Pronalaženje najkraćih putova u grafu korištenjem hibridne CPU-GPU platforme

Karaula, Mislav

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike i računarstva

Mislav Karaula

Pronalaženje najkraćih putova u grafu korištenjem hibridne CPU-GPU platforme

Diplomski rad

Osijek, 2017.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike i računarstva

Mislav Karaula

Pronalaženje najkraćih putova u grafu korištenjem hibridne CPU-GPU platforme

Diplomski rad

Mentor: doc. dr. sc. Domagoj Ševerdija

Osijek, 2017.

Sadržaj

1 UVOD

Traženje najkraćih putova u grafu postala je svakodnevnica svake osobe u modernom društvu, bili mi toga svjesni ili ne. Razlog tomu je što se podaci raznih problema iz naše okoline mogu modelirati grafovima, a rješavanje istih je tada ekvivalentno traženju najkraćeg puta u grafu. Primjeri ovakvih problema uključuju navigaciju na mobilnim uređajima ili u automobilima, primjene na društvenim mrežama, prometnicama, itd.

Dakle, bitno je pronaći optimalne algoritme za rješavanje problema koji se mogu modelirati kao traženje najkraćih putova u grafu. Ovaj rad, nakon osnova teorije grafova potrebnih za definiranje daljnjih pojmova iznesenih u *Poglavlju 2*, bavi se upravo navedenom problematikom, pa je tako detaljno analiziran Floyd-Warshall algoritam u *Poglavlju 3* kao rješenje problema najkraćih putova među svim parovima vrhova nekog grafa.

U svrhu vremenski što efikasnijeg izvršavanja Floyd-Warshall algoritma na nekom danom grafu, osim `C++` implementacije koja je analogon algoritmu pisanom u pseudojeziku, u radu je dalje analizirana hibridna CPU-GPU platforma, gdje se uz pomoć `NVIDIA CUDA` arhitekture došlo do efikasne paralelne implementacije ovog algoritma, što je predstavljeno u *Poglavlju 3*.

U zadnjem poglavlju su izneseni eksperimentalni rezultati, te su u *Prilogu* priloženi kôdovi svih triju verzija implementacija Floyd-Warshall algoritma.

2 TEORIJA GRAFOVA

2.1 Osnovni pojmovi u teoriji grafova

U ovom poglavlju definira se teorijska podloga matematičkog područja teorije grafova kao baza za rješavanje problema najkraćih putova u grafu, koji je tema ovog rada.

Prema [8]:

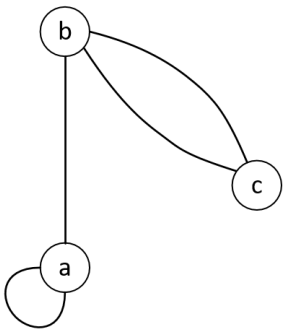
Definicija 1. Graf G je uređena trojka $G = (V(G), E(G), \psi_G)$ koja se sastoji od nepraznog skupa $V = V(G)$, čiji su elementi vrhovi od G, skupa $E = E(G)$ disjunktnog sa $V(G)$, čiji su elementi bridovi od G i funkcije incidencije ψ_G koja svakom bridu od G pridružuje neuređeni par (ne nužno različitih) vrhova od G.

Ako $u, v \in V(G)$ i $e \in E(G)$ tako da $\psi_G(e) = \{u, v\}$, kaže se da e spaja u i v, a u i v su krajevi od e. Za krajeve u, v brida e kaže se da su **incidentni** sa bridom e. Štoviše, v i e su incidentni ako je v jedan kraj brida e. Nadalje, dva vrha se nazivaju **susjednima** ukoliko su incidentni s istim bridom, a dva brida susjednima ukoliko su incidentni s istim vrhom.

Iz *Definicije 1* slijede sljedeće tvrdnje:

- $|V| \geq 1$, tj. graf mora sadržavati najmanje jedan vrh
- moguće je da $E = \emptyset$, tj. graf ne mora sadržavati bridove
- skupovi $V(G)$ i $E(G)$ ne moraju biti konačni; ukoliko su oba skupa konačna, tada kažemo da je G konačan graf
- ako $e, f \in E$, $e \neq f$, i $u, v \in V$, $u \neq v$, tada je moguće da $\psi(e) = \psi(f) = \{u, v\}$, tj. funkcija ψ ne mora biti injekcija (moguće je da dva različita vrha budu spojena sa više bridova, tj. da graf sadrži višestruke bridove); ukoliko je to slučaj, tada graf G zovemo **multigrafom**
- ako $e \in E$ i $v \in V$, tada je moguće da $\psi(e) = \{v, v\}$, tj. brid može spajati neki vrh sam sa sobom i zove se **petlja**.

Primjer 1. *Neka je dan graf G kao na Slici 2.1. Može se primijetiti da je dani graf sa skupom vrhova $V = \{a, b, c\}$ i bridovima $\psi(e_1) = \{a, a\}$, $\psi(e_2) = \{a, b\}$, $\psi(e_3) = \{b, c\}$ i $\psi(e_4) = \{b, c\}$ multigraf u kojemu postoji petlja.*



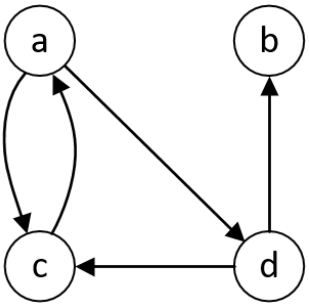
Slika 2.1: Jednostavan primjer multigrafa s petljom

Obzirom da se algoritmi za najkraće putove korišteni u ovom radu fokusiraju na usmjerene grafove, prema [8]:

Definicija 2. Usmjereni graf ili **digraf** G je uređena trojka $(V(G), A(G), \psi_G)$ koja se sastoji od nepraznog skupa $V(G)$ vrhova, skupa $A(G)$ lukova (ili usmjerenih bridova) i funkcije incidencije ψ_G koja svakom luku a pridružuje uređeni par (ne nužno različitih) vrhova u, v spojenih sa a. Vrh u je početni, a v krajnji vrh od a.

Primjedba 1. Svi iskazi u nastavku rada mogu se odnositi na graf, kao i na usmjereni graf (digraf).

Primjer 2. *Neka je dan usmjereni graf G kao na Slici 2.2, te neka su $u, v \in V$ neka njegova dva vrha. Vodeći se opće prihvaćenim oznakama, usmjerene strijelice među parovima vrhova predstavljaju usmjerene bridove. Obzirom na prirodu usmjerenih grafova, bitno je naglasiti da oni mogu sadržavati brid $\{u, v\}$, kao i brid $\{v, u\}$ koji je suprotnog smjera.*



Slika 2.2: Jednostavan primjer usmjerenog grafa

Nadalje, iz [8]:

Definicija 3. Šetnja u grafu G je netrivijalan konačan niz $W := v_0e_1v_1e_2v_2 \dots e_kv_k$ čiji su članovi naizmjence vrhovi v_i i bridovi e_i, tako da su krajevi od e_i vrhovi v_{i-1} i v_i, za svako i, $1 \leq i \leq k$. Broj k zove se **duljina šetnje** W, a šetnja je **zatvorena** ako je $v_0 = v_k$. Ukoliko su svi bridovi u šetnji W međusobno različiti, onda se W zove **staza** duljine k, a ako su i svi vrhovi međusobno različiti, W se naziva **putom** duljine k.

Primjedba 2. Česta alternativna notacija za šetnju, stazu ili put $W = v_0e_1v_1 \dots e_kv_k$ u grafu G je oblika $v_0 \to v_1 \to \dots \to v_{k-1} \to v_k$, ukoliko između svaka dva vrha u danom nizu postoji jedinstveni (usmjereni) brid.

Definicija 4. Vrh u može **dohvatiti** vrh v (i v je **dohvatljiv** iz u) ako postoji put koji počinje u vrhu u i završava u vrhu v.

Definicija 5. Udaljenost $d_G(u, v)$ dvaju vrhova $u, v \in V(G)$ je duljina najkraćeg (u, v)-puta u G. Ako ne postoji takav put u G, dogovorno se piše $d_G(u, v) = \infty$. Ako $u = v$, (u, v)-put je trivijalan. Dva vrha u grafu G su **povezana** ako postoji (u, v)-put u G, a neprazan graf G se naziva **povezanim grafom** ako između svaka dva para vrhova iz G postoji put. Graf G za kojega ovo ne vrijedi naziva se **nepovezanim grafom**.

Također, iz [4] se definira:

Definicija 6. Neka je $v_0e_0v_1 \dots v_{k-1}e_{k-1}v_k$ put i $k \geq 2$. Ako od vrha v_k prema vrhu v_0 postoji brid e_k, tada se graf $C := v_0e_0v_1 \dots v_ke_kv_0$ zove **ciklus**.

Definicija 7. Potpun graf je jednostavan graf u kojem je svaki par vrhova spojen bridom. Oznaka za potpun graf s n vrhova je K_n. Kažemo da je graf **planaran** ako ga je moguće nacrtati u ravnini tako da mu se bridovi sijeku samo u vrhovima. **Aciklički graf** ili **šuma** je graf koji ne sadrži cikluse, a **stablo** je povezan aciklički graf.

Primjedba 3. Graf G naziva se **rijetkim** ako je broj bridova puno manji od kvadrata broja vrhova. Inače, graf se naziva **gustim** grafom. Često se **gustoća** D **usmjerenog grafa** G računa kao:

$$\rho(G) = \frac{|E(G)|}{|V(G)| \cdot (|V(G)| - 1)},$$

pri čemu $D(G) \in [0, 1]$.

2.2 Težinski grafovi

Obzirom da algoritmi za pronalaženje najkraćih putova pretpostavljaju težinski graf, prema [8] se definira:

Definicija 8. Težinski graf G^w je graf G čijim su bridovima pridruženi neki realni brojevi, tj. postoji težinska funkcija $w : E(G) \to \mathbb{R}$ pri čemu broj $w(e)$ zovemo težinom brida $e \in E(G)$. Ako promotrimo sve (u, v)-putove u G^w, $u, v \in V(G^w)$, tada broj

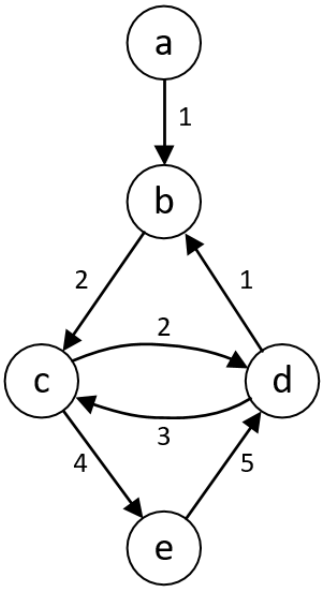
$$d_{G^w}(u, v) = \min\{w(W) : \text{W je put od vrha } u \text{ do vrha } v\}$$

zovemo **minimalna težinska udaljenost** između vrhova u i v. Ukoliko (u, v)-put ne postoji, tada $d_{G^w}(u, v) = \infty$.

Primjer 3. *Neka je dan težinski usmjereni graf G kao na Slici 2.3 sa skupom vrhova $V = \{a, b, c, d, e\}$, skupom bridova $E = \{e_{a,b}, e_{b,c}, e_{c,d}, e_{c,e}, e_{d,b}, e_{d,c}, e_{e,d}\}$, te pripadnim težinama $w(e_{a,b}) = 1, w(e_{b,c}) = 2, w(e_{c,d}) = 2, w(e_{c,e}) = 4, w(e_{d,b}) = 1, w(e_{d,c}) = 3$ i $w(e_{e,d}) = 5$. Na njemu se jednostavno mogu vidjeti primjeri pojmova u iskazanim Definicijama 3, 5, 6 i 8. Tako je moguće zaključiti, primjerice, sljedeće:*

- *jedan put u grafu G duljine 4 ($d_G(a, d) = 4$) je put $a \to b \to c \to e \to d$, te je težinska udaljenost za taj put jednaka 12, ali treba primijetiti kako to nije minimalna težinska udaljenost između početnog i krajnjeg vrha tog puta; minimalna težinska udaljenost između vrhova a i d je 5, te se ona postiže putom $a \to b \to c \to d$*
- *jedan ciklus u grafu G je staza $b \to c \to d \to b$, a težinska udaljenost ovog ciklusa je 5*
- *kako ne postoji (e, a)-put, prema Definicijama 6 i 8, vrijedi: $d_G(e, a) = \infty$ i $d_{G^w}(e, a) = \infty$.*

Do sličnih zaključaka moguće je doći i za druge primjere putova, staza i ciklusa na danom težinskom usmjerenom grafu.



Slika 2.3: Primjer težinskog grafa

2.3 Problem najkraćeg puta i njegove varijante

Sukladno *Definiciji 8*, upravo minimalna težinska udaljenost je predmet rješavanja problema najkraćeg puta koji se, prema [8], definira kao:

Definicija 9. Neka je dan povezan graf G^w sa težinskom funkcijom $w : E(G) \to \mathbb{R}$. Potrebno je odrediti $d_{G^w}(u, v)$, tj. minimalnu težinsku udaljenost između vrhova u i v u grafu G^w. Ovakav problem naziva se problem najkraćeg puta.

Bridovi s negativnim težinama u ovakvim problemima su dozvoljeni, sve dok ne postoje ciklusi s negativnom težinom u kojima se nalaze vrhovi dohvatljivi iz izvorišnog vrha. Razlog tomu je što, ukoliko postoji ciklus s negativnom težinom, traženje najkraćeg puta između vrhova $u, v \in V(G)$ među kojima se takav ciklus nalazi bi moglo rezultirati beskonačnom petljom gdje bi vrijedilo da je $d_{G^w}(u, v) = -\infty$.

Varijante problema najkraćeg puta u (usmjerenom) grafu, prema [3], su:

- najkraći put iz jednog izvora (engl. *Single Source Shortest Path - SSSP*) - problem u kojemu je potrebno pronaći najkraći put iz izvorišnog vrha $s \in V$ do preostalih vrhova grafa
- najkraći put do jednog odredišta (engl. *Single Destination Shortest Path - SDSP*) - problem u kojemu je potrebno pronaći najkraći put od svih vrhova grafa do jednog odredišta $t \in V$
- najkraći put između dva vrha (engl. *Single Pair Shortest Path - SPSP*) - problem u kojemu je potrebno pronaći najkraći put između dva dana vrha u i v, gdje su $u, v \in V$
- najkraći put među svim parovima vrhova (engl. *All Pairs Shortest Paths - APSP*) - problem u kojemu je potrebno pronaći najkraći put od u do v, $\forall u, v \in V$.

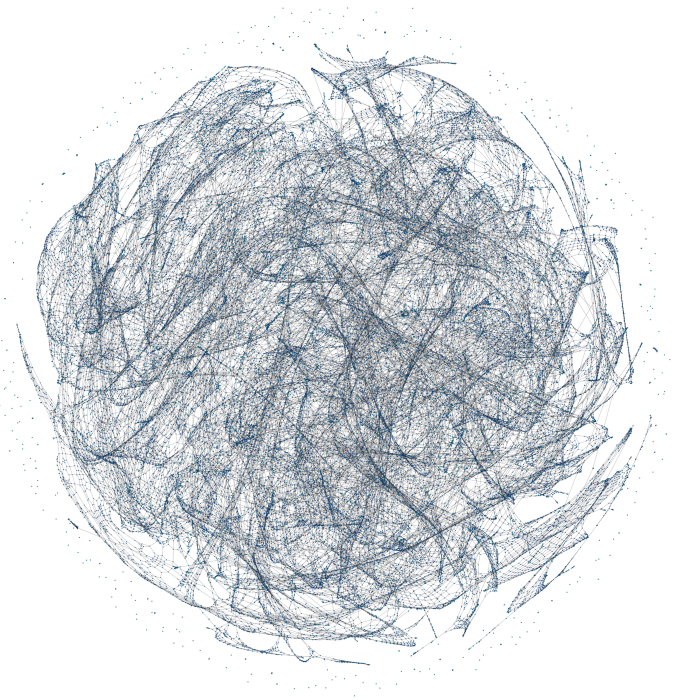
U nastavku ovog rada biti će analizirana posljednja navedena varijanta problema najkraćeg puta, odnosno problem najkraćih putova među svim parovima vrhova u grafu.

2.3.1 Primjene algoritma najkraćih putova među svim parovima vrhova

Mnogobrojni problemi mogu se modelirati kao grafovi, njihova čvorišta kao vrhovi i neke veze među čvorištima kao bridovi, pa se često rješenja ovakvih problema, osim modeliranja, svode na traženje najkraćih putova među svim parovima vrhova. U nastavku su navedeni neki takvi problemi.

Cestovne mreže: Raskrižja se mogu modelirati vrhovima, a ceste bridovima. Također, bridovima se mogu pridružiti težine u obliku očekivanih vremena putovanja ili udaljenosti. Kako cestovne mreže često sadrže razne mostove i tunele, pripadajući grafovi obično nisu planarni. Jedna od karakteristika cestovnih mreža je da je na svakom raskrižju broj ulica koje se sijeku relativno malen, pa je, primjerice, planiranje efikasnih ruta pri navigaciji vrlo izazovno jer modeli cestovnih mreža mogu biti iznimno veliki (imati milijune vrhova) i dinamični (očekivana vremena putovanja su podložna utjecaju raznih faktora poput trenutne situacije u prometu ili stanja održavanja cesta). Takav primjer je i *Slika 2.4*, gdje se vidi graf cestovnih mreža grada Chicaga.

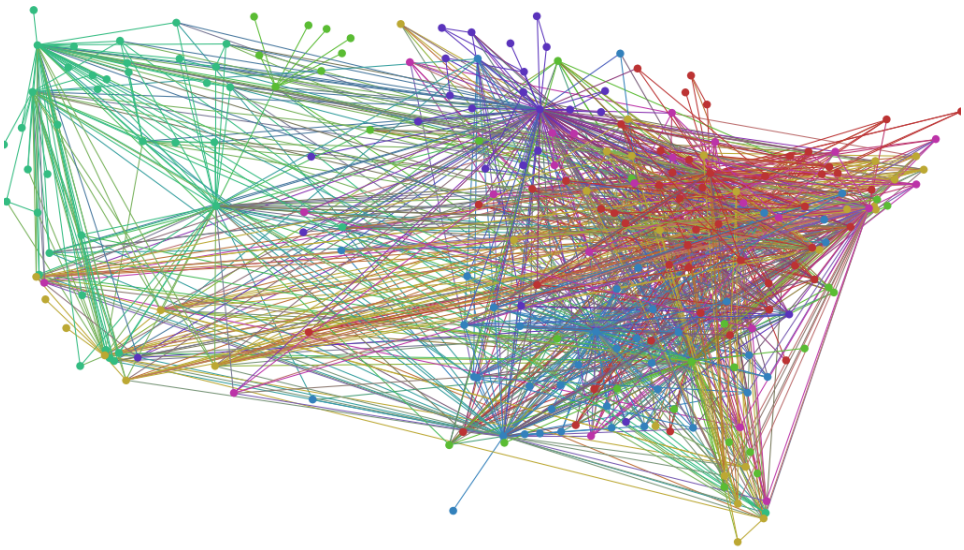
Željezničke mreže: Analogan model onome za cestovne mreže bio bi modeliranje željezničkih stanica vrhovima, a pruga bridovima, dok bi putovanje vlakom bilo reprezentirano praćenjem brida između dva vrha. Bridovima se i ovdje mogu pridružiti težine u obliku očekivanih vremena putovanja ili udaljenosti. No, postoje i realističniji modeli. U jednom takvom su, primjerice, vrhovima modelirani ujedno i lokacija i određeno vrijeme. Jedna stanica tada predstavlja više vrhova u grafu, dok bridovi imaju više tipova - praćenje brida između dva vrha reprezentira putovanje vlakom, čekanje na stanici ili šetnju do drugog kolosijeka na istoj stanici. Težine bridova bi mogle predstavljati vremena putovanja ili cijene karata. Općenito gledajući, modeliranje željezničke mreže je puno kompleksnije od modeliranja cestovne mreže.



Slika 2.4: Cestovna mreža Chicaga *(Izvor: The University of Illinois at Chicago*[1]*)*

[1] `http://uic.edu/`, 20. 11. 2016.

