SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Programiranje masivno paralelnih procesora

Josip Hucaljuk

Voditelj: *Mile Šikić*

Zagreb, svibanj 2011.

Sadržaj

1. Uvod 1
2. Arhitektura 2
3. Razvojna okolina 9
4. Razvoj aplikacije 10
5. Rezultati 12
6. Zaključak 13
7. Literatura 14

1. Uvod

Od samih početaka, razvoj grafičkih procesora, kao i ostale računalne elektronike, pratio je poznati Mooreov zakon[1], prema kojemu se svakih 12 mjeseci broj integriranih tranzistora udvostručuje po jedinici površine, te svakih 18 mjeseci udvostručuje računalna snaga. Taj tempo održavao se sve do unazad par godina, kada dolazi do znatnog usporenja. Glavni razlog tome je približavanje proizvođača tehnološkim granicama postojećih proizvodnih procesa, što uzrokuje brojne probleme prilikom dizajniranja i proizvodnje.

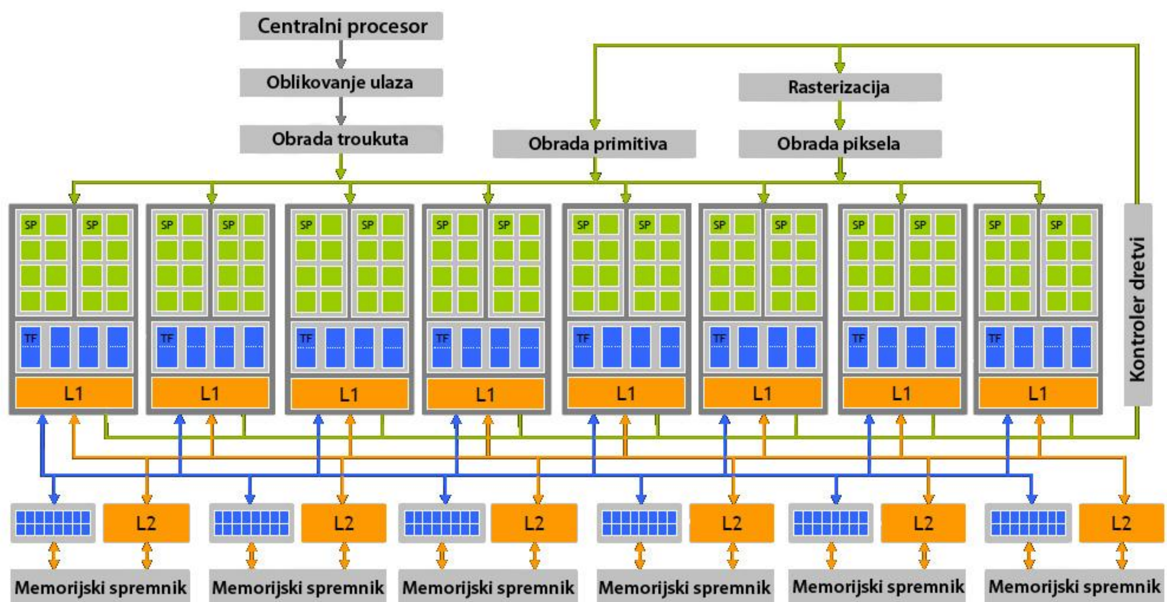
Budući da više nije moguće poboljšavati performanse u željenim koracima, industrija se okreće pružanju veće funkcionalnosti. Tako danas grafički procesori, koji su se do prije par godina koristili isključivo za prikaz računalne grafike, pružaju mogućnost korištenja i u neke opće svrhe. Zbog svoje visoko paralelizirane arhitekture, današnji moderni grafički procesori omogućavaju značajna ubrzanja izvođenja određenih poslova u odnosu na centralni procesor (CPU). Ostvareno ubrzanje omogućava korištenje algoritama koji prije nisu bili primjenjivi zbog vremenskih razloga. U sklopu ovoga rada programski je ostvaren jedan primjer koji demonstrira koje razine ubrzanja se mogu postići izvođenjem koda na grafičkim procesorima.

