Matematički fakultet / Univerzitet u Beogradu

SEMINARSKI RAD

Predmet: Razvoj softvera – napredni koncepti
predmetni nastavnik: doc dr Saša Malkov

IMPLEMENTACIJA PARALELNOG GENETSKOG ALGORITMA U DELJENOJ CUDA MEMORIJI

Aleksandar KARTELJ
Matematički fakultet,
Univerzitet u Beogradu, Srbija
kartelj@matf.bg.ac.rs

U Beogradu, 2.7.2011

SADRŽAJ

SADRŽAJ 2
O DOKUMENTU 3
1. CUDA 4
a. Istorijat 4
b. Arhitektura 4
c. Kerneli, niti, memorija 5
d. "Zdravo CUDA!" 7
2. GA UOPŠTENO 8
a. Istorijat i osnovni GA 8
b. Kodiranje 8
c. Funkcija prilagođenosti 9
d. Operator selekcije 9
e. Operator ukrštanja 9
f. Operator mutacije 10
g. Kriterijum zaustavljanja 10
3. GA IMPLEMENTACIJA 11
a. Implementacija sekvencijalnog algoritma 12
b. Kodiranje (implementacija) 13
c. Funkcija prilagođenosti (implementacija) 14
d. Operator selekcije (implementacija) 14
e. Operator ukrštanja (implementacija) 14
f. Operator mutacije (implementacija) 15
g. Kriterijum zaustavljanja (implementacija) 15
4. CUDA+GA 16
a. Paralelizacija funkcije cilja 17
b. Paralelizacija ostalih funkcija 18
c. Uvođenje deljene memorije – revizija koda 19
d. Reorganizacija – smanjivanje broja pristupa globalnoj memoriji 20
e. Redukovanje memorijskih zahteva 22
5. EKSPERIMENTALNI REZULTATI 24
a. Instance problema i parametri aplikacije 24
b. Rezultati 24
6. ZAKLJUČAK 28
a. Budući rad 28
7. DODATAK 29
8. KORIŠĆENA LITERATURA 29

O DOKUMENTU

U ovom dokumentu opisana je implementacija genetskog algoritma (GA) na paralelnoj CUDA platformi. CUDA predstavlja relativno novu tehnologiju, koja koristi neke komparativne prednosti GPU (grafičkih procesora) u odnosu na klasične CPU. Najpre je opisana sekvencijalna varijanta algoritma, koja je potom transformisana na paralelnu arhitekturu. Transformacija je izvršena sa namerom da sve relevantne informacije potrebne za izvršavanje genetskog algoritma stanu u deljenu memoriju (eng. shared memory) CUDA podržane grafičke kartice. Ovo je doprinelo određenom ubrzanju kada je u pitanju vreme izvršavanja algoritma. Ipak, nivo podizanja performansi nije velik, a deo krivice se svakako može pripisati i činjenici da je korišćena grafička karta Nvidia 8400 GS jedna od „pionirskih" kada su u pitanju CUDA standardi. Stoga, verujem da bi implementacija na nekoj od novijih grafičkih kartica više doprinela performantnosti sistema. Drugi razlog leži u tome da je transformacija zamišljena tako da paralelni algoritam konceptualno zadrži sva svojstva polaznog, a da se posle u nekoj narednoj iteraciji ova implementacija iskoristi kao gradivni blok npr. „ostrvskog" modela GA, koji se razlikuje od klasičnog algoritma. Konačno, ključni razlog je to što je razmatrani problem imao „previše" jednostavnu funkciju cilja, tako da je dobit od paralelizovanog načina rada bila potisnuta troškovima kreiranja niti. Međutim, korišćenjem veštački otežane funkcije cilja, pokazano je značajno ubrzanje, koje sugeriše da ne treba vršiti paralelizaciju po svaku cenu, već je potrebno napraviti dobar kompromis. Jedna od smernica je dakle, paralelizovati ukoliko je funkcija cilja iole kompleksnija i kompenzuje troškove kreiranja niti. Srećna okolnost implementiranja na „malo starijoj" Nvidia 8400 GS kartici je da je uspela da natera autora da pažljivije „kodira", i da na teži, ali verovatno kvalitetniji način shvati neke od principa programiranja na GPU. U tekstu će se ponekad pričati o nekim na prvi pogled banalnim zapažanjima, ipak pokušaću da pokažem da i najmanje sitnice mogu biti od važnosti kada je u pitanju rad sa ovako osetljivim API-jem. Mislim da ovaj dokument može potencijalno da bude koristan osobama koje tek počinju da se bave CUDA programiranjem iz dva razloga: 1) naglašavaću situacije u kojima sam nailazio na probleme i davaću opise kako se rešavaju (uglavnom, znanje prikupljeno na forumima); 2) pokušaću kroz način izlaganja da provučem pre svega generalnu ideju i način razmišljanja s obzirom da je CUDA programiranje još uvek prilično „trusno" područje, pa nema smisla previše ulaziti u tehničke aspekte.

Tekst je podeljen na sledeće celine:

1. CUDA – poglavlje sadrži neke informacije o toku razvoja, kratki opis arhitekture, programskog modela, memorije i drugih mehanizama. Poglavlje zaključujemo sa uvodnim „Zdravo CUDA!". U ovom delu će biti dat i opis hardverske specifikacije opisane CUDA standardom (u ovom slučaju opis Nvidia 8400 GS).
2. GA UOPŠTENO – u ovom delu dat je opis prostog genetskog algoritma i genetskih operatora.
3. GA IMPLEMENTACIJA – na početku je izložen problem koji rešavamo, a zatim je dat sažet opis implementacije genetskog algoritma (po tačkama iz **2.**) na programskom jeziku C, sa pratećim fragmentima bitnijih delova koda.
4. CUDA+GA – opisana je implementacija GA na CUDA. Dati su detaljniji opisi i razmatranja vezana za transformaciju osnovnog GA. Izvršena je iscrpnija analiza fragmenata koda, posebno na mestima gde je performantnost od velikog značaja. Najveći naglasak je stavljen na korišćenje deljene memorije.
5. EKSPERIMENTALNI REZULTATI – analiza toga što je urađeno.
6. ZAKLJUČAK – završna reč i budući pravci razvoja.
7. DODATAK – uputstvo za korišćenje programa.
8. KORIŠĆENA LITERATURA

1. CUDA

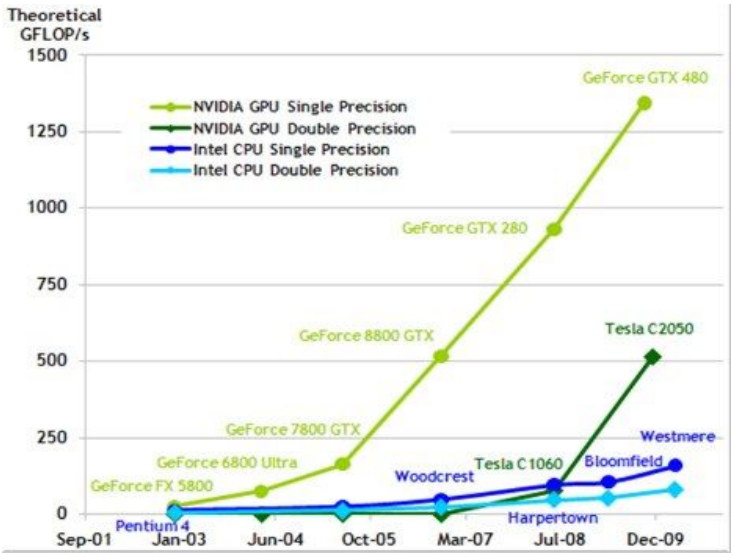
a. Istorijat

CUDA (eng. Compute Unified Device Architecture) je arhitektura za paralelna izračunavanja, razvijena od strane Nvidia-e. Ova arhitektura koristi grafički procesor GPU (eng. Graphics Processing Unit), koji ima određene komparativne prednosti u odnosu na klasične CPU (eng. Central Processing Unit). CUDA predstavlja i niz standarda, a sa programerskog stanovištva, svakako je najbitniji programski model koji nudi ekstenzije za rad sa C i C++ programskim jezicima. Postoje i ne-oficijelni (eng. third party) paketi za podršku sledećim jezicima: Python, Perl, Fortran, Java, Ruby, Lua, Matlab, IDL i neke ekstenzije za programski paket Mathematica. CUDA ima i konkurente u vidu OpenCL standarda, koji razvija Khronos Group, kao i Microsoft-ovog DirectCompute sistema.

CUDA nudi pristup virtuelnom skupu instrukcija i specifičnom memorijskom modelu koji je sastavni deo CUDA GPU elemenata za paralelna izračunavanja. Napori Nvidia-e u poslednjih nekoliko godina bili su pre svega usmereni ka podizanju nivoa abstrakcije u pogledu programiranja. Neki od ranijih standarda kao što je npr. OpenGL nisu imali za cilj da iskoriste GPU za namene generalnog programiranja, već su se ticali uske oblasti 2D i 3D grafike, razvoja igrica i slično. Međutim, određeni krugovi IT stručnjaka prepoznali su prednosti koje leže u srži grafičkih procesora za generalna izračunavanja. Nasuprot klasičnih procesorskih jedinica, koje imaju velike skupove instrukcija i centralizovani pristup, grafički procesori imaju mali skup primitivnih operacija, ali distributivno orijentisan model izračunavanja. Nvidia je razvojem svog CUDA standarda uspela da olakša razvoj aplikacija i ponudi ga široj zajednici programera.

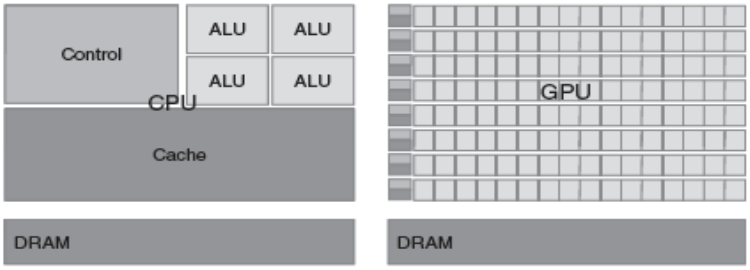
b. Arhitektura

Počev od 2003. godine razvoj mikroprocesora je zauzeo dva komplamentarna pravca. Multi-jezgarska linija razvoja (eng. Multicore processors) pokušava da proširi broj jezgara u okviru klasičnih CPU i na taj način omogući izvršavanje većeg broja sekvencijalnih programa. Trenutni lider u toj oblasti je Intel Core i7 procesor, koji ima 4 jezgra, a svako od njih ima podršku za puni x86 skup instrukcija. Sa druge strane, tu su mnogo-jezgarski (many-core) sistemi, koji su se fokusirali na poboljšavanje paralelnog izvršavanja. Ovakvi sistemi imaju veliki broj malih jezgara, a trenutni je trend da se taj broj udvostručava sa pojavom svake nove generacije. Dakle, za razliku od CPU, na ovom polju i dalje važi „Murov zakon“. Trenutni lider je Nvidia GeForce GTX 280, sa 240 jezgara, od kojih svako ima mogućnost višenitnog izvršavanja. Na Slici 1. je predstavljen komparativni dijagram razvoja CPU i GPU počev od 2003. godine.



Slika 1. Evolutivni razvoj GPU (U GFLOPS/s)

Jednostavnija jezgra nisu jedina „uslovna“ prednost GPU, organizacija memorije ima bitan uticaj na brzinu paralelnog izvršavanja. Na Slici 2. možemo videti da je CPU pristup memoriji vrlo diferenciran. I pored toga što postoji registarska i keš memorija, sam mehanizam pristupa nije „dovoljno“ distribuiran kao što je to slučaj kod GPU.



