SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1489

PRIMJENA GRAFIČKOG SKLOPOVLJA ZA OBAVLJANJE OPERACIJA OBRADE SLIKA

JOSIP HUCALJUK

Zagreb, srpanj 2010

Sadržaj

1. Uvod

Od samih početaka, razvoj grafičkih procesora, kao i ostale računalne elektronike, pratio je poznati Mooreov zakon, 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).

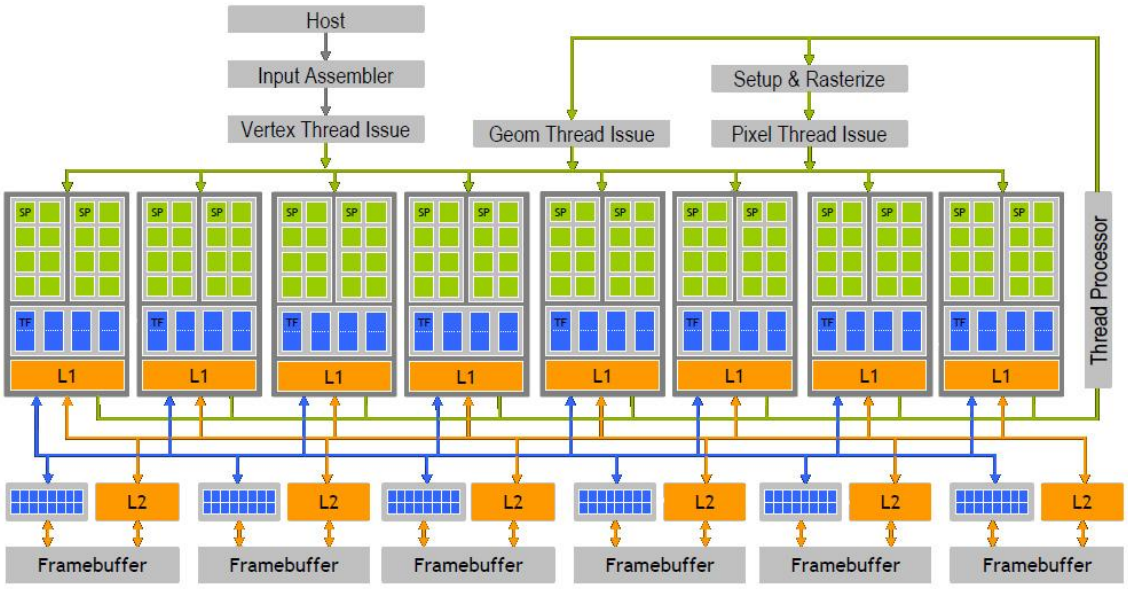
Jedno od područja gdje se uporaba grafičkih procesora naročito isplati je obrada slike. Kako se slike sastoje od mnogo slikovnih elemenata predstavljenih u memoriji numeričkim vrijednostima, a procesi obrade slika predstavljaju niz jednostavnih matematičkih operacija koje se vrše nad tim istim elementima, visoko paralelizirana arhitektura grafičkih procesora omogućava potencijalno značajna ubrzanja.

Kroz niz sljedećih poglavlja bit će pobliže opisana arhitektura modernih grafičkih procesora (2. poglavlje), CUDA razvojna okolina (3. poglavlje), uobičajeni algoritmi koji se koriste pri obradi slika, te njihove implementacije prilagođene za izvršavanje na grafičkim procesorima (4. poglavlje). Na kraju će biti prikazani i sami eksperimentalni rezultati dobiveni izvođenjem ostvarenog programskog rješenja (5. poglavlje).

2. Arhitektura grafičkih procesora

2.1. Arhitektura modernih grafičkih procesora

Kao što je rečeno u uvodu, današnje grafičke procesore karakterizira masivno paralelizirana arhitektura. Na slici 2.1 [8] prikazana je konceptualna shema modernog grafičkog procesora.



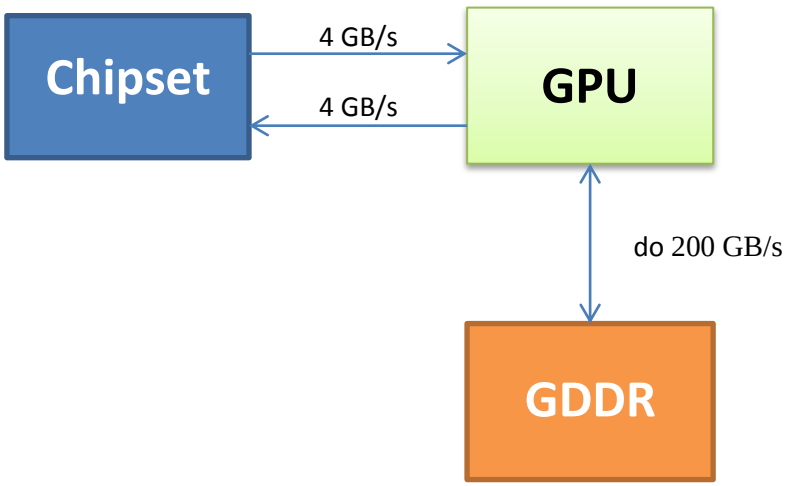
Slika 2.1 - Shema modernog grafičkog procesora

Visoka paraleliziranost grafičkog procesora ostvarena je velikim brojem jednostavnih procesorskih jezgri nazvanih *Streaming Processors* (SP)[1]. Radi jednostavnijih kontrolnih mehanizama i pristupa memoriji, jezgre su grupirane u veće nakupine (uobičajeno po 8 SP-ova) koje se nazivaju *Streaming Mulitprocessors* (MP). 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 današnji moderni grafički procesori iz sredine ponude imaju uobičajeno 128 takvih jezgri, koje rade na frekvencijama preko 1 GHz, lako se može doći do brojke od preko 500 GFLOPS-a (**G**iga **Fl**oating Point **Op**erations Per **S**econd), odnosno 500 milijardi operacija sa pomičnim zarezom u sekundi. Kod nedavno izašle nove generacije Nvidia GeForce grafičkih procesora, ta vrijednost

doseže i 1,5 TFLOPS, što ulazi već u domenu do sada rezerviranu isključivo za superračunala.

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 (*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 i do 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 – Komunikacija

2.2. CUDA

2.2.1. Općenito

Ideja korištenja grafičkih procesora u svrhu obrade općih podataka, postojala je i prije pojave CUDA arhitekture. Najveći problem koji je sprječavao širu uporabu bilo je ograničenje komunikacije 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.

