Paralelni algoritmi za QR faktorizaciju

Rukavina, Vedran

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Vedran Rukavina

PARALELNI ALGORITMI ZA QR FAKTORIZACIJU

Diplomski rad

Voditelj rada:
Doc. dr. sc. Nela Bosner

Zagreb, Rujan, 2019

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

QR dekompozicija matrice jako je važan alat u numeričkoj linearnoj algebri i zato nam je bitno da se algoritam za njeno računanje implementira na efikasan način. Standardni serijski algoritmi bazirani su na blokiranju Householderovih reflektora i Givensovih rotacija, tako da optimiziraju komunikaciju između brze cache memorije i sporije RAM memorije na CPU. Ovaj rad obrađuje paralelne algoritme za računanje QR faktorizacije na grafičkim karticama (GPU). Kod paralelnih algoritama optimizacija komunikacije među procesima ili dretvama, kao i među nivoima složene hijerarhije memorije od presudne je važnosti. Zbog toga opisujemo varijante QR algoritma koje reorganiziraju cijeli postupak u podzadatke koji se mogu neovisno i paralelno odvijati. Glavna ideja je što više podzadataka izvršiti paralelno ili minimizirati količinu podataka koja se šalje između procesora kao i prema globalnoj memoriji. Algoritmi su implementirani u C - u s CUDA sučeljem, a njihova se je efikasnost zatim usporedila s efikasnošću standardnih algoritama.

Ključne riječi - QR faktorizacija, Gram-Schmidt, Householderovi reflektori, TSQR, CAQR, CUDA, GPU, CPU

Poglavlje 1

Osnove

U ovom poglavlju definiramo termine koje ćemo koristiti kroz cijeli rad i dati ćemo osnove numeričke linearne algebre vezane uz QR faktorizaciju. Prvo krećemo s terminima.

1.1 Termini

U ovom djelu upoznajemo se s terminima koje koristimo kroz cijeli rad.

GPU - (*eng. Graphics Processing Unit*) Programabilan logički čip (procesor) specijaliziran za funkcije prikazivanja grafike. GPU se ne koristi samo za prikazivanje grafike. GPU-ovi su u mogućnosti obavljanja paralelnih operacija na više skupova podataka i sve se više koriste kao vektorski procesori za negrafičke aplikacije koje zahtijevaju ponavljajuća računanja.

CPU - (*eng. Central Processing Unit*) CPU je jedinica koja izvršava većinu procesa unutar računala. Služi za kontrolu ulaza i protoka podataka iz drugih dijelova računala. U mogućnosti je obavljati paralelne operacije na više skupova podataka.

CUDA - (*eng. Compute Unified Device Architecture*) CUDA (za više informacija pogledati [4]) je paralelna računalna platforma i model programskog sučelja (API) kreiran od strane Nvidia korporacije. Omogućuje programerima softvera i softverskim inženjerima da koriste GPU s CUDA podrškom za opću svrhu - pristup nazvan GPGPU (*eng. General-Purpose computing on Graphics Processing Units*). CUDA platforma softverski je sloj koji omogućava izravan pristup GPU virtualnom skupu instrukcija i paralelnim računalnim elementima.

LAPACK - (*eng. Linear Algebra PACKage*) LAPACK (za više informacija pogledati [11]) je napisan u FORTRANU 90 i nudi rutine za rješavanje sustava istodobno nekoliko linearnih

jednadžbi, rješenja najmanjih kvadrata linearnih sustava jednadžbi, problema svojstvenih vrijednosti i problema singularnih vrijednosti. Nudi rutine i za pridružene matrične faktorizacije (LU, Cholesky, QR, SVD, Schur, generalizirani Schur), kao i srodne proračune poput preuređene Schur-ove faktorizacije i procjene brojeva uvjetovanosti. Ima podršku za guste i rijetko popunjene matrice. U svim su područjima slične funkcionalnosti predviđene za realne i kompleksne matrice u jednostrukoj i dvostrukoj preciznosti.

BLAS - (*eng. Basic Linear Algebra Subprograms*) BLAS (za više informacija pogledati [1]) je skupina rutina koje pružaju standardne građevne blokove za izvođenje osnovnih operacija vektora i matrice. BLAS1 izvodi skalarne, vektorske i vektor vektor operacije. BLAS2 izvodi matrica-vektor operacije. BLAS3 izvodi matrica-matrica operacije. Budući da je BLAS učinkoviti, prenosiv i široko dostupan, obično se koristi u razvoju visokokvalitetnog softvera linearne algebre, na primjer, LAPACK.

host - CPU i njegova memorija (host memory)

device - GPU i njezina memorija (device memory)

kernel - Funkcija koja će biti izvršena na GPU.

1.2 Definicije

U ovom dijelu navest ćemo osnovne definicije koje ćemo koristiti kroz cijeli rad. Krenimo s jednostavnim definicijama koje su dobrim djelom preuzete iz [7] i [12].

Definicija 1.2.1. *Za prirodne brojeve m i n, preslikavanje $\mathcal{A} : \{1, 2, ..., m\} \times \{1, 2, ..., n\} \to \mathbb{F}$ naziva se matrica tipa (m, n) (reda $m \times n$) s koeficijentima iz polja $\mathbb{F}$.*

Napomena 1.2.2. *Obično se funkcije $\mathcal{A}$ pišu tablično u m redaka i n stupaca, tako da se funkcijska vrijednost $\mathcal{A}(i, j)$, koju najčešće označavamo s a_{ij} piše u i-ti redak i j-ti stupac.*

Definicija 1.2.3. *Neka je dana matrica $A \in \mathbb{C}^{m\times n}$. Adjungirana matrica matrice A je matrica $A^* \in \mathbb{C}^{n\times m}$ za koju vrijedi $A^* = (\bar{a_{ji}})_{ij}$. Ako je $A \in \mathbb{R}^{m\times n}$ tada $A^* = A^T \in \mathbb{R}^{n\times m}$ nazivamo transponirana matrica matrice A za koju vrijedi $A^T = (a_{ji})_{ij}$.*

Definicija 1.2.4. *Za matricu $Q \in \mathbb{C}^{n\times n}$ kažemo da je unitartna ako je $Q^*Q = QQ^* = I$, to jest ako su stupci matrice Q ortonormirani s obzirom na standardni skalarni umnožak. Ako matrica $Q \in \mathbb{R}^{n\times n}$ zadovoljava $Q^TQ = QQ^T = I$ tada je matrica Q ortogonalna.*

Definicija 1.2.5. *Matrica $A \in \mathbb{C}^{n\times n}$ je hermitska ako vrijedi $A^* = A$. Ako je matrica $A \in \mathbb{R}^{n\times n}$ za koju vrijedi $A^* = A^T = A$ tada kažemo da je simetrična.*

Definicija 1.2.6. *Hermitska matrica $A \in \mathbb{C}^{n\times n}$ je pozitivno definitna ako vrijedi $x^*Ax > 0$ za svaki $x \in \mathbb{C}^n \setminus \{0\}$. Simetrična matrica $A \in \mathbb{R}^{n\times n}$ je pozitivno definitna ako vrijedi $x^TAx > 0$ za svaki $x \in \mathbb{R}^n \setminus \{0\}$.*

Definicija 1.2.7. *Hermitska matrica $A \in \mathbb{C}^{n\times n}$ je pozitivno semidefinitna ako vrijedi $x^*Ax \geq 0$ za svaki $x \in \mathbb{C}^n$. Simetrična matrica $A \in \mathbb{R}^{n\times n}$ je pozitivno semidefinitna ako vrijedi $x^TAx \geq 0$ za svaki $x \in \mathbb{R}^n$.*

Definicija 1.2.8. *Kažemo da je matrica $A \in \mathbb{F}^{n\times n}$ regularna ili invertibilna ako postoji matrica B tako da vrijedi $AB = BA = I$, gdje I označava jediničnu matricu. Matrica B je inverzna matrica matrice A i označavamo je s A^{-1}.*

Definicija 1.2.9. *Permutacijska matrica je kvadratna binarna matrica koja u svakom retku i svakom stupcu točno na jednom mjestu ima element* 1*, a na svim ostalim mjestima ima-elemente* 0*.*

Definicija 1.2.10. *Za regularnu matricu $A \in \mathbb{F}^{n\times n}$ definiramo veličinu $\kappa_2(A) = \left\|A^{-1}\right\|_2 \|A\|_2$ koju nazivamo uvjetovanost matrice A u normi* 2*.*

Napomena 1.2.11.

- *Uvjetovanost matrice je uvijek veća ili jednaka* 1*. Zaista zbog konzistentsnosti matrične norme vrijedi:*

$$\kappa_2(A) = \left\|A^{-1}\right\|_2 \|A\|_2 \geq \left\|A^{-1}A\right\|_2 = \|I\|_2 = 1$$

- *Taj broj nam govori koliko je rješavanje sustava osjetljivo na perturbacije desne strane.*
- *Broj $\kappa_2(A)^{-1}$ daje udaljenost matrice A od skupa singularnih matrica (u normi* 2*)*
- *Ako je $\kappa_2(A) \gg 1$ tada kažemo da je matrica A loše uvjetovana (blizu je singularnosti i rješavanje sustava s tom matricom je osjetljivo na greške u desnoj strani sustava kojom god metodom ga rješavali).*

1.3 QR faktorizacija

Mnogi složeniji postupci koji uključuju operacije s matricama ne mogu se riješiti na učinkovit način niti se može jamčiti stabilnost zbog konačne strojne preciznosti.

Matrične dekompozicije (matrični rastavi) su metode kojima se matrica rastavlja na sastavne dijelove koji olakšavaju računanje složenijih matričnih operacija. Metode matričnih rastava, koje se nazivaju i metode matrične faktorizacije, temelj su linearne algebre na računalima, čak i za osnovne operacije poput rješavanja sustava linearnih jednadžbi, računanje matričnog inverza i računanje determinante matrice. To je pristup koji može pojednostaviti složenije operacije s matricom, koje se mogu lakše izvoditi nad sastavnim dijelovima, dok su na izvornoj matrici te složene operacije zahtijevnije.

Uobičajena analogija za rastav matrice je faktorizacija brojeva, kao što je faktorizacija od $6 = 2x3$. Zbog toga se dekompozicija matrice naziva i matrična faktorizacija. Postoji mnogo načina da se matrica rastavi kao i niz različitih tehnika dekompozicije matrice. Dvije jednostavne i široko korištene metode matrične dekompozicije su `LU` i `QR`.

