Računanje svojstvenih vrijednosti na masovno-paralelnom računalu

Lehpamer, Mihael

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:824674

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mihael Lehpamer

RAČUNANJE SVOJSTVENIH VRIJEDNOSTI NA MASOVNO-PARALELENOM RAČUNALU

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mihael Lehpamer

Matični broj: 45137/16–R

Studij: Informacijsko i programsko inženjerstvo

RAČUNANJE SVOJSTVENIH VRIJEDNOSTI NA MASOVNO-PARALELNOM RAČUNALU

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Ivan Hip

Varaždin, rujan 2022

Mihael Lehpamer

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Problem svojstvenih vrijednosti rješava se pomoću mnogih algoritama, dok je u samom radu naglasak na pronalazak optimalnog rješenja koristeći programski model za paralelno programiranje *CUDA* kojeg pruža proizvođač grafičkih procesora *NVIDIA.* Implementiran je *Jacobijev algoritam* na standardan, odnosno serijski, način koji se izvršava na *CPU* te paralelna verzija koja se izvršava na *GPU*. Paralelna verzija je konfigurirana da koristi 8^2, 16^2, 24^2 ili 32^2 dretve po bloku. Vremena njihovih izvršavanja su zatim uspoređena kako bismo vidjeli nudi li nam paralelno programiranje prednosti u izvršavanju naspram serijske verzije. Kako bismo kontrolirali točnost rješenja, korištena je *LAPACK* biblioteka koja pruža već gotove rutine za izračun svojstvenih vrijednosti.

Ključne riječi: paralelno programiranje, svojstvene vrijednosti, *LAPACK*, *CUDA*, programski jezik *C, Jacobijev algoritam*

Sadržaj

1. Uvod

Računanje svojstvenih vrijednosti je važan problem u numeričkoj matematici jer se pojavljuje u brojnim granama znanosti i inženjerstva. Svojstvene vrijednosti predstavljaju karakteristične frekvencije mehaničkih sustava [1], a posebno su važne u kvantnoj mehanici gdje predstavljaju energijske nivoe koje je moguće izravno usporediti s rezultatima eksperimenata.

Postoje mnogo algoritama za izračunavanje svojstvenih vrijednosti. Neki od tih algoritama se mogu dodatno optimizirati za primjenu u masovno-paralelnim računalima.

Moderne grafičke kartice opremljene grafičkim procesorima (engl. *Graphics Processing Unit - GPU*) zapravo su masovna-paralelna računala. Navedena računala imaju mogućnost pokretanja na stotine pa i tisuće manjih paralelnih procesa i rasterećuju resurse na centralnom procesoru računala (engl. *Central Processing Unit - CPU*). Računala koje sadrže navedeni hardver imaju mogućnost ubrzavanja algoritama kod kojih se neki ili svi koraci mogu izvršavati istovremeno.

U ovome radu bit će implementirano računanje svojstvenih vrijednosti na više načina. Prvo će biti implementirana serijska verzija u programskom jeziku *C*, zatim će biti implementirana verzija uz pomoć rutina koju pruža biblioteka *LAPACK* (engl. *Linear Algebra PACKage*) te vlastita implementacija paralelne verzije uz pomoć programskog modela *CUDA* čija sintaksa je bliska programskom jeziku *C*.

2. Metode i tehnike rada

Okolina u kojoj će se provesti implementacija je operacijski sustav *Linux* ili točnije distribucija *Ubuntu 20.04 LTS*. Način na koji je instaliran *Linux* operativni sustav je pomoću tehnologije *WSL* (engl. *Windows Subsystem for Linux*). Unutar njega je instalirano nekoliko biblioteka koji će biti potrebni za realizaciju rada.

2.1. WSL

Instalacijama *Podsustava Windowsa za Linux* (engl. *Windows Subsystem for Linux*) razvojnim programerima daje mogućnost pokretanja *Linux* operacijskog sustava na već postojećoj platformi *Windows* bez da se treba napustiti navedena platforma. Instalirana platforma dolazi s većinom alata naredbenog retka, programa, aplikacija i drugo. Navedena instalacija nam omogućuje direktan pristup *Linux* sustavu bez potrebe pokretanja virtualnog stroja ili konfiguriranja dvostrukog pokretanja (engl. *dualboot*) što nam uvelike olakšava sam proces pristupanja operacijskom sustavu i daje nam punu kontrolu nad hardverskim i softverskim dijelovima računala. U ovome radu nam je potrebna potpuna kontrola hardvera te njegove pune performanse. U našem slučaju potrebne su nam performanse grafičke kartice (engl. *Graphics Processing Unit - GPU*) [2].

Za instalaciju *WSL*-a potrebno je imati verziju veću od *Windows 10 version 2004* (*Build 19041 and higher*). Ako zadovoljavamo taj uvjet slobodno možemo unijeti sljedeću komandu *u Windows PowerShell* ili naredbeni redak (engl. *Command Prompt - CMD*) s privilegijama administratora: **wsl --install**. Nakon unesene naredbe trebamo ponovno pokrenuti računalo kako bi se instalacija aplicirala. Nakon ponovnog pokretanja računala otvorit će se naredbeni prozor koji će napraviti dovršavanje instalacije te će biti instalirana zadana verzija *Linuxa* tj. *Ubuntu Linux* [2].

Za naše potrebe zadana verzija *Linux*-a će poslužiti, iako naredbom: **wsl --install -d <Distribution Name>** možemo instalirati i ostale dostupne verzije.

Kada smo jednom uspješno pokrenuli *WSL* započet ćemo s instalacijom softvera potrebnog za izvršavanje programa u *NVIDIA CUDA*.

2.2. NVIDIA CUDA

Instalacija tehnologije *NVIDIA CUDA* (engl. *Compute Unified Device Architecture*) omogućava *GPU* akceleraciju unutar *WSL* podsustava. *GPU* akceleracija nam pruža direktan pristup hardveru što je nama u ovome radu izričito potrebno.

Kao što smo za instalaciju *WSL*-a trebali imati preduvjete tako će biti i ovdje. Operacijski sustav na kojem se izvršava *WSL* mora imati instaliranu verziju *Windows version 21H2* ili veću. Također trebamo imati i fizičku komponentu, *GPU* proizvođača *NVIDIA*. Verzija Linuxa trebala bi biti veća od *Ubuntu 20.04* također na *WSL 2* verziji. Uz to, još nam je potrebna konekcija na *Internet* jer trebamo preuzeti nekoliko gigabajta (*GB*) podataka.

Najprije je potrebno preuzeti driver koji će biti instaliran putem *Windowsa* i izvršne datoteke *.exe*. Datoteka koja se treba preuzeti je *NVIDIA CUDA on WSL driver* te nam daje mogućnost instaliranja *NVIDIA CUDA* u sljedećem koraku [3].

Sljedećim komandama instalirat ćemo verziju *CUDA toolkit version 11.6*.

- **sudo apt-key del 7fa2af80** – brišemo stari *GPG* (*GnuPG*) ključ zbog verzije *WSL*-a
- **wget https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86_64/cuda-wsl-ubuntu.pin** - preuzimanje paketa iz repozitorija *CUDA*-e
- **sudo mv cuda-wsl-ubuntu.pin /etc/apt/preferences.d/cuda-repository-pin-600** – premještamo datoteku
- **sudo apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86_64/3bf863cc.pub** - dohvaćamo javni ključ
- **sudo add-apt-repository "deb https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86_64/ /"** – dohvaćamo ažuriranje
- **sudo apt-get update** – radimo ažuriranje
- **sudo apt-get -y install cuda** – instalacija *CUDA*-e

Time smo završili instalaciju te možemo testirati radi li sve kako spada. *CUDA* nam pruža otvoreni repozitorij na *GitHub*-u te možemo ondje preuzeti neki od primjera [3][4].

2.3. LAPACK

LAPACK (engl. *Linear Algebra PACKage*) je softverski paket koji sadrži brojne rutine za rješavanje sustava linearnih jednadžbi te računanje svojstvenih i singularnih vrijednosti [5]. U radu će se koristiti jedna rutina za računanje svojstvenih vrijednosti simetrične matrice.

LAPACK-ove rutine pisane su na način da se što više obavljaju operacije pozivom *BLAS* (engl. *Basic Linear Algebra Subprograms*) potprograma. Dizajniran je na *BLAS3* razini koji sadrži neke od specifikacija jezika *FORTRAN* te one rade različita množenja matrica i daju rješenja za trokutaste sustave s višestrukim desnim stranama.

U ovome radu pozivat će se rutina koja daje rješenje za svojstvene vrijednosti matrice. Priprema za poziv rutina će biti u sklopu programskog jezika *C* gdje će se generirati matrica temeljene na slučajnim brojevima s opcijom *sjemena* kako bi svaka matrica mogla biti ista kod svakog pokretanja sve dok se *sjeme* ne promijeni.

Za početak rada s paketom *LAPACK* potrebno je samo preuzeti *.zip* datoteku s njihove web stranice [5] te uključiti potrebne datoteke u naš program ili možemo isto tako napraviti instalaciju pomoću naredbe **sudo apt install liblapacke-dev.**

Prethodnom naredbom smo napravili instalaciju biblioteke *LAPACK**E*** koji se razlikuje od biblioteke *LAPACK*. Naime *LAPACK* je biblioteka koja je pisana u programskom jeziku *FORTRAN* te da bi se pristupilo toj biblioteci u programskom jeziku *C* napravljena je nova biblioteka. Ta biblioteka nam daje pristup do originalnih rutina pisanih u *FORTRAN*-u kroz programski jezik *C*.

2.4. Kompajler

Pošto radimo u operacijskom sustavu *Linux - Ubuntu* koristit ćemo *GNU*-ov *kompajler* za programski jezik *C*. Verzija u trenutku izrade rada je bila sljedeća: *gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0.* Pozivom naredbe **gcc naziv_datoteke.c** radimo prevođenje programa.

Za prevođenje programa koji su pisani u *CUDA-i* koristimo *kompajler* koji je bio instaliran zajedno s instalacijom *CUDA-e*. Verzija koja je korištena u trenutku rada je sljedeća: *release 10.1, V10.1.243*. Pozivom naredbe **nvcc naziv_datoteke.cu** vrši se prevođenje programa pisanog pomoću programskog modela *CUDA*.

Kako ne bismo morali raditi kompajliranje svakog programa posebno napravljena je skripta pomoću alata za kompajliranje programa naziva *Makefile*. *Makefile* nam daje jednostavno i brzo rješenje za kompajliranje pomoću seta određenih pravila. Pokretanje se izvršava na *Linux*-u pozivom naredbe **make -f Makefile**, gdje je *Makefile* naziv datoteke s pravilima ili jednostavno samo **make** ako je datoteka ispravno nazvana (*Makefile*).

3. GPGPU

GPGPU (engl. General Purpose Graphics Processing Unit) je procesna grafička jedinica koja izvodi nespecijalizirane izračune koji se inače izvršavaju na centralnom procesoru računala (*CPU*) [6]. U današnje vrijeme je preuzeo brojne zadatke s *CPU*-a kao što su enkripcija i dekripcija, znanstveni izračuni, izračuni kripto valuta kao što je Bitcoin, itd. U suštini sve što zahtijeva veći izračun i troši puno resursa [6].

Razlog tomu je što je *GPU* dizajniran da podržava masivni paralelizam, što možemo vidjeti prilikom iscrtavanja slike na ekranu. Naime iste jezgre koje omogućuju brojnim pikselima da budu istodobno prikazani i sinkronizirani na ekranu, mogu istovremeno, zbog paralelizacije, obraditi nekoliko tokova podataka. Iako su jezgre *CPU*-a su složenije od jezgri *GPU*-a, njih ima tek 8 ili 12, dok ih *GPU* može imati na tisuće [6].

Kako tehnologija napreduje stavlja se sve veći fokus na razvoj programskih modela za *GPGPU*-a. Dva najpoznatija programska modela su *OpenCL* proizvođača *AMD/ATI* i *CUDA* proizvođača *NVIDIA*. *CUDA* model, koji je i korišten u ovom radu, temelji se na pozivima *API* (engl. Application Programing Interface) metoda.

4. Svojstvene vrijednosti

Iako se pojmovi svojstvene vrijednosti i svojstveni vektori većinom vežu za algebru, prvo se u 17. stoljeću koriste u analitičkoj geometriji. Stoljeće kasnije pronalazimo ih u procesu rješavanja diferencijalnih jednadžbi [8]. *U kvantnoj mehanici svojstvena vrijednost je diskretna energijska svojstvena vrijednost ili svojstvena vrijednost drugih veličina stacionarnih stanja dobivenih rješavanjem Schrödingerove jednadžbe* [9], a energijske svojstvene vrijednosti su: *vrijednosti ukupne energije E nekoga dinamičkoga sustava koje pripadaju svojstvenim funkcijama Hamiltonova operatora H toga sustava* [10].

4.1. Problem svojstvenih vrijednosti matrice

Za samo razumijevanje svojstvenih vrijednosti (engl. *eigenvalue*), prije je potrebno definirati nekoliko pojmova.

Prvo nam je potrebna kvadratna matrica, koja je zapravo matrica s jednakim brojem stupaca i redaka. Taj broj označavamo s *N* i nazivamo red matrice. U daljnjem tekstu matricu označavamo slovom *A* koja u sebi sadrži članove A_{ij}, pri čemu *i* predstavlja redak, dok *j* predstavlja stupac u danoj matrici. Vektor je jednostupčana matrica iste dimenzije kao i dimenzije *N*, tako da se može definirati množenje *Ax*. Skalar je, s druge strane, tek brojčana vrijednost.

Imajući sve navedene pojmove na umu, svojstvene vrijednosti λ od matrice *A* definiramo kao neki skalar tako da vrijedi

$$Ax = \lambda x$$

pri čemu *x* predstavlja vektor koji je različit od 0. Navedeni vektor *x* još se naziva i svojstvenim vektorom (engl. *eigenvector*).

Iz prethodne jednadžbe trebali bi dobiti elemente vektora *x* i skalara λ koji su različiti od nule te bi trebali zadovoljavati lijevu stranu jednadžbe i to zovemo problemom svojstvenih vrijednosti. Skalar (λ) koji množi vektor (*x*) zovemo svojstvenom vrijednošću (engl. eigenvalue) ili karakteristična vrijednost matrice *A*. Svaka matrica veličine *NxN* gdje je *N* red matrice sadrži *N* svojstvenih vrijednosti. Skup svih svojstvenih vrijednosti matrice *A* naziva se spektar od *A.* Najveća apsolutna svojstvena vrijednost se naziva spektralnim radijusom matrice.

Kako bismo si lakše mogli vizualizirati napisano će biti objašnjeno kroz jednostavan primjer. Uzmimo matricu *A* sljedećih vrijednosti:

$$A = \begin{bmatrix} -6 & 4 \\ 4 & -1 \end{bmatrix}$$

Navedenu matricu možemo također zapisati i prema ranijoj formuli:

$$Ax = \begin{bmatrix} -6 & 4 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Nadalje sada to možemo gledati kao dvije zasebne jednadžbe:

$$-6x_1 + 4x_2 = \lambda x_1$$

$$4x_1 - 1x_2 = \lambda x_2$$

Sljedeći korak nam je iz obje jednadžbe izlučiti x_1 i x_2:

$$(-6 - \lambda)x_1 + 4x_2 = 0$$

$$4x_1 + (-1 - \lambda)x_2 = 0$$

Nakon izlučivanja možemo uvidjeti da se navedeni izrazi mogu zapisati i pomoću matričnih izraza gdje bi sadržaj unutar izlučenih zagrada bio zapisan sljedećom notacijom:

$$(A - \lambda I)x = 0$$

Te bi takav izraz mogli uvrstiti u determinantu zapisa:

$$D(\lambda) = det(A - \lambda I) = \begin{vmatrix} -6 - \lambda & 4 \\ 4 & -1 - \lambda \end{vmatrix}$$

Determinanta nije ništa više nego homogeni sustav jednadžbi koji možemo riješiti poprilično lagano.

$$(-6 - \lambda)(-1 - \lambda) - 16 = 0$$

$$\lambda^2 + 7\lambda - 10 = 0$$

$$\lambda_1 = 1.21699$$

$$\lambda_2 = -8.21699$$

Rješavanjem jednadžbe dobivamo svojstvene vrijednosti λ_1 i λ_2 te ih također možemo zapisati kao matricu:

$$\begin{bmatrix} 1.21699 & 0 \\ 0 & -8.21699 \end{bmatrix}$$

Ovdje nam je zapravo bio cilj dobiti brojeve na glavnoj dijagonali dok su ostali elementi matrice jednaki nuli [3].

4.2. Algoritmi

Postoje brojni algoritmi koji rješavaju problem svojstvenih vrijednosti, a u samom praktičnom radu implementiran je algoritam koji rješava problem svojstvenih vrijednosti za simetrične matrice. Algoritme dijelimo na simetrične i nesimetrične odnosno algoritme za matrice koje su simetrične ili nesimetrične.

4.2.1. Simetrični algoritam

Simetrični algoritmi su algoritmi koji rješavanju problem za kvadratne matrice *NxN* koje na mjestima izvan glavne dijagonale N_{ij} i N_{ji} sadrže iste elemente. Također vrijedi i pravilo da je $N^T = N$ tj. transponirana matrica, matrica gdje se zamjenjuju redovi sa stupcima, je jednaka početnoj matrici.

Svaku simetričnu matricu karakteriziraju tri vrijednosti koje ih razlikuju od nesimetričnih matrica [11]:

1. Simetrične matrice imaju realne svojstvene vrijednosti.
2. Svojstveni vektori koji odgovaraju svojstvenim vrijednostima su ortogonalni.
3. Spektralni teorem – simetrične matrice se uvijek daju dijagonalizirati.

4.2.1.1. *Jacobijev* algoritam

Jacobijev algoritam je iterativni algoritam koji računa svojstvene vrijednosti i svojstvene vektore realne simetrične matrice. Prvi puta se javlja 1846. godine od strane *Carl Gustav Jacob Jacobi* koji je ujedno i tvorac iste.

Ideja samog algoritma je smanjiti sumu kvadrata elementa iznad ili ispod glavne dijagonale na što manju vrijednost (npr. 10^{-20}). Takvi elementi se zovu elementi izvan glavne dijagonale *(engl. off-diagonal elements*), a suma njihovih kvadrata je:

$$off(A) = \sqrt{\sum_{i=1}^{n} \sum_{j=1;\, j \neq i}^{n} {a_{ij}}^2}$$

Gdje je *A* matrica za koju se provjerava je li zadovoljen uvjet konvergiranja kvadrata elemenata izvan dijagonale u što manju vrijednost.

Kako bi se došlo do gore opisanog stanja trebamo raditi određene transformacije na zadanoj matrici *A*.

