SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2321

SWIG - Poravnanje struktura korištenjem iterativne primjene Smith-Waterman algoritma

Bruno Rahle

Zagreb, lipanj 2012.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

Popis slika **vi**

Popis tablica **vii**

1. Uvod **1**

2. Poravanavanje sturktura **3**

3. Smith-Watermanov algoritam **4**

3.1. Needleman-Wunschov algoritam 4

3.1.1. Ulazni podaci 4

3.1.2. Izlazni podaci 5

3.1.3. Algoritam 5

3.1.4. Primjer 6

3.1.5. Analiza složenosti 7

3.2. Smith-Watermanov algoritam 7

3.2.1. Ulazni podaci 7

3.2.2. Izlazni podaci 7

3.2.3. Algoritam 7

3.2.4. Primjer 8

3.2.5. Analiza složenosti 9

3.3. Procjepi 9

3.4. Memorijska optimizacija 10

3.4.1. Hirschbergov algoritam 11

3.5. Substitucija substitucijske matrice 12

4. Algoritam simuliranog kaljenja **13**

4.1. Algoritam 13

4.1.1. Odabir susjeda 14

4.1.2. Računanje energije 15

4.1.3. Računanje vjerojatnosti prihvaćanja 15
4.2. Modifikacije 17
4.2.1. Više susjeda 17
4.2.2. Ponovo pokretanje 18
4.3. Parametari 18

5. CUDA tehnologija **19**

6. Implementacija **21**
6.1. Implementacija simuliranog kaljenja 21
6.2. Implementacija Smith-Watermanovog algoritma 22

7. Rezultati **24**

8. Zaključak **25**

Literatura **26**

POPIS SLIKA

POPIS TABLICA

1. Uvod

Živimo u doba kada velikih promjena. Današnjim generacijama mladih nezamisliv je život bez Interneta i društvenih mreža. Internet, a kamoli društvene mreže, bili su nezamislivi njihovim roditeljima dok su bili mladi. Računala su bila (većinom i ostala) neshvatljiva čuda djedovima i bakama, dok je njihovim roditeljima čak i električna energija strana.

Nije samo računarstvo doživjelo ogromne promjene. Tek pred nešto više od stotinu godinu izumljen je automobil. Danas gotovo da ne postoji obitelj koja ne posjeduje barem jedan.

Još je veliki hrvatski pjesnik Petar Preradović u pjesmi "Mujezin" zapisao "Stalna na tom svijetu samo mijena jest." Ono što danas smatramo znanstvenom fantastikom za nekoliko bi godina moglo postati stvarnost. Kako se povećava ljudsko znanje, tako se povećava i broj pitanja na koje ljudi traže odgovore. Često problemi s kojima se znanost susreće prelaze granice isključivo jedne discipline. Zbog toga je postao običaj da ljudi različitih profila sudjeluju na istom istraživanju.

Postoji čitav niz problema iz biologije koji se danas više ili manje efikasno rješavaju uz pomoć kompjutera. Ti problemi spadaju u područje koje nazivamo bioinformatikom. Glavni problema koje rješava bioinformatika su:

- Analiza nizova (engl. *sequence analysis*).
- Označavanje genoma (engl. *genome annotation*).
- Računska evolucijska biologija (engl. *computational evolutionary biology*).
- Dubinska analiza teksta (engl. *literature analysis*, *data mining*).
- Analiza izražaja gena (engl. *analysis of gene expression*).
- Analiza nadzora (engl. *analysis of regulation*).
- Analiza izražaja proteina (engl. *analysis of protein expression*).
- Analiza mutacija u raku (engl. *analysis of mutations in cancer*).
- Usporedna genomika (enl. *comparative genomics*).
- Modeliranje bioloških sistema (engl. *modeling biological systems*).
- Visoko-propusna anliza slika (engl. *high-throughput image analysis*).
- Predviđanje struktura proteina (engl. *protein structure prediction*).

– Interakcija među molekulama (engl. *molecular Interaction*).

Problem koji mi rješavamo, poravnavanje struktura, može se staviti u više područja koje smo naveli, budući da im se područja preklapaju. Taj problem bitan je jer, ako ga uspješno riješimo, napravilo smo veliki posao jer bi

- pričaj još o: zašto je problem koji si rješavao bitan, kome koristi, kako se može koristiti i potencijalno za što se sve može koristiti.

- spomeni i grafičke kartice

- cca 2 stranice

- nešto sitno i o grafičkim karticama

2. Poravanavanje sturktura

U ovom radu bavit ćemo se rješavanjem slijedećeg problema:

Zadana su nam dva proteina, tj. pozicije u prostoru svakog od atoma koje protein sadrži. Zanimaju nas samo atomi ugljika, a ostali atomi nam nisu interesantni. Želimo jednog od njih transforimarati koristeći samo translacije i rotacije tako da se što je moguće više preklapa s drugim. Protein koji ćemo rotirati zvat ćemo protein B, a protein A bit će onaj s kojim ga želimo preklopiti.

Kao rješenje želimo dobiti preklapanje koje će poravnati neki podniz elemenata iz proteina A i B i koje će biti maksimalno za ta dva proteina. Zbog toga nas zanima i rekonstrukcija rješenja.

Da bismo dobili rješenje, koristit ćemo simulirano kaljenje zajedno sa Smith-Watermanovim algoritmom za proračun energije. Taj par algoritama trebao bi moći u "pristojnom" vremenu pronaći neko poravnanje koje je relativno blisko optimalnom.

Inovativnost ovog rada temelji se na tome da ćemo kao arhitekturu koristi GPU, tj. grafičku procesnu jedinicu. Prednost GPU arhitekture u odnosu na CPU jest to što je GPU masivno paralelan, što znači da imamo velik broj dretvi (broje se u stotinama!) koje nam omogućavaju da stvari rješavamo paralelno. Najzrelija tehnologija koja nudi korištenje grafičkih kartica u svrhu procesiranja podataka je CUDA. Unatoč nekim ograničenjima (npr. radi samo na Nvidijinim grafičkim karticama), odlučili smo koristiti upravo nju za potrebe implementacije rješenja ovog problema.

- detaljan opis problema - cca 1-2 stranice

3. Smith-Watermanov algoritam

Smith-Watermanov algoritam služi nam da bismo pronašli lokalno poravnanje. Osmislili su ga Temple F. Smith i Michael S. Waterman 1981. Smith (1981). Temelji se na Needleman-Wunschevom algoritmu (Needleman (1970)) te i sam spada u kategoriju algoritama dinamičkog programiranja. Glavna razlika između ta dva algoritma jest što Needleman-Wunschov algoritam prolazi globalno poravananje.