Neka nam je Φ zajednička oznaka za $\mathbb{R}$ i $\mathbb{C}$ kada god ne bude bilo potrebno precizirati konkretan izbor. Sljedeći teorem će nam dati definiciju i egzistenciju `QR` faktorizacije.

Teorem 1.3.1. ***(QR faktorizacija)*** *Neka je $A \in \Phi^{m\times n}$ tako da je $m \geq n$. Tada postoji matrica $Q \in \Phi^{m\times m}$ takva da je*

$$Q^{-}A = R = \begin{bmatrix} R_0 \\ 0 \end{bmatrix},$$

gdje je $R \in \Phi^{m\times n}$, a $R_0 \in \Phi^{n\times n}$ gornje trokutasta matrica s nenegativnim dijagonalnim elementima. Matrica Q je unitarna ($\Phi = \mathbb{C}$), odnosno ortogonalna ($\Phi = \mathbb{R}$), dok nam Q^{-} označava adjungiranu ($Q^{-} = Q^{*}$, $\Phi = \mathbb{C}$), odnosno transponiranu ($Q^{-} = Q^{T}, \Phi = \mathbb{R}$) matricu matrice Q. Vrijedi $A = QR$.

Napomena 1.3.2. *Matricu Q možemo particionirati i to na sljedeći način:*

$$Q = \begin{array}{c} \begin{matrix} \overset{n}{} & \overset{m-n}{} \end{matrix} \\ \begin{bmatrix} Q_0 & Q_1 \end{bmatrix} \end{array}$$

pa nam iz teorema 1.3.1 slijedi:

$$A = \begin{bmatrix} Q_0 & Q_1 \end{bmatrix} \begin{bmatrix} R_0 \\ 0 \end{bmatrix} = Q_0 R_0.$$

Uočimo da `QR` faktorizaciju tada možemo zapisati u kraćem obliku

$$A = Q_0 R_0,$$

gdje nam je $Q_0 \in \Phi^{m\times n}$ ortonormirana matrica, a $R_0 \in \Phi^{n\times n}$ gornje trokutasta matrica s nenegativnim dijagonalnim elementima. Ovakav rastav matrice naziva se reducirana `QR` faktorizacija. Potpuna `QR` faktorizacija je oblika

$$A = QR$$

gdje je Q matrica dimenzije $m \times m$ s ortonormiranim stupcima (unitarna matrica, odnosno ortogonalna u realnom slučaju), a R gornjetrokutasta matrica dimenzije $m \times n$. Potpuna QR *faktorizacija se dobiva iz reducirane i to na način da se matrici Q doda $m - n$ ortonormiranih stupaca koji s ostalim stupcima čine bazu u Φ^m, a matrici R se dodaju nul-retci kako bi se dopunila do matrice dimenzije $m \times n$.*

Postoji nekoliko tehnika dolaženja do QR faktorizacije. Neki od tih tehnika su opisani u poglavlju 3.

Poglavlje 2

Problem najmanjih kvadrata i QR

U ovom poglavlju ćemo opisati poveznicu između problema najmanjih kvadrata i QR faktorizacije. Nadalje, u ovom djelu koristimo rezultate iz [8].

U praksi postoji nekoliko načina rješavanja problema najmanjih kvadrata. Metode koje se najčešće koriste su:

- QR faktorizacija
- sustav normalnih jednadžbi
- dekompozicija singularnih vrijednosti (SVD)
- transformacija u linearni sustav.

Kako je ovaj rad posvećen QR faktorizaciji, opisati ćemo samo prvu od gore navedenih metoda u rješavanju problema najmanjih kvadrata.

Teorem 2.0.1. *Neka je zadan linearan sustav $Ax = b$ od m jednadžbi i n nepoznanica. Neka su stupci matrice A linearno nezavisni, odnosno $rang(A) = n$. Tada je rješenje x problema najmanjih kvadrata $\|Ax - b\| \rightarrow min$ ujedno i jedinstveno rješenje normalne jednadžbe $A^TAx = A^Tb$.*

Napomena 2.0.2. *Matrica A^TA je simetrična i pozitivno semidefinitna, a sustav normalnih jednadžbi je uvijek konzistentan, jer je*

$$A^Tb \in Im(A^T) = Im(A^TA).$$

Pretpostavimo da je A^TA pozitivno definitna i A punog stupčanog ranga. Promotrimo sljedeće:

$$\begin{aligned} A^TAx &= A^Tb \\ x &= (A^TA)^{-1}A^Tb \end{aligned}$$

Zatim napišimo QR faktorizaciju matrice A:

$$A = QR = Q_0 R_0$$

gdje nam je Q_0 ortogonalna matrica tipa $m \times n$, a R_0 gornje trokutasta tipa $n \times n$ i uvrstimo u rješenje. Važno je napomenuti da ako je A punog stupčanog ranga tada je R_0 regularna. Slijedi

$$\begin{aligned} x =& (A^T A)^{-1} A^T b = (R_0^T Q_0^T Q_0 R_0)^{-1} R_0^T Q_0^T b \\ &= (R_0^T R_0)^{-1} R_0^T Q_0^T b = R_0^{-1} R_0^{-T} R_0^T Q_0^T b \\ &= R_0^{-1} Q_0^T b, \end{aligned}$$

točnije, x se dobiva primjenom "invertirane" skraćene QR faktorizacije od A na b (po analogiji s rješavanjem linearnih sustava, samo što A ne mora imati inverz).

Preciznije, da bismo našli x, rješavamo trokutasti linearni sustav

$$R_0 x = Q_0^T b.$$

Na ovakav način najčešće se rješavaju problemi najmanjih kvadrata. Ukupan broj aritmetičkih operacija je $2mn^2 - \frac{2}{3}n^3$.

QR faktorizacija može se koristiti i za problem najmanjih kvadrata kad matrica A nema puni stupčani rang, ali tada se koristi QR faktorizacija sa stupčanim pivotiranjem (na prvo mjesto dovodi se stupac čiji "radni dio" ima najveću normu). Ako matrica A ima rang $r < n$, onda njena QR faktorizacija sa stupčanim pivotiranjem ima oblik

$$AP = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

gdje je P matrica permutacije $n \times n$, R_{11} regularna reda r, a R_{12} neka $r \times (n - r)$ matrica. Zbog grešaka zaokruživanja, umjesto pravog R, izračunamo

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \\ 0 & 0 \end{bmatrix}$$

Naravno, željeli bismo da je $\|R_{22}\|_2$ vrlo mala, reda veličine $\epsilon \|A\|_2$, pa da ju možemo "zaboraviti", tj. staviti $R_{22} = 0$ i tako odrediti rang od A. Nažalost, to nije uvijek tako. Za primjer uzmimo bidijagonalnu matricu koja na glavnoj dijagonali sadrži $\frac{1}{2}$, a na gornjoj sporednoj sadrži 1. Takva matrica je skoro singularna ($det(A) = 2^{-n}$), njena QR faktorizacija je $Q = I$, $R = A$ i nema niti jednog R_{22} koji bi po normi bio malen.

Zbog toga koristimo pivotiranje, koje R_{11} pokušava držati što bolje uvjetovanim, a R_{22} po normi što manjim.

Poglavlje 3

Serijski i paralelni algoritmi

U ovom poglavlju navodimo neke serijske i paralelne algoritme za određivanje *QR* faktorizacije. Neke od tih algoritama smo implementirali, a dobivene rezultate smo dali u poglavlju 5.

3.1 Serijski algoritmi

U ovom djelu navodimo serijske algoritme koji koriste Gram-Schmidtov postupak ortogonalizacije i Householderove reflektore kako bi izračunali *QR* faktorizaciju.

Gram-Schmidtov postupak ortogonalizacije i QR

Gram-Schmidtov postupak je metoda u linearnoj algebri koja služi za ortogonalizaciju skupa vektora u zadanom euklidskom prostoru. U ovom poglavlju uglavnom koristimo rezultate opisane u [12]

Neka je $\mathcal{B} = \{a_1, a_2, ..., a_n\}$ baza za neki unitarni prostor $\mathcal{S}$, tada je Gram-Schmidtov niz zadan sa:

$$q_1 = \frac{a_1}{\|a_1\|}$$
$$q_k = \frac{a_k - \sum_{i=1}^{k-1} \langle q_i, a_k \rangle q_i}{\left\| a_k - \sum_{i=1}^{k-1} \langle q_i, a_k \rangle q_i \right\|}, \text{ za k=2,...,n}$$

i čini ortonormiranu bazu za $\mathcal{S}$.

Primjenimo gornji postupak na matricu $A = [a_1\ a_2\ ...\ a_n]$, gdje smo s a_i, $i = 1, 2, ..., n$ označili stupce pripadne matrice A. Želimo dobiti faktorizaciju $A = Q_0 R_0$

Postupak možemo provesti u dva glavna koraka:

1. Zbog $a_1 \neq 0$ definiramo
$$q_1^{'} = a_1,\ q_1 = \frac{q_1^{'}}{\left\|q_1^{'}\right\|}$$

2. u j-tom koraku imamo već ortonormirane vektore $q_1, ..., q_{j-1}$, koji razapinju isti potprostor kao i stupci $a_1, ..., a_{j-1}$ matrice A. Sada definiramo novi vektor i normiramo ga:
$$q_j^{'} = a_j - \sum_{i=j}^{j-1} \langle a_j, q_i \rangle q_i,\ q_j = \frac{q_j^{'}}{\left\|q_j^{'}\right\|}$$

Stupci matrice A su linearno nezavisni, što osigurava $q_j^{'} \neq 0$. Stavljanjem $Q_0 = [q_1\ q_2\ ...\ q_n]$ dobivamo $m \times n$ matricu (ortonormirani stupci). Uz oznaku za skalarne produkte i norme iz prethodne formule imamo

$$r_{ij} = \langle a_j, q_i \rangle = q_i^T a_j,\ r_{jj} = \left\|q_j^{'}\right\|$$

polazni stupac a_j možemo zapisati kao lineranu kombinaciju prvih j vektora q_i ortonormirane baze, u obliku

$$a_j = \sum_{i=1}^{j} r_{ij} q_i.$$

Koeficijenti r_{ij} su upravo elementi tražene matrice R_0.

