SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2126

Implementacija Smith Waterman algoritma koristeći grafičke kartice s CUDA arhitekturom

Matija Korpar

Zagreb, lipanj 2011.

Umjesto ove stranice umetnite izvornik Vašeg rada.

Da biste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Hvala Ivani Kajić što mi je preporučila Mileta za mentora.
Hvala Miletu na znanju, usmjeravanju, velikom trudu oko prakse i ćevapima.

SADRŽAJ

Popis slika vi

Popis tablica vii

1. Uvod 1

2. *Smith-Watermanov algoritam* 3

2.1. *Needleman-Wunschov algoritam* . . . 3

2.2. Procjepi . . . 4

2.3. Supstitucijske matrice . . . 5

2.4. Formalna definicija *Needleman-Wunschova algoritma* . . . 5

2.5. Formalna definicija *Smith-Watermanova algoritma* . . . 7

3. Primjena *Hirschbergova algoritma* na rekonstrukciju rješenja *Smith-Watermanova algoritma* 9

3.1. Svođenje lokalnog poravnanja na globalno poravnanje . . . 9

3.2. Algoritam . . . 10

4. *CUDA* 13

4.1. Arhitektura . . . 13

4.2. Memorija . . . 14

5. Implementacija *Smith-Watermanova* algoritma na tehnologiji *CUDA* 15

5.1. Ocjenjivanje sličnosti . . . 15

5.1.1. Raspodjela posla . . . 15

5.1.2. Pristup memoriji . . . 17

5.2. Rekonstrukcija rješenja . . . 19

6. Primjena genetskog algoritma na traženje optimalnih parametara algoritma 22
6.1. Primjena na traženje parametara algoritma 22
6.2. Utjecaj dobivenih parametara na ubrzanje 23

7. Rezultati 25
7.1. Ispitni skupovi 25
7.2. Usporedba brzina izvođenja i potrošnji memorije 26
7.3. Usporedba s postojećim radovima 29

8. Zaključak 30

Literatura 31

Popis slika

2.1. Primjer rješenja *Needleman-Wunschova algoritma*. 4
2.2. Matrica rješavanja *Needleman-Wunschova algoritma*. 7
2.3. Matrica rješavanja *Smith-Watermanova algoritma*. 8

3.1. Svođenje lokalnog poravnanja na globalno poravnanje. 10
3.2. Prvi korak *Hirschbergova algoritma*. 12
3.3. Drugi korak *Hirschbergova algoritma*. 12

5.1. Postupak rješavanja vanjskih dijagonala *CUDA* implemenetacije algoritma. 17
5.2. Odnos ćelija i elemenata koje obrađuju i prebacivanje ćelija. 18
5.3. Duga i kratka faza izvođenja algoritma. 19
5.4. Hibridna rekonstrukcija rješenja. 20

Popis tablica

6.1. Usporedba brzina izvođenja algoritma za ocjenjivanje sličnosti s neoptimiranim parametrima i verzije s genetskim algoritmom optimiranim parametrima. . . . 24

7.1. Ispitni skup proteina. . . . 25
7.2. Ispitni skup *DNK-a*. . . . 26
7.3. Usporedba brzina izvođenja klasične neparalelizirane verzije, neparalelizirane verzije s uporabom *Hirschbergova algoritma* i paraleliziranog algoritma. . . . 27
7.4. Usporedba memorijske potrošnje klasične neparalelizirane verzije, neparalelizirane verzije s uporabom *Hirschbergova algoritma* i paraleliziranog algoritma. . . . 27
7.5. Usporedba potrošnja memorije i vremena izvođenja neparaleliziranih verzija s vremenom izvođenja i potrošnjom memorije paralelizirane verzije. . . . 28
7.6. Usporedba brzine izvođenja algoritma ocjenjivanja s pronađenim najbržim radom. . . . 29
7.7. Usporedba brzine izvođenja algoritma ocjenjivanja s *algoritmom BLAST*. 29

1. Uvod

Bioinformatika se kao znanstvena disciplina razvila s potrebom biologa za prikupljanjem, pohranjivanjem te obradom vrlo velikog broja informacija. Zbog količine podataka kojima se raspolaže (red veličine 10^{12} stanica u tijelu, tri milijarde nukleotida u genu), jednostavni algoritmi iziskuju previše resursa i vremena kod obrade informacija [8]. Kako se količina prikupljenih informacija eksponencijalno povećava, brzina, zahtjevi i točnost budućih algoritama odigrat će važnu ulogu u razvoju same biologije. Omogućiti obradu informacija uz potrošnju realnih resursa i vremena jedna je od zadaća bioinformatike.

Algoritmi ocjenjivanja sličnosti, tj. poravnanja sljedova igraju jednu od ključnih uloga u bioinformatici. Zadaća im je poravnanje dvaju ili više sljedova *DNK*, *RNK* i proteina te davanje ocjene sličnosti na temelju poravnanja, dok im je važnost u tome što njihovi rezultati mogu ukazivati na funkcionalne, strukturalne i evolucijske veze između danih sljedova [17]. Sljedove možemo shvatiti kao niz znakova abecede. Iako naizgled jednostavni, navedeni algoritmi zahtijevaju ogromne resurse i njihovo izvođenje traje veoma dugo.

Primjenjuju se:

– *u procjeni evolucijske srodnosti* - ukoliko primjena algoritama na dva slijeda *DNK-a* daje visoku ocjenu sličnosti, nositelji tih *DNK* se ocjenjuju evolucijski srodnima. Daljnje analize s drugim *DNK* sljedovima mogu pronaći i zajedničke pretke [17]

– *u predviđanju svojstava mutiranih ili novo otkrivenih DNK, RNK i proteina* - Pronalaženje sličnih sljedova s poznatim svojstvima pomoću algoritama pomaže pri procjeni svojstava novih sljedova [17].

– *u nebioinformatici* - poravnavanje sljedova znakova dugo je poznat problem u svijetu obrade teksta, zbog čega se algoritam primjenjuje u računarstvu.

Algoritmi poravnanja i ocjenjivana sljedova *DNK-a*, *RNK-a* i proteina čine jedan od središnjih stupova bioinformatike zbog svoje primjene na traženje gena. *Smith-*

Waterman [20], deterministički algoritam lokalnog poravnanja sljedova po mišljenju je mnogih najvažniji algoritam tog područja, no zbog svoje je velike složenosti $O(nm)$ često zamjenjivan puno bržim *BLAST algoritmom* [3] koji ne garantira točne rezultate [3]. Još jedan od problema algoritma velika je potrošnja memorije, konkretnije $O(nm)$. Primjenom *Hirschbergova algoritma* u kombinaciji s *Needleman-Wunschovim algoritmom* za rekonstrukciju rješenja *Smith-Watermanova algoritma*, moguće je, nauštrb brzine, smanjiti memorijsku potrošnju na $O(\min(m, n))$. *CUDA (Compute Unified Device Architecture)*, tehnologija paralelizacije poslova na grafičkim karticama, omogućila je ponovni povrat *Smith-Watermanu*. Središnja tema rada bit će izvršavanje ocjenjivanja *Smith-Watermanova algoritma* pomoću tehnologije *CUDA* te memorijski efikasna rekonstrukcija poravnanja.

2. *Smith-Watermanov algoritam*

U ovom poglavlju daje se kratki opis *Smith-Watermanova algoritma*, objašnjava se problem odabira njegovih parametara te se daje formalna matematička definicija algoritma. Problem odabira svodi se na problem ocjenjivanja, tj. kažnjavanja procjepa i na odabir matrica ocjenjivanja. Budući da je *Smith-Watermanov algoritma* nastao iz *Needleman-Wunschova algoritma* [18], prvo će biti dana definicija *Needleman-Wunsch algoritma*, a nakon toga će biti definirane razlike između njega i *Smith-Watermanova algoritma*.

2.1. *Needleman-Wunschov algoritam*

Sama je teorija poravnanja dvaju sljedova, temeljena na biološkim dokazima izvedenih eksperimentima, kompleksna. Pokazalo se ipak da se navedena teorija može kvalitetno aproksimirati veoma jednostavnim modelom. *Needleman-Wunschov algoritam* traži globalno poravnanje, tj. cilj mu je optimalno poravnanje. Optimalno poravnanje ono je poravnanje koje algoritam ocijeni najvećom ocjenom. Algoritam su objavili 1970. godine Saul Needleman i Christian Wunsch.

Prilikom poravnanja dvaju sljedova cilj je da se što više elemenata prvog slijeda poklopi sa što većim brojem elemenata drugog slijeda [20]. Da bismo to postigli, potrebno je umetati praznine u sljedove, tj. stvarati procjepe. Procjepi naravno nisu pogodni i njihovo se umetanje kažnjava. Kao primjer pretpostavimo da se svako umetanje procjepa kažnjava u konačnoj ocjeni s -1. Ostalo je još odrediti ocjene podudaranja i nepodudaranja dvaju elemenata. Kao primjer uzmimo da će se podudaranje dvaju elemenata ocijeniti s 1, dok će se nepodudaranje ocjenjivati ocjenom 0.

