UNIVERZITET U BEOGRADU - ELEKTROTEHNIČKI FAKULTET

MULTIPROCESORKI SISTEMI (13S114MUPS, 13E114MUPS)

DOMAĆI ZADATAK 3 – CUDA

Izveštaj o urađenom domaćem zadatku

Predmetni asistent:

doc. dr Marko Mišić

Kandidati:

Marija Kostić 2015/0096

Stefan Milanović 2015/0361

Beograd, januar 2019.

SADRŽAJ

1. PROBLEM 1 - SGEMM

U okviru ovog poglavlja je dat kratak izveštaj u vezi rešenja zadatog problema 1.

1.1. Tekst problema

Paralelizovati program koji vrši jednostavno generalizovano množenje matrica u jednostrukoj preciznosti *Single precision floating General Matrix Multiply* (SGEMM). SGEMM operacije je definisana sledećom formom:

$$C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C$$

Program se nalazi u datoteci sgemm.c u arhivi koja je priložena uz ovaj dokument. Proces sa rangom 0 treba da učita ulazne podatke, raspodeli posao ostalim procesima, na kraju prikupi dobijene rezultate i ravnopravno učestvuje u obradi. Za razmenu podataka, koristiti rutine za kolektivnu komunikaciju. Program testirati sa parametrima koji su dati u datoteci run. [1, N]

1.2. Delovi koje treba paralelizovati

1.2.1. Diskusija

Kod koji rešava ovaj problem se može razbiti na tri velike celine – najpre se vrši učitavanje ulaznih matrica iz dva tekstualna fajla, vrši se obrada ulaznih podataka u vidu kreiranja izlazne matrice koja predstavlja proizvod unetih matrica, i na samom kraju se vrši ispis izlazne matrice u tekstualni fajl.

Prvu i treću celinu nije moguće paralelizovati – ovi delovi koda predstavljaju rad sa I/O sistemom koji se mora obaviti isključivo unutar jedne niti.

Jedini deo koda nad kojim se može vršiti paralelizacija jeste deo koda koji radi obradu, tj. funkcija *void basicSgemm(args)*. U skladu sa postavkom zadatka, paralelizacija će se vršiti na dva načina – prvi način deli posao ravnopravno između procesa (uključujući i master proces, koji vrši učitavanje i prikupljanje rezultata, a takođe vrši i obradu), dok drugi način podrazumeva korišćenje *manager-worker* modela, pri čemu master predstavlja proces gospodara (menadžera).

1.2.2. Način paralelizacije

Zadatak 1

Rešenje ovog zadatka nalazi se u datoteci *dz3z1.cpp*. U ovom zadatku svaka nit koja se izvršava na grafičkoj kartici obrađuje jedan element rezultujuće matrice.

Ovo je postignuto tako što svi procesi pozivaju kernel funkciju *basicSgemm_par().* Unutar ove funkcije svaki proces najpre odredi četiri identifikatora koje koristi – koordinate bloka kojem pripada u okviru rešetke (ovaj broj zavisi od ukupnog broja blokova i pozicije bloka unutar rešetke), i same njegove koordinate unutar bloka (broj od 0 do 1023). Nakon ovoga svaka nit u *shared* memoriju smesti odgovarajući element iz originalne matrice. Ovo omogućava brz pristup ovim elementima od strane ostalih niti u okviru istog bloka. Nakon što se dohvate elementi matrice, radi se sinhronizacija (ova sinhronizacija se vrši na nivou bloka). Time se garantuje da će sve niti imati u *shared* memoriji sve podatke koje su im neophodne za računicu. Sada svaka nit za sebe računa vrednost parametra *sum*. Na kraju glavne *for* petlje će sve niti izračunati svoju vrednost *sum* promenljive. Nakon ovoga svaka nit u globalnu linearizovanu matricu upiše izračunatu vrednost. Kada sve niti iz svih blokova završe ovaj korak, u linearizovanoj matrici *cudaC* će se naći rezultat programa.

Na samom kraju neophodno je samo kopirati nazad rezultujuću matricu iz globalne memorije grafičke kartice na host procesor koji je pozvao kernel kod. Na samom procesoru osim pripreme podataka (i kopiranja na grafičku karticu), poziva kernela, i zatim kopiranja rezultata nazad u adresni prostor procesora, nije neophodno ništa više uraditi. Jedino što se sada radi je proveravanje validnosti rešenja i ispis samog rezultata.