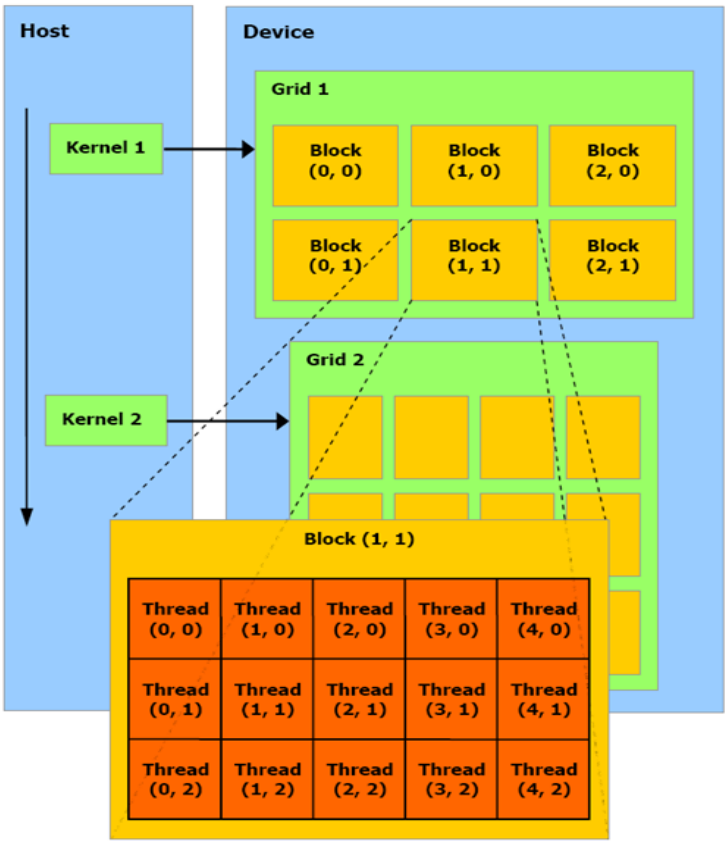
Veliki iskorak u GPGPU industriji dogodio se 2006. godine, predstavljanjem CUDA arhitekture. Po prvi puta dio silicija grafičkog procesora bio je posvećen za olakšavanje GPGPU programiranja. Uvedeno je posebno hardversko sučelje, kao i svi potrebni alati za efikasno paralelno programiranje. Od 2006.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

2.2.2. Logička struktura

Kod izrade paralelnih aplikacija korištenjem CUDA-e, moguće je razlikovati dvije vrste programskog koda: kod koji će se izvršavati na centralnom procesoru (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 isti kod, ali nad različitim podatcima. 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. [7]



Slika 2.3 - Logička struktura

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 MP-u. 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.

2.2.3. Memorijska hijerarhija

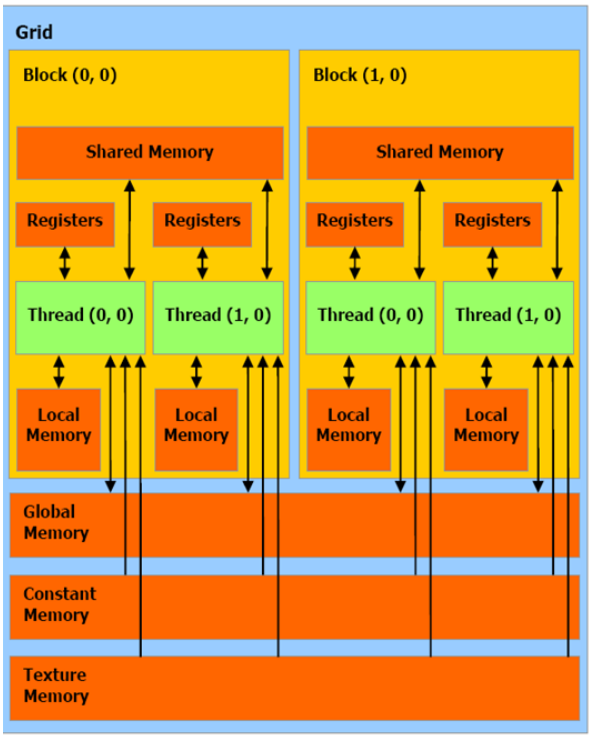
Kod pohranjivanja podataka korisniku na raspolaganju dostupne su sljedeće memorije (sortirane po brzini pristupa):

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

Na slici 2.4 prikazana je shema memorijske hijerarhije. [7]

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 65 KB po mreži blokova, a unos podataka moguć je samo od strane centralnog procesora.



Slika 2.4 - Memorijska hijerarhija

Dijeljena memorija definirana je na razini blokova, te je vrlo ograničene veličine (16 KB po bloku), ali omogućava vrlo brzi pristup podatcima (3 - 4 ciklusa). Što efikasnije rukovanje dijeljenom memorijom nužno je da bi se zaobišlo ograničenje memorijske propusnosti.

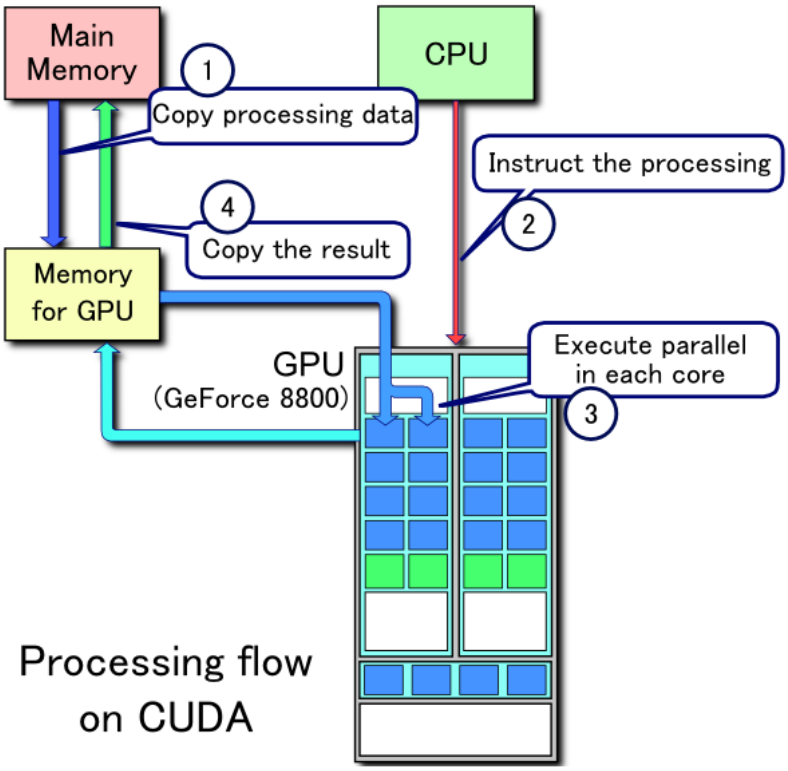
Zadnja, ujedno i najbrža, memorijska lokacija su registri. Registri su definirani na razini bloka dretvi, te ih je ukupno 8192 raspoloživo po bloku. Raspoloživi registri dijele se u jednakim dijelovima među dretvama u bloku. U slučaju da dretve koriste više registara nego što bi smjele, smanjuje se broj

raspoloživih warpova koji se mogu izvršavati na jednom SM-u. 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.