Zrakoplovne mreže: Veze među zračnim lukama također se mogu modelirati grafovima: zračne luke se modeliraju vrhovima, postojanje zračnih linija među parovima zračnih luka bridovima, a vremena putovanja težinama bridova. U ovakvom grafu, skoro svi vrhovi su povezani. Za razliku od cestovnih mreža gdje se u jednom vrhu spaja mali broj bridova, zračne luke mogu imati i stotine zračnih linija za različite destinacije, pa su grafovi kojima se modeliraju zrakoplovne mreže iznimno gusti. U takvim mrežama većina vrhova je incidentna sa samo nekoliko bridova, ali od svakog pojedinog vrha postoji (kratak) put do svih drugih vrhova jer ima nekoliko vrhova koji su incidentni sa jako puno bridova (a time su im i većina drugih vrhova susjedi). Interpretacija navedenoga u ovom problemu je da se avio kompanije većinom odlučuju imati mali broj zračnih luka, čvorišta, kroz koje će teći većina njihovog prometa, a sve ostale zračne luke imaju direktne linije do takvih čvorišta. Ovo je primjetno i na *Slici 2.5* na kojoj je prikaz modela ovog problema za zračne luke u Sjedinjenim Američkim Državama. Boje predstavljaju zračnu udaljenost, pri čemu je sve što je jednake boje također i jednako udaljeno od grada Chicaga.



Slika 2.5: Mreža zračnih luka u SAD-u *(Izvor: Stanford University*[2]*)*

Citiranje među znanstvenim radovima: Graf citata među znanstvenim radovima može se jednostavno modelirati tako da vrh koji je pridružen jednom radu ima usmjereni brid prema vrhu pridruženom nekom drugom radu, ako i samo ako se u prvom radu citira drugi. Citiranje među znanstvenim radovima se može smatrati mjerom utjecaja, jer rad koji se citira u puno drugih radova je zasigurno imao utjecaj na mnoge znanstvenike ili istraživače. Jednu od zanimljivijih primjena istoga pogledati u [16].

Internet mreže: Usmjerivači se u Internet mrežama modeliraju vrhovima, a kablovi bridovima. Iako je infrastruktura vrlo jasna, ne postoji precizna mapa Interneta, ali ju znanstvenici pokušavaju što vjernije prikazati na način da šalju signale nasumičnim računalima, te prate kroz koje je sve rutere signal prošao.

Društvene mreže: Struktura društvenih mreža sastoji se od ljudi koji se modeliraju vrhovima i njihovih odnosa koji se modeliraju bridovima. Društvene mreže generiraju

[2] `http://stanford.edu/`, 20. 11. 2016.

ogromne količine podataka koje su od interesa prodajnim stručnjacima. Upravo ti podaci se koriste za ciljani marketing, te se ponekad, nakon anonimizacije podataka, oni ustupe i znanstvenicima radi istraživanja.

Baze podataka: Zbog rastućeg broja problema koje je lako modelirati kao grafove, u razvoju softvera se sve više počinju koristiti graf baze podataka. Ideja je povećati skalabilnost sustava ukoliko dođe do naglog povećanja obima podataka, te koristeći strukturu grafa efikasno izvršavati upite nad podacima. Detaljnije pogledati u [19].

Navedeni problemi su samo neki od primjera gdje se može postići računarsko poboljšanje prikazom problema u obliku grafa, te primjenom algoritma najkraćih putova među svim parovima vrhova kako bi se dobili neki zaključci.

2.3.2 Neki algoritmi za rješavanje problema najkraćih putova među svim parovima vrhova

Općenito, kako je navedeno u *Poglavlju 2.3*, problem najkraćih putova među svim parovima vrhova u grafu G svodi se na računanje udaljenosti između svih parova vrhova (u, v), $u, v \in V(G)$. Neka je $n := |V(G)|$. Kako postoji $\binom{n}{2} = \Theta(n^2)$ parova vrhova za koje se trebaju naći najkraći putovi, vremenska složenost mora biti barem $\Omega(n^2)$. Rješavanju ovoga problema može se pristupiti sa dva stajališta: rješavanjem problema najkraćeg puta iz jednog izvora za svaki vrh zasebno, ili direktnim rješavanjem problema najkraćih putova među svim parovima vrhova.

Prvim pristupom, uzmemo li Dijkstrin algoritam za rješavanje problema najkraćeg puta iz jednog izvora koji ima složenost $\mathcal{O}(|E(V)| + |V(G)| \cdot \log |V(G)|)$ (vidi: [5] i [11]), za rješenje problema najkraćih putova među svim parovima vrhova dobivamo ukupnu složenost: $\mathcal{O}(|E(G)| \cdot |V(G)| + |V(G)|^2 \cdot \log |V(G)|)$. Štoviše, za vrlo rijetke grafove za koje vrijedi $|E(G)| = \mathcal{O}(|V(G)|)$, vrijeme se svede na $\mathcal{O}(|V(G)|^2 \log |V(G)|)$.

No, ukoliko se ne radi o vrlo rijetkom grafu, moguće je napraviti i bolje rješenje. Taj drugi pristup se, za razliku od onog prvog, ne bazira na usporedbi putova, već na dinamičkom programiranju[3]. Najpoznatiji takav algoritam je Floyd-Warshallov algoritam kojemu je složenost $\mathcal{O}(|V(G)|^3)$, a koji će biti detaljnije opisan u *Poglavlju 3.*

Vremenom su znanstvenici za problem najkraćih putova među svim parovima vrhova u grafu pronalazili algoritme efikasnije od već navedenih. Tako je T. Hagerup 2000. godine definirao algoritam u $\mathcal{O}(|E(G)| \cdot |V(G)| + |V(G)|^2 \cdot \log \log |V(G)|)$ vremenu pri čemu težine bridova moraju biti prirodni brojevi (detaljnije vidi [7]), S. Pettie 2004. godine algoritam u jednakom vremenu, ali za razliku od Hagerupovog algoritma težine bridova mogu biti realni brojevi ukoliko ne postoje negativni ciklusi (detaljnije vidi [20]), a R. Williams je 2014. godine definirao algoritam u $\mathcal{O}(|V(G)|^3/2^{\Omega(\log n)^{1/2}})$ vremenu u kojemu težine bridova moraju biti prirodni brojevi (detaljnije vidi [27]).

2.3.3 Reprezentacija grafova

Grafove je potrebno reprezentirati u obliku koji je pogodan za spremanje u računalu. Najčešće se tu radi o matrici susjedstva i njenim težinskim varijantama. Radi toga, iz [4] se

[3]Za detaljniji opis dinamičkog programiranja pogledati *Poglavlje 3.*

definira:

Definicija 10. Ako je G proizvoljan graf sa skupom vrhova $V(G) = \{v_1, v_2, \ldots, v_n\}$ i skupom bridova $E(G) = \{e_1, e_2, \ldots, e_n\}$, matrica susjedstva $B(G) = [b_{i,j}]_{n\times n}$ grafa G je kvadratna $n \times n$ matrica za čije elemente vrijedi:

$$b_{i,j} = \begin{cases} 1, & \text{ako je vrh } v_i \text{ incidentan s bridom } e_j \\ 0, & \text{inače} \end{cases}.$$

Definicija 11. Ako je G proizvoljan graf sa skupom vrhova $V(G) = \{v_1, v_2, \ldots, v_n\}$ i skupom bridova $E(G) = \{e_1, e_2, \ldots, e_n\}$, matrica udaljenosti $D(G)$ grafa G je kvadratna $n\times n$ matrica sa elementima $d_G(i, j)$ koji su jednaki broju bridova između vrhova v_i i v_j, $i, j = 1, \ldots, n$.

Primjer 4. *Matrica udaljenosti za graf G sa Slike 2.3, uz pretpostavku da se u redovima i stupcima nalaze redom vrhovi a, b, c, d, e, izgleda kako slijedi:*

$$D(G) = \begin{bmatrix} 0 & 1 & 2 & 3 & 3 \\ \infty & 0 & 1 & 2 & 2 \\ \infty & 2 & 0 & 1 & 1 \\ \infty & 1 & 1 & 0 & 2 \\ \infty & 2 & 2 & 1 & 0 \end{bmatrix}. \tag{1}$$

S druge strane, matrica udaljenosti težinskog grafa na poziciji (i, j) ima težinu brida koja spaja vrhove i i j umjesto broja bridova između tih vrhova. Sljedeća varijanta takve matrice, obzirom na *Definiciju 8*, biti će korištena u implementaciji algoritama za pronalaženje najkraćih putova:

$$D(G^w) = [d_{G^w}(i, j)]_{n\times n}, i, j = 1, \ldots, n.$$

Interpretacija ovakve matrice: na poziciji (i, j) nalazi se minimalna težinska udaljenost između vrhova i i j. Za ovu varijantu matrice udaljenosti težinskog grafa će se dalje u radu koristiti izraz težinska matrica udaljenosti.

Primjer 5. *Nastavno na Primjer 4, težinska matrica udaljenosti grafa G sa Slike 2.3, uz pretpostavku da se u redovima i stupcima nalaze redom vrhovi a, b, c, d, e, izgleda kako slijedi:*

$$D(G^w) = \begin{bmatrix} 0 & 1 & 3 & 5 & 7 \\ \infty & 0 & 2 & 4 & 6 \\ \infty & 3 & 0 & 2 & 4 \\ \infty & 1 & 3 & 0 & 7 \\ \infty & 6 & 8 & 5 & 0 \end{bmatrix}. \tag{2}$$

Treba primijetiti kako najkraći put između neka dva vrha ne mora biti jedinstven. Tako, primjerice, $d_{G^w}(e, c) = 8$ nije jedinstven jer se ista minimalna težinska udaljenost postiže na putu $e \to d \to b \to c$, kao i na putu $e \to d \to c$. Analogno vrijedi i za $d_{G^w}(d, c) = 3$ (putovi $d \to b \to c$ i $d \to c$), kao i za $d_{G^w}(d, e) = 7$ (putovi $d \to b \to c \to e$ i $d \to c \to e$).

Problemi kod kojih su ulazni parametri grafovi poput onoga sa *Slike 2.3*, a izlazi težinske matrice udaljenosti poput one iz *Primjera 5*, upravo su problemi najkraćih putova među svim parovima vrhova u grafu i njihova rješenja. Algoritam za pronalaženje takvog rješenja opisan je u sljedećem poglavlju.

3 FLOYD-WARSHALL ALGORITAM

U ovom poglavlju detaljno će biti analiziran Floyd-Warshall algoritam kao najpoznatiji algoritam za rješavanje problema najkraćih putova među svim parovima vrhova. Kako je on klasičan primjer dinamičkog programiranja, ono će također biti objašnjeno, te će biti prikazana jednostavna CPU implementacija Floyd-Warshall algoritma.

Izvor termina dinamičko programiranje nema skoro nikakvu vezu s današnjim programiranjem. Richard Bellman je prvi upotrebljavao ovaj izraz u 50-im godinama prošlog stoljeća, vrijeme kada je programiranje u današnjem smislu riječi bilo toliko rijetko da nije postojao niti termin za takvu aktivnost. Tada je programiranje značilo planiranje, pa bi dinamičko programiranje bilo planiranje optimalnih procesa u više faza.

Pri rješavanju nekog problema dinamičkim programiranjem, najbitnije pitanje je: "Što je potproblem danog problema?". Dinamičko programiranje se svodi na to da za dani problem treba postojati uređeni niz potproblema i relacija koja pokazuje kako riješiti potproblem koristeći rješenja manjih potproblema, odnosno onih koji su se pojavili prije njega u tom uređenom nizu. Ako postoji ovakav uređeni niz, onda je za bilo koji problem vrlo lako definirati algoritam - jednostavno se iterativno rješavaju potproblemi redom, jedan za drugim.

3.1 Floyd-Warshall algoritam

Kao što je već nekoliko puta spomenuto u ovom radu, Floyd-Warshallov algoritam rješenje je problema najkraćih putova među svim parovima vrhova u usmjerenom težinskom grafu bazirano na dinamičkom programiranju. Ovaj algoritam je, u danas uvriježenom obliku, objavljen 1962. godine od strane Roberta Floyda. No, suštinski je ekvivalentan algoritmima koje su ranije objavili Bernard Roy 1959. godine i Stephen Warshall 1962. godine za traženje tranzitivnog zatvarača grafa[4], te je vrlo sličan Kleene-ovom algoritmu objavljenom 1956. godine koji služi za pretvaranje deterministčkog konačnog automata u regularni izraz[5]. Štoviše, moderna formulacija Floyd-Warshall algoritma, koja se sastoji od tri *for* petlje kao što se može vidjeti u nastavku ovog poglavlja, je prvotno opisana od strane Petera Ingermana 1962. godine. Zbog navedene povijesti, ovaj algoritam se često još naziva i sljedećim imenima: Floydov algoritam, Roy-Warshall algoritam, Roy-Floyd algoritam ili WFI algoritam.

Dakle, za dani graf $G := G^w$ rješenje problema je pronaći težinsku matricu udaljenosti, kao što je navedno u *Primjeru 5.* Za ovaj algoritam uobičajeno se koristi notacija kako slijedi.

Neka je $G := G^w$ graf. Definiramo $W = [w_{i,j}]_{n\times n}$, $i, j = 1, \ldots, n$, kao težinsku matricu gdje je:

$$w_{i,j} = \begin{cases} 0, & \text{ako } i = j \\ \text{težina usmjerenog brida } (i,j), & \text{ako } i \neq j \text{ i } (i,j) \in E(G) \\ \infty, & \text{ako } i \neq j \text{ i } (i,j) \notin E(G) \end{cases}.$$

[4]Često se rješavani problem svodi na provjeru postojanja puta između neka dva vrha, ne obazirući se na težine. Ovakav problem naziva se problem tranzitivnog zatvarača grafa.

[5]Pojmovi deterministčkog konačnog automata i regularnog izraza su izvan domene ovog rada. Za definiciju deterministčkog konačnog automata, vidi *[17], str. 24*, a za definiciju regularnog izraza, vidi *[17], str. 53.*

Ova matrica je, zapravo, inicijalno stanje grafa G obzirom na težine bridova.

Nadalje, neka je $D = [d_{i,j}^{(k)}]_{n\times n}$ matrica najkraćih putova, gdje je $d_{i,j}^{(k)}$ težina najkraćeg puta od vrha i do vrha j, pri čemu se svi posredni vrhovi[6] nalaze u skupu $\{1, 2, \ldots, k\}$.

Vrijedi primijetiti da za najkraći put od vrha i do vrha j, takav da su svi posredni vrhovi iz skupa $\{1, 2, \ldots, k\}$, postoje dvije mogućnosti:

- k nije vrh na danom putu, pa je najkraći takav put duljine $d_{i,j}^{k-1}$
- k je vrh na danom putu, pa je najkraći takav put duljine $d_{i,k}^{k-1} + d_{k,j}^{k-1}$.

Obzirom na te zaključke, $d_{ij}^{(k)}$ se može definirati kao:

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j}, & \text{ako } k = 0 \\ \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}), & \text{ako } k \geq 1 \end{cases}.$$

Definira se i matrica prethodnika $\Pi^{(k)} = [\pi_{i,j}^{(k)}]_{n\times n}$, gdje za prethodnika vrha j na najkraćem putu od vrha i, pri čemu su svi posredni vrhovi iz skupa $\{1, 2, \ldots, k\}$, vrijedi:

$$\pi_{i,j}^{(0)} = \begin{cases} \text{NIL}, & \text{ako } i = j \text{ ili } w_{i,j} = \infty \\ i, & \text{ako } i \neq j \text{ i } w_{i,j} < \infty \end{cases}, \quad \pi_{i,j}^{(k)} = \begin{cases} \pi_{i,j}^{(k-1)}, & \text{ako } d_{i,j}^{(k-1)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \\ \pi_{k,j}^{(k-1)}, & \text{ako } d_{i,j}^{(k-1)} > d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \end{cases}.$$

Koristeći definirane matrice W, D i Π, slijedi Floyd-Warshallov algoritam pisan u pseudojeziku:

Algorithm 1 Floyd-Warshall(G)

```
1: izračunati matrice W i Π^(0)
2: n ← dimenzija kvadratne matrice W
3: D^(0) ← W
4: for k = 1 to n do
5:     for i = 1 to n do
6:         for j = 1 to n do
7:             if d_{i,j}^(k) > d_{i,k}^(k) + d_{k,j}^(k) then
8:                 d_{i,j}^(k) ← d_{i,k}^(k) + d_{k,j}^(k)
9:                 π_{i,j}^(k) ← π_{k,j}^(k)
10:            end if
11:        end for
12:    end for
13: end for
14: return D^(n)
```

[6]Posredni vrh puta $W = v_0e_0v_1 \ldots v_{k-1}e_{k-1}v_k$ je bilo koji vrh osim v_0 i v_k, tj. vrh koji se nalazi u skupu $\{1, 2, \ldots, k\}$.

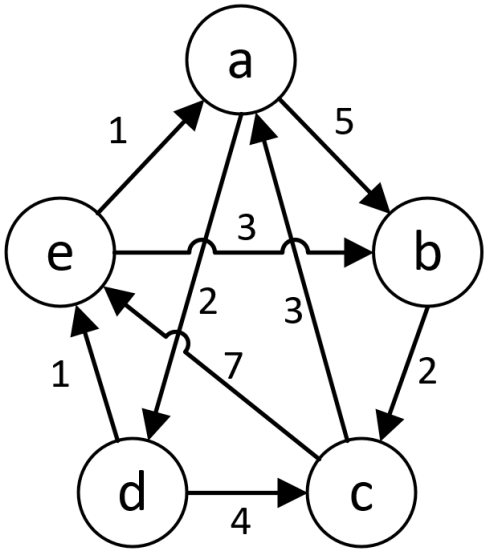
Vremenska složenost algoritma: Vrlo je jednostavno za zaključiti, zbog tri *for* petlje preko svih vrhova grafa, da je vremenska složenost jednaka $\mathcal{O}(n^3)$, pri čemu je n dimenzija kvadratne matrice W, odnosno $n = |V(G)|$. Upravo zbog ove činjenice, što vremenska složenost ne ovisi o broju bridova, je Floyd-Warshallov algoritam jako dobar za guste grafove. Kod rijetkih grafova on gubi tu svoju prednost nad, recimo, prvim navedenim pristupom u *Poglavlju 2.3.2* (pokretanje nekog algoritma kao problem najkraćeg puta iz jednog izvora za svaki vrh posebno).

Dokaz korektnosti algoritma: Dokaz se provodi indukcijom. Cilj je dokazati da je nakon k iteracija u matrici D element $d_{i,j}^{(k)}$ upravo težina najkraćeg puta od i do j koji ne uključuje vrhove iz skupa $\{k+1, \ldots, n\}$.

Baza indukcije: Neka je $k = 0$, tj. još nije prošla niti jedna iteracija algoritma. Tada ne postoje posredni vrhovi na putu između i i j, pa bi trebalo vrijediti $d_{i,j} = w_{i,j}$, što inicijalizacijski korak algoritma upravo i radi.

Pretpostavka indukcije: Pretpostavlja se da je nakon k iteracija element $d_{i,j}^{(k)}$ matrice D upravo težina najkraćeg puta od vrha i do vrha j pri čemu niti jedan posredni vrh nije iz skupa $\{k+1, \ldots, n\}$.