U sljedećim poglavljima ukratko će biti opisana arhitektura modernih grafičkih procesora (2. poglavlje), razvojna okolina potrebna za razvoj CUDA aplikacija (3. poglavlje), programsko ostvarenje odabranog primjera (4. poglavlje), eksperimentalni rezultati dobiveni izvođenjem ostvarenog programskog rješenja (5. poglavlje), te na kraju i sami zaključak.

[1] Zakon je imenovan po suosnivaču Intel korporacije Gordonu E. Mooreu.

2. Arhitektura

Kao što je rečeno u uvodu, današnje grafičke procesore karakterizira masivno paralelizirana arhitektura. Na slici 2.1 [4] prikazana je konceptualna shema modernog grafičkog procesora.

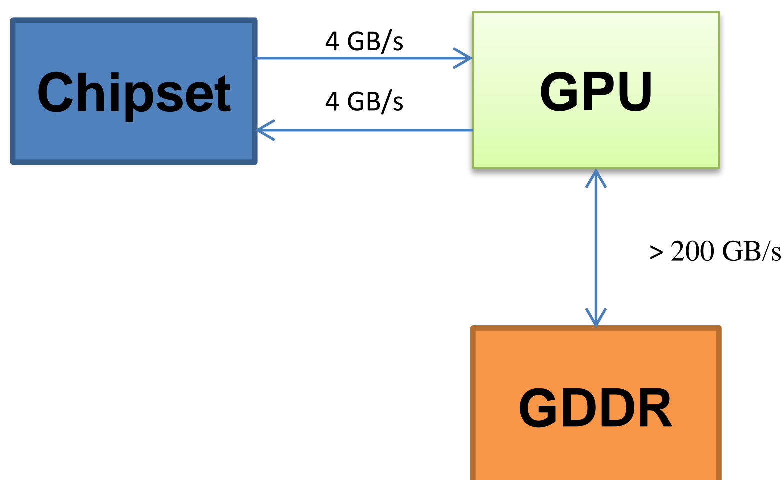


Slika 2.1 - Shema modernog grafičkog procesora

Visoka paraleliziranost grafičkog procesora ostvarena je velikim brojem jednostavnih procesorskih jezgri (eng. *Streaming Processors*). Radi jednostavnijih kontrolnih mehanizama i pristupa memoriji, jezgre su grupirane u veće nakupine (uobičajeno po 32 SP-ova u trenutnoj generaciji) koje se nazivaju multiprocesori (eng. *Streaming Multiprocessors*). Svaka procesorska jezgra sastoji se od 1 jedinice za zbrajanje i množenje (MAD) i 1 jedinice koja vrši samo množenje (MUL). Kad se uzme u obzir da današnje moderne grafičke kartice iz višeg segmenta imaju i do 1024 jezgre, koje rade na frekvencijama preko 1 GHz, lako se može doći do brojke od preko 3 TFLOPS-a (**T**era **Fl**oating Point **Op**erations **P**er **S**econd), odnosno 3 bilijuna operacija sa pomičnim zarezom u sekundi. Takva računalna snaga do sada je bila raspoloživa isključivo na vrlo skupim superračunalima. Detaljniji opis strukture grafičkog procesora može se naći u [1] i [2].

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

Podatci prije obrade moraju se na neki način premjestiti u grafičku memoriju. To se ostvaruje preko PCI Express sučelja, koje naspram propusnosti na relaciji GPU – memorija ima izrazito mali iznos od 4 GB/s u oba smjera. Iako na prvi pogled to se može činiti kao izrazito usko grlo, podatci se preko njega prenose relativno rijetko naspram komunikacije GPU – memorije, te stoga i ne predstavlja poseban problem. Komunikacija sa grafičkim procesorom prikazana je na slici 2.2.