1.3. Rezultati

U okviru ove sekcije su izloženi rezultati paralelizacije problema 1.

1.3.1. Logovi izvršavanja

Ovde su dati logovi izvršavanja za definisane test primere.

Listing 1. Zadatak 1

```
Opening file:data/small/input/matrix1.txt
Matrix dimension: 128x96
Opening file:data/small/input/matrix2t.txt
```

Matrix dimension: 160x96

Opening file:result_small_par.txt for write.

Matrix dimension: 128x160

**********DZ3Z1**********

Elapsed time - SEQ: 0.0093673.

Elapsed time - PAR: 0.000979168.

TEST PASSED

****

Opening file:data/medium/input/matrix1.txt

Matrix dimension: 1024x992

Opening file:data/medium/input/matrix2t.txt

Matrix dimension: 1056x992

Opening file:result_medium_par.txt for write.

Matrix dimension: 1024x1056

**********DZ3Z1**********

Elapsed time - SEQ: 13.4862

Elapsed time - PAR: 0.0555644.

TEST PASSED

****

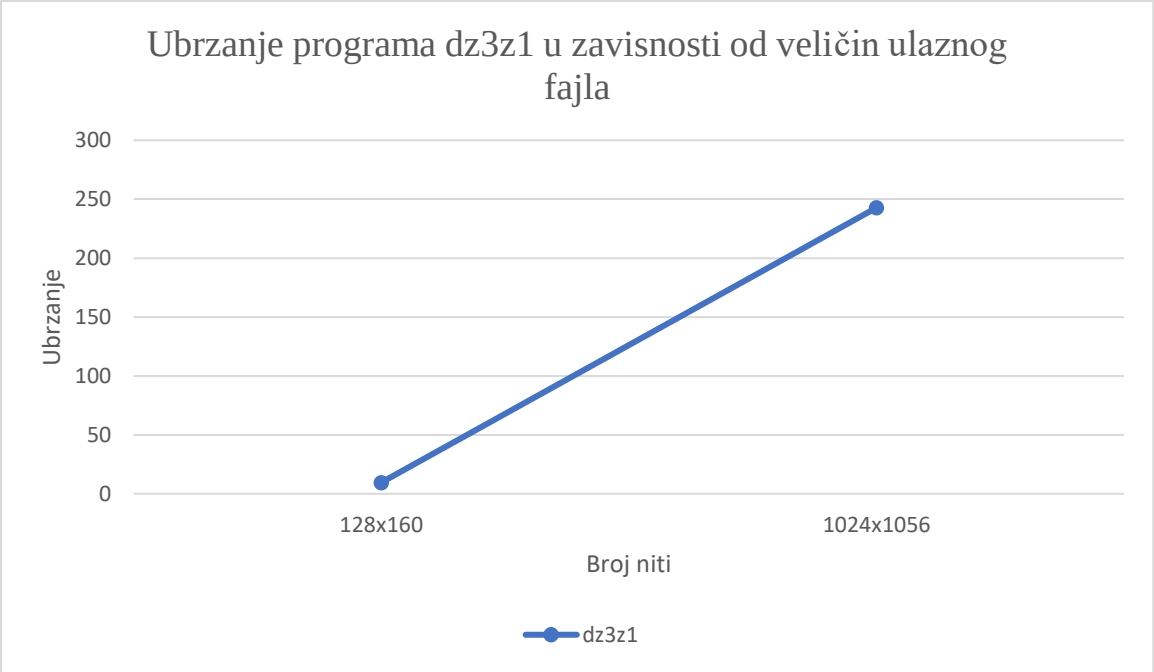
****

1.3.2. Tabela trajanja programa

	small input			medium input		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 1 – 1 nit	0, 0093673	0,000979168	9,56659122	13,4862	0,0555644	242,71296

1.3.3. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja paralelnih implementacija u odnosu na sekvencijalnu implementaciju. Ubrzanje na svim narednim graficima predstavljeno je kao odnos trajanja sekvencijalne implementacije programa i trajanja paralelizovane implementacije.