- napisi jos teksta ovdje.

3.1. Needleman-Wunschov algoritam

Algoritam, kao što je već rečeno, traži globalno poravnanje. To znači da se svi članovi ulaznih nizova moraju poravnati. Dopuštene operacije kada tražimo poravnanje su preklapanje s elementom iz suprotnog niza i ubacivanje praznina u neki od nizova. Sve parove elemenata u dobivenom preklapanju bodujemo i na osnovu te ocjene određujemo sličnost nizova. Konačan rezultat ovog algoritma jest poravnanje koje maksimizira takvu ocjenu, tj. daje maksimalno globalno poravnanje.

Zanimljivost je da je to prvi algoritam dinamičkog programiranja ikada primjenjen u bioinformatici.

3.1.1. Ulazni podaci

1. Dva niza (A i B) proteina ili nukleotida. Zbog jednostavnosti, pretpostavit ćemo da su to nukleotidi iz DNK - adenin (A), timin (T), gvanin (G) i citozin (C). U primjeru ćemo koristiti $A = "ATGCCGTA"$ i $B = "TGCACTA"$. Dužinu niza A označit ćemo s N, a dužinu niza B s M.

2. Supstitucijska matrica S, koja nam daje bodove koje dobijemo kada jedan nukletoid preklopimo s drugim. U našem će slučaju imati dimenzije 4×4, budući da ćemo razmatrati slučaj kada imamo samo četiri nukleotida. U principu će na dijagonali imati pozitivne brojeve, a na ostalim poljima negativne. To znači da nam se najviše isplati

preklapati nukletide istog tipa, jer za to dobivamo bodove, a inače ih gubimo. Primjer jedne takve matrice koju ćemo koristiti i u primjeru:

	A	C	G	T
A	10	-3	-9	-1
C	-5	8	-8	-7
G	-5	-4	7	-5
T	-4	-11	-8	9

3. Negativan broj d, koji označava bodove koje dobijemo (tj. izgubimo) kada nukleotid preklopimo s prazninom. U primjeru ćemo koristiti $d = -5$.

3.1.2. Izlazni podaci

1. Broj H, ocjena najboljeg globalnog poravnanja.

2. Dva nova niza jednake dužine, A' i B', nastala ubacivanjem praznina (označenih najčešće sa '-') u nizove A i B koja predstavljaju najbolje pronađeno poravnanje.

3.1.3. Algoritam

Neka nam matrica F služi za računanje poravnanja. Tada će nam $F_{i,j}$ označavati maksimalan broj bodova koje možemo dobiti kada poravnamo prvih i članova niza A i prvih j članova niza B. $F_{i,j}$ možemo računati rekurzijom na slijedeći način:

$$F_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ i } j = 0 \\ F_{i-1,j} + d & \text{ako je } i > 0 \text{ i } j = 0 \\ F_{i,j-1} + d & \text{ako je } i = 0 \text{ i } j > 0 \\ max \begin{pmatrix} F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ F_{i-1,j} + d \\ F_{i,j-1} + d \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

U $F_{N,M}$ će nam stoga pisati maksimalno globalno poravnanje. Primjetite da u niti jednom slučaju nećemo poravnati dvije praznine. Ako bismo to učinili, samo bismo izgubili bodove, budući da je d nužno negativan broj.

Da bismo znali rekonstruirati rješenje, koristit ćemo matricu R. U polju $R_{i,j}$ pisat će koje smo polje matrice F koristili da bi došli u polje $F_{i,j}$. Kako su jedine mogućnosti $F_{i-1,j}$, $F_{i,j-1}$, $F_{i-1,j-1}$ i da nismo došli iz nikojeg polja (to vrijedi jedino za polje $F_{0,0}$), koristit ćemo redom oznake A, B, O i X.

$$R_{i,j} = \begin{cases} X & \text{ako je } \begin{pmatrix} i = 0 \\ j = 0 \end{pmatrix} \\ O & \text{ako je } \begin{pmatrix} i > 0 \\ j > 0 \\ F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \geq F_{i-1,j} + d \\ F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \geq F_{i,j-1} + d \end{pmatrix} \\ A & \text{ako je } \begin{pmatrix} i > 0 \\ j = 0 \end{pmatrix} \text{ili } \begin{pmatrix} i > 0 \\ j > 0 \\ F_{i-1,j} + d > F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ F_{i-1,j} + d \geq F_{i,j-1} + d \end{pmatrix} \\ B & \text{ako je } \begin{pmatrix} i = 0 \\ j > 0 \end{pmatrix} \text{ili } \begin{pmatrix} i > 0 \\ j > 0 \\ F_{i,j-1} + d > F_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ F_{i,j-1} + d > F_{i-1,j} + d \end{pmatrix} \end{cases}$$

Rekonstrukciju provodimo tako krenemo iz polja $R_{N,M}$ i krećemo se po matrici unazad dok ne dođemo do polja na kojem piđe X, tj. $R_{0,0}$. Ako na polju pročitamo O, pomičemo se po dijagonili, tj. u izlazni niz spremimo par (A_{i-1}, B_{j-1}) te smanjimo i i i j. Ako pročitamo A, spremamo par $(A_{i-1}, -)$ te smanjimo samo i za jedan. U slučaju da pročitamo B, spremamo par $(-, B_{j-1})$ te smanjujemo j za jedan. Ako smo pročitali X, došli smo do kraja i generirali smo izlazni niz, ali u obrnutom redosljedu.

3.1.4. Primjer

Za prethodno navedene ulazne podatke, matrice F i R izgledat će ovako:

$F =$

	-	T	G	C	A	C	T	A
-	0	-5	-10	-15	-20	-25	-30	-35
A	-5	-1	-6	-11	-5	-10	-15	-20
T	-10	4	-1	-6	-10	-15	-1	-6
G	-15	-1	11	6	1	-4	-6	-6
C	-20	-6	6	19	14	9	4	-1
C	-25	-11	1	14	14	22	17	12
G	-30	-16	-4	9	9	17	17	12
T	-35	-21	-9	4	5	12	26	21
A	-40	-26	-14	-1	14	9	21	36

$R =$

	-	T	G	C	A	C	T	A
-	X	B	B	B	B	B	B	B
A	A	O	B	B	O	B	B	O
T	A	O	B	B	A	A	O	B
G	A	A	O	B	B	B	A	O
C	A	A	A	O	B	O	B	B
C	A	A	A	O	O	O	B	B
G	A	A	O	A	O	A	O	O
T	A	O	A	A	O	A	O	B
A	A	A	A	A	O	B	A	O

Iz tih podataka lagano je napraviti rekonstrukciju (polja označena sivom bojom). Stoga zaključujemo da je traženo globlano poravnanje $A' =$ "ATGCCGTA" i $B' =$ "-TGCACTA".