Do rješenja vidljivog na slici došli smo ispitivanjem svih mogućnosti umetanja procjepa. Navedeno ispitivanje svih mogućih umetanja procjepa središnji je problem algoritma. U praksi je problem kažnjavanja procjepa, kao i problem ocjenjivanja sličnosti dvaju elementa nešto kompleksniji. Ocjenjivanje sličnosti najčešće se radi *supstitucijskim matricama*. Takva implementacija ocjenjivanja sličnosti praktična je jer se

A	C	T	T	C	C	-	-	A	G	A
A	G	T	T	C	C	G	G	A	G	G
1	0	1	1	1	1	-1	-1	1	1	0

$\underbrace{\qquad\qquad}_{\sum=5}$

Slika 2.1: Rješenje *Needleman-Wunschova algoritma* na sljedovima s_0=ACTTCCAGA i s_1=AGTTCCGGAGG.

većina drugih modela ocjenjivanja može svesti na nju.

2.2. Procjepi

Problem određivanja cijene procjepa svodi se na pronalaženje dovoljno jednostavnog matematičkog modela koji bi opisivao eksperimentalno dobivene podatke, zbog velike složenosti eksperimentlano dobivene funkcije. U praksi postoje tri načina određivanja cijene, tj. kazne koju izaziva neki procjep:

– *linearna ocjena procjepa* - ako je k broj bodova koji dodajemo za otvaranje procjepa veličine 1, tada je ocjena procjepa veličine n upravo nk

– *konstantna ocjena procjepa* - bez obzira na veličinu, uvijek dodajemo isti broj bodova za bilo koji procjep

– *Afina ocjena procjepa* - uvode se dva parametra: d kao ocjena otvaranja novog procjepa i e kao ocjena svakog dodatnog proširenja procjepa nakon što je jednom otvoren. Ocjena procjepa duljine n po ovom je modelu $d + (n - 1)e$.

Empirijski dokazana formula za određivanje cijene procjepa jest parabola, stoga je lako uočiti da najbolje rezultate daje Afina ocjena, koja se iz istog razloga u praksi najčešće i koristi. Korištenjem Afine funkcije račun se u određenoj mjeri komplicira. Linearno ocjenjivanje procjepa najčešće se koristi na vrlo velikim sljedovima gdje do izražaja ne dolazi ocjenjivanje procjepa. Konstantno ocjenjivanje smatra se zastarjelom fazom razvoja algoritma, s obzirom na to da uvodi komplikacije u račun i daje rezultate netolerantno različite od idealnog modela. Međutim, mora se uzeti u obzir da je konstantno ocjenjivanje procjepa dovelo do Afine funkcije. Uzmemo li da je parametar kod Afine funkcije $e = 0$, tada dobivamo upravo konstantno ocjenjivanje [21]. U daljnjim poglavljima bit će objašnjena verzija *Needleman-Wunschova algoritma* koja koristi Afinu ocjenu, a takav je način ocjenjivanja također korišten u implementaciji.

2.3. Supstitucijske matrice

U prethodnom primjeru pokazan je veoma jednostavan model ocjenjivanja sličnosti, konstanta za podudaranje elemenata i analogno drukčija konstanta za njihovo nepodudaranje. Praksa je pokazala da je takvo ocjenjivanje pogodno za uspoređivanje sljedova *DNK-a* ili pak *RNK-a*, dok se zbog većeg broja elemenata i njihove različitosti takav model ne može adekvatno primjeniti na uspoređivanje proteina. Rješenje za uspoređivanje proteina prilično je intuitivna ideja matrice koja za svaka dva elementa određuje njihovu ocjenu sličnosti. Takve matrice nazivaju se *supstitucijske matrice* [9]. Eksperimentalno su definirane često korištene supstitucijske matrice:

- *BLOSUM45*
- *BLOSUM50*
- *BLOSUM62*

Za potrebe ovog rada pri usporedbi proteina koristit će se matrica *BLOSUM62* [1].

2.4. Formalna definicija *Needleman-Wunschova algoritma*

Needleman-Wunsch je deterministički algoritam i rješava se dinamičkim programiranjem. Rješava se matrično, a definicija vrijednosti elementa na nekoj poziciji određena je rekurzivnom formulom [20]. Budući da se za ocjenjivanje procjepa koristi Afino ocjenjivanje, nije dovoljno za svaki element pamtiti samo jednu vrijednost, već tri. Stoga možemo reći da postoje tri odvojene matrice.

Definirajmo pojmove:

s_1 - prvi slijed

s_2 - drugi slijed

m - duljina s_1

n - duljina s_2

d - ocjena novog procjepa kod Afina ocjenjivanja

e - ocjena produženog procjepa kod Afina ocjenjivanja

$p(x, y)$ - funkcija koja vraća ocjenu sličnosti elemenata x i y, $x \in s_1, y \in s_2$

Matrica koja se rješava veličine je $(m+1)(n+1)$ i svakom su elementu pridodane matrice H, E i F.

$$H_{i,j} = \begin{cases} 0 & i = 0 \land j = 0 \\ -(d + (i-1)e) & j = 0 \\ -(d + (j-1)e) & i = 0 \\ max \begin{cases} E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + p(s_{1,i-1}, s_{2,j-1}) \end{cases} & \text{inače} \end{cases} \tag{2.1}$$

$$E_{i,j} = \begin{cases} -\infty & i = 0 \\ max \begin{cases} E_{i,j-1} - e \\ H_{i,j-1} - d \end{cases} & \text{inače} \end{cases} \tag{2.2}$$

$$F_{i,j} = \begin{cases} -\infty & j = 0 \\ max \begin{cases} F_{i-1,j} - e \\ H_{i-1,j} - d \end{cases} & \text{inače} \end{cases} \tag{2.3}$$

Rekonstrukcija rješenja dobiva se pronalaženjem puta od ćelije rješenja matrice prema gornjem lijevom kutu matrice. Put se definira kao pomaci u svakoj ćeliji, dok se pomak u svakoj ćeliji definira s obizrom na to koja je od matrica H, E ili F generirala rješenje te ćelije. Očito je da za rekonstrukciju rješenja moramo uvesti još jednu matricu jednakih dimenzija kao i matrice H, E i F, nazovimo je M. Matrica M će pamtiti pomake ćelija.

$$M_{i,j} = \begin{cases} \text{zaustavljanje} & H_{i,j} = 0 \\ \text{pomak lijevo} & H_{i,j} = E_{i,j} \lor i = 0 \\ \text{pomak gore} & H_{i,j} = F_{i,j} \lor j = 0 \\ \text{pomak gore lijevo} & \text{inače} \end{cases} \tag{2.4}$$

Iz formula 2.1, 2.2, 2.3 te 2.4 uviđamo da put u rekonstrukciji rješenja *Needleman-Wunschova algoritma* završava u gornjem lijevom kutu matrice, čime je osigurano da se iskoriste svi elementi obaju sljedova, tj. da je poravnanje globalno.

		A	C	T	A
	0	-2	-4	-6	-8
A	-2	3	1	-1	-3
T	-4	1	2	4	2
A	-6	-1	0	2	7
T	-8	-3	-2	3	5
G	-10	-5	-4	1	2

Slika 2.2: Matrica rješavanja *Needleman-Wunschova algoritma*. Parametri d i e postavljeni su na 2. Funckija $p(x, y)$ vraća 3 ukoliko su x i y jednaki, -1 inače. Sivim ćelijama označen je put.

2.5. Formalna definicija *Smith-Watermanova algoritma*

Temeljna je razlika *Smith-Watermanova algoritma* i *Needleman-Wunschova algoritma* u tome da *Smith-Watermanov algoritam* traži najbolje lokalno poravnanje dvaju sljedova, tj. rezultat poravananja ne mora se nužno sastojati od svih elemenata obaju sljedova, dok *Needleman-Wunschov algoritam* traži najbolje globalno poravnanje dvaju sljedova, tj. rezultat se nužno sastoji od svih elemenata obaju sljedova. Razlika navedenih algoritama u formalnoj definiciji očituje se u definicijima matrica H i M. *Smith-Watermanov algoritam* objavili su 1981. godine Temple Smith i Michael Waterman.

$$H_{i,j} = \begin{cases} 0 & i = 0 \vee j = 0 \\ max \begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + p(s_{1,i-1}, s_{2,j-1}) \end{cases} & \text{inače} \end{cases} \tag{2.5}$$

$$M_{i,j} = \begin{cases} \text{zaustavljanje} & H_{i,j} = 0 \vee i = 0 \vee j = 0 \\ \text{pomak lijevo} & H_{i,j} = E_{i,j} \\ \text{pomak gore} & H_{i,j} = F_{i,j} \\ \text{pomak gore lijevo} & \text{inače} \end{cases} \tag{2.6}$$

Za rješenje s vrijedi $s = \max\ (H)$, $s \in H$.