Slika 2. Razlika između CPU i GPU memorijske organizacije

Protok podataka (eng. bandwidth) između DRAM (eng. dynamic random access memory) i GPU kod GT200 kartice iznosi 150 GB/s, a bitno je napomenuti da se beleži konstantan trend rasta, kada je ovaj parametar u pitanju. Sa druge strane za CPU sisteme se ne predviđa rast preko 50 GB/s u naredne tri godine.

c. Kerneli, niti, memorija

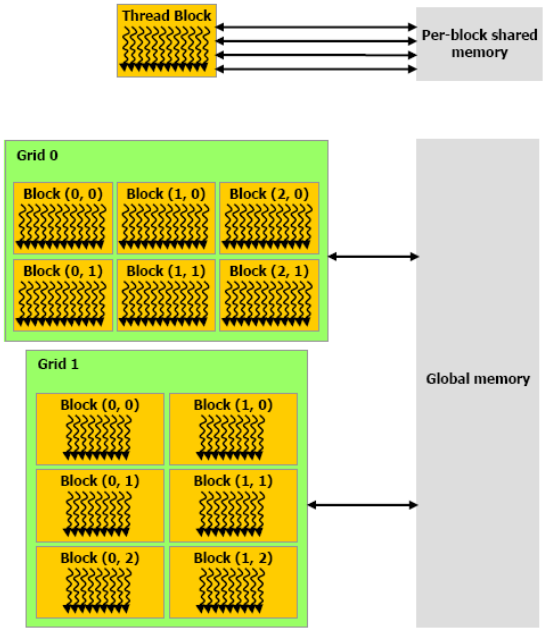
Kerneli su funkcije, koje bivaju izvršene od strane više niti. Pitanje se dakle postavlja gde je ta separacija na paralelni način izvršavanja. CUDA implementira tzv. SIMT (eng. Single Instruction Multiple Thread) model, što je u suštini SIMD (eng. Single Instruction Multiple Data). Dakle, podela „posla“ se ne vrši na nivou obavljanja različitih vrsta aktivnosti, već na nivou različitog skupa ulaznih podataka. Niti su u ovom modelu izuzetno jednostavne i ne treba ih poistovećivati sa CPU nitima. Vreme potrebno za njihovo je kreiranje je kratko, a efikasan način kontrole niti omogućava brzo prebacivanje sa jedne na drugu.

Terminološka opaska.
Device ⇔ GPU
Host ⇔ CPU
ID⇔ Jedinstvena oznaka

Kernel izvršava niz niti, pri čemu se svaka od njih karakteriše svojim jedinstvenim ID-em. Niti među sobom mogu da sarađuju samo indirektno, korišćenjem neke od dozvoljenih memorija na GPU:

- Globalna (kvalifikator `__device__`) je najveća i ima najsporije vreme pristupa. Može joj se pristupati sa GPU i CPU u bilo kom momentu tokom izvršavanja aplikacije.
- Deljena memorija (nalazi se na čipu, kvalifikator `__shared__`) je izuzetno brza memorija, oko 150 brža od globalne. Međutim, ograničene je količine, svega 16KB po jednoj multi-procesorskoj jedinici. (8400 GS ima jednu multi-procesorsku jedinicu). Vidljiva je u okviru određene grupe niti, koja se zove blok.
- Registarska memorija (nalazi se na čipu) je barem toliko brza, koliko i deljena. Ima je manje, a vidljiva je samo niti u kojoj je kreirana. Nema poseban kfalifikator, jednostavno je kreirana u okviru tela GPU funkcije.
- Konstantna memorija je sporija od deljene i registarske, ali ne mnogo. Nalazi se u globalnoj memoriji, ali se kešira na čipu, tako da je dosta brža od globalne (kvalifikator `__constant__`)

Niti se grupišu u blokove niti, na 8400 GS maksimalna veličina bloka je 512 niti. Prednost korišćenja niti u istom bloku je u tome što sve mogu da vide deljenu memoriju. Stoga je ovaj parameter grafičke kartice od izuzetnog značaja, posebno ukoliko je potrebno praviti visoko kooperativnu aplikaciju. Način grupisanja može biti bilo jedno-, dvo- ili trodimenzionalan (po x, y i z osi), jedino je bitno je da ukupan broj iskorišćenih jedinica ne premašuje maksimalan broj. Blokovi se dalje grupišu (jedno- ili dvodimenziono) u tzv. rešetke (eng. grids), čiji redosled izvršavanja nije deterministički sa aspekta programiranja. Ovakvu organizaciju je povoljno koristiti za relativno nezavisne elemente aplikacije.



Slika 3. Vrste memorije na GPU

U nastavku su prikazane informacije o korišćenoj grafičkoj kartici, a u dodatku na kraju dokumenta i program koji omogućava ovaj ispis.

```
--- General Information for device 0 ---
Name:                                          GeForce 8400 GS
Compute capability:                            1.1
Clock rate:                                    1350000
Device copy overlap:                           Disabled
Kernel execition timeout :                     Enabled
 --- Memory Information for device 0 ---
Total global mem:                              536543232
Total constant Mem:                            65536
Max mem pitch:                                 2147483647
Texture Alignment:                             256
 --- MP Information for device 0 ---
Multiprocessor count:                          1
Shared mem per mp:                             16384
Registers per mp:                              8192
Threads in warp:                               32
Max threads per block:                         512
Max thread dimensions:                         (512, 512, 64)
Max grid dimensions:                           (65535, 65535, 1)
```

d. "Zdravo CUDA!"

Posmatrajmo problem sabiranja dva niza brojeva. Složenost izvršavanja ovakve operacije na sekvencijalnoj mašini je O(n), jer program mora da prođe kroz svaki element oba niza redom. Skraćena verzija CUDA koda koji rešava ovakav problem paralelno izgleda ovako:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
int main()
{
...
// Kernel invocation with N threads
VecAdd<<<1, N>>>(A, B, C);
}
```

Zapažamo da je zbog same prirode problema kod vrlo jednostavno paralelizovati, sve operacije su nezavisne. Sabiramo dva elementa na lokaciji koja odgovara niti sa ID-em `threadIdx.x`, te ćemo ako "pustimo" N niti, nakon samo jednog kernelskog poziva na izlazu imati zbir dva niza. Prefiks poziva `VecAdd<<<1, N>>>` znači da će se aktivirati 1 blok niti, sa N niti u sebi. U ovom slučaju bilo bi ekvivalentno pozvati i `VecAdd<<<N,1>>>`, jer se ne koristi deljena memorija, pa nije bitno da li imamo 1 blok i N niti, ili N blokova sa po jednom niti.

2. GA UOPŠTENO

U ovom delu teksta biće izložene opšte informacije o genetskim algoritmima. Čitaocu koji je upoznat sa GA opštim konceptima, predlažem da „preskoči“ ovo poglavlje i pređe na poglavlje **3. GA IMPLEMENTACIJA**, gde se razmatra problem koji rešavamo i vrši pregled implementacije sekvencijalnog GA.

a. Istorijat i osnovni GA

Genetski algoritam je razvio Džon Holand (eng. John Holland) šezdesetih godina prošlog veka . Originalni cilj razvoja genetskog algoritma nije bila praktična primena zarad rešavanja nekog specifičnog problema, već formalna studija o evoluciji i adaptaciji u prirodi i načinima na koje je tu logiku moguće ubaciti u računarstvo. U njegovoj knjizi iz 1975, prikazan je genetski algoritam kao abstrakcija biološke evolucije.

Da bismo detaljnije opisali osnove genetskog algoritma, uvešćemo deo biološko računarske terminologije vezane za temu biološke evolucije. Svi živi organizmi se sastoje od ćelija, i svaka ćelija sadrži skup hromozoma, koji su izgrađeni od lanaca DNK (eng. DNA - Deoxyribonucleic acid) koji kodiraju informacije o odgovarajućem organizmu. Hromozom dalje može da se deli u gene, koji su zapravo funkcionalni blokovi DNK, gde svaki gen predstavlja odgovarajuće svojstvo organizma. Različite mogućnosti za svako svojstvo, npr. različita boja za oči se nazivaju genski aleli. Svaki gen je postavljen na određenoj lokaciji hromozoma. Tokom reprodukcije dešava se ukrštanje gena: oni bivaju razmenjeni između parova roditeljskih hromozoma i daju gene potomka. Potomstvo dalje postaje subjekat mutacije, gde su pojedinačni segmenti DNK skloni promenama. “Prilagođenost” ili “dobrota” (eng. fitness) organizma se može kvantifikovati kao verovatnoća da će organizam opstati i dobiti mogućnost da se reprodukuje i ostavi svoje potomstvo u narednoj generaciji. Skup svih hromozoma u datom momentu se naziva populacija. Evolucija se postiže mutacijama i ukrštanjem između različitih jedinki, pri čemu najbolje jedinke imaju pravo da učestvuju u kreiranju naredne generacije. U računarstvu, genetski algoritmi se koriste za pronalaženje „dobrih“ rešenja iz velikog skupa mogućih. Osnovni genetski algoritam je dat sledećim pseudokodom:

```
P←InicijalnaPopulacija()
while uslovi zaustavljanja nisu ispunjeni do
          for each jedinka in P do
             pi←funkcija_cilja()
          end for each
          IzracunajFunkcijuPrilagododjenosti()
          Selekcija()
          Ukrštanje()
          Mutacija()
end while
```

b. Kodiranje

Kao što je opisano, osnova prirodne genetike je hromozom. Prilikom projektovanja ove ideje na genetske algoritme, potrebno je razmišljati o mapiranju svake jedinke u okviru populacije u nešto nalik hromozomu. Uobičajeni način da se ovo uradi je korišćenjem niza bitova. Na taj način svaka lokacija u okviru hromozoma predstavlja gen, a aleli odgovaraju vrednostima jedan ili nula. Prednost ovog kodiranja je jednostavnost i ono se često koristi kod problema kombinatorne optimizacije. Na primer, ukoliko želimo da dobijemo sumu brojeva što bližu vrednosti x, sumiranjem brojeva od jedan do dvadeset, tako da zbir brojeva bude što manji, možemo koristiti hromozom od dvadeset bitova, gde je svaki broj predstavljen odgovarajućom lokacijom u okviru

hromozoma (ovaj optimizacioni problem je lak, spomenut je tek zbog intuitivnog opisa koncepta, obično se genetski algoritmi primenjuju rešavanju NP teških problema). Tada alel sa vrednošću jedan, na odgovarajućoj lokaciji znači da broj ulazi u sumu. Drugi način je da koristimo niz prirodnih brojeva. Takvo kodiranje je dobro za rad sa permutacionim problemima, tj. problema kod kojih je bitan redosled, a ne samo pripadnost skupu rešenja, na primer, problem trgovačkog putnika (eng. travelling salesman person – TSP). Broj načina kodiranja nije ograničen sa ove dve konvencije - ovo su samo dve najčešće korišćenje i najpogodnije za prikaz. Kodiranje može biti i drugačije zamišljeno, ali uvek mora zadržati mogućnosti primene druga dva operatora: ukrštanja i mutacije.

c. Funkcija prilagođenosti

U cilju određivanja koliko je koja jedinka u okviru populacije dobra, definiše se funkcija prilagođenosti. Uobičajena primena genetskog algoritma je optimizacija neke funkcije. Nažalost, optimizacioni problemi su retko kad jednostavni i genetski algoritam je, pre svega, orijentisan ka optimizaciji kompleksnih funkcija više promenljivih i tzv. funkcija nad ne-numeričkim podacima, za koje ne postoje efikasna egzaktna numerička rešenja. Prirodno, što je kompleksniji problem, komplikovanija je i funkcija prilagođenosti. Kada se radi o numeričkim podacima, funkcija prilagođenosti može biti lako utvrđena iz samog optimizacionog problema. Najkomplikovanije funkcije prilagođenosti su one potrebne za evaluaciju ne-numeričkih podataka, kod kojih se mora tražiti adekvatna metrika za evaluaciju rešenja.

d. Operator selekcije

Ovaj operator određuje jedinke koje će steći pravo da ostave potomstvo za sledeću generaciju i obično se definiše tako da one jedinke, koje imaju bolju prilagođenost, imaju veće šanse. Najjednostavniji način definisanja operatora selekcije je čisto elitistički pristup, kod kojeg se biraju jedinke sa najvećom vrednošću funkcije prilagođenosti. Ovaj pristup je jednostavan za shvatanje i implementaciju. Međutim, ovo nije uvek najbolji izbor jer često može da uzrokuje „zaglavljivanje“ u lokalnom optimumu. Drugi, češće korišćeni metod, je zasnovan na selekciji koja je proporcionalna prilagođenosti, ali sa dozom slučajnosti. Može se implementirati korišćenjem „ruletskog točka“, gde svaka jedinka dobija svoj odsečak na točku veličine koja je propocionalna funkciji prilagođenosti. Na ovaj način, jedinke sa većom vrednošću funkcije prilagđenosti imaju veće šanse da budu izabrane. Ipak, često se koristi i kombinacija ova dva metoda, prema kojem određeni broj najboljih jedinki garantovano prelazi u narednu generaciju zarad ubrzavanja algoritma, dok ostali birane gore predloženim probabilističkim metodom, kako bi se osigurala raznolikost populacije. Jedna od popularnijih tehnika je i turnirska selekcija kod koje se populacija deli u grupe od po n jedinki, koje se zatim „nadmeću“, a „pobednik ostaje“. Ponekad je problem odabrati odgovarajući broj n, te postoji i metod tzv. fino gradirane turnirske selekcije, koji omogućava da u proseku taj broj ne bude ceo.

e. Operator ukrštanja

Operator ukrštanja se primenjuje na dva hromozoma roditelja i kreira dva potpuno nova hromozoma, tj. njihovo potomostvo, koje sadrži kombinovana svojstva roditelja. Ukrštanje se primenjuje na slučajno odabranim lokacijama hromozoma, tzv. tačkama prekida. Operator ukrštanja vrši transpoziciju podnizova pre i posle odabrane lokacije. Na primer, predpostavimo da imamo hromozome $\mathbf{x_1x_2x_3...x_n}$ i $\mathbf{y_1y_2y_3...y_n}$, i odabrali smo lokaciju k, k<n za tačku prekida. Potomstvo će u tom slučaju predstavljati dva hromozoma $\mathbf{x_1x_2..x_ky_{k+1}..y_n}$ i $\mathbf{y_1y_2..y_kx_{k+1}...x_n}$. Vodeći se ovim primerom, lako je zamisliti ukrštanje sa dve, tri, u opštem slučaju m tačaka prekida.

Koeficijent ukrštanja predstavlja verovatnoću da će se ukrštanje primeniti nad nekim parom hromozoma. Parovi hromozoma se utvrđuju po unapred ili slučajno određenim indeksima, a zatim se nad tim parovima vrši ukrštanje ukoliko je slučajni generisani broj manji od koeficijenta ukrštanja.

f. Operator mutacije

Mutacija je način kreiranja novih individua iz populacije pravljenjem manjih promena alela na slučajnim lokacijma već postojećih individua (hromozoma). U slučaju binarnog kodiranja, osnovna mutacija podrazumeva promenu 0 u 1 i obrnuto. Na primer, možemo imati niz bitova 010100, i mutirati njegovu treću lokaciju pri čemu ćemo dobiti 011100. Kada niz sadrži prirorodne brojeve, mutacija može podrazumevati zamenu njihovih vrednosti na dve lokacije, dakle jednostavna transpozicija. No, bez obzira na to kako je mutacija zamišljena, dobijeni rezultat uvek mora biti legitimna individua, tj. mora biti "neko" rešenje posmatranog problema. Za svaku mutaciju uvek postoji definisana verovatnoća primene, tzv. koeficijent mutacije (eng. mutation rate). Kao i u prirodi, mutacije su neželjene u najvećem broju slučajeva, tako da su koeficijenti mutacije uglavnom dosta niski. O tačnom stepenu mutacije treba dobro razmisliti, jer prevelike ili premale verovatnoće primene mutacije mogu rezultirati problemima. Ako je ona prevelika, algoritam će bez jasnog fokusa pretraživati prostor rešenja, u suprotnom ako je premala, populacija će iz generacije u generaciju ostajati vrlo slična, jer ne postoji dovoljno razlika među individuama, što obično dovodi do lokalnog optimuma.

g. Kriterijum zaustavljanja

Genetski algoritam ne pretražuje prostor rešenja sistematski kao što je to slučaj kod potpune enumeracije, te je potrebno definisati kriterijum zaustavljanja. Neki od poznatijih kriterijuma su: maksimalni broj generacija, maksimalni broj ponavljanja najbolje jedinke, stepen sličnosti jedinki, ograničeno vreme izvršavanja, dostizanje optimalnog rešenja (pod uslovom da je unapred poznato) i druge.

3. GA IMPLEMENTACIJA

GA spada u klasu metaheuristika, a to znači da predstavlja rešavač opšte klase optimizacionih problema. Iz tog razloga opis samog problema koji rešavamo ne bi trebao da bude od prevelikog značaja, tj. pri razvoju samog algoritma, programer ne bi trebao da ulazi u pitanje specifikacije samog problema, koji će taj algoritam kasnije rešavati. U razvoju sam se držao ovih principa, tako da je veza između rešavača i samog problema lokalizovana na svega jednu funkciju. To je naravno funkcija cilja, koja preslikava niz bitova (lokuse hromozoma) u numeričku vrednost.

Problem koji rešavam se naziva problem minimalne energetske povezanosti (eng. Strong Minimum Energy Topology - SMET) i tiče se optimizacije potrošnje energije u bežičnim senzorskim mrežama. Formulacija je sledeća:

Definicija 1 *SMET: Za dati skup senzora u ravni, odrediti vrednost energije koju treba dodeliti svakom senzoru, tako da između svakog uređenog para čvorova senzora postoji najmanje jedan usmereni put i ukupna potrošnja energije je minimalna.*

Energija prenosa (transimisije) između senzora (čvorova) obeleženog indeksom *i* i čvora sa indeksom *j* se obično definiše kao:

$$f_{i,j} = f(d_{i,j}) = t_j(d_{i,j})^{\alpha}$$

gde $d_{i,j}$ predstavlja meru udaljenosti između čvorova, t_j prag osetljivosti (eng. detection sensitivity treshold) senzora *j*, tj. potrebna vrednost jačine signala dovoljna da ga senzor *j* detektuje. Kada je u pitanju prag osetljivosti, u literaturi se obično podrazumeva da je on jednak za sve senzore te se često njegova vrednost normalizuje na 1. Ovo ima za posledicu da problem, koji posmatramo, postaje simetričan. Konstanta α je vezana za stepen gubitka energije signala u zavisnosti od povećanja udaljenosti (eng. path loss), i ona obično ima vrednost 2 ili 4. Ako posmatramo čvor *i* i njemu dodeljenu energiju z_i, za svaki čvor *j* za koji važi $f_{i,j} \leq z_i$, kažemo da pripada oblasti signala čvora *i*, tj. moguće je poslati signal sa čvora *i* na čvor *j*. Neka je $G_d = (V, E)$ kompletan digraf, gde je V skup čvorova (senzora), a E je skup svih usmerenih grana. Grafovska formulacija SMET-a je sledeća:

Definicija 2 *SMET**:** Za dati kompletan digraf G_d i funkciju energije $f_{i,j}$ odrediti pridružene vrednosti energije u svakom od čvorova grafa $\{z_1, z_2, ..., z_n\}$, tako da je podgraf dat sa $G_d = (V, E')$, gde je $E' = \{f_{i,j} \mid f_{i,j} \leq z_i, i, j \in V\}$ strogo povezan i ukupna energija data sa $\sum_{k \in N} z_k$ minimalna.*

U ovoj implementaciji razmatran je slučaj kada je t_j normalizovan, tj. kada je problem simetričan, tj. Ovo je vrlo racionalna pretpostavka, jer su često senzori u nekoj bežičnoj senzorskoj mreži istih karakteristika, pa samim tim i istog praga osetljivosti. Konkretne informacije o korišćenim instancama problema će biti pojašnjene u poglavlju sa eksperimentalnim rezultatima. Zasad, bitno je shvatiti se problem opisuje skupom čvorova i grana sa pridruženim težinama, a da će ulazna datoteka problema sadržati informacije sledećeg formata:

```
broj_cvorova(Nn) broj_grana(Nl)
#cvor1 #cvor2 tezina_cvor1_cvor2
#cvor1 #cvor3 tezina_cvor1_cvor3
```

```
...
#cvor1 #cvor(Nn) ...
#cvor2 #cvor3...
...
```

a. Implementacija sekvencijalnog algoritma

Dalja organizacija teksta pratiće onu iz poglavlja **2,** tj. redom će biti objašnjen svaki od koncepata GA: kodiranje, operatori itd. Platforma koju ću prikazati razvijena je „od nule“, sa tim da je nekoliko funkcija ili fragmenata koda preuzeto sa interneta. U okviru moje master teze koju sam radio prošle godine, implementiran je sekvencijalni genetski algoritam, koji na žalost nije bio pogodan za prebacivanje na CUDA kod, a razlog je prevelika robusnost pomenutog sistema. Novi sekvencijalni algoritam je kasnije transformisan u CUDA paralelni algoritam, bez i jedne konceptualne promene u radu genetskih operatora i drugih funkcija. Ovo je nadam se, doprinelo validnosti uporednih rezultata, jer kada se ukloni presek svih sličnosti ostaje samo paralelna transformacija.

Resurs.
Sekvencijalni GA – u okviru master teze: http://www.math.rs/p/files/19-master.zip

