)}, \quad G_{ij}^{(\ell+1)} = G_{ij}^{(\ell)} \widehat{Z}_{ij}^{(\ell)}, \tag{3.19}$$

where $\widehat{Z}_{ij}^{(\ell)}$ is a nonsingular matrix of order $k_i + k_j$. This transformation matrix is a blocked analogue of the pointwise Hari–Zimmermann transformation $\widehat{Z}_\ell$ from (3.7). If these transformations have to be accumulated as in (3.2), the global transformation matrix $Z^{(\ell)}$ of order k is partitioned in the same way as the columns of F and G, and only the block-columns $Z_i^{(\ell)}$ and $Z_j^{(\ell)}$ need to be updated. By putting

$$Z_{ij}^{(\ell)} = [Z_i^{(\ell)}, Z_j^{(\ell)}],$$

the block-column update can be written as

$$Z_{ij}^{(\ell+1)} = Z_{ij}^{(\ell)} \cdot \widehat{Z}_{ij}^{(\ell)}. \tag{3.20}$$

The matrix $\widehat{Z}_{ij}^{(\ell)}$ in (3.19) is taken as a product, or a sequence of a certain number of pointwise transformations, all of which have the same form as in (3.7). Each pointwise transformation orthogonalizes a pair of pivot columns, and the whole sequence of such transformations in a blocked algorithm is chosen to accomplish a similar task on the block-level, i.e., to make all the columns in both pivot blocks more orthogonal than before. Since $k_i + k_j$ is, generally, greater than 2, the non-blocked (pointwise) algorithm can now be mimicked in two different ways—both pivot blocks can be "fully" orthogonalized, or just a single sweep of pointwise transformations can be applied.

1. If the block-sizes are chosen to be sufficiently small, most of the required data resides in cache, and the pivot pair $(\widehat{A}_{ij}^{(\ell)}, \widehat{B}_{ij}^{(\ell)})$ can be diagonalized efficiently by the non-blocked two-sided Hari–Zimmermann algorithm. In the one-sided terminology, the columns of pivot blocks $F_{ij}^{(\ell)}$ and $G_{ij}^{(\ell)}$ are orthogonalized. This approach is called the *full block* algorithm.
2. The offdiagonal elements of the pair $(\widehat{A}_{ij}^{(\ell)}, \widehat{B}_{ij}^{(\ell)})$ can be annihilated exactly once, i.e., only one sweep of the transformations is applied to the right-hand side of $F_{ij}^{(\ell)}$ and $G_{ij}^{(\ell)}$. If the block-sizes are chosen to be moderate, this, the so-called *block-oriented* algorithm, is a powerful alternative to the full block algorithm.

The one-sided transformations of block-columns can be applied from the right-hand side on $(F_{ij}^{(\ell)}, G_{ij}^{(\ell)})$ in (3.17), or from the left-hand side on $\big((F_{ij}^{(\ell)})^T, (G_{ij}^{(\ell)})^T\big)$. The choice of the side depends on the programming language. For example, Fortran stores matrices by columns, and the right-handed transformations are suitable, while C keeps matrices by rows, and the left-handed transformations are more appropriate.

Both blocked versions of the Hari–Zimmermann algorithm are recursive in nature, as they use the pointwise algorithm at the bottom (or the inner) level, to compute the transformations at the block-level. It is a well-known fact that the inner one-sided Jacobi-type algorithms are most efficient when applied to square matrices, because the columns that appear in inner products (3.14) and column updates (Algorithm 5) are as short as possible. Because the block-columns $F_{ij}^{(\ell)}$ and $G_{ij}^{(\ell)}$ are "tall-and-skinny" matrices, suitable square matrices $R_F^{(\ell)}$ and $R_G^{(\ell)}$ (pivot indices ij are omitted to keep the notation readable) can be computed either

1. from (3.17), by the QR factorization of $F_{ij}^{(\ell)}$ and $G_{ij}^{(\ell)}$, or
2. from (3.18), by the Cholesky factorization of $\widehat{A}_{ij}^{(\ell)}$ and $\widehat{B}_{ij}^{(\ell)}$.

Since the original $F_{ij}^{(\ell)}$ and $G_{ij}^{(\ell)}$ are needed in the algorithm for the block-column updates (3.19), they should be copied and saved before the QR factorization, because the QR factorization of a given matrix is performed in-place, i.e., it destroys the original matrix. In the second alternative, this extra copying of tall blocks is not needed. An additional storage is required only for the small square pivot submatrices to store the matrix multiply results, and the Cholesky factorization is performed in-place on these results. Besides, the "multiply + in-place Cholesky" approach turns out to be much faster in practice, so it is used in all blocked algorithms to compute $R_F^{(\ell)}$ and $R_G^{(\ell)}$.

Regardless of the choice, the use of pivoting is desirable in both approaches, to improve the numerical stability and the speed of convergence of the blocked algorithm. Since the pivoting must be the same for both matrices, a direct application of the LAPACK routines for the pivoted QR or the pivoted Cholesky factorization of each matrix, as in [69], is not possible. Instead of using pivoting in the factorization algorithms, which becomes quite complicated when multiple levels of blocking are present, the pivoting is incorporated directly into the (pointwise) orthogonalization at the bottom level of the algorithm, i.e., into the inner loop of `HZ_Orthog` from Algorithm 5.

One way to do this is similarly as de Rijk [16] incorporated pivoting in the ordinary Jacobi SVD computation. In that algorithm, each sweep orthogonalizes k columns of a given matrix R in a row-cyclic manner. To ensure that all pairs of the original columns are indeed transformed, a sweep is implemented like the selection sort algorithm over the columns of the current working matrix (initially R), in the decreasing order of their norms (see Algorithm 6, where the sweep with pivoting is implemented in-place—on the working matrix denoted by R). The sweep is divided into $k-1$ subsweeps. In each subsweep, the first pivot column is the one with the largest norm among the remaining columns, and then all the remaining pairs are transformed, which changes the norms of the transformed columns. The number of necessary column interchanges in each sweep is at most $k-1$.

Explicit interchanges of columns can cause a significant slowdown of the inner orthogonalization algorithm, so they should be avoided, if possible. Such a trick is used by Novaković [55] for the Jacobi SVD on graphics processing units (see Algorithm 7, again written in-place, on the working matrix R). Since this strategy is strictly local, i.e., it compares only a pair of columns to be transformed, it is suitable for parallel algorithms—any prescribed ordering of pairs (i,j) in a sweep can be used, not just a cyclic one.

More importantly, in the local pivoting strategy there are no column interchanges

Algorithm 6: Pivoting strategies for the SVD: sequential de Rijk [16]

```
for $i = 1$ to $k - 1$ do
    find a column $r_{\max}$ such that $\|r_{\max}\|_2 = \max_{j=i,\ldots,k} \|r_j\|_2$;
    interchange the columns $r_i$ and $r_{\max}$;
    for $j = i + 1$ to $k$ do
        orthogonalize the columns $r_i$ and $r_j$;
    end for
end for
```

Algorithm 7: Pivoting strategies for the SVD: local Novaković [55]

```
for each pair $(i, j)$ in a sweep do
    compute a $2 \times 2$ rotation that orthogonalizes $r_i$, $r_j$;
    compute $\|r_i'\|_2$ and $\|r_j'\|_2$ as they would be after the transformation \
      (but without applying it);
    if $\|r_i'\|_2 < \|r_j'\|_2$ then
        swap the columns of the computed $2 \times 2$ rotation;
    end if
    apply the rotation to $r_i$ and $r_j$ to obtain $r_i'$ and $r_j'$;
end for
```

before the transformation, and the actual ordering is achieved by permuting the pointwise transformation itself, when required. In contrast to de Rijk's sequential strategy, which begins with a "sorted" pair of columns (with respect to their norms), the local reordering strategy ends with a "sorted" pair of columns after each pointwise transformation.

In the GSVD algorithm, the same pivoting has to be applied on two matrices $R_F^{(\ell)}$ and $R_G^{(\ell)}$, instead of only to one matrix R in the ordinary SVD algorithm. To describe the modified reordering strategy, that is suitable for the GSVD computation, it is sufficient to consider the state before and after one pointwise transformation. For simplicity, the pointwise transformation index is omitted in the discussion below, and the following convention is adopted: all quantities without a prime refer to the state before the transformation, while all primed quantities refer to the state after the transformation.

The GSVD pivoting is designed in such a fashion that the ratio between the norms of the pivot columns from R_F and R_G is decreasing after the transformation, i.e.,

$$\frac{\|(R_F')_i\|_2}{\|(R_G')_i\|_2} \geq \frac{\|(R_F')_j\|_2}{\|(R_G')_j\|_2}, \quad i < j.$$

Additionally, in the one-sided Hari–Zimmermann algorithm, the norms of all computed columns in R_G' are equal to 1, so it is sufficient to compare the norms of the computed pivot columns in R_F'.

To accomplish the desired ordering, the sweep phase of Algorithm 5 should be modified so that the pivot submatrices $[f_i, f_j]$, $[g_i, g_j]$, and $[z_i, z_j]$ are multiplied either by $\widehat{Z}$ or by $\widehat{Z}P$, where

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

depending on the sizes of $\|f_s'\|_2 = \|(R_F')_s\|_2$, for $s = i, j$. These two norms can be computed very quickly (with a reasonable accuracy) before the actual transformation. The elements of pivot submatrices from (3.4) are already at our disposal, as they are needed in the computation of $\tan(2\vartheta)$ in (3.8). Then it is easy to compare the scaled squares of norms

$$a_{ss}'' := \bigl(1 - b_{ij}^2\bigr) a_{ss}' = \bigl(1 - b_{ij}^2\bigr) \, \|(R_F')_s\|_2^2, \quad s = i, j,$$

where a_{ii}'' and a_{jj}'' are computed as

$$\begin{aligned} a_{ii}'' &= \cos^2\varphi \, a_{ii} - 2\cos\varphi \sin\psi \, a_{ij} + \sin^2\psi \, a_{jj}, \\ a_{jj}'' &= \sin^2\varphi \, a_{ii} + 2\sin\varphi \cos\psi \, a_{ij} + \cos^2\psi \, a_{jj}. \end{aligned}$$

In the later stages of the orthogonalization, it can be expected that the columns are already sorted in such a way, so that the permutations do not occur any more, and the algorithm will produce the generalized singular values in the decreasing order.

Subsequently, it is assumed that `HZ_Orthog` in Algorithm 5 incorporates the local reordering of pivot columns as in Algorithm 7, combined with any chosen pointwise pivoting strategy. Such a modification brings an additional speedup of up to 20%.

In any blocked version of the algorithm, the matrix $\widehat{Z}_{ij}^{(\ell)}$ that transforms the columns of $R_F^{(\ell)}$ and $R_G^{(\ell)}$ can be computed in several different ways (see [39, 40]). Numerical tests show that the explicit accumulation of all used transformations is the best option regarding the accuracy, while the solution of one of the linear systems

$$R_F^{(\ell)} \widehat{Z}_{ij}^{(\ell)} = R_F^{(\ell+1)}, \qquad R_G^{(\ell)} \widehat{Z}_{ij}^{(\ell)} = R_G^{(\ell+1)}, \tag{3.21}$$

after all transformations, is the best option regarding the speed (about 20% faster for sequential execution of blocked algorithms), with a negligible loss of accuracy (see [27] for details). The first system in (3.21) can be singular, while the second one is certainly not, since G is of full column rank.

When the multiple levels of blocking are present, the decision which option is better may vary from level to level. For instance, if the blocks are spread over a distributed memory, then local matrix multiplication in each process (executed in parallel), is much faster than the parallel solution of a distributed linear system.

Both blocked versions of the one-sided Hari–Zimmermann algorithm are given in Algorithm 8.

Algorithm `HZ_BOrthog_Lv`$_n$ transforms F and G by one of the blocked algorithms. The step index ℓ is omitted. To accommodate multiple levels of blocking, the algorithm is recursive in nature—each pivot block is transformed by the calling essentially the same algorithm `HZ_BOrthog_Lv`$_{n-1}$ at the next level of blocking. At the bottom level, `HZ_BOrthog_Lv`$_0$ is the pointwise algorithm `HZ_Orthog`.

Algorithm 8: Implicit blocked Hari–Zimmermann algorithm for the GSVD.

```
HZ_BOrthog_Lv$_n$($inout$ :: $F$, $G$, $Z$,  $in$ :: $k$, $acc_n$, $max\_sw_n$);
begin
    partition matrices $F$, $G$, and (if needed) $Z$, as in (3.16);
    set $acc_{n-1}$ to true, if $\widehat{Z}_{ij}$ will be accumulated,
            or to false, if $\widehat{Z}_{ij}$ will be obtained by solving the linear system;
    $it = 0$;
    if algorithm is full block then
        $max\_sw_{n-1} = 50$;
    else if algorithm is block-oriented then
        $max\_sw_{n-1} = 1$;
    end if
    repeat  // sweep loop
        $it = it + 1$;
        choose a block pivot strategy block_piv;
        for $\forall$ $(i, j)$ in a block-sweep, according to block_piv (parallel) do
            // block-transformation: in parallel implementation,
            // a thread or a process job
            compute the factors $R_F$ and $R_G$ of
                    the submatrices $\widehat{A}_{ij}$ and $\widehat{B}_{ij}$, respectively;
            if $acc_{n-1}$ then
                $\widehat{Z}_{ij} = I$;
            else
                save $R_F$ to $T_F$, or $R_G$ to $T_G$;
            end if
            call HZ_BOrthog_Lv$_{n-1}$($R_F$, $R_G$, $\widehat{Z}_{ij}$, $k_i + k_j$, $acc_{n-1}$,
             $max\_sw_{n-1}$);
            if not $acc_{n-1}$ then
                solve the linear system $T_F\widehat{Z}_{ij} = R_F$, or $T_G\widehat{Z}_{ij} = R_G$
            end if
            // transform the pivot blocks of $F$ and $G$
            $F_{ij} = F_{ij} \cdot \widehat{Z}_{ij}$;    $G_{ij} = G_{ij} \cdot \widehat{Z}_{ij}$;
            // if needed, accumulate $Z$
            if $acc_n$ then $Z_{ij} = Z_{ij} \cdot \widehat{Z}_{ij}$;
        end for
    until (no transf. in this sweep) or ($it \geq max\_sw_n$);
end
```

The driver routine `HZ_GSVD` is the same as in Algorithm 5, except that it now calls the topmost blocked algorithm `HZ_BOrthog_Lv`$_n$, instead of the pointwise algorithm `HZ_Orthog`.