Prvi korak koji trebamo primijeniti su *Jacobijeve rotacije* koje su zapravo iste kao i *Givensove rotacije*, a služe za rotaciju vektora u nekoj ravnini. Rotacije vektora vrše se na sljedeći način [12][13]:

$$J(p,q,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ p \\ \\ q \\ \\ \\ \end{matrix}$$

Uzmimo matricu *J* za koju odabiremo indekse parova (*p,q*) koji zadovoljavaju uvjet:

$$1 \leq p < q \leq n$$

te za njih radimo izračun sinusa/kosinusa (*c,s u matrici J na mjestima p,q*) tako da se zadovolji sljedeća jednadžba:

$$\begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

λ_1 i λ_2 predstavljaju svojstvene vrijednosti matrice te se do njih dolazi sređivanjem jednadžbe iznad:

$$\begin{bmatrix} a_{pp}c^2 + a_{qq}s^2 + 2csa_{pq} & cs(a_{qq} - a_{pp}) + a_{pq}(c^2 - s^2) \\ cs(a_{qq} - a_{pp}) + a_{pq}(c^2 - s^2) & a_{pp}s^2 + a_{qq}c^2 + 2csa_{pq} \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

Zatim slijedi:

$$\frac{a_{pp} - a_{qq}}{2a_{pq}} = \frac{c^2 - s^2}{2cs} = \frac{cos2\theta}{sin2\theta} \equiv \beta$$

Napravimo zamjenu tako da slijedi $t = tg\theta$ te dobivamo:

$$t^2 + 2t\beta - 1 = 0$$

$$t = \frac{sign\beta}{|\beta| + \sqrt{\beta^2 + 1}}$$

iz toga slijedi:

$$c = \frac{1}{\sqrt{t^2 + 1}}$$

$$s = ct$$

Uvrštavanjem brojeva u gornje jednadžbe dobivamo *s* i *c* koje zatim možemo uvrstiti u matricu *J* na određena mjesta.

Nova matrica *B* koja nastaje ovom transformacijom je zapravo jednaka matrici J^TAJ gdje je $J = J(p,q,\theta)$ te se svaki sljedeći korak radi na novo dobivenoj matrici *B* ili možemo reći da svaka sljedeća matrica je:

$$A_{i+1} = J_i^T A_i J_i$$

Može se pokazati da svaka *Jacobijeva* transformacija smanjuje *off(A)* [13] i algoritam završava kada je zadovoljen uvjet $off(A) < \varepsilon$ (npr. 10^{-20}).

Implementacija algoritma napravljena je pomoću serijske verzije kao i paralelne te će svaka biti opisana posebno u svojem poglavlju.

4.2.2. Nesimetrični algoritam

Nesimetrični ili asimetrični algoritmi mogu riješiti problem svojstvenih vrijednosti za nesimetrične matrice, ali i za simetrične. Primjer takvog algoritma koji je jako popularan je QR algoritam.

4.2.2.1. *QR* algoritam

QR algoritam rješava problem svojstvenih vrijednosti za nesimetrične matrice, ali se isti može i iskoristiti za simetrične matrice. Računanje vrijednosti temelji se na rastavljanju matrice na faktore. Takvo rastavljanje se naziva *QR faktorizacija* matrice. Drugim riječima rastavljamo matricu *A* na:

$$A = QR$$

Gdje je *Q* ortogonalna matrica, a *R* gornja trokutasta matrica. Rastavljanje matrice na *QR faktorizaciju* traje $O(n^3)$ iteracija. Za ortogonalnu matricu vrijedi:

$$QQ^T = I$$

Za matricu kažemo da je gornje trokutasta ako su elementi ispod glavne dijagonale jednaki nuli. Zbog ortogonalnosti matrice *Q* vrijedi pravilo:

$$Q^TA = R$$

Ako sredimo jednadžbu iznad tako da je pomnožimo s *Q*, tj. ako primijenimo *Q* s desne strane jednadžbe dobit ćemo:

$$A_{n+1} = Q^T A_n Q_n$$

Ponavljanje iznad navedenog procesa zove se *QR iteracija*. Svaka sljedeća matrica se postepeno transformira prema gornje trokutastoj matrici koja je slična početnoj matrici. Za transformaciju matrice na *QR faktorizaciju* mogu se koristiti *Givensove rotacije* koje su jednake *Jacobijevim rotacijama*.

Algoritam *QR iteracije* bi izgledao:

```
for k = 1,2,… do
      A_{k-1} =: Q_k R_k;
      A_k := R_k Q_k;
```

Kada je algoritam završio na glavnoj dijagonali ćemo dobiti svojstvene vrijednosti, a svaka nova matrica dobivena *QR iteracijom* je slična početnoj matrici. Navedena sličnost se može zapisati pomoću jednadžbe:

$$A_{k+1} = R_k Q_k = Q_k^* Q_k R_k Q_k = Q_k^* A_k Q_k = Q_k^{-1} A_k Q_k$$

5. Potprogrami

Za izradu praktičnog rada prvotno su napravljeni potprogrami koji će olakšati implementaciju zadatka te će se izbjeći višestruko definiranje sličnih metoda. Struktura pomoćnih programa napisana je u programskom jeziku *C* te se uključuju u ostale programe pomoću strukture zaglavlja (*engl. header*).

5.1. Matrix helper

Matrix helper sadrži programe koji generiraju matrice, rade ispise matrica i polja. Sastoji se zapravo od dva zaglavlja:

- *Matrix_generate_random.h*
- *Print_results.h*

5.1.1. Matrix generate random

U spomenutoj datoteci nalaze se metode za generiranje matrice tipa *double*. Generiranje matrica je dinamično tj. memorija se alocira prema veličini matrice.

Pokretanjem glavnog djela programa prvo se postavi veličina matrice koja se želi generirati te se po potrebi ta veličina može i promijeniti. Način na koji je to napravljeno je pomoću jednostavne varijable kombinirana sa *set* i *get* metodama.

Metoda za generiranje matrice je dvostruki pokazivač te je tipa *double*. Za generiranje brojeva tj. elemenata u matrici koristi se ugrađena metoda *drand48()* biblioteke *stdlib* koja generira slučajne brojeve između nula i jedan. Prije generiranja brojeva postavimo *sjeme* kroz poziv metode *srand48()* kako bismo kod svakog generiranja matrice imali zapravo iste brojeve na pripadajućim pozicijama.

Postavljanje brojeva na pozicije se vrši kroz dvije petlje te je potrebno zapravo samo generirati jednu stranu matrice, iznad ili ispod dijagonale, pošto je matrica simetrična te kopirati broj na simetrično mjesto.

Potrebna nam je još jedna od metoda kako bismo imali sve potrebno za izračun svojstvenih vrijednosti. Potrebno je generirati još jednu matricu koja će biti jedinična, a takva matrica ima jedinice na glavnoj dijagonali dok su ostali elementi jednaki nuli. Matrica se generira slično kao i ostale samo bez metode slučajnih brojeva.

Izvoz svih potrebnih metoda stavljen je u zaglavlje te se može po potrebi uključiti u ostale dijelove programa. U prilogu 1 možemo vidjeti programsku implementaciju.

5.1.2. Print results

Print_results je potprogram koja sadrži metode za ispis matrica, polja i svojstvenih vrijednosti.

Metoda za ispis svojstvenih vrijednosti iz matrice prolazi kroz matricu i traži elemente na glavnoj dijagonali. Pošto ti elementi nisu sortirani prvo je napravljeno spremanje u pomoćno jednodimenzionalno polje koje će se kasnije sortirati pomoću *qsort()* ugrađene u *stdlib* biblioteku. U prilogu 2 možemo vidjeti programsku implementaciju.

5.2. Time

Time je jedan od potprograma koji nam omogućava mjerenje između dvije točke u programu [14].

Implementacija koristi ugrađenu funkciju programskog jezika *C clock_t clock(void)* koja vraća broj otkucaja sata proteklih od pokretanja programa. Ako želimo dobiti vrijeme u sekundama trebamo podijeliti broj otkucaja s varijablom, također ugrađenom u biblioteku *C* jezika, CLOCKS_PER_SEC. Program je dosta jednostavan te vraća precizno mjerenje vremena.

Isto tako je napravljeno i mjerenje stvarno proteklog vremena kako bismo zapravo mogli usporediti koliko je prvo vrijeme precizno. Za takvu vrstu mjerenja odabrana je metoda *gettimeofday()* biblioteke *time.h*.

Korištenje programa je jednostavno. Za početak mjerenja vremena pozovemo metodu *startTimer()*, a za završetak *endTimer()*. Za dohvat rezultata pozovemo metodu *getTime()* ako želimo vrijeme koje je proteklo na *CPU*, a za stvarno proteklo vrijeme pozivamo metodu *getWallClockTime().* U prilogu 3 možemo vidjeti programsku implementaciju.

6. LAPACK

LAPACK (engl. *Linear Algebra PACKage*) [5] u ovom programu služi za kontrolu točnosti svojstvenih vrijednosti s vlastitim implementiranim verzijama. Za potrebe računanja svojstvenih vrijednosti je već ranije pripremljena okolina te je sada jednostavno pozvati odgovarajuću rutinu.

Treba nam rutina za dijagonalizaciju simetrične matrice koja će biti tipa *double*. Rutina koja radi izračun za taj tip podataka počinju sa slovom *d (*engl. *double precision)* pa je naziv rutine koja se poziva DSYEV.

6.1. DSYEV

Navedena rutina može poslužiti za izračun svojstvenih vrijednosti matrice, ali i također svojstvenih vektora matrice. Nama trebaju samo svojstvene vrijednosti te će kroz izvorni kod biti objašnjen postupak pozivanja.

```
void dsyev()
{
    MATRIX_ORDER = getMatrixOrder();
    double **a = symetric_matrix_double();
    lapack_int info, n, lda;

    char jobz = 'N';        // -- Eigenvalues only
    char uplo = 'U';        // -- Upper triangle is stored
    n = MATRIX_ORDER;       // -- Order of the matrix
    lda = MATRIX_ORDER;     // -- The leading dimension of the array
    double w[MATRIX_ORDER]; // -- Used to store eigenvalues results

    print_matrix_double(a, MATRIX_ORDER, "Generated matrix:");
    startTimer();
    info = LAPACKE_dsyev(LAPACK_COL_MAJOR, jobz, uplo, n, *a, lda, w);
    endTimer();
    /* Print Solution */
    print_eigenvalues_double("EIGENVALUES FOR MATRIX DSYEV: ", MATRIX_ORDER,
w);
    printf("\n");
```

```
    free(*a);
    free(a);
    printf("Total time: %f\n", getTime());
}
```

Prije poziva rutine trebamo provesti potrebne inicijalizacije. Najprije trebamo dohvatiti red matrice koju je korisnik unio prilikom pokretanja programa. Zatim generiramo matricu na prethodno objašnjen način te inicijaliziramo varijable potrebne za rutinu. Rutina prima 7 argumenata koji su sljedeći:

- *LAPACK_COL_MAJOR* – označava raspored (engl. *layout*) matrice.
- *JOBZ* – znak koji označava želimo li izračunati svojstvene vrijednosti zajedno sa svojstvenim vektorima ili samo svojstvene vrijednosti.
- *UPLO* – znak koji govori je li spremljen gornji ili donji trokut matrice.
- *N* – red matrice.
- *A* – matrica tipa *double* s ispunjenim elementima.
- *LDA* – broj koji nam govori vodeću dimenziju matrice.
- *W* – polje brojeva u koje će se zapisati rezultati.

Prije poziva rutine pokrećemo mjerenje vremena. Pozivom rutine s iznad definiranim parametrima dobivamo rezultat koji kasnije trebamo samo ispisati uz pomoć također ranije kreiranih pomoćnih metoda. Prije nego što program završi trebamo očistiti memoriju koju smo zauzeli kako resursi koje smo koristili ne bi zauzimali mjesto u memoriji. Zaustavljanjem vremena dobivamo proteklo vrijeme te ga možemo usporediti s ostalim vremenima.

7. Serijska verzija

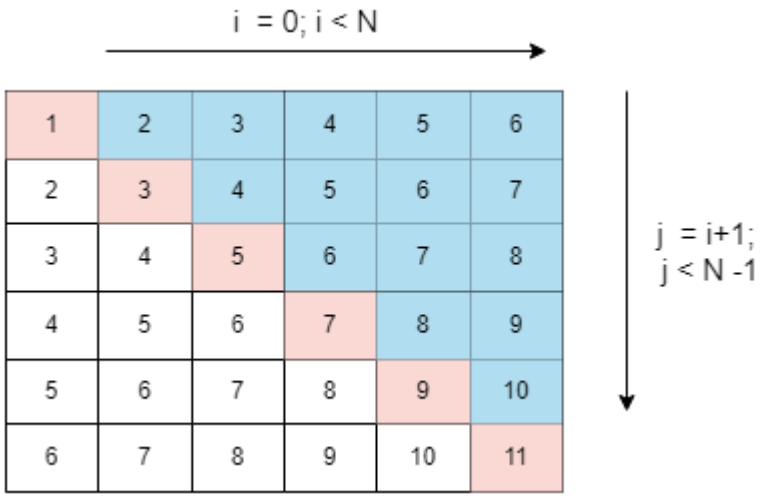
Za serijsku verziju implementiran je *Jacobijev algoritam* u programskom jeziku *C* te okruženju *Ubuntu* opisanom u poglavlju metode i tehnike rada. Algoritam radi na temelju ranije opisanih *Jacobijevih rotacija*. Implementirana verzija rješava problem svojstvenih vrijednosti tako da na mjestima N_{ij} i N_{ji} postavlja nule te na odgovarajući način modificira stupce *i* i *j* te redove *j* i *i*.

Prije nego što započnemo s transformacijom matrice, potrebno je definirati kriterij zaustavljanja algoritma.

Kriterij za zaustavljanja algoritma je opisan ranije te je on korijen iz sume kvadrata svih elemenata iznad ili ispod glavne dijagonale. Ta suma se izračunava:

```
  double precision(double **matrix, int matrix_order)
{
    double suma = 0;
    for (int i = 0; i < matrix_order - 1; i++)
    {
        for (int j = i + 1; j < matrix_order; j++)
        {
            suma += pow(matrix[i][j], 2);
        }
    }
    return sqrt(suma);
}
```

Metoda *double precision(double **matrix, int matrix_order)* kao ulaz prima matricu, koja je transformirana *Jacobijevim rotacijama,* te red matrice. Algoritam se sastoji od dvije petlje koje zapravo prolaze kroz svaki element iznad dijagonale te ga kvadriraju, kako bi dobili apsolutne vrijednosti, te pamte sumu elemenata. Unutarnja petlja počinje od prvog sljedećeg elementa na dijagonali.



Slika 1: Suma elemenata iznad dijagonale (Izvor: Vlastita izrada)

Prilikom svake iteracije/transformacije matrice radi se navedena provjera pomoću petlje *while* koja završava kada je uvjet zadovoljen.

Unutar uvjeta radi se transformacija matrice koja započinje od prvog reda matrice pa sve do predzadnjeg. Zadnji red ne trebamo zapravo transformirati jer u njemu nemam više plavih elemenata (vidjeti sliku 2) koje treba poništiti. Odabirom redova, počevši od prvog pa sve do predzadnjeg, radi se odabir stupaca koji kreće od indeksa retka uvećanog za jedan. Da bismo izveli navedeno implementiramo dvije petlje od kojih indeks *i* predstavlja redak, a indeks *j* predstavlja stupac.

```
while (precision(matrix, MATRIX_ORDER) > 1e-20)
{
    for (int i = 0; i < MATRIX_ORDER - 1; i++)
    {
        for (int j = i + 1; j < MATRIX_ORDER; j++)
        {
```

Tako ćemo dobiti sve parove N_{ij}, N_{ji}, N_{ii} i N_{jj} koji su potrebni za izračun sinusa i kosinusa. Na temelju opisanih formula u poglavlju *Jacobijevog algoritma* radimo izračun vrijednosti *c* i *s* unutar petlja od kojih jedna prolazi kroz redove, a druga kroz stupce.

```
if (matrix[i][j] != 0)
{
      double b = (matrix[i][i] - matrix[j][j]) / (2.0 * matrix[i][j]);
```

```
    double t = ((b > 0) ? 1.0 : ((b < 0) ? -1.0 : 0.0)) / ((fabs(b)) +
sqrt(pow(b, 2) + 1.0));
    double c = 1.0 / (sqrt(pow(t, 2) + 1.0));
    double s = c * t;
```

Prije nego što radimo izračun vrijednosti *c* i *s,* radimo i provjeru je li navedeni element jednak nuli kako bismo preskočili element koji je već poništen. Kada smo uspješno dobili vrijednosti, možemo krenuti s transformacijom elemenata u matrici.

Najprije radimo promjenu elemenata koji su na dijagonali gledanog pravokutnika, a čine ga trenutni indeksi.

Nii	2	3	4	Nij	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
Nji	6	7	8	Njj	10
6	7	8	9	10	11

Slika 2: Trenutni odabir stupca i retka (Izvor: Vlastita izrada)

Prije nego što na određenim mjestima izračunamo nove vrijednosti, trebamo prethodne zapamtiti jer su nam potrebne za izračun novih. Ako unesemo nove vrijednosti bez da smo prethodne zapamtili dobit ćemo krive rezultate. Iz navedenog razloga uvodimo pomoćne varijable koje će sadržavati vrijednosti potrebne za izračun novih elemenata.

```
double mii = matrix[i][i];
double mij = matrix[i][j];
double mjj = matrix[j][j];

matrix[j][j] = (pow(s, 2) * mii) - (2.0 * c * s * mij) + (pow(c, 2) * mjj);
matrix[i][i] = (pow(c, 2) * mii) + (2.0 * c * s * mij) + (pow(s, 2) * mjj);
```

Na mjestima koja nisu na dijagonali, ali su u kutu trenutno odabranog pravokutnika slobodno možemo upisati kao vrijednosti broj nula jer se ti elementi prema *Jacobijevim rotacijama* poništavaju.

```
matrix[i][j] = matrix[j][i] = 0;
```

Ono što zapravo radimo je sljedeće:

$$A_{ij} = A_{ji} = (c^2 - s^2)A_{ij} + cs(A_{jj} - A_{ii}) = 0$$

Sljedeći korak nam je proći kroz sve elemente stupca i redova indeksa *i* i *j* iz razloga zato što se *Jacobijevim rotacijama* mijenjaju odabrani redovi i odabrani stupci što proizlazi iz

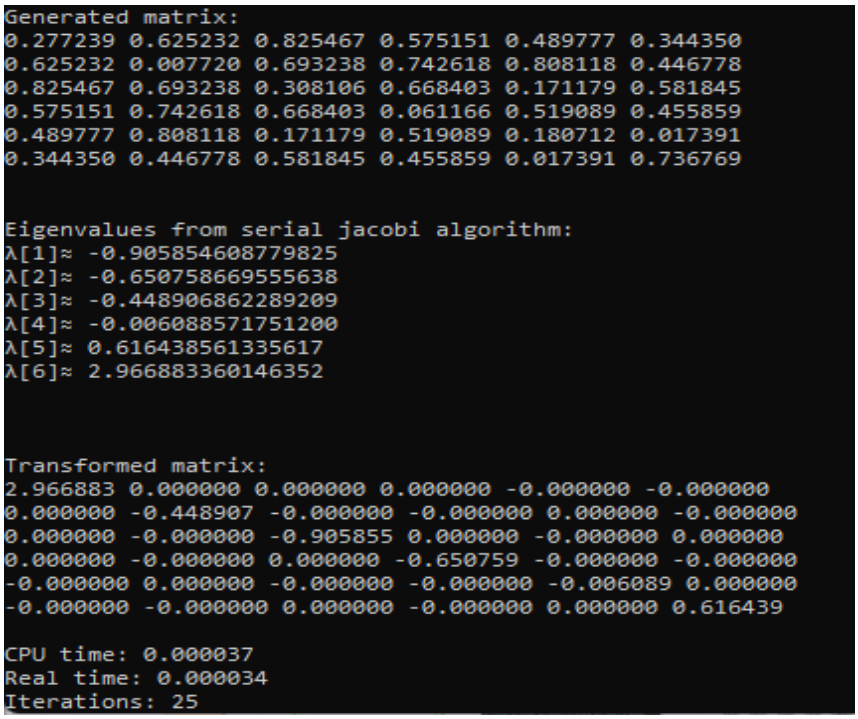
$$A_i = J^T A_{i-1} J$$

gdje je *J Jacobijeva* matrica s vrijednostima *c* i *s*.