3.1.5. Analiza složenosti

Trivijalno je vidljivo da su memorijska i vremenska složenost opisanog algoritma jednake $O(NM)$.

3.2. Smith-Watermanov algoritam

Razlika njega i prethodno opisanog Needleman-Wunschevog algoritma jest u tome što ovaj algoritam traži najbolje lokalno poravnanje. To znači da ne koristi nužno cijele nizove proteina ili nukleotida već samo najsličnije uzastopne podnizove. U praksi se koriste nešto poboljšane verzije ovog algoritma.

3.2.1. Ulazni podaci

Vidi odlomak 3.1.1.

3.2.2. Izlazni podaci

Vidi odlomak 3.1.2.

3.2.3. Algoritam

U ovom ćemo algoritmu koristiti matricu H za računanje poravnanja. Za razliku od Needleman-Wunschevog algoritma, $H_{i,j}$ ovaj će put označavati rezultat najboljeg lokalnog poravnanja koje koristi A_{i-1} i B_{j-1}. Matricu ćemo popuniti na slijedeći način:

$$H_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ ili } j = 0 \\ \max \begin{pmatrix} 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H_{i-1,j} + d \\ H_{i,j-1} + d \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

Primjetite bitnu razliku u odnosu na Needleman-Wunschev algoritam - u ovom slučaju matrica H neće sadržavati negativne brojeve. Rješenje, tj. najbolje lokalno poravnanje više neće pisati na polju $H_{N,M}$. Ono će biti na polju (t, u) na kojem se nalazi najveći broj u matrici.

Da bismo znali rekonstruirati takvo lokalno poravnanje, koristit ćemo matricu R koju gradimo na sličan način kao i kod prethodnog algoritma:

$$R_{i,j} = \begin{cases} X & \text{ako je} H_{i,j} = 0 \\ O & \text{ako je} \begin{pmatrix} i > 0 \\ j > 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \geq H_{i-1,j} + d \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \geq H_{i,j-1} + d \end{pmatrix} \\ A & \text{ako je} \begin{pmatrix} i > 0 \\ j = 0 \end{pmatrix} \text{ili} \begin{pmatrix} i > 0 \\ j > 0 \\ H_{i-1,j} + d > H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H_{i-1,j} + d \geq H_{i,j-1} + d \end{pmatrix} \\ B & \text{ako je} \begin{pmatrix} i = 0 \\ j > 0 \end{pmatrix} \text{ili} \begin{pmatrix} i > 0 \\ j > 0 \\ H_{i,j-1} + d > H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H_{i,j-1} + d > H_{i-1,j} + d \end{pmatrix} \end{cases}$$

Rekonstrukcija se, slično kao i kod Needleman-Wunschevog algoritma, izvodi tako da krenemo iz polja (t, u) i krećemo se po matrici R dok ne dođemo do polja na kojem piše X koje ovaj put ne mora nužno biti polje $R_{0,0}$. Dobiveni niz okrenemo i dobit ćemo traženo poravnanje.

3.2.4. Primjer

Za prethodno navedene ulazne podatke, matrice H i R izgledat će ovako:

$H =$

	-	T	G	C	A	C	T	A
-	0	0	0	0	0	0	0	0
A	0	0	0	0	10	5	0	10
T	0	9	4	0	5	0	14	9
G	0	4	16	11	6	1	9	9
C	0	0	11	24	19	14	9	4
C	0	0	6	19	19	27	22	17
G	0	0	7	14	14	22	22	17
T	0	9	4	9	10	17	31	26
A	0	4	0	4	19	14	26	41

$R =$

	-	T	G	C	A	C	T	A
-	X	X	X	X	X	X	X	X
A	X	X	X	X	O	B	B	O
T	X	O	B	X	A	A	O	B
G	X	A	O	B	B	O	A	O
C	X	X	A	O	B	O	B	O
C	X	X	A	O	O	O	B	B
G	X	X	O	A	O	A	O	O
T	X	O	B	A	O	A	O	B
A	X	A	O	A	O	B	A	O

Prema tome, traženo poravnanje jest $A' =$ "TGC-CGTA" i $B' =$ "TGCAC-TA".

3.2.5. Analiza složenosti

Trivijalno je vidljivo da su memorijska i vremenska složenost i ovog algoritma jednake $O(NM)$.

3.3. Procjepi

Do sada smo razmatrali samo slučaj kada je cijena otvaranja procjepa i njegova proširivanja jednaka (d). Taj slučaj nazivamo *linearnom ocjenom procjepa*, budući da, ako je k dužina procjepa, dk je cijena za taj procjep. U praksi se primjenjuju još dva načina ocjenjivana procjepa.

Jedan je da za svaki procjep, bez obzira na njegovu dužinu, platimo fiksnu cijenu (c), a nazivamo ga *konstantnom ocjenom procjepa*.

Drugi je kombinacija prethodna dva načina: za svaki procjep plaćamo fiksnu cijenu (c), ali za njegovo produženje plaćamo neku drugu cijenu (d). Takvu funkciju ocjene procjepa nazivamo *Afinom ocjenom procjepa*. Prema tome, za procjep dužine k, platit ćem cijenu jednaku $c + (k-1)d$. Kako je empirijski pokazano da je ta funkcija u biti parobala, ovakva je ocjena ujedno i najpreciznija. Nažalost, ona otežava računanje matrice H (i R). Problem je najlakše riješiti tako da uvedemo dvije nove matrice H^A i H^B koje će nam pomoći u računanju cijene procjepa. Matrice ćemo onda računati na slijedeći način:

$$H_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ ili } j = 0 \\ \max \begin{pmatrix} 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H^A_{i-1,j} + c \\ H^B_{i,j-1} + c \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

$$H^A_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ ili } j = 0 \\ \max \begin{pmatrix} 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H^A_{i-1,j} + d \\ H^B_{i,j-1} + c \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

$$H^B_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ ili } j = 0 \\ \max \begin{pmatrix} 0 \\ H_{i-1,j-1} + S_{A_{i-1},B_{j-1}} \\ H^A_{i-1,j} + c \\ H^B_{i,j-1} + d \end{pmatrix} & \text{ako je } i > 0 \text{ i } j > 0 \end{cases}$$

Rekonstrukcijska matrica vrlo je slična onoj u izvorno opisanom algoritmu, a kako ćemo u udućem potpoglavlju maknuti potrebu za njom, nećemo je posebno navoditi.