Cd d

1.3.4. Diskusija dobijenih rezultata

Rezultati se mogu posmatrati tako da se zasebno gledaju izmerene performanse kada se na ulaz sistema dovedu matrice malih dimenzije i kada se na ulaz sistema dovedu matrice većih dimenzija.

Što se manjeg ulaznog fajla tiče, iz dobijenih rezultata se može videti da je ubrzanje oko 10 puta u odnosu na sekvencijalno izvršavanje. Ovo je, u odnosu na paralelizaciju koristeći MPI i OpenMP, ubedljivo najbolje ubrzanje u poređenju. Slično se može primetiti i za sve ostale rezultate ostalih programa sa različitim ulaznim parametrima. Ovo ubrzanje se uvećava sa povećanjem veličine ulaznih matrica.

Za veće ulazne matrice performanse paralelnih programa su mnogo bolje nego kada su ulazi manje matrice i to znatno. Dobijeni rezultati ukazuju na to da je ubrzanje paralelne u odnosu na sekvencijalnu verziju **300 puta.** Ovo daleko prevazilazi dobijena ubrzanja iz prethodnih domaćih zadataka. Ovo je verovatno tako zato što je opravdan odnos izračunavanja računskih operacija i pristupa memoriji od strane niti na grafičkom procesoru, pa je i sama paralelna obrada brža i efikasnija.

2.PROBLEM 2 - JACOBI

U okviru ovog poglavlja je dat kratak izveštaj u vezi rešenja zadatog problema 2.

1.4. Tekst problema

Paralelizovati program koji vrši jednostavno generalizovano množenje matrica u jednostrukoj Paralelizovati program koji rešava sistem linearnih jednačina *A * x = b* Jakobijevim metodom. Kod koji treba paralelizovati se nalazi u datoteci *jacobi.c* u arhivi koja je priložena uz ovaj dokument. Ukoliko je moguće, koristiti rutine za neblokirajuću komunikaciju za razmenu poruka. Program testirati sa parametrima koji su dati u datoteci run. [1, N]

1.5. Delovi koje treba paralelizovati

1.5.1. Diskusija

Za razliku od paralelne implementacije koristeći OpenMP (a analogno kao kod implementacije kod MPI paralelizovanog rešenja), ovde je neophodno više izmeniti način obrade kako bi se omogućila najveća moguća paralelizacija. Opet, moguće je izbeći korišćenje niza *b* u potpunosti – ovaj niz se nikad ne menja (iz njega se samo čita) i od početka programa ima vrednosti 0 za sve elemente niza osim za poslednji, koji sadrži vrednost *n+1*. Samim tim moguće je ovaj niz ukloniti, a zatim koristiti vrednost *n+1* umesto *b[i]* samo u procesu koji obrađuje poslednji element poslednjeg dela niza *x*. Ovim se izbegava *scatter* niza *b* od mastera ka ostalim procesima ili njegova inicijalizacija u svim procesima.

Za razliku od MPI paralelizacije, ovde se ne može izbeći fizičko kopiranje iz niza *xnew* u niz *x* zato što se ovi nizovi nalaze na memoriji na grafičkoj kartici (i podaci koji se nalaze u nizovima su perzistentni između više kernel poziva). Ovo omogućava da se svo kopiranje i stavljanje rezultata iz novog niza u stari radi na grafičkoj kartici.

Inicijalizacija početnog niza *x* se radi na host procesoru, a zatim se kopira tako inicijalizovani niz na grafičku karticu. Kao rezultat neophodno je samo jednom uraditi kopiranje na početku, i samo jednom na kraju obrade – nije neophodno raditi kopiranje u toku svih iteracija.