To radimo tako da pokrenemo novu petlju koja će ići po svim elementima redova i stupaca te računati nove vrijednosti. Također, prije računanja novih vrijednosti, trebamo spremiti prethodne vrijednosti kako ih ne bi prebrisali kod zamjene jer su nam potrebne za izračun novih. To radimo na sljedeći način:

```
for (int k = 0; k < MATRIX_ORDER; k++)
{
	if (k != i && k != j)
	{
		double mik = matrix[i][k];
		double mjk = matrix[j][k];
		double mki = matrix[k][i];

		matrix[i][k] = matrix[k][i] = c * mik + s * mjk;
		matrix[j][k] = matrix[k][j] = c * mjk - s * mki;
	}
}
```

Dovoljno je samo napraviti izračun na mjestu N_{ik} i N_{jk}, zatim ih zapisati na njihovo simetrično mjesto. Nakon promjene svakog elementa u stupcima *i* i *j* te redovima *i* i *j* završili smo s transformacijom matrice te bi time elementi matrice na dijagonali trebali biti bliži svojstvenim vrijednostima matrice. Algoritam završava kada je zadovoljen uvjet *off(A)*. Svojstvene vrijednosti se nalaze na glavnoj dijagonali matrice. Rezultate možemo vidjeti na slici niže koja prikazuje slučajno generiranu simetričnu matricu te njene svojstvene vrijednosti.



Slika 3: Svojstvene vrijednosti pomoću *Jacobijevog* algoritma (Izvor: Vlastita izrada)

8. CUDA

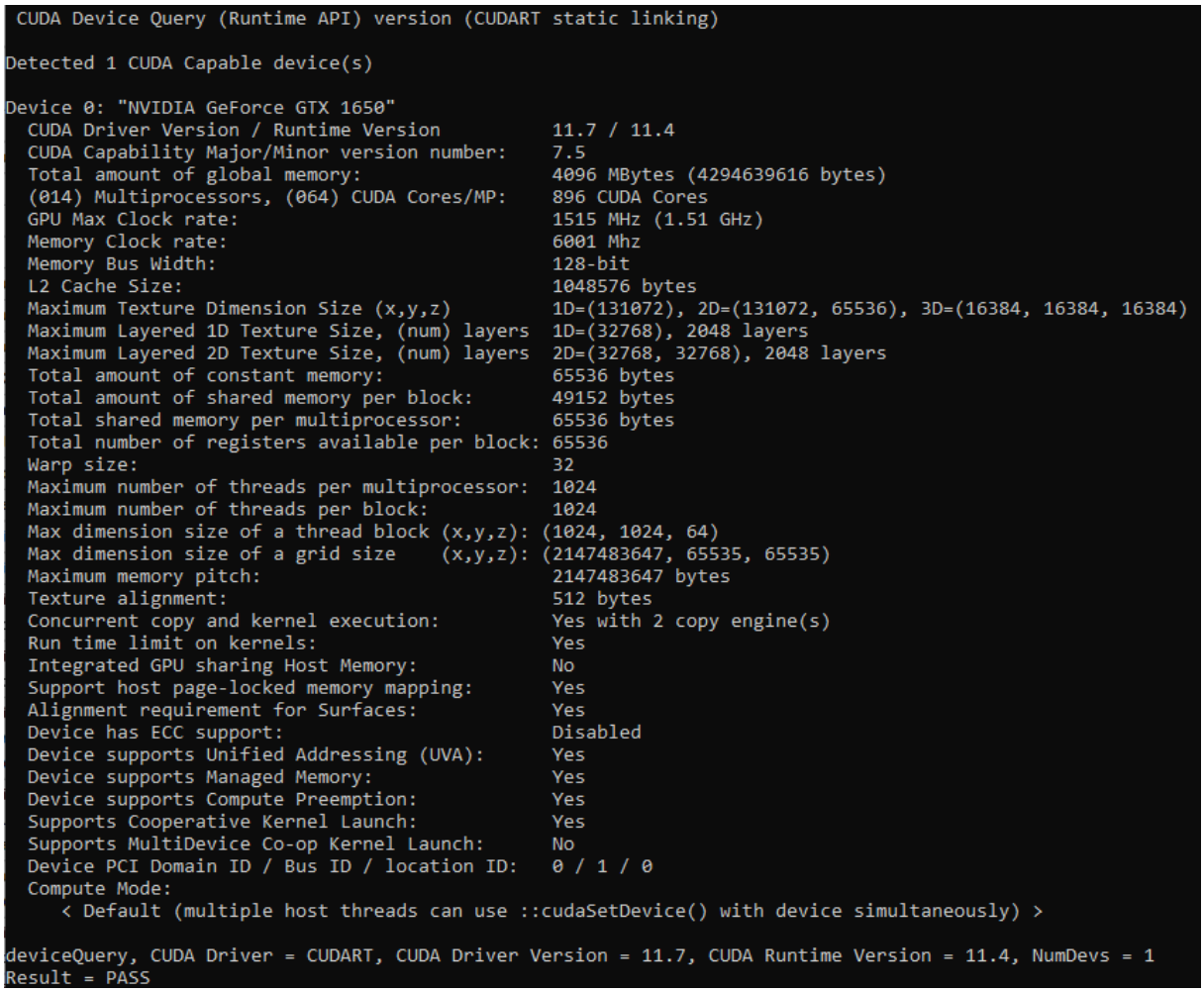
Za realizaciju paralelne verzije izračuna svojstvenih vrijednosti matrice izabran je programski model *CUDA* (*Compute Unified Device Architecture*) proizvođača grafičkih procesora *NVIDIA*. Navedena arhitektura nudi nam mogućnost paralelnog programiranja.

Paralelno programiranje je proces kojim se problem rastavlja na nekoliko dijelova te se svaki od tih procesa izvodi istovremeno. Takvo programiranje nam nudi novu perspektivu. Pokretanjem paralelnih procesa zauzimaju se novi resursi u računalu kojih ima konačan broj. U današnje vrijeme se može izvoditi na *CPU*-u zbog sve većeg broja jezgri. Iako taj broj jezgra nije veći od nekoliko desetaka, rastavljene procese može izvršiti puno brže od serijskog programiranja.

Za razliku od *CPU*, grafički procesor (*GPU)* je masovno paralelno računalo koje dolazi s mogućnosti raspodjele procesa na nekoliko tisuća pa čak i desetaka tisuća pojedinih jezgri.

NVIDIA je razvila programski model koji kroz *API* sustav nudi mogućnost veoma efikasnog paralelnog programiranja. Pozivom određenih *API* metoda procesi se mogu raspodijeliti na određen broj dretava koji ovise o verziji *CUDA* programskog modela zajedno s vrstom grafičkog procesora. Na slici 4 možemo vidjeti s kojim resursima se raspolaže u ovome radu.

Maksimalan broj dretva koje se mogu kreirati je 1024, ali to nije maksimalan broj na koji možemo raspodijeliti procese. Taj se broj odnosi na broj dretvi unutar jednog bloka u jednoj dimenziji. Kompletna struktura i način na koji se određuje broj dretva po bloku će biti objašnjena kroz praktičan primjer.



Slika 4: Resursi za masovno računalno u programu *CUDA* (Izvor: Vlastita izrada)

8.1. Struktura CUDA i primjer

Struktura programa temelji se na standardnom *C* jeziku. Za izradu i razumijevanje *CUDA* programa potrebno nam je za početak samo znanje *C* jezika.

Terminologija kojom se raspolaže kada pričamo o *CUDA* programu je:

- Domaćin (engl. *Host*) – Procesor računala sa svojom memorijom (*CPU*, *RAM*, *HDD*)
- Uređaj (engl. *Device*) – Grafička kartica sa svojom memorijom (*GPU*, *VRAM*)

Jednostavan *CUDA* program sadrži glavnu metodu (*main*) koja se poziva kod pokretanja programa kao i svaki drugi *C* program. Unutar *main* metode možemo raditi pozive metoda koje se izvršavaju na *GPU*-u. Metode za koje se želi da se izvrše na grafičkom

procesoru zovu se *jezgre* (engl. *kernel*) te sadrže prefiks *__global__* prije svojega naziva. Primjer takve metode:

```
__global__ void kernel(void){}
```

Prefiks *__global__* označava nam da se metoda/jezgra može pozvati s domaćina tako da se specificira broj dretva i blokova s kojima će se izvršavati. Poziv jezgre s domaćina se radi na sljedeći način:

```
int main(void){
	int blocks = 1;
	int threads = 1;
	kernel<<<blocks, threads >>>();
	return 0;
}
```

U programskom kodu iznad pozvali smo metodu koja će sadržavati jednu dretvu po bloku. Možemo uočiti kako se broj blokova i dretva definira unutar trostrukih zagrada.

8.1.1. Upravljanje memorijom

Da bi se podacima s kojima želimo raditi moglo pristupiti putem grafičkog procesora potrebno je alocirati dovoljno memorije na uređaju te zatim prenijeti podatke s domaćina na uređaj. Navedeno se radi putem ugrađenih metoda koje su slične alociranju memorije na domaćinu. Metode koje se mogu pozvati su:

- *cudaMalloc*() – alocira se memorija na GPU-u
- *cudaFree*() – oslobađa se zauzeta memorija na GPU-u
- *cudaMemcpy*() – vrši se kopiranje memorije (postoji više vrsta)

Razlog zašto se memorija treba alocirati je zato što svi ulazni parametri s kojima će se raditi na *GPU*-u će biti pokazivači (engl. *pointer*) na memoriju uređaja.

Kako bismo prenijeli podatke s domaćina na uređaj radimo sljedeće:

```
__global__ void sum(int *a, int *b, int *c){
	*c = *a + *b;
}

int main(void){
	int a,b,c;
```

```
	int *dev_a, *dev_b,*dev_c;
	int size = sizeof(int);

	cudaMalloc((void**)&dev_a,size);
	cudaMalloc((void**)&dev_b,size);
	cudaMalloc((void**)&dev_c,size);

	a = 2;
	b = 2;

	cudaMemcpy(dev_a,&a,size, cudaMemcpyHostToDevice);
	cudaMemcpy(dev_b,&b,size, cudaMemcpyHostToDevice);

	sum<<<1,1>>>(dev_a,dev_b,dev_c);

	cudaMemcpy(&c,dev_c,size, cudaMemcpyDeviceToHost);

	cudaFree(dev_a);
	cudaFree(dev_b);
	cudaFree(dev_c);

	return 0;
}
```

Ono što smo zapravo napravili je da smo prvo definirali varijable koje će biti potrebne za domaćina i uređaj te smo ih imenovali tako da ih možemo razlikovati (prefiks *dev* za varijable na uređaju). Zatim smo alocirali memoriju potrebnu za varijablu tipa *int* (*sizeof(int)*). Kada smo to napravili možemo kopirati sadržaj domaćina na uređaj s metodom *cudaMemcpy()* gdje je smjer kopiranja: domaćin → uređaj. Isto tako postoji kopiranje memorije: uređaj → domaćin, uređaj → uređaj te domaćin → domaćin. Pozivom metode *sum* zapravo zapisujemo rezultat operacije u memoriju uređaja kojega kasnije možemo kopirati natrag na domaćina te smo na taj način uspješno napravili aritmetičku operaciju putem uređaja. Ako smo završili s izračunom ne smijemo zaboraviti osloboditi memoriju uređaja s *cudaFree()*.

Opisani primjer je koristio brojeve tipa *int* dok se mogu koristiti i ostali tipovi. Za ostale tipove radimo isto alociranje memorije samo što onda moramo alocirati mjesta za taj tip podataka npr. *sizeof(double)*. Ako imamo pokazivač koji će biti polje tada trebamo alocirati memoriju za *veličina_varijable*veličina_polja* ili *sizeof(int) * N*.

8.1.2. Primjer s više blokova

Sada kada smo shvatili kako funkcionira upravljanje memorijom i kako se podaci prenose s domaćina na uređaj možemo krenuti s malo kompleksnijim primjerom. Želimo iskoristiti funkcionalnost podjele procesa koju imamo, a to možemo napraviti na način da povećamo ili broj dretva ili broj blokova kod poziva jezgre. Ono što možemo napraviti je da iskoristimo indeks bloka i/ili dretve koji se izvršava kao indeks definiranog polja.

Za pristup indeksu dretve/bloka koji se trenutno izvršavaju, programski model *CUDA* nudi ugrađene varijable. Ako imamo specificiranu samo jednu dimenziju pristup je slijedeći:

- blockIdx.x – indeks bloka koji je trenutno u izvršavanju
- threadIdx.x – indeks dretve unutar bloka koji je trenutno u izvršavanju
- blockDim.x – veličina jednog bloka

Ako imamo drugu ili treću dimenziju na kraju varijable slovo *x* zamijenimo s *y/z*. Npr. *threadIdx.y/threadIdx.z*.

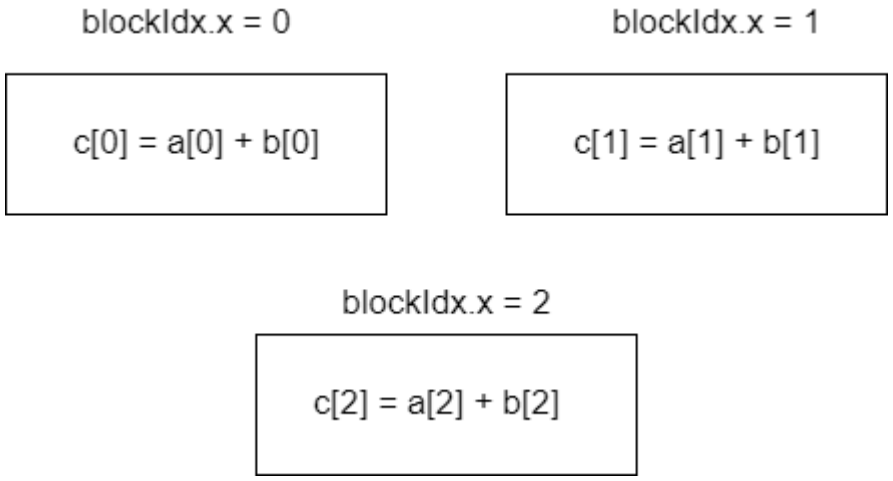
```
__global__ void sum(int *a, int *b, int *c){
      int indeks = blockIdx.x;
      c[indeks] = a[indeks] + b[indeks];
}
```

Ako uzmemo u obzir da kod poziva globalne metode *sum* postavimo da se metoda izvršava s tri paralelna bloka, od kojih svaki blok sadrži jednu dretvu, u varijablu *c* na mjesto *indeksa* će se zapisati vrijednost sume druga dva polja. Kod poziva te metode moramo paziti da smo ispravno alocirali memoriju za *N* broj blokova.

```
int main(void)
{
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;
    int N = 3;
    int size = N * sizeof(int);

    cudaMalloc((void **)&dev_a, size);
    cudaMalloc((void **)&dev_b, size);
    cudaMalloc((void **)&dev_c, size);

    a = (int *)malloc(size);
```