2.2.4. Dijagram toka CUDA aplikacije

Na slici 2.5 prikazan je dijagram toka standardne CUDA aplikacije [9]. 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



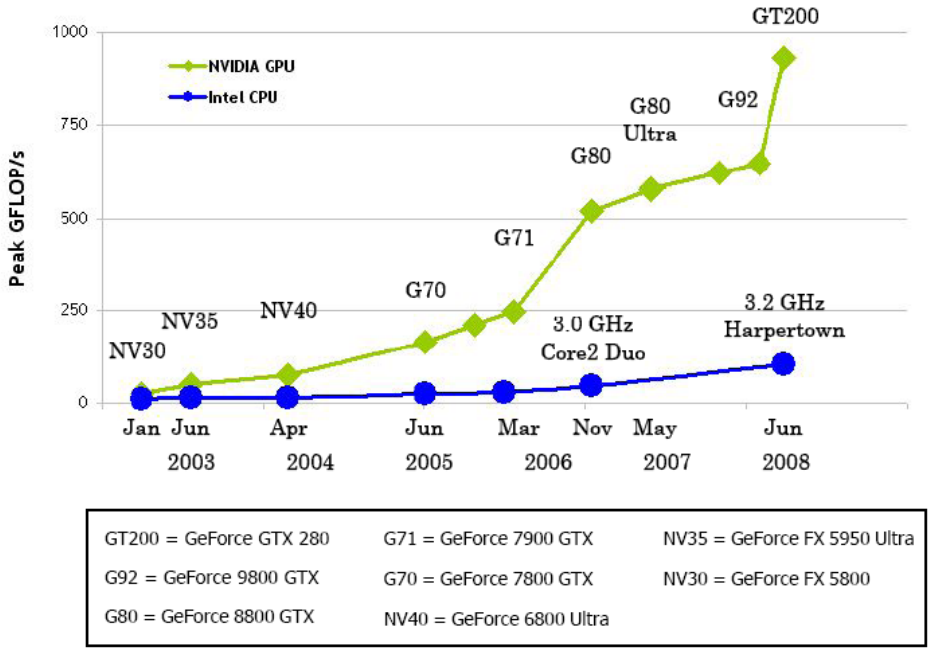
Slika 2.5 - Dijagram toka

2.3. Usporedba sa centralnim procesorom

Veliki broj procesorskih jezgri koje uz to rade na relativno visokoj frekvenciji omogućuju grafičkim procesorima obavljanje izrazito mnogo operacija u sekundi, znatno više nego što to može centralni procesor. Dok današnji centralni procesori iz najvišeg segmenta razvijaju tek oko 60 GFLOPS-a, grafički procesori srednje klase obavljaju 10 puta više operacija u sekundi i sve to po 5 puta manjoj cijeni!

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

Slika 2.5 prikazuje razvoj centralnih i grafičkih procesora u razdoblju od 2003. do 2008. godine. [10]



Slika 2.5 - Razvoj CPU i GPU

Razlika u budućnosti će se nastaviti povećavati. Razlog tome može se naći u činjenici da veliki dio centralnog procesora obavlja kompleksne upravljačke

operacije (npr. predviđanje grananja), te mnogo tranzistora troši se na priručne memorije raznih razina. Sve to ograničava broj jezgri koje je moguće integrirati, a da pri tome potrošnja ostane u prihvatljivim granicama.

Prema prognozama CEO Nvidia-e snaga grafičkih procesora kroz sljedećih 6 godina porasti će 570 puta! U istome razdoblju centralni procesori povećati će 3 puta snagu. Točan razlog i temelj za tako hrabru prognozu u ovome trenutku još nije poznat, ali ako se ostvari i dio obećanog pojačanja, omogućit će nevjerojatne pomake u industriji. [11]

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

Što to sve znači? Postaju li centralni procesori nepotrebni?

Na to pitanje teško je odgovoriti. Danas još uvijek postoji puno aplikacija koje centralni procesor može puno brže i bolje napraviti od grafičkog, dosta poslova nije ni moguće izvesti na grafičkom, ali oni poslovi koji se mogu prilagoditi za rad na grafičkim procesorima nerijetko pokazuju performanse 10 do 100 puta bolje od onih na centralnim procesorima. Također, sa svakom novom generacijom, grafički procesori dobivaju sve više elemenata centralnog procesora, čime se razlika dodatno smanjuje. Potvrda tome je posljednja generacija kodnog imena Fermi, koja koristi L1 i L2 priručne memorije, te omogućuje nativno izvođenje C++ programskog koda.

3. Razvojna okolina

3.1. Općenito

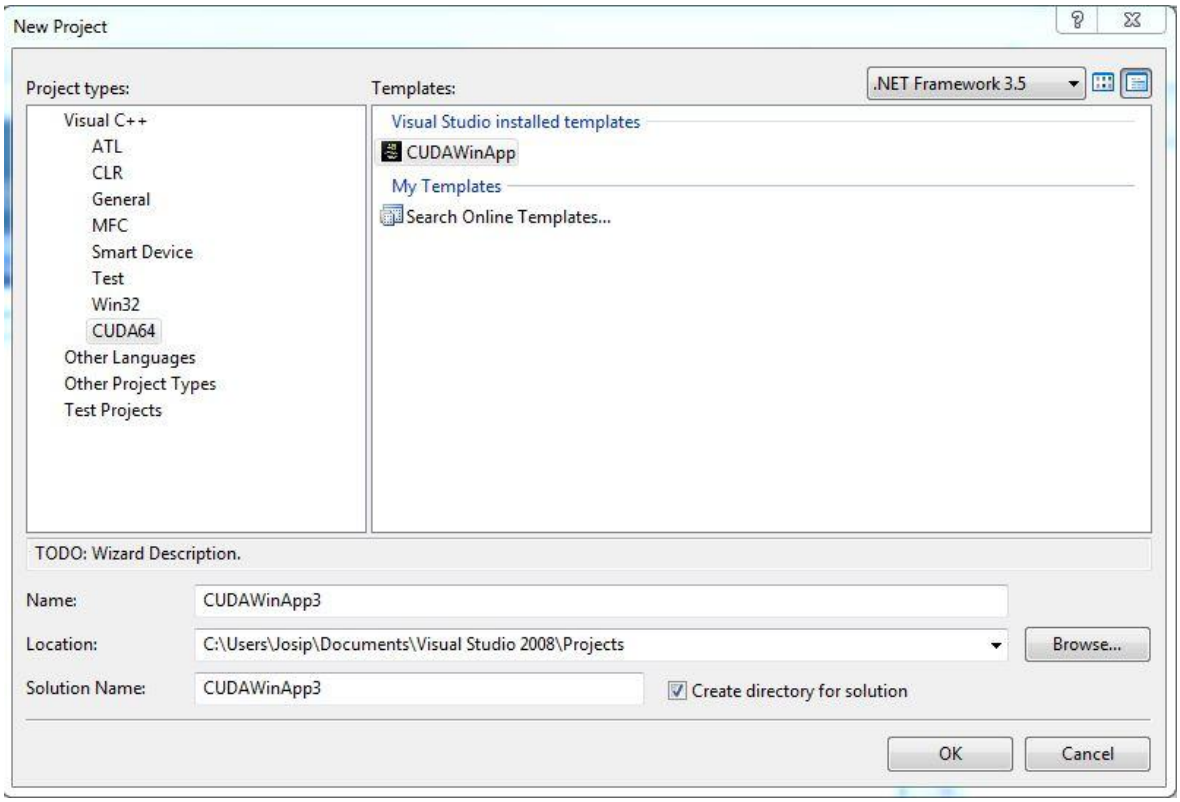
Prije nego li se može krenuti sa razvojem CUDA aplikacija potrebno je prilagoditi razvojnu okolinu.

