University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Davor Davidović

SOLVING LARGE DENSE SYMMETRIC EIGENPROBLEM ON HYBRID ARCHITECTURES

DOCTORAL THESIS

Zagreb, 2014

University of Zagreb
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Davor Davidović

SOLVING LARGE DENSE SYMMETRIC EIGENPROBLEM ON HYBRID ARCHITECTURES

DOCTORAL THESIS

Supervisors:
Associate Professor Domagoj Jakobović, PhD
Professor Karolj Skala, PhD

Zagreb, 2014

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Davor Davidović

RJEŠAVANJE VELIKOG PUNOG SIMETRIČNOG SVOJSTVENOG PROBLEMA NA HIBRIDNIM ARHITEKTURAMA

DOKTORSKI RAD

Mentori:
izv. prof. dr. sc. Domagoj Jakobović
prof. dr. sc. Karolj Skala

Zagreb, 2014.

The dissertation was made at the University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Electronics, Microelectronics, Computer and Intelligent systems in cooperation with Ruđer Bošković Institute, Centre for Informatics and Computing, and University Jaume I, Spain.

Supervisors: Associate Professor Domagoj Jakobović, PhD
Professor Karolj Skala, PhD

The dissertation has: 126 pages

The dissertation number: ___________

About the Supervisors

Domagoj Jakobović was born in Našice in 1973. He received B.Sc., M.Sc. and Ph.D. degrees in computer science from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER), Zagreb, Croatia, in 1996, 2001, and 2005, respectively.

From April 1997 he is working at the Department of electronics, microelectronics, computer and intelligent systems at FER. In February 2012 he was promoted to Associate Professor. He participated in 3 scientific projects and has supervised two informatics projects financed by the Ministry of Science, Education and Sports of the Republic of Croatia. Currently he is a member of the research project "MultiClod" financed by the Croatian science foundation. He published more than 40 papers in journals and conference proceedings in the area of application of stochastic optimization in scheduling and cryptography, as well as development of parallel evolutionary algorithms.

Prof. Jakobović is a member of IEEE and ACM. He is member of a journal editorial board and he serves as a technical reviewer for various international journals.

Karolj Skala was born in 1951 in Subotica. He received B.Sc., M.Sc. and Ph.D. degrees in the field of electrical engineering from University of Zagreb, Faculty of Electrical Engineering (ETF), in 1974, 1979, and 1983, respectively.

From November 1974 he is working at Ruđer Bošković Institute in the Department for laser research and development and since 1991 he is head of the Centre for informatics and computing as a consultant in permanent title from 2008. From 1996 to 2012 he is a lecturer at the Faculty of Graphic Arts in the position of full professor. Since 2013, he is a professor in permanent title at the University of Osijek. He was principal investigator in four research projects of the Ministry of Science, Education and Sports of the Republic of Croatia. He participated in framework programmes of the European Union, five EU FP6 and seven EU FP7 projects. Presently, he is a consortium partner - coordinator for three EU FP7 projects: "Application Information Services for Distributed Computing Environments (AIS DC)", "Scientific gateway Based User Support (SCI-BUS)", and "Embedded Computer Engineering Learning Platform (E2LP)". He is a national representative of COST IC1305 - "Network for Sustainable Ultrascale Computing (NESUS)". Furthermore, he was the national representative in the European Commission in the Research Infrastructure Programme Committee EC FP7 (2010-2013). Currently he represents Croatia in EC at European Research Infrastructure Consortium (ERIC). He published more than 80 scientific papers in the fields of laser communications, scientific visualization and distributed computing systems.

Prof. Skala is presidency member of MIPRO association and regular member of Croatian Academy of Engineering (HATZ). He acts as a chair of the annual conference "Distributed Computing and Visualization Systems" and participates as a reviewer in a number of international projects and scientific journals. In 2010, he received the medal of the Hungarian Academy of Sciences.

O mentorima

Domagoj Jakobović rođen je u Našicama 1973. godine. Diplomirao je, magistrirao i doktorirao u polju računarstva na Sveučilištu u Zagrebu, Fakultetu elektrotehnike i računarstva (FER), 1996., 2001. odnosno 2005. godine.

Od travnja 1997. godine radi na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave FER-a. U veljači 2012. godine izabran je u zvanje izvanrednog profesora. Sudjelovao je na tri znanstvena projekta i bio voditelj dva informatička projekta Ministarstva znanosti, obrazovanja i sporta Republike Hrvatske. Trenutno je suradnik na projektu "MultiClod" koji financira Hrvatska zaklada za znanost. Objavio je više od 40 radova u časopisima i zbornicima konferencija u području primjene stohastičke optimizacije u problemima raspoređivanja i kriptografiji te razvoja paralelnih evolucijskih algoritama.

Prof. Jakobović član je stručnih udruga IEEE i ACM. Član je uredničkog odbora znanstvenog časopisa te sudjeluje kao recenzent u desetak inozemnih časopisa.

Karolj Skala rođen je 1951. u Subotici. Diplomirao je, magistrirao i doktorirao u polju elektrotehnike na Sveučilištu u Zagrebu, Fakultetu elektrotehnike (ETF), 1974., 1979. odnosno 1983. godine.

Od studenog 1974. godine radi na Institutu Ruđer Bošković u Odjelu za laserska i atomska istraživanja i razvoj, a od 1991. je voditelj Centra za informatiku i računarstvo. Od 2008. godine zaposlen je kao znanstveni savjetnik u trajnome zvanju. Od 1996. do 2012. bio je predavač na Grafičkom fakultetu u Zagrebu u naslovnom zvanju redovitog profesora. Od 2013. godine je profesor u trajnome zvanju Sveučilišta u Osijeku. Bio je voditelj četiri znanstvena projekta Ministarstva znanosti, obrazovanja i sporta Republike Hrvatske. Sudjelovao je u okvirnim programima Europske unije, pet EU FP6 i sedam EU FP7 projekata. Trenutno je konzorcijski partner–kordinator na tri EU FP7 projekta: “Application Information Services for Distributed Computing Environments” (AIS DC), “Scientific gateway Based User Support” (SCI-BUS) i “Embedded Computer Engineering Learning Platform” (E2LP). Aktivno sudjeluje i na projektu COST IC1305 - “Network for Sustainable Ultrascale Computing” (NESUS). Bio je nacionalni predstavnik pri Europskoj komisiji u Research Infrastructure Programme Committee EC FP7 (2010-2013), a trenutno obnaša funkciju predstavnika u European Research Infrastructure Consortium (ERIC). Objavio je više od 80 znanstvenih radova iz područja laserskih komunikacija, znanstvene vizualizacije i distribuiranih računalnih sustava.

Prof. Skala je član predsjedništva udruge MIPRO i redoviti član Akademije tehničkih znanosti Hrvatske (HATZ). Voditelj je godišnje konferencije *Distributed Computing and Visualisation Systems* te sudjeluje kao recenzent na većem broju inozemnih projekata i časopisa. Dobitnik je medalje Mađarske Akademije znanosti 2010. godine.

Acknowledgements

The past six years of my life were extremely intensive, filled with a numerous travels, projects, and research activities, with this dissertation as a point to the end. During these six years I have met many interesting people that took a part in or influenced my future research directions and ideas as well as made their small contribution to my life, both personal and professional.

First, I would like to sincerely thank my supervisors, Professor Domagoj Jakobović and Professor Karolj Skala on their guidance and everyday support in overcoming the challenges I have faced during my long journey towards the genesis of this PhD dissertation. This work would not be possible without their firm leadership and encouraging advices that guided me throughout the whole PhD research. I would also like to express special thanks to Professor Enrique Quintana-Ortí from University Jaume I, Spain who hosted me in Castellón on several occasions and who was my advisor in the shadow. Without his advice, enthusiasm, and kind hospitality the majority of this research would not have been possible nor would the papers gotten published.

I would like to mention Professor Sanja Singer from the University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture, my supervisor for Masters degree. She is the one that introduced me to the world of parallel computing and advanced linear algebra and who encouraged me to undertake this PhD. Also, I would like to thank Professor Daniel Kressner who hosted me at ETH Zurich and helped me with defining my PhD research direction.

To all my colleague from Ruđer Bošković Institute who supported me, helped and encouraged me to continue my research: Jelena, Ivan S., Ivan G., Tomislav, Enis, Igor, Bojan, and finally Gita who was my loyal lunch-companion. A special thanks to Tomislav on his constructive questioning of my research and advice that helped me improve both myself as well as my research. Thanks to all guys from the University Jaume I and HPC&A group, who are still there or world-wide, José Aliaga, Sergio Iserte, and Ruyman Reyes, among others, and all the guys that kept the GPU cluster at UJI running that ultimately allowed me to run all my experiments.

Finally, I would like to extend a special gratitude to my family, especially to the three most important persons in my life, my parents and my wife Martina: to my parents who always encouraged me to do my best and to believe in myself, and to my wife Martina, the best wife one can imagine, who gave me her unlimited support to finish this dissertation and her patience during the long sunny days I spent at home writing so that this dissertation can see the light of day.

Abstract

Dense symmetric eigenproblem is one of the most significant problems in the numerical linear algebra that arises in numerous research fields such as bioinformatics, computational chemistry, and meteorology. In the past years, the problems arising in these fields become bigger than ever resulting in growing demands in both computational power as well as the storage capacities. In such problems, the eigenproblem becomes the main computational bottleneck for which solution is required an extremely high computational power. Modern computing architectures that can meet these growing demands are those that combine the power of the traditional multi-core processors and the general-purpose GPUs and are called hybrid systems. These systems exhibit very high performance when the data fits into the GPU memory; however, if the volume of the data exceeds the total GPU memory, i.e. the data is out-of-core from the GPU perspective, the performance rapidly decreases.

This dissertation is focused on the development of the algorithms that solve dense symmetric eigenproblems on the hybrid GPU-based architectures. In particular, it aims at developing the eigensolvers that exhibit very high performance even if a problem is out-of-core for the GPU. The developed out-of-core eigensolvers are evaluated and compared on real problems that arise in the simulation of molecular motions. In such problems the data, usually too large to fit into the GPU memory, are stored in the main memory and copied to the GPU memory in pieces. That approach results in the performance drop due to a slow interconnection and a high memory latency.

To overcome this problem an approach that applies blocking strategy and re-designs the existing eigensolvers, in order to decrease the volume of data transferred and the number of memory transfers, is presented. This approach designs and implements a set of the block-oriented, communication-avoiding BLAS routines that overlap the data transfers with the number of computations performed. Next, these routines are applied to speed-up the following eigensolvers: the solver based on the multi-stage reduction to a tridiagonal form, the Krylov subspace-based method, and the spectral divide-and-conquer method.

Although the out-of-core BLAS routines significantly improve the performance of these three eigensolvers, a careful re-design is required in order to tackle the solution of the large eigenproblems on the hybrid CPU-GPU systems. In the out-of-core multi-stage reduction approach, the factor that mostly influences the performance is the band size of the obtained band matrix. On the other hand, the Krylov subspace-based method, although it is based on the memory-bound BLAS-2 operations, is the fastest method if only a small subset of the eigenpairs is required. Finally, the spectral divide-and-conquer algorithm, which exhibits significantly higher arithmetic cost than the other two eigensolvers, achieves extremely high performance since it can be performed completely in terms of the compute-bound BLAS-3 operations. Furthermore, its high arithmetic cost is further reduced by exploiting the special structure of a matrix.

Finally, the results presented in the dissertation show that the three out-of-core eigensolvers, for a set of the specific macromolecular problems, significantly overcome the multi-core variants and attain high flops rate even if data do not fit into the GPU memory. This proves that it is possible to solve large eigenproblems on modest computing systems equipped with a single GPU.

Keywords: numerical linear algebra, eigenproblems, out-of-core, GPU, high-performance computing

Rješavanje velikog punog simetričnog svojstvenog problema na hibridnim arhitekturama

Problem efikasnog rješavanja svojstvenih problema jedan je od najvažnijih pravaca istraživanja iz područja numeričke linearne algebre. Svojstveni problemi javljaju se u brojnim granama istraživanja kao što su bioinformatika, računalna kemija ili meteorologija. Zbog iznimno velike složenosti algoritama za njihovo rješavanje (često kubne složenosti) rješavanje svojstvenog problema postaje jedan od računalno najzahtjevnijih zadataka koji tradicionalni višejezgreni procesori teško mogu efikasno i brzo izvršiti. Za rješavanje takvih problem danas se uobičajeno koriste velika paralelna računala s distribuiranom memorijom (eng. distributed memory systems). Takva računala pružaju dovoljno procesorske snage za efikasno rješavanje svojstvenih problema, međutim relativno ih je teško programirati, zahtijevaju velike troškove održavanja i troše puno električne energije potrebne za rad i hlađenje. Pojavom grafičkih kartica opće namjene (GPGPU) problem potrebe za velikom računalnom snagom je djelomično riješen, međutim, zbog sve većih problema s kojima se moderna znanost susreće i relativno malog memorijskog kapaciteta grafičkih kartica, računalna snaga GPU kartice se ne može u potpunosti iskoristiti. Kako bi se riješio taj problem, u zadnjih nekoliko godina počelo se intenzivno raditi na razvoju hibridnih algoritama koji istovremeno koriste računalnu snagu i memoriju višejezgrenih procesora i grafičkih kartica. Međutim, jedan od glavnih problema grafičkih kartica, relativno mali kapacitet memorije te mala propusnost (eng. bandwidth) između glavne memorije procesora i memorije grafičke kartice, i dalje predstavlja usko grlo računanja. Taj problem pogotovo odlazi od izražaja prilikom rješavanje problema čiji memorijski zahtjevi uvelike premašuju kapacitet grafičkih kartica, često zvani i problemi vanjske memorije (eng. out-of-core problems). Jedno od uskih područja istraživanja je i razvoj algoritama koji će omogućiti brzo rješavanje problema vanjske memorije. Uvidom u trenutno stanje tehnike dolazi se do zaključka da se, iako postoje implementacije koje su sposobne postići vrlo visoke performanse na grafičkim procesorima, postižu loše ili uopće ne mogu riješiti probleme koji premašuju kapacitet memorije grafičkih kartica. Jedan od takvih problema je i rješavanje gustih simetričnih svojstvenih problema, koji je i glavni cilj ovog doktorskog istraživanja.

Glavni cilj ovog doktorskog istraživanja je razviti i analizirati programske strategije koje će poboljšati performanse postojećih algoritama za rješavanje velikih punih simetričnih problema svojstvenih vrijednosti na GPU procesorima za probleme čiji memorijski zahtjevi premašuju kapacitet memorije grafičkih kartica. Tri specifična cilja su: dizajn i razvoj novih programskih strategija i algoritama linearne algebre za probleme vanjske memorije, redizajn i poboljšanje postojećih algoritama za velike pune simetrične problem na hibridnim arhitekturama temeljenima na GPU, te evaluacija performansi novih algoritama na velikim problemima koji se javljaju u molekularnoj dinamici.

U Poglavlju 1 (Uvod) predstavljena je motivacija koja leži iza provedenog istraživanja te se uvode neki temeljne pojmove nužni za razumijevanje ovog doktorskog rada. Također u ovom poglavlju dan je i kratki pregled najnovijih istraživanja iz područja numeričke linearne algebre, računarstva visokih performansi te razvoja naprednih algoritama prilagođenih izvršavanju na grafičkim karticama opće namjene. Predstavljeni su postojeći algoritmi i biblioteke za rješavanje problema svojstvenih vrijednosti na hibridnim arhitekturama, kao i njihovi nedostaci, pogotovo iz perspektive rješavanje vrlo velikih problema, odnosno problema vanjske memorije. U nastavku su navedeni glavni ciljevi rada te poželjna svojstva koja novi algoritmi, razvijeni u okviru rada, trebaju ispunjavati.

U Poglavlju 2 uvode se temeljni matematički pojmovi i notacija potrebni za definiranje i razumijevanje problema svojstvenih vrijednosti, kao što su vektori, matrice, matrična norma te neka njihova svojstva. Zatim se definiraju svojstvene vrijednosti i svojstveni vektori, te se objašnjavaju temeljne matematičke metode kao što su spektralni rastav (dekompozicija), te Krilovljevi i invarijantni potprostori. Na kraju poglavlja dana je i kratka analiza i ocjena grešaka dobivenih svojstvenih vrijednosti te se definira pojam povratne stabilnosti (eng. backward stability) algoritama.

Poglavlje 3 opisuje arhitekturu modernih računalnih sustava temeljenih na grafičkim karticama opće namjene (GPGPU) poznatih pod nazivom hibridni računalni sustavi. Ovo poglavlje podijeljeno je u tri veća potpoglavlja, te je u svakom opisan jedan od tri dijela hibridnog računalnog sustava. Prvo potpoglavlje donosi sažeti pregled povijesti razvoja modernih računalnih sustava, od prvih mikroprocesora do današnjih modernih višejezgrenih procesora. Nadalje, opisani su razni tipovi paralelnih arhitektura, načini ostvarivanja paralelizma na razini sklopovlja i programskoj razini, te memorijska hijerarhija koja je od posebne važnosti za ostvarivanje visokih performansi u današnjim modernim računalnim sustavima. Drugo potpoglavlje opisuje arhitekturu i programske modele najnovijih GPU kartica, s posebnim naglaskom na NVIDIA tehnologije, te detaljno pojašnjava zašto su takve računalne arhitekture višestruko brže nego tradicionalni višejezgreni procesori. U zadnjem potpoglavlju je iznesen kratki pregled dizajna jedne takve hibridne računalne arhitekture zasnovane na grafičkim karticama. Također, obrađen je i problem latencije te brzine prijenosa podataka između glavne memorije i memorije grafičke kartice te njihov utjecaj na ukupne performanse sustava. Na kraju ovog poglavlja uvodi se nova metodologija u kojoj se memorija grafičke kartice promatra kao velika i brza „cache“ memorija što predstavlja osnovu daljnjem istraživanju ovog doktorskog rada.

U Poglavlju 4 detaljno su opisane najnovije metode i algoritmi za rješavanje gustih simetričnih problema svojstvenih vrijednosti. U uvodnom potpoglavlju opisani su algoritmi i njihove implementacije koje se koriste za transformaciju generaliziranog svojstvenog problema u standardnu formu. U sklopu istraživanja oblikovana su tri nova algoritma za rješavanje velikih problema na grafičkim karticama koji se temelje na postojećim metodama opisanima u ovom poglavlju, a to su: višestupanjska (eng. multi-stage) metoda, Lanczova metoda temeljena na Krilovljevnim potprostorima i spektralna podijeli-i-vladaj (eng. divide-and-conquer) metoda. Za svaku od postojećih metoda dan je detaljan opis, najnovija dostignuća i implementacije, ali i diskutirani glavni nedostaci za efikasno rješavanje velikih problema, pogotovo takvih za koje je potrebna memorija veća od kapaciteta memorije grafičkih kartica.

Poglavlje 5 donosi detaljan opis novih algoritama za rješavanje velikih gustih simetričnih svojstvenih problema na hibridnim arhitekturama. Za svaki od postojeća tri algoritma, opisanih u poglavlju 4, detaljno su opisana poboljšanja te promjene u dizajnu, a koji su nužni kako bi se postiglo da se i problemi koji premašuju memoriju grafičkih kartica mogu ubrzati uz istovremeno postizanje vrlo visokih performansi, kao što su kraće vrijeme izvođenja ili postizanje visokog broja izvršenih računskih operacija u sekundi (eng. flops). Neke od metoda unaprijeđenja opisanih u ovom poglavlju su blok-algoritmi za GPU, smanjenje volumena podataka i broja memorijskih prijenosa preoblikovanjem toka podataka u postojećim algoritmima te smanjenje složenosti algoritama iskorištavajući specifičnu strukturu matrica (npr. u spektralnoj podijeli-i-vladaj metodi). Izvedba predloženih algoritama, kao i postojećih metoda opisanih u prethodnom poglavlju, temelji se na dostupnim programskim primitivima prilagođenima izvođenju u hibridnom okruženju,

što omogućava izravnu primjenu novih algoritama u hibridnim računalnim sustavima.

Analiza performansi i vrijeme izvršavanje novih algoritama opisanih u 5. poglavlju su dani u Poglavlju 6. Analize su provedene nad stvarnim problemima proizašlima iz molekularne dinamike, konkretno iz simulacije gibanja velikih makro-molekularnih struktura. Svaki od tri nova algoritma je posebno analiziran te dobiveni rezultati detaljno diskutirani uz napomenu kako postići maksimalne performanse. U sklopu provedenih analiza pokazano je da sva tri nova algoritma ostvaruju vrlo visoke performanse nad specifičnim problemima čak i kad dani problem ne stane u memoriju grafičke kartice. Također, iznesena je i kratka diskusija u kojim slučajevima je koja metoda bolja te koji su optimalni parametri za postizanje najboljih rezultata.

Završno poglavlje, Poglavlje 7, donosi zaključno razmatranje, ukratko analizira ostvarenje očekivanih doprinosa te predstavlja smjernice za daljnje istraživanje.

Rješavanje velikih problema jedan je od ključnih ciljeva računale znanosti. Jedan od pristupa rješavanju problema je korištenjem hibridnih arhitektura temeljenih na grafičkim procesorima opće namjene. Takav pristup se pokazao kao iznimno učinkovit za rješavanje problema čiji memorijski zahtjevi ne premašuju kapacitet GPU memorije. Međutim, pokazalo se da najnovije implementacije ili ne postižu željene performanse ili uopće ne mogu riješiti probleme koji premašuju kapacitet GPU memorije. Stoga je problem korištenja GPU procesora za računanje vrlo velikih problema glavni predmet istraživanja ovog doktorskog rada. Tijekom istraživanja razvijena su tri nova algoritma, temeljena na postojećim metoda, koji omogućavaju rješavanje vrlo velikih problem, tj. problema vanjske memorije, na grafičkim procesorima.

Kako bi se postojeći algoritmi i metode uspješno redizajnirali u svrhu rješavanje problema vanjske memorije na GPU procesorima, razvijeno je nekoliko metoda i tehnika koje omogućuju izvršavanje osnovnih problema linearne algebre, kao što su npr. matrično množenje. Pokazano je da se algoritmi, koji se temelje na matričnim operacija, konkretno BLAS-3 operacijama, mogu vrlo efikasno implementirati tako da u potpunosti sakriju negativne posljedice sporog kopiranja između glavne memorije i GPU memorije te tako postignu vrlo visoke performanse. Korištenjem takvih tehnika, uz redizajn postojećih metoda, moguće je dobiti vrlo brze i efikasne algoritme za rješavanje velikih punih simetričnih problema svojstvenih vrijednosti.U višestupanjskoj metodi, odlučujući faktor koji utječe na performanse je širina vrpce tražene vrpčaste matrice. Nadalje, Lanczosova metoda, iako se ne može opisati u pogledu BLAS-3 operacija, pokazala se kao najbrža metoda za dani testni skup problem, ali samo za slučajeve kad se traži samo mali broj svojstvenih vrijednosti. Spektralna podijeli-i-vladaj metoda postiže daleko najbolje performanse u pogledu broj računskih operacija u sekundi (FLOPS) međutim, zbog iznimno velike složenosti samog algoritma, pokazala se kao najsporija. Sve tri razvijene metode pokazale su da se veliki problemi mogu efikasno rješavati na hibridnim arhitekturama temeljenima na GPU procesorima, te da su značajno brže od postojećih višejezgrenih implementacija.

Ključne riječi: numerička linearna algebra, svojstvene vrijednosti, tehnike vanjske memorije, GPU, računarstvo visokih performansi

Contents

1. Introduction **1**
- 1.1. Motivation 1
- 1.2. State-of-the-art 2
- 1.3. Objectives 4
- 1.4. Structure of the document 5

2. Mathematical Background **8**
- 2.1. Basic notations and matrix operations 8
- 2.2. Eigenvalues and eigenvectors 12
- 2.3. Invariant subspaces 15
- 2.4. Krylov subspaces 16
- 2.5. Overview of perturbation theory 17

3. GPU-based Hybrid Computing Architectures **21**
- 3.1. Hybrid architectures 21
  - 3.1.1. The evolution of computing architectures 21
  - 3.1.2. Parallel architectures and parallelism 23
  - 3.1.3. Memory hierarchy 26
- 3.2. General–purpose GPU computing 28
  - 3.2.1. Architecture of modern GPUs 29
  - 3.2.2. NVIDIA programming model 31
- 3.3. A GPU-based hybrid system 34

4. Dense Symmetric Eigensolvers **37**
- 4.1. Dense generalized eigenproblem 38
- 4.2. Direct eigensolvers 39
  - 4.2.1. One–stage approach 40
  - 4.2.2. Multi–stage approach 42
- 4.3. Krylov subspace based approaches 48
- 4.4. Spectral divide-and-conquer based algorithms 53

5. Dense Eigensolvers for Hybrid Architectures **58**
- 5.1. Multi-stage reduction to tridiagonal form 59
  - 5.1.1. Reduction to band form 60
  - 5.1.2. Reduction to narrower band form 67
- 5.2. The Krylov subspace method 69
  - 5.2.1. Introduction to the algorithm 69
  - 5.2.2. Explicit Krylov subspace algorithm (KE) 70
  - 5.2.3. Implicit Krylov subspace algorithm (KI) 71

5.2.4. Algorithm improvements . . . 72
5.3. Spectral divide-and-conquer based on QDWH . . . 73
5.3.1. Out-of-core spectral D&C algorithm . . . 74
5.3.2. OOC QDWH algorithm . . . 75
5.3.3. Subspace iteration . . . 81

6. Numerical Experiments **84**
6.1. Macromolecular motions . . . 84
6.1.1. IC-NMA method . . . 85
6.2. Benchmarks . . . 86
6.3. Multi-stage reduction . . . 88
6.4. The Krylov subspace approach . . . 94
6.5. Spectral D&C approach . . . 99
6.6. Comparison of OOC-GPU algorithms . . . 103

7. Conclusion **107**
7.1. General contributions . . . 107
7.1.1. OOC multi-stage reduction . . . 109
7.1.2. OOC Krylov subspace-based method . . . 110
7.1.3. OOC spectral divide–and–conquer . . . 110
7.2. Future research . . . 111

Appendices **113**

A. A List of the BLAS Routines **113**

Bibliography **115**

Biography **124**

Životopis **126**

List of Figures

2.1. Backward and forward errors for $y = f(x)$. 17

3.1. Growth in clock rate of microprocessors since 1978.* 22
3.2. Example of two types of SMP architectures. A uniform memory architecture where all processors share the common resources (i.e. main memory) uniformly (a) and a non-uniform memory architecture where each processor has its local main memory (b). 25
3.3. The architecture of multi-core processor, left) separate L2 cache, right) shared L2 cache. 26
3.4. Memory hierarchy of standard single-node computing system. The most recent processors have additional L4 cache level. 27
3.5. Conceptual difference between traditional multi-core processor and general-purpose graphic processor. 30
3.6. NVIDIA Fermi architecture 31
3.7. The program work-flow in the host-device programming model 32
3.8. The organization of threads into blocks and grids and the memory hierarchy. 33
3.9. An overview of computational units, memories and interconnections in modern CPU-GPU hybrid system. 34
3.10. Multilevel memory hierarchy of a standard CPU-GPU hybrid architecture. 36

4.1. Partitioning of the matrix during one iteration of routine SYRDB for the reduction to band form. 44
4.2. Annihilation of the outmost sub–diagonals of the first n_b columns of the band matrix and updating the rest of the matrix (a). Bulge chasing of the fill-in block (b) and (c) 46
4.3. Band matrix form with the first n_b columns reduced to the narrower band form. 47
4.4. A schematic preview of the spectral divide and conquer algorithm 54

5.1. OOC reduction of dense symmetric matrix to band form. 61
5.2. Left: Annihilation of the outermost sub–diagonals and updating the rest of the matrix, Right: bulge chasing part of the first fill-in block 68
5.3. Illustration of the QR_OOC factorization on GPU. 77
5.4. Illustration of the kernels UPDATE_OOC and QR_Hybrid 79
5.5. Illustration of modified GEMM where C is symmetric and A upper triangular matrix. 80

6.1. Two SYRDB variants for the reduction form dense to band form. (a) and (b) present total execution time and (c) and (d) GFLOPS. 91

6.2. Performance of the in-core GPU TM3 for `1cwp` problem ($n = 30504$) with different values for algorithmic block size nb. 92

6.3. Performance of the 2-stage and 3-stage GPU OOC eigensolvers for MD problems Table 6.1. 100 smallest eigenpairs are computed. 93

6.4. Performance scaling of 3-stage (black) and 2-stage (green) OOC variants for different matrix sizes for `1cwp` MD collection. 94

6.5. Execution time of in-core GPU and multi-core versions of Krylov-subspace variants KE and KI. 96

6.6. Execution time (left) and GFLOPS (right) of the two GPU out-of-core Krylov-subspace variants. Only 100 largest eigenvalues are computed. . . . 96

6.7. Execution time and GFLOPS rate for KE (a),(c) and KI (b),(d) on GPU for MD collection `1cwp` with 10, 100 and 500 largest eigenvalues computed. 98

6.8. Execution time (left) and Gflops rate (right) of improved OOC-GPU QR algorithm compared to MAGMA OOC-GPU implementation. 100

6.9. Performance of the QDWH1 (QR fact.), QDWH2 (construction of Q) and QDWH3 (update X_{i+1}) kernels. 101

6.10. Speedup of three OOC-GPU eigensolvers relative to 1-stage CPU eigensolver. Only 1% smallest eigenvalues are computed. 104

6.11. GFLOPS rate of the OOC-GPU eigensolvers applied to MD problems. Only 1% smallest eigenvalues are computed. 105

List of Tables

3.1. Flynn's taxonomy. 24
3.2. Typical capacities and access time for different memory levels (per core). . 28

6.1. Benchmark of large-scale macromolecules. 87
6.2. The list of our GPU-based eigensolvers with average flops counts. 88
6.3. Routines required in reduction of dense symmetric matrix to tridiagonal form on GPU. TD: one-stage reduction, TM: multi-stage reduction. 89
6.4. The variants of QR factorization and two-sided update with their GPU memory requirements and number of data transfered, where n is matrix dimension, w band size and b algorithmic block size. 89
6.5. The cost of accumulating the transformations in Q_1 for n=35,000 and the percentage of the total execution time of SYRDB. 92
6.6. Execution time of the in-core GPU implementation of TM2 for `1cwp` problem ($n = 30504$) compared to SBR routine SBRDT employing 8 cores. 93
6.7. Routines required to build Krylov subspace approaches. GS: reduction to standard eigenproblem, KE: explicit Krylov, KI: implicit Krylov, BT: back-transformation. 95
6.8. Total time in seconds for the implicit (KI) and explicit (KE) Krylov subspace variants on multi-core processors and GPU. 97
6.9. Routines required to build spectral divide–and–conquer OOC-GPU eigensolver. 99
6.10. Splitting points for σ strategies SD&C-A, SD&C-B and SD&C-C. 102
6.11. Number of iterations and execution time (in seconds) for QDWH algorithm applied on MD collection UTUB20. 103

A.1. Description and the number of the floating-point operation of the BLAS-2 routines used in the dissertation. 113
A.2. Description and the number of the floating-point operation of the BLAS-3 routines used in the dissertation. 114

List of Algorithms

4.1. Direct eigensolver for standard eigenproblem 40
4.2. Multi–stage eigensolver 43
4.3. Reduction from band to narrower band form, $toBand(A, n, w_1, d, n_b)$ 46
4.4. Multi-step reduction from band to narrower band matrix 48
4.5. Basic Arnoldi algorithm 51
4.6. Basic Lanczos algorithm 52
4.7. QDWH based spectral D&C algorithm 55
4.8. Subspace iteration algorithm 57
5.1. OOC reduction to band form 60
5.2. Hybrid QR decomposition 62
5.3. OOC variant 1: $X_1 := A_2W$ 65
5.4. OOC variant 2: $X_1 := A_2W$ 66
5.5. OOC variant 2: $X_3 = X_1 + YX_2$ 66
5.6. OOC variant 2: Symmetric 2k update 67
5.7. In-core GPU reduction to narrower band form 68
5.8. Explicit Krylov subspace algorithm 71
5.9. Implicit Krylov subspace algorithm 72
5.10. QDWH based OOC spectral D&C algorithm 75
5.11. QDWH algorithm for polar decomposition 76
5.12. QR_OOC(n, k, b, D) 77
5.13. Update_OOC(n, k, b, E, F) 78
5.14. QR_Hybrid(n, k, b, E); 78
5.15. OOC $C = \alpha BA^T + \beta C$ 81
5.16. Out-of-core subspace iteration algorithm 82

Chapter 1

Introduction

1.1 Motivation

Computing the eigenvalues and eigenvectors of a matrix is one of the central problems in linear algebra. In many problems that arise in a number of research fields such as quantum mechanics, molecular dynamics, dense functional theory, and principal component analysis, computing the eigenvalues becomes the main computational bottleneck. The complexity of solving the eigenproblem depends on the number of sought-after eigenvalues and eigenvectors, size of the problem, i.e. dimension of the representation matrix, and structure of the eigenproblem. In some problems, the complexity can be significantly reduced by exploiting the sparsity structure of the matrix. However, for some eigenproblems, such as ones arising in molecular dynamics, the complexity cannot be reduced since all entries are non-zeros. Such problems are called dense eigenproblems and their efficient solution requires a large amount of storage space and computational power. A special type of dense eigenproblem, covered by this dissertation, is the symmetric dense eigenproblem that is the most computationally intensive component of macromolecular motion simulations.

Traditionally, in order to speed-up solving of the dense symmetric eigenproblem, different parallel programming models have been employed that can exploit the performance of parallel computing architectures such as multi-core processors. When solving very large dense eigenproblems, supercomputers and large distributed systems, composed of a large number of computational units (processors and/or cores) and a large amount of memory, are used. The main disadvantages of such systems are very high costs of procurement, maintenance, and energy for power and cooling. Moreover, distributed systems are very hard to be fully exploited due to different scalability problems as well as latency and bandwidth between different distributed memory locations.

The most significant milestone in the high performance computing occurred in 2006, with the release of the first version of CUDA architecture. This novel architecture facilitated the rapid deployment of GPUs as general-purpose computational units capable of performing billions of floating point operations per second (FLOPS). The performance of modern GPU devices exceeds those of the most recent multi-core processors. Because of their extremely high performance and a hardware design that is most suitable for matrix and vector operations, many highly tuned implementations of basic linear algebra algorithms have been developed paving the way for a strong development of GPU-based kernels for advanced linear algebra routines. In addition, these GPU-based kernels significantly reduce the computational time of dense eigenproblems. When the problem size increases, a main drawback appears due to the limited size of the on-board GPU memory which may prevent problems that cannot fit into the GPU memory to exploit its high performance. Therefore, a possible solution to tackle large dense eigenproblem is to utilize a distributed-memory system.

An alternative to distributed systems are hybrid computing systems. Today, the most popular hybrid systems are GPU-based hybrid systems equipped with one or more multi-core processors and one or more GPU devices. The reason for their popularity lies in the high performance of modern GPU processors that significantly outperforms those of CPUs. The top-end GPU devices consist of more than 2000 processing units and can easily deliver more than 1 TFLOPS (10^{12} floating-point arithmetic operations per seconds or FLOPS) while modern multi-core processors exhibit performance in order of a few dozen GFLOPS. Because of their extremely high performance, two supercomputers in the most recent Top 10 supercomputers list are accelerated by GPUs.

The GPU-based hybrid systems are popular, not only because of very high performance exhibited by the attached GPU devices but the possibility to perform a fine-tuned scheduling between the two different types of computational units, multi-core processors and GPUs. This allows applications to balance the workload by off-loading the computationally intensive parts to the GPU while simultaneously performing other operations on the multi-core CPU. Thus, for example, the most computationally intensive parts of eigensolvers, such as matrix-matrix operations, are executed on GPUs, while the memory-bounded parts are performed on multi-core. This design allows the concurrent execution on all computational units potentially exhibiting much higher performance. In this scenario, the main bottleneck in GPU execution, a memory transfer between the system's main memory and the GPU memory, can be hopefully overlapped with the useful computations.

Modern GPU-based eigensolvers for hybrid architectures can attain very high performance by off-loading computationally intensive parts to the GPU devices. However, only a few of them can compute eigenvalues if the problem cannot fit into the GPU memory, i.e. the problem is out-of-core (OOC) from the GPU perspective. Generally, a problem is regarded as out-of-core, with the respect to a certain memory space, if the problem size exceeds the memory capacity. Opposite to the OOC problems, the in-core problems are those that can entirely fit into the certain memory space. The data of the out-of-core problem, too large to fit into the GPU memory, are stored in the main memory and transferred to/from the GPU memory in pieces which may result in large communication overhead. The development of the high performance algorithms that efficiently overcome the communication bottleneck is an extremely challenging task.

1.2 State-of-the-art

The history of graphic processors as general-purpose processors (GPGPU) dates back to 2001 and the paper of Larsen et al. [1]. In that paper the authors introduced the first successful implementation of the matrix-matrix multiplication for graphic processing units. This paper was followed by other papers [2, 3] that proposed novel optimization strategies for basic linear algebra operations and the first LU factorization. These works increased the appeal interest of linear algebra implementations on GPU.

The major breakthrough in the implementation of linear algebra routines on GPUs was initiated with the introduction of the CUDA architecture in 2006 by NVIDIA. This novel architecture turned programming GPUs as general-purpose computational devices easier and thus more accessible to a wider community. One of the most referenced papers in high-performance linear algebra was published by Volkov and Demmel [4]. In that paper, the authors analyzed the performance results of modern GPUs and the implementations of the most common linear algebra routines: matrix-matrix multiplication, and the LU, QR, and Cholesky factorizations.

Together with the adaption of the specific linear algebra routines on GPUs, big vendors, such as NVIDIA, published their computational libraries for basic linear algebra routines. The most widely used is the CUBLAS computational library for NVIDIA GPU architectures. The authors in [5] analyze the performance of the CUBLAS implementations of the level-3 BLAS operations and introduce some optimizations techniques based on hybrid approach that off-loads computations between the CPU and the GPU.

With the evolution of the hybrid computing platforms based on the GPU processors, new methods were developed to exploit the massive hardware concurrency of these emerging computing platforms.The authors in [6] developed an approach for a dense linear algebra routines for platforms equipped with multi-core processors and multiple hardware accelerators. This work was followed by scalable implementation of the Cholesky, LU and QR factorizations for a hybrid system equipped with several GPUs [7] that was the basis for the MAGMA library [8, 9]. The aim of MAGMA library is to implement LAPACK routines for hybrid computing architectures base on GPU and multi-GPUs.

Specifically, conventional eigenvalue solvers can be found in LAPACK [10], ScaLAPACK [11], and PLAPACK [12] for both shared-memory and distributed memory systems. Furthermore, vendor-specific libraries such as MKL [13] and ACML [14] provide a set of eigensolvers highly tuned for Intel and AMD processors, respectively. Recent developments in eigenvalue solvers aim at accelerating both the reduction from dense symmetric matrix to tridiagonal form and the subsequent tridiagonal eigensolver applied for solution of tridiagonal eigenproblem. The reduction is accelerated by introducing a two-step reduction, in which matrices are first reduced to band form, and then consecutively, to tridiagonal form. This approach exhibits higher computational cost but can be efficiently accelerated by off-loading the computations on the GPU. In [15] and [16] the authors off-load the computational intensive task of the reduction phase, such as matrix-matrix multiplication and two-sided update, on the GPU, for one-stage and the multi-stage reduction to tridiagonal form, respectively. Furthermore, the authors in [17] introduce a novel implementation that reduces a symmetric dense matrix to tridiagonal form based on the tile strategy. The improvements in the tridiagonal eigensolvers are mostly based on developing strategies for multi-core architectures, such as that presented in [18] for the four most commonly used tridiagonal eigensolvers: QR iteration [19, 20], bisection and inverse iteration [21, 22], the divide-and-conquer method [23, 24], and the method of multiple relatively robust representation (MRRR) [25, 26, 27].

None of the previous work addresses the solution of very large eigenproblems that exceed the GPU memory, which require the development of novel programming strategies. One approach is to port existing eigensolvers to large memory-distributed computing clusters in which each node is equipped with one or more graphic processors. Such strategy, however, is still unexploited. One of the first approaches to exploit the performance of multi-GPUs was presented in [28] and is part of FLAME project and LIBFLAME library [29]. The dense generalized symmetric eigensolvers presented in [30] outperform those of ScaLAPACK by improving the multi-stage reduction on a large hybrid multi-GPU system. In [31] the authors proposed a multi-stage method that exhibits significantly lower computational cost than the standard multi-stage methods. The implementation in the associated ELPA library [32] targets a distributed system with the nodes equipped with multi-core processors, and assume that the problem data are distributed between the main memories of the nodes.

Hybrid systems based on GPUs have become more and more popular as an alternative to computer clusters. The research in this area aims to enable solving of a very large problem on single-node computing systems by developing an out-of-core strategy. One of the first out-of-core strategies that solve eigenproblems whose data are too large to fit into the main memory of the CPU-only system were presented in [33]. The improvement is based on scheduling techniques that optimize sequential data accesses from the disc and the reuse of data stored in main memory. Out-of-core implementations of LU, QR and Cholesky factorization for distributed memory platforms were presented in [34]. However, the research in the direction of out-of-core GPGPU techniques and algorithms is still a widely exploited area. An initial research of out-of-core techniques that efficiently compute Cholesky and LU factorization for matrices that cannot fit into GPU memory was performed in [35]. A state-of-the-art out-of-core QR factorization can be found in the MAGMA library. Further research was done by the author of this thesis who is one of the authors of the papers [36, 37] that introduce different out-of-core optimization techniques to improve dense symmetric eigensolvers. These techniques aim at reducing the amount of data transferred to/from the main memory and improve the reuse of data stored in the GPU memory.

1.3 Objectives

In the past few years, hybrid computing platforms based on the graphic processors have become very powerful computing systems used in every-day scientific computations. To efficiently solve a large variety of problems on these platforms, numerous specialized high-performance algorithms have been developed. Nevertheless, the majority of the developed algorithms, though yielding very high performance, cannot solve problems whose required storage space exceeds the available GPU on-board memory. That is an obvious disadvantage for numerous linear algebra routines, especially when large problems are to be solved whose performance can be significantly improved by employing GPUs. Such large problems, too large to entirely fit into the GPU memory, are stored in the main memory and transferred in pieces. The development of strategies and algorithms that will hide the memory latency and amortize it with a sufficiently large number of floating-point arithmetic operations (FLOPS) performed on the GPU is extremely challenging.

The general goal of this thesis is thus *to design, develop, and evaluate programming strategies to improve the performance of existing eigenvalue solvers for dense symmetric eigenproblems on a single GPU when the required storage space exceeds the available GPU memory.* This general goal is divided into 3 specific objectives:

1. Design and develop programming strategies and basic linear algebra kernels for out-of-core GPU execution.
2. Re-design and improve the performance of existing eigensolvers for dense symmetric eigenproblems on GPU-based hybrid architectures.
3. Evaluate and compare the performance of the improved eigensolvers on real large eigenvalue problems arising in molecular dynamics.

The performance improvements of advanced linear algebra kernels, e.g. eigenvalues solvers, mostly come from the improvements that are made in the basic linear algebra kernels, such as those from the BLAS computational library. The first objective of this research is to develop a set of highly-tuned BLAS routines, such as GEMM, TRMM and SYMM, that will exhibit high performance even when the matrix cannot fit into the GPU memory. The optimization will be based on extending the blocking strategy, used in cache-aware algorithms, to improve the performance on the GPU by achieving better reuse of data stored in the GPU memory. In addition, a new model will be designed to exploit the matrix structure, if possible, and introduce a new passing strategy that will reduce the number of data transfers between the CPU and GPU.

The second objective is to re-design and improve the performance of existing eigensolvers. The aim of this thesis is not to pursue the improvement of a full set of eigensolvers; however, the improved BLAS kernels and new programming methods, obtained within this research, can be used to improve the performance of other eigensolvers as well. The following three eigensolvers will be addressed in this thesis: eigensolver based on multi-stage reduction to tridiagonal form, implicitly restarted Lanczos method based on Krylov-subspace approach, and spectral divide-and-conquer based eigensolver. These algorithms are improved under the following assumptions:

- The algorithms must be hybrid CPU-GPU based eigensolvers that attain very high performance by employing both CPU and GPU processors concurrently.
- The eigensolvers must be out-of-core algorithms, i.e. the algorithms capable to exploit a GPU even when the problem is too large to fit into the GPU memory.
- The algorithms must include specific strategies that decrease the number and volume of transfers between the main memory and GPU memory in order to attain high performances.

In order to fulfill all three goals, the constructed algorithms have to be re-designed to leverage the highly-optimized out-of-core BLAS routine implementations obtained from the previous objective. In addition, by applying a blocking strategy and increasing the operational intensity (i.e. ratio between data transfered and floating point operations performed) on GPU, it is expected to significantly increase the scalability of the existing eigenvalue solvers across all computational devices of a hybrid platform independently of the problem size.

The final objective is to evaluate the eigensolvers showing how these algorithms improve the solution of the real dense symmetric eigenvalue problems arising in macromolecular motion simulations. Furthermore, we intend to demonstrate that, by carefully re-designing the existing eigenvalue solvers and by employing highly-tuned BLAS and LAPACK out-of-core GPU routines, it is possible to efficiently solve very large problems, even on the modest computational hardware equipped with only one GPU.

1.4 Structure of the document

This thesis is structured in seven chapters. Chapter 1 motivates the research conducted and described in this manuscript. Section 1.1 describes the motivation that encouraged the development of novel eigensolvers that could efficiently solve problems by employing general-purpose graphics processing units (GPGPU). Furthermore, this section pursues the idea of employing a modest computational system equipped with a single GPGPU in the solution of very large eigenproblems. Following the motivation, Section 1.2 survey the state-of-the-art tools and libraries that are traditionally employed for the solution of eigenproblems. The main goals of this research are addressed in Section 1.3 and the structure of the document is given in Section 1.4.

Chapter 2 provides the mathematical background and theory necessary to understand the standard methods that are applied in the solution of the eigenvalue problem. The chapter starts by introducing the basic mathematical notation, matrix operations, and properties that are required to understand the rest of the thesis (Section 2.1). The main mathematical problem of this thesis, computing the eigenvalues and the corresponding eigenvectors, is formulated in Section 2.2. Furthermore, this section brings the theory underlying the symmetric spectral decomposition that aims at reducing the problem into a similar diagonal problem. The invariant subspace is introduced in Section 2.3. It is a theoretical background for many powerful methods and is based on dividing the eigenproblem into subproblems whose eigenvalues are easier to compute. Another approach for the solution of eigenproblems is based on Krylov subspaces. Section 2.4 presents this class of iterative eigensolvers. Finally, Section 2.5 discusses error approximations of obtained eigenvalues, backward error and the stability of the methods.

Chapter 3 describes the architecture of modern hybrid computing systems equipped with one or more general-purpose graphic processing units. The chapter is divided into 3 sections, each describing a different part of a GPU-based hybrid architecture. The host part of a hybrid computing platform, a traditional multi-core processor, is introduced in Section 3.1, from its early days to the most recent multi-core multi-socket architectures capable of executing both data-parallel as well as task-parallel applications. Furthermore, this section also surveys the types of parallel architectures, levels of parallelism, and memory hierarchy of modern computing systems. Section 3.2 describes the most recent GPU architecture and programming models, explaining why GPUs are superior in performance to modern CPU architectures. In Section 3.3 an overview is given of the architectural design of a GPU-based hybrid platform. Moreover, it shows that the memory latency and the bandwidth between CPU and GPU memory are two major constraints on system performance. A new idea is presented there that considers GPU memory as a large “cache” memory.

Chapter 4 addresses the methods employed for the solution of large dense symmetric eigenproblems. Since the problems arising in molecular dynamics are given in generalized eigenproblem form, this chapter starts with the methods and kernels that efficiently transforms this critical problem into standard form by employing optimized Cholesky and matrix-matrix operations, Section 4.1. Once the eigenproblem is in standard form, the standard eigensolvers are employed. These are divided into 3 groups described in the following sections. Section 4.2 presents two direct eigensolvers based on the direct reduction to tridiagonal form. The first solver is the one-stage approach in which half of the flops are performed in terms of memory-bounded level-2 BLAS kernels. To overcome the shortcomings of the one-stage, the multi-stage eigensolver is introduced, which casts the majority of operations in terms of highly-tuned matrix-matrix multiplication based kernels. Opposite to direct eigensolvers, an iterative solver based on the Krylov subspaces is addressed in Section 4.3. The algorithm, implemented in the ARPACK computational library, exhibits a much lower FLOPS rate but is significantly less expensive than direct solvers when only a small subset of eigenvalues is required. Finally, Section 4.4 introduces the spectral divide-and-conquer algorithm that can be cast completely in terms of level-3 BLAS operations.

Chapter 5 improves the state-of-the-art algorithms at the previous chapter by offloading their computationally intensive parts on the graphics processing units. The Chapter pursues the design and implementation of algorithms to efficiently solve the eigenproblems even if required storage space exceeds the available GPU memory. Section 5.1 improves the multi-stage reduction by performing the first stage, reduction from dense to band matrix form, out-of-core from the GPU viewpoint. The GPU OOC blocked algorithms are designed to improve the performance of the QR factorization and the two-side update. The bandwidth is chosen so that the second stage, reduction from band to narrower band form, can be executed in-core from the GPU perspective. Section 5.2 introduces two variants of the algorithm based on the Krylov-subspace, employing explicit or implicit reduction to standard eigenproblem form. The reduction is done by moving the computationally intensive parts to the GPU while memory-bounded Lanczos iterations are performed in-core for GPU or by employing multi-threaded LAPACK library. Section 5.3 reorganizes the spectral divide-and-conquer algorithm by designing an out-of-core QR factorization for matrices with the special structure that appear in computing the polar decomposition.