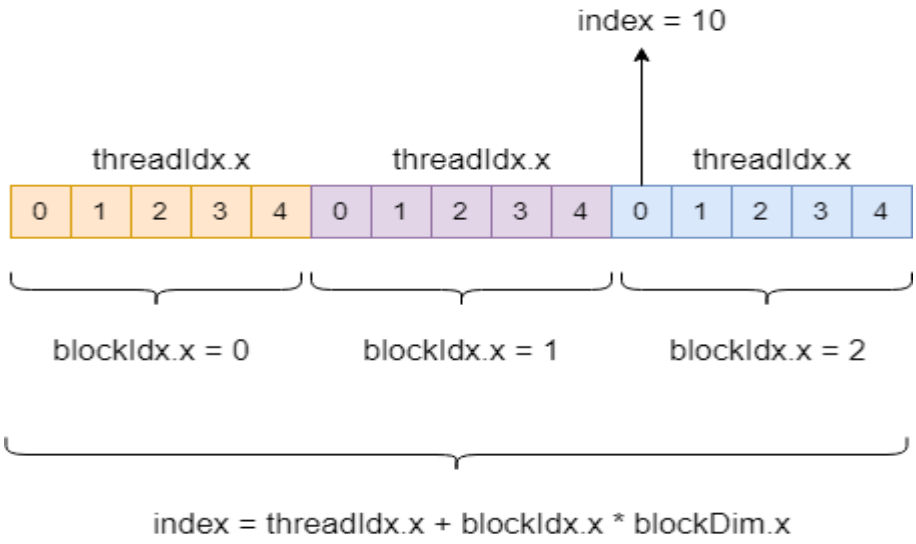
```
    b = (int *)malloc(size);
    c = (int *)malloc(size);

    random_ints(a, N);
    random_ints(b, N);

    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

    sum<<<N ,1>>>(dev_a, dev_b, dev_c);

    cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Vizualizacija izvršavanja bi izgledala kao na slici 5:



Slika 5: Paralelni blokovi (Izvor: Vlastita izrada)

Budući da se unutar jednog bloka može povećati broj dretvi, trebamo pratiti određen način indeksiranja istih. Ako na primjer želimo pokrenuti metodu koja sadrži 5 dretava na 3 bloka struktura će izgledati kao na slici 6:



Slika 6: Blokovi s nekoliko dretava (Izvor: Vlastita izrada)

8.1.3. Vrste metoda i varijabla

Već smo upoznali jedan prefiks za deklariranje varijabli i metoda naziva *__global__*. To je prefiks koji nam omogućuje određivanje metode koja će biti jezgra, ali isto tako može biti i za određivanje globalne varijable. Globalna metoda može se pozvati samo s domaćina. Izuzev deklariranja globalne metode postoje još dva prefiksa za deklariranje metoda. Jedan on njih je *__device__*. On označava metodu koja se može pozvati samo s uređaja. Isto tako postoji i prefiks *__host__* koji označava metodu koja se može pozvati i s uređaja i s domaćina.

Dosta česta implementacija paralelnog programiranja, te jedna od prvih implementacija prilikom proučavanja kako funkcioniraju dretve, a kako blokovi. Izrada je jezgre koja radi skalarno množenje vektora. Rješenje se može implementirati na više načina, dok će ovdje biti objašnjeno kroz korištenje prefiksa *__shared__* te sinkronizacije dretva. Jezgru ćemo pokrenuti s *N* dretva te samo jednim blokom. Ulaz u jezgru će biti dva polja dimenzije *N*, a izlaz će biti skalarni produkt.

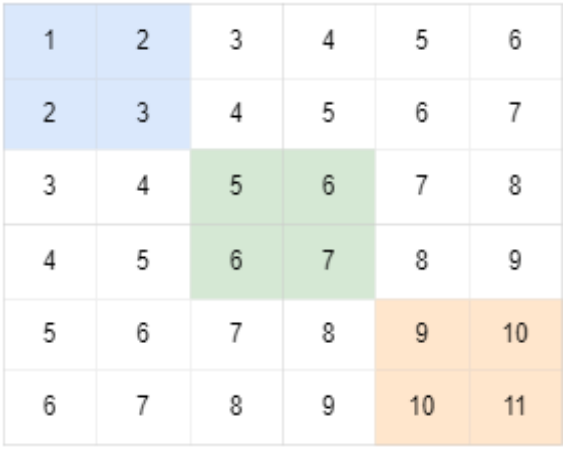
```
__global__ void mulitply( int *a, int *b, int *c ) {
	__shared__ int temp[N];
	int indeks = threadIdx.x;
	temp[indeks] = a[indeks] * b[indeks];
	__syncthreads();
	if( 0 == threadIdx.x )
	{
		int sum = 0;
		for( int i = 0; i < N; i++ )
			sum += temp[i];
		*c = sum;
	}
}
```

Prvo dohvaćamo indeks trenutne dretve u izvršavanju. Dalje deklariramo varijablu *temp* koja ima prefiks *__shared__* te on omogućuje dretvama da imaju pristup istoj varijabli. Ako deklariramo varijablu bez navedenog prefiksa, pojedine dretve neće moći pristupiti varijabli jer će biti definirana kao *private*. Zatim iskoristimo indeks dretve kako bismo napravili množenje skalara na odgovarajućim mjestima te rezultat zapišemo u polje *temp*, također na odgovarajuće mjesto. Kako bismo bili sigurni da su sve dretve završile s izvršavanjem i napravile željenu operaciju pozivamo metodu __syncthreads(). Sav kod koji je pisan nakon ovog poziva će se izvršiti tek nakon što su sve dretve izvršile svoj zadatak tj. kada su se sinkronizirale. Zadnji korak nam je iskoristiti prvu dretvu koja će pomoću *for* petlje napraviti sumu elemenata u polju.

Razlog zašto želimo koristiti dretve u kombinaciji s blokovima je jer nam dretve, za razliku od blokova, nude mogućnost sinkronizacije i komuniciranja.

8.2. Paralelna implementacija *Jacobijevog* algoritma

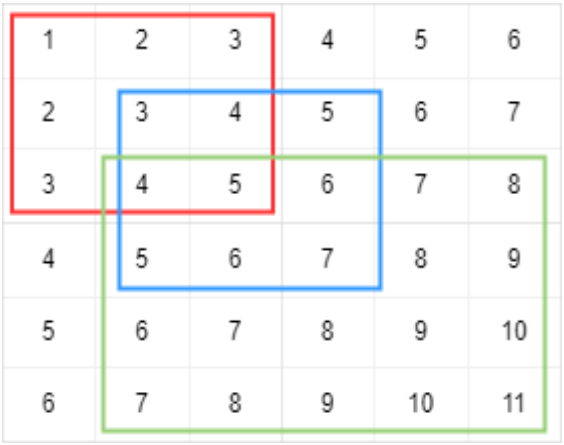
Premisa implementacije paralelnog *Jacobijevog algoritma* leži u teoriji odabiru parova *i* i *j* koji se međusobno ne ometaju. Pošto se prilikom odabira parova u početnoj matrici mijenjaju samo redovi i stupci na odabranim pozicijama, pametnim odabirom istih možemo paralelizirati i ubrzati proces pronalaženja svojstvenih vrijednosti. Broj takvih parova u kvadratnoj matrici iznosi *N/2*. Parovi koje se međusobno ne ometaju izgledali bi ovako:



Slika 7: N/2 ne ometajućih parova (Izvor: Vlastita izrada)

Indekse možemo spariti i na drugi način. Bitno je da elementi na pozicijama N_{ii} i N_{jj} nisu kutni elementi drugih parova.

Npr.

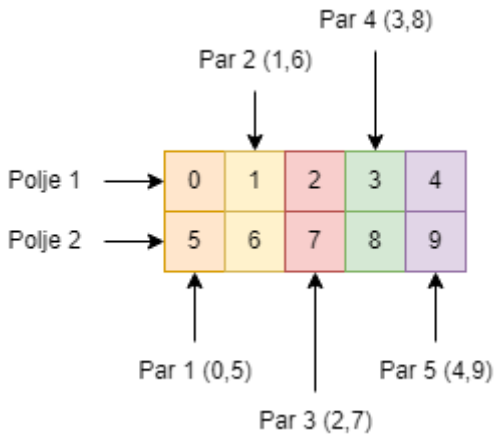


Slika 8: *N/2* ne ometajućih parova s preklapanjem (Izvor: Vlastita izrada)

U svrhu odabira takvih parova implementiran je algoritam *chess ordering tournament* [15].

8.2.1. *Chess ordering tournament* algoritam

Kako bismo uspješno prošli kroz sve elemente matrice te ih spojili, napravljena su dva polja prirodnih brojeva koja sadrže indekse parova. Početno su polja napunjena brojevima koja izgledaju kao na slici ispod.



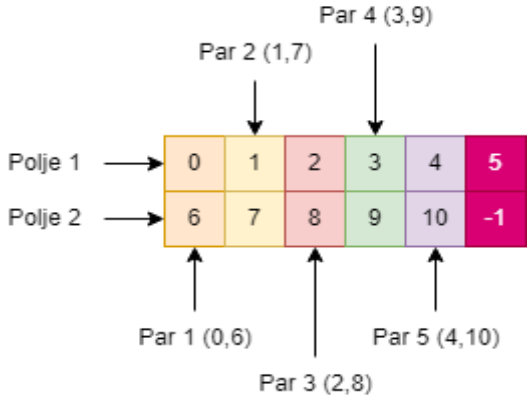
Slika 9: Odabir parova – početno stanje (Izvor: Vlastita izrada)

Inicijalizacija parova u polje je napravljena na domaćinu te je kasnije početno stanje kopirano na uređaj gdje se radio pomak. Implementacija na domaćinu je sljedeća:

```
void initializeMatrixPairs(int *array1, int *array2, int matrix_order)
{
    int j = 0;

    if (matrix_order % 2 != 0)
    {
        matrix_order++;
    }

    for (int i = 0; i < matrix_order / 2; i++)
    {
        array1[i] = i;
    }
    for (int i = matrix_order / 2; i < matrix_order; i++)
    {
        array2[j] = i;
        j++;
    }

    if (matrix_order % 2 != 0)
    {
```

```
            array2[matrix_order / 2 - 1] = -1;
        }
}
```

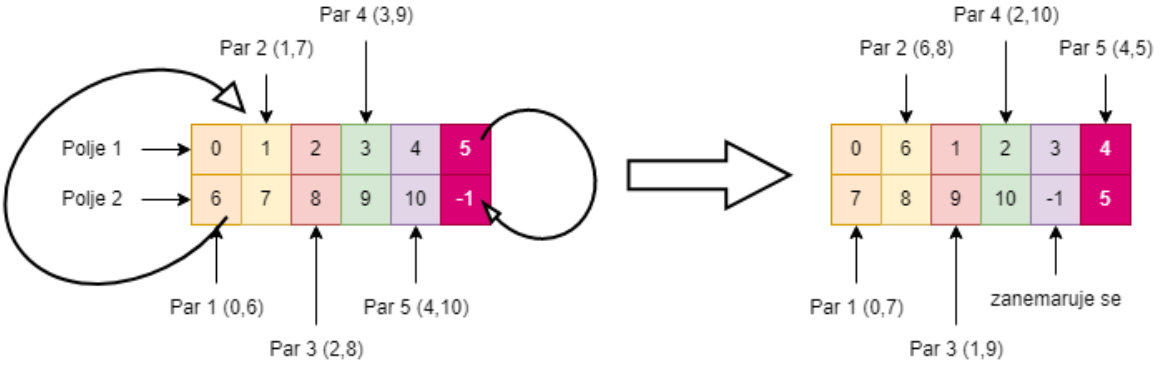
Slika 9 nam prikazuje kako izgledaju parovi kada je red matrice paran, a slika 10 prikazuje nam odabir parova za neparan red matrice.



Slika 10: Parovi za neparan red matrice (Izvor: Vlastita izrada)

Vidljivo je da ako je red matrice jednak 9, dodajemo jedan novi element u svako polje i u jednom od tih elemenata stavit ćemo zastavicu, u ovom slučaju broj -1, koja će nam govoriti da taj par trebamo preskočiti.

Sada nam samo ostaje kod svake iteracije generirati nove parove. Funkcionira u smjeru kazaljke na satu na način da zamijenimo mjesta elementima, dok element prvog polja ostaje na svojem mjestu. Prvi pomak vidimo na slici 11:



Slika 11: Pomak parova (Izvor: Vlastita izrada)

Pomak parova se vrši na jezgri uz pomoć *N/2* dretvi te pomoćnih polja. Pomoćna polja nam služe za pamćenje prethodnog stanja, odnosno prije nego što ih prepišemo. Ovaj način je odabran jer se u istom trenutku radi pristup svim poljima te nema lakog načina pamćenja prethodnog stanja osim spremanja prethodnog stanja u navedena polja.

```
__global__ void shiftJacobiPairs(int *pair1, int *pair2, int *temp1, int *temp2, double **matrix_Host, int matrix_order)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int size = matrix_order;

    if (matrix_order % 2 != 0)
    {
        size++;
    }

    if (index < size / 2)
    {
        if (index == 0)
        { // If it is first element
            pair2[index] = temp2[index + 1];
        }
        else if (index == 1)
        { // If it is second element
            pair1[index] = temp2[0];
            pair2[index] = temp2[index + 1];
        }
        else if (index == (size / 2 - 1))
        { // If it is last element
            pair1[size / 2 - 1] = temp1[size / 2 - 2];
            pair2[size / 2 - 1] = temp1[size / 2 - 1];
        }
        else
        { // Every other element
            pair1[index] = temp1[index - 1];
            pair2[index] = temp2[index + 1];
        }
    }
}
```

Jezgra se pokreće s ranije definiranim brojem blokova i dretava koja će se koristiti kroz cijelu implementaciju algoritma.

```
shiftJacobiPairs<<<blocks,threads>>>(device_pair1,device_pair2,device_temp1
,device_temp2, matrix_Device, _MATRIX_ORDER);
```

pri čemu se *threads* konfigurira, a *blocks* je jednak *(N* + threads - 1) / threads. Broj dretva (*threads*) će se konfigurirati kako bi se pronašlo optimalno izvršavanje, a on će iznositi 8,16,24 i 32 dretve.

Prije poziva jezgre alociramo sva potrebna polja i pokazivače na domaćinu i kopiramo početno stanje polja na sljedeći način:

```
int size;
if (_MATRIX_ORDER % 2 != 0)
{
    size = ((_MATRIX_ORDER + 1) / 2) * sizeof(int);
}
else
{
    size = (_MATRIX_ORDER / 2) * sizeof(int);
}
int *pair1 = (int *)malloc(size);
int *pair2 = (int *)malloc(size);

cudaMalloc((void **)&precision_Device, sizeof(double));

int *device_pair1, *device_temp1;
int *device_pair2, *device_temp2;

cudaMalloc((void **)&device_pair1, size);
cudaMalloc((void **)&device_pair2, size);
cudaMalloc((void **)&device_temp1, size);
cudaMalloc((void **)&device_temp2, size);

initializeMatrixPairs(pair1, pair2, _MATRIX_ORDER);

cudaMemcpy(device_pair1, pair1, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(device_pair2, pair2, size, cudaMemcpyHostToDevice);
cudaMemcpy(device_temp1, pair1, size, cudaMemcpyHostToDevice);
cudaMemcpy(device_temp2, pair2, size, cudaMemcpyHostToDevice);
```

8.2.2. Potrebni podaci

Kako bismo uspješno mogli krenuti na računanje svojstvenih vrijednosti najprije trebamo napraviti inicijalizaciju potrebnih varijabli na domaćinu te sukladno generiranim podacima i unesenim parametrima trebamo prebaciti sve podatke s domaćina na uređaj.

Krećemo od generiranja dviju potrebnih matrica. Prva matrica će sadržavati elemente za koje želimo naći svojstvene vrijednosti, dok će druga sadržavati vrijednosti *c* i *s* na mjestima parova koja će biti generirana pomoću *chess ordering tournament* algoritma. Matrica koja će sadržavati vrijednosti *c* i *s* je na početku generirana kao jedinična matrica.

```
int _MATRIX_ORDER = getMatrixOrder();
double **matrix_Host = symetric_matrix_double();
double **matrixOut_Host;
double **jacobiMatrix_Host = identityMatrixDouble();

double **matrix_Device;
double **jacobiMatrix_Device;

double *precision_Device;

double doubleSize = sizeof(double);
double *precision_Host = (double *)malloc(doubleSize);

/*Allocate double pointer host memory */
matrixOut_Host = (double **)malloc(_MATRIX_ORDER * sizeof(double *));
allocateMatrix(matrixOut_Host, _MATRIX_ORDER);
```

Za prijenos podataka s domaćina na uređaj koji su dvostruki pokazivači potrebna nam je petlja zajedno s pomoćnom varijablom. Takav prijenos se radi tako da prvo alociramo memoriju na uređaju pomoću petlje i pomoćnog polja. Zatim napravimo još jedno alociranje za željenu varijablu te kopiramo sadržaj privremenog polja na prethodno alociranu memoriju uređaja. Kada smo to napravili trebamo još kopirati sadržaj domaćina na uređaj ponovno pomoću jedne petlje.

```
double **tmp = (double **)malloc(_MATRIX_ORDER * sizeof(tmp[0]));
double **tmp_1 = (double **)malloc(_MATRIX_ORDER * sizeof(tmp_1[0]));

for (int i = 0; i < _MATRIX_ORDER; i++)
{
      cudaMalloc((void **)&tmp[i], _MATRIX_ORDER * sizeof(tmp[0][0]));
      cudaMalloc((void **)&tmp_1[i], _MATRIX_ORDER *sizeof(tmp_1[0][0]));
}

cudaMalloc((void **)&matrix_Device,_MATRIX_ORDER*sizeof(matrix_Device[0]));
cudaMalloc((void**)&jacobiMatrix_Device,_MATRIX_ORDER*sizeof(jacobiMatrix_D
evice[0]));

cudaMemcpy(matrix_Device, tmp, _MATRIX_ORDER * sizeof(matrix_Device[0]),
cudaMemcpyHostToDevice);
cudaMemcpy(jacobiMatrix_Device, tmp_1, _MATRIX_ORDER *
sizeof(jacobiMatrix_Device[0]), cudaMemcpyHostToDevice);

for (int i = 0; i < _MATRIX_ORDER; i++)
{
      cudaMemcpy(tmp[i],matrix_Host[i],_MATRIX_ORDER*sizeof(matrix_Device[0
      ][0]), cudaMemcpyHostToDevice);

     cudaMemcpy(tmp_1[i],jacobiMatrix_Host[i],_MATRIX_ORDER*sizeof(jacobiMa
     trix_Device[0][0]), cudaMemcpyHostToDevice);
    }
```

Navedena funkcionalnost kopiranja podataka s domaćina na uređaj je rezultat dugog proučavanja kako funkcioniraju dvodimenzionalna polja na grafičkim procesorima i memoriji koja nije dokumentirana na službenim stranicama tehnologije. Većina izvora za korištenje matrica u *CUDA* programskom jeziku su jednodimenzionalna polja kod kojih je svaki novi redak matrice spremljen u pomaku od *N* polja.

Iz razloga bolje preglednosti te jasnijeg razumijevanja daljnjim čitateljima, odlučeno je ostati na dvodimenzionalnim poljima te naći rješenje kako se vrši alokacija dvostrukih pokazivača.

Kako bismo imali dovoljan broj dretva po blokovima definiran je unaprijed fiksni broj dretva (može se konfigurirati), dok se blokovi određuju prema redu matrice. U nekim dijelovima

programa potrebna nam je druga dimenzija te iz tog razloga uvodimo novu dimenziju i njih definiramo na sljedeći način:

```
int threads = 32;
int blocks = (_MATRIX_ORDER + threads - 1) / threads;
dim3 THREADS(threads, threads);
dim3 BLOCKS_COLS(blocks / 2 + 1, blocks);
dim3 BLOCKS_ROWS(blocks, blocks / 2 + 1);
dim3 BLOCKS(blocks, blocks);
```

Upotreba novo uvedenih dimenzije bit će objašnjena kroz poziv jezgre u nastavku rada.

8.2.3. Izračun matrice transformacija

Kao što smo ranije naveli potrebna nam je matrica koja će sadržavati izračunate vrijednosti *c* i *s* za parove koji se ne preklapaju. Za dohvat parova definiramo *jezgru* compute*JacobiMatrix* koji će imati 32 dretve po bloku, dok je izračun blokova ranije naveden i ovisi o redu matrice.