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. U trenutku pisanja ovog rada, proširenje je bilo dostupno isključivo za Visual Studio 2008, te se nalazilo u beta fazi testiranja.

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 (slika 3.1).



Slika 3.1 - CUDA projekt

3.2. Primjer CUDA aplikacije

Programski kod koji će se izvršavati na grafičkom procesoru, nalazi se u datotekama sa .cu ekstenzijom. Također, u njima je moguće napisati i kod koji će se izvršavati na centralnom procesoru. Da bi prevodioc mogao razlikovati programske kodove, kod koji će se izvršavati na grafičkom procesoru ima specifičnu sintaksu. Sljedeći primjer ilustrira jednu funkciju (*kernel*) koja se izvršava na grafičkom procesoru.

```
__global__ void GPUMultiplication (int a, int b, int *rezultat)
{

	*rezultat = a * b;

	// Funkcije koje vrše ispis poput printf
	// ili čitanje podataka (scanf), NISU podržane
	// Funkcije ne mogu vraćati vrijednost
}
```

Poziv odgovarajuće funkcije vrši se na sljedeći način:

```
int main()
{
	...

	dim3 dimGrid(3,3); // Blok od 9 dretvi, 3x3

	dim3 dimBlock(2,2);  // Mreža od 4 bloka, 2x2

	...

	GPUMultiplication<<<dimGrid, dimBlock>>>(a, b, rezultat);

	...
}
```

Ono što je bitno uočiti je da prilikom poziva funkcije je nužno definirati koliko dretvi će se stvoriti, te njihovu logičku strukturu. Blokovi i mreže mogu se definirati u najviše 3 dimenzije, uz još poneka ograničenja. Jedan blok može imati najviše (ovisno o generaciji) 512 dretvi, koje se mogu rasporediti u 1D, 2D ili 3D polje. Kod definiranja mreža, ograničenje je znatno slabije, te tako moguće je dizajnirati mrežu sa najviše 65536 x 65536 x 1 blokova.

Budući da nam na raspolaganju stoje dvije vrste potpuno odvojenih memorija, radna memorija glavnog procesora i memorija grafičke kartice, potrebno je prije pozivanja kernela osigurati prostor za inicijalne podatke i rezultat u memoriji grafičke kartice, te prekopirati podatke u taj prostor. Razlog tome je činjenica da grafički procesor ne može pristupiti direktno podatcima u radnoj memoriji centralnog procesora preko pokazivača, kao i obrnuto. Iz toga razloga je potrebno podatke prekopirati prije izvođenja. Nakon izvršavanja kernela, rezultat obrade je pohranjen u grafičkoj memoriji. Da bi centralni procesor mogao dalje raditi sa tim podatcima, potrebno ih je prekopirati iz grafičke memorije u radnu memoriju centralnog procesora. Cjeloviti kod prikazan je na sljedećem primjeru.

```
__global__ void GPUMultiplication (int a, int b, int *rezultat)
{
       //množenje 2 ulazna broja
       *rezultat = a * b;
}
int main()
{
      int a = 2;
      int b = 3;

      int *rezultatCPU  = (int *)malloc(sizeof(int));

      //rezerviranje grafičke memorije za rezultat
      int *rezultat;
      cudaMalloc((void**)&rezultat, sizeof(int));

      dim3 dimGrid(3,3); // Blok od 9 dretvi, 3x3
      dim3 dimBlock(2,2);  // Mreža od 4 bloka, 2x2

      //predaju se jednostavne vrijednosti, pa nije potrebno
      //rezervirati prostor na grafičkoj memoriji za a i b
      GPUMultiplication<<<dimGrid, dimBlock>>>(a, b, rezultat);

      //kopiranje umnoška iz grafičke memorije u radnu memoriju
      cudaMemcpy(rezultatCPU, rezultat, sizeof(int),
                  cudaMemcpyDeviceToHost);

      printf("Rezultat množenja je: ", *rezultatCPU);
      return 0;
}
```

4. Algoritmi i programska implementacija

4.1. Konvolucija

Jedna od najčešće korištenih i najvažnijih operacija koje se vrše pri procesu obrade slika jest konvolucija. Matematički gledano konvolucija dviju funkcija je funkcija koja predstavlja u kojoj mjeri se te dvije ulazne funkcije „preklapaju“. Formulom zapisano to izgleda na sljedeći način:

$$(f * g)(t) = \int f(t-n)g(n)dn \qquad (4.1)$$

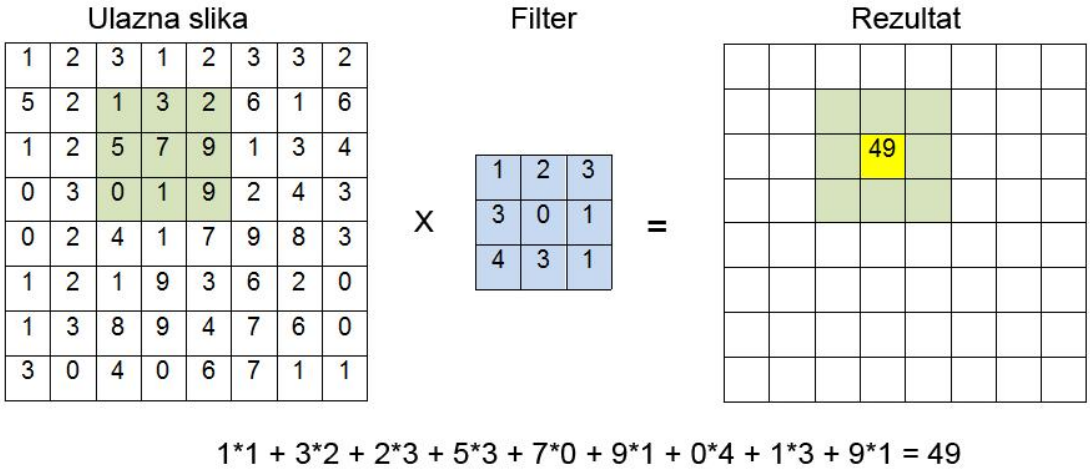
$$(f * g)(k) = \sum_n f(k-n)g(n) \qquad (4.\ 2)$$