Korak indukcije: Nakon k-te iteracije iz pretpostavke indukcije, u $(k+1)$-oj iteraciji dopušteno je koristiti vrh $k+1$ u bilo kojem putu. Za sve parove vrhova (i, j), težinski najkraći put od i do j koristi vrh $k+1$ ako i samo ako postoji težinski najkraći put od i do $k+1$ i od $k+1$ do j gdje niti jedan posredan vrh nije iz skupa $\{k+2, \ldots, n\}$. No, koristeći pretpostavku indukcije, težinski najkraći put od i do $k+1$, bez korištenja vrhova iz skupa $\{k+2, \ldots, n\}$, je duljine $d_{i,k+1}^{(k+1)}$. Analogno, težinski najkraći put od $k+1$ do j, bez korištenja vrhova iz skupa $\{k+2, \ldots, n\}$, je duljine $d_{k+1,j}^{(k+1)}$. Prema tome, za težinski najkraći put od vrha i do vrha j treba koristiti vrh $k+1$ ako i samo ako vrijedi $d_{i,k+1}^{(k+1)} + d_{k+1,j}^{(k+1)} < d_{i,j}^{(k+1)}$ (jer je najkraći put prije razmatranja vrha $k+1$ bio $d_{i,j}^{(k+1)}$). Kako je ovo upravo ono što se računa u $(k+1)$. iteraciji algoritma, a kako za $k = n$ algoritam terminira, a to znači da se za najkraći put koriste svi vrhovi što je i traženo rješenje, Floyd-Warshallov algoritam je korektan. □



Slika 3.1: Jednostavan primjer težinskog grafa

Primjer 6. *Neka je zadan graf G kao na Slici 3.1, uz pretpostavku da se u redovima i stupcima nalaze redom vrhovi a, b, c, d, e. Floyd-Warshall algoritam je vrlo jednostavan za pratiti, te slijedi:*

$$W = \begin{bmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{bmatrix}$$

Nadalje, za $k = 0$ vrijedi:

$$D^{(0)} = \begin{bmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & \infty & 0 & \infty & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & \infty & 0 \end{bmatrix}, \Pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & \text{NIL} & 1 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} & \text{NIL} \\ 3 & \text{NIL} & \text{NIL} & \text{NIL} & 3 \\ \text{NIL} & \text{NIL} & 4 & \text{NIL} & 4 \\ 5 & 5 & \text{NIL} & \text{NIL} & \text{NIL} \end{bmatrix},$$

za $k = 1$:

$$D^{(1)} = \begin{bmatrix} 0 & 5 & \infty & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & \infty & 3 & 0 \end{bmatrix}, \Pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & \text{NIL} & 1 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} & \text{NIL} \\ 3 & 1 & \text{NIL} & 1 & 3 \\ \text{NIL} & \text{NIL} & 4 & \text{NIL} & 4 \\ 5 & 5 & \text{NIL} & 1 & \text{NIL} \end{bmatrix},$$

za $k = 2$:

$$D^{(2)} = \begin{bmatrix} 0 & 5 & 7 & 2 & \infty \\ \infty & 0 & 2 & \infty & \infty \\ 3 & 8 & 0 & 5 & 7 \\ \infty & \infty & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{bmatrix}, \Pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 2 & 1 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} & \text{NIL} \\ 3 & 1 & \text{NIL} & 1 & 3 \\ \text{NIL} & \text{NIL} & 4 & \text{NIL} & 4 \\ 5 & 5 & 2 & 1 & \text{NIL} \end{bmatrix},$$

za $k = 3$:

$$D^{(3)} = \begin{bmatrix} 0 & 5 & 7 & 2 & 14 \\ 5 & 0 & 2 & 7 & 9 \\ 3 & 8 & 0 & 5 & 7 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{bmatrix}, \Pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 2 & 1 & 3 \\ 3 & \text{NIL} & 2 & 1 & 3 \\ 3 & 1 & \text{NIL} & 1 & 3 \\ 3 & 1 & 4 & \text{NIL} & 4 \\ 5 & 5 & 2 & 1 & \text{NIL} \end{bmatrix},$$

za $k = 4$:

$$D^{(4)} = \begin{bmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 7 & 12 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{bmatrix}, \Pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 1 & 4 \\ 3 & \text{NIL} & 2 & 1 & 4 \\ 3 & 1 & \text{NIL} & 1 & 4 \\ 3 & 1 & 4 & \text{NIL} & 4 \\ 5 & 5 & 2 & 1 & \text{NIL} \end{bmatrix},$$

i za $k = 5$:

$$D^{(5)} = \begin{bmatrix} 0 & 5 & 6 & 2 & 3 \\ 5 & 0 & 2 & 7 & 8 \\ 3 & 8 & 0 & 5 & 6 \\ 2 & 4 & 4 & 0 & 1 \\ 1 & 3 & 5 & 3 & 0 \end{bmatrix}, \Pi^{(5)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 1 & 4 \\ 3 & \text{NIL} & 2 & 1 & 4 \\ 3 & 1 & \text{NIL} & 1 & 4 \\ 5 & 5 & 4 & \text{NIL} & 4 \\ 5 & 5 & 2 & 1 & \text{NIL} \end{bmatrix}.$$

Kao što piše u algoritmu, izlaz, tj. rješenje, je matrica $D^{(5)}$ koja je matrica najkraćih putova između svih parova vrhova u grafu danom na Slici 3.1. Ovaj primjer je relativno malen pa je jednostavno pratiti gore navedene iteracije, kao i provjeriti krajnji rezultat. No, algoritam se analogno računa za bilo koju veličinu grafa, neovisno o broju vrhova. Tako, primjerice, neki grafovi u Poglavlju 5 na kojima će se testirati implementacije ovih algoritama imaju na tisuće vrhova.

3.2 Jednostavna implementacija u `C++` programskom okruženju

Za implementaciju kôda koji je vjerna preslika algoritma pisanog u pseudojeziku danog u *Poglavlju 3.1* korišteno je `C++` programsko okruženje. Radi se o rješenju koje se, za razliku od hibridnih rješenja koja slijede u *Poglavlju 4*, izvršava samo na procesoru računala i u njegovoj radnoj memoriji.

Cjeloviti kôd ove implementacije nalazi se u *Prilogu 1*[7], u kojemu je moguće primijetiti implementirane dodatke logici algoritma pisanog u pseudojeziku, odnosno neke dodatne metode. Radi se o metodama `checkSolutionCorrectness`, te u njoj korištenoj metodi `getPath`, koje za izračunate rezultate koristeći matricu prethodnika i matricu najkraćih putova provjeravaju, za svaki element dobivenog matričnog rješenja, može li se rekurzivno zaista doći do početnog stanja matrice najkraćih putova, odnosno, prema nazivlju iz *Poglavlja 3.1*, do matrice $D^{(0)} := W$. Drugim riječima, provjerava se ispravnost dobivenih rješenja. Identične metode nalaze se i u implementacijama iz *Poglavlja 4.4 (Priloga 2)* i *Poglavlja 4.5 (Priloga 3)*.

Valja naglasiti da su matrice W, D i Π korištene u algoritmu, koje su dimenzija $n \times n$, radi jednostavnosti spremištene kao nizovi duljine n^2. Također, u *Prilogu 1* je vidljivo da se mjeri egzaktno vrijeme izvršavanja algoritma (kôda) za učitani graf, te se krajnji rezultat, odnosno matrice $D^{(n)}$ i $\Pi^{(n)}$, kao i izmjereno vrijeme izvršavanja, zapisuju u datoteku kako bi se rezultati mogli kasnije analizirati u *Poglavlju 5*.

[7] Izvorni kôd dostupan i na repozitoriju `https://github.com/mislavkaraula/cuda_floyd_warshall` (14. 01. 2017.).

4 CUDA - PLATFORMA ZA PARALELNO RAČUNARSTVO

CUDA je platforma za paralelno računanje koja omogućava softver inženjerima i programerima korištenje CUDA-omogućenih grafičkih kartica (engl. *GPU - Graphics Processing Unit*) za programiranje u opće svrhe. Ovakav pristup naziva se GPGPU (engl. *General-Purpose computing on Graphics Processing Units*). CUDA platforma je softverski sloj koji korisniku daje direktan pristup virtualnom skupu instrukcija na grafičkoj kartici i elementima za paralelno računarstvo, što omogućuje izvršavanje tzv. *kernela*[8].

CUDA je dizajnirana da radi s programskim jezicima poput C, C++ i Fortran. Koristeći takve programske jezike, GPU-akcelerirane aplikacije pokreću sekvencijalni dio kôda na procesoru (engl. *CPU - Central Processing Unit*) koji je optimiziran za sekvencijalne zadatke, a GPU koriste za akceleriranje paralelnih procesa. Ovakav način kôdiranja postiže se korištenjem CUDA ključnih riječi koje omogućuju masovnu paralelizaciju dijela aplikacije koji se pokreće kao kernel na GPU. Opisani oblik računarstva se naziva "GPU akcelerirano računarstvo" (engl. *GPU computing*), a sama platforma na kojoj se takve aplikacije pokreću naziva se hibridna CPU-GPU platforma, kako i stoji u imenu ovog diplomskog rada.

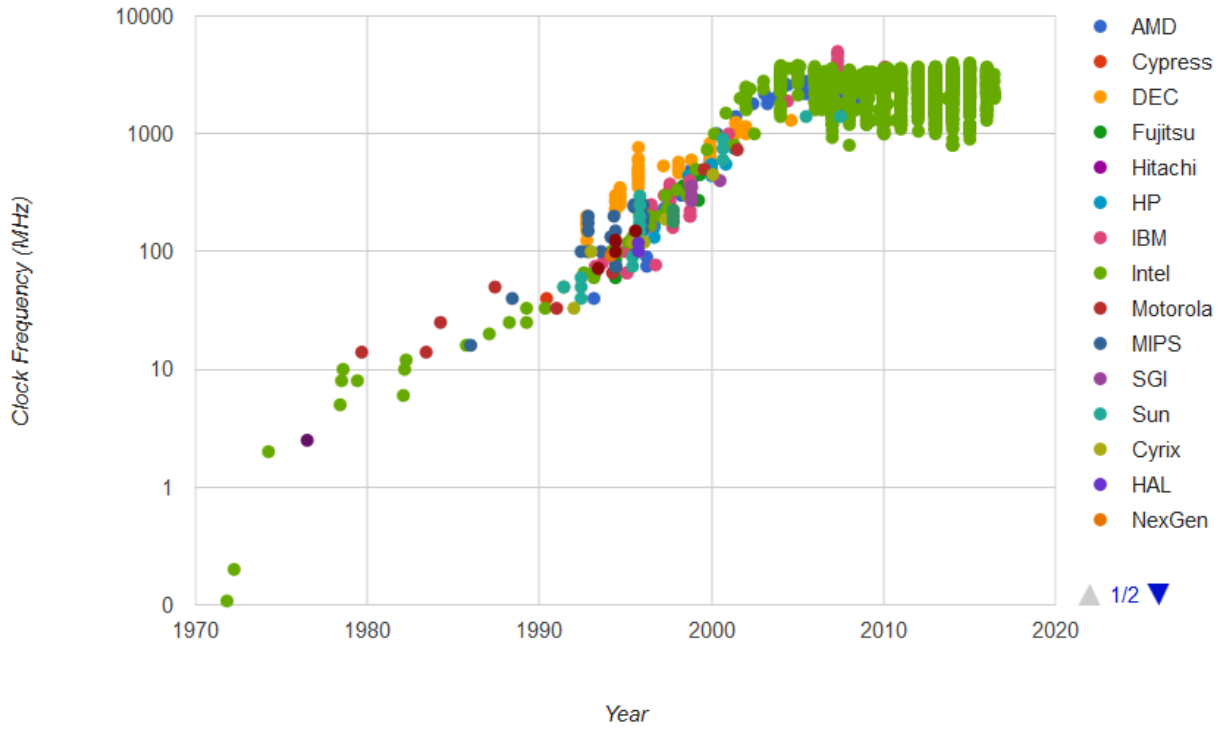
4.1 Povijest paralelnog računarstva

Posljednjih godina u tehnološkoj industriji dogodio se pomak svih vrsta potrošačkih tehnologija na razne oblike paralelnog računarstva koji prestaju biti prisutni samo u superračunalima[9]. Razvoj softvera se okreće razvijanju paralelnih procesa na svim vrstama platformi, te se od, primjerice, današnjih mobilnih uređaja očekuje da simultano mogu pokretati više procesa odjednom - glazbu, navigaciju (GPS servise), Internet, itd.

Oko 30 godina, jedna od najvažnijih metoda poboljšanja performansi uređaja široke uporabe bilo je povećavanje brzine procesora - počevši sa prvim osobnim računalima ranih 80-ih, kada su procesori imali brzinu od oko 1MHz, do današnjih procesora koji imaju brzine između 1GHz i 4GHz (po jezgri), te su skoro tisuću puta brži od prvog osobnog računala. No, posljednjih godina se proizvođači okreću alternativnim načinima poboljšavanja performansi, jer povećanje brzine procesora više nije moguće zbog raznih ograničenja materijala i integriranih krugova od kojih se procesori proizvode (primjerice, zbog toplinskih restrikcija). Proizvođači su se za inspiraciju ugledali na superračunala koja su desetljećima poboljšavala svoje performanse ne samo povećanjem brzine procesora, nego i njihovog broja. 2005. godine vodeći proizvođači procesora počeli su nuditi procesore s dvije jezgre umjesto jedne, da bi kroz sljedećih nekoliko godina pratili takav razvoj performansi, pa danas nije neobično u osobnom računalu imati procesor sa četiri, osam ili čak šesnaest jezgara. Štoviše, bilo bi vrlo izazovno pokušati naći novo računalo u prodaji koje ima jednu jezgru. Na *Slici 4.1* može se vidjeti opisani trend u povećanju brzine (jezgre) procesora, te kako je isti počeo stagnirati oko 2005. godine kada su se proizvođači okrenuli povećanju broja jezgara.

[8] U računarstvu, *kernel* je rutina (metoda) kompajlirana za akceleratore, tj. mikročipove visoke propusnosti, poput grafičkih kartica. *Kerneli* su odvojeni od glavnog programa (kojega najčešće izvršava procesor), no glavni program ih pokreće i koristi.

[9] Superračunalom se naziva računalo koje ima izuzetne performanse obrađivanja velikih količina podataka u odnosu na većinom zastupljena računala za osobnu upotrebu.



Slika 4.1: Brzine procesora kroz posljednjih 40-ak godina *(Izvor: Stanford University*[10]*)*

S druge strane, GPU-ovi su relativno nova platforma za procesiranje podataka naspram cjelokupnog polja računarstva, ali je njihov razvoj doživio naglu revoluciju. U kasnim 80-ima i ranim 90-ima, popularnost grafičkih operativnih sustava poput *Microsoft Windowsa* pomogla je stvoriti tržište za novu vrstu procesorskih jedinica. Otprilike u isto vrijeme, kroz 80-e, tvrtka *Silicon Graphics* koja se profesionalno bavila računarstvom, popularizirala je korištenje trodimenzionalne grafike u razne profesionalne primjene, uključujući znanstvene i tehničke vizualizacije, (tada) zapanjujuće kinematografske efekte, itd. Ista tvrtka je 1992. javnosti predstavila svoju `OpenGL` platformu za koju su htjeli da bude standard za primjenu trodimenzionalne grafike neovisno o operativnom sustavu na kojem se pokreće, pa je bilo samo pitanje vremena kada će se takva tehnologija naći u primjeni u potrošačkoj tehnologiji dostupnoj svima.

Do sredine 90-ih godina, potražnja u potrošačkoj tehnologiji koja zahtjeva 3D grafiku je eskalirala, što je dovelo do dva vrlo značajna razvojna koraka u ovom polju računarstva. Prvo, do razvoja 3D igara za računala koje su dodatno potakle na progresivan razvoj sve realističnijih 3D tehnologija za buduće igre. S druge strane, u isto vrijeme, tvrtke poput *NVIDIA*, *ATI Technologies* i *3dfx Interactive* počeli su proizvoditi grafičke akceleratore koji su bili pristupačni cijenom. Ovi razvojni smjerovi potvrdili su da će 3D grafička tehnologija biti jedno od velikih područja budućeg računarstva, kao što je to primjetno i danas.

Razvoj *NVIDIA GeForce 256* grafičke kartice je pokazao dodatno poboljšanje potrošačkog grafičkog hardvera, jer su se podaci o transformaciji svjetlosti počeli obrađivati direktno na grafičkoj kartici. Kasnije, dolaskom *NVIDIA GeForce 3* serije grafičkih kartica 2001. godine, došlo je do najvažnijeg pomaka u GPU tehnologiji. Naime, to je bio prvi čip koji je implementirao, tada novi, *Microsoft DirectX 8.0* standard, te su po prvi puta softver

[10] `http://cpudb.stanford.edu/`, 03. 12. 2016.

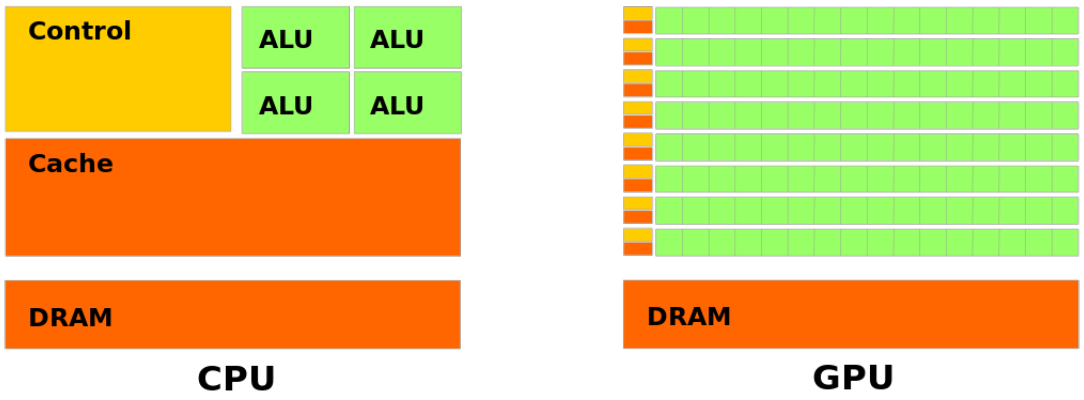
inženjeri imali kontrolu na nekim konkretnim izračunima koji će se izvršavati na grafičkim čipovima.

Pojavljivanje GPU-ova koji su bili programabilni privuklo je mnoge istraživače jer su napokon mogli iskoristiti pravi potencijal ovakve arhitekture. GPU računarstvo prije toga se zasnivalo na, u osnovi, varanju grafičke kartice na način da joj se predavao skup brojeva koji je grafičkoj kartici bio podatak o boji piksela[11], a koji je istraživačima ili softverskim inženjerima bio skup ulaznih podataka nekog algoritma. Nakon toga, s tim podacima su se mogli raditi razni izračuni, da bi na kraju rezultat obrade podataka bio izlazni podatak algoritma - koji se mogao očitati iz boje određenog piksela. Ovakvi trikovi bili su vrlo domišljati, no iznimno zamršeni za postizanje željene obrade podataka.

Oko pet godina nakon predstavljanja *NVIDIA GeForce 3* serije grafičkih kartica, GPU računarstvo je doživjelo nagli napredak. U studenom 2006. godine, *NVIDIA* je predstavila svoju *GeForce 8800 GTX* grafičku karticu, prvu u svijetu pogonjenu s *DirectX 10*. Ova grafička kartica je također i prva bazirana na *NVIDIA CUDA* arhitekturi, koja je uključivala nekoliko novih komponenti dizajniranih upravo radi GPU računarstva u cilju ublažavanja i micanja nekih ograničenja koje su prethodne GPU imale (npr. maskiranje *GPGPU* računanja kao obradu grafike), a koje su bile glavni razlog ne korištenja grafičkih čipova u GPGPU (engl. *General-Purpose computing on Graphics Processing Units*) svrhe.

4.2 `CUDA` arhitektura

Kao što je već spomenuto, cilj `CUDA`-e je omogućavanje softver inženjerima što jednostavniji pristup *NVIDIA* grafičkim karticama za paralelno računanje koje se manifestira hibridnom CPU-GPU platformom. Da bi se kôd namijenjen grafičkom čipu paralelno izvršio, `CUDA` program će pokrenuti kernel na njemu. Svaki kernel generira skup potprocesa na GPU koji se zovu niti (engl. *threads*). Svaka nit u cijelosti izvršava kôd unutar pokrenutog kernela pojedinačno, odvojeno od ostalih niti koje se izvršavaju na GPU.

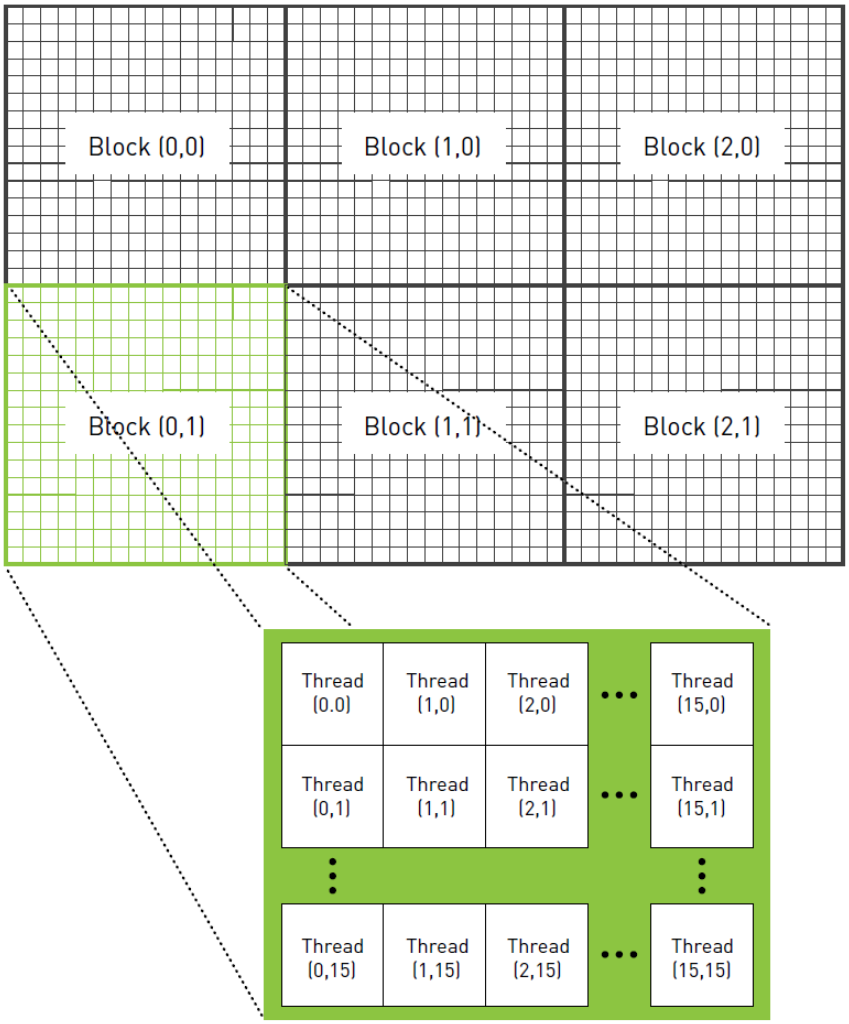


Slika 4.2: Usporedba CPU i GPU arhitekture *(Izvor: Wikipedia[12])*

Niti koje je kernel poziv stvorio su organizirane u hijerarhijskoj strukturi sa više razina kako bi se mogle konfigurirati za što bolju propusnost i korištenje hardvera bilo koje `CUDA`-omogućene GPU. Niti su organizirane u jednodimenzionalne, dvodimenzionalne ili trodi-

[11] Točka (piksel) u računalnoj grafici označava najmanju (osnovnu) jedinicu od koje je slika sastavljena.
[12] `https://cs.wikipedia.org/wiki/CUDA`, 14. 12. 2016.