Slika 2.2 – Shema komunikacije grafičkog procesora sa ostalim dijelovima sustava

2.1. CUDA

2.1.1. Općenito

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

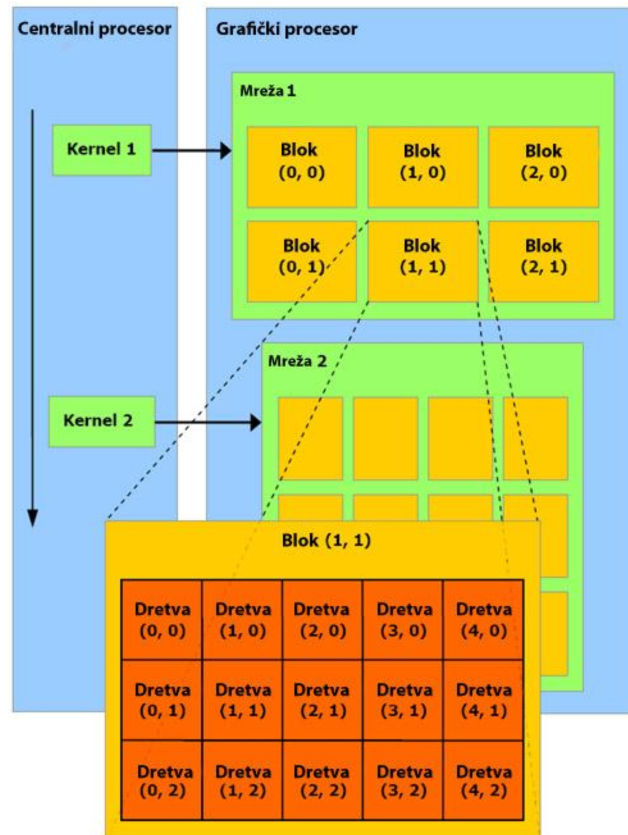
Veliki iskorak u GPGPU industriji dogodio se 2006. godine, predstavljanjem CUDA arhitekture. Po prvi puta dio silicija grafičkog procesora bio je posvećen za olakšavanje GPGPU programiranja. Uvedeno je posebno hardversko sučelje, kao i svi potrebni alati za efikasno paralelno programiranje. Od 2006.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

2.1.2. Logička struktura

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

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

Logička struktura prikazana je na slici 2.3. [3]



Slika 2.3 - Logička organizacija grafičkog procesora

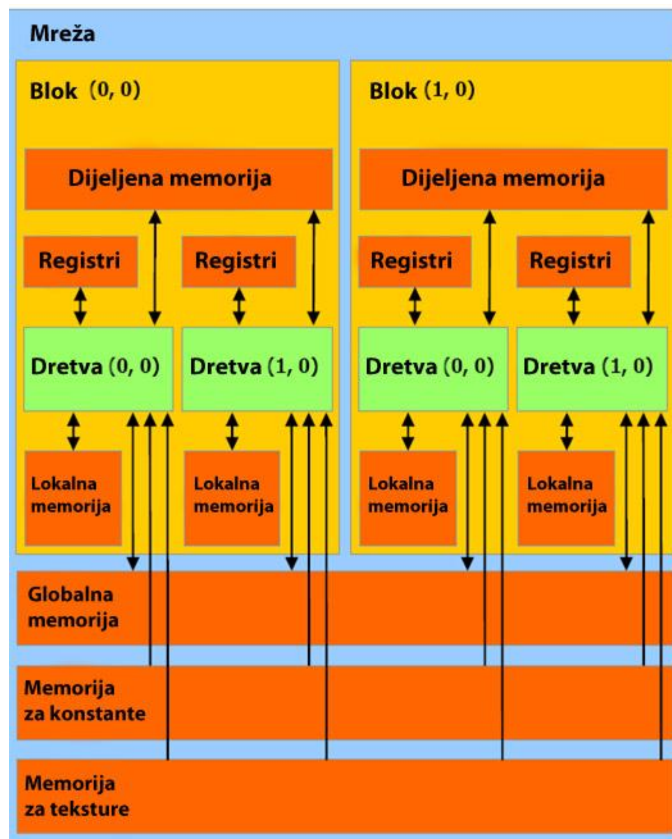
Glavni razlog odabira ovakve logičke strukture su performanse. Na ovaj način maksimizira se iskorištenost procesorskih jezgri, odnosno minimizira vrijeme koje jezgre provode čekajući na obavljanje neke visoko latentne operacije poput pristupa glavnoj memoriji. U trenutku izvršavanja nekog kernela svaki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

2.1.3. Memorijska hijerarhija

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

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

Na slici 2.4 [3] prikazana je shema memorijske hijerarhije. Detaljniji opis dostupan je u članku [3].