Formula (4.1) odnosi se na kontinuirane funkcije, a (4.2) na diskretne. Budući da je naš cilj primijeniti konvoluciju na slike, a slike su pohranjene u memoriji računala u obliku niza brojeva, gdje svaki podniz od 4 broja pripada jednom pikselu slike, logično se nameće da će se koristiti formula za diskretne funkcije sa malom modifikacijom. Modifikacija je potrebna iz razloga što je slika zapravo dvodimenzionalno polje, a formula (4.2) primjenjiva je samo za jednodimenzionalan slučaj. Konačna formula koja je temelj za implementaciju:

$$(f * g)(i,j) = \sum_n \sum_m f(i-n, j-m)g(n,m) \qquad (4.\ 3)$$

Koja je veza između konvolucije, odnosno formule (4.3), i obrade slike? Kod obrade slika, interno nad početnom slikom primjenjujemo neki filter koji vrši transformaciju podataka početne slike i na taj način dobivamo željeni efekt. Direktno preslikavajući to u formulu (4.3), funkcija f(i,j) odgovara podatcima ulazne slike, a funkcija g(i,j) je zapravo željeni filter. Najlakši način za shvatiti to je zamisliti ulazne podatke i filter u obliku dviju matrica (što zapravo odgovara implementaciji), a konvoluciju kao proces pri kojem „pomičemo“ matricu filtera iznad matice ulazne slike i pri tome množimo direktno preklopljene elemente te ih sve skupa sumiramo. Sumu pohranjujemo u izlaznu matricu na mjesto središnjeg elementa matrice ulaznih podataka.

Postupak je prikazan na slici 4.1.



Slika 4.1 2D konvolucija

Iz praktičnih razloga sve matrice bit će kvadratne.

4.2. Osnovna programska implementacija

Implementacija konvolucije za rad na centralnom procesoru je prilično jednostavna, te se neće opisivati. Osnovna ideja kod implementacije na grafičkom procesoru je stvoriti dovoljno dretvi, tako da svaka dretva izračunava jedan element izlazne matrice. Znači, ako je ulazna matrica dimenzija 1024 x 1024, broj stvorenih dretvi će biti 1024^2. Za osnovnu implementaciju, kod proračuna će se koristiti podatci spremljeni u globalnoj grafičkoj memoriji. Ulazni podatci spremljeni su u polje struktura pixelData. Sadržaj strukture prikazan je sljedećim odsječkom:

```
typedef struct {
      unsigned char red;
      unsigned char green;
      unsigned char blue;
}pixelData;
```

Sljedeći programski kod ilustrira implementaciju osnovnog algoritma.

```
__global__ void GPUconvolute(pixelData *imageSource, pixelData
                  *deviceResult, int *fil, int width, int height)
{
      int bx = blockIdx.x;
      int by = blockIdx.y;

      int tx = threadIdx.x;
      int ty = threadIdx.y;

      int dimX = bx * BLOCK + tx;
      int dimY = by * BLOCK + ty;

int sum_r = 0;
int sum_g = 0;
int sum_b = 0;

int index = dimY * width + dimX;

for(int y = -FILTER_DIAM/2; y <= FILTER_DIAM/2; y++)
{
      for(int x = -FILTER_DIAM/2; x <= FILTER_DIAM/2; x++)
      {
            if(((dimX + x)>= 0 ) && ((dimX + x) < width) &&
                  ((dimY + y) >= 0) && ((dimY + y) < height))
            {
                  int indexFil = (FILTER_DIAM/2 + y) * FILTER_DIAM +
                                    FILTER_DIAM/2 + x;

                  int indexData = index + y * width + x;

                  sum_r += imageSource[indexData].red *
                              fil[indexFil];

                  sum_g += imageSource[indexData].green *
                              fil[indexFil];

                  sum_b += imageSource[indexData].blue *
                              fil[indexFil];
            }
      }
}

deviceResult[index].red = sum_r / FILTER_SUM;

deviceResult[index].green = sum_r / FILTER_SUM;

deviceResult[index].blue = sum_r / FILTER_SUM;

}
```

4.3. Optimizacije

Kao što može i naslutiti iz programskog kod osnovne implementacije, postoji dosta prostora za poboljšanje performansi.

4.3.1. Optimizacije memorijskog pristupa

Jedan od najvećih problema osnovne izvedbe je učestalo dohvaćanje podataka iz globalne memorije. Kao posljedica toga dolazi do situacije gdje većina dretvi čeka da se dohvate potrebni podatci za računanje, a u isto vrijeme procesorske jezgre stoje „nezaposleno“. Ovo ujedno i predstavlja najčešći problem kod programiranja na grafičkim procesorima. Na koji način zaobići to memorijsko ograničenje te iskoristiti puni potencijal grafičkog hardvera?

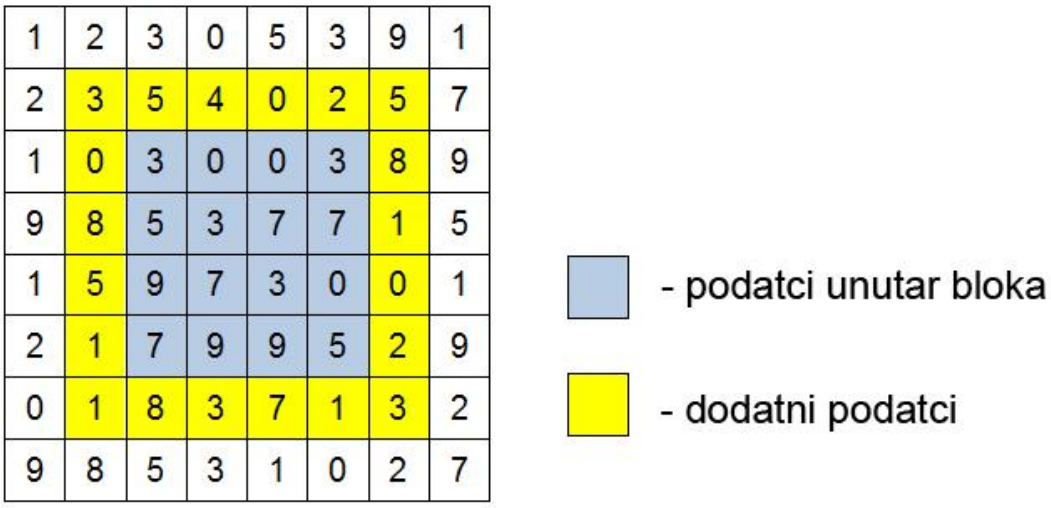
Kao rješenje navedenog problema logično se nameće korištenje dijeljene memorije koju odlikuje izuzetno nisko vrijeme odziva, ali i vrlo mali kapacitet. Stoga ključ za postizanje dobrih performansi je efikasno iskoristiti dostupnu dijeljenu memoriju. Implementacija je izvršena na sljedeći način. Blok dretvi sadrži 256 dretvi (16 x 16), od kojih svaka dretva čita iz glavne memorije podatke za jedan piksel (3 byte-a). Budući da su nam pri konvoluciji potrebni i podatci koji okružuju piksel za koji se računa konvolucija, očito je da će biti potrebno za piksele na granicama bloka učitati i podatke izvan bloka (slika 4.2). Stoga, sve dretve koje su udaljene od granice bloka manje od pola duljine filtera, učitavaju i još jedan element sa vanjske strane. Ukupno količina učitanih podataka jest :

$$C = (16 + \text{duljina filtera} - 1)^2 * 3 \quad [\text{byte}]$$

Ako znamo da je dijeljena memorija ograničena na 16 KB, maksimalna duljina filtera jest 56 elemenata. Ovime je postignuto da za računanje konvolucije 256 piksela slike je potrebno (16 + duljina_filtera - 1)^2 pristupa globalnoj memoriji, dok kod osnovne implementacije taj broj je 16^2 * (duljina_filtera)^2. Ako uzmemo za primjer da radimo sa filterom duljine 15 elemenata, ušteda je sljedeća:

$$n = \frac{16^2 * 15^2}{(16 + 15 - 1)^2} = \frac{57600}{900} = 64$$

Za zadani primjer, 64 puta manje pristupa se vrši prema globalnoj memoriji, što u velikoj mjeri popravlja performanse. Testiranjem je utvrđeno ubrzanje u prosjeku oko 10 puta u odnosu na osnovnu implementaciju.



