SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 373

GPU implementacija vremenski efikasnog algoritma za lokalno poravnavanje s linearnom memorijskom složenosti

JOSIP HUCALJUK

Zagreb, srpanj 2012

Sadržaj

1. Uvod

Bioinformatika je znanstveno područje koje obuhvaća primjenu računarske znanosti i informacijskih tehnologija u području biologije i medicine. Tipični zadatci karakteristični za bioinformatiku su pohranjivanje velikih skupova bioloških podataka, razvijanje algoritama za obradu tih podataka, strojno učenje, vizualiziranje podataka i rezultata itd.

Početkom genetske revolucije pojavila se potreba pohrane velike količine novih bioloških podataka, što je ujedno i predstavljalo glavnu zadaću bioinformatike. Razvoj bioloških znanosti, kao i težnja da se što bolje shvate aktivnosti na staničnoj razini dovelo je do toga da je danas glavni zadatak bioinformatike analiza i kombiniranje prikupljenih bioloških podataka pri čemu algoritmi poravnavanja sekvenci igraju jednu od ključnih uloga. Njihov zadatak je poravnavanje dvaju ili više sekvenci DNK podataka, koji su prikazani u obliku slijeda znakova abecede, tako da se postigne maksimalna ocjena sličnosti. Na temelju toga se onda mogu lakše uočiti razne evolucijske veze između danih sekvenci.

Iako naizgled zvuči jednostavan problem, s računarske strane radi se o složenom procesu koji iziskuje velike količine resursa i vremena. Problemu složenosti pokušava se doskočiti na 2 načina: razvojem učinkovitijih algoritama, te korištenjem bržeg sklopovlja.

Glavna tema ovoga rada je implementacija prostorno i vremenski efikasnog algoritam za lokalno poravnavanje sekvenci na grafičkim procesorima. Današnje moderne grafički procesore karakterizira masivno paralelizirana arhitektura koja u usporedbi s centralnim procesorima omogućava par redova veličine bolje performanse. Iz toga razloga grafički se procesori sve više koriste kod računski intenzivnih zadataka.

U sljedećim poglavljima ukratko će biti opisani najznačajniji algoritmi za poravnavanje sekvenci (2. poglavlje), arhitektura grafičkih procesora (3. poglavlje), implementacija algoritma (4. poglavlje), te na kraju i sami rezultati ostvarenog rješenja.

2. Algoritmi za poravnavanje sekvenci

Kao što je rečeno u uvodu, algoritmi za poravnavanje sekvenci igraju vrlo važnu ulogu u bioinformatici. Većina algoritama zasniva se na dinamičkom programiranju koji daju egzaktne rezultate, ali zbog toga imaju dugo vrijeme izvođenja. Najpoznatiji predstavnici su Needleman-Wunschov [1] i Smith-Waterman [2] algoritmi. Alternativno, postoje algoritmi koji koriste različite heuristike kako bi smanjili složenost i vrijeme izvođenja, ali pri tom ne garantiraju točnost rezultata. BLAST algoritam [3] je najpoznatiji među njima.

U sklopu ovoga rada bit će opisani samo algoritmi koji daju egzaktna rješenja. U nastavku će biti opisani Needleman-Wunschov algoritam, koji vrši globalno poravnavanje, te Smith-Waterman algoritam koji pak izvodi lokalno poravnavanje sekvenci. Dodatno će bit opisana modifikacija Smith Waterman algoritma s linearnom prostornom složenošću koja je i implementirana.

2.1. Ocjena sličnosti

Ocjena sličnosti kvantitativna je mjera koja nam govori koliko su dvije sekvence slične, odnosno u kojoj se mjeri elementi elementi prvog slijeda poklapaju s elementima drugog slijeda. Ocjena se radi prema skupu pravila. Najjednostavnija shema ocjenjivanja je dodjeljivanje 1 za poklapanje elemenata, -1 u slučaju nepoklapanja, te 0 ukoliko se umeće procjep. Primjer ocjenjivanja za sekvence ATGGCGT i ATGAGT.

ATGGCGT
1+1+1+0-1+1+1 = 4
ATG-AGT

Navedeno rješenje ujedno predstavlja i najbolje poravnavanje. Do njega se dolazi ispitivanjem svih mogućih kombinacija umetanja procjepa iz čega i proizlazi velika složenost algoritama.

U praksi se koriste složenija pravila za ocjenjivanje temeljena na supstitucijskim matricama te naprednijim modelima kažnjavanja umetanja procjepa.

2.2. Supstitucijske matrice

Korištenje konstantne vrijednosti u slučaju poklapanja i nepoklapanja elemenata primjenjivo je u slučaju poravnavanja DNK sekvenci. Kod poravnavanja sekvenci proteina koriste se naprednije supstitucijske matrice zbog većeg broja elemenata i njihove složenosti. Supstitucijske matrice temelje se na svojstvima aminokiselina:

- Veličina
- Polaritet
- Naboj
- Hidrofobnost

U praksi se najčešće koriste evolucijske supstitucijske matrice koje možemo podijeliti u dvije skupine:

- PAM matrice: PAM250, PAM120
- BLOSUM matrice: BLOSUM62, BLOSUM50

Na slici 1 prikazan je sadržaj BLOSUM62 matrice. [4]

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Slika 1 - BLOSUM62 supstitucijska matrica

2.3. Procjepi

U prethodnom primjeru utjecaj procjepa se ignorirao. Procjepi odgovaraju ubacivanju praznine u neku sekvencu na određeno mjesto. U pravilu kazna za ubacivanje procjepa je puno veća nego za nepodudaranje elemenata jer se unošenjem procjepa prekida lanac polimera. Kaznu za više procjepa može se odrediti na sljedeće načine [5]:

- Konstanta cijena – pridjeljuje se ista cijena bez obzira na duljinu procjepa
- Linearna cijena – ako je cijena otvaranja procjepa jednaka **k**, onda cijena otvaranja procjepa duljine **n** jednaka je **n x k**
- Afina cijena – uvodi se cijena za otvaranje procjepa **o** te cijena za svako jedinično proširenje procjepa c. Ukupna cijena je jednaka **o + (n – 1) x e**

Dodjeljivanje kazni za višestruke procjepe po konstantnoj cijeni u praksi se više ne koristi zbog velikog odstupanja od idealnog modela. Linearni model najčešće se koristi kod jako dugačkih sekvenci gdje ne dolazi do izražaja toliko utjecaj ocjenjivanja procjepa. Afin model ocjenjivanja najčešće se koristi jer pruža najbolje rezultate iako komplicira izračun u određenoj mjeri.

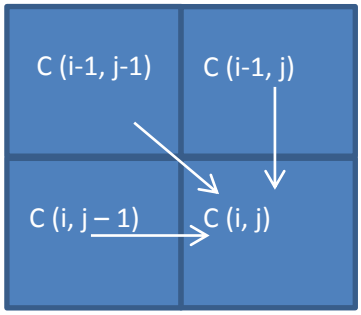
2.4. Needleman-Wunsch algoritam

Needleman-Wunschov algoritam je deterministički algoritam koji daje egzaktno rješenje globalnog poravnavanja dviju sekvenci. Rješava se dinamičkim programiranjem. Osnovna ideja je da se najbolje globalno poravnavanje postiže korištenjem najboljih poravnavanja kraćih sekvenci. Navedena ideja odgovara 'podijeli i vladaj' strategiji; osnovni problem se podijeli na manje probleme, svaki potproblem riješi se optimalno i onda se na temelju rješenja potproblema izgradi rješenje za početni problem.

U postupku se koriste dvije matrice dimenzija (N+1) x (M+1) (N i M su duljine ulaznih sekvenci). Jedna matrica se koristi za pohranjivanje izračunatih rezultata, a druga za pohranjivanje poteza kako bi se mogla napraviti rekonstrukcija najboljeg poravnavanja. Cijeli algoritam može se podijeliti u 3 koraka:

1. Inicijaliziranje matrice rezultata
2. Proračun i popunjavanje matrice rezultata i poteza
3. Rekonstrukcija poravnavanja na temelju matrice poteza

Za izračun vrijednosti jednog elementa matrice rezultata potrebno je znati vrijednosti neposredno lijeve, gornje i sjeverozapadno dijagonalne ćelije.



Slika 2 - Ovisnost podataka pri izračunu vrijednosti ćelije

Vrijednost ćelije na poziciji (i,j) računa se na sljedeći način:

$$C(i,j) = \max \begin{cases} C(i-1,j-1) + S(i,j) \rightarrow dijagonalna\ ćelija \\ C(i-1,j) - g \rightarrow gornja\ ćelija \\ C(i,j-1) - g \rightarrow lijeva\ ćelija \end{cases}$$

S(i,j) je vrijednost koja se dobije iz supstitucijske matrice za znakove *i* i *j,* dok je *g* kazna za otvaranje procjepa. U navedenoj definiciji koristi se linearno kažnjavanje procjepa. Dodavanjem afine ocjene, postupak se nešto komplicira.

Primjer izvođenja algoritma za poravnavanje sekvenci SEND i AND. Koristi se supstitucijska matrica BLOSUM62, a kazna za otvaranje procjepa je 10.

		S	E	N	D
	0	-10	-20	-30	-40
A	-10				
N	-20				
D	-30				

	S	E	N	D
kraj	lijevo	lijevo	lijevo	lijevo
gore				
gore				
gore				

Slika 3 - Inicijalni sadržaj matrica u Needleman-Wunsch algoritmu

Izračun vrijednosti za polje (2,2):

$$C(2,2) = max\begin{cases} C(1,1) + S(S,A) = 0 + 1 = 1 \\ C(1,2) - g = -10 - 10 = -20 \\ C(2,1) - g = -10 - 10 = -20 \end{cases}$$

Najveći rezultat je 1 i to se upisuje u matricu rezultata. Do njega se dolazi na temelju dijagonalnog elementa te se stoga u tablicu poteza dodaje potez 'diag'.

Isti postupak provodi se za sve preostale elemente matrice. Konačni sadržaj matrica prikazan je na sljedećoj slici.

		S	E	N	D
	0	-10	-20	-30	-40
A	-10	1	-9	-19	-29
N	-20	-9	-1	-3	-13
D	-30	-19	-11	2	3

	S	E	N	D
kraj	lijevo	lijevo	lijevo	lijevo
gore	diag	lijevo	lijevo	lijevo
gore	diag	diag	diag	lijevo
gore	gore	diag	diag	diag

Slika 4 - Konačni sadržaj matrica na kraju Needleman-Wunsch algoritma

Rekonstrukcija poravnavanja izvodi se na način da se krene od donjeg desnog elementa, te se prate potezi prema gornjem lijevom elementu. Konačni rezultat za zadani primjer je:

```
SEND
A-ND
```

2.5. Smith Waterman algoritam

Smith-Waterman algoritam za razliku od Needleman-Wunsch algoritma lokalno poravnava sekvence. Lokalno poravnavanje sekvenci ne mora se nužno sastojati od svih elemenata dviju sekvenci kao što je to kod globalnog. Algoritam predstavlja varijaciju prethodno opisanog Needleman-Wunsch algoritma, što znači da se i dalje radi o algoritmu temeljenom na dinamičkom programiranju koji nudi egzaktno rješenje. Sami postupak rješavanja sličan je Needleman Wunschovom algoritmu s dvije razlike:

- Negativni rezultati u matrici rezultata postavljaju se na vrijednost 0
- Rekonstrukcija kreće od elementa s najvećom vrijednosti i zaustavlja se kad se dosegne element s vrijednošću 0

Za primjer iz prethodnog poglavlja, inicijalizacija matrice bi bila sljedeća:

		S	E	N	D
	0	0	0	0	0
A	0				
N	0				
D	0				

	S	E	N	D
kraj	lijevo	lijevo	lijevo	lijevo
gore				
gore				
gore				

Slika 5 - Inicijalni sadržaj matrica u Smith-Waterman algoritmu

Nakon što se provede postupak za sve elemente dobiju se sljedeće matrice:

		S	E	N	D
	0	0	0	0	0
A	0	1	0	0	0
N	0	0	0	0	0
D	0	0	0	2	3

	S	E	N	D
kraj	lijevo	lijevo	lijevo	lijevo
gore	diag	lijevo	lijevo	lijevo
gore	diag	diag	diag	lijevo
gore	gore	diag	diag	diag

Slika 6 - Konačni sadržaj matrica na kraju Smith-Waterman algoritma

Za dani primjer element s najvećom vrijednosti je upravo donji desni element, tako da je rezultat jednak onome koji daje i Needleman-Wunsch algoritam,

odnosno lokalno poravnavanje je jednako globalnom. Za duže sekvence mala je vjerojatnost da će rezultati lokalnog i globalnog poravnavanja biti jednaki.

2.6. Modifikacija Smith Watermana

Glavni problem postojećih algoritama za lokalno poravnavanje temeljenih na dinamičkom programiranju je prostorna složenost koja je jednaka umnošku duljina sekvenci. Takva kvadratna prostorna složenost ograničava upotrebljivost algoritama samo na kratke sekvence. Za globalno poravnavanje poznati su algoritmi koji imaju linearnu prostornu složenost, odnosno složenost im je jednaka sumi duljina sekvenci (Hirschbergov algoritam [6]).

Za lokalno poravnavanje situacija je malo složenija. Huang je 1990. godine [7] predstavio algoritam za lokalno poravnavanje koji je koristio Myers-Miller postupak [8] kako bi sveo prostornu složenost na linearnu. Iako je time bio riješen problem prostorne složenosti, algoritam nije bio primjenjiv zbog velike vremenske složenosti koja je iznosila O(kMN), pri čemu su M i N duljine ulaznih sekvenci, a k željeni broj najboljih poravnavanja. Konačno Huang i Miller su 1991 [9] predstavili algoritam koji zadržava linearnu prostornu složenost i smanjuje vremensku složenost na $O(MN + \sum_{n=1}^{k} L_n^2)$, pri čemu je L_n duljina n-og dobivenog poravnavanja. Navedeni algoritam ostvaruje puno bolje performanse u slučaju kad je K velik, a dobivena poravnavanja kratka u odnosu na ulazne sekvence. U nastavku će biti detaljnije opisan algoritam.

2.6.1. Huang-Miller algoritam

Prvi korak algoritma je provođenje Smith-Waterman algoritma kako bi se popunila matrica rezultata s razlikom da umjesto najboljeg rezultata, pamtimo k najboljih rezultata. Pri tome treba voditi računa da putovi koji vode prema najboljim rezultatima ne dijele niti jednu točku u grafu. Postavljanjem zadnjeg uvjeta gubi se garancija da će K pronađenih putova u grafu iz jednog prolaza Smith Waterman algoritma uistinu biti i najbolji putovi. Navedeni problem rješava se višestrukim provođenjem Smith-Waterman algoritma na manjem lokaliziranom području.

Za svaki čvor C koji je dobiven u prvom koraku algoritma dodatno pamtimo i čvor F iz kojeg počinje put prema njemu. Početni čvor F potreban je kako bi se

moglo rekonstruirati put prema čvoru C jer za razliku od osnovnog Smith Waterman algoritma ovdje nemamo pohranjenu tablicu poteza na temelju koje bi mogli provest rekonstrukciju. Čvor F računa se tako da se prilikom izračuna ocjene sljedećeg čvora na nekom putu jednostavno naslijedi čvor F od prethodnog. Ocjena nekog čvora i njegov početni čvor jedinstveno ga smještaju u neku klasu ekvivalencije. Rezultat prvog koraka algoritma je podjela grafa u klase ekvivalencije pri čemu se samo K najboljih pamti. Klasa ekvivalencije S definirana je kao sedmorka $< C, F, u, T, B, L, R >$

- C = ocjena(S)
- F = F(s) za sve čvorove $s \in S$
- F(u) = F i C(u) = ocjena(S)
- [T, B] x [L, R] sadrži sve $s \in S$ s C(s) > W, pri čemu je W ocjena najlošije klase koja se pamti

Nakon što se popuni lista najboljih klasi prelazi se na rekonstruiranje poravnavanja. Iz liste se uzimaju klase od najbolje prema lošijima, rekonstruira se njihov put Myers i Miller algoritmom (rezultat toga je zapravo samo poravnavanje), te se brišu bridovi tog puta iz cijelog grafa. Nakon brisanja bridova određuje se područje grafa nad kojim će se ponovo provesti Smith Waterman algoritam kako bi se pronašli dobri putovi koji su možda bili skriveni u prvom prolazu algoritma. Tu se javlja problem kako odrediti koje područje ponovo istražiti. Jedan pristup bi bio provjeriti cijeli graf, što garantira da ćemo pronaći neki bolji put ako on uistinu postoji, ali rezultat toga bi bila prevelika vremenska složenost. Huang i Miller su pokazali da je dovoljno provjeriti samo čvorove u blizini obrisanog puta, jer samo za njih postoji vjerojatnost da im je promijenjena ocjena. Točan algoritam za to je sljedeći:

- Pokrećemo Smith Waterman algoritam od krajnjeg čvora obrisanog puta, ali u obrnutom smjeru.
- Algoritam se provodi sve dok svi čvorovi na gornjoj i lijevog granici ne budu imali krajnji čvor izvan područja [T', S.B] x [L', S.R]
- Nakon što se dobiju L i T granice, provodi se još jedan prolaz Smith Watermana algoritam nad tim dobivenim područjem i pri tome se osvježava lista klasi ukoliko se nađe neka klasa sa ocjenom većom od W

Navedeni postupak provodi se K puta, odnosno za svako poravnavanje. Okvirni pseudokod algoritma:

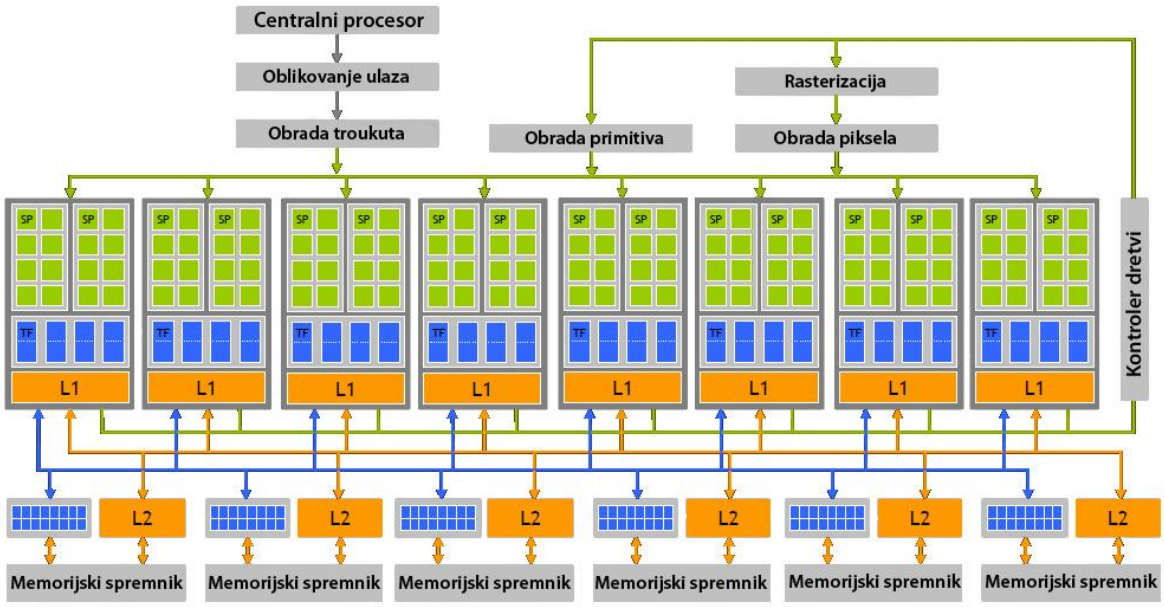
```
W = 0;
for( i = 0; i < M; i++ )
{
      for( j = 0; j < N; j++ )
      {
            izračunaj C(i,j) i Fc(i,j)
            if( C(i,j) > W )
                  W = ubaci_u_listu_klasi(C(i,j), F(i,j))
      }
}

for( n = 1; n < k; n++ )
{
      S = uzmi_najbolju_klasu_iz_liste
      rekonstruiraj_put(S)
      ako (n != k)
      {
            odredi granice T i L tako da svaki put koji započinje izvan
            [T, S.B] x [L, S.R], a završava unutar [S.T, S.B] x [S.L, S.R]
            ima ocjenu manju od W
            for( i = T; i < S.B; i++ )
            {
                  for( j = L ; j < S.R; j++ )
                  {
                        izračunaj C(i,j) i Fc(i,j)
                        if( C(i,j) > W & (i,j) se nalazi u području [S.T,
                        S.B] x [S.L, S.R])
                              w = ubaci_u_listu_klasi(C(i,j), F(i,j))
                  }
            }
      }
}
```

Složenost kompletnog algoritma u najgorem slučaju je O(kMN) i to onda kad je svako dobiveno poravnavanje po duljini vrlo blizu ulaznim sekvencama. U praksi se algoritam pokazao puno brži jer samo par poravnavanja bude relativno dugačko. Vrijeme izvođenja algoritma također uvelike ovisi o kažnjavanju procjepa i supstitucijskim matricama. Kada se nepoklapanja lagano kažnjavaju, područja koja se moraju dodatno provjeriti su vrlo velika te se samim time vrijeme izvođenja algoritma produžuje.

3. Arhitektura grafičkih procesora

3.1. Arhitektura modernih grafičkih procesora

Kao što je rečeno u uvodu, današnje grafičke procesore karakterizira masivno paralelizirana arhitektura. Na slici 7 [10] prikazana je konceptualna shema modernog grafičkog procesora.



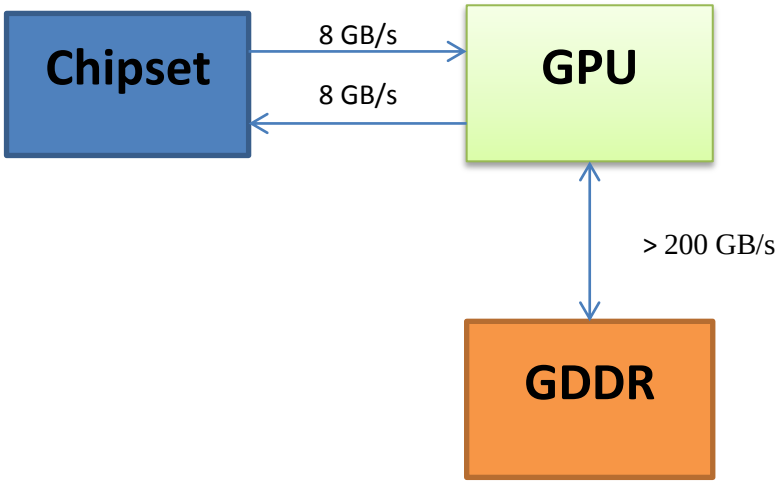
Slika 7 - Shema modernog grafičkog procesora

Visoka paraleliziranost grafičkog procesora ostvarena je velikim brojem jednostavnih procesorskih jezgri (engl. *Streaming Processors*). Radi jednostavnijih kontrolnih mehanizama i pristupa memoriji, jezgre su grupirane u veće nakupine (uobičajeno po 192 SP-ova u trenutnoj Kepler generaciji) koje se nazivaju multiprocesori (engl. *Streaming Multiprocessors*). Svaka procesorska jezgra sastoji se od 1 jedinice za zbrajanje i množenje (MAD) i 1 jedinice koja izvodi samo množenje (MUL). Kad se uzme u obzir da današnje moderne grafičke kartice iz višeg segmenta imaju i do 3072 takve jezgre, koje rade na frekvencijama preko 1 GHz, jednostavnim računom može se doći do brojke od preko 6 TFLOPS-a (**T**era **Fl**oating Point **Op**erations **P**er **S**econd), odnosno 6 bilijuna operacija s pomičnim zarezom u sekundi. Takva računalna snaga do sada je bila raspoloživa

isključivo na vrlo skupim superračunalima. Detaljniji opis strukture grafičkog procesora može se naći u [11] i [12].

Sva ta raspoloživa računalna snaga ne bi bila od prevelike koristi bez podataka nad kojima bi se ona iskorištavala. Svi podatci koji će se koristiti pri proračunima spremaju se u glavnu grafičku memoriju (engl. *global memory*) čiji kapacitet doseže i do 4 GB. Budući da je sustav jak koliko i najslabija karika u sustavu, izrazito bitno je ostvariti brzi pristup podatcima u memoriji. U tu svrhu koriste se brze verzije DDR memorija (frekvencije preko 6 GHz efektivno), koje imaju i posebnu oznaku GDDR (**G**raphics **D**ouble **D**ata **R**ate), te izrazito široke sabirnice (do 384 bita). Rezultat je propusnost preko 200 GB/s kod najbržih modela. Naravno, bitno je napomenuti, da je to teoretski najveća ostvariva brzina koja se postiže optimalnim pristupom memoriji. Pod optimalnim pristupom podrazumijeva se čitanje podataka iz susjednih memorijskih lokacija. U slučaju nasumičnog čitanja iz memorije, propusnost u znatnoj mjeri opada.

Podatci prije obrade moraju se na neki način premjestiti u grafičku memoriju. To se ostvaruje preko PCI Express Gen3 sučelja, koje naspram propusnosti na relaciji GPU – memorija ima izrazito mali iznos od 8 GB/s u oba smjera. Iako na prvi pogled to se može činiti kao izrazito usko grlo, podatci se preko njega prenose relativno rijetko naspram komunikacije GPU – memorije, te stoga i ne predstavlja poseban problem. Komunikacija s grafičkim procesorom prikazana je na slici 8.



Slika 8 - Shema komunikacije i prijenosa podataka prema grafičkom procesoru

3.2. CUDA

3.2.1. Općenito

Ideja korištenja grafičkih procesora u svrhu obrade općih podataka, postojala je i prije pojave CUDA arhitekture. Najveći problem koji je sprječavao širu uporabu bilo je ograničenje komunikacije s GPU-om isključivo preko grafičkog sučelja (OpenGL, DirectX). To je donosilo sa sobom brojne probleme, prvenstveno sa predočavanjem podataka. Rezultate obrade nije bilo moguće vratiti u prikladnom numeričkom obliku, jer rezultat obrade preko grafičkog sučelja bio je niz piksela, odnosno slika.

Veliki iskorak u GPGPU industriji dogodio se 2006. godine, predstavljanjem CUDA arhitekture. Po prvi puta dio silicija grafičkog procesora bio je posvećen za olakšavanje GPGPU programiranja. Uvedeno je posebno hardversko sučelje, kao i svi potrebni alati za efikasno paralelno programiranje. Od 2006. godine do danas prodano je više od 200 milijuna grafičkih kartica sa CUDA arhitekturom.

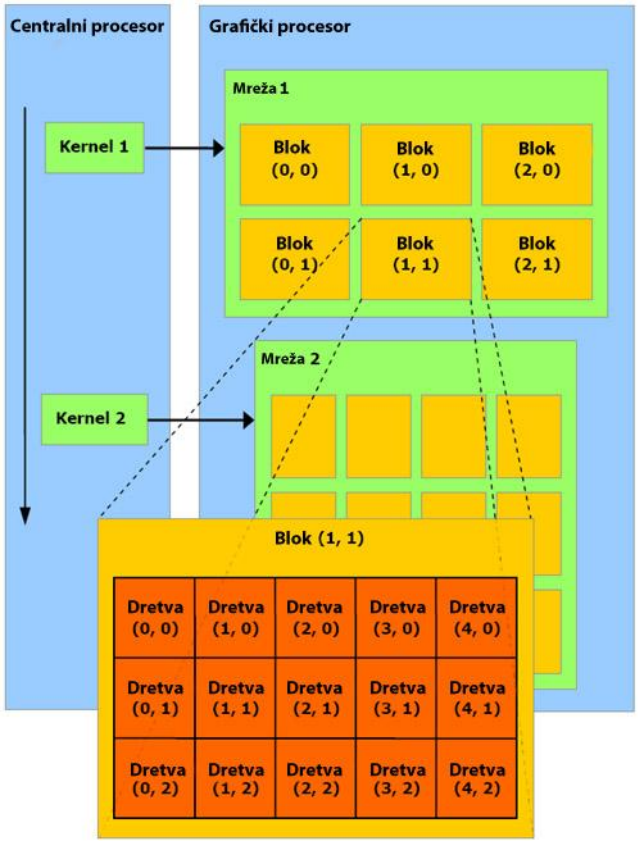
3.2.2. Logička struktura

Kod izrade paralelnih aplikacija korištenjem CUDA-e, moguće je razlikovati dvije vrste programskog koda: kod koji će se izvršavati na centralnom procesoru (engl. *host code*), te kod koji se izvršava na grafičkom procesoru (engl. *device code*). Osnovna ideja kod dizajniranja sustava je dio posla koji se mora sekvencijalno izvršavati prilagoditi za izvođenje na centralnom procesoru, a dio posla koji uključuje izvršavanje aritmetičkih operacija nad velikom količinom podataka istovremeno, prilagoditi za izvođenje na grafičkom procesoru.

Programski kod koji se izvršava na grafičkom procesoru sastoji se od niza funkcija koje se nazivaju jezgre (*engl. kernel*). Prilikom poziva jedne takve jezgre, sve stvorene dretve izvršavaju iste instrukcije, ali nad različitim podatcima što je karakteristično za SIMD arhitekturu. Koje podatke dretva treba koristiti određuju identifikatori *BlockID* i *ThreadID*. Stvorene dretve organizirane su u dvije razine. Na nižoj razini nalaze se *blokovi.* Blokovi su skupine dretvi koje mogu direktno komunicirati preko zajedničke memorije (engl. *shared memory*). Na višoj razini nalazi se mreža blokova (engl. *grid*). Dretve koje se nalaze u odvojenim blokovima

mogu komunicirati isključivo preko glavne grafičke memorije (engl. *global memory*).

Logička struktura prikazana je na slici 9. [13]



Slika 9 - Logička organizacija grafičkog procesora

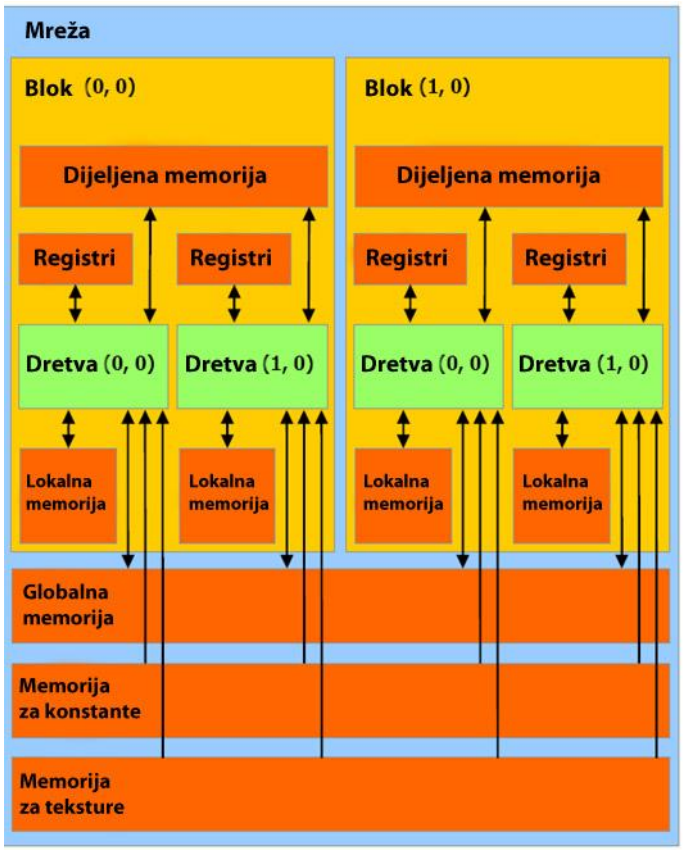
Glavni razlog odabira ovakve logičke strukture su performanse. Na ovaj način maksimizira se iskorištenost procesorskih jezgri, odnosno minimizira vrijeme koje jezgre provode čekajući na obavljanje neke visoko latentne operacije poput pristupa glavnoj memoriji. U trenutku izvršavanja neke jezgre svaki blok dretvi dijeli se na *warpove,* odnosno nakupine od 32 dretve. Svaki warp može se direktno izvoditi na jednom multiprocesoru. U svrhu postizanja maksimalnih performansi, broj dretvi u jednom bloku trebao bi biti djeljiv sa 32, jer u suprotnom warp koji nije potpun neće u potpunosti iskoristiti raspoložive procesorske jezgre.

3.2.3. Memorijska hijerarhija

Kod pohranjivanja podataka korisniku na raspolaganju dostupne su sljedeće memorije (sortirane po brzini pristupa):

- Registri
- Dijeljena memorija
- Memorija za konstante
- Teksture
- Globalna memorija

Na slici 10 [13] prikazana je shema memorijske hijerarhije. Detaljniji opis dostupan je u članku [13].



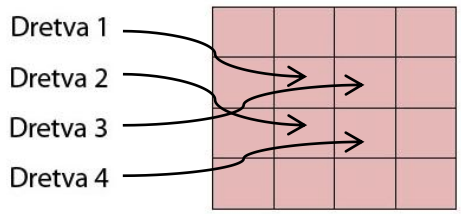
Slika 10 -Shema memorijske hijerarhije

Ključ za postizanje visokih performansi je efikasno korištenje raspoložive memorije. Globalna memorija ima veliki kapacitet, ali i vrlo dugo vrijeme pristupa

(oko 600 ciklusa). U idealnom slučaju koristi se za inicijalni prijenos podataka, te za vraćanje rezultata obrade.

Memorija za konstante specifični je dio globalne memorije, koji je posebno indeksiran što omogućava brži pristup (isključivo čitanje). Veličina je ograničena na 64 KB po mreži blokova, a unos podataka moguć je samo od strane centralnog procesora.

Još jedan tip memorije koji se može samo čitati od strane grafičkog procesora su teksture. Teksture po svojstvima predstavljaju kombinaciju globalne memorije (veliki kapacitet) i memorije za konstante (indeksirani pristup i isključivo čitanje). Indeksiranje je provedeno na poseban način kako bi ubrzao *lokalizirani* pristup memoriji. Slika 11 ilustrira pristup memoriji koji je prikladan za korištenje teksturi.



Slika 11 - Lokalizirani pristup memorijskim lokacijama

Pravilnom uporabom tekstura moguće je ubrzati izvođenje programskog koda i do 50% u odnosu na varijantu sa globalnom memorijom.

Dijeljena memorija definirana je na razini blokova, te je vrlo ograničene veličine (do 48 KB po bloku), ali omogućava vrlo brzi pristup podatcima (3 - 4 ciklusa). Što efikasnije rukovanje dijeljenom memorijom nužno je kako bi se zaobišlo ograničenje memorijske propusnosti.

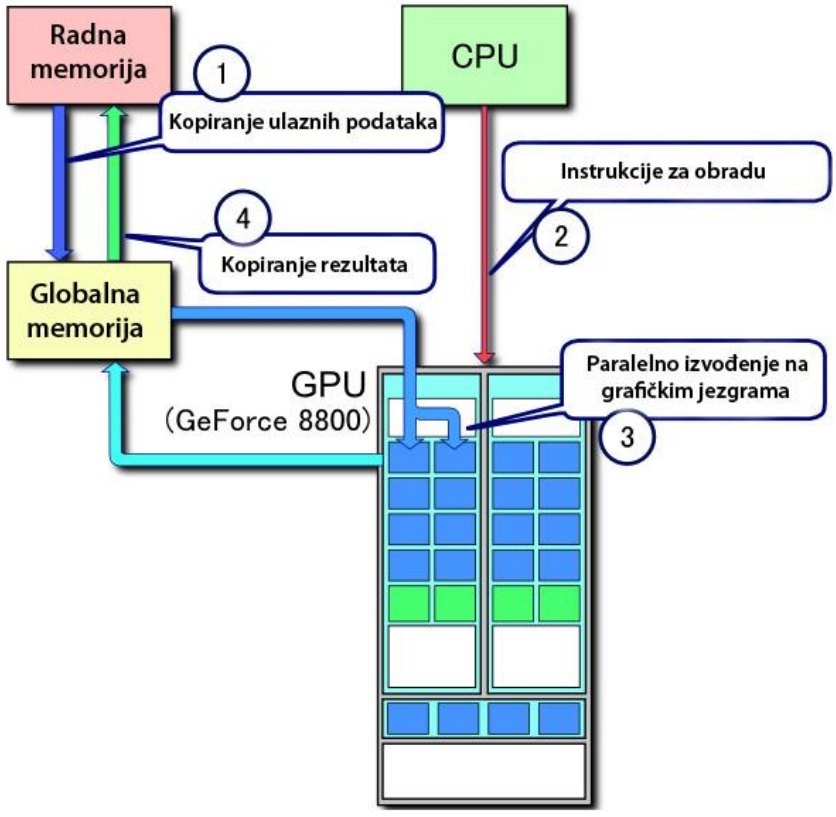
Zadnja, ujedno i najbrža, memorijska lokacija su registri. Registri su definirani na razini bloka dretvi, te ih je ukupno 65536 raspoloživo po bloku. Raspoloživi registri dijele se u jednakim dijelovima među dretvama u bloku. U slučaju da dretve koriste više registara nego što bi smjele, smanjuje se broj raspoloživih warpova koji se mogu izvršavati na jednom multiprocesoru. Svi jednostavni tipovi podataka (char, int, float...) spremaju se u registre, što omogućuje vrlo brzi pristup varijablama (1 ciklus). Složeni tipovi podataka (polja, stringovi itd.), ako nije

drugačije naznačeno, spremaju se u globalnu memoriju (na slici 10 označeno kao lokalna memorija), te o tome treba voditi računa prilikom programiranja.

3.2.4. Dijagram toka CUDA aplikacije

Na slici 12 [14] prikazan je dijagram toka standardne CUDA aplikacije. Moguće je identificirati sljedeće aktivnosti:

- Rezerviranje prostora u globalnoj memoriji za inicijalni skup podataka i rezultat izvođenja
- Prijenos podataka nad kojima će se vršiti obrada u globalnu memoriju
- Pozivanje odgovarajuće jezgre
- Prijenos rezultata obrade iz globalne memorije
- Oslobađanje zauzete memorije



Slika 12 - Dijagram toka izvođenja CUDA aplikacije

3.3. Razvojna okolina

Prije nego li se može krenuti sa razvojem CUDA aplikacija potrebno je prilagoditi razvojnu okolinu.

Prvi i osnovni uvjet bez kojeg nije moguće izvoditi napisane aplikacije je grafička kartica sa CUDA arhitekturom. S obzirom da većina Nvidia grafičkih kartica izdanih u posljednjih 5 godina podržava CUDA-u te ima dovoljnu količinu memorije to i ne predstavlja neki problem. U slučaju korištenja neke starije kartice koja podržava CUDA-u, ali ne zadovoljava minimalnih 256MB grafičke memorije, izvođenje je moguće, ali samo u slučajevima da se radi o jednostavnim operacijama nad malom količinom podataka.

Sljedeće po redu potrebno je pribaviti posebne upravljačke programe (eng. *driver*), te CUDA razvojne alate (engl. *CUDA Toolkit*) koji uključuju potrebne prevodioce, razne matematičke biblioteke, te još neke dodatne alate. Za testiranje postavljene okoline moguće je pokrenuti primjere koji dolaze s CUDA SDK-om (**S**oftware **D**evelopment **K**it).

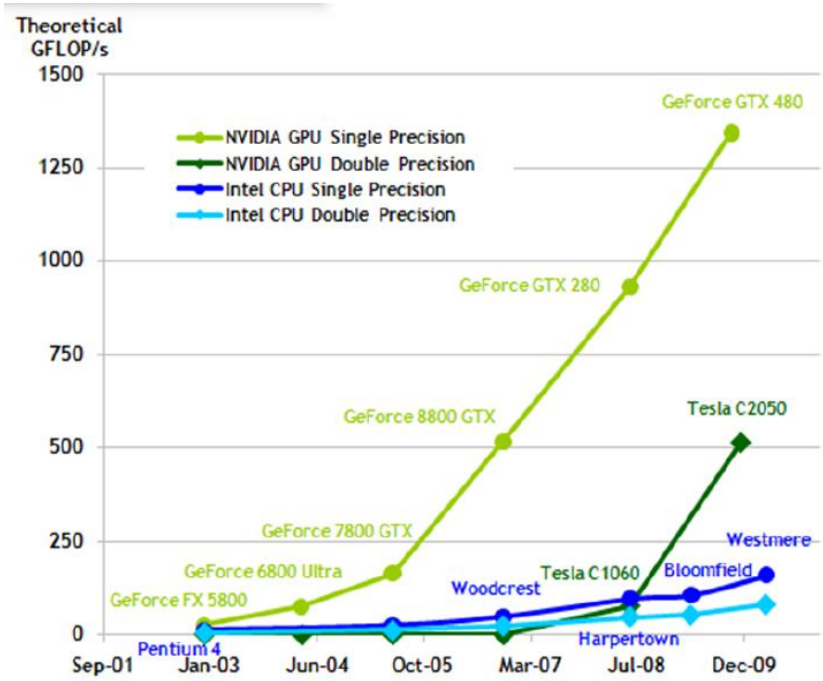
Razvoj CUDA aplikacija vrši se pomoću programskog jezika C sa CUDA specifičnim proširenjima (engl. *C for CUDA*). Odabir razvojne okoline prepušten je korisniku, te u suštini bilo koji C IDE (**I**ntegrated **D**evelopment **E**nvironment), poput MS Visual Studia ili Eclipse-a, poslužit će svrsi. Prilikom stvaranja novog projekta potrebno je podesiti putanje do gcc i Pathscale prevodioca, putanje do dodatnih biblioteka, kao i još neke manje bitne opcije. Ukoliko se koristi MS Visual Studio prilikom razvoja, moguće je instalirati aplikaciju *CUDA VS Wizard*. Pomoću nje, prilikom odabira vrste projekta, moguće je odabrati CUDA aplikaciju kod koje su sve opcije već podešene.

3.4. Usporedba sa centralnim procesorom

Veliki broj procesorskih jezgri koje uz to rade na relativno visokoj frekvenciji omogućuju grafičkim procesorima obavljanje izrazito mnogo operacija u sekundi, znatno više nego što to može centralni procesor. Dok današnji centralni procesori iz najvišeg segmenta razvijaju tek oko 100 GFLOPS-a, grafički procesori srednje klase obavljaju 10 puta više operacija u sekundi i sve to po 5 puta manjoj cijeni!

Razlika se očituje i u komunikaciji sa memorijom. Dok komunikacija između centralnog procesora i radne memorije se odvija brzinom do 12 GB/s, grafičke kartice imaju propusnost sve do 200 GB/s. Ipak, u obzir treba uzeti i činjenicu da komunikacija s centralnim procesorom ima manju latenciju, ali razlika i u tom slučaju ostaje više nego očigledna. Predviđa se da će kroz 3 godine propusnost centralnih procesora i radne memorije porasti na 50 GB/s, što je još uvijek znatno manje nego što to već danas ostvaruju grafičke kartice.

Slika 13 prikazuje razvoj centralnih i grafičkih procesora u razdoblju od 2003. do 2009. godine. [15]



Slika 13 - Razvoj CPU i GPU

Razlika u budućnosti će se nastaviti povećavati. Razlog tome može se naći u činjenici da veliki dio centralnog procesora obavlja kompleksne upravljačke operacije (npr. predviđanje grananja), te se mnogo tranzistora troši na priručne memorije raznih razina. Sve to ograničava broj jezgri koje je moguće integrirati, a da pri tome potrošnja ostane u prihvatljivim granicama.

Prema prognozama CEO Nvidia-e od prije 2 godine snaga grafičkih procesora kroz sljedećih 6 godina porasti će 570 puta! U istome razdoblju centralni procesori povećati će 3 puta snagu. Točan razlog i temelj za tako hrabru prognozu u ovome trenutku još nije poznat, ali ako se ostvari i dio obećanog pojačanja, omogućit će nevjerojatne pomake u industriji. [16]

Još jedna od bitnih razlika je mogućnost skaliranja performansi postojećeg softvera. Kod softvera napisanog za izvođenje na grafičkom procesoru, povećanjem broja raspoloživih procesorskih jezgri performanse rastu proporcionalno, bez ikakvih potrebnih modifikacija. Kod softvera napisanog za izvođenje na centralnom procesoru situacija je u drukčija. U većini slučajeva softver napisan za izvođenje na dvojezgrenom procesoru neće imati dvostruko bolje performanse ako bi se koristio na četverojezgrenom procesoru.

Što to sve znači? Postaju li centralni procesori nepotrebni?

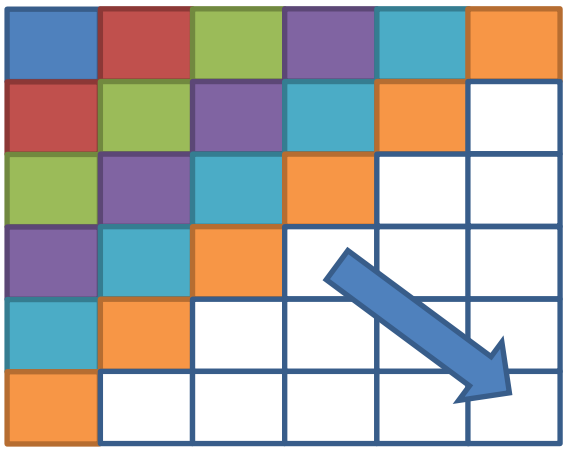
Na to pitanje teško je odgovoriti. Danas još uvijek postoji puno aplikacija koje centralni procesor može puno brže i bolje napraviti od grafičkog, dosta poslova nije ni moguće izvesti na grafičkom, ali oni poslovi koji se mogu prilagoditi za rad na grafičkim procesorima nerijetko pokazuju performanse 10 do 1000 puta bolje od onih na centralnim procesorima.

4. Implementacija

Huang-Millerov algoritam kao i svi ostali algoritmi temeljeni na dinamičkom programiranju, nije baš jednostavan za paralelizirati zbog svoje ovisnosti o podatcima izračunatim u prethodnim koracima. Ipak, detaljnijom analizom algoritma moguće je uočiti dijelove koji se mogu istovremeno izvoditi.

4.1. Osnovna implementacija

Osnovna ideja paralizacije je ubrzati one dijelove algoritma koji imaju najduže vrijeme izvođenja. U ovom slučaju radi se o inicijalnom prolasku kroz matrice pri kojem se pronalazi K najboljih klasi ekvivalencije, te ponovnim manjim prolascima prilikom kojih se traže potencijalno dobri putovi koji nisu uočeni prilikom inicijalnog prolaska. [17] Kao što je već opisano u poglavlju 2.6., proračun svake ćelije matrice ovisi o vrijednosti ćelije neposredno lijevo, iznad i sjeverozapadno po dijagonali. Ako se pogleda izvođenje algoritma malo iz daljega, može se uočiti da postoje elementi čije se vrijednosti mogu istovremeno računati. Ti elementi posloženi su po sporednoj dijagonali matrice, te predstavljaju jedine kandidate za paralelno izvođenje. Na slici 14 prikazana je ideja provođenja paralelizacije algoritma.



Slika 14 - Ideja izvođenja algoritma

Osnovna ideja je svakoj ćeliji po sporednoj dijagonali pridijeliti jednu dretvu koja će joj izračunat vrijednost. Kod takve paralelizacije javljaju se dva problema.

Prvi problem je provođenje sinkronizacije, odnosno osiguravanje da sve dretve izračunaju svoj element prije nego krenu računat novi red. Taj uvjet je nužan kako bi se osigurala ispravnost svih izračunatih rezultata. Osiguravanje zadovoljavanja uvjeta ostvareno je na način da u jednom pozivu jezgre sve dretve računaju samo jedan red. Takav način izvođenja ima i dobre i loše efekte na performanse. Loša posljedica je nemogućnost korištenja dijeljene memorije, koja ima znatno kraće vrijeme odziva od globalne, jer se sadržaj dijeljene memorije gubi između uzastopnih poziva iste jezgrene funkcije. Također, prilikom svakog poziva jezgre gubi se određeno vrijeme. Iako se vremenski gubitak uslijed pozivanja mjeri u mikrosekundama [18], u slučaju velikog broja poziva to može potrajati. Dobra pak strana ove izvedbe je poprilično jednostavna izvedbe sinkronizacije u sustavima s više grafičkih procesora, što će biti detaljnije objašnjeno u sljedećem poglavlju.

Drugi problem je neiskorištenost svih dretvi na početku i kraju postupka. Naime, prilikom prvog poziva funkcije, samo prva dretva koja obrađuje gornji lijevi element ima sve podatke potrebne za izračun vrijednosti. U drugoj iteraciji će 2 dretve moći računat itd. Dakle, ako uzmemo da imamo N dretvi, tek nakon N poziva funkcije će sve pokrenute dretve moći raditi istovremeno. Ako uzmemo u obzir da je druga sekvenca duljine M (što odgovara broju redaka u matrici), bit će potrebno M + N poziva funkcije kako bi se izračunala cijela matrica. Broj poziva funkcije moguće je smanjiti i to će biti objašnjeno u sljedećem poglavlju.

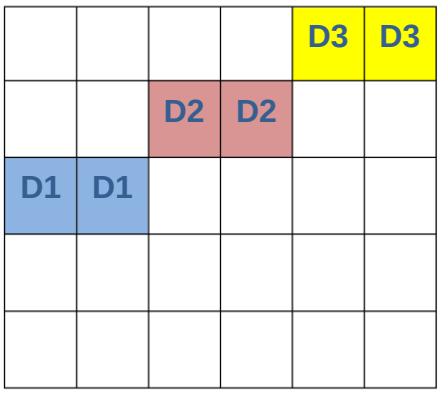
4.2. Optimizacije

Iako već i osnovna izvedba paralelizacije pruža znatno bolje performanse od sekvencijalne verzije, uvijek postoji prostor za poboljšanja. Kako bi se ostvarile bolje performanse isprobane su sljedeće optimizacije:

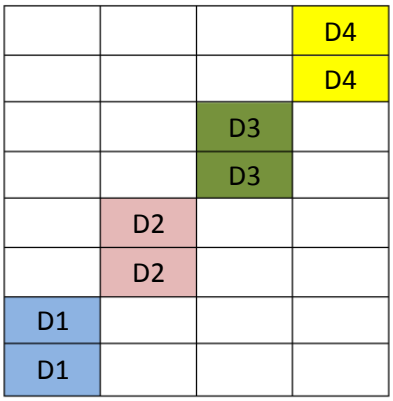
- Povećanje posla po dretvi
- Podjela posla na više grafičkih procesora
- Memorijska optimizacije

4.2.1. Povećanje posla

Prva ideja za poboljšanje performansi je povećati količinu posla po dretvi. Na taj način umanjio bi se gubitak vremena koji nastaje u početnoj i završnoj fazi algoritma gdje nisu sve dretve aktivne. Povećanje posla je moguće izvršiti na 2 načina: povećanjem broja stupaca koji dretva obrađuje i povećanjem broja redova koji se obrađuju u jednoj iteraciji algoritma. Na slikama 15 i 16 prikazane su ideje paralelizacije (ćelije obojane u istoj boji računaju se u jednoj iteraciji):



Slika 15 - Povećanje po stupcima



Slika 16 - Povećanje po redovima

Na malo iznenađenje, rezultati testiranja su pokazali da niti jedna verzija ne pruža značajno bolje performanse nego osnovna izvedba. Razlog za to kod verzije s povećanjem po stupcima je lošiji način pristupanja memoriji. Kod osnovne izvedbe sve dretve čitaju podatke na uzastopnim lokacijama, tako da je moguće u jednom ciklusu pročitati podatke za više dretvi. Kod verzije s većim brojem

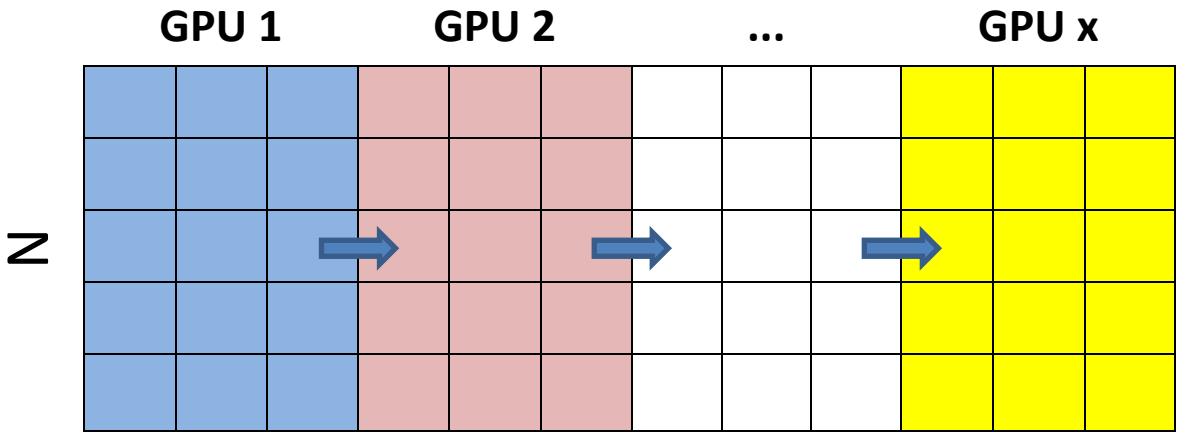
stupaca to nije slučaj, već postoje rupe u pristupanju. Konkretno na slici to znači da se u prvom koraku pristupaju podatcima prvog, trećeg i petog stupca, koji fizički u memoriji nisu spremljeni uzastopno. Kod verzije s obradom više redaka radi se o sličnom problemu. Iako se zbog obrade više redaka odjednom može smanjiti broj pristupa globalnoj memoriji kopirajući bitne podatke u registre i onda radeći s njima, isto tako je potrebno više podataka na početku pročitat iz globalne memorije tako da se gubi većina prednosti.

4.2.2. Podjela posla na više grafičkih kartica

S obzirom da se grafički procesori odlikuju masivno paraleliziranom arhitekturom, dodavanje dodatnih procesora koji će obrađivati podatke paralelno ne bi u načelu trebalo predstavljati neki posebni problem. U ovom slučaju problemi se javljaju zbog potrebe za razmjenom podataka među grafičkim procesorima. Kako jedan grafički procesor vidi samo podatke pohranjene u svojoj globalnoj memoriji, potrebno je provesti sinkronizaciju podataka pomoću centralnog procesora. Ta sinkronizacija se relativno jednostavno provodi zbog činjenice da se nakon obrade svakog reda prekida izvođenje funkcije, te se vraća kontrola centralnom procesoru. Ako pretpostavimo da imamo dvije sekvence duljina M i N i dva grafička procesora, tijek izvođenja bi bio sljedeći:

- Prvih M / 2 iteracija izvođenja radio bi prvi grafički procesor (faza propagiranja podataka dok sve dretve ne budu mogle raditi). Ne postoji potreba za prijenosom podataka među grafičkim procesorima
- Provodi se prva sinkronizacija podataka među grafičkim procesorima
- Sljedećih M / 2 iteracija sve dretve prvog grafičkog procesora su zaposlene, a na drugom grafičkom procesoru dolazi do propagiranja podataka. U svakoj iteraciji dolazi do sinkronizacije podataka
- Nakon N – M / 2 iteracija prvi grafički procesor je gotov s računanjem
- Nakon M / 2 iteracija završava i drugi grafički procesor s radom
- Završna sinkronizacija pronađenih najboljih klasi ekvivalencije

U načelu ovakva izvedba paralelizacije nema ograničenje na broj grafičkih procesora. Slika 17 prikazuje općeniti koncept podjele podataka.



Slika 17 - Multi GPU sustav

4.2.3. Memorijske optimizacije

Kao što je već uvodu u poglavlje rečeno, zbog načina na koji se provodi sinkronizacija podataka, nije moguće koristiti dijeljenu memoriju. To nije baš poželjno, jer dijeljena memorija pruža velike brzine pristupa uz dovoljan kapacitet. Kako bi se što bolje nadoknadio ovaj nedostatak kao i zaobišla ograničenja globalne memorije, korišteni su cjelobrojni tipovi podataka veličine 4 okteta. Takav pristup omogućava maksimalno iskorištavanje dostupnih registara grafičkog procesor, a da se pritom ne smanjuje broj dretvi koje se mogu istovremeno izvoditi. Također, može se identificirati dio podataka koji sve dretve učestalo isključivo čitaju. Radi se o supstitucijskim matricama, te su iste smještene u memoriju za konstante. Sve navedene optimizacije rezultirale su smanjenjem vremena izvođenja za 30%.

4.3. Programska implementacija

Ostvarenu programsku implementaciju možemo ugrubo podijeliti u dvije kategorije; kod koji implementira sekvencijalne algoritme i programski kod koji implementira paralelne algoritme. Sekvencijalni kod pisan je u C programskom jeziku. U sljedećoj tablici navedene su datoteke, najbitnije funkcije za svaku i kratki opis šta je implementirano u svakoj.

Tablica 1 - Struktura sekvencijalnog koda

Datoteka	Funkcije	Implementacija
Main.c	-	Osnovna logika
backPass.c	locate	Algoritam za određivanje granica područja na kojem će se ponovno provest ocjenjivanje
fwdPass.c	fwdPassCPU, smallFwdPassCPU	Implementacija Smith Waterman algoritma. Provodi se ocjenjivanje.
Hisrch.c	globalAlign	Implementacija algoritma za globalno poravnavanje.
LIST.c	addnode	Implementacija LIST strukture podataka u koju se pohranjuju najbolje klase ekvivalencije.
Lib.c	loadSeq, displaySeq	Implementacija općenitih ulazno / izlaznih funkcija
Globals.c	-	Globalno dostupne strukture poput supstitucijske matrice i liste zabranjenih poteza

Većina algoritama je implementirana tako da se koriste samo polja jednostavnih tipova podataka. Jedini izuzetak je datoteka LIST.c koja implementira operacije nad LIST strukturom u koju se pohranjuju najbolje klase ekvivalencije. LIST struktura ostvarena je kao polje sljedeće strukture.

```
typedef struct NODE
{
        int   SCORE;                    // score
        int   STARI;                    // First(i,j)
        int   STARJ;                    // First(i,j)
        int   ENDI;                     // end point
        int   ENDJ;                     // end point
        int   TOP;                      // top boundary
        int   BOT;                      // bottom boundary
        int   LEFT;                     // left boundary
        int   RIGHT;                    // right boundary
}  eqClass
```

Struktura eqClass sadrži sve članove koji opisuju jednu klase ekvivalencije, a koji su detaljnije objašnjeni u poglavlju 2.6.1.

Programski kod koji implementira paralelizirane verzije algoritama napisani su u CUDA tehnologiji o kojoj je više rečeno u poglavlju 3. Programska implementacija podijeljena je na sljedeće datoteke.

Tablica 2 - Struktura paraleliziranog koda

Datoteka	Funkcije	Implementacija
Gpu.cu	forwardPassGPU, smallForwardPassGPU	Prepoznaje se dostupna hardverska konfiguracija, te pozivaju prikladne funkcije
fwdPassGPU_sg.cu	forwardPassSingleGPU	Implementacija paralelizirane verzije algoritma ocjenjivanja na jednom grafičkom procesoru
fwdPassGPU_mg.cu	forwardPassMultiGPU	Implementacija paralelizirane verzije algoritma ocjenjivanja na više grafičkih procesora
smallFwdPassGPU.cu	smallFwdPassGPU	Implementacija paralelizirane verzije algoritma ocjenjivanja na ograničenom području.

Osnovna izvedba na jednom grafičkom procesoru koristi samo jednu posebnu strukturu podataka:

```
typedef struct
{
      int i;                          // current row to calculate
      short ready;                    // data available flag
}threadStatus;
```

Svaka dretva koja se izvodi na grafičkom procesoru ima po jednu takvu strukturu. U strukturi je pohranjen indeks reda koji se treba obraditi, te zastavica koja određuje može li dretva krenut s izvođenjem. Izvođenje dretve uvjetovano je dostupnošću potrebnih podataka. Po završetku izvođenja dretve koja obrađuje stupac X, postaju dostupni svi podatci potrebni za sljedeću dretvu X+1. Ta dostupnost signalizira se tako da dretva X na kraju izvođenja zapiše 1 u ready

zastavicu dretve X+1. Na kraju izvođenja dretva X također inkrementira vrijednost varijable *i* u strukturi.

Izvedba na više grafičkih procesora je ponešto kompliciranija jer je potrebno izvesti sinkronizaciju podataka između grafičkih procesora, te između dretvi centralnog procesora. Svaki grafički procesor koji će se koristiti zahtjeva zasebnu dretvu centralnog procesora. S obzirom da različiti grafički procesori ne mogu komunicirati direktno, komunikacija se izvodi tako da se prvo podatci prenesu s grafičkih procesora u radnu memoriju. Nakon toga se ti podatci izmijene među dretvama centralnog procesora, te se na kraju novi podatci šalju grafičkim procesorima.

Kako bi se što jednostavnije i efikasnije prenijeli svi potrebni podatci, uvedene su sljedeće dvije strukture:

```
typedef struct
{
	int c;				// score at (i,j)
	int ci;				// First(i,j)
	int cj;				// First(i,j)
	int f;				// score at (i-1,j)
	int fi;				// First(i-1,j)
	int fj;				// First(i-1,j)
	int p;				// score at (i-1,j-1)
	int pi;				// First(i-1,j-1)
	int pj;				// First(i-1,j-1)
}interThreadData;
```

```
typedef struct
{
	int deviceID;			// device ID to run kernels on
	int M;				// first sequence length
	int N;				// second sequence length
	char *A;			// first sequence
	char *B;			// second sequence
	int o;				// opening gap penalty
	int r;				// extending gap penalty
	int qr;				// o+r
	int len;			// sequence length
	short *ready;			// ready to compute flag
	interThreadData *iData;		// sync data for host threads
	eClass *classes;		// computed equivalence classes
	int k;				// number of classes
}gpuData;
```

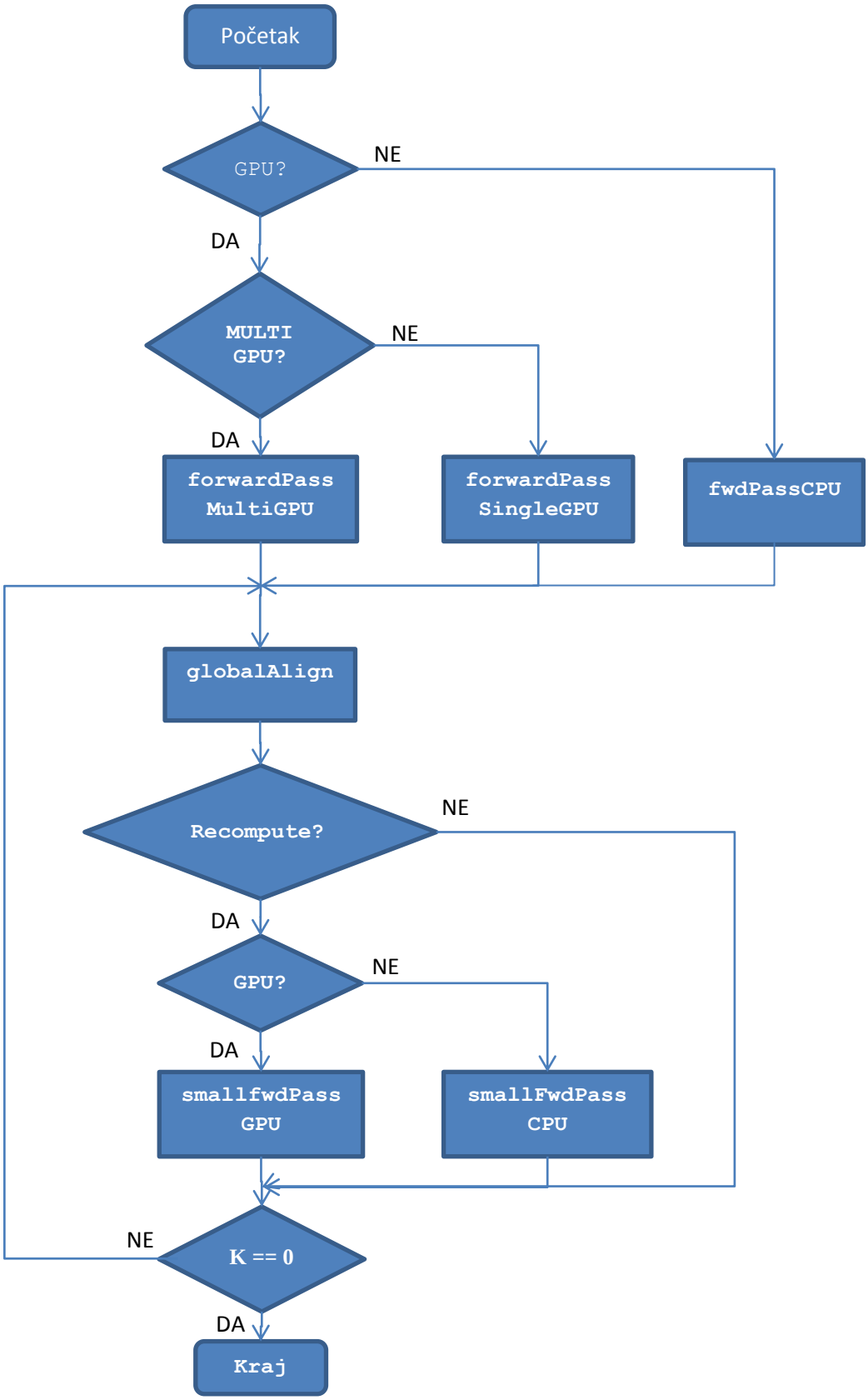
InterThreadData struktura sadrži rubne podatke koji su potrebni sljedećem grafičkom procesoru kako bi mogao krenuti s izvođenjem. Ako na raspolaganju imamo 2 grafička procesora, navedenu strukturu će popuniti dretva koja obrađuje zadnji stupac na prvom grafičkom procesoru, a pročitati će prva dretva drugog grafičkog procesora. Taj postupak ponoviti će se M puta, odnosno onoliko puta koliko imamo redaka u matrici koja se ocjenjuje.

Prilikom stvaranja novih dretvi na centralnom procesoru, svakoj dretvi se predaje funkcija koja će se izvoditi, te pokazivač na strukturu koja treba sadržavati sve podatke potrebne novo stvorenoj dretvi. Tu se koristi druga struktura, gpuData sljedećeg sadržaja:

- **deviceID** varijabla označava s kojim grafičkim procesorom će dretva komunicirati
- **M** i **N** varijable predstavljaju duljine ulaznih sekvenci
- **A** i **B** su pokazivači na polja ulaznih sekvenci
- Varijable **o**, **r** i **qr** predstavljaju kazne za otvaranje procjepa, jedinično proširenje te za otvaranje procjepa duljine 2
- Varijabla **len** označava broj stupaca koje grafički procesor treba obraditi
- Polje **ready** je polje zastavica koje određuju može li dretva pozvati jezgrenu funkciju ovisno o tome je li primila sve podatke od susjednog grafičkog procesora (interThreadData)
- Pokazivač **iData** na interThreadData strukturu koja je već opisana
- Polje **classes** u koje će biti pohranjene sve pronađene klase ekvivalencije od strane grafičkog procesora
- Varijabla **k** koja označava traženi broj najboljih klasi ekvivalencije

Kada svi grafički procesori završe s radom, sve dretve centralnog procesora osim početne se gase. Potom početna dretva prolazi kroz sve pronađene klase ekvivalencije i pokušava ih spremiti u LIST strukturu pozivajući **addNode** funkciju. Po završetku u LIST strukturi će ostati samo K najboljih pronađenih klasi.

Kompletno izvođenje programskog koda prikazano je na sljedećem dijagramu toka (slika 18).



Slika 18 - Dijagram toka aplikacije

5. Rezultati

5.1. Konfiguracija testiranja

Kako bi se ocijenile performanse ostvarenog programskog rješenja, testiranje će bit izvršeno nad sekvencama različitih duljina i s različitim brojem najboljih poravnavanja. Ovakva konfiguracija testova omogućuje nam ne samo usporedbu performansi izvođenja algoritma na centralnom i grafičkom procesoru, već i uvid kako ulazni parametri algoritma utječu na vrijeme izvođenja.

Za potrebe testiranja odabrane su sekvence sljedećih duljina: 10k, 50k, 100k , 250k i 500k. Testiranje utjecaja broja pronađenih najboljih poravnavanja zbog ograničenog vremena testiranja bilo je ograničeno na samo 3 vrijednosti: 10, 25 i 100. Sve sekvence korištene za testiranje nasumično su generirane.

Što se tiče korištenog sklopovlja, testovi programskog koda sekvencijalne izvedbe provedeni su na Intelovom dvojezgrenom centralnom procesoru E6300, kojem je u ovom slučaju podignut osnovni takt na 3,8 GHz. Za potrebe testiranja paraleliziranog koda i koda za više grafičkih procesora korištena je grafička kartica GTX 590 koja se sastoji od 2 grafička procesora. Svaki procesor raspolaže sa 512 jezgri koje rade na 1,2 GHz.

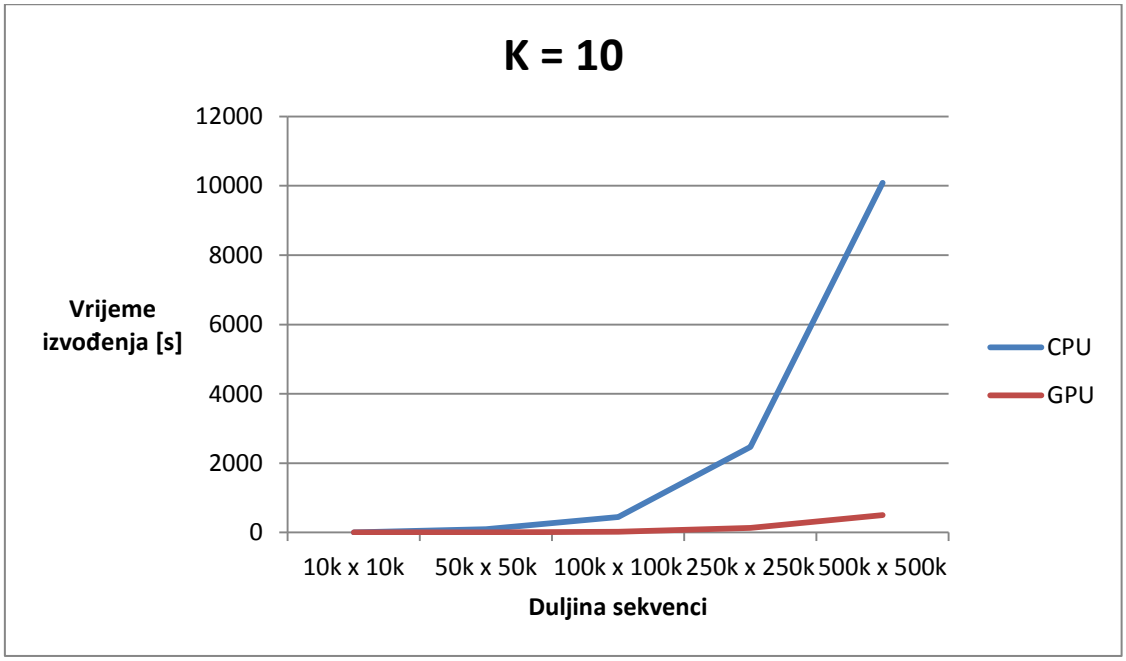
Prilikom testiranja osim brzine izvođenja zabilježeni su i memorijski zahtjevi. Na kraju testiranja provjerena je i osjetljivost performansi algoritma na parametre ocjenjivanja poput kazne za otvaranje procjepa. Iz opisa algoritma navedeni parametri u slučaju preblagog kažnjavanja bi trebali imati popriličan utjecaj na performanse.

```
CTNNNTNANTCNCGTAGTACCANNCNTGGACAATCNTGACAGTTGCNCAN
GCNNTTANGNTNCTGTGAAGCCNAATTNCTACCTAGNAANCTATCNACNTT
GACGAGACCGTTCNNNATNAGGAGCCAGCTACACNTCTNGTGTGNTTCNT
GNTNTNTTTGTNCTACNGGGGTGCATATCCCNANGAAGTNGCGAGACGGC
GNGNTGCTTTACTATTTTTACCGCCTCNACAAAANTAGCTTGGCNACGNGA
....
```

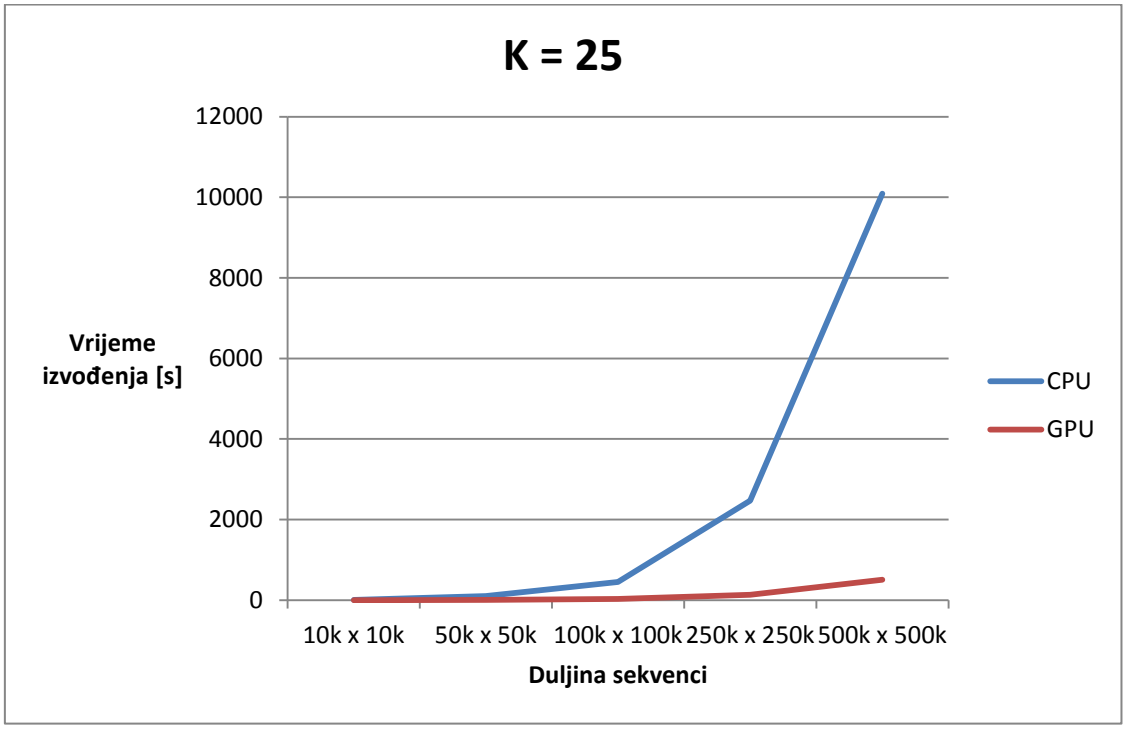
Slika 19 - Primjer testirane sekvence

5.2. Rezultati testiranja

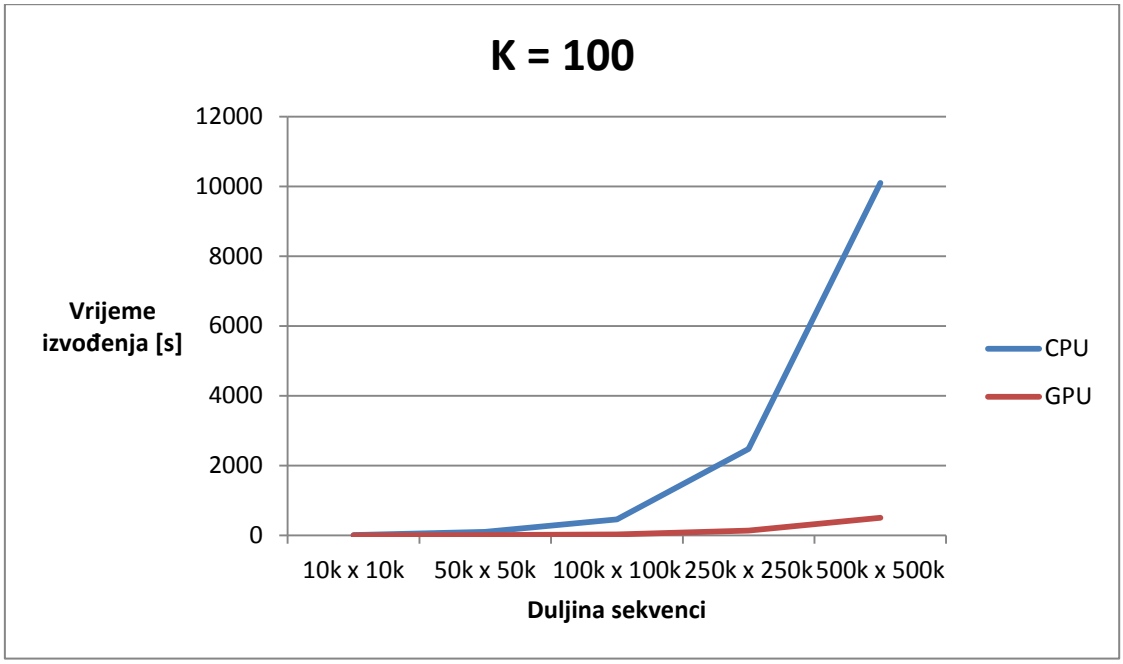
Na sljedeće 3 slike (za K = 10, 25 i 100) prikazani su dobiveni rezultati testiranja.



Slika 20 - Rezultati testiranja za 10 najboljih poravnavanja

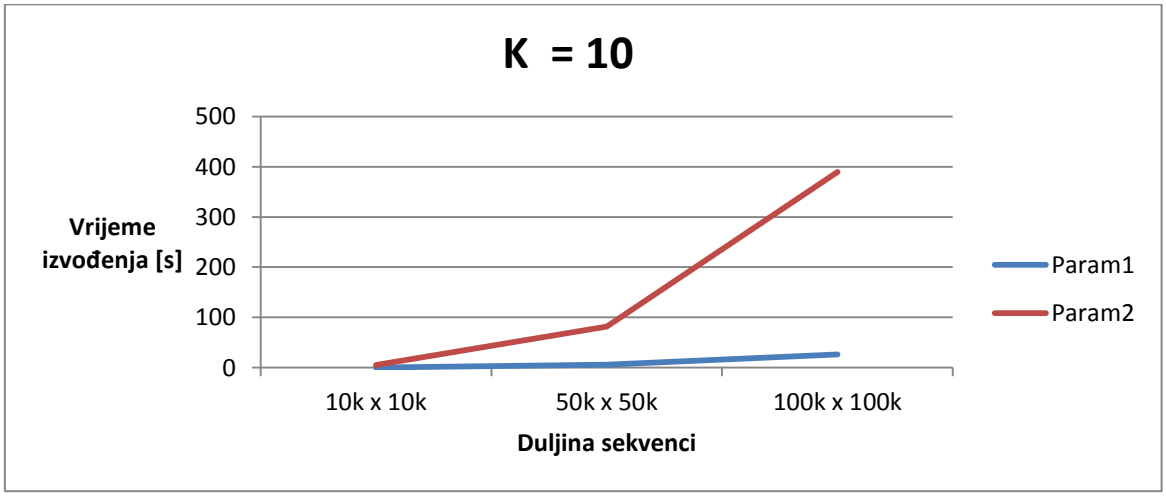


Slika 21 - Rezultati testiranja za 25 najbolji poravnavanja



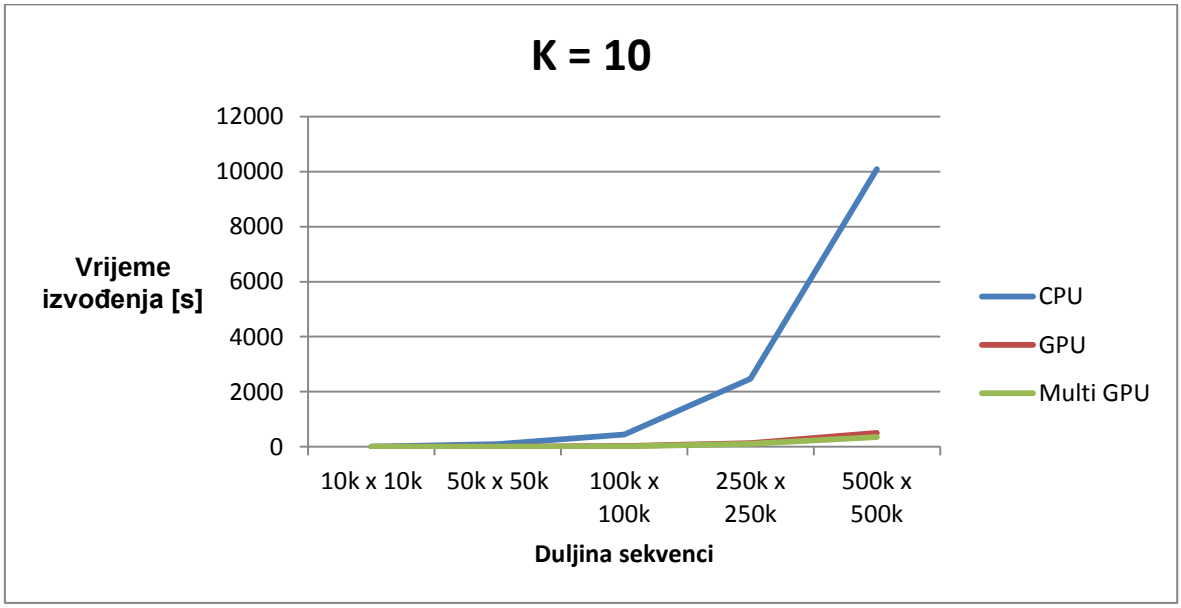
Slika 22 - Rezultati testiranja za 100 najboljih poravnavanja

Iz priloženih rezultata testiranja može se uvidjeti da paralelizirana izvedba na grafičkom procesoru pruža otprilike 20 puta bolje performanse nego sekvencijalna verzija. Navedeni rezultati su otprilike u skladu s očekivanjima, posebice ako se uzme u obzir da u paraleliziranoj izvedbi postoji dio koji se sekvencijalno izvodi. Također, iz rezultata se može primijetit da ne postoji neka velika razlika u vremenu izvođenja u ovisnosti o broju traženih poravnavanja. Razlog tome su visoko postavljene kazne za otvaranje procjepa, zbog čega je većina pronađenih poravnavanja relativno kratka u odnosu na ulazne sekvence. Smanjujući kazne za otvaranje i produživanje procjepa postignuti su sljedeći rezultati (zbog ograničenog vremena testiranje je provedeno samo na paraleliziranoj izvedbi za K = 10).

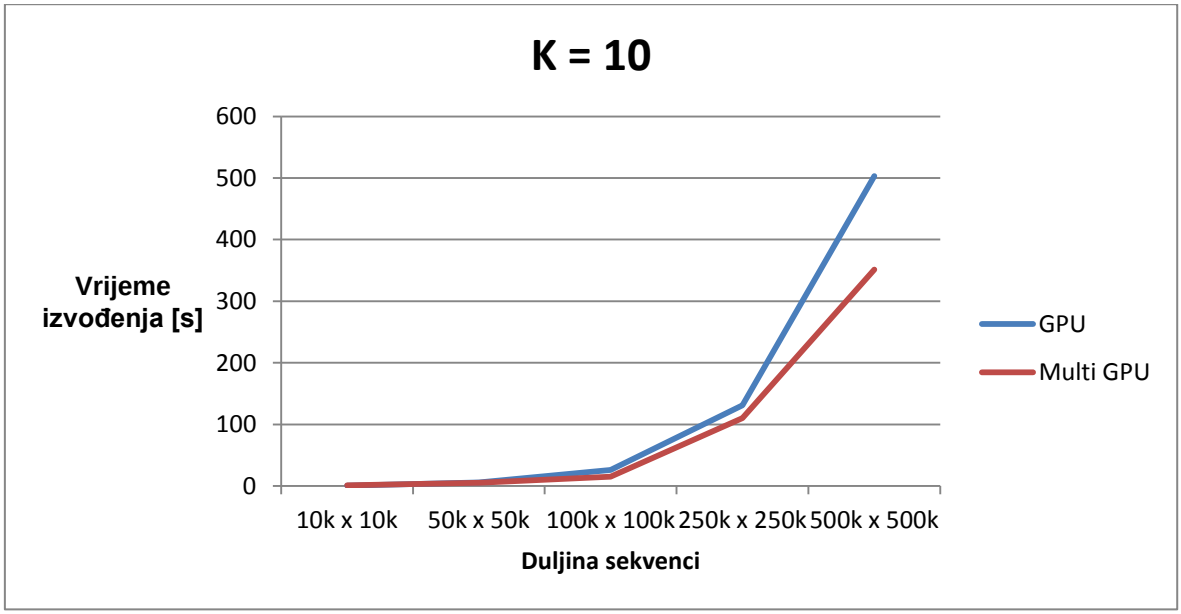


Slika 23 - Usporedba vremena izvođenja u ovisnosti o parametrima

Od provedenih optimizacija najveće ubrzanje postiglo se prilagodbom za izvođenje na više grafičkih procesora. Na slici 24 prikazana je usporedba brzine izvođenja s centralnim procesorom i osnovnom izvedbom na grafičkom procesoru. Iz grafa se vidi da se postižu oko 30 puta bolje performanse u odnosu na izvedbu na centralnom procesoru, što predstavlja značajno ubrzanje. Na slici 25 prikazan je odnos s osnovnom izvedbom na jednom grafičkom procesoru. Ostvareno je ubrzanje od 1.5 puta s trendom rasta s porastom dužina ulaznih sekvenci.



Slika 24 - Rezultati testiranja za 10 najboljih poravnavanja



Slika 25 - Usporedba vremena izvođenja u slučaju korištenja jednog i više grafičkih procesora

U sljedećoj tablici dane su numeričke vrijednosti svih testova kako bi se dobio bolji uvid u rezultate:

Tablica 3 - Rezultati testiranja

Duljina sekvenci	K	CPU [s]	GPU [s]	MultiGPU [s]	Memorija [MB]	GPU s parametrima
10k x 10k	10	5,594	0,409	0,851	0,86	5,157
	25	5,918	0,534	0,863		-
	100	6,008	0,788	0,944		-
50k x 50k	10	97,7	5,734	5,266	3,00	82,046
	25	98,374	5,991	5,321		-
	100	98,746	6,133	5,535		-
100k x 100k	10	449,02	26,207	14,976	5,80	389,67
	25	450,23	26,447	15,109		-
	100	453,11	26,771	15,333		-
250k x 250k	10	2469,11	130,993	109,924	14,1	-
	25	2471,74	131,265	110,322		
	100	2474,787	131,627	110,936		
500k x 500k	10	10083,23	503,063	351,339	27,9	-
	25	10087,19	503,233	351,998		
	100	10098,44	503,579	353,621		

6. Zaključak

U ovome radu ukratko je opisano područje bioinformatike, te konkretnije problematika poravnavanja sekvenci. Opisano je par osnovnih i najvažnijih algoritama za poravnavanje, a potom i poseban algoritam koji se odlikuje linearnom prostornom složenošću i prosječno kvadratnom vremenskom. Nakon toga je opisana ukratko arhitektura modernijih grafičkih procesora, te programski model CUDA koji je korišten pri implementaciji paralelizirane verzije. Sami detalji i ideje oko implementacije detaljnije su objašnjene, kao i provedene optimizacije.

Iz dobivenih rezultata vidljivo je da se isplati provesti paralelizaciju na grafičkim procesorima. Prosječno ostvareno ubrzanje po svim testovima se kretalo oko 20 puta u odnosu na sekvencijalnu inačicu, odnosno oko 30 puta u slučaju korištenja više grafičkih procesora. Korištenjem novijih kartica više klase realno se mogu očekivati i ubrzanja do 50 puta. Sa takvim ubrzanjem smanjuje se potreba za heurističkim algoritmima poput BLAST-a koji ipak ne garantiraju točnost rezultata.

7. Literatura

[1] V. Likić*, The Needleman-Wunsch algorithm for sequence alignment,* www.ludwig.edu.au/course/lectures2005/likic.pdf, 30.4.2012

[2] E. Ayguade, J. Navarro, D. Gonzalez*, Smith-Waterman Algorithm,* http://docencia.ac.upc.edu/master/AMPP/slides/ampp_sw_presentation.pdf, 30.4.2012

[3] S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman, *Basic local alignment search tool*

[4] *BLOSUM62,* http://birec.org/sandbox/sites/default/files/images/BLOSUM62.JPG, 26.5.2012

[5] Lubica Benuskova*, Sequence Alignment,* http://www.cs.otago.ac.nz/cosc348/alignments/Lecture05-06_Alignment.pdf, 30.4.2012

[6] D. S. Hirschberg, *A linear space algorithm for computing maximal common subsequences*

[7] X. Huang, R. C. Hardison, W. Miller, *A space-efficient algorithm for local similarities*

[8] E. W. Myers, W. Miller, *Optimal alignments in linear space*

[9] X. Huang, W. Miller*, A Time-Efficient, Linear Space Local Similarity Algorithm*

[10] *NVIDIA GeForce 8800 GTX/GTS Tech Report*, http://www.rojakpot.com/showarticle.aspx?artno=358&pgno=1 , 24.5.2012

[11] Kirk, D.B., Hwu, W.W..*Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, USA, 2010.

[12] Sanders J., Kandrot E., CUDA By Example

[13] Song, Y., *Nvidia Graphics Processingg Unit (GPU)*, 9.9.2009. http://www2.engr.arizona.edu/~yangsong/gpu.htm , 24.5.2012.

.

[14] Yeo, K., *CUDA from NVIDIA – Turbo-Charging High Performance Computing*, 23.1.2009.

http://www.hardwarezone.com/articles/view.php?cid=3&id=2793, 20.5.2012.

[15] *Future of computing: GPGPU?*

http://gridtalk-project.blogspot.com/2010/07/future-of-computing-gpgpu.html, 25.5.2012

[16] Parrish, K., *Nvidia Predicts 570X GPU Performance Increase*¸ 26.8. 2009.,

http://www.tomshardware.com/news/Nvidia-GPU-Huang-570x,8544.html,

20.5.2012.

[17] X. Huang, W. Miler, S. Schwartz, R. Hardison*, Parallelization of a local similarity algorithm*

[18] *CUDA kernel overhead*

http://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html, 20.5.2012

8. Sažetak / Abstract

U ovome radu ukratko je opisano područje bioinformatike te algoritmi za poravnavanje sekvenci koji imaju vrlo bitnu ulogu u bioinformatici. Kako su pri implementaciji paralelizirane verzije programskog rješenja korišteni grafički procesori, također je opisana arhitektura modernih grafičkih procesora. Sama implementacija je detaljnije opisana, kao i provedene optimizacije. Ostvareni rezultati pokazuju ubrzanje od 20 puta u odnosu na sekvencijalnu verziju, što je u skladu s očekivanjima.

Ključne riječi: bioinformatika, Smith-Waterman, Needleman-Wunsch, CUDA, poravnavanje sekvenci

This paper briefly describes the area of bioinformatics and sequence alignment algorithms which play a major role in bioinformatics. Since graphics processing units have been used to run parallel version of the algorithm, this paper also describes the architecture of a modern graphics processing unit. The implementation is described in more details as well as the performed optimizations. Achieved results show acceleration of 20 times in comparison to sequential version, which is in line with the expectations.

Keywords: bioinformatics, Smith-Waterman, Needleman-Wunsch, CUDA, sequence alignment