Chapter 6 analyzes the performance of the three new out-of-core GPU eigensolvers applied to the solution of larger dense symmetric eigenproblems. The eigenvalue problems used as the test cases in the chapter are obtained from the real problems arising in macromolecular motions. Section 6.1 gives a short introduction to macromolecular motions and the basic tools and methods, in particular the normal mode analysis (NMA), that describe the collective motions at macromolecular level. The macromolecular collections used in this research are given in Section 6.3. The same section describes the testing computing system and the libraries used in the experiments. Sections 6.3, 6.4 and 6.5 offer experimental results of each of three OOC eigensolvers, respectively, multi-stage, Krylov subspace, and spectral divide-and-conquer eigensolver. The performance of the new OOC eigensolvers is discussed by comparing it with that of the state-of-the-art eigensolvers. Finally, Section 6.6 compares the performance of the three OOC eigensolvers and discusses which approach is the best to use for a specific eigenvalue problem.

Chapter 7 presents the main conclusions from this research and summarizes the main contributions of the thesis that go beyond the state-of-the-art. Finally, it discusses a few open research directions and future work related to the work done.

Chapter 2

Mathematical Background

This chapter gives a short overview of the mathematical background required to better understand the central problem of this thesis, the eigenvalue problem. This problem belongs to the mathematical discipline called linear algebra and is one of its most studied and widely encountered problems. The eigenvalues and eigenvectors are of special interest in quantum mechanics, molecular dynamics, dense functional theory, and principal component analysis, just to name a few. For example, in molecular dynamics, the eigenvalues and the corresponding eigenvectors describe the collective motions of a large macromolecular system.

In this chapter we will be briefly introduce the mathematical theory that lies behind the methods for solving of eigenvalue problem necessary for the construction of efficient high-performance eigensolvers. For a more detailed description of the underlying linear algebra theory, readers are encouraged to refer to [20, 38].

This chapter is organized as follows. Section 2.1 introduces the basic mathematical notations, matrix operations, and properties that are required in the following section. Eigenvalues and eigenvectors are introduced in Section 2.2. Furthermore, this section defines eigenvalue decomposition together with some useful properties. Sections 2.3 and 2.4 introduce the theoretical background required for the two methods that compute the eigenvalues of symmetric matrices, based on invariant subspaces and the Krylov subspace, respectively. Finally, Section 2.5 offers the perturbation theory and discusses the accuracy of the obtained eigenvalues.

2.1 Basic notations and matrix operations

Before we define the basic mathematical objects, we introduce a term: field. A field is a set of elements that satisfies the field axioms, associativity, commutativity, distributivity, identity, and inverse, for both addition and multiplication. An element of the field is called scalar and represents the most basic object in linear algebra. The most commonly used fields are the field of real numbers ($\mathbb{R}$) and complex numbers ($\mathbb{C}$), but this work will consider objects over the field of real numbers only. In general, scalars of the field $\mathbb{R}$ will be denoted with lowercase Greek letters ($\alpha, \beta, \ldots$). A vector is defined as an element of the finite n-dimensional space $\mathbb{R}^n$ and will be denoted with a lowercase Roman letter ($x, y, a, b, \ldots$),

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix},$$

where $x_i \in \mathbb{R}$ is the i-th component of the vector x. The standard way of presenting vectors are as column-vectors in which $\mathbb{R}$ is identified with $\mathbb{R}^{n\times 1}$. On the other hand, the elements of the vector space $\mathbb{R}^{1\times n}$ are row-vectors, $x = [x_1,\ldots,x_n]^T$. The vectors of the canonical basis of vectors space $\mathbb{R}^{n\times 1}$ are denoted as $e_1, e_2, \ldots, e_n$, where each e_i has all elements zero except the i-th element which is equal to 1.

The vector space denoted by $\mathbb{R}^{m\times n}$ represents the set of all real matrices with m rows and n columns. Matrices will be denoted with uppercase Latin letters $(A, B, \ldots)$, and their elements with the corresponding double-indexed lowercase Latin letter, e.g. (a_{ij}) denotes the element of matrix A positioned in the i-th row and the j-th column. Moreover, the element at position (i, j) is usually denoted as $A(i, j)$ as well. In summary,

$$A \in \mathbb{R}^{m\times n}, \quad A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}, \quad \text{with} \quad a_{ij} \in \mathbb{R}.$$

The submatrix $A(r_i : r_{i+k}, c_j : c_{j+z})$ is defined as the matrix starting at position $A(r_i, c_j)$ with k rows and z columns.

Matrices are one of the main objects in the linear algebra. Different practical problems can be represented in matrix form; for example, any system of linear equations can be represented as a matrix. Matrices have many features that can be exploited during the computations like special structure or properties. The exploitation of the matrix structure, as will be demonstrated in the forthcoming sections, can significantly reduce the number of flops and decrease the execution time. Furthermore, a special property of the matrix (e.g., symmetry) can significantly influence the decision, at which method is the best to apply to a specific eigenvalue problem.

Basic matrix forms

A matrix is said to be square if $m = n$ and rectangular if $m \neq n$. Two basic types of square matrices are the identity matrix, denoted by I, and the diagonal matrix, denoted by D. The diagonal matrix has the off-diagonal equal to zero, while the identity matrix is a special case of diagonal matrix with all diagonal entities equal to 1. The standard notation is $D = \mathrm{diag}(d_1, d_2, \ldots, d_n)$ with d_i a diagonal element. For the solution of eigenvalue problem it is important to define some other matrix structures, called canonical forms:

- A *tridiagonal matrix* is a matrix with entries $a_{ij} = 0$ when $|j - i| > 1$.
- A *band matrix* is a matrix with entries $a_{ij} = 0$ when $|j - i| > w$, where $w \leq n$ is denoted as the band size.
- An *upper (lower) triangular* matrix has all its elements below (above) the main diagonal equal to zero.
- An *upper (lower) Hessenberg* matrix is an upper (lower) triangular matrix plus at most one non-zero subdiagonal (superdiagonal).

Matrices have also some interesting properties that can be exploited as well. An invertible or non-singular matrix is a square matrix A for which there exists a unique matrix B such that:

$$AB = BA = I. \tag{2.1}$$

Then B is called the inverse of the matrix A and is denoted by A^{-1}. If the matrix A is not invertible then it is called singular.

A real square matrix is symmetric if it satisfies:

$$A = A^T,$$

where A^T stands for transpose. Thus, for each i and j, $a_{ij} = a_{ji}$. In other words, a matrix is symmetric if the matrix with exchanged rows and columns is the same as the original matrix. The matrix is antisymmetric if $A = -A^T$.

A matrix A is orthogonal if:

$$AA^T = A^T A = I.$$

From $AA^T = A^T A$ follows that A is also square. Furthermore, the orthogonal matrix A is always invertible since $A^T = A^{-1}$.

Matrix norms

Matrix norms are frequently used in the analysis of the algorithms operating with matrices. In many cases it is important to quantify the "size" of a matrix or the "distance" between two matrices that is not necessarily related to their number of columns/rows. For example, in the error analysis of matrix decomposition, the quality of the computed solution is computed by determining how far it is from the exact solution.

Most of the matrix norms are defined in terms of vector norms. The most frequently-used vector norms are p-norms, defined as

$$\|x\|_p := (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}},$$

where x is a vector and $p \geq 1$. The most commonly-used p-norms are the 1, 2 and ∞ norms, defined as:

$$\begin{aligned} \|x\|_1 &= |x_1| + \cdots + |x_n|, && (2.2) \\ \|x\|_2 &= (|x_1|^2 + \cdots + |x_n|^2)^{\frac{1}{2}}, \text{and} && (2.3) \\ \|x\|_\infty &= \max_{1 \leq i \leq n} |x_i|. && (2.4) \end{aligned}$$

A vector x for which $\|x\| = 1$ holds is called a *unit* vector.

The linear operator $f : \mathbb{R}^{m \times n} \mapsto \mathbb{R}$ is called a matrix norm if the following properties hold:

$$\begin{aligned} f(A) &\geq 0, && A \in \mathbb{R}^{m \times n}, \\ f(A) &> 0, && \text{for} A \neq 0, \\ f(A+B) &\leq f(A) + f(B), && A, B \in \mathbb{R}^{m \times n}, \\ f(\alpha A) &= |\alpha| f(A), && \alpha \in \mathbb{R}, A \in \mathbb{R}^{m \times n}, \end{aligned}$$

and is denoted as $f(A) = \|A\|$.

The most frequently-used matrix norms in linear algebra are:

- Frobenius norm:

$$\|A\|_F := \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|^2},$$

- p-norm:

$$\|A\|_p := \sup_{\neq 0} \frac{\|Ax\|_p}{\|x\|_p}$$

where p is $1 \leq p \leq \infty$. Note that the p-norms are defined in terms of vector norms. The proof that both the Frobenius and p-norms satisfy the norm properties is trivial. Similar to the vector norms, the three special p-norms, apart from the Frobenius norm, that are most frequently-used in matrix analysis are the 1, 2 and ∞ norms. From (2.1) we can deduce that Frobenius norm is easy to compute. However some other norms are not. The Frobenius norm and p-norm satisfy some inequalities that are important in the analysis of matrix computations. For $A \in \mathbb{R}^{m\times n}$, the following properties are defined:

$$\|A\|_2 \leq \|A\|_F \leq \sqrt{n}\|A\|_2,$$

$$\max_{i,j}|a_{ij}| \leq \|A\|_2 \leq \sqrt{mn},$$

$$\|A\|_1 = \max_{1\leq j\leq n}\sum_{i=1}^{m}|a_{ij}|,$$

$$\|A\|_\infty = \max_{1\leq i\leq m}\sum_{j=1}^{n}|a_{ij}|,$$

$$\frac{1}{\sqrt{n}}\|A\|_\infty \leq \|A\|_2 \leq \sqrt{m}\|A\|_\infty,$$

$$\frac{1}{\sqrt{m}}\|A\|_1 \leq \|A\|_2 \leq \sqrt{n}\|A\|_1.$$

The 1, ∞ and Frobenius norm are mostly used since they have nice properties and they are easy to compute. The 2-norm can be roughly estimated in terms of other norms, but is considerably more complicated to compute. For more details, the readers are referred to Golub and van Loan's book [20].

Basic matrix operations

On vector space $\mathbb{R}^{m\times n}$, the basic matrix computations defined next are used as building blocks for more advanced matrix operations. The basic matrix operations are:

- *Transposition* of matrix $A \in \mathbb{R}^{m\times n}$ into matrix $C \in \mathbb{R}^{n\times m}$

$$C = A^T \Rightarrow c_{ij} = a_{ji}.$$

- *Addition* of two matrices $A, B \in \mathbb{R}^{m\times n}$

$$C = A + B \Rightarrow c_{ij} = a_{ij} + b_{ij}.$$

- *Scalar-matrix multiplication* with $\alpha \in \mathbb{R}$ and $A, C \in \mathbb{R}^{m\times n}$

$$C = \alpha A \Rightarrow c_{ij} = \alpha a_{ij}.$$

- *Matrix-matrix multiplication* with matrices $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$ and $C \in \mathbb{R}^{m\times n}$

$$C = A \cdot B \Rightarrow c_{ij} = \sum_{p=1}^{k} a_{ip}b_{pj}.$$

In basic matrix operations we can also include two operations that are very commonly used in different matrix analysis: trace and determinant. The trace of a matrix $A \in \mathbb{R}^{n\times n}$ is defined as the sum of all diagonal elements:

$$\mathrm{tr}(A) = \sum_{i=1}^{n} a_{ii}.$$

The determinant of the square matrix A is a real number denoted by $\det(A)$ or $\|A\|$ and it is computed through the process called "*Laplacian expansion*"

$$\det(A) = \sum_{i=1}^{n} (-1)^{i+j} a_{ij} M_{ij},$$

where M_{ij} is the minor of A formed by eliminating row i and column j. The determinant of a 1×1 matrix $A = (a)$ is defined as the scalar a. Although this technique is efficient for small matrices, other approaches such as Gaussian elimination, are much more efficient when the matrix size becomes large.

The basic properties of determinants for square matrices $A, B \in \mathbb{R}^{n \times n}$ and a scalar c are:

$$\begin{aligned}
&\det(AB) = \det(A)\det(B),\\
&\det(A^T) = \det(A),\\
&\det(cA) = c^n \det(A).\\
&\det(A) \neq 0 \Leftrightarrow A \text{ is nonsingular},\\
&\det(A^{-1}) = \frac{1}{\det(A)} = \det(A)^{-1}.
\end{aligned}$$

Although computing the determinant via Gaussian eliminations or directly via Laplacian expansion is complex, for some simple matrix forms, the determinant is quite easy to obtain. The determinant of the identity matrix $I \in \mathbb{R}^{n \times n}$ equals 1 and this property can be easily proven by applying the basic properties of determinants. If we choose any nonsigular matrix A, such that $I = AA^{-1}$, then:

$$\det(I) = \det(AA^{-1}) = \det(A)\det(A^{-1}) = \det(A)\det(A)^{-1} = \det(A)\frac{1}{\det(A)} = 1.$$

Furthermore, if matrix A is diagonal or upper/lower triangular, its determinant is the product of diagonal elements, i.e. $\det(A) = a_{11}a_{22}\cdots a_{nn}$.

2.2 Eigenvalues and eigenvectors

Definition (Eigenvalues and eigenvectors): A real number $\lambda \in \mathbb{R}$ is an eigenvalue of the matrix $A \in \mathbb{R}^{n \times n}$ if there exists a non-zero vector $x \in \mathbb{R}^n$ such that:

$$Ax = \lambda x. \tag{2.5}$$

The vector x is called the right eigenvector associated with the eigenvalue λ and (λ, x) is called an eigenpair of matrix A. The set of all eigenvalues of a given matrix A is called the *spectrum* and is denoted as $\sigma(A)$.

Definition An eigenspace associated to the eigenvalue λ is defined as

$$\mathcal{P}_\lambda = \{u \in \mathbb{R}^n \ : \ Au = \lambda u\}.$$

In other words, the eigenspace $\mathcal{P}_\lambda$ contains all the eigenvectors associated with the eigenvalue λ. An alternative definition is that the eigenvalues of a matrix $A \in \mathbb{R}^{n \times n}$ are the roots of its characteristic polynomial:

$$p(\lambda) = \det(A - \lambda I) = 0.$$

The characteristic polynomial is an n-th order polynomial equation with the n solutions, i.e. eigenvalues. The *spectrum* is usually represented by a diagonal matrix $\Lambda(A)$ with the eigenvalues λ_i on its diagonal. A shorter notation Λ can be also used if it is clear to which matrix the spectrum corresponds to.

Note also that all the eigenvalues of a matrix are not necessarily real numbers. For example, observe a real matrix:

$$A = \begin{bmatrix} 3 & -2 \\ 4 & -1 \end{bmatrix},$$

whose characteristic polynomial is $\det(A - \lambda I) = \begin{vmatrix} 3-\lambda & -2 \\ 4 & -1-\lambda \end{vmatrix} = \lambda^2 - 2\lambda + 5 = 0$. Since the eigenvalues are roots of the characteristic polynomial, for A those are $\lambda_1 = 1 + 2i$ and $\lambda_2 = 1 - 2i$. However, if A is real and symmetric, then the symmetry guarantees that all eigenvalues are real and that there exists an orthonormal basis of eigenvectors.

A more convenient way of representing the eigenvalues and the corresponding eigenvectors from Equation (2.5) is by using the matrix representation. If the matrix X is constructed such that its columns are eigenvectors x_i of A, then the Equation (2.5) can be reformulated as:

$$AX = X\Lambda, \tag{2.6}$$

where the diagonal elements of matrix Λ are the eigenvalues and the columns of X are the unknown eigenvectors associated with Λ.

The trivial cases are identity and diagonal matrices whose eigenvalues are their diagonal elements. The proof for the identity matrix is trivial since this matrix maps each vector x to itself, i.e. $Ix = x$. Thus, the eigenvalues are all equal to 1. To prove the statement for the diagonal matrix, let's observe the eigenvalue definition via the characteristic polynomial: λ is eigenvalue only if it is a root of the polynomial $\det(D - xI) = 0$. Furthermore, the determinant is equal to zero only if the matrix is singular. From that, it is obvious that $\lambda = d_i$, for each $i = 1, \ldots, n$, is a root of the characteristic polynomial since matrix $D - d_i I$ is singular because i-th row and column are equal to zero. The same holds for the upper (lower) triangular matrices whose eigenvalues are also their diagonal elements.

The property of diagonal matrices to allow "reading-off" the eigenvalues from the diagonal presents the basic idea in the solution of symmetric eigenvalue problems.

Computing the eigenvalues directly from a square matrix via the Equation (2.5) or the characteristic polynomial is impractical and slow when the dimension of the matrix is large and when all eigenvalues are required. Many approaches are based on the reduction of the matrix into simpler forms, such as diagonal or tridiagonal form, with the property that the eigenvalues are simply "read-off" from the diagonal. Before we start with the eigenvalue decomposition, we introduce similarity transformations that will define connection between the eigenvalues and eigenvectors of a square matrix and a diagonal matrix similar to it.

Definition (Similarity transformation): A square matrix $B \in \mathbb{R}^{n \times n}$ is said to be similar to $A \in \mathbb{R}^{n \times n}$ if there exists a non-singular matrix $X \in \mathbb{R}^{n \times n}$ such that:

$$B = X^{-1} A X.$$

The mapping $A \longmapsto B$ is called a similarity transformation.

The definition holds for other direction as well. If B is similar to A, then A is similar to B because of $A = XBX^{-1}$, since X is invertible (non-singular). The similarity is necessary in solving the eigenproblems since it preserves some useful properties of the matrices. Two similar matrices A and B share the following properties:

- $\Lambda(A) = \Lambda(B)$.
- $A^T \longmapsto B^T$, $A^{-1} \longmapsto B^{-1}$.
- $\det(A) = \det(B)$, $\operatorname{tr}(A) = \operatorname{tr}(B)$.
- $p_A(x) = p_B(x)$.

If we observe the characteristic polynomial of B, then we can see that:

$$\begin{aligned} \det(B - \lambda I) &= \det(X^{-1}AX - \lambda X^{-1}X) \\ &= \det(X^{-1}(A - \lambda I)X) \\ &= \det(X^{-1})\det(A - \lambda I)\det(X) \\ &= \det(A - \lambda I), \end{aligned}$$

so A and B have the same characteristic polynomial and hence the same eigenvalues. However, the similarity transformation changes eigenvectors such that if v is an eigenvector of B then Xv is the eigenvectors of A and vice versa, for each eigenvector z of A, $X^{-1}z$ is the eigenvector of B. From the similarity of A and B follows that exists X such that $A = XBX^{-1}$. Suppose that (λ, z) is an eigenpair of A, then:

$$\begin{aligned} Az &= \lambda z, \\ XBX^{-1}z &= \lambda z, \\ B(X^{-1}z) &= \lambda(X^{-1}z), \\ Bv &= \lambda v. \end{aligned}$$

The eigenvalue of B is equal to λ and $v = X^{-1}z$ is the corresponding eigenvector. The idea behind the similarity is to make matrix A as simple as possible while preserving its essential properties, such as the spectrum. This property is applied in the eigenvalue decompositions such as the Schur decomposition, for general square matrices, or spectral decomposition, in the case the matrix is symmetric.

Symmetric eigenvalue decomposition

For general square matrices, the Schur decomposition [20] (Theorem 7.1.3) states that for every square matrix there exists a similar orthogonal upper triangular matrix. The special case of the Schur decomposition, when the matrix is symmetric, is called *symmetric Schur decomposition*, or *spectral decomposition* and implies that the matrix is similar to a diagonal matrix. The following theorem describes the spectral decomposition.

Theorem 2.2.1 (Spectral decomposition) *If $A \in \mathbb{R}^{n\times n}$ is symmetric, then there exists an orthogonal matrix $Q \in \mathbb{R}^{n\times n}$ such that*

$$Q^T AQ = D. \tag{2.7}$$

where $D = \operatorname{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ is a diagonal matrix.

Proof: The proof can be found in [20], Theorem 8.1.1.

The diagonal elements of D are the eigenvalues and the columns of Q are the associated eigenvectors. The matrix is **diagonalizable** if it is similar to a diagonal matrix. The eigenvalues of a symmetric matrix satisfy the following property.

Theorem 2.2.2 (Courant-Fisher) *If* $A \in \mathbb{R}^{n\times n}$ *is symmetric with eigenvalues* $\lambda_1 < \lambda_2 < \ldots < \lambda_n$*, then*

$$\lambda_k = \max_{\dim(X)=k} \min_{0\neq y\in X} \frac{y^T A y}{y^T y}, \qquad k = 1,\ldots,n \tag{2.8}$$

Proof: The proof can be found in [20], Theorem 8.1.2.

If matrix A is positive definite, i.e. $x^T Ax > 0$ for all $x \neq 0$, then the fraction in (2.8) is strictly positive and all eigenvalues λ_k are greater than zero.

2.3 Invariant subspaces

Definition (Invariant subspace): The subspace $\mathcal{S} \subseteq \mathbb{R}^n$ is said to be invariant for A if $A\mathcal{S} \subseteq \mathcal{S}$. In precise mathematical notation, $\mathcal{S}$ is A-invariant if

$$\forall x \in \mathcal{S} \Rightarrow Ax \in \mathcal{S}.$$

The most trivial examples of invariant subspaces are the zero-space 0 and $\mathbb{R}^n$, which are always A-invariant. Note that the space $\mathcal{X} = \text{span}\{v_1, v_2, \ldots, v_n\}$, spanned by the eigenvectors of A, is also A-invariant. For every $x \in \mathcal{X}$ there exist coefficients $\alpha_1, \alpha_2, \ldots, \alpha_n$ such that $x = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n$, i.e. x is a non-trivial linear combination of v_i. If we recall that each v_i is an eigenvector of A, then:

$$\begin{aligned} Ax &= A(\alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n) \\ &= \alpha_1 Av_1 + \alpha_2 Av_2 + \cdots + \alpha_n Av_n \leftarrow v_i \text{ is eigenvector} \\ &= \alpha_1 \lambda_1 v_1 + \alpha_2 \lambda_2 v_2 + \cdots + \alpha_n \lambda_n v_n. \end{aligned}$$

Thus, for each $x \in \mathcal{X}$, the resulting vector Ax is the linear combination of vectors v_i with coefficients $\alpha_i \lambda_i$.

In addition, each eigenspace $\mathcal{P}_\lambda$ is A-invariant as for $\forall u \in \mathcal{P}_\lambda$ is $Au = \lambda u \in \mathcal{P}_\lambda$. Let $\mathcal{X}$ be A-invariant subspace that is spanned with the columns of matrix X. Then, a unique matrix B exists such that $AX = XB$. Due to this similarity with the one-dimensional case, we refer to (B, X) as an eigenpair of A. Furthermore, for each eigenpair (λ, y), $By = \lambda y$ implies that $A(Xy) = \lambda(Xy)$ and (λ, Xy) is the eigenpair of A. Thus, if X has a full column rank then $AX = XB$ implies that $\Lambda(B) \subseteq \Lambda(A)$. Additionally, if X is square and non-singular, then $\Lambda(A) = \Lambda(B)$ and A and $B = X^{-1}AX$ are similar. The following lemma describes the decoupling of the spectrum of a given matrix into smaller eigenproblems.

Theorem 2.3.1 *Suppose that* $A, Q \in \mathbb{R}^{n\times n}$*,* A *is symmetric, and* Q *orthogonal matrix partitioned as* $Q = [Q_1\, Q_2]$ *with* $Q_1 \in \mathbb{R}^{n\times r}$*. If* $\text{span}(Q_1)$ *is an* A*-invariant subspace, then*

$$Q^T AQ = D = \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix},$$

with $D_1 \in \mathbb{R}^{r\times r}$ *and* $D_2 \in \mathbb{R}^{(n-r)\times(n-r)}$ *and* $\Lambda(A) = \Lambda(D_1) \cup \Lambda(D_1)$

Proof: The proof can be found in [20] Lemma 8.1.9.

The theorem states that any given symmetric matrix A can be decoupled by applying orthogonal transformations into smaller subproblems and that the spectrum of A is equal to the union of the their spectra. The next lemma connects the invariant subspaces and the eigenvalue decoupling.

Lemma 2.3.2 *If* $A \in \mathbb{R}^{n\times n}$*,* $B \in \mathbb{R}^{p\times p}$*, and* $X \in \mathbb{R}^{n\times p}$ *satisfy*

$$AX = XB, \qquad \operatorname{rank}(X) = p, \tag{2.9}$$

then there exists an orthogonal matrix $Q \in \mathbb{R}^{n\times n}$ *such that*

$$Q^T AQ = T = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix}, \tag{2.10}$$

where $T_{11} \in \mathbb{R}^{p\times p}$ *and* $\Lambda(T_{11}) = \Lambda(A) \cap \Lambda(B)$*.*

Proof: The proof can be found in [20], Lemma 7.1.2.

The conclusion of Lemma 2.3.2 is that if we have an orthonormal basis X of A-invariant subspace $\mathcal{X}$ then we can reduce matrix A to a block triangular form by using similarity transformations. If $\mathcal{X}$ is an A-invariant subspace then Equation (2.9) is satisfied and matrix Y is chosen such that $Q = [X, Y]$ is orthogonal.

2.4 Krylov subspaces

An important class of methods for solving the eigenvalue problems are based on Krylov subspaces. Today, these methods are mainly used when only a few eigenvalues of large sparse matrices are required or in the solution of large systems of linear equations. Generally, the methods start with an arbitrary vector b and compute vector Ab in the first step. In the following steps, the resulting vector is consecutively multiplied with A from the left, producing a sequence of vectors $\{b, Ab, A^2 b, \ldots\}$. All methods that use that same simple principle are referred to as the Krylov subspace methods.

A Krylov subspace is defined as follows.

Definition (Krylov subspace): Let $A \in \mathbb{R}^{n\times n}$ and $u \in \mathbb{R}^n$ with $u \neq 0$, then

$$K_m(A, u) = [u\ Au\ A^2 u\ \cdots\ A^{m-1} u]$$

is the m^{th} Krylov matrix associated with A and u. The corresponding subspace

$$\mathcal{K}_m(A, u) \equiv \operatorname{span}\{u, Au, A^2 u, \ldots, A^{m-1} u\}$$

is the m^{th} Krylov subspace associated with A and u. If it is clear which matrix is concerned, we can omit the matrix name and write $\mathcal{K}_m(u)$.

Before we start with the properties of the Krylov subspaces, let's recall that the polynomial p applied on vector with respect to a matrix A is

$$p(A)u = (\alpha_0 I + \alpha_1 A + \alpha_2 A^2 + \ldots + \alpha_{k-1} A^{k-1})u,$$

where k is the degree of the polynomial. Furthermore, the minimal polynomial of vector u with respect to A is a nonzero monic polynomial p of the lowest degree such that $p(A)u = 0$.

The following lemmas present some basic properties of Krylov subspaces. The proofs are skipped as some of them are trivial while the more complex ones can be found in Saad's book [39].

Lemma 2.4.1 *Let $A \in \mathbb{R}^{n\times n}$ and $u \in \mathbb{R}^n$ and $u \neq 0$, then*

1. *The Krylov subspace $\mathcal{K}_k(u)$ is the subspace of all vectors $x \in \mathbb{R}^n$ which can be expressed as $x = p(A)u$, where p is a polynomial of degree not exceeding $k-1$.*
2. *$\mathcal{K}_k(u) \subseteq \mathcal{K}_{k+1}(u)$, for any polynomial of degree k.*
3. *$A\mathcal{K}_k(u) \subseteq \mathcal{K}_{k+1}(u)$.*
4. *$\mathcal{K}_l(u) = \mathcal{K}_{l+1}(u)$, implies that $\mathcal{K}_l(u) = \mathcal{K}_k(u)$ for all $k \geq l$.*

The last statement in Lemma 2.4.1 implies, because of the $A\mathcal{K}_k(u) \subseteq \mathcal{K}_{k+1}(u)$, that $\mathcal{K}_k(u)$ is invariant subspace of A. If μ is the degree of minimal polynomial of u, then $\mathcal{K}_\mu$ is A-invariant and $\mathcal{K}_k = \mathcal{K}_\mu$ for all $k \geq \mu$.

Lemma 2.4.2 *The Krylov subspace $\mathcal{K}_m$ is of dimension m if and only if the degree of minimal polynomial of u with respect to A is larger than $m-1$.*

2.5 Overview of perturbation theory

Most of the eigenproblems come from real applications that arise from different domains in which data are collected and generated before some computation is applied. During the process of collecting, storing, and computing with the experimental data various errors are introduced at different stages of the research process. Generally, we can divide errors in three categories based on how and where they occurred. In the first group are those errors introduced in the measurement and data generation process with the input data collected and generated with different imprecise instruments. The errors of the second group are introduced during the computational process due to inaccurate algorithms. The errors from the last group occur when storing the data as floating-point numbers on computer storage which introduces truncation and rounding errors due to the finite arithmetic precision. Therefore, the question we ask ourselves is how accurate is the solution obtained from our experiments, or in other words, how good is the solution compared to exact one. To answer that question we give a short overview of error analysis.

Assume that $\hat{y}$ is an approximated solution of $y = f(x)$ computed in finite-precision arithmetic with precision ϵ, where f is a function and x an input data. The distance between the computed and the exact solution, $\Delta y = y - \hat{y}$, is called the *forward error*. However, the forward error is hard to estimate due to the inability to have an exact solution and because it usually leads to overestimate errors. Instead of focusing on errors in $\hat{y}$ we can analyze for which problem $\hat{x}$, $\hat{y} = f(\hat{x})$ is the exact solution. In other words, for what small perturbation Δx we have $\hat{y} = f(x + \Delta x)$. The quantity $\Delta x = x - \hat{x}$ is called the *backward error*. Figure 2.1 illustrates this concept.

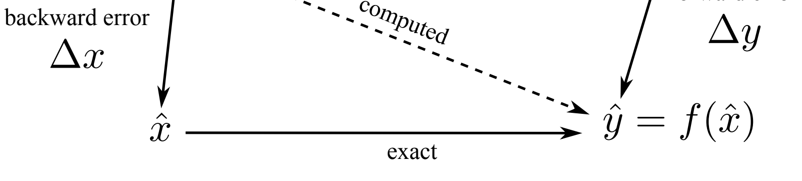


Figure 2.1: Backward and forward errors for $y = f(x)$.

Based on the error analysis, the stability of algorithm is defined as follows. An algorithm is said to be forward stable if:

$$\frac{\|y-\hat{y}\|}{\|y\|} \leq c\epsilon,$$

for some small constants c, where y and $\hat{y}$ are the exact and computed solutions, respectively. The algorithm is backward stable if:

$$\frac{\|x-\hat{x}\|}{\|x\|} \leq k\epsilon,$$

for some small constant k, where x is the original problem and $\hat{x}$ perturbed problem. The difference between forward and backward error analysis is in the focus. While forward analysis is concerned with the quality of the method output, backward analysis looks at the problem being solved. For example, if a certain problem is unstable, the forward error analysis will blame the method itself since it produces inaccurate results. On the other hand, the backward analysis says that the instability occurs due to the error (Δx) in the input data.

To demonstrate the influence of small perturbation in the matrix entries to the quality of the final solution, let's observe this simple example:

Example Let the matrix A and E be as follows:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad E = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \varepsilon & 0 & 0 & 0 \end{bmatrix},$$

where A is the exact matrix and E represents a small perturbation (error) matrix introduced into matrix A. The matrix $A+E$ is called perturbed matrix. We can set $\varepsilon = 10^{-16}$ to the machine precision which is the usual size of an error introduced when storing data using a finite-precision arithmetic. Matrix A has four eigenvalues all equal to zero, while the perturbed matrix $A+E$ has four different eigenvalues. Thus, we can conclude that the perturbation of size $\|E\| = \varepsilon$ introduced an error of size $\varepsilon^{1/4}$ that is 12 orders of magnitude greater than the exact solutions. Furthermore, in this example, the introduced error even changed the properties of the matrix such that $A+E$ has become diagonalizable.

Definition (Closed disc): For a square complex matrix A, a closed disc $D_i \subseteq \mathbb{R}$ centered at the element a_{ii} is defined as

$$D_i = \{z \in \mathbb{R}^n : |z - a_{ii}| \leq d_i\}$$

where

$$d_i = \sum_{j=1, j\neq i}^{n} |a_{ij}|,$$

is the sum of absolute values of non-diagonal elements of the i-th row. The closed disc D_i is called Gershgorin disc.

Theorem 2.5.1 (Gershgorin) *Let $A \in \mathbb{R}^{n\times n}$, then all eigenvalues of A are located in the union of n Gershgorin discs*

$$\sigma(A) \subseteq \bigcup_{i=1}^{n} D_i, \tag{2.11}$$

where D_i is the closed disc centered at the element A_{ii}.

Furthermore, if a union of k discs, with $k < n$, forms a connected region that is disjoint from all the remaining $n-k$ discs, then there are precisely k eigenvalues of A in this region.

Proof: The proof can be found in [40], Theorem 6.1.1.

From the theorem it is clear that every eigenvalue of A lies within at least one Gershgorin disc. The consequence is that each eigenvalue is bounded and can be roughly approximated. For example, matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & -2 \end{bmatrix}$$

has eigenvalues $\{-2.3028, 1.3028\}$ and the Gershgorin's discs are $[-3,-1]$ and $[0,2]$. The Gershgorin theorems gives very rough estimations of eigenvalues.

On the other hand, if we apply the theorem on a matrix whose off-diagonal entries have small norms, then each radius d_i around diagonal element would be small and the bounds would be much closer to the eigenvalue. Since matrix $S^{-1}AS$ is similar to A, whenever S is invertible, we can apply Gershgorin theorem on the matrix $S^{-1}AS$. Because of the similarity, matrices $S^{-1}AS$ and A have the same eigenvalues. Furthermore, S can be chosen so that off-diagonal entries have small norms. In the spectral decomposition, Theorem (2.2.1), if we set $S = Q$ then the off-diagonal entries become zero.

The algorithms, like Jacobi [41], proposed by Veselić, apply a sequence of similarity transformation to reduce a given matrix to diagonal form by annihilating the off-diagonal entries. The iterative process stops when the bounds (d_i) are smaller than a given value (e.g. machine precision). At the end of the reduction process the eigenvalue approximations appear on the diagonal and all off-diagonal entries are close to zero (machine precision).

For the next theorem we need to define normal matrices. A matrix $A \in \mathbb{C}^{n\times n}$ is normal if $AA^* = A^*A$, where A^* denotes the conjugate-transpose. In the real case, $A \in \mathbb{R}^{n\times n}$ is normal if $AA^T = A^TA$. This definition is analogous to state that there exists a unitary (orthogonal) matrix U (Q) such that UAU^* (QAQ^T) is diagonal.

Corollary 2.5.2 *Matrix $A \in \mathbb{R}^{n\times n}$ is normal if and only if every matrix orthogonally similar to A is normal.*

Proof: Suppose A is normal and $B = QAQ^T$ where Q is orthogonal. Then

$$BB^T = Q^TAQQ^TA^TQ = Q^TAA^TQ,$$

and since A is normal

$$BB^T = Q^TA^TAQ = Q^TA^TQQ^TAQ = B^TB.$$

On the other side, if B is normal then $BB^T = B^TB$ gives $Q^TAA^TQ = Q^TA^TAQ$. By multiplying with Q^{-1} from the left and Q^{-1} from the right, we obtain that $AA^T = A^TA$. ■

The corollary states that every normal matrix A is diagonalizable, i.e. there exists an orthogonal matrix Q such that $D = QAQ^T$ where D is diagonal. Furthermore, every symmetric and every orthogonal matrix is also normal. However, other direction is not applied since not all normal matrices are symmetric or orthogonal. The following theorem gives a rough approximation of the eigenvalues of the perturbed matrix $A+E$ if A and E are normal.

Theorem 2.5.3 (Wielandt-Hoffman) *Let $A,E \in \mathbb{R}^{n\times n}$, A and $A+E$ normal, and let $\lambda_1,\lambda_2,\ldots,\lambda_n$ denote the eigenvalues of A and $\hat{\lambda_1},\hat{\lambda_2},\ldots,\hat{\lambda_n}$ the eigenvalues of $A+E$ in any given order. Then there exists a permutation σ such that*

$$\sum_{i=1}^{n}(\hat{\lambda}_{\sigma(i)} - \lambda_i)^2 \leq \|E\|_F^2. \tag{2.12}$$

Proof: The proof can be found in [42].

This theorem does not specify which permutation σ will satisfy the inequality nor how to choose this permutation. Not every permutation will fulfill the inequality. If the matrices are symmetric then the inequality holds for the natural (decreasing or increasing) order of eigenvalues.

Corollary 2.5.4 *If $A,E \in \mathbb{R}^{n\times n}$, A symmetric and $A+E$ normal with $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ eigenvalues of A, and $\hat{\lambda_1} \leq \hat{\lambda_2} \leq \ldots \leq \hat{\lambda_n}$ eigenvalues of $A+E$ then*

$$\sum_{i=1}^{n}(\hat{\lambda}_i - \lambda_i)^2 \leq \|E\|_2^2. \tag{2.13}$$

This corollary states that if a symmetric matrix A is perturbed by a symmetric matrix E then its eigenvalues do not change by more than $\|E\|$.

The following two corollaries define the maximum possible difference between the computed (approximate) eigenvalues of perturbed problem $A+E$ and the exact solution of A.

Corollary 2.5.5 *If $A,E \in \mathbb{R}^{n\times n}$ are symmetric, then*

$$\lambda_k(A) + \lambda_n(E) \leq \lambda_k(A+E) \leq \lambda_k(A) + \lambda_1(E),$$

for $k = 1,\ldots,n$.

The corollary states that the obtained eigenvalue $\lambda_k(A+E)$ of the perturbed matrix $A+E$ differ from the $\lambda_k(A)$ for no more than $\lambda_1(E) - \lambda_n(E)$. The next corollary defines the upper bound of the absolute error.

Corollary 2.5.6 *If $A,E \in \mathbb{R}^{n\times n}$ are symmetric, then*

$$\|\lambda_k(A+E) - \lambda_k(A)\| \leq \|E\|_2$$

for $k = 1,\ldots,n$.

Chapter 3

GPU-based Hybrid Computing Architectures

The idea to design hybrid architectures came from the need to fulfill the growing demands of modern scientific problems that generate enormous amounts of data, for which large processing power is required. The traditional computing architectures that consist of only one type of processing units, e.g. multi-core CPU-based system, are usually not best suited for all types of high demanding applications. Therefore, various computational units are joined into one system forming a hybrid architecture that can take advantage of all its computational units.

One of the hottest topic in high-performance computing are GPU-based hybrid systems which are usually made of one or more traditional multi-core processors and at least one general-purpose GPU device. In such architectures, the GPU devices are utilized as co-processors or accelerators for processing the compute-bounded applications because of their unique architecture. The popularity of GPU-based hybrid systems started with the release of the NVIDIA Compute Unified Device Architecture (CUDA) and the extensions of industry-standard programming languages, such as C, C++ and Fortran, which made the GPUs easier to program, allowing developers to exploit the computational power of modern GPU devices.

The chapter is structured as follows. The evolution of hybrid architectures is discussed in Section 3.1 from the first computing systems till the most recent computing architectures. In addition, this section explains the different levels of parallelism and memory hierarchy and how they are exploited for high performance of hybrid architectures. Section 3.2 gives an overview of GPU architectures as well as the GPGPU programming model. Finally, Section 3.3 details the architecture design and the memory hierarchy of GPU-based hybrid systems.

3.1 Hybrid architectures

3.1.1 The evolution of computing architectures

The breakthrough of the modern science was always limited by the availability of the computational power. The first problem that was solved, on what we now regard as modern computer, was computing artillery firing tables for the U.S. Army. The calculation was done on the first digital computer finished in 1946, ENIAC. Although the first digital computer was the Atanasoff-Berry [43] computer, constructed in 1942, this was never fully operational. The first computers were designed to use thousands of vacuum tubes and relays, occupied a lot of space and had a very modest performance, almost five orders of magnitude slower than the average modern computing system. The first major breakthrough in computer science happened in 1947, with the invention of digital transistor for which the inventors Bardeen, Brattain and Shockley were awarded the Nobel prize in physics in 1957. This invention drastically reduced space requirements and increased the speed of logic gates, making computers smaller and more energy efficient. The invention of the integrated circuit in 1958 revolutionized the world of electronics as this further reduced space and power requirements. It also provides the proportional growth of the computational power. The integrated circuit or, as currently referred to *chip*, is today used in almost all electronic equipments and presents the base for modern processing computing units (CPUs).

Following these developments, Gordon Moore postulated his famous law [44] in 1965. He predicted that the number of transistors that can be placed on a single chip with an affordable cost would double every two years. This trend is still valid and the law is applicable on modern processors. Since then, the performance of processors was increased by adding more transistors on a single chip that resulted in proportional growth of the processor clock speed (frequency), illustrated in Figure 3.1. In the following decades, especially during 1990s, performance improvement was tightly connected to the increase of processor frequency. The application execution time decreased with each new version of processor and no modifications of the code were required.

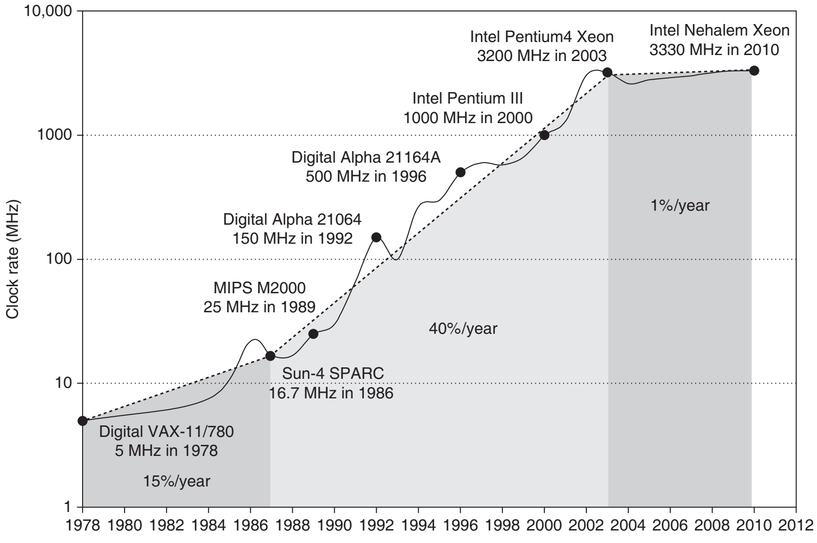


Figure 3.1: Growth in clock rate of microprocessors since 1978.*

The trend of scaling the single CPU performance reached its peak in 2004 and since then it is almost flat due to the technological physical constraints. The key constraint was power density or as more often referred to the *power wall* [45]. The power wall is reaching the maximum power dissipation of air-cooled chips that prevents adding more transistors on a chip. That results in stopping or even reversing the increase of the processor performance.

In 2004, all major processor vendors decided to abandon further development of single-core high-performance CPUs and turned towards on the development of multi-core CPU. The main difference from the previous architecture design was the integration of two or

---

*The figure is taken from [46], Chapter 1 more independent computational units, called *cores*, on the same die (chip), each capable of performing different task simultaneously. The multi-core architecture launched a new era in which the increase of performance lies in exploiting the parallelism of independent computational units rather than CPU frequency. Because of the new processor design, the existing sequential applications were not capable of exploiting the parallelism in multi-core processors without changing the application structure. That started the evolution of parallel programming models (e.g. OpenMP) and called for the re-implementation of the existing programs and computational libraries (e.g. LAPACK) to exploit the parallelism and increase performance in novel architectures.

The latest step in the evolution of computing architectures are hybrid architectures. The idea for the hybrid architectures came as a response to overcome limitations in high-performance computing of traditional multi-core CPU-based architectures. The multi-core CPUs are designed to efficiently operate with the complex program flows resulting in spending most of their clocks (computational power) on performing non-computational tasks. In addition, because of the CPU memory design, they are not highly efficient for applications that require high memory throughput. Hybrid or heterogeneous systems refer to computing systems that comprise different types of computational units, joined together to maximize the performance. The term heterogeneous refers to various computational systems, such as distributed systems, that consist of non-uniform (heterogeneous) computing nodes connected through a network (e.g. grid and cluster systems) as well as single machines that encompass different computational units connected via a system bus.

A single-node or single hybrid system encompasses general-purpose processors, usually one or more classic CPU processors like Intel Xeon or AMD Opteron, special-purpose processors (i.e. digital signal processor or graphics processing unit) that serve as co-processors and/or accelerators (e.g. field-programmable gateway arrays). The hybrid architecture, assembled of different computational units, has changed the traditional processor design path and encouraged the evolution of new programming models and paradigms thus opening new challenges and opportunities in high-performance computing. One of the hottest topics in HPC computing are the GPU-based hybrid architectures that deliver very high performance by putting together general-purpose multi-core CPUs and specific-purpose GPU processors for computationally and data-throughput intensive applications.

3.1.2 Parallel architectures and parallelism

Nowadays, parallelism is ubiquitous in computing systems. It is exploited at different levels and is the main driving force when seeking for computational performance. Basically, parallelism can be divided into two main types:

- Data-Level parallelism (DLP) that deals with a large amount of data that can be operated in the same time,
- Task-Level parallelism (TLP) exploits independent tasks that can be executed in parallel.

Parallel architectures such as vector processors (superscalar processors) or general-purpose GPUs, usually called the data-parallel architectures, are designed so that they can efficiently exploit the parallelism that appears in data processing. On the other hand, multi-core processors are more efficient in processing task-parallel applications. The way how the computing architectures support these two types of parallelism was first observed in 1966 by Michael Flynn [47]. He found a simple classification of computer systems based upon the number of concurrent instructions and how they operate on data, presented in

Table 3.1.

Stream type	**Single instruction**	**Multiple instruction**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Table 3.1: Flynn's taxonomy.

Traditional sequential processors, capable of performing a single instruction on a single data at the time, are considered a SISD architecture. The main characteristic of SIMD architectures is that they can perform one instruction on many data at the same time and are referred as data-parallel architectures. Two examples of SIMD architectures are vector and array processors. Most general-purpose parallel computers, such as distributed and shared memory systems (e.g. clusters, MPPs and data centers) as well as traditional multi-core processors, are MIMD architectures capable of performing multiple tasks on many data in parallel. Finally, the MISD type architecture does not exist but rounds up the classification.

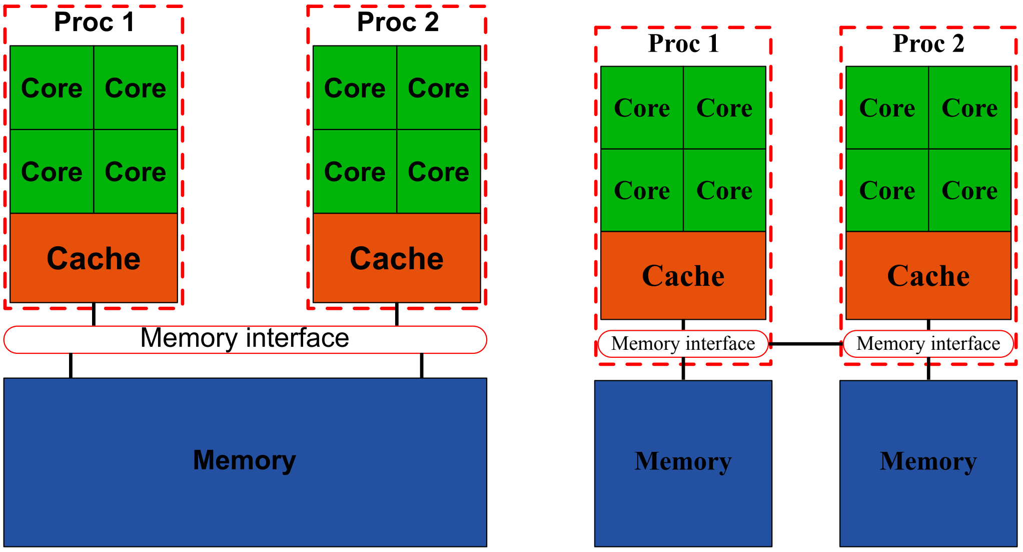
Today, Flynn's taxonomy is considered obsolete because the majority of modern computing architectures are a mix of different architectures and it is hard to make a clear distinction between them. Although Flynn's taxonomy is depreciated nowadays, it is still used as a basic architecture classification.

A more recent approach in the classification of computational hardware, on the processors level, is based on the number and the capability of computational units. Therefore, we can distinguish two types: multi-core processors and many-core processors. Recent multi-core processors, such as Intel Xeon "Ivy-Bridge" or i7 processor families, consist of up to 8 independent computational units that can perform multiple tasks in parallel on multiple data (MIMD type of architecture). Multi-core processors are usually referred to as tasks-parallel architectures because they are more efficient in exploiting the parallelism between the tasks than between the data. Many-core processors are the processors that consist of many light-weight processors (usually more than several dozens) and are capable of working together on the same tasks or instruction set on many data in parallel. However, light-weight processors are usually not capable of performing different tasks in parallel. The examples of many-core processors are general-purpose GPUs and Intel Xeon Phi co-processors. Further discussion on parallelism of GPUs, as the main example of the many-core architectures, is given in Section 3.2.

The parallelism on multi-core processors can be exploited at different hardware levels. The top-most is a *chip* or processor, that refers to a single physical package. A chip contains one or more *cores* that denote physical processing cores. Parallelism can be further exploited in a system that has two or more processors or on a core level by using multiple hardware threads and concurrent instructions. In this thesis, the research was done on single computing node, thus only the node-level parallelism will be presented next. The parallelism on single computing node can be exploited at the following hardware levels:

- Multi-processor (chip).
- Multi-core.
- Thread.
- Instruction.

A **Multi-processor** system or symmetric multi-processor system is a single computer with several homogeneous physical processors (chips), centralized shared memory called the *main memory*, and an operating system (OS). A closely related term is *symmetric multiprocessing* (SMP) that refers to the parallelism where two or more homogeneous processors share the main memory, controlled by a single OS, working on different programs/tasks and different data concurrently. In SMP systems, each processor can execute any task, no matter where in memory data are located. The workload efficiency can be improved by easily moving tasks between the processors. Figure 3.2(a) illustrates the common SMP system where the processors share a system bus.



(a) Dual-socket Intel Core2 architecture (UMA).