```
computeJacobiMatrix<<<blocks, threads>>>(matrix_Device, jacobiMatrix_Device,
device_pair1, device_pair2, _MATRIX_ORDER, device_temp1, device_temp2);
```

Kao parametre prosljeđujemo početnu matricu ili, ako korak nije prvi, ulaznu matricu koja je već transformirana i konvergira u gornjem trokutu u nulu. Drugi parametar nam je matrica u koju ćemo zapisati izračunate vrijednosti. Sljedeća dva parametra su *device_pair1* i *device_pair2* koji sadrže indekse elemenata koji se mogu izvršavati u paraleli, red matrice te zadnja dva parametra su pomoćna polja.

```
__global__ void computeJacobiMatrix(double **inputMatrix, double
**jacobiMatrix, int *pair1, int *pair2, int matrix_order, int *pair1_temp,
int *pair2_temp)
{
    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int size = matrix_order;
    if (matrix_order % 2 != 0)
    {
        size++;
    }
    if (index < size / 2)
    {
        int i = pair1[index];
```

```
        int j = pair2[index];

        pair1_temp[index] = pair1[index];
        pair2_temp[index] = pair2[index];

        if (i > j)
        {
            int temp = i;
            i = j;
            j = temp;
        }

        if (i != -1 && inputMatrix[i][j] != 0)
        {
            double b = (inputMatrix[i][i] - inputMatrix[j][j]) / (2.0 *
            inputMatrix[i][j]);

            double t = ((b > 0) ? 1.0 : ((b < 0) ? -1.0 : 0.0)) /
            ((fabs(b)) + sqrt(pow(b, 2) + 1.0));

            double c = 1.0 / (sqrt(pow(t, 2) + 1.0));
            double s = c * t;

            jacobiMatrix[i][j] = s;
            jacobiMatrix[j][i] = -s;
            jacobiMatrix[j][j] = c;
            jacobiMatrix[i][i] = c;
        }
    }
}
```

Najprije počinjemo s dohvaćanjem indeksa dretve koja se izvršava u paraleli. On će nam poslužiti za dohvat parova koji se ne ometaju. Dakle, moramo raditi provjeru da je taj indeks dretve manji od *N/2*. Navedenu provjeru radimo iz razloga jer se ne pokreće točan broj dretvi koji bi trebao za izvršavanje. Način na koji se radi izračun dretvi i blokova je konfiguriran tako da se mogu izračunati svojstvene vrijednosti za matrice koju su veće od 1024. Možemo reći da iz konfiguracije proizlazi uvjet za provjeru, iako je isprobana verzija kada bi se jezgra pokretala s *N/2* blokova, međutim za rezultat daje lošije vrijeme. U slučaju neparnog reda matrice imamo *(N+1)/2* parova te iz tog razloga prvo radimo provjeru je li to slučaj. Ovdje

također iskorištavamo ovu jezgru za kopiranje trenutnih parova indeksa u pomoćna polja kako ih ne bi trebali kopirati pozivom nove jezgre. Kada smo dohvatili indeks dretve, dohvaćamo indeks parova te radimo provjeru je li broj *i* veći od broja *j*. Indeks *i* bi u teoriji trebao biti manji od indeksa *j* zato što bi inače dobili krive parove od kojih bi zapravo indeks *j* predstavljao prvi element koji je na slici 8 vidljiv unutar crvenog pravokutnika. Ako je to slučaj trebamo im zamijeniti mjesta. Takav slučaj proizlazi iz *chess ordering tournament* algoritma koji radi spajanje svih parova, ali ih kao takve ne sortira.

Ako imamo sve potrebno iz ulazne matrice možemo izračunati vrijednosti *c* i *s*, na način opisan u poglavlju *Jacobijevog algoritma,* pa zatim te vrijednosti upisati na odgovarajuća mjesta u *Jacobijevu matricu (J)*. Tim potezom smo uspješno zapisali *c* i *s* za *N/2* parova koji se ne preklapaju u paraleli te imamo potrebne podatke za sljedeći korak.

8.2.4. Transformacija redova i stupaca

Nakon izračuna matrice transformacija sljedeći nam je korak da iskoristimo tu matricu transformacija za promjenu svih redova i stupaca.

Prvotna verzija ovog koraka je uključivala množenje triju matrica pozivom jezgre koja bi prvim pozivom množila matrice $J^T * A = TempA$ te bi zatim se novim pozivom iste jezgre pomnožila novo dobivena matrica $TempA * J = A$ kako bismo zadovoljili uvjet *Jacobijevog algoritma* $A_{i+1} = J^T A_i J$. To množenje matrica je bilo implementirano uz pomoć uvođenja rutine koja je koristila dvije dimenzije za izračun.

```
__global__ void multiplyMatrix(double **A_d, double **B_d, double **C_d, int
matrix_order)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;

    if ((i < matrix_order) && (j < matrix_order))
    {
        double tmp = 0;
        for (int z = 0; z < matrix_order; z++)
        {
            tmp = tmp + A_d[i][z] * B_d[z][j];
        }
        C_d[i][j] = tmp;
    }
```

```
}
```

Takav pristup množenju matrica je bio potpuno optimiziran, ali u našem slučaju nije bilo potrebno raditi množenje cijelih matrica iz razloga zato što se kod pojedine *Jacobijeve* transformacije mijenjaju samo dva reda matrice, a drugim množenjem dva stupca.

Iz toga razloga napravljene su dvije nove jezgre koje će u prvom pozivu promijeniti redove unesene matrice, a drugim pozivom promijeniti stupce. Indeksi stupaca i redova koji se trebaju promijenit spremljeni su u polja, a izračunati su pomoću *chess ordering tournament* algoritma te su oni kao takvi proslijeđeni metodama. Oba poziva jezgre izgledaju slično samo što svaka metoda ima definiranu svoju veličinu i dimenziju dretva za pokretanje.

Poziv jezgre za promjene redova i stupaca:

```
updateRows<<<BLOCKS_ROWS, THREADS>>>(matrix_Device, jacobiMatrix_Device, device_pair1, device_pair2, _MATRIX_ORDER);
updateColumns<<<BLOCKS_COLS, THREADS>>>(matrix_Device, jacobiMatrix_Device, device_pair1, device_pair2, _MATRIX_ORDER);
```

Možemo vidjeti da su ulazi identični. Prvi parametar je matrica na kojoj se vrši transformacija, drugi je matrica transformacija, sljedeća dva parametra su polja indeksa parova, dok je zadnji red matrice. Također možemo primijetiti s koliko blokova i dretvi pokrećemo jezgru tj. u kojoj dimenziji. Potrebne su nam zapravo dvije dimenzije od kojih jedna dimenzija ima *N/2* dretava. Navedenih *N/2* dretava prve dimenzije se koristi za dohvat indeksa, dok druga dimenzija služi za dohvat elemenata u matrici kojih ima *N* po stupcu/retku.

```
__global__ void updateRows(double **matrix_A, double **jacobi_matrix, int *pair_1, int *pair_2, int matrix_order)
{
    int index = threadIdx.y + blockDim.y * blockIdx.y;
    int k = threadIdx.x + blockDim.x * blockIdx.x;

    int size = matrix_order;
    if (matrix_order % 2 != 0)
    {
        size++;
    }
    if (index < size / 2)
    {
        int i = pair_1[index];
```

```
        int j = pair_2[index];

        if (i > j)
        {
            int temp = i;
            i = j;
            j = temp;
        }
        if (i < matrix_order && j < matrix_order && i != -1 &&
matrix_A[i][j] != 0)
        {

            double s = jacobi_matrix[j][i];
            double c = jacobi_matrix[j][j];

            if (k < matrix_order)
            {
                double mik = matrix_A[i][k];
                double mjk = matrix_A[j][k];

                matrix_A[i][k] = (c * mik) - (s * mjk);
                matrix_A[j][k] = (s * mik) + (c * mjk);
            }
        }
    }
}
```

Prvo pozivamo metodu za promjenu redova gdje nam dretve dimenzije *y* označavaju indeks u poljima za parove kojih ima *N/2*. Dretve druge dimenzije iskoristiti ćemo za pristup stupcima koje treba promijeniti. Za razliku od serijske verzije gdje se pristup određenom redu i određenom stupcu vrši kroz dvije petlje, ovdje je te dvije petlje zamijenila dimenzija *y* zato što dohvatom određenog para znamo pomoću kojeg elementa u matrici trebamo napraviti transformaciju redova. U serijskoj verziji, kada znamo pomoću kojeg elementa radimo transformaciju dva reda, pokrećemo još jednu petlju koja će raditi transformaciju svih elemenata u dva prikladna reda. Dimenzija *x* zamjenjuje tu petlju te se pomoću indeksa dretvi radi transformacija na odgovarajućim redovima. Iskorištene su obje dimenzije kako bi se smanjilo vrijeme izračuna svojstvenih vrijednosti. Rješenje se moglo postići samo s jednom

dimenzijom, ali bi onda trebalo definirati petlju unutar *jezgre* što bi utrošilo puno vremena. Naime, svako definiranje petlje unutar jezgre nam produžuje vrijeme za *O(n)*.

Metoda koja radi ažuriranje stupaca je zapravo identična samo što se radi zamjena indeksa *k* te indeksa *j,i* kako bismo pristupili elementima stupaca, a ne redova. Također je zbog efikasnosti napravljeno da dimenzija *y* dohvaća redove, a dimenzija *x* predstavlja indeks u poljima za parove. Nakon prve iteracije dobili smo nule na *N/2* mjesta što možemo vidjeti na slici niže. Također možemo vidjeti kako su stupci jednaki redovima na odgovarajućim mjestima tj. tako transformirana matrica je i dalje simetrična.

```
Matrix order: 6


After one iteration:
0.754412 0.237079 0.553102 -0.000000 1.165463 1.063630
0.237079 -0.718518 0.145662 0.070164 -0.000000 0.492524
0.553102 0.145662 -0.097628 -0.085292 0.306310 0.000000
-0.000000 0.070164 -0.085292 -0.416007 0.179716 0.100187
1.165463 -0.000000 0.306310 0.179716 0.906950 0.593441
1.063630 0.492524 0.000000 0.100187 0.593441 1.142503

CPU time : 0.000385
Real time: 0.001772
```

Slika 12: Jedan korak iteracije (Izvor: Vlastita izrada)

8.2.5. Provjera uvjeta

Kako bismo znali je li algoritam gotov, u paraleli pokrećemo novu jezgru koja računa sumu kvadrata gornjeg trokuta matrice. Navedena jezgra je optimizirana tako da koristi novu dimenziju za dohvat elemenata iznad dijagonale bez petlje.

```
sumOffset<<<BLOCKS,THREADS>>>(matrix_Device,precision_Device,_MATRIX_ORDER;
```

Implementacija jezgre je sljedeća:

```
__global__ void sumOffset(double **matrix, double *precision, int matrix_order)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col < matrix_order && row < col)
```

```
        {
            double mij = matrix[row][col];
            atomicAdd(precision, pow(mij, 2));
        }
    }
```

Pomoću druge dimenzije dohvaćamo sve elemente koji se nalaze iznad dijagonale. Element koji je dohvaćen se nadodaje u varijablu *precision* na način da se prvo kvadrira kako bismo dobili apsolutnu vrijednost. Način na koji radimo sumiranje elemenata je pomoću ugrađene *CUDA* metode *atomicAdd()* koja dozvoljava dretvama da u isto vrijeme rade promjene nad varijablama. Ako koristimo notaciju `precision += pow(mij,2)` dobiti ćemo krive rezultate. Metoda *atomicAdd()* na trenutno instaliranoj verziji i arhitekturi ne podržava tip podataka *double* kao ulaz. Problem se može riješiti ako se kod prevođenja programa dodaje zastavica *-arch=sm_60*. Verzijom *CUDA* 8 nadograđena je metoda za arhitekturu *SM_6X (*Pascal) *GPU* te sada podržava za ulaz tip *double*, ali se i dalje prevođenje programa mora raditi sa zastavicom.

Metoda se po analogiji sa serijskom verzijom poziva svakih N-1 iteracija.

```
if (iterations % (_MATRIX_ORDER - 1) == 0)
{
    double zero = 0.0;
    cudaMemcpy(precision_Device,&zero,doubleSize,cudaMemcpyHostToDevice);

    sumOffset<<<BLOCKS,THREADS>>>(matrix_Device,precision_Device,_MATRIX_
    ORDER);
    cudaDeviceSynchronize();
    cudaMemcpy(precision_Host,precision_Device,doubleSize,cudaMemcpyDevic
    eToHost);
}
```

Prije pokretanja jezgre resetiramo varijablu *precision_device*, a nakon izračuna kopiramo rezultat natrag na uređaj pomoću *cudaMemcpy()* metode.

8.2.6. Poziv jezgri

Poziv jezgri se odvija unutar *while* petlje koja završava uvjetom da je suma kvadrata iznad ili ispod dijagonale približno jednaka nuli.

```
do
{
	computeJacobiMatrix<<<blocks,
	threads>>>(matrix_Device,jacobiMatrix_Device, device_pair1,
	device_pair2, _MATRIX_ORDER, device_temp1, device_temp2);
	cudaDeviceSynchronize();

	updateRows<<<BLOCKS_ROWS, THREADS>>>(matrix_Device,
	jacobiMatrix_Device, device_pair1, device_pair2, _MATRIX_ORDER);
	cudaDeviceSynchronize();

	updateColumns<<<BLOCKS_COLS, THREADS>>>(matrix_Device,
	jacobiMatrix_Device, device_pair1, device_pair2, _MATRIX_ORDER);
	cudaDeviceSynchronize();

	shiftJacobiPairs<<<blocks, threads>>>(device_pair1, device_pair2,
	device_temp1, device_temp2, matrix_Device, _MATRIX_ORDER);
	cudaDeviceSynchronize();

	if (iterations % (_MATRIX_ORDER - 1) == 0)
	{
		double zero = 0.0;
		cudaMemcpy(precision_Device, &zero, doubleSize,
		cudaMemcpyHostToDevice);
		sumOffset<<<BLOCKS, THREADS>>>(matrix_Device, precision_Device,
		_MATRIX_ORDER);
		cudaDeviceSynchronize();
		cudaMemcpy(precision_Host, precision_Device, doubleSize,
		cudaMemcpyDeviceToHost);
		printf("Offset(A) : %6.20f\n", sqrt(*precision_Host));
	}
```

```
        iterations++;
} while (sqrt(*precision_Host) > 1e-20);
```

Između poziva jezgri možemo uočiti poziv još jedne metode imena *cudaDeviceSynchronize()*. Navedenu metodu pozivamo u slučaju kada želimo pričekati da prethodna jezgra završi te tek tada pokrenemo novu. Poziv metode se uključuje u program jer izvršavanje jezgre je asinkrono tj. njihovim pozivima se ne blokiraju daljnja izvršavanja u programu. Neki izvori tvrde da se sljedeća jezgra u nizu pozivom stavlja u red čekanja te se prilikom završetka prethodne pokrene nova, dok službeni izvor savjetuje za upotrebu sinkronizacije između poziva jezgara.

8.2.7. Ispis rezultata i oslobađanje memorije

Kada navedeni algoritam završi uspješno smo dobili rezultat koji je pohranjen u početnoj matrici te se vrijednosti nalaze na glavnoj dijagonali. Za sada je ta memorija na uređaju pa je trebamo prebaciti prvo na domaćina na isti način kako smo i radili kopiranje memorije dvostrukih pokazivača uz pomoćno polje.

```
    double **tmp_2 = (double **)malloc(_MATRIX_ORDER * sizeof(tmp_2[0]));
    for (int i = 0; i < _MATRIX_ORDER; i++)
    {
        cudaMalloc((void **)&tmp_2[i], _MATRIX_ORDER * sizeof(tmp_2[0][0]));
    }

    cudaMemcpy(tmp_2, matrix_Device, _MATRIX_ORDER * sizeof(matrix_Device[0]), cudaMemcpyDeviceToHost);
    for (int i = 0; i < _MATRIX_ORDER; i++)
    {
        cudaMemcpy(matrixOut_Host[i], tmp_2[i], _MATRIX_ORDER * sizeof(matrix_Device[0][0]), cudaMemcpyDeviceToHost);
    }
```

Kopiranjem memorije na domaćina uspješno smo završili zadatak računanja svojstvenih vrijednosti pomoću paralelnog algoritma. Prije nego što završimo s programom moramo očistiti memoriju adekvatno tome kako smo je i alocirali.

Za dvostruke pokazivače trebamo prvo očistiti sva pomoćna polja kroz petlju.

```
    for (int i = 0; i < _MATRIX_ORDER; i++)
    {
```

```
    cudaFree(tmp[i]);
    cudaFree(tmp_1[i]);
    cudaFree(tmp_2[i]);
}
```

A svu ostalo memoriju očistimo tako samo da pozovemo *free(),* tj. *cudaFree()*

```
cudaFree(tmp);
cudaFree(tmp_1);
cudaFree(tmp_2);

free(*matrix_Host);
free(*matrixOut_Host);
free(*jacobiMatrix_Host);

free(pair1);
free(pair2);

free(matrix_Host);
free(matrixOut_Host);
free(jacobiMatrix_Host);

cudaFree(precision_Device);
cudaFree(device_pair1);
cudaFree(device_temp1);
cudaFree(device_pair2);
cudaFree(device_temp2);

cudaFree(matrix_Device);
cudaFree(jacobiMatrix_Device);
```

Nakon uspješnog oslobađanja memorije program se može prekinuti.

9. Usporedba

U ovom poglavlju radit će se usporedba svih prethodno opisanih implementacija. Putem tabličnog prikaza zajedno s grafičkim, vidjet ćemo koji od algoritama daje najbolje vrijeme za određeni red matrice.

Prilikom izvršavanja programa veliku ulogu imaju performanse računala. Omjeri između serijske i paralelne verzije mogu dati velike razlike vremena u odnosu na međusobno izvršavanje, a sve ovisi o konfiguraciji računala na kojem se izvršavaju. Konfiguracija računala na kojem su obavljena mjerenja je sljedeća:

- Operacijski sustav:
  - *Windows* 11 Pro 64-bit u kojemu je instaliran *WSL* opisan u poglavlju 2.
- *CPU*
  - AMD *Ryzen 5 4600H with Radeon Graphics*, 3.00 GHz
  - 6 jezgri
  - 12 dretvi ili logičkih procesora
  - L2 cache: 3MB
  - L3 cache: 8MB
- *RAM*
  - 16GB
  - DDR4-3200 (1600MHz)
- *GPU*
  - *NVIDIA* GeForce GTX 1650
  - *VRAM*: 4GB GDDR5
  - *VRAM* brzina: 8000Gbps
  - Veličina sabirnice: 128-bit
  - Širina pojasa (engl. *Bandwidth*): 128GB/s
  - *CUDA* jezgre: 896
  - Bazni sat (engl. *Base clock*): 1485MHz
  - Za ostalu *CUDA* specifikaciju pogledati sliku 5

Paralelnu implementaciju *Jacobijevog* algoritma podijelit ćemo na više dijelova. Podjela će biti napravljena na način da ćemo promijeniti broj dretva koji se pokreće po jednom bloku. Broj dretva će biti: 8^2, 16^2, 24^2 i 32^2. Maksimalan broj dretva u našem slučaju je 32 koji proizlazi iz fizičke barijere uređaja na kojem se izvršava program te verzije *CUDA* programskog modela.

Veličina koja je dozvoljena u našem slučaju je 1024 dretve, te kroz našu konfiguraciju kada definiramo novu dimenziju zauzimamo točno toliko resursa.