The general blocked algorithm `HZ_BOrthog_Lv`$_n$, at a certain level n, can be implemented either sequentially or in parallel, depending on the choice of the block-pivot strategy.

The hierarchy of blocking levels in Algorithm 8 corresponds to the memory hierarchy in modern computer architectures. In that respect, level 1 (the bottom level of true blocking) is, usually, the sequential blocked algorithm, aimed at exploiting the local cache hierarchy of a single processing unit. The next two levels of blocking correspond to the shared and distributed memory parallelism, respectively. The choice of the topmost level depends on the size of the pair and the available hardware.

In a multi-level blocked algorithm, only the topmost level “sees” the block-columns of the original square matrices F and G, while the lower levels “see” only the shortened square matrices $R_F^{(\ell)}$ and $R_G^{(\ell)}$, resulting from some pivot blocks at the level above.

Note that the bottom pointwise level $\texttt{HZ\_BOrthog\_Lv}_0$ = `HZ_Orthog` can be implemented in a similar fashion as all the blocked levels. However, for pivot blocks with only two columns, it is much faster to implement the transformations directly (as in Algorithm 5), without a reduction to square matrices of order 2.

The most time-consuming part of the blocked algorithm is the update of both pivot blocks (3.19), and the update of the transformation matrix (3.20), if it is accumulated. All updates are done by the BLAS-3 routine `xGEMM`. Since the original matrices cannot be overwritten by `xGEMM`, a temporary $k \times 2k_0$ matrix is needed, where

$$k_0 = \max_{i=1,\dots,nb} k_i. \tag{3.22}$$

This is sufficient for the sequential blocked algorithms (level 1). If these two or three updates are to be performed in parallel (at higher levels), then the required temporary storage is $k \times 2k_0$ *per update*, times the maximal number of concurrent updates to be performed.

For example, on an Intel Xeon Phi 7210 (Knights Landing) machine, for the matrices of order 8000 in MCDRAM, with 32 OpenMP threads and the sequential MKL routine calls in `P-HZ-BO-50` (see section 3.4.), the `DGEMM`s take about 36.4%, while the scalar products take about 28.9%, and the `DSYRK`s (in the Gram matrix formation part of shortening phase) take about 13% of the total CPU time.

At parallel blocked levels of the algorithm, the number of blocks nb is determined by the number of cores or processes available (seesection 3.3.), which determines the value of k_0 in (3.22).

For the sequential blocked algorithms, the optimal value of k_0 very much depends on the cache hierarchy, but also on the fine implementation details of the algorithm itself. Therefore, the optimal k_0 in (3.22) can be determined only by extensive numerical testing. Generally, the performance of each algorithm is almost constant for a reasonably wide range of nearly optimal block-sizes. In the full block algorithm, this range is narrower and covers smaller values of k_0, than in the block-oriented algorithm. Moreover, the range width decreases as the matrix order k grows. For example, for moderately sized matrices of order 5000, the nearly optimal range is 16–40 for the full block, and 32–128 for the block-oriented algorithm. On the other hand, for smaller matrices of order 2000, the difference between the ranges of nearly optimal block-sizes is minimal (see Figure 3.1).

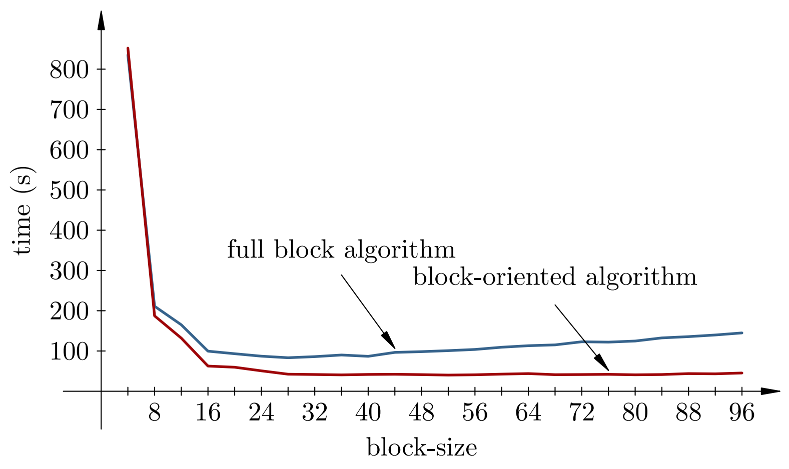


Figure 3.1: Effect of the chosen block-size k_0 for the sequential block-oriented and full block algorithms on a matrix of order 2000, with threaded MKL routines on 12 cores.

3.3. The parallel algorithms

It is well known that the one-sided Jacobi type algorithms are easy to parallelize, because independent (disjoint) pivot blocks can be transformed simultaneously. This is implemented by choosing one of the so-called parallel pivoting strategies at the appropriate level of blocking, or even at the pointwise level of the algorithm.

3.3.1. A shared memory algorithm

To maximize the efficiency on multicore computers, the blocked algorithms should be parallelized adequately. The aim here is to exploit the shared memory parallelism—available on almost any modern computing architecture. To this end, the OpenMP interface in the Fortran routines is used.

The basic principle of a parallel blocked algorithm is to divide the columns of F and G into blocks, as in (3.16), such that $nb = 2p$, where p is the number of available cores. For simplicity, it is assumed that each core runs (at most) one computational OpenMP thread. In each step, every thread transforms two block-columns of F, and two block-columns of G, with the same indices for both pairs of block-columns. These block-columns are regarded as “pivots” in each thread (as in (3.17)), and the sequential algorithm can be used to transform them.

However, if, for example, 6 threads are available, with two matrices of order 12000, then each thread would work on two 12000×2000 pivot blocks $F_{ij}^{(\ell)}$ and $G_{ij}^{(\ell)}$, and they would be shortened to square matrices $R_F^{(\ell)}$ and $R_G^{(\ell)}$ of order 2000. Depending on the cache memory available for each thread, these blocks can be too large for an efficient execution of the pointwise Hari–Zimmermann algorithm `HZ_Orthog`. Therefore, the matrices $R_F^{(\ell)}$ and $R_G^{(\ell)}$ should be subdivided into a second level of blocking and transformed by the sequential blocked algorithm `HZ_BOrthog_Lv`$_1$. Finally, the transformation matrix is applied from the right-hand side on both pivot blocks. So, the topmost parallel shared memory algorithm is actually given by `HZ_BOrthog_Lv`$_2$.

The main difference between the sequential blocked algorithms and the parallel shared memory algorithms is the choice of pivot blocks. In the parallel case, each thread (among p of them) transforms a block assigned to it. By choosing an appropriate parallel pivot strategy, that maps disjoint pivot blocks to the computational threads in each parallel step, the block-transformation in Algorithm 8 is the task performed by each thread. If $max\_sw_1 = 1$, then `HZ_BOrthog_Lv`$_2$ is the block-oriented algorithm, otherwise, for sufficiently large max_sw_1, it is the full block algorithm.

For `HZ_BOrthog_Lv`$_1$ in each thread, again it is sufficient to use a nearly optimal block-size k_0 in (3.22), because the differences in performance are not essential for any block-size in the nearly optimal range. Figure 3.1 shows that for matrices of order 12000, and 6 available threads, the computation time of the block-oriented and the full block parallel algorithm are almost constant if k_0 is chosen in the common range 24–40 for both algorithms. On machines with different cache configurations, the conclusion about the flexible range of (nearly) optimal k_0 values remains the same, even though there may be a significantly different time ratio between these two variants of the algorithm.

Up until now the discussion of pivot strategies has been omitted, i.e., the order in which cores choose the pivot pairs. The ideal parallel algorithm would simultaneously transform $nb = 2p$ pivot blocks by using some parallel block-strategy, while performing the transformation of columns inside the block by using some sequential strategy, usually, row- or column-wise.

Parallel strategies can be classified as either pointwise or blockwise. As of now, the convergence results mostly exist for the pointwise strategies, and they are based on proving their weak equivalence (see [38] for the definition) to the ordinary row- or column-cyclic strategies. Formal proofs of convergence for blockwise strategies have started to appear very recently [40], but only for the standard Jacobi algorithm that diagonalizes a single symmetric matrix.

The pointwise modulus strategy, described in [50], simultaneously annihilates the elements of A and B on each antidiagonal. This parallel strategy is known to be globally convergent.

Theorem 3.5. *If G is of full column rank, the implicit Hari–Zimmermann method for the pair (F, G), under the modulus strategy, is globally convergent.*

Proof. It suffices to prove that the modulus strategy is weakly equivalent to the row-cyclic strategy, and thus convergent. This proof can be found in [56, Appendix]. □

If the order k of matrices A and B is even, the pointwise modulus strategy is not optimal, in the sense that it annihilates the maximal possible number of elements ($k/2$) only on one half of the antidiagonals, and one element less ($k/2 - 1$) on the other half of the antidiagonals. This nonoptimality is illustrated in Figure 3.2, left, where an additional element that can be annihilated is presented in a lighter hue. The strategy that annihilates these additional elements, as well, is called the modified modulus strategy.

The same principle is used for the corresponding block-strategy that operates on blocks, with a slight modification to include the diagonal blocks (see Figure 3.2, right), thus providing a pivot pair for all cores at every parallel step. A complete description of the block-layout per core in each parallel step, and all the necessary data transfer (when needed) is given in [69, Algorithm 3.1]. The `Send_Receive` part of Algorithm 3.1 from [69] is not needed for the shared memory algorithms, but will be used in the MPI version of the algorithm.

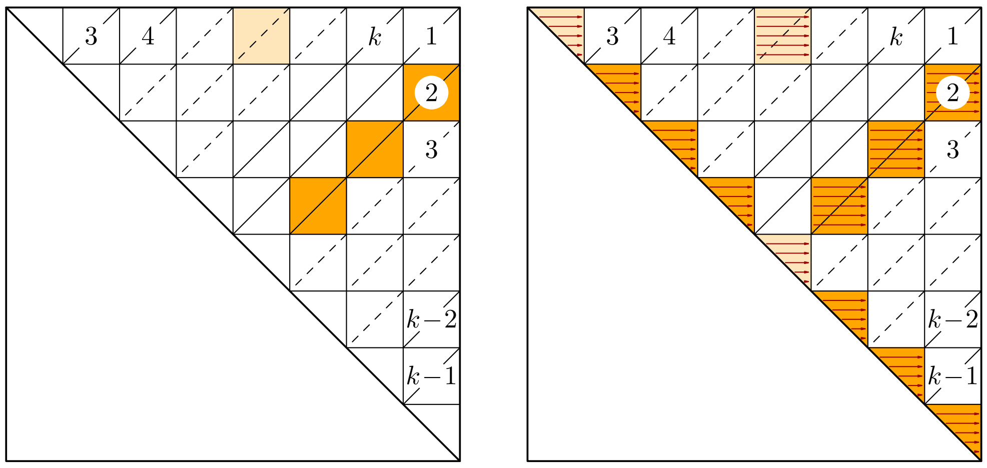


Figure 3.2: Left: modified pointwise modulus strategy on A (and B), right: modified block-modulus strategy on A (and B).

This block-strategy, combined with the row-cyclic inner strategy, is weakly equivalent to the row-cyclic block-strategy (with the same inner strategy). Therefore, it can be expected that the proof of convergence of these blocked algorithms is similar to the proofs for the ordinary block-Jacobi method [39, 40]. A mathematical technique suitable for the proofs of convergence of block-processes is developed in [38].

A halting criterion for the sequential algorithms is that the algorithm stops if no transformations have been performed in any block during the sweep, i.e., the off-diagonal elements in every pivot submatrix (3.4) are negligible, relative to the diagonal ones. This strong criterion can lead to unnecessary block-sweeps in the final phase of the parallel blocked algorithm, where only a few blocks are acted upon in the concluding block-sweeps, with only a few transformations per block. These transformations are caused by small rounding errors in factorizations and postmultiplications of the pivot blocks, but they do not alter the computed generalized singular values.