(b) Dual-socket Intel Westmere architecture (NUMA).

Figure 3.2: Example of two types of SMP architectures. A uniform memory architecture where all processors share the common resources (i.e. main memory) uniformly (a) and a non-uniform memory architecture where each processor has its local main memory (b).

Traditionally, the SMP is usually referred to as Uniform Memory Access (UMA) architecture because all the processors share the physical memory uniformly. The advantages are the common memory address space and memory location access time that is independent of which processor makes the request. To reduce the system bus traffic, each processor has an associated cache memory that reduces the number of data accesses to the main memory. However, the cache is usually very small and the shared system bus remains the main bottleneck.

An alternative is the Non-Uniform Memory Access (NUMA) architecture; see Figure 3.2(b). In a NUMA architecture the memory is physically divided between processors and the access time depends on data location. The access time for each processor to its local memory is faster than to the memory of other processors. With this architecture, the memory throughput can be significantly improved as long as data are kept in the processor's local memory.

At the **multi-core** level parallelism is exploited on a single processor/chip within the computational cores. The communication between cores is less expensive than between processors as the cores share common resources such as on-chip cache memory; Figure 3.3. Each core executes its own instructions (add/move data, processing, and branching) independently of the other cores in the same processor enabling concurrent execution of different programs/tasks or parallel applications.

The performance gain of parallel algorithms on multi-core processors mostly depends on the algorithms and their implementation. The implementation of such algorithms depends on the processor architecture, specifically on the amount cache memory and if the cache (e.g. L2) is private (Figure 3.3, left) or shared between cores (Figure 3.3, right). If the required data is found in cache, the access time is faster due to the lower cache latency. On the other hand, if the data are not found in given cache level, e.g. L1, they are fetched from the first lower memory level, L2, resulting in the core waiting for data. The access time is even worse if data are fetched from the off-chip memory (e.g. the main memory). Thus, the best performance is achieved for applications whose data can fit into the cache and for so-called embarrassingly parallel applications (applications that can be naturally split into a number of independent tasks).

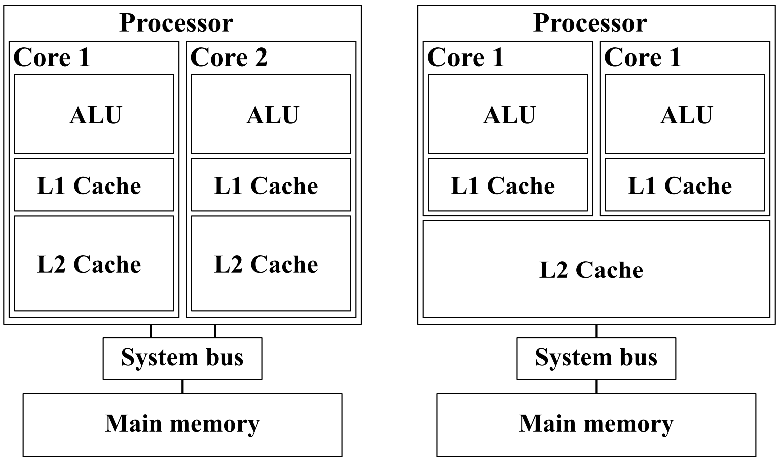


Figure 3.3: The architecture of multi-core processor, left) separate L2 cache, right) shared L2 cache.

The parallelism at **thread level** or multi-threading is a model of parallelism where multiple threads are executed on a single core. The aim of multi-threading is to increase the utilization of a single core. If a thread experiences a lot of cache misses and is waiting for data to be fetched from off-chip memory, other threads can take advantage of the unused core which results in faster overall execution and decreases idle time. There are several types of multi-threading concepts such as *block multi-threading*, *interleaved*, and *simultaneous* multi-threading that applies to superscalar processors. More information on thread-level parallelism can be found in [46].

The **instruction level** parallelism (ILP) is an ability to overlap the execution of multiple instructions of the same computational unit. The ILP tries to improve parallelism between the instructions. A good example is within instructions in the loop. The dependences between instructions are examined and the independent instructions are processed in parallel. ILP parallelism is usually exploited within superscalar processors.

3.1.3 Memory hierarchy

Memory hierarchy has one of the most important roles in the overall computer performance. Peak performance of modern processors is worthless if the data are not available to the processing units. In such case, the processing is paused resulting in processor idling while waiting for data to be transferred from the memory. Because of memory latency, hundreds of computing cycles can be wasted in waiting for data and not performing any useful operation. This huge gap in the performance between the memory and the CPU occurred because the improvements of processor performance were much faster than the improvements in memory performance. This was the result of the archaic approach from the early days of computing, in which floating-point operations were considered expensive and retrieving data almost free. To mid 1990s the gap between processor and memory performance grew up to 50% [48] per year. In most recent systems, processor bandwidth is much higher than memory bandwidth, for example, the bandwidth of Intel i7 processors is up to 409.6 GB/s, compared to only 25 GB/s main memory bandwidth. In such systems the CPU idle time can easily be up to a few thousand CPU cycles.

To overcome the memory bottleneck, different memory levels were introduced between the CPU and the main memory to address both the bandwidth and the latency problem. The idea is to keep the requested data in fast, low latency memories as long as possible and thus exploiting data locality. The locality rapidly increases the access to recently used data and keeps the processing units busy. The ultimate solution would be that all memory is made as fast as possible, but due to technological and commercial constraints, faster memories are more expensive to produce, this is not feasible. The rule of thumb is, the faster the memory is, the smaller capacity it has. Therefore, memory is organized in a hierarchy as illustrated in Figure 3.4.

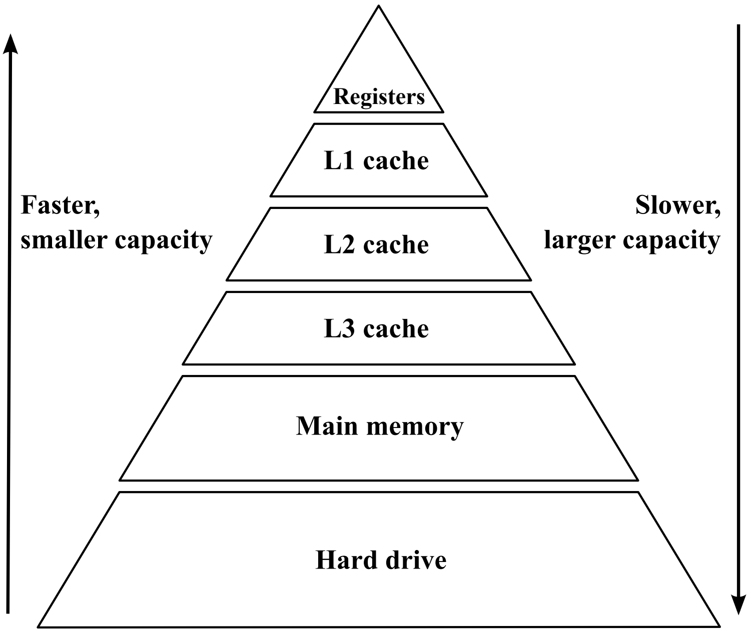


Figure 3.4: Memory hierarchy of standard single-node computing system. The most recent processors have additional L4 cache level.

The memory hierarchy is organized in a way that faster and smaller memories are placed closer to the computational units, while slower and larger memories are in the lower memory levels, farther from the computational units. With such organization of memory levels it is possible to alleviate the influence of the memory latency by keeping and working on data in the top-most memory levels while transferring other data from lower levels. Table 3.2 shows typical capacities and access times for different memory levels. Note that the access time changes from picoseconds for registers to milliseconds for hard drive, that is an increase by a factor of 10^9! Similarly the capacity ranges from

1 KB for registers to a few TB for hard drives. For example, on a high-end multi-core processors, like the Intel Xeon E5-2470, the L3 cache capacity is up to 25 MB.

Mem. level	**Access time**	**Capacity**
Registers	300 ps	1 KB
L1	1-2 ns	64 KB
L2	3-10 ns	256 KB
L3	10-20 ns	2-4 MB
Main Memory	50-100 ns	4-16 GB
Hard drive	5-10 ms	1-16 TB

Table 3.2: Typical capacities and access time for different memory levels (per core).

Two terms are closely connected to memory performance, *bandwidth* and *latency*. Memory bandwidth or throughput is the total amount of bytes per second transferred between two memory levels. Latency or response time is the time for a single access. In other words, memory latency is the delay time between the moment the request for a particular memory module (data) is sent and the moment the transfer is started.

The memory hierarchy is organized as follows. The computing core can operate only with the data that are in the top-most memory level, the registers. Without lose of generality we will assume that the top-most level is L1 cache. If data requested by the processors is not found in the L1 cache, it must be fetched from the first lower level in the hierarchy. If the data are also not there they are fetched from the next lower level and so on till the data are found. The total time needed to fetch the data is called *cache access time*, and if data are not in the top-most cache, a *miss*. If data are missed a *miss penalty* is paid, the extra time needed to fetch the data from the lower memory levels. The opposite of a cache miss is a cache hit, i.e. if the data are found in the cache. The total number of cache misses per application is called *miss rate*. The cache miss and hit rates depend on the implementation but also on how the cache organizes and stores data. Detailed information on cache organization can be found in [46], chapter 2.

To measure the cache performance, the average memory access time ($Access_{avg}$) is computed as:

$$Access_{avg} = Hit\,time + Miss\,rate \times Miss\,penalty.$$

If the miss rate is large, the total access time is bounded by the time required for miss penalty. With multiple cache levels it is possible to reduce miss penalties. For example, if two cache levels are present in the system, L1 and L2, then the average memory access time is calculated as follows:

$$Access_{avg} = Hit\,time_{L1} + Miss\,rate_{L1} \times (Hit\,time_{L2} + Miss\,rate_{L2} \times Miss\,penalty_{L2}).$$

The idea is that the L1 cache is small but fast enough to keep the pace with the processor clock cycle time, while the L2 (or even L3) cache is large enough to keep more data, and to decrease the number of access that would go to the main memory.

3.2 General–purpose GPU computing

In the last few years general-purpose GPU computing has become one of the hottest topics in high performance computing. That is because of the low-cost and very-high performance of the modern GPU cards compared to traditional CPUs. Indeed GPUs have become so popular that the second fastest supercomputer on Top500 [49] list, November 2013, is based on NVIDIA GPUs. Moreover, the top 10 supercomputers on the Green500 [50] list (the most energy efficient supercomputers) are based on GPUs. Nowadays, GPUs offer not only high-performance but are also more energy efficient than traditional processors. GPUs are SIMD architectures that can efficiently exploit data-level parallelism. The performance of GPUs comes from their unique hardware design that is equipped with a few thousand light-weight computational units.

Without loss of generality, this section gives a brief description of NVIDIA GPU architecture and the CUDA (Compute Unified Device Architecture) programming model [51] as well as reviews the features of modern graphic processors. The reason we do not include other GPU vendors is because NVIDIA technology is the most mature and provides various computing libraries that make programming much easier. For that reason, all the research in this thesis is conducted on NVIDIA graphic cards and by utilizing the CUDA-based computational libraries.

3.2.1 Architecture of modern GPUs

Traditionally, the main purpose of GPUs was to render computer graphics and the GPU architecture was adjusted to solve that specific type of problem. The work was done in a pipeline that consisted of four strictly pre-defined stages. The input for each stage was the output from the previous stage and, because of these dependencies, the stages were executed in sequential order. The four main stages were *vertex transformation*, *primitive assembly and rasterisation*, *fragment color and texture*, and the final stage *raster operations*. The older versions of GPUs had specialized execution units (shaders) for each stage while, in more recent GPUs, each stage can be executed by any execution unit. The most important computational units were vertex and pixel shaders. These traditional GPUs were very hard or even impossible to program due to the fixed design of the stages which prevented any modifications.

The first graphic cards that allowed some level of modification were released in 2001 by NVIDIA. These cards were the first that allowed the programmers to modify the stages of the pipeline. The programmable stages were the first step in what we know now as the GPU kernels, i.e. functions or parts of code executed on a GPU device. The revolution started in 2006 with the release of an NVIDIA GPU that had unified vertex and pixel processors (shaders) and in which all stages were performed by that unified processing unit. That novelty soon become obvious for general-purpose computing by offering a large number of computational units capable of solving computational-intensive problems in parallel.

An overview of GPU architecture

The graphics processing unit (GPU) is defined as a multi-threaded multi-processor composed of an array of simple computational cores with very high memory bandwidth. These features give GPUs a remarkable advantage in solving computational and data intensive applications over standard multi-core processors. As presented in Figure 3.5 a modern GPU device consists of a large number of light-weight computational units and small cache and control units. On the other hand, standard multi-core processors have a few computational units or cores (8–16 in the most recent processors) capable of solving complex tasks because of their large cache and control unit. Thus, the multi-core processor spends most of its transistors on flow and execution control as well as cache memory while the GPU is more oriented toward vast computation.

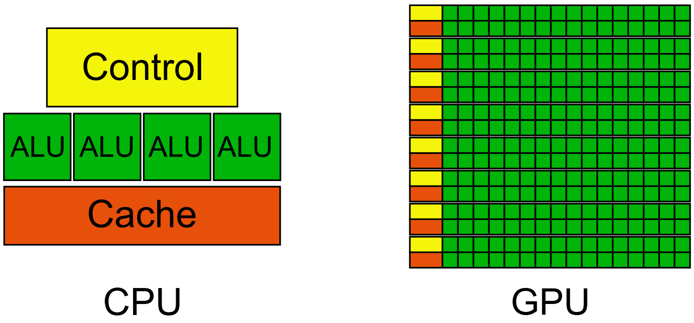


Figure 3.5: Conceptual difference between traditional multi-core processor and general-purpose graphic processor.

The elementary GPU computational unit is called Streaming Processor (SP) and it is capable of running a single thread. A set of SPs are grouped into a Streaming Multiprocessor (SM) which presents the main computational unit of the GPU. Each SM is responsible for the creation, execution, and destruction of threads assigned to its SPs. Figure 3.6 illustrates the NVIDIA Fermi architecture which consists of 14 to 16 streaming multiprocessors. Each Fermi SM has 32 SPs sharing the common on-the-SM memory that is also used as L1 cache memory. In the latest NVIDIA GPUs architecture (e.g. Kepler) the SMs are organized into groups of four in a so-called *graphic processor cluster* (GPC), each totaling 192 SPs.

As a main computational unit, the SM is responsible for managing the thread execution on its SPs. In order to manage a large number of threads, the SM employs Single-Instruction Multiple-Thread (SIMT) architecture that applies one instruction across multiple threads in parallel. Each SM manages threads divided into special groups called *warps*. A warp is a group of 32 threads that concurrently execute the same instruction on multiple data. In other words, each SM executes one instruction across a warp (32 threads) at the time.

For example, NVIDIA Tesla C2050 consists of 448 SPs divided into 14 SMs. Each SM has two additional SFUs (Special Functional Unit - for transcendental functions such as sine and cosine), instruction and constant caches, and 48 Kbytes of on-chip shared memory. On a single SM, up to 1536 concurrent threads can be run (organized into warps) with practically no scheduling overhead. The scheduling overhead is negligible because all threads share the SM's local memory that is equally shared among all threads. Furthermore, the SM's instruction fetch and issue unit is also shared across threads and to start a new set of threads (a warp) on a SM no additional copying to instruction and memory registers is required. This architectural design reduces the unnecessary overhead that appears due to the expensive thread context switch that is applied in the traditional processors.

The streaming processors (SMs) are light-weight computational units or cores with both integer and floating-point arithmetic units. Each SP is capable of running multiple threads simultaneously, which share the common SM's register file. To fully utilize all the available cores on each SM, a fine-grained parallelism is required with a very large number of threads. The number of threads usually goes far beyond the number of physical cores on all SMs. With the GPU design that relies on a large number of concurrent threads, it is possible to maximally alleviate the GPU memory latency. Thus, while one warp is

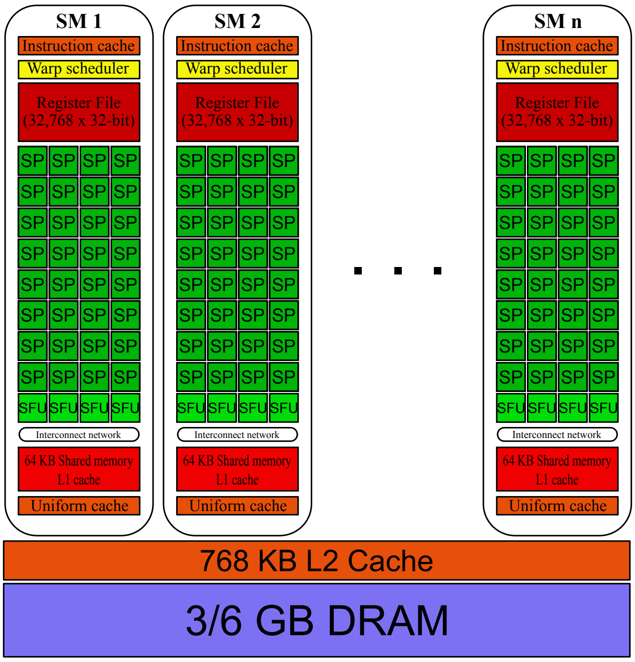


Figure 3.6: NVIDIA Fermi architecture

executed, a second one is prepared, i.e. memory transfers are performed in the background.

Modern GPUs are not only characterized by their vast multi-threaded parallelism but also by their specific memory hierarchy and very high memory bandwidth between the memory and the processing units. Modern graphics cards consist of three memory levels. The fastest but the smallest memories are the 32-bit registers dedicated to each SP. The registers of one SP are divided among the active threads executed on that SP. The second level encompasses the shared memory, constant cache, and the texture memory placed on the SMs. The shared memory is accessible to all threads of the same SM. The cache and texture memories are read-only memory and used mostly for graphic computations. The last level is global or device memory. It is available to all threads that participate in the computation on all SMs. For example, the TESLA C2050 graphic card has 3 GB of global memory. Each SM has 48 Kbyte of shared memory and 32768 registers, resulting in 1024 registers attached to each SP. Furthermore, C2050 has 384-bit memory bus width providing very high data throughput.

3.2.2 NVIDIA programming model

With the release of the first general-purpose GPU card in 2006, NVIDIA introduced CUDA (Compute Unified Device Architecture) [52], a general-purpose parallel computing programming model and platform. CUDA allows developers to use GPUs as general-purpose processing units. It provides CUDA-accelerated libraries, compiler directives, and extension for standard programming languages C, C++, and Fortran.

Without loss of generality, the NVIDIA CUDA terminology will be used in the following text. The GPU device attached to a traditional multi-core system as a co-processor (accelerator) is called *device*, while the multi-core system is called *host*. The host and device have their own separate memory spaces, the main memory and the GPU memory (DRAM memory), called *host memory* and *device memory*, respectively. For more detailed information on the CUDA programming model, the readers are encouraged to consult the "CUDA C Programming Guide" [53].

Figure 3.7 illustrates a typical CUDA execution flow. The main program is run on the host system that also controls the parts of code to be executed on the device as well as data management. The function called from the host that is executed on the device is called *kernel*. Moreover, because of separate memory spaces, explicit device memory allocation and deallocation as well as data transfers between host and device have to be performed, illustrated in Figure 3.7 with yellow boxes.

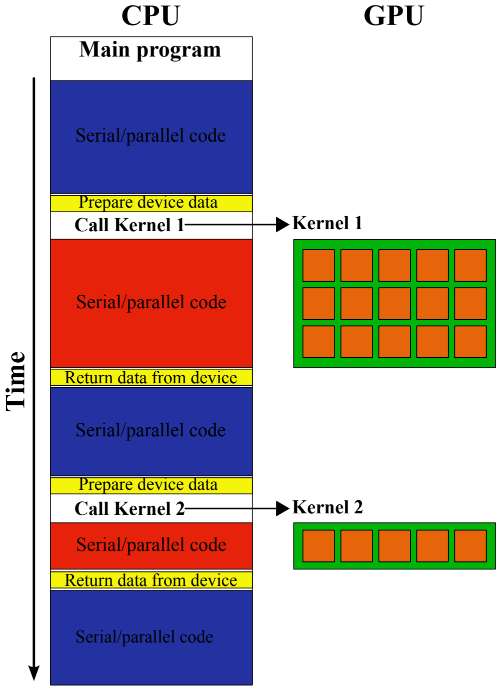


Figure 3.7: The program work-flow in the host-device programming model

The kernel call can also be made asynchronously, i.e. the program control is immediately returned to the host, thus allowing concurrent execution. In Figure 3.7, the red boxes illustrate the parts of the host code that can be run in parallel with the GPU execution. In addition, starting with the NVIDIA Fermi architecture, two or more kernels can be executed concurrently on the same device but on different streaming multiprocessors thus increasing the utilization of the GPU device. This ability adds more parallelism and improves the utilization of GPU device.

The kernel is executed in parallel by a number of CUDA threads defined by programmer as a kernel input parameter. All threads engaged in the kernel are divided into blocks, called *thread blocks*. Each thread block is scheduled, in random order, to any streaming multiprocessor given the kernel execution to be independent on the architecture of a specific device, i.e. independent on a number of SMs. If a kernel is executed on the GPU with less streaming multiprocessors it will simply be executed in more time, or in less time if the GPU has more multiprocessors. The benefit of this approach is that the GPU kernels are extremely scalable to different GPU architectures, and they will automatically benefit from GPUs with a larger number of multiprocessors.

The organization of kernel execution into blocks of threads allows a problem to be divided into sub-problems where each sub-problem is processed by one thread block. Therefore, problems that can be sub-divided in this way, for example data-parallel applications such as linear algebra operations, can perfectly suit the GPU programming model.

A thread block can be one-dimensional, two-dimensional, or three-dimensional. Each thread in the block is identified by its unique one-, two- or three-dimensional thread index by which, each thread within block can be accessed uniformly. This presents a natural way to describe the computation on elements such as vector, matrix or volume.

The number of threads per block is limited by the design of the GPU multiprocessor and its limited memory resources that are shared among all cores. In modern GPUs, the maximum number of threads per block is 1024. Blocks are organized into a one-, two-, or three-dimensional *grid* of blocks, as in Figure 3.8. Usually, the size of the problem being processed on the GPU determines the number of blocks in a grid.

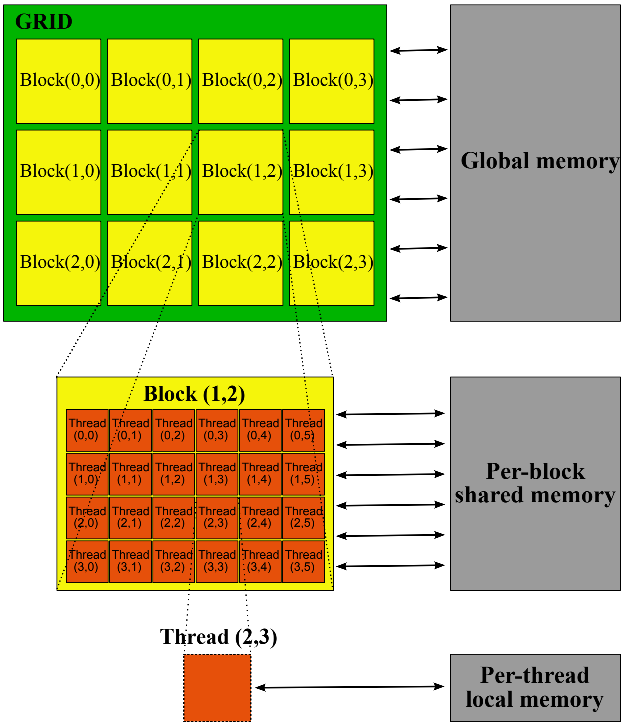


Figure 3.8: The organization of threads into blocks and grids and the memory hierarchy.

Each thread has its private memory; see Figure 3.8. The number of threads per block is limited by the number of registers per multiprocessor which are equally divided among all the threads of the block. Furthermore, all threads of the same block have access to the shared memory through which they can communicate and exchange data. The device global memory is visible to all the threads of the same kernel (all thread blocks). Through the global memory, threads from different blocks can communicate. Nevertheless, access time is proportional to the distance between the thread and the data. Therefore, the fastest read/write is to the own thread's local memory, then to the shared memory of the SM, and finally, the slowest is to the global memory. The best programming practice is to keep the local data required by each block of thread into the SM's shared memory and decrease the number of calls to global memory. For more information refer to "CUDA C Best Practices Guide" [51].

3.3 A GPU-based hybrid system

Hybrid systems based on the GPUs are among the most popular hybrid architectures. This is because of the GPU extreme performance that largely over paces the performance of recent multi-core processors. A GPU hybrid system is made of a traditional CPU-based system with one or more GPUs attached. In such system each GPU card is managed and operated by the host system. The host system (CPU) runs the operating system and manages data transfers between the system components. The GPU, in hybrid systems, works in conjunction with other components such as the CPU, chipset, system memory as well as other GPUs. This type of hybrid system will be referred as CPU-GPU hybrid system. An example of CPU-GPU hybrid system, based on an Intel platform, is illustrated in Figure 3.9.

Current CPU-GPU hybrid systems usually consist of one or more multi-core processors and one or more separated GPU devices connected to the host through a high speed bus, the PCIExpress (PCIe) bus. The PCIe, as shown later, has become the main bottleneck in achieving the maximum performance of GPU devices. The future trends aim at integrating the traditional processors with the graphic processors on the same die (chip), such as the Intel Haswell and the AMD Richland architectures. The goal is to override the bandwidth and latency bottleneck caused by PCIe, thus achieving faster communication between CPU and GPU processing units.

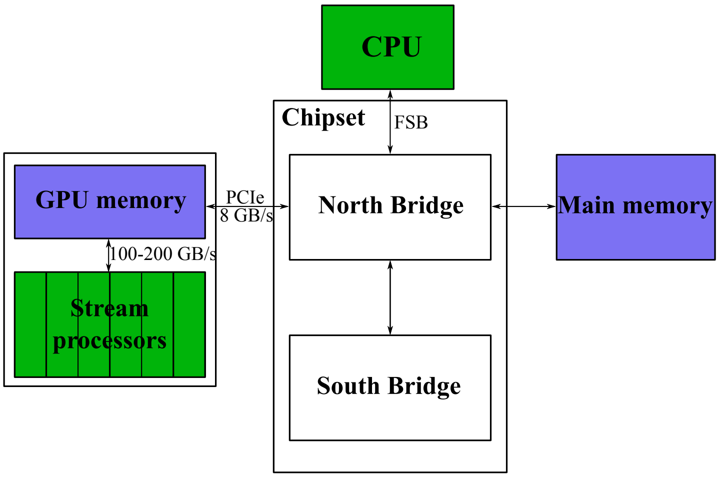


Figure 3.9: An overview of computational units, memories and interconnections in modern CPU-GPU hybrid system.

The GPU is connected directly to the host chipset (i.e. North Bridge memory controller) through PCIExpress bus, Figure 3.9, that provides the peak performance of 8 GB/s in both directions (for PCIx 2.0 standard). The most recent GPUs, such as NVIDIA GeForce TITAN, support the PCIExpress 3.0 standard, allowing up to 15.75 GB/s in both directions. If more than one GPUs is attached, the PCIe bus is shared between them resulting in a possible communication bottleneck. The GPU memory is GDDR5 (Graphics Double Data Rate, version 5) and its size varies between 3 and 6 GBytes. The total memory bandwidth of the top-most high-performance GPUs, such as the NVIDIA Tesla K40, is up to 288 GB/s. In contrast, the bandwidth of the main memory is an order of magnitude less than those of GPUs and varies between 8 and 15 GB/s. Because of very high GDDR5 memory bandwidth, the applications that require high bandwidth, such as graphic rendering and high-performance computations (e.g. dense linear algebra routines), are perfectly suitable for GPU devices.

As illustrated in Figure 3.9, the PCIe bus bandwidth is relatively small compared with GPU memory bandwidth and, as will be shown later, is proved to be a significant bottleneck for certain applications. In all GPU-based hybrid systems, the main issue is managing the data transfers between the main memory and the GPU memory. Transferring data from one device to another takes additional time and has to be reduced. Although GPUs allow asynchronous data copying and execution, this is not always possible to achieve. In that case, the transfer time can easily overpower the GPU execution time or force GPU streaming processors to idle while waiting for data.

The total transfer time is made of two components, latency and bandwidth, and is computed as follows:

$$T_t = T_L + T_B, \tag{3.1}$$

where T_L is the latency time and T_B is the time to transfer data. The latency time (T_L) is constant and does not depend on the amount of transferred data. On the other hand, T_B depends on the bandwidth (B) and data transferred (T), and is computed as $T_B = T/B$. It is obvious from (3.1) that the total transfer time can be reduced by sending a smaller number of larger blocks of data. For example, if a payload of n data is divided into four blocks and sent through PCIe to GPU memory, the total time is $4T_L$ compared with $1T_L$ if only one transfer is run.

The transfer time becomes even a larger problem if the amount of data increases. That especially comes to the fore when solving large-scale problem and/or problems when the amount of data exceeds the capacity of the GPU memory. This problem is similar to the cache miss problem, described in Section 3.1.3, when the cache is too small to store all the required data. If the data are not found in the memory, i.e. the cache miss occurred, the data are fetched from the first lower memory level. In the CPU-GPU communication, fetching the data is not managed by the system automatically but an explicit transfer has to be introduced in the code. As is the case with the cache memory, preserving data locality in the GPU memory increases the performance by reducing the number of transfers.

The memory hierarchy of a standard GPU-based hybrid system is illustrated in Figure 3.10. The hierarchy consists of the traditional memory levels, such as hard drive, main memory, and cache and the memory system of the attached GPUs. The lowest two levels, the hard drive and the main memory are shared by both CPU and GPU. Then the memory hierarchy splits into two directions, one towards the CPU and the other towards the GPU. A CPU has one, two, three or even four levels of cache memories and registers that are on-chip, while the GPU has global memory, shared memory/L1 cache, for each streaming multiprocessor (SM), and registers that are shared between all cores of the SM.

Taking into account that the GPU global memory has a huge bandwidth and small latency, the GPU memory can be looked at as a "cache" level memory. The idea is the same as with the traditional cache levels: preserve data locality and hide latency. In that context, the GPU memory serves as a component that reduces the cache misses and decreases the number of data accesses to the main memory. The term *cache miss* then refers to a moment when certain data, not stored in the GPU memory, are required; and *data access* is the time required to transfer the data from the main memory to the GPU. The main difference is that the GPU memory management is not done by the operating system, as for the traditional cache, but it is performed in the application run-time by explicitly invoking the data transfers at the certain point of the code.

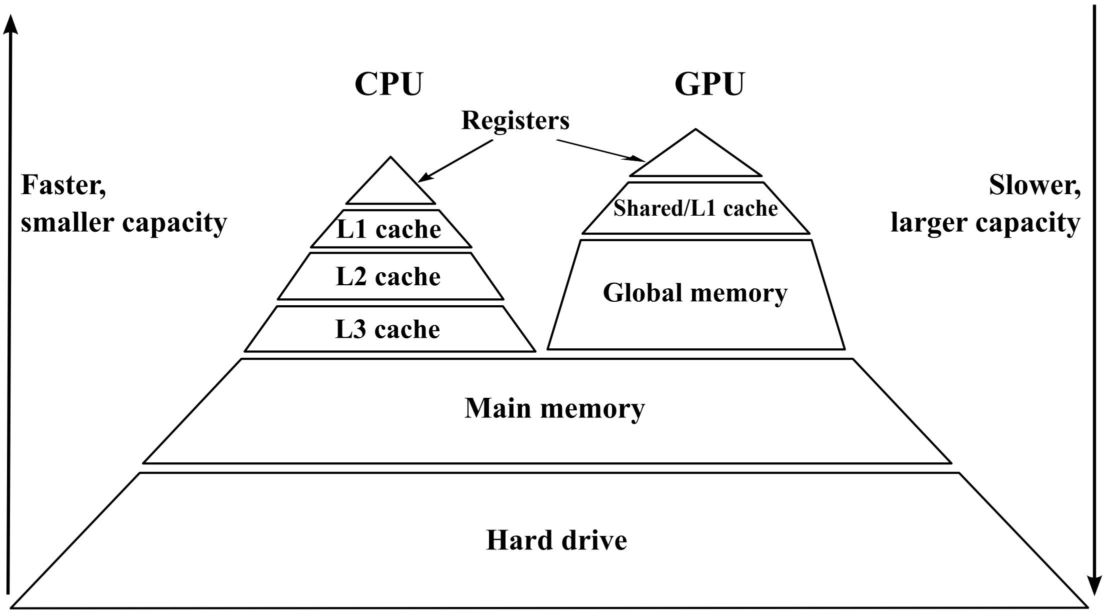


Figure 3.10: Multilevel memory hierarchy of a standard CPU-GPU hybrid architecture.

Chapter 4

Dense Symmetric Eigensolvers

The dense symmetric eigensolvers are algorithms that aim to compute the eigenvalues and corresponding eigenvectors of dense symmetric eigenproblems, i.e. the problems whose given matrix is symmetric and dense. However, in many cases, the eigenproblems appear in a form different from the standard eigenproblem presented in Equation (2.5) as, e.g., occurs with the generalized eigenproblem form. Dense generalized eigenproblems arise in many fields of science, such as macromolecular motion or dense functional theory.

In this thesis, we focus on the special case where the generalized eigenproblem is dense and symmetric. The solution of the generalized case can be directly obtained by computing the so-called generalized eigenvalues and the corresponding generalized eigenvectors but, in the case the problem is symmetric, other approaches can be used.

In particular, instead of computing the eigenvalues/eigenvectors of a given dense symmetric generalized eigenproblem, the solution of the corresponding standard eigenproblem is computed. The method of transforming from generalized to standard eigenproblem is described in Section 4.1. Since the transform is composed of the Cholesky factorization and matrix multiplication whose implementations, even for the out-of-core problems, attain very high performance on almost all computing architectures, this stage will not be covered in this thesis. Instead, the methods for the solution of the corresponding standard dense symmetric eigenproblem will be considered, since they are computationally more expensive and widely unexplored from the out-of-core perspective.

For the purpose of this thesis, the eigensolvers are divided into three groups: direct, Krylov-based and spectral divide–and–conquer. Although the traditional division of eigensolvers [54] is into direct and iterative eigensolvers, this partition is made for simplicity and to enable easier understanding of the novel GPU-based eigensolvers that are introduced in Chapter 5.

The eigenvalue algorithms can be roughly divided into 3 categories based on the size of the problem:

- Algorithms for *small* matrices,
- Algorithms for *medium* matrices,
- Algorithms for *large* matrices.

The categorization based on the matrix size mostly depends on the available computing architecture. Small matrices are those that fit into the main memory of the computing system; on the other hand, the medium and large matrices refer to those that require a large amount of memory (storage) that, sometimes, can even exceed the available system memory.

Traditionally, the algorithms for *small* matrices aim at computing all the eigenvalues by applying orthogonal similarity transformations in order to reduce the matrix to diagonal form. These algorithms target matrices that are dense (full) and unstructured as they cannot exploit the sparsity or any other special matrix structure. The most popular algorithms in this group are the QR algorithm [55, 56, 57] for general and Hermitian matrices, the Jacobi algorithm [58, 59], and the divide–and–conquer approach [23]. However, during the last decades, the Jacobi algorithm [60, 61, 62, 63] has become competitive even for medium size matrices, especially with the maturity of the GPU cards [64].

The algorithms for *medium* matrices focus on matrices that are too large to be efficiently diagonalized by applying orthogonal similarity transformations, but are small enough to use approximate solutions of the linear systems for computing the eigenpairs. Opposite to the algorithms for small matrices, these algorithms aim at computing only a subset of all eigenvalues. The computations are based on matrix–vector operations that can be efficiently performed if the sparsity and the structure of matrices are exploited. An example is the Jacobi–Davidson algorithm [65].

The last group consists of the algorithms that are applied on *large* matrices. These are matrices that require huge memory space and the operations on them exhibit extremely high computational cost ($\mathcal{O}(n^3)$). Therefore, these algorithms rely mostly on matrix-vector computations that are less computationally intensive (complexity $\mathcal{O}(n^2)$). The Krylov subspace based algorithms, such as Arnoldi-type algorithms [39], are an example for large eigenproblem solvers.

This chapter is structured as follows. Section 4.1 introduces the method and the kernels for the reduction of a dense generalized eigenproblem to its corresponding standard form. Two direct algorithms, based on one-stage and multi-stage reductions, are presented in Section 4.2. Section 4.3 describes the Krylov subspace-based algorithm for efficient solution of a subset of eigenvalues for large matrices. Finally, Section 4.4 shows an algorithm based on the spectral divide–and–conquer method that can be entirely performed in terms of level-3 BLAS kernels.

4.1 Dense generalized eigenproblem

The generalized eigenproblem is defined as:

$$AX = BX\Lambda, \tag{4.1}$$

where $A, B \in \mathbb{R}^{n\times n}$ are given, $\Lambda \in \mathbb{R}^{s\times s}$ is a diagonal matrix with the s sought-after eigenvalues on the diagonal, and the columns of $X \in \mathbb{R}^{n\times s}$ contain the corresponding eigenvectors. In case matrices A and B are symmetric, the problem (4.1) is referred to as a symmetric generalized eigenproblem. Instead of computing the eigenvalues of the generalized eigenproblem, the eigenvalues of the corresponding standard eigenproblem are computed. If the eigenvectors are aslo required, then a simple back-transformation is applied on the obtained eigenvectors of standard eigenproblem.

The reduction to a standard symmetric eigenproblem is performed as follows. First, consider the Cholesky factorization of B given by

$$B = U^T U, \tag{4.2}$$

where $U \in \mathbb{R}^{n\times n}$ is upper triangular [20]; then the generalized problem can be transformed into the standard one

$$CY = Y\Lambda, \tag{4.3}$$

by exchanging B with U^TU in (4.1) and some manipulation:

$$\begin{aligned} AX &= U^TUX\Lambda && (4.4)\\ U^{-T}AX &= UX\Lambda && (4.5)\\ U^{-T}AU^{-1}\,UX &= UX\Lambda, && (4.6)\end{aligned}$$

where $C = U^{-T}AU^{-1} \in \mathbb{R}^{n\times n}$ is symmetric, and $Y = UX$ contains the eigenvectors associated with the standard problem. While the eigenvalues of the generalized eigenproblem (4.1) and the standard one (4.3) are the same, the eigenvectors X of generalized problem can be easily recovered from the eigenvectors Y of the standard problem by solving the upper triangular linear system:

$$X := U^{-1}Y. \tag{4.7}$$

Once the generalized eigenproblem is reduced into the standard form (4.3), the sought-after eigenvalues and the corresponding eigenvectors can be computed directly by applying any standard eigensolver. The transformation to standard form exhibits a computational cost of $\mathcal{O}(n^3)$ flops and can be completely performed in terms of Cholesky factorization and triangular solvers, which attain very high performance on a number of computing architectures. In particular, various implementations of the Cholesky factorization [20] efficiently exploit the maximum performance of different computing platforms, such as the out-of-core implementation for CPU-only systems [34], hybrid CPU-GPU based systems for the in-core and the out-of-core problems [8, 66, 67], and the approaches targeting distributed GPU-based systems [29, 68].

4.2 Direct eigensolvers

The direct eigensolvers compute the eigenvalues/eigenvectors based on the orthogonal decomposition. Traditionally, the direct eigensolvers were used when all eigenvalues and the corresponding eigenvectors were required. In the general case, when the matrix is real, the direct eigensolvers compute the Schur decomposition ([20], Theorem 7.1.3), and in case the matrix is symmetric, the spectral decomposition (Section 2.2, Theorem 2.2.1):

$$Q^TAQ = D, \tag{4.8}$$

where Q is orthogonal and D is a diagonal matrix with the eigenvalues of A on the diagonal from which the eigenvalues can be directly read off. A simple backward substitution can be used to compute the eigenvectors of the matrix A by applying the orthogonal transforms in Q [20].

The state-of-the-art algorithms do not directly decompose the matrix to the diagonal form, but instead first reduce the matrix to tridiagonal form. This is done because computing the eigenvalues of a tridiagonal matrix is fast and can be done in $\mathcal{O}(n^2)$ flops. The eigensolver based on tridiagonalization comprises the three stages, described in Algorithm 4.1. In the first stage (line 1), the symmetric matrix A is reduced to a symmetric tridiagonal matrix T

$$Q^TAQ = T, \tag{4.9}$$

by a sequence of orthogonal similarity transforms Q. In the second step (line 2), a tridiagonal eigensolver, such as MRRR [25, 69], Bisection/Inverse Iteration, Divide–and–Conquer or QR iteration [18], is applied to compute the eigenvalues and, if required, the associated eigenvectors of the tridiagonal matrix T. Finally, if the eigenvectors of A are required, the back-transformation (line 3) is applied to the eigenvectors of T. In particular, if $TX_T = X_T\Lambda$, with the matrix X_T representing the eigenvectors of T, then $X = QX_T$ are the eigenvectors of A. Since matrices T and A are similar, the eigenvalues of A equal those of T.

Algorithm 4.1 Direct eigensolver for standard eigenproblem

```
Input: Dense symmetric matrix $A$
Output: Eigenvalues $\Lambda$ and the corresponding eigenvectors $X$ of $A$
1: Reduce $A$ to tridiagonal matrix $T$
2: Compute the eigenvalues (and eigenvectors) of $T$
3: Compute the eigenvectors of $A$ (back-transformation)
```

The first and the last step of the Algorithm 4.1 require $\mathcal{O}(n^3)$ flops and are the main computational bottlenecks. The cost of the second stage is negligible and requires only $\mathcal{O}(n^2)$ flops. The third stage can be performed in terms of matrix–matrix (GEMM) multiplications which is of the most studied operations in the linear algebra. The GEMM routine achieves very high computational performance, especially when executed on the general-purpose GPUs [70]. From the computational point of view, the first step, i.e. the reduction to tridiagonal form is considerably more expensive. The reduction of a dense symmetric matrix to tridiagonal form can be performed via two approaches, the one–stage approach and the multi–stage approach.

The one–stage approach computes a sequence of $n-2$ Householder transforms [20] that annihilate all entries below the first subdiagonal of the matrix A. The algorithm requires $4n^3/3$ flops, half of which can be performed in terms of call to level-3 BLAS routines. The remaining $2n^3/3$ flops are performed in terms of level-2 BLAS routines, which achieve poor performance due to the low data locality of level-2 BLAS routines. This one–stage approach is implemented in the routine SYTRD that is part of the LAPACK computational library [10, 71].

In contrast to the one–stage approach, the multi–stage approach first reduces the matrix A into the intermediate band matrix B and then subsequently transforms the matrix B into a sequence of narrower band matrices. Finally, the small band matrix is reduced to tridiagonal form. The advantage of this approach is that the first step can be performed completely in terms of level-3 BLAS routines while the cost of the successive reduction to narrower band form is negligible if moderate bandwidths are chosen.

The following sections describe the one–stage and multi–stage approaches as implemented in the LAPACK library and the SBR Toolbox [72], respectively.

4.2.1 One–stage approach

The one–stage approach was the first direct method to tridiagonalize real symmetric matrices and the easiest to implement. The most popular implementation, that presents a base for all other implementations, is LAPACK SYTRD kernel for multi-core architectures. The idea of the one–stage approach is to annihilate all the entries of the symmetric matrix A that are below the first subdiagonal and right of the first super-diagonal. The reduction is done column by column (or by row, because of symmetry) until only those entries on diagonal and the first super- and subdiagonal remain as non-zeros.

Concretely, the tridiagonalization of matrix A is computed by a series of Householder reflectors $H_1, H_2, \ldots, H_{n-2}$. Each reflector is an orthogonal matrix of the form $H_j = I - \beta_j u_j u_j^T$, where $\beta_j \in \mathbb{R}$, $u_j \in \mathbb{R}^n$, with the first j entries equal to zero, and where I denotes the square identity matrix. The purpose of each Householder reflector H_j is to annihilate the entries below the subdiagonal in the j^{th} column of sub-matrix A_{j-1}, where $A_{j-1} = H_{j-1}^T \cdots H_2^T H_1^T A H_1 H_2 \cdots H_{j-1}$ is obtained by applying all previous reflectors $H_1, H_2, \ldots, H_{j-1}$ on A from both sides.

The method described next is a blocked algorithm or algorithm-by-blocks [73], in which Householder reflectors are computed and applied for a block (panel) of columns instead for a single column. By improving data locality, this approach significantly increases the performance of the reduction [74] due to less cache-misses. Let b be the algorithmic block size and assume that the first $j-1$ columns of T, Equation (4.9), are already computed, i.e. the entities below the subdiagonal in the first $j-1$ columns are already annihilated. Consider the following partition

$$H_{j-1}^T \cdots H_2^T H_1^T A H_1 H_2 \cdots H_{j-1} = A_{j-1} = \left(\begin{array}{c|cc} T_{00} & T_{10}^T & 0 \\ \hline T_{10} & A_{11} & A_{21}^T \\ 0 & A_{21} & A_{22} \end{array}\right),$$

where $T_{00} \in \mathbb{R}^{(j-1)\times(j-1)}$ is tridiagonal and $A_{11} \in \mathbb{R}^{b\times b}$. With this partitioning, all entries of T_{10} are zero except for its top-right corner. The following steps are computed during the reduction of the current panel $\left(\begin{smallmatrix} A_{11} \\ A_{21} \end{smallmatrix}\right)$ to tridiagonal form.

1. The panel $\left(\begin{smallmatrix} A_{11} \\ A_{21} \end{smallmatrix}\right)$ is reduced to tridiagonal form by a sequence of b orthogonal transforms $H_j, H_{j+1}, \ldots, H_{j+b-1}$ such that

$$H_{j+b-1}^T \cdots H_{j+1}^T H_j^T \left(\begin{array}{c|cc} T_{00} & T_{10}^T & 0 \\ \hline T_{10} & A_{11} & A_{21}^T \\ 0 & A_{21} & A_{22} \end{array}\right) H_j H_{j+1} \cdots H_{j+b-1}$$
$$= \left(\begin{array}{c|cc} T_{00} & T_{10}^T & 0 \\ \hline T_{10} & T_{11} & T_{21}^T \\ 0 & T_{21} & A_{22} - UW^T - WU^T \end{array}\right)$$

   where T_{11} is tridiagonal and all entries of T_{21}, except its top-right corner, are zero. Matrices U,$W \in \mathbb{R}^{n-j-b+1\times b}$, required in the next step, are built concurrently with the reduction of the panel.
2. The submatrix A_{22} is updated as $A_{22} := A_{22} - UW^T - WU^T$, where, in order to exploit symmetry, only the lower (or the upper) half of the matrix A_{22} is updated.

The computation of U and W, which are defined by Householder reflectors, along with the reduction in Step 1 are required in Step 2 of the unreduced part of the matrix A_{j-1}. The columns of matrix U are the vectors $u_j, u_{j+1}, \ldots, u_{j+b-1}$ of the Householder reflectors $H_j, H_{j+1}, \ldots, H_{j+b-1}$. Much more work is done in the construction of matrix W. For each reduced column in the panel, a new column of W is generated. This requires four panel-vector multiplications and one symmetric matrix-vector multiplication with the submatrix A_{22} as the operand. The latter operation, computed with the level-2 BLAS routine SYMV, is the most expensive one, requiring roughly $2(n-j)^2b$ flops. Step 2 also requires $2(n-j)^2b$ flops, but is entirely performed by the level-3 BLAS kernel SYR2K for the symmetric rank-$2b$ update. The overall cost of SYTRD is $4n^3/3$ flops provided $b \ll n$.

Note that there is no need to construct the orthogonal factor $Q = H_1 H_2 \cdots H_{n-2}$ explicitly. Therefore, the vectors u_j defining the Householder reflectors H_j can be stored in the annihilated entries of A and no additional workspace is required. However, additional workspace is needed to store the scalars β_j, but this requires only $\mathcal{O}(n)$ entries and is thus negligible. If the eigenvectors are requested, the back-transform QX_T is computed with an additional cost of $2n^3$ flops. If Q is applied via the WY representation [75], the back–transformation can be performed almost entirely in terms of calls to BLAS-3 kernels.

The LAPACK one–stage implementation is characterized by two successive computational steps: the panel factorization (step 1) and the update of the trailing submatrix (Step 2). The panel factorization computes Householder reflectors almost entirely in terms of calls to level-2 BLAS kernels and accumulates them so that they can be applied onto the trailing submatrix using level-3 BLAS kernels. The parallelism in LAPACK resides within the multi-threaded BLAS library, such as GotoBLAS [76] or the most recent OpenBLAS [77] library, which follow the expensive fork and join model. This produces unnecessary synchronization points between the panel factorization and the trailing submatrix update phases. The serious coarse granularity is also a significant drawback and prevents from attaining a higher degree of parallelism [17].

4.2.2 Multi–stage approach

Despite the performance improvement of the one–stage approach obtained with the recent MAGMA library, a main drawback remains in that the number of computations performed as level-2 BLAS kernels is roughly 50%. This drawback is diminished with the multi-stage approach [72] but in exchange for an increment in the computational cost. The algorithm based on two-stage approach (a special case of multi-stage approach) achieves higher performance that the one–stage approach on the multi-threaded architectures [78] as well as GPU-based architectures [79].

In particular, instead of directly computing tridiagonal form, the algorithms based on multi–stage approach first reduce matrix to band form

$$Q_1^T A Q_1 \to A_{w_1}, \tag{4.10}$$

where $A_{w_1} \in \mathbb{R}^{n\times n}$ is a matrix of bandwidth w_1 and $Q_1 \in \mathbb{R}^{n\times n}$ collects the corresponding orthogonal transforms [20]. In the two-stage variant, this band matrix is then reduced to tridiagonal form

$$Q_2^T A_{w_1} Q_2 \to T, \tag{4.11}$$

so that $Q = Q_1 Q_2 \in \mathbb{R}^{n\times n}$ yields the reduction/orthogonal transform in (4.9). A truly multi-stage algorithm can be employed to successively transform A into a series of matrices of narrower band, $w_1 > w_2 > w_3 \ldots > w_r$, as in

$$Q_1^T A Q_1 \to A_{w_1}, \quad Q_2^T A_{w_1} Q_2 \to A_{w_2}, \quad Q_3^T A_{w_2} Q_3 \to A_{w_3}, \ldots, Q_r^T A_{w_{r-1}} Q_r \to A_{w_r}, \tag{4.12}$$

followed with the reduction of the band matrix A_{w_r} to tridiagonal form

$$Q_t^T A_{w_r} Q_t = T, \tag{4.13}$$

yielding the desired tridiagonal matrix T and the orthogonal transforms accumulated in $Q = Q_1 Q_2 \cdots Q_r Q_t$.