```
dim3 THREADS(threads,threads)
```

Iz programskog koda iznad dobivamo da je $32^2 = 1024$. Broj blokova se određuje prema formuli:

```
blocks = (N + threads -1) / threads
```

Izračun broja blokova i dretava je isti kod svakog reda matrice te je tako definiran da bi se mogle izračunati svojstvene vrijednosti za red matrice veći od 1024.

Simbologija:

- N → Red matrice
- sjacsy → Jacobi serijska verzija
- pjacsy8 → Jacobi paralelna verzija s 8 dretva po bloku
- pjacsy16 → Jacobi paralelna verzija sa 16 dretva po bloku
- pjacsy24 → Jacobi paralelna verzija s 24 dretve po bloku
- pjacsy32 → Jacobi paralelna verzija s 32 dretve po bloku
- LDSYEV → LAPACK rutina DSYEV

Vremena su iskazana u sekundama.

Tablica 1: Usporedba vremena

N	sjacsy	pjacsy8	pjacsy16	pjacsy24	pjacsy32	LDSYEV
6	0.000201	0.001726	0.001866	0.001898	0.001689	0.000030
8	0.000287	0.002787	0.001773	0.002848	0.002686	0.000025
16	0.000836	0.004234	0.003814	0.007353	0.006625	0.000029
32	0.003613	0.015351	0.007787	0.015042	0.009094	0.000058
64	0.018093	0.028620	0.017022	0.017594	0.030000	0.000217
128	0.121071	0.053058	0.044031	0.054187	0.068352	0.001070
256	0.917536	0.124783	0.107085	0.116034	0.129344	0.006374
512	7.236952	0.364611	0.322662	0.480717	0.349803	0.042223
1024	268.667739	3.563670	3.310981	3.737042	3.067069	0.331653
2048	2915.481633	29.045720	28.517810	32.467522	26.970892	2.409138
4096	26260.121564	286.706526	245.780152	283.582742	221.093344	17.446649

(Izvor: Vlastita izrada)

Kao što možemo vidjeti iz tablice 1 najbolje vrijeme ima rutina programskog paketa *LAPACK* koja koristi *QR algoritam* za izračun svojstvenih vrijednosti.

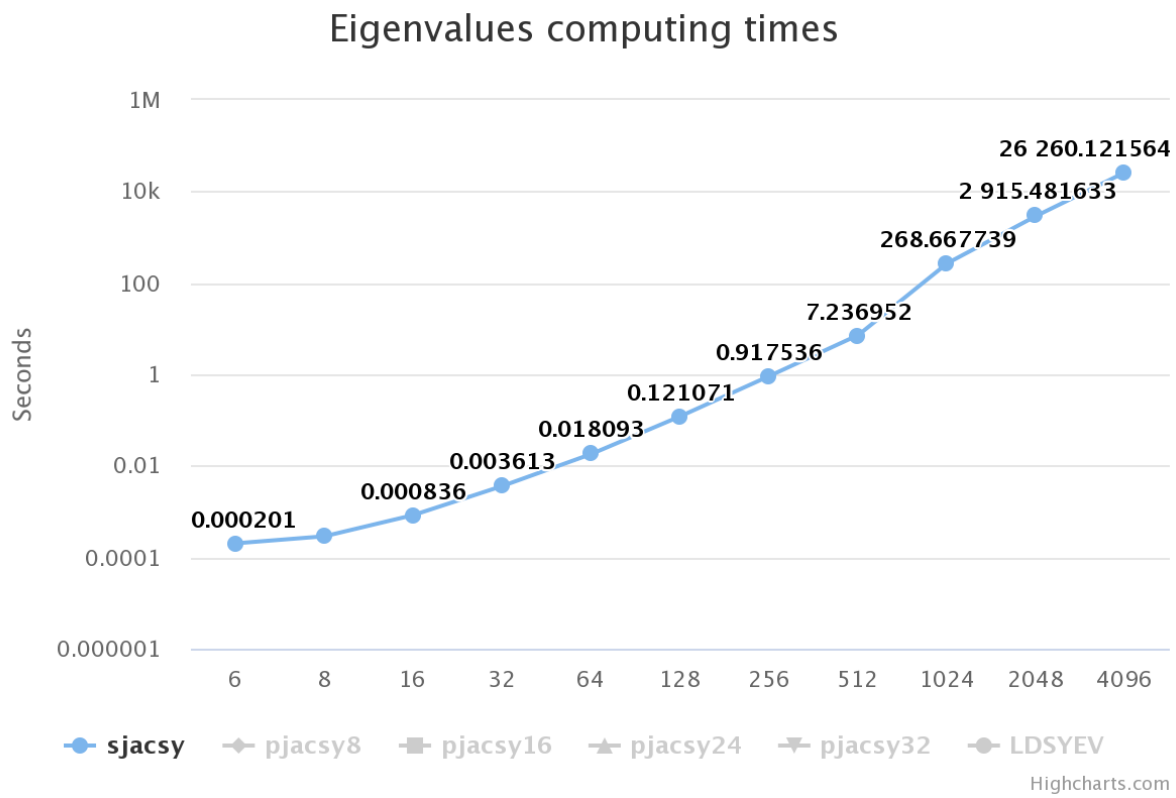
Povećanjem reda matrice se povećava kompleksnost *Jacobijevog algoritma* dok kod *QR algoritma* to nije slučaj te je iz tog razloga *LAPACK* rutina brža kada se povećava red matrice. Sljedeće jednadžbe opisuju kompleksnost obaju algoritma gdje je *n = red matrice* [17]:

$$O_{QR}(n) = 9n^3 + (30n + n + 6n^2) * (broj\ iteracija)$$

$$O_{JAC}(n) = (broj\ prolaza) * (6n^3 + 81n^2 - 87n)$$

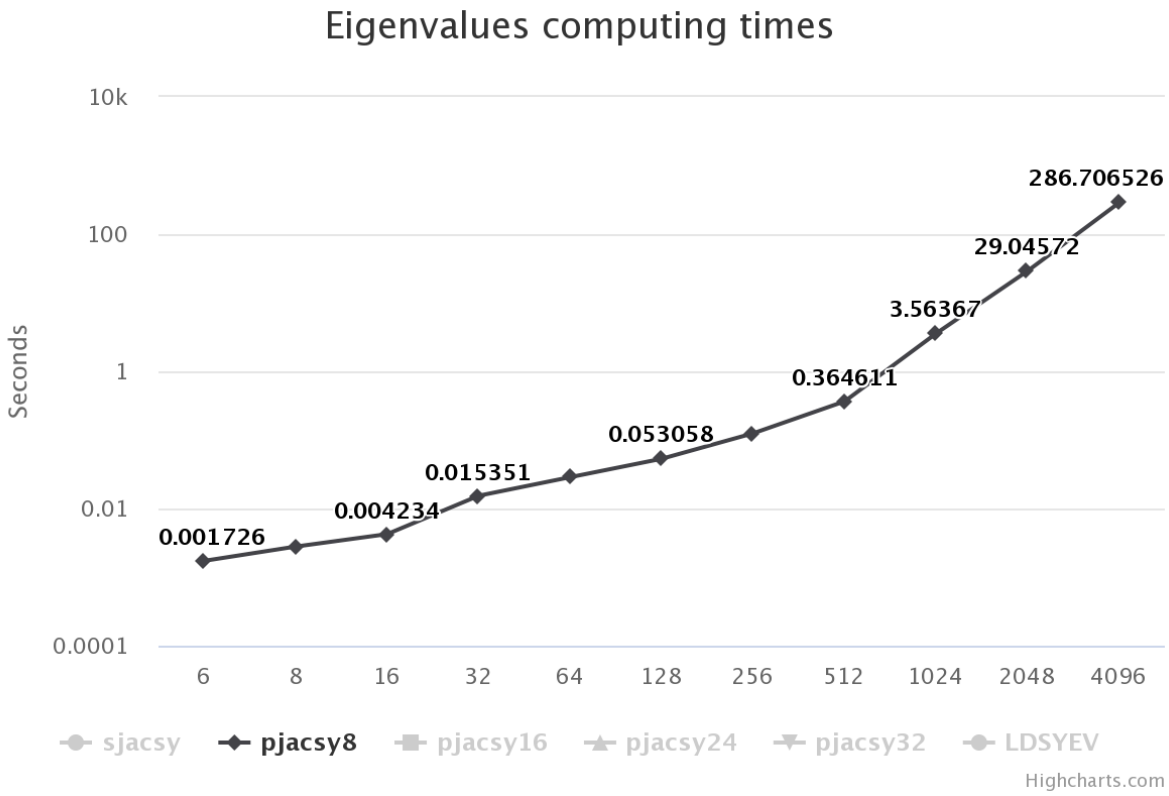
Broj prolaza lagano raste ovisno o veličini matrice, odnosno što je broj *n* veći. Pa tako imamo za red matrice 512 broj prolaza je 11, za 1024 je 12, a za 2048 i 4096 je 13.

Ovisnost vremena izvršavanja o redu matrice kod serijskog *Jacobijevog algoritma* se može vidjeti na grafu niže gdje se vrijeme povećava najmanje s trećom potencijom reda matrice pa je stoga prikazano pomoću logaritamske skale na y-osi.

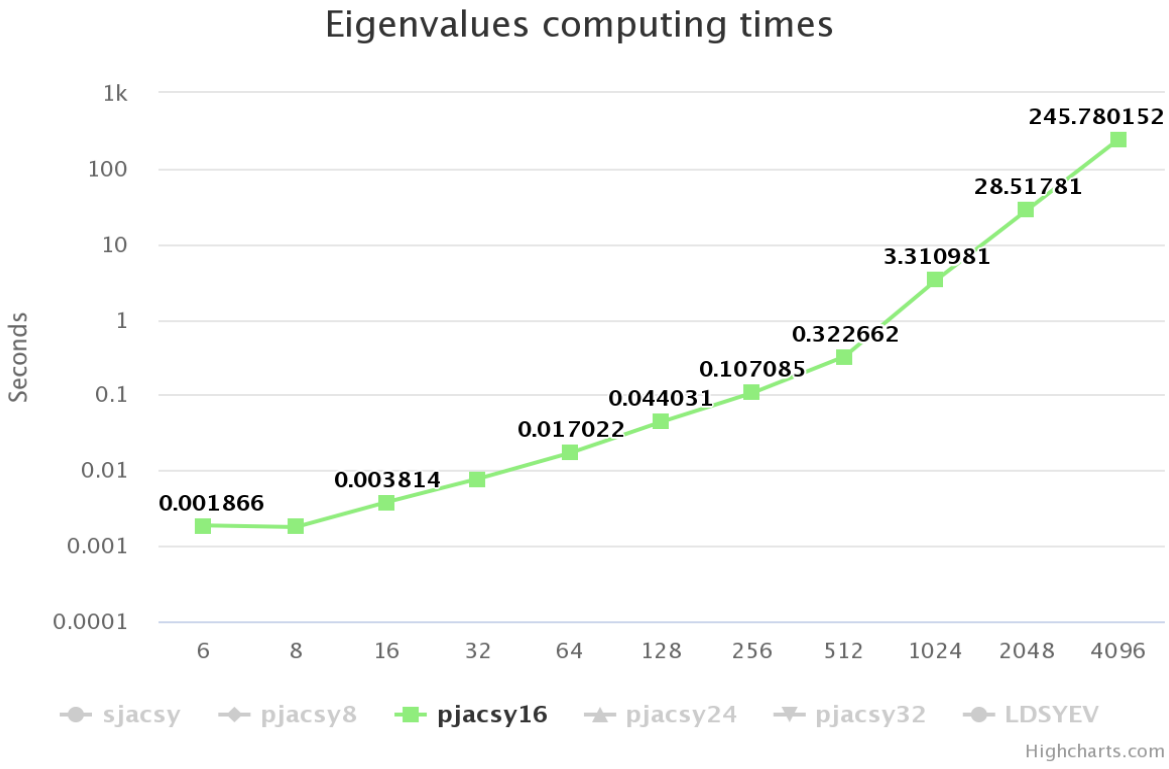


Slika 13: sjacsy (Izvor: Vlastita izrada)

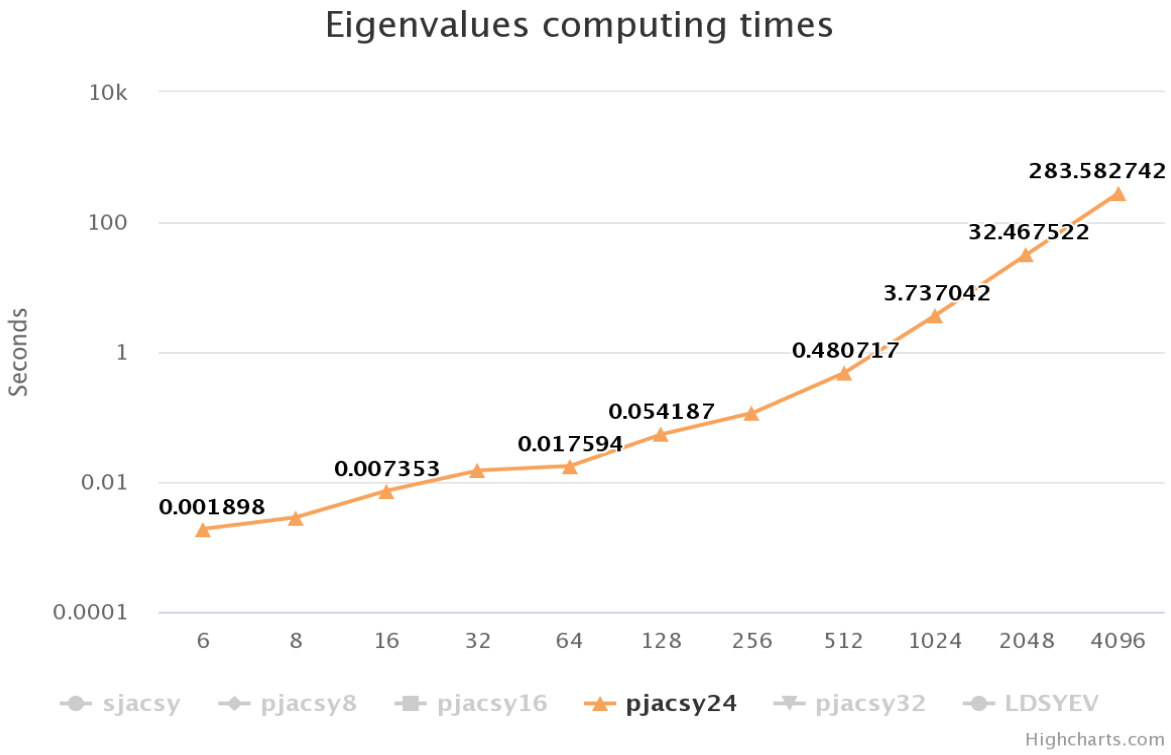
Za razliku od serijske verzije *Jacobijevog* algoritma, paralelna verzija je nešto brža kada red matrice bude veći od 64. U nastavku su prikazani grafovi za paralelnu implementaciju *Jacobijevog algoritma* koji se pokreću s različitim brojem dretva.



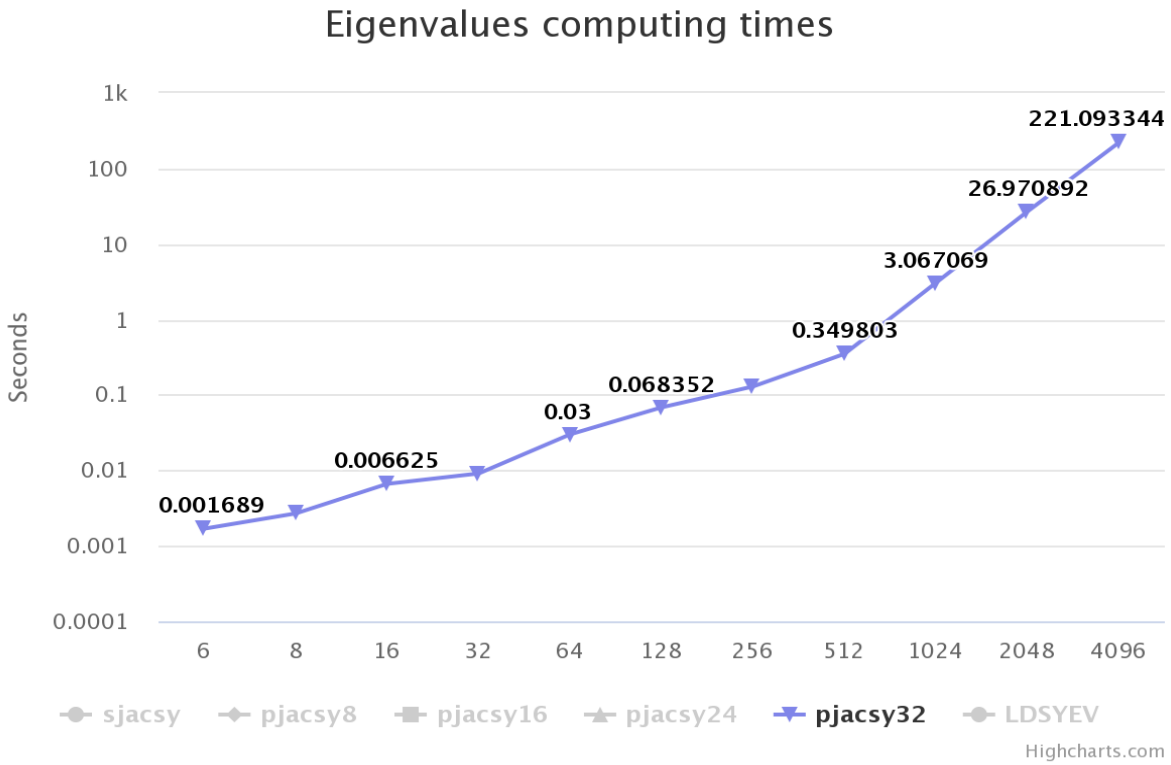
Slika 14: PJACSY8 (Izvor: Vlastita izrada)