To prevent unnecessary transformations to incur extraneous sweeps, the shared memory and distributed memory parallel algorithms are terminated when a sweep (at the highest level of blocking) has had all its transformations with their cosines computed as 1. A similar heuristics is applied in subsection 2.3.6. for the multilevel blocked SVD. Note that the corresponding sines might still be different from 0 in such a case; in fact, their magnitudes may be of order $\sqrt{\varepsilon}$ at most. Then, as suggested by Vjeran Hari[1], one extra sweep might be beneficial for the accuracy of the singular vectors. It would cost about 5–10% of the execution time shown in the timing results here. In Table 3.1 the total number of pointwise transformations in block-sweeps of the full block parallel algorithm is compared to the number of transformations with their cosines different from 1.

The number of block-sweeps depends on the size of the pivot blocks $\widehat{A}_{ij}^{(\ell)}$ and $\widehat{B}_{ij}^{(\ell)}$ in (3.18), as well as the cuts between the generalized singular values induced by the block-partition (3.16). This causes a slight variation in the number of block-sweeps, when the block-size is changed.

Table 3.1: Number of transformations per sweep for the full block parallel algorithm on a matrix of order 5000 with 4 cores.

sweep number	number of all transformations	number of transformations with cosines different from 1
1	480628807	294997040
2	358157606	207835477
3	263922745	143114173
4	154679964	71401218
5	61192932	20656585
6	15891242	2317820
7	1139311	0

3.3.2. A distributed memory algorithm

The parallel shared memory algorithms presented here can also be used as the building blocks in a hybrid MPI/OpenMP algorithm for the GSVD, that would target machines with the distributed, multi-level memory hierarchy and many CPU cores per machine.

The main principle is similar to the shared memory algorithm, i.e., the columns of F, G, and Z (if needed) are divided into blocks, as in (3.16), such that $nb = 2p_m$, where p_m is the number of available MPI processes. Therefore, each MPI process, as in all blocked variants of the algorithm so far (sequential and thread-parallel ones), requires memory storage for four block-columns per matrix (F, G, and Z), and the auxiliary storage for the block-factors $R_F^{(\ell)}$, $R_G^{(\ell)}$, and the transformation matrix $\widehat{Z}_{ij}^{(\ell)}$. Two block-columns hold the input data, and also serve as a receive buffer in the communication phase of the algorithm, and the other two block-columns store the postmultiplied (updated) data, and also serve as a send buffer in the communication phase.

The only conceptual difference between `HZ_BOrthog_Lv`$_3$ (the distributed memory algorithm) and `HZ_BOrthog_Lv`$_2$ (the shared memory algorithm) is that the

[1]private communication while reviewing this thesis

updated block-columns have to be exchanged among the MPI processes after completing one step of the algorithm, according to the chosen parallel pivot strategy, while with the threaded algorithm the new pairs of block-columns are readily available in the shared memory. The exchange is achieved by a single `MPI_SENDRECV` operation per block-column on the Cartesian topology of the one-dimensional torus of processes, according to the modified modulus pivot strategy. The communication pattern is the same as in the three-level MPI-parallel Jacobi-type SVD computation, described in [68, 69], so it is omitted here for brevity.

To eliminate the need for communication among the processes residing on the same computational node, there should be only one process allocated per node—either a physical one, or a NUMA domain. Here, the latter has been chosen. Typically, in either case, an MPI process has several CPU cores available for execution, and it is assumed there is an even number p of them. In the setup, there is a single CPU with $p = 12$ cores per process available.

Note that the block-factors $R_F^{(\ell)}$ and $R_G^{(\ell)}$ may be computed concurrently, each in its own thread, under each MPI process. Such a thread may further employ the threaded BLAS interface with $p/2$ threads, if nested parallelism is available, to utilize fully all the cores available.

When the block-factors have been computed, the thread-parallel blocked algorithm `HZ_BOrthog_Lv`$_2$ with p threads is called—either the block-oriented or the full block one. In order to have a balanced execution time among the processes, the block-oriented variant is preferable. After that, the block-columns $F_{ij}^{(\ell)}$, $G_{ij}^{(\ell)}$, and $Z_{ij}^{(\ell)}$ are postmultiplied by $\widehat{Z}_{ij}^{(\ell)}$, using the threaded BLAS with p threads. Finally, a single block-column of each matrix is exchanged with the neighboring processes in the communication ring, creating an implicit synchronization point of the (otherwise concurrent) MPI processes, which concludes the actions of a single step of the distributed memory algorithm.

Table 3.2 shows the actual levels of blocking used in the implementations of the one-sided Hari–Zimmermann algorithm, as described in Algorithm 8. The table also specifies the pivoting strategy used at each level, and the corresponding value of acc_n that selects the method of computing the transformation matrix (accumulation or solution of a linear system). The choice acc_n = true at all levels is motivated by the fact that the Jacobi-type methods are known for their high accuracy of the computed results, so accuracy, rather than speed, has been targeted.

Table 3.2: Levels of blocking for the one-sided Hari–Zimmermann algorithm.

blocking level n	type of the algorithm	pivoting strategy	comm.	transformations acc_n
3	distributed (MPI)	modified modulus	yes	true
2	shared memory parallel	modified modulus	no	true
1	sequential blocked	row-cyclic	–	true
0	pointwise	row-cyclic	–	true

3.4. Numerical testing

Numerical testing has been performed on a cluster with InfiniBand interconnect, and with the nodes consisting of two Intel Xeon E5-2697 v2 CPUs, running at 2.7 GHz. Each CPU is equipped with 12 cores. The cores share 30 MB of level-3 cache. Each core has 256 kB of private instruction/data level-2 cache, and 32+32 kB of private level-1 data and instruction caches. Each CPU belongs to a separate NUMA domain with 64 GB of RAM.

The software stack consists of Intel Composer XE 2015.1.133 with the Fortran compiler under 64-bit Linux, and the BLAS and LAPACK routines from Intel MKL library, in the sequential and thread-parallel modes.

The test matrices[2] were generated in the following manner: the diagonal matrices Σ_F and Σ_G were generated with uniformly distributed values from $[10^{-5}, 10^{3}]$, giving the generalized singular values from 10^{-5} to 10^{4}. Afterwards, these diagonal matrices were multiplied by random orthogonal matrices U and V, and a nonsingular, reasonably well-conditioned matrix X, as in (1.1), resulting in relatively ill-conditioned matrices F_0 and G_0.

To alleviate possible round-off errors, all multiplications of matrices were carried out in Intel's `extended` floating-point type with 80 bits, and only the final matrices F_0 and G_0 were rounded to `double` precision.

Since LAPACK's `DTGSJA` Kogbetliantz-based routine requires triangular starting matrices, F_0 and G_0 are preprocessed by LAPACK's `DGGSVP` to obtain the starting matrices F and G for all algorithms. The time needed for this step is not measured.

First, the non-blocked and blocked sequential versions of the Hari–Zimmermann algorithm have been compared to `DTGSJA` (see Table 3.3, left). In all cases, the threaded BLAS interface with 12 computational threads was used. Since the differences between the execution times of the proposed routines and `DTGSJA` are already evident for matrices of order 5000, further comparisons with `DTGSJA` were not performed, because `DTGSJA` is too slow for larger matrices.

The threaded BLAS routines, however, do not guarantee the maximum performance on the multi-core hardware. Therefore, the parallel versions of the algorithms have been developed and tested, to see if a further speedup is possible. The right-hand side of Table 3.3 shows the execution times for the parallel (shared memory) versions of blocked routines.

To demonstrate the value of explicit parallelization, the fastest sequential routine `HZ-BO-32` with threaded MKL on 12 cores is compared to the shared memory parallel routines on $p = 2, \ldots, 12$ cores (with p even). Figures 3.3 and 3.4 show that the block-oriented parallel algorithm outperforms the fastest sequential routine when $p \geq 6$, and the full block algorithm does the same when $p \geq 10$.

It is interesting to observe that the fastest sequential routine is more than 15 times faster than the pointwise routine, both running in the same environment (with threaded BLAS on 12 cores). Likewise, the $2p$-core parallel algorithm is more than 2 times faster than the p-core algorithm. These superlinear speedups are caused by more efficient exploitation of the available memory hierarchy. The exact relationship

[2]Available for download at http://euridika.math.hr:1846/Jacobi/FLdata/

Table 3.3: The running times of LAPACK's `DTGSJA` algorithm and various sequential and parallel implementations of the Hari–Zimmermann algorithm: pointwise (`pointwise HZ`), full block with block-size 32 (`HZ-FB-32`), and block-oriented with block-size 32 (`HZ-BO-32`). The shared memory parallel versions of the algorithms running on 12 cores are denoted by prefix `P-`.

	with threaded MKL (12 cores)				with sequential MKL	
k	`DTGSJA`	`pointw. HZ`	`HZ-FB-32`	`HZ-BO-32`	`P-HZ-FB-32`	`P-HZ-BO-32`
500	16.16	3.17	4.36	2.03	1.41	0.88
1000	128.56	26.89	18.50	7.65	4.78	2.02
1500	466.11	105.31	42.38	19.31	14.57	5.99
2000	1092.39	273.48	86.01	41.60	30.02	12.13
2500	2186.39	547.84	139.53	73.07	53.13	22.34
3000	3726.76	1652.14	203.00	109.46	86.78	36.08
3500	6062.03	2480.14	294.58	186.40	129.37	55.20
4000	8976.99	3568.00	411.71	239.89	180.32	86.36
4500	12805.27	4910.09	553.67	343.58	249.92	119.74
5000	20110.39	6599.68	711.86	426.76	320.39	159.59

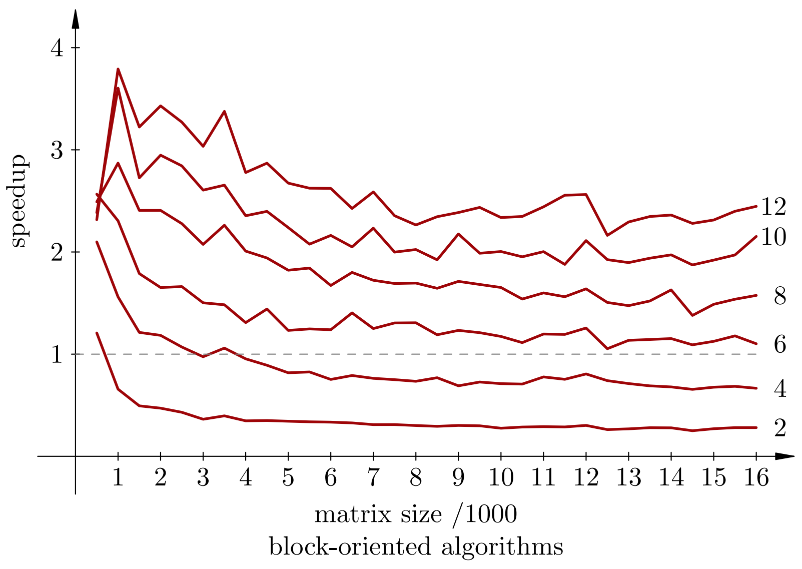


Figure 3.3: Speedup of the shared memory block-oriented algorithm on 2–12 cores vs. the sequential block-oriented Hari–Zimmermann algorithm (with MKL threaded on 12 cores).

between various versions of the algorithm is extremely machine-dependent, due to many intricacies of modern computer architectures, and thus it is very hard to predict without the actual testing.

The final speed test compares the distributed memory MPI algorithm to the

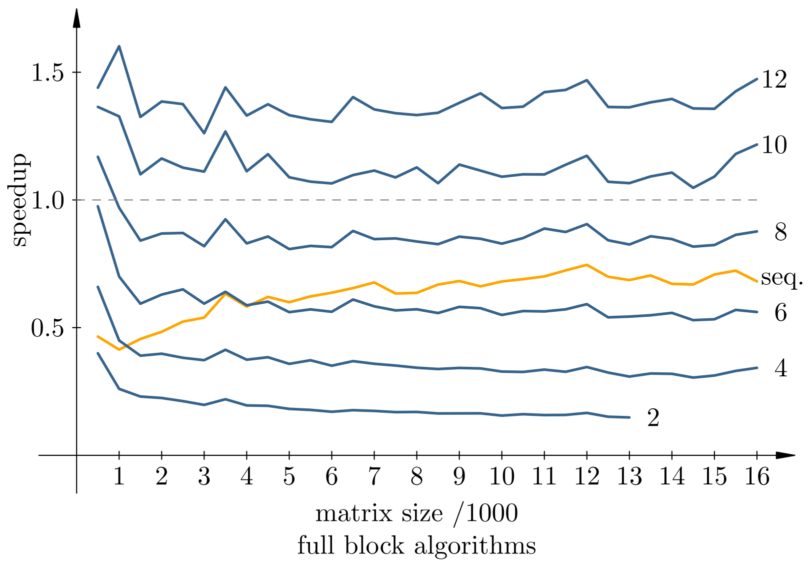


Figure 3.4: Speedup of the shared memory full block algorithm on 2–12 cores vs. the sequential block-oriented Hari–Zimmermann algorithm (with MKL threaded on 12 cores). An additional curve (denoted by seq.) shows the "speedup" of the sequential full block algorithm vs. the block-oriented algorithm. A few missing values for 2 cores are caused by the limit of execution time on the cluster (12 hours).