The eigensolver based on the multi–stage reduction approach is described in Algorithm 4.2. In the first step (1), as defined with Equation (4.10), the dense symmetric matrix is reduced to the band matrix form. Thereafter, the given band form is successively reduced to narrower band forms (2) and, in case the eigenvectors are required, the orthogonal transforms are accumulated (4). The narrower band form, obtained from the previous reductions, is finally reduced to tridiagonal form (6). Note that, if the eigenvectors are required, the orthogonal transforms are accumulated explicitly (4), adding n^2b-2nb^2i flops in each iteration. The last two steps are the same as in the one–stage reduction, where eigenvalues and the corresponding eigenvectors of the given tridiagonal matrix are directly computed with minimal computational cost by utilizing tridiagonal eigensolvers. The last step (8), the back–transformation, is performed only if eigenvectors are required.

Algorithm 4.2 Multi–stage eigensolver

```
Input: Dense symmetric matrix $A$
Output: Eigenvalues $\Lambda$ and the corresponding eigenvectors $X$
1: Reduce $A$ to $A_{w_1}$ ($Q_1^T A Q_1 = A_{w_1}$, $Q := Q_1$)
2: for $i = 2 : r$ do
3:     Reduce $A_{w_{i-1}}$ to $A_{w_i}$ ( $Q_i^T A_{w_{i-1}} Q_i = A_{w_i}$)
4:     Accumulate transforms ($Q = QQ_i$)
5: end for
6: Reduce from band to tridiagonal form ($Q_t^T A_{w_r} Q_t = T$, $Q = QQ_t$)
7: Compute eigenpairs of $T \to \Lambda$, $X_t$
8: Back-transformation ($X := QX_t$)
```

Although the multi–stage reduction can be performed via LAPACK kernels, the implementation from the Successive Band Reduction (SBR) toolbox [80] is better suited for large-scale problems. Thus, the SBR will serve as a base for the multi–stage eigensolver for the out-of-core problems on GPU.

The SBR toolbox

The Successive Band Reduction (SBR) [72, 80] is a software package for symmetric band reduction via orthogonal transforms. The toolbox includes routines for the reduction of dense symmetric matrices to band form (SYRDB), and the reduction of band matrices to narrower band form (SBRDB) or tridiagonal form (SBRDT). Since the symmetric matrices in band form have all entries below the band equal to zero, it is possible to store a band matrix in more economical way. Thus, the SBR provides two routines for repacking a symmetric band matrix from conventional storage to the LAPACK lower band storage scheme [10]. The benefit of repacking is not only in decreasing the workspace but also in faster access to consecutive column elements that are stored in column-major format (e.g. in FORTRAN).

Reduction to band form. Consider that the first $j-1$ columns of the matrix A have been already reduced to band form with bandwidth w. Let b denote the algorithmic block size, and assume for simplicity that $j+w+b-1 \leq n$, and n, w are integer multiples of w, b, respectively; see Figure 4.1. Then, during the current iteration of routine SYRDB, b new columns of the band matrix are computed as follows:

1. Compute the QR factorization of $A_0 \in \mathbb{R}^{k \times b}$, $k = n-(j+w)+1$:

$$A_0 = Q_0 R_0, \tag{4.14}$$

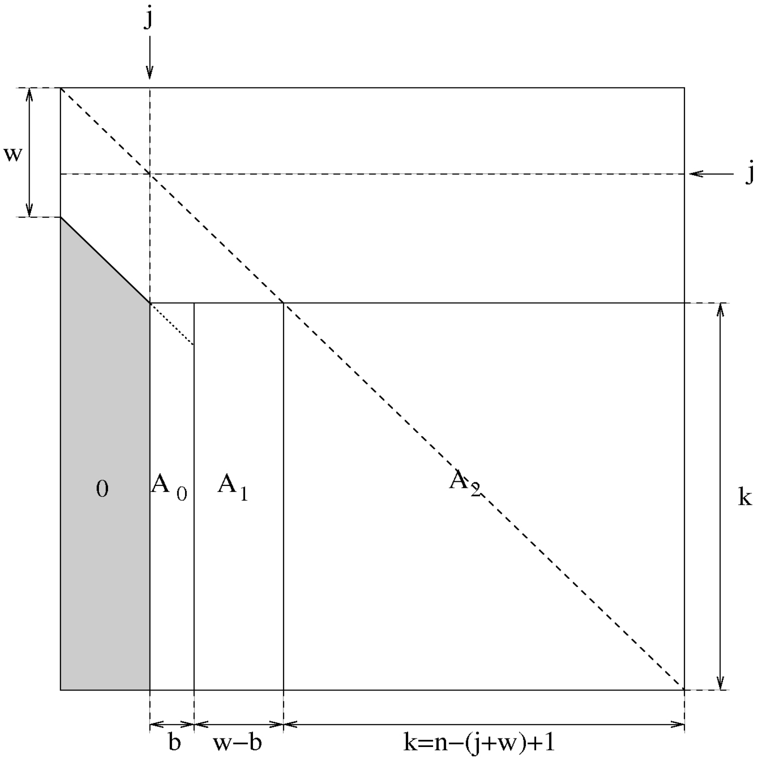


Figure 4.1: Partitioning of the matrix during one iteration of routine SYRDB for the reduction to band form.

where $R_0 \in \mathbb{R}^{b\times b}$ is upper triangular and the orthogonal factor Q_0 is implicitly stored as a sequence of b Householder vectors. The cost of this first step is $2b^2(k-b/3)$ flops.

2. Construct the factors W and Y of the WY representation [81] of the orthogonal matrix $Q_0 = I_k + WY^T$, with $W, Y \in \mathbb{R}^{k\times b}$. The cost of this step is kb^2 flops.
3. Apply the orthogonal matrix to $A_1 \in \mathbb{R}^{k\times w-b}$ from the left:

$$A_1 := Q_0^T A_1 = (I_k + WY^T)^T A_1 = A_1 + Y(W^T A_1). \tag{4.15}$$

   By performing the operations in the order specified in the rightmost expression of (4.15), the cost of this step becomes $4kb(w-b)$ flops. In case the bandwidth equals the block size ($w = b$), A_1 comprises no columns and, therefore, no operation is performed in this step.
4. Apply the orthogonal matrix to $A_2 \in \mathbb{R}^{k\times k}$ from both the left and right:

$$\begin{array}{rcl} A_2 & := & Q_0^T A_2 Q_0 = (I_k + WY^T)^T A_2 (I + WY^T) \\ & = & A_2 + YW^T A_2 + A_2 WY^T + YW^T A_2 WY^T. \end{array} \tag{4.16}$$

   In particular, during this step only the lower (or the upper) triangular part of A_2 is updated. In order to do this, (4.16) is computed as the following sequence of level-3

BLAS operations:

$$
\begin{aligned}
&(\textsc{symm}) & X_1 &:= A_2 W, & (4.17)\\
&(\textsc{gemm}) & X_2 &:= \frac{1}{2} X_1^T W, & (4.18)\\
&(\textsc{gemm}) & X_3 &:= X_1 + Y X_2, & (4.19)\\
&(\textsc{syr2k}) & A_2 &:= A_2 + X_3 Y^T + Y X_3^T. & (4.20)
\end{aligned}
$$

The major computational cost of the fourth step is in the computation of the symmetric matrix product (4.17) and the symmetric rank-$2k$ update (4.20), each with a cost of $2k^2b$ flops. The matrix products (4.18) and (4.19) require $2kb^2$ flops each. The overall cost of this step is $4k^2b + 4kb^2$ flops, which is higher than the cost of the remaining Steps 1, 2 and 3 which require $\mathcal{O}(kb^2)$, $\mathcal{O}(kb^2)$, $\mathcal{O}(\max(kb^2, kbw))$, respectively.

In summary, provided that b and w are both small compared to n, the global cost of the reduction of a full matrix to band form is $4n^3/3$ flops. Furthermore, the bulk of the computation is performed in terms of the level-3 BLAS kernels SYMM and SYR2K in (4.17) and (4.20), so that high performance can be expected in case a tuned BLAS is used.

The orthogonal matrix $Q_1 \in \mathbb{R}^{n\times n}$ that reduces A to the band matrix A_{w_1} (4.10), can be explicitly constructed by accumulating the involved Householder reflectors at a cost of an additional $2n^3 - 2n^2w_1$ flops. Once again, the compact WY representation helps in casting this computation almost entirely in terms of calls to level-3 BLAS. The SBR toolbox implements this functionality in routine SYGTR.

The accumulation of the Householder reflectors can be performed completely in terms of matrix-matrix multiplication. The performance of the routine can be increased by applying the multi-threaded implementations from OpenBLAS and LAPACK libraries. Furthermore, the reduction to band form performs more floating-point operations than the one–stage approach, especially when Q_1 is required, but outperforms the one–stage approach [78] as most of operations are performed as BLAS-3 operations.

Reduction from band to narrower band form. The reduction from band to narrower band form (4.11) is implemented in the SBR routine SBRDB. The idea of this algorithm is to repeatedly remove sets of outmost sub–diagonals from the symmetric band matrix A_{w_1}, obtained in the first step of the multi–stage algorithm (4.10).

A symmetric $n \times n$ band matrix A_{w_1} with the band width w_1 is reduced to the band matrix with bandwidth $w_2 = w_1 - d$, with $1 \leq d < n$. The parameter d is the number of sub(super)–diagonals that are to be peeled off from the band. Because of the symmetry, only the lower triangle of the symmetric band matrix and the corresponding sub–diagonals are observed.

The algorithm is based on an *annihilate–and–chase* strategy, similar to the algorithms proposed by Rutishauser [82], Schwartz [83], Murata and Horikoshi [84], and Lang [85]. Householder transforms are used to annihilate the d outmost sub–diagonals, and in case $w_2 > 1$, the WY or compact WY representation of the transforms can be used to improve the data locality. Basically, two annihilate and strategy algorithms are recognized; the first algorithm removes a subset of sub–diagonals from a band matrix and is called the one-step algorithm because sub–diagonals are peeled off in one pass (step). The second algorithm is called the multi-step algorithm that successively reduces the starting band matrix to the final narrower band form (4.12). One can also envisage the multi-step

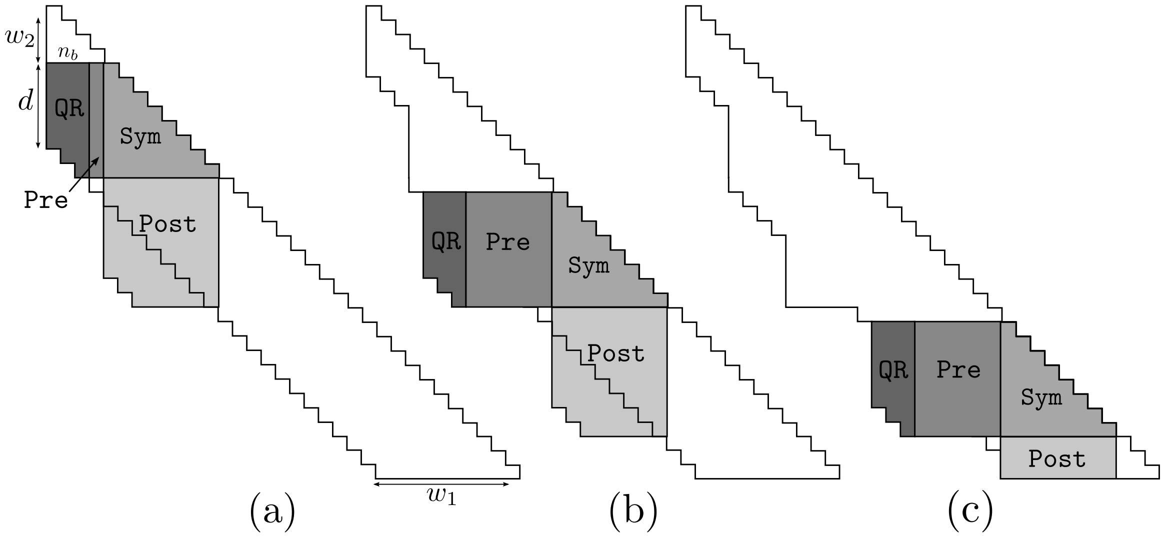


Figure 4.2: Annihilation of the outmost sub–diagonals of the first n_b columns of the band matrix and updating the rest of the matrix (a). Bulge chasing of the fill-in block (b) and (c)

algorithm as the one-step algorithm called successively on intermediate band matrices until the final band matrix is attained.

In the one step algorithm, each reduction sweep has two parts:

- Annihilation of several columns to reduce to a narrower band form
- Bulge-chasing to restore the band form from the trailing matrix.

Assume that $w_2 > 1$ and $n_b \leq w_2$. First, the d outmost sub–diagonals from the first n_b columns of A_{w_1} are annihilated. This can be done via the QR factorization of an $h \times n_b$ upper trapezoidal block "`QR`" where $h = d + n_b$, (Algorithm 4.3, line 4). Then, the WY representation of the block $Q = I + WY^T$ is generated. To complete the similarity transforms, the block Q is applied from the left and from the right to A_{w_1}. This requires applying Q from the left to the $h \times (d - n_b)$ block "`Pre`" (line 5), from both sides to the $h \times h$ lower triangular block "`Sym`" (line 6), and from the right to the $w_1 \times h$ block "`Post`" (line 7). A graphical illustration of how the similarity transforms Q are applied to the blocks "`Pre`", "`Sym`" and "`Post`" is given in Figure 4.2 (a).

Algorithm 4.3 Reduction from band to narrower band form, *toBand*(A, n, w_1, d, n_b)

Input: Symmetric band matrix $A \in \mathbb{R}^{n \times n}$ with bandwidth w_1, $1 < w_1 < n$, d the number of sub–diagonals to eliminate and n_b the block size, $1 \leq n_b \leq w_1 - d$

Output: Symmetric band matrix with the bandwidth $w_2 = w_1 - d$

```
1: for $j = 1 : n - w_2 - 1 : n_b$ do
2:     $j_1 = j$, $j_2 = j_1 + n_b - 1$, $i_1 = j + w_2$, $i_2 = \min(j + w_1 + n_b - 1, n)$
3:     while $i_1 < n$ do
4:         Perform QR on the block $B \equiv A(i_1 : i_2, j_1 : j_2)$ in place (QR)
5:         Replace the block $B \equiv A(i_1 : i_2, j_2 + 1 : i_1 - 1)$ by $Q^T B$ (Pre)
6:         Replace the block $B \equiv A(i_1 : i_2, i_1 : i_2)$ by $Q^T B Q$ (Sym)
7:         Replace the block $B \equiv A(i_2 + 1 : \min(i_2 + w_1, n), i_1 : i_2)$ by $BQ$ (Post)
8:     end while
9: end for
```

The application of the transforms on block "`Post`" fill-ins d diagonals below the band; see Figure 4.2 (a), the light grey area below diagonal of "`Pre`" block. During the application to "`Post`", the d diagonals below the band are filled-in with non–zero values, deforming the band structure. Therefore, the first n_b columns of the fill-in are removed by another QR factorization (Figure 4.2 (b)), followed by the corresponding application to "`Pre`", "`Sym`" and "`Post`". The process is repeated for the first n_b columns of the newly generated fill-in that was generated in the previous "`Post`" block (Figure 4.2 (c)), and so on. The process is repeated along the diagonal until the filled-in block is pushed off the matrix and the band form with bandwidth w_1 is restored. Then, the new *annihilate–and–chase* sweep can start.

Figure 4.3 shows the matrix after the first sweep. In the first n_b columns the d outmost diagonals are removed and the upper left sub-matrix is in the final narrower band form with bandwidth w_2. The rest of the matrix is block tridiagonal with diagonal blocks, except for the last one, of order b and with the off–diagonal blocks whose first n_b columns of each fill-in block (positions of the `Post` blocks in the Figure 4.2) are restored to the band form (w_1). The rest of the non–zero entries below the band in the fill-in blocks are left and are not chased down the diagonal as these positions will be filled-in with non–zero entries in the next sweeps and annihilated. Therefore, in each sweep, only the fill-in columns that will be accessed by the `QR` in the next sweep are restored to the band form w_1.

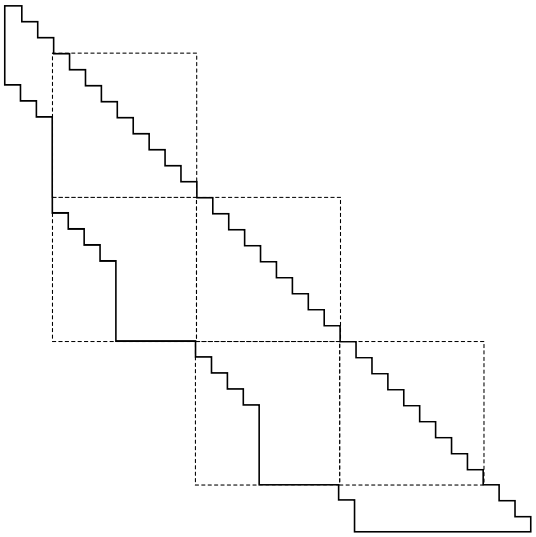


Figure 4.3: Band matrix form with the first n_b columns reduced to the narrower band form.

After describing the one step algorithm for the reduction from band to narrower band form we next proceed with the multi-step algorithm that peels off chunks of sub–diagonals. The multi-step reduction to narrower band form is described in Algorithm 4.4. The input for the algorithm is a sequence of k positive integers $d^{(i)}$, $d = \sum_{i=1}^{k} d^{(i)} < w_1$, the number of the sub–diagonals to be eliminated in each step and a sequence of k integers $n_b^{(i)}$.

In practice, the multi-step algorithm can be applied on a matrix that is not in tridiagonal form. One can see that, for $d^{(k)} = b^{(k-1)}$, the band matrix is reduced to tridiagonal matrix because all sub–diagonals, except the first one, are annihilated. Moreover, if $w_1 = n-1$ and $d = w_1 - 1$, the algorithm performs the one–stage reduction of a dense matrix to tridiagonal form. However, by choosing different values for $d^{(i)}$ and $b^{(i)}$ one can expect to achieve some good algorithmic properties such as a better exploitation of the level-3 BLAS kernels. For example, for large d, i.e. when the difference between w_1 and w_2 is considerable, the block `QR`, Figure 4.2 (a), becomes tall and data locality is poorly

Algorithm 4.4 Multi-step reduction from band to narrower band matrix

```
Input: A symmetric band matrix $A \in \mathbb{R}^{n \times n}$ with the bandwidth $w_1$, a sequence of positive integers $d^{(1)}, d^{(2)}, \ldots, d^{(k)}$ and a sequence $n_b^{(1)}, n_b^{(2)}, \ldots, n_b^{(k)}$
Output: A symmetric band matrix with the bandwidth $w_2$
$b^{(1)} = w_1$
for $i = 1 : k$ do
    Call $toBand(A, n, b^{(i)}, d^{(i)}, n_b^{(i)})$
    $b^{(i+1)} = b^{(i)} - d^{(i)}$
end for
```

exploited because the block size n_b is small ($d < w_1 - d$). On the other hand, if d is small, the data locality is better exploited but the algorithmic complexity is increased as more steps are needed to reduce the band matrix to tridiagonal form. Thus, the parameters $b^{(i)}$ and $d^{(i)}$ have significant influence on performance of the algorithm for the reduction from band to narrower band form.

Reduction to tridiagonal form. Routine SBRDT in the SBR toolbox implements the reduction of a band matrix A_{w_r} to a tridiagonal form, Algorithm 4.2 (line (6)). Let Q_t denote the orthogonal transforms which yield this reduction, that is $Q_t^T A_{w_r} Q_t = T$, Equation (4.13). On exit, the routine returns the tridiagonal matrix T and, upon request, accumulates these transforms, forming the matrix $Q = Q_b Q_t \in \mathbb{R}^{n \times n}$, where $Q_b = Q_1 Q_2 \cdots Q_r$ is the accumulation of orthogonal transforms from all the previous steps, so that $Q^T A Q = Q_t^T (Q_b^T A Q_b) Q_t = Q_t^T A_{w_r} Q_t = T$.

Matrix T is constructed in routine SBRDT one column at a time: at each iteration the elements below the first subdiagonal of the current column are annihilated using a Householder reflector; the reflector is then applied to both sides of the matrix, resulting in a bulge which has to be chased down along the band. The computation is cast in terms of level 2 BLAS operations at best (SYMV and SYR2 for two-sided updates, and GEMV and GER for one-sided updates) and the total cost is $6n^2w + 8nw^2$ flops.

If the eigenvectors are desired, the orthogonal transforms computed in the reduction the reduction to tridiagonal form are accumulated from the left to the matrix Q_b obtained from the first two steps (reduction from full to band form and successively to narrower band form) such that $Q = Q_b Q_t$. These accumulations require $\mathcal{O}(n^3)$ flops and can be cast almost entirely in terms of calls to level-3 BLAS kernels, even though this reformulation is less trivial than that involved in the first step [72]. Furthermore, the accumulated orthogonal transforms Q are used and applied in the back-transform stage, to obtain the eigenvectors of A, $X := QX_t$, adding $2n^3$ flops to the overall computational cost.

Some recent researches address communication-avoiding successive band reduction to tridiagonal form. These researches aim at reducing the communication (i.e. data movement) by chasing the multiple bulges at the time and achieve 2 to 6× [86] speedup compared to the state-of-the-art implementation [72].

4.3 Krylov subspace based approaches

The second large group of methods are the iterative eigensolvers. Unlike the direct solvers, the iterative solvers iteratively approximate the eigenvalues and the associated eigenvectors. Traditionally, these methods have been used to find eigenpairs of sparse matrices and/or when only a subset of eigen–spectrum is required. However, there is a very thin line between direct and iterative solvers. In fact, each direct eigensolver has an iterative part. For example, the direct solvers based on the reduction to tridiagonal form iteratively compute the eigenvalues and the associated eigenvectors of the tridiagonal form. The methods to compute eigenpairs of the tridiagonal form (e.g. MRRR and bisection algorithms) are iterative algorithms. Generally, we can say that the direct solvers are those that solve problems in a finite sequence of iterations while the iterative solvers produce a sequence that converges to the eigenvalues and the corresponding eigenvectors.

The algorithms described in the Section 4.2 are based on orthogonal transforms and are very efficient for small to medium-sized eigenproblems. In case the matrix size is too large, the direct eigensolvers become impractical because of their expensive QR factorization that is applied on a large dense matrix. Opposite to the direct methods, Krylov-based methods avoid the computationally intensive matrix-matrix operations and rather rely on matrix-vector operations that perform in $\mathcal{O}(n^2)$ flops. Due to a low ratio between the number of flops performed and the amount of data transfered ($\mathcal{O}(n^2)$), Krylov-based algorithms are regarded as memory-bounded algorithms, i.e. algorithms whose performance is constrained by the data transfers. Although the algorithms based on Krylov subspace are memory-bounded, they can be highly competitive as they exhibit much lower arithmetic cost compared to compute-bound direct eigensolvers.

This section is focused on two Krylov-based algorithms, the Arnoldi and Lanczos methods. These algorithms have low computational cost and very fast convergence when only a small subset of eigenvalues/eigenvectors is required. This kind of problems are typically coming from molecular dynamics simulation problems and dense functional theory.

The Arnoldi method

The Arnoldi method belongs to the class of methods based on the idea of Krylov subspaces. It was first introduced in 1951 by Arnoldi [87] to reduce a dense matrix to Hessenberg form. Arnoldi expected that this method could give a good approximation of some eigenvalues but it later showed out that this method yields a good technique for finding eigenvalues of general large sparse matrices.

The idea for the Arnoldi method came up after trying to solve some drawback of the power method [20, 39]. The power method calculates the sequence $Au, A^2u, A^3u, \ldots$ iteratively, storing the result in u on every turn. This sequence converges to the eigenvector corresponding to the largest eigenvalue, and most computations are spent in forming the final result $A^{n-1}u$. One of the solutions for this drawback is to use Krylov matrix instead:

$$K_k(A,u) = [u, Au, A^2u, \ldots, A^{k-1}u].$$

When k increases, the vectors $A^k u$ converge to an eigenvector corresponding to the largest eigenvalue as in power method. To avoid the columns of Krylov matrix become linearly dependent, an orthogonal basis for a Krylov subspace is chosen via Gram-Schmidt orthogonalization. The orthogonalization is done as follows.

Suppose that the set $\{u_1, u_2, \ldots, u_j\}$ is an orthonormal basis for the Krylov subspace $\mathcal{K}^j(u)$. The extension of the basis with the vector u_{j+1} is done by first orthogonalizing the $A^j u$ against $u_1, \ldots, u_j$,

$$y_j := A^j u - \sum_{i=1}^{j} u_i u_i^T A^j u, \tag{4.21}$$

and then normalizing the vector

$$u_{j+1} = y_j / \|y_j\|.$$

The set $\{u_1, \ldots, u_j, u_{j+1}\}$ is an orthonormal basis for $\mathcal{K}^{j+1}(u)$ and is called the Arnoldi basis. In case A is symmetric, it is called the Lanczos basis. The vectors u_i for the orthonormal basis are called Arnoldi or Lanczos vectors.

The computation of vector u_{j+1} is computationally intensive because of the expensive product $A^j x$. Instead of $A^j x$, we can compute u_{j+1} by orthogonalizing Aq_j against $u_1, \ldots, u_j$, and (4.21) then becomes:

$$w_j := Au_j - \sum_{i=1}^{j} u_j (u_i^T A u_j). \tag{4.22}$$

If w_j is null vector, we have found an invariant subspace spanned by the vectors $\{u_1, \ldots, u_{j+1}\}$.

The orthogonalization can be done explicitly or implicitly. In the explicit Gram-Schmidt orthogonalization, the constructed basis $K_k(A, u)$ often suffers from inaccuracy and loss of orthogonality because of finite-precision arithmetic. A more numerically stable approach is to implicitly construct the orthogonal basis for Krylov subspace. Theorem 4.3.1 provides the basic idea for this approach.

Theorem 4.3.1 *Let the columns of*

$$U_{k+1} = [u_1, u_2, \ldots, u_{k+1}] \in \mathbb{R}^{n \times (k+1)}$$

form an orthonormal basis for $\mathcal{K}_{k+1}(A, u_1)$*. Then, there exists a* $(k+1) \times k$ *unreduced upper Hessenberg matrix* H_k *so that*

$$A\, U_k = U_{k+1}\, H_k.$$

Conversely, a matrix U_{k+1} *with orthonormal columns satisfies a relation of the above form only if the columns of* U_{k+1} *form a basis for* $\mathcal{K}_{k+1}(A, u_1)$*.*

From Theorem 4.3.1 we can define the Arnoldi decomposition that will directly lead to the construction of the basic Arnoldi method.

Definition (Arnoldi decomposition): Let the columns of $U_{k+1} = [U_k, u_{k+1}] \in \mathbb{R}^{n \times (k+1)}$ form an orthonormal basis. If there exists an (unreduced) Hessenberg matrix $H_k \in \mathbb{R}^{(k+1) \times k}$ so that

$$AU_k = U_{k+1} H_k \tag{4.23}$$

then Equation (4.23) is an (unreduced) Arnoldi decomposition of order k.

Following Theorem 4.3.1 and the definition of the Arnoldi decomposition, we can provide the basic Arnoldi algorithm (Algorithm 4.5) for computing an orthonormal basis of a Krylov subspace. The cost of the algorithm is dominated by the matrix-vector multiplication (3), which is the main and only computational bottleneck of the algorithm. The required storage space is $n \times (k+1)$ for storing the columns of U_{k+1}. The iteration is stopped when U_j spans an invariant subspace, i.e. when $h_{j+1,j} = 0$ (null vector) (10). The eigenvalues of H_j are the approximations of those of A. If the obtained approximations of eigenvalues are not good, the iteration is restarted with a random unit vector orthogonal to U_j.

Algorithm 4.5 Basic Arnoldi algorithm

Input: Matrix $A \in \mathbb{R}^{n \times n}$, vector $u_1 \in \mathbb{R}^n$, such that $\|u_1\|_2 = 1$, and integer $k \leq n$

Output: A matrix $U_{k+1} = [u_1 \ldots u_{k+1}] \in \mathbb{R}^{n \times (k+1)}$ with orthonormal columns and the upper Hessenberg matrix $H_k = (h_{ij}) \in \mathbb{R}^{(k+1) \times k}$ defining an Arnoldi decomposition (4.23) of k^{th} order

```
 1: $U_1 = [u_1]$
 2: for $j = 1 : k$ do
 3:     $w = Au_j$
 4:     for $i = 1 : j$ do
 5:         (Gram-Schmidt orthogonalization)
 6:         $h_{i,j} = u_i^T w$
 7:         $w := w - u_i h_{i,j}$
 8:     end for
 9:     $h_{j+1,j} = \|w\|_2$
10:     if $h_{j+1,j} == 0$ then
11:         $U_j$ is $A$-invariant, exit
12:     end if
13:     $u_{j+1} = w / h_{j+1,j}$
14:     $U_{j+1} = [U_j \; u_{j+1}]$
15: end for
```

The Lanczos method

The Lanczos method [88, 89] is a version of the Arnoldi method for the case when matrix is symmetric. The symmetry of the matrix gives some additional properties that cause the Lanczos method to be simpler than Arnoldi's. If the basic Arnoldi algorithm is applied on a symmetric matrix A, then the Hessenberg matrix H_k is real, tridiagonal, and symmetric. This is proved by the fact that the Hessenberg matrix is formed by applying the orthogonal transform to A such that $H_k = U_k^T A U_k$ which yields that H_k is also symmetric (because of the similarity with the symmetric matrix A). Furthermore, by the definition, the Hessenberg matrix has all entries below the first subdiagonal equal to zero, and because it is also symmetric, all the entries above the first superdiagonal have to be zero as well.

For simplicity we will use different notation for the elements of the matrix H_k so that $\alpha_j = h_{j,j}$ for the diagonal elements and $\beta_j = h_{j-1,j}$ for the subdiagonal elements. The Lanczos method is described in Algorithm 4.6.

From the computational and storage points of view, the Lanczos algorithm requires less storage space compared with the Arnoldi method, because only three vectors are stored. Furthermore, because of the tridiagonal symmetric matrix, the Gram-Schmidt orthogonalization process requires less flops (lines 5 and 6), only 2 inner products instead of $\mathcal{O}(n)$ as in the case in the Arnoldi method.

Restarting the Arnoldi algorithm

One of the main drawbacks of the Arnoldi method, presented in Algorithm 4.5, is that the number of the iterations k required to converge to the sough-after eigenvalues, is unknown at the beginning of the algorithm. Furthermore, if the eigenvectors are required, then the storage space, U_{k+1}, required for storing vectors of Arnoldi basis, u_i, is unknown

Algorithm 4.6 Basic Lanczos algorithm

```
Input: A symmetric $A \in \mathbb{R}^{n\times n}$, vector $u_1 \in \mathbb{R}^n$, such that $\|u_1\|_2 = 1$, and integer $k \leq n$
Output: A matrix $U_{k+1} = [u_1 \dots u_{k+1}] \in \mathbb{R}^{n\times(k+1)}$ with orthonormal columns and $\alpha, \beta \in \mathbb{R}$
$\beta_0 = 0$
$U_1 = [u_1]$
for $j = 1 : k$ do
    $w = Au_j$
    $\alpha_j = u_j^T w$
    $w := w - \beta_{j-1}u_{j-1} - \alpha_j u_j$
    $\beta_j = \|w\|_2$
    if $\beta_j == 0$ then
        $U_j$ is $A$-invariant, exit
    end if
    $u_{j+1} = w/\beta_j$
    $U_{j+1} = [U_j\ u_{j+1}]$
end for
```

and increases with the number of iterations performed. The size of the matrix U_{k+1} is $n\times(k+1)$ and depends on the number of iterations k. Thus, the size of matrix U_{k+1} may exceed the available system memory long before the desired eigenvalues are approximated well enough. As a result, a large eigenvalue subproblem represented with the Hessenberg matrix H_k is solved at the cost of $\mathcal{O}(n^3)$ flops. A similar approach is applied in the Lanczos method, but with $H_k \in \mathbb{R}^{(k+1)\times k}$ symmetric.

Suppose that n iterations of the basic algorithm are completed and that the approximated eigenvalues/eigenvectors are obtained. Based on the known eigenvector approximations, a new initial vector u_1, orthogonal to the obtained eigenvectors, is chosen and the basic Arnoldi algorithm is re-run for the next n steps. If the new eigenpairs are not well approximated, the process is repeated till the desired accuracy is attained. This version is called the Restarted Arnoldi algorithm [39]. The restarting technique plays a significant role in speeding up the convergence of the Arnoldi-based algorithms. Two approaches for restarting are the implicitly and explicitly restarted Arnoldi/Lanczos method [54, 90].

The explicit restarting was proposed by Saad [91] and is based on the polynomial acceleration scheme for the iterative solution of linear systems. The process includes the so-called *filter polynomial* p based on the information of the eigenvalues of H_k. The Arnoldi method is restarted with the new initial vector $u_1 = p(A)u_1/\|p(A)u_1\|_2$. With an increasing number of Arnoldi steps k, the number of eigenvalue approximations (eigenvalues of H_k) gets larger and larger, and many of them are useless as approximations of the sought-after eigenvalues. Thus, the purpose of the polynomial p is to remove those unwanted approximations and decrease the number of the performed floating-point operations. Further details on the restarted Arnoldi algorithm can be found in [90].

The implicitly restarted method [92] is more efficient and numerically stable than the explicitly restarted variant. It is a method that combines the implicitly shifted QR mechanism with a k-step Arnoldi/Lanczos method to obtain a truncated form of the implicitly shifted QR-iteration. With this algorithm it is possible to compute a few eigenvalues (largest/smallest, or in some range). For more details on the implicitly restarted Arnoldi algorithm, refer to [92]. One of the major practical implementation of these methods was done with the Implicitly Restarted Arnoldi Method (IRAM) [93] available in the ARPACK [94] library.

The Implicitly restarted Arnold method (IRAM) and Implicitly restarted Lanczos method (IRLM) are implemented in the ARnoldi PACKage (ARPACK) library [94, 95] for the solution of non-symmetric and symmetric eigenvalue problems, respectively. The IRAM and IRLM methods achieve high performance when only a small subset of eigenvalues/eigenvectors are required because of the fast convergence and the fact that the computation is based on the computationally cheap matrix-vector product (e.g. LAPACK's `xSymv` routine). Traditionally, these methods were applied to the solutions of large and structured eigenvalue problems in which the matrix structure can be exploited to further decrease the computational cost of the matrix-vector product.

ARPACK exploits the parallelism of the traditional multi-core processors by utilizing the underlying internal BLAS parallelism based on multi-threading. The distributed-memory PARPACK [96] library is an extension of ARPACK based for message passing systems. It supports BLACS [97] and the MPI [98] communication layers.

One of the main features of the ARPACK library is the reverse communication [99]. This is a technique by which the implementation details and various operations can be hidden from the implementation of the iterative method itself. In the Arnoldi/Lanczos method, the matrix-vector multiplication is hidden from the method's implementation giving the user the possibility to choose any convenient data structure for the matrix representation or to choose any available subroutine to perform the matrix-vector multiplication. Therefore, any architecture specific implementation of the matrix-vector product can be used depending the problem requirements. That gives ARPACK the robustness to exploit the parallelism of different computing architectures. For example, in case a very large scale eigenproblem is solved on a distributed-memory system, PLAPACK [100] or ScaLAPACK [101] library implementations of matrix-vector multiplication can be used.

4.4 Spectral divide-and-conquer based algorithms

The direct algorithms in Section 4.2 are very efficient in terms of number of floating-point operations per second, especially the multi-stage approach, but are impractical when only a small subset of eigenvalues/eigenvectors is required. For these methods, the difference in the arithmetic cost and the overall execution time of computing the full set or a subset of eigenvalues/eigenvectors is negligible. The efficiency of the direct algorithms comes from the fact that most of the computations can be performed in terms of level-3 BLAS operations. On the other hand, the Krylov-based eigensolvers are the algorithms of choice when only a small subset of the eigenvalues is required. The main drawback of Krylov-based algorithms is that they are completely performed in terms of level-2 BLAS kernels which have very low computational intensity. This particularly comes to fore when solving large-scale eigenproblems that are out-of-core for the GPU or even the main memory, and thus, a big penalty has to be paid to transfer the data resulting in a transfer time that can easily overcome the computational time.

In contrast to the previously described eigensolvers, the spectral divide-and conquer algorithm tries to overcome some of these problems by relying on a divide and conquer design pattern. The main idea of the spectral divide and conquer approach is to divide a problem into two subproblems by computing the invariant subspace for a subset of the eigenvalues. The algorithm is recursively applied on the two subproblems until they are simple enough to be solved directly by applying any traditional direct eigensolver. At the

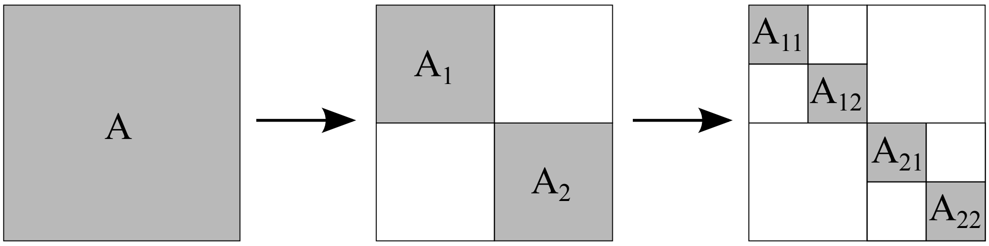


Figure 4.4: A schematic preview of the spectral divide and conquer algorithm

end of the recursion process, the solution of the subproblems are then combined into the final solution of the problem.

The first divide and conquer algorithm used the matrix sign function to split the spectrum of shifted matrices about the imaginary axis [102] but showed to be numerically instable for certain matrix types. The numerical stability was improved with the algorithm by Lin and Zmijewski [103] that employs the orthogonal basis. The major improvement in the algorithm construction was done with the inverse-free algorithm based only on rank revealing QR factorization and matrix multiplication [104] that is especially suited for generalized eigenvalue problems.

The parallel spectral divide and conquer algorithms for symmetric matrices were developed in the PRISM project [105]. In [106] the authors proposed a new parallel algorithm for computing the invariant subspaces of Hermitian matrices using only matrix-matrix multiplications and the QR factorization.

In general, the divide-and-conquer eigensolvers present a significantly higher arithmetic cost than the direct eigensolvers based on the reduction to condensed form. Furthermore, none of the algorithms have been proven to be backward stable. Recently, a divide-and-conquer algorithm based on the polar decomposition [107] was published, which has been proved to be backward stable. This method, called the spectral divide–and–conquer (SD&C) algorithm, can be implemented almost entirely in terms of matrix-matrix kernels and the QR factorization thus exhibiting very high performance.

Algorithm description

The main idea of the SD&C algorithm is described in the Figure 4.4 and is similar to the previously described divide–and–conquer ideas. Matrix A is split into two subproblems, A_1 and A_2, by computing invariant subspaces via the polar factor for a user-defined splitting point. The process is then recursively applied on each subproblem, with new splitting points, until the submatrix A_i is diagonal or small enough to directly compute the eigenvalues.

The polar factor is computed via the QDWH (QR-based dynamically weighted Halley) algorithm [108]. This is an algorithm based on QR factorization, which computes the polar decomposition and will be presented later. The positive and the negative invariant subspaces, that correspond to A_1 and A_2, are computed via a polar factor obtained from QDWH. The key point is that the invariant subspaces can be efficiently computed via the polar factor. Algorithm 4.7 shows the QDWH-based SD&C algorithm.

The algorithm starts with choosing the splitting point σ (1). The aim of the splitting point is to make A_1 and A_2 of similar dimension in order to provide the recursion balanced. Therefore, σ is a point that splits the spectrum in two parts, the left side with the eigenvalues smaller, and the right side with the eigenvalues that are larger than the splitting point. If all the eigenvalues of A are computed, the optimal splitting point would be the median of the spectrum. Instead of computing the median of the eigenvalues of A, which is very costly, the median of diag(A) can be used as a good estimate. Also, other σ strategies can be used, like the median of the eigenvalues of the tridiagonal part of A, or the center of the interval containing the eigenvalues. The QDWH algorithm is employed to compute the orthogonal polar factor U_p of the matrix A shifted by the σ factor (2). In the steps (3)-(4) of the algorithm, the subspace iteration is used to compute the orthogonal matrix V, which is then applied from the left and the right to compute A_1 and A_2 such that:

$$\begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} A\,[V_1 V_2] = \left(\begin{array}{c|c} A_1 & E^T \\ \hline E & A_2 \end{array} \right),$$

where $\|E\|_F \approx u$, the machine unit roundoff. In the last step (5), the SD&C algorithm is recursively called on submatrices A_1 and A_2. The recursion is stopped once the diagonal blocks (submatrices) are small enough so that any standard eigensolver can be invoked to compute the eigenpairs. However, the recursion can be run till the diagonal blocks are scalars. In the practical implementation, the computational time for the final stages of the algorithm, when the recursion is run to the scalar level, is negligible [107] and is comparable to a method when the standard eigensolvers are invoked on small-size diagonal blocks.

Algorithm 4.7 QDWH based spectral D&C algorithm

```
Input: Symmetric dense matrix $A \in \mathbb{R}^{n\times n}$
Output: Spectral decomposition of $A = VDV^T$
1: Choose $\sigma$
2: Compute polar factor $U_p$ of $A - \sigma I$
3: Compute orthogonal matrix $V = [V_1 V_2]$ such that $\frac{1}{2}(U_p + I) = V_1 V_1^T$
4: Compute $A_1 = V_1^T A V_1$ and $A_2 = V_2^T A V_2$
5: Repeat steps (1)-(4) for $A_1$ and $A_2$
```

The QDWH-based SD&C algorithm is the first algorithm for which backward stability has been proven. The proof can be found in [107](Theorem 3.1). Furthermore, the algorithm has low communication costs and the main computational routines are matrix-matrix operations and the QR factorization that can achieve high computational performance on the modern memory hierarchy architectures.

Polar decomposition

For any rectangular matrix $A \in \mathbb{R}^{m\times n}$ there exists a polar decomposition

$$A = U_p H, \tag{4.24}$$

where U_p has orthonormal columns and H is symmetric positive semidefinite. The matrix U_p is called a polar factor of the matrix A. The decomposition is unique if A has full column rank [109].

In the SD&C algorithm the polar decomposition is computed via the QDWH algorithm described by Nakatsukasa [108]. The QDWH algorithm iteratively computes the sequence of iterations:

$$X_{k+1} = X_k(a_k I + b_k X_k^T X_k)(I + c_k X_k^T X_k)^{-1}, \quad X_0 = A/\alpha, \tag{4.25}$$

where the limit of the sequence X_k is the polar factor U_p.

The factor α is an estimate of $\|A\|_2$ such that $\alpha \gtrsim \|A\|_2$. Setting $a_k = 3, b_k = 1, c_k = 3$ gives the Halley iteration. It is proven that the sequence X_k converges globally to the polar factor and that the convergence rate is cubic [110, 111].

The parameters a_k, b_k, c_k are dynamically chosen in each iteration to speed up the convergence such that

$$\begin{aligned} a_k &= h(l_k), && (4.26) \\ b_k &= (a_k - 1)^2/4, && (4.27) \\ c_k &= a_k + b_k - 1, && (4.28) \end{aligned}$$

where $h(l) = \sqrt{(1+\gamma)} + \frac{1}{2}(8 - 4\gamma + 8(2 - l^2)/()l^2\sqrt{1+\gamma})$ with $\gamma = (4(1 - l^2/l^4))^{1/3}$. The parameter l_k is computed from the recurrence:

$$l_k = l_{k-1}(a_{k-1} + b_{k-1}l_{k-1}^2)/(1 + c_{k-1}l_{k-1}^2), \tag{4.29}$$

for $k \geq 1$. The parameter l_k is a lower bound for the smallest singular value of X_k. The computation cost of the parameters a_k, b_k, c_k is negligible and does not require any matrix computation. From the description of l_k, the starting value l_0 is chosen to be the lower bound for the smallest singular value of X_0 and can be obtained via a condition number estimator. With the parameters defined as above, at most 6 iterations are needed to converge to U_p with the tolerance close to the unit roundoff for IEEE double-precision arithmetic.

The QDWH iteration (4.25) has a practical form that is based on the QR factorization and is more eligible for the implementation:

$$X_0 = A/\alpha, \tag{4.30}$$

$$\begin{bmatrix} \sqrt{c_j}X_j \\ I_n \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R, \quad X_{k+1} = \frac{b_k}{c_k}X_k + \frac{1}{\sqrt{c_k}}\left(a_k - \frac{b_k}{c_k}\right)Q_1Q_2^T \tag{4.31}$$

The main cost in the (4.31) is the QR factorization of the $(m+n) \times n$ matrix on the left hand side. Note that the matrix is symmetric and that the lower half of the matrix is equal to the identity matrix.

Invariant subspace

The invariant subspace of a symmetric matrix, corresponding to the positive (or negative) eigenvalues, is computed using the polar decomposition obtained in the QDWH algorithm. Therefore, it is necessary to establish the connection between the polar decomposition and the spectral decomposition of the symmetric matrix A. First, assume that A is nonsingular. In case A is singular, the subspace iteration finds an invariant subspace V_1 that corresponds to the eigenvalues equal or larger than zero [107] (subsection 5.4.).

Let $A = U_pH$ be the polar decomposition and $A = V\Lambda V^T$ be an spectral decomposition. Furthermore, Λ is divided as $\Lambda = \mathrm{diag}(\Lambda_+, \Lambda_-)$ where the matrices Λ_+ and Λ_- contain the positive and negative eigenvalues of A, respectively. If we assume that there are k positive eigenvalues then

$$\begin{aligned} A &= V\,\mathrm{diag}(\Lambda_+, \Lambda_-)V^T \\ A &= V\,\mathrm{diag}(I_k, -I_{n-k})V^T \times V\,\mathrm{diag}(\Lambda_+, |\Lambda_-|)V^T \\ &\equiv U_pH \end{aligned}$$

If we consider that the U_p is computed by invoking the QDWH algorithm (subsection 4.4) and that $V = [V_1, V_2]$ is partitioned as Λ, then

$$U_p + I = [V_1 V_2] \begin{bmatrix} I_k & 0 \\ 0 & -I_{n-k} \end{bmatrix} [V_1 V_2]^T + I = [V_1 V_2] \begin{bmatrix} 2I_k & 0 \\ 0 & 0 \end{bmatrix} [V_1 V_2]^T = 2V_1 V_1^T, \quad (4.32)$$

where the symmetric matrix $C = \frac{1}{2}(U_p + I) = V_1 V_1^T$ is an orthogonal projector onto the subspace spanned by the columns of V_1 and that is the invariant subspace corresponding to the positive eigenvalues. The matrix V_1 can be computed as an orthogonal basis for the column space of C and the subspace iteration algorithm [19] is applied. The pseudo-code for subspace iteration is given in Algorithm 4.8. The matrix $C = V_1 V_1^T$ is the orthogonal projection onto an invariant subspace of A and k is the number of positive eigenvalues.

Algorithm 4.8 Subspace iteration algorithm

```
Input: Symmetric matrices $A \in \mathbb{R}^{n\times n}$ and $C = \frac{1}{2}(U_p + I) \in \mathbb{R}^{n\times n}$, $k = \|C\|_F^2$
Output: $V_1 \in \mathbb{R}^{n\times k}$
Choose initial matrix $X \in \mathbb{R}^{n\times k}$
Compute QR factorization $X = [V_1 V_2] \begin{bmatrix} R \\ 0 \end{bmatrix}$
Form $E = V_2^T A V_1$
If $\|E\|_F / \|A\|_F = \epsilon$, stop
$X := CV_1$, go to (2)
```

The number of eigenvalues $k = \|C\|_F^2$ is an integer since the eigenvalues of C are either 0 or 1. The algorithm converges with the convergence rate $|\lambda_{r+1}|/|\lambda_k|$ for the kth eigenvalue. As all eigenvalues are either 0 or 1, a single iteration is required. In practice, due to roundoff errors, a few iterations are required to converge. The initial matrix X is chosen as the first k columns of C with the largest norms. The subspace iterations stops if the norm of the off–diagonal block is close to the machine unit roundoff or some user defined precision ϵ.

In the SD&C algorithm only two computationally intensive kernels are used, one for the QR factorization and a second for matrix multiplication. Since these two kernels asymptotically minimize both bandwidth and latency costs [112], i.e. communication, the SD&C algorithm minimizes the communication as well. By choosing the splitting points with care, so that at each iteration of the recursive procedure subproblems A_1 and A_2 are approximately of the same size, the computational cost rapidly decreases with the iteration count. In such case, even if the recursion is applied till scalar level, the average cost of the full solution of the eigenproblem requires about $27n^3$ flops. The flop count of each QDWH iteration (4.31) is $26/3n^3$, considering all matrices are dense and symmetric. However, if the symmetry of the matrices X_{k+1} and the product of $Q_1 Q_2^T$ is exploited as well as the sparsity structure in the QR factorization of $2n \times n$ matrix, the flops count drops to only $5n^3$ flops.

Chapter 5

Dense Eigensolvers for Hybrid Architectures

Large-scale dense eigenvalue problems are in the focus of this research because of their extremely high computational cost and vast amount of memory required for their efficient solution. The solution of such eigenproblems asks for systems with very high computational performance, capable of tackling such high demanding problems. Traditionally, distributed-memory systems and supercomputers were used to solve large-scale eigenvalue problems. Such systems are equipped with a large number of computational units that can deliver the computational power needed for the solution of these problems. However, such systems are expensive to acquire and maintain, especially in terms of energy consumption.