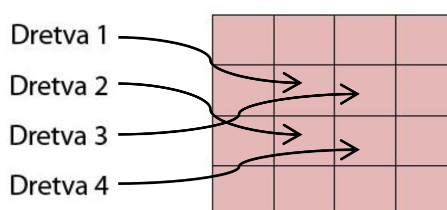
Slika 2.4 - Shema memorijske hijerarhije

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

600 ciklusa). Koristi se za inicijalni prijenos podataka, te za vraćanje rezultata obrade.

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

Još jedan tip memorije koji se može samo čitati od strane grafičkog procesora su teksture. Teksture po svojstvima predstavljaju kombinaciju globalne memorije (veliki kapacitet) i memorije za konstante (indeksirani pristup i isključivo čitanje). Indeksiranje je provedeno na poseban način kako bi ubrzao *lokalizirani* pristup memoriji. Slika 2.5 ilustrira pristup memoriji koji je prikladan za korištenje teksturi.



Slika 2.5 – Lokalizirani pristup memorijskim lokacijama

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

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

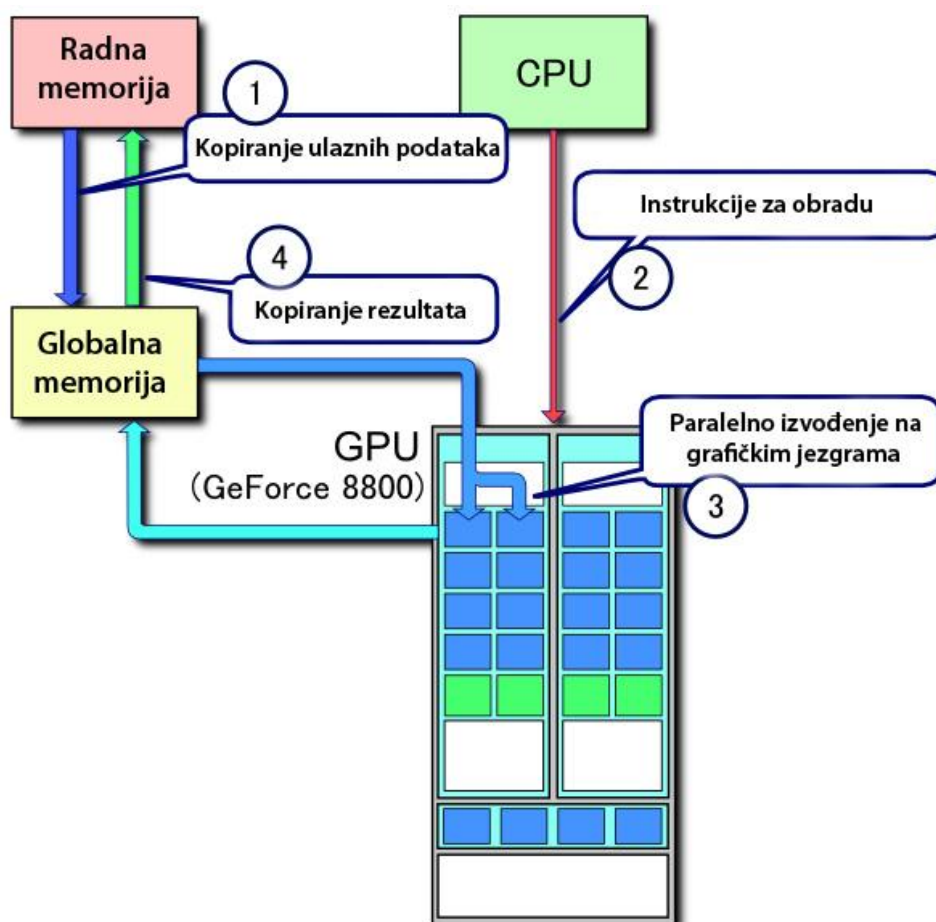
Zadnja, ujedno i najbrža, memorijska lokacija su registri. Registri su definirani na razini bloka dretvi, te ih je ukupno 32768 raspoloživo po bloku. Raspoloživi registri dijele se u jednakim dijelovima među dretvama u bloku. U slučaju da dretve koriste više registara nego što bi smjele, smanjuje se broj raspoloživih warpova koji se mogu izvršavati na jednom multiprocesoru. Svi jednostavni tipovi podataka (char, int, float...) spremaju se u registre, što omogućuje vrlo brzi pristup varijablama (1 ciklus).