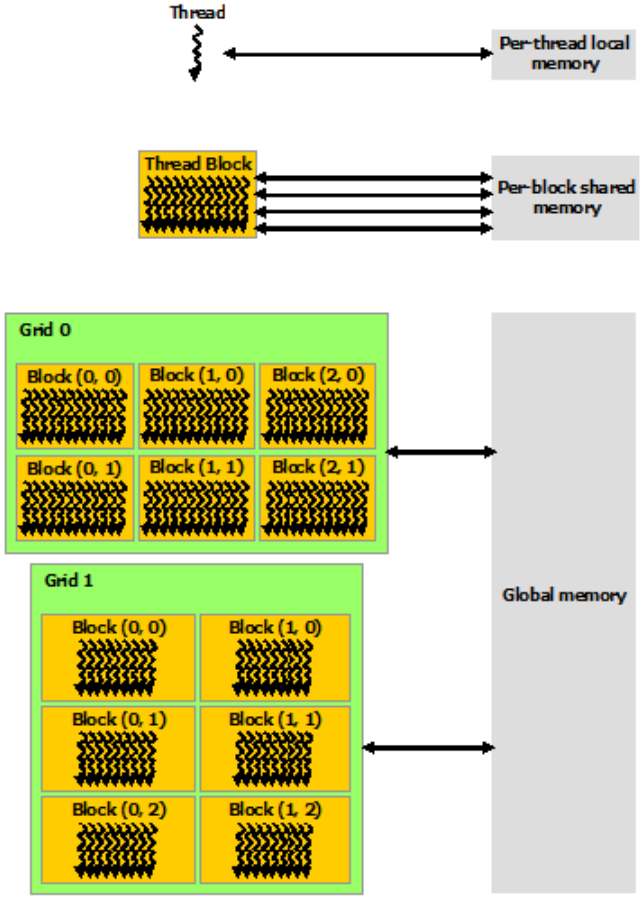
menzionalne nizove zvane blokovi (engl. *block*) koji se odvojeno izvršavaju na (potencijalno) različitim procesorima unutar GPU. Razlika između arhitekture CPU i GPU, osobito u broju procesora, vidljiva je na *Slici 4.2*. Upravo njihova arhitektura je razlog tomu što je na GPU kôd moguće vrlo efikasno paralelizirati. Svakoj niti je dodijeljen jedinstven identifikacijski broj za pojedinu dimenziju bloka, te se taj broj za svaku nit spremišti kao lokalna varijabla (u programskom okruženju). Svi blokovi unutar kernela su, kao i niti u blokovima, organizirani u jednodimenzionalne, dvodimenzionalne ili trodimenzionalne nizove zvane mreže (engl. *grid*). Blokovima su također unutar kernela dodijeljeni jedinstveni identifikacijski brojevi za svaku dimenziju u mreži. Za jednostavnije shvaćanje spomenutih pojmova, na *Slici 4.3* može se vidjeti primjer dvodimenzionalne mreže u kojoj se nalaze dvodimenzionalni blokovi niti.



Slika 4.3: Dvodimenzionalna mreža sa dvodimenzionalnim blokovima *(Izvor: [21])*

Ovisno o konkretnoj grafičkoj kartici i njenim specifikacijama, postoji granica u broju blokova, kao i u broju niti, koji se mogu izvršavati simultano. Ako je broj jezgri koje ona ima manji od broja blokova koje implementirano rješenje nekog problema pokreće, (nasumičan) podskup blokova će se izvršiti simultano, te će se po njihovom završetku početi izvršavati ostali blokovi. Ovaj proces traje dok svi blokovi u kernelu ne budu izvršeni. S druge strane, ako GPU ima velik broj jezgri, moguće je da će se svi blokovi izvršiti istovremeno. Ovakav način organiziranja blokova pruža fleksibilnost u prilagođavanju izvršavanja nekog kôda na grafičkim karticama različitih specifikacija, tj. različitog broja dostupnih jezgri.

Memorija unutar `CUDA`-e je distribuirana na sličan hijerarhijski način. Memorija mreže, koja se naziva globalna memorija (engl. *global memory*), postoji na najvišoj globalnoj razini. Svaka nit i svaki blok unutar kernel poziva imaju pristup ovoj memoriji, pa je ona zato analogna radnoj memoriji (engl. *RAM - Random Access Memory*) za CPU. Na razini ispod toga, svaki blok ima svoju dijeljenu memoriju (engl. *shared memory*) koja je dostupna svakoj niti u tom bloku, te je ona puno brža od globalne memorije. Dijeljena memorija se često koristi poput *cache memorije*[13] za operacije koje bi inače zahtjevale mnogo redudantnih pisanja i čitanja iz globalne memorije. Ona postoji za vrijeme trajanja izvršavanja bloka, te se oslobađa kada ono završi. Na najnižoj razini, svaka nit posjeduje malu količinu svoje privatne memorije kojoj samo ta nit može pristupiti. Ovakva memorija je najbrža, ali i najviše ograničena vrsta memorije dostupna na *NVIDIA* grafičkim čipovima. Slikovni prikaz vrsta memorija i shema mogućnosti pristupa istima nalazi se na *Slici 4.4*.



Slika 4.4: Vrste memorija na *NVIDIA* grafičkim čipovima *(Izvor: [2])*

Model izvršavanja kôda je generalizacija "jednostruka naredba, višestruki podaci" modela (engl. *SIMD - Single instruction, multiple data*) zvana "jednostruka naredba, višestruke niti" (engl. *SIMT - Single instruction, multiple threads*). Kod SIMD arhitekture, skup jezgri izvršava jednake operacije na različitim podacima u istim vremenskim intervalima. Ovakve arhitekture su prikladne za masivno paraleliziranje gdje se neke operacije ponavljaju na manjim podskupovima velikog skupa podataka, i gdje je svaka operacija nezavisna od ostalih.

[13]Priručna memorija (predmemorija, brza memorija; engl. *cache*), mala je memorija koja služi za pohranu podataka koji se često koriste.

Kada sve jezgre obave dodijeljene im operacije, one se sinkroniziraju te program nastavlja izvršavanje. Oprečno tome, model "jednostruke naredbe, jednostruki podaci" (engl. *SISD - Single instruction, single data*) nalaže da se svaka nit sinkronizira na kraju svog izvršavanja, što rezultira sekvencijalnim izvršavanjem programa. Iz perspektive pisanja kôda, SIMT model kojeg koristi `CUDA` identičan je SIMD modelu, zbog čega nije potrebna dodatna edukacija kako bi se iskoristila `CUDA` arhitektura.

`CUDA` podržava i proizvoljno sinkroniziranje pri izvršavanju, tj. moguće je zaustaviti CPU ili GPU izvršavanje i sinkronizirati sve niti koristeći jednostavne metode prije nego se program nastavi izvoditi. Ova značajka daje kontrolu tako da je moguće zaustaviti CPU izvršavanje dok GPU završi, kao što je moguće postići i simultano izvršavanje na CPU i GPU. Tek optimiziranjem kôda postavljanjem pomno planiranih sinkronizacija niti se postiže optimalno paralelno rješenje nekog problema.

Uvođenjem `CUDA 5` standarda, *NVIDIA* je omogućila i dinamičko paraleliziranje, dopuštajući izvršavanje kernel poziva unutar drugih kernel poziva. U prijašnjim verzijama bilo je moguće postići rekurziju jedino korištenjem GPU metoda koje su mogle pozivati niti u kernelu. Korištenje ovakvih funkcija bilo je relativno sporo, ali pojavom dinamičkog paraleliziranja jedna nit može pozvati kernel i stvoriti novi skup niti za potpuno paralelan rekurzivan algoritam. Dinamičko paraleliziranje je ograničeno brojem razina rekurzije koje je moguće postići kroz jednu nit. Kada nit napravi kernel poziv, ona dalje nastavlja računanje bez čekanja da se pozvani kernel izvrši, a to je ponašanje jednako pozivanju kernela sa CPU-a.

4.3 Prednosti i mane obrađivanja podataka na grafičkom čipu

Najveća prednost je vrlo dobra performantnost pri paraleliziranju velikog broja procesa. Paraleliziranje je, također, jednostavnije kontrolirati na grafičkim čipovima nego na CPU jer dok, primjerice, raspored izvršavanja niti i sinkronizacija na CPU nisu trivijalni, grafički čipovi su dizajnirani kako bi ove probleme rješavali prirodno (nativno) na hardveru, smanjujući vrijeme koje pisanje kôda i sama aplikacija troše na paraleliziranje. Kod grafičkih čipova, programeri ne moraju organizirati pojedine niti, već se one jednostavno kontroliraju kroz kernel pozive koji alociraju niti i blokove, te ih po alokaciji izvršavaju simultano. Nativna paralelizacija GPU arhitekture omogućuje skaliranje bez velikih hardverskih promjena. Unatoč tome što su jezgre na grafičkom čipu u pravilu slabije od CPU jezgara, velika propusnost grafičkih čipova u većini problema nadoknađuje tu razliku. Također, pomnim planiranjem struktura u kojima se podaci spremaju iz `CUDA` niti je moguće pristupiti uzastopnim dijelovima globalne memorije (engl. *coalesced memory access*), što poboljšava performanse.

S druge strane, dok se neki modeli mogu jako dobro modelirati u GPU računarstvu, pisanje efikasnog kôda nije bez svojih izazova. Jedan od najvećih je prijenos podataka sa CPU na GPU, jer, u usporedbi, na CPU program učitava podatke u radnu memoriju ili sa tvrdog diska (ili iz nekog drugog izvora), i brzo im može pristupiti. No, GPU prvo mora prenijeti sve podatke koji su joj potrebni u svoju memoriju prije nego računanje može početi, a kako neće moći ništa izvršavati dok se podaci prenose, ovisno o količini podataka i učestalosti prijenosa, može doći do značajnih padova u performansama. Također, dio kôda kojeg je potrebno sekvencijalno izvršavati neće imati koristi od GPU arhitekture jer će u tom slučaju sve jezgre osim jedne morati čekati, tj. biti neaktivne dok se sekvencijalni dio ne izvrši na jednoj jezgri, pa je optimalno rješenje da se takvi dijelovi izvršavaju na CPU.

4.4 Jednostavna CUDA implementacija Floyd-Warshall algoritma

Vrlo slično implementaciji iz *Poglavlja 3.2*, jednostavna CUDA implementacija Floyd-Warshall algoritma je najbliži mogući analogon algoritmu pisanom u pseudojeziku danom u *Poglavlju 3.1*. No, oprečno spomenutoj CPU implementaciji, osnovna razlika nalazi se u iskorištavanju hibridne CPU-GPU arhitekture - paralelizaciji. Naime, vanjska *for* petlja algoritma izvršava se na CPU, dok su unutrašnje dvije paralelizirane na GPU što je efikasnije moguće, koristeći maksimum broja niti po bloku za dani grafički čip.

Paralelizacija se dostiže na način da se pokreće onoliko CUDA niti koliko ima elemenata u kvadratnim matricama W, D i Π, pri čemu svaka nit izvršava logiku Floyd-Warshall algoritma za element matrice kojem je ona dodijeljena. Na taj način se algoritam istovremeno može izvršavati za onoliko elemenata matrice koliko grafička kartica na kojoj se izvršava to dopušta svojim specifikacijama.

Ovakav način implementacije potaknut je radom [6]. Postignuto je značajno poboljšanje u vremenu izvršavanja na testnim grafovima, kako je vidljivo u *Poglavlju 5*, a cjeloviti kôd ove implementacije nalazi se u *Prilogu 2*[14].

4.5 Napredna CUDA implementacija Floyd-Warshall algoritma

Napredni, tzv. blok (engl. *blocked*) algoritam, predložen u radu [26], nije dizajniran i optimiziran za radnu memoriju, nego za cache memoriju, što je bio standard 2003. godine. Isti algoritam pruža veliko ubrzanje naspram standardnog Floyd-Warshall algoritma u rješavanju problema najkraćih putova među svim parovima vrhova u grafu. Dodatno poboljšanje ovog algoritma i njegovo prilagođavanje CUDA arhitekturi predloženo je u radu [12], 2008. godine. U nastavku se nalaze kombinirani rezultati iz oba spomenuta rada.

Algoritam počinje particioniranjem matrice $W := D^{(0)}$ iz *Poglavlja 3.1* u podmatrice veličine $B \times B$, gdje se B naziva *blok faktorom*. Uobičajeno je da B dijeli $|V(G)|$. U svakom koraku algoritma određuje se tzv. primarna podmatrica, koja je jedna od spomenutih podmatrica matrice D. Primarne podmatrice u svim koracima algoritma nalaze se na dijagonali matrice, počevši od podmatrice koji počinje elementom $(1, 1)$. Primarna podmatrica sastoji se od elemenata koji se nalaze između $(p_{početak}, p_{početak})$ i (p_{kraj}, p_{kraj}), gdje vrijedi:

$$p_{početak} = \frac{\text{redni broj primarne podmatrice koju obrađujemo} \times \text{dimenzija podmatrica}}{\text{ukupan broj primarnih podmatrica}}$$

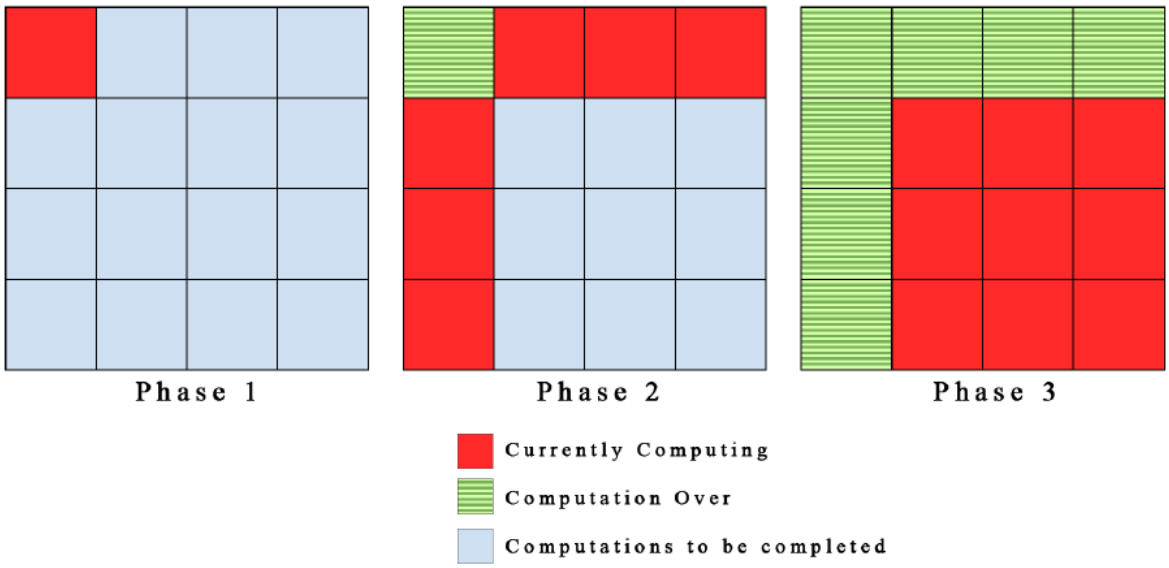
i

$$p_{kraj} = p_{početak} + \left(\frac{\text{dimenzija podmatrica}}{\text{ukupan broj primarnih podmatrica}} \right) - 1.$$

Da bi se olakšalo shvaćanje algoritma, u radu [26] predloženo je da se svaki korak algoritma promatra kroz 3 faze.

U fazi 1 prve iteracije, Floyd-Warshall algoritam poput onog iz *Poglavlja 3.1* koristi se za izračun najkraćih putova elemenata primarne podmatrice (gledajući na nju kao na posebnu matricu), gdje su i, j i k iz Floyd-Warshall algoritma između $p_{početak}$ i p_{kraj}. Računanje se

[14] Izvorni kôd dostupan i na repozitoriju `https://github.com/mislavkaraula/cuda_floyd_warshall` (14. 01. 2017.).



Slika 4.5: Faze blok algoritma kada primarni blok počinje elementom $(1, 1)$ *(Izvor: [12])*

odvija u jednom `CUDA` bloku. Po završetku Floyd-Warshall algoritma, nađeni su najkraći putovi među svim parovima vrhova između $p_{početak}$ i p_{kraj}.

U fazi 2 koristi se modificirana verzija Floyd-Warshall algoritma iz *Poglavlja 3.1* da bi se izračunale ostale podmatrice koje ovise samo o svojim elementima i elementima primarne podmatrice, a sve one su u istom stupcu ili retku kao primarna podmatrica. Ovo se može primijetiti jer za neku podmatricu koji se nalazi u istom retku kao primarna podmatrica, k se nalazi u rasponu između $p_{početak}$ i p_{kraj}, j u rasponu između $p_{početak}$ i p_{kraj}, a i u rasponu između $c_{početak}$ i c_{kraj}, gdje su $c_{početak}$ i c_{kraj} zapravo početni i krajnji vrh trenutne podmatrice prema x-osi. Analogno vrijedi i za podmatrice koji se nalaze u istom stupcu kao primarna podmatrica. Za podmatrice u istom retku se koristi formula:

$$d_{i,j}^{(k)} = \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(B)} + d_{k,j}^{(k-1)}\}.$$

Analogno, za podmatrice u istom stupcu koristi se formula:

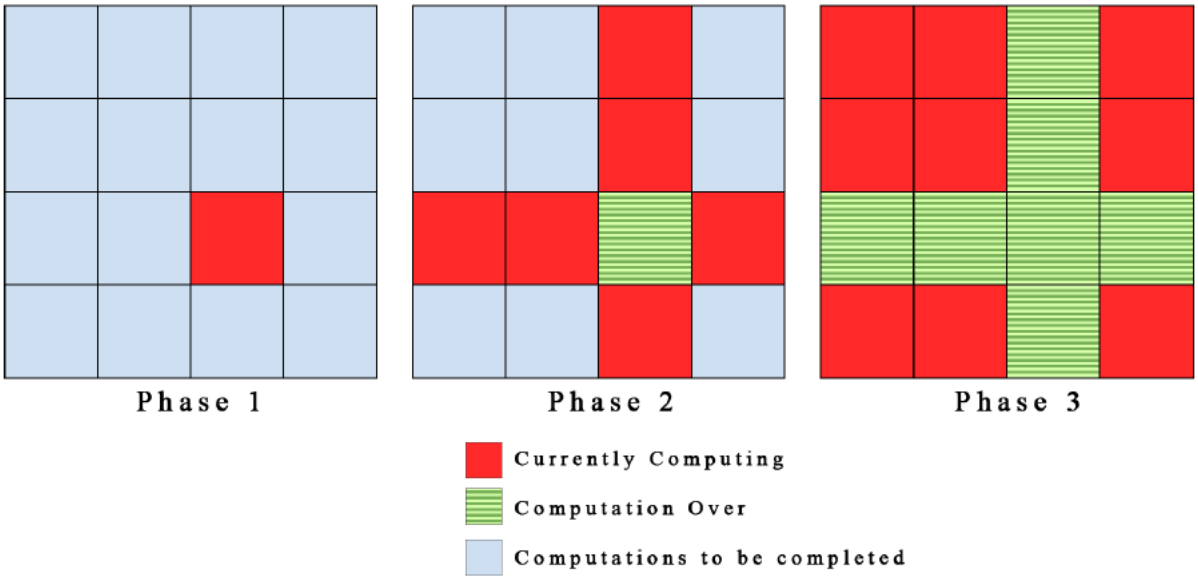
$$d_{i,j}^{(k)} = \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(B)}\}.$$

Ovime su, u fazi 2, izračunata rješenja svih parova vrhova za sve podmatrice koje dijele redak ili stupac sa primarnom, sa vrijednostima k između $p_{početak}$ i p_{kraj}, pri čemu se svaka podmatrica računala paralelno na GPU.