A cheaper and eligible alternative to large expensive systems are CPU-GPU hybrid architectures, which have recently evolved into powerful computing systems. These systems, equipped with one or two multi-core processors and one or more general-purpose GPUs accelerators, have a very high theoretical peak performance that can even overcome the performance of some moderate-scale systems. Because of their high performance, a considerable research effort has been already put in the development of solvers for dense symmetric eigenproblems and the kernels needed for their execution. However, the main disadvantage of all these efforts is that they cannot solve problems that require an amount of memory exceeding the available GPU memory. Therefore, although the GPU-based hybrid platform can deliver enough performance, these large-scale problems cannot be solved by using the GPU accelerators.

The goal of this research is to develop novel high performance algorithms that can efficiently solve large-scale eigenproblems on GPU-based hybrid platforms, even if the problem data are too large to entirely fit into the GPU memory, i.e. are out-of-core from the perspective of the GPU memory. The novel algorithms presented here are based on the three specific methods presented in Chapter 4: multi-stage reduction to tridiagonal form, the Krylov-subspace method, and the spectral divide–and–conquer algorithm.

The chapter is organized as follows. Section 5.1 describes the new GPU out-of-core algorithm based on the multi-stage reduction to tridiagonal form. Section 5.2 shows the two versions of modified Arnoldi-type algorithms that off-load the computationally-intensive matrix-vector multiplications on the GPU. Finally, Section 5.3 introduces the out-of-core spectral divide–and–conquer algorithm.

5.1 Multi-stage reduction to tridiagonal form

The direct eigensolvers, namely, one–stage and multi–stage reduction, are mostly used for small problems and when all eigenvalues and eigenvectors are computed. However, when solving large-scale eigenvalue problems, their performance drastically drops with the increase of problem size, due to the high arithmetic cost of those methods ($\mathcal{O}(n^3)$) as well as large storage requirements. One advantage of the eigensolvers based on the reduction to tridiagonal form is that the obtained tridiagonal matrix is negligible in terms of storage requirements ($3n$ compared to n^2 needed for a full dense matrix). Furthermore, instead of the costly computation of the eigenvalues of a dense matrix, the eigenvalues of tridiagonal matrix are computed by applying fast specialized solvers, e.g. MRRR. This feature yields a huge benefit when solving large dense eigenproblems that can not fit into the GPU memory and whose performance is bounded by the memory transfers between the main memory and GPU memory. Thus, the reduction to tridiagonal form remains an out-of-core problem for the GPU, while finding the eigenvalues of tridiagonal matrix, because of the low memory requirements, is a standard in-core problem. Therefore, the main computational bottleneck is not the computation of the eigenpairs but the reduction from dense to tridiagonal form.

Two approaches for the reduction to tridiagonal form were presented in Section 4.2. The first approach is the one-stage reduction and requires $4/3n^3$ flops, half of which can be cast in terms of communication efficient level-3 BLAS operations [112]. The remaining $2/3n^3$ flops are spent on symmetric matrix-vector product, a memory-bound operation that renders the global algorithm quite inefficient on current general-purpose architectures. The memory bottleneck is partially alleviated in MAGMA [8] project, by off-loading the level-2&3 BLAS operations on the GPU while at the same time partially overlapping the memory transfer with the GPU execution. Although the GPU implementation attains much higher performance than the multi-core implementation, the main drawback remains as half of the operations are still performed in terms of level-2 BLAS operations. Furthermore, the existing MAGMA implementations cannot solve problems whose problem size exceeds GPU memory.

The second approach is the multi-stage reduction to tridiagonal form described in Section 4.2.2. This approach is almost entirely cast in terms of level-3 BLAS operations that better preserve data locality and therefore decrease the number of memory transfers between the local and global memories of the systems. For the out-of-core GPU algorithms that implies less memory transfers between the main memory and GPU memory resulting in a better utilization of the GPU. The multi-stage reduction algorithm consists of three main steps: the reduction of the dense matrix to band form, successive reduction to narrower band form, and the reduction of band matrix to tridiagonal form. The first two steps are rich in level-3 BLAS routines, while the third step is strictly sequential and level-2 BLAS oriented. The first stage, as the most computationally-intensive part of the reduction, can be easily off-loaded to the GPU [16]. In this GPU implementation, the two-sided update of the trailing submatrix, step 4.16 (Subsection 4.2.2), is performed on the GPU while the other steps are executed on the multi-core CPU. However, this approach can not deal with the out-of-core problems, from the point of view of GPU, as it considers that the problem can fit into the GPU memory. Highly optimized GPU kernels for basic operations such as matrix multiplication, QR factorization and 2-sided update can be found in MAGMA and CUBLAS [113] libraries.

The multi-stage eigensolver for the OOC problems from the GPU perspective is similar to Algorithm 4.2. There, steps 1, 2 (for-loop) and 6 have to be redesigned so that they can handle out-of-core problems on GPUs. Generally, an out-of-core algorithm cannot attain better performance that its in-core counterpart. Therefore, the idea is to transform the out-of-core problem into the in-core as soon as possible. In the multi-stage reduction the first step, i.e. the reduction to band form, is always performed out-of-core; however, the subsequent steps, successive reductions or direct reduction to tridiagonal form, can be made in-core by carefully deciding on the output bandwidth for the last stage so that the obtained matrix in the band form can always fit into GPU memory.

Finally, the last step, i.e. the reduction from band to tridiagonal form, is rich in memory-bounded level-2 BLAS operations that exhibit very low operational intensity. In such operations, the communication between the CPU and GPU can not be efficiently overlapped by applying any out-of-core technique since the main bottleneck is memory bandwidth, not the latency. Thus, the reduction from band to tridiagonal matrix form is performed entirely on the CPU. However, in the case the eigenvectors are required, the accumulation of the orthogonal factor produced by this step can be postponed, and applied by blocks in order to increase the operational intensity. Therefore, the accumulations can be entirely cast in terms of matrix-matrix multiplications and can be performed by applying any of the highly-tuned implementations from CUBLAS or MAGMA computational libraries.

5.1.1 Reduction to band form

The algorithm for the reduction of dense symmetric matrix to band form for problems that exceed the GPU memory, presented in [36], is based on the SBR toolbox routine SYRDB, as presented in Subsection 4.2.2 and illustrated in Figure 4.1. In order to decrease the number of memory transfers, and to increase data locality in the GPU memory, the algorithmic block size is set to be large enough so that the GPU memory is kept full and the GPU stream processors are busy. However, the algorithmic block size can not be larger than the required bandwidth w. Therefore the algorithmic block size is set to be equal to w. That results in the sub-matrix A_1 as in Figure 4.1 being void so that only the sub-matrix A_2 has to be updated. The algorithm is composed of two main steps, the QR factorization of slab A_0 and the two-sided update of A_2, as illustrated in Figure 5.1.

Algorithm 5.1 illustrates the basic reduction of a dense symmetric matrix to tridiagonal form. Matrix A is divided into slab A_0, with w columns, and sub-matrix A_2, where w denotes the algorithmic block size as well as the targeted bandwidth. For each slab A_0, the QR decomposition is computed and the factors W and Y of the WY-representation (2) constructed. The upper triangle of A_0 is non-zero (factor R) while all other entires are zeroed, bringing the first w columns of A into the required band form. The sub-matrix $A_2 = A(j+w\colon n, j+w\colon n)$ that is to the right of the current slab A_0 is updated from both sides by the orthogonal matrix $Q = I + WY^T$ such that $A_2 := Q^T A_2 Q$.

Algorithm 5.1 OOC reduction to band form

```
Input: Real symmetric matrix $A \in \mathbb{R}^{n\times n}$, bandwidth $w$
Output: Matrix $A$ is overwritten with the resulting band-matrix
for $j = 1 : n-1 : w$ do
    QR decomposition of $A_0 = A(j+w\colon n, j\colon j+w-1) \rightarrow W, Y$
    Two-sided update: $A_2 := (I + WY^T)^T A_2 (I + WY^T)$
end for
```

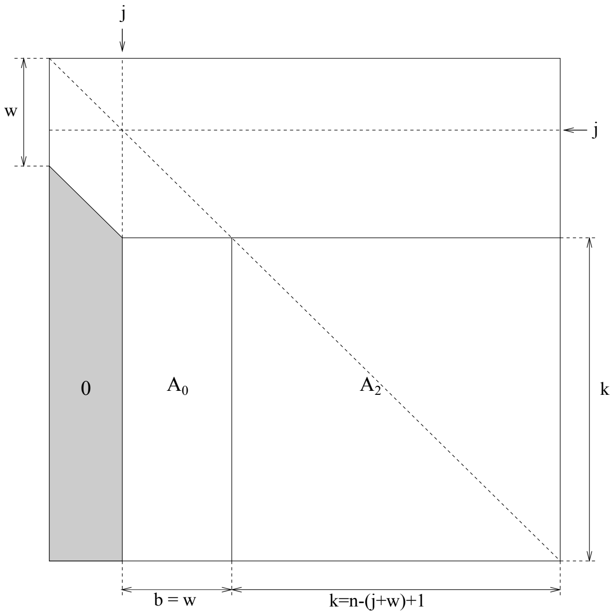


Figure 5.1: OOC reduction of dense symmetric matrix to band form.

The dimension of the panel A_0 is chosen so that it can fit into the GPU memory while the trailing sub-matrix A_2 exceeds the capacity of the GPU memory. While advancing through the matrix column-blocks (slabs), the size of the submatrix A_2 decreases and the problem eventually becomes in-core. As mentioned before, by choosing the slab-width to be equal to the band, the update of slab A_1 performed in the original SBR algorithm is omitted. This trick increases the size of the factors W and Y and assures better computational intensity that finally yields better performance for the overall algorithm.

Furthermore, as discussed before, it is always more efficient to solve problem in-core than out-of-core because an out-of-core algorithm will never attain higher performance than its in-core counterpart. Thus, in the reduction to band form, the bandwidth w can be chosen such that the following steps (e.g. reduction from band to narrower band form) can be completely performed as in-core problems. This can be achieved if the matrix is re-arranged in to the band storage representation that requires only $n \times w$ memory spaces instead of relying on standard storage representation requiring n^2 memory spaces.

The algorithm is dominated by two computational routines: the QR decomposition (2) and the two-sided updated (3). The QR decomposition operates on a large slab that fits into the GPU while at the same time it constructs factors W and Y on the GPU requiring two additional storage spaces that can not fit the GPU. A novel hybrid slab-oriented QR decomposition is presented next.

Hybrid (in-core) QR decomposition

Let us consider the QR factorization of the $k \times w$ matrix A_0 and the construction of the corresponding factors W and Y of the same dimension; see Figure 5.1. For clarity, in the following discussion W_i stands for $W(i\colon k, i\colon i+\tilde{b}-1)$, Y_i for $Y(i\colon k, i\colon i+\tilde{b}-1)$ and A_{0_i} for $A_0(i\colon k, i\colon i+\tilde{b}-1)$, where $\tilde{b}$ is the algorithmic block dimension and k number of rows of A_0. The prefix "d" identifies matrices that are stored in the GPU memory; all operations on these matrices imply the execution on the GPU without further explicit mention of it. For simplicity, we also assume that w is an exact multiple of $\tilde{b}$.

Our implementation of the hybrid QR decomposition is illustrated in Algorithm 5.2. The procedure operates on column blocks of width $\tilde{b}$, and at the beginning, A_0 resides in both main and GPU memory (dA). Let's consider that the first $i-1$ columns of the (rectangular) matrix A_0 have already been factorized. At the beginning of each iteration, the columns $dA(1:k, i:i+\tilde{b})$ are copied back to CPU (3). Then, the QR factorization on the panel A_{0_i} is performed (4) (factor R in stored in the upper triangle of A_{0_i}) and the factors W_i and Y_i generated (5) on the CPU. The entries of $W(1\colon i-1;\ i\colon i+\tilde{b}-1)$ and $Y(1\colon i-1, i\colon i+\tilde{b}-1)$ are set to zero (6) as the current factors do not change these upper positions. Then, W_i and Y_i are transferred to the GPU (7) to update from the left the submatrix $dA(i\colon k, i+\tilde{b}\colon k)$ (8). Finally, the matrix Y is simply updated by appending the obtained columns of Y_i and dW_i is updated by applying all the previous factors stored in dW (9) and then appended to dW. The total cost of the algorithm is $4(k^2w - kw^2 + w^3/3)$ flops. At the end of the for-loop (2) the updated A_0 and Y reside on both CPU and GPU, while the factor dW is stored on the GPU and has to be transferred back to the CPU once the algorithm is finished (11).

Algorithm 5.2 Hybrid QR decomposition

Input: Real matrix $A_0 \in \mathbb{R}^{k\times w}$, algorithmic block size $\tilde{b}$

Output: Matrix A_0 overwritten with factor R. Factors $W, Y \in \mathbb{R}^{k\times w}$ of the WY-representation

```
 1: copy $A_0 \rightarrow dA$
 2: for $i = 1 : w : \tilde{b}$ do
 3:     copy $dA(i\colon k, i\colon i+\tilde{b}-1) \rightarrow A_{0_i}$
 4:     compute QR factorization of block $A_{0_i}$ (xGEQRL)
 5:     construct $W_i$ and $Y_i$ (xGEWYG)
 6:     $W(1\colon i-1, i\colon i+\tilde{b}-1) := 0$, $Y(1\colon i-1, i\colon i+\tilde{b}-1) := 0$
 7:     copy $W_i \rightarrow dW_i$, and $Y_i \rightarrow dY_i$
 8:     $dA(i\colon k, i+\tilde{b}\colon w) := (I + dW_i dY_i^T)^T dA(i\colon k, i+\tilde{b}\colon w)$
 9:     two-sided update: $dW_i := dW_i + dW(i\colon k, 1\colon i-1) dY(i\colon k, 1\colon i-1)^T dW_i$
10: end for
11: copy $dW \rightarrow W$
```

The QR decomposition in Algorithm 5.2 illustrates the hybrid CPU-GPU algorithm that off-loads arithmetically-expensive tasks, such as the update of dA and dW that are based on matrix multiplications, on the GPU, while low operationally-intensive, i.e. those with a small ratio between flops and data transferred, are performed on the CPU. Two versions of the hybrid QR factorizations are developed depending on the number of data transfers and the amount of the workspace required in the GPU memory.

Variant QR-1

This version requires memory space on the GPU to hold two matrices, dA and dW, of size $k \times w$ (with $k \leq n$), and an additional workspace dZ of size $w \times w$. At each iteration of Algorithm 5.2, the factor dY_i is actually stored overwriting the entries of dA_i since, once dA_i is copied to A_{0_i} it is not referenced anymore. The updated matrix A_{0_i} is now stored on the CPU so that the associated space on the GPU can be reused for dY_i. The update of the rest of dA (8) is split into two parts, with each being performed via a single invocation of the GEMM kernel. Specifically, dZ is used as workspace for the first matrix product $dZ := dW_i^T dA(i\colon k, i+\tilde{b}\colon w)$ while, in the second product, $dA(i\colon k, i+\tilde{b}\colon w) := dA(i\colon k, i+\tilde{b}\colon w) + dY\, dZ$. The application of the previous transforms to the slab W_i can also be divided into two parts, performed by two calls of the GEMM kernel on GPU such that $dW(i\colon k, 1\colon i-1)$ and $dY(i\colon k, 1\colon i-1)$.

The advantage of this version is that it performs a reduced number of data transfers to/from the GPU. However, it requires that both matrices dA and dW fit into the GPU memory. As the matrix dimension n grows, the bandwidth w has to be reduced so that the slab A_0 fits into the GPU memory. For very large matrices, this results in operating with tall column blocks of A_0 as well as narrow factors W_i and Y_i, which eventually decreases the ratio between computation and data transfers performed, and impairs the overall performance. Since the leading dimension of A_0 depends on the dimension k, the limiting factor is the amount of the GPU memory.

Variant QR-2

This version requires less storage space on the GPU: a workspace of size $k \times w$ for dA, a panel dY of size $k \times \tilde{b}$ for a single column block of Y_i, and an additional workspace dZ of size $w \times w$. The factor dW_i is now stored overwriting dA once the entries of this panel have been copied to A_{0_i} (in the previous version we did this for dY_i).

The update of the rest of the matrix dA, Algorithm 5.2 step (8), is performed in the same way as in the Variant QR-1 using dZ as a temporary workspace. As the matrix Y is not stored in the GPU memory, the application of the previous transforms to the slab W_i differs from that of the previous version. In particular, this computation is split into the following two parts:

$$\begin{aligned} dZ &:= dY(i\colon k, 1\colon i-1)^T\, dW_i, & (5.1)\\ dW_i &:= dW_i + dW(i\colon k, 1\colon i-1)\, dZ, & (5.2) \end{aligned}$$

where dW is stored in $dA(1\colon k, 1\colon i+\tilde{b}-1)$. The update (5.1) is performed by copying the column blocks $Y(i\colon k, p\colon p+\tilde{b}-1)$ to dY and then computing:

$$dZ(p\colon p+\tilde{b}-1, :\,) := dY^T\, dW_i,$$

for $p = 1, \tilde{b}+1, 2\tilde{b}+1, \ldots, i-\tilde{b}$. Once all blocks of Y have been processed and dZ has been computed, the update in (5.2) can be performed by invoking a single call of to GEMM on GPU.

The advantage of this approach is that it requires less memory space on the GPU than Variant QR-1, enabling the use of a larger bandwidth w. However, the workspace on the GPU is still proportional to the number of rows k and, as the problem size grows, the bandwidth w has to be reduced as well. Compared with the previous version, the drawback of Variant QR-2 is that it requires more transfers to/from the GPU as, for each p, one panel of size $(k-i) \times \tilde{b}$ is copied to the GPU, which results in additional overhead. Still, this version is expected to exhibit an acceptable computation/transfer ratio as long as the bandwidth remains large.

OOC two-sided update of the symmetric matrix

The second computational routine of the out-of-core reduction of a dense symmetric matrix to band form is the two-sided update of the $k \times k$ symmetric trailing submatrix

$A_2 = A(i+w\colon n, i+w\colon n)$, Algorithm 5.1 step (3). The update $A_2 = (I+WY^T)^T A_2 (I+WY^T)$ can be performed in 4 steps:

$$\begin{aligned} &(\textsc{symm}) & X_1 &:= A_2 W, && (5.3)\\ &(\textsc{gemm}) & X_2 &:= \frac{1}{2} X_1^T W, && (5.4)\\ &(\textsc{gemm}) & X_3 &:= X_1 + Y X_2, && (5.5)\\ &(\textsc{syr2k}) & A_2 &:= A_2 + X_3 Y^T + Y X_3^T. && (5.6) \end{aligned}$$

Consider a partitioning of matrix A_2 in Figure 5.1 into square blocks of size $\hat{b} \times \hat{b}$, with the algorithmic block size $\hat{b}$ being fixed so that a "few" of these blocks fit into the GPU memory (the number of blocks will be discussed later). Moreover, the size $\hat{b}$ depends only on the available GPU memory and is independent of the dimension of the problem A_2. That feature gives the algorithm the advantage to scale regardless of the matrix dimension. Theoretically, the only limiting factor is the capacity of the main memory in which the problem originally resides.

For simplicity, let's denote blocks $A_2(i\colon i+\hat{b}-1, j\colon j+\hat{b}-1)$ with $A_{2_{ij}}$, $W(i\colon i+\hat{b}-1, 1\colon w)$ and $Y(i\colon i+\hat{b}-1, 1\colon w)$ with W_i and Y_i, respectively, and $X(i\colon i+\hat{b}-1, 1\colon w)$ with X_i. In the following text, two variants of the out-of-core GPU-accelerated two-sided updates are presented. These variants differ in the GPU storage requirements and the amount of data transferred between the main and GPU memory, which has direct impact on the performance.

Variant Update-1

The first variant of the two-sided update aims at decreasing the number of memory transfers between CPU and GPU by keeping one of the factors, W or Y, and the workspace matrix on the GPU. Since the factors W and Y are required in all four steps and are relatively small when the bandwidth is small, it is cheaper to keep them in the GPU memory and thus reduce unnecessary copying of small memory chunks to/from the GPU. The drawback of this variant becomes obvious with the increase of the dimension k, resulting in large tall factors W and Y kept in GPU memory which decrease the block size of A_2. The smaller block size results in more memory transfers and lower ratio between flops performed and data copied on the GPU. The workspace requirements on GPU are two panels of size $k \times w$ for keeping W/Y and the workspace matrix of $\hat{b} \times \hat{b}$.

Consider first the computation of X_1 in Equation (5.3). This step operates with factors W and Y in the GPU memory and the out-of-core A_2 divided into square blocks of order $\hat{b}$. The implementation is given in Algorithm 5.3 where, for simplicity, X_1 is denoted as an X. Note that A_2 is symmetric and only the lower triangle is referenced. Furthermore, once the block $A_{2_{ji}}$ is copied to the GPU, we can update blocks X_i and X_j (8)-(9) of X because blocks $A_{2_{ji}}$ and $A_{2_{ij}}$, i.e $A_{2_{ji}}^T$, respectively, participate in their update. The diagonal blocks are updated in advance (3). Thus, to update X_1 the lower block triangle of A_2 and factor W have to be transferred only once to the GPU.

After the first step, both matrices W and X_1 are stored in the GPU memory, while A_2 is unchanged. The next two steps, corresponding to Equations (5.4) and (5.5), can be performed completely as in-core GPU problems via two calls to the GEMM routine. No additional copy is required in step (5.4) since X_1 and W are already on the GPU and the intermediate matrix X_2 (of dimension $w \times w$) is stored in the workspace dA. The block

Algorithm 5.3 OOC variant 1: $X_1 := A_2 W$

```
Input: Real symmetric matrix $A_2 \in \mathbb{R}^{k \times k}$, factor $W \in \mathbb{R}^{k \times w}$
Output: Real matrix $X_1 \in \mathbb{R}^{k \times w}$
for $i = 1 : n : \hat{b}$ do
    Copy $A_{2_{ii}} \rightarrow dA$
    $dX_i := dA\, dW_i$
end for
for $i = 1 : n : \hat{b}$ do
    for $j = i + \hat{b} : n : \hat{b}$ do
        Copy $A_{2_{ji}} \rightarrow dA$
        $dX_i := dX_i + dA^T\, dW_j$
        $dX_j := dX_j + dA\, dW_i$
    end for
end for
```

size $\hat{b}$ is chosen so that $w <= \hat{b}$. In step (5.5) the workspace dW is re-used for keeping Y and the resulting X_3 is written into dX requiring not additional workspace on the GPU.

The last step (5.6) is tackled out-of-core by dividing A_2 into blocks and updating them on the GPU as follows:

$$A_{2_{ij}} := A_{2_{ij}} + X_{3_i} Y_j^T + Y_i X_{3_j}^T. \tag{5.7}$$

Because of the symmetry of A_2 only the lower block triangular has to be copied to the GPU. Since only one block of A_2 can fit into the GPU memory, after each update the block is returned to the main memory and the next block transferred to GPU and updated.

The advantage of this version is that we keep W/Y and one of X_1, X_2 or X_3 in the GPU memory, and only one block A_{ij} is copied to/from the GPU. This significantly reduces the number of memory transfers and enables data reuse once it resides on the GPU. However, with the increase of w, the block size is reduced which leads to a loss of efficiency as the number of copy transfers grows and the algorithm operates on smaller blocks.

Variant Update-2

The second variant tries to overcome the scalability limitations of variant Update-1 that occur with the increasing dimension k. The main limitation of the previous variant is that full-size auxiliary matrices X_1 and X_3 are kept in the GPU memory. That is partially alleviated by dividing X_1/X_3 into blocks and keeping only one of them in the GPU memory at the time, but with the cost of increased number of data transfers. Hopefully, the memory transfers can be overlapped with the computation as the block size remains large enough.

This version requires one block of size $\hat{b} \times \hat{b}$ to keep block $A_{2_{ij}}$, one slab of size $k \times w$, and one additional slab of size $\hat{b} \times w$. Therefore, this variant requires less storage than the previous one, concretely $(k - \hat{b}) \times w$ less data, thus the block size $\hat{b}$ can be larger. The drawback of the new variant is in increased number of memory transfers because full slabs W/Y are not kept permanently in GPU memory while X_1 and X_3 are transferred by blocks.

In step (5.3) of Variant Update-2 only one block ($\hat{b} \times w$) of X_1 is stored on GPU at the time. Thus, the advantage of updating two blocks of X_1 with the current $A_{2_{ij}}$ block in the GPU memory is not possible anymore. The new variant of the algorithm is listed in Algorithm 5.4. Each block $A_{2_{ip}}$ is transferred to the GPU (not only the diagonal and lower triangle blocks). That results in twice as much copying than in Variant Update-1. In each pass of i-loop a block of $\hat{b}$ rows of X_1, stored in positions dX, is updated (8). When this update is completed, block X_{1_i}, is returned to the main memory. In order to store the updated X_1 an additional workspace of size $k \times w$ is required. At the end of these steps, matrix W is stored on the GPU while X_1 and A_2 reside in the main memory.

Algorithm 5.4 OOC variant 2: $X_1 := A_2 W$

Input: Real symmetric matrix $A_2 \in \mathbb{R}^{k \times k}$, factor $W \in \mathbb{R}^{k \times w}$
Output: Real matrix $X_1 \in \mathbb{R}^{k \times w}$

```
 1: Copy $W \to dW$
 2: for $i = 1 : n : \hat{b}$ do
 3:     Copy $A_{2_{ii}} \to dA$
 4:     $dX := dA\, dW_i$
 5:     for $j = 1 : n : \hat{b}$ do
 6:         if $j \neq i$ then
 7:             Copy $A_{2_{ji}} \to dA$
 8:             $dX := dX + dA\, dW_i$
 9:         end if
10:     end for
11:     Copy $dX \to X_1(i\colon i+\hat{b}-1, :\,)$
12: end for
```

In step (5.4) X_2 is updated as:

$$X_2 := X_2 + \frac{1}{2}\, dX_{1_i}^T\, dW_i.$$

where $i = 1, \hat{b}+1, 2\hat{b}+1, \ldots, k-\hat{b}$ is the number of blocks X_1 is divided into. Factor W is already on GPU and X_1 is transferred by blocks. As in the previous variant, dA is used to keep X_2 and no additional workspace is required. The only requirement is that $\hat{b} \geq w$, so that dA is large enough to keep matrix X_2.

Step (5.5) is also performed out-of-core since matrices X_1 and X_3 do not fit into the GPU memory. Matrix X_1 is brought into the GPU memory (dX) by blocks $X_1(i\colon i+\hat{b}-1, 1\colon w)$ while Y is copied once into the storage space dW. Matrix X_2 is already stored in dA from the last step. The update of the block $X_{3_i} = X_3(i\colon i+\hat{b}-1, 1\colon w)$ is listed in Algorithm 5.5.

Algorithm 5.5 OOC variant 2: $X_3 = X_1 + Y X_2$

```
Copy $Y \to dW$
for $i = 1 : k-\hat{b} : \hat{b}$ do
    $dX := dX + dW_i\, dA$
    Copy $dX \to X_{3_i}$
end for
```

At the end of each iteration (4), the computed block $X_3(i\colon i+\hat{b}-1, 1\colon w)$ stored in dX is returned to the main memory. X_3 is then transferred into the same workspace as X_1 (as this block is not required anymore).

In the last step (5.6), blocks $X_i = X_3(i\colon i+\hat{b}-1, 1\colon w)$ and $X_j = X_3(j\colon j+\hat{b}-1, 1\colon w)$ are required to update each block $A_{2_{ij}}$. Once the block is updated it is returned to the main memory. Thus, three memory transfers to and one transfer from the GPU memory are required to update one block $A_{2_{ij}}$. Algorithm 5.6 illustrates the update of one block $A_{2_{ij}}$. Note that Y that is already stored in dW from the previous step can be re-used with no additional copying.

Algorithm 5.6 OOC variant 2: Symmetric 2k update

```
Copy $A_{2_{ij}} \to dA$
Copy $X_i \to dX$
$dA := dA + dX\, dW_j^T$
Copy $X_j \to dX$
$dA := dA + dW_i\, dX^T$
Copy $dA \to A_{2_{ij}}$
```

5.1.2 Reduction to narrower band form

The second stage of the multi-stage approach is also the computationally-intensive phase of the reduction rich in level-2 operations that can efficiently exploit the GPU. Since the width of the band is chosen in the previous stage to be small enough so that the band matrix fits into the GPU memory, the problem becomes in-core regarding the GPU. Although the reduction to narrower band form is now GPU in-core, it is still expensive, especially when the eigenvectors are required, because of the arithmetically-intensive bulge-chasing phase that is used in the reduction. The algorithm is based on the SBR routine SBRDB that reduces a band matrix to narrower band form.

The algorithm operates on the band matrix B, with bandwidth w_1, stored in the band storage representation that requires $w_1 \times n$ storage spaces and is much lower than in the standard representation if $w_1 \ll n$. In the band storage format, the main diagonal is placed in the first row, and the following subdiagonals are stored in successive rows of this matrix. In other words, if $\hat{B}$ is a band storage representation of the band matrix B, then element $B(i,j)$ is saved in the position $\hat{B}(1+i-j, j)$.

Assume that we want to reduce a band matrix B_{w_1} of bandwidth w_1, into a band matrix B_{w_2}, with bandwidth $w_2 < w_1$. The goal is to annihilate $d = w_1 - w_2$ subdiagonals of B_{w_1}, i.e. subdiagonals from w_2 to w_1. The procedure is based on the “annihilate-and-chase” strategy and employs Householder transforms for this purpose. The reduction is done by blocks, annihilating one block of subdiagonals at a time as illustrated in Figure 5.2. The idea of the out-of-core multi-stage reduction was to perform the reduction from dense to band form out-of-core, while the following steps are performed as in-core GPU problems. Therefore, the bandwidth w_1 was chosen such that the matrix $\hat{B}$ in band storage representation can fit into the GPU memory.

At the beginning of the reduction, matrix $\hat{B}$ is stored in both the main and GPU memory. The first d outermost subdiagonals from the first n_b columns, block designated as “QR” in Figure 5.2 (left), are annihilated on the CPU by performing the QR factorization and the transformation $Q = I + WY^T$ in the WY-representation generated (2). The

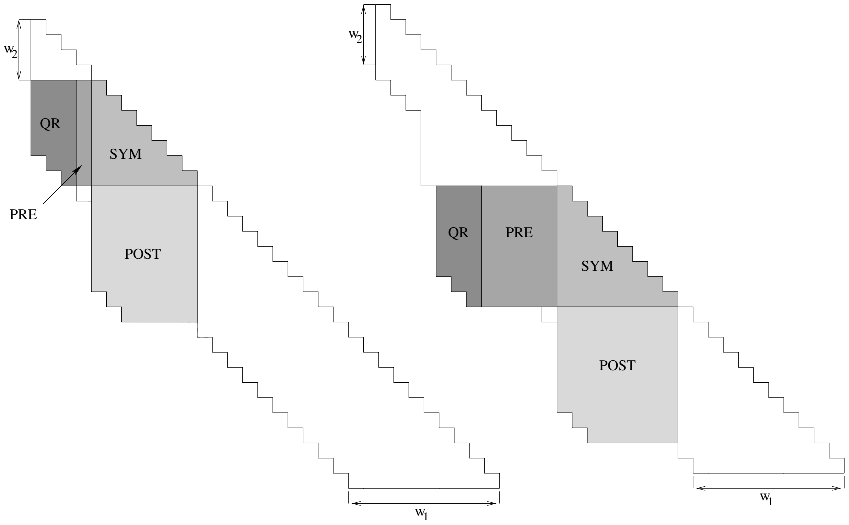


Figure 5.2: Left: Annihilation of the outermost sub–diagonals and updating the rest of the matrix, Right: bulge chasing part of the first fill-in block

transforms are then transferred to the GPU and applied from the left to block “PRE” (3). Furthermore, the transforms are applied from both sides to the symmetric block “SYM” (note that only the lower triangular is updated), and from the right to block “POST” (5) on the GPU.

Algorithm 5.7 In-core GPU reduction to narrower band form

```
Input: Band matrix $B_{w_1} \in \mathbb{R}^{w_1 \times n}$
Output: Matrix $B_{w_2}$
1: Copy block “QR” GPU → CPU
2: Compute QR on the block “QR” and generate factors $W$ and $Y$ (in-core CPU)
3: Copy $W, Y$ CPU → GPU, update block “PRE”(in-core GPU)
4: for $k = i + w_2 : n : w_1$ do
5:     Update blocks “SYM” and “POST” (in-core GPU)
6:     Copy fill-in block “QR” GPU → CPU
7:     Compute QR and generate associated $W, Y$ (in-core GPU)
8:     Copy $W, Y$ CPU → GPU, update block “PRE” (in-core GPU)
9: end for
```

The first n_b columns are brought to band form with bandwidth w_2. However, note that in the block “POST” the first d subdiagonals below the band (i.e. subdiagonal w_1) are filled-in with non-zero elements. At this stage, the first n_b annihilated columns are located on the CPU, while the rest of the updated matrix $\hat{B}$ are on the GPU. To restore the $\hat{B}$ to band form, the annihilation and chase step is performed to remove these non-zero elements below the band. The bulge-chasing step is illustrated in Figure 5.2 (right). First, the n_b updated columns, block “QR”, is returned from the GPU to CPU (6). Then, the QR decomposition is applied on “QR” to annihilate the fill-ins of the d subdiagonals below the band and the corresponding transforms generated (7) and Figure 5.2 (right) on CPU. Then the current transforms are copied to the GPU and applied to the blocks marked with “PRE” (8), “SYM”, and “POST” (5). This generates new fill-ins below the second “POST” block, Figure 5.2 (right), and the process is repeated, chasing down the fill-ins until they are pushed off the matrix. Upon completion of the bulge-chasing phase, a new block of n_b columns is reduced to band form w_2 and the process starts with retrieving the updated block "QR" to the main memory (1). Upon completion, the band matrix B_{w_2} is stored in the main memory. Additional storage is required in the GPU to keep the W and Y factors as well as a $b_1 \times n$ workspace for the fill-ins generated during the successive annihilate-and-chase steps.

5.2 The Krylov subspace method

The Krylov subspace approach was traditionally used when only a subset of the eigenvalues and eigenvectors was required, or in case the problem was structured or sparse. This approach is particularly efficient for the solution of large sparse eigenproblems in which the sparsity structure of the matrix can be exploited and the arithmetic complexity decreased.

For the solution of symmetric eigenproblems, the Krylov subspace method employs a Lanczos-based method [20] to iteratively construct an orthogonal basis of the Krylov subspace. Upon convergence, the eigenvalues of a resulting tridiagonal matrix H_k, Algorithm 4.6, of reduced dimension $k \geq 2s$ (with s being the number of sought-after eigenvalues), approximate certain eigenvalues of A. Computing the eigenvalues and eigenvectors from H_k adds little cost to the computations since in practice the number of sought-after eigenvalues is much smaller than n. For symmetric matrices, this method exhibits fast convergence, low computational cost ($\mathcal{O}(n^2)$ flops per iteration) and requires minor additional storage space. The in-core implementation of Lanczos method [94] on multi-core systems outperforms the multi-stage approaches even for dense eigenproblems especially when hardware accelerators, such as GPUs, are used [37].

The Krylov subspace-based methods are completely cast in terms of level-2 BLAS operations thus exhibiting low computational cost compared to the direct eigensolvers. However, when solving large-scale eigenproblems, the main drawback of the Krylov subspace based methods (e.g. Lanczos) is that a significant part of their computations is spent in terms of memory-bounded matrix-vector product. For a matrix of size $n \times n$, this kernel roughly performs $2n^2$ flops on n^2 data, i.e. the ratio between flops performed and data transfers is $\mathcal{O}(1)$. When operating with out-of-core data, these methods are limited by data movement and therefore attain very low performance. Thus, even if their computation is accelerated by exploiting a GPU better performance will not be achieved as the method is bounded by data movements over a relatively slow PCIe bus.

Although level-2 BLAS routines are memory-bound, and cannot improve the performance if data are out-of-core from the GPU perspective, they outperform multi-core implementations by an order of magnitude if data are in-core. Because the majority of computations in Lanczos methods are cast in terms of symmetric matrix-vector products, they can greatly benefit from using the GPUs as long as data are in-core from the viewpoint of the GPU. The GPU-based Lanczos algorithm has been proved not only to be faster than its multi-core variants but also to be competitive with GPU-based direct eigensolvers even for dense symmetric problems when a small subset of eigenvalues is required [37].

5.2.1 Introduction to the algorithm

The Krylov subspace methods described in Section 4.3 with aim to solve the standard eigenproblem. However, this thesis is focused in the solution of generalized symmetric eigenvalues problems for very large-scale instances. Although the ARPACK package provides routines for the solution of generalized eigenproblems, these cannot exploit the GPU performance and, moreover, cannot take advantage of different approaches that can ultimately transform the Krylov subspace method into an efficient algorithm even for the solution of dense generalized eigenproblems.

In the direct solvers based on the multi-stage reduction to tridiagonal form, the generalized eigenproblem $AX = BX\Lambda$ was explicitly transformed into the standard eigenproblem $CY = Y\Lambda$, where $C = U^{-T}AU^{-1}$ and U is obtained from Cholesky factorization of $B = U^TU$. A similar approach can be applied to Krylov subspace-based approach. However, an explicit reduction to a standard eigenproblem is expensive, especially for very large problems, since it requires $7/3n^3$ flops. As the computational cost of the Krylov subspace methods is $\mathcal{O}(n^2)$ flops, this explicit transformation could easily overpace the iterative phase of the Lanczos procedure and become the most expensive part.

In the reduction to standard eigenproblem, the Cholesky factorization is initial step that can not be skipped. Fortunately, this stage is known to deliver very high performance on a large variety of HPC architectures, including multi-core processors and GPUs, and its functionality is covered by current numerical libraries (e.g., LAPACK, libflame, ScaLAPACK, PLAPACK, etc.) including some OOC extensions (SOLAR, POOCLAPACK). The construction of C can be performed via two calls to the triangular systems solve (TRSM routine) but the construction can be avoided during Lanczos iterations. This approach removes part of the arithmetic complexity from the first stage of the algorithm, but introduces additional flops into the Lanczos iteration. Depending on whether C is constructed explicitly at the beginning of the method or used implicitly within the Lanczos iterations, two GPU-based in-core Krylov subspace methods are developed in this dissertation: the Explicit Krylov subspace (KE) and Implicit Krylov subspace algorithms (KI).

Both algorithms consist of three parts: transformation from a generalized to standard eigenvalue problem, Krylov subspace based procedure, and back-transformation that recovers the eigenvectors of the generalized eigenproblem from those of the standard eigenproblem. The first two stages are described in the following subsection as part of two variants of algorithm while the back-reduction is skipped, since it can be performed via a single call to a matrix-matrix multiplication.

5.2.2 Explicit Krylov subspace algorithm (KE)

This variant of the algorithm strictly follows the transformation to the standard eigenproblem and explicitly builds matrix C, as illustrated in Equation (4.3) and presented in Algorithm 5.8. The algorithm starts with the transformation to the standard eigenproblem by computing the Cholesky factorization of B (1) and explicit construction of C (2) as in Equation (4.6). The construction can be performed by two calls to the level-3 BLAS kernel TRSM

$$\begin{aligned} A &:= AU^{-1} && (5.8)\\ C &:= U^{-T}A, && (5.9)\end{aligned}$$

with the arithmetic cost of $2n^3$ flops. Because of the symmetry of C, only the lower or upper triangle is stored in the storage space of A that is not used in the subsequent steps. A Lanczos procedure is then applied, Algorithm 5.8 (4) and (5), that returns the eigenvalues and eigenvectors of the standard eigenproblem $CY = Y\Lambda$. The stopping criterion for the Lanczos iterations is the relative accuracy of the Ritz values that approximate the eigenvalues of the standard eigenproblem and is provided at the beginning of the iteration process.

Algorithm 5.8 Explicit Krylov subspace algorithm

```
Input: Input matrices $A, B \in \mathbb{R}^{n\times n}$
Output: $k$ eigenvalues and associated eigenvectors
1: $B = U^T U$
2: $C = U^{-T} A U^{-1}$
3: while not stop criterion do
4:     call SAUPD(...)
5:     $w_{k+1} = C u_k$
6: end while
7: Solve eigenproblem $T$
8: Compute eigenvectors of generalized problem (back-transformation)
```

ARPACK leverages reverse communication where the Lanczos step is performed by ARPACK routine SAUPD (4) while the symmetric matrix-vector multiplications is done by a separate call (5). At each iteration, the vector u_k (as in Algorithm 4.6 (4)) generated by the SAUPD routine is multiplied by C (5) in $2n^2$ flops. The resulting vector w_{k+1} is forwarded to the next call of the SAUPD routine and a new iteration starts. The SAUPD routine implements the Lanczos iteration that is described in the `for` loop of Algorithm 4.6.

The total computational cost of SAUPD is negligible compared with other parts of the algorithm. The computation of u_{k+1} from w_{k+1}, which is required for the next step, only requires a few operations of linear cost in n. The re-orthogonalization (in case it is needed) can be done in $\mathcal{O}(n)$ to $\mathcal{O}(mn)$ flops, for the best and worst case, respectively and can potentially contribute substantially to the total computational time for moderate values of m, where m is the number of sought-after eigenvalues. In case implicit restarts are needed after the Lanczos augmentation step, an additional cost of $\mathcal{O}(nm^2)$ flops per restart is required, as each restart involves the application of the QR iteration to the tridiagonal subproblem T. After the stopping criterion is reached, the converged eigenvalues and the corresponding eigenvectors are returned (7) and the back-transform (8) is performed to obtain the eigenvectors of the starting generalized eigenproblem, Equation (4.7), with an additional cost of n^3 flops.

The memory requirements of the Explicit Krylov approach is $2n^2$ elements for storing both matrices A and B, although after a careful design and re-implementation of some BLAS routines used in this algorithm, only n^2 memory entries are required. After the Cholesky factorization, only the lower or upper triangular matrix is stored instead of full matrix B. Furthermore, the matrix A is symmetric so only half of the matrix needs to be stored. The memory needed to store the eigenvectors is $\mathcal{O}(nm)$ and is negligible as long as the number of sought-after eigenvalues m is small.

5.2.3 Implicit Krylov subspace algorithm (KI)

This variant of the algorithm is similar to the explicit one except that the matrix C is not formed explicitly but is constructed during Lanczos iterations. The complete method is described in Algorithm 5.9. The main difference with the previous variant is that, within each Lanczos iteration, matrix C is implicitly "applied", as illustrated in Algorithm 5.9 (4). This construction can be performed by the following three operations:

$$\begin{aligned} (\textsc{trsv}) \quad y &:= U^{-1}u_k \quad (5.10) \\ (\textsc{symv}) \quad y &:= Ay \quad (5.11) \\ (\textsc{trsv}) \quad w_{k+1} &:= U^{-T}y. \quad (5.12) \end{aligned}$$

The first and the third steps can be performed by invoking the BLAS kernel TRSV and the second step by a single call to the SYMV routine.

In the Implicit Krylov subspace approach no initial cost is payed for the explicit construction of C at the beginning. However, the cost per iteration for computing vector w_{k+1} doubles with respect to the previous case, from $2n^2$ to $4n^2$ flops. Computing the vector u_{k+1} from w_{k+1} at each Lanczos step (SAUPD) requires $\mathcal{O}(n)$ flops, and the aforementioned re-orthogonalization costs $\mathcal{O}(nm)$ flops; in addition, each of the restarting steps performs $\mathcal{O}(nm^2)$ flops. The stopping criterion for the variant KI is the same as for the explicit variant.

Algorithm 5.9 Implicit Krylov subspace algorithm

```
1: $B = U^T U$ (Cholesky factorization of the matrix B)
2: while not stop criterion do
3:     call SAUPD(...) (Implicitly restarted Lanczos algorithms)
4:     $w_{k+1} = (U^{-T}(A(U^{-1}u_k)))$, (implicitly construction of C)
5: end while
6: Calculate eigenvalues and eigenvectors
7: Calculate eigenvectors of the generalized problem (back transformation)
```

The memory requirements are the same as in the case of the explicit variant KE and the amount of $2n^2$ elements. Likewise, the memory requirements can be decreased to n^2 if the symmetry of A and upper/lower triangular form B, are exploited. If we compare the total flops performed by these two variants of the Krylov subspace algorithms, one can notice that the total flops for the explicit variant is $7/3n^3 + k \times 2n^2$ and for implicit is $1/3n^3 + k \times 4n^2$. If we remove from the calculation the parts of the code that are the same for both cases: Cholesky, back transform, and Lanczos steps (SAUPD routine), the number of flops becomes $k \times 2n^2 + 2n^3$ for KE and $k \times 4n^2$ for KI, where k is the total number of Lanczos iterations performed. The performance of the algorithm depends on the number of iterations k. If the $k \ll n$ then KI performs less flops; in the case $k = n$ both versions perform $4n^3$ flops; and in the case $k \geq n$ the KE becomes the algorithm of choice. The last case may occur if a large number of eigenvalues is required or the convergence is slow due to the matrix condition.

5.2.4 Algorithm improvements

The Krylov subspace algorithms based on Lanczos procedures are mostly composed of the memory-bound symmetric matrix-vector products that attain very low performance for large size problems stored in the main memory. The Krylov subspace methods, applied to OOC problems, cannot improve their performance by exploiting the GPU since the bottleneck is not in the number of operations performed but in the communication overhead. However, even memory-bound algorithms can significantly increase their performance by exploiting GPU if their data fits fits into the GPU memory.

In the Krylov subspace method, the straight-forward approach is to exchange all level-2&3 BLAS routines, such as Cholesky, TRSM, TRSV and SYMV, with the calls to the corresponding CUBLAS and MAGMA routines. However, as mentioned before, all CUBLAS and MAGMA routines, expect of a few, cannot solve problems whose required storage space exceeds the GPU memory. Despite these constraints, large-scale eigenproblems can be solved by the GPU-based version of Krylov subspace on GPU. All parts of the Lanczos-based method, except the symmetric matrix-vector multiplication, can efficiently exploit a GPU even if data are out-of-core. Namely, the first step in both Krylov subspace algorithms, explicit KE and implicit KI, is a compute-bound Cholesky factorization. The out-of-core implementations from MAGMA library and some independent research [35] showed that memory transfers can be efficiently overlapped with computations and deliver very high performance on GPUs for this operation. Furthermore, in the explicit Krylov subspace algorithm, matrix C, Equation (5.9) is explicitly constructed via a communication-efficient TRSM routine. This routine can be efficiently implemented to achieve very high performance even if C is out-of-core for the GPU. The construction of C can be easily implemented by applying blocked algorithms that divide matrices C, U and A into square blocks that are processed one at the time on the GPU while minimizing the communication between the main and GPU memories. However, a highly efficient out-of-core general matrix-matrix multiplication on GPU, developed for spectral divide–and–conquer algorithm, Section 5.3, can be used as well but at the cost of a higher arithmetic cost. The computation of eigenvectors of generalized eigenproblem (back-transformations) can also be computed via out-of-core implementations of general matrix-matrix multiplication.

The only memory-bound part that cannot be efficiently processed on GPU, if the data is OOC from the GPU perspective, is the symmetric matrix-vector multiplication. However, the Krylov-subspace algorithms exhibit very fast convergence and low computational cost ($2n^2$ flops per iteration) when only a small subset of the spectrum is required. In particular, if the KE variant is applied for the solution of a large-scale eigenproblem, the total computational time for the iterative part, Algorithm 5.8 (3)–(6), becomes negligible compared with the explicit construction of C. The construction, as presented earlier, can be cast in terms of highly-tunned Cholesky and matrix-matrix multiplication kernels that attain very high performance on the GPU even when C cannot entirely fit into the GPU memory.

5.3 Spectral divide-and-conquer based on QDWH

The state-of-the-art spectral divide–and–conquer (D&C) algorithm solves a symmetric eigenproblem by recursively decoupling it into subproblems using successive invariant subspaces for a subset of spectrum. The development of this algorithm is justified because of its parallelism and reduced communication cost. However, the spectral D&C presents significantly higher arithmetic cost than the direct and Krylov subspace-based eigensolvers. The arithmetic cost of SD&C was drastically reduced by the recent spectral divide–and–conquer algorithm based on QR-based dynamically weighted Halley algorithm (QDWH) [107]. Furthermore, this algorithm achieves lower bounds on communication cost since most of its operations can be cast in terms of level-3 BLAS operations. The QDWH algorithm, which computes the polar decomposition, is based on the QR factorization and matrix-matrix operations that can be efficiently implemented in a communication-optimal manner [112, 114]. Compared it with the iterative Krylov subspace-based approach, the SD&C algorithm based on QDWH is a compute-bound eigensolver and therefore could attain very high performance on GPU even when the problem is out-of-core from the GPU viewpoint [115].

The out-of-core SD&C eigensolver is based on the algorithm presented in Section 4.4 and published in [107] that compute all eigenvalues and the corresponding eigenvectors. If only a subset of all eigenvalues is required, certain modifications are required. The original algorithm recursively reduces the subproblems till scalar level or a pre-defined subproblem size. In the latter case, a standard eigensolver (e.g. one-stage eigensolver) is next applied to obtain the eigenvalues. In other words, the algorithm is called recursively till the matrix becomes diagonal (scalar level) or nearly diagonal (small-size subproblem). Note that if only a subset of eigenvalues is required, then there is no need to go down with the full diagonalization but only one subproblem has to be diagonalized thus reducing the arithmetic cost.

5.3.1 Out-of-core spectral D&C algorithm

The out-of-core spectral D&C algorithm described next is applied when only a small subset of the eigenvalues is required; however, it can be used when all eigenvalues are computed but at increased arithmetic cost. The main idea when tackling an out-of-core problem is to reduce it to an in-core problem as soon as possible and thus to remove the expensive data transfers. The same idea is applied to our novel out-of-core spectral D&C algorithm that decouples problem into subproblems, such that they fit into the GPU memory.