Napomena 3.1.1. *U praksi se nikad ne koristi klasični Gram–Schmidtov postupak ortogonalizacije (skraćeno CGS), jer*

- *vektore a_j ortogonalizira obzirom na prethodne vektore q_i.*
- *Zbog toga je numerički nestabilan kad su stupci od A skoro linearno zavisni, to jest kada je A loše uvjetovana.*

Umjesto CGS-a, može se koristiti tako zvani modificirani Gram–Schmidtov postupak (skraćeno MGS)

- koji vektor a_j kojemu su već odstranjene komponente u smjeru $q_1, ..., q_{i-1}$ ortogonalizira u odnosu na q_i, pa je mnogo stabilniji.
- No, i kod njega se može dogoditi da je izračunati Q_0 vrlo daleko od ortogonalnog, tj. $\left\|Q_0^T Q_0 - I\right\| \gg \epsilon$, kad je A vrlo loše uvjetovana.

Napomena 3.1.2. *Obje spomenute varijante Gram-Schmidt algoritma su ekvivalentne u egzaktnoj aritmetici, ali različite u aritmetici konačne preciznosti.*

U nastavku dajemo pseudokodove klasičnog i modificiranog Gram-Schmidtovog algoritma.

Algoritam 1 Klasični Gram-Schmidtov algoritam - CGS

Ulaz: $a_1 \dots a_n$
Izlaz: Vraća matrice Q_0 i R_0

for $j \leftarrow 1$ to n **do**
  $q_j^{'} \leftarrow a_j$
  for $i \leftarrow 1$ to $j-1$ **do**
    $r_{ij} \leftarrow q_i^T * a_j$
    $q_j^{'} \leftarrow q_j^{'} - r_{ij} * q_i$
  end for
  $r_{jj} \leftarrow \left\| q_j^{'} \right\|$
  if $r_{jj} > 0$ **then**
    $q_j \leftarrow \frac{q_j^{'}}{r_{jj}}$
  else
    Matrica R_0 je singularna – stani!
  end if
end for

Algoritam 2 Modificirani Gram-Schmidtov algoritam - MGS

Ulaz: $a_1 \dots a_n$
Izlaz: Vraća matrice Q_0 i R_0

for $j \leftarrow 1$ to n **do**
  $q_j^{'} \leftarrow a_j$
  for $i \leftarrow 1$ to $j-1$ **do**
    $r_{ij} \leftarrow q_i^T * q_j^{'}$
    $q_j^{'} \leftarrow q_j^{'} - r_{ij} * q_i$
  end for
  $r_{jj} \leftarrow \left\| q_j^{'} \right\|$
  if $r_{jj} > 0$ **then**
    $q_j \leftarrow \frac{q_j^{'}}{r_{jj}}$
  else
    Matrica R_0 je singularna – stani!
  end if
end for

Napomena 3.1.3. *$r_{jj} = 0$ je ekvivalentno s tim da je a_j linearna kombinacija prethodnih stupaca matrice A (linearna zavisnost stupaca, pad ranga). Također elementi ispod glavne dijagonale su jednaki* 0.

Householderovi reflektori i QR

Ovaj dio posvećujemo važnosti Householderovih reflektora i njegovoj ulozi u računanju `QR` faktorizacije. U ovom poglavlju koristimo većinom rezultate iz [12].

Householderovi reflektori poništavaju sve osim jednog elementa u (skraćenom) stupcu. Algoritam koji koristi Householderove reflektore kako bi se izračunala `QR` faktorizacija može dati skraćenu i punu `QR` faktorizaciju.

Definicija 3.1.4. *Za zadani jedinični vektor $u \in \mathbb{R}^m$, matrica H definirana s*

$$H = H(u) := I - 2uu^T,\ \|u\|_2 = 1,$$

zove se Householderov reflektor.

Svojstva matrice H su:

- simetrična,
- ortogonalna

Napomena 3.1.5. *H je ortogonalna. Zaista, vrijedi:*

$$\begin{aligned} HH^T = H^2 &= (I - 2uu^T)(I - 2uu^T) \\ &= I - 4uu^T + 4u(u^Tu)u^T \\ &= I - 4uu^T + 4\|u\|_2^2 uu^T \\ &= I \end{aligned}$$

Nadalje H gradimo tako da vrijedi $Hx = \alpha e_1$ za neku konstantu α i $e_1 = [1\ 0\ 0\ \cdots\ 0]^T$. Kako smo pokazali da nam je H ortogonalna, $\|Hx\| = \|x\|$ i $\|\alpha e_1\| = |\alpha|\,\|x\| = |\alpha|$. Stoga je $\alpha = \pm\|x\|$. Predznak biramo tako da ima suprotan predznak od x_1. Vektor u koji tražimo je sada:

$$u = \begin{bmatrix} x_1 + sign(x_1)\|x_1\| \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

S jediničnim vektorom v definiranim kao $v = \frac{u}{\|u\|}$ pripadna Householderova refleksija poprima izraz:

$$H(x) = I - 2vv^T = I - 2\frac{uu^T}{u^Tu}$$

Metodom koja koristi Householderove reflektore za QR faktorizaciju pronalazimo odgovarajuće matrice H i množimo s lijeva originalnu matricu A kako bi dobili gornje trokutastu matricu R. Kao što smo vidjeli ranije, za razliku od Gram-Schmidtovog postupka, ovim pristupom ne formiramo izričito matricu Q. Međutim, matrica Q se može pronaći uzimajući matrični produkt svake sukcesivne matrice H.

$$Q = H_1 H_2 \cdots H_{n-1} H_n$$

U gornjem prikazu matrice Q smo koristili da nam je matrica A $m \times n$, $m \gg n$, što znači da imamo n Householderovih reflektora za svaki stupac.

U nastavku dajemo naše C implementacije glavnih dijelova Householderovog algoritma za QR faktorizaciju. Implementacije su nastale po uzoru na [10].

Algoritam 3 Matrix structure

```
/**
 * Matrix structure.
 */
typedef struct Matrix {
    int M;
    int N;
    double *elem;
}Matrix;
```

Algoritam 4 QR decomposition by Householder reflectors

```
/**
 * We initiate matrix R_=A and Q and I are initiate as identity matrix.
 */
for (int j=0; j<minMN; ++j)
{
    qrhr_gen(R_->elem, &(*Q)->elem, ((*Q)->M)-j, j*((*Q)->M)+j);
    qrhr_gen_H(&R_, (*Q)->elem, minMN, j, ((*Q)->M)-j);
    for(int i=j+1; i<R_->M; ++i) {
        R_->elem[j*(R_->M) + i] = 0.0;
    }
}
house_apply(&I, Q);
```

Algoritam 5 Apply Householder reflector

```
/**
 * Generate Householder reflection.
 * We compute reflektor as I - uu' and norm of vector u is sqrt(2).
 * param x: Type of 'const double *'. Represent vector x.
 * param u: Type of 'double **'. Vector for storing reflection.
 * param n: Type of 'const int'. Length of vectors x and u.
 * param start_pos: Type of 'const int'. Starting position of vectors x and u.
 * Result will be stored in vector u.
 */
void qrhr_gen(const double *x, double **u, const int n, const int start_pos) {

    double nu = l2_norm(x+start_pos, n);

    for(int i = 0; i<n; ++i) { (*u)[start_pos+i] = x[start_pos+i]; }
    if (nu != 0.0) {
        (*u)[start_pos] = (*u)[start_pos]/nu;
        (*u)[start_pos] = (*u)[start_pos] + sgn((*u)[start_pos]);
        double sqrt_abs_u0 = sqrt(fabs((*u)[start_pos]));
        (*u)[start_pos] = (*u)[start_pos]/sqrt_abs_u0;
        double nu_sqrt_abs_u_o = nu*sqrt_abs_u0;
        for (int i=1; i<n; ++i) {
            (*u)[start_pos+i] = (*u)[start_pos+i]/(nu_sqrt_abs_u_o);
        }
    }
    else {
        (*u)[start_pos] = sqrt(2.0);
    }
}
```

Algoritam 6 Apply Householder reflection

```
/**
 * Generate Householder matrix H.
 * param R: Type of 'double **'. Represent vector x.
 * param u: Type of 'double *'. Vector for storing reflection.
 * param minMN: Type of 'const int'. Number of columns;
 * param j: Type of 'const int'. Starting point of block.
 * param length: Type of 'const int'. Starting row length of block.
 * Result will be stored in apropriate block of matrix R.
 */
void qrhr_gen_H(Matrix **R, double *u, const int minMN,
                const int j, const int length) {
    // H = @(u,x) x - u*(u'*x);
    int start_pos = j*((*R)->M)+j;
    for (int i=j; i<minMN; ++i)
    {
        double dot_val = dot(u+start_pos, (*R)->elem+i*((*R)->M)+j, length);
        for (int k=0; k<length; ++k)
        {
            (*R)->elem[i*((*R)->M)+j+k] -= u[start_pos+k]*dot_val;
        }
    }
}
```

Algoritam 7 Product of Householder reflectors

```
/**
 * Apply matrix U on I.
 * param U: Type of 'Matrix **'. We apply U on I.
 * param I: Type of 'Matrix **'. Economic storage of Q from Householder algoritm.
 */
void house_apply(Matrix **U, Matrix **I) {
    for(int i=((*I)->N)-1; i>-1; --i){
        qrhr_gen_H(U, (*I)->elem, (*I)->N, i, ((*I)->M)-i);
    }
}
```

3.2 Paralelni algoritmi

Motivaciju za ovaj dio crpimo iz [2]. Prvo ćemo opisati originalni algoritam iz [2], a zatim u poglavlju 4 dati nešto ograničenu i manje skalabilnu implementaciju.

U pogledu postizanja dobrih performansi, posebno izazovan je slučaj QR faktorizacije za ”mršave” i ”visoke” matrice. To su matrice gdje je broj redaka mnogo veći od broja stupaca. Neka nam je A matrica s m redaka i n stupaca, definirajmo broj

$$r_{m,n} = \frac{m}{n}.$$

Broj $r_{m,n}$ će nam označavati omjer broja redaka matrice u odnosu na broj stupaca. Na primjer, neka matrica A ima 200 redaka i 20 stupaca, tada je $r_{200,20} = 10$. Što je $r_{m,n}$ veći to je matrica ”višlja” i ”mršavija”. QR faktorizacija ovakvih matrica, više od bilo kojih drugih, zahtijevaju veliku količinu komunikacije između procesora u paralelnom okruženju. To znači da većina računalnih biblioteka kada se suoče s ovim problemom, koriste pristupe koji su ograničeni propusnim opsegom (*eng. bandwith-bound*) i događa se zasićenje procesora s konačnom aritmetikom. Matrice s ekstremnim $r_{m,n}$ možda izgleda kao rijedak slučaj, međutim to je česta pojava u primjenama. Najčešći je primjer problem najmanjih kvadrata, koji su sveprisutni u gotovo svim granama znanosti i inženjerstva i mogu se riješiti QR-om (pogledati 2). U [2] je naveden i primjer oduzimanja stacionarne pozadine u videozapisima.