		A	C	T	A
	0	0	0	0	0
A	0	3	1	0	3
T	0	1	2	4	2
A	0	3	1	2	7
T	0	1	2	4	5
G	0	0	0	2	3

Slika 2.3: Matrica rješavanja *Smith-Watermanova algoritma*. Parametri d i e postavljeni su na 2. Funckija $p(x, y)$ vraća 3 ukoliko su x i y jednaki, -1 inače. Sivim ćelijama označen je put.

3. Primjena *Hirschbergova algoritma* na rekonstrukciju rješenja *Smith-Watermanova algoritma*

U poglavlju će se objasniti osnovni princip rada *Hirschbergova algoritma*[12] te problem njegova prilagođavanja *Smith-Watermanovu algoritmu*.

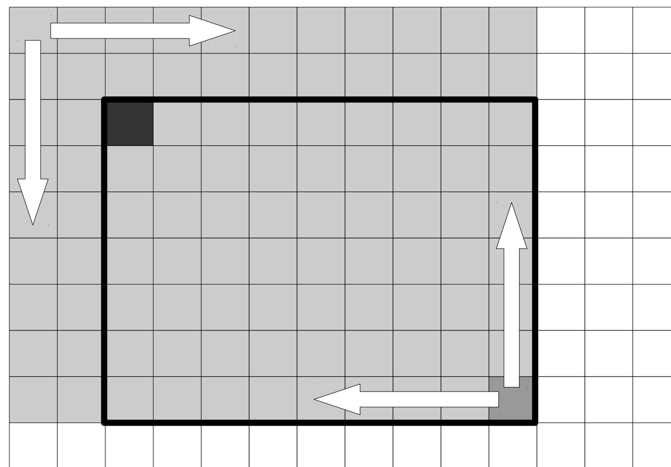
Hirschbergov algoritam, koji je dobio ime po svojem autoru *Danu Hirschbergu*, verzija je *Needleman-Wunsch algoritma*. Za razliku od memorijske složenosti $O(nm)$ *Needleman-Wunsch algoritma Hirschbergov algoritam* ima memorijsku složenost $O(\min(m, n))$ no istovremeno i nešto veću vremensku složenost. *Hirschbergov algoritam* daje kao rezultat globalno poravnanje, dok *Smith-Watermanov algoritam* daje lokalno. U nastavku je opisan problem svođenja lokalnog poravnanja na globalno poravnanje.

3.1. Svođenje lokalnog poravnanja na globalno poravnanje

Prilikom svođenja lokalnog poravnanja na globalno poravnanje koristi se jednostavna činjenica da je svako lokalno poravnanje gledajući iz određene domene globalno. Problem se, koristeći tu činjenicu, svodi na pronalaženje granica rekonstrukcije *Smith-Watermanova algoritma*, tj. na problem koje podsljedove sljedova s ulaza *Smith-Watermanova algoritma* proslijediti na ulaz *Hirschbergova algoritma*.

Samim saznanjem da put rekonstrukcije *Smith-Watermanova algoritma* završava u rezultantnoj ćeliji (što je objašnjeno u poglavlju 2.5), sa sigurnošću se zna da je to kraj lokalnog poravnanja, koje će biti ujedno i kraj novog globalnog poravnanja, tj. redak i stupac rezultantne ćelije uzimaju se kao kraj novog globalnog poravnanja. Kako naći početak poravnanja složeniji je problem jer je gledajući samo formalnu definiciju je-

dino rješenje rekonstukcija puta, što je memorijski i vremenski skupo, a i sam postupak bi tada bio jednak klasičnom postupku rekonstrukcije. Valja uočiti da formule dane u poglavlju 2.5 osiguravaju da algoritam za neke ulazne sljedove daje jednak rezultat kao i za iste reverzne sljedove. Ta će se činjenica koristiti i u algoritmu u poglavlju 3.2. Koristeći tu činjenicu, početnu ćeliju novog globalnog poravnanja dobivamo pokretanjem *Smith-Watermanova algoritma* nad reverznim ulaznim sljedovima s početkom u retku i stupcu kraja novog globalnog poravnanja. Početna ćelija rezultantna je ćelija tako pokrenutog algoritma [12].



Slika 3.1: Svođenje lokalnog poravnanja na globalno poravnanje. Tamnosiva ćelija predstavlja rezultantnu ćeliju pri rješavanju *Smith-Watermanova algoritma*. Nakon toga pokreće se *Smith-Waterman algoritam* nad svijetlosivim ćelijama tako da je tamnosiva ćelija početak, a gornji lijevi kut matrice kraj. Takvo pokretanje daje drugu rezultantnu ćeliju na slici označenu crnom bojom. *Hirschbergov algoritam* obavlja se na crno uokvirenom dijelu.

Znajući početnu i završnju ćeliju novog globalnog poravnanja, skraćuju se ulazni sljedovi prema indeksima navedenih ćelija. Nad navedenim podsljedovima pokreće se *Hirschbergov algoritam*.

3.2. Algoritam

Glavna prednost *Hirschbergova algoritma* njegova je linearna memorijska složenost $O(\min(m, n))$ [12]. Kako bi se postigla takva složenost, algoritam koristi strategiju podijeli pa vladaj. *Dan Hirschberg*, kreator algoritma, pokazao je da se podjelom *Needleman-Wunschova algoritma* u dva dijela, rješavanjem, pronalaskom odgovaraju-

ćih ćelija najveće sume, pronalaze komponente puta rekonstrukcije. Važno je napomenuti da *Needleman-Wunschov algoritam* koristi samo zadnja dva retka rješevanja koja su potrebna za određivanje elemenata matrice. Takva *zaboravna* verzija *Needleman-Wunschova algoritma* odgovara pri implementaciji *Hirschbergova algoritma* jer je bitan samo zadnji redak, što je objašnjeno u narednim odjeljcima.

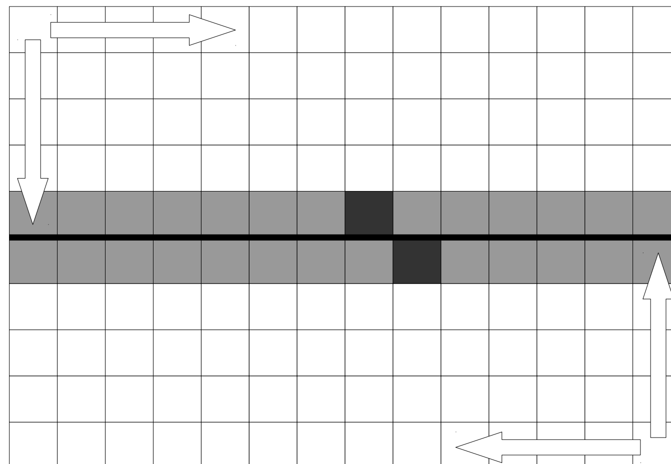
Prvi od koraka algoritma jest podijeliti matricu na dva dijela. Kako je memorijska složenost algoritma $O(\min(m, n))$, uviđa se da je memorijski najefikasnije matricu dijeliti na dva približno jednaka dijela te se ta podjela najčešće radi po stupcima. Nakon te podjele pokrećemo *Needleman-Wunschov algoritam* nad sljedovima koji odgovaraju gornjoj polovici matrice te pokrećemo drugi *Needleman-Wunschov algoritam* nad invertiranim sljedovima koji odgovaraju donjoj polovici matrice. Tim postupkom dobivaju se dva retka matrice. Zbrajajući njihove odgovarajuće ćelije, dolazimo do ćelija koje imaju najveću sumu. Označimo zadnji redak H i F matrice gornje podmatrice s X_H i X_F te analogno zadnji redak H i F matrice donje podmatrice s Y_H i Y_F. Duljinu retka označimo s n. Uzimajući u obzir afino ocjenjivanje procjepa te formule i simbole definirane u 2.3, ćelija s najvećom sumom c definirana je formulom:

$$c = arg_{max}(\max(X_{Hi} + Y_{Hi}, X_{Fi} + Y_{Fi} + d - e)), i \in 0..n \tag{3.1}$$

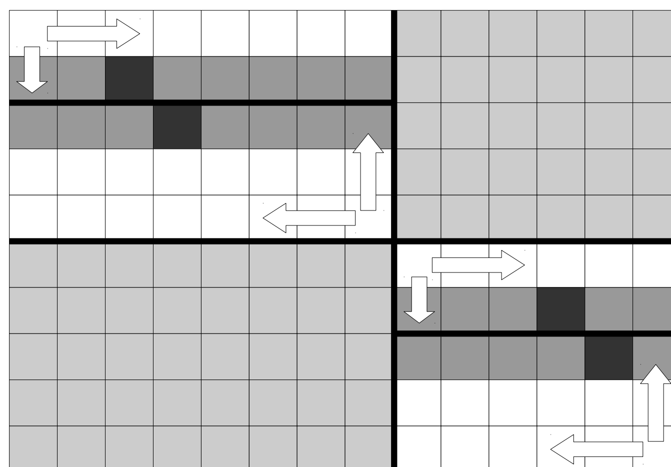
Za svaki par ćelija moramo provjeriti jesu li donja i gornja podmatrica u tom paru završile procjepom. Ako su obje ćelije završile procjepom, rezultat se ispravlja jer se kaznilo otvaranje procjepa s obje strane, a potrebno ga je kazniti samo s jedne. Time je pronađena jedna komponenta puta rekonstrukcije. Demonstracija navednog postupka prikazana je na slici 3.2.