Konačno, u fazi 3 računaju se preostale podmatrice. Slično kao i u fazi 2, koristi se modificirana verzija Floyd-Warshall algoritma iz *Poglavlja 3.1.* Konkretno, koristi se formula:

$$d_{i,j}^{(k)} = \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(B)} + d_{k,j}^{(B)}\}.$$

Iz dane formule, kao i temeljem onoga što je već izračunato u fazi 2, sada se računaju elementi za sve podmatrice koje nisu primarna i ne dijele redak ili stupac s istom. Njihovi elementi ovisni su o elementima podmatrica dobivenih u fazi 2, i to svaka preostala podmatrica o jednoj podmatrici koja je bila u istom retku kao i primarna, te o jednoj podmatrici koja je bila u istom stupcu. Slikovni prikaz ovakvog pristupa podacima vidljiv je na *Slici 4.7.*



Slika 4.6: Faze blok algoritma kada je neki k-ti blok primaran *(Izvor: [12])*

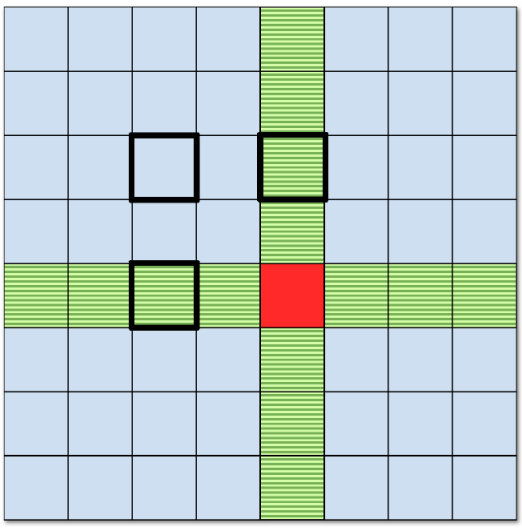
Po završetku faze 3, svi elementi matrice imaju izračunate vrijednosti svih parova vrhova za k između $p_{početak}$ i p_{kraj}. Primarna podmatrica se tada pomiče po dijagonali matrice i proces se ponavlja. Kada su se sve tri faze izračunale za svaku primarnu podmatricu, nađeno je konačno rješenje najkraćih putova među svim parovima vrhova u grafu.

Skica dokaza korektnosti naprednog Floyd-Warshall algoritma: Neka je sa D označena matrica udaljenosti naprednog Floyd-Warshall algoritma, sa D' matrica udaljenosti običnog Floyd-Warshall algoritma, te $n := |V(G)|$. Da bi se pokazala korektnost naprednog algoritma, potrebno je pokazati $D^n(i,j) = D'^n(i,j), \forall i,j \in \{1,\ldots,n\}$. Točnije, pokaže se da $D^k(i,j) = D'^k(i,j)$ za svake i i j kada je k višekratnik blok faktora B. Definira se $k = qB$, te se dokaz provodi indukcijom po q pri čemu se može pokazati da $D^k(i,j) = D'^k(i,j)$, za sve i i j za $0 \leq q \leq n/B$. Za više, pogledati [13] ili [26].

Komentar vremenske složenosti naprednog Floyd-Warshall algoritma: Još uvijek ne postoji model koji jasno opisuje `CUDA` implementacije. No, kako se generalno uzima da `CUDA` predstavlja CREW PRAM (engl. *Concurrent Read Exclusive Write, Parallel Random Access Machine*) model[15] (detaljnije vidi [6]) u kojemu se zanemaruju sinkronizacije niti u blokovima i komunikacija među njima, dobiva se teoretsko $\mathcal{O}(n)$ vrijeme izvršavanja algoritma.

Iskorištavanje dijeljene memorije: U svakom koraku algoritma broj primarne podmatrice prosljeđuje se u kernel. Svaki blok niti mora odrediti koju podmatricu trenutno obrađuje i koje podatke mora učitati u dijeljenu memoriju. U fazi 1, obzirom da je primarna podmatrica jedina relevantna, samo nju je potrebno učitati u dijeljenu memoriju. Svaka nit može učitati element podmatrice koji se podudara sa njenim identifikacijskim brojem, te spremiti tu vrijednost natrag u globalnu memoriju po završetku faze 1.

[15]PRAM model predstavlja analogiju RAM modela za paralelno računanje. Detaljnije vidi [10].

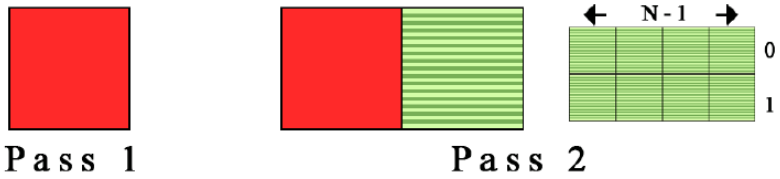


Slika 4.7: Pristup podacima u fazi 3 blok Floyd-Warshall algoritma *(Izvor: [12])*

U fazi 2, primarna podmatrica se učitava u dijeljenu memoriju zajedno sa trenutnom podmatricom, pri čemu svaka nit u svoju privatnu memoriju učitava po jedan element iz obje. Po završetku obrade, svaka nit sprema rezultantni element iz trenutne podmatrice u globalnu memoriju. Oblik mreže u ovoj fazi određuje jednostavnost obrade, pa tako, ukoliko su podmatrice dimenzija $B \times B$, onda je $2 \cdot (B-1)$ blokova niti obrađeno paralelno. Ovi blokovi su postavljeni u mrežu dimenzije $(B-1) \times 2$ (vidljivu na *Slici 4.8*), te se nalaze u globalnoj memoriji. Prvi redak u mreži obrađuje podmatrice koje su u istom retku kao primarna podmatrica, dok drugi redak obrađuje one koji su u istom stupcu. Prema tome, identifikator bloka po y-osi određuje poziciju trenutnog bloka naspram primarnog, pa se ova vrijednost može koristiti kao uvjet pri indeksiranju podataka. Konfiguracija dijeljene memorije iz faze 2, kao i one iz faze 1, vidljiva je na *Slici 4.8*. Bitno je, također, pravilno indeksirati podmatrice prema x-osi, te je važno da se "preskoči" primarna podmatrica, što se može postići dodavajući sljedeću vrijednost na identifikator bloka prema x-osi kada se učitava trenutni blok:

$$\textit{preskoči primarnu podmatricu} = \min\left(\frac{\textit{identifikator trenutne podmatrice} + 1}{\textit{identifikator primarne podmatrice} + 1}, 1\right),$$

gdje su *identifikator trenutne podmatrice* i *identifikator primarne podmatrice* vrijednosti obzirom na x-os. Vrijednost za *preskoči primarnu podmatricu* evaluirati će se u 0 ako je identifikator primarne podmatrice veći od identifikatora trenutne podmatrice, te u 1 kada je identifikator primarne podmatrice manji ili jednak onomu trenutne podmatrice.



Slika 4.8: Konfiguracija dijeljene memorije u fazama 1 i 2 *(Izvor: [12])*

U fazi 3 definira se mreža blokova veličine $(B-1) \times (B-1)$ (vidljiva na *Slici 4.9*) koja se nalazi u globalnoj memoriji. Slično fazi 2, i ovdje je potrebno preskočiti primarnu

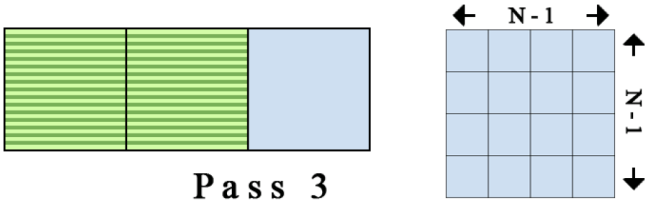
podmatricu, ali i sve podmatrice u istom retku ili stupcu sa primarnom podmatricom izračunate u fazi 2. Ovo se postiže sljedećim formulama analognima onoj gore:

$$\textit{preskoči primarnu podmatricu } (x) = \min\left(\frac{\textit{identifikator trenutne podmatrice.}x + 1}{\textit{identifikator primarne podmatrice.}x + 1}, 1\right)$$

i

$$\textit{preskoči primarnu podmatricu } (y) = \min\left(\frac{\textit{identifikator trenutne podmatrice.}y + 1}{\textit{identifikator primarne podmatrice.}y + 1}, 1\right).$$

Definirane vrijednosti dodaju se identifikatoru bloka prema x, odnosno y-osi. Dijeljena memorija u fazi 3 mora sadržavati tri bloka - trenutni blok, te blokove koji su dijelili isti stupac, odnosno redak s primarnim blokom. Ovakva konfiguracija vidljiva je na slici *Slici 4.9.*



Slika 4.9: Konfiguracija dijeljene memorije u fazi 3 *(Izvor: [12])*

Obzirom na sve iznesene zaključke o dijeljenoj memoriji i blokovima niti potrebnim za ovakav algoritam, on se na GPU može izvoditi uz sljedeća dva uvjeta:

- dimenzija podmatrica jednaka je najviše trećini ukupne dostupne dijeljene memorije - odnosno, tri podmatrice određene dimenzije moraju moći stati u dijeljenu memoriju; ovo je posljedica zahtjeva za dijeljenom memorijom iz faze 3
- dimenzija podmatrica ograničena je odozgo brojem niti koje se mogu pokrenuti u jednom bloku.

Oba navedena ograničenja ovise o modelu grafičke kartice.

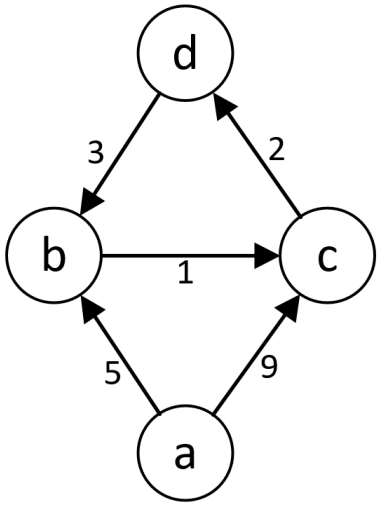
Cjeloviti kôd ovakve implementacije nalazi se u *Prilogu 3*[16].

Primjer 7. *Neka je zadan graf G kao na Slici 4.10.*

Napredni (blok) Floyd-Warshall algoritam, uz blok faktor $B = 2$ (trenutna primarna podmatrica biti će naznačena tako da su njeni elementi podebljani), te uz pretpostavku da se u redovima i stupcima nalaze redom vrhovi a, b, c, d, radi kako slijedi:

$$W = \begin{bmatrix} 0 & 5 & 9 & \infty \\ \infty & 0 & 1 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & 3 & \infty & 0 \end{bmatrix}$$

[16]Izvorni kôd dostupan i na repozitoriju `https://github.com/mislavkaraula/cuda_floyd_warshall` (14. 01. 2017.).



Slika 4.10: Jednostavan primjer težinskog grafa

Nadalje, za $B = 0$ *prije faze 1 vrijedi:*

$$D^{(0)} = \begin{bmatrix} \mathbf{0} & \mathbf{5} & 9 & \infty \\ \mathbf{\infty} & \mathbf{0} & 1 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & 3 & \infty & 0 \end{bmatrix}, \Pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 3 \\ \text{NIL} & 4 & \text{NIL} & \text{NIL} \end{bmatrix},$$

za $B = 0$*, nakon faze 1 i prije faze 2:*

$$D^{(0)} = \begin{bmatrix} \mathbf{0} & \mathbf{5} & 9 & \infty \\ \mathbf{\infty} & \mathbf{0} & 1 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & 3 & \infty & 0 \end{bmatrix}, \Pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 3 \\ \text{NIL} & 4 & \text{NIL} & \text{NIL} \end{bmatrix},$$

za $B = 0$*, nakon faze 2 i prije faze 3:*

$$D^{(0)} = \begin{bmatrix} \mathbf{0} & \mathbf{5} & 6 & \infty \\ \mathbf{\infty} & \mathbf{0} & 1 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & 3 & \infty & 0 \end{bmatrix}, \Pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 3 \\ \text{NIL} & 4 & \text{NIL} & \text{NIL} \end{bmatrix},$$

za $B = 0$*, nakon faze 3:*

$$D^{(0)} = \begin{bmatrix} \mathbf{0} & \mathbf{5} & 6 & \infty \\ \mathbf{\infty} & \mathbf{0} & 1 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & 3 & 4 & 0 \end{bmatrix}, \Pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 3 \\ \text{NIL} & 4 & 2 & \text{NIL} \end{bmatrix},$$

za $B = 1$*, nakon faze 1 i prije faze 2:*

$$D^{(1)} = \begin{bmatrix} 0 & 5 & 6 & \infty \\ \infty & 0 & 1 & \infty \\ \infty & \infty & \mathbf{0} & \mathbf{2} \\ \infty & 3 & \mathbf{4} & \mathbf{0} \end{bmatrix}, \Pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 3 \\ \text{NIL} & 4 & 2 & \text{NIL} \end{bmatrix},$$

za $B = 1$, nakon faze 2 i prije faze 3:

$$D^{(1)} = \begin{bmatrix} 0 & 5 & 6 & 8 \\ \infty & 0 & 1 & 3 \\ \infty & 5 & \mathbf{0} & \mathbf{2} \\ \infty & 3 & \mathbf{4} & \mathbf{0} \end{bmatrix}, \Pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 2 & 3 \\ \text{NIL} & \text{NIL} & 2 & 3 \\ \text{NIL} & 4 & \text{NIL} & 3 \\ \text{NIL} & 4 & 2 & \text{NIL} \end{bmatrix},$$

te na kraju, za $B = 1$, nakon faze 3:

$$D^{(1)} = \begin{bmatrix} 0 & 5 & 6 & 8 \\ \infty & 0 & 1 & 3 \\ \infty & 5 & \mathbf{0} & \mathbf{2} \\ \infty & 3 & \mathbf{4} & \mathbf{0} \end{bmatrix}, \Pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 2 & 3 \\ \text{NIL} & \text{NIL} & 2 & 3 \\ \text{NIL} & 4 & \text{NIL} & 3 \\ \text{NIL} & 4 & 2 & \text{NIL} \end{bmatrix}.$$

Izlaz, tj. rješenje algoritma je matrica $D^{(1)}$ koja je matrica najkraćih putova između svih parova vrhova u grafu danom na Slici 4.10. Ovaj primjer je relativno malen pa je jednostavno pratiti gore navedene iteracije, kao i provjeriti krajnji rezultat. No, algoritam se analogno računa za bilo koju veličinu grafa, neovisno o broju vrhova. Tako, primjerice, neki grafovi u Poglavlju 5 na kojima će se testirati implementacija ovog algoritma imaju na tisuće vrhova.

5 EKSPERIMENTALNI REZULTATI

Na grafovima raznih veličina s nasumičnim težinama bridova mjerena su vremena izvršavanja implementacija iz *Poglavlja 3.2, 4.4 i 4.5*. Računalo na kojem se izvodilo ovo testiranje nalazi se na Odjelu za matematiku Sveučilišta J. J. Strossmayera u Osijeku, te je specifikacija kako slijedi: *Intel Core i5-760 (8M Cache, 2.80 GHz)* procesor, 8GB radne memorije, *NVIDIA Tesla C2050/C2070* grafička kartica koja ima 448 `CUDA` jezgara, podržava raspored niti u tri dimenzije unutar bloka, te tri dimenzije bloka unutar mreže. Na njoj je moguće pokrenuti 1024 niti po bloku i ima 64KB dijeljene memorije, što je bitno za naprednu `CUDA` implementaciju iz prošlog poglavlja. Ova grafička kartica bazirana je na tzv. *Fermi* mikro-arhitekturi, te se radi o `CUDA` arhitekturi 2.0. U vrijeme pisanja ovog rada, već su postojale grafičke kartice na `CUDA` 6.0 arhitekturi, tzv. *Pascal* mikro-arhitektura.

Eksperimentalno su uzeti grafovi iz tri kategorije: gusti grafovi, rijetki grafovi, te grafovi koji su srednje gusti, pri čemu se za gustoću grafa koristila jednakost iz *Primjedbe 3*. Sve tri implementacije Floyd-Warshall algoritma su svaki od grafova obrađivale dva puta u različitim vremenskim intervalima kako bi se dobilo realno srednje vrijeme izvršavanja, tj. kako bi se smanjila odstupanja radi ostalih procesa na računalu. Rezultati ovih mjerenja (srednje vrijednosti izračunatih vremena) vidljivi su u *Tablici 1* za guste grafove, *Tablici 3* za rijetke grafove i *Tablici 5* za grafove koji su srednje gusti, pri čemu se *CPU* stupac odnosi na jednostavnu implementaciju Floyd-Warshall algoritma iz *Poglavlja 3.2*, *CUDA v1* stupac na jednostavnu `CUDA` implementaciju iz *Poglavlja 4.4*, a *CUDA v2* stupac na naprednu `CUDA` implementaciju iz *Poglavlja 4.5*.

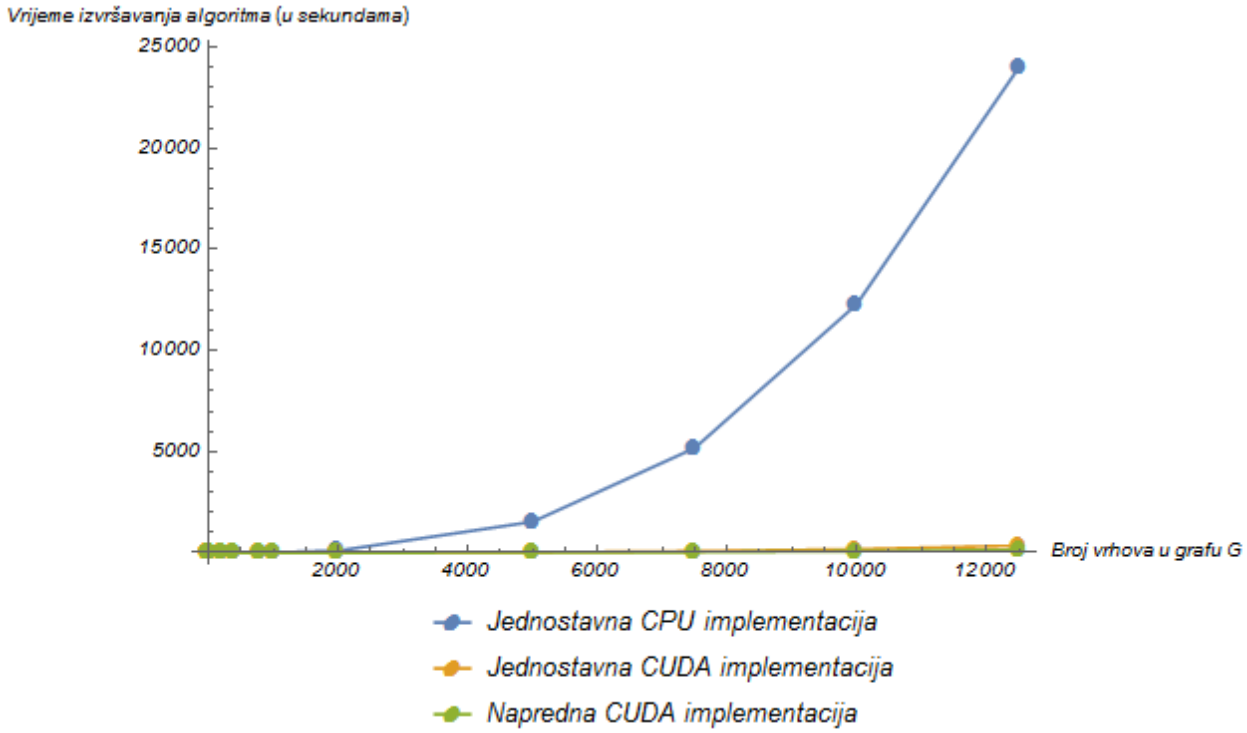
|V(G)|	**|E(G)|**	**ρ(G)**	**CPU**	**CUDA v1**	**CUDA v2**
5	10	50%	3e-06	0.05	0.05
10	40	44.44%	1.6e-05	0.05	0.05
20	160	42.11%	1.18e-04	0.05	0.05
40	640	41.03%	9.48e-04	0.05	0.05
200	6000	15.08%	0.1	0.06	0.05
400	24000	15.04%	0.81	0.07	0.06
800	96000	15.02%	6.46	0.15	0.09
1000	150000	15.02%	12.59	0.26	0.14
2000	400000	10.01%	99.66	1.51	0.71
5000	500000	2%	1531.97	22.5	9.93
7500	1125000	2%	5181.18	76.98	33.12
10000	2000000	2%	12297.75	175.58	78.08
12500	3125000	2%	24007.1	350.86	151.96

Tablica 1: Rezultati vremena izvršavanja za guste grafove (izražene vrijednosti vremena su u sekundama)

Osim usporedbe vremena izvršavanja pojedinih implementacija u sekundama, u *Tablicama 2, 4 i 6* dane su usporedbe efikasnosti implementacija, odnosno za koliko je puta, za pojedini graf, jednostavna `CUDA` implementacija bila brža od jednostavne `CPU` implementacije, te za koliko je puta napredna `CUDA` implementacija bila brža od jednostavne `CUDA` implementacije.