CAQR (*eng. Communication-Avoiding QR*) je algoritam za računanje QR faktorizacije koji je optimalan s obzirom na količinu obavljene komunikacije. To znači da algoritam minimizira količinu podataka koja mora biti poslana između procesora u paralelnom okruženju ili alternativno količinu podataka koji se prenose iz globalne memorije. Kao rezultat, algoritam CAQR prirodno je pogodan za ”visoke” i ”mršave” matrice ($r_{m,n}$ jako velik) gdje je komunikacija obično usko grlo.

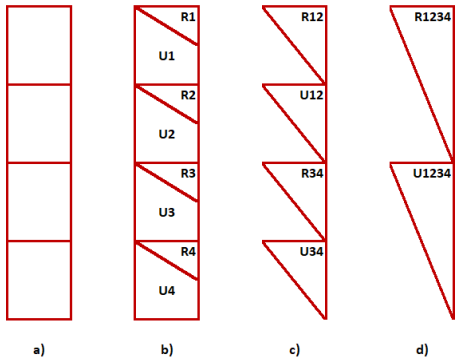
Većina postojećih implementacija QR faktorizacije za GPU koristi Householderove reflektore, kao i LAPACK. Jedno povoljno svojstvo Householderovog algoritma je da se on može organizirati na takav način da koristi BLAS3.

Prvo ćemo opisati TSQR (*eng. Tall-Skinny QR*) algoritam, zatim CACQR.

TSQR

TSQR algoritam reorganizira faktorizaciju ”visoke” i ”mršave” matrice (kao što su stupčane ploče) kako bi minimizirao pristup memoriji. Umjesto direktnog računanja jednog Householderovog vektora za svaki stupac, ”visoku” i ”mršavu” matricu dijelimo okomito na manje blokove, vidi sliku 3.1 pod a).. Zatim pomoću Householderovih reflektora vršimo faktorizaciju svakog bloka neovisno. Na ovaj način stvaramo manje reprezentacije matrice Q Householderovim matricama, koje nazivamo U, i gornje trokutastu matricu R za svaki blok, vidi sliku 3.1 pod b). Želja nam je ukloniti sve matrice R ispod gornje najviše dijagonale, tako da ih možemo grupirati i zatim primijeniti Householderov algoritam na svaku grupu, vidi sliku 3.1 pod c). Ovaj postupak grupiranja nastavljamo sve dok ne dođemo do jedne grupe. Ta zadnja jedna grupa sadržava gornje trokutastu matricu R i niz malih matrica U koji se prema potrebi mogu koristiti za eksplicitno računanje matrice Q, vidi sliku 3.1 pod d). Kao što već možemo i naslutiti TSQR algoritam nam je sklon paraleliziranju. Svaki blok unutar ploče neovisno može biti obrađen drugim procesorom. TSQR nam također omogućuje da problem podijelimo na manje dijelove radi lakše kontrole veličine. Ako odaberemo veličine blokova koji se uklapaju u predmemoriju, možemo postići značajnije performanse.

Vizualno, korake algoritma možemo vidjeti na slici 3.1. Kao što vidimo na slici 3.1, matrice R se eliminiraju pomoću binarnog stabla. Međutim, to se može učiniti bilo kojim oblikom stabla. Optimalni oblik može se razlikovati ovisno o karakteristikama arhitekture. Na primjer, na strojevima s više jezgara može se korititi redukcija binomnog stabla, dok su u [2] koristili redukciju četveronožnog stabla.

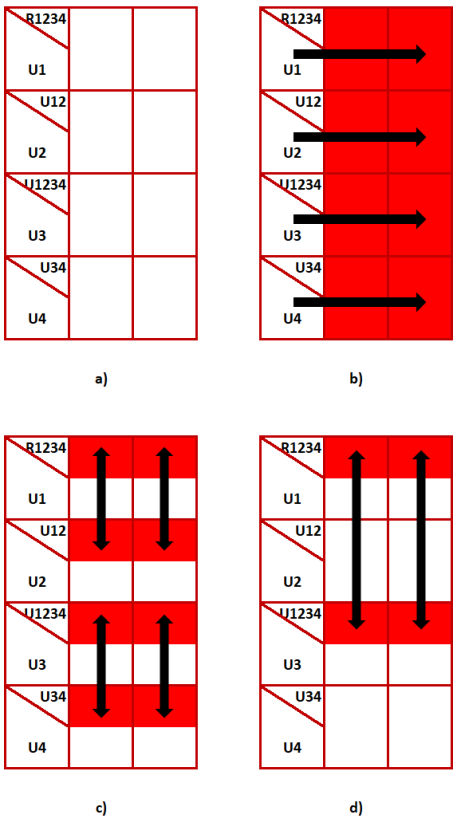


Slika 3.1: Koraci u TSQR algoritmu.

CAQR

CAQR algoritam je proširenje TSQR algoritma za matrice proizvoljne veličine. Ovog puta dijelimo matricu u mrežu manjih blok stupaca. Kao i kod blokirajućeg Householdera, CAQR uključuje faktorizaciju ploča i ažuriranje ostatka matrice. Faktorizacija ploča vrši se pomoću TSQR, prikazanog na slici 3.2 a). Zatim, moramo izvršiti ažuriranje ostatka matrice, što znači primijeniti Q^T ploče na ostatak matrice. Imajmo na umu da budući da TSQR djeluje na blokove unutar stupčane ploče, ažuriranje ostatka matrice može započeti prije dovršetka faktorizacije cijele ploče. Ovo uklanja sinkronizaciju u standardnom pristupu blokirajućeg Householdera i više je podložno paralelizmu. Nažalost, ne može se na jednostavan način koristiti množenje "velikih" matrica kao što se to može u blokirajućem Householderu. To je zbog distribuiranog formata u kojem TSQR proizvodi svoju matricu Q. Umjesto toga vrše se manja ažuriranja u svakom manjem bloku preostale podmatrice.

Ovdje gledamo dvije vrste ažuriranja matrica, a to su vodoravna ažuriranja i ažuriranja stabla. Vodoravna ažuriranja su jednostavniji slučaj. Ovdje uzimamo Householderove vektore stvorene iz prve faze TSQR algoritma i primjenjujemo ih vodoravno na ostatak matrice, prikazano na slici 3.2 b). Ova je operacija dosta ujednačena, a ažuriranje svakog bloka u ostatku matrice je neovisno. Nešto izazovnije je ažuriranje stabla. Ovdje uzimamo Householderove vektore generirane tijekom svake razine redukcije stabla TSQR algoritma i primjenjujemo ih na ostatke matrice. To uključuje miješanje manjih komada ostataka matrice, kao što je prikazano na slici 3.2 c) i d). Redovi ažurirane matrice razlikuju se ovisno o razini stabla redukcije. To može biti izazovno na GPU, jer su pristupi matrici nepravilniji i pomalo rijetki.



Slika 3.2: Koraci u CAQR algoritmu.

Nakon što se ažuriraju ostatci matrice, možemo prijeći na sljedeću ploču. Moramo paziti da nam se mijenja domena djelovanja, što odražava činjenicu da ostatak matrice nakon svakog koraka postaje kraći i uži.

Postoje različite mogućnosti za mapiranje CAQR u trenutne heterogene sustave. Navest ćemo neke od njih. Razmatramo heterogeni sustav koji sadrži jedan ili više CPU jezgri s DRAM-om, GPU s DRAM-om i fizičku povezanost dviju memorija. GPU općenito ima više mogućnosti za računanje i propusnost od CPU-a, dok CPU-ovi općenito imaju veću predmemoriju, veću mogućnost iskorištavanja paralelizma na razini instrukcija i bolje su opremljeni za rukovanje nepravilnim pristupima računanju i podacima.

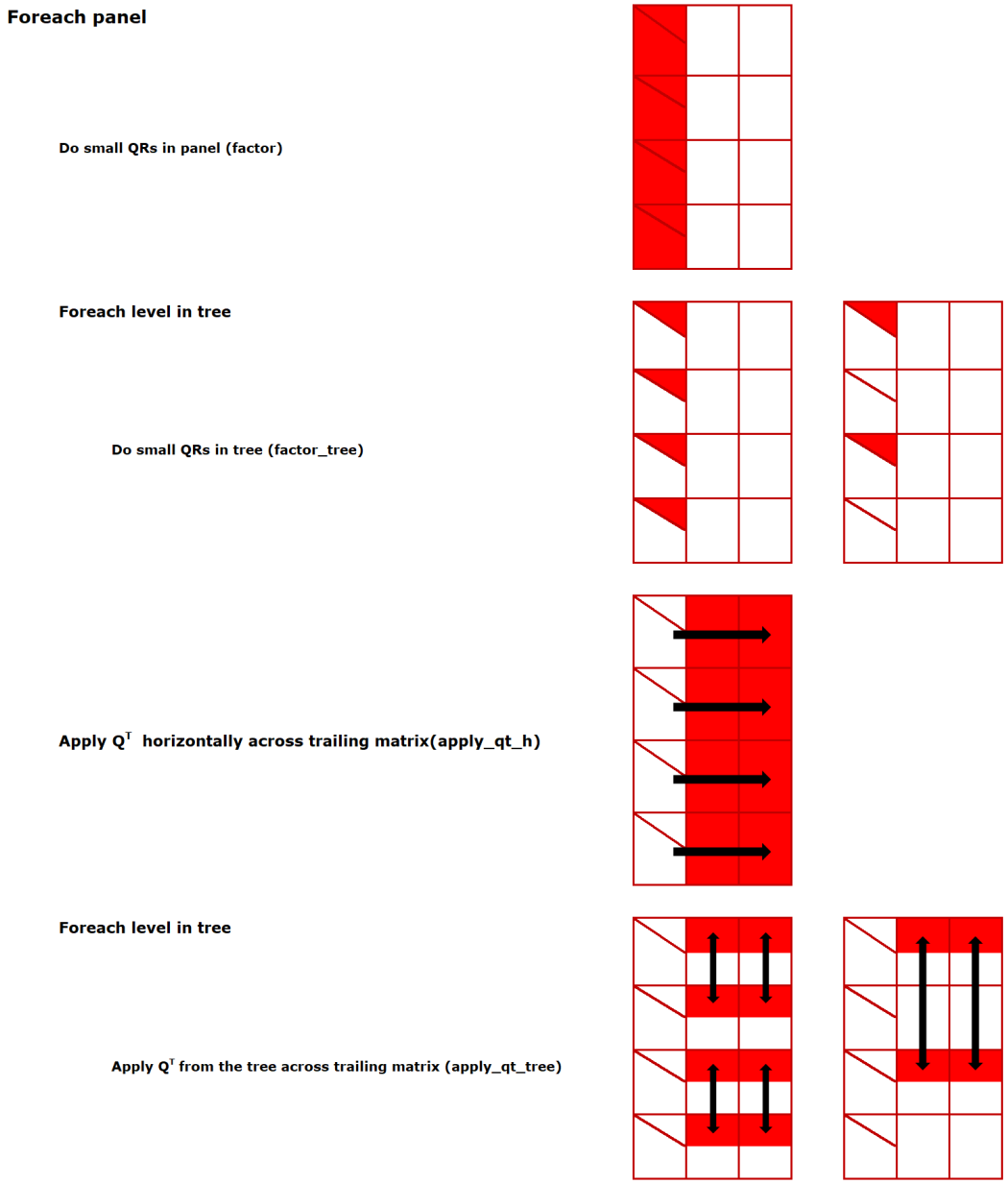
Nameću se dva glavna prirodna pitanja vezana uz CAQR algoritam:

- Gdje trebamo obaviti svaki korak računanja?
- Gdje bismo trebali pohraniti podatke?

Navedimo strategije mapiranja:

- **Faktorizacija ploča pomoću CPU-a i ažuriranje ostatka matrice pomoću GPU-a.** S ovim pristupom algoritam je sljedeći. Tanja ploča šalje se CPU-u, ako je potrebno, a CPU radi TSQR faktorizaciju. Rezultat faktorizacije šalje se natrag na GPU i koristi se za ažuriranje ostatka matrice. Potencijalno, CPU bi mogao započeti faktorizaciju sljedeće ploče dok je GPU i dalje zauzet primjenom ažuriranja generiranih kod faktorizacije prethodne ploče na ostatak matrice. Glavna prednost ovog pristupa je da prebacivanje rada na CPU omogućava preklapanje rada GPU-a i CPU-a. To nam omogućuje korištenje cijelog sustava. Faktorizacija ploče pomoću TSQR algoritma može se dobro iskoristiti pomoću CPU-u zbog nepravilne prirode redukcije stabla. Ažuriranje ostatka matrice je regularno, redovito i može se učinkovito obaviti na GPU-u. Jedan nedostatak ovog pristupa je prijenos podataka između memorije CPU-a i GPU-a, što nam može uvelike narušiti efikasnost. Problem je što se u takvom sustavu povećava kašnjenje koje može smanjiti radnu snagu za "mršavije" matrice. Ako se poslovi računanja CPU-a i GPU-a ne preklapaju, slanje faktorizacije ploče na CPU znači da za te dijelove izračuna ne možemo koristiti superiorne mogućnosti GPU-a.

- **Čitava faktorizacija na GPU.** S ovim pristupom, cijela se faktorizacija vrši na GPU. To uključuje faktorizaciju ploče pomoću TSQR algoritma i ažuriranje ostatka matrice. Pod pretpostavkom da je matrica u potpunosti u GPU memoriji, ovaj pristup uklanja problem kašnjenja u prijenosu podataka. To znači da možemo dobiti dobre performanse čak i na "mršavim" matricama. Također možemo imati koristi od većih računalnih mogućnosti GPU-a za faktorizaciju ploče. Nažalost, ovaj je pristup mnogo teže isprogramirati. To je prije svega zato što ne možemo ponovo koristiti postojeće podešene CPU programske pakete. Također, određeni dijelovi QR algoritma uključuju više nepravilnih izračuna i zbog toga su izazovniji i manje učinkoviti za provođenje u programerskim i izvedbenim modelima GPU-a. Pseudokod na slici 3.3, ilustrira neke nepravilne radnje potrebne za GPU pristup.



Slika 3.3: Pseudokod CAQR algoritma s redukcijom binarnog stabla.

Kao što je vidljivo sa slike 3.3, postoje četiri glavna dijela CAQR algoritma. Opišimo svaki od njih.

1. `factor`. Izvršava QR faktorizaciju manjeg bloka u bržoj memoriji koristeći prilagođene

BLAS2 rutine. Prepisuje Householderove vektore i gornje trokutastu matricu R preko originalne manje ulazne matrice.

2. factor_tree. Prikuplja gornje trokutaste matrice R generiranih factor-om i pohranjuje ih u brzu memoriju. Potom izvršava QR faktorizaciju na tom manjem bloku (sastoji se od dvije trokutaste matrice spojene u jednu), kao što radi i factor. Oblik rezultirajućih Householderovih vektora i matrice R je također skupina gornje trokutastih matrica i tako se mogu prebrisati R - ovi koji su bili učitani u brzoj memoriji.

3. apply_qt_h. Primjenjuje Q^T dobivenu od Householderovih vektora generiranih u factor dijelu vodoravno na manje blokove ostatka matrice. Vraća ažurirane blokove matrica na mjesta odakle su učitana.

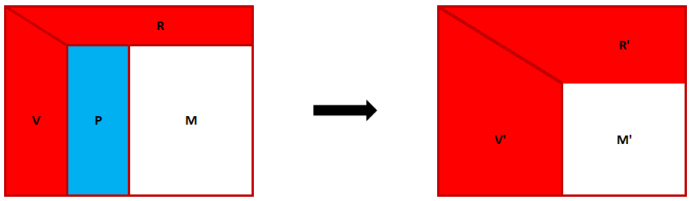
4. apply_qt_tree. Primjenjuje Q^T generiranu iz Householderovih vektora dobivenih od factor_tree tijekom redukcije stabla TSQR algoritma na odgovarajuća mjesta u ostatku matrice. Da bismo to učinili, prikupljamo distribuirane komponente ostatka matrice koja se ažurira, kao i distribuirane Householderove vektore. Primijenimo Q^T i natrag zapisujemo ažurirane blokove matrice na iste distribuirane lokacije s kojih su pročitani.

Poglavlje 4

Implementacija

U ovom dijelu ćemo dati opis jednostavnije, ograničene i manje skalabilne implementacije algoritma CAQR iz poglavlja 3. Radi jednostavnosti nazvat ćemo je *HRCAQR*. U poglavlju 5 ćemo dati rezultate dobivene korištenjem *HRCAQR* algoritma u odnosu na druge standardne algoritme.

Implementacija je slična opisanom `CAQR` algoritmu. Važno je znati da su nam sve matrice učitane u memoriju u stupčanom obliku i u `host` i u `device` memoriju. Razlika između `HRCAQR` i `CAQR` algoritma je ta što ne koristimo `TSQR` algoritam opisan u 3.2. U `HRCAQR` radimo `TSQR` ploče, ali tako da reduciramo niz preklapajućih pločica od dolje prema gore. Konkretno neka je matrica $A = [a_1,\ a_2,\ ...\ a_n]$, A je $m \times n$ matrica, gdje su nam $a_i,\ i = 1, 2, ...n$ stupci matrice A. Matricu A podijelimo na ploče gdje nam svaka ploča sadrži `PC_WIDTH` stupaca, to jest imamo $n/$`PC_WIDTH` ploča. Svaku ploču podijelimo na pločice od kojih je svaka dimenzije `PR_WIDTH` $\times$ `PC_WIDTH` imajući na umu da broj pločica u ploči ovisi u kojoj ploči se nalazimo. Na primjer, ako se nalazimo u prvoj ploči, tada imamo maksimalan broj pločica, u drugoj imamo jednu pločicu manje u odnosu na prvu, u trećoj jednu manje u odnosu na drugu i tako dalje. Kako idemo od dolje prema gore unutar ploče tada nam je početni redak prve pločice $m -$ `PR_WIDTH`, druge $m -$ 2`PR_WIDTH` + `PC_WIDTH`, treće $m -$ 3PR_WIDTH + 2`PC_WIDTH` i tako dalje. Odmah možemo uočiti da se pločice preklapaju. Početni redak pločice označimo s `pr`, a početni stupac (zapravo i ploču u kojoj se nalazimo) s `pc`. Od dolje prema gore unutar ploče idemo sve dok je `pr`+PR_WIDTH > `pc` i `pr` >= 0. Za svaku ploču računamo `QR` faktorizaciju odgovarajućih pločica unutar ploče krenuvši od dolje prema gore, a zatim vršimo ažuriranje odgovarajućih pločica ostatka matrice. Jedan korak `HRCAQR` algoritma možemo vidjeti na slici 4.1.



Slika 4.1: Korak `HRCAQR` algoritma. Radi se faktorizacija ploče `P` čiji rezultat su gornje trokutasti faktori `R` i reflektori `V` koji se koriste za ažuriranje podmatrice `M`.

Detaljan opis `kernel` - a je u odjeljku 4.1 i 4.2.
Glavni dio algoritma (`host`) je prikazan u algoritmu 8.

Algoritam 8 HRCAQR - main part

```
// FACTOR_NUM_THREADS = 512 ---> defined at compiled time
// UPDATE_NUM_THREADS = 1024 ---> defined at compiled time
// Adev is matrix A stored in device memory
// tauDev is a vector of coefficient (I-tau*vv^T) stored in device memory
// W is matrix from WY Householders representation
// m is number of rows of matrix A, n is number of columns of matrix A
int pc_count = 0;
int kernel1shared = (FACTOR_NUM_THREADS + 16 + 2 * PR_WIDTH_H * PC_WIDTH_H + PR_WIDTH_H) *
    sizeof(double);
int kernel2shared = (2 * PR_WIDTH_H * PC_WIDTH_H + PR_WIDTH_H) * sizeof(double);
for(int pc = 0; pc < n; pc += PC_WIDTH_H) {
    int pr_count = 0;
    for(int pr = m - PR_WIDTH_H; pr + PR_WIDTH_H > pc && pr >= 0; pr -= PR_PC_DIFF_H) {
        double* panelTau = &tauDev[(rowPanels * pc_count + pr_count) * PC_WIDTH_H];
        panel_householder<<<1, FACTOR_NUM_THREADS, kernel1shared>>>(Adev, panelTau, W, m,
            n, pr, pc);
        if(pc + PC_WIDTH_H < n) {
            int blocks = ceildiv(n - pc - PC_WIDTH_H, PR_WIDTH_H);
            trailing_update<<<blocks, UPDATE_NUM_THREADS, kernel2shared>>>(Adev, W, m, n,
                pr, pc);
        }
        pr_count++;
    }
    pc_count++;
}
```

Kao što vidimo u gornjem glavnom dijelu algoritma nailazimo na ograničenja i to su korištenja samo jednog `CUDA` bloka i fiksan broj dretvi po bloku u pozivanju `kernel` - a `panel_householder`, dok za `kernel trailing_update` koristimo više `CUDA` blokova, ali opet fiksan broj dretvi po bloku.