Postupak nastavljamo rekurzivnim pozivom algoritma na podmatricama. Ako matricu podijelimo na četiri dijela s obzirom na redak i stupac ćelija s najvećom odgovarajućom sumom, dobivamo četiri manje podmatrice. Zbog definicije algoritma važno je primijetiti da se donja lijeva podmatrica te gornja desna podmatrica mogu odbaciti, tj. nisu potrebne pri postupku rekonstrukcije. Rekurzivno se ponavlja postupak nad gornjom lijevom podmatricom te donjom desnom podmatricom. Takav postupak ponavlja se sve dok se ne dođe do svih komponenti puta. Drugi korak rekurzije navedenog postupka prikazan je na slici 3.3.

Valja uočiti da takav algoritam uz *zaboravnu* verziju *Needleman-Wunschova algoritma*, tj. takvu verziju u kojoj se ne pamte pomaci i ne pamti se ništa drugo osim zadnjeg i predzadnjeg retka koji su potrebni za rješavanje, nikada neće premašiti memorijsku složenost od $m + n + 2\min(m, n)$. Memorijski prostor od $m + n$ potreban je



Slika 3.2: Prvi korak *Hirschbergova algoritma.* Pokrene se *Needleman-Wunschov algoritam* nad polovicama matrice i time se izračunaju vrijednosti svijetlosivih ćelija. Zbrajanjem odgovarajućih ćelija tih dvaju redaka dolazimo do rezultata da su tamnosive ćelije upravo komponente rekonstrukcije puta.



Slika 3.3: Drugi korak *Hirschbergova algoritma.* Matrice dobivene u prvom koraku dijele se na pola i u svakoj se takvoj matrici traže odgovarajuće ćelije s najvećom sumom.

za pohranjivanje informacije o sljedovima. Analizirajmo prostor od $2\min(m, n)$. Prvi korak rekurzije za pamćenje dva potrebna retka troši upravo toliko memorije. Važno je primijetiti da se ta memorija nakon rješavanja može otpustiti prije idućih koraka rekurzije. Budući da se idući koraci sigurno obavljaju na podsljedovima manje duljine, možemo zaključiti da je najgora memorijska složenost $m+n+2\min(m, n)$, što upravo odgovara $O(mn)$.

4. *CUDA*

CUDA (Compute Unified Device Architecture) je opće primjenjiva arhitektura paralelizacije razvijena od strane *NVIDIA-e* [16]. Za razliku od klasičnih sustava *CUDA* iskorištava procesore grafičkih kartica. Specifičnost sustava procesora na grafičkim karticama u odnosu na obične procesore predstavlja činjenica da sustav procesora na grafičkoj kartici sadrži velik broj slabijih procesora koji imaju mogućnost paralelnog izvođenja.

NVIDIA je otvorila svijet paralelizacije širokoj javnosti, s obzirom na to da je cijena potrebne opreme realna.

4.1. Arhitektura

Grafička kartica koja podržava tehnologiju *CUDA* može izvršavati algoritme sa značajnim ubrzanjem u odnosu na vrijeme njihova izvođenja na običnom procesoru. Navedena arhitektura temelji se na velikom broju procesora koji istovremeno mogu izvršavati velik broj dretvi te učinkovito izvršavati aritmetičke operacije nad podacima spremljenim u jedinstveno organiziranoj memoriji [19].

U *CUDA-i* dretve su grupirane u blokove, dok su blokovi grupirani u mreže. Dretve unutar istog bloka mogu razmjenjivati informacije preko brze dijeljene memorije i mogu biti sinkronizirane. Dretve unutar različitih blokova dijele informacije preko spore globalne memorije i rijetko se sinkroniziraju. Sama *CUDA* nadogradnja je na programski jezik *C*, što omogućava lako programiranje jezgri [19]. Jezgre su funkcije programa koje se izvršavaju na grafičkoj kartici, tj. standardni program napisan na tehnologiji *CUDA* kombinira izvršavanje i na standardnom procesoru i na procesoru grafičke jedinice.

4.2. Memorija

Ključna je pri ubrzavanju s tehnologijom *CUDA* organizacija memorije. Arhitektura memorije nešto je složenija, a različite vrste memorija razlikuju se po veličini i brzini pristupa. Postoji više vrsta memorije, poredanih po brzini pristupa:

- *memorija registara* - interna memorija svake dretve, nevidljiva svim ostalim dretvama, dostupno 16k registara [19]
- *memorija konstanti* - memorija koja služi samo za čitanje, dostupna svim dretvama, do 64kB [19]
- *dijeljena memorija* - memorija vidljiva dretvama unutar jednog bloka, do 48kB [19]
- *globalna memorija* - memorija vidljiva svim dretvama, velika i najsporija [19]

Ključ je uspjeha pri ubrzanju u korištenju što više memorije s vrha popisa, a što manje memorije s dna popisa.

5. Implementacija *Smith-Watermanova* algoritma na tehnologiji *CUDA*

Smith-Waterman, kao i većina algoritama koji pripadaju vrsti dinamičkog programiranja, zbog svoje ovisnosti pri izvođenju, tj. zbog rekurzivne definicije, nije jednostavan za implementaciju na tehnologiji *CUDA*.

5.1. Ocjenjivanje sličnosti

5.1.1. Raspodjela posla

S obzirom na to da prilikom računanja svi elementi matrice nisu međusobno ovisni, možemo ih grupirati u grupe neovisnih elemenata, pri čemu su elementi unutar jedne grupe oni elementi koji se mogu izračunati u istom trenutku neovisno jedni o drugima. Kod *Smith-Watermanova algoritma* važno je primijetiti da su elementi na sporednim dijagonalama neovisni, s obzirom na to da svaki element ovisi o donjem, gornjem te lijevo-gore dijagonalnom elementu. Na temelju te će se grupacije vršiti i izvođenje. Izvođenja jezgri rješavati će sporedne dijagonale, čime se osigurava točnost rješenja, tj. ni u jednom trenutku neće se istovremeno računati 2 ili više ovisnih elemenata [15].

Kada bi se dijagonale izvodile ćeliju po ćeliju, postojao bi velik broj upisa i čitanja iz globalne memorije, što je s strane ubrzanja nedopustivo. Važno je primijetiti da ako rješavamo prema sporednim dijagonalama, nema razloga da se dretve kreću po matrici po sporednim dijagonalama. Mnogo je zgodnije da se dretve rješavaju horizontalno. Primjenom rješavanja po sporednim dijagonalama n-ta dretva će biti na koordinatama (i, j) dok će $(n+1)$-va dretva u tom trenutku biti na indeksu $(i+1, j-1)$, pri čemu je i indeks retka, dok je j indeks stupca. Uočljivo je da je ovisnost izvršavanja zadovoljena.

Temeljem tog zaključka uvodimo pojmove:

- *vanjska dijagonala* - dijagonala gledana iz prespektive blokova
- *unutarnja dijagonala* - dijagonala gledana iz perspektive dretvi.

Eksperimentalno je uspostavljeno da dolazi do značajnih ubrzanja ako jedna dretva ne računa samo jedan redak u matrici. Nazovimo broj redaka koje obrađuje jedna dretva α.