3.4. Memorijska optimizacija

S obzirom da nizovi nukleotida mogu sadržavati i do nekoliko milijuna elemenata, matrica dimenzija NM fizički ne stane u memoriju. Međutim, čim primjetimo da za potrebe generiranja matrice H (a ujedno i H^A odnosno H^B) koristimo samo polja iz trenutnog i prethodnog retka, postaje jasno da nam ostali reci koje smo obradili ranije njih nisu potrebni. Stoga možemo pamtiti samo ta dva retka i time korištenu memoriju za proračun matrice H smanjimo na samo $O(\min(N, M))$. Ipak, problem se javlja kada trebamo rekonstruirati rješenje pomoću matrice R, budući da se prilikom rekonstrukcije potencijalno vraćamo se kroz sva njena polja. U nastavku ćemo opisati rješenje tog problema.

-TODO: slika koja prikazuje da možemo koristiti samo dva posljednja retka

3.4.1. Hirschbergov algoritam

Dan Hirschberg osmislio je algoritam koji rješava problem rekonstrukcije globalnog poravnanja u $O(NM)$ vremena koristeći $O(\min(N, M))$ memorije. Njegova ideja temeljena je na metodi *podijeli-pa-vladaj*.

No, prije nego je objasnimo, promotrimo što će se dogoditi s globalnim poravnanjem ako okrenemo nizove A i B. Okrenute nizove nazivat ćemo A^R i B^R, a matrice F^R i R^R. Očito je da se ocjena poravnanja neće promijeniti, a prilikom rekonstrukcije dobit ćemo obrnuti niz. Uz sitne modifikacije, možemo izračunati matricu F^R bez da fizički okrećemo nizove.

Hirschbergov algoritam radi na slijedeći način:

1. Bez smanjenja općenitosti, možemo pretpostaviti da je niz A duži od niza B. Neka nam p označava polovicu dužine niza A, zaokruženu na dolje. Podijelimo niz A na dvije polovice, dužine p.

2. Na obje polovice niza ($A_{0..p}$ i $A_{p..N}$) pokrenemo gore opisani algoritam za pronalaženje lokalnog poravnanja koje koristi $O(\min(N, M))$ memorije, ali bez rekonstrukcije. Nad drugom polovicom niza radimo algoritam za obrnuto nizove, tako da je za obje polovice posljednji izračunati red onaj odabran u prethodnom koraku.

3. Definirajmo funkciju $\phi(j)$ kao sumu lokalnih poravnanja obje matrice u redu p, tj. kao:

   $$\phi(j) = F_{p,j} + F^R_{p,j}$$

   Barem će jedan član globalnog poravnanja, biti u retku p, a Hirschberg je pokazao da će to biti onaj u stupcu j za kojeg je $\phi(j)$ najveći. Time smo pronašli jedan član koji je sigurno u globalnom poravnanju.

4. S obzirom na svojstva globalnog poravnanja, intuitivno je jasno da polja koja se nalaze gore-desno i dolje-lijevo od pronađenog polja sigurno nisu u točnom globalnom poravnanju, stoga ta polja možemo izbaciti iz daljnjeg razmatranja. Ako rekurzivno primjenimo algoritam na parove podnizova $A_{0..p}$ i $B_{0,j}$ te $A_{p..N}$ i $B_{j..M}$, možemo ponavljajući postupak napraviti rekonstrukciju cijelog niza.

Ono što možda nije jasno jest vremenska složenost gore opisanog algoritma. Pokazuje se da pri odbacivanju polja odbacimo u pravlu polovicu trenutno razmatranih polja. Prema tome, broj operacija koje napravimo možemo napisati u obliku geometrijskog niza:

$$NM + NM/2 + NM/4 + ... < 2NM \in O(NM)$$

Time dolazimo do zaključka da je složenost i dalje ostala jednaka do razlike u konstantni, ali je znanto smanjena količina iskorištene memorije, stoga se ova ušteda isplatiti koristiti.

- TODO: slike svih koraka

3.5. Substitucija substitucijske matrice

- TODO: neko bolje ime za ovo potpoglavlje

U ovom ćemo radu umjesto supstitucijske matrice za usporedbu dva elementa koristiti fizičku udaljenost između dva atoma. Stoga će nizovi A i B biti zapravo nizovi koordinata iz kojih ćemo moći za svaki par elemenata izračunati koordinate.

Neka nam d označava udaljenost između dva atoma, a d_0 neka nam je konstanta (recimo $d_0 = 10$). Tada ćemo im udaljenost ocijeniti s $30 - e^{d/d_0}$.

4. Algoritam simuliranog kaljenja

Algoritam simuliranog kaljenja generička je metoda bazirana na vjerojatnosti koja traži ekstrem neke funkcije u velikom prostoru traženja. Zbog toga što nam ne garantira da će pronaći najbolje rješenje, obično se koristi kada nam je želimo dobiti rješenje koje je prihvatljivo u relativno kratkom vremenu.

Inspiracija za algoritam dolazi iz metalurgije. Metali se prvo grubo obrađuju na visokoj temperaturi, a, kako se hlade, tako obrada postaje sve finija i finija. Drugim riječima, *željezo se kuje dok je vruće*. Taj proces pokušavamo simulirati u ovom algoritmu.

4.1. Algoritam

Uvedimo nekoliko definicija:

- S - trenutno stanje.
- S' - stanje koje je susjedno trenutnom.
- $E(s)$ - funkcija koja računa energiju nekog stanja. Želimo pronaći stanje s najmanjom (ili najvećom) energijom.
- S_n - najbolje pronađeno stanje, tj. stanje u kojem je energija najmanja.

– e - energija trenutnog stanja, tj. $E(S)$.

– t - vrijeme proteklo od početka pokusa.

– T - trenutna temperatura. Definiramo je kao $T_0 T_1^t$, gdje je T_0 početna temperatura, a T_1 faktor koji govori koliko se temperatura brzo smanjuje.

– $P(e, e', T)$ - vjerojatnost da prijeđemo u stanje koje ima energiju e' iz stanja koje ima energiju e u trenutku T.

Algoritam se sastoji od ponavljanja slijedećeg niza operacija dok nismo dobili stanje koje ima traženu energiju ili dok vrijeme nije isteklo.