Table 3.4: Left table: the running times of the hybrid MPI/OpenMP version of the Hari–Zimmermann algorithm for a matrix pair of order 16000. Each MPI process executes the OpenMP algorithm with 12 threads. Each thread performs the block-oriented algorithm with the inner block-size of 32. Right table: the running times for the full block and block-oriented shared memory algorithms for the same matrix.

number of MPI processes	number of cores	time MPI-HZ-BO-32
2	24	15323.72
4	48	8229.32
6	72	6049.77
8	96	4276.65
10	120	3448.90
12	144	3003.39
14	168	2565.29
16	192	2231.71

number of cores	time P-HZ-FB-32	time P-HZ-BO-32
2	–	42906.93
4	35168.73	18096.72
6	21473.00	10936.10
8	13745.17	7651.86
10	9901.96	5599.25
12	8177.90	4925.56

shared memory algorithms. Based on the results of earlier tests, the topmost (distributed memory) level of the algorithm executes the block-oriented algorithm, with the modified modulus strategy, on a collection of MPI processes. On the second (shared memory) level, each MPI process executes the block-oriented thread-parallel algorithm, with the same strategy, on the given block-columns of F and G.

This comparison was carried out only for the largest matrix size of 16000, and the results are given in Table 3.4. Because the communication between the MPI processes is relatively slow (compared to everything else), the MPI algorithm becomes faster than the shared memory block-oriented algorithm when the number of processes p_m is at least 8. At this point the number of cores used is 96, which is huge in comparison to only 12 cores used by the shared memory algorithm. Therefore, the MPI algorithm is designed for really huge matrices, that cannot fit into the shared memory, or the parts of matrices given to a single thread are too big for efficient orthogonalization.

Finally, the accuracy of algorithms has been measured, by comparing the computed generalized singular values with the original ones, set at the beginning of the matrix generation. The test was carried out on a matrix pair of order 5000, which is the largest size that could be tested on all algorithms, including `DTGSJA`. This particular pair is moderately ill-conditioned, with a condition measure $\max \sigma_i / \min \sigma_i \approx 6.32 \cdot 10^5$. The relative errors (see Figures 3.5–3.8) are U-shaped over the ordering of values (displayed in the logarithmic scale)—the largest errors occur at both ends of the range, which is understandable, since either $(\Sigma_F)_{ii}$ or $(\Sigma_G)_{ii}$ is small there.

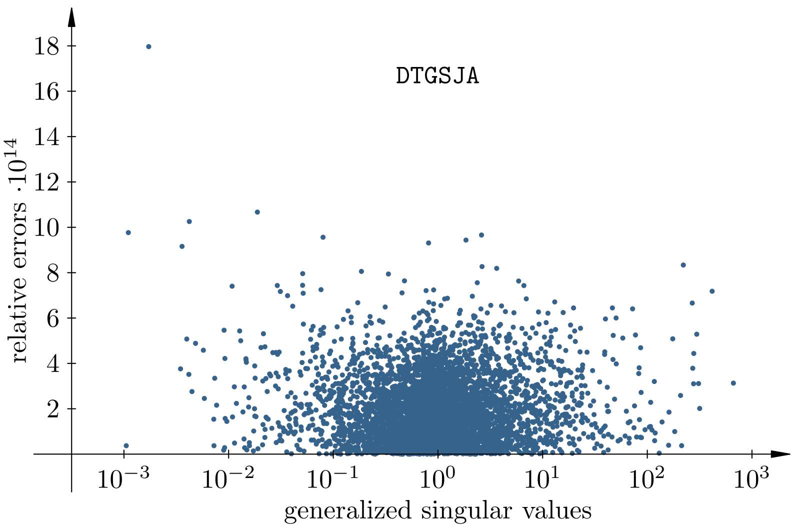


Figure 3.5: Relative errors in the computed generalized singular values for a matrix pair of order 5000 by `DTGSJA`.

Quite unexpectedly for the Jacobi-type methods, the pointwise algorithm, which is usually more accurate than any of the blocked algorithms, turns out to be slightly less accurate here. The sequential blocked algorithms are a bit more accurate than the parallel ones, as expected, while the differences in accuracy between the block-oriented and the full block versions are negligible. Maximal relative errors, and average relative errors for different routines are given in Table 3.5. Note that the average relative error for `DTGSJA` is about 5 times larger than for the blocked Hari–Zimmermann algorithms.

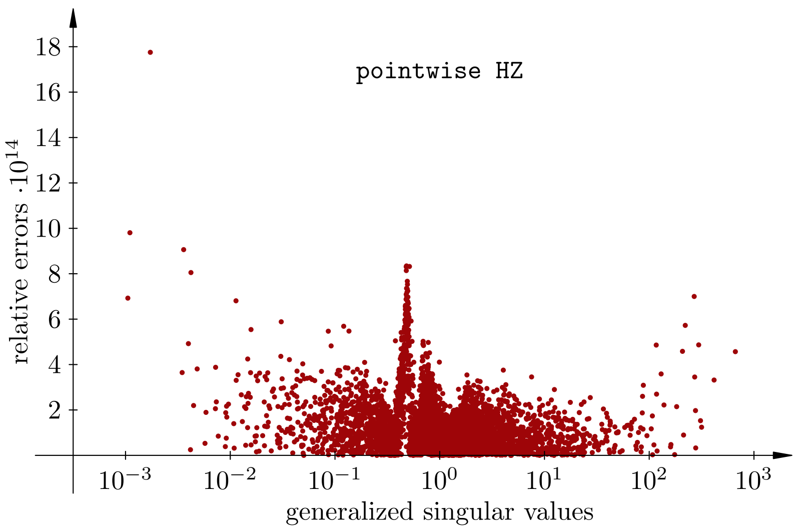


Figure 3.6: Relative errors in the computed generalized singular values for a matrix pair of order 5000 by `pointwise HZ`.

The maximal relative errors grow with the condition of the problem, and very slightly with the matrix size. For a matrix pair of order 16000 with $\max\sigma_i / \min\sigma_i \approx 4.51 \cdot 10^8$, the maximal and average relative errors are given in Table 3.6. Only the first few and the last few generalized singular values are responsible for the maximal relative error in all algorithms (the average values are much smaller). If the 5 smallest and the 5 largest generalized singular values are ignored, the ratio between the largest and the smallest σ_i drops to $2.41 \cdot 10^6$, and the obtained errors are similar to those in Table 3.5.

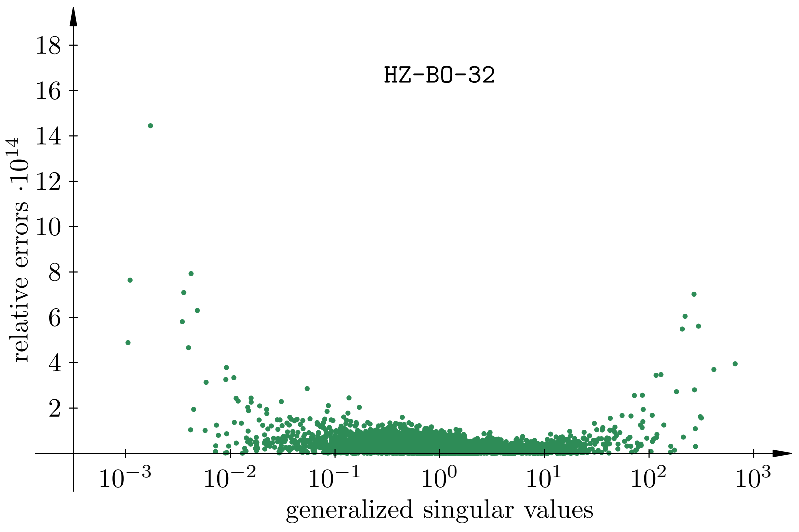


Figure 3.7: Relative errors in the computed generalized singular values for a matrix pair of order 5000 by `HZ-BO-32`.

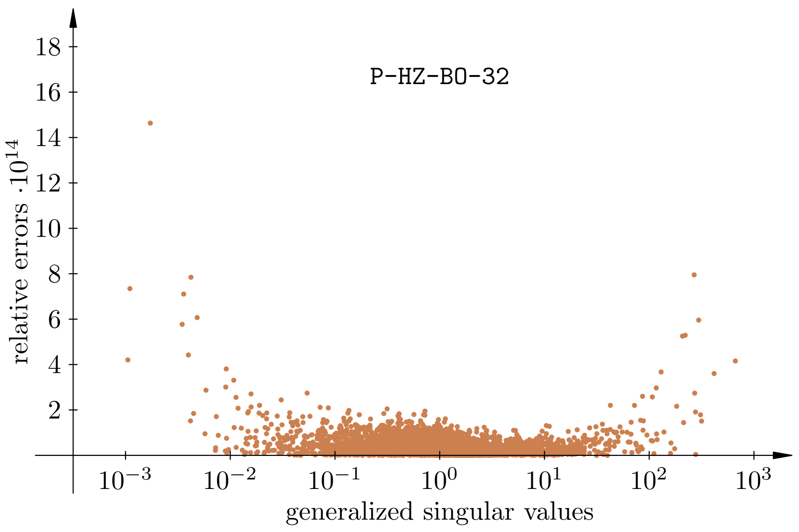


Figure 3.8: Relative errors in the computed generalized singular values for a matrix pair of order 5000 by `P-HZ-BO-32`.

Table 3.5: Maximal relative errors, and average relative errors for a matrix pair of order 5000.

	`DTGSJA`	`pointwise HZ`	`HZ-BO-32`	`P-HZ-BO-32`
max_rel	$1.79666 \cdot 10^{-13}$	$1.77529 \cdot 10^{-13}$	$1.44462 \cdot 10^{-13}$	$1.46348 \cdot 10^{-13}$
avg_rel	$1.92500 \cdot 10^{-14}$	$1.25585 \cdot 10^{-14}$	$3.50042 \cdot 10^{-15}$	$4.07475 \cdot 10^{-15}$

Note that `DTGSJA` is unable to handle such large matrices in any reasonable time. On the largest matrices that can be used for comparison, the fastest sequential threaded routine `HZ-BO-32` is more than 47 times faster than LAPACK's (on

Table 3.6: Maximal relative errors, and average relative errors for a matrix pair of order 16000. If the 5 smallest and the 5 largest generalized singular values are ignored, the respective values are denoted by (6 : 15995).

	`HZ-BO-32`	`P-HZ-BO-32`	`MPI-HZ-BO-32`
max_rel	$1.45480 \cdot 10^{-11}$	$1.45499 \cdot 10^{-11}$	$1.45576 \cdot 10^{-11}$
avg_rel	$7.91808 \cdot 10^{-15}$	$7.30987 \cdot 10^{-15}$	$7.62689 \cdot 10^{-15}$
max_rel (6 : 15995)	$2.70628 \cdot 10^{-13}$	$2.64252 \cdot 10^{-13}$	$2.65734 \cdot 10^{-13}$
avg_rel (6 : 15995)	$6.35910 \cdot 10^{-15}$	$5.75219 \cdot 10^{-15}$	$6.06843 \cdot 10^{-15}$

this particular architecture), and more accurate, as well. For the fastest explicitly parallel shared memory algorithm `P-HZ-BO-32`, the speedup factor is 126!

Bear in mind that, these speedup factors do not include the time required for triangularization of both matrices, which is mandatory for `DTGSJA`, but not necessary for the Hari–Zimmermann method, when G is of full column rank.

The tests conducted on several different architectures give similar results, only the speedup factors are somewhat different, due to differences in architecture.

3.5. An implicit Hari–Zimmermann algorithm for the GPU(s)

The blocked, parallel, implicit Hari–Zimmermann algorithm can be implemented on the GPU(s) much along the same lines as the blocked one-sided Jacobi SVD has already been detailed in chapter 2.

For brevity, only the full GSVD algorithm for a single GPU will be sketched here, and only the differences from the Jacobi SVD will be highlighted. The GSVD algorithm performs all computation on a GPU, and consists of three kernels:

1. `initZ` – acts as if it sets Z to I_k, computes $||g_i||_2$ for each $1 \le i \le k$, and scales the columns f_i of F, g_i of G, and z_i of Z, by $1/||g_i||_2$;
2. `pStep` – invoked once for each p-step in a block-sweep;
3. `Sigma` – rescales Z after each block-sweep, and optionally rescales F and G and extracts the generalized singular values at the end of computation, as per (3.10), (3.11), and (3.12).