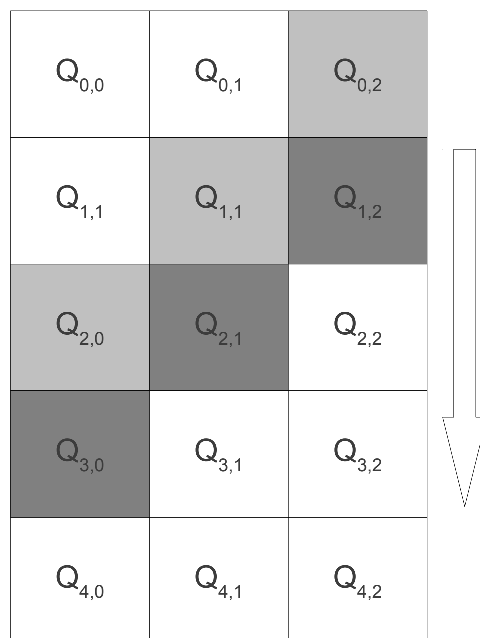
Pretpostavimo da se broj dretvi i blokova određuje samo na temelju dužeg ulaznog niza, što će biti objašnjeno u sljedećem poglavlju rada. Označimo broj blokova s B i broj dretvi s T. Potrebno je odrediti dimenzije bloka. Označimo duljinu ulaznih nizova s m i n, tako da je $n < m$. Označimo širinu bloka s C i visinu s R. Iz objašnjenja internih i eksternih dijagonala vidljivo je da je $R = \alpha T$. Izračun širine komplicira se ako n nije djeljivo s B, no kao primjer pretpostavimo da n dijeli B bez ostataka. Tada je širina bloka $C = n/B$. Takvom konfiguracijom dobivamo da je broj vanjskih dijagonala $D = B + \lceil m/(\alpha T) \rceil - 1$. Sada možemo podijeliti matricu na ćelije RC. Za svaku dijagonalu D_k tada možemo reći da blokovi u tom trenutku obrađuju RC ćelije s indeksom (i, j) takvim da zadovaljava $i + j = k$, tj. ako je ćelija s indeksom (i, j) jednaka $Q_{i,j}$, tada vrijedi $D_k = Q_{i,j} | i + j = k$. Uviđamo da će kod prvih B dijagonala i zadnjih B dijagonala postojati blokovi koji ne rade ništa zbog strukture sporednih dijagonala. Kada algoritam nije u takvom stanju kažemo da je u punoj paralelizaciji [13].

S jedne strane, problem kod internih dijagonala pojavljuje se kod rješavanja prvih elemenata u bloku. Neka dretva u bloku, označimo je s t_n, mora čekati da prva dretva t_0 obradi n elemenata, tj. da dretva t_{n-1} obradi element o kojem je prvo njezino izvođenje ovisno. S druge strane, imamo problem da će na kraju matrice, ili pak na dijelovima prijašnje dijagonale, ostajati neobrađene ćelije iz istog razloga. Bez većih posljedica možemo preraspodijeliti dretve koje čekaju na neobrađene elemente u ćelijama prijašnje dijagonale [15]. Ako je ćelija na lijevom rubu matrice dretve, prebacujemo na ćeliju B redova više na desnom rubu matrice. Navedeni postupak nazovimo gornjim prebacivanjem. Iz takvog prebacivanja dretvi proizlazi uvjet $2BT \leq n$. Zbog gornjeg prebacivanja moramo u svakom koraku provjeriti moramo li se za određenu dretvu prebaciti natrag u originalni red. Taj postupak nazovimo vraćanjem prebacivanja.

Iako iskorištava sve dretve, takav pristup u slučaju potpune paralelizacije naravno izaziva problem sinkronizacije. Uočimo da dretve prvog bloka dijagonale mogu ovisiti o dretvama koje rješavaju ćelije prijašnjih dijagonala [14]. Važno je napomenuti da se taj problem događa samo u prvih T koraka svake dijagonale. Kako bismo osigurali sigurno izvođenje, možemo odvojiti algoritam u dvije faze:

– *kratka faza* - prvih T koraka

– *duga faza* - od T-tog do koraka C.



Slika 5.1: Postupak rješavanja vanjskih dijagonala *CUDA* implemenetacije algoritma. Svijetlosivom bojom označena je dijagonala D_2, dok je tamnosivom označena dijagonala D_3. Svi blokovi navedenih dijagonala iskorišteni su, dok će u npr. dijagonali D_0 i D_1 biti neiskorištenih blokova.

Jedna je od optimizacija da u dugoj fazi sigurno ne može doći do gornjeg prebacivanja i vraćanja prebacivanja, što zbog velikog broja obrada prilikom izvršavanja rezultira znatnim ubrzanjem.

5.1.2. Pristup memoriji

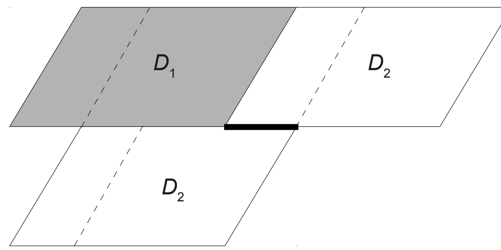
Sljedovi koji se analiziraju zbog svoje se veličine spremaju u globalnu memoriju. Prilikom pokretanja svake jezgre dretve učitavaju odgovarajući podniz slijeda koji odgovara stupcima u dijeljenoj memoriji. Važno je primijetiti da svaka dretva mora najviše dvaput učitati podatke sljedova koji odgovaraju retcima, nakon čega se te informacije spremaju u registarsku memoriju.

Označimo osnovnu memorijsku jedinicu pri rješavanju K. Za riješiti α elemenata, jer svaka dretva rješava upravo toliko, potreban je jedan K. Kako su za svaki od α elemenata potrebna tri elementa i kako je svaki od α redaka poredan jedan ispod drugog, svaki K zapravo obuhvaća podatke iz $2(\alpha+1)$ elemenata. Za svaki od tih elemenata ne moramo pamtiti vrijednosti svih matrica. Nazovimo element na indeksu $(0,0)$ u K-u

Slika 5.2: Odnos ćelija i elemenata koje obrađuju i prebacivanje ćelija. Debelim crnim obrubom uokvirene su ćelije vanjskih dijagonala. Naizmjenično sivom i bijelom bojom prikazane su dijagonale i elementi koje obrađuju. Tamnosivom bojom označeni su slučajevi prebacivanja ćelija.

K_{diag}, element na indeksu $(0, 1)$ K_{up}, elemente u lijevom stupcu, tj. one s indeksima $(1..\alpha, 0)$, K_{left} i ostale elemnte u desnom stupcu, tj. one s indeksima $(1..\alpha, 1)$, K_{cur}. Kod K nije važna veličina, s obzirom na to da je puno manja od registarske memorije u koju se sprema, već je važno ono što se prenosi. Početne vrijednosti K_{left}-ova i K_{diag}-a i vrijednost K_{up}-a u svakom koraku moraju se učitati iz neke od zajedničkih memorija. Početne vrijednosti K_{left}-ova prenose se iz K_{cur}-ova, dok K_{diag} se prenosi iz K_{up}-a prethodne ćelije. Naravno, postupak se odvija kroz globalnu memoriju jer se podaci prenose tijekom izvođenja različitih jezgri. Nazovimo prostor za prijenos tih vrijednosti vertikalnom sabirnicom. Za svaki K_{left} moramo prenijeti po dvije 4-bajtna vrijednosti, vrijednost H i E matrica elemenata, dok je za svaki K_{diag} potrebna samo jedna 4-bajtne vrijednost, vrijednost H matrice elementa. Ukupna veličina vertikalne sabirnice tada je $(8\alpha + 4)BT$ bajtova [13].

K_{up} se osvježava svakim korakom. Prva dretva u bloku, tj. najviša dretva, svoje podatke čita iz globalne memorije, dok ostale dretve čitaju iz dijeljene. Analogno tome zadnja dretva, tj. najniža dretva, piše u globalnu memoriju, dok ostale pišu u dijeljenu memoriju. Svaka dretva mora u svakom koraku prenijeti dvije 4-bitne vri-



Slika 5.3: Duga i kratka faza izvođenja algoritma. Debelom crnom linijom označeno je područje na kojem može doći do konflikta. Budući da su blokovi potpuno neovisni, ne možemo tvrditi da će bijela ćelija riješiti crne elemente prije nego što ih bijeli ćelija u donjoj liniji dohvati. Isprekidanom crtom odvojene su duge i kratke faze izvođenja. Takvim odvajanjem faza riješen je navedeni problem.

jednosti, vrijednosti H i F matrica. Nazovimo globalni prostor za prijenos navedenih podataka horizontalna sabirnica. Dakle, prva dretva čita iz horizontalne sabirnice, dok zadnja dretva piše u horizontalnu sabirnicu, ostali prijenosi obavljaju se kroz dijeljenu memoriju. Veličina je horizontalne sabirnice $8n$ bajtova.

Ukupno memorijsko ograničenje zbog relativno malog broja blokova i dretvi svodi se na horizontalnu sabirnicu i veličinu samih lanaca. Kada se to zbroji, memorijsko ograničenje ispada $9n + m$. Ovakva složenost prihvaća usporedbu nizova do reda veličine 10^9, što je i više nego zadovoljavajuće jer je dužina DNK-a u kromosomu čovjeka reda veličine $3 \cdot 10^8$.

5.2. Rekonstrukcija rješenja

Kod rekonstrukcije rješenja koristi se *Hirschbergov algoritam*, detaljno opisan u poglavlju 3. Dio tog algoritma koji u implementaciji koristi jezgru *CUDA* upravo je dohvaćanje graničnih redaka, tj. zadnjeg reda gornje i donje polovice matrice rješavanja.