Slika 4.2 Potrebni podatci

Implementacija gore opisane optimizacije zahtjeva sljedeće preinake na osnovnoj izvedbi:

- Rezerviranje dijeljene memorije

```
__shared__ pixelData matrix[15 + FILTER_DIAM][15 + FILTER_DIAM];
```

- Učitavanje podataka unutar bloka

```
//učitavanje osnovnih podataka
      matrix[ty + FILTER_DIAM/2][tx + FILTER_DIAM/2] =
                                                    imageSource[index];
```

❖ Učitavanje okvira

```
if(((dimX - FILTER_DIAM/2) >= 0) && ((dimY - FILTER_DIAM/2) >= 0))
{
    if((tx < (FILTER_DIAM/2)) && (ty < (FILTER_DIAM/2)))
          matrix[ty][tx] = imageSource[(dimY-FILTER_DIAM/2) * width
                                  + (dimX - FILTER_DIAM/2)];
}

...

// učitavanje preostalih dijelova okvira

...
```

❖ Izmjena mjesta odakle se čitaju podatci pri proračunu

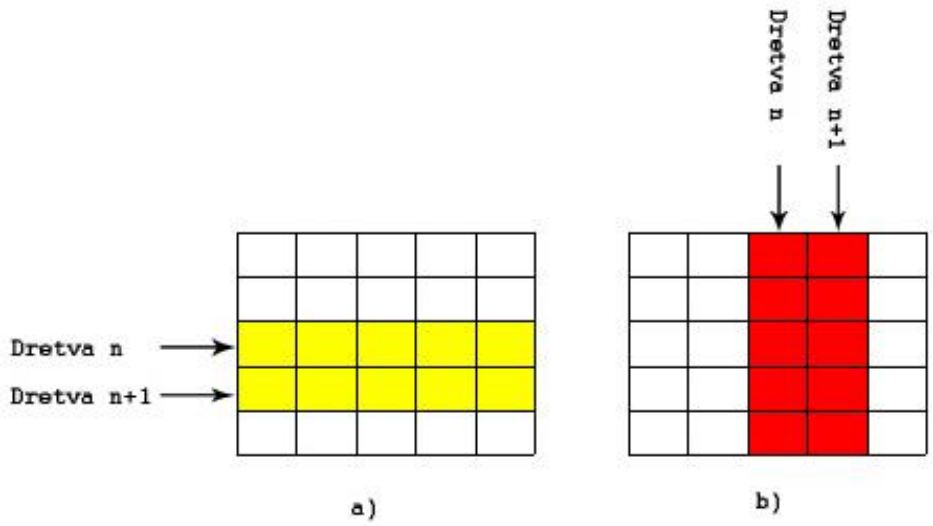
```
...
for(int y = -FILTER_DIAM/2; y <= FILTER_DIAM/2; y++)
{
      for(int x = -FILTER_DIAM/2; x <= FILTER_DIAM/2; x++)
      {
            if(((dimX + x)>= 0 ) && ((dimX + x) < width) &&
                  ((dimY + y) >= 0) && ((dimY + y) < height))
            {
                  int indexFil = (FILTER_DIAM/2 + y) * FILTER_DIAM +
                                    FILTER_DIAM/2 + x;

                  sum_r += matrix[ty + FILTER_RADIUS/2 + y][tx +
                           FILTER_RADIUS/2 + x].red * fil[indexFil];

                  sum_g += matrix[ty + FILTER_RADIUS/2 + y][tx +
                           FILTER_RADIUS/2 + x].green * fil[indexFil];

                  sum_b += matrix[ty + FILTER_RADIUS/2 + y][tx +
                           FILTER_RADIUS/2 + x].blue * fil[indexFil];
            }
      }
}
```

Iako su memorijske performanse znatno poboljšane sa prethodnom optimizacijom, prostor za daljnje poboljšanje postoji. Kako nije moguće dalje smanjiti broj pristupa globalnoj memoriji, potrebno je onda učiniti pristup memoriji što učinkovitijim. Kao što je rečeno u poglavlju 2.1, performanse memorije uvelike ovise o načinu pristupanja podatcima, odnosno o njihovom razmještaju u memoriji. Kako bi se postigle maksimalne performanse nužno je čitati podatke iz uzastopnih memorijskih lokacija. Za ilustraciju principa koristit će se slika 4.3.



Slika 4.3 Pristup memoriji

Dvodimenzionalne strukture poput 2D polja pohranjuju se u memoriji sekvencijalno na način da se prvo pohrani prvi red, pa onda slijedi drugi red itd. Ako bismo podatcima pristupali kao što je prikazano na slici 4.3 pod a) gdje svaka dretva čita matricu po stupcima, interno čitanje iz memorije bilo bi neefikasno jer podatci nisu slijedno pohranjeni u memoriji. U jednom pristupu memoriji mogao bi se pročitati samo jedan element matrice, te u slučaju velikog broja dretvi, odnosno dimenzija matrice, memorijska propusnost bi drastično opala. Bolji način pristupa memoriji prikazan je u b) primjeru, gdje svaka dretva čita podatke matrice redak po redak. Na taj način podatci koji se trebaju pročitati interno su pohranjeni slijedno u memoriji, te je moguće pročitati više njih u samo jednom pristupu. Implementacija ove optimizacije svodi se na mijenjanje rasporeda *if* blokova koji učitavaju dijelove okvira. Rezultat optimizacije je 40% ubrzanje izvođenja.

4.3.2. Ubrzanje aritmetičkih operacija

Nakon što je memorijsko ograničenje svedeno na minimum na redu je pokušati ubrzati izvođenje aritmetičkih operacija.