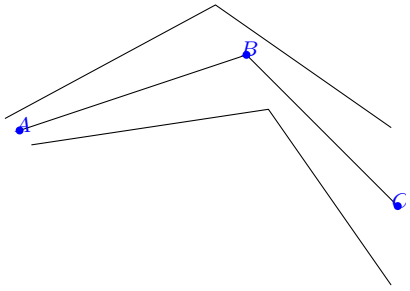
1. Odaberi stanje S' koje je susjedno stanju S.
2. Izračnaj energiju novog stanja $e' = E(S')$.
3. Provjeri ima li novo stanje veću energiju od najboljeg do sada pronađenog stanja. Ako ima, onda je $S_n = S'$.
4. Izračunaj vjerojatnost napredovanja u stanje S': $p = P(e, e', T)$.
5. Odaberi nasumičan broj iz intervala $[0, 1]$ i ako je manji od p, pomakni trenutno stanje u S'.
6. Povećaj vrijeme t.

4.1.1. Odabir susjeda

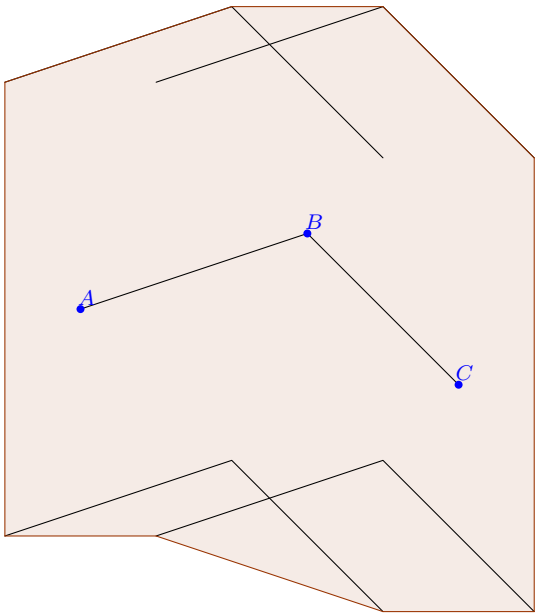
U našem slučaju, odabir susjeda radimo tako uzmemo koordinate atoma te im translatiramo sve tri koordinati, a cijeli niz potom rotiramo oko sve tri osi. Time smo osigurali da možemo konstruirati bilo koji drugi izomorfan niz u prostoru.

Da bismo bili sigurni da ne radimo prevelike skokove, svaka translacija ili rotacija događa se u određenom intervalu, to jest, nećemo raditi translacije koje za više od x niti rotacije za više od α stupnjeva.

- slika koja prikazuje pomak po kutevima



- slika koja prikazuje pomak po koordinatama



4.1.2. Računanje energije

Za računanje energije, koristimo prethodno opisani Smith-Waterman algoritam. Jedan ulazni niz je onaj koji želimo dobiti transformacijama, dok je drugi onaj za kojeg računamo energiju. Budući da nam rekonstrukcija nije potrebna jer nas zanima samo energija, nju niti ne računamo. Time ćemo dobiti ubrzanje od nekoliko puta.

4.1.3. Računanje vjerojatnosti prihvaćanja

Rješavanje ovog problema srž je algoritma simuliranog kaljenja. Jasno je da želimo otići u stanje koje ima bolju energiju, pa će nam ova funkcija kada je nova energija veća od trenutne, uvijek vratiti 1, što znači da ćemo sigurno otići u to stanje. Ono što razlikuje ovaj algoritam od klasičnog algoritma *penjanja na brdo* (engl. *hill-climbing*) jest ponašanje u slučaju da je nova energija manja od trenutne.

Simulirano kaljenje, kao što je već rečeno, ponekad ode i u lošija stanja. Hoće li se to dogoditi, ovisi o implementaciji funkcije P. Kako nam inspiracija dolazi iz obrade metala, vjerojatnost da prihvatimo loše stanje veća je na početku nego pri kraju. Također, ako je rješenje puno lošije, ne isplati nam ga se prihvatiti kao trenutno.

Razmotrimo način na koji možemo modelirati vrijeme. Pretpostavimo da nam q označava trenutnu vrijednost funkcije P. Također, uvedimo ograničenje da q mora biti iz intervala $[0, 1]$. Želimo, dakle, da P pada kako vrijeme odmiče. Najjednostavnije što možemo napraviti jest da od q oduzmemo t. Međutim, kako P mora biti u granicama $[0, 1]$ to nije dobro rješenje. Alternativno, ako oduzmemo t/t_max, dobili smo rješenje koje je potencijalno u intervalu $[-1, 1]$ i linearno pada ako mijenjamo samo vrijeme. Pokazalo se, međutim, da u

"prirodi" nije baš tako. Vjerojatnost, naime, pada eksponencijalno.