Složeni tipovi podataka (polja, stringovi itd.), ako nije drugačije naznačeno, spremaju se u globalnu memoriju (na slici 2.4 označeno kao lokalna memorija), te o tome treba voditi računa prilikom programiranja.

2.1.4. Dijagram toka CUDA aplikacije

Na slici 2.6 [5] prikazan je dijagram toka standardne CUDA aplikacije. Moguće je identificirati sljedeće aktivnosti:

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



Slika 2.6 - Dijagram toka izvođenja CUDA aplikacije

3. Razvojna okolina

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

Prvi i osnovni uvjet bez kojeg nije moguće izvoditi napisane aplikacije je grafička kartica sa CUDA arhitekturom. Većina Nvidia kartica izdanih u zadnje 3 godine podržava CUDA-u, te stoga jedini uvjet na koji treba paziti je da kartica ima barem 256 MB grafičke memorije. Izvođenje je moguće napraviti i na karticama sa manjom količinom memorije, ali samo u slučajevima da se radi o jednostavnim operacijama nad malom količinom podataka.

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

Razvoj CUDA aplikacija vrši se pomoću programskog jezika C sa CUDA specifičnim proširenjima (eng. *C for CUDA*). Odabir razvojne okoline prepušten je korisniku, te u suštini bilo koji C IDE (**I**ntegrated **D**evelopment **E**nvironment), poput MS Visual Studia, poslužit će svrsi. U okviru ovog rada korišten je MS Visual Studio Professional 2008 koji je i službeno preporučen od strane proizvođača. Dodatno, moguće je i instalirati proširenje koje omogućuje debugiranje programskog koda koji se izvršava na grafičkom procesoru (*Nexus Debbuger*).

Prilikom stvaranja novog projekta potrebno je podesiti putanje do gcc i Pathscale prevodioca, putanje do dodatnih biblioteka, kao i još neke manje bitne opcije. Alternativno moguće je instalirati aplikaciju *CUDA VS Wizard*. Pomoću nje, prilikom odabira vrste projekta, moguće je odabrati CUDA aplikaciju kod koje su sve opcije već podešene.