The `pStep` kernel is similar to the one for the Jacobi SVD. The thread blocks have to operate on three 32×32 matrices in the shared memory, roughly corresponding to $\widehat{F}$, $\widehat{G}$, and $\widehat{Z}$, which makes for 24 kB in the case of $\mathbb{F} = \mathbb{R}$, or for 48 kB the case of $\mathbb{F} = \mathbb{C}$ (both in `double` precision). The strategy tables and the mapped CPU memory for the convergence statistics stay the same. Computation in a thread block proceeds in the three major phases:

1. `factorize` – shortens the pivot block-pairs $\begin{bmatrix} F_{\mathsf{p}} & F_{\mathsf{q}} \end{bmatrix}$ and $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$, according to (3.17) and (3.18), either by forming the Gram matrices and by factorizing them using the Cholesky factorizations, or by the QR factorizations of the block-columns, into the triangular factors $\widehat{F}_{\mathsf{pq}}$ and $\widehat{G}_{\mathsf{pq}}$ of order 32, initializes $\widehat{Z}_{\mathsf{pq}} = I_{32} \cdot \operatorname{diag}(1/||\hat{g}_1||_2, \ldots, 1/||\hat{g}_{32}||_2)$, and scales the columns of $\widehat{F}_{\mathsf{pq}}$ and $\widehat{G}_{\mathsf{pq}}$ by the same inverses of the column norms of $\widehat{G}_{\mathsf{pq}}$ used for $\widehat{Z}_{\mathsf{pq}}$, i.e., by $1/||\hat{g}_i||_2$, with $1 \le i \le 32$;
2. `orthogonalize` – orthogonalizes $\widehat{F}_{\mathsf{pq}}$ and $\widehat{G}_{\mathsf{pq}}$ by the pointwise implicit Hari-Zimmermann method, according to the block-oriented or the full block variant of the algorithm, accumulating the applied transformations into $\widehat{Z}_{\mathsf{pq}}$. After one inner sweep (the block-oriented variant), or as many inner sweeps as needed until the local convergence (the full block variant), $\widehat{Z}_{\mathsf{pq}}$ is rescaled according to (3.11) and (3.12), into $\widehat{Z}'_{\mathsf{pq}}$. The singular value extraction and rescaling of

$\widehat{F}_{\mathsf{pq}}$ and $\widehat{G}_{\mathsf{pq}}$ is not needed, since those matrices are discarded anyway;

3. `postmultiply` – postmultiplies $\begin{bmatrix} F_{\mathsf{p}} & F_{\mathsf{q}} \end{bmatrix}$, $\begin{bmatrix} G_{\mathsf{p}} & G_{\mathsf{q}} \end{bmatrix}$, and $\begin{bmatrix} Z_{\mathsf{p}} & Z_{\mathsf{q}} \end{bmatrix}$, by $\widehat{Z}'_{\mathsf{pq}}$, according to (3.19) and (3.20).

Figure 3.9 summarizes the `pStep` kernel from a perspective of data placement and movement. Subfigures (a) to (f) show the `factorize` phase, subfigures (g) and (h) show the `orthogonalize` phase, and subfigures (i) to (k) show the `postmultiply` phase. RAM refers to the GPU's main memory, and `*` to the matrix multiplication.

Note that on subfigure (a) the last 32×32 shared memory block is unused. That is a deficiency of the prototype code, that reuses the Gram matrix formation from Algorithm 3, i.e., for performing $A^T A$ operation on the 64×32 chunks. Another procedure should be written for 96×32 chunks, to fully utilize all the shared memory available. A similar remark relates to subfigures (i) to (k), where the `postmultiply` phase currently reuses the 32×32 Cannon-like multiplication of Algorithm 4, instead of performing $(64 \times 32) \times (32 \times 32)$ multiplication, to fully exploit the shared memory.

A transformation of a pivot pair will be computed and applied if

$$|\hat{g}_p^T \hat{g}_q| \geq \varepsilon\sqrt{32} \quad \text{or} \quad |\hat{f}_p^T \hat{f}_q| \geq ||f_p||_2 ||f_q||_2 \varepsilon\sqrt{32},$$

with ε being the machine unit roundoff for the datatype. The convergence criterion looks for the absolute values of the scaled cosines, i.e., the absolute values on the diagonal of a 2×2 transformation matrix $\widehat{Z}$, to differ from one (at least one of those elements), in order to mark a transformation as the "proper" one. Apart from that, handling the convergence information is exactly the same as in the SVD algorithm.

A recent GPU card (Pascal or newer) was not available at the time of writing the code to perform a comparative numerical test against the CPU algorithm. With a consumer Kepler GPU (GeForce GT 750M)[3], CUDA 9.0 RC[4], $\mathfrak{R}^{\|}$ p-strategy, and a 1024×1024 matrix pair, the block-oriented variant takes 26.176350 seconds for 14 block-sweeps, and the full block variant takes 30.792716 seconds for 12 block-sweeps. The block-oriented variant seems to be the faster one on the GPUs, as well as on the CPUs (cf. Table 3.3), but further testing on newer GPUs is certainly warranted.

A multi-GPU implicit Hari-Zimmermann GSVD algorithm should be a straightforward generalization of the one-sided multi-GPU Jacobi SVD and the single-GPU implicit Hari-Zimmermann GSVD algorithms. It has not been implemented yet, but it makes a reasonable and attainable target for the short-term future work.

A note on the floating-point issues. Computing the dot-products and the vector (i.e., column) norms here suffers from the same issues as in the one-sided Jacobi SVD algorithm, and the same remedies apply. Also, to compute the scaling S from (3.11), a quantity $\sqrt{||f_i||_2^2 + ||g_i||_2^2}$, and its reciprocal one, are needed. If the squares of the column norms, or their sum, may overflow, it is advisable to use the `hypot` operation, which has to protect from the overflow, albeit possibly introducing the rounding error of a couple of `ulps`. But if no overflow is expected,

[3]With such GPUs, in double precision, at most 1/24 of single precision performace is achievable.

[4]Release Candidate, not publicly available, so any results may change with the future versions.

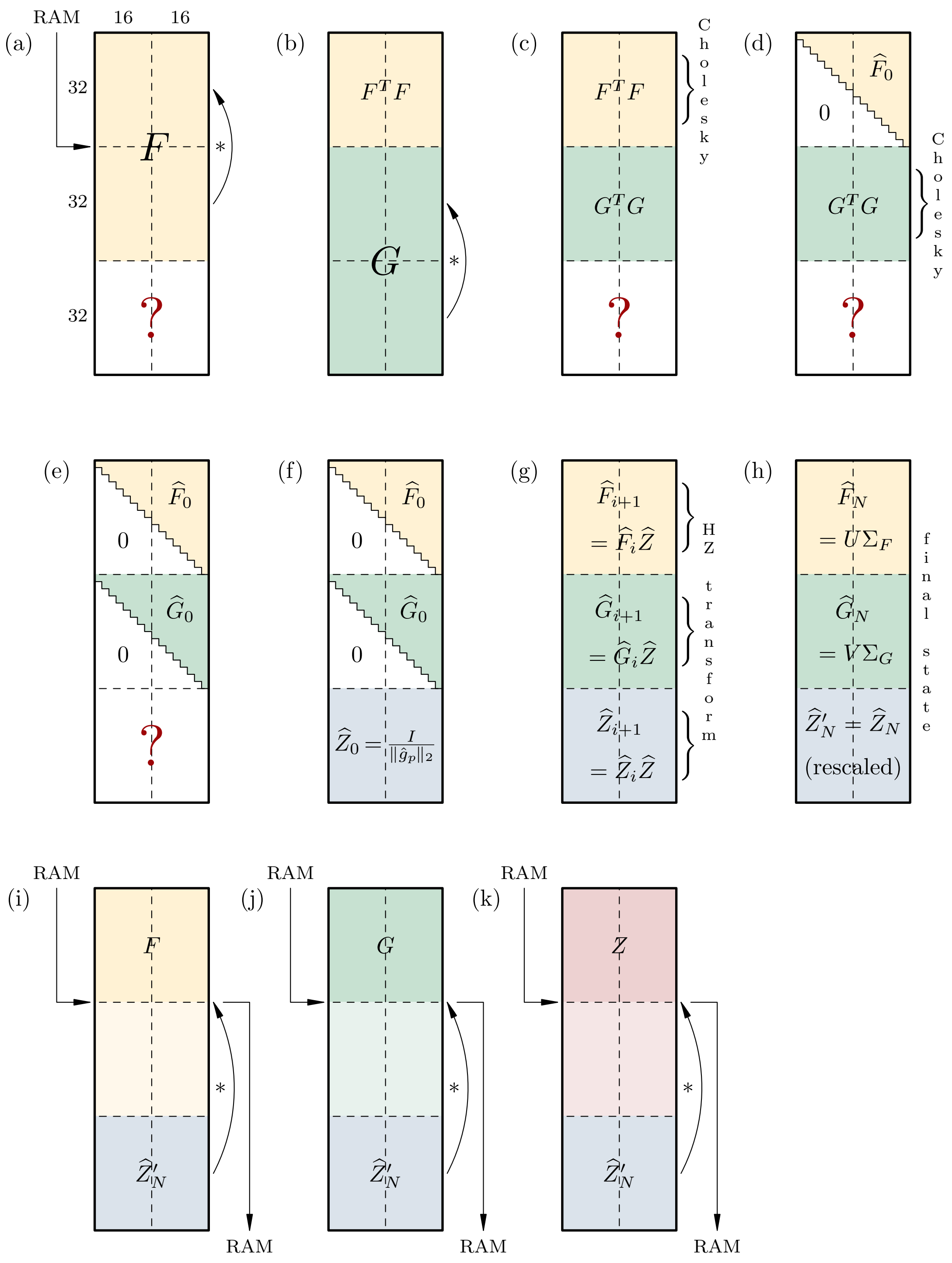


Figure 3.9: A single-GPU implicit Hari–Zimmermann algorithm, `pStep` kernel.

and the squares of the norms are already available, it may be faster to sum them and compute the square root or the reciprocal square root (`rsqrt`) directly.

Generally speaking, for both the SVD and the GSVD algorithms, a balance between the guaranteed correctness in all cases (i.e., even with a badly scaled input) and the decent performace can be struck as follows: run the faster version of the algorithm (the one that uses the Cholesky factorizations, does not protect from the possible overflows induced by the dot-product based computation of the vector norms, etc.) and check from time to time if any overflow or an invalid operation is signalled. If so, restart the whole computation with the original input (preserved in a backup workspace), but with a more careful variant of the algorithm (the one that uses the QR factorization, protects from the avoidable overflows and underflows, etc.). For this approach to be facilitated, the floating-point operations have to set the corresponding exception flags in the status register. Unlike the CPUs, that is not possible on the GPUs for now, but a check for a non-finite floating-point quantity during the computation should not be too expensive.

4. Conclusions and future work

4.1. Conclusions

In this thesis a set of new parallel Jacobi strategies has been developed, both faster and more accurate than the widely used ones. The new strategies may be seen as the generalizations of the Mantharam–Eberlein block-recursive strategy [51] to all even matrix orders. These new strategies are combined with the multi-level blocking and parallelization techniques explored in [39, 40, 69, 68, 56], to deliver the Jacobi-type (H)SVD algorithms for the graphics processing unit(s), competitive with the leading hybrid (CPU + GPU) alternatives, like MAGMA. The new algorithms are carefully designed to use a CPU primarily as a controlling unit. To this end, a collection of the auxiliary shared-memory routines for the concurrent formation of the Gram matrices, the Cholesky and QR factorizations, and the numerically robust vector 2-norm computations are proposed. The numerical results confirm that in the massively parallel GPU case the Jacobi-type methods retain all the known advantages [28, 29], while exhibiting noteworthy performance.

In the following part of the thesis the implicit Hari–Zimmermann method has been developed, for computation of the generalized singular values of matrix pairs, where one of the matrices is of full column rank. The method is backward stable, and, if the matrices permit, computes the generalized singular values with small relative errors. Moreover, it is easy to parallelize. A part of the work is motivated by the fact that the current alternatives for the GSVD computation—the best known is `DGGSVD` from LAPACK, are extremely slow for larger matrices.

Unlike the triangular routine `DTGSJA` from LAPACK, which is Kogbetliantz-based, the Hari–Zimmermann method needs no preprocessing to make both matrices triangular. Even when the matrices are preprocessed to a triangular form, the sequential pointwise Hari–Zimmermann method is, for matrices of a moderate size, significantly faster than `DTGSJA`. On a particular hardware, where threaded MKL (on 12 cores) is used, the pointwise Hari–Zimmermann method is about 3 times faster than `DTGSJA`, and the computed generalized singular values have, on average, smaller relative errors.

A significant speedup is obtained by blocking of the algorithm, to exploit the efficiency of BLAS-3 operations. For example, the sequential block-oriented version of the algorithm, in the same hardware/software environment, is 15 times faster and even more accurate than the pointwise version, for matrices of order 5000.

Further gains in speed are obtained by explicit shared memory parallelization of blocked algorithms, now with the sequential MKL. The shared memory block-oriented algorithm gives an additional speedup factor of about 2.5, even for much larger matrices. A distributed memory MPI version of the algorithm has also been developed, which has been designed for really huge matrices that do not fit into the available shared memory, as well as a GPU version, as an extension of the GPU SVD Jacobi-type algorithm from the first part of the thesis.

The thesis contains several theoretical results about the implicit Hari–Zimmermann method. The convergence of the pointwise algorithm with the row-cyclic, column-cyclic, and modulus strategies follows from the results in [37] for the two-sided generalized eigenproblem algorithm. An extension of these results for blocked algorithms is an open problem. The proofs of convergence for various blocked versions of the algorithm are under development, by using the techniques from [38].

4.2. A work in progress

In this section the ongoing and possible future work is briefly described, with an emphasis on the benefits of, and programming for, the vector extensions to the CPU architectures, in the context of the Jacobi-type SVD processes.

4.2.1. A new hierarchy for the Jacobi-type processes

As the hardware architectures become ever more complex, it seems necessary to revisit, from time to time, the assumptions about the appropriate number of the blocking levels for the Jacobi-type algorithms.

Driven by the many levels of caches present in a typical modern CPU, alongside SIMD (vectorization) capabilities, and the (still shared but) non-uniform memory architectures (NUMA), where also a distinction between a cache level and the RAM is blurred by the advent of the large memory buffers (like MCDRAM for the Intel KNLs), it may be required for an optimal design of a Jacobi-type algorithm to consider three to four memory hierarchy levels, as follows.