CUDA pruža niz alternativnih operacija tradicionalno „skupim“ aritmetičkim operacijama poput množenja, dijeljenja, korjenovanja itd. Popis dostupnih funkcija je prikazan u sljedećoj tablici: [5]

Tablica 4.1 - Popis CUDA operacija

__fdividef(x,y)	__exp(x)
__log(x)	__log2(x)
__log10(x)	__sin(x)
__cos(x)	__tan(x)
__pow(x,y)	__mul24(x,y)
__umul24(x,y)	__mulhi(x,y)
__umulhi(x,y)	__int_as_float(x)
__float_as_int(x)	__saturate(x)

Sve CUDA alternativne funkcije započinju sa dvostrukom podvlakom kako bi ih se razlikovalo od standardnih matematičkih funkcija. U usporedbi sa standardnim funkcijama, izvode se u manjem broju ciklusa, ali kao posljedica toga postoji i mogućnost odstupanja od potpuno točnog rezultata u određenim granicama. Budući da su vrijednosti tih granica izrazito malene, njihovo korištenje u ovom radu je potpuno opravdano. Ubrzanje koje se može postići korištenjem ovih funkcija ovisi od funkcije do funkcije. Dok standardno množenje 32 bitnih cjelobrojnih brojeva zahtjeva 16 ciklusa, množenje pomoću ***__mul24*** zahtjeva samo 4. Dijeljenje brojeva sa pomičnim zarezom zahtjeva u prosjeku 36 ciklusa, dok korištenjem ***__fdividef*** to se svodi na 20 ciklusa. U slučaju da je potrebno izvršiti dijeljenje cjelobrojnih brojeva, preporučeno je koristiti bitovne operatore što je više moguće. Funkcije *__int_as_float* i *__float_as_int* posebno su zanimljive, jer prema specifikaciji [5] izvode se u 0 ciklusa, što omogućava korištenje funkcija posebno optimiziranih za brojeve sa pomičnim zarezom bez ikakvog dodatnog kašnjenja.

4.3.3. Optimizacija algoritma konvolucije

Jedan od nedostataka klasične dvodimenzionalne konvolucije je polinomijalna složenost. Za svaki element slike potrebno je izvršiti n^2 množenja i sve to sumirati. U slučaju slika velikih dimenzija i velikih filtera, broj potrebnih operacija postaje strašno velik.

Korištenjem separabilnih filtera moguće je smanjiti tu složenost na 2*n operacija množenja, što kod većih filtera se itekako osjeti. Separabilni filteri su filteri koje je moguće prikazati kao umnožak stupca i vektor retka. Primjer jednog takvog filtera prikazan je formulom (4.1).

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \qquad (4.1)$$

Efekt koji bi nastao primjenom filtera $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ nad slikom možemo postići korištenjem prvo filtera $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$, a potom $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$. Ušteda koja se na taj način postiže određena je omjerom (za kvadratne matrice) :

$$\frac{n^2}{2n}$$

Za filtere veličine 15 elemenata, broj potrebnih operacija je 7.5 puta manji. Sa povećanjem veličine filtera ta razlika se još znatno povećava.

Nažalost, nije moguće sve željene efekte postići sa matricama koje se mogu na taj način rastaviti. Tipičan predstavnik filtera koje je moguće rastaviti je Gaussov filter, kao i Sobelov filter za detekciju rubova (4.1).

4.4. Rezultati obrade

U nastavku su prikazani neki efekti koji se mogu postići sa različitim vrijednostima filtera.

a) Gauss filter – $[1 \quad 3 \quad 7 \quad 10 \quad 7 \quad 3 \quad 1]$

Slika 4.4 Gauss filter

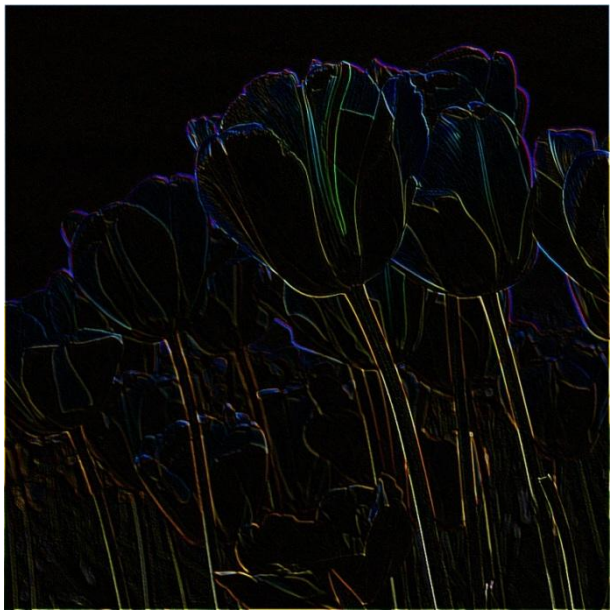
b) Izoštravanje - $\begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & 2 & 8 & 2 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}$

Slika 4.5 Filter za izoštravanje

c) Emboss filter - $\begin{bmatrix} -1 & -1 & -1 & -1 & 0 \\ -1 & -1 & -1 & 0 & 1 \\ -1 & -1 & 0 & 1 & 1 \\ -1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$

Slika 4.6 Emboss filter

d) Sobel filter - $\begin{bmatrix} -1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & -1 \end{bmatrix}$



Slika 4.7 Sobel filter za detekciju rubova

5. Eksperimentalni rezultati

5.1. Općenito

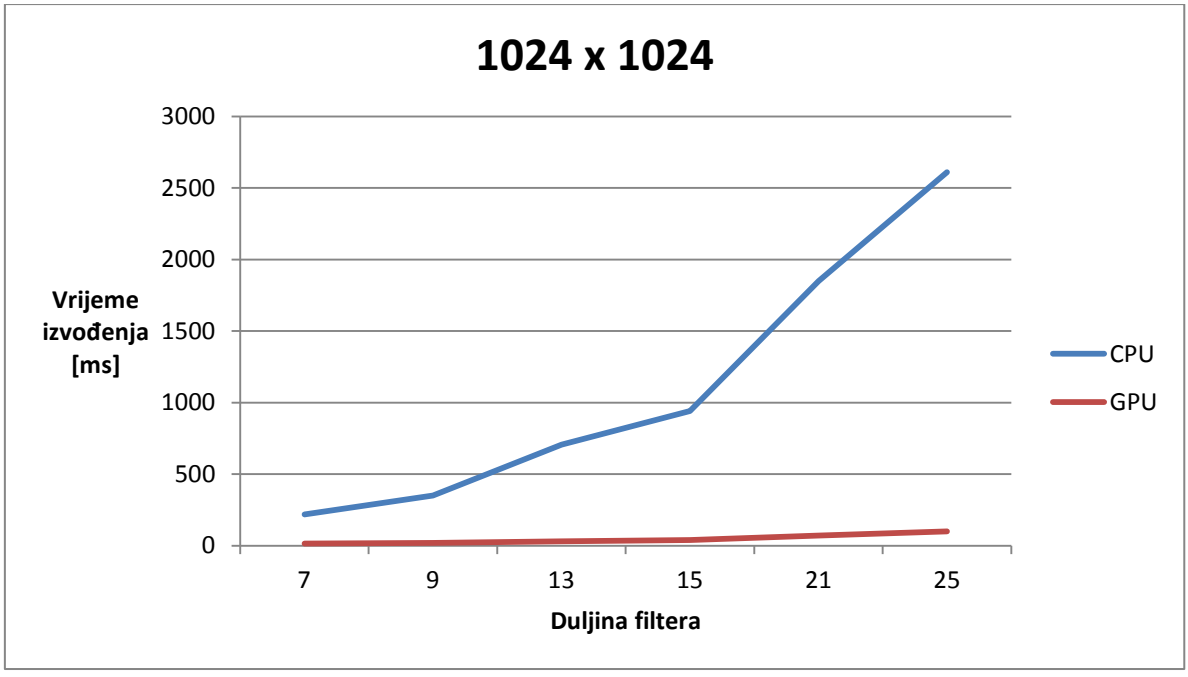
Kako bi se ocijenile performanse ostvarenog programskog rješenja, testiranje je izvršeno nad slikama različitih rezolucija i različitih veličina filtera. To nam omogućuje ne samo usporedbu performansi izvođenja na centralnom i grafičkom procesoru, već i uvid kako koja veličina utječe na cjelokupne performanse.