Slika 15: pjacsy16 (Izvor: Vlastita izrada)

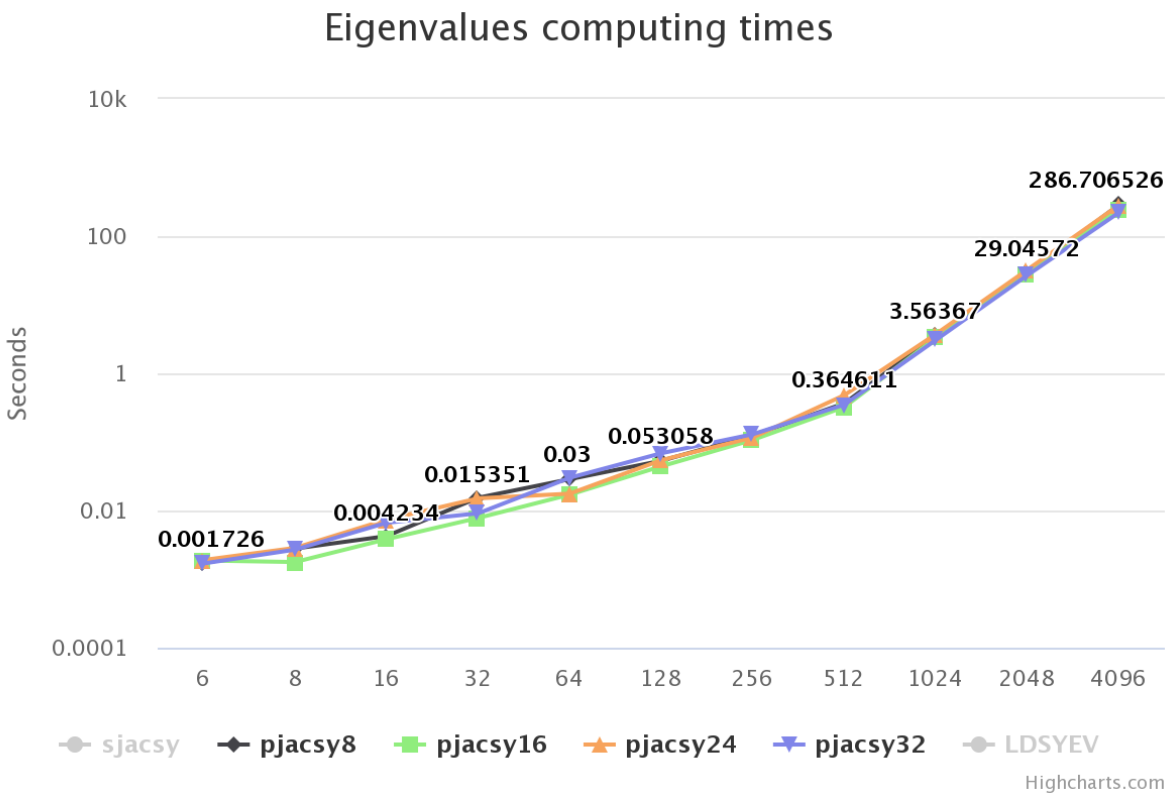


Slika 16: pjacsy24 (Izvor: Vlastita izrada)



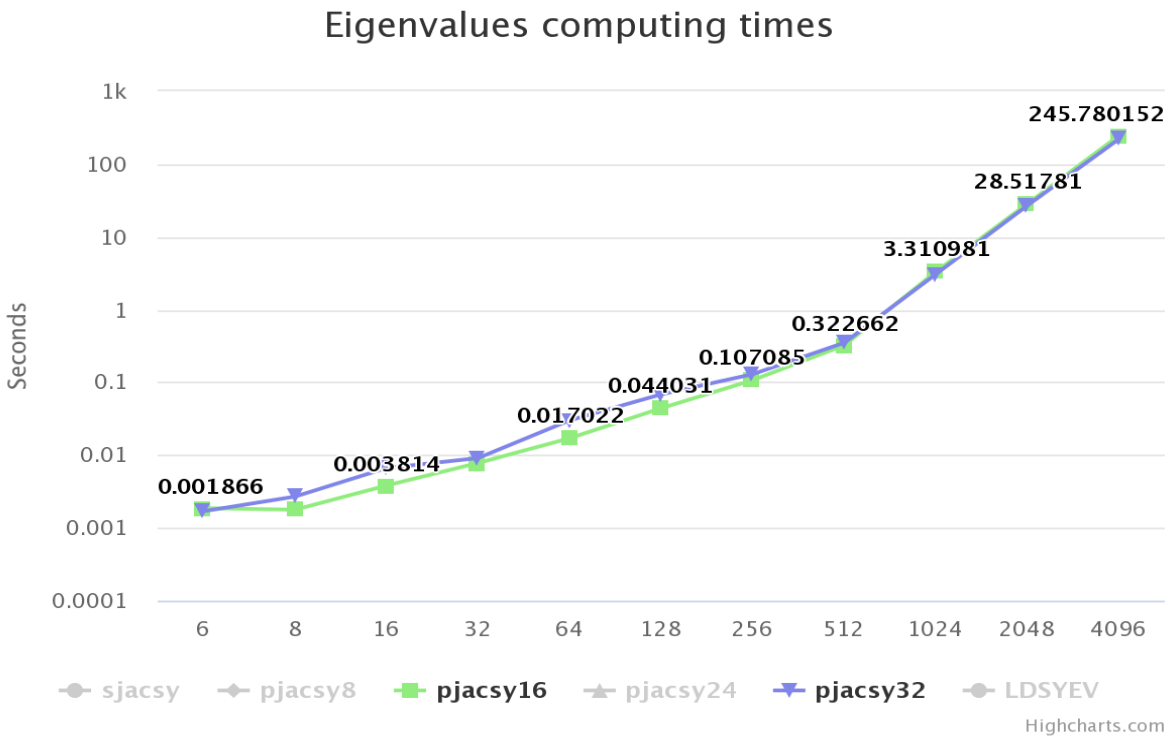
Slika 17: pjacsy32 (Izvor: Vlastita izrada)

Tri različite konfiguracije paralelnog algoritma daju slične rezultate vremena, iako se tu ističe konfiguracija od 32 dretve po bloku (pjacsy32).



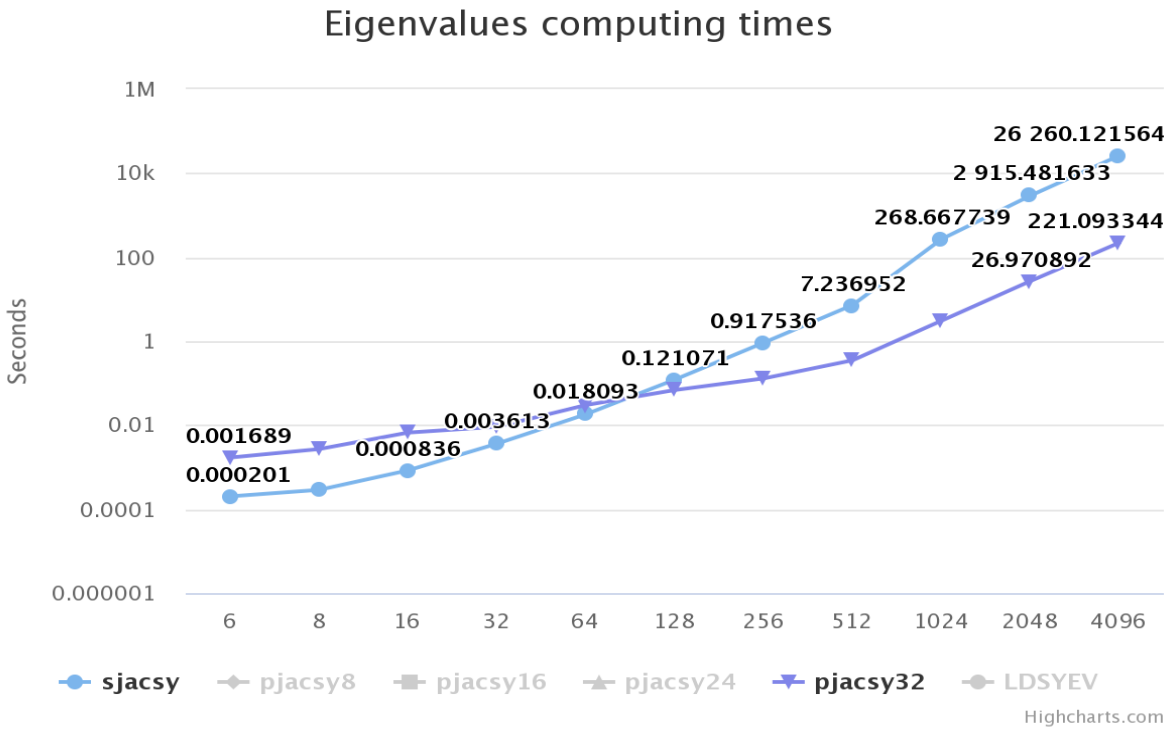
Slika 18: Usporedba paralelnih algoritma različitih konfiguracija (Izvor: Vlastita izrada)

Prije reda matrice 512 vremena paralelnih implementacija variraju za sitne intervale, dok se poslije reda matrice 512 može vidjeti kako *pjacsy32* daje nešto bolja vremena. Za taj prikaz su uzete dvije najbolje konfiguracije *pjacsy16* i *pjacsy32*.



Slika 19: Usporedba paralelnih algoritama pjacsy16 s pjacsy32 (Izvor: Vlastita izrada)

Pošto najbolje rezultate daje verzija s 32 dretve ona će biti uspoređena sa serijskom verzijom *Jacobijevog* algoritma.

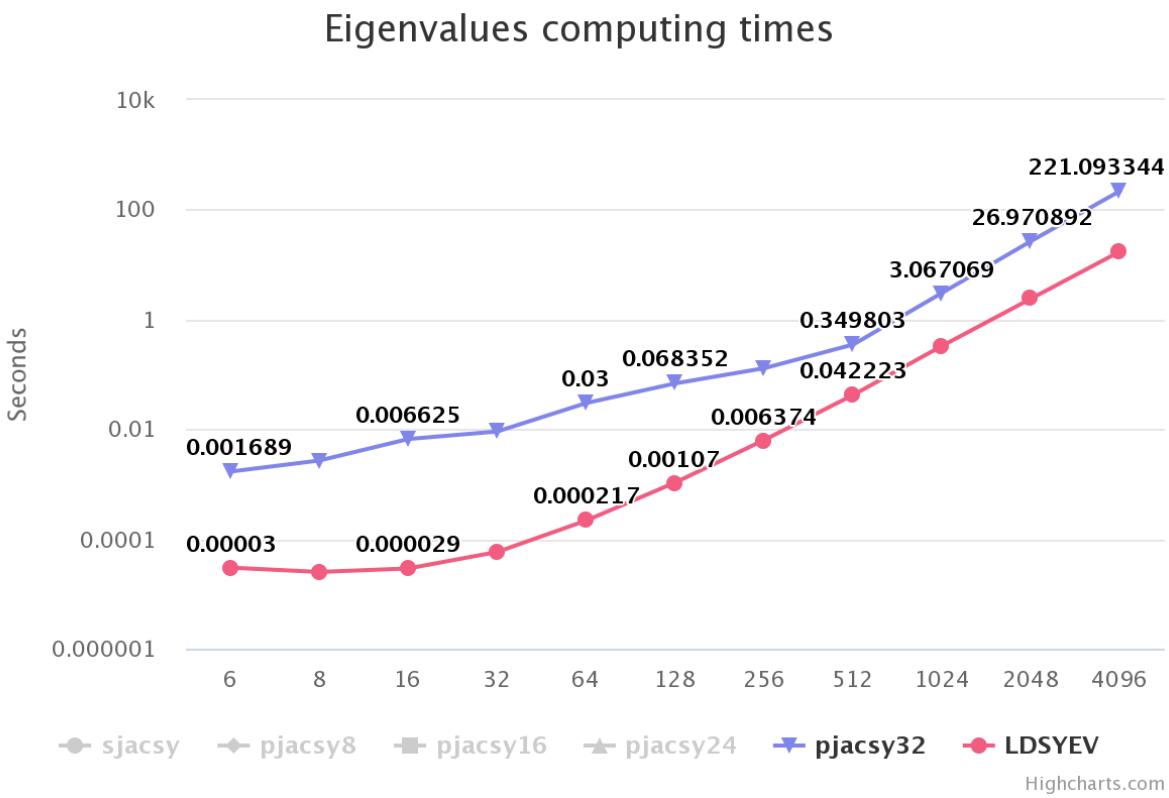


Slika 20: Usporedba sjacsy s pjacsy32 (Izvor: Vlastita izrada)

Zanimljivo je vidjeti kako na manjim matricama serijska verzija daje bolje rezultate iako su te razlike jako male. Nešto veće vrijeme za paralelnu verziju se može opravdati samom strukturom *CUDA* modela. Svaki poziv nove jezgre, kopiranje memorije i preraspodjela procesa na uređaju troši vrijeme koje je zapravo jako malo, ali se na ovom grafu vidi kako na manjim redovima matrice čini jedan dobar dio izvršavanja programa. Povećanjem reda matrice suma svih poziva je zanemariva. Navedeno se najbolje vidi prilikom prelaska reda matrice sa 64 na red matrice 128 kada je vrijeme sa serijskom verzijom gotovo identično.

Kako se povećava red matrice tako paralelna verzija preuzima vodstvo nad serijskom verzijom. Razlike su već zamjetnije te se broje u sekundama. Za primjer možemo pogledati graf na slici 25 za red matrice 2048 kada serijskoj verziji treba približno 50 minuta, a paralelnoj verziji samo 26 sekundi. Iz navedenog možemo zaključiti da je paralelna verzija sto puta brža.

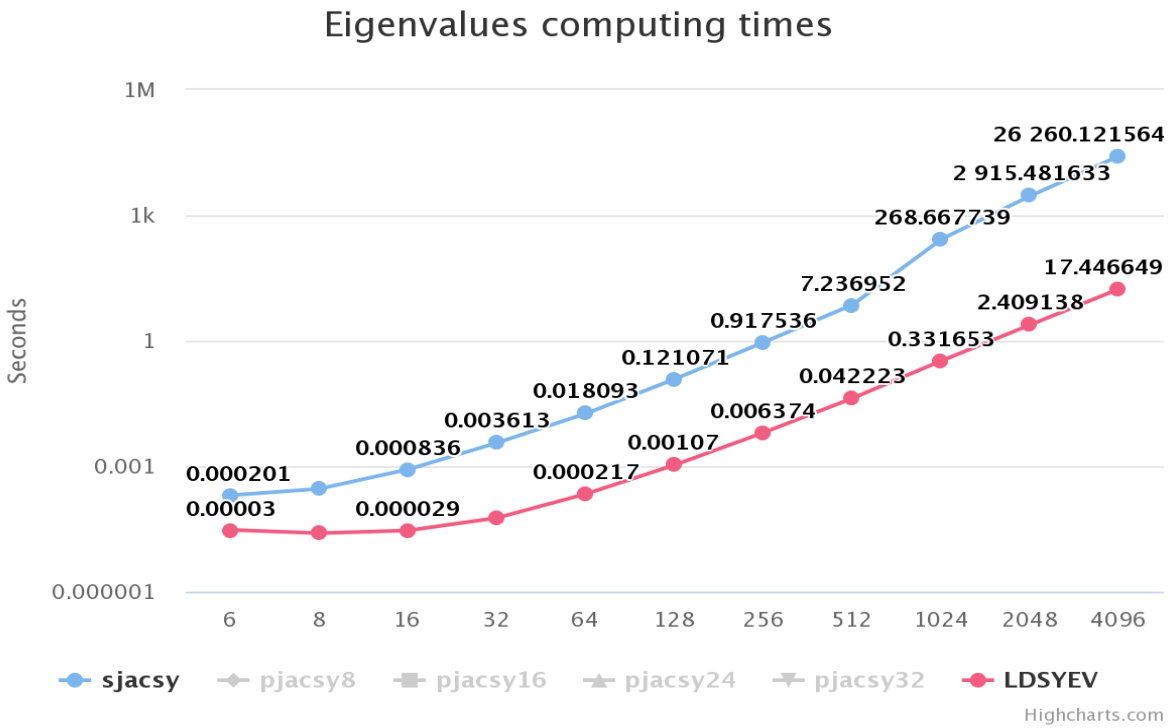
Sljedeća usporedba je paralelna verzija *Jacobijevog algoritma* (pjacsy32) s *LAPACK* rutinom *DSYDEV* (LDSYEV).



Slika 21: Usporedba pjacsy32 s LDSYEV (Izvor: Vlastita izrada)

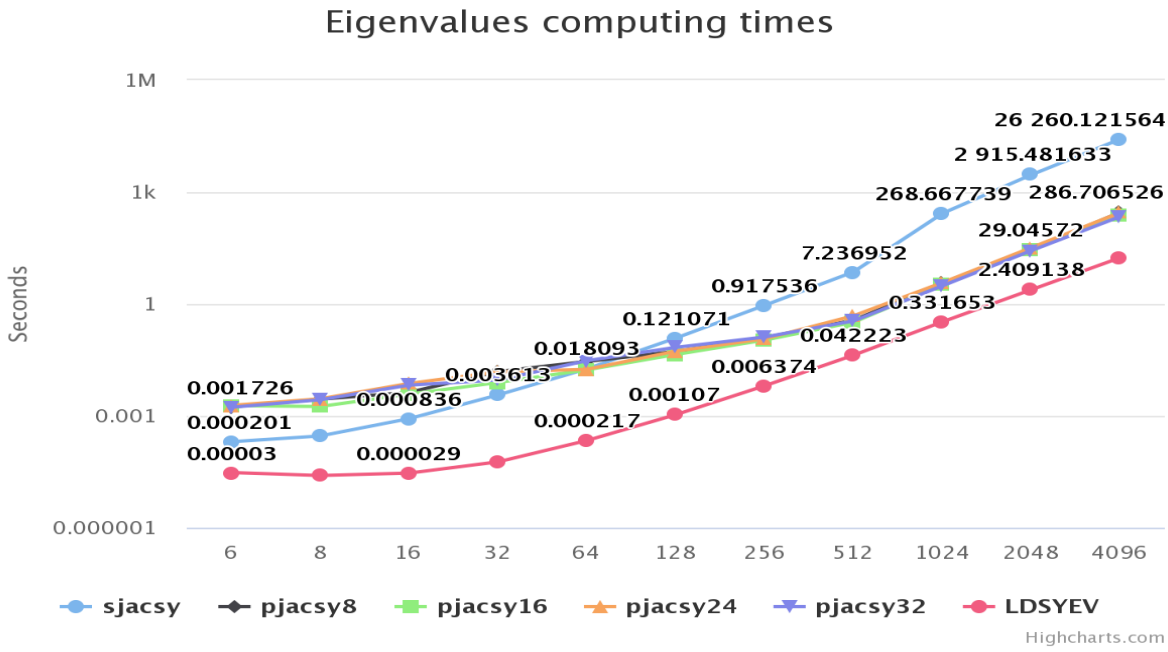
Na slici 21 možemo vidjeti kako je *LAPACK* rutina uvjerljivo brža od paralelne implementacije *Jacobijevog algoritma* u izračunu svojstvenih vrijednosti matrice.

Na slici 22 također možemo vidjeti rutinu *LAPACK*-a u usporedbi sa serijskom verzijom *Jacobijevog algoritma*.



Slika 22: Usporedba LDSYEV sa sjacsy (Izvor: Vlastita izrada)

Usporedba svih vremena na grafu vidljiva je na slici 23.



Slika 23: Sve implementacije (Izvor: Vlastita izrada)

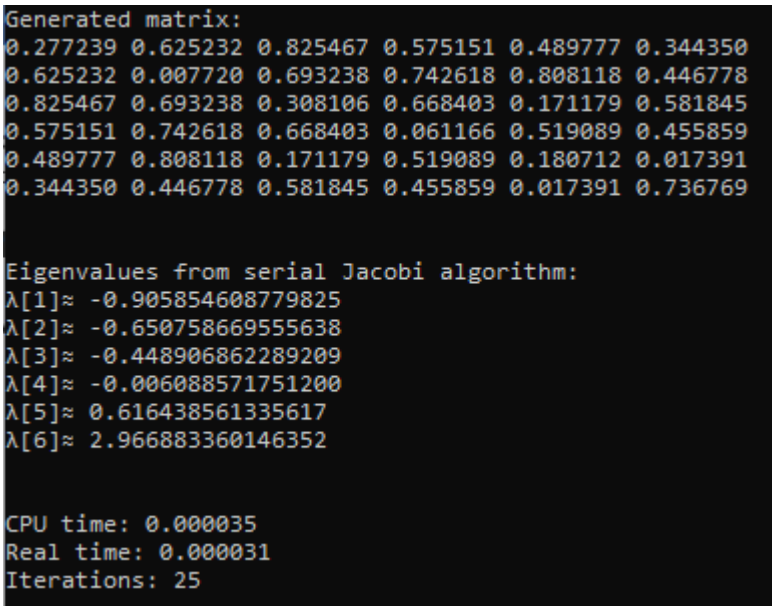
Sve navedene implementacije daju točne svojstvene vrijednosti. Kod *Jacobijevog* algoritma ta točnost leži u *Jacobijevim* rotacijama gdje se elementi iznad/ispod dijagonale poništavaju u nulu.