$\|V(G)\|$	$\|E(G)\|$	$\rho(G)$	$\frac{\text{CUDA v1}}{\text{CPU}}$	$\frac{\text{CUDA v2}}{\text{CUDA v1}}$
5	10	50%	6.034e-05	0.98
10	40	44.44%	2.98e-04	1.03
20	160	42.11%	2.29e-03	0.97
40	640	41.03%	0.02	1.08
200	6000	15.08%	1.80	1.2
400	24000	15.04%	11.91	1.14
800	96000	15.02%	43.21	1.59
1000	150000	15.02%	49.18	1.9
2000	400000	10.01%	66.19	2.12
5000	500000	2%	68.1	2.27
7500	1125000	2%	67.31	2.32
10000	2000000	2%	70.04	2.25
12500	3125000	2%	68.42	2.31

Tablica 2: Ubrzanja za guste grafove (izražene vrijednosti ubrzanja su u broju puta)



Slika 5.1: Vizualizacija podataka iz *Tablica 1 i 2* (gusti grafovi)

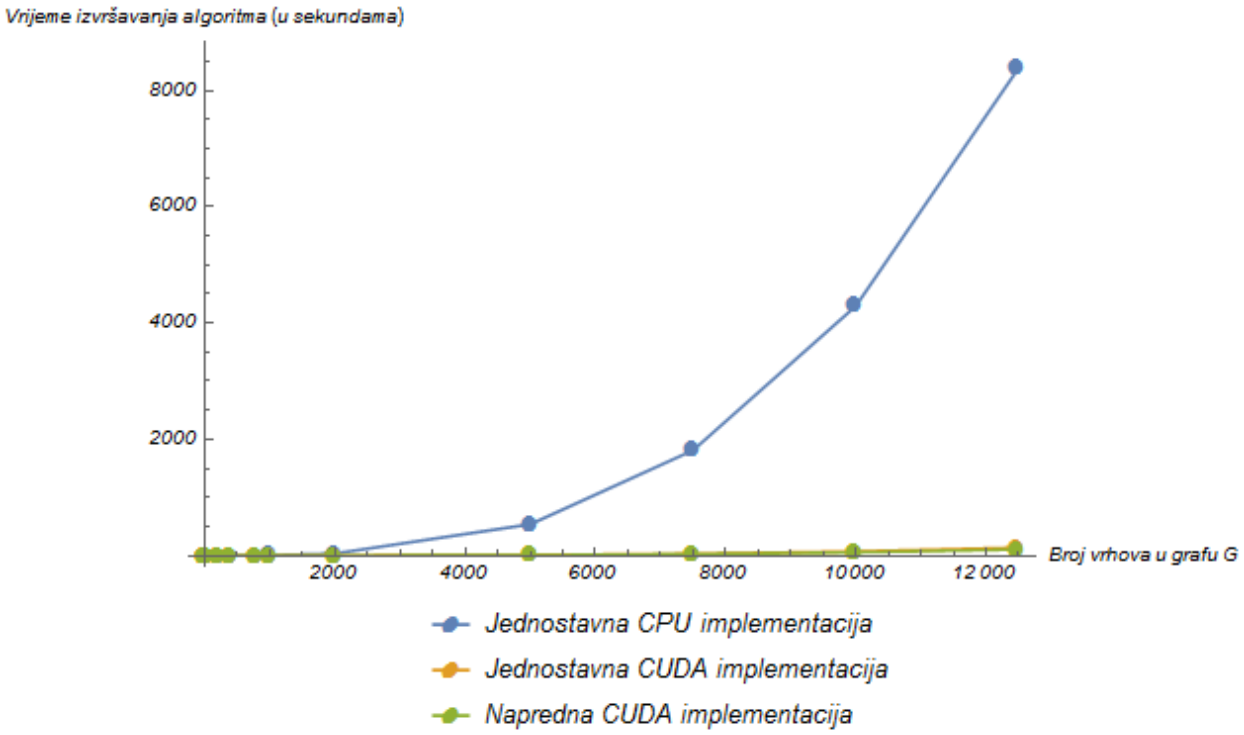
Povećanjem broja vrhova vidljivo je teoretski deklarirano kubno vrijeme izvršavanja Floyd-Warshall algoritma u jednostavnoj CPU implementaciji iz *Poglavlja 3.2.* S druge strane, obje `CUDA` implementacije se vrlo dobro skaliraju, pa povećanje broja vrhova ne utječe drastično na njihovo vrijeme izvršavanja. Uspoređujući naprednu i jednostavnu `CUDA` implementaciju, kod većine testiranih primjera je dobiveno skoro stopostotno poboljšanje. U nastavku, na *Slikama 5.1, 5.2 i 5.3*, nalaze se vizualni reprezentanti tablica iz ovog poglavlja na kojima su, ovisno o gustoći grafa, jasno vidljiva poboljšanja između tri različite implementacije Floyd-Warshall algoritma.

|V(G)|	**|E(G)|**	**ρ(G)**	**CPU**	**CUDA v1**	**CUDA v2**
5	2	10%	2e-06	0.05	0.05
10	5	5.56%	7e-06	0.05	0.05
20	10	2.63%	3.95e-05	0.05	0.05
40	20	1.28%	2.97e-04	0.05	0.05
200	100	0.25%	0.04	0.06	0.05
400	200	0.13%	0.28	0.06	0.05
800	400	0.06%	2.21	0.1	0.08
1000	500	0.05%	4.31	0.15	0.11
2000	1000	0.03%	34.51	0.7	0.5
5000	2500	0.01%	537.3	9.27	6.78
7500	3750	6.67e-03%	1811.47	30.49	22.6
10000	5000	5e-03%	4292.49	69.29	53.38
12500	6250	4e-03%	8384.87	135.19	103.4

Tablica 3: Rezultati vremena izvršavanja za rijetke grafove (izražene vrijednosti vremena su u sekundama)

|V(G)|	**|E(G)|**	**ρ(G)**	$\frac{\textbf{CUDA v1}}{\textbf{CPU}}$	$\frac{\textbf{CUDA v2}}{\textbf{CUDA v1}}$
5	2	10%	3.923e-05	1.04
10	5	5.56%	1.2e-04	1.1
20	10	2.63%	7.79e-03	1
40	20	1.28%	0.01	1.01
200	100	0.25%	0.65	1.07
400	200	0.13%	4.35	1.21
800	400	0.06%	22.38	1.19
1000	500	0.05%	29.29	1.38
2000	1000	0.03%	49.13	1.41
5000	2500	0.01%	57.93	1.37
7500	3750	6.67e-03%	59.41	1.35
10000	5000	5e-03%	61.94	1.3
12500	6250	4e-03%	62.03	1.3

Tablica 4: Ubrzanja za rijetke grafove (izražene vrijednosti ubrzanja su u broju puta)



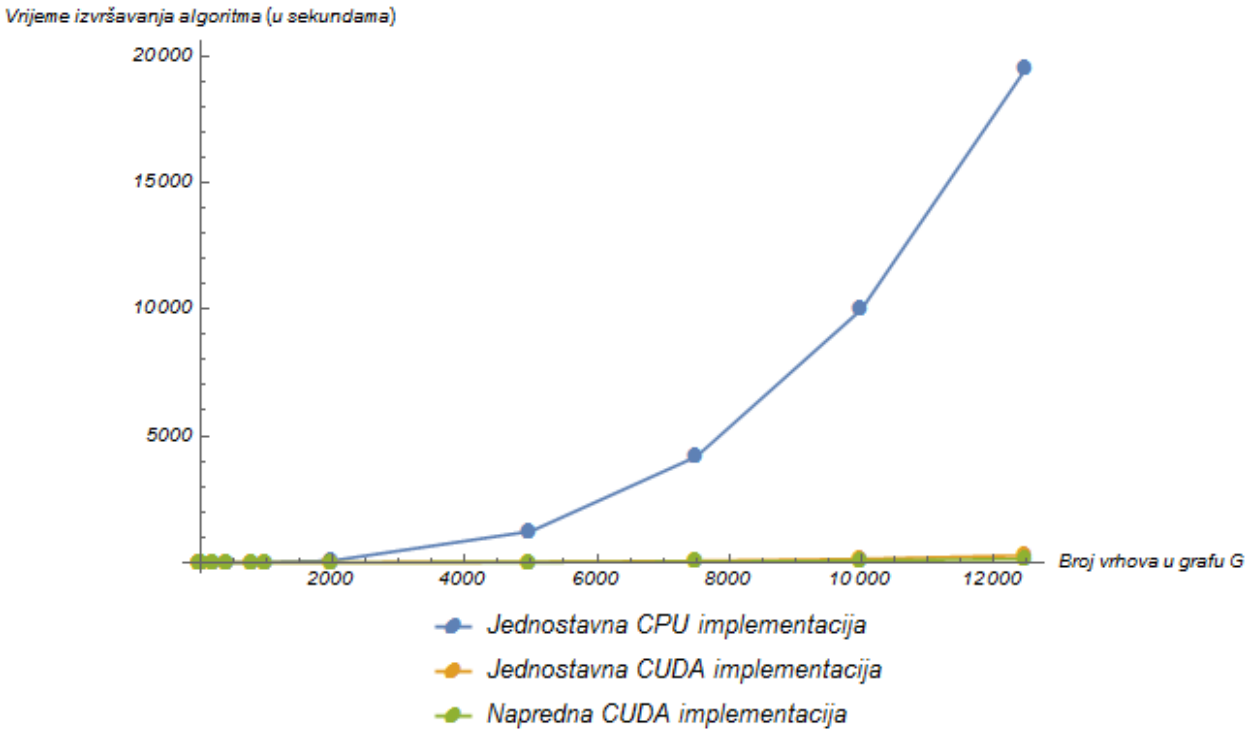
Slika 5.2: Vizualizacija podataka iz *Tablica 3 i 4* (rijetki grafovi)

|V(G)|	**|E(G)|**	**ρ(G)**	**CPU**	**CUDA v1**	**CUDA v2**
5	5	25%	2.5e-06	0.05	0.05
10	15	16.67%	1.2e-05	0.05	0.05
20	40	10.53%	7.6e-05	0.05	0.05
40	100	6.41%	4.83e-4	0.05	0.05
200	700	1.76%	0.07	0.06	0.05
400	1600	1%	0.6	0.06	0.05
800	3200	0.5%	4.42	0.12	0.09
1000	4500	0.45%	9.23	0.21	0.13
2000	10000	0.25%	75.83	1.19	0.66
5000	30000	0.12%	1234.1	18.11	9.29
7500	45000	0.08%	4177.94	61.43	31.02
10000	65000	0.07%	9992.2	141.69	73.23
12500	81250	0.05%	19538.15	281.27	142.83

Tablica 5: Rezultati vremena izvršavanja za grafove koji su srednje gusti (izražene vrijednosti vremena su u sekundama)

$\|V(G)\|$	$\|E(G)\|$	$\rho(G)$	$\frac{\text{CUDA v1}}{\text{CPU}}$	$\frac{\text{CUDA v2}}{\text{CUDA v1}}$
5	5	25%	4.65e-06	1.05
10	15	16.67%	2.39e-04	0.97
20	40	10.53%	1.5e-03	0.97
40	100	6.41%	0.01	0.98
200	700	1.76%	1.27	1.09
400	1600	1%	9.64	1.17
800	3200	0.5%	35.53	1.37
1000	4500	0.45%	44.26	1.64
2000	10000	0.25%	63.75	1.81
5000	30000	0.12%	68.14	1.95
7500	45000	0.08%	68.02	1.98
10000	65000	0.07%	70.52	1.94
12500	81250	0.05%	69.46	1.97

Tablica 6: Ubrzanja za grafove koji su srednje gusti (izražene vrijednosti ubrzanja su u broju puta)



Slika 5.3: Vizualizacija podataka iz *Tablica 5 i 6* (grafovi srednje gustoće)

Rezultati neporecivo dokazuju nadmoć GPU baziranih rješenja u ovakvim problemima. Uz grafičku karticu novije generacije od one na kojoj su ovi rezultati dobiveni, koja može pokrenuti više niti po bloku i ima više dijeljene memorije, rascjep između CPU i GPU implementacija bi bio još drastičniji, a onaj između jednostavne i napredne `CUDA` implementacije također još primjetniji. Potrebno je odabrati i adekvatne strategije korištenja danih implementacija kako bi se što bolje iskoristila GPU arhitektura.

6 ZAKLJUČAK

Teorija grafova matematička je podloga koja pruža mogućnost modeliranja raznih skupova podataka vrhovima i bridovima grafa, a pronalaženje najkraćih putova u grafu tada je rješenje velikom broju problema iz naše okoline. Zbog toga, bitno je pronaći optimalan algoritam koji je i skalabilan.

Najpoznatiji algoritam koji rješava problem najkraćih putova među svim parovima vrhova je Floyd-Warshall algoritam čija teorijska vremenska složenost govori kako njegovo vrijeme izvršavanja ovisi isključivo o broju vrhova grafa. Također, ovaj algoritam primjer je dinamičkog programiranja. Temeljem svega ovoga, Floyd-Warshall algoritam podoban je kandidat za paralelizaciju na nekoj hibridnoj CPU-GPU platformi kao što je `CUDA`. Osim toga, zbog načina na koji Floyd-Warshall radi, uviđeno je da je ovaj algoritam moguće dodatno paralelizirati i na algoritamskoj razini i time još više smanjiti brzinu izvršavanja za neke konkretne grafove.

Svi izneseni zaključci dovode do vrlo efikasnog rješenja koje svoj daljnji napredak ima u sve boljim grafičkim karticama koje proizvodi *NVIDIA* - u njihovim efikasnijim mikroarhitekturama, većem mogućem broju pokrenutih niti u jednom `CUDA` bloku, te većoj dijeljenoj memoriji potrebnoj za posljednju navedenu verziju Floyd-Warshall algoritma. Također, u vrijeme pisanja ovog rada su u testnoj fazi usluge u oblacima računala (engl. *cloud computing*) koje nude virtualne strojeve (engl. *virtual machine*) s najboljim `CUDA`-omogućenim grafičkim karticama koje postoje, što znači da će u skoroj budućnosti najbolje od ove tehnologije postati, na zahtjev, dostupno svima.

7 PRILOG

U ovom poglavlju priložen je programski kôd za sve 3 spomenute implementacije Floyd-Warshall algoritma: jednostavna `CPU` implementacija, jednostavna `CUDA` implementacija i napredna `CUDA` implementacija. Čitav programski projekt nalazi se u nastavku ovog poglavlja, kao i na GitHub repozitoriju: `https://github.com/mislavkaraula/cuda_floyd_warshall/`.

Prilog 1. Jednostavna CPU implementacija Floyd-Warshall algoritma iz *Poglavlja 3.2*:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <cstring>
#include <limits>    // radi definiranja beskonačnosti
#include <ctime>     // radi mjerenja vremena izvršavanja
using namespace std;

/* Definiramo beskonacnost kao najveci moguci integer broj. */
#define infty   std::numeric_limits<int>::max()

void printMatrix (int* G, unsigned int dim) {
  cout << "\r\n";
  for (int i = 0; i < dim*dim; i++) {
    if (G[i] < infty) {
      cout << G[i] << "\t";
    }
    else {
      cout << "INF" << "\t";
    }
    /* Ako je ispisao sve za jedan vrh, prijedi na sljedeci u novi redak.
       */
    if ((i+1)%dim == 0) {
      cout << "\r\n";
    }
  }
}

void Floyd_Warshall (int* W, int* D, int* PI, unsigned int dim) {
  //cout « "---- Floyd-Warshall ----" ;
  for (int k = 0; k < dim; k++) {
    for (int i = 0; i < dim; i++)
      for (int j = 0; j < dim; j++) {
        /* Kako ne bismo dobili overflow, umjesto provjere:
            D[i*dim+j] > D[i*dim+k] + D[k*dim+j]
           radimo donju provjeru. */
        if (D[i*dim+k] < infty && D[k*dim+j] < infty) {
          if (D[i*dim+j] > D[i*dim+k] + D[k*dim+j]) {
            D[i*dim+j] = D[i*dim+k] + D[k*dim+j];
            PI[i*dim+j] = PI[k*dim+j];
          }
        }
      }
    //cout « "--- k = " « k « " --- " ;
    //printMatrix(D, dim); printMatrix(PI, dim);
  }
  //cout « "---------------" ;
}
```

```
/* Metoda koja rekonstruira tezinu najkraceg puta za dani par vrhova
    koristeci matricu prethodnika PI i matricu inicijalnih tezina W. */
int getPath (int* W, int* PI, int i, int j, unsigned int dim) {
  if (i == j) {
    return 0;
  }
    else if (PI[i*dim+j] == -1) {
    return infty;
  }
  else {
    int recursivePath = getPath(W, PI, i, PI[i*dim+j], dim);
    if (recursivePath < infty) {
      return recursivePath + W[PI[i*dim+j]*dim+j];
    }
    else {
      return infty;
    }
  }
}

/* Za svaki par vrhova pokrece getPath metodu koja rekonstruira tezinu
    najkraceg puta izmedu njih koristeci matricu prethodnika PI. Tu tezinu
    onda usporeduje sa dobivenom tezinom za isti par vrhova u matrici
    najkracih putova D. */
bool checkSolutionCorrectness (int* W, int* D, int* PI, unsigned int dim)
   {
  for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
      if (getPath(W, PI, i, j, dim) != D[i*dim+j]) {
        return false;
      }
    }
  }

  return true;
}


int main() {

  /*
     V - broj vrhova
     E - broj bridova
     u - prvi vrh pri ucitavanju grafa iz datoteke
     v - drugi vrh pri ucitavanju grafa iz datoteke
     w - tezina izmedu v1 i v2 pri ucitavanju grafa iz datoteke
  */
  unsigned int V, E;
  int u, v, w;

  ifstream inputGraphFile; inputGraphFile.open("graphFile.txt");
  ofstream outputFile; outputFile.open("output.txt");

  inputGraphFile >> V >> E;
  cout << "V = " << V << ", E = " << E << "\n";

  unsigned int n = V*V;
```

```
/* Inicijalizacija grafova u memoriji. */
int* W = (int*)malloc(n*sizeof(int));
int* D = (int*)malloc(n*sizeof(int));
int* PI = (int*)malloc(n*sizeof(int));


/* Postavljanje inicijalnih vrijednosti za matricu prethodnika PI(0),
   matricu tezina W i matricu najkracih putova D(0). */
fill_n(W, n, infty);
fill_n(PI, n, -1);

for (int i = 0; i < E; i++) {
  inputGraphFile >> u >> v >> w;
  //cout « u « " <- " « w « " -> " « v « " | ";

  W[u*V+v] = w;
  if (u != v) {
    PI[u*V+v] = u;
  }
}

for (int i = 0; i < V; i++) {
  W[i*V+i] = 0;
}

/* D(0) = W na pocetku. */
memcpy (D, W, n*sizeof(int));

// printMatrix(W, V); printMatrix(D, V); printMatrix(PI, V);

/* Pocetak mjerenja izvrsavanja Floyd-Warshall algoritma. */
clock_t begin = clock();

/* Pozivamo Floyd-Warshall CPU algoritam nad ucitanim grafom. */
Floyd_Warshall(W, D, PI, V);

/* Kraj mjerenja izvrsavanja Floyd-Warshall algoritma. */
clock_t end = clock();
double elapsedTime = double(end - begin) / CLOCKS_PER_SEC;

//printMatrix(W, V); printMatrix(D, V); printMatrix(PI, V);

/* Ispis rezultata u datoteku. */
outputFile << "|V| = " << V << ", |E| = " << E << " \n ";
for (int i = 0; i < n; i++) {
  if (i%V==0) outputFile << " \n ";
  if (D[i] < infty)
    outputFile << D[i] << "\t";
  else
    outputFile << "INF" << "\t";
}
outputFile << " | ";
for (int i = 0; i < n; i++) {
  if (i%V==0) outputFile << " | ";
  outputFile << PI[i] << "\t";
}
```