Level 0

Let V be the largest vector length for a given datatype (e.g., for `double`, it is 8 for Intel KNL, and 2 for IBM POWER8), and $1 \le \mathsf{v} \le \mathsf{V}$ a chosen vector length. Let C_1^d be the level-1 data (L1d) cache line size, counted in number of elements of the datatype that fit into a cache line. For `double`, $\mathsf{C}_1^d = 8$ on KNL, and $\mathsf{C}_1^d = 16$ on POWER8. Then, $\mathsf{C}_1^d/\mathsf{v}$ is a number of vectors that fit into a cache line. Let $\mathsf{V}_c = \mathsf{C}_1^d/\mathsf{V}$, and assume $\mathsf{v} = \mathsf{V}$. The Level 0 algorithm should process exactly $k\mathsf{V}_c$, $k \ge 1$, pairs of columns, to keep full vectorization for computing the rotations' parameters. Being the innermost level, it should avoid as much argument checking and branching as possible. Therefore, the input matrix should have $2k\mathsf{V}_c$ columns, or be bordered (i.e., expanded by an identity matrix in the lower right corner, and by zeroes elsewhere) up to that size.

A parameter k should be chosen to maximize L1d cache utilization, bearing in mind that two "small" matrices, G and V (or three matrices: F, G, and Z, in the GSVD case), and a couple of vectors holding the temporary results, as well as the sines (or tangents) and the cosines, have to fit into L1d cache. One should be aware of the cache associativity, false sharing, and similar hidden dangers, also!

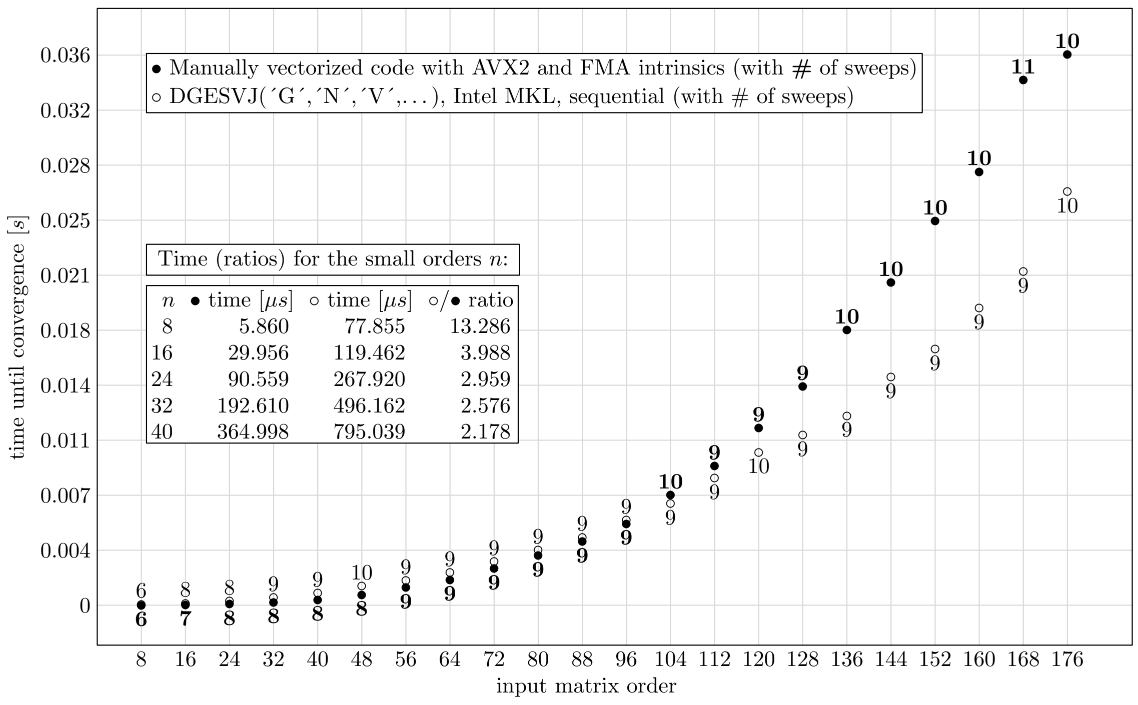
It is worth exploring if hyperthreading [52] (or a similar non-Intel technology for running more than one thread per core) with $h > 1$ threads per core, where h

divides k, improves instruction throughput versus no hyperthreading (i.e., $h = 1$). Hyperthreading should split k tasks among h threads, but the overhead of threading might be to high for such an approach to be viable.

To demonstrate that manual vectorization pays off, the following example compares the performance of a simple, hand-written (using Intel's AVX2 compiler intrinsics[1], that closely resemble the actual machine instructions generated) vectorized version of the pointwise one-sided Jacobi SVD in `double` ($\mathsf{v} = \mathsf{V} = 4$) to the performance of the MKL's optimized single-threaded `DGESVJ` routine from LAPACK.

In the hand-written code, contrary to DGESVJ, there is no final computation of the column norms (to extract the singular values) nor there is rescaling (i.e., normalizing) of the columns, since only the matrix of the accumulated rotations is sought for, in an expectation that it will be used further up the blocking hierarchy.

That final postprocessing requires only a quadratic number of operations in terms of the matrix order, and should not affect much the time ratios shown in Figure 4.1.



n	● time [μs]	○ time [μs]	○/● ratio
8	5.860	77.855	13.286
16	29.956	119.462	3.988
24	90.559	267.920	2.959
32	192.610	496.162	2.576
40	364.998	795.039	2.178

Figure 4.1: Execution times and sweep counts of the manually vectorized Jacobi code with AVX2 and FMA intrinsics, vs. `DGESVJ` from Intel MKL 2017.2.163 (sequential), on Intel Core i7-4850HQ CPU @ 2.30GHz, with macOS 10.12.4 and `clang`-802.0.38. Measured with `CLOCK_THREAD_CPUTIME_ID` POSIX clock, per run, and averaged over 100 runs; see https://github.com/venovako/JACSD/tree/master/test for more.

As can be seen on Figure 4.1, the effort of avoiding any argument checking, the function calls inside the algorithm, and non-vectorized code as much as possible, gives a speedup of several times for the very small matrix orders (where all data is

[1]See the Intel Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

expected to fit into L1d cache), although any speedup is lost for the “larger” matrix orders (104 and beyond, in this case). It is therefore advisable to pay the most attention to the innermost blocking level of the Jacobi-type processes.

So, how to perform the vectorization of the pointwise one-sided Jacobi SVD? In effect, the SIMD vector architecture can be seen as a very restricted parallel machine, where any and all parallelism comes from the scalar operations performed simultaneously on all elements (also called “lanes”) of one, two, or three vector registers, or on a subset (defined by a bitmask) of the elements thereof.

In the present architectures, the operations within the same lane across the registers involved (i.e., “vertical” instructions) are readily available, but the intra-register, inter-lane (i.e., “horizontal”) operations are either not supported, or (where they are) may be more expensive and less versatile than the vertical ones. That can be compared to the thread warps on a GPU, where the operations across threads in a warp require manual shuffling of data between threads, as opposed to the scalar operations within each warp lane.

Therefore, a similar parallelization approach as with the inner level of blocking on a GPU is required. Ignoring the technical details, there are three important phases in the pointwise Jacobi SVD: the dot-product computation, the calculation of the Jacobi rotations, and their application to the matrices G (and V).

First, the pivot pairs are chosen from the current p-step of the chosen p-strategy. There should be as many independent pivot pairs as there are vector lanes available (e.g., 4 for the `doubles` on the AVX2 architecture). If the matrix order n is equal to 8, then all pivot pairs from a p-step can be processed as it is described below. If n is smaller, the uppermost vector lanes will be unutilized during the computation of the rotations, and in the processing of the final chunk of rows during the dot-product and transformation phases. Else, if $n > 8$, then a p-step should be sliced into $\lceil n/(2\mathsf{V}_c)\rceil$ batches of pivot pairs, where each batch is processed as to be described. The batches can be iterated over sequentially, or they can be partitioned among the hyperthreads (where available and enabled) for parallel execution.

Obviously, it pays off to process matrices of orders which are the exact multiples of 8 (for AVX2, or 16 for AVX-512). Otherwise, a special code path is required for loading and storing a part of a vector from memory at the end of the loops over the matrix rows, and for masking the unused lanes elsewhere in the computation.

The pointwise algorithm vectorized. For each pivot pair (in sequential, i.e., one pivot pair after another), three dot-products are computed. Here, the vectorization is not used to handle the pivot pairs simultaneously, but to compute, e.g., 4 *scalar* `fmas` (updates of the 4 partial sums) in one go, one chunk of 4 consecutive row indices in the columns of G at a time. Once a chunk of g_p and g_q is loaded in the vector registers, three *vector* `fmas` are enough to update the partial sums of $g_p^T g_p$, $g_p^T g_q$, and $g_q^T g_q$, each in its own vector register. So, e.g., one register will hold the partial sums for $g_p^T g_q$, where the lane i will accumulate the products of the elements of g_p with the elements of g_q, with the row indices equal to $i \bmod 4$, for $0 \le i \le 3$.

The tricky part, as with the GPUs, is how to reduce the partial sums, held in each of the three aforementioned registers. Here, the “horizontal” reductions are needed, either readily available as an instruction or an intrinsic, or manually implemented.

For the AVX2 architecture, Listing 4 shows the entire computation, with one possible realization of the $+$-reduction. For each pivot pair, one vector containing the three dot products, and additionally $|g_p^T g_q|$, is obtained.

Those vectors have to be rearranged for the second phase. There, each lane will be in charge of computing the Jacobi rotation's parameters for a particular pivot pair. The idea is to get the parameters analogously to the scalar case, but for all the pivot pairs simultaneously. That effectively means to compute 4 tangents and 4 cosines for (about) the price of one each. For that, all 4 values of $g_p^T g_p$ have to be in the same vector, occupying a lane associated with their pivot pair. The same goes for $g_p^T g_q$, $g_q^T g_q$, and $|g_p^T g_q|$.

Such a rearrangement comes with a price. In the prototype implementation, it has been realized by storing the dot-product vectors back to memory, and vector-gathering their pieces into the new variables described above. Once that is done, it is decided whether *all* the columns are relatively orthogonal. If so, no transformation is needed. If not, it does not hurt at all to proceed with computing the rotations even for the lanes (i.e., the pivot pairs) where the columns are relatively orthogonal, since the vector operations will take the same (or similar) amount of time, regardless of the number of the "active" lanes. Those results can (and will) be discarded anyway.

It is however assumed that any possible floating-point exceptions occurring in the operations over those lanes will not affect the execution flow of the code. If that was not the case, a careful masking of the "offending" lanes should be considered before proceeding to the computation of the rotations' parameters, shown in Listing 5.

The rotations' parameters of the pivot pairs whose columns are nor relatively orthogonal have to be reshuffled again. For each such tangent, it has to be spread out ("broadcast") over the whole new register (i.e., in 4 copies, one in each lane), in order to be ready for vector multiplication with 4 consecutive rows of G (and V). The same holds for each such cosine.

Finally, applying the rotations is done in a conceptually similar way to computing the dot-products, one pivot pair (i.e., one rotation) a time (if the transformation is needed), but with 4 elements of G (and V) transformed at once.

Level 1

An input matrix is split into block columns, with approximately $k\mathsf{V}_c$ columns per a block column. The input should be sized in accordance to the level-2 cache size (L2), i.e., two same-size matrices, G and V, and a workspace, should fit into L2 cache. The workspace has to contain "small" blocks and vectors for the Level 0.

Hyperthreading might prove more beneficial at this level than at the level below, but then for each thread there should be its private input, output, and workspace memory for Level 0 set aside, as described. So, each thread t would need a "small" G_t, V_t, and W_t (workspace) arrays, while *all* such arrays, for *all* threads on a core, should simultaneously be present in L1d cache. Therefore, hyperthreading might push k down, close to 1, which means that k itself is a varying parameter to the Level 0 algorithm, not a constant deducible at compile-time.