By using the SD&C approach, the problem can be decoupled such that all required eigenvalues, the smallest or largest s, are a subset of the eigenvalues of one of the subproblems. The idea underlying of our out-of-core SD&C is to decouple the out-of-core problem into two subproblems, one of which is small enough to fit into the GPU memory but big enough to superset all the required eigenvalues. Then any eigensolver, direct, iterative or even SD&C can be applied to compute the sought-after eigenvalues in-core.

The main problem is to find a good splitting point σ that divides the original problem into two subproblems, A_1 or A_2, such that one of them contains a superset of the sought-after eigenvalues. Since σ splits the spectrum into two parts, one containing the eigenvalues that are smaller than σ and the other with those that are larger, in order to compute the s largest eigenvalues, σ should be smaller than the s-th largest eigenvalue and larger than s-th smallest eigenvalue if the s smallest eigenvalues is required.

The idea of choosing σ to be a close bound of the s-th eigenvalue gives a subproblem of dimension $k \times k$, where, if the smallest eigenvalues are desired, $k = \lceil \sigma \rceil \geq \lambda_s$ (s smallest eigenvalues). If the number of sought-after eigenvalues s is also small, i.e. $s \ll n$, then the subproblem is small and the out-of-core recursion can be stopped.

Algorithm 5.10 illustrates the out-of-core spectral D&C algorithm based on the QDWH and presented in [115] that computes the smallest s eigenpairs. The algorithm computes partial spectral decomposition of $A = V_s D_s V_s^T$ where D_s and V_s are the sought-after eigenvalues and eigenvectors, respectively, computed directly from the obtained subproblem A_1. Opposite to the algorithm described in Section 4.4, only one recursion step is performed corresponding to the out-of-core decoupling of matrix A. The splitting point σ is chosen (1) so that matrix A_1 contains a superset of sought-after eigenvalues. Then, the polar factor U_p of the matrix $A - \sigma I$ is computed via the QDWH algorithm (2). In case A cannot fit into the GPU memory, QDWH is executed on the GPU. If a good σ is chosen, the computation of the subspace A_1 can be performed entirely in-core. That includes computing only the orthogonal matrix V_1 (3) and the construction of A_1 (4). If the dimension k of subproblem A_1, i.e. number of columns of V_1, is smaller than the number of required eigenvalues s, the algorithm has to be repeated with a larger σ. Given that $s \ll n$, a good selection of σ that provides small k, ensures that A_1 can computed in-core. Therefore, if k is larger than s, any standard eigensolver can be applied to compute the s smallest eigenpairs of the subproblem A_1 (6), for example, the DSYEVR routine from LAPACK. If the eigenvectors are required, they can be computed from those of A_1 by applying a matrix-matrix multiplication (7). The computational cost for steps (6) and (7) becomes negligible provided s is small.

Algorithm 5.10 QDWH based OOC spectral D&C algorithm

```
Input: Symmetric dense matrix $A \in \mathbb{R}^{n \times n}$, $s$ number of sought-after eigenpairs
Output: Partial spectral decomposition of $A = V_s D_s V_s^T$
 1: Choose $\sigma$
 2: Compute polar factor $U_p$ of $A - \sigma I$ via QDWH (OOC GPU)
 3: Compute orthogonal matrix $V = [V_1 V_2]$ such that $\frac{1}{2}(U_p + I) = V_1 V_1^T$ (in-core GPU)
 4: Compute $A_1 = V_1^T A V_1$ (OOC GPU)
 5: if $k \geq s$ then
 6:     Compute eigenpairs of $A_1$ (in-core CPU)
 7:     Compute eigenvectors of $A$ from those of $A_1$ (back-transformation) (in-core GPU)
 8: else
 9:     Go to the step (1) and choose larger $\sigma$
10: end if
```

As illustrated in Algorithm 5.10, the only computationally-intensive part is the polar decomposition via the QDWH algorithm (2). This operates on matrix A of dimension $n \times n$. From Equation (4.31), the computationally-intensive part is the QR factorization of a $2n \times n$ matrix. In the following subsections we describe the implementation of the OOC QDWH algorithm on the GPU. Furthermore, the arithmetic and communication cost can be significantly reduced by exploiting the special structure of the $2n \times n$ matrix.

5.3.2 OOC QDWH algorithm

In the out-of-core spectral D&C eigensolver, the QDWH algorithm for computing polar decomposition remains the main computational bottleneck. The algorithm operates on problems for which matrix A or an auxiliary matrix of dimension $2n \times n$, required in the QDWH iteration, do not fit into the GPU memory. The pseudo-code is given in Algorithm 5.11.

As discussed in Section 4.4, the QDWH iteration is run until the difference between two steps is smaller than the user-defined tolerance "*tol*" (7) or if the maximum number of iterations, "*maxIter*", is reached. If we analyze the algorithm, the QR factorization of $\hat{X}_i$ (5) and update of X_{i+1} (6) are the only computationally-intensive parts of the QDWH algorithm. For n large enough, the QR factorization becomes an out-of-core problem for the GPU. In the matrix one can notice that the upper $n \times n$ part of $\hat{X}_i$ is symmetric and the lower $n \times n$ part is the identity matrix. While there exist QR implementations that can deliver very high performance on GPUs [8], they lack specialized kernels that can exploit such matrix structure and therefore perform additional flops. By applying

Algorithm 5.11 QDWH algorithm for polar decomposition

```
Input: Symmetric dense matrix $A \in \mathbb{R}^{n\times n}$
Output: Polar factor $U_p$
 1: $X_0 = A/\alpha$
 2: for  $i = 0 : maxIter$ do
 3:     Calculate factors $\gamma$, $a_i$, $b_i$ and $c_i$
 4:     Set matrix $\hat{X}_i = \begin{bmatrix} \sqrt{c_i} X_i \\ I_n \end{bmatrix}$
 5:     $\hat{X}_i \xrightarrow{QR} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$
 6:     $X_{i+1} = \frac{b_k}{c_k} X_i + \frac{1}{\sqrt{c_k}}(a_k - \frac{b_k}{c_k}) Q_1 Q_2^T$
 7:     if $\|X_{i+1} - X_i\|_F < tol$ then
 8:         break
 9:     end if
10: end for
11: $U_p = X_{i+1}$
```

the routines from traditional computational libraries, the QR factorization of a $2n \times n$ matrix costs $\frac{10}{3}n^3$ flops. If the eigenvectors are required then an additional $\frac{10}{3}n^3$ flops are required to explicitly construct factor Q.

The second arithmetically expensive part is the matrix-matrix multiplication (6) performed in the update of X_{i+1}. Taking into account that X_{i+1} is symmetric, we can exploit this structure to develop a new out-of-core symmetric matrix-matrix multiplication (based on LAPACK's routine GEMM):

$$C = \alpha A B^T + \beta C.$$

This routine is similar to the corresponding routine from BLAS but with C symmetric. By employing the square block strategy, it is possible to further reduce both the communication and algorithmic costs by referencing only the lower or upper triangular part of C and thus reduce volume of data copied to and from GPU memory.

The total cost of one QDWH iteration, with all improvements applied, is roughly $5n^3$ per iteration and includes $4n^3$ flops for the QR factorization and generating $Q = [Q_1\, Q_2]$, and $2/3n^3 + bn^2$ flops for the symmetric matrix-matrix multiplication, where b is the algorithmic block size. The memory requirements are n^2 elements for storing matrices A/X_i (both are symmetric and can be stored into one storage space by keeping only the upper/lower triangle of each and the diagonal of one of them in an additional array with the dimension n), the additional workspace of size $3n^2$ for storing $\hat{X}_i$ $(2n^2)$ and a few auxiliary arrays (n^2). In the following we offer a detailed description of a novel out-of-core QR factorization for structured matrices and the OOC symmetric matrix-matrix multiplication.

OOC-GPU QR factorization

The OOC-GPU algorithm for the QR factorization encodes a left-looking, slab-oriented approach [33] that transfers data by column blocks (slabs) of width k, and operates on blocks of width $b \ll k$. This approach is implemented as an blocked algorithm consisting of two-levels, in which the first block-level, operating on slabs of width k, optimizes the transfers to/from the GPU memory and improves the utilization of GPU. The second level, or cache-level blocking, optimizes the execution on the multi-core CPU by improving cache data locality.

Let us denote the $2n \times n$ matrix $\hat{X}_i$ in Algorithm 5.11 as D, and consider a partitioning of this matrix into blocks of dimension $s \times s$ each, where D_{ij} denotes the (i,j)-th block and, for simplicity, we assume that n is an integer multiple of k. Here, the parameter k is chosen so that a slab of size $(n+k) \times k$ fits into the GPU memory. Routine QR_OOC in Algorithm 5.12 and Figure 5.3 describe how to leverage the upper triangular structure of the bottom $n \times n$ half of D during the computation of the QR factorization of this matrix using our OOC-GPU algorithm.

Algorithm 5.12 QR_OOC(n, k, b, D)

```
1: $r = n/k$
2: for $i = 1 : r$ do
3:     Copy $D(1 : r+i, i)$ to GPU
4:     for $j = 1 : i-1$ do
5:         $D(j : r+j, i) = \text{Update\_OOC}(n, k, b, D(j : r+j, j), D(j : r+j, i))$
6:     end for
7:     $D(i : r+i, i) = \text{QR\_Hybrid}(n, k, b, D(i : r+i, i))$
8:     Copy $D(i : r+i, i)$ to main memory
9: end for
```

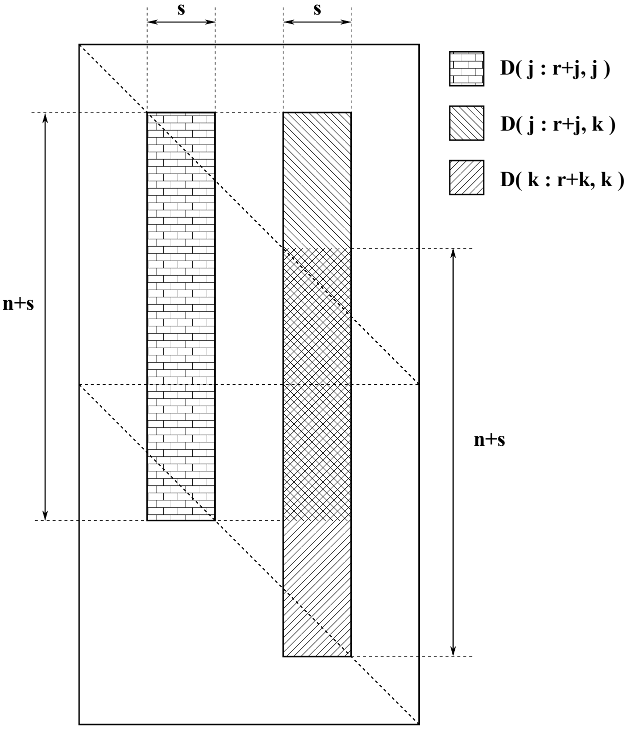


Figure 5.3: Illustration of the QR_OOC factorization on GPU.

At each iteration of loop (2), a slab $D(1 : r+i, i)$ of size $(n+k) \times k$ is updated. The update starts with a copy of the slab to the GPU memory and the application of all the previous transformations, loop (4), to the left of the slab (left-oriented QR factorization). The previous transforms are divided into slabs of width k and applied to the corresponding fraction of the slab $D(:, i)$ from the left, invoking routine UPDATE_OOC (5) for that purpose. As the whole set of previous transfers to not fit into the GPU memory they are divided into slabs that can fit the GPU memory and applied one at a time. When the update is done, the slab $D(i:r+i,i)$ is factorized (7) by invoking the QR_HYBRID routine. Note that the QR factorization can be performed completely in-core since the working slab fits into the GPU memory. Upon the completion, the slab $D(i:r+i,i)$ is copied back to main memory (8).

The algorithm requires an $n+k\times k$ memory space to store the working slab $D(j:r+j,i)$ and an $n+k\times b$ space for a slab of the previous transformations. Therefore, the update of the current slab is performed out-of-core while the QR factorization can be performed as a hybrid in-core GPU problem.

The bottom sparse structure of matrix D partially exploited by dividing it into large slabs of width k that fits the GPU memory. In Algorithm 5.13, Algorithm 5.14, and Figure 5.4 we present routines UPDATE_OOC and QR_HYBRID. These algorithms further exploit the zero-structure of the bottom half of D by introducing a second level of the blocking. In these routines, matrices E and F are partitioned into blocks of size $b\times b$, where E_{ij}, F_{ij} stand for the (i,j)-th blocks of the corresponding matrix. For simplicity, we assume that s is an integer multiple of b. The UPDATE_OOC routine operates on slab F (slab $D(j:r+j,i)$ from Algorithm 5.12) which is stored in GPU memory, streaming blocks of E; i.e. previous transforms $(D(j:r+j,j)$ from Algorithm 5.12) from main memory to the GPU and updating F from the left. The QR_HYBRID routine computes a QR factorization of E (slab $D(i:r+i,i)$, stored in the GPU), using a conventional blocked left-looking procedure with block size b, so that the block factorizations and orthogonal transforms are computed in the CPU (4), while the update of the trailing submatrix is performed in the GPU (6).

Algorithm 5.13 Update_OOC(n, k, b, E, F)

```
$r = n/b$; $t = k/b$
for $i = 1:t$ do
    Copy $E[i:r+i,i]$, containing $Q_k$, to main memory
    $F[i:r+i,:] = Q_i^T F[i:r+i,:]$ (in-core GPU)
end for
```

Algorithm 5.14 QR_Hybrid(n, k, b, E);

```
$r = n/b$; $t = k/b$
for $i = 1:t$ do
    Copy $E[i:r+i,i]$ to main memory
    $E[i:r+i,i] = R_i/Q_i = QR(E[i:r+i,i])$ (in-core CPU)
    Copy $E[i:r+i,i]$, containing $Q_i$, CPU → GPU
    $E[i:r+i,i+1:r] = Q_i^T E[i:r+i,i+1:r]$ (in-core GPU)
end for
```

Improvements in QR_OOC

While performing the QR factorization of D, only the orthogonal matrix Q is stored while the upper triangular factor R is not needed and thus is not stored. The QR_OOC algorithm is a left-looking variant that applies all previous transforms to the current slab,

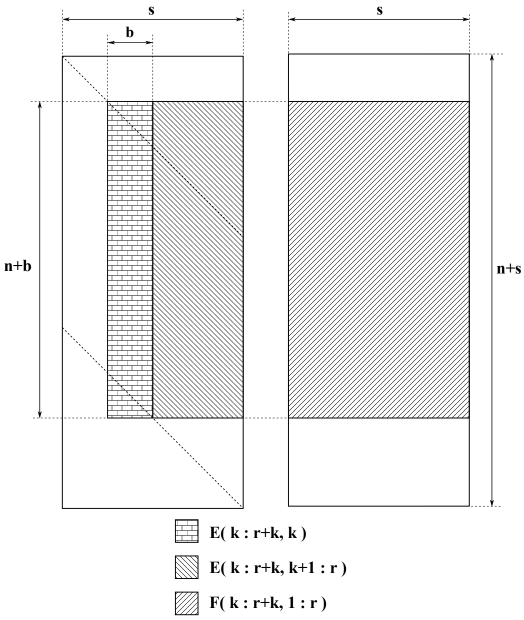


Figure 5.4: Illustration of the kernels UPDATE_OOC and QR_Hybrid

in contrast with the traditional right-looking approach that immediately propagates the transforms to the right of the current slab. Left-looking OOC variants in general obtains a smaller number of transfers [33]. When operating on the OOC data, the traditional right-looking approach requires copying all the slabs that are to the right of the current slab, to the GPU, update and move them back to the main memory, which requires two additional transfers. Despite the increased number of memory transfers, the traditional right-looking approach performs a large number of copying from GPU memory to the main memory which is slower than the copying in opposite direction (up to 2×). Therefore, copying from the GPU to the main memory should be avoided as much as possible. In our approach, the number of data transfers between main memory and GPU memory is reduced by a factor of 2 if the left-looking approach is applied.

The data locality can be preserved by exploiting the special structure of the slab $D(:,i)$ (Figure 5.3) and thus decrease even further the number of transfers. Concretely, at each step of the inner loop of routine QR_OOC, the slab $D(j:r+j,i)$ is stored in the GPU memory, and for the next iteration of the j-loop (Algorithm 5.12) the additional block $D(r+j+1,i)$ of size $k \times k$ is required while the top-most block $D(j,i)$ of the current slab $D(j:r+j,i)$ is not needed. Therefore, instead of returning the updated slab $D(j:r+j,i)$ to the main memory and copying back slab $D(j+1,r+j+1,i)$ to the GPU, the upper block $D(j,i)$ is removed from the GPU and the new block $D(r+j+1,i)$ is copied there. With this approach, at each update step, only one block of size $k \times k$ is sent to the GPU, instead of the whole slab of size $(n+k) \times k$, resulting in a significant reduction of data transfers. Furthermore, one transfer back to the main memory is spared as the block $D(j,i)$ is not needed in any subsequent computation.

Improvements in QR_Hybrid

This routine computes the QR factorization of a large slab $D(k:r+k,k)$, that can fit into the GPU memory, with the collaboration of both CPU and GPU. The slab is divided into smaller slabs ($E(:,k)$) of width b, Figure 5.4 (left). At each iteration of routine QR_Hybrid, Algorithm 5.14, the orthogonal factor of the slab $E(k:r+k,k)$ is computed

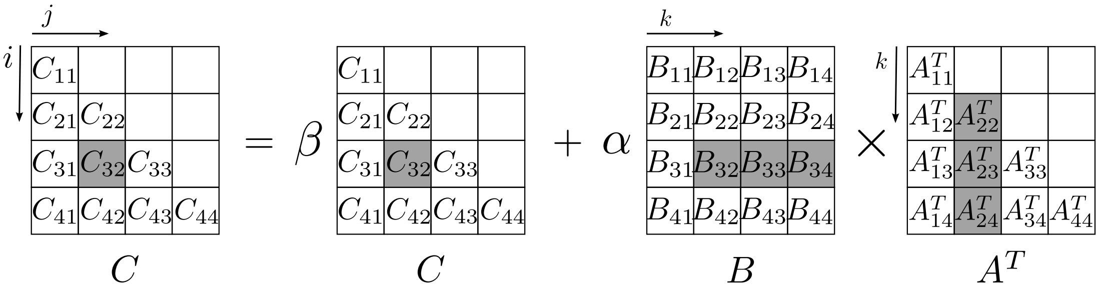

j
i
k
k
C_{11}
C_{21} C_{22}
C_{31} C_{32} C_{33}
C_{41} C_{42} C_{43} C_{44}
$= \beta$
$+ \alpha$
$\times$
B_{11} B_{12} B_{13} B_{14}
B_{21} B_{22} B_{23} B_{24}
B_{31} B_{32} B_{33} B_{34}
B_{41} B_{42} B_{43} B_{44}
A^T_{11}
A^T_{12} A^T_{22}
A^T_{13} A^T_{23} A^T_{33}
A^T_{14} A^T_{24} A^T_{34} A^T_{44}
C
C
B
A^T

Figure 5.5: Illustration of modified GEMM where C is symmetric and A upper triangular matrix.

on the CPU, transferred to the GPU, and finally submatrix $E[k:r+k,k]$ is updated from the right on the GPU. The update can be performed with only one call to routine GEMM since the current slab $D(k:r+k,k)$ is already in the GPU memory. Following this approach, the strictly lower triangle of E, with all zeros entries, can be efficiently exploited with little overhead. The overhead depends on the ratio between b and k (in practice, $b \leq 128 \ll k$), getting smaller as the difference between b and k grows.

OOC update of QDWH iteration

After the QR factorization of the large $2n \times n$ matrix, the second most expensive part of the QDWH algorithm is the update of X_{i+1}, Algorithm 5.11 (6). To update X_{i+1} the following matrix-matrix operation has to be performed:

$$C = \beta C + \alpha B A^T, \tag{5.13}$$

where the right-hand side C is symmetric, A is an upper triangular matrix, and the memory space required for C, A and B oversize the GPU memory. The update can be computed with a single call to the LAPACK/MAGMA routine GEMM or SYMM, and matrix-matrix addition. However, the state-of-the-art MAGMA implementation cannot deal with problems whose size exceeds the available GPU memory. To overcome this limitation, a new matrix-matrix multiplication algorithm based on GEMM routine is developed. The algorithm exploits the symmetry and upper triangular structure of the matrices in a communication-optimal manner, as illustrated in Figure 5.5

The matrices are divided into $b \times b$ blocks where $dim = n/b$ denotes the dimension of block-matrix. Blocks A_{ij}, B_{ij}, C_{ij} stand for the (i,j)-th block of the corresponding matrices. Each block $C_{i,j}$ is computed as the product of the i-th row of B and the j-th column of A^T. Since C is symmetric, only the lower (or upper) block-triangle is updated. Furthermore, because A is upper triangular, only the blocks $B_{i,j}, \ldots, B_{i,n/b}$ and $A^T_{j,j}, \ldots, A^T_{j,dim}$ take part in the update of $C_{i,j}$.

The pseudo–code of our OOC matrix-matrix multiplication is given in Algorithm 5.15. The notation dA, dB and dC is used to point to the storage spaces of the corresponding matrices in the GPU memory.

The lower triangle of matrix C is divided into $dim \times dim$ blocks of size $b \times b$ that are processed by rows (1)-(2). The update of each block starts by copying blocks C_{ij}, B_{ij} and A_{ij} to the GPU, steps (3)-(7), and updating block dC with a single call to the MAGMA routine GEMM (8) performed in the GPU. The block size b is chosen so that 3 blocks of size $b \times b$ fit into the GPU memory. In order to decrease the number of memory transfers, the order of the following updates (9) is reversed, so that, at the end of loop, block B_{ij+1},

Algorithm 5.15 OOC $C = \alpha B A^T + \beta C$

Input: $A \in \mathbb{R}^{n \times n}$ upper triangular, $B \in \mathbb{R}^{m \times n}$, b block size, factors α, β
Output: Symmetric matrix C

```
 1: for $i = 1 : dim$ do
 2:     for $j = 1 : i$ do
 3:         Copy $C_{ij} \rightarrow dC$
 4:         if $j == 1$ then
 5:             Copy $B_{ij} \rightarrow dB$
 6:         end if
 7:         Copy $A_{ij} \rightarrow dA$
 8:         $dC = \beta dC + \alpha dB dA^T$ (in-core GPU)
 9:         for $k = dim : j+1 : -1$ do
10:             Copy $B_{ik} \rightarrow dB$
11:             Copy $A_{jk}[\rightarrow dA$
12:             $dC = dC + \alpha dB dA^T$ (in-core GPU)
13:         end for
14:         Copy $dC \rightarrow C_{ij}$
15:     end for
16: end for
```

required in the next j-loop iteration, is already in the GPU memory. Thus, block B_{ij} is copied explicitly (5) to the GPU only once at the beginning of j-loop (i.e. when $j = 1$). At each pass of i-loop (i.e. for each computed block-row) $i - 1$ copies of block B_{ij} to the GPU are spared.

5.3.3 Subspace iteration

In this section we describe an out-of-core algorithm that computes the orthogonal matrix V, Algorithm 5.10 (3) and constructions subproblem A_1 (4) (if the smallest eigenvalues are required). As discussed earlier, the subspace iteration can be performed entirely in-core if the splitting point σ is carefully chosen.

As illustrated in Algorithm 4.8, Section 4.4, the most expensive parts are the QR factorization of matrix $n \times k$ and the matrix-matrix multiplication of dimension $n \times n$ and $n \times k$. Given that $k \ll n$ V can be computed entirely in-core in the GPU. However, if k gets larger or n is extremely large, some parts of the subspace iteration algorithm might become out-of-core problems for the GPU. In the following discussion we will refer to our algorithm as an out-of-core algorithm although some or all parts of the algorithm might be inherently in-core for GPU.

Algorithm 5.16 illustrates the out-of-core subspace iteration algorithm for the orthogonal matrix V. The routine computes the orthogonal matrix V from the polar factor C obtained from the QDWH algorithm. The algorithm operates on the polar factor C obtained from the OOC QDWH algorithm described in Subsection 5.3.2. The dimension k of the subproblem A_1, is obtained as the sum of the square of the Frobenious norm of A and a small constant $\hat{k}$ that is used as a safeguard (1). If k is less than the number of sought-after eigenvalues s (2), i.e. all sought-after eigenvalues are not in the spectrum of subspace A_1, the algorithm is stopped and the polar decomposition (OOC QDWH algorithm) is run again with a larger σ.

Algorithm 5.16 Out-of-core subspace iteration algorithm

```
Input: Symmetric matrices $A \in \mathbb{R}^{n\times n}$ and $C = \frac{1}{2}(U_p + I) \in \mathbb{R}^{n\times n}$, $s$
Output: Orthogonal matrix $X := V = [V_1 \, V_2]$
 1: Compute $k = \|C\|_F^2 + \hat{k}$
 2: if ($k < s$) then
 3:     exit algorithm
 4: end if
 5: Choose initial matrix $X \in \mathbb{R}^{n\times k}$
 6: while (true) do
 7:     Compute QR factorization of $X$ (in-core GPU)
 8:     Construct $V = [V_1 \, V_2]$ (OOC GPU)
 9:     Form error matrix $E = V_2^T A V_1$ (OOC GPU)
10:     if ($\|E\|_F / \|A\|_F < tol$) then
11:         stop
12:     end if
13:     $X := CV_1$, go to (7) (OOC GPU)
14: end while
```

The iterations starts with the construction of an initial matrix X (5). A good choice for X is one whose columns lie near to the column space of C. Thus, in the algorithm the columns of X are set to be the first k columns of C. In the original algorithm [107], X is set to the first k columns with the largest norm. However, the simplest approach is to generate a random X and form $X := CX$ (X has to lie in the column space of C) but introduces matrix-matrix operations the exhibit additional $2n^2k$ flops. The QR factorization (7) is performed by calling the GEQRF routine from the LAPACK, CUBLAS or MAGMA libraries. In case the matrix X cannot fit into the GPU memory, the out-of-core MAGMA implementation (MAGMA_DGEQRF_OOC kernel) can be applied. In recent GPU cards the QR factorization will always be performed in-core as far as a good splitting point is chosen and the number of eigenvalues is small. However, in older GPU cards or those with fewer memory, the QR factorization can easily become an out-of-core problem and then the out-of-core MAGMA implementation has to be used.

The orthogonal matrix V is explicitly constructed by applying routine ORGQR(8) to construct it from the elementary reflectors stored in lower triangle of X obtained in (7). The dimension of V is $n \times n$ and, therefore, this is an out-of-core problem. However, recent linear algebra libraries do not provide any routine capable of explicitly constructing the orthogonal V from the Householder reflectors if the data are out-of-core. A straightforward out-of-core implementation of ORGQR routine is done. This algorithm is similar to the QR_OOC factorization, Algorithm 5.12, describing the two-level blocked algorithm implementation, but with the difference that the algorithm is started from the rightmost panel and moves to the left and that V is square matrix.

Once the QR factorization and construction of orthogonal matrix V are finished the backward error is checked (10). Note that $V = [V_1 \, V_2]$ and $V^T A V = \begin{bmatrix} A_1 & E^T \\ E & A_2 \end{bmatrix}$, where E should be a zero-matrix. Thus, we can check the accuracy of the orthogonal matrix V by checking if $\|E\|_F / \|A\|_F = \epsilon$. Instead of using the machine roundoff error, we use a user-defined tolerance, step (10). Typically, the subspace iteration algorithm converges in just one iteration but in finite-precision arithmetic, it usually requires more iterations.

The construction of the backward error E (9) can be done in two steps, $X = AV_1$ and $E = V_2^T X$, by calling routines SYMM for the first and GEMM for the second. Both functions potentially operate on matrices that do not fit into the GPU memory. For the first operation, the required memory space is $n^2 + nk + nk$ for A, V_1 and X; while the second equation requires $n(n-k) + nk + (n-k)k$ for V_2, X and E. Even if k is small, each operation requires more than n^2 storage entries (if k is small then $n-k$ is close to n) and they are both out-of-core problems for large n.

If the stopping criteria is not matched, X is constructed in (13) by multiplying C from the right with V_1. The construction can be done via a single call to the MAGMA in-core routine SYMM or out-of-core implementation if A is too large to fit into GPU memory. In case the algorithm converges to invariant subspace, the subspace A_1 is computed such that $A_1 := V_1^T A V_1$. For the construction of A_1 only one call to SYMM is required since the product $X_t := AV_1$ is already performed and stored when the error matrix E was constructed (9), therefore, only $A_1 := V_1^T X_t$ has to be computed. By performing only one matrix product instead of two the arithmetic cost is decreased by n^3 flops. That significantly reduces the execution time when A is out-of-core, since only one out-of-core matrix computation is performed instead of two.

The original CPU-based algorithm presented in Algorithm 4.8 is almost unchanged. The main difference is that standard multi-core LAPACK routines, such as the QR factorization, are replaced with their GPU-based out-of-core or in-core routines form MAGMA or CUBLAS computational libraries. However, other routines, like matrix computations (e.g. GEMM and SYMM) and the routine that generates orthogonal matrix from the Householder reflectors (e.g. ORGQR), which standard GPU computational libraries do not implement, were implemented from the scratch in order to efficiently solve the out-of-core problem from the GPU viewpoint.

If a very small number of eigenvalues is required and the given eigenproblem exceeds the GPU memory, the computational cost of our GPU-based algorithm for the invariant subspace A_1 becomes negligible compared to the QDWH part of the spectral D&C algorithm. Despite some parts of the subspace iteration are still out-of-core, the number of flops are much lower compared to the polar decomposition (computed via QDWH algorithm). The only computational intensive part of the subspace iteration is in the while loop. At each pass of the loop, the QR factorization exploits $2nk^2$ flops, the explicit construction of V costs $4n^2k$ flops, computing E costs $4n^2k - 2nk^2$ flops for both matrix computations, and $2n^2k$ flops are required to update X. The total cost for one iteration is therefore $10n^2k$ flops and since usually no more than two iterations are required to converge, the average flops for the subspace iteration is $\sim 20n^2k$ (the final computing of the subspaceA_1 is negligible and requires only $2nk^2$ flops).

Chapter 6

Numerical Experiments

In this chapter we will analyze the performance for our three GPU out-of-core eigensolvers (OOC-GPU) that are based on the multi-stage reduction to tridiagonal form (Section 5.1), the Krylov-subspace eigensolvers (Section 5.2), and the spectral divide–and–conquer based on QDWH algorithm for computing polar decomposition (Section 5.3).

The performance and scalability of OOC-GPU algorithms will be demonstrated using problems arising in the simulation of molecular motions. The problems arising in this application are very large and computationally demanding with the solution of the generalized dense symmetric eigenproblem as their central computational bottleneck. We will show that, by employing our OOC-GPU eigensolvers, the execution time of traditional eigensolvers used for the solution of larger dense symmetric eigenproblem can be significantly reduced. Furthermore, by exploiting the power of GPU devices, even large problems can be solved while preserving very high performance even if the problem does not fit into the GPU memory.

This chapter is organized as follows. Section 6.1 gives a short introduction into macromolecular motion and the basic tools and methods, such as Normal Mode Analysis, that captures the collective motions at the molecular level. The molecular dynamics problems that will serve as test cases for OOC-GPU eigensolvers and the decription of the testing computing system, are presented in Section 6.2. Sections 6.3, 6.4 and 6.5 analyze the performance of each of the three OOC-GPU eigensolvers, respectively, eigensolver based on multi-stage reduction, Krylov-subspace based eigensolver and SD&C eigensolver. Finally, Section 6.6 offers the side-by-side comparison of three OOC-GPU eigensolvers applied to macromolecular test cases, and gives some concluding remarks of the overall performance of the algorithms.

6.1 Macromolecular motions

Macromolecular motions are very important to understand the biological processes and functions. A macromolecule is an assemble of proteins and nucleic acids that are formed in long unbranched chains of amino acids and nucleotides. These macromolecules support the main biological functions, for example, the ribosomal machinery produces new proteins according to the genetic code; the chaperonin proteins assist the folding process of the newly constructed proteins; and tubulin and actin filaments support the cellular shape. The biological activities of these components can be studied by analyzing their dynamics and interactions at molecular level.

A traditional way to predict conformational changes is by direct experimentation of the functional motions but this approach showed to be quite complex. Instead of direct experimentation, Molecular Dynamic (MD) simulations are more often used. MD is a computer simulation of physical movement of atoms and molecules at a given period of time, during which the atoms and molecules interact between themselves. MD provides information of fluctuations and conformational changes by using 3D atomic structures. However, the large size of macromolecules, i.e. the large number of atoms they are assembled of, and the long time scales of their motions makes MD simulations too costly or even impossible.

In order to decrease the cost of MD simulations, coarse-grained (CG) models are used. In CG models, instead of explicitly representing every atom of the systems, atoms are grouped into simplified entities or pseudo-atoms that results in a reduced number of variables and significant computational savings. In recent years, CG model merged with normal mode analysis (NMA) [116] has become a powerful and popular approach to simulate collective motions of macromolecules at extended time scales [117]. CG-NMA simulation has been successfully applied to calculate different bio-molecular simulations [118, 119, 120] and it has been proved, in practice, to be a powerful alternative to perform costly simulations using all atoms.

For large molecular systems, the main bottleneck of NMA approach is the diagonalization step. To reduce the diagonalization, an internal coordinate (IC) method is used instead of Cartesian coordinates [121]. IC method requires at least one-third less degrees of freedom (DoF) and thus reduces both computational time and memory usage. Although IC methods reduce the number of DoF, the diagonalization step remains a major computational problem for large macromolecular simulations.

6.1.1 IC-NMA method

IC-NMA method describes collective motions by approximating the potential (Hessian) and kinetic energies as quadratic functions of the atomic positions and velocities. This approach allows the decomposition of motions into a series of vectors which encode the deformation or potential displacement of atoms. These vectors, or modes, are obtained by diagonalizing the second derivate matrices of both potential and kinetic energies. The resulting eigenvectors present the modes and the eigenvalues the corresponding frequencies. The modes with high frequencies represent localized dislocation, whereas low-frequency modes correspond to collective conformational changes.

In the IC-NMA model, the molecular system is a set of pseudo-atoms interconnected by harmonic springs. The motion of the system can be described as a combination of n normal modes obtained by solving the generalized eigenproblem:

$$HX = TX\Lambda, \tag{6.1}$$

where the eigenvectors $X = (x_1, x_2, \ldots, x_n)$ represent normal modes and are associated with the eigenvalues $\Lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$. Each eigenvalue is related to the frequency ν_k by $\lambda_k = (2\pi\nu_k)^2$. The matrices H and T represent the potential and kinetic energies, respectively. The potential energy of the systems is formulated as:

$$V = \sum_{i<j} F_{ij}(r_{ij}^t - r_{ij}^0)^2, \tag{6.2}$$

where F_{ij} is the spring stiffness matrix, r_{ij} is the distance between pseudo-atoms i and j, and super-indices t and 0 are current and equilibrium conformations, respectively. The potential energy expressed in n internal coordinates (ICs) is given by

$$V = \frac{1}{2} q \mathbf{H} q^T, \tag{6.3}$$

where vector q represents the coordinate displacements from the equilibrium conformation at a given energy minimum q^0, $q = q_\alpha - q_\alpha^0$, $\alpha = 1, \ldots, n$ and H. The Hessian matrix H is defined as:

$$\mathbf{H} = \frac{\partial^2 V}{\partial q_\alpha \partial q_\beta}, \tag{6.4}$$

where α and β represent any internal coordinate indices. In a similar way, the kinetic energy is expressed as:

$$T = \frac{1}{2} \dot{q} \mathbf{T} \dot{q}^T, \tag{6.5}$$

where the time differentiation is indicated with the dot operator and the matrix $\mathbf{T}$ representing the kinetic energy of the system is computed as:

$$\mathbf{T} = \sum_i m_i \frac{\partial r_i}{\partial q_\alpha} \frac{\partial r_i}{\partial q_\beta}. \tag{6.6}$$

The parameter m_i indicates the mass of the i-th atom and r_i is the corresponding Cartesian coordinate. The diagonalization of Lagrange's Equations (6.1) yields solutions of the form:

$$q_k^t = q_k^0 + \sum_{k=1}^{n} a_k x_k \cos(2\pi \nu_k t + \delta_k), \tag{6.7}$$

where a_k and δ_k depend on the initial conditions and ν_k is angular frequency associated with the normal mode x_k. The direct computation of matrices H and T using Equations (6.3) and (6.5) requires $\mathcal{O}(n^4)$ and $\mathcal{O}(n^3)$ floating-point operations, respectively. If the multipurpose tool chest iMod [121] is used, the computational cost can be reduced to only $\mathcal{O}(n^2)$ flops. In consequence, the solution of the generalized eigenproblem (6.1) becomes the main computational bottleneck. Furthermore, not all eigenvalues are required since the functional motions are only encoded in the lowest frequency part, i.e. only a set of the smallest eigenvalues (frequencies) and the corresponding eigenvectors (normal modes) are to be computed.

6.2 Benchmarks

The key numerical problem in the analysis of the macromolecular functional motions via IC-NMA method is the solution of a generalized eigenproblem of the form

$$AX = BX\Lambda,$$

where $A, B \in \mathbb{R}^{n \times n}$ correspond to Hessian and kinetic matrices, $\Lambda \in \mathbb{R}^{s \times s}$ is a diagonal matrix with s sought-after eigenvalues corresponding to frequencies, and $X \in \mathbb{R}^{n \times s}$ contains unknown eigenvectors (i.e. normal modes). Furthermore, in IC-NMA, matrices A and B and dense and symmetric positive definite (SPD). Since functional motions are presented with the lowest frequencies, only the smallest eigenpairs are required ($\approx 1\%$) for large macromolecules ($n > 10,000$).

In order to validate the novel GPU-based algorithms presented in this thesis we chose a benchmark set of large macromolecules including filaments, compartamental or chaperonin macromolecules, and viral capsids; see Table 6.1. The maximum number of variables (DoF - Degrees of Freedom) are chosen such that the examples can fit into the main memory of the testing computing system. Although the coarse-grained (CG) approach is used to reduce the number of variables (or pseudo-atoms), some examples still exceed the available memory and thus the number of variables is randomly decreased in the problem generation stage by applying iMOD. However, in the experiments we will use the same examples but with less DoF giving problems of smaller size whose computation is cheaper but provides a final conformation that is less accurate. Therefore, the final goal is to obtain a more accurate solution by providing more DoF that control the directions to inspect conformational changes.

Name	Acronym	PDB id	DOF
Sus scrofa	1Tub	utub10	29,622
Sus scrofa	1Tub	utub20	31,178
Sus scrofa	1Tub	utub40	34,297
Nudaurelia capensis omega virus	NwV	1ohf	33,352
Hong Kong 97 virus Head II	HK97	2ft1	31,858
Hepatitis B virus	HBV	1qgt	30,785
Cowpea chlorotic mottle virus	CCMV	1cwp	30,504

Table 6.1: Benchmark of large-scale macromolecules.

The main computational bottleneck in IC normal mode analysis is the generalized eigenvalues problem. The computation starts with the reduction to a standard eigenvalue problem by applying the steps in Equation (4.2) - (4.6) and finishes with a back-transformation of the eigenvectors of the generalized eigenproblem, Equation (4.7) in Section 4.1. The first step of the reduction, the Cholesky factorization, is not the focus of this research and will not be covered in the experiments. As the construction of C is postponed in one version of the GPU-based Krylov subspace algorithm and performed implicitly within Lanczos steps, for that particular example, the construction of C will be tested and included in the final measurements.

Once the problem is reduced to standard form, any of the GPU-based eigensolvers presented in Chapter 5 are applied to compute a subset of the smallest eigenpairs. Table 6.2 presents the novel GPU-based algorithms and the acronyms that will be used in further experimentations. Furthermore, the table also offers a total number of floating point operations required by each algorithm. The Cholesky factorization is performed in $n^3/3$ flops and is not included in the total flop count.

The multi-stage reduction algorithm is a direct algorithm and, therefore, its total flops count is constant and independent on the number of eigenpairs computed. The total flops for the Krylov subspace-based algorithms KE and KI are based on the number of iterations performed. The KE algorithm pays an initial cost of $2n^3$ flops for the explicit construction of C while KI doubles the cost performed at each Lanczos iteration. The flops for SPECDIV depends on the number of iterations k_2 performed to compute a polar factor, but in practice, the number of iterations is no more than 8, thus yielding $48n^3$

Algorithm	Acronym	Type	Flops
Multi-stage reduction	MULTIS	Direct	$14n^3/3$
Krylov-subspace implicit	KI	Iterative	$k_1 \times 4n^2$
Krylov-subspace explicit	KE	Iterative	$2n^3 + k_1 \times 2n^2$
Spectral D&C based on QDWH	SPECDIV	Divide-and-conquer	$k_2 \times 6n^3$

Table 6.2: The list of our GPU-based eigensolvers with average flops counts.

flops at worst. In all the experiments, both the eigenvalues and eigenvectors are computed unless otherwise stated.

The target computational platform is a NUMA workstation with two Intel Xeon E5520 quad-core CPUs running at 2.27 GHz, with 48 GB of main memory, and a theoretical peak performance of 74.6 GFLOPS in double-precision. The testing system is equipped with an NVIDIA Tesla C2050 graphics processors consisting of 448 streaming processors running at 1.15 GHz, with 2.8 GB on-board memory (ECC on), and a theoretical peak performance of 515 GFLOPS in double-precision. The aggregated peak theoretical performance of the workstation is 590 GFLOPS. The Intel chipset and the GPU board are connected via a PCI-Express Gen2 interface with a peak bandwidth of 4.6 GB/second.

For the computations performed on the Intel cores, the GNU C compiler (gcc 4.1.2), GotoBLAS2 and LAPACK (version 3.4.0) were employed. For the computations performed on the Tesla GPU accelerator, the NVIDIA CUBLAS (version 5.5) and MAGMA (version 1.2.1) libraries built on top of CUDA (version 5.5) were used. All tests were performed in double precision arithmetic.

All experiments corresponding to the macromolecules listed in the Table 6.1 are out-of-core for all the most recent GPUs and therefore our testing environment equipped with Tesla C2050 GPU. The required memory space for a problem with ∼40,000 DoF is 12GB just for keeping a single matrix (e.g. C) in double-precision arithmetic.

6.3 Multi-stage reduction

In the multi-stage reduction to tridiagonal form only the reduction from dense symmetric matrix to tridiagonal form is considered. Table 6.3 lists the required routines/steps for the multi-stage (TM) approach. For illustration, here we will present only the two-stage and three-stage approaches; however, a general multi-stage approach can be leveraged as well, with multiple calls to the TM2 step (reduction from band to narrower band form) that gradually decreases the number of non-zero super/sub-diagonals of the band matrix W_i, where i presents the number of non-zero super/sub-diagonals. The last two steps, TM4 and TM5 are LAPACK routines applied to compute the eigenpairs of the tridiagonal matrix T and the back-transform that obtains the eigenvectors of A from those of T, respectively. Although these two routines are not part of this research, for completeness, their execution time and performance will be included in the final testings.

In Chapter 5 we have presented the multi-stage reduction approach. This algorithm is designed so that the first phase, i.e. reduction form dense symmetric matrix to band matrix, is performed completely out-of-core from the GPU viewpoint. The number of non-zero sub-diagonals, or bandwidth w, is chosen so that the band matrix, of dimension $n \times w$, can entirely fit into the GPU memory and, thus, the reduction to narrower band

Var.		Operation	Routine	Library
TD	TD1	$Q^TCQ = T$	DSYTRD	LAPACK
	TD2	$TZ = Z\Lambda \rightarrow T, Z$	DSTEMR	LAPACK
	TD3	$Y := QZ$	DORMTR	LAPACK
TM	TM1	$Q_1^TCQ_1 = W_1$	OOC__GPU__DSYRDB	–
	TM2	$Q_2^TW_1Q_2 = W_2$	OOC__GPU__DSBRDB	–
	TM3	$Q_3^TW_2Q_3 = T$	GPU__DSBRDT	–
	TM4	$TZ = Z\Lambda \rightarrow \Lambda, Z$	DSTEMR	LAPACK
	TM5	$Y := Q_1Q_2Q_3Z$	OOC__GPU__DGEMM	–

Table 6.3: Routines required in reduction of dense symmetric matrix to tridiagonal form on GPU. TD: one-stage reduction, TM: multi-stage reduction.

form (TM2) can be performed in-core on GPU. By exploiting this strategy, it is possible to avoid the overhead costs of memory transfers to and from GPU memory. Furthermore, the main bottleneck of TM2 is the bulge chasing phase that introduces additional flops, especially when the eigenvectors are required in which case the orthogonal transforms have to be explicitly accumulated.

Let us start with the examination of step TM1, the OOC reduction of a dense symmetric matrix to band form. In Section 5.1.1, we have distinguished two main parts, the QR factorization that is applied to the panel/slabs currently computed and the two-sided update of the rest of the matrix. Furthermore, for each of these two parts, we have developed two variants that differ in the GPU memory requirements and the size of the working arrays stored on the GPU. Table 6.4 lists the QR and two-sided update variants with their total memory requirements and the number of data transferred in the worst case. This case occurs at the beginning of the reduction when the first slab, that has to be reduced to band form, is $n - w \times w$. After that the algorithm proceeds with the two-sided update of the $(n-w) \times (n-w)$ submatrix A_2. In each following step, the height of the slab and the dimension of the submatrix A_2 are decreased by w. At a certain point, the leading dimension of the working slab is small enough to that the QR and the OOC two-sided update can be performed entirely in-core from the GPU perspective.

Function	Variant	Mem. req.	Data transfered
QR. fact.	QR-1	$w^2 + 2nw$	$4(n-w)w$
	QR-2	$w^2 + nw + bw$	$5(n-w)w$
Two-sided update	UPDATE-1	$b^2 + 2(n-w)w$	$3/2n^2 + n(3w + 3/2b)$
	UPDATE-2	$b^2 + (b+n-w)w$	$(3/2 + 3w/b)n^2 + n(5w + 3/2b)$
Accumulation of Q_1	ACC	$3b^2$	$4n^3/b + 2nb$

Table 6.4: The variants of QR factorization and two-sided update with their GPU memory requirements and number of data transfered, where n is matrix dimension, w band size and b algorithmic block size.

In the experiments, we consider two variants for the reduction from full symmetric matrix to band form, SYRDB-1, that employs variants QR-1 and Update-1, and variant SYRDB-2, that employs QR-2 and UPDATE-2. Although both variants exhibit the same computational cost, SYRDB-2 requires less storage space on the GPU than SYRDB-1, at the cost of an increased number of memory transfers to/from the GPU; see Table 6.4. Moreover, in the hybrid QR factorization the in-core GPU problem is divided into smaller slabs that are factorized on the CPU. The width of these slabs is chosen in order to improve the CPU execution. Our tests experimentally show that the optimal block size, on our testing system, is 128.

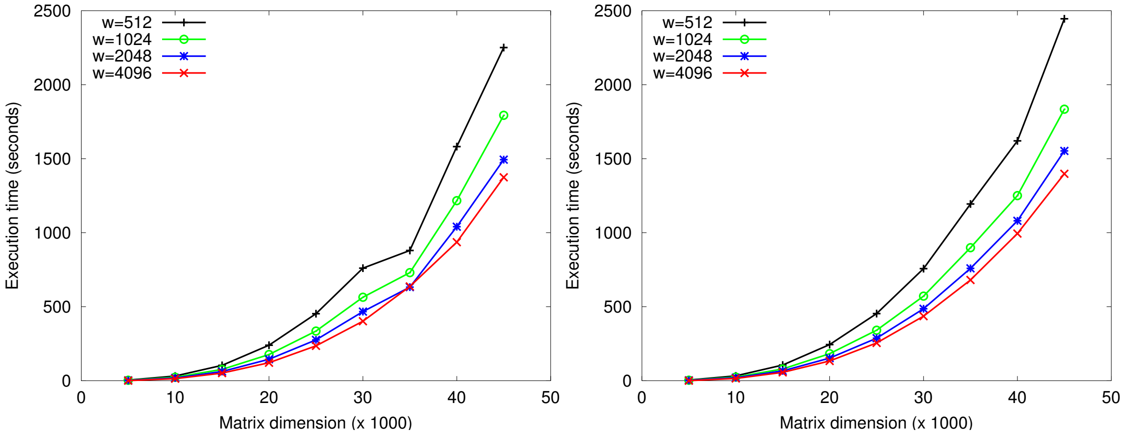
The accumulation of orthogonal transforms ACC, that is common to both SYRDB variants, can be cast completely in terms of the level-3 BLAS kernels GEMM and SYMM. Our out-of-core ACC is implemented as two calls of out-of-core GEMM routine. The OOC GEMM routine operates on square blocks and requires three workspaces of size $b \times b$ on GPU for keeping the multiplication factors A_{ij}, B_{ij} and C_{ij}. The block size does not depend on the problem dimension n and could be scaled to any problem size (eventually, the bottleneck would be the amount of main memory, not GPU memory, for very large matrices). In comparison to a straight-forward out-of-core GEMM implementation that performs $3n^3/b$ memory transfers, our implementation exploits better data locality exhibiting only $2n^3/b + nb$ data transfers. The data locality is preserved by implementing the cyclic update of blocks C_{ij} that re-uses the last block B_{ij} from the update of the previous block C_{i-1j}.

Figure 6.1 illustrates the performance of the two SYRDB variants including the accumulation of the orthogonal transforms in Q_1. As expected, SYRDB-1 achieves slightly better performance than SYRDB-2, Figure 6.1(a) and 6.1(b). This is because SYRDB-2 exhibits $nw - w^2$ more memory transfers from/to the GPU, at the same time, with the larger operational block size b. For example, in our testing computing system, SYRDB-2 achieves more than 182 GFLOPS (495 sec) for $n = 35000$ and $w = 8192$, while SYRDB-1 achieves 160 GFLOPS but with the maximum band size of only $w = 4400$. In other words, SYRDB-2 variant can be used to reduce larger matrices with the same band size or the same problem but with larger w.