```
  cout << "Vrijeme izvrsavanja Floyd-Warshall algoritma: "
          << elapsedTime << "s. ";

  if (checkSolutionCorrectness(W, D, PI, V) == true)
    cout << "Svi najkraci putevi su tocno izracunati!";
  else
    cout << "Najkraci putevi nisu tocno izracunati.";

  inputGraphFile.close();
  outputFile.close();
  free(W); free(D); free(PI);

  return 0;
}
```

Prilog 2. Jednostavna CUDA implementacija Floyd-Warshall algoritma iz *Poglavlja 4.4*:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <cstring>
#include <limits>    // radi definiranja beskonačnosti
#include <ctime>     // radi mjerenja vremena izvršavanja
#include <cmath>     // radi "strop" funkcije
using namespace std;

/* Definiramo beskonacnost kao najveci moguci integer broj. */
#define infty   std::numeric_limits<int>::max()

void printMatrix (int* G, unsigned int dim) {
  cout << "\r\n";
  for (int i = 0; i < dim*dim; i++) {
    if (G[i] < infty) {
      cout << G[i] << "\t";
    }
    else {
      cout << "infty" << "\t";
    }
    /* Ako je ispisao sve za jedan vrh, prijedi na sljedeci u novi redak.
       */
    if ((i+1)%dim == 0) {
      cout << "\r\n";
    }
  }
}

/* Kernel za device koji paralelizira unutarnje dvije for petlje
    Floyd-Warshall algoritma. */
__global__ void FW_Cuda(int k, int* D, int* PI, unsigned int dim) {
    int i = blockDim.y * blockIdx.y + threadIdx.y;
    int j = blockDim.x * blockIdx.x + threadIdx.x;

  if (i < dim && j < dim && D[i*dim+k] < INT_MAX && D[k*dim+j] < INT_MAX)
     {
    if (D[i*dim+j] > D[i*dim+k] + D[k*dim+j]) {
      D[i*dim+j] = D[i*dim+k] + D[k*dim+j];
      PI[i*dim+j] = PI[k*dim+j];
    }
  }
}


void Floyd_Warshall_Cuda (int* W, int* D, int* PI, unsigned int dim) {
  unsigned int n = dim*dim;

    /* Error varijabla za handleanje CUDA errora. */
    cudaError_t err = cudaSuccess;

  /* Alociranje device varijabli matrica D i PI. */
    int* d_D = NULL;
    err = cudaMalloc((void**) &d_D, n*sizeof(int));
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno alociranje matrice D
    (error code %s)!\n", cudaGetErrorString(err));
```

```
        exit(EXIT_FAILURE);
    }

    int* d_PI = NULL;
    err = cudaMalloc((void**) &d_PI, n*sizeof(int));
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno alociranje matrice PI
    (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

/* Kopiranje podataka iz host matrica u device. */
    err = cudaMemcpy(d_D, D, n*sizeof(int), cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno kopiranje matrice D iz hosta u device
    (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaMemcpy(d_PI, PI, n*sizeof(int), cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno kopiranje matrice PI iz hosta u device
    (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

/* Pozivanje CUDA kernela. */
int blockDim = 32;  /* Ako je dim. bloka 32, 1024 threada su po bloku.
    */
/* Racuna se kolika mora biti dim. grida ovisno o dim. blokova. */
int gridDim = ceil((float)dim/(float)blockDim);

cout << "CUDA kernel se pokrece sa " << gridDim*gridDim <<
    " blokova i " << blockDim*blockDim << " threadova po bloku.\r\n";

/* Vanjsku petlju Floyd-Warshall algoritma vrtimo na CPU,
     unutarnje dvije paraleliziramo. */
for (int k = 0; k < dim; k++) {
    FW_Cuda<<<dim3(gridDim, gridDim, 1),
      dim3(blockDim, blockDim, 1)>>> (k, d_D, d_PI, dim);

    err = cudaGetLastError();
    if (err != cudaSuccess) {
      fprintf(stderr, "Neuspjesno pokrenuta kernel metoda
        (error code %s)!\n", cudaGetErrorString(err));
      exit(EXIT_FAILURE);
    }

    /* Sinkronizacija threadova kako bi se zavrsila k-ta iteracija,
        te kako bi se preslo na (k+1). iteraciju. */
    cudaThreadSynchronize();
}

/* Kopiranje podataka iz device matrica u host. */
    err = cudaMemcpy(D, d_D, n*sizeof(int), cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno kopiranje matrice D iz devicea u host
    (error code %s)!\n", cudaGetErrorString(err));
```

```
        exit(EXIT_FAILURE);
    }

    err = cudaMemcpy(PI, d_PI, n*sizeof(int), cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno kopiranje matrice PI iz devicea u host
    (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

  /* Dealociranje device varijabli matrica D i PI. */
  err = cudaFree(d_D);
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno dealociranje matrice D
    (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaFree(d_PI);
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno dealociranje matrice PI
    (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

  /* Reset CUDA devicea i zavrsavanje CUDA Floyd-Warshalla. */
    err = cudaDeviceReset();
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno resetiranje devicea (zavrsavanje sa
    CUDA FW, priprema za sljedece pokretanje)!
    error=%s\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

}


/* Metoda koja rekonstruira tezinu najkraceg puta za dani par vrhova
    koristeci matricu prethodnika PI i matricu inicijalnih tezina W. */
int getPath (int* W, int* PI, int i, int j, unsigned int dim) {
  if (i == j) {
    return 0;
  }
    else if (PI[i*dim+j] == -1) {
    return infty;
  }
  else {
    int recursivePath = getPath(W, PI, i, PI[i*dim+j], dim);
    if (recursivePath < infty) {
      return recursivePath + W[PI[i*dim+j]*dim+j];
    }
    else {
      return infty;
    }
  }
}
```

```
/* Za svaki par vrhova pokrece getPath metodu koja rekonstruira tezinu
    najkraceg puta izmedu njih koristeci matricu prethodnika PI. Tu tezinu
    onda usporeduje sa dobivenom tezinom za isti par vrhova u matrici
    najkracih putova D. */
bool checkSolutionCorrectness (int* W, int* D, int* PI, unsigned int dim)
   {
  for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
      if (getPath(W, PI, i, j, dim) != D[i*dim+j]) {
        return false;
      }
    }
  }

  return true;
}


int main() {

  /*
     V - broj vrhova
     E - broj bridova
     u - prvi vrh pri ucitavanju grafa iz datoteke
     v - drugi vrh pri ucitavanju grafa iz datoteke
     w - tezina izmedu v1 i v2 pri ucitavanju grafa iz datoteke
  */
  unsigned int V, E;
  int u, v, w;

  ifstream inputGraphFile; inputGraphFile.open("graphFile.txt");
  ofstream outputFile; outputFile.open("output_cuda.txt");

  inputGraphFile >> V >> E;
  cout << "V = " << V << ", E = " << E << "\r\n";

  unsigned int n = V*V;

  /* Inicijalizacija grafova u memoriji. */
  int* W = (int*)malloc(n*sizeof(int));
  int* D = (int*)malloc(n*sizeof(int));
  int* PI = (int*)malloc(n*sizeof(int));

  /* Postavljanje inicijalnih vrijednosti za matricu prethodnika PI(0),
     matricu tezina W i matricu najkracih putova D(0). */
  fill_n(W, n, infty);
  fill_n(PI, n, -1);

  for (int i = 0; i < E; i++) {
    inputGraphFile >> u >> v >> w;
    //cout « u « " <- " « w « " -> " « v « " | ";

    W[u*V+v] = w;
    if (u != v) {
      PI[u*V+v] = u;
    }
  }
```

```
  for (int i = 0; i < V; i++) {
    W[i*V+i] = 0;
  }

  /* D(0) = W na pocetku. */
  memcpy (D, W, n*sizeof(int));

  // printMatrix(W, V); printMatrix(D, V); printMatrix(PI, V);

  /* Pocetak mjerenja izvrsavanja Floyd-Warshall algoritma. */
  clock_t begin = clock();

  /* Pozivamo Floyd-Warshall CPU algoritam nad ucitanim grafom. */
  Floyd_Warshall_Cuda(W, D, PI, V);

  /* Kraj mjerenja izvrsavanja Floyd-Warshall algoritma. */
  clock_t end = clock();
  double elapsedTime = double(end - begin) / CLOCKS_PER_SEC;

  //printMatrix(W, V); printMatrix(D, V); printMatrix(PI, V);

  /* Ispis rezultata u datoteku. */
  outputFile << "|V| = " << V << ", |E| = " << E << "\r\n\r\n";
  for (int i = 0; i < n; i++) {
    if (i%V==0) outputFile << "\r\n";
    if (D[i] < infty)
      outputFile << D[i] << "\t";
    else
      outputFile << "infty" << "\t";
  }
  outputFile << "\r\n\r\n";
  for (int i = 0; i < n; i++) {
    if (i%V==0) outputFile << "\r\n";
    outputFile << PI[i] << "\t";
  }

  cout << "Vrijeme izvrsavanja Floyd-Warshall algoritma: " <<
    elapsedTime << "s.\r\n";

  if (checkSolutionCorrectness(W, D, PI, V) == true)
    cout << "Svi najkraci putevi su tocno izracunati!\r\n";
  else
    cout << "Najkraci putevi nisu tocno izracunati.\r\n";

  inputGraphFile.close();
  outputFile.close();
  free(W); free(D); free(PI);

  return 0;
}
```

Prilog 3. Napredna (blok) CUDA implementacija Floyd-Warshall algoritma iz *Poglavlja 4.5*:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <cstring>
#include <limits>    // radi definiranja beskonačnosti
#include <ctime>     // radi mjerenja vremena izvršavanja
#include <cmath>     // radi "strop" funkcije
using namespace std;

/* Definiramo beskonacnost kao najveci moguci integer broj. */
#define infty   std::numeric_limits<int>::max()

void printMatrix (int* G, unsigned int dim) {
  cout << "\r\n";
  for (int i = 0; i < dim*dim; i++) {
    if (G[i] < infty) {
      cout << G[i] << "\t";
    }
    else {
      cout << "infty" << "\t";
    }
    /* Ako je ispisao sve za jedan vrh, prijedi na sljedeci u novi redak.
       */
    if ((i+1)%dim == 0) {
      cout << "\r\n";
    }
  }
}

/* Kernel za device koji implementira prvu fazu blocked
    Floyd-Warshall algoritma. */
__global__ void FW_Cuda_Phase1(int B, int* D, int* PI,
  int dim, int primaryBlockDim) {
  extern __shared__ int shrd[];

  int* sh_D = &shrd[0];
  int* sh_PI = &shrd[primaryBlockDim*primaryBlockDim];

  int i = B*primaryBlockDim + threadIdx.y;
  int j = B*primaryBlockDim + threadIdx.x;

  int sub_dim = primaryBlockDim;
  int sub_i = threadIdx.y;
  int sub_j = threadIdx.x;

  __syncthreads();

  if (sub_i < sub_dim && sub_j < sub_dim && sub_i*sub_dim+sub_j <
  sub_dim*sub_dim && i < max(sub_dim, dim) && j < max(sub_dim, dim)) {
    sh_D[sub_i*sub_dim+sub_j] = D[i*dim+j];
    sh_PI[sub_i*sub_dim+sub_j] = PI[i*dim+j];
  }
  else {
    sh_D[sub_i*sub_dim+sub_j] = INT_MAX;
    sh_PI[sub_i*sub_dim+sub_j] = -1;
  }
```

```
  __syncthreads();

  for (int sub_k = 0; sub_k < min(sub_dim, dim); sub_k++) {
    __syncthreads();
    if (i < max(sub_dim, dim) && j < max(sub_dim, dim) &&
    sh_D[sub_i*sub_dim+sub_k] < INT_MAX &&
    sh_D[sub_k*sub_dim+sub_j] < INT_MAX) {
      if (sh_D[sub_i*sub_dim+sub_j] >
        sh_D[sub_i*sub_dim+sub_k] + sh_D[sub_k*sub_dim+sub_j]) {
        sh_D[sub_i*sub_dim+sub_j] = sh_D[sub_i*sub_dim+sub_k] +
        sh_D[sub_k*sub_dim+sub_j];
        sh_PI[sub_i*sub_dim+sub_j] = sh_PI[sub_k*sub_dim+sub_j];
      }
    }
  }

  __syncthreads();

  if (sub_i < min(sub_dim, dim) && sub_j < min(sub_dim, dim) &&
  sub_i*sub_dim+sub_j < sub_dim*sub_dim &&
  i < max(sub_dim, dim) && j < max(sub_dim, dim)) {
    D[i*dim+j] = sh_D[sub_i*sub_dim+sub_j];
    PI[i*dim+j] = sh_PI[sub_i*sub_dim+sub_j];
  }
}

/* Kernel za device koji implementira drugu fazu
    blocked Floyd-Warshall algoritma. */
__global__ void FW_Cuda_Phase2(int B, int* D, int* PI,
  int dim, int primaryBlockDim) {
  extern __shared__ int shrd[];

  /* Sve varijable koje imaju prefiks "p_" pripadaju primarnom podbloku,
     sve koje imaju "c_" pripadaju trenutnom bloku. */
  int* p_sh_D = &shrd[0];
  int* p_sh_PI = &shrd[primaryBlockDim*primaryBlockDim];
  int* c_sh_D = &shrd[2*primaryBlockDim*primaryBlockDim];
  int* c_sh_PI = &shrd[3*primaryBlockDim*primaryBlockDim];

  int p_i = B*primaryBlockDim + threadIdx.y;
  int p_j = B*primaryBlockDim + threadIdx.x;

  /* Ako je trenutni blok prije primarnog, skipCenterBlock biti ce 0.
     Inace, ako je primarni ili neki nakon njega, biti ce 1. */
  int skipCenterBlock = min((blockIdx.x+1)/(B+1), 1);

  int c_i, c_j;

  /* Ako je y koordinata bloka u gridu jednaka 0,
      onda on pripada istom retku kao i primarni blok.
      Ako je y koordinata bloka u gridu jednaka 1,
      pripada istom stupcu kao i primarni blok. */
  if (blockIdx.y == 0) {
    c_i = p_i;
    c_j = (blockIdx.x+skipCenterBlock)*primaryBlockDim + threadIdx.x;
  }
  else {
```

```
  c_i = (blockIdx.x+skipCenterBlock)*primaryBlockDim + threadIdx.y;
  c_j = p_j;
}

int sub_dim = primaryBlockDim;
int sub_i = threadIdx.y;
int sub_j = threadIdx.x;

__syncthreads();

p_sh_D[sub_i*sub_dim+sub_j] = D[p_i*dim+p_j];
p_sh_PI[sub_i*sub_dim+sub_j] = PI[p_i*dim+p_j];

if (sub_i < sub_dim && sub_j < sub_dim &&
sub_i*sub_dim+sub_j < sub_dim*sub_dim &&
c_i < max(sub_dim, dim) && c_j < max(sub_dim, dim)) {
  c_sh_D[sub_i*sub_dim+sub_j] = D[c_i*dim+c_j];
  c_sh_PI[sub_i*sub_dim+sub_j] = PI[c_i*dim+c_j];
}
else {
  c_sh_D[sub_i*sub_dim+sub_j] = INT_MAX;
  c_sh_PI[sub_i*sub_dim+sub_j] = -1;
}

__syncthreads();

for (int sub_k = 0; sub_k < min(sub_dim, dim); sub_k++) {
  __syncthreads();
  /* Pripada istom stupcu kao i primarni blok. */
  if (blockIdx.y == 1) {
    if (c_i < max(sub_dim, dim) && c_j < max(sub_dim, dim) &&
    c_sh_D[sub_i*sub_dim+sub_k] < INT_MAX &&
    p_sh_D[sub_k*sub_dim+sub_j] < INT_MAX) {
      if (c_sh_D[sub_i*sub_dim+sub_j] > c_sh_D[sub_i*sub_dim+sub_k] +
      p_sh_D[sub_k*sub_dim+sub_j]) {
        c_sh_D[sub_i*sub_dim+sub_j] = c_sh_D[sub_i*sub_dim+sub_k] +
        p_sh_D[sub_k*sub_dim+sub_j];
        c_sh_PI[sub_i*sub_dim+sub_j] = p_sh_PI[sub_k*sub_dim+sub_j];
      }
    }
  }
  /* Pripada istom retku kao i primarni blok. */
  if (blockIdx.y == 0) {
    if (c_i < max(sub_dim, dim) && c_j < max(sub_dim, dim) &&
    p_sh_D[sub_i*sub_dim+sub_k] < INT_MAX &&
    c_sh_D[sub_k*sub_dim+sub_j] < INT_MAX) {
      if (c_sh_D[sub_i*sub_dim+sub_j] > p_sh_D[sub_i*sub_dim+sub_k] +
      c_sh_D[sub_k*sub_dim+sub_j]) {
        c_sh_D[sub_i*sub_dim+sub_j] = p_sh_D[sub_i*sub_dim+sub_k] +
        c_sh_D[sub_k*sub_dim+sub_j];
        c_sh_PI[sub_i*sub_dim+sub_j] = c_sh_PI[sub_k*sub_dim+sub_j];
      }
    }
  }
  __syncthreads();
}

__syncthreads();
```

```
  if (sub_i < min(sub_dim, dim) && sub_j < min(sub_dim, dim) &&
  sub_i*sub_dim+sub_j < sub_dim*sub_dim &&
  c_i < max(sub_dim, dim) && c_j < max(sub_dim, dim)) {
    D[c_i*dim+c_j] = c_sh_D[sub_i*sub_dim+sub_j];
    PI[c_i*dim+c_j] = c_sh_PI[sub_i*sub_dim+sub_j];
  }
}

/* Kernel za device koji implementira trecu fazu
    blocked Floyd-Warshall algoritma. */
__global__ void FW_Cuda_Phase3(int B, int* D, int* PI,
  int dim, int primaryBlockDim) {
  extern __shared__ int shrd[];

  /* Sve varijable koje imaju prefiks "p1_" pripadaju primarnom
      podbloku 1 izracunatom u fazi 2, sve koje imaju prefiks
      "p2_" pripadaju primarnom podbloku 2 izracunatom u fazi 2,
      a sve koje imaju "c_" pripadaju trenutnom bloku. */
  int* p1_sh_D = &shrd[0];
  int* p1_sh_PI = &shrd[primaryBlockDim*primaryBlockDim];
  int* p2_sh_D = &shrd[2*primaryBlockDim*primaryBlockDim];
  int* p2_sh_PI = &shrd[3*primaryBlockDim*primaryBlockDim];
  int* c_sh_D = &shrd[4*primaryBlockDim*primaryBlockDim];
  int* c_sh_PI = &shrd[5*primaryBlockDim*primaryBlockDim];

  /* Ako je trenutni blok prije primarnog, skipCenterBlock biti ce 0.
     Inace, ako je primarni ili neki nakon njega, biti ce 1.
     U ovoj fazi to radimo po obje osi. */
  int skipCenterBlockX = min((blockIdx.x+1)/(B+1), 1);
  int skipCenterBlockY = min((blockIdx.y+1)/(B+1), 1);

  int c_i = (blockIdx.y+skipCenterBlockY)*primaryBlockDim+threadIdx.y;
  int c_j = (blockIdx.x+skipCenterBlockX)*primaryBlockDim+threadIdx.x;

  int p1_i = c_i;
  int p1_j = B*primaryBlockDim + threadIdx.x;
  int p2_i = B*primaryBlockDim + threadIdx.y;
  int p2_j = c_j;

  int sub_dim = primaryBlockDim;
  int sub_i = threadIdx.y;
  int sub_j = threadIdx.x;

  __syncthreads();

  p1_sh_D[sub_i*sub_dim+sub_j] = D[p1_i*dim+p1_j];
  p1_sh_PI[sub_i*sub_dim+sub_j] = PI[p1_i*dim+p1_j];
  p2_sh_D[sub_i*sub_dim+sub_j] = D[p2_i*dim+p2_j];
  p2_sh_PI[sub_i*sub_dim+sub_j] = PI[p2_i*dim+p2_j];

  if (sub_i < sub_dim && sub_j < sub_dim &&
  sub_i*sub_dim+sub_j < sub_dim*sub_dim &&
  c_i < dim && c_j < dim) {
    c_sh_D[sub_i*sub_dim+sub_j] = D[c_i*dim+c_j];
    c_sh_PI[sub_i*sub_dim+sub_j] = PI[c_i*dim+c_j];
  }
  else {
```

```
    c_sh_D[sub_i*sub_dim+sub_j] = INT_MAX;
    c_sh_PI[sub_i*sub_dim+sub_j] = -1;
  }