4. Razvoj aplikacije

4.1. Mandelbrotov skup

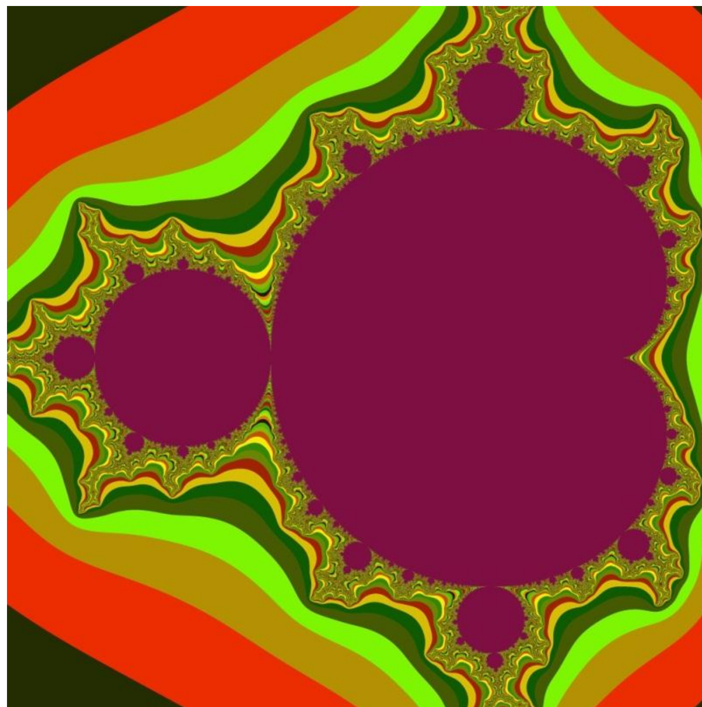
Kako bi se prikazalo potencijalno ubrzanje koje je moguće postići izvođenjem programskog koda na grafičkim procesorima ostvaren je program koji pronalazi elemente Mandelbrotovog skupa.

Mandelbrotov skup čine elementi iz skupa kompleksnih brojeva koji konvergiraju prilikom iterativnog preslikavanja, odnosno matematički zapisano, oni elementi **c** za koje vrijedi da apsolutna vrijednost izraza:

$$z_{n+1} = {z_n}^2 + c, \qquad z_0 = 0$$

konvergira kad se n pusti u beskonačnost.

Mandelbrotov skup vjerojatno je najpoznatiji po svojoj matematičkoj vizualizaciji. Iscrtavanjem elemenata Mandelbrotovog skupa ostvaruju se dvodimenzionalni fraktali vrlo specifičnog izgleda (slika 2.7).



Slika 2.7 Mandelbrotov fraktal

4.2. Programsko ostvarenje

Sljedeći programski odsječak predstavlja programski kod koji provjerava pripadnost nekog kompleksnog broja **c** Mandelbrotovom skupu.

```
__global__ void gpuMandel(unsigned char *data)
{
	int dimX = blockIdx.x * BLOCK_DIM + threadIdx.x;
	int dimY = blockIdx.y * BLOCK_DIM + threadIdx.y;

	float uMin, uMax, vMin, vMax;

	uMin = -1.5;
	uMax = 0.5;
	vMin = -1;
	vMax = 1;

	int k = -1;

	float u = ((uMax - uMin) * dimX) / REZ_X + uMin;
	float v = ((vMax - vMin) * dimY) / REZ_Y + vMin;

	complex c = {u,v};
	complex z = {0,0};

	do
	{
		k++;
		complex kvadrat = multiply2(z,z);
		z.real = kvadrat.real + c.real;
		z.img = kvadrat.img + c.img;
	}while((z.real * z.real + z.img * z.img) < EPS && (k < MAX_ITER));

	data[dimY * REZ_X + dimX] = k;
}
```

Ukupno se treba provjeriti REZ_X * REZ_Y kompleksnih brojeva koji su jednoliko uzorkovani iz područja kompleksne ravnine omeđene varijablama uMin, uMax, vMin i vMax. S obzirom da se svaki broj mora provjeriti i to nezavisno o drugima, problem je trivijalan za paralelizirat. U ovoj implementaciji svaki broj koji treba provjeriti provjerava jedna dretva. Tako u slučaju 16 milijuna brojeva koje želimo provjeriti, imamo 16 milijuna dretvi. Varijable **dimX** i **dimY** određuju relativnu poziciju dretve u 2D prostoru svih dretvi, dok varijable **u** i **v** predstavljaju njihovo preslikavanje u omeđeni kompleksni prostor. Rezultat provjere konvergencije sprema se u polje **data**. Oni elementi polja koji imaju vrijednost manju od (MAX_ITER – 1), predstavljaju točke u kompleksnoj ravnini koje divergiraju, jer apsolutna vrijednost varijable **z** prelazi prag konvergencije EPS.