1.5.2. Način paralelizacije

Paralelizacija ovog problema podeljena je na faze. Sve faze se izvršavaju onoliko puta koliko je zadat broj iteracija u argumentima komandne linije. Pre svih faza, radi se *malloc* neophodnih struktura podataka na grafičkoj kartici (uz eventualni *memcpy* kada je neophodna i inicijalizacija). Nakon svih faza, radi se kopiranje rezultata iz memorije grafičke kartice u memoriju procesora. Faze koje se izvršavaju su ujedno i pozivi kernel funkcija (ovu funkciju izvršava po jedna nit na grafičkom procesoru unutar blokova koji sadrže najviše 1024 niti): *KernelJacobi, KernelOverwrite, KernelCalculateR,* i *KernelReduction.*

KernelJacobi služi za generisanje novog niza *x*_new, uz eventualno generisanje i vrednosti *d* za tu nit. Svaka nit na osnovu svog identifikatora određuje koji element originalnog niza joj pripada i ko su joj susedi. Uz ovo, svaka nit dovlači svoj element iz globalne memorije u *shared* memoriju tog bloka za brži pristup. Ukoliko se radi o niti koja treba da izračuna vrednost graničnog elementa bloka (prvi ili poslednji element), ona će takođe iz globalne memorije dohvatiti vrednost susednog podatka, ukoliko postoji. Nakon računice nove vrednosti elementa niza, ukoliko je neophodno, računa i lokalnu vrednost za d koja se pamti u zasebnom nizu u globalnoj memoriji grafičke kartice. Ova vrednost će se kasnije koristiti za redukciju u vrednost *d* na host procesoru.

KernelOverwrite je kernel funkcija koja služi za prepisivanje niza *xnew* u niz *x.* Ovo se radi isključivo u memoriji grafičke kartice procesora i nije neophodno kopirati ovaj niz nazad na host procesor zato što je memorija grafičke kartice perzistentna između više poziva kernela. Kopiranje nazad rezultujućeg niza u memoriju host procesora radi se jedino kada je završena sva obrada, kada se u nizu *cuda_x* nalazi rezultat.

KernelCalculateR je kernel funkcija koja služi za računanje vrednosti *r* i funkcioniše na skoro isti način kao i funkcija KernelJacobi, samo što se sada vrši računanje vrednosti *r* za svaku nit i ona se smešta u privremeni, zasebni niz koji se nalazi u memoriji grafičke kartice. Radiće se redukcija i ovog niza kako bi se dobila konačna vrednost promenljive *r* za tu iteraciju.

KernelReduction je kernel funkcija koja služi za redukciju vrednosti *d* i *r* koje je neophodno vratiti nakon redukcije host procesoru. Pošto su oba privremena niza koja čuvaju međurezultate ove redukcije veliki ne može se redukcija svesti na jednu vrednost pomoću

samo jednog poziva kernel koda. Ovde se problem može rešiti na dva načina: može se ponovo pozivati kernel kod sve dok se ne dobije jedna konkretna vrednost (koja ujedno predstavlja i rezultat redukcije), ili se može vratiti dosta kraći niz međurezultata procesoru koji onda kompletira redukciju i kod sebe nalazi jedinstvene vrednosti za *r* i *d.* U ovom zadatku je to tako i urađeno – kada je neophodna redukcija, vrši se jedan poziv kernel koda za redukciju koji oba niza od *n* elemenata svodi na onoliko elemenata koliko ima blokova u rešetki, a zatim se ova dva manja niza kopiraju nazad u memoriju procesora gde se radi konačna redukcija.

Sve ove faze se ponavljaju dok se ne završe sve iteracije. Tada se vrši kopiranje rezultujućeg niza nazad u niz koji se nalazi u adresnom prostoru host procesora, koji onda ispisuje rezultat i proverava validnost rešenja.

1.6. Rezultati

U okviru ove sekcije su izloženi rezultati paralelizacije problema 2.

1.6.1. Logovi izvršavanja

Ovde su dati logovi izvršavanja za definisane test primere.

Listing 1. Zadatak 2

```
**********DZ3Z2**********
Elapsed time - SEQ: 0.104365.
Elapsed time - PAR: 0.137206.
TEST PASSED
**********
**********DZ3Z2**********
Elapsed time - SEQ: 0.33041.
Elapsed time - PAR: 0.2106.
TEST PASSED
**********
**********DZ3Z2**********
Elapsed time - SEQ: 7.51316.
Elapsed time - PAR: 1.04395.
TEST PASSED
**********
**********DZ3Z2**********
Elapsed time - SEQ: 33.6498.
Elapsed time - PAR: 2.09594.
TEST PASSED
```