  __syncthreads();

  for (int sub_k = 0; sub_k < min(sub_dim, dim); sub_k++) {
    __syncthreads();
    if (c_i < max(sub_dim, dim) && c_j < max(sub_dim, dim) &&
    p1_sh_D[sub_i*sub_dim+sub_k] < INT_MAX &&
    p2_sh_D[sub_k*sub_dim+sub_j] < INT_MAX) {
      if (c_sh_D[sub_i*sub_dim+sub_j] > p1_sh_D[sub_i*sub_dim+sub_k] +
      p2_sh_D[sub_k*sub_dim+sub_j]) {
        c_sh_D[sub_i*sub_dim+sub_j] = p1_sh_D[sub_i*sub_dim+sub_k] +
        p2_sh_D[sub_k*sub_dim+sub_j];
        c_sh_PI[sub_i*sub_dim+sub_j] = p2_sh_PI[sub_k*sub_dim+sub_j];
      }
    }
    __syncthreads();
  }

  __syncthreads();

  if (sub_i < min(sub_dim, dim) && sub_j < min(sub_dim, dim) &&
  sub_i*sub_dim+sub_j < sub_dim*sub_dim &&
  c_i < max(sub_dim, dim) && c_j < max(sub_dim, dim)) {
    D[c_i*dim+c_j] = c_sh_D[sub_i*sub_dim+sub_j];
    PI[c_i*dim+c_j] = c_sh_PI[sub_i*sub_dim+sub_j];
  }
}

void Blocked_Floyd_Warshall_Cuda (int* W, int* D, int* PI, unsigned int
   dim) {
  unsigned int n = dim*dim;

    /* Error varijabla za handleanje CUDA errora. */
    cudaError_t err = cudaSuccess;

  /* Alociranje device varijabli matrica D i PI. */
    int* d_D = NULL;
    err = cudaMalloc((void**) &d_D, n*sizeof(int));
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno alociranje matrice D (error
  code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    int* d_PI = NULL;
    err = cudaMalloc((void**) &d_PI, n*sizeof(int));
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno alociranje matrice PI (error
  code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

  /* Kopiranje podataka iz host matrica u device. */
    err = cudaMemcpy(d_D, D, n*sizeof(int), cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
```

```
      fprintf(stderr, "Neuspjesno kopiranje matrice D iz hosta u device
(error code %s)!\n", cudaGetErrorString(err));
      exit(EXIT_FAILURE);
  }

  err = cudaMemcpy(d_PI, PI, n*sizeof(int), cudaMemcpyHostToDevice);
  if (err != cudaSuccess) {
      fprintf(stderr, "Neuspjesno kopiranje matrice PI iz hosta u device
(error code %s)!\n", cudaGetErrorString(err));
      exit(EXIT_FAILURE);
  }

/* Pozivanje CUDA kernela. */
int blockDim = 32;
/* Broj (primarnih) blokova u blocked Floyd-Warshall algoritmu. */
int numberOfBlocks = ceil((float)dim/(float)blockDim);

cout << "Blocked Floyd-Warshall algoritam se pokrece sa " <<
   numberOfBlocks << " primarna bloka po dijagonali.\r\n";
cout << "CUDA kerneli se pokrecu kako slijedi: \r\n \t Faza 1: grid
   dimenzije 1x1 \r\n";
cout << "\t Faza 2: grid dimenzije " << numberOfBlocks-1 << "x2";
if (numberOfBlocks-1 == 0) cout << " (Faza 2 se nece izvrsiti zbog
   dimenzija grida)";
cout << "\r\n";
cout << "\t Faza 3: grid dimenzije " << numberOfBlocks-1 << "x" <<
   numberOfBlocks-1;
if (numberOfBlocks-1 == 0) cout << " (Faza 3 se nece izvrsiti zbog
   dimenzija grida)";
cout << "\r\n";
cout << "Svi blokovi se pokrecu s " << blockDim*blockDim << " threada po
    bloku.\r\n";

/* Iteriranje po blokovima radimo na CPU, ostalo paraleliziramo. */
for (int B = 0; B < numberOfBlocks; B++) {

  /* Velicina shared memorije je blockDim*blockDim za matricu D
      i za matricu PI. */
  FW_Cuda_Phase1<<<dim3(1, 1, 1), dim3(blockDim, blockDim, 1),
  2*blockDim*blockDim*sizeof(int)>>> (B, d_D, d_PI, dim, blockDim);

  err = cudaGetLastError();
  if (err != cudaSuccess) {
    fprintf(stderr, "Neuspjesno pokrenuta kernel metoda FW_Cuda_Phase1
    (error code %s)!\n", cudaGetErrorString(err));
    cout << "\r\n B = " << B << "\r\n";
    exit(EXIT_FAILURE);
  }

  cudaThreadSynchronize();

  /* Velicina shared memorije je blockDim*blockDim za primarnu matricu D
     ,
      trenutnu matricu D, primarnu i trenutnu za matricu PI. */
  if (numberOfBlocks-1 > 0) {
    FW_Cuda_Phase2<<<dim3(numberOfBlocks-1, 2, 1),
    dim3(blockDim, blockDim, 1),
    4*blockDim*blockDim*sizeof(int)>>> (B, d_D, d_PI, dim, blockDim);
```

```
    err = cudaGetLastError();
    if (err != cudaSuccess) {
      fprintf(stderr, "Neuspjesno pokrenuta kernel metoda FW_Cuda_Phase2
      (error code %s)!\n", cudaGetErrorString(err));
      cout << "\r\n B = " << B << "\r\n";
      exit(EXIT_FAILURE);
    }
  }

  cudaThreadSynchronize();

  /* Velicina shared memorije je blockDim*blockDim za trenutnu matricu D
     ,
      dvije primarne matrice D izracunate u fazi 2, te za pripadne
         matrice PI. */
  if (numberOfBlocks-1 > 0) {
    FW_Cuda_Phase3<<<dim3(numberOfBlocks-1, numberOfBlocks-1, 1),
    dim3(blockDim, blockDim, 1), 6*blockDim*blockDim*sizeof(int)>>>
    (B, d_D, d_PI, dim, blockDim);

    err = cudaGetLastError();
    if (err != cudaSuccess) {
      fprintf(stderr, "Neuspjesno pokrenuta kernel metoda FW_Cuda_Phase3
      (error code %s)!\n", cudaGetErrorString(err));
      cout << "\r\n B = " << B << "\r\n";
      exit(EXIT_FAILURE);
    }
  }

  /* Sinkronizacija threadova kako bi se zavrsila B-ta iteracija,
      te kako bi se preslo na (B+1). iteraciju. */
  cudaThreadSynchronize();
}

/* Kopiranje podataka iz device matrica u host. */
  err = cudaMemcpy(D, d_D, n*sizeof(int), cudaMemcpyDeviceToHost);
  if (err != cudaSuccess) {
      fprintf(stderr, "Neuspjesno kopiranje matrice D iz devicea u host
(error code %s)!\n", cudaGetErrorString(err));
      exit(EXIT_FAILURE);
  }

  err = cudaMemcpy(PI, d_PI, n*sizeof(int), cudaMemcpyDeviceToHost);
  if (err != cudaSuccess) {
      fprintf(stderr, "Neuspjesno kopiranje matrice PI iz devicea u host
(error code %s)!\n", cudaGetErrorString(err));
      exit(EXIT_FAILURE);
  }

/* Dealociranje device varijabli matrica D i PI. */
err = cudaFree(d_D);
  if (err != cudaSuccess) {
      fprintf(stderr, "Neuspjesno dealociranje matrice D
(error code %s)!\n", cudaGetErrorString(err));
      exit(EXIT_FAILURE);
  }
```

```
    err = cudaFree(d_PI);
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno dealociranje matrice PI
  (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

  /* Reset CUDA devicea i zavrsavanje CUDA Floyd-Warshalla. */
    err = cudaDeviceReset();
    if (err != cudaSuccess) {
        fprintf(stderr, "Neuspjesno resetiranje devicea (zavrsavanje sa
  CUDA FW, priprema za sljedece pokretanje)! error=%s\n",
  cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

}

/* Metoda koja rekonstruira tezinu najkraceg puta za dani par
    vrhova koristeci matricu prethodnika PI i matricu inicijalnih tezina W
       . */
int getPath (int* W, int* PI, int i, int j, unsigned int dim) {
  if (i == j) {
    return 0;
  }
    else if (PI[i*dim+j] == -1) {
    return infty;
  }
  else {
    int recursivePath = getPath(W, PI, i, PI[i*dim+j], dim);
    if (recursivePath < infty) {
      return recursivePath + W[PI[i*dim+j]*dim+j];
    }
    else {
      return infty;
    }
  }
}

/* Za svaki par vrhova pokrece getPath metodu koja rekonstruira
    tezinu najkraceg puta izmedu njih koristeci matricu prethodnika PI.
    Tu tezinu onda usporeduje sa dobivenom tezinom
   za isti par vrhova u matrici najkracih putova D. */
bool checkSolutionCorrectness (int* W, int* D, int* PI, unsigned int dim)
   {
  for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
      if (getPath(W, PI, i, j, dim) != D[i*dim+j]) {
        return false;
      }
    }
  }

  return true;
}

int main() {
```

```
/*
   V - broj vrhova
   E - broj bridova
   u - prvi vrh pri ucitavanju grafa iz datoteke
   v - drugi vrh pri ucitavanju grafa iz datoteke
   w - tezina izmedu v1 i v2 pri ucitavanju grafa iz datoteke
*/
unsigned int V, E;
int u, v, w;

ifstream inputGraphFile; inputGraphFile.open("graphFile.txt");
ofstream outputFile; outputFile.open("output_cuda_blocked.txt");

inputGraphFile >> V >> E;
cout << "V = " << V << ", E = " << E << "\r\n";

unsigned int n = V*V;

/* Inicijalizacija grafova u memoriji. */
int* W = (int*)malloc(n*sizeof(int));
int* D = (int*)malloc(n*sizeof(int));
int* PI = (int*)malloc(n*sizeof(int));


/* Postavljanje inicijalnih vrijednosti za matricu prethodnika PI(0),
   matricu tezina W i matricu najkracih putova D(0). */
fill_n(W, n, infty);
fill_n(PI, n, -1);

for (int i = 0; i < E; i++) {
  inputGraphFile >> u >> v >> w;
  //cout « u « " <- " « w « " -> " « v « " | ";

  W[u*V+v] = w;
  if (u != v) {
    PI[u*V+v] = u;
  }
}

for (int i = 0; i < V; i++) {
  W[i*V+i] = 0;
}

/* D(0) = W na pocetku. */
memcpy (D, W, n*sizeof(int));

// printMatrix(W, V); printMatrix(D, V); printMatrix(PI, V);

/* Pocetak mjerenja izvrsavanja Floyd-Warshall algoritma. */
clock_t begin = clock();

/* Pozivamo Floyd-Warshall CPU algoritam nad ucitanim grafom. */
Blocked_Floyd_Warshall_Cuda(W, D, PI, V);

/* Kraj mjerenja izvrsavanja Floyd-Warshall algoritma. */
clock_t end = clock();
double elapsedTime = double(end - begin) / CLOCKS_PER_SEC;
```

```
  //printMatrix(W, V); printMatrix(D, V); printMatrix(PI, V);

  /* Ispis rezultata u datoteku. */
  outputFile << "|V| = " << V << ", |E| = " << E << "\r\n\r\n";
  for (int i = 0; i < n; i++) {
    if (i%V==0) outputFile << "\r\n";
    if (D[i] < infty)
      outputFile << D[i] << "\t";
    else
      outputFile << "infty" << "\t";
  }
  outputFile << "\r\n\r\n";
  for (int i = 0; i < n; i++) {
    if (i%V==0) outputFile << "\r\n";
    outputFile << PI[i] << "\t";
  }

  cout << "Vrijeme izvrsavanja Blocked Floyd-Warshall algoritma: " <<
  elapsedTime << "s.\r\n";

  if (checkSolutionCorrectness(W, D, PI, V) == true)
    cout << "Svi najkraci putevi su tocno izracunati!\r\n";
  else
    cout << "Najkraci putevi nisu tocno izracunati.\r\n";

  inputGraphFile.close();
  outputFile.close();
  free(W); free(D); free(PI);

  return 0;
}
```

8 LITERATURA

[1] *CUDA Parallel Computing Platform* (`http://www.nvidia.com/object/cuda_home_new.html`, 07. 12. 2016.)

[2] *CUDA Toolkit Documentation v8.0* (`http://docs.nvidia.com/cuda/index.html`, 12. 12. 2016.)

[3] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, *Introduction to Algorithms*, McGraw-Hill Book Company, 3rd edition, Cambridge, Massachusetts, SAD, 2009.

[4] R. Diestel, *Graph Theory*, Springer-Verlag, 3rd edition, New York, SAD, 2005.

[5] M. L. Fredman, R. E. Tarjan, *Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms*, 2013 IEEE 54th Annual Symposium on Foundations of Computer Science, 1984.

[6] P. Harish, P. J. Narayanan, *Accelerating large graph algorithms on the GPU using CUDA*, International Institute of Information Technology Hyderabad, India, 2007.

[7] T. Hagerup, *Improved Shortest Paths on the Word RAM*, ICALP '00 Proceedings of the 27th International Colloquium on Automata, Languages and Programming (pp. 61-72), Springer-Verlag, London, UK, 2000.

[8] T. Harju, *Lecture Notes on Graph Theory*, CreateSpace Independent Publishing Platform, 2014.

[9] S. Hong, S. K. Kim, T. Oguntebi, K. Olukotun, *Accelerating CUDA Graph Algorithms at Maximum Warp*, Computer Systems Laboratory, Stanford University, Stanford, California, SAD, 2011.

[10] J. JaJa, *Introduction to Parallel Algorithms*, Addison-Wesley Professional, 1st edition, 1992.

[11] D. B. Johnson, *Efficient Algorithms for Shortest Paths in Sparse Networks*, Journal of the ACM, 1977.

[12] G. J. Katz, J. T. Kider, *All-Pairs Shortest-Paths for Large Graphs on the GPU*, University of Pennsylvania, Philadelphia, Pennsylvania, SAD, 2008.

[13] M. J. Kemp, *All-Pairs Shortest Path Algorithms Using CUDA*, Durham University, Durham, North East England, UK, 2012.

[14] Kent State University, Ohio, SAD - Department of Computer Science, nastavni materijali kolegija *Graph Theory* (`http://www.personal.kent.edu/~rmuhamma/GraphTheory/MyGraphTheory/planarity.htm`, 25. 11. 2016.)

[15] KTH Royal Institute of Technology, Stockholm, Švedska - School of Engineering Sciences, nastavni materijali kolegija *Parallel Computations for Large-Scale Problems* (`http://www.math.kth.se/na/SF2568/`, 25. 11. 2016.)

[16] T. Lappas, K. Liu, E. Terzi, *Finding a team of experts in social networks*, KDD '09 Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 467-476), New York, New York, SAD, 2009.

[17] A. Maheshwari, M. Smid, *Introduction to Theory of Computation*, School of Computer Science, Carleton University, Ottawa, Canada, 2016.

[18] Massachusetts Institute of Technology, Cambridge, SAD - nastavni materijali seminara *Undergraduate seminar in Discrete Mathematics* (`http://math.mit.edu/~rothvoss/DiscreteMath1PMSpring2013.html`, 20. 11. 2016.)

[19] *Neo4j: The World's Leading Graph Database* (`https://neo4j.com/`, 10. 01. 2017.)

[20] S. Pettie, *A new approach to all-pairs shortest paths on real-weighted graphs*, Theoretical Computer Science - Special issue on automata, languages and programming, Volume 312, Issue 1 (pp. 47–74), Elsevier Science Publishers Ltd., Essex, UK, 2004.

[21] J. Sanders, E. Kandrot, *CUDA By Example - An Introduction to General-Purpose GPU Programming*, Addison-Wesley, Ann Arbor, Michigan, SAD, 2011.

[22] C. Sommer, *Approximate Shortest Path and Distance Queries in Networks*, Department of Computer Science, University of Tokyo, Japan, 2010.

[23] C. Stevens, *GPU-Optimized Graph Theory Analysis of Allosteric Protein Signaling Networks*, Department of Computer Science, Wake Forest University, Winston-Salem, North Carolina, SAD, 2015.

[24] P. Tang, *Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-core Computers*, Department of Computer Science, University of Arkansas, Little Rock, Arkansas, SAD, 2014.

[25] *Tesla C2050 / C2070 GPU Computing Processor Specification* (`http://www.nvidia.com/docs/io/43395/nv_ds_tesla_c2050_c2070_apr10_final_lores.pdf`, 17. 12. 2016.)

[26] G. Venkataraman, S. Sahni, S. Mukhopadhyaya, *A Blocked All-Pairs Shortest-Paths Algorithm*, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, Florida, SAD, 2003.

[27] R. Williams, *Faster all-pairs shortest paths via circuit complexity*, STOC '14 Proceedings of the forty-sixth annual ACM symposium on Theory of computing (pp. 664-673), New York, New York, SAD, 2014.

SAŽETAK

U ovom diplomskom radu definirani su osnovni pojmovi teorije grafova kao baza za problem najkraćeg puta u grafu. Opisane su varijante tog problema, te način reprezentacije grafova u računalu, a potom je detaljno analiziran Floyd-Warshall algoritam za rješavanje problema najkraćih putova među svim parovima vrhova. Definirano je dinamičko programiranje obzirom da je Floyd-Warshall algoritam primjer istog. Nakon toga, detaljno je analizirana `CUDA` platforma za hibridno CPU-GPU računarstvo - njena arhitektura, njene prednosti i mane, a obzirom da se radi o platformi za paralelno računarstvo dana je i povijest istog. Temeljem svih zaključaka, implementirane su tri verzije Floyd-Warshall algoritma: jednostavna `C++` implementacija algoritma koja je analogon danom algoritmu pisanom u pseudojeziku, jednostavna `CUDA` implementacija koja je iskoristila prednosti arhitekture grafičke kartice, te napredna `CUDA` implementacija koja je na algoritamskoj razini dodatno paralelizirala Floyd-Warshall algoritam. Na kraju, izloženi su eksperimentalni rezultati sve tri implementacije na grafovima raznih veličina i razne gustoće.

Ključne riječi: najkraći putovi u grafu, Floyd-Warshall algoritam, `CUDA` akcelerirano računarstvo

SUMMARY

At the beginning of this thesis some elementary terms from graph theory are defined as a foundation for the shortest path problem. A few varieties of the aforementioned problem are described, as well as the method of representing graphs in computer memory. After that, Floyd-Warshall algorithm for all-pairs shortest-path problem is analized in detail. Dynamic programming is also defined as Floyd-Warshall algorithm is a prime example of such a method. Additionally, `CUDA` platform for hybrid CPU-GPU computing is analized - it's architecture, it's advantages and disadvantages, but also, since it is a parallel computing platform, the history of parallel computing is given. Based on all the conclusions, three versions of Floyd-Warshall algorithm have been implemented: a simple `C++` implementation of the algorithm which is analogous to the given pseudocode, a simple `CUDA` implementation which used the advantages of the graphics card architecture and an advanced (blocked) `CUDA` implementation which further made use of the Floyd-Warshall's algorithm parallel nature. At the end, the experimental results of all three implementations on different sized graphs and graphs of various densities have been laid out.

Key words: shortest paths in a graph, Floyd-Warshall algorithm, `CUDA` accelerated computing

ŽIVOTOPIS

Mislav Karaula je rođen 05. studenog 1992. godine u Vinkovcima. Osnovnu školu je pohađao u Osnovnoj školi Antun Gustav Matoš u Vinkovcima, te je osnovnoškolsko obrazovanje završio 2007. godine. Iste godine upisao je prirodoslovno-matematičku gimnaziju u Gimnaziji Matije Antuna Reljkovića u Vinkovcima, gdje 2011. godine završava srednjoškolsko obrazovanje. Tijekom osnovne i srednje škole sudjelovao je na općinskim i županijskim natjecanjima iz više nastavnih predmeta, pretežno iz matematike i informatike. Po završetku srednjoškolskog obrazovanja, 2011. godine, upisao je sveučilišni preddiplomski studij matematike na Odjelu za matematiku Sveučilišta Josipa Jurja Strossmayera u Osijeku kojeg uspješno završava 2014. godine. Daljnje obrazovanje nastavlja na Odjelu za matematiku gdje je iste godine upisao sveučilišni diplomski studij matematike, smjer Matematika i računarstvo. U ljeto 2014. godine sudjelovao je u međunarodnoj razmjeni studenata prirodnih i tehničkih znanosti preko studentske organizacije IAESTE, temeljem čega je proveo 8 tjedana na odjelu School of Engineering Sveučilišta u Cardiffu, Velika Britanija, a tijekom prve godine diplomskog studija odradio je tromjesečnu stručnu praksu u tvrtki Span d.o.o.