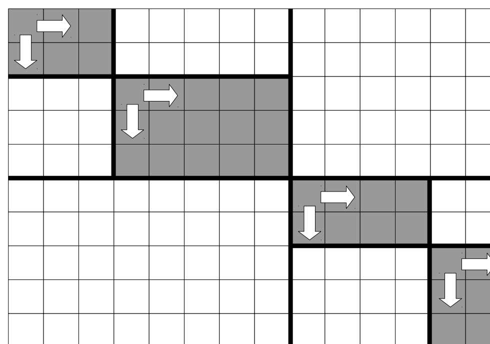
Zbog memorijskog pristupa praktičnije je, tj. bolje, da se algoritam uvijek razvija prema memorijski idućim lokacijama, tj. da se ne vraća. Stoga se donje polovice matrica rješavanja odvijaju nad reverznim sljedovima. Kako se ne bi trošilo vrijeme na invertiranje, u memoriji je inicijalno zapamćen svaki slijed i u reverznom obliku. Utrošak memorije povećava se za $m + n$, no to i dalje ne narušava memorijsku složenost, a značajno štedi vrijeme.

Sama jezgra dohvaćanja zadnjeg retka uvelike je slična jezgri rješavanja algoritma

objašnjenoj u poglavlju 5.1. Informacije koje su potrebne, tj. vrijednosti zadnjeg retka algoritma rješavanja, nalaze se u horizontalnoj sabirnici i glede tog nije potrebno modificirati algoritam. No ipak postoje tri temeljne razlike između te dvije jezgre:

- α je postavljena na 1. Kako su potrebne informacije zadnjeg retka, ne može se pustiti da zadnji *fantomski* retci, tj. oni retci koji se pojavljuju zbog nedjeljivosti m s α, upisuju podatke u horizontalnu sabirnicu. Važno je naglasiti da je postavljanje α na 1 mnogo vremenski jeftinije nego neprekidne provjere je li redak *fantomski*
- nema potrebe za spremanjem najvećih rezultata ćelija. Sve što je potrebno od navedene jezgre upravo je horizontalna sabirnica. Dijelovi koji traže i zapisuju rješenje ćelije izbacuju se iz jezgre i time se dobiva značajno vremensko poboljšanje jer zapis rezultata teče kroz globalnu memoriju
- početni uvjeti algoritma moraju odgovarati *Needleman-Wunschovu algoritmu*. Za razliku od jezgre ocjenjivanja, vertikalna i horizontalna sabirnica postavljaju se na početne uvjete *Needleman-Wunschova algoritma*. Praktično je što se horizontalna sabirnica postavlja incijalno jednom izvan same jezgre, no vertikalna sabirnica mora se postavljati dinamački unutar jezgre.

Uz navedene promjene jezgra rekonstrukcije puno je vremenski efikasnija od jezgre ocjenjivanja, što je veoma korisno zbog velikog broja pozivanja takvih jezgri što je posljedica rekurzivne definicije *Hirschbergova algoritma*.



Slika 5.4: Hibridna rekonstrukcija rješenja. Granice sivih blokova dobivene su pomoću *CUDA* jezgri. Nakon što se tim granicama dovoljno fino podijeli matrica rješavanja, dobivene sive podmatrice rješavaju se klasičnim *Needleman-Wunschovim algoritmom*.

Uvjet stajanja algoritma danog u poglavlju 3 nalaženje je svih komponenti puta. Vidljivo je da takav pristup prilikom rada s jezgrama *CUDA* nema smisla jer su jezgre

CUDA predviđene za rad s velikom količinom podataka i njihovo bi pokretanje nad malim količinima podataka bilo vremenski puno skuplje nego pokretanje iste količine podataka nad klasičnim algoritmom. Rješenje koje se nameće jest neka vrsta hibridnog rješavanja, prepusti se *CUDA* jezgrama da dovoljno fino podijele matricu rješavanja i nakon toga se nad dobivenim malim matricama pokreće klasičan *Needleman-Wunschov algoritam*. Nadovezivanjem rješenja takvih blokova dobiva se ukupno rješenje, tj. put.

Ovakav algoritam rekonstrukcije puta memorijski može podnijeti isto što i algoritam ocjenjivanja sličnosti opisan u poglavlju 5.1, što znači da je rekonstrukcija rješenja primjenjiva na gotovo sve praktične sljedove iz prirode, za razliku od klasične verzije gdje memorijski zahtjevi postaju nerealni već na redu veličine proteina.

6. Primjena genetskog algoritma na traženje optimalnih parametara algoritma

Genetski algoritam heuristička je metoda optimiranja koja oponaša prirodni evolucijski proces. Evolucija je proces pretraživanja prostora rješenja. Živa se bića tijekom evolucije prilagođavaju uvjetima u prirodi, tj. životnoj okolini. Analogija evolucije kao prirodnog procesa i genetskog algoritma kao metode optimiranja očituje se u procesu selekcije i genetskim operatorima. Mehanizam odabira nad nekom vrstom živih bića u evolucijskom procesu čine okolina i uvjeti u prirodi. U genetskim algoritmima ključ je selekcije funkcija cilja koja na odgovarajući način predstavlja problem koji se rješava. Slično kao što su okolina i uvjeti u prirodi ključ selekcije nad nekom vrstom živih bića, tako je i funkcija cilja ključ selekcije nad populacijom rješenja u genetskom algoritmu. Naime, u prirodi jedinka koja je najbolje prilagođena uvjetima i okolini u kojoj živi ima najveću vjerojatnost preživljavanja i parenja, a time i prenošenja svojega genetskog materijala na potomke. Selekcijom se odabiru dobre jedinke koje se prenose u sljedeću populaciju, a manipulacijom genetskog materijala stvaraju se nove jedinke. Takav proces selekcije, reprodukcije i manipulacije genetskim materijalom jedinki ponavlja se sve dok nije zadovoljen uvjet zaustavljanja evolucijskog procesa. Konačan rezultat populacija jedinki je. Najbolja jedinka u zadnjoj iteraciji predstavlja rješenje optimiranja [11].

6.1. Primjena na traženje parametara algoritma

Kako se implementacija algoritma sastoji od dva glavna dijela, ocjenjivanja sličnosti i rekonstrukcije rješenja, koji imaju veoma slične jezgre *CUDA*, genetski algoritam zbog praktičnosti je primijenjen samo na algoritam ocjenjivanja sličnosti. Kod rekonstrukcije rješenja ima mnogo dijelova algoritma koji se ne izvršavaju paraleleno i ti bi

dijelovi nepotrebno utjecali na rezultate mjerenja.

Kako postoji velik broj implementacija genetskih algoritama, potrebno je okvirno opisati korištenu implementaciju. Parametri *CUDA*, tj. parametri algoritma, koje je potrebno optimirati jesu broj dretvi T i broj blokova B. Brzina izvođenja t direktno je ovisna o B i T i predstavlja nam jedinu ocjenu kakvoće jedinke. Jedinka s najmanjom brzinom izvođenja bit će najbolja jedinka. Sama arhitektura tehnologije *CUDA* predstavlja ograničenja prilikom odabira B i T. Većina kartica podržava da je broj dretvi manji od 512, tj. $1 \leq T \leq 512$ [2] jer, naravno, uvijek mora postojati barem jedna dretva. B je za slučaj jednodimenzionalnih konfiguracija koje su korištene manji od 65536, tj. $1 \leq B \leq 65536$ [2] jer, kao i kod dretvi, uvijek mora postojati barem jedan blok. Dodatno ograničenje objašnjeno je u poglavlju 5.1.1, a ono govori da $2BT \leq n$, pri čemu je n duljina kraćeg slijeda.

Genetski algoritam odvija se u generacijama. Broj generacija G ručno je definiran, kao i veličina populacije P. Sljedeća generacija stvara se na temelju odabira najboljih D jedinki prethodne generacije i u novu se generaciju umeće C djece na mjesto C najgorih jedinki prethodne generacije. Djeca su genetski manipulirane jedinke prethodne generacije. Manipulacija se sastoji od mutacije i križanja. Mutacija se događa uz vjerojatnost od P_m. Mutirano dijete dijete je neovisno roditeljima, tj. ima slučajno generirane parametre. Križanje se događa uz vjerojatnost P_k. Za razliku od mutacije, križanje ovisi o dvama roditeljima. Roditelji se odabiru troturnirskom selekcijom, tj. iz populacije se izvlače tri slučajne jedinke od kojih se odabiru dvije najbolje. Parametri djeteta dobivenog križanjem predstvaljaju aritmetičku sredinu parametara roditelja.