5. Rezultati

Kako bi se ocijenile performanse ostvarenog programskog rješenja testiranje je provedeno na sljedećim veličinama skupa: 2048 x 2048, 4096 x 4096, 8192 x 8192 i 16384 x 16384. Svaki test je ponovljen 10 puta i kao rezultat je uzeta srednja vrijednost.

Testiranje je provedeno na dvojezgrenom Intelovom centralnom procesoru takta 3,8 GHz, te na grafičkim karticama Nvidia Geforce GTX 460 i Nvidia Geforce GTX 570. Navedene grafičke kartice predstavljaju najpopularnije predstavnike u cjenovnom razredu do 1500 kn, odnosno 3000 kn.

Ostvareni rezultati prikazani su u sljedećoj tablici:

Tablica 1 Rezultati testiranja

Veličina skupa	**CPU [ms]**	**GTX 460 [ms]**	**GTX 570 [ms]**
2048 x 2048	6 830	15,25	7,49
4096 x 4096	27 002	59,97	29,33
8192 x 8192	109 705	237,08	115,75
16384 x 16384	479 815	939,99	458,53

Kao što se može vidjeti iz priloženih rezultata razlika u brzini izvođenja je zaista značajna. U slučaju slabije grafičke kartice vrijeme izvođenja je čak 500 puta kraće nego na centralnom procesoru. Skuplja kartica je još duplo brža te u usporedbi sa centralnim procesorom omogućava ubrzanje i do 1000 puta.

Ipak treba napomenuti da odabrani problem gotovo idealno pogoduje implementaciji na grafičkim karticama zbog svoje male memorijske ovisnosti. Svaki broj se provjerava sa vrlo malom interakcijom sa grafičkom memorijom, koja u većini slučajeva predstavlja usko grlo. Ubrzanja za neke učestalije probleme iz prakse bit će ipak manja i ovisit će ponajviše o samom problemu.

6. Zaključak

U ovome radu ukratko je opisana arhitektura modernih grafičkih procesora te mogućnost njihove primjene za obavljanje poslova opće namjene. Kako bi se demonstriralo ubrzanje koje je moguće postići izvođenjem programskog koda na grafičkom procesoru izrađen je i jedan jednostavan program koji ispituje pripadnost elementa Mandelbrotovom skupu.

Rezultati testiranja implementiranog primjera pokazuju značajno ubrzanje prilikom izvođenja na grafičkom procesoru. U slučaju slabije kartice ostvareno je ubrzanje oko 500 puta, dok kartica iz višeg segmenta ponude ostvaruje ubrzanje i do 1000 puta u odnosu na implementaciju na centralnom procesoru.

Ostvareni rezultati najbolje pokazuju zašto područje primjene grafičkih procesora danas uvelike nadilazi samo obradu grafike. Poslovi koji su se prije izvodili danima ili tjednima, izvođenjem na grafičkim karticama mogu se obaviti u samo par minuta. Danas vjerojatno najpoznatiji projekt distribuiranog računanja koji se uvelike oslanja na grafičke kartice jest Folding@Home. Glavni cilj spomenutog projekta je istraživanje uzroka raznih bolesti poput Alzheimerove, Huntingtonove i raznih oblika raka. Zbog svojih visokih performansi, niske potrošnje i cijene, grafičke kartice se danas sve češće koriste kao zamjena za superračunala.

Buduće generacije grafičkih kartica obećavaju još bolje performanse, nižu potrošnju i lakše programiranje, što je u ovom trenutku možda i jedini nedostatak.

7. Literatura

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

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

[3] Song, Y., *Nvidia Graphics Processingg Unit (GPU)*, 9.9.2009.

http://www2.engr.arizona.edu/~yangsong/gpu.htm , 22.5.2010.

[4] *NVIDIA GeForce 8800 GTX/GTS Tech Report*,

http://www.rojakpot.com/showarticle.aspx?artno=358&pgno=1 , 22.5.2010.

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

http://www.hardwarezone.com/articles/view.php?cid=3&id=2793, 20.4.2010.