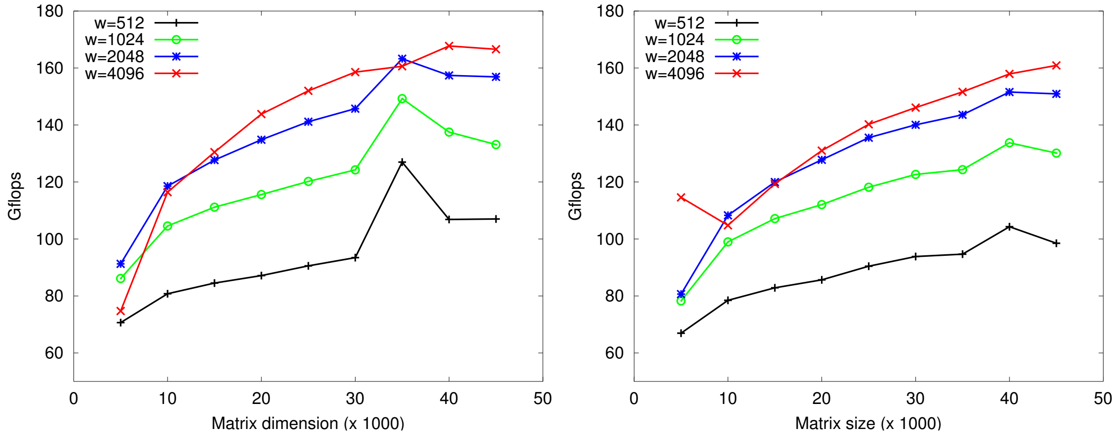
The performance of the reduction also depends on the bandwidth w, Figure 6.1(c) and 6.1(d). When the bandwidth grows, the performance increases as well. That is because the number of sub-diagonals that have to be eliminated decreases as the band is getting larger, resulting in less computations. In addition, the height, i.e the number of rows of the working slab for the QR factorization and the dimension of the sub-matrix, decreases as well. That introduce less memory transfers as more that more elements of the working matrices are kept in the GPU memory.

The majority of the computational cost of both SYRDB variants is spent in the accumulation of the orthogonal factors. The computational cost of the accumulation is $2n^3 - 2n^2w$ flops and doubles the total cost of SYRDB. Table 6.5 lists the performance and the percentage of total execution time spent in the accumulation. The accumulation weight in the total execution time slightly decreases with the band size due to the smaller number of sub-diagonals that have to be annihilated. In that case, the entries of Q_1, that do not participate in the annihilation, are simply filled with zeros, thus decreasing the portion of Q_1 that has to be updated. Moreover, with an increase of the bandwidth w the total execution time decreases while GFLOPS rate increases. Generally, the performance of the reduction form dense symmetric matrix to band matrix W_1 significantly improves as the bandwidth grows.

The reduction from band to tridiagonal form, routine TM3, is an algorithm that does not apply memory-aware blocked kernels. However, the accumulation of the orthogonal transforms can be delayed and accumulated by blocks. The algorithmic block size nb defines the size of the delay, i.e. the number of transforms accumulated in one turn. The execution time drops with the enlargement of both band size and algorithmic block size nb, Figure 6.2.



(a) SYRDB-1 variant time. (b) SYRDB-2 variant time.



(c) SYRDB-1 variant GFLOPS. (d) SYRDB-2 variant GFLOPS.

Figure 6.1: Two SYRDB variants for the reduction form dense to band form. (a) and (b) present total execution time and (c) and (d) GFLOPS.

The best execution time was achieved for the band size $w = 256$ and a block dimension $nb = 256$ (green). However, more than half of the total execution time is spent in the accumulation of orthogonal transforms. For example, in case $w = 256$ and $nb = 256$, 508.16 seconds out of a total 865.39 are spent in the accumulation (58%). The contribution of the accumulation to the total execution time decreases with the band size; e.g. for $w = 512$, $nb = 512$ only 37% of total time is spent in the accumulation. When the band size is larger, the accumulation operates with larger orthogonal factors thus achieving better utilization of the GPU but the execution time for the annihilation step increases because of a larger number of computations performed in terms of inefficient non-blocked level-2 BLAS routines. On the other hand, with smaller band, the annihilation phase is faster due to a better utilization of small CPU cache memory but the accumulation is slower because smaller updates are performed resulting in more memory transfers to/from GPU. To achieve the best performance for the TM3 phase, the compromise between CPU and GPU performance has to be investigated by carefully choosing the best match of band size and size of delay.

The best performance for TM1 is achieved when the bandwidth is large, while TM3 benefits from a small bandwidth. To bridge that gap, the TM2 routine that reduces a band matrix W_1 to a narrower band matrix W_2, is applied successively to reduce a large

w	SYRDB-1			SYRDB-2		
	time	GFLOPS	%	time	GFLOPS	%
512	674.88	125.20	0.56	653.92	129.21	0.54
1024	475.02	175.24	0.53	465.18	178.95	0.51
2048	374.79	215.40	0.51	370.73	217.77	0.48
4096	309.45	244.67	0.48	308.27	245.62	0.45

Table 6.5: The cost of accumulating the transformations in Q_1 for n=35,000 and the percentage of the total execution time of SYRDB.

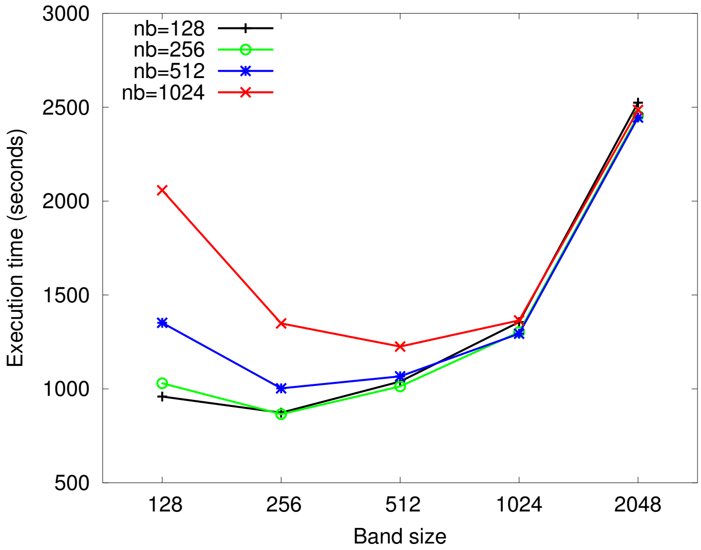


Figure 6.2: Performance of the in-core GPU TM3 for `1cwp` problem ($n = 30504$) with different values for algorithmic block size nb.

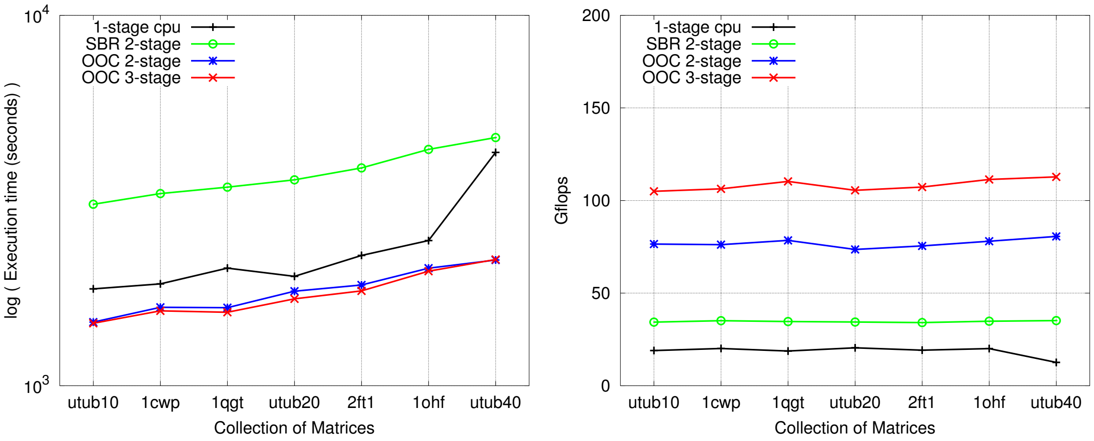
band matrix obtained in TM1 to the small band matrix required by the TM3 routines. The majority of computations in TM2 is performed within the bulge-chasing phase that becomes very expensive, especially if the eigenvectors are required. In such case the accumulation of orthogonal transforms in matrix Q_2 costs an additional $2n^3 - 2n^2w_2$ flops, with w_2 the number of non-zero sub-diagonals in the narrower band matrix W_2. The accumulation is done by directly updating the transformation matrix Q_1, obtained in TM1, from the right, i.e. $Q = Q_1Q_2$.

To increase performance and reduce the number of memory transfers performed in the bulge-chasing phase, TM2 is performed in-core. However, the accumulation of orthogonal transforms is still performed out-of-core from the GPU viewpoint since Q_1 is dense and cannot entirely fit into the GPU memory. The band size in TM1 is chosen such that a matrix W_1, in band storage format, can fit into the GPU memory. For our testing collection of MD matrices with maximum matrix size ∼30,000 and the Tesla C2050 GPU equipped with 2.6 GB of memory, the maximum band size w_1 is ∼3000. As expected, if the starting band w_1 is larger, the execution time increases because of a larger number of sub-diagonals that have to be annihilated. If the GPU is employed for both reduction and accumulation, Table 6.6, the execution time is greatly improved compared with the multi-core SBR implementation. The same occurs when the narrower band w_2 is increased as well. The best performance for TM2 is achieved when the bandwidth w_1 is large and w_2 small. The larger band w_1 introduces a negligible increase in the total execution time, if the computation is performed on the GPU.

w_1	$w_2 = 256$		$w_2 = 512$	
	CPU	GPU	CPU	GPU
1024	965.29	242.4	933.86	248.78
2048	1014.16	257.4	1012.34	303.9
3072	1074.75	273.38	1083.4	343.46

Table 6.6: Execution time of the in-core GPU implementation of TM2 for `1cwp` problem ($n = 30504$) compared to SBR routine SBRDT employing 8 cores.

Let us analyze the performance of the overall multi-stage out-of-core eigensolvers. We have tested two specific versions, the 2-stage and 3-stage out-of-core variants. The 2-stage OOC algorithm consists of the TM1, TM3, TM4 and TM5 steps, listed in Table 6.3; while the 3-stage OOC includes an additional step – TM2. The performance of the two OOC multi-stage eigensolvers for the MD collection of matrices in Table 6.1 is presented in Figure 6.3. In all testings, the CPU-only algorithms, 1-stage LAPACK algorithm and 2-stage SBR algorithm, are executed on all 8 available cores of the target computing system by employing a multi-threaded BLAS library. The total arithmetic costs for the OOC 2-stage and 3-stage eigensolvers are $8/3n^3$ and $14/3n^3$ flops, respectively, including the accumulation of orthogonal transforms.



(a) (b)

Figure 6.3: Performance of the 2-stage and 3-stage GPU OOC eigensolvers for MD problems Table 6.1. 100 smallest eigenpairs are computed.

The multicore 1-stage routine from the LAPACK library is used as the state-of-the-art routine for the solution of large eigenproblems. Although the highly optimized MAGMA or CUBLAS implementations of TD1 can be used as well, this implementation cannot solve problems whose size exceeds the total available GPU memory. In our testing computing systems, the largest in-core problem that can be solved on GPU have n=18,000.

In the 3-stage OOC algorithm, TM2 acts like an intermediate step that enables larger band to be used in the first step, reduction to band form, while providing a small band, required for the reduction to tridiagonal form. Therefore, a 3-stage OOC attains much higher performance than the 2-stage OOC variant, Figure 6.3(b). In the 2-stage OOC variant, the best performance was attained for band size 512, while for the 3-stage OOC in all MD collections, the band w_1 was set to maximum feasible size, $w_1 = 3072$, and $w_2 = 256$. Thus, by enabling larger band size in the TM1, the 3-stage OOC exhibits higher performance than the 2-stage OOC. As discussed above, the maximum feasible size for w_1 in the 3-stage OOC variant depends only on the amount of available GPU memory. Therefore, on GPUs equipped with more memory, w_1 can be made larger and thus attain even higher performance.

Furthermore, by carefully orchestrating the computation and the number of data transfers between the main memory and the GPU memory it is possible to efficiently overcome the drawbacks of the increased number of data transfers when solving the out-of-core problems on GPU, Figure 6.4. Moreover, by applying OOC techniques, we not only efficiently overlap data transfers, but also attain significantly higher performance and improve the scalability of the algorithms with the increasing matrix dimension.

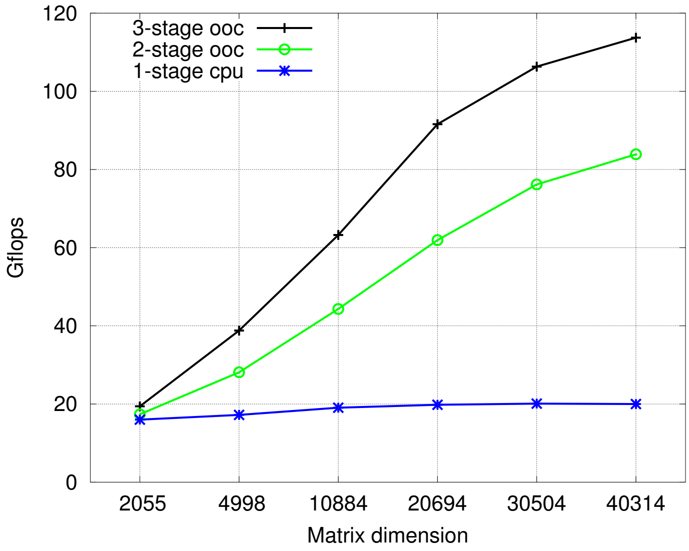


Figure 6.4: Performance scaling of 3-stage (black) and 2-stage (green) OOC variants for different matrix sizes for `1cwp` MD collection.

6.4 The Krylov subspace approach

In this section we present the experimental results of the Krylov-subspace approaches applied on the collection of MD problems, Table 6.1. In Section 5.2 are presented two versions of the Krylov subspace approach. The first approach, KE, explicitly constructs matrix $C := U^{-T}AU^{-1}$ from the generalized eigenvalue problem, while the KI approach implicitly operates with C within each Lanczos iteration. Note that the reduction from generalized to standard eigenproblem is a part of the Krylov-subspace approaches and is included in the experiments results. Table 6.7 lists all routines required for both Krylov subspace approaches.

In NMA the main computational bottleneck is the computation of the s smallest eigenvalues, representing frequencies, and the corresponding eigenvectors or normal modes. However, the Krylov-subspace approach, as an extension of power method, converges towards the largest eigenvalue. Thus, in order to accelerate the convergence of the Lanczos iteration we compute the s largest eigenpairs of the inverse eigenproblem:

$$BX = AX\Lambda^{-1}, \tag{6.8}$$

where A and B correspond to the Hessian and kinetic matrices, respectively, and Λ contains the eigenvalues of the generalized problem. In the following experiments, instead of computing the smallest s eigenvalues of the original problem, we compute the largest s eigenvalues of the inverse problem.

The routines required for the reduction to an standard eigenproblem, GS1 and GS2, as well as the back-transform BT, are computationally-bound routines and thus appropriate to be executed on the GPU even if matrices A and B exceed the total available GPU memory. Since the Cholesky factorization (GS1) is beyond the scope of this research, we use a GPU out-of-core implementation of the MAGMA computational library. Although the TRSM routine required in steps GS2 and BT is highly appropriate for an out-of-core GPU execution, the implementations from MAGMA and CUBLAS computational libraries can only be applied if the required storage spaces of A and U do not exceed the total GPU memory. Otherwise we use our own OOC_GPU_DTRSM routine for GS2 and BT that efficiently operates even if the matrices cannot fit into the GPU memory.

Var.	Operation		Routine	Library
GS	GS1	$B = U^TU$	MAGMA_DPOTRF	MAGMA
	GS2	$C := U^{-T}AU^{-1}$	OOC_GPU_DTRSM	–
KE	KE1	$z_{k+1} := Cw_k$	DSYMV	MAGMA/CUBLAS
	KE2	$z_{k+1} \rightarrow w_{k+1}$	DSAUPD	ARPACK
	KE3	$T_m, V_m \rightarrow \Lambda, Y$	DSEUPD	ARPACK
KI	KI1	$\bar{w}_k := U^{-1} w_k$	DTRSV	LAPACK/MAGMA/CUBLAS
	KI2	$\hat{w}_k := A\bar{w}_k$	DSYMV	LAPACK/MAGMA/CUBLAS
	KI3	$z_{k+1} := U^{-T}\hat{w}_k$	DTRSV	LAPACK/MAGMA/CUBLAS
	KI4	$z_{k+1} \rightarrow w_{k+1}$	DSAUPD	ARPACK
	KI5	$T_m, V_m \rightarrow \Lambda, Y$	DSEUPD	ARPACK
BT	BT	$X := U^{-1}Y$	OOC_GPU_DTRSM	–

Table 6.7: Routines required to build Krylov subspace approaches. GS: reduction to standard eigenproblem, KE: explicit Krylov, KI: implicit Krylov, BT: back-transformation.

Note that all algorithms required by the KE and KI variants are memory-bound level-2 BLAS routines with KE1, KI1, KI2 and KI3 the most time consuming and computationally costly parts. Nevertheless, the MAGMA and CUBLAS implementations of the routines required in these steps can attain much higher performance than the LAPACK multi-core CPU variants as far as the input matrix (C, U or A) can fit into the GPU memory; see Figure 6.5. The maximum in-core GPU problem that can be solved by using conventional GPU computational libraries, on our testing system, is n =13,000.

Figure 6.5 illustrates the potential of the GPU to accelerate the execution of both Krylov-subspace variants. The major drawback of both Krylov-subspace variants is that the main computational routines, TRSV and SYMV, cannot achieve further improvement in performance if the problem exceeds the total GPU memory. Moreover, the out-of-core GPU implementations are an order of magnitude slower than the multi-core implementations due to the very poor ratio between data transferred and computations performed on the GPU. Therefore, for the solution of large eigenproblems, our GPU Krylov-subspace variants KE and KI use the traditional multi-core implementations fro TRSV and SYMV

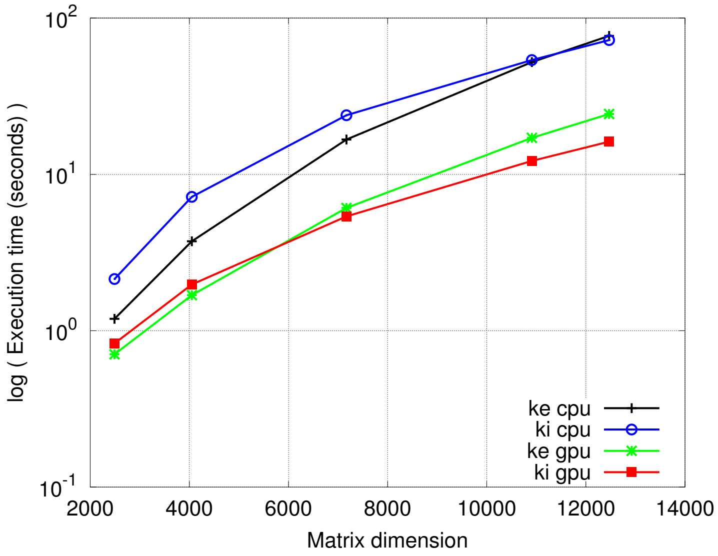
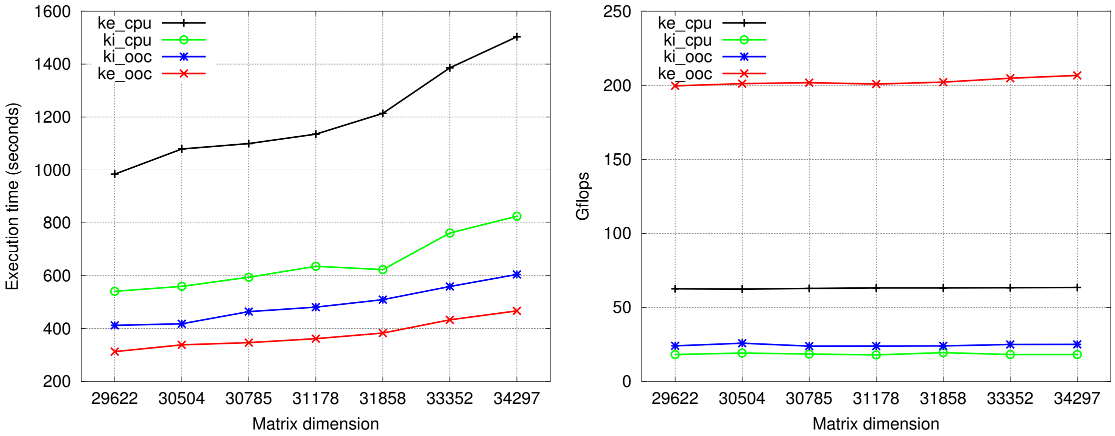


Figure 6.5: Execution time of in-core GPU and multi-core versions of Krylov-subspace variants KE and KI.

from LAPACK. On the other hand, the reduction of generalized to standard eigenvalue form, which consists of compute-bound routines, can be efficiently accelerated even when data are too large for the GPU memory. The total algorithmic cost for the KE and KI variants are $7/3n^3 + (2k+s)n^2$ and $n^3/3 + (4k+s)n^2$, respectively, where s is the number of sought-after eigenvalues, and k number of iterations.

Figure 6.6 illustrates the performance of our hybrid GPU Krylov-subspace variants in which the traditional Cholesky (POTRF) and TRSM kernels are replaced with the MAGMA out-of-core Cholesky implementation and our out-of-core OOC__GPU__DTRSM routine. Although steps KE1 and KI1–3 are performed on the multi-core CPUs, the out-of-core Krylov variants achieve much higher performance than the multi-core variants. The execution time here includes the reduction of generalized to standard eigenproblem as well as the data transfers to/from GPU.



(a) (b)

Figure 6.6: Execution time (left) and GFLOPS (right) of the two GPU out-of-core Krylov-subspace variants. Only 100 largest eigenvalues are computed.

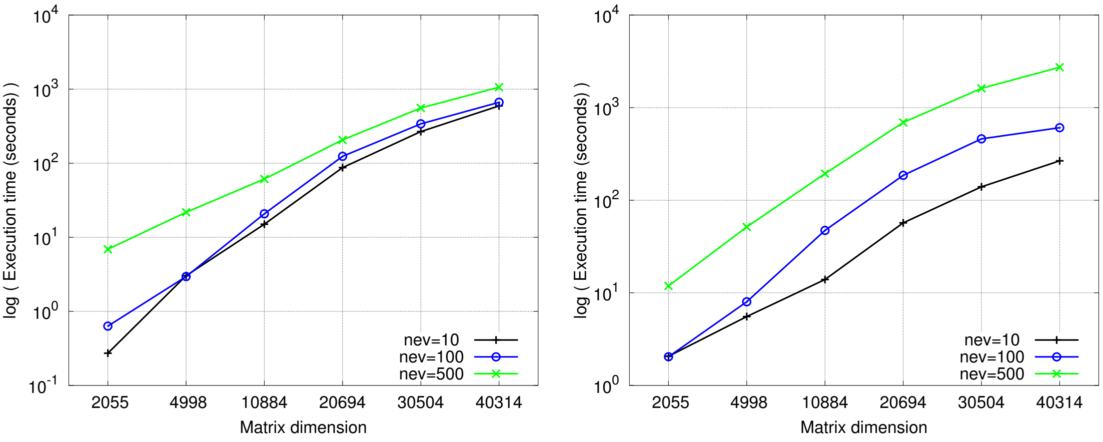
The significant speed-up of both OOC KE and KI variants is achieved because the computationally intensive GS1 and GS2 routines are performed on the GPU. The Krylov-subspace kernels KE and KI are still performed completely on the multi-core CPU. Multi-core and OOC variants of the KI approach deliver much lower performance, Figure 6.6(b), than the KE variant since the majority of the computation is spent in the level-2 BLAS kernels required in steps KI1–3. In the explicit Krylov subspace variant, the memory-bound computations from steps KI1 and KI3 are performed in advance in step GS2 via two calls to the compute bound level-3 BLAS kernel TRSM; see Table 6.8. In the multi-core implementation of KE most of the execution time is spent in steps GS1–2, and thus this routine exhibits a significant speed-up if it is executed on GPU (3.7× speed-up for n=40,314). In the KI variant, half of the execution time is spent in steps KI1 and KI3. However, these two steps cannot benefit from the out-of-core GPU execution and remain unchanged. KI outperforms KE on the multi-core but the KE variant outperforms KI if GPU is employed for almost all cases. This is because the KI variant only benefits from the out-of-core Cholesky factorization (GS1) on the GPU while the construction of C is performed on the CPU during each Lanczos iteration.

CPU	KI				KE			
dimension	GS1	KE1+KE3	KI2	Total	GS1	GS2	KE1	Total
2,055	0.06	1.43	0.24	2.01	0.06	0.36	0.22	0.84
4,998	0.77	6.81	1.1	9.44	0.79	5.17	1.09	7.67
10,884	8.52	28.45	4.93	42.98	8.52	35.38	4.94	50.06
20,694	62.62	110.57	17.69	194.34	62.63	246.67	17.65	330.32
30,504	207.48	218.32	38.52	470.89	207.51	790.59	38.87	1042.42
40,314	481.4	443.09	69.13	1002.64	481.42	1781.3	68.39	2339.69
OOC GPU	KI				KE			
dimension	GS1	KE1+KE3	KI2	Total	GS1	GS2	KE1	Total
2,055	0.03	1.31	0.22	1.72	0.03	0.15	0.19	0.58
4,998	0.27	6.12	1.05	7.81	0.26	1.25	1.06	2.95
10,884	1.97	28.7	4.94	46.58	1.96	11.02	4.94	19.14
20,694	12.49	111.9	17.39	144.67	12.24	67.55	17.29	99.97
30,504	38.57	256.55	40.57	345.97	38.27	209.63	40.47	295.94
40,314	87.46	446.69	68.28	607.98	87.33	470.08	69.53	635.96

Table 6.8: Total time in seconds for the implicit (KI) and explicit (KE) Krylov subspace variants on multi-core processors and GPU.

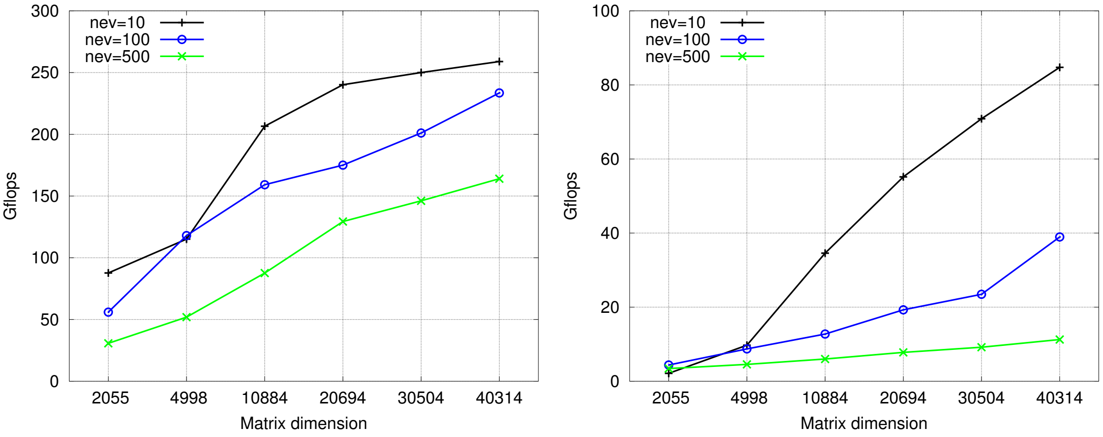
In molecular dynamics only the s smallest (in our case s largest of the inverse eigenproblem) eigenvalues are required. However, for each MD collection, one can decide on the number of DoF. With a larger freedom number, the collection problems become larger. Nevertheless, in practice, only the 1% smallest eigenvalues are required. In all previous tests we compute only the largest 100 eigenvalues; however, for specific MD problem, with dimension over 30,000, more eigenvalues are required. Figure 6.7 presents the performance of our OOC Krylov subspace variant when a different number of eigenvalues is computed.

In Table 6.8 we showed that the OOC KE variant is faster than the KI variant for problems of dimension up to 40,000 when only the 100 smallest eigenvalues are required. However, if a larger number of eigenvalues is required (e.g. 500), then KE is faster. If only 10 eigenvalues are required, KI is almost twice faster than KE. The fast increase



(a) Execution time in seconds for KE.

(b) Execution time in seconds for KI.



(c) GFLOPS rate for KE.

(d) GFLOPS rate for KI.

Figure 6.7: Execution time and GFLOPS rate for KE (a),(c) and KI (b),(d) on GPU for MD collection `1cwp` with 10, 100 and 500 largest eigenvalues computed.

in execution time of the KI variant when a larger number of eigenvalues is required is because of its slower convergence; i.e. a higher number of Lanczos iterations. For example, if 10 eigenvalues are required, the average number of Lanczos iterations is 85; for 100 eigenvalues is between 250 and 325; and for 500 eigenvalues is 1250. Note that in the KE variant, the GS2 step, explicit construction of C, is performed at the beginning and does not depend on the number of eigenvalues required, i.e. the number of iterations performed. However, in the KI variant the work performed in steps KI1 and KI3, which cannot be accelerated with GPUs rapidly increases with the number of iterations resulting in longer execution time.

Furthermore, the GFLOPS rates illustrated in Figure 6.7(c) show that we cannot expect any further raise in performance for the KE variant by working with larger problems. However, the performance of the KI variant continues to grow with the problem dimension, Figure 6.7(d). This is because the performance of the KE variant is dominated by the level-3 BLAS steps GS1 and GS2 which are independent of the number of iterations while the performance of KI when only 10 eigenvalues are required, rapidly grows because a small number of Lanczos iterations do not still dominate execution part, as is the case when 500 eigenvalues are computed. The conclusion is that the KI GPU OOC variant is more efficient when a very small number of eigenvalues (no more than 100) is required for very large matrices. On the other hand, the explicit Krylov-subspace approach is the method of choice when a larger number of eigenvalues is required.

6.5 Spectral D&C approach

The last out-of-core GPU eigensolver is the spectral divide-and-conquer (D&C) QDWH based algorithm. It belongs to a group of eigensolvers that implement divide-and-conquer strategy. The division of the starting eigenproblem into subproblems is realized through an iterative algorithm. Therefore, this algorithm can be also considered as an iterative method. In contrast to Krylov subspace iterative eigensolvers, which are mostly composed of memory inefficient level-2 BLAS routine, that achieve very poor performance when the data are out-of-core for GPU, this approach can be cast completely in terms of level-3 BLAS operations, Table 6.9. Thus, the SD&C algorithm can deliver a much higher GFLOPS rate by employing a GPU than the Krylov-subspace approaches, but requires significantly higher computational cost; see Table 6.2.

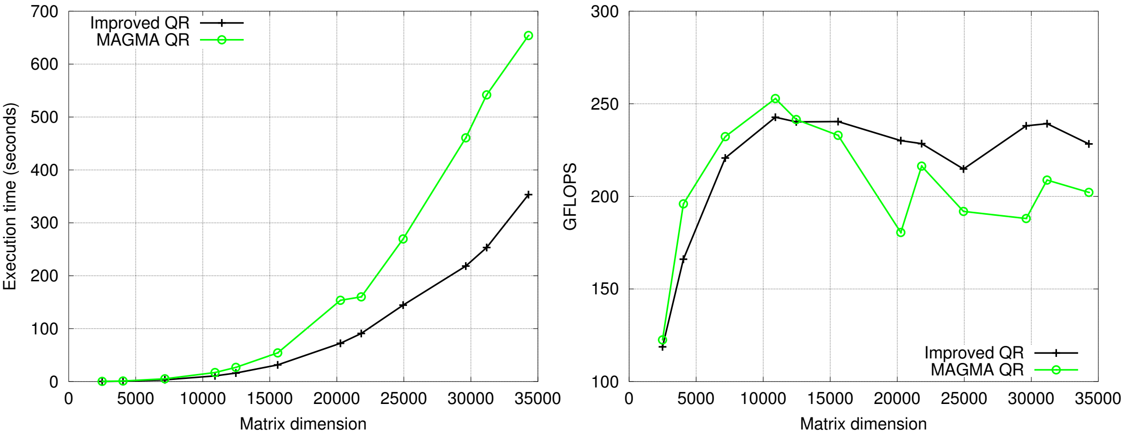
Executional step	Operation		Routine	Library
QDWH	QDWH1	$X_i \rightarrow \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$	DGEQRF	–
	QDWH2	Construct Q	DORGQR	–
	QDWH3	$X_{i+1} = \alpha X_i + \beta Q_1 Q_2^T$	DGEMM	–
SUBS	SUBS1	$X \rightarrow QR$	DGEQRF	MAGMA
	SUBS2	Construct Q	DORGQR	–
	SUBS3	$E = V_2^T A V_1$	DGEMM/DSYMM	MAGMA/CUBLAS
	SUBS4	$X := CV_1$	DSYMM	MAGMA/CUBLAS
	SUBS5	$A_1 := V_1^T A V_1$	DGEMM/DSYMM	MAGMA/CUBLAS
EIG	EIG	$A_1 \rightarrow \Lambda, Y$	DSYEVR	LAPACK

Table 6.9: Routines required to build spectral divide–and–conquer OOC-GPU eigensolver.

As discussed in Section 5.3, from the computationally point of view, most significant part of the OOC-GPU SD&C eigensolver is the computation of polar factor via the QDWH algorithm (QDWH step), Subsection 5.3.2, Algorithm 5.11. This step employs three computational routines: QR factorization of a $2n \times n$ matrix (QDWH1), explicit construction of the orthogonal transforms Q (QDWH2), and update of X_{i+1} (QDWH3), with a computational complexity $\mathcal{O}(n^3)$. The kernels required for QDWH3 were already presented in the previous sections and they are proved to achieve very high performance even when the data exceed GPU memory. An efficient kernel for the QDWH1 step, when X_i cannot fit into the GPU memory, can be found in MAGMA. Although the MAGMA kernel achieves a very high GFLOPS rate, the execution time is the main drawback, since the step QDWH1 is repeated at each QDWH iteration.

As discussed in Subsection 5.3.2, the QR factorization can be improved by exploiting the sparsity of X_i (note that, by construction, $X_i(n+1:2n, 1:n)$ is the identity matrix). For that reason we constructed a new left-looking, slab-oriented OOC-GPU QR factorization whose performance is illustrated in Figure 6.8. If we compare our algorithm with the OOC MAGMA implementation, the former exhibits a lower computational time, but a higher GFLOPS rate. This proves that our QR factorization is scalable with the problem dimension and delivers more than 230 GFLOPS. Note that our approach requires only $2n^3$ flops compared to the $10/3n^3$ flops required by OOC MAGMA QR factorization.

The algorithmic cost is reduced by introducing the two-level blocked kernels that operate only on non-zero entries of X_i, while the lower triangular is not referenced. The first level of blocking, in which the panel width s is large enough (up to 12,000 on the testing system) so that one panel can fit into the GPU memory, optimizes the data transfers to/from GPU memory. The second level of blocking, with the panel width $b \leqslant 128$, improves the performance on the multi-core (QR factorization of small panels and constructing Q). The size of the panels is chosen so that the lower triangle of X_i is never referenced.



(a) Execution time.

(b) Gflops rate.

Figure 6.8: Execution time (left) and Gflops rate (right) of improved OOC-GPU QR algorithm compared to MAGMA OOC-GPU implementation.

Because the OOC QR algorithm exhibits less flops, the computational time is lower and becomes more significant as the dimension increases, Figure 6.8(a). For large matrices the computational time for MAGMA QR is almost twice slower than our approach. Although MAGMA QR explicitly overlaps data transfers to GPU with execution (using CUDA streams), our approach delivers higher performance, Figure 6.8(b). This is because in our approach we compensate data transfers with better data re-use, once they are in the GPU memory, and because of the reduced number of transfers required between the main memory and GPU.

At the time of writing of this thesis, there is no GPU implementation of the DORGQR routine, required by QDWH2. However, this kernel is mostly made of level-3 BLAS kernels and thus can achieve very high performance even if the matrix exceeds the GPU memory. Our OOC-GPU kernel is a left-looking, slab-oriented approach that incurs less memory transfers and exhibits the same performance as our OOC-GPU QR factorization; see Figure 6.9. All three kernels required by QDWH exhibit a very high GFLOPS rate. This is because they are all cast completely in terms of compute bound level-3 BLAS operations that successfully overcome the negative effect of data transfers, even if the data exceed the available GPU memory.

The subspace iteration part (SUBS) requires the same routines as QDWH. However, in this part all matrices are dense and symmetric; therefore the kernels from MAGMA library can be employed, e.g. the MAGMA out-of-core QR factorization. If the splitting point σ is chosen so that the number of columns of V_1 is small, the SUBS part becomes an

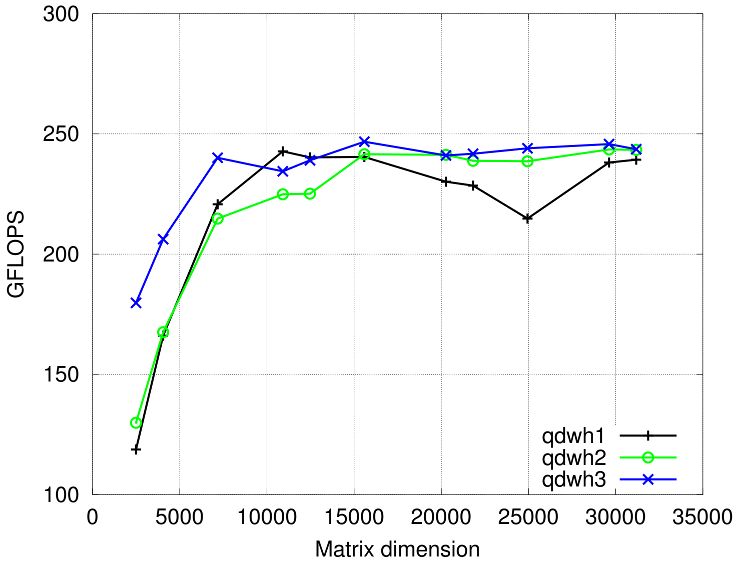


Figure 6.9: Performance of the QDWH1 (QR fact.), QDWH2 (construction of Q) and QDWH3 (update X_{i+1}) kernels.

in-core problem from the GPU viewpoint. In that case, MAGMA and CUBLAS kernels are employed in all calculations, expect for the kernel SUBS2, whose GPU implementation does not exist in any conventional GPU computational library.

The last executional part, EIG, computes the eigenvalues and the corresponding eigenvectors of the subproblem A_1. Since the number of eigenpairs required in normal mode analysis simulations is small, this step is negligible and can be computed in-core on GPU or via multi-core processor by employing standard LAPACK kernels.

Without loss of generality, in the following experiments we compute the largest s eigenpairs of problem A. The main idea of the OOC-GPU algorithm lies in choosing a splitting point σ that separates the problem A into subproblems in which subproblem A_1 contains a superset of the desired eigenvalues. With a small subproblem A_1, finding the sought-after eigenpairs from it becomes negligible even if traditional computational libraries, such as LAPACK, are employed.

Thus, σ has to be chosen such that the sought-after eigenvalues are a subset of the eigenvalues of A_1; i.e. the dimension of A_1 is slightly larger than s. The key is to select the value of σ that will separate the spectrum of A into two subsets (sub–matrices) A_1 and A_2, with the dimension of A_1 being equal to or only slightly larger than s. In [115] the author of this thesis, together with other authors, presented three σ strategies for the specific macromolecular problem:

SD&C-A. $\sigma = \text{mean}(\text{diag}(A))$, where $\text{mean}(\cdot)$ denotes the arithmetic average value of its argument.

SD&C-B. $\sigma = 4 \times \text{mean}(\text{diag}(A))$.

SD&C-C. In this case, given a macromolecule, we use IMOD to generate the Hessian and kinetic matrices pair (A, B) for different values of n, with larger dimensions offering much higher accuracy and reliability in the subsequent simulation. Concretely, we generate a matrix pair (A, B) for a problem of much smaller dimension, say $m \approx 1{,}000$, and use this to investigate the distribution of the eigenvalues of the problem by computing them. We choose σ as the $(s \cdot m/n)$-th largest eigenvalue of the small problem, where s is the number of sought-after eigenvalues.

The third strategy allows us to use a conservative value of σ, which can then be applied in the "extended" problem to split the eigenvalues at the appropriate point. In the subspace iteration algorithm, the value $k = \frac{1}{2}\|U_p + I_n\|_F^2$ indicates the number of eigenvalues in A_1. Therefore, in case $k < s$, the QDWH iterate has to be recomputed with a larger value for σ, but in the practical MD problem this case never occurred. After the first successful split, the eigenpairs of A_1 are computed directly by employing a direct in-core eigensolver based on the reduction to tridiagonal form with a negligible cost compared with that of the initial stage.

The splitting points for different σ strategies are presented in the Table 6.10 for the MD collection problems UTUB10, UTUB20 and UTUB40 with different values for the problem dimension. The SD&C-A splitting, the strategy used in the original algorithm [107] when all eigenvalues are required, aims at splitting the problem into two equal-size subproblems. As expected, this strategy does not provide a good approximation of the splitting point required by the OOC-GPU algorithm. For larger problem dimensions, the splitting point approaches 10,000 giving a large dimension for subproblem A_1. Therefore, to obtain a better guess, we should move up, from the middle of the spectrum, to the left edge of the spectrum if the smallest eigenvalues are required (e.g. by dividing by 4) and to the right edge for largest eigenvalues (e.g. times 4) as in the SD&C-B strategy. For most problems, this strategy gives the best splitting points. The SD&C-C strategy computes all the eigenvalues of the smaller problems, with the size $m = \{2490, 4050, 7170\}$, and then chooses σ to be the $(s \times m/n)$-th smallest eigenvalue of the smaller problem. This strategy gives good splitting points for small size matrices, but for larger matrices the splitting point gets large. The reason is that the eigenvalues, for the particular MD collection, are clustered around the smallest eigenvalue. As illustrated in Table 6.10, the SD&C-B and SD&C-C strategies provide a good splitting point that keeps the dimension of obtained subproblem A_1 small enough so that the required eigenpairs can be computed directly without performing any further splittings into smaller subproblems. By choosing these two splitting strategies, the QDWH remains the only computationally-intensive part on which more than 80% of the total SD&C execution time is spent.

Problem dimension	Splitting point k		
	SD&C-A	SD&C-B	SD&C-C
2490	704	105	–
4050	1160	137	–
7170	2204	185	–
10910	3816	272	347
12469	4523	359	536
15588	4881	537	508
20266	7215	488	709
21822	7158	533	1322
24943	8678	712	1024
29622	9006	936	1409
31178	9288	815	2511

Table 6.10: Splitting points for σ strategies SD&C-A, SD&C-B and SD&C-C.

In the second experiment we analyze the performance of the iterative QDWH algorithm for the polar factor U_p. The total executional time, proportional to the number of iterations of QDWH, depends on the values of parameters α and l_0, Section 4.4. These values are chosen such that $\alpha \gtrsim \|A\|_2$ and $l_0 \lesssim \sigma_{min}(A)$; i.e. l_0 is smaller than the smallest eigenvalue of the problem A. If an eigenvalue of A exists that is smaller than l_0, then more iterations are required to converge to U_p, see Table 6.11.

The value for α can be easily approximated by computing the square of the product of the 1-norm and infinity norm of matrix A. Approximating the parameter l_0 is more complicated. In `matlab` and `octave` it is computed by calling the `condest` function that estimates the 1-norm condition number. In our experiments with SD&C-C approach, we exploit the eigenvalues of the smaller problem of the same collection. The value for l_0 is set to be slightly smaller than the smallest eigenvalue of the same MD collection but with smaller problem dimension, usually $n < 1000$. Therefore, we can guarantee that the value for l_0 will always be lower than the smallest eigenvalue, and that QDWH will perform no more than 7 iterations [107].

Problem dimension	Strategy					
	SD&C-A		SD&C-B		SD&C-C	
	#iter	time	#iter	time	#iter	time
4050	7	17.06	7	17.11	–	–
12469	7	335.48	7	336.84	6	283.55
21822	8	1834.80	7	1608.97	6	1414.34
31178	10	6218.09	7	4370.31	6	3967.78

Table 6.11: Number of iterations and execution time (in seconds) for QDWH algorithm applied on MD collection UTUB20.

The number of iterations performed does not depend on the σ factor, as the strategy names suggests. However, in the SD&C-C strategy, we exploited the eigenvalues from a problem with a smaller dimension to make a better guess for parameter l_0. Observe that the SD&C-C strategy requires less iterations than other strategies. This is in-line with the work of Nakatsukasa [107], in which a number of iterations is shown to be necessary less than 7 for all practical cases, if l_0 is strictly smaller than the smallest eigenvalues. That implies that the guesses for l_0 (we use $l_0 = \sqrt{n\epsilon}$, ϵ with the machine roundoff error) in strategies SD&C-A and SD&C-B are not always the best. Despite the better σ guess for the SD&C-B strategy that delivers smaller subproblem A_1, the SD&C-C strategy requires less iterations to compute the polar factor and thus significantly outperforms the SD&C-B strategy.

6.6 Comparison of OOC-GPU algorithms

In this section we give a side-by-side comparison of the three OOC-GPU eigensolvers developed. The goal of this analysis is to present the performances and possibilities to tackle large macromolecular problems on a modest computing infrastructure equipped with one or two multi-core processors and one GPU computing device. The macromolecular dynamics test cases, Table 6.1, are generated using IMOD so that the test cases do not fit into the GPU memory but are small enough for the main memory. Since our testing system is equipped with 48 GB of the main memory, the largest MD problems range from 29,000 to 34,000 degrees of freedom (Table 6.1). Furthermore, since the functional motions are described by the smallest frequencies, in practice, only 1% of the smallest eigenvalues are required to describe the functional motions of macromolecules. Thus, in our MD test cases, we compute ≈ 300 smallest eigenvalues.

In our analysis, we consider a multi-core implementation of the one-stage approach (routine DSYEVR from LAPACK library) as the state-of-the-art eigensolver. Although different approaches and implementations that exploit GPU devices exist, none of them is capable of solving problems whose storage space exceeds that of the GPU memory. Therefore, the LAPACK implementation is the only one competitive eigensolver capable of solving large eigenproblems ($n >$30,000) on one-node computing systems.

In the previous section we analyzed the OOC multi-stage eigensolver, the Krylov subspace and the spectral divide–and–conquer based on the QDWH eigensolver. For these experiments, we choose the version of each OOC-GPU eigensolver that requires the lowest computational time; therefore, for the solution of the MD test cases we choose OOC-GPU 3-stage eigensolver, explicit Krylov-subspace approach, KE and spectral D&C with the SD&C-A strategy. Figure 6.10 shows the speed-up of our OOC-GPU eigensolvers compared with the multi-core variant of the one-stage eigensolver ran on all 8 cores of the testing system. The analysis of all three OOC-GPU eigensolvers includes the cost of the reduction from generalized to standard form as well as the back-transform. The explicit Krylov-based eigensolver achieves almost a 10× speed-up compared with LAPACK, for the largest test cases (UTUB40 $n = 34297$). In general all OOC-GPU eigensolvers achieve significant speed-ups for the largest test cases because the execution time for the 1-stage eigensolver rapidly increases for the largest test cases; see Figure 6.3(a). When the problem dimension increases, the 1-stage eigensolver pays a price due to the slow memory-bound level-2 BLAS operations that also manifest in the decrease of GFLOPS rate, Figures 6.3(b) and 6.11.

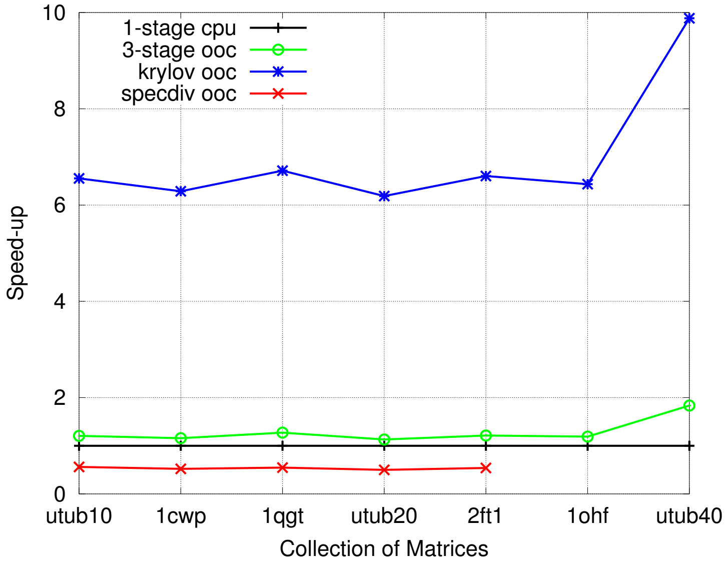


Figure 6.10: Speedup of three OOC-GPU eigensolvers relative to 1-stage CPU eigensolver. Only 1% smallest eigenvalues are computed.

On the other hand, the OOC-GPU eigensolvers maintain very high performance even for the largest MD problems, Figure 6.11. At this point, it is worth noticing that the higher GFLOPS ratio for SD&C does not imply a shorter execution time, Figure 6.10, as this method also exhibits a much higher flop cost ($30n^3$ if only 6 iterations are performed) than the 3-stage ($14/3n^3$) and the Krylov subspace ($7/3n^3$) alternatives. In any case, the experiment serves its purpose, showing that the OOC-GPU multi-stage, Krylov and SD&C eigensolvers deliver, respectively, sustained rates of 110, 200 and 210 GFLOPS for the largest problem size. More importantly, the trends revealed by this experiment indicate we can hardly expect a raise in the GFLOPS rate when working with larger problems (even if they fit into the main memory) as the performance rates are quite flat for the largest two problem sizes.