```
Generated matrix:
0.277239 0.625232 0.825467 0.575151 0.489777 0.344350
0.625232 0.007720 0.693238 0.742618 0.808118 0.446778
0.825467 0.693238 0.308106 0.668403 0.171179 0.581845
0.575151 0.742618 0.668403 0.061166 0.519089 0.455859
0.489777 0.808118 0.171179 0.519089 0.180712 0.017391
0.344350 0.446778 0.581845 0.455859 0.017391 0.736769

Eigenvalues for LAPACK routine DSYEV:
λ[1]≈ -0.905854608779825
λ[2]≈ -0.650758669555639
λ[3]≈ -0.448906862289209
λ[4]≈ -0.006088571751200
λ[5]≈ 0.616438561335618
λ[6]≈ 2.966883360146347


CPU time: 0.000137
Real time: 0.000137
```

Slika 24: Svojstvene vrijednosti LAPACK rutine DSYEV (Izvor: Vlastita izrada)

```
Generated matrix:
0.277239 0.625232 0.825467 0.575151 0.489777 0.344350
0.625232 0.007720 0.693238 0.742618 0.808118 0.446778
0.825467 0.693238 0.308106 0.668403 0.171179 0.581845
0.575151 0.742618 0.668403 0.061166 0.519089 0.455859
0.489777 0.808118 0.171179 0.519089 0.180712 0.017391
0.344350 0.446778 0.581845 0.455859 0.017391 0.736769

Calculating...

Matrix order: 6
Offset(A) : 1.90159619744853247703
Offset(A) : 0.29497022354278662482
Offset(A) : 0.00712821275001513949
Offset(A) : 0.00000002381473392689
Offset(A) : 0.00000000000000000339
Offset(A) : 0.00000000000000000000


Eigenvalues from parallel Jacobi algorithm:
λ[1]≈ -0.905854608779826
λ[2]≈ -0.650758669555639
λ[3]≈ -0.448906862289209
λ[4]≈ -0.006088571751200
λ[5]≈ 0.616438561335617
λ[6]≈ 2.966883360146347


CPU time : 0.002763
Real time: 0.002763
NUMBER OF ITTERATIONS: 26
```

Slika 25: Svojstvene vrijednosti paralelnog Jacobijevog algoritma (Izvor: Vlastita izrada)



Slika 26: Svojstvene vrijednosti serijskog Jacobijevog algoritma (Izvor: Vlastita izrada)

10. Zaključak

U ovom radu prikazane su usporedbe različitih implementacija izračuna svojstvenih vrijednosti. Prvo je razvijena i testirana serijska implementacija *Jacobijevog algoritma*, a potom se krenulo u implementaciju njegove paralelne verzije.

Paralelna verzija je implementirana pomoću *CUDA* programskog modela kojeg je razvio proizvođač grafičkih procesora *NVIDIA*. Pomoću poziva njegovih *API* metoda nudi se mogućnost paralelnog programiranja. Prije nego što se krenulo u samu implementaciju paralelne verzije *Jacobijevog algoritma* napravljeno je nekoliko testnih programa kako bi se savladale osnove *CUDA* programskog modela. Stečeno znanje zatim je primijenjeno na *Jacobijev algoritam* koji je uspješno paraleliziran.

Za kontrolu izračunatih svojstvenih vrijednosti korištena je biblioteka *LAPACK* koja pruža već gotove rutine. Koristila se rutina koja daje točne svojstvene vrijednosti za *double precision,* koji je korišten i kod implementacije serijskog i paralelnog *Jacobijevog algoritma*. Dobivene svojstvene vrijednosti su zatim uspoređene kako bi se ustanovila njihova točnost i preciznost. Preciznost koju pruža *double precision* je na 15 znamenki te je taj uvjet zadovoljen.

Osim preciznosti svojstvenih vrijednosti, fokus je također stavljen na vrijeme izvođenja izračuna svojstvenih vrijednosti. Usporedba vremena se vršila između paralelne i serijske verzije *Jacobijevog algoritma* te *LAPACK* rutine. Mjerenjima je ustanovljeno da *LAPACK* rutina ima najbrže vrijeme. Kako *LAPACK* rutina koristi *QR algoritam*, vremena koja će se uspoređivati vezana su za paralelnu i serijsku implementaciju *Jacobijevog algoritma*.

Paralelnom implementacijom *Jacobijevog algoritma* ubrzan je proces izračuna svojstvenih vrijednosti za nekoliko puta kod matrica reda većeg od 128, a kod velikih matrica, npr. reda 2048 paralelna implementacija brža je od serijske više od sto puta.

Također je za paralelnu verziju *Jacobijevog algoritma* eksperimentirano s brojem dretvi unutar jednog bloka. Prilikom definiranja broja dretvi potrebno je obratiti pozornost na arhitekturu grafičkog procesora na kojima se žele koristiti, jer neki grafički procesori podržavaju veći broj dretva od drugih. Za različite konfiguracije vremena su drugačija, što postavlja pitanje: bi li se dodatnom/drugačijom raspodjelom dretvi moglo pronaći optimalnije izvođenje programa koje bi dalo još bolja vremena?

Popis literature

[1] „*Some Applications of the Eigenvalues and Eigenvectors of a square matrix*“, (bez dat.) [Na internetu]. Dostupno: https://www.cpp.edu/~manasab/eigenvalue.pdf [pristupano: 07.09.2022].

[2] Microsoft, „*WSL documentation*“, 2022. [Na internetu]. Dostupno: https://docs.microsoft.com/en-us/windows/wsl/[pristupano 10.5.2022]

[3] Canonical, „*Enabling GPU acceleration on Ubuntu on WSL2 with the NVIDIA CUDA Platform*“,2022. [Na internetu]. Dostupno: https://ubuntu.com/tutorials/enabling-gpu-acceleration-on-ubuntu-on-wsl2-with-the-nvidia-cuda-platform#1-overview [pristupano 10.5.2022]

[4] NVIDIA, „*CUDA TOOLKIT*“,2022. [Na internetu]. Dostupno: https://developer.nvidia.com/cuda-toolkit [pristupano 10.5.2022]

[5] E. Anderson, Z. Bai, S. C. Bischof i Blackford, J. Demmel, J.J. Dongarra, D. Croz, A. Greenbaum, „*LAPACK*“, 1999. [Na internetu]. Dostupno: https://netlib.org/lapack/ [pristupano 1.6.2022]

[6] "*What is GPGPU (general purpose graphics processing unit)*“,2015. [Na internetu]. Dostupno: https://www.techtarget.com/whatis/definition/GPGPU-general-purpose-graphics-processing-unit [pristupano: 07.09.2022].

[7] „*Bump Mapping*", 2022. [Na internetu]. Dostupno: https://g.co/kgs/ieVVsn [pristupano: 07.09.2022].

[8] M. M. Pandur i M. Pilj, „*Generalizirani svojstveni problem*“, 2018. [Na internetu]. Dostupno: https://hrcak.srce.hr/file/300069 [pristupano: 26.7.2022]

[9] Hrvatska enciklopedija, mrežno izdanje, „*Svojstvena vrijednost*“, Leksikografski zavod Miroslav Krleža, 2021. [Na internetu]. Dostupno: https://www.enciklopedija.hr/Natuknica.aspx?ID=59149 [pristupano: 07.09.2022].

[10] Hrvatska enciklopedija, mrežno izdanje, „*Energijske svojstvene vrijednosti*“, Leksikografski zavod Miroslav Krleža, 2021. [Na internetu]. Dostupno: https://www.enciklopedija.hr/natuknica.aspx?ID=17936 [pristupano: 07.09.2022].

[11] X. Zhang, „*The properties and application of symmetric matrice*“, 2021. [Na internetu]. Dostupno: https://towardsdatascience.com/the-properties-and-application-of-symmetric-matrice-1dc3f183de5a [pristupano: 10.8.2022]

[12] J. Lambers, „*Jacobi Methods*“, 2010. [Na internetu]. Dostupno: https://web.stanford.edu/class/cme335/lecture7.pdf [pristupano 30.6.2022]

[13] Gene H. Golub i Charles F. Van Loan, „*Matrix Computations“*. Baltimore i London: The Johns Hopkins University Press. 1996.

[14] TutorialsPoint, „*C library function – clock()*“, 2022. [Na internetu]. Dostupno: https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm [pristupano 1.6.2022]

[15] B. Imana i P. Yoon, „*GPU-Accelerated Jacobi-like Algorithm for Eigendecomposition of General Complex Matrices*“ . [Na internetu]. Dostupno: http://sc16.supercomputing.org/sc-archive/src_poster/poster_files/spost163s2-file1.pdf [pristupano: 3.7.2022]

[16] M. T. Romero i A. M. Viveros, „*Parallel QR Factorization using Givens Rotations in MPI-CUDA for Multi-GPU*“, 2020. [Na internetu]. Dostupno: https://thesai.org/Downloads/Volume11No5/Paper_78-Parallel_QR_Factorization_using_Givens_Rotations.pdf [pristupano 1.7.2022]

[17] L. S. Chien, „*Jacobi-Based Eigenvalue Solver on GPU*“, 2017. [Na internetu]. Dostupno: https://on-demand.gputechconf.com/gtc/2017/presentation/s7121-lung-sheng-chien-jacobi-based-eigenvalue-solver.pdf [pristupano 3.7.2022]

[18] E. Kreyszig, „*Matrix Eigenvalues Problems*“, 2011. [Na internetu]. Dostupno: http://www.che.ncku.edu.tw/facultyweb/changct/html/teaching/Engineering%20Math/Chapter%208%20(Sec8.1,Sec8.3,Sec8.4).pdf [pristupano 10.8.2022]

[19] G.R. Lindfield i J.E.T. Penny, „*Linear Equations and Eigensystems*“, 2012. [Na internetu]. Dostupno: https://www.sciencedirect.com/topics/mathematics/eigenvalue-problems [pristupano 27.7.2022]

[20] J. Sanders, „*Introduction to CUDA C*“, 2012. [Na internetu]. Dostupno: https://www.nvidia.com/content/GTC-2010/pdfs/2131_GTC2010.pdf#page=1&zoom=auto,-119,482[pristupano: 25.6.2022]

[21] M. T. Heath, „*Parallel Numerical Algorithms*“, 2015. [Na internetu]. Dostupno: https://courses.engr.illinois.edu/cs554/fa2015/notes/12_eigenvalue_8up.pdf [pristupano 10.8.2022]

Popis slika

Popis tablica

Prilog 1 – matrix_generate_random

matrix_generate_random.h

```
#ifndef _MATRIX_GENERATE_RANDOM_
#define _MATRIX_GENERATE_RANDOM_

#ifdef __cplusplus
// when included in C++ file, let compiler know these are C functions
extern "C"
{
#endif
    void setMatrixOrder(int matrix_order);
    int getMatrixOrder();

    double **symetric_matrix_double();
    double **identityMatrixDouble();

#ifdef __cplusplus
}
#endif

#endif /* _MATRIX_GENERATE_RANDOM_*/
```

matrix_generate_random.c

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <stdbool.h>

#define SEED 112333121
int MATRIX_ORDER = 0;

/**
 * Set order of the matrix
 */
```

```
void setMatrixOrder(int matrix_order)
{
    MATRIX_ORDER = matrix_order;
}

/**
 * Get order of the matrix
 */
int getMatrixOrder()
{
    return MATRIX_ORDER;
}

/**
 * Generate symetric matrix with random double numbers
 */
double **symetric_matrix_double()
{
    double *values = calloc(MATRIX_ORDER * MATRIX_ORDER, sizeof(double));
    double **rows = malloc(MATRIX_ORDER * sizeof(double *));

    srand48(SEED);

    for (int i = 0; i < MATRIX_ORDER; ++i)
    {
        rows[i] = values + i * MATRIX_ORDER;
    }

    for (int i = 0; i < MATRIX_ORDER; ++i)
    {
        for (int j = i; j < MATRIX_ORDER; ++j)
        {
            rows[i][j] = drand48();
            rows[j][i] = rows[i][j];
        }
    }
    return rows;
}
```

```
double **identityMatrixDouble()
{
    double **matrix = malloc(MATRIX_ORDER * sizeof(double *));
    for (int i = 0; i < MATRIX_ORDER; i++)
    {
        matrix[i] = (double *)malloc(MATRIX_ORDER * sizeof(double));
    }

    for (int i = 0; i < MATRIX_ORDER; ++i)
    {
        for (int j = i; j < MATRIX_ORDER; ++j)
        {

            if (i == j)
            {
                matrix[i][j] = 1;
            }
            else
            {
                matrix[i][j] = 0;
                matrix[j][i] = 0;
            }
        }
    }

    return matrix;
}
```

Prilog 2 – printResults

printResults.h

```
#ifndef _PRINT_RESULT_
#define _PRINT_RESULT_

#ifdef __cplusplus
// when included in C++ file, let compiler know these are C functions
extern "C"
{
#endif
    void print_matrix_double(double **matrix, int _MATRIX_ORDER, char *desc);
    void  print_eigenvalues_double(char  *desc,  int  _MATRIX_ORDER,  double
*mat);
    void      print_eigenvalues_double_from_matrix(double      **matrix,int
_MATRIX_ORDER,char *desc);
    void print_array_double(double *array, int _MATRIX_ORDER, char *desc);

#ifdef __cplusplus
}
#endif

#endif /* _PRINT_RESULT_*/
```

printResults.c

```
#include <stdio.h>
#include <stdlib.h>

/**
 * Sort function
 */
int cmpfunc(const void *a, const void *b)
{
    if (*(double *)a > *(double *)b)
        return 1;
    else if (*(double *)a < *(double *)b)
        return -1;
```

```
    else
        return 0;
}

/**
 * Print eigenvalues of matrix (double)
 */
void print_eigenvalues_double(char *desc, int _MATRIX_ORDER, double *mat)
{
    printf("\n%s\n", desc);
    for (int i = 0; i < _MATRIX_ORDER; i++)
    {
        printf("%s%d%s %6.15f\n", "λ[", i + 1, "]≈", mat[i]);
    }
}

/**
 * Method that prints matrix
 */
void print_matrix_double(double **matrix, int _MATRIX_ORDER, char *desc)
{
    printf("\n%s\n", desc);
    for (int i = 0; i < _MATRIX_ORDER; i++)
    {
        for (int j = 0; j < _MATRIX_ORDER; j++)
        {
            printf("%f%s", matrix[i][j], " ");
        }
        printf("\n");
    }
}

/**
 * Method that prints eigenvalues extracted from matrix diagonal
 */
void print_eigenvalues_double_from_matrix(double **matrix,int _MATRIX_ORDER,
char *desc)
{
```

```
    double array[_MATRIX_ORDER];
    for (int i = 0; i < _MATRIX_ORDER; i++)
    {
        array[i] = matrix[i][i];
    }
    qsort(array, _MATRIX_ORDER, sizeof(double), cmpfunc);
    print_eigenvalues_double(desc, _MATRIX_ORDER, array);
    printf("\n");
}

/**
 *Print array of double type
 */
void print_array_double(double *array, int _MATRIX_ORDER, char *desc)
{
    printf("\n%s\n", desc);
    for (int i = 0; i < _MATRIX_ORDER; i++)
    {
        printf("%f%s", array[i], " ");
    }
    printf("\n");
}
```

Prilog 3 – time

time.h

```
#ifndef _GET_TIME_
#define _GET_TIME_

#ifdef __cplusplus
// when included in C++ file, let compiler know these are C functions
extern "C"
{
#endif

    void startTimer();
    void endTimer();
    double getTime();
    double getwallClockTime();

#ifdef __cplusplus
}
#endif

#endif /* _GET_TIME_*/
```

time.c

```
#include <time.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>

clock_t start_time, end_time;
double total_t;
long mtime, seconds, useconds;

struct timeval start, end;

/**
```

```
 * Method to start the timer.
 */
void startTimer()
{
    start_time = clock();
    gettimeofday(&start, NULL);
}

/**
 * Method to stop the timer.
 */
void endTimer()
{
    end_time = clock();
    gettimeofday(&end, NULL);
}

/**
 * Method to get the time elapsed on CPU.
 */
double getTime()
{
    total_t = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    return total_t;
}

/**
 *Get real time
 */
double getwallClockTime()
{
    seconds = end.tv_sec - start.tv_sec;
    useconds = end.tv_usec - start.tv_usec;
    double duration;
    duration = seconds + useconds / 1000000.0;
    return duration;
}
```