Listing 4: Computing the dot products with Intel AVX2 & FMA intrinsics

```c
static inline __m256d avx2_fma_ddots(const unsigned m,
const double *const restrict Gp, const double *const restrict Gq)
{ // to be called for each pivot pair (p,q) in sequence

  // The data has to be aligned in memory to addresses that are
  // a multiple of the vector size (32 B), for faster load/store.
  register const double *const Gp_i = // start of Gp
    (const double*)__builtin_assume_aligned(Gp, 32);
  register const double *const Gq_i = // start of Gq
    (const double*)__builtin_assume_aligned(Gq, 32);

  register __m256d Gpp = _mm256_setzero_pd();
  register __m256d Gqq = _mm256_setzero_pd();
  register __m256d Gpq = _mm256_setzero_pd();

  for (register unsigned i = 0u; i < m; i += 4u) {
    // load the next chunk of Gp and the next chunk of Gq
    register const __m256d Gpi = _mm256_load_pd(Gp_i + i);
    register const __m256d Gqi = _mm256_load_pd(Gq_i + i);
    // update the partial sums
    Gpp = _mm256_fmadd_pd(Gpi, Gpi, Gpp);
    Gqq = _mm256_fmadd_pd(Gqi, Gqi, Gqq);
    Gpq = _mm256_fmadd_pd(Gpi, Gqi, Gpq);
  }

  // horizontal +-reductions, with a twist to get |Gpq| as well

  // (Gpq[2]+Gpq[3], Gpp[2]+Gpp[3], Gpq[0]+Gpq[1], Gpp[0]+Gpp[1])
  register const __m256d GppGpq = _mm256_hadd_pd(Gpp, Gpq);
  // (Gpq[2]+Gpq[3], Gqq[2]+Gqq[3], Gpq[0]+Gpq[1], Gqq[0]+Gqq[1])
  register const __m256d GqqGpq = _mm256_hadd_pd(Gqq, Gpq);
  // (Gpq[2]+Gpq[3], Gpq[0]+Gpq[1], Gpp[2]+Gpp[3], Gpp[0]+Gpp[1])
  register const __m256d GppGpq_ = // swap the middle lanes
    _mm256_permute4x64_pd(GppGpq, 0xD8);
  // (Gpq[2]+Gpq[3], Gpq[0]+Gpq[1], Gqq[2]+Gqq[3], Gqq[0]+Gqq[1])
  register const __m256d GqqGpq_ = // swap the middle lanes
    _mm256_permute4x64_pd(GqqGpq, 0xD8);
  // (Gpq, Gpq, Gqq, Gpp)
  register const __m256d intm = _mm256_hadd_pd(GppGpq_, GqqGpq_);
  // a bit of sign-masking (taking abs) needed for |Gpq|
  // x &~ -0.0 clears the sign bit, ~(-0.0): all-1s, leading 0
  register const __m256d mask = // 0.0 negated: all-1s
    _mm256_set_pd(-0.0, 0.0, -0.0, -0.0);

  // out[0] = Gpp; out[1] = Gqq; out[2] = Gpq; out[3] = |Gpq|;
  return _mm256_andnot_pd(mask, intm);
}
```

Listing 5: Computing the Jacobi rotations with Intel AVX2 & FMA intrinsics

```
__m256d ones = _mm256_set_pd( 1.0, 1.0, 1.0, 1.0);
__m256d twos = _mm256_set_pd( 2.0, 2.0, 2.0, 2.0);
__m256d neg0 = _mm256_set_pd(-0.0,-0.0,-0.0,-0.0);
// $||G_{\vec{q}}||_2^2 - ||G_{\vec{p}}||_2^2$, if the squares of the norms are available
__m256d Gq_p = _mm256_sub_pd(Gqq, Gpp);
// else, with the norms, $(||G_{\vec{q}}||_2 - ||G_{\vec{p}}||_2)(||G_{\vec{q}}||_2 + ||G_{\vec{p}}||_2)$
[[ _mm256_mul_pd(_mm256_sub_pd(Gq_,Gp_),_mm256_add_pd(Gq_,Gp_)); ]]
// $\cot 2\vec{\varphi} = (||G_{\vec{q}}||_2^2 - ||G_{\vec{p}}||_2^2)/(2 \cdot G_{\vec{p}}^T G_{\vec{q}})$; where available and if faster,
// replace multiplication by 2 with a scalbn()-like instruction
__m256d Ctg2 = _mm256_div_pd(Gq_p, _mm256_mul_pd(Gpq, twos));
// $\sqrt{\cot^2 2\vec{\varphi} + 1}$; only two roundings due to FMA
__m256d tmp0 = _mm256_sqrt_pd(_mm256_fmadd_pd(Ctg2, Ctg2, ones));
// extract the sign bit from each of the cotangents of $2\vec{\varphi}$
__m256d sgnb = _mm256_and_pd(Ctg2, neg0);
// transfer the sign bits onto the square roots; their sign bits
// being 0, it is enough to bitwise-OR them with the desired ones
__m256d tmp1 = _mm256_or_pd(tmp0, sgnb);
// $\cot \vec{\varphi} = \cot 2\vec{\varphi} + \mathrm{sign}(\sqrt{\cot^2 2\vec{\varphi} + 1}, \cot 2\vec{\varphi})$
__m256d Ctg = _mm256_add_pd(Ctg2, tmp1);
// $\tan \vec{\varphi} = 1/\cot \vec{\varphi}$; here, a ‘reciprocal’ instruction would be useful
__m256d Tan = _mm256_div_pd(ones, Ctg);
// $\sqrt{\tan^2 \vec{\varphi} + 1}$; only two roundings due to FMA
__m256d tmp2 = _mm256_sqrt_pd(_mm256_fmadd_pd(Tan, Tan, ones));
// $\cos \vec{\varphi} = 1/\sqrt{\tan^2 \vec{\varphi} + 1}$; here, there is 1 division (or reciprocal),
// and 1 square root; it would be beneficial to have CORRECTLY
// ROUNDED ‘rsqrt’, $x^{-1/2}$, instruction, to halve the number of ex-
// pensive operations to 1, and have only 2 roundings altogether.
__m256d Cos = _mm256_div_pd(ones, tmp2);
// new $\vec{p}$-th diagonal entries of $G^T G$: $||G'_{\vec{p}}||_2^2 = ||G_{\vec{p}}||_2^2 + \tan \vec{\varphi} \cdot (G_{\vec{p}}^T G_{\vec{q}})$
Gpp = _mm256_fmadd_pd(Tan, Gpq, Gpp);
// new $\vec{q}$-th diagonal entries of $G^T G$: $||G'_{\vec{q}}||_2^2 = ||G_{\vec{q}}||_2^2 - \tan \vec{\varphi} \cdot (G_{\vec{p}}^T G_{\vec{q}})$
Gqq = _mm256_fnmadd_pd(Tan, Gpq, Gqq);
```

Level 2

At this level the algorithm becomes thread-parallel (e.g., using OpenMP), if not already so due to the hyperthreading. The threads are bound to cores, one-to-one, in a single NUMA domain. Usually, that would be one CPU socket in an SMP machine and its associated memory, but it may get more complicated with the KNLs.

This is also the most appropriate level for introducing any kind of hybrid processing (e.g., CPU&GPU, CPU&MIC, CPU&FPGA, ...), since the attached devices share a NUMA domain with their neighboring CPU, but what part(s), if any, of the entire computation to offload to a coprocessing device still remains a separate research topic.

Level 3

At this level the algorithm operates in the “classical” MPI data exchange regime, and can span multiple NUMA domains (still with shared memory), or multiple SMP nodes in a cluster (with distributed memory). Each MPI process is threaded, in accordance to Level 2.

Alternatively, a single OpenMP-threaded process can be in charge for all NUMA domains (and, optionally, coprocessing devices) within a single node, by carefully setting thread affinities and allocating/using the memory local to a domain throughout execution, save for the column exchanges at the end of each block-sweep. Such a process should still employ MPI to communicate to other nodes in a cluster, though. That approach might be considered to be Level 2.5.

NUMA and the network (i.e., cluster) topologies may also be the drivers for the new classes of the parallel strategies, as it has been in the past (e.g., with the Mantharam-Eberlein strategy for the hypercube topologies, or with the modified modulus strategy for the one-dimensional torus), to minimize the wall-clock time of the data exchange phases.

4.3. A note on the figures and the software

Figures 2.2, 2.3, 2.4, 2.5, 2.6, and 2.7 have been generated by `circo` tool of Graphviz[2] 2.40.1, Figures 2.9 and 4.1 by METAPOST[3] 1.9991, as well as Figures 2.1, 2.10, 2.11, 2.12, 2.18, 2.19, and 3.9, which have been drawn by Sanja Singer from the author’s sketches. The graphs and schematics in Figures 2.20, 2.21, 2.22, 2.23, 2.24, 2.25, 2.26, 2.27, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, and 3.8 have also been plotted by her. Figure 2.8 has been generated by Wolfram Mathematica[4] 10.4.1. The matrix heatmaps of Figures 2.13, 2.14, 2.15, 2.16, and 2.17 have been created manually[5], as bitmaps, as well as the colorbar[6]. The rest of the thesis has been prepared with LaTeX. The final version will be available at http://venovako.eu site.

The prototype software implementations of the algorithms described in this thesis (and some related ones) can be found on https://github.com/venovako GitHub, under repositories GPUJACHx (a single-GPU CUDA-based one-sided Jacobi SVD, HSVD, and the Hari-Zimmermann GSVD code), hz-gsvd (an OpenMP shared-memory code for the Hari-Zimmermann GSVD), JACSD (a collection of multi-level OpenMP codes for the one-sided Jacobi SVD and a lot of support routines), and JKogb (a Kogbetliantz-like, i.e., two-sided experimental code for the HSVD).

[2]http://www.graphviz.org

[3]https://www.tug.org/metapost.html

[4]http://www.wolfram.com/mathematica/

[5]See, in https://github.com/venovako/JACSD repository, `vn` library’s tools for bitmap (BMP) generation and matrix visualization.

[6]The colormap is based on Matlab’s `jet`, with the first and the last color not shown and reserved for $-\infty$ and $+\infty$ (or `NaN`s), respectively (of which neither are present in the plotted data).

5. Bibliography

[1] O. Alter, P. O. Brown, and D. Botstein, *Generalized singular value decomposition for comparative analysis of genome-scale expression data sets of two different organisms*, P. Natl. Acad. Sci. USA, 100 (2003), pp. 3351–3356.

[2] A. A. Anda and H. Park, *Fast plane rotations with dynamic scaling*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 162–174.

[3] M. Anderson, G. Ballard, J. W. Demmel, and K. Keutzer, *Communication-avoiding QR decomposition for GPUs*, in Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011), Anchorage, AK, USA, May 2011, pp. 48–58.

[4] M. Ashworth, J. Meng, V. Novaković, and S. Siso, *Early application performance at the Hartree Centre with the OpenPOWER architecture*, in High Performance Computing, M. Taufer, B. Mohr, and J. M. Kunkel, eds., vol. 9945 of Lecture Notes in Computer Science, Frankfurt, Germany, June 2016, Springer, pp. 173–187.

[5] Z. Bai and J. W. Demmel, *Computing the generalized singular value decomposition*, SIAM J. Sci. Comput., 14 (1993), pp. 1464–1486.

[6] Z. Bai and H. Zha, *A new preprocessing algorithm for the computation of the generalized singular value decomposition*, SIAM J. Sci. Comput., 14 (1993), pp. 1007–1012.

[7] M. Bečka, G. Okša, and M. Vajteršic, *Dynamic ordering for a parallel block–Jacobi SVD algorithm*, Parallel Comput., 28 (2002), pp. 243–262.

[8] P. Benner, V. Novaković, A. Plaza, E. S. Quintana-Ortí, and A. Remón, *Fast and reliable noise estimation for hyperspectral subspace identification*, IEEE Geosci. Remote Sens. Lett., 12 (2015), pp. 1199–1203.

[9] K. Bhuyan, S. B. Singh, and P. K. Bhuyan, *Application of generalized singular value decomposition to ionospheric tomography*, Ann. Geophys., 22 (2004), pp. 3437–3444.

[10] J. L. Blue, *A portable fortran program to find the Euclidean norm of a vector*, ACM Trans. Math. Software, 4 (1978), pp. 15–23.

[11] M. Botinčan and V. Novaković, *Model-based testing of the Conference Protocol with Spec Explorer*, in 9th International Conference on Telecommunications, Zagreb, Croatia, June 2007, IEEE, pp. 131–138.

[12] R. P. Brent and F. T. Luk, *The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 69–84.

[13] R. P. Brent, F. T. Luk, and C. F. Van Loan, *Computation of the singular value decomposition using mesh–connected processors*, J. VLSI Comput. Syst., 1 (1985), pp. 242–270.

[14] J. R. Bunch and B. N. Parlett, *Direct methods for solving symmetric indefinite systems of linear equations*, SIAM J. Numer. Anal., 8 (1971), pp. 639–655.

[15] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm*, PhD thesis, Montana State University, Bozeman, MT, USA, 1969.

[16] P. P. M. de Rijk, *A one–sided Jacobi algorithm for computing the singular value decomposition on a vector computer*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 359–371.

[17] A. Deichmöller, *Über die Berechnung verallgemeinerter singularer Werte mittels Jacobi-ähnlicher Verfahren*, PhD thesis, FernUniversität–Gesamthochschule, Hagen, 1990.

[18] J. W. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou, *Communication–optimal parallel and sequential QR and LU factorizations*, Technical Report UCB/EECS–2008–89, Electrical Engineering and Computer Sciences University of California at Berkeley, Aug. 2008.

[19] ——, *Communication–optimal parallel and sequential QR and LU factorizations*, SIAM J. Sci. Comput., 34 (2012), pp. A206–A239.

[20] J. W. Demmel and H. D. Nguyen, *Fast reproducible floating-point summation*, in Proceedings of the 21st IEEE Symposium on Computer Arithmetic (ARITH), Austin, TX, USA, April 2013, pp. 163–172.

[21] J. W. Demmel and K. Veselić, *Jacobi's method is more accurate than QR*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 1204–1245.

[22] M. Doko and V. Novaković, *Izračunljivost i apstraktni strojevi*, Hrvatski matematički elektronski časopis math.e, 9 (2006).

[23] F. M. Dopico, P. Koev, and J. M. Molera, *Implicit standard Jacobi gives high relative accuracy*, Numer. Math., 113 (2009), pp. 519–553.

[24] Z. Drmač, *Computing the Singular and the Generalized Singular Values*, PhD thesis, FernUniversität–Gesamthochschule, Hagen, 1994.

[25] ——, *Implementation of Jacobi rotations for accurate singular value computation in floating point arithmetic*, SIAM J. Sci. Comput., 18 (1997), pp. 1200–1222.

[26] ——, *A tangent algorithm for computing the generalized singular value decomposition*, SIAM J. Numer. Anal., 35 (1998), pp. 1804–1832.