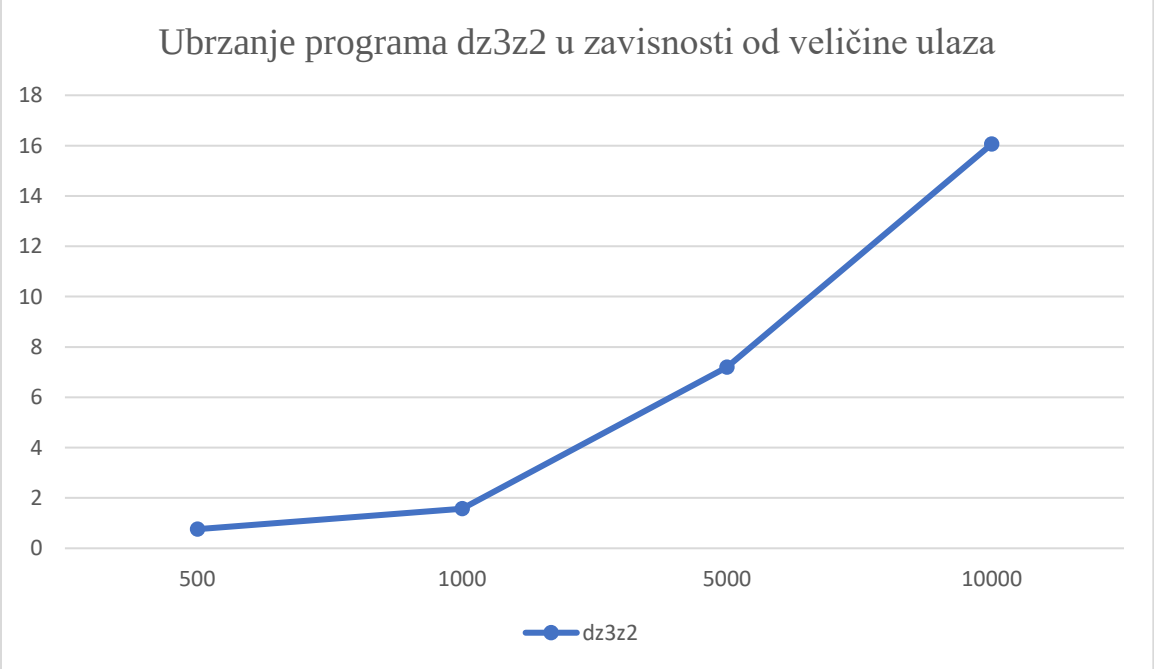
****

1.6.2. Tabela trajanja programa

	input 500			input 1000		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 2	0,104365	0,137206	0.760644	0,33041	0,2106	1,568898
	input 5000			input 10000		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 2	7,51316	1,04395	7,19685	33,6498	2,09594	16,054534

1.6.3. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja paralelnih implementacija u odnosu na sekvencijalnu implementaciju. Ubrzanje na svim narednim graficima predstavljeno je kao odnos trajanja sekvencijalne implementacije programa i trajanja paralelizovane implementacije.



1.6.4. Diskusija dobijenih rezultata

Kao i što je primećeno za prvi problem, i kod *Jacobi* problema CUDA rešenje donosi daleko veća ubrzanja za sve veličine ulaznih podataka.

Kao i kod OpenMP i MPI implementacija, ubrzanje CUDA paralelne implementacije zavisi od veličine niza nad kojim se radi, ali ovde ubrzanje sa dosta većim stepenom raste nego što je to slučaj u prethodna dva domaća zadatka.

Za manje ulazne nizove (parametar *n*) dobija se vreme koje je približno jednako sekvencijalnom vremenu (ali je već sada malo brže u odnosu na sekvencijalnu implementaciju). Kada se posmatraju primeri gde je *n* reda 5 000, 10 000, ili više, primećuje se zaista veliko ubrzanje od **skoro 17x**. Ovo je približno **4.3x veće** ubrzanje nego što je to slučaj kod OpenMP i MPI paralelizovanog rešenja.

3.PROBLEM 3 - KMEANS

U okviru ovog poglavlja je dat kratak izveštaj u vezi rešenja zadatog problema 3.