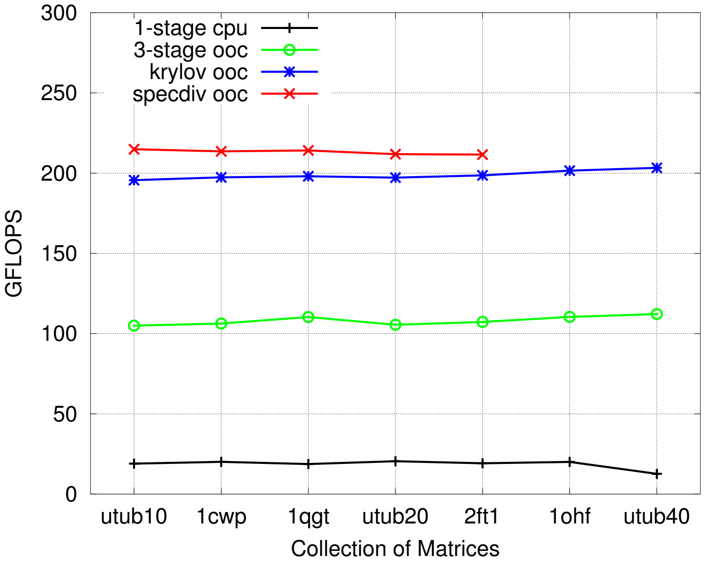


Figure 6.11: GFLOPS rate of the OOC-GPU eigensolvers applied to MD problems. Only 1% smallest eigenvalues are computed.

Although we stated that the Krylov subspace-based approaches cannot efficiently exploit the GPU performance when the data are out-of-core from the GPU viewpoint, KE variant exhibits a very high GFLOPS rate. This is because the majority of the computations in KE is performed in the reduction from generalized to standard form, by employing highly optimized Cholesky and DTRSM routines that successfully overlap the data transfers with computation. Moreover, the iteration step, which is exclusively level-2 BLAS oriented, does have a strong impact on the total execution time. Thus, the performance of the KE variant is not limited with memory-bound but with the compute-bound kernels. However, if the number of sought-after eigenvalues increases, the impact of the memory-bound kernel (DSYMV) becomes larger and performance is expected to drop.

The 3-stage OOC-GPU eigensolver does not achieve the best Gflops rate of all eigensolvers, as it was expected. This is mainly because of the last step, reduction from narrower to tridiagonal form that, unlike reduction to band form, becomes the most time consuming part of the algorithm. However, the 3-stage eigensolver demonstrates excellent scalability with the problem dimension. The spectral divide-and-conquer eigensolver, although exhibits much higher computational cost, attains the highest GFLOPS rate. With some eventual development of the polar decomposition algorithm, it is expected that this approach becomes competitive with the multi-stage and Krylov-subspace based eigensolvers.

The conclusion is that the explicit Krylov-subspace approach is the best eigensolver for the targeted MD problems and their dimension. However, its performance drops with the number of the eigenvalues because of computationally inefficient matrix-vector product. On the other hand, if larger problems or more eigenvalues are required then the OOC 3-stage eigensolver is a better solution. Furthermore, the 3-stage eigensolver has a great potential of being extended to other computing platforms equipped with various memory levels. In such systems, the block size can be fine-tuned to better exploit the underlying computational infrastructure, for example in order to optimize the memory transfers between disk and the main memory or between local and distributed memories, by adjusting block sizes with respect to the memory capacity and bandwidth.

Chapter 7

Conclusion

Nowadays large dense symmetric eigenproblems are solved on various computing architectures, ranging from scalable supercomputers to large computing clusters that consist of a number of nodes equipped each with one or more multi-core processors. These large systems can provide enough computing power and the level of scalability to efficiently meet the needs of large eigenvalue problems. Naturally, with the maturity of general-purpose GPUs, these traditional computing systems were extended by adding GPU accelerators to each computational node thus providing further speedup over CPU-only systems. A key factor to exploit these systems, called hybrid systems, is to improve load balancing among the different computational units, such as processing cores, multi-core processors, GPUs and computational nodes.

The GPUs, because of their unique architecture, are especially suitable for vector parallel, compute-bound operations, that a majority of the linear algebra operations are composed of. Modern optimization approaches aim to improve the performance of computationally-intensive parts by off-loading them to GPUs while performing less computationally-intensive and memory-bound operations on more suitable multi-core processors. Such approaches can attain very high performance, as far as the required data are kept in the GPU memory. However, if the problem size exceeds the GPU memory, the performance drastically drops due to the PCI-e latency and bandwidth constraints.

The keep the pace with the demands on high performance of GPUs even for problems whose data are too large to fit the GPU memory, current efforts aim to improve the hardware scalability, i.e. improve the algorithms that efficiently scale to a very large number of GPU accelerators. Thus, it is expected that such algorithms will greatly benefit by adding more GPU accelerators to the system. However, this approach has two main drawbacks: procurement and maintenance costs and the limited amount of aggregated memories of all attached GPUs. The latter is the cause of many problems connected with the scalability and expensive memory transfers between the main memory and the GPU memory. The consequence of this is that, the GPU accelerators cannot be applied to if the problems are large enough, i.e. out-of-core for GPU.

7.1 General contributions

Opposite to modern approaches that mostly tend to improve the scalability and the performance of the eigensolvers by improving the memory distributions, overlapping transfers and executions on large computational resources this thesis proposes an alternative path by revitalizing the out-of-core techniques applied to GPUs. Although out-of-core techniques are presently not mainstream in high performance computing they are becoming more important. In particular, as the number of computational cores grows, the amount of memory per core is decreasing. The memory bandwidth and latency thus become the main bottlenecks since the computational units are waiting for data to be transferred to the local memory (e.g. cache or GPU memory). In this research, the GPU memory is considered as the local memory and utilized as a cache memory for GPU processors. This concept is feasible since GPU memory has very low latency and exhibits very high bandwidth providing very high performance for data already residing in the GPU memory.

This thesis is devoted to the solution of large dense symmetric eigenproblems on a CPU-GPU hybrid system equipped with a single GPU. The main motivation of this work is that recent dense symmetric eigensolvers, though attaining significant speedup compared to the tuned versions for modern multi-core CPUs, cannot solve problems if the required storage space exceeds the GPU memory. To address this obstacle, this thesis aims to improve a set of advanced linear algebra routines by applying out-of-core techniques from the perspective of the GPU. These developed out-of-core routines are then applied to improve the existing dense symmetric eigensolvers for the GPU-based hybrid systems. In addition, the out-of-core routines developed within this research, such as matrix-matrix multiplications and QR factorization, can improve the performance of other linear algebra problems that do fit into the GPU memory. However, to exploit the potential of GPU accelerators to their maximum, additional improvements in eigensolvers are necessary.

In this thesis we have demonstrated that real eigenproblems, for example those that arise in molecular dynamics and were traditionally solved on clusters of multi-core processors, can also be efficiently solved on a hybrid computing systems equipped with a single GPU accelerator. The developed out-of-core algorithms exploit the performance of the GPU accelerators even when the problem cannot entirely fit into the GPU memory. Moreover, the obtained performances are comparable with those of the in-core GPU variants. The experimental evaluations show that the large eigenproblems can be solved even on modest computational hardware with a minimal decrease in performance.

Specifically, this thesis addresses three symmetric dense eigensolvers: the eigensolver based on the multi-stage reduction, the Krylov subspace-based method, and the spectral divide–and–conquer algorithm. For two eigensolvers; the multi-stage reduction and the spectral divide–and–conquer, it is shown that by careful orchestrating the transfers between the main and the GPU memory, very large eigenproblems can be efficiently solved on a hybrid system equipped with a single GPU without significant drop in the overall performance. Furthermore, the negative effects of data transfers between local memory (i.e. GPU memory) and the main memory are efficiently overcome by carefully preserving data locality. The three targeted out-of-core eigensolvers significantly outperform their highly tuned multi-core counterpart versions even if the data is too large to fit into the GPU memory. Furthermore, the peak performance of the out-of-core eigensolvers is comparable with those of the in-core GPU versions for problems that can fit into the GPU memory.

The performance of the three out-of-core eigensolvers is demonstrated on real problems that arise in the molecular dynamics. The eigenproblems to be solved in molecular dynamics experiments are dense and symmetric requiring only a small subset of eigenvalues and the corresponding eigenvectors to be computed. Furthermore, the starting problem is in generalized form and its reduction to standard eigenvalue problem exhibits extremely high performance when executed as out-of-core. The developed out-of-core routines applied to the reduction to standard eigenvalue form are perfectly scalable to any problem dimension and deliver a constant GFLOPS rate, comparable to the in-core variants, for large enough matrices.

A detailed comparison of the developed out-of-core eigensolvers with molecular dynamics cases shows that the out-of-core Krylov-subspace eigensolver though a memory-bound operation, is the fastest approach. However, as the number of the required eigenvalues and eigenvectors increases, the execution time of the Krylov subspace eigensolver increases as well due to an increased number of iterations. Generally, the out-of-core Krylov-subspace method is the method of choice for any problem dimension if the number of sought-after eigenvalues is small (experimentally, the best performance are achieved when at most 1% eigenpairs are required). On the other hand, if all eigenvalues or a large subset of eigenpairs are required, then the multi-stage method is the best solution, since the time for the reduction to tridiagonal form is constant and does not depend on the number of eigenpairs. The cost of finding eigenvalues and eigenvectors of tridiagonal form is negligible compared to the rest of the algorithm.

In addition to the general contributions of the thesis, some specific contributions for each eigensolver are presented in the following subsections.

7.1.1 OOC multi-stage reduction

Traditionally, direct eigensolvers, such as the multi-stage reduction, are applied to small eigenproblems. The reason for that is in its high arithmetic cost ($\mathcal{O}(n^3)$) flops, especially when the eigenvectors are required as well, due to the explicit accumulation of the Householder reflectors. The evaluation reveals that the most time consuming stages are reduction from dense symmetric to band form and from band to narrower band form. By off-loading these two stages to the GPU, a significant speedup is expected to be achieved. However, for large enough problems, the straight-forward solutions, such as the solutions that can be found in the MAGMA computational library, are not applicable since the problem does not fit into the GPU memory.

To address this out-of-core scenario, a new reduction from dense symmetric to band form was designed and implemented. The new reduction algorithm re-organizes the state-of-the-art SBR toolbox routines for reduction to band matrix form. Specifically, a hybrid CPU-GPU QR factorization of the leading panel and a two-sided updated of the trailing matrix were designed. The second stage, the reduction from band to narrower band form, though rich in BLAS-3 operations, has an expensive bulge-chasing phase, which requires a lot for copying to and from the memory if the eigenvectors are required. Therefore, to maximize the performance and decrease the number of memory transfers, this stage was designed and implemented as an in-core GPU algorithm. In order to optimize the execution of the second stage, the bandwidth has to be small enough so that the matrix, stored in the band storage representation, entirely fits into the GPU memory.

The experiments demonstrated that the size of the band has a significant role in the algorithm's performance. In the 3-stage reduction, for all problem sizes, the best performances are achieved for large values of the band size. This parameter is chosen to be the maximum feasible so that the band matrix, involved in the second stage, fits into the GPU memory. This approach can be generalized to any two memory levels, e.g. distributed memory of the cluster and the local (main) memory of one computing node, by choosing the largest possible band size so that the band matrix fits into the smaller memory, thus decreasing the number of memory transfers between the two memory levels with different bandwidths and latencies.

The multi-stage out-of-core eigensolver, in particular the 3-stage variant, although exhibits more flops that the one-stage and Krylov subspace approaches, outperforms both of them if all or a large subset of eigenvalues are required. Furthermore, the out-of-core GPU multi-stage approach significantly outperforms the multi-core variants for large enough eigenproblems.

7.1.2 OOC Krylov subspace-based method

The Krylov-subspace based methods are traditionally applied to the solution of sparse eigenproblems and when only a small subset of eigenvalues and eigenvectors is required because of their fast convergence and low computational cost, $\mathcal{O}(n^2)$ flops. Moreover, the Krylov methods are based on the memory-bound matrix-vector operations, rather that on the compute-bounded matrix-matrix operations. Therefore, they are not considered as the methods of choice for the solution of large dense eigenproblems because the operational intensity of matrix-vector operations is $\mathcal{O}(1)$ and can not be efficiently implemented on GPU when the matrix is too large to entirely fit into the GPU memory.

Nevertheless, in this thesis we demonstrated that the Krylov-subspace base eigensolvers can be very competitive to direct eigensolvers for molecular dynamics problems, even for very large dense matrices. Note that the molecular dynamic problems are generalized eigenvalue problem which require a reduction to standard form. Opposite to memory-bound matrix-vector operations, these operations are compute-bound and perfectly suitable for an out-of-core execution on the GPUs. With the explicit construction of matrix C on the GPU, for very large matrices, the execution time of the Krylov-subspace methods significantly decreases. Furthermore, if a small subset of eigenpairs is required, the convergence is fast (i.e. number of iterations) and the standard multi-threaded matrix-vector multiplication, from BLAS, provides fair speedup.

The experiments show that the execution time of the iterative stage of the Krylov methods, due to matrix-vector multiplication, is negligible compared with the reduction to standard eigenvalue form and back-transformation. Furthermore, the explicit, multi-core construction of C for large matrices is the slowest method, while on the other hand the GPU out-of-core variant proved to be the fastest method.

7.1.3 OOC spectral divide–and–conquer

The spectral divide–and–conquer QDWH-based algorithm is a novel highly accurate method to compute the eigenvalues and the corresponding eigenvectors of a dense symmetric matrix. The algorithm is based on the polar decomposition and implements the divide–and–conquer strategy by dividing the problem into smaller subproblems. The divide–and–conquer approach can be very useful when dealing with a large problem since it can be reduced to subproblems that can be efficiently solved in-core by applying state-of-the-art solution from different computational libraries. The spectral D&C algorithm exhibits a significantly higher computational cost than other eigensolvers. However, it attains a very high GFLOPS rate since all operations can be cast in the terms of level-3 BLAS operations that are suitable for the out-of-core execution on the GPUs.

In our approach we modified the recent spectral D&C algorithm by choosing the splitting point for spectral decomposition to be slightly larger than the largest sought-after eigenvalue giving that the subspace iteration part is performed as in-core algorithm with problem dimension close to the number of the required eigenvalues. This turns the polar decomposition as the only computationally-intensive part that has to be re-designed to efficiently exploit a GPU, even if the problem exceeds the GPU memory and the time for subspace iteration becomes negligible compared to the QDWH part.

The QDWH algorithm is an iterative algorithm with three computationally-intensive parts: QR factorization of $2n \times n$ matrix, explicit construction of Q, and updating for the next iteration. Furthermore, the QR factorization operates on a tall matrix with all entries in its lower triangle equal to zero. The sparsity of the matrix can be exploited in order to decrease both computational cost and number of data transferred. Thus, we designed an out-of-core, left-looking, slab-oriented QR factorization that requires less data transfers than the right-looking variant. The computational cost is decreased by applying a two-level blocked strategy. In the first level of blocking, the block size is chosen such that one one slab can fit into the GPU memory. This is a coarse-grained blocking that optimizes the number of data transfers between the main memory and the GPU memory. The second level of blocking is used to eliminate nonzero entires in the lower triangle the matrix and thus optimize the QR factorization of small slabs on the CPU. Instead of working on tall slabs with zeros in the lower positions, the blocking allows us to operate only on the non-zero entries, thus decreasing both the computational cost and the amount of data transferred to the GPU. By exploiting the sparsity of the matrix, our out-of-core QR factorization features lower computational cost, and achieves a higher GFLOPS rate and significant speedup over the most recent out-of-core QR variant from the MAGMA computational library.

In general, the spectral D&C algorithm, though exploits much larger computational costs that the other two methods, exhibits higher GFLOPS rate. However, because of its high cost, the execution time is not comparable with other two methods. However, this method is perfectly scalable to the problem dimension and with some future research that may lead towards the decrease in the algorithm complexity, can be competitive for the solution of dense symmetric eigenproblems when a small subset of eigenpairs is required.

7.2 Future research

Out-of-core GPU computing is a relatively novel research line in the domain of high-performance computing. This area is still largely unexploited and there are many open questions that still wait to be answered. In this thesis we tackled one small segment in this research line, dealing with the efficient solution of large dense symmetric eigenproblems. The following list details some open questions that burst out from this thesis:

- Applying out-of-core techniques and routines to the solution of the generalized non-symmetric eigenvalue problems. Specifically, the developed routines, such as QR factorization, matrix-matrix multiplication, and the update of the trailing matrices can be efficiently applied to the solution of large dense eigenvalue problems. Furthermore, the blocked algorithms that improve the utilization of GPUs and decrease the number of transfers can be used in more general cases.
- Extend the research to problems that are out-of-core from the main memory point of view. Since the main memory is also a limiting factor, one of the possible improvement would be to examine if it is possible to overcome and hide the latency of the hard disk for problems that do not fit into the main memory of the system. Together with the existing GPU out-of-core execution, would it be desirable to compute at the rate close to those of the GPU and store data on disc. This approach,

if possible, would completely hide the hard disc latency and enable the efficient solution for any problem size.

- Overlapping data transfers and computation on the GPU. This open question is closely related with the previous one. In this thesis we did not implement any explicit overlapping between data transfers and computing on the GPU. Some routines such as matrix-matrix multiplications, the QR factorization and Cholesky can greatly benefit from explicit overlapping by introducing CUDA streams and asynchronous kernel calls.
- Dynamic scheduling can be efficiently applied to in the bulge chasing phase in the multi-stage reduction. The bulge chasing phase is very expensive, especially if eigenvectors are required. Future research may include fine-grain bulge-chasing, describing it as direct acyclic graph (DAG) that can be scheduled between CPUs and a GPU or between multiple GPUs.
- Novel eigensolvers based on combinations of the iterative and direct eigensolvers. This thesis showed that the multi-stage approach, though achieves a high GFLOPS rate in the first stages, in the last stage, i.e. the reduction from the band to the tridiagonal form, it exhibits very poor performance and becomes the most time consuming part. An open research line is how accelerate this stage by applying the Krylov subspace-based approach to compute the eigenvalues of a band matrix.
- An open research line is to exploit multi-GPU out-of-core solutions. This approach will boost the performance by combining the power of the multiple GPUs on extremely large problems on systems equipped with more GPUs. An open question is how to efficiently synchronize load balancing (by blocks) on multiple GPUs.

Appendix A

A List of the BLAS Routines

The Basic Linear Algebra Subprograms (BLAS) are routines that perform the most common linear algebra problems such as the scalar product of two vectors, the solution of triangular system, or the matrix-matrix multiplication. The BLAS subroutines are commonly used as basic building blocks for the development of complex dense linear algebra routines as well as in a wide variety of applications in science and engineering. Since the beginning of the development of the BLAS library in early 1970s, it has become *de facto* a standard for the basic, low-level linear algebra libraries and routines. Numerous highly-tuned, hardware-specific implementation of the BLAS library were developed on a top of the original BLAS. All these implementations share the common application programming interface (API) and the specification defined since the original BLAS definition [122].

The BLAS routines are divided into three levels depending on whether they operate on vectors, matrices or both. The first level, referred to as Level-1 BLAS or BLAS-1, encompasses the routines that implement basic operations on vectors such as copying, scaling, vector scalar product, or computing the vector norms. In these routines, the number of operations and the amount of data increase linearly with the problem size and attain low operational intensity. Therefore, because their performance is dictated by the memory bandwidth, these routines are referred as the memory-bounded routines.

The routines from the second level, known as Level-2 BLAS or BLAS-2, are those that exhibit matrix-vector operations. These routines perform the quadratic order of operations on the quadratic amount data. As in case of BLAS-1 routines, the operational intensity is low and is of order of $\mathcal{O}(1)$. However, an efficient implementation can reduce the number of memory accesses by improving the reuse of data stored in the memory, e.g., registers or cache. The list of the BLAS-2 routines most commonly used in this dissertation, is given in Table A.1.

BLAS-2			
Routine	Operation	Comment	flops
xGEMV	$y := \alpha\,\mathrm{op}(A)\,x + \beta\,y$	$\mathrm{op}(A) = A, A^T$, A is $m \times n$	$2mn$
xSYMV	$y := \alpha\,A\,x + \beta\,y$	$A = A^T$	$2n^2$
xTRMV	$x := \mathrm{op}(A)\,x$	$\mathrm{op}(A) = A, A^T$, A upper/lower triangular	n^2
xTRSV	$\mathrm{op}(A)\,x = b$	$\mathrm{op}(A) = A, A^T$, A upper/lower triangular	n^2
xSYR2	$A := \alpha\,x\,y^T + \alpha\,y\,x^T + A$	$A = A^T$	$2n^2 + n$

Table A.1: Description and the number of the floating-point operation of the BLAS-2 routines used in the dissertation.

The main drawback of the BLAS-1 and the BLAS-2 routines is that their operational intensity, i.e. the ratio between the number of floating-point operations and the number of memory accesses, is low. As a result, their performance is limited by the speed at which the data can be provided to the processing units. Therefore, the libraries build on top of the BLAS-1 and BLAS-2 routines cannot attain high performance on modern systems with a hierarchical organization of the system memory. In late 1980s the BLAS specification proposed a set of new operations that exploit cubic number of floating-point operations on a quadratic amount of data. These new routines are called Level-3 BLAS or BLAS-3 routines and are oriented on matrix-matrix operations. With the unbalance between the number of operations performed and memory accesses, these routines better exploit the data locality in the architectures with the multiple memory levels. In practice, this enable the development of algorithms-by-blocks to efficiently hide the memory latency and thus exhibits much higher performance, usually very close to the peak performance delivered by the processors. The BLAS-3 routines that are used in the dissertation are listed in Table A.2.

BLAS-3			
Routine	Operation	Comment	flops
xGEMM	$C := \alpha\, \mathrm{op}(A)\, \mathrm{op}(B) + \beta\, C$	$\mathrm{op}(X) = X, X^T$, C is $m \times n$	$2mkn$
xSYMM	$C := \alpha\, A\, B + \beta\, C$ $C := \alpha\, B\, A + \beta\, C$	C is $m \times n$, $A = A^T$	$2nm^2$ $2mn^2$
xTRMM	$C := \mathrm{op}(A)\, C$ $C := C\, \mathrm{op}(A)$	C is $m \times n$, $\mathrm{op}(A) = A, A^T$	nm^2 mn^2
xTRSM	$\mathrm{op}(A)\, X = \alpha\, B$ $X\, \mathrm{op}(A) = \alpha\, B$	X, B is $m \times n$, $\mathrm{op}(A) = A, A^T$	nm^2 mn^2
xSYR2K	$C := \alpha\, A\, B^T + \alpha\, B\, A^T + \beta\, C$ $C := \alpha\, A^T\, B + \alpha\, B^T\, A + \beta\, C$	A, B in $n \times k$, $C = C^T$ A, B in $k \times n$, $C = C^T$	$2n^2 + k$

Table A.2: Description and the number of the floating-point operation of the BLAS-3 routines used in the dissertation.

Bibliography

[1] Larsen, E., McAllister, D., "Fast Matrix Multiplies Using Graphics Hardware", in Supercomputing, ACM/IEEE 2001 Conference. ACM, Nov 2001, pp. 43–43.

[2] Fatahalian, K., Sugerman, J., Hanrahan, P., "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication", in Graphics Hardware, McCool, M. D., Akenine-Möller, T., (ur.). Eurographics Association, 2004, pp. 133–137.

[3] Galoppo, N., Govindaraju, N. K., Henson, M., Manocha, D., "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware", in Proceedings of the 2005 ACM/IEEE conference on Supercomputing – SC2005, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 3.

[4] Volkov, V., Demmel, J. W., "Benchmarking GPUs to tune dense linear algebra", in SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[5] Barrachina, S., Castillo, M., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., "Evaluation and Tuning of the Level 3 CUBLAS for Graphics Processors", in Proceedings of the 10th IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC 2008, 2008, pp. 1–8.

[6] Quintana-Ortí, G., Igual, F. D., Quintana-Ortí, E. S., van de Geijn, R., "Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators", in ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'09), 2009, to appear.

[7] Tomov, S., Nath, R., Ltaief, H., Dongarra, J., "Dense linear algebra solvers for multicore with GPU accelerators", in IPDPS Workshops. IEEE, 2010, pp. 1–8.

[8] "MAGMA Matrix Algebra on GPU and Multicore Architectures", http://icl.cs.utk.edu/magma/, 2014.

[9] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S., "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects ", Vol. 180, No. 1, 2009.

[10] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, D., McKenney, A., Sorensen, D., LAPACK Users' Guide, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[11] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R. C., ScaLAPACK Users' Guide. SIAM, 1997.

[12] van de Geijn, R. A., Using PLAPACK: Parallel Linear Algebra Package. The MIT Press, 1997.

[13] "Intel Math Kernel library", http://software.intel.com/en-us/intel-mkl, 2014.

[14] "AMD Core Math Library (ACML)", http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/, 2014.

[15] Tomov, S., Nath, R., Dongarra, J., "Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing", Parallel Computing, Vol. 36, No. 12, 2010, pp. 645–654.

[16] Bientinesi, P., Igual, F. D., Kressner, D., Quintana-Ortí, E. S., "Reduction to Condensed Forms for Symmetric Eigenvalue Problems on Multi-core Architectures", in PPAM (1), ser. Lecture Notes in Computer Science, Vol. 6067. Springer, 2009, pp. 387–395.

[17] Haidar, A., Ltaief, H., Dongarra, J., "Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels", in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 8:1–8:11.

[18] Demmel, J., Marques, O., Parlett, B. N., Vömel, C., "Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers", SIAM J. Scientific Computing, Vol. 30, No. 3, 2008, pp. 1508–1526.

[19] Parlett, B. N., The Symmetric Eigenvalue Problem. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[20] Golub, G. H., Loan, C. F. V., Matrix Computations, 3rd ed. Baltimore: The Johns Hopkins University Press, 1996.

[21] Ipsen, I. C., "Computing an eigenvector with inverse iteration", SIAM review, Vol. 39, No. 2, 1997, pp. 254–291.

[22] Dhillon, I. S., "Current inverse iteration software can fail", BIT Numerical Mathematics, Vol. 38, No. 4, 1998, pp. 685–704.

[23] Cuppen, J. J. M., "A divide and conquer method for the symmetric tridiagonal eigenproblem", Numerische Mathematik, Vol. 36, No. 2, 1980, pp. 177–195.

[24] Gu, M., Eisenstat, S. C., "A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem", SIAM Journal on Matrix Analysis and Applications, Vol. 16, No. 1, 1995, pp. 172–191.

[25] Dhillon, S., Parlett, N., Vomel, "The Design and Implementation of the MRRR Algorithm", ACM Trans. Math. Soft., Vol. 32, No. 4, 2006, pp. 533–560.

[26] Dhillon, I. S., Parlett, B. N., “Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices”, Linear Algebra and its Applications, Vol. 387, 2004, pp. 1–28.

[27] Parlett, B. N., Dhillon, I. S., “Relatively robust representations of symmetric tridiagonals”, Linear Algebra and its Applications, Vol. 309, No. 1–3, 2000, pp. 121–151.

[28] Igual, F. D., Chan, E., Quintana-Ortí, E. S., Quintana-Ortí, G., van de Geijn, R. A., Zee, F. G. V., “The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations”, J. Parallel Distrib. Comput., Vol. 72, No. 9, 2012, pp. 1134–1143.

[29] Gunnels, J. A., Gustavson, F. G., Henry, G. M., van de Geijn, R. A., “FLAME: Formal Linear Algebra Methods Environment”, ACM Transactions on Mathematical Software, Vol. 27, No. 4, December 2001, pp. 422–455.

[30] Solcà, R., Haidar, A., Tomov, S., Schulthess, T. C., Dongarra, J., “A Novel Hybrid CPU-GPU Generalized Eigensolver for Electronic Structure Calculations Based on Fine Grained Memory Aware Tasks”, in SC Companion. IEEE Computer Society, 2012, pp. 1338–1339.

[31] Auckenthaler, T., Blum, V., Bungartz, H.-J., Huckle, T., Johanni, R., Krämer, L., Lang, B., Lederer, H., Willems, P. R., “Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations”, Parallel Computing, Vol. 37, No. 12, 2011, pp. 783–794.

[32] ELPA, “Eigenvalue SoLvers for Petaflop-Applications (ELPA)”, http://elpa.rzg.mpg.de/, 2014.

[33] Toledo, S., “A survey of out-of-core algorithms in numerical linear algebra”, in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1999.

[34] D’Azevedo, E. F., Dongarra, J., “The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines”, Concurrency - Practice and Experience, Vol. 12, No. 15, 2000, pp. 1481–1493.

[35] D’Azevedo, E., Huang, A., Wong, K., Wu, W., “Out-of-core algorithms for dense matrix factorization on GPGPU”.

[36] Davidović, D., Quintana-Ortí, E. S., “Applying OOC techniques in the reduction to condensed form for very large symmetric eigenproblems on GPUs”, in 20th Euro. Conf. PDP 2012, 2012, pp. 442–449.

[37] Aliaga, J., Bientinesi, P., Davidović, D., Di Napoli, E., Igual, F., Quintana-Ortí, E. S., “Solving dense generalized eigenproblems on multi-threaded architectures”, Applied Mathematics and Computation, Vol. 218, No. 22, 2012, pp. 11 279–11 289.

[38] Strang, G., Introduction to Linear Algebra, 4th ed. Wellesley, MA: Wellesley-Cambridge Press, 2009.

[39] Saad, Y., Numerical Methods for Large Eigenvalue Problems. SIAM Press, 2011, revised edition.

[40] Horn, R. A., Johnson, C. R., Matrix Analysis. Cambridge University Press, 1990.

[41] Veselić, K., “A Jacobi eigenreduction algorithm for definite matrix pairs”, Numer. Math., Vol. 64, No. 1, 1993, pp. 241–269.

[42] Wilkinson, J. H., The Algebraic Eigenvalue Problem. Oxford, England: Oxford University Press, 1965.

[43] Gustafson, J., “Reconstruction of the Atanasoff-Berry computer”, in The First Computers: History and Architectures. MIT Press, 2000, pp. 91–106.

[44] Moore, G. E., “Cramming more components onto integrated circuits”, Electronics, Vol. 38, No. 8, April 1965.

[45] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R. S., Yelick, K., “ Exascale computing study: Technology challenges in achieving exascale systems”, Tech. Rep., 2008.

[46] Hennessy, J. L., Patterson, D. A., Computer Architecture: A Quantitative Approach, 5th ed. Elsevier, 2012.

[47] Flynn, M. J., “Very High-Speed Computing Systems”, Proceedings of the IEEE, Vol. 54, No. 12, December 1966, pp. 1901–1909.

[48] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., Yelick, K., “A case for intelligent RAM”, IEEE Micro, Vol. 17, No. 2, 1997, pp. 34–44.

[49] Top500, “Top500 supercomputer sites”, Nov. 2013.

[50] “The Green 500”, http://www.green500.org/, Nov. 2013.

[51] NVIDIA, “CUDA Toolkit Documentation, CUDA C Best Practices Guide”, http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, NVIDIA, 2013.

[52] NVIDIA, “NVIDIA CUDA Parallel Computing Platform”, http://www.nvidia.com/object/cuda_home_new.html, NVIDIA, 2013.

[53] NVIDIA, “CUDA Toolkit Documentation, CUDA C Programming Guide v5.5”, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, NVIDIA, 2013.

[54] Kressner, D., “Numerical Methods for General and Structured Eigenvalue Problems”, Vol. 46, 2005, pp. 258.

[55] Kublanovskaya, V. N., “On some algorithms for the solution of the complete eigenvalue problem”, USSR Computational Mathematics and Mathematical Physics, Vol. 1, No. 3, 1962, pp. 637–657.

[56] Francis, J., “The QR Transformation, I”, The Computer Journal, Vol. 4, No. 3, 1961, pp. 265–271.

[57] Francis, J., “The QR Transformation, II”, The Computer Journal, Vol. 4, No. 4, 1962, pp. 332–345.

[58] Jacobi, C., “Über ein leichtes Verfahren, die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen”, J. Reine Angew. Math., Vol. 30, 1846, pp. 51–94.

[59] de Rijk, P. P. M., “A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer”, SIAM journal on scientific and statistical computing, Vol. 10, No. 2, 1989, pp. 359–371.

[60] Arbenz, P., Slapničar, I., “An analysis of parallel implementations of the block-Jacobi algorithm for computing the SVD”, in Proceedings of the 17th International Conference on Information Technology Interfaces ITI, Vol. 95, 1995.

[61] Demmel, J., Veselić, K., “Jacobi’s method is more accurate than QR”, SIAM Journal on Matrix Analysis and Applications, Vol. 13, No. 4, 1992, pp. 1204–1245.

[62] Singer, S., Singer, S., Novaković, V., Davidović, D., Bokulić, K., Uščumlić, A., “Three-level parallel J-Jacobi algorithms for Hermitian matrices”, Applied Mathematics and Computation, Vol. 128, No. 9, Jan. 2012, pp. 5704–5725.

[63] Singer, S., Hari, V., Bokulić, K., Davidović, D., Jurešić, M., Uščumlić, A., “Advances in Speedup of the Indefinite One-Sided Block Jacobi Method”, in Numerical Analysis and Applied Mathematics: International Conference on Numerical Analysis and Applied Mathematics, Vol. 936, 2007, pp. 519–522.

[64] Novaković, V., Singer, S., “A GPU-based hyperbolic SVD algorithm”, BIT Numerical Mathematics, Vol. 51, No. 4, 2010, pp. 1009–1030.

[65] Sleijpen, G. L. G., der Vorst, H. A. V., “A Jacobi-Davidson iteration method for linear eigenvalue problems”, SIAM Journal on Matrix Analysis and Applications, Vol. 17, 1996, pp. 401–425.

[66] Ltaief, H., Tomov, S., Nath, R., Du, P., Dongarra, J., “A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators”, in VECPAR, ser. Lecture Notes in Computer Science, Palma, J. M. L. M., Daydé, M. J., Marques, O., Lopes, J. C., (ur.), Vol. 6449. Springer, 2010, pp. 93–101.

[67] Volkov, V., Demmel, J., “LU, QR and Cholesky factorizations using vector capabilities of GPUs”, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May 2008.

[68] Zee, F. G. V., Chan, E., van de Geijn, R. A., Quintana-Ortí, E. S., Quintana-Ortí, G., “The libflame Library for Dense Matrix Computations”, Computing in Science and Engineering, Vol. 11, No. 6, 2009, pp. 56–63.

[69] Bientinesi, P., Dhillon, I. S., van de Geijn, R., “A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations”, SIAM J. Sci. Comput., Vol. 27, No. 1, 2005, pp. 43–66.

[70] Nath, R., Tomov, S., Dongarra, J., “An Improved Magma Gemm For Fermi Graphics Processing Units”, IJHPCA, Vol. 24, No. 4, 2010, pp. 511–515.

[71] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D., “LAPACK: A Portable Linear Algebra Library for High-Performance Computers”, in Proceedings Supercomputing ’90. Los Alamitos, California: IEEE Computer Society Press, 1990, pp. 2–11.

[72] Bischof, C. H., Lang, B., Sun, X., “Algorithm 807: The SBR Toolbox—software for successive band reduction”, Vol. 26, No. 4, 2000, pp. 602–616.

[73] Elmroth, E., Gustavson, F., Jonsson, I., Kagstrom, B., “Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software”, SIAM Review, Vol. 46, No. 1, 2004, pp. 3–45.

[74] Dongarra, J. J., Hammarling, S. J., Sorensen, D. C., “Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations”, Journal of Computational and Applied Mathematics, Vol. 27, 1989.

[75] Bischof, C., Van Loan, C., “The WY Representation for Products of Householder Matrices”, SIAM J. Sci. Stat. Comput., Vol. 8, No. 1, January 1987, pp. s2–s13.

[76] Goto, K., van De Geijn, R. A., “High-performance implementation of the level-3 BLAS”, ACM Trans. Math. Softw., Vol. 35, No. 1, 2008.

[77] Xianyi, Z., Qian, W., Chothia, Z., “OpenBLAS”, http://www.openblas.net/, 2013.

[78] Luszczek, P., Ltaief, H., Dongarra, J., “Two-Stage Tridiagonal Reduction for Dense Symmetric Matrices Using Tile Algorithms on Multicore Architectures”, in IPDPS. IEEE, 2011, pp. 944–955.

[79] Bientinesi, P., Igual, F. D., Kressner, D., Petschow, M., Quintana-Ortí, E. S., “Condensed forms for the symmetric eigenvalue problem on multi-threaded architectures”, Concurrency and Computation: Practice and Experience, Vol. 23, No. 7, 2011, pp. 694–707.

[80] Bischof, C. H., Lang, B., Sun, X., “A Framework for Symmetric Band Reduction”, ACM Trans. Math. Soft., Vol. 26, No. 4, 2000, pp. 581–601.

[81] Schreiber, R., Van Loan, C., “A Storage-Efficient WY Representation for Products of Householder Transformations”, SIAM J. Sci. Stat. Comput., Vol. 10, No. 1, January 1989, pp. 53–57.

[82] Rutishauser, H., “On Jacobi rotation patterns”, in Proceedings of Symposia in Applied Mathematics, Vol. 15, 1963.

[83] Schwarz, H. R., “Tridiagonalization of a symetric band matrix”, Numerische Mathematik, Vol. 14, No. 2, 1968, pp. 231–241.

[84] Murata, K., Horikoshi, K., “A new method for the tridiagonalization of the symmetric band matrix”, Information Processing in Japan, Vol. 15, 1975, pp. 108–112.

[85] Lang, B., “A parallel algorithm for reducing symmetric banded matrices to tridiagonal form”, SIAM Journal on Scientific Computing, Vol. 14, No. 6, 1993, pp. 1320–1338.

[86] Ballard, G., Demmel, J., Knight, N., “Avoiding Communication in Successive Band Reduction”, Tech. Rep., 2013.

[87] Arnoldi, W. E., “The principle of minimized iterations in the solution of the matrix eigenvalue problem”, Quart. Appl. Math, Vol. 9, No. 1, 1951, pp. 17–29.

[88] Lanczos, C., “An iteration method for the solution of the eigenvalue problem of linear differential and integral operators”, J. Res. Nat. Bur. Stand., Vol. 45, 1950, pp. 255–82.

[89] Saad, Y., “On the Lanczos Method for Solving Symmetric Linear Systems with Several Right Hand Sides”, Mathematics of Computation, Vol. 4, No. 178, 1987, pp. 651–662.

[90] Sorensen, D. C., Implicitly restarted Arnoldi/Lanczos methods for large scale eigenvalue calculations. Springer Netherlands, 1997.

[91] Saad, Y., “Chebyshev Acceleration Techniques for Solving Nonsymmetric Eigenvalue Problems”, Mathematics of Computation, Vol. 42, No. 166, April 1984, pp. 567–588.

[92] Sorensen, D. C., “Implicit application of polynomial filters in a k-step Arnoldi method”, SIAM Journal on Matrix Analysis and Applications, Vol. 13, No. 1, 1992, pp. SIAM Journal on Matrix Analysis and Applications.

[93] Lehoucq, R. B., Sorensen, D. C., “Deflation Techniques for an Implicitly Restarted Arnoldi Iteration”, SIAM Journal on Matrix Analysis and Applications, Vol. 17, No. 4, 1996, pp. 789–821.

[94] Lehoucq, R., Maschhoff, K., Sorensen, D., Yang, C., “ARPACK home page”, http://www.caam.rice.edu/software/ARPACK/, 2014.

[95] Lehoucq, R. B., Sorensen, D. C., Yang, C., ARPACK users’ guide - solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods, ser. Software, environments, tools. SIAM, 1998, Vol. 6067.

[96] Maschhoff, K. J., Sorensen, D. C., “P_ARPACK: An Efficient Portable Large Scale Eigenvalue Package for Distributed Memory Parallel Architectures”, in PARA, ser. Lecture Notes in Computer Science, Wasniewski, J., Dongarra, J., Madsen, K., Olesen, D., (ur.), Vol. 1184. Springer, 1996, pp. 478–486.

[97] Dongarra, J. J., Whaley, R. C., “LAPACK Working Note 94 A User’s Guide to the BLACS v1”, Tech. Rep., 1997.

[98] MPI, “he Message Passing Interface (MPI) standard”, http://www.mpi-forum.org/, 2013.

[99] Dongarra, J., Eijkhout, V., Kalhan, A., “Reverse communication interface for linear algebra templates for iterative methods”, Tech. Rep., May 1995.

[100] Baker, G., Gunnels, J., Morrow, G., Riviere, B., van de Geijn, R., “PLAPACK: High Performance through High Level Abstraction”, in Proceedings of ICCP98, 1998.

[101] Choi, J., Dongarra, J. J., Pozo, R., Walker, D. W., “Scalapack: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers”, in Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation. IEEE Comput. Soc. Press, 1992, pp. 120–127.

[102] Beavers, A., Denman, E., “A computational method for eigenvalues and eigenvectors of a matrix with real eigenvalues”, Numerische Mathematik, Vol. 21, No. 5, 1973, pp. 389–396.

[103] Lin, C.-C., Zmijewski, E., “A Parallel Algorithm for Computing the Eigenvalues of an Unsymmetric Matrix on an SIMD Mesh of Processors”, University of California, Tech. Rep., 1991.

[104] Bai, Z., Demmel, J., Gu, M., “An inverse free parallel spectral divide and conquer algorithm for nonsymmetric eigenproblems”, Numerische Mathematik, Vol. 76, No. 3, 1997, pp. 279–308.

[105] Bischof, C., Huss-Lederman, S., Sun, X., Tsao, A., “The PRISM project: Infrastructure and algorithms for parallel eigensolvers”. IEEE, Oct. 1993, pp. 123–131.

[106] Zhang, Z., Zha, H., Ying, W., “Fast parallelizable methods for computing invariant subspaces of Hermitian matrices”, Journal of Computational Mathematics, Vol. 25, No. 5, 2007, pp. 583–594.

[107] Nakatsukasa, Y., Higham, N. J., “Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD”, SIAM Journal on Scientific Computing, Vol. 35, No. 3, 2013, pp. A1325–A1349.

[108] Nakatsukasa, Y., Bai, Z., Gygi, F., “Optimizing Halley’s iteration for computing the matrix polar decomposition”, SIAM Journal on Matrix Analysis and Applications, Vol. 31, No. 5, 2010, pp. 2700–2720.

[109] Higham, N. J., Functions of Matrices: Theory and Computation. SIAM, 2008.

[110] Gander, W., “On Halley’s iteration method”, The American Mathematical Monthly, Vol. 92, No. 2, 1985, pp. 131–134.

[111] Gander, W., “Algorithms for the polar decomposition”, SIAM journal on scientific and statistical computing, Vol. 11, No. 6, 1990, pp. 1102–1115.

[112] Ballard, G., Demmel, J., Holtz, O., Schwartz, O., “Minimizing Communication in Numerical Linear Algebra”, SIAM J. Matrix Analysis Applications, Vol. 32, No. 3, 2011, pp. 866–901.

[113] NVIDIA, “cuBLAS”, https://developer.nvidia.com/cublas, 2014.

[114] Demmel, J., Grigori, L., Hoemmen, M., Langou, J., “Communication-optimal Parallel and Sequential QR and LU Factorizations”, SIAM J. Scientific Computing, Vol. 34, No. 1, 2012.

[115] Aliaga, J. I., Davidović, D., Quintana-Ortí, E. S., “Out-of-Core Solution of Eigenproblems for Macromolecular Simulations”, in Parallel Processing and Applied Mathematics, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 490–499.

[116] Cui, Q., Bahar, I., Normal mode analysis: theory and applications to biological and chemical systems, ser. Mathematical and Computational Biology Series. Chapman & Hall/CRC press, 2010.

[117] Bahar, I., Rader, A. J., “Coarse-grained normal mode analysis in structural biology”, Current opinion in structural biology, Vol. 15, No. 5, 2005, pp. 586–592.

[118] Chacon, P., Tama, F., Wriggers, W., “ Mega-dalton biomolecular motion captured from electron microscopy reconstructions”, Journal of Molecular Biology, Vol. 362, No. 2, 2003, pp. 485–492.

[119] Delarue, M., Dumas, P., “On the use of low-frequency normal modes to enforce collective movements in refining macromolecular structural models”, Vol. 101, No. 18, 2004, pp. 6957–6962.

[120] May, A., Zacharias, M., “ Energy minimization in low-frequency normal modes to efficiently allow for global flexibility during systematic protein-protein docking”, Proteins, Vol. 70, No. 8, 2008, pp. 794–809.

[121] Lopez-Blanco, J., Garzon, J. I., Chacon, P., “iMOD: multipurpose normal mode analysis in internal coordinates”, Bioinformatics, 2011, to appear.

[122] “BLAS (Basic Linear Algebra Subprograms)”, http://netlib.org/blas/, 2014.

Biography

Davor Davidović was born on 14th of June 1984 in Zagreb. He graduated in 2008 at the Faculty of Science, Department of Mathematics, University of Zagreb. From May 2008 till May 2009 he was technical associate at the Ruđer Bošković Institute at the Center for Informatics and Computing working on two EU FP7 projects SEEGRID-SCI and EGEE-III. Since May 2009 he is research assistant on the project "Methods of scientific visualization" funded by the Ministry of Science, Education and Sports of the Republic of Croatia. He was actively involved in more than 5 international research projects that focus on distributed computing, high-performance computing, and the development of advanced scalable numerical linear algebra algorithms. While working at the Institute he has established strong collaborations with international research institutions and spent more than 8 months abroad as a guest researcher or at the professional training courses. Between them are several months-long stays as a guest researcher at the University Jaume I, Spain and ETH Zurich, Switzerland.

He has published more than 13 original peer-reviewed research and technical papers in the international journals and conference proceedings, of which 2 have been published in the journals indexed in CC.

List of Publications

Journal papers

1. Fosin, J., Davidović, D., Carić, T., *"A GPU implementation of local search operations for symmetric traveling salesman problem"*, Promet – Traffic & Transportation. Vol. 25, No. 3, 2013, pp. 225-234.
2. Aliaga, José I., Bientinesi, P., Davidović, D., Di Napoli, E., Igual Peña, Francisco D., Quintana-Ortí, E. S., *"Solving Dense Generalized Eigenproblems on Multi-threaded Architectures"*, Applied mathematics and computation. Vol. 218, No. 22, 2012, pp. 11279-11289.
3. Singer, S., Singer, S., Novaković, V., Davidović, D., Bokulić, K., Uščumlić, A., *"Thee-Level Parallel J-Jacobi Algorithms for Hermitian Matrices"*, Applied Mathematics and Computations. Vol. 218, No. 9, 2012, pp. 5704-5725.
4. Davidović, D., Skala, K., Belušić, D., Telišman-Prtenjak, M., *"Grid implementation of the Weather Research and Forecasting model"*, Earth Science Informatics. Vol. 3, No. 4, 2010, pp. 199-208.

Conference proceedings

1. Aliaga, José I., Davidović, D., Quintana-Ortí, E. S., *"Out-of-core solution of eigenproblems for macromolecular simulations on GPUs"*, Parallel Processing and Ap-

plied Mathematics, Lecture Notes in Computer Science. Vol. 8384, 2014, pp. 490-499.

2. Afgan, E., Skala, K., Davidović, D., Lipić, T., Sović, I., *"CloudMan as a tool execution framework for the cloud"*, Proceedings of the 36th International Convention MIPRO, 2013, pp. 437-441.
3. Davidović , D., Lipić, T., Skala, K., *"AdriaScience Gateway: Application Specific Gateway for Advanced Meteorological Predictions on Croatian Distributed Computing Infrastructures"*, Proceedings of the 36th International Convention MIPRO, 2013, pp. 237-241.
4. Davidović, D., Quintana-Ortí, E. S., *"Applying OOC Techniques in the Reduction to Condensed Form for Very Large Symmetric Eigenproblems on GPUs"* Euromicro Conference on Parallel, Distributed and Network-based Processing, IEEE Computer Society CPS, 2012, pp. 442-449.
5. Davidović, D., Skala, K., *"Implementation of the WRF-ARW prognostic model on the Grid"*, Proceedings of the 33rd International Convention MIPRO, 2010, pp. 253-258.
6. Singer, S., Singer, S., Hari, V., Bokulić, K., Davidović, D., Jurešić, M., Uščumlić, A., *"Advances in Speedup of the Indefinite One-Sided Block Jacobi Method"*, Numerical Analysis and Applied Mathematics: International Conference of Numerical Analysis and Applied Mathematics, Vol. 936, 2007, pp. 519-522.

Other publications and presentations

1. "Grid implementation and application of the WRF-ARW prognostic model." Oral presentation at the 5th EGEE User Forum (April 12-16, 2010), Uppsala University, Sweden
2. "Feature – Forecasting weather on the grid", On-line publication in ISGTW international science grid this week (2010).
3. "Grid implementation of the Weather Research and Forecasting model. " Oral presentation at the SEEGRID-SCI User Forum (December 9-10, 2009), Boğaziçi University, Turkey

Životopis

Davor Davidović rođen je 14. lipnja 1984. godine u Zagrebu. Diplomirao je 2008. godine na Matematičkom odsjeku Prirodoslovno-Matematičkog fakulteta Sveučilišta u Zagrebu, smjer računarstvo. Od svibnja 2008. od svibnja 2009. godine radio je kao stručni suradnik na Institut Ruđer Bošković u Centru za informatiku i računarstvo na dva EU FP7 projekta SEEGRID-SCI i EGEE-III. Od svibnja 2009. godine je zaposlen kao znanstvenog novak na projektu "Metode znanstvene vizualizacije" financiranog od strane Ministarstva znanosti, obrazovanja i sporta Republike Hrvatske. Aktivno je sudjelovao na preko 5 međunarodnih projekata iz područja distribuiranog računarstva, računarstva visokih performansi te razvoju naprednih skalabilnih algoritama iz područja numeričke linearne algebre. Tijekom rada na Institutu ostvario je brojne međunarodne suradnje te boravio ukupno preko 8 mjeseci na inozemnim sveučilištima u svrhu stručnog usavršavanja te provođenja znanstvenih istraživanja. Među njima posebno se ističu višemjesečna gostovanja na Sveučilištu Jaume I., Španjolska te na ETH Zurich, Švicarska.

Do sada je objavio preko 13 znanstvenih i stručnih radova u časopisima i zbornicima skupova s međunarodnom recenzijom od kojih su 2 objavljena u časopisima indeksiranima u CC.