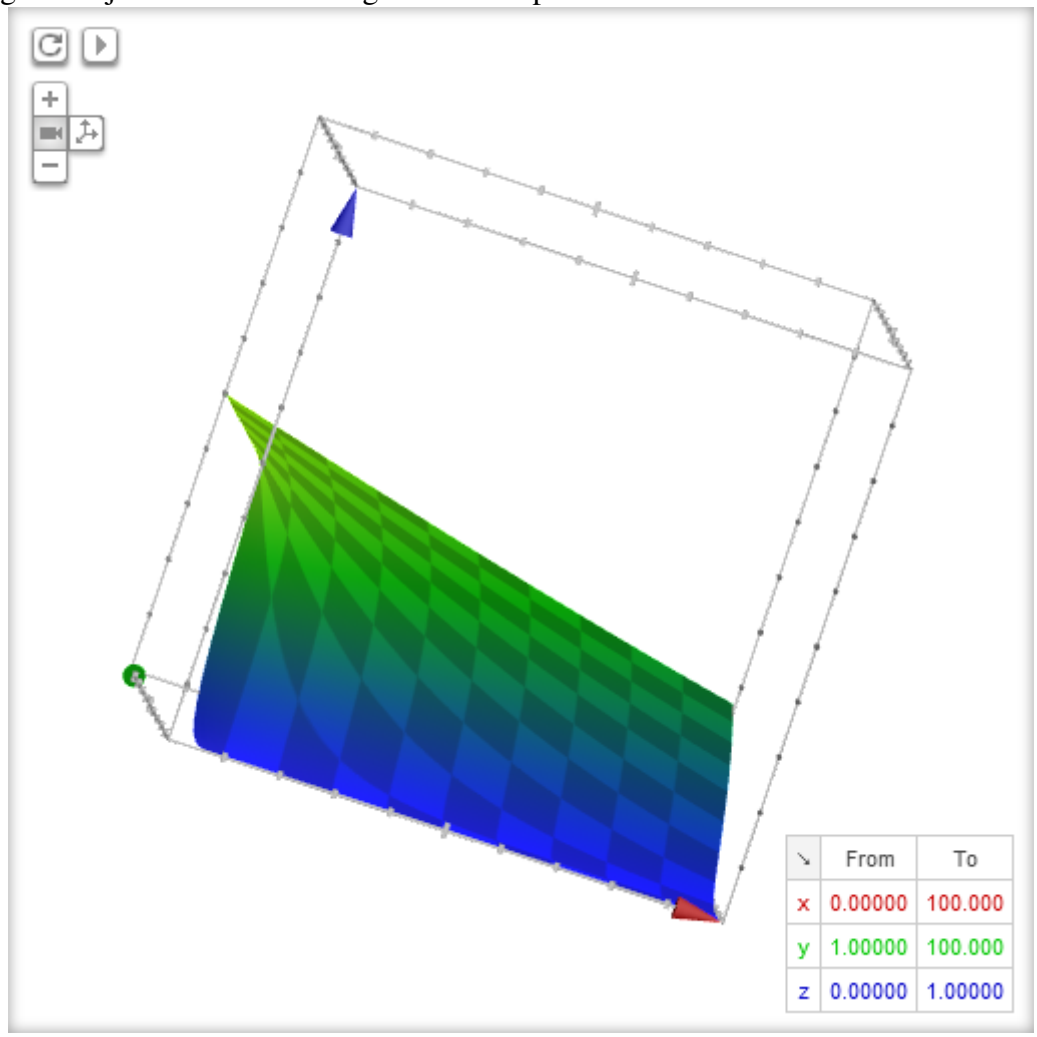
Slično razmišljanje vrijedi i za modeliranje razlike u energiji. Što je razlika veća, to nam vjerojatnost prijelaza treba biti manja.

Obično se uzima slijedeća funkcija:

$$P(e, e', T) = \begin{cases} 1 & \text{ako je } e' > e \\ \frac{1}{1+\exp\frac{e'-e}{T}} & \text{inače} \end{cases}$$

Primjetite da se, umjesto vremena, u njoj govori o temperaturi, koja eksponencijalno pada kako vrijeme odmiče.

U nastavku je dan graf funkcije $P(\Delta e, T)$, gdje je $\Delta e = e' - e$. Os x predstavlja Δe, os y je T (bitno je ponoviti da vremenom ova vrijednost pada) te os z prikazuje vrijednost P. Budući da 3D graf prikazujemo na 2D površini, pobojali smo ga u gradijent s obzirom na njegovu vrijednost kako bismo ga lakše interpretirali.

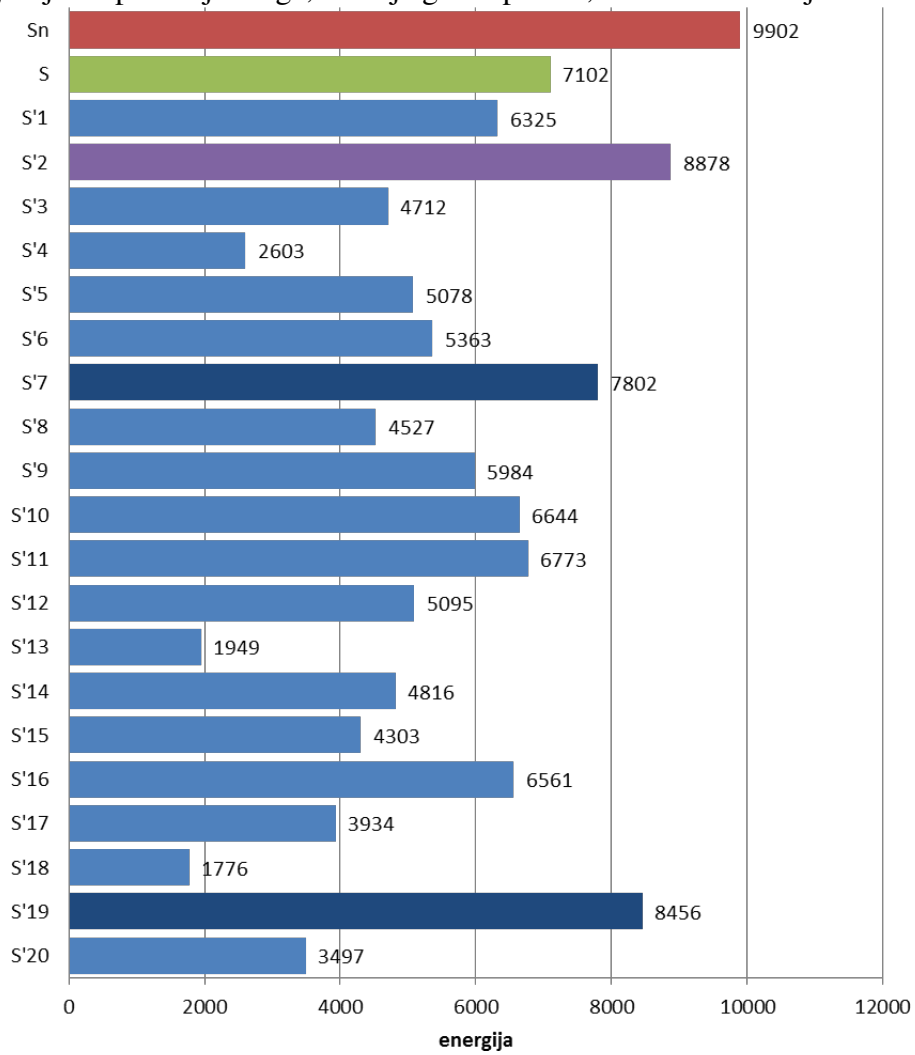


4.2. Modifikacije

Ovaj algoritam smo nešto prilagodili našoj upotrebi, kako bismo dobili prihvatljivo rješenje što je moguće prije. Budući da je cijeli algoritam baziran na heuristici, a izmjene koje smo napravili ne utječu na točnost, smijemo ih koristiti jer jedina stvar na što utječu su brzina algoritma.