1.7. Tekst problema

Paralelizovati program koji vrši k-means klasterizaciju podataka. Klasterizacija metodom k-srednjih vrednosti (eng. k-means clustering) je metod koji particioniše n objekata u k klastera u kojem svaki objekat pripada klasteru sa najbližom srednjom vrednošću. Objekat se sastoji od niza vrednosti - osobina (eng. features). Podelom objekata u potklastere, algoritam predstavlja sve objekte pomoću njihovi srednjih vrednosti (tzv. centroida potklastera). Inicijalni centroid za svaki potklaster se bira ili nasumično ili pomoću odgovarajuće heuristike. U svakoj iteratiji, algoritam pridružuje svaki objekat najbližem centroidu na osnovu definisane metrike. Novi centroidi za sledeću iteraciju se izračunavaju usrednjavanjem svih objekata unutar potklastera. Algoritam se izvršava sve dok se makar jedan objekat pomera iz jednog u drugi potklaster. Program se nalazi u direktorijumu kmeans u arhivi koja je priložena uz ovaj dokument. Program se sastoji od više datoteka, od kojih su od interesa datoteke kmeans.c, cluster.c i kmeans_clustering.c. Analizirati dati kod i obratiti pažnju na različite mogućnosti i nivoe na kojima se može obaviti paralelizacija koda, kao i na deo koda za generisanje novih centroida u svakoj iteraciji unutar datoteke kmeans_clustering.c. Ulazni test primeri se nalaze u direktorijumu data, a način pokretanja programa u datoteci run. [1, N]

1.8. Delovi koje treba paralelizovati

1.8.1. Diskusija

Kod koji sekvencijalno rešava ovaj problem razdvojen je u tri fajla – *kmeans.c, cluster.c,* i *kmeans_clustering.*c. Sama *main* funkcija koja se nalazi u *kmeans.c* fajlu samo vrši učitavanje svih neophodnih argumenata iz komandne linije i iz odgovarajućih fajlova i alokaciju prostora, a zatim kroz određeni broj iteracija (trenutno postavljeno na 1) poziva funkciju *cluster* iz *cluster.c* fajla. Pored toga, kod iz *main* funkcije samo vrši ispis i dealokaciju prostora. U paralelnoj verziji, većinu ovog posla radi proces sa rangom master. On će pokrenuti sekvencijalnu implementaciju programa i zapamtiti vreme izvršavanja, a zatim će odgovarajuće vrednosti pomoću funkcije *broadcast* proslediti i ostalim procesima koji rade

obradu. Na kraju, proces master vrši ispis dobijenih rezultata i upoređuje validnost dobijenih vrednosti sa onim dobijenih iz sekvencijalne implementacije, a i ispisuje vremena izvršavanja obe implementacije.

Funkcija *cluster* iz *cluster.c* fajla je samo *wrapper* funkcija koja alocira niz *membership* koji se koristi u *kmeans_clustering* funkciji, generiše *seed* random generatora, i generiše rezultat tako što poziva funkciju *kmeans_clustering*. Povratna vrednost funkcije *kmeans_clustering* se zatim vraća u *main* funkciju preko promene argumenta *cluster_centres* koji je prosleđen po referenci. U slučaju paralelizacije koristeći MPI svi procesi će pozvati funkciju *cluster_par*, samo će u njoj alocirati *membership* koji odgovara samo objektima koji su dodeljeni tom procesu, a zatim se unutar *kmeans_clustering_par* funkcije vrši komunikacija između mastera i ostalih procesa.

U ovom zadataku najveće ubrzanje dobija se paralelizacijom funkcije koja se u sekvencijalnoj implementaciji nalazi u fajlu *kmeans_clustering.c*. Ovo uključuje paralelizaciju funkcije *float** kmeans_clustering(args).* Funkcije *int find_nearest_point(args), float euclid_dist_2(args)* su pomoćne funkcije koje ne treba paralelizovati. Način na koji je izvršena paralelizacija za navedenu funkciju opisan je u narednom poglavlju.