```
start=clock();
/*inicijalizacija pocetne populacije*/
init();
evpop();
order(0);
lastBest=INF;
for(i=0;i<num;i++)
{
          if((MINIMIZATION && popcurrent[0].fit<lastBest)
          ||(!MINIMIZATION && popcurrent[0].fit>lastBest))
          {
              lastBest=popcurrent[0].fit;
              found=clock();
              bestRepeated=1;
              modifiedMutationProb=MUTATIONPROB;
          }else{
              bestRepeated++;
              if(bestRepeated%1000==0)
                    modifiedMutationProb+=0.01;
          }
          printf("Najbolje:\t%f\ti = %d\n",lastBest,i);
          /*selekcija*/
          selection();
          /*ukrstanje*/
          crossover(1);
          /*mutacija nad linkovima*/
          mutation();
          /*racunamo f-je cilja i fitness-e za sve jedinke*/
          evpop();
          /*uredjivanje*/
          order(0);
          /*ako su se pojavili duplikati, izbacujemo ih jer
          narusavaju
          diverzitet genetskog koda*/
          replaceEqual();
          /*sortiramo ponovo*/
          order(0);
}
end=clock();
```

U kodu iznad se jasno vidi struktura genetskog algoritma, koji odgovara opštoj „teorijskoj“ strukturi. Ovom delu koda prethode neke funkcije koje nisu prikazane: učitavanje parametara komandne linije, učitavanje instanci problema koji rešavamo, a nakon datog dela koda, vrši se ispisivanje rezultata u datoteku, oslobađanje memorije i pokazivača na fajlove.

Resurs.
GA – ova implementacija: http://www.math.rs/~kartelj/PGA.zip

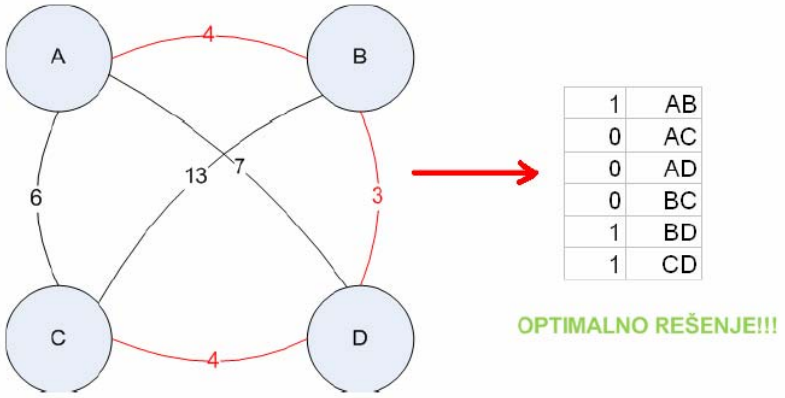
b. Kodiranje (implementacija)

U razmatranom problemu, na sreću, nema nikakave dileme o načinu kodiranja. Svakoj komunikacionoj liniji između dva senzora (grani grafa između dva čvora) jednostavno dodeljujemo vrednost 0 ili 1, u zavisnosti da li pripada ili ne pripada rezultujućoj topologiji mreže (rezultujućem podgrafu).

Funkcija cilja se potom izračunava kao zbir dodeljenih energija svakom senzoru u mreži. Iz definicije problema od ranije se vidi da je dodeljena energija za svaki pojedinačni senzor jednaka energiji maksimalne izlazne komunikacione linije (izlaznoj grani sa maksimalnom težinom).

Terminološka opaska. Na levoj strani su dati grafovski, a na desnoj termini iz prakse. U daljem tekstu, biće korišćena grafovska terminologija.
Čvor ⇔ Senzor
Grana ⇔ Komunikaciona linija
Težinski graf ⇔ Senzorska mreža
Rezultujući podgraf ⇔ Rezultujuća topologija mreže senzora

Na Slici 4. je dat primer grafa sa četiri čvora i 6 mogućih grana. Prikazano je kodiranje rešenja, a kao što se vidi četiri grane (obojene crvenom) su odabrane. Funkcija cilja za ovaj rezultujući podgraf je 16, jer je težina najveće izlazne grane u svakom od čvorova jednaka 4.



Slika 4. Primer kodiranja grafa na niz 0 i 1

Međutim, ne treba zaboraviti ni uslov povezanosti. Ukoliko npr. na grafu sa Slike 4. ne bi bila odabrana grana AB (ili BD ili CD), graf ne bi bio povezan (za proveru povezanosti korišćen je iterativni BFS algoritam). Tada bi dobili bitovsku reprezentaciju nevalidnog hromozoma. U tom slučaju možemo ili da obacimo hromozom iz populacije, ili da pokušamo da ga popravimo. U ovoj implementaciji vršeno je odbacivanje.

Zbog jednostavnosti algoritma za vršenje evaluacije funkcije cilja, kod neće biti prikazan, jedino vredno pomena su strukture koje su koriišćene za opis hromozoma i grane, kao i parametri GA:

```
typedef struct Chrom
{
          char *lbit; /*niz bitova koji odgovaraju odabiru grana*/
          int *z; /*energije dodeljene svakom od cvorova*/
          double obj; /*f-ja  cilja*/
          double fit; /*f-ja prilagodjenosti*/
          char valid; /* da li je hromozom korektan ('0' ili
          '1')*/
}chrom;

typedef struct Link{
          int from;
          int to;
          int weight;
}link;

/*velicina populacije hromozoma*/
#define POPSIZE 256
/*broj elitnih jedinki*/
#define NELITE 26
/*broj igraca u turnirskoj selekciji*/
#define NCOMPET 8
/*pocetna verovatnoca mutacije, moze se povecavati kasnije*/
#define MUTATIONPROB 0.005
/*verovatnoca ukrstanja*/
#define CROSSOVERPROB 0.85
/*beskonacno*/
#define INF 10000000
/*minimizacija = 1 / maksimizacija =0*/
#define MINIMIZATION 1
/*gustina grafa, koristi se pri inicijalizaciji
vrednosti. Sto je manje, graf ce biti redji*/
#define GRAPHDENSITY 0.5
```

c. Funkcija prilagođenosti (implementacija)

Funkcija prilagođenosti je ovde ista kao i funkcija cilja. Ovo, dakako ne mora uvek biti slučaj, ponekad je praktično izvršiti skaliranje funkcije cilja na manji ili veći opseg, ili neku drugu vrstu transformacije. Najčešće transformacija iz funkcije cilja u funkciju prilagođenosti održava uređenje.

d. Operator selekcije (implementacija)

Primenjena je kombinacija elitističke i turnirske selekcije. Naime, prvih NELITE najboljih jedinki biva odabrano direktno. Nakon toga se za ostale organizuje turnirska selekcija sa po NCOMPET slučajno odabrana učesnika.

e. Operator ukrštanja (implementacija)

Nakon izvršene selekcije, vršimo ukrštanje hromozoma. Isključujemo iz procesa najboljih NELITE hromozoma, a za ostatak odabranih hromozoma vršimo slučajna uparivanja i ukrštanja nad formiranim parom. Ovo radimo bez provere da li je neki hromozom već učestvovao u uparivanju. Kada je par formiran, sa verovatnoćom CROSSOVERPROB vršimo jednopoziciono ukrštanje, sa slučajno odabranom pozicijom ukrštanja.

f. Operator mutacije (implementacija)

Implementirana je jednostavna mutacija, operator se primenjuje sa verovatnoćom `MUTATIONPROB` po svakom bitu, svih hromozoma u okviru populacije.

g. Kriterijum zaustavljanja (implementacija)

Izvršavanje GA se završava ukoliko broj generacija (iteracija) dostigne neku maksimalnu vrednost. Ta vrednost se određuje prilikom pokretanja aplikacije, tako da je nije moguće sada specifikovati.

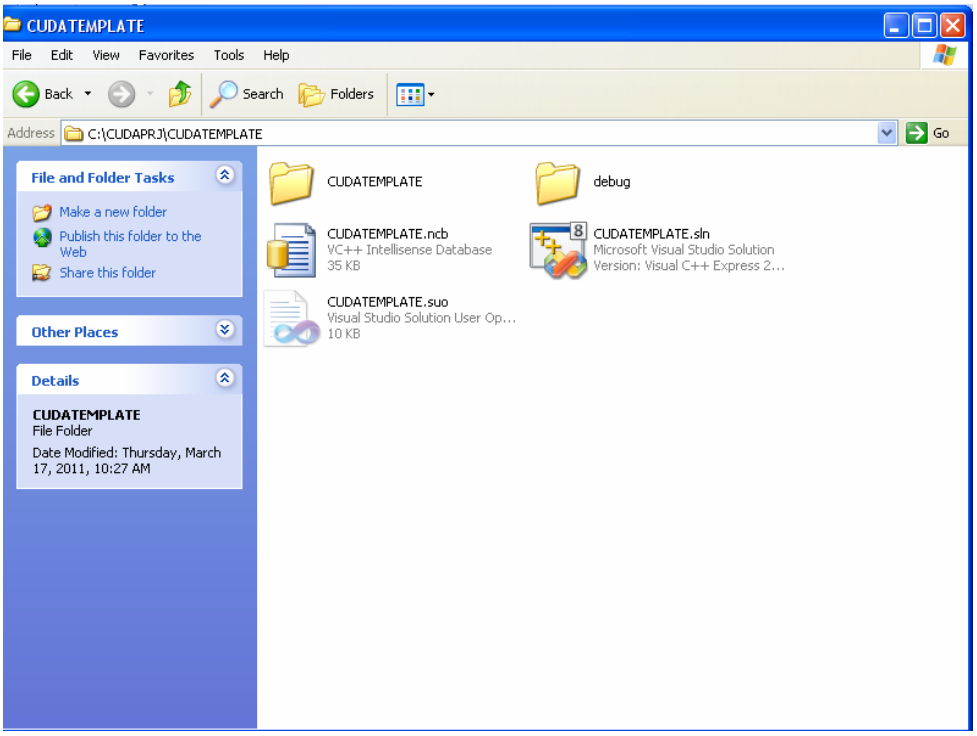
4. CUDA+GA

Dolazimo do centralnog poglavlja ovog dokumenta, u njemu će biti izložena transformacija GA na CUDA sistem. U procesu prebacivanja iskusio sam mnogo poteškoća, pre svega u pogledu otkrivanja i ispravljanja grešaka (eng. debugging).

Nakon što sam odabrao da radim projekat optimizacije na CUDA sistemu, moj prvi korak je bio da pronađem hardversku podršku za izvođenje istog. Prvi kandidat je bio i „najjeftiniiji“, Nvidia 8400 GS. Specifikacija parametara relevantnih za rad sa ovom karticom, data je u poglavlju 1. Nakon ispunjavanja hardverskih zahteva, potrebno je instalirati CUDA drajvere, koji se distribuiraju zajeno sa CUDA kompajlerom i potebnim bibliotekama u okviru CUDA Toolkit-a. Nakon toga pokušao sam da nađem adekvatno, po mogućstvu udobno okruženje za rad sa CUDA API-jem. Nvidia nudi platformu pod nazivom „Parallel Nsight“, koja je zapravo dodatak (eng. plugin) za Visual Studio 2008. On poseduje bogato okruženje, a najbitniji mi je bio alat za otkrivanje grešaka. Nažalost, imao sam problema sa softverskim ograničenjima, Parallel Nsight je zahtevao Windows 7 i Visual Studio 2008.

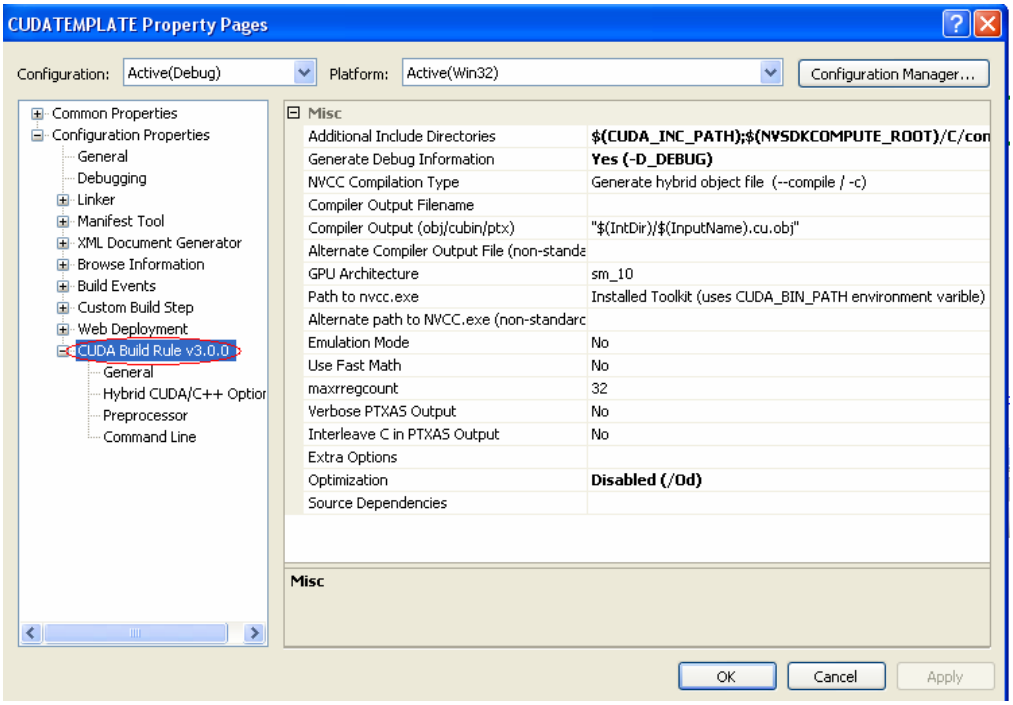
Resurs.
CUDA toolkit - http://developer.nvidia.com/cuda-toolkit-40
Parallel Nsight - http://developer.nvidia.com/nvidia-parallel-nsight
CUDA project template – može se naći pod ovim ili pod nazivom CUDA wizard na internetu

Potražio sam alternativu, i našao je: Windows XP+Visual Studio 2005, što je baš odgovaralo mojim trenutnim softverskim resursima. CUDA ne nudi dodatak za VS 2005, međutim, na internetu se može pronaći VS 2005 projekat pod nazivom „CUDA project template“, koji se potom samo otvori iz VS 2005 (Slika 5).



Slika 5. CUDA template za VS 2005

U okviru podešavanja projekta (eng. property pages, Slika 6.) pored standardnih stavki, sada postoji i jedna nova „CUDA Build Rule v3.0.0“. Ona sadrži podešavanja koja se tiču uključenih putanja, potrebnih biblioteka itd. Jedna od bitnih stavki je i putanja do nvcc.exe datoteke, koja zapravo predstavlja C kompajler, prilagođen radu sa CUDA. Ukoliko je CUDA toolkit uredno instaliran pre otvaranja CUDA template projekta, ne bi trebalo da bude problema u prepoznavanju lokacije kompajlera. Uz ovako podešeno okruženje, možemo da počnemo da programiramo.



Slika 6. Podešavanja projekta (eng. property pages)

a. Paralelizacija funkcije cilja

U sekvencijalnom GA može se uočiti da svaki od hromozoma vrši neki svoj nezavisni skup aktivnosti. U te aktivnosti spada i evaluacija funkcije cilja, naime, svaki od njih poseduje dovoljne informacije, a to je niz bitova (genetski kod hromozoma), za koji se vrši izračunavanje opisano u **3.b** sekciji. Ideja je dakle, svakom hromozomu dodeliti zasebnu nit, tj. napisati kernel funkciju i izvršiti je paralelno u POPSIZE niti. Sekvencijalni kod:

```
init();
evpop();
order(0);
lastBest=INF;
for(i=0;i<num;i++)
{
...
          selection();
          crossover(1);
          mutation();
          /*racunamo f-je cilja i fitness-e za sve jedinke*/
          evpop();
...
}

void ev(int j){
```

```
            obj(&popcurrent[j]);
            fitness(&popcurrent[j]);
}
void evpop()
{
            int j;
            for(j=0;j<POPSIZE;j++)
            {
                ev(j);
            }
}
```

se transformiše u paraleizovani:

```
init();
evpop<<<1,POPSIZE>>>(popcurrent);
order(0);
lastBest=INF;
for(i=0;i<num;i++)
{
...
            selection();
            crossover(1);
            mutation();
            /*racunamo f-je cilja i fitness-e za sve jedinke*/
            evpop<<<1,POPSIZE>>>(popcurrent);
...
}

__device__ void ev(chrom *chr){
            (*chr).obj=obj(chr->lbit);
            (*chr).fit=fitness(chr->lbit);
}
__global__ void evpop(chrom *popcurrent)
{
            int tid=threadIdx.x + blockIdx.x * blockDim.x;
            if(tid<POPSIZE){
                ev(&popcurrent[tid]);
            }
}
```

Dakle, prvo zapažanje je da nema više petlje koja se „vrti“ POPSIZE puta u funkciji **evpop**. Ona je zamenjena pozivom kernel funkcije sa 1 blokom od POPSIZE niti. U telu funkcije **evpop**, određuje se jedinstveni identifikator niti. Svaka nit sadrži nekoliko privatnih informacija, koje određuju ID niti. ID se računa kao `threadIdx.x+blockIdx.x* blockDim.x`. Ipak ovde je ID bloka redudantan, jer je kernel pozvan za samo jedan blok, pa je kod svih niti ta vrednost jednaka 0. Ranije je napomenuto da se niti mogu grupisati i u dvo- i trodimenzione blokove, a blokovi u dvodimenzione mreže, tako da se mogu pojaviti i sledeći identifikatori: `threadIdx.y`, `threadIdx.z`, `blockIdx.y`.

Kernelska funkcija može biti pozvana samo iz host funkcije, dakle kernel ne može pozvati neki drugi kernel. Međutim, kerneli mogu pozivati pomoćne funkcije, koje se obeležavaju kvalifikatorom `__device__`. Takođe, `__device__` funkcije ne mogu pozivati samu sebe, CUDA kompajler ne zna da radi sa rekurzijom!

b. Paralelizacija ostalih funkcija

Po sličnom modelu, paralelizujemo i ostale funkcije sekvencijalnog GA. Neću prikazivati kako je to urađeno za svaku pojedinačno, jer verujem da je ideja jasna iz paralelizacije funkcije cilja. Međutim, moram da naglasim da nailazimo na uslovno usko grlo u situacijama kada vršimo neku agregatnu operaciju, kao što je npr. selekcija ili ukrštanje. Tada niti nisu više nezavisne, tj. da bi se npr. izvršila selekcija, sve niti pre

toga moraju da imaju izračunatu funkciju cilja, kako bi mogle da se pored. U ovoj situaciji koristimo funkciju koja sinhronizuje rad niti.

```
__global__ void selection(chrom *popcurrent, chrom *poptemp)
{
          int tx=threadIdx.x;
          int k, intChosen, i;
          int intChamp, intCompet;
          int ind=0;
          //alociramo deljenu memoriju, jer je brza
          __shared__ int sfit[POPSIZE];
          sfit[tx+NELITE]=popcurrent[tx+NELITE].fit;
          //cekamo da sve niti upisu vrednosti f-je cilja u ---
          //deljenu memoriju
          __syncthreads();
          k =NELITE+tx;
          //vrsimo turnirsku selekciju za neelitne hromozome
          intChamp= k_rand(intItem)%(POPSIZE-NELITE)+NELITE;
          for ( intChosen=1; intChosen<NCOMPET; intChosen++ ){
              intCompet= k_rand(k)%(POPSIZE-NELITE)+NELITE;
               //bolja jedinka (manja vredn. F-je prilag. pobedjuje
              if (sfit[intCompet]<sfit[intChamp])
                     intChamp= intCompet;
          }
          ind=intChamp;
          //poptemp je kopija ove populacije, potrebna nam je jer
          //vrsimo razmestaj hromozoma
          popcurrent[intItem].fit=poptemp[ind].fit;
          //na trenutnu poziciju stavljamo pobednicki hromozom
          for(i =0; i <probNl; i ++){
                 popcurrent[k].lbit[i]=popcurrent[ind].lbit[i];
          }
}
```

__syncthreads sinhronizuje rad niti, u smislu da, sve niti koje paralelno izvršavaju trenutni kernel čekaju na ovoj funkciji sve ostale niti. Dakle, ova barijera se ne prolazi dok god sve niti ne završe svoj posao. Poziv **__syncthreads** funkcije je obezbedio da se ne ide dalje u algoritam selekcije, dok god svi hromozomi ne iskopiraju svoju funkciju prilagođenosti u deljenu memoriju. Deljena memorija ima kraće vreme pristupa, te je pogodno na početku keširati f-je prilagođenosti svih hromozoma u okviru populacije `(popcurrent)` iz globalne memorije u deljenu memoriju. U narednoj sekciji izložiću dalje korake u optimizaciji izvršavanja, a priča će posebno biti fokusirana na korišćenje deljene memorije.

c. Uvođenje deljene memorije – revizija koda

Nakon izvršene prve faze paralelizacije, kod izgleda ovako:

```
chrom *popcurrent; //globalne promenljive (spor pristup)
chrom *poptemp;
...
init();
evpop<<<1,POPSIZE>>>(popcurrent);
order(0);
lastBest=INF;
for(i=0;i<num;i++)
{
...
   //pravimo kopiju populacije hromozoma i upisujemo je u poptemp
   fillTempPop<<<1,POPSIZE>>>(popcurrent, poptemp);
   //selekcija
   selection<<<1,(POPSIZE-NELITE)>>>(popcurrent, poptemp);
   //ukrstanje
```

```
   crossover<<<1,(POPSIZE-NELITE)>>>(1, popcurrent);
   //mutacija
   mutation<<<1,(POPSIZ-ENELITE)>>>(popcurrent,cMutatProb);
   //izracunavanje f-je cilja i prilagodjenosti
   evpop<<<1,POPSIZE>>>(popcurrent);
   //sort po f-ji cilja. Na izlazu je sortirajuca permutacija
   bitonicSort<<<1,POPSIZE>>>(popcurrent, sortedPerm);
   //primena sortirajuce permutacije
   reorder<<<1,POPSIZE>>>(popcurrent,
                          poptemp,sortedPermutation,cBest);
...
}
```

Problem sa ovim algoritmom je to što ne iskorišćava u potpunosti prednosti deljene memorije. Iako unutar samih funkcija, a to smo videli na primeru selekcije, koristimo deljenu memoriju, problem je u tome što se globalna memorija previše često referiše. Kao što se vidi iz koda, prilikom svakog poziva neke od funkcija, u okviru nje se prepisuje vrednost iz globalne memorije u deljenu, onda se ta deljena memorija koristi za potrebe algoritma, i na kraju opet ažurira globalna memorija. Sve to se radi u petlji, tako da je ukupan broj pristupa globalnoj memoriji izuzetno velik. Dakle, može to i bolje!

d. Reorganizacija – smanjivanje broja pristupa globalnoj memoriji

Već sam napomenuo ranije da je pristup globalnoj memoriji izuzetno neefikasan. Prema nekim izvorima, taj pristup je čak do 150 puta sporiji nego pristup deljenoj i registarskoj memoriji. Ako imamo nekoliko hiljada iteracija petlje, i pritom skoro svaka od navedenih funkcija koristi kopiranje globalne memorije u deljenu, može se zaključiti koliko to zapravo usporava algoritam. Upravo efekat usporenja sam i iskusio u prvim fazama testiranja, dok nisam izvršio reorganizaciju koda. Ideja je bila smanjiti nekako broj pristupa deljenoj memoriji.

Da bih postigao željeni efekat izvršio sam nekoliko reorganizacija. Najpre sam uveo samo jednu kernelsku funkciju, a u nju sam stavio sve ostale funkcije, koje su sada postale pomoćne funkcije. Prebacivanje podataka je tako vršeno samo na ulasku u tu jednu funkciju, a nakon toga je već učitana deljena memorija prosleđivana kao parametar pomoćnim funkcijama. Međutim, i pored ove promene broj pristupa je bio prevelik, tako da je prva ideja bila da petlju koja „vrti“ generacije ubacim u kernelsku funkciju. Na taj način bi se broj pristupa globalnoj memoriji sveo na 2, jedno čitanje i jedno upisivanje na kraju. Na žalost, ova ideja nije uspela, a razlog je jednostavan: CUDA ima vremensko ograničenje za izvršavanje kernelske funkcije, koje iznosi nekoliko sekundi. Ovo je samo jedno od ograničenja koje CUDA uvodi, postoje i određena ograničenja za broj ugnježdenih petlji, beskonačne petlje (sa break komandom) itd. Neka od ograničenja bivaju prepoznata u fazi kompilacije, npr. ograničenje količine deljene memorije o kojem ću biti reči nešto kasnije. Ograničenja vezana za petlje, se uglavnom ne proveravaju u fazi kompilacije, već se jednostavno desi izuzetak u fazi izvršavanja. Rešenje problema je bio kompromis između navedenog ograničenja i broja pristupa memoriji. Razbio sam spoljnu petlju na dve petlje, tako da se jedna nalazu u glavnom programu, a druga u kernelskoj funkciji. Pritom je kritičan broj iteracija u unutrašnjoj petlji, takoreći treba da bude što je veći moguć, a da pri tom ne izaziva pomenuti izuzetak za maksimalno vreme izvršavanja. Testiranjem je utvrđeno da povoljan broj iteracija unutrašnje (kernelske petlje iznosi oko 50).