Konfiguracija korištenog genetskog algoritma:

G - 200
P - 50
D - 30
C - 20
P_k - 0.95
P_m - 0.05

6.2. Utjecaj dobivenih parametara na ubrzanje

Ocjenu ubrzanja ovom metodom treba shvatiti relativno. Referentna brzina koja se koristi pri izračunavanju ubrzanja rezultat je brzine algoritma s parametrima odabranima na temelju iskustva, tj. subjektivno. Zanimljivo je to što se pokazalo da dobiveni B i T kod *DNK-a* ne ovise o veličini ulaznih sljedova, dok je kod proteina zbog uvjeta

$2BT \leq n$ potrebno smanjiti B. Genetski algoritam pokrenut je na trima parovima ulaznih sljedova različitih redova veličine. U prosjeku dobiveno ubrzanje iznosi 20%, dok se u prosjeku izvođenja algoritma s optimiranim parametrima na većem ulaznom skupu ubrzanje svodi na 15-20%. Genetski algoritam nije pokrenut nad cijelim skupom testiranih ulaznih parova zbog njegova izrazito dugog izvršavanja.

Tablica 6.1: Usporedba brzina izvođenja algoritma za ocjenjivanje sličnosti s neoptimiranim parametrima t_1 i verzije s genetskim algoritmom optimiranim parametrima t_2.

Duljine sljedova	t_1	t_2	Ubrzanje
35213 x 36805	1.12s	0.91s	18%
162114 x 171823	3.47s	2.78s	19%
536165 x 542868	19.74s	15.28s	23%

7. Rezultati

Dobiveni rezultati, koji su prikazani u nastavku, omjeri su brzina izvođenja i memorijske potrošnje klasičnoga neparaleliziranog algoritma, neparaleliziranog algoritma s uporabom *Hirschbergovog algoritma* te paraleliziranog algoritma, čija je implementacija opisana u poglavlju 5. U nastavku su najprije definirani ispitni skupovi, posebno proteini i *DNK-a*, nad kojima se mjeri brzina i potrošnja memorije algoritama. Dan je prikaz omjera brzina i potrošnje memorije te je prikazano vrijeme izvođenja paraleliziranе verzije na većim sljedovima. Također je dana usporedba paraleliziranog algoritma s postojećim implementacijama i usporedba s *BLAST* algoritmom.

7.1. Ispitni skupovi

Ispitni skupovi podijeljeni su na proteine i *DNK* sljedove. Dužine proteinskih sljedova sežu od 100 pa do okvirno 37000. Izabrana su tri proteinska slijeda s najvećom dužinom radi demonstracije ubrzanja algoritma. Kod odabira *DNK-a*, uzeta je u obzir brzina izvođenja neparaleliziranog algoritma, tako da duljina *DNK* sljedova testiranih na navedenoj verziji ne prelazi $6 \cdot 10^5$ i dodatno je uzeto nekoliko duljih sljedova za prikaz brzine izvođenja isključivo na paraleliziranoj verziji algoritma.

Tablica 7.1: Ispitni skup proteina. [7]

Oznaka	Porijeklo	Duljina
D3DPG0	Čovjek	34942
A2ASS6	Miš	35213
Q3ASY8	Chlorobium chlorochromatii	36805

Tablica 7.2: Ispitni skup *DNK-a.* [6]

Oznaka	Porijeklo	Duljina
NC_000898	Ljudski retrovirus 6B	162114
NC_007605	Ljudski retrovirus 4	171823
NC_003064	Agrobacterium tumefaciens	542868
NC_000914	Rhizobium sp.	536165
CP000051	Chlamydia trachomatis	1044459
AE002160	Chlamydia muridarum	1072950
BA000035	Corynebacterium efficiens	3147090
BX927147	Corynebacterium glutamicum	3282708

7.2. Usporedba brzina izvođenja i potrošnji memorije

Rezultati mjerenja algoritama ovise ponajviše o samom sklopovlju na kojem se izvode. Kod neparaleliziranih verzija algoritma relevantno je sklopovlje procesor, dok je kod paralelizirane verzije relevantno sklopovlje grafička kartica. Dodatno brzina izvođenja paralelizirane verzije ovisi o odabiru parametara *CUDA-e*. Vrijeme je mjereno na operacijskom sustavu *Ubuntu 10.04* u njegovoj jezgri alatom *time* [4] dok je memorija mjerena alatom *top*[5]. Za paraleliziranu verziju odabrani su parametri dobiveni primjenom genetskog algoritma opisanom u poglavlju 6.

Sklopovlje korišteno pri mjerenju:

- *Intel Core™2 Duo CPU P6400 s 2.13GHz* x 2
- *nVidia Corporation GF110 [Geforce GTX 570]*
- *4GB* radne memorije

S jedne strane, izmjerene brzine kod neparaleliziranih verzija, prikazanih u tablici 7.3 kao t_1 i t_2, rastu eksponencijalno s porastom duljine analaziranih sljedova. Takvo je ponašanje i očekivano, s obzirom na složenost izvođenja $O(nm)$, pri čemu su m i n duljine ulaznih sljedova. S druge strane, brzine paralelizirane verzije, označene u istoj tablici kao t_3, nije moguće tako lako opisati. Brzina izvođenja na tehnologiji *CUDA* ovisi uvelike o broju čitanja i pisanja u globalnu memoriju. Ako se rezultat često mijenja, tj. ako se rezultat algoritma nalazi blizu donjeg desnog ruba matrice nm, takav će slučaj biti sporiji od slučaja u kojem je rješenje blizu gornjeg lijevog ruba matrice nm, uz pretpostavku da su m i n u oba slučaja vrlo slični. Valja primijetiti da

Tablica 7.3: Usporedba brzina izvođenja klasične neparalelizirane verzije t_1, neparalelizirane verzije s uporabom *Hirschbergova algoritma* t_2 i paraleliziranog algoritma t_3, čija je implementacija opisana u poglavlju 5. Znak "-" označava mjerenja koja su zahtijevala više memorije nego što je dostupno na testnom sklopovlju ili ona mjerenja čije izvršavanje ne završava u realnom vremenu.

Oznaka	Oznaka	Duljine	t_1	t_2	t_3
D3DPG0	A2ASS6	34942 x 35213	42.375s	92.235s	3.513s
D3DPG0	Q3ASY8	34942 x 36805	58.566s	110.149s	4.163s
A2ASS6	Q3ASY8	35213 x 36805	59.002s	113.162s	4.106s
NC_000898	NC_007605	162114 x 171823	-	660.835s	2.805s
NC_000914	NC_003064	536165 x 542868	-	9090.448s	21.92s
CP000051	AE002160	1072950 x 1072950	-	-	73.01s
BA000035	BX927147	3147090 x 3282708	-	-	630.958s

Tablica 7.4: Usporedba memorijske potrošnje klasične neparalelizirane verzije m_1, neparalelizirane verzije s uporabom *Hirschbergova algoritma* m_2 i paraleliziranog algoritma m_3, čija je implementacija opisana u poglavlju 5. Znak "-" označava mjerenja koja su zahtijevala više memorije nego što je dostupno na testnom sklopovlju ili ona mjerenja čije izvršavanje ne završava u realnom vremenu.

Oznaka	Oznaka	Duljine	m_1	m_2	m_3
D3DPG0	A2ASS6	34942 x 35213	1.1GB	205KB	359KB
D3DPG0	Q3ASY8	34942 x 36805	1.2GB	215KB	342KB
A2ASS6	Q3ASY8	35213 x 36805	1,2GB	217KB	411KB
NC_000898	NC_007605	162114 x 171823	-	1.5MB	2.7MB
NC_000914	NC_003064	536165 x 542868	-	12MB	25MB
CP000051	AE002160	1072950 x 1072950	-	-	42MB
BA000035	BX927147	3147090 x 3282708	-	-	67MB

prvi navedeni slučaj uzrokuje mnogo više pisanja po globalnoj memoriji.

Memorijska potrošnja kod klasične verzije algoritma, prikazana u tablici 7.4 kao m_1, visoka je zbog njegove memorijske složenosti $O(nm)$. Takva složenost uzrokuje nerealne resurse već pri manjim sljedovima *DNK-a*, pa je u tim slučajevima navedena verzija neprimjenjiva. Paralelizirani i neparalelizirani algoritam koji koriste *Hirsch-*

bergov algoritam za memorijsku optimizaciju rekonstrukcije rješenja i time postižu složenost $O(\min(m, n))$, imaju gotovo jednaku memorijsku potrošnju. Memorijske potrošnje navedenih algoritama, prikazanih u tablici 7.4 kao m_2 i m_3, veoma su niske što uzrokuje da s realnim memorijskim resursima navedeni algoritmi mogu rješavati i najveće sljedove *DNK-a*. Razlika između m_2 i m_3 leži u potrošnji memorije same *CUDA* arhitekture.

Tablica 7.5: Usporedba potrošnja memorije i vremena izvođenja neparaleliziranih verzija t_1, odnosno m_1, pri čemu je t_1 vrijeme bržeg, na testnom sklopovlju izvedivog, neparaleliziranog algoritma, s vremenom izvođenja i potrošnjom memorije paralelizirane verzije t_2, odnosno m_2.

Duljine	t_1	t_2	t_1/t_2	m_1	m_2	m_1/m_2
34942 x 35213	42.375s	3.513s	12.06	1.1GB	359KB	3212.9
34942 x 36805	58.566s	4.163s	14.07	1.2GB	342KB	3679.21
35213 x 36805	59.002s	4.106s	14.39	1.2GB	411KB	3061.53
162114 x 171823	660.835s	2.805s	235.59	1.5MB	2.7MB	0.55
536165 x 542868	9090.448s	21.92s	414.71	12MB	25MB	0.48

Iz tablice 7.5 vidljivo je da su ubrzanja značajna i da je memorijska potrošnja paralelizirane verzije algoritma veoma niska što ga čini upotrebljivim, za razliku od neparaleliziranih ili pak memorijski neoptimiranih verzija.

7.3. Usporedba s postojećim radovima

Važno je napomenuti da će se prilikom usporedbe koristiti samo dio algoritma za ocjenjivanje, jer nije pronađena paralelizirana implemenatacija rekonstrukcije rješenja. Od svih pronađenih algoritama ocjenjivanja najbrži je algoritam autora *Edansa Flaviusa de O. Sandesa* i *Albe Cristine M. A. de Meloa* [10]. U tablici 7.6 dana je usporedba brzina izvođenja. Algoritmi su testirani na različitim grafičkim karticama, zbog nemogućnosti nabavke jednakih. Algoritam s kojim se uspoređuje testiran je na grafičkoj kartici *GeForce GTX 280*. Budući da je ta kartica nešto slabija od navedenog testnog sklopovlja, rezultate u tablici 7.6 treba smatrati približno jednakim.

Tablica 7.6: Usporedba brzine izvođenja algoritma ocjenjivanja t_2 s pronađenim najbržim radom t_1.

Duljine	t_1	t_2	t_2/t_1
162114 x 171823	1.7s	2.0s	0.85
536165 x 542868	15.2s	12.3s	1.23
1072950 x 1072950	56.6s	44.3s	1.27
3147090 x 3282708	512s	390s	1.31

Tablica 7.7: Usporedba brzine izvođenja algoritma ocjenjivanja t_2 s *algoritmom BLAST* t_1. Budući da je *algoritam BLAST* heuristički, dana je i ocjena točnosti rezultata za svaki primjer.

Duljine	t_1	t_2	t_2/t_1	točan
162114 x 171823	0.67s	2.0s	2.98	DA
536165 x 542868	0.78s	12.3s	15.76	DA
1072950 x 1072950	2.89s	44.3s	15.32	NE
3147090 x 3282708	7.1s	390s	54.92	NE

8. Zaključak

U radu je opisan *Smith-Watermanov algoritam*, osnove tehnologije *CUDA* i dana je implementacija algoritma na *CUDA* tehnologiji. Razređeni su i komentirani problemi koji su se pojavili prilikom implementacije algoritma. Algoritam je ubrzan u eksponencijalnoj ovisnosti prema klasičnom algoritmu. Rezultati ubrzanja prikazani su u tablici 7.5, gdje je na primjerima pokazano ubrzanje od 10 do oko 400 puta. U implementaciji je korišten genetski algoritam objašnjen u poglavlju 6.1 pri odabiru parametara tehnologije *CUDA*. Navedenom implementacijom pronađeni su parametri koji su ubrzali algoritam daljnjih 15-20%. Također, uprabom *Hirschbergova algoritma* memorijska složenost algoritma spuštena je s $O(nm)$ na $O(\min(m, n))$. Memorijska složenost jezgre *CUDA* prikazane implementacije dovoljno je malena da algoritam radi na većini grafičkih kartica. Algoritam radi sa sljedovima do reda veličine 10^9 s realnom količinom memorije, što je i više nego zadovoljavajuće jer je duljina *DNK-a* u kromosomu čovjeka reda veličine $3 \cdot 10^8$. Dobivena ubrzanja omugućuju rad u stvarnom vremenu pri analizi proteina i svode analizu malih i srednjih *DNK* sljedova na realno vrijeme.

Literatura

[1] *BLOSUM62 Substitution Matrix*, 2006. URL http://www.uky.edu/Classes/BIO/520/BIO520WWW/blosum62.htm.

[2] *Cuda Reference Manual*, 2010. URL http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/CudaReferenceManual.pdf.

[3] *Bioinformatics explained: BLAST versus Smith-Waterman*, 2010. URL http://www.clcbio.com/index.php?id=1098.

[4] *Time manual*, 2011. URL http://ss64.com/bash/time.html.

[5] *Top manual*, 2011. URL http://unixhelp.ed.ac.uk/CGI/man-cgi?top.

[6] *Nucleotide database*, 2011. URL http://www.ncbi.nlm.nih.gov/.

[7] *Protein knowledgebase*, 2011. URL http://www.uniprot.org/.

[8] Nair Achuthsankar. *Computational Biology and Bioinformatics - A gentle Overview*, 2007. URL http://print.achuth.googlepages.com/BINFTutorialV5.0CSI07.pdf.

[9] M. O. Dayhoff, R. M. Schwartz, i B. C. Orcutt. *A model of evolutionary change in proteins*. 1978.

[10] Edans Flavius de O. Sandesa i Alba Cristina M. A. de Meloa. *CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences*, 2010.

[11] Marin Golub. *Genetski algoritam*, 2010. URL http://www.zemris.fer.hr/~golub/ga/ga.html.

[12] Dan S. Hirschberg. *A linear space algorithm for computing maximal common subsequences*. 1975.

[13] Lukasz Ligowski i Witold Rudnicki. *An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In IEEE International Workshop on High Performance Computational Biology*. 2009.

[14] Weiguo Liu, B. Schmidt, G. Voss, A. Schroder, i W. Muller-Wittig. *Bio-sequence database scanning on a gpu*. 2006.

[15] Yongchao Liu, Douglas Maskell, i Bertil Schmidt. *Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units*. 2009.

[16] Shane McGlaun. *NVIDIA Clears Water Muddied by Larrabee*. 2008. URL `http://www.dailytech.com/NVIDIA+Clears+Water+Muddied+by+Larrabee/article12585c.htm`.

[17] Mount. *Bioinformatics: Sequence and Genome Analysis*. 2004.

[18] Saul B. Needleman i Christian D. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. 1970.

[19] Jason Sanders i Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 2010.

[20] Temple F. Smith i Michael S. Waterman. *Identification of Common Molecular Subsequences*. 1981. URL `http://www.cray.com/downloads/SmithWaterman.pdf`.

[21] Taylor i Munro. *Multiple sequence threading: conditional gap placement*. 1997.

Implementacija Smith Waterman algoritma koristeći grafičke kartice s CUDA arhitekturom

Sažetak

Smith-Waterman, deterministički algoritam ocjenjivanja sličnosti i poravnavanja sljedova, jedan od najvažnijih algoritama bionformatike, zbog svoje velike složenosti $O(nm)$ često je zamjenjivan puno bržim heurističkim *algoritmom BLAST*. *CUDA (Compute Unified Device Architecture)*, tehnologija paralelizacije posla na grafičkim karticama, omogućila je ponovni povrat *Smith-Watermanu*. U ovom radu predstavljena je jedna od mogućih implementacija *Smith-Watermanova algoritma* na tehnologiji *CUDA*. Dobiveno ubrzanje raste eksponencijalno s porastom veličine sljedova, od 10 puta pri većim proteinima, do 400 puta na manjim sljedovima *DNK-a*. Problem odabira parametara *CUDA* tehnologije riješen je primjenom genetskog algoritma čiji su rezultati uzrokovali daljnje ubrzanje algoritma za 15-20%. Uporabom *Hirschbergova algoritma* memorijska složenost algoritma spuštena je s $O(nm)$ na $O(\min(m, n))$.

Ključne riječi: Smith-Waterman, CUDA, paralelizacija, bioinformatika, poravnavanje sljedova

Smith-Waterman algorithm implementation using graphic cards with CUDA architecture

Abstract

Smith-Waterman, deterministic algorithm for searching the local alignment of sequences, considered by many as one of the most important algorithm in bioinformatics, due to its high complexity $O(nm)$ is often replaced with the faster heuristic *BLAST algorithm*. *CUDA (Compute Unified Device Architecture)*, parallelization technology which works with graphic cards allows the return of *Smith-Waterman*. The resulting acceleration grows exponentially, from 10 times with longest proteins, up to 400 times with shorter *DNA* sequences. The problem of selecting the parameters for *CUDA* technology is solved using a genetic algorithm whose results caused further acceleration of the algorithm by 15-20%. Usage of *Hirschberg's algorithm* has reduced memory complexity from $O(nm)$ to $O(\min(m, n))$.

Keywords: Smith-Waterman, CUDA, parallelization, bioinformatics, sequence alignment