4.2.1. Više susjeda

Klasičan algoritam, kako je to gore objašnjeno, koristi samo jedan susjed u svakom koraku. Mi ćemo u našem rješenju koristiti više (k) njih. Od tako dobivenih, uzimamo onaj koji je najbolji i uspoređujemo ga, kako je gore opisano, s trenutnim stanjem.



S obzirom da nam je svaki korak ocjenjivanja relativno skup, time se možemo osigurati rijetko odlazimo u slabija stanja. Međutim, takav pohlepan način odabira može nas dovesti u stanje lokalnog minimuma, iz kojeg se nećemo izvući. Da bismo pokušali otkloniti taj problem, u 10% generirat ćemo samo jednog susjeda u kojeg ćemo se, ako zadovljava uvjete, proširiti.

Također, pokazat će se u nastavku, ovime možemo nešto uštedjeti i na korištenju memorije i time povećati efikasnost koju dobijemo kada koristimo grafički procesor za računanje traženih podataka.

4.2.2. Ponovo pokretanje

Jedna od metoda da se izvučemo iz potencijalnog lokalnog ekstrema jest da ponovo pokrenemo algoritam iz neke prethodne točke. Često se to implementira na način da se promijeni vrijeme (smanji se, tj. postavi se u prošlost). Time se poveća vjerojatnost da izađemo iz nekog lokalnog ekstrema koristeći neka lošija stanja.

U slučaju da se najbolje rješenje nije promjenilo u zadnjih 15 koraka, vrijeme ćemo resetirati da bismo se pokušali pomaknuti iz mogućeg lokalnog ekstrema.

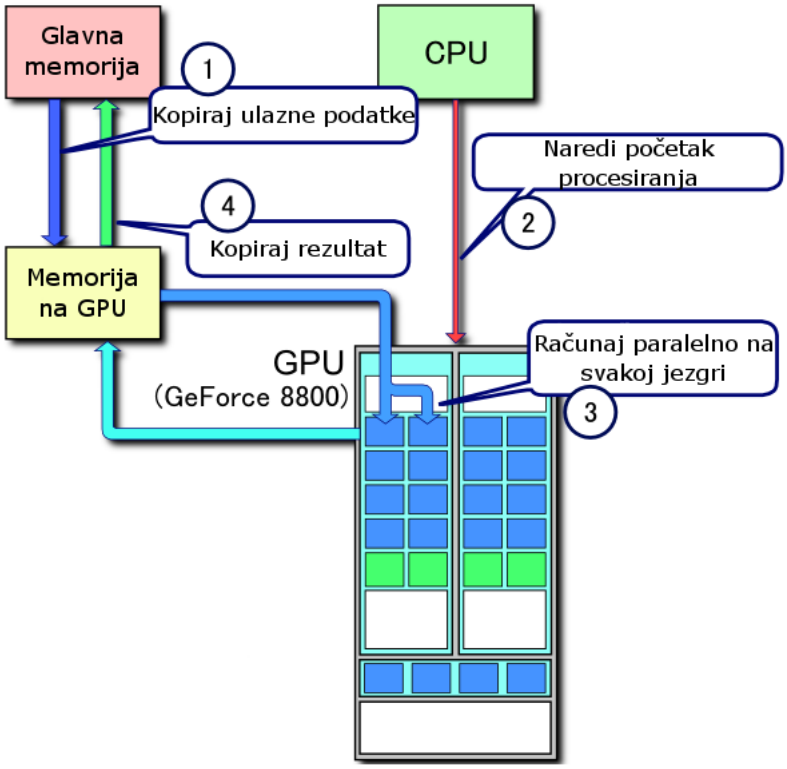
4.3. Parametari

S obzirom da naglasak ovog rada nije bio isključivo na ovom algoritmu, a i na vremenske rokove postavljene na pisanje rada, parametri koji su se koristili u implementaciji nisu previše istraženi.

5. CUDA tehnologija

Compute Unified Device Architecture (engl.; računski objedinjena arhitektura uređaja) ili skraćeno CUDA, tehnologija je koja nam omogućuje da upregnemo moć grafičkih čipova na problemima za koje se tipično koristio CPU (odnosno procesor). Kopanija Nvidija razvila ju je za svoje grafičke kartice.

Prva verzija CUDA-inog SDK-a (engl. ***s**oftware **d**evelopment **k**it* - oprema za razvoj programske podrške) izdana je 15. veljače 2007. Ona je omogućila korištenje resursa grafičkih kartica jedino u programskom jeziku C. Od tada do danas izdane su još dvije velike iteracije i nekoliko manjih te danas možemo koristiti i podskup jezika C++. Osim službenog SDK-a, za čitav niz programskih jezika, poput Fortrana, Jave, Haskella, Perla, Pythona, Matlaba, napisane su neslužbene poveznice.



Pri radu na CUDA-i, moramo razumjeti slijedeće pojmove:

– Domaćin (engl. *host*) - računalo u kojem se nalazi grafička kartica. Kada kažemo, npr. memorija domaćina mislimo na RAM.

– Uređaj (engl. *device*) - grafička kartica.

Prednosti koje ova tehnologija (odnosno grafičke kartice koje su sposobne koristiti ju) ima nad ostalim GPGPU tehnologijama su slijedeće:

– Nudi pristup bilo kojem dijelu memorije.

– Dretve mogu koristiti brzu zajedničku dijelnu memoriju.

– Podrška za račun s cijelobrojnim tipovima podataka, uključujući i operacije s bitovima.

Nažalost, ništa u životu nije savršeno, pa tako nije niti CUDA. Treba imati na umu da i ona ima i neka ograničenja:

– Učestalo kopiranje memorije između uređaja i domaćina ima loš utjecaj na brzinu izvođenja programa.

– U praksi se pokazalo da trebamo koristiti barem 32 dretve paralelno da bismo dobili brži program nego onaj na CPU-u. Ukupan broj dretvi koje koristimo trebao bi se brojati u tisućama.

– Sklopovlje je dostupno isključivo od jednog proizvođača - Nvidije.

– Zbog nezrelosti prevoditelja i optimizatora valjani C/C++ kôd ne radi nužno i na CUDA-i.

– Računanje s brojevima s posmačnim zarezom nije do kraja implementirano.

6. Implementacija

6.1. Implementacija simuliranog kaljenja

Simulirano kaljenje temelji se na postupku koji je identičan za sve tipove podataka. To nas navodi da implementaciju riješimo koristeći oblikovni obrazac okvirna metoda. Ideja tog oblikovnog obrasca jest da imamo jednu metodu (funkciju) koja izvodi neki postupak pozivajući generičke korake. Svaki korak može biti drugačije implementiran, ovisno o postupku koji želimo.