Prije detaljnog opisa gore navedenih `kernel` - a, važan nam je sljedeći rezultat koji je vezan uz *WY* reprezentaciju Householderovih reflektora.

Lema 4.0.1. *Pretpostavimo da je $Q = I + WY^T$ $n \times n$ ortogonalna matrica gdje su $W, Y \in \mathbb{R}^{n \times j}$ za neki $j \in \mathbb{N}$. Ako je $P = I - \beta vv^T$, gdje je $v \in \mathbb{R}^n$, $z = -\beta Qv$ i $\beta \in \mathbb{R}$, tada je*

$$Q_+ = QP = I + W_+ Y_+^T,$$

gdje su $W_+ = [W\ z]$ i $Y_+ = [Y\ v]$ obje $n \times (j+1)$.

Detaljnije o WY reprezentaciji produkta Householderovih reflektora kao i o algoritmima za njeno računanje možete vidjeti u [5].

Sljedeći dijelovi ovog poglavlja nam detaljnije opisuju svaki od gore spomenutih `kernel` - a uz gore navedene pretpostavke vezane uz matricu A.

4.1 kernel - panel_householder

U ovom djelu opisujemo `panel_householder kernel` koji nam računa `QR` faktorizaciju pločice. Implementacija je isključivo vezana za `device`.

`kernel` nam računa blok Householderovu faktorizaciju prvih `PC_WIDTH` stupaca i prvih `PR_WIDTH` redaka počevši od stupca `pc` i retka `pr`. Ulazni parametri `kernel` - u su: `A`, `tau`, `W`, `m`, `n`, `pr`, `pc`. `A` nam označava našu $m \times n$ matricu za koju želimo izračunati `QR` faktorizaciju. `W` nam je matrica iz WY reprezentacije Housholderove matrice. `tau` nam označava vektor skalara ($H = I - \tau vv^T$).

Koristimo zajedničku `shared` memoriju gdje god možemo. Svaka dretva učitava jedan dio pločice s odgovarajućih mjesta matrice `A` u `shared` memoriju. Dimenzija pločice je `PR_WIDTH` $\times$ `PC_WIDTH`. Označimo s `panel` tu pločicu učitanu u `shared` memoriju.

Podijelimo izvršavanje kernela u dvije faze:

- Prva faza. Iteracija po svim stupcima `panel` - a. Prvo računamo `tau` i vektor `v` (Fragment 11). Ovaj dio nam sadrži računanje skalarnog produkta i norme stupca u `panel` - u. Skalarni produkt računamo tako da svaka dretva izračuna lokalni dio skalarnog produkta, a zatim pomoću redukcije se zbroje sve vrijednosti lokalnog računa i tako dobijemo traženu vrijednost. Kada god izračunamo `tau` i `v` tada ažuriramo `panel` kako bi sadržavao pripadajući gornji trokut R i pripadajuće reflektore `v`. U Fragment 12 je prikazan fragment koda za računanje skalarnog produkta i norme. Slijedi nam računanje `z` vrijednosti pomoću WY reprezentacije Householderovih reflektora (Fragment 13). Svaka dretva računa jednu `z` vrijednost. Izračunate `z` vrijednosti zatim spremamo na odgovarajuće mjesto u `W_shared` matricu kako bi ih mogli koristiti u kasnijem računu. Korak koji slijedi je primjena izračunatog reflektora stupca na preostale stupce u `panel` - u (Fragment 14). U ovom koraku nam se opet mijenja `panel`.

- Druga faza. Spremanje `W_shared` matrice u matricu `W` i `panel` u `A` na pripadajuća mjesta i time je izvršavanje `kernel` - a gotovo (Fragment 15).

Kompletan `kernel` je dan u algoritmu 9.

Algoritam 9 kernel - panel_householder

```
/**
 * Do block Householder factorization of the first PC_WIDTH columns of A,
 * starting at pc (col) and pr (row)
 * param A: Type of "double *". Matrix on which we will perform block Householder
     factorization.
 * param tau: Type of "double *". Vector in which we will store Householders coefficients.
 * param W: Type of "double *". From Q=I-WY^T.
 * param m: Type of "int". Number of rows of matrix A.
 * param n: Type of "int". Number of columns of matrix A.
 * param pr: Type of "int". Starting row point for Householder factorization.
 * param pc: Type of "int". Starting column point for Householder factorization.
 */
__global__ void panel_householder(double* A, double* tau, double* W, int m,
                                  int n, int pr, int pc) {
  int pr_mult_pc = PR_WIDTH * PC_WIDTH;
  // define shared memory in share space with size passed from the host
  extern __shared__ double shared_buffer[];
  // used in computing scalar product
  double* to_reduce = shared_buffer;
  // used in computing scalar product
  double* final_reduce = shared_buffer + blockDim.x;
  int final_reduce_num = 16;
  // panel in which we will load part of A
  double* panel = final_reduce + final_reduce_num;
  // W_shared (W) from WY Householder representation applyed on panel
  double* W_shared = panel + pr_mult_pc;
  // copy of the updating column of panel
  double* A_col = W_shared + pr_mult_pc;
  // now we determine where the panel lies with respect to the matrix A
  bool bottom_panel = pr == m - PR_WIDTH;
  bool top_panel = pr < pc;

  /*
   * this is where Fragment 10 comes into.
   */

  // here we need synchronization to ensure that all parts of shared memory are
  // loaded in the proper places (panel and W_shared)
  __syncthreads();
  for(int col = 0; col < PC_WIDTH && pc + col < n; ++col) {
   /*
    * this is where Fragment 11 comes into.
    */
   // syncronization needed because panel is changed and we need changed panel
   // later in computation
    __syncthreads();
    // v is now fully computed and stored back to panel
    /*
     * this is where Fragment 13 comes into.
     */
    // we need syncronization because W_shared is changed and we need W_shared
    // later in computation
    __syncthreads();
    /*
     * this is where Fragment 14 comes into.
     */
  }
  // we need syncronization here cecause we write all computation back to global
  __syncthreads();
  /*
   * this is where Fragment 15 comes into.
   */
}
```

Fragment 10 Set `W_shared` matrix to zero and load part of `A` into panel

```
...
// set W_shared matrix to zero
// each thread in block set their own part of W_shared to 0.0
for(int i = 0; i < pr_mult_pc; i += blockDim.x) {
  int index = i + threadIdx.x;
  if(index < pr_mult_pc)
    W_shared[index] = 0.0;
}

// load panel into shared
// each thread per block load their own part of A into panel
for(int i = 0; i < pr_mult_pc; i += blockDim.x) {
  int index = i + threadIdx.x;
  if(index < pr_mult_pc) {
    int col = index / PR_WIDTH;
    int row = index % PR_WIDTH;
    double mat_val = 0;
    if(pc + col < n && pr + row >= 0) {
      mat_val = A[(pc + col) * m + pr + row];
    }
    panel[row + col * PR_WIDTH] = mat_val;
  }
}
...
```

Fragment 11 Compute `tau` and `v`

```
...
// starting point of vector v
int v_start, v_end;
if(top_panel && bottom_panel) {
  v_start = pc - pr + col;
  v_end = PR_WIDTH;
}
else if(!top_panel && bottom_panel) {
  v_start = col;
  v_end = PR_WIDTH;
}
else if(top_panel && !bottom_panel) {
  // v_start needs to be at or below A's diagonal, even if
  // panel boundaries extends above it
  v_start = pc - pr + col;
  v_end = PR_PC_DIFF + col + 1;
}
else {
  // neither top nor bottom panel
  v_start = col;
  v_end = PR_PC_DIFF + col + 1;
}
int vlen = v_end - v_start;
/*
 * this is where Fragment 12 comes into
 */
double leading = panel[col * PR_WIDTH + v_start];
double sign = (leading < 0) ? -1.0 : 1.0;
double u = leading + sign * norm;
double this_tau = sign * u / norm;
// compute entire v vector in-place, storing it back to A subdiag
for(int i = v_start; i < v_end; i += blockDim.x) {
  int index = i + threadIdx.x;
  if(index == v_start) {
    // thread 0 uniquely responsible for setting R diagonal entry and tau
    tau[col] = this_tau;
    panel[col * PR_WIDTH + v_start] = -sign * norm;
  }
  else if(index < v_end) {
    panel[col * PR_WIDTH + index] /= u;
  }
}
...
```

Fragment 12 Scalar product and norm

```
...
// compute the inner product and norm of column
double inner_prod = 0.0;
{
  double local_inner_prod = 0.0;
  // use a cyclic row distribution for perfect coalesced accesses
  for(int i = v_start; i < v_end; i += blockDim.x)
  {
    int index = i + threadIdx.x;
    if(index < v_end)
    {
      local_inner_prod += panel[index + col * PR_WIDTH] * panel[index + col * PR_WIDTH];
    }
  }
  // now, sum up the local_inner_prods across the whole block write the
  // partial sums to shared, then do linear reduction
  to_reduce[threadIdx.x] = local_inner_prod;
  // we need synchronization here
  __syncthreads();
  if(threadIdx.x < final_reduce_num)
  {
    local_inner_prod = 0.0;
    for(int i = 0; i < blockDim.x; i += final_reduce_num)
    {
      int index = i + threadIdx.x;
      if(index < blockDim.x)
        local_inner_prod += to_reduce[index];
    }
    final_reduce[threadIdx.x] = local_inner_prod;
  }
  // we need synchronization here
  __syncthreads();
  // now, every thread sums up final_reduce to get inner_prod
  for(int i = 0; i < final_reduce_num; ++i)
    inner_prod += final_reduce[i];
}
double norm = sqrt(inner_prod);
...
```