1.8.2. Način paralelizacije

Kod ovog problema ideja je da se kernel poziva unutar glavne *do-while* petlje koja se nalazi u *kmeans_clustering_par()* funkciji, pri čemu bi host procesor proveravao uslov izlaska iz petlje. Da bi se ovo omogućilo, pre same petlje će se izvršiti inicijalizacija svih neophodnih struktura podataka i njihovo kopiranje u memoriju grafičke kartice, onda će se pokretati odgovarajuće kernel funkcije koje će raditi obradu, pripremu pomoćnih nizova za redukciju, i samu redukciju. Nakon ovih koraka (a pre završetka *do-while* petlje) će se samo new-centers, new_centers_len, i delta vraćati procesoru da bi se utvrdila nova vrednost za *clusters* niz i da bi se utvrdilo da li treba nastaviti dalje sa obradom tako što će se izvršiti konačni deo redukcije vrednosti *delta* i onda to proveriti sa uslovom petlje.

Ukoliko uslov nije ispunjen (to jest, treba ponoviti još jednu iteraciju) vrši se kopiranje *clusters* niza u memoriju grafičkog procesora i ponavlja se izvršavanje sve dok redukovana vrednost *delta* nije manja od vrednosti *threshold.* Tada host procesor proverava validnost rešenja i ispisuje rezultate.

Sama paralelizacija je raspoređena tako da prvi deo originalne *do-while* petlje bude kernel funkcija. To znači da će svaka nit izvršavati ovaj kod, pri čemu ima onoliko niti na GPU koliko postoji objekata u ulaznoj datoteci. Pošto se u ovom kodu računa vrednost promenljive *index* (ona može imati bilo koju vrednost u opsegu od 0 do *num_clusters – 1*) koristeći funkcije *find_nearest_point()* i funkcije *euclid_dist(),* neophodno je ove dve funkcije obezbediti na grafičkom procesoru. Ovo se radi korišćenjem direktiva __*device*__ prilikom definisanja funkcija.

Kako bi se omogućila redukcija, iskoristićemo prostor u globalnoj memoriji grafičke kartice. Naime, neophodno je opet ručno uraditi redukciju. Ovaj put, za razliku od Jacobi problema, nije dovoljna redukcija koja se svodi samo na jednu promenljivu, već je neophodno raditi redukciju čitavog niza (*new_centers_len*) i čitave matrice (*new_centers*). Pošto svaka nit može upisati u bilo koji deo ovog niza i ove matrice, neophodno je da u memoriji grafičke kartice za svaku nit imamo njenu „lokalnu“ kopiju ovog niza i ove matrice. Nakon toga, kada sve niti izvrše obradu, vršimo redukciju za jednu promenljivu (kako bismo našli konačnu vrednost *delta*), za jedan niz (kako bismo našli konačni niz *new_centers_len*), i za jednu matricu (kako bismo našli konačnu matricu *new_centers*). Redukciju u sva tri slučaja završavamo na host procesoru.

Drugi deo glavne *do-while* petlje se može raditi na host procesoru, zato što je veličina korišćenih nizova i matrica jako mali. Ukoliko uslov za izlazak iz glavne petlje nije ispunjen, vraćamo se na vrh petlje kada se ponavlja obrada. Tada se nova izračunata *clusters* matrica vraća u memoriju grafičke kartice kako bi se ponovo radila obrada nad novim, korektnim vrednostima.

1.9. Rezultati

U okviru ove sekcije su izloženi rezultati paralelizacije problema 3.

1.9.1. Logovi izvršavanja

Ovde su dati logovi izvršavanja za definisane test primere i različit broj niti. Obavezno uključiti u ispis i vremena izvršavanja. Logove pojedinačno uokviriti i obeležiti.