[27] ——, *A posteriori computation of the singular vectors in a preconditioned Jacobi SVD algorithm*, IMA J. Numer. Anal., 19 (1999), pp. 191–213.

[28] Z. Drmač and K. Veselić, *New fast and accurate Jacobi SVD algorithm. I*, SIAM J. Matrix Anal. Appl., 29 (2008), pp. 1322–1342.

[29] ——, *New fast and accurate Jacobi SVD algorithm. II*, SIAM J. Matrix Anal. Appl., 29 (2008), pp. 1343–1362.

[30] P. J. Eberlein, *A one–sided Jacobi methods for parallel computation*, SIAM J. Alg. Disc. Meth., 8 (1987), pp. 790–796.

[31] S. Falk and P. Langemeyer, *Das Jacobische Rotationsverfahren fur reelsymmetrische Matrizenpaare I*, in Elektronische Datenverarbeitung Folge 7, H. K. Schuff, ed., Friedr. Vieweg & Sohn, Braunschweig, 1960, pp. 30–34.

[32] ——, *Das Jacobische Rotationsverfahren fur reelsymmetrische Matrizenpaare II*, in Elektronische Datenverarbeitung Folge 8, H. K. Schuff, ed., Friedr. Vieweg & Sohn, Braunschweig, 1960, pp. 35–43.

[33] G. R. Gao and S. J. Thomas, *An optimal parallel Jacobi–like solution method for the singular value decomposition*, in Proceedings of the 1988 International Conference on Parallel Processing, St. Charles, IL, USA, vol. 3, August 1988, pp. 47–53.

[34] G. Gose, *Das Jacobi–Verfahren für $Ax = \lambda Bx$*, Z. Angew. Math. Mech., 59 (1979), pp. 93–101.

[35] S. Graillat, C. Lauter, P. T. P. Tang, N. Yamanaka, and S. Oishi, *Efficient calculations of faithfully rounded l_2-norms of n-vectors*, ACM Trans. Math. Software, 41 (2015), p. Article 24.

[36] E. R. Hansen, *On cyclic Jacobi methods*, J. Soc. Indust. Appl. Math., 11 (1963), pp. 448–459.

[37] V. Hari, *On Cyclic Jacobi Methods for the Positive Definite Generalized Eigenvalue Problem*, PhD thesis, FernUniversität–Gesamthochschule, Hagen, 1984.

[38] ——, *Convergence to diagonal form of block Jacobi-type methods*, Numer. Math., 129 (2015), pp. 449–481.

[39] V. Hari, S. Singer, and S. Singer, *Block-oriented J-Jacobi methods for Hermitian matrices*, Linear Algebra Appl., 433 (2010), pp. 1491–1512.

[40] ——, *Full block J-Jacobi methods for Hermitian matrices*, Linear Algebra Appl., 444 (2014), pp. 1–27.

[41] M. R. Hestenes, *Inversion of matrices by biorthonalization and related results*, J. Soc. Indust. Appl. Math., 6 (1958), pp. 51–90.

[42] N. J. Higham, *Functions of Matrices: Theory and Computation*, SIAM, Philadelphia, 2008.

[43] R. A. Horn and C. R. Johnson, *Matrix Analysis*, Cambridge University Press, Cambridge, 1985.

[44] *IEEE 754-2008, Standard for Floating-Point Arithmetic*, New York, NY, USA, Aug. 2008.

[45] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, *On generating all maximal independent sets*, Inform. Process. Lett., 27 (1988), pp. 119–123.

[46] E. G. Kogbetliantz, *Solution of linear equations by diagonalization of coefficients matrix*, Quart. Appl. Math., 13 (1955), pp. 123–132.

[47] S. R. Kuo, W. Yeih, and Y. C. Wu, *Applications of the generalized singular-value decomposition method on the eigenproblem using the incomplete boundary element formulation*, J. Sound. Vib., 235 (2000), pp. 813–845.

[48] S. Lahabar and P. J. Narayanan, *Singular value decomposition on GPU using CUDA*, in Proceedings of the 23rd IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009), Rome, Italy, May 2009.

[49] F. T. Luk and H. Park, *On parallel Jacobi orderings*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 18–26.

[50] ——, *A proof of convergence for two parallel Jacobi SVD algorithms*, IEEE Trans. Comput., C–38 (1989), pp. 806–811.

[51] M. Mantharam and P. J. Eberlein, *Block recursive algorithm to generate Jacobi–sets*, Parallel Comput., 19 (1993), pp. 481–496.

[52] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, *Hyper-threading technology architecture and microarchitecture*, Intel Technology Journal, 6 (2002), pp. 4–15.

[53] W. F. Mascarenhas, *On the convergence of the Jacobi method for arbitrary orderings*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 1197–1209.

[54] L. Nazareth, *On the convergence of the cyclic Jacobi method*, Linear Algebra Appl., 12 (1975), pp. 151–164.

[55] V. Novaković, *A hierarchically blocked Jacobi SVD algorithm for single and multiple graphics processing units*, SIAM J. Sci. Comput., 37 (2015), pp. C1–C30.

[56] V. Novaković and S. Singer, *A GPU-based hyperbolic SVD algorithm*, BIT, 51 (2011), pp. 1009–1030.

[57] V. Novaković, S. Singer, and S. Singer, *Estimates for the spectral condition number of cardinal B-spline collocation matrices*, Math. Commun., 15 (2010), pp. 503–519.

[58] ——, *Blocking and parallelization of the Hari–Zimmermann variant of the Falk–Langemeyer algorithm for the generalized SVD*, Parallel Comput., 49 (2015), pp. 136–152.

[59] NVIDIA, *CUDA C Programming Guide 5.5*, July 2013.

[60] R. Onn, A. O. Steinhardt, and A. Bojanczyk, *The hyperbolic singular value decomposition and applications*, IEEE Trans. Signal Process., 39 (1991), pp. 1575–1588.

[61] C. C. Paige, *Computing the generalized singular value decomposition*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 1126–1146.

[62] C. C. Paige and M. A. Saunders, *Towards a generalized singular value decomposition*, SIAM J. Numer. Anal., 18 (1981), pp. 398–405.

[63] B. N. Parlett, *The Symmetric Eigenvalue Problem*, no. 20 in Classics in Applied Mathematics, SIAM, Philadelphia, 1998.

[64] H. Rutishauser, *The Jacobi method for real symmetric matrices*, Numer. Math., 9 (1966), pp. 1–10.

[65] A. H. Sameh, *On Jacobi and Jacobi–like algorithms for a parallel computer*, Math. Comp., 25 (1971), pp. 579–590.

[66] G. Shroff and R. S. Schreiber, *On the convergence of the cyclic Jacobi method for parallel block orderings*, SIAM J. Matrix Anal. Appl., 10 (1989), pp. 326–346.

[67] S. Singer, S. Singer, V. Hari, K. Bokulić, D. Davidović, M. Jurešić, and A. Uščumlić, *Advances in speedup of the indefinite one-sided block Jacobi method*, in AIP Conf. Proc. – Volume 936 Numerical Analysis and Applied Mathematics, T. E. Simos, G. Psihoyios, and C. Tsitouras, eds., Melville, New York, 2007, AIP, pp. 519–522.

[68] S. Singer, S. Singer, V. Novaković, D. Davidović, K. Bokulić, and A. Uščumlić, *Three-level parallel J-Jacobi algorithms for Hermitian matrices*, Appl. Math. Comput., 218 (2012), pp. 5704–5725.

[69] S. Singer, S. Singer, V. Novaković, A. Uščumlić, and V. Dunjko, *Novel modifications of parallel Jacobi algorithms*, Numer. Alg., 59 (2012), pp. 1–27.

[70] I. Slapničar, *Componentwise analysis of direct factorization of real symmetric and Hermitian matrices*, Linear Algebra Appl., 272 (1998), pp. 227–275.

[71] I. Slapničar and V. Hari, *On the quadratic convergence of the Falk–Langemeyer method*, SIAM J. Math. Anal., 12 (1991), pp. 84–114.

[72] G. W. Stewart, *Computing the CS decomposition of a partitioned orthogonal matrix*, Numer. Math., 40 (1982), pp. 297–306.

[73] ——, *Computing the CS and the generalized singular value decompositions*, Numer. Math., 46 (1985), pp. 479–491.

[74] ——, *On the early history of the singular value decomposition*, SIAM Rev., 35 (1993), pp. 551–566.

[75] B. D. Sutton, *Stable computation of the CS decomposition: Simultaneous bidiagonalization*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 1–21.

[76] S. Tomov, R. Nath, and J. Dongarra, *Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing*, Parallel Comput., 36 (2010), pp. 645–654.

[77] C. F. Van Loan, *Generalizing the singular value decomposition*, SIAM J. Numer. Anal., 13 (1976), pp. 76–83.

[78] ——, *The block Jacobi method for computing the singular value decomposition*, in Computational and combinatorial methods in systems theory, Sel. Pap. 7th Int. Symp. Math. Theory Networks Syst., Stockholm 1985, 1986, pp. 245–255.

[79] K. Veselić, *A Jacobi eigenreduction algorithm for definite matrix pairs*, Numer. Math., 64 (1993), pp. 241–269.

[80] H. Zha, *A note on the existence of the hyperbolic singular value decomposition*, Linear Algebra Appl., 240 (1996), pp. 199–205.

[81] K. Zimmermann, *Zur Konvergenz eines Jacobiverfahren für gewönliche und verallgemeinerte Eigenwertprobleme*, dissertation no. 4305, ETH, Zürich, 1969.

IZJAVA O IZVORNOSTI RADA

Ja, VEDRAN NOVAKOVIĆ, student/ica Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu, s prebivalištem na adresi II. Maksimirsko naselje 15, HR-10000 Zagreb, OIB ______, JMBAG 1191004541, ovim putem izjavljujem pod materijalnom i kaznenom odgovornošću da je moj završni/diplomski/[✓ doktorski] rad pod naslovom: Parallel Jacobi-type algorithms for the singular and the generalized singular value decomposition (Paralelni algoritmi Jacobijeva tipa za singularnu i generaliziranu singularnu dekompoziciju), isključivo moje autorsko djelo, koje je u potpunosti samostalno napisano uz naznaku izvora drugih autora i dokumenata korištenih u radu.

U Zagrebu, 1. prosinca 2017.

Potpis

6. Biography

Vedran Novaković was born on October 10$^{\text{th}}$, 1982 in Zagreb, Croatia, where he attended XV Gymnasium and afterwards obtained a diploma from Department of Mathematics, Faculty of Science, University of Zagreb (the equivalent of Master of informatics and mathematics in present-day terms) in May 2006.

From July to December 2006 he was employed as a technical associate at Department of Mathematics, working on the "CRO-GRID Applications" project. From January 2007 to November 2007 he was a technical associate at Chair of Mathematics, Faculty of Mechanical Engineering and Naval Architecture, working on a collaborative project with industry, implementing a Java-based Open Mobile Alliance Digital Rights Management solution. From December 2007 to November 2014 he was a teaching assistant at the Faculty for the mathematical courses, while also working as a honorary teaching assistant at Department of Mathematics for the C programming labs and the parallel computing courses, as well as a researcher on the "Numerical methods in geophysical models" project of the science ministry.

From February 2015 to October 2017 he was employed as a computational scientist at Daresbury Laboratory of the Science and Technology Facilities Council, United Kingdom, for the "Square Kilometre Array" and "Numerical Linear Algebra for Exascale" (Horizon 2020) projects. He also participates in the "Matrix Factorizations and Block Diagonalization Algorithms" project of the Croatian Science Foundation.

He has coauthored 7 research papers in SCIE-indexed journals [57, 56, 69, 68, 55, 8, 58], 2 in conference proceedings [4, 11], and 1 professional paper [22].

Životopis

Vedran Novaković rođen je 10. 10. 1982. u Zagrebu, Hrvatska, gdje je pohađao XV. gimnaziju, a potom je stekao diplomu na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu (ekvivalent današnjeg magistra računarstva i matematike) u svibnju 2006.

Od srpnja do prosinca 2006. bio je zaposlen kao stručni suradnik na Matematičkom odsjeku, gdje jer radio na tehnološkom projektu “CRO-GRID Aplikacije”. Od siječnja do studenog 2007. bio je stručni suradnik na Katedri za matematiku Fakulteta strojarstva i brodogradnje, radeći na suradnom projektu s industrijom za implementaciju Open Mobile Alliance Digital Rights Management rješenja u jeziku Java. Od prosinca 2007. do studenog 2014. imao je poziciju asistenta za matematičku grupu predmeta na Fakultetu, radeći ujedno kao honorarni asistent na Matematičkom odsjeku za Programiranje (C) i kolegije o paralelnom računanju, te kao istraživač na znanstvenom projektu “Numeričke metode u geofizičkim modelima”.

Od veljače 2015. do listopada 2017. bio je zaposlen kao računalni znanstvenik u Daresbury Laboratory, Science and Technology Facilities Council, Ujedinjeno Kraljevstvo, na projektima “Square Kilometre Array” i “Numerical Linear Algebra for Exascale” (Obzor 2020). Također je vanjski suradnik na projektu “Matrične faktorizacije i blok dijagonalizacijski algoritmi” Hrvatske zaklade za znanost.

Koautor je 7 znanstvenih radova u časopisima indeksiranima u SCIE [57, 56, 69, 68, 55, 8, 58], 2 rada u zbornicima konferencija [4, 11], i jednog stručnog rada [22].