Fragment 13 Compute z

```
...
// compute z vector using W,Y
// each thread will compute one entry in z
for(int i = 0; i < PR_WIDTH; i += blockDim.x)  {
  int index = i + threadIdx.x;
  if(index < PR_WIDTH) {
    double z_val = 0.0;
    //set z_val to v[index]
    if(index == v_start) { z_val = -this_tau; }
    else if(index > v_start && index < v_end) {
      z_val = -this_tau * panel[col * PR_WIDTH + index];
    }
    // finish computing entry i of z
    // compute z_val as (W * Y^T * v)(i)
    double wyt_v_i = 0;
    for(int j = 0; j < PR_WIDTH; ++j)  {
      // need inner product of row i of W and row j of Y
      // this is (WY^T)(i, j)
      // use the fact that only the first col+1 columns of W and Y are nonzero
      double wyt = 0;
      for(int k = 0; k < col; ++k) {
        double y_val = 0;
        //find the set of rows for column k of panel
        int v_start_k, v_end_k;
        if(top_panel && bottom_panel) {
          v_start_k = pc - pr + k;
          v_end_k = PR_WIDTH;
        }
        else if(!top_panel && bottom_panel) {
          v_start_k = k;
          v_end_k = PR_WIDTH;
        }
        else if(top_panel && !bottom_panel) {
          // v_start needs to be at or below A diagonal, even if
          // panel boundaries extends above it
          v_start_k = pc - pr + k;
          v_end_k = PR_PC_DIFF + k + 1;
        }
        else {
          // neither top nor bottom panel
          v_start_k = k;
          v_end_k = PR_PC_DIFF + k + 1;
        }
        if(j > v_start_k && j < v_end_k) { y_val = panel[k * PR_WIDTH + j]; }
        else if(j == v_start_k) { y_val = 1; }
        wyt += W_shared[k * PR_WIDTH + index] * y_val;
      }
      double v_val = 0.0;
      if(j == v_start) { v_val = 1; }
      else if(j > v_start && j < v_end) { v_val = panel[col * PR_WIDTH + j]; }
      wyt_v_i += wyt * v_val;
    }
    z_val -= this_tau * wyt_v_i;
    W_shared[col * PR_WIDTH + index] = z_val;
  }
}
...
```

Fragment 14 Apply reflector

```
...
// apply reflector in col to remaining columns in panel
for(int apply_col = col + 1; apply_col < PC_WIDTH && pc + apply_col < n; ++apply_col) {
  // Create a copy of the updating column of A which will
  // persist while each entry is computed
  // Only the height range [v_start, m) is read, used and written back
  for(int i = 0; i < vlen; i += blockDim.x) {
    int index = i + threadIdx.x;
    if(index < vlen) {
      A_col[index] = panel[apply_col * PR_WIDTH + v_start + index];
    }
  }
  __syncthreads();
  for(int apply_row = v_start; apply_row < v_end; apply_row += blockDim.x) {
    int index = apply_row + threadIdx.x;
    if(index < v_end) {
      double val = A_col[index - v_start];
      double v_index = 0;
      if(index == v_start) { v_index = 1; }
      else { v_index = panel[col * PR_WIDTH + index]; }
      for(int i = v_start; i < v_end; ++i) {
        double v_i = 0;
        if(i == v_start) { v_i = 1; }
        else { v_i = panel[col * PR_WIDTH + i]; }
        val -= this_tau * v_index * v_i * A_col[i - v_start];
      }
      panel[apply_col * PR_WIDTH + index] = val;
    }
  }
}
...
```

Fragment 15 Write out `W` and `A`.

```
...
// write out W and panel back to global
for(int i = 0; i < pr_mult_pc; i += blockDim.x) {
  int index = i + threadIdx.x;
  if(index < pr_mult_pc) {
    W[index] = W_shared[index];
  }
}
for(int i = 0; i < pr_mult_pc; i += blockDim.x) {
  int index = i + threadIdx.x;
  if(index < pr_mult_pc) {
    int row = index % PR_WIDTH;
    int col = index / PR_WIDTH;
    A[pr + row + (pc + col) * m] = panel[row + col * PR_WIDTH];
  }
}
...
```

4.2 kernel - trailing_update

U ovom djelu opisujemo `trailing_update` `kernel` koji nam vrši ažuriranje ostatka matrice. Implementacija je isključivo vezana za `device`.

`kernel` nam vrši ažuriranje ostatka matrice počevši od retka `pr` i stupca `pc`. Ulazni parametri `kernel` - u su: `A`, `W`, `m`, `n`, `pr`, `pc`. S reprezentacijom ulaznih parametara smo se upoznali u odjeljku 4.1.

Kao i u odjeljku 4.1, koristimo zajedničku `shared` memoriju gdje god možemo.

Podijelimo `kernel` u nekoliko faza:

- Prva faza. Kreiramo matricu `Y_shared` koja nam dolazi od *WY* Householderove reprezentacije i koja nam nakon kreiranja ostaje konstantna do kraja izvršavanja `kernel` - a (Fragment 17). Matrica `Y_shared` je učitana u `shared` memoriji. Stupci od `Y_shared` su nam reflektori ispod dijagonale matrice `A` (gledamo one stupce od `A` koje smo ažurirali `kernel` - om u 4.1).

- Druga faza. Kopiramo vrijednosti matrice `W` u matricu `W_shared`. `W_shared` nam je spremljena u `shared` memoriji.

- Treća faza. Neka je A=`A(pr:pr+PR_WIDTH,pc+PC_WIDTH:)`. Ažuriramo ostatak matrice $A = (I +$ `Y_shared`$\texttt{W\_shared}^T)A$ (Fragment 18).

Kompletan `kernel` je dan u algoritmu 16.

Algoritam 16 kernel - `trailing_update`

```
/**
 * Do trailing matrix update of A, starting at pc.
 * param A: Type of "double *". Matrix on which we will perform block Householder
     factorization.
 * param tau: Type of "double *". Vector in which we will store Householders coefficients.
 * param W: Type of "double *". Matrix from WY Householder representation.
 * param m: Type of "int". Number of rows of matrix A.
 * param n: Type of "int". Number of columns of matrix A.
 * param pr: Type of "int". Starting point for determining where are we.
 * param pc: Type of "int". Starting point for Householder factorization.
 */
__global__ void trailing_update(double* A, double* W, int m, int n,
                                int pr, int pc) {
  int pr_mult_pc = PR_WIDTH * PC_WIDTH;
  // define shared memory in share space with size passed from the host
  extern __shared__ double shared_buffer[];
  // W_shared (W) from WY Householder representation
  double* W_shared = &shared_buffer[0];
  // Y_shared (Y) from WY Householder representation
  double* Y_shared = &W_shared[pr_mult_pc];
  // copy of the updating column of panel
  double* A_col = &Y_shared[pr_mult_pc];
  // now we determine where the panel lies with respect to the matrix A
  bool bottom_panel = pr == m - PR_WIDTH;
  bool top_panel = pr < pc;
  // update trailing columns of A: A = (I + YW^T)A
  int block_col = pc + PC_WIDTH + blockIdx.x * PR_WIDTH;
  /*
   * this is where Fragment 17 comes into.
   */
  // load block of W into shared (from the global W)
  for(int i = 0; i < pr_mult_pc; i += blockDim.x) {
    int index = i + threadIdx.x;
    if(index < pr_mult_pc) {
      W_shared[index] = W[index];
    }
  }
  __syncthreads();
  //For each column to update...
  for(int apply_col = 0; apply_col < PR_WIDTH && apply_col + pc + PC_WIDTH < n; ++
      apply_col) {
    //Save a copy of the column to A_col
    for(int j = 0; j < PR_WIDTH; j += blockDim.x) {
      int index = j + threadIdx.x;
      if(index < PR_WIDTH) {
        if(pr + index < m && block_col + apply_col < n) {
          A_col[index] = A[pr + index + (block_col + apply_col) * m];
        }
        else { A_col[index] = 0; }
      }
    }
    __syncthreads();
    /*
     * this is where Fragment 18 comes into.
     */
    __syncthreads();
  }
}
```

Fragment 17 Load `Y_shared`.

```
...
// load Y_shared block from A
// it will stay constant for whole kernel
for(int i = 0; i < pr_mult_pc; i += blockDim.x) {
  int index = i + threadIdx.x;
  if(index < pr_mult_pc) {
    int row = index % PR_WIDTH;
    int col = index / PR_WIDTH;
    int v_start;
    int v_end;
    if(top_panel && bottom_panel) {
      v_start = pc - pr + col;
      v_end = PR_WIDTH;
    }
    else if(!top_panel && bottom_panel) {
      v_start = col;
      v_end = PR_WIDTH;
    }
    else if(top_panel && !bottom_panel) {
      // v_start needs to be at or below A's diagonal, even if
      // panel boundaries extends above it
      v_start = pc - pr + col;
      v_end = PR_PC_DIFF + col + 1;
    }
    else {
      // neither top nor bottom panel
      v_start = col;
      v_end = PR_PC_DIFF + col + 1;
    }
    if(!bottom_panel) { v_end = PR_PC_DIFF + col + 1; }
    int mat_row = pr + row;
    int mat_col = pc + col;
    // Y_shared's columns are simply the reflectors stored in A's subdiagonal.
    // this reads back the implicit 0/1 entries
    double y_val = 0;
    if(mat_row < m && mat_col < n) {
      if(row > v_start && row < v_end) { y_val = A[mat_row + m * mat_col]; }
      else if(row == v_start) { y_val = 1; }
    }
    Y_shared[row + col * PR_WIDTH] = y_val;
  }
}
...
```

Fragment 18 Update triling matrix.

```
...
// Compute the updated (I + Y_shared * W_shared^T) * A_col
for(int i = 0; i < PR_WIDTH; i += blockDim.x) {
  int entry = i + threadIdx.x;
  if(entry < PR_WIDTH) {
    double val = A_col[entry];
    for(int j = 0; j < PR_WIDTH; ++j) {
      double ywt = 0;
      for(int k = 0; k < PC_WIDTH; ++k) {
        ywt += Y_shared[entry + k * PR_WIDTH] * W_shared[j + k * PR_WIDTH];
      }
      val += ywt * A_col[j];
    }
    // can safely write this back immediately
    if(pr + entry < m && block_col + apply_col < n) {
      A[pr + entry + (block_col + apply_col) * m] = val;
    }
  }
}
...
```

Poglavlje 5

Radno okruženje i rezultati

U ovom dijelu donosimo rezultate našeg rada. Implementirali smo serijske algoritme za QR faktorizaciju pomoću klasičnog Gram-Schmidta (CGS) i pomoću Householderovih reflektora i paralelni HRCAQR algoritam. Rezultate smo usporedili i s MATLAB - ovom implementacijom za računanje QR faktorizacije.