Izdvojimo apstraktne korake iz algoritma:

1. Odabir susjeda.
2. Računanje energije stanja.
3. Računanje vjerojatnosti prihvaćanja.

Kako točno rješavamo svaki od tih koraka, napisalo smo u prethodnom poglavlju.

Implementacijski gledano, sva tri postupka su funkcije. Prvi je funkcija koja prima jedan parametar, trenutno stanje te vraća transformirano stanje koje dobijemo primjenjujući translaciju i/ili rotaciju. Drugi prima takvo transformirano stanje i računa njegovu energiju. To radi tako da primjeni prethodno opisani Smith-Waterman algoritam nad transforimarnim i ciljnem nizom, ali ne radimo rekonstrukciju. Treći prima energiju početnog i krajenjeg niza te trenutnu temperaturu te iz toga računa vjerojansto prihvaćanja novog niza.

Zahvaljujući genericima, tj. predlošcima, u C++ je lagano ostaviriti ortogonalnost. Njihovom upotrebom, omogućavamo da kôd algoritma napišemo samo jednom i da ga potom možemo koristiti na više mjesta, za više tipova podataka, što upravo i želimo.

Takva implementacija omogućuje nam da ovaj dio programskog rješenja testiramo i na nekom jednostavnijem problemu, što uvelike olakšava razvoj programske potpore jer ne moramo čekati za završetak cijelog sustava da bismo vidjeli radi li. Osim toga, prilikom bubolova (engl. *debugging*) nam je lakše tražiti pogreške ako primjer možemo vizualizirati. Nažalost, kako ovo nije problem u kojem je vizualizaciju lagano napraviti, autor je koristio neke jednostavnije probleme da bi ispravio pogreške u ovom dijelu koda. Primjeri tih prob-

lema su traženje težišta poligona, Prilagodba algoritma za tu svrhu svela se na reimplantaciju funkcija za računanje energije i odabir susjeda.

6.2. Implementacija Smith-Watermanovog algoritma

Do sada smo razmatrali samo situaciju kada računamo jedno po jedno polje matrice R. Pretpostavka je bila da su sva polja u prošlim retcima i trenutnom retku do trenutnog stupca izračunata i da ih možemo koristiti u daljnjem računu.

Međutim, kada koristimo masivno paralalene procesore, želimo odjednonom računati više od jednog polja jer inače ne uprežemo svu snagu koju nam oni nude.

Ideja na kojoj nam se temelji rješenje jest da primjetimo koja su polja nezavisna, tj. koja možemo koristiti u računu ako smo do sada već izračunali neki dio rješenja. Definirajmo novu matricu W u kojoj nam $W_{i,j}$ označava broj polja matrice R koje moramo izračunati prije nego izračunamo polje $R_{i,j}$. Na početku će vrijediti slijedeće:

$$W_{i,j} = \begin{cases} 0 & \text{ako je } i = 0 \text{ i } j = 0 \\ 3 & \text{ako je } i > 0 \text{ i } j > 0 \\ 1 & \text{inače} \end{cases}$$

Jasno je da možemo izračunati vrijednosti samo onih polja za koje je $W_{i,j} = 0$. Na početku je takvo samo jedno polje: $W_{0,0}$. Međitim, kada njega izračunamo, smanjit ćemo vrijednosti tri polja u matrici W: $W_{1,0}$, $W_{0,1}$ i $W_{1,1}$. To će uzrokovati da $W_{1,0}$ i $W_{0,1}$ postanu jednaki 0, što znači da sada smijemo i njih izračanati. Kada to učinimo, i osvježimo vrijednosti u matrici W, vidjet ćemo da smo dobili tri nove nule, na pozicijama $W_{2,0}$, $W_{0,2}$ i $W_{1,1}$. Pravilnost se već lagano počinje nazirati. Naime, sve elemente koji se nalaze na istoj sporednoj dijagonali moći ćemo izračunati u istom koraku. Stoga ćemo paralelizaciju temeljiti upravo na tome da u pojedinom koraku računamo sve elemente na sporednoj dijagonali.

Vremenska složenost takvog bi algoritma u optimalnom slučaju bila bi $O(M + N)$. Nažalost, realan je slučaj daleko od optimalnog. Problem nam stvara to što nemamo dovoljno procesora da bismo mogli pokrenuti računanja nad svim elmentima dijagonale, ako je ona prevelika. Na sreću, CUDA nam ne stvara prevelike probleme ako koristimo stvorimo više blokova (do najviše 65536) nego što grafička kartica fizički može izračunati. S obzirom da dužina nizova vjerojatno neće biti puno veća od 15000 elemenata, takvo ograničenje je sasvim prihvatljivo.

Isto tako, ako u memoriju grafičke kartice ne stanu oba niza (što će se dogoditi, nasreću, samo za iznimno duge nizove koje nećemo promatrati), morat ćemo prilagoditi algoritam. Jedna efikasna ideja za rješavanje tog problema jest da podijelimo nizove na manje dijelove koje možemo u potpunosti riješiti te da kombiniramo tako dobivena rješenja. Međutim, to izlazi izvan teme ovog rada, pa nećemo ovdje detaljno objasniti kako se to radi.

Također, moramo biti svjesni da će konstanta koja je sakrivena u O notaciji biti puno veća nego kod obične implementicje na procesoru, budući da moramo koristiti barijere za sinkronizaciju. Pozivanje funkcija također troši vrijeme, a i pristup globalnoj memoriji uređaja jest skuplji nego pristup radnoj memoriji računala.

Što se memorijske složenosti tiče, i nju možemo ovdje ostvariti kao $O(M + N)$. Dovoljno nam je da, umjesto posljednja dva reda, pamtimo posljednje tri sporedne dijagonale. Rekonstrukcija se rješava Hirschbergovim algoritmom, koji će sada

7. Rezultati

- cca 1-2 starnice

8. Zaključak

Zaključak. - cca 1-2 stranice

- CUDA kao tehnologija jos je u zacecima, ali napreduje enormno brzo
- mnogo stavri je neistarzeno
- spomeni thrust i povuci paralelu s STL-om

LITERATURA

Christian D. Needleman, Saul B.; Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins, 1970.

Michael S. Smith, Temple F.; Waterman. Identification of common molecular subsequences, 1981.

SWIG - Poravnanje struktura korištenjem iterativne primjene Smith-Waterman algoritma

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: Ključne riječi, odvojene zarezima.

Title

Abstract

Abstract.

Keywords: Keywords.