Rezolucije odabrane za izvođenje testova su: 1024 x 1024, 2048 x 2048 i 4096 x 4096 piksela, dok za veličine filtera su odabrane sljedeće vrijednosti: 7, 9, 13, 15, 21 i 25 elemenata. Za svaku kombinaciju rezolucije i veličine filtera, test je ponovljen 1000 puta, te je kao rezultat uzeta srednja vrijednost.

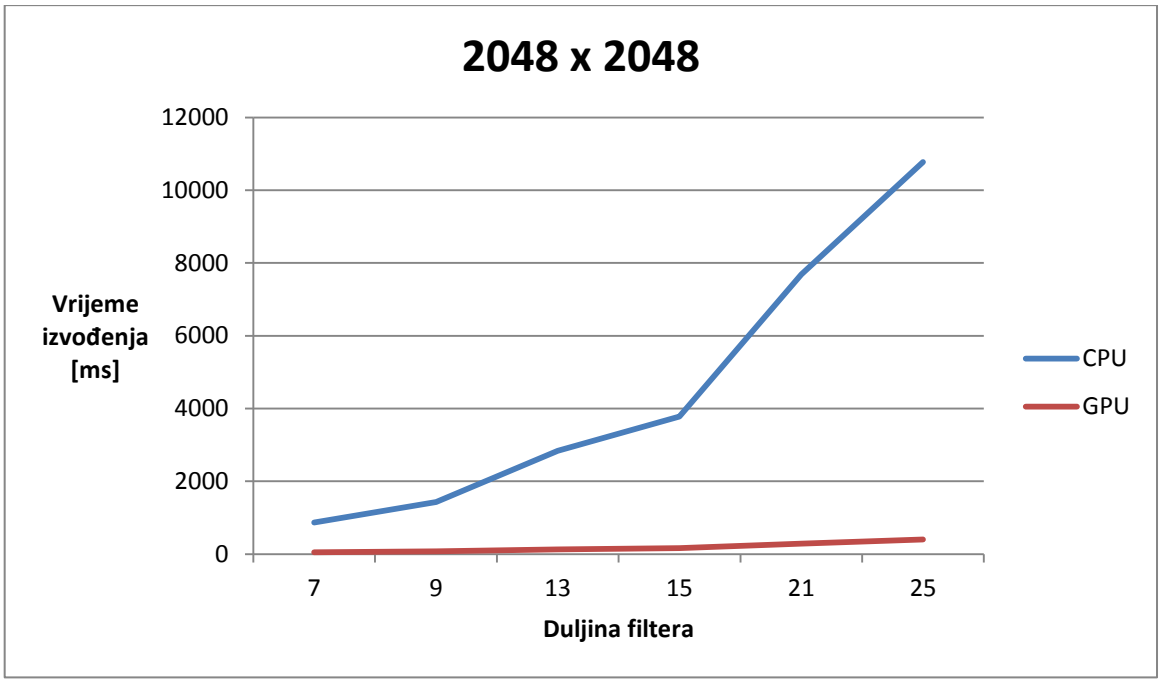
Sa hardverske strane, testovi programskog koda prilagođenog izvođenju na centralnom procesoru izvedeni su na procesoru Intel Core 2 Duo T7500 takta 2,2 GHz, a testovi programskog koda prilagođenog izvođenju na grafičkom procesoru, izvedeni su na grafičkoj kartici Nvidia Geforce 8800GTS 512.

5.2. Testovi brzine izvođenja

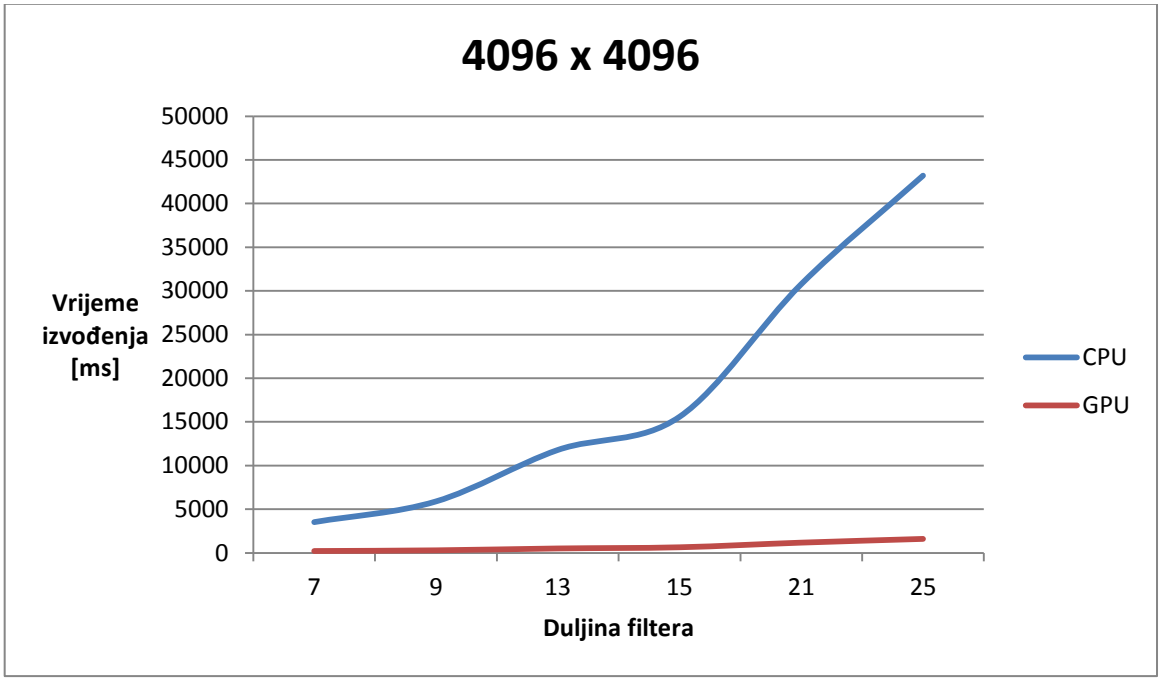
Na sljedeća 3 grafa bit će prikazani rezultati izvođenja programskih rješenja za sve 3 rezolucije.



Slika 5.1 Rezultati izvođenja za rezoluciju 1024 x 1024



Slika 5.2 Rezultati izvođenja za rezoluciju 2048 x 2048