5.1 Arhitektura

Algoritmi su testirani na sljedećoj arhitekturi.

GeForce GTX 1050 Ti

GPU Engine Specs	
CUDA Cores:	768
Graphics Clock (MHz):	1290
Processor Clock (MHz):	1392
Graphics Performance:	high-6747
Memory Specs	
Memory Clock:	7 Gbps
Standard Memory Config:	4 GB
Memory Interface:	GDDR5
Memory Interface Width:	128-bit
Memory Bandwidth (GB/sec):	112
Feature Support	
Supported Technologies:	CUDA, 3D Vision, PhysX, NVIDIA G-SYNC™, Ansel

Intel Core i7-7700HQ Processor

`Performance`	
# of Cores:	4
# of Threads:	8
Processor Base Frequency:	2.80 GHz
Max Turbo Frequency	3.80 GHz
Cache	6 MB
Bus Speed	8 GT/s DMI
`Memory Specifications`	
Max Memory Size (dependent on memory type):	64 GB
Memory Types:	DDR4-2400, LPDDR3-2133, DDR3L-1600
Max # of Memory Channels	2
Max Memory Bandwidth	37.5 GB/s
ECC Memory Supported:	No

Ostalo

OS:	Windows 10 Pro
System Type:	64-bit Operating system, x64-based proccesor
Installed Memory (RAM):	16.0GB (15.9GB usable)
CUDA version:	V10.0.130

5.2 Rezultati

U ovom djelu prikazujemo rezultate serijskih `QR` algoritama pomoću klasičnog Gram-Schmidta (`CGS`) i Householderovih reflektora, pararelni `HRCAQR` i `MATLAB` - ovog algoritma. Važno je napomenuti da rezultati ovise o samoj implementaciji algoritama, među parametrima i načinu kontrola veličina.

Konfiguraciju zajedničke memorije (`shared memory`) za `GPU` smo postavili na 8 bajtova (standardna postavka je 4 bajta).

Za početak krenimo s jednostavnijim testiranjem `HRCAQR`.

m	n	$r_{m,n}$	PR_WIDTH	PC_WIDTH	rpg	cpu_time(sec)
100	100	1	64	32	$6.4229e-15$	0.001000
512	256	2	64	32	$2.4700e-14$	0.000000
512	512	1	64	32	$3.7087e-14$	0.001000

Tablica 5.1: Rezultati testiranja `HRCAQR - a`. Testiranje relativne povratne greške ($rpg = ||A - QR|| \,/\, ||A||$).

m	n	$r_{m,n}$	cpu_time(sec)
1024	256	4	0.328000
1024	512	2	1.296000
2048	256	8	0.641000
2048	512	4	2.641000
4096	256	16	1.328000
4096	512	8	5.281000
8192	256	32	2.640000
8192	512	16	10.531000
8192	1024	16	42.047000

Tablica 5.2: Rezultati testiranja serijskiog QR - a pomoću klasičnog Gram-Schmidta (CGS).

m	n	$r_{m,n}$	cpu_time(sec)
1024	256	4	0.250000
1024	512	2	0.906000
2048	256	8	0.516000
2048	512	4	1.953000
4096	256	16	1.062000
4096	512	8	4.141000
8192	256	32	2.156000
8192	512	16	8.469000
8192	1024	16	32.953000

Tablica 5.3: Rezultati testiranja serijskog QR - a pomoću Householderovih reflektora.

m	n	$r_{m,n}$	PR_WIDTH	PC_WIDTH	cpu_time(sec)
1024	256	4	64	32	0.001000
1024	512	2	64	32	0.003000
2048	256	8	64	32	0.003000
2048	512	4	64	32	0.004000
4096	256	16	64	32	0.005000
4096	512	8	64	32	0.010000
8192	256	32	64	32	0.010000
8192	512	16	64	32	12.992000
8192	512	16	128	16	0.011000
8192	1024	8	128	16	32.347000
8192	1024	8	128	32	0.002000

Tablica 5.4: Rezultati testiranja HRCAQR - a.

m	n	$r_{m,n}$	cpu_time(sec)
1024	256	4	0.024600
1024	512	2	0.317700
2048	256	8	0.078500
2048	512	4	0.378100
4096	256	16	0.255300
4096	512	8	0.426400
8192	256	32	1.153300
8192	512	16	1.658900
8192	1024	16	2.456700

Tablica 5.5: Rezultati testiranja MATLAB - ovog QR-a.

5.3 Zaključak

Kao što možemo vidjeti u djelu 5.2 implementacija HRCAQR je polučila puno bolje rezultate nego serijski algoritmi. U prvom dijelu 5.2 kod testiranja relativne povratne greške (tablica 5.1) svjedočimo točnosti HRCAQR algoritma. Nadalje, u nekim dijelovima polučeni su bolji rezultati čak i od MATLAB - ove implementacije QR algoritma što je zahtjevalo dodatno podešavanje određenih parametara samog HRCAQR algoritma.

m	n	PR_WIDTH	PC_WIDTH	MATLAB	QR CGC	QR Householder
1024	256	64	32	24.6	328	250
1024	512	64	32	105.9	432	302
2048	256	64	32	26.16	213.6	172
2048	512	64	32	94.525	660.25	488.25
4096	256	64	32	51.06	256.6	212.4
4096	512	64	32	42.64	528.1	414.1
8192	256	64	32	115.33	264	215.6
8192	512	64	32	0.128	0.81	0.652
8192	512	128	16	150.81	957.36	769.9
8192	1024	128	16	0.076	1.3	1.02
8192	1024	128	32	1228.35	21023.5	16476.5

Tablica 5.6: Tablica ubrzanja HRCAQR algoritma u odnosu na ostale iz odjeljka 5.2.

Kao što smo napomenuli i ranije, HRCAQR je dosta ograničen i manje skalabilan i tu ima puno prostora za poboljšanje.

Bibliografija

[1] *BLAS (Basic Linear Algebra Subprograms)*, http://www.netlib.org/blas/.

[2] Michael Anderson, Grey Ballard, James Demmel i Kurt Keutzer, *Communication-Avoiding QR Decomposition for GPUs.*, (2011), http://www.netlib.org/lapack/lawnspdf/lawn240.pdf.

[3] Austin R. Benson, David F. Gleich i James Demmel, *Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures*, CoRR **abs/1301.1071** (2013), http://arxiv.org/abs/1301.1071.

[4] NVIDIA Corporation, *CUDA*, https://developer.nvidia.com/cuda-zone.

[5] Gene H. Golub i Charles F. Van Loan, *Matrix Computations (3rd Ed.)*, Johns Hopkins University Press, Baltimore, MD, USA, 1996, ISBN 0-8018-5414-8.

[6] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo i Jack Dongarra, *Enhancing Parallelism of Tile QR Factorization for Multicore Architectures*, (2010), https://www.researchgate.net/publication/255572539_Enhancing_Parallelism_of_Tile_QR_Factorization_for_Multicore_Architectures.

[7] Saša Singer i Nela Bosner, *Numerička analiza 4. predavanje*, Sveučilište u Zagrebu, PMF - Matematički odjel, 2011.

[8] Zlatko Drmač i Vjeran Hari i Miljenko Marušić i Malden Rogina i Sanja Singer i Saša Singer, *Numerička analiza*, Sveučilište u Zagrebu, PMF - Matematički odjel, 2003, https://web.math.pmf.unizg.hr/~singer/num_mat/num_anal.pdf.

[9] Andrew Kerr, Dan Campbell i Mark Richards, *QR Decomposition on GPUs*, Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), GPGPU-2, ACM, 2009, str. 71–78, ISBN 978-1-60558-517-8, http://doi.acm.org/10.1145/1513895.1513904.

[10] Cleve Moler, *Householeder Reflections and the QR Decomposition*, (2016), https://blogs.mathworks.com/cleve/2016/10/03/householder-reflections-and-the-qr-decomposition/.

[11] Univ. of Tennessee, Univ. of California, Berkeley, Univ. of Colorado Denver, i NAG Ltd.., *LAPACK (Linear algebra PACKage)*, http://www.netlib.org/lapack/.

[12] Saša Singer, *Numerička matematika 8. predavanje*, Sveučilište u Zagrebu, PMF - Matematički odjel, 2019.

Sažetak

U ovom radu opisujemo serijske i paralelne algoritme za računanje QR faktorizacije. Opisujemo serijski algoritam za računanje QR faktorizaciju pomoću klasičnog i modificiranog Gram-Schmidtovog postupka te serijski algoritam koji koristi Householderove reflektore za računanje već spomenute faktorizacije. Opisujemo paralelne algoritme za računanje QR faktorizacije kao što su TSQR, CAQR i HRCAQR koja je ograničena varijanta CAQR. Neke od algoritama smo implementirali koristeći C programski jezik, a za paralelni algoritam smo koristili C s CUDA sučeljem. Na kraju rada su pokazani rezultati implementiranih algoritama uspoređeni s MATLAB - ovom implementacijom QR algoritama. Cijeli rad smo zaključili s diskusijom o dobivenim rezultatima.

Summary

In this paper, we describe serial and parallel algorithms for computing QR factorization. We describe a serial algorithm for computing QR factorization using the classical and modified Gram-Schmidt methods and a serial algorithm that uses Householder reflectors to calculate the factorization that we have already mentioned. We describe parallel algorithms for computing QR factorization such as TSQR, CAQR, and HRCAQR which is a restricted variant of CAQR. Some of the algorithms were implemented using the C programming language, and for the parallel algorithm, we used C with the CUDA interface. At the end of the paper, the results of the implemented algorithms are shown compared to MATLABs implementation of QR algorithms. We concluded the whole paper with a discussion of the obtained results.

Životopis

Vedran Rukavina rođen je 29. ožujka 1991. u Virovitici. Pohađao je osnovnu školu Josipa Kozarca u Slatini, a nakon toga upisuje Prirodoslovno-matematičku gimnaziju Petra Preradovića u Virovitici, te maturira 2010. godine. Iste godine upisuje preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu (PMF) Sveučilišta u Zagrebu. 2015. godine završava preddiplomski studij matematike i postaje sveučilišni prvostupnik (baccalaurea) matematike. Iste godine u jesen, upisuje diplomski studij Primijenjene matematike na PMF-u u Zagrebu. Od 15. ožujka 2016. zaposlen (studentski) u Ericsson Nikola Tesla d.d..