Listing 1. Zadatak 3

```
I/O completed
numObjects=100, numclusters=5
```

==========SEQ==========

number of Clusters 5

number of Attributes 34

Time for process: 0.000792

==========PAR==========

number of Clusters 5

number of Attributes 34

Time for process: 0.015899

TEST PASSED

I/O completed

numObjects=494016, numclusters=5

==========SEQ==========

number of Clusters 5

number of Attributes 34

Time for process: 16.420168

==========PAR==========

number of Clusters 5

number of Attributes 34

Time for process: 1.813536

TEST FAILED

I/O completed

numObjects=204796, numclusters=5

==========SEQ==========

number of Clusters 5

number of Attributes 34

Time for process: 3.624799

==========PAR==========

number of Clusters 5

number of Attributes 34

Time for process: 0.365338

TEST PASSED

I/O completed

numObjects=819196, numclusters=5

==========SEQ==========

number of Clusters 5

number of Attributes 34

Time for process: 15.35736

==========PAR==========

number of Clusters 5

number of Attributes 34

Time for process: 1.640451

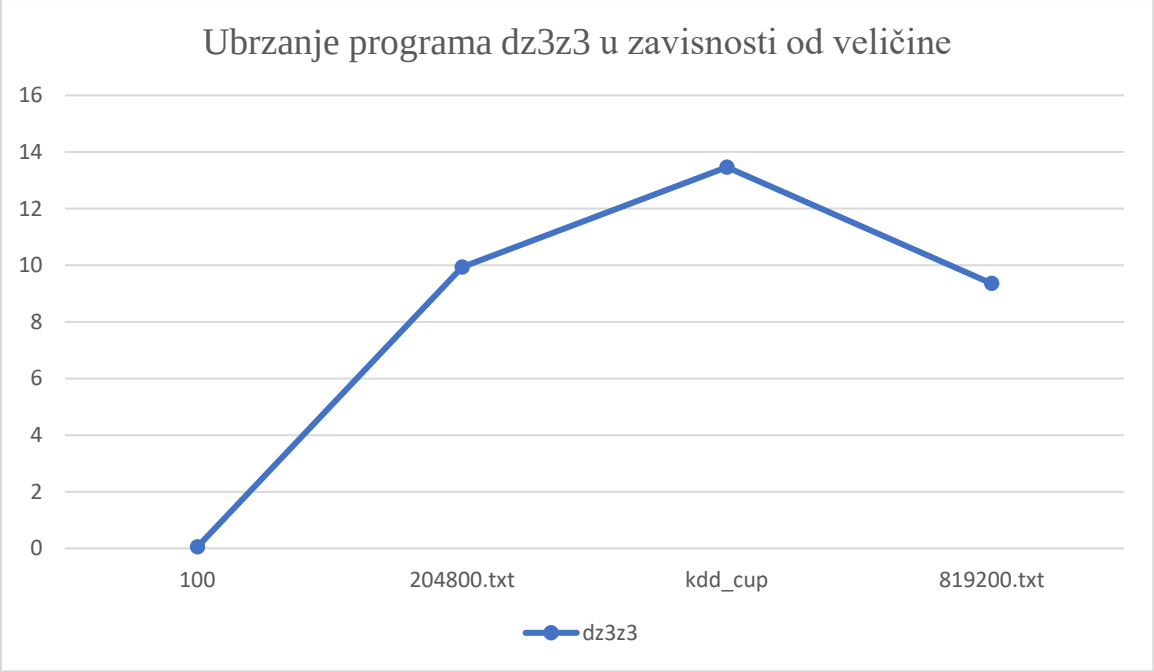
TEST FAILED

1.9.2. Tabela trajanja programa

	100			kdd_cup		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 3	0.000792	0.015899	0.049814	24,41608	1,813536	13.463245
	204800.txt			819200.txt		
	Sekvencijalno vreme	Paralelno vreme	Ubrzanje	Sekvencijalno vreme	Parelelno vreme	Ubrzanje
Zadatak 3	3.627499	0.365338	9.927800	15.35736	1.640451	9.361669

1.9.3. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja paralelnih implementacija u odnosu na sekvencijalnu implementaciju. Ubrzanje na svim narednim graficima predstavljeno je kao odnos trajanja sekvencijalne implementacije programa i trajanja paralelizovane implementacije.



1.9.4. Diskusija dobijenih rezultata

Kao što je slučaj bio u prvom i drugom zadatku, i ovde je dobijena dosta velika paralelizacija za date ulazne primere, pri čemu je ubrzanje opet zavisilo od veličine ulaznih fajlova i podataka. Za male ulazne fajlove je dobijeno usporenje u odnosu na sekvencijalnu

verziju, ali se kasnije za najveće ulazne primere paralelizovani program izvršava **10x do 15x brže.** Dobijeni rezultati ukazuju na to da je uspešno izvršena paralelizacija i da je odnos između računanja i pristupa memoriji niti koje se izvršavaju na grafičkoj kartici dovoljno veliki da je opravdan ovakav način pokretanja kernela. pokretanje mehanizma kreiranja niti i njihovog puštanja u rad opravdano jer su dobici u brzini izvršavanja dosta veliki.