Sveučilište u Zagrebu,
Fakultet elektrotehnike i računarstva

Računanje lokalnog poravnanja koristeći Smith-Waterman algoritam implementiran na CUDA platformi

Bioinformatika - projekt

Članovi tima:
Dario Sitnik
Franjo Matković
Matej Crnac

17.01.2018.

Sadržaj

Sadržaj 2

Uvod 3

Opis algoritama 4

Implementacija 6

Rezultati 9

Zaključak 11

Literatura 12

1. Uvod

U ovom radu opisana je implementacija računanja lokalnog poravnanja koristeći Smith-Waterman algoritam po uzoru na (Korpar M.,Šikić M.) [1]. Traženje poravnanja između dva niza je čest problem u bioinformatici. Jedno od poznatijih rješenja tog problema je Needleman-Wunsch algoritam koji pronalazi globalno poravnanje. Ipak, za neke slučajeve globalno poravnanje i Needleman-Wunsch algoritam nisu dovoljni dobri te je potrebno tražiti lokalno poravnanje.

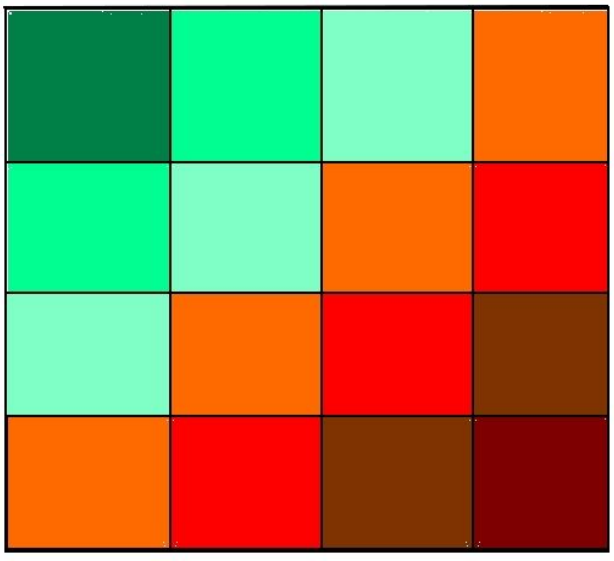
Rješenje tog problema je Smith-Waterman algoritam koji je, poput Needleman-Wunscha, jedan od najpoznatijih algoritama u bioinformatici. Njegov cilj je pronaći regije koje čije će poravnanje imati najveći rezultat. Zbog toga se radi maksimizacija sličnosti. Kako se prolazi kroz oba slijeda sličnost će rasti i padati. Ako sličnost padne ispod 0 može se govoriti da sličnost više ne postoji. Kad se iz krajnje točke bude vraćalo prema početnoj, takvi elementi neće doprinositi rezultatu nego će ga smanjivati. Zbog toga se elementi koji bi trebali biti negativni stavljaju na 0. Nakon što se izračuna cijela tablica sličnosti, iz najveće točke se kreće prema početku tablice prateći put kojim je došao do te točke. Tako se pronalazi poravnanje.

Često se u bioinformatici koriste jako veliki nizovi (10^6). Računanje lokalnog poravnanja na tako velikim nizovima rezultira velikim padom performansi i velikim memorijskim zauzećem. Zbog toga su s vremenom smišljeni i različiti optimizacijski postupci. Jedan primjer svođenja algoritma na linearnu memorijsku složenost je Hirschbergov algoritam [2]. Pomoću njega se matrica podijeli na 2 jednaka dijela po horizontalnoj osi. Zatim se rezultati oba dijela tablice računaju u isto vrijeme, s tim da se drugi dio računa od kraja prema početku. Još jedan zanimljiv pristup je računanje s ograničenim pojasom. To znači da će se sva računanja raditi unutar ograničenog pojasa koji ide u smjeru glavne dijagonale. Tako se zanemaruju elementi koji se nalaze u donjem lijevom rubu i gornjem desnom rubu (engl. *pruning*) što će znatno ubrzati rad algoritma. Zadnjih nekoliko godina sve više se koristi paralelizacija uz pomoć CUDA-e, jedan takav način bit će objašnjen u idućem poglavlju.

2. Opis algoritama

Implementacija opisana u ovom dokumentu temelji se na (Korpar M.,Šikić M.) [1] te će se u ovom poglavlju opisati algoritam predstavljen u navedenom radu. Kao što je već rečeno, u zadnjih nekoliko godine se sve više koristi paralelizacija koristeći CUDA-u. To je moguće zato što grafičke kartice imaju veliki broj jezgri koje mogu pokrenuti veliki broj dretvi, a CUDA je programski paket namijenjen za korištenje grafičkih kartica u računanju. Tako se i u navedenom radu iskorištava sposobnost CUDA-e za paralelizaciju.

U radu se računanje lokalnog poravnanja dijeli na tri faze: fazu računanja, fazu pronalaska i fazu rekonstrukcije. U fazi računanja računa se tablica sličnosti između nizova. Već ovdje se iskorištava mogućnost paralelizacije uvođenjem wavefront metode paralelizacije. To znači da se svi elementi na sporednoj dijagonali računaju u isto vrijeme. To je moguće napraviti zato što rezultat svakog elementa matrice ovisi samo o elementu lijevo, elementu iznad i elementu dijagonalno gore lijevo. Prikaz rada wavefront metode može se vidjeti na slici 1. Blokovi koji su prikazani istom bojom računaju se paralelno. Računanje idućeg seta blokova pokreće se tek kada prethodni set završi. Kreće se u smjeru sporedne dijagonale. Takav način paralelizacije znatno ubrzava računanje matrice. Dodatni korak optimizacije je i korištenje podrezivanja (engl. *pruning*). Rezultat faze računanja je izračunata tablica sličnosti krajnja točka poravnanja.



Slika 1. Prikaz rada wavefront metode

Faza pronalaska treba pronaći puteve u tablici sličnosti kojim se kretao rezultat. Prvo se tablica podijeli horizontalnom linijom po pola te se računaju krajnja točka i rezultat gornje polovice, krajnja točka i rezultat donje polovice i srednja krajnja točka i srednji rezultat. Zatim se između dobivenih vrijednosti rezultata izračuna maksimalan. Idući korak je modificiranim Smith-Waterman algoritam s obrnutim nizovima pronaći startnu točku tako da krene iz krajnje točke. Kao i u prvoj fazi i tu se koriste wavefront metoda te podrezivanje.

Na kraju je u fazi rekonstrukcije potrebno pronaći najbolji put kroz tablicu sličnosti te time dobiti poravnanje dva niza. Tijekom rekonstrukcije razmatraju se samo elementi između početne točke i krajnje točke. U ovoj fazi koristi se, također, wavefront metoda za prolazak kroz tablicu.

3. Implementacija

Implementacija u radu se razlikuje od prethodno objašnjenog algoritma. U računanju tablice sličnosti između dva genetska slijeda koriste se dvije razine paralelizacije. Program prvo primi dva genetska slijeda koja uspoređuje. Tada računa broj blokova koje će rješavati svaka jezgra grafičke kartice. Broj se dobije tako da se izračuna po svakom slijedu, a to je: (dužina_slijeda_1/32), te (dužina_slijeda_2/32), pomnožimo ta dva broja pa se dobije ukupan broj blokova koji se moraju riješiti.

Prije samoga računanje broja blokova genetski sljedovi se nadopunjuju tako da je dužina pojedinog sljeda višekratnik broja 32, a to je broj elemenata u jednoj dimenziji matrice koji računa jedna jezgra grafičke kartice. Maksimalan broj jezgri koje mogu raditi paralelno smo ograničili na 512. S tim smo dobili da u svakom trenutnu grafička kartica može računati sličnosti sljedova dužine do 16000.

Prva razina paralelizacije jest na razini blokova (tj. broja jezgri grafičke kartice). Nakon što smo dobili broj blokova koje svaka jezgra mora riješiti to prosljeđujemo jednom kernelu (jezgri grafičke kartice) koji će organizirati redoslijed računanja memorijskih blokova po sporednoj dijagonali. Unutar početnog kernela računa se koji blokovi se moraju rješavati u isto vrijeme s obzirom na sporednu dijagonalu. Nakon što to izračuna unutar for petlje ići će po sporednim dijagonalama i pozivati nove kernele koji će onda u isto vrijeme rješavati blokove na sporednim dijagonali.

Početni kernel ujedno računa i početne pozicije memorijskog bloka koje će pojedini kerneli rješavati. To se računa na način: ako je trenutna pozicija u prvom stupcu, u listu se dodaje početna pozicija bloka desno i početna pozicija bloka ispod, u suprotnom se dodaje samo početna pozicija bloka desno. Tijekom tog izračuna ispituje se da li je trenutni blok zadnji u redu ili zadnji u stupcu. Ako je, onda se ne izvršava nikakav izračun.

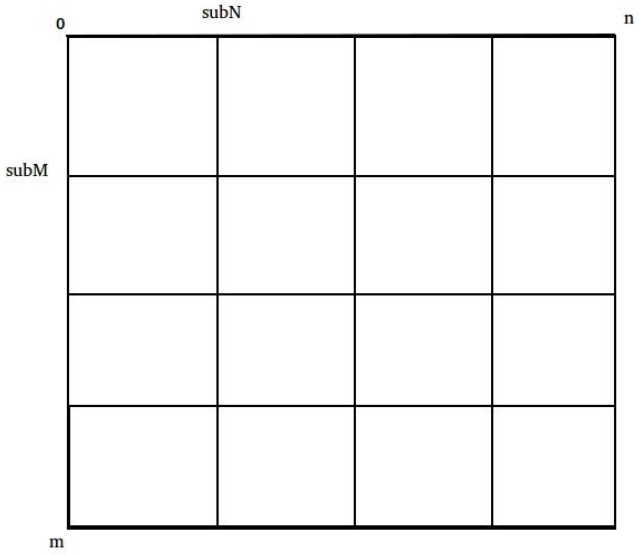
Formula po kojoj početni kernel izračuna koliko ima antidijagonala jest:

brojDijagonala = (dimX + dimY - 1)

Početni kernel tijekom pozivanja kernela po anti-dijagonali pazi da u niti jednom trenutku ne može biti više kernela nego jest veličina anti-dijagonale i to se pazi sa sljedećom usporedbom:

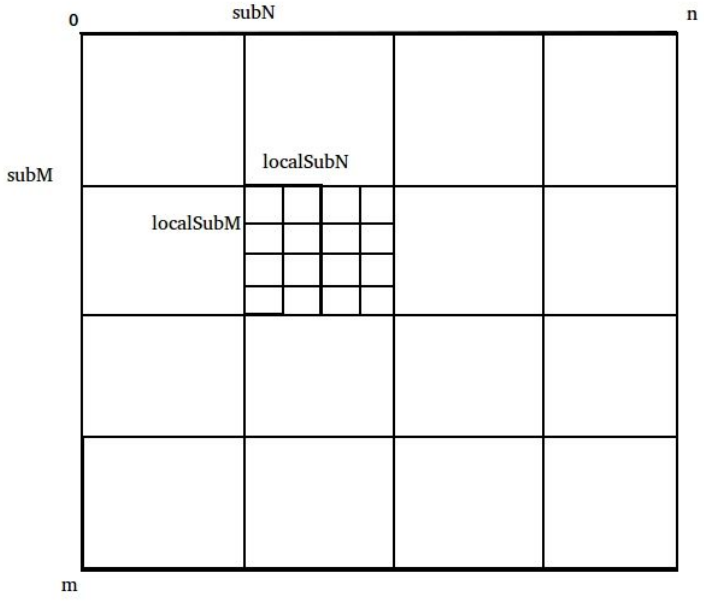
numBlocks=min(numBlocks_m,min(i,numBlocks_n));

Parametar numBlocks_m označava broj blokova po vertikalnoj dimenziji, numBlocks_n označava broj blokova po horizontalnoj dimenziji, a “i” označava iteraciju sporedne dijagonale. Na slici 2 se vidi podjela blokova koje će rješavati pojedini kerneli.



Slika 2. Prikaz dodijeljivanja blokova jezgrama

Druga razina paralelizacije je na razini dretvi. Svaka jezgra ima određeni broj dretvi (maksimalan broj dretvi po jezgri je 1024). Kada se to podijeli po dimenzijama, dobije se da svaki kernel rješava 32 elementa po dimenziji. Organizacija na razini dretvi je napravljena tako da je svakoj dretvi dodijeljena jedna memorijska lokaciju koju ona rješava. Sinkronizacija je napravljena na sljedeći način: prvo se izračuna koliko će biti sporednih dijagonala a to je po istoj formuli kao i na razini blokova (broj_elemenata_u_horizontalnoj_dimenziji + broj_elemenata_u_vertikalnoj_dimenziji - 1). Svaka dretva ima svoj identifikacijski broj (threadIdx.x) te po tome nalazi memorijski element koji računa. Dretva zna u kojem trenutku može izračunati svoj element, a to je kada je u iteraciji njezin broj aktivan. Broj se dobije po formuli (threadIdx.x/BlockSize_n + threadIdx.x%BlockSize_n).



Slika 3. Prikaz dodijeljivanja blokova dretvama

Na slici 3. subN i subM označavaju koordinate početka memorijske lokacije koju jezgra rješava, dok localSubN i localSubM označavaju početke memorijske lokacije koje rješavaju pojedine dretve. Tablica sličnosti genetskih sekvenci je u računalnoj (grafičkoj) memoriji spremljeno je kao jednodimenzionalno polje veličine (dužina prve sekvence + 1)*(dužina druge sekvence + 1). Dužine su uvećane za 1 zato što su po algoritmu potrebni dodatni početni redak i početni stupac koji su inicijalizirani na 0. Pozicija koja se računa u memoriji dana je sljedećom formulom:
(index/n + threadIdx.x/BlockSize_n)*n + (index%n + threadIdx.x%BlockSize_n)

gdje je:

- n dužina gornje sekvence
- Index - početni položaj bloka
- blockSize_n - broj elemenata u bloku kojeg jezgra rješava
- threadIdx.x - identifikacijski broj dretve

S prethodnim funkcijama računali smo matricu cijene. Nakon matrice cijene ostaje nam da pronađemo memorijski element s najvećim poklapanjem te se od tog elementa pomičemo po matrici dok ne rekonstruiramo poravnanje. Sama funkcija je napravljena tako da prima matricu cijene preko pokazivača, kao i poziciju najvećeg elementa, veličinu reda matrice te sekvence 1 i 2. Počinje tako da od trenutne pozicije uzima onu koja se nalazi na poziciji iznad, slijeva te lijevo gore. Po prioritetima ispituje prvo dijagonalno, gore pa lijevo. Uzme najveći od tih elemenata, pomakne se na tu poziciju te spremi kakva je operacija potrebna, bilo ona podudaranje, nepodudaranje, brisanje ili umetanje.

Poravnanje radimo na procesoru. Poravnanje se računa tako da se uzme pozicija najvećeg elementa u matrici. Taj element je početna pozicija. Tada se uspoređuju vrijednosti memorijskih elemenata iznad, lijevo, te iznad-lijevo u dijagonali. Od tih elemenata uzima se onaj koji ima najveću vrijednost. Tada se obilježava prikladna operacije, bila ona poklapanje, ne-poklapanje, umetanje ili brisanje i pomičemo se na tu poziciju. To se ponavlja dok se ne dođe do pozicije gdje je vrijednost memorijskog elementa jednaka 0.

4. Rezultati

S obzirom na to da se naša implementacija temelji na radu (Korpar M.,Šikić M.) [1], naš rezultat uspoređivat ćemo sa SW# bibliotekom koja je implementacija tog rada. Na početku će se prikazati izračun tablice sličnosti našeg riješenja za dva manja niza (6 x 4).

Primjer prikazuje nizove “ATATTA” i “ATAT”. Rezultat se može vidjeti u Tablici 1. Kao što se može vidjeti, tablica je veća nego što treba biti jer je bila proširena kako bi bio efikasniji izračun.

		A	T	A	T	T	A				
	0	0	0	0	0	0	0	0	0	0	0
A	0	1	0	1	0	0	1	0	0	0	0
T	0	0	2	0	2	1	0	0	0	0	0
A	0	1	0	3	1	0	2	0	0	0	0
T	0	0	2	1	4	2	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0

Tablica 1. Rezultati izračuna za nizove ATATTA i ATAT

Rezultat matrice cijene je dobar za usporedbu manjih sekvenci. Uvidom u takve sekvence čije su matrice čitljive potvrdili smo da su cijene dobre. Rekonstrukcija puta s dohvaćene pozicije na kojoj se nalazi maksimalni element je dobar, i ona ovisi o matrici cijene. Rekonstrukcija se radi na način da se krene od pronađene najveće cijene u matrici i uvidom u okolne elemente (dijagonalno lijevo, lijevo i gore) se ispisuje “m” za “match” i “mismatch” te “i” za umetanje i “d” za brisanje. Također, ispis poravnanja je dobar te ispisuje za svaki match ili mismatch znakove pripadajuće sekvence, odnosno “-” ako se radi o brisanju ili umetanju u pripadajućoj sekvenci. Ograničenje broja jezgara u CUDA arhitekturi je 512.

U izračun vremenskog izvođenja ušlo je izvršavanje SW algoritma nad matricom, rekonstrukcija poravnanja, kao i vrijeme inicijaliziranja memorije i varijabli. Memorijsko složenost našeg algoritma je kvadratna O(n*m), dok je memorijska složenost SW# algoritma O(n+9*m). Zbog memorijske složenosti i ograničenja memorije na grafičkoj kartici nismo bili u mogućnosti ispitati sekvence veće od 10^4. Za sekvence veličine između 10^2 i

10^3 elemenata naša implementacija je 56 puta sporija od SW#, dok je na sekvencama između 10^3 i 10^4 sporija 4.69 puta.

Vremensko izvođenje

Raspon duljine sekvenci	Vrijeme SW# (s)	Vrijeme naše implementacije (s)
10^2 - 10^3	0.0007920	0.0448604
10^3 - 10^4	0.0846200	0.3969080
10^4 - 10^5	0.1977960	-
10^5 - 10^6	12.500969	-

Tablica 2. Usporedba rezultata vremenskog izvođenja na SW# algoritmu i naše implementacije

Memorijsko zauzeće

Raspon duljine sekvenci	Zauzeće naše implementacije (Mbyte)
10^2 - 10^3	820.875
10^3 - 10^4	820.625
10^4 - 10^5	-
10^5 - 10^6	-

Tablica 3. Prikaz memorijskog zauzeća na grafičkoj kartici

U tablici 3. može se vidjeti memorijsko zauzeće na grafičkoj kartici za našu implementaciju. Rezultati pokazuju da naša implementacija nije memorijski efikasna i ima kvadratnu složenost.

5. Zaključak

Kroz projekt su se svi članovi tima detaljno upoznali s algoritmima globalnih i lokalnih poravnanja. Budući da su svi članovi bili bez iskustva u paralelizaciji, radu na razini grafičke kartice te programiranjem u CUDA programskom paketu, projekt možemo smatrati uspješnim jer smo stekli puno znanja u tom području.

Iako projekt nije do kraja uspješno implementiran možemo zaključiti da dobro izvodi wavefront metodu te da dobro sinkronizira blokove. Uspješno smo ubrzali računanje za matricu cijene. Za dovršetak implementacije potrebno je popraviti wavefront metodu na grafičkoj, implementirati traženje optimalnog puta te rekonstrukciju wavefront metodom. Mi smo na najjednostavniji način implementirali rekonstrukciju i poravnanje, a to je na procesoru kroz jednu while petlju.

U usporedbi s SW# algoritmom [1] naša implementacija puno sporije računa poravnanja. Uspoređujući rezultate s online alatima poput [3] na malim sekvencama algoritam radi dobro.

Literatura

1. M Korpar, M Šikić: SW#GPU-enabled exact alignments on genome scale, Bioinformatics 29 (19), 2494–2495
2. Hirschberg, Daniel S. "A linear space algorithm for computing maximal common subsequences." *Communications of the ACM* 18.6 (1975): 341-343.
3. http://bioinformaticnotes.com/Needle-Water/