```
#define KERNELLOOP 50

...
/*spoljasnja petlja*/
for(i=0;i<(num/KERNELLOOP);i++)
{
...
```

```
            GAallInOneOperator<<<1,POPSIZE>>>(popcurrent,cMutProb
            ,cBest);
...
}
...

__global__ void GAallInOneOperator(chrom *popcurrent, float
            *cModifiedMutationProb, int *cBest){
            int i,j;
            int tid=threadIdx.x;
            /*deljena memorija za originale i kopije populacije*/
            __shared__ unsigned  short int sfit[POPSIZE];
            __shared__ unsigned char slbit[POPSIZE][EDGEMEMNUM];
            __shared__ unsigned  short int sfit_t[POPSIZE];
            __shared__ unsigned char slbit_t[POPSIZE][EDGEMEMNUM];
            ...
            /*kopiramo u deljenu memoriju, samo jedno kopiranje po
            kernelskom pozivu*/
            sfit[tid]=popcurrent[tid].fit;
            for(i=0; i<POPSIZE; i++){
                slbit[tid][i]=popcurrent[tid].lbit[i];
            }
            /*kopiramo i u pomocne nizove*/
            sfit_t[tid]=sfit[tid];
            for(i=0; i<slbytes; i++){
                slbit_t[tid][i]=slbit[tid][i];
            }
            /*cekamo da svi upisu*/
            __syncthreads();
            /*unutrasnja petlja (kernelska petlja)*/
            for(int p=0; p<KERNELLOOP; p++){
                   selection(tid,sfit,slbit);
                   __syncthreads();
                   crossover(tid,sfit,slbit);
                   __syncthreads();
                   mutation(tid,slbit,*cModifiedMutationProb);
                   sfit[tid]=fitness(slbit[tid]);
                   __syncthreads();
                   bitonicSort(tid, sfit, slbit);
                   __syncthreads();
                   reorder(tid,sfit,slbit);
                   __syncthreads();
                   bitonicSort(tid, sfit, slbit);
                   __syncthreads();
            }
            popcurrent[tid].fit=sfit[tid];

            /*vracamo vrednosti u globalnu memoriju, samo jednom po
                  kernelskom pozivu*/
            for(j=0; j<probNl; j++){
                  setBit(j,popcurrent[tid].lbit,
                           getBit(j,slbit[tid]));
            }
            *cBest=sfit[0];
}
```

A nova poboljšana varijanta operatora selekcije izgleda ovako:

```
__device__ void selection(int tid, unsigned short int
            sfit[POPSIZE], unsigned char slbit[POPSIZE][EDGEMEMNUM])
{
            int intItem, intChosen, intTemp;
            int intChamp, intCompet;
            int indSelected=0;
```

```
            unsigned short int fitTemp;
            unsigned char lbitTemp[EDGEMEMNUM];

            intItem=tid;
            if(intItem>=NELITE && intItem<POPSIZE){
                intChamp= k_rand(intItem)%(POPSIZE-NELITE)+NELITE;
                for ( intChosen=1; intChosen<NCOMPET; intChosen++ ){
                       intCompet= k_rand(intItem)%(POPSIZE-
            NELITE)+NELITE;
                       if ((MINIMIZATION &&
            sfit[intCompet]<sfit[intChamp])
                              || (!MINIMIZATION &&
            sfit[intCompet]>sfit[intChamp]))
                              intChamp= intCompet;
                }
                indSelected=intChamp;
                fitTemp=sfit[indSelected];
                for(intTemp=0; intTemp<probNl; intTemp++){

                setBit(intTemp,lbitTemp,getBit(intTemp,slbit[indSelec
            ted]));
                }
                __syncthreads();
                //sada kada su svi dobili lokacije, treba prekopirati
            //te vrednosti
                sfit[intItem]=fitTemp;
                for(intTemp=0; intTemp<probNl; intTemp++){

                setBit(intTemp,slbit[intItem],getBit(intTemp,lbitTemp
            ));
                }
            }
}
```

Ukupan broj čitanja i upisa u globalnu memoriju na taj način značajno smanjen, što je doprinelo podizanju performansi. O konkretnim ubrzanjima biće reči u narednom poglavlju.

e. Redukovanje memorijskih zahteva

Deljena memorija je vrlo ograničen resurs, svega 16KB po multiprocesorskoj jedinici. U slučaju Nvidia 8400 GS, postoji samo jedan multiprocesor, tako da je ovo bilo sve što sam imao na raspolaganju. Analizom memorijskih zahteva utvrdio sam da postoje mesta na kojima može da se izvrši značajna redukcija. Ranije opisana struktura hromozoma je izgledala ovako:

```
typedef struct Chrom
{
            char *lbit; /*niz bitova koji odgovaraju odabiru grana*/
            int *z; /*energije dodeljene svakom od cvorova*/
            double obj; /*f-ja  cilja*/
            double fit; /*f-ja prilagodjenosti*/
            char valid; /* da li je hromozom korektan ('0' ili
            '1')*/
}chrom;
```

A posle redukcije ovako:

```
typedef struct Chrom
{
```

```
        unsigned char *lbit; /*niz bitova koji odgovaraju
        odabiru grana*/
        int fit; /*fitness f-ja*/
}chrom;
```

Ranije je pomenuto da je funkcija prilagođenosti jedna funkciji cilja, tako da je vrednost za funkciju cilja izbačena iz koda. Energije dodeljene svakom od čvorova nisu relevantne u toku izvršavanja algoritma, već samo na kraju kada se ispisuje rezultat, tako da mogu jednokratno da se izračunaju. Informacija da li je hromozom korektan (rezultujući podgraf povezan) može da se inkorporira u samu funkciju prilagođenosti. Tako da je u slučaju da jedinka nije korektna prilagođenost postavljana na maksimalnu vrednost. Na taj način jedinka bi bivala implicitno isključivana iz razmatranja kroz operator selekcije, jer ne bi mogla da pobedi ni na jednom turniru (osim ukoliko bi se baš takmičila sa isto nevalidnim hromozomima, ali ovaj slučaj nije moguć jer se vrši uklanjanje duplikata posle svake iteracije). Budući da je vrednost funkcije prilagođenosti uvek ceo broj, barem za ovaj problem, prebacio sam je u ceo broj. Grane su ranije bile kodirane nizom karaktera '0' i '1', ovo nije bio problem za program koji je radio na CPU, memorija hipa (eng. heap) je bila i više nego dovoljna za ovakav „luksuz". Razmotrimo graf sa 25 čvorova. Broj grana je u tom slučaju 25*24/2=300, dakle 300 karaktera, što iznosi 300 bajtova. To znači da u deljenu memoriju pod pretpostavkom da nemamo drugih podataka može da stane 16KB/300B≈55 hromozoma. Uvođenjem bitovske reprezentacije uspeo sam da smanjim potrebnu količinu memorije 8 puta.

```
/*maksimalan broj cvorova*/
#define MAXNODES 25
/*maksimalan broj grana*/
#define MAXEDGES (MAXNODES*(MAXNODES-1)/2)
/*broj karaktera koje ostavljamo za bitovski niz grana +2 (+1 zbog
        zaokruzivanja na gore, +1 zbog term. nule*/
#define EDGEMEMNUM (MAXEDGES/EDGEMEMSIZE+2)

/*postavlja val (0 ili 1) na indeks idx.
__device__ void setBit(int idx,unsigned char *bitArr, int val){
        if(val==1){
            bitArr[idx / EDGEMEMSIZE]|=(1<<(idx%EDGEMEMSIZE));
        }
        else if(val==0){
            bitArr[idx / EDGEMEMSIZE]&=(~(1<<(idx%EDGEMEMSIZE)));
        }
}

/*cita vrednost bita na indeksu idx*/
__device__ int getBit(int idx,unsigned char *bitArr){
        return
        (bitArr[idx/EDGEMEMSIZE]&(1<<(idx%EDGEMEMSIZE)))>0;
}
```

Za opis 300 grana je sada potrebno 300/8+2 karaktera, tj. broj grana podeljen sa brojem bitova u bajtu plus 2 (+1 umesto zaokruživanja na gore, +1 za terminirajuću nulu, jer se radi o nizu karaktera).

Ovim završavam izlaganje o postupku transformacije GA na CUDA paralelni algoritam. Od elemenata CUDA API-ja koji su bitni, a nisam ih izložio izdvaja se korišćenje konstantne memorije. Nju sam upotrebio za podatke koji su bili često referisani, a pritom nisu promenjlive prirode. Radi se o strukturi problem i informacijama o granama i njihovim težinama. Brzina konstantne memorije je manja od brzine deljene ili registarske memorije, međutim zbog velike količine ovih podataka, konstantna memorija je bila adekvatan kompromis. Druga stvar koju nisam pomenuo, a interesantna je, je tzv. bitonik-sort algoritam (eng. bitonic sort). On zapravo predstavlja sortirajuću mrežu, a konceptualno je dosta sličan mergesort-u kod sekvencijalnih algoritama. Bitonik-sort sam preuzeo sa interneta i prilagodio svojim potrebama.

5. EKSPERIMENTALNI REZULTATI

Glavna odrednica u organizovanju korektnog i efikasnog testiranja mi je bila uporedivost. Sam razvoj aplikacije je pratio tu neophodnu karakteristiku bilo kakve komparativne studije. Koristio sam transformacije koje ne unapređuju CUDA algoritam u odnosu na sekvencijalni konceptualno, već ga samo paralelizuju. Npr. jedan od glavnih nosilaca vremenske složenosti je sortirajući algoritam, koji se izvršava dva puta u okviru svake iteracije. Naime, jedanput nakon izračunavanja funkcije cilja, a drugi put nakon uklanjana duplikata. Merenje ne bi bilo korektno, ukoliko bi za sortiranje kod sekvencijalnog koristili $O(n^2)$ algoritam, npr. sortiranje selekcijom (eng. selection sort), a kod paralelne implementacije neki brži algoritam, npr. bitonik ili brzo sortiranje (eng. quicksort), složenosti $O(n\log(n))$, $O(n\log(n)^2)$ respektivno. Iz tog razloga ovde je korišćeno brzo sortiranje u sekvencijalnom i bitonik sortiranje u paralenom algoritmu.

Redukcije memorije koje su opisane ranije, ne smanjuju vremensku složenost paralelnog algoritma. One su jednostavno proistekle iz potrebe da se uštedi na memoriji, kako bi aplikacija mogla da rešava i probleme malo više dimenzije. Transformacija niza karaktera, koji su korišćeni za opisivanje skupa grana grafa na bitovsku reprezentaciju je doprinela smanjivanju potrebne deljene memorije za 8 puta. Pritom se vremenska složenost nije promenila, konstantan je pristup elementima niza i u jednom i u drugom slučaju.

a. Instance problema i parametri aplikacije

Korišćene su slučajno generisane simetrične instance sa 10, 45, 105, 190, 300, 435, 595, 780 i 990 grana (što su ustvari kompletni neorijentisani težinski grafovi sa 5, 10, 15, 20, 25, 30, 35, 40 i 45 čvorova). Težine grana su celobrojne i pripadaju intervalu [0,10000]. Korišćenjem parametra komandne linije –**graphdensity-** moguće je podesiti prosečan broj grana u generisanju inicijalne populacije, uglavnom je postavljen na 0.5. Kriterijum izlaska iz programa je fiksiran na broj iteracija, u ovom slučaju 5000. Izuzetak je nekoliko situacija kada sekvencijalni algoritam nije mogao da završi u razumnom vremenu, tako da je na tim mestima kriterijum izlaska smanjen na 1000 iteracija. Napravljene su po 3 izvršne datoteke za sekvencijalni i CUDA algoritam, sa različitim brojem hromozoma u populaciji: 128, 256, 384. Veličina populacije se, dakle, nije prosleđivala kao parametar, jer je rad sa dinamičkom deljenom memorijom problematičan.

b. Rezultati

Najpre su obavljeni eksperimenti sa populacijom od 256 hromozoma. Na ovoj aplikaciji testirane su instance od 10, 45, 105, 190, 300 i 435 grana. S obzirom na jednostavnu funkciju cilja, koja je u slučaju simetričnog SMETP-a proveravanje poveznosti BFS algoritmom, koji je složenosti $O(|V|+|E|)$, tj. linearan u odnosu na broj grana i čvorova, ispostavilo se da dostignuta ubrzanja nisu na zavidnom nivou. Iz tog razloga uvedene su dodatni veštački „pojačivači“ vremenske složenosti funkcije cilja na sledeći način:

```
__device__ int Pvalid(unsigned char *lbit, int P, int tid){
          if(P>0){
              int complexity=1;
              for(int p=0; p<P;p++)
                     complexity*=probNn;
              int sum=0;
              for(int i=0; i<complexity; i++){
                     if(k_rand(tid)%2==0)
                            sum+=(k_rand(tid)%100);
                     else
                            sum-=(k_rand(tid)%100);
              }
```

```
            }
            return valid(lbit, tid);
}
```

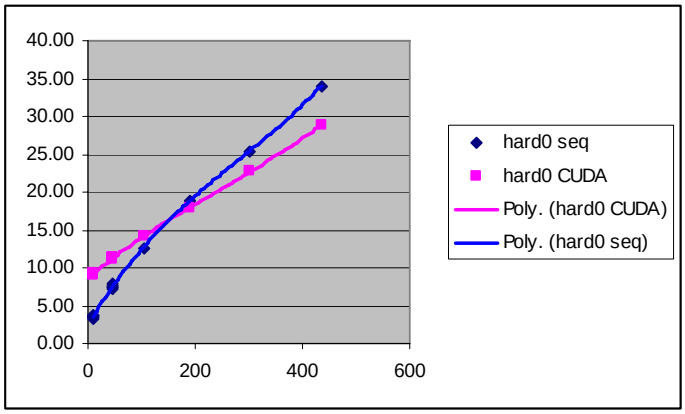
Identična funkcija, uvedena je i u sekvencijalnom algoritmu. Ideja je da se pored provere povezanosti (poslednja naredba), vrši i redudantna petlja od $|V|^P$ iteracija, gde se P prosleđuje kao parametar. Rezultati koji će sada biti predstavljeni se karakterišu test instancom, brojem hromozoma u populaciji (256, za 128 i 384 nisu predstavljeni, ali se nalaze u priloženim datotekama) i težinom funkcije cilja (**hard0**, ako je samo proveravana povezanost; **hard1**, ako je izvršavana redudantna petlja linearna po broju čvorova; **hard2**, kvadratna po broju čvorova i najviše **hard3**, tj. kubna). Za svaku od varijanti je data funkcija cilja koju je dostigao najbolji hromozom, kolona „najb.“, i vreme ukupnog izvršavanja „uk. vr“. Vremena dostizanja najbolje funkcije cilja nisu prikazana, jer nisu toliko relevantna za prikaz ubrzanja algoritma, već za kvalitet konvergencije. Detaljni rezultati izvršavanja se mogu naći u priloženim datotekama.

Tabela 1. Instance veličine između 10 i 435 grana. Izvršeno u 5000 iteracija sa 256 hromozoma.

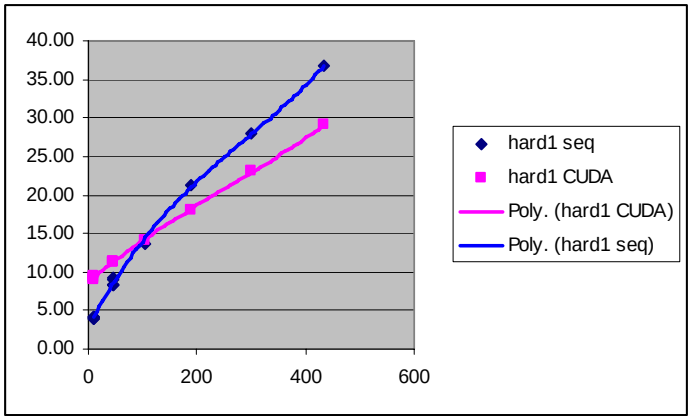
		Sekvencijalni		CUDA		sekvencijalni		CUDA		
		hard0		hard0		hard1		hard1		
instance	|E|	najb.	uk. vr	najb.	uk. vr	najb.	uk. vr	najb.	uk. vr	opt
sim_t5_112	10	**1667**	3.65	**1667**	9.22	**1667**	4.21	**1667**	9.41	1667
sim_t5_342	10	**1468**	3.66	**1468**	9.09	**1468**	4.15	**1468**	9.20	1468
sim_t5_368	10	**2676**	3.59	**2676**	9.30	**2676**	4.19	**2676**	9.09	2676
sim_t5_394	10	**1693**	3.34	**1693**	9.16	**1693**	3.91	**1693**	9.11	1693
sim_t5_651	10	**1800**	3.40	**1800**	9.16	**1800**	3.90	**1800**	9.14	1800
sim_t10_167	45	2915	7.77	2915	11.30	2915	9.13	2915	11.41	2808
sim_t10_288	45	2586	7.11	2586	11.27	2586	8.40	2586	11.36	2576
sim_t10_409	45	2493	7.43	2493	11.30	2493	8.41	2493	11.38	2439
sim_t10_530	45	2347	7.81	**2243**	11.28	2347	8.33	**2243**	11.39	2243
sim_t10_652	45	2648	7.35	2648	11.27	2788	8.99	2648	11.33	2562
sim_t15_128	105	3290	12.58	3278	14.09	3295	13.58	3278	14.16	-
sim_t20_139	190	3652	18.80	3728	17.80	3647	21.32	3688	17.92	-
sim_t25_169	300	4018	25.45	3956	22.75	4024	27.87	4005	23.08	-
sim_t30_199	435	5255	34.04	5284	28.80	5169	36.84	5339	29.03	-
		hard2		hard2		hard3		hard3		
	|E|	najb.	uk. vr	najb.	uk. vr	najb.	uk. vr	najb.	uk. vr	opt
sim_t5_112	10	**1667**	6.72	**1667**	9.58	**1667**	17.75	**1667**	9.91	1667
sim_t5_342	10	**1468**	6.65	**1468**	9.34	**1468**	18.75	**1468**	9.81	1468
sim_t5_368	10	**2676**	6.58	**2676**	9.25	**2676**	18.84	**2676**	9.89	2676
sim_t5_394	10	**1693**	6.04	**1693**	9.27	**1693**	17.22	**1693**	9.80	1693
sim_t5_651	10	**1800**	6.04	**1800**	9.27	**1800**	17.10	**1800**	9.78	1800
sim_t10_167	45	2915	17.29	2915	11.78	2915	109.58	2915	16.45	2808
sim_t10_288	45	2586	17.56	2586	11.78	2586	117.89	2586	16.41	2576
sim_t10_409	45	2493	17.87	2493	11.86	2578	123.08	2493	16.42	2439
sim_t10_530	45	**2243**	19.28	**2243**	11.84	2243	123.71	**2243**	16.41	2243
sim_t10_652	45	2648	18.77	2648	11.80	2648	132.47	2648	16.39	2562
sim_t15_128	105	3332	33.52	3278	15.23	3538	373.31	3278	31.08	-
sim_t20_139	190	3693	55.00	3581	19.83	3703	849.64	3634	58.52	-
sim_t25_169	300	4005	77.78	3988	25.97	3993	1516.55	3971	100.34	-
sim_t30_199	435	5247	106.46	5408	33.34	*5853	*414.11	5231	160.56	-

(*) Izvršeno u 1000 iteracija umesto u 5000. Na grafiku koji sledi vreme ove instance je pomnoženo faktorom 5.

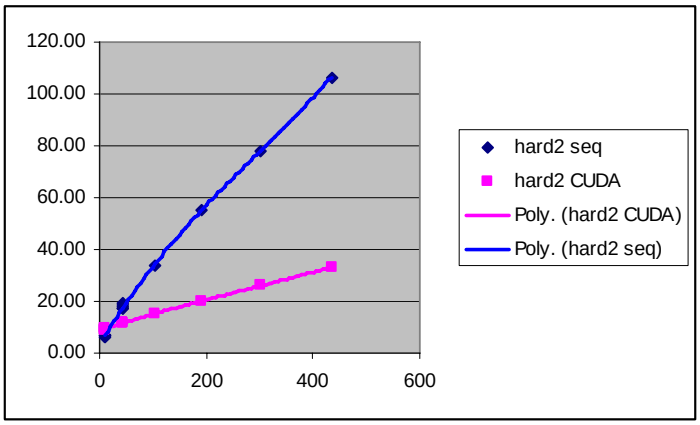
U Tabeli 1. vidimo najpre, da za najmanje instance (10 grana), oba algoritma dostižu optimalna rešenja (provera optimalnosti je izvršena pomoću CPLEX-a). Takođe vidimo da su dobijene vrednosti skoro na svim instancama dosta bliske, što je svakako posledica konceptualno istih algoritama (ako izuzmemo paralelizaciju). U idealnom slučaju, ova rešenja bi se potpuno poklopila, jer su niti adekvatno sihronizovane. Ipak, nepotpuno poklapanje rešenja sekvencijalnog algoritma i CUDA algoritma je najverovatnije posledica korišćenja različitih generatora slučajnih brojeva. Kada je u pitanju vremenska efikasnost, vidimo da za bazičnu funkciju cilja (**hard0**) i linearno pojačanu (**hard1**, što opet implicira linearnu složenost funkcije cilja), CUDA „prestigne“ sekvencijalni algoritam tek kod instance sa 190 grana. Kod **hard2** instanci CUDA je brža za oko 46% na primeru sa 45 grana, a na 435 grana za oko 220%. Konačno na instancama koje su otežane za $O(|V|^3)$, CUDA je već u startu bolja za 80%, poboljšanje dostiže 1400% na primeru sa 300 grana. Za graf sa 435 grana, **hard3** rešenje nije moglo biti pronađeno u „razumnom“ vremenu, tako da je izvršen algoritam u samo 1000 iteracija. Pritom je vreme izvršavanja iznosilo 414 sekundi. Procenjena dužina trajanja algoritma u 5000 iteracija je oko 2000 sekundi, tj. za 1150% sporije od CUDA. Postavlja se pitanje, da li to znači da se CUDA ubrzanje u ovim okvirima i stabilizuje? Na slikama 7, 8, 9 i 10 predstavljene su zavisnosti vremena izvršavanja od broja grana posmatranog grafa, za funkcije cilja različite težine. Primećujemo da se sa porastom težine funkcije cilja povećava i poboljšanje CUDA algoritma u odnosu na sekvencijalni algoritam.



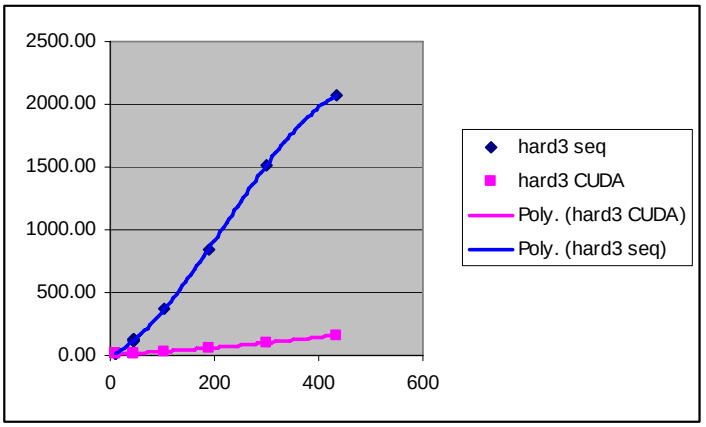
Slika 7. Normalna funkcija cilja (linearna složenost)



Slika 8. Linearno pojačana funkcija cilja (linearna složenost)

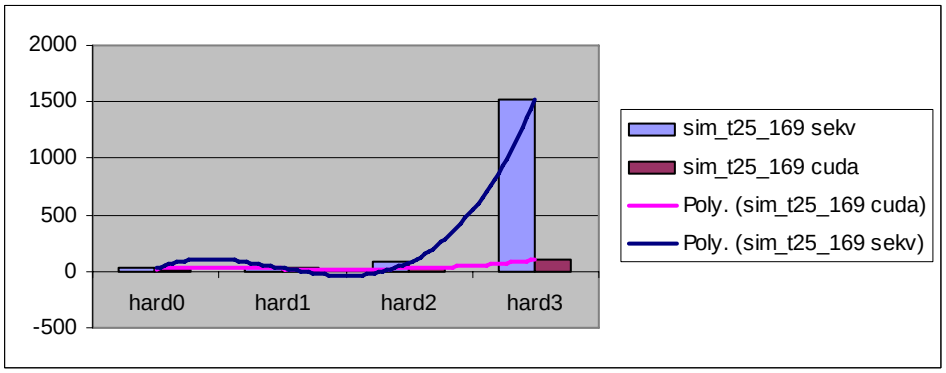


Slika 9. Kvadratno pojačana funkcija cilja (kvadratna složenost)



Slika 10. Kubno pojačana funkcija cilja (kubna složenost)

Na Slici 11. se može videti zavisnost vremena izvršavanja od stepena težine funkcije cilja. Fiksirana je instanca sa 300 grana (sim_t25_369), a zatim rešena korišćenjem sekvencijalnog i CUDA algoritma za sve 4 težine funkcije cilja. Primećuje se značajno ubrzanje CUDA algoritma u odnosu na sekvencijalni kod funkcije cilja sa kubnom vremenskom složenošću.



Slika 11. Zavisnost vremena izvršavanja od težine funkcije cilja

Primećeno je i da se povećanjem populacije hromozoma, za istu instancu problema vreme izvršavanja sekvencijalnog algoritma naglo povećava, dok CUDA GA ostaje skalabilniji.

6. ZAKLJUČAK

U ovom dokumentu su izloženi neki osnovni principi programiranja na CUDA API-ju, kroz implementaciju paralelnog genetskog algoritma. Predstavljeni rezultati su predočili da paralelizaciju ne treba vršiti po svaku cenu, tj. da li će do poboljšanja doći ne zavisi samo od načina implementacije, već i od specifične prirode problema koji rešavamo. Uvođenjem veštačkih funkcija cilja date složenosti utvrđeno je da se dobit od paralelizacije povećava otežavanjem iste. Tako su za linearno složene funkcije cilja, poboljšanja gotovo i neprimetna na ovoj platformi, dok za kubnu složenost funkcije cilja dobijamo red ubrzanja i od 15 puta. Trend poboljšanja bi se verovatno nastavio uvođenjem funkcije cilja reda $O(n^4)$, ipak dužine izvršavanja su postale poprilično dugotrajne, tako da sam se zaustavio sa testiranjem na složenosti $O(n^3)$. Drugi faktor koji je bitan u aspektu paralelizacije GA je veličina populacije. Utvrđena je očekivana pozitivna korelacija između dostignutog ubrzanja i veličine populacije hromozoma. Konačno veličina instance je takođe imala efekta, što je opet logično, jer veći graf povećava i vreme izvršavanja funkcije cilja, tako da se to svodi pod prvi zaključak. Jednom rečju, veća dimenzionalnost problema znači veći benefit od paralelizacije, jer na taj način troškovi kreiranja niti i rada sa memorijom postaju sve zanemarljiviji u odnosu na računarske zahteve drugih delova algoritma: funkcije cilja pre svega, ali i sortiranja populacije, selekcije itd.

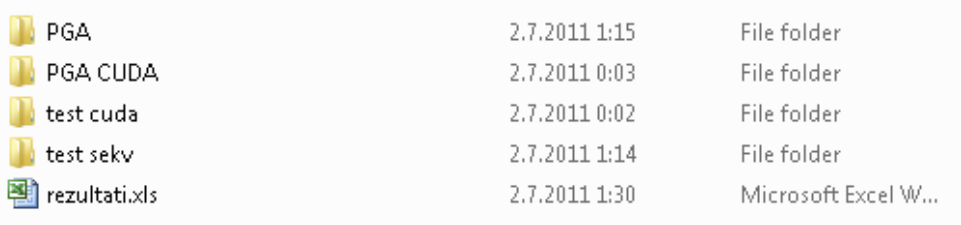
a. Budući rad

CUDA počev od verzije 1.2 (trenutno su aktuelne verzije 2.x) nudi rad sa atomičkim operacijama nad deljenom memorijom. One omogućavaju veću fleksibilnost u kreiranju kooperativnih zadataka između niti. Jedna od mogućih primena u CUDA GA bi bio rad sa zaleđenim bitovima, koji održavaju diversifikovanost populacije, jer ne dopuštaju da se neki gen izgubi iz čitave populacije. Međutim, Nvidia 8400 GS ne podržava 1.2 standarde (već „stari“ 1.1), tako da je ova ideja ostala da se realizuje u budućnosti.

Ostaje ideja o transformaciji postojećeg paralelnog algoritma na model ostrva, u kojem bi se radilo sa više blokova niti, i na taj način povećalo iskorišćenje GPU. Svakom bloku be se npr. dodelila jedna populacija, koja bi se razvijala izolovano od drugih, a zatim povremeno razmenjivala genetski materijal sa drugim populacijama. Problem koji bi se verovatno javio u ovakvom modelu GA je kvalitet dobijenih rešenja: više manjih populacija bi verovatno proizvelo lošija rešenja nego jedna velika, tako da bi ovde morao da se napravi vrlo fleksibilan mehanizam parametrizacije. Sa druge strane, brzina bi korišćenjem ovog modela mogla da bude višestruko povećana. Takođe ću pokušati da nađem neke klase problema koje su pogodnije za paralelno izvršavanje.

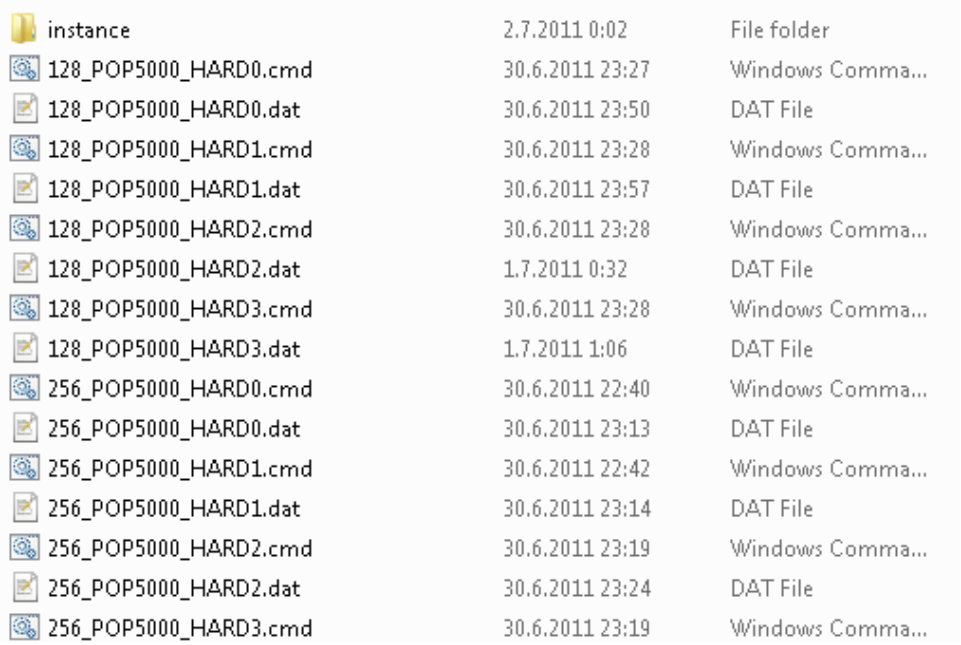
7. DODATAK

Sve datoteke, uključujući i ovaj dokument nalaze se u arhivi **pga.zip**. Nakon što je otpakujete struktura direktorijuma bi trebala da izgleda kao na Slici 12:



Slika 12.

U direktorijumu **„PGA“** se nalazi kod sekvencijalnog algoritma koji je kompajliran u VS2010, tako da verovatno neće moći da se otvori iz neke starije verzije Visual Studio-a. U tom slučaju predlažem da se jednostavno kreira konzolni C++ projekat u okruženju koje imate, pa naknadno dodajte datoteke sa izvornim kodom. U direktorijumu **„PGA CUDA“,** nalazi se kod paralelnog algoritma. Projekat je napravljen u VS2005, a da bi se kompajlirao morate postaviti i CUDA okruženje i preporučljivo je imati i adekvatnu Nvidia karticu. Postoji i režim rada u emulaciji, koji simulira CUDA izvršavanje, ali je po mom saznanju vrlo spor, a dodatno je naravno nemoguće testirati performanse paralelnog izvršavanja emulacijom. Direktorijumi **„test cuda“** i **„test sekv“** sadrže po tri izvršna programa sa različitim veličinama populacija hromozoma. U njemu se nalaze i batch skriptovi, koji su služili za pokretanje programa nad različitim skupom parametara (Slika 13).



Slika 13. Batch skriptovi za pokretanje

Program se može pokrenuti i pojedinačno iz komandne linije:

```
<prog.exe> <ulFajl> <seed> <brGen> <tezina[0-N]> <gustGrafa[0,1]>
```

Npr, jedan mogući poziv sekvencijalno programa bi bio:

```
PGA_128 „instance/gsmet_sim_t10_167.txt“ 55343 1000 2 0.3
```

Što odgovara pokretanju sekvencijalnog program za instancu od 10 čvorova tj. 45 grana, gde je inicijalizator slučajnih brojeva (eng. rand seed) postavljen na 5534, broj generacija je 1000, težina funkcije cilja $O(n^2)$, a očekivana gustina grafa 30%.

8. KORIŠĆENA LITERATURA

Navodim samo najbitnije resurse, one koji su imali značajnijeg uticaja na razvoj aplikacije.

Knjige:

- David Kirk and Wen-mei Hwu, "Programming Massively Parallel Processors, A Hands-on Approach".
- Jason Sanders and Edward Kandrot, "CUDA by example – an introduction to general-purpose GPU programming".
- Nvidia, "Nvidia CUDA™ - Programming guide".
- Nvidia, "CUDA Technical Training - Volume I: Introduction to CUDA Programming".
-

Radovi i naučni resursi:

- Qizhi Yu, Chongcheng Chen, and Zhigeng Pan, "Parallel Genetic Algorithms on Programmable Graphics Hardware".
- Pawan Harish and P. J. Narayanan: "Accelerating large graph algorithms on the GPU using CUDA".
- Daniele G. Spampinato, Anne C. Elster and Thorvald Natvig, "Modelling Multi-GPU Systems".
- Petr Pospíchal, "GPU-based acceleration of the genetic algorithm"

Internet resursi:

- Dr. Dobbs.
- Nvidia CUDA forum.
- Nvidia SDK samples.
- Stackoverflow forum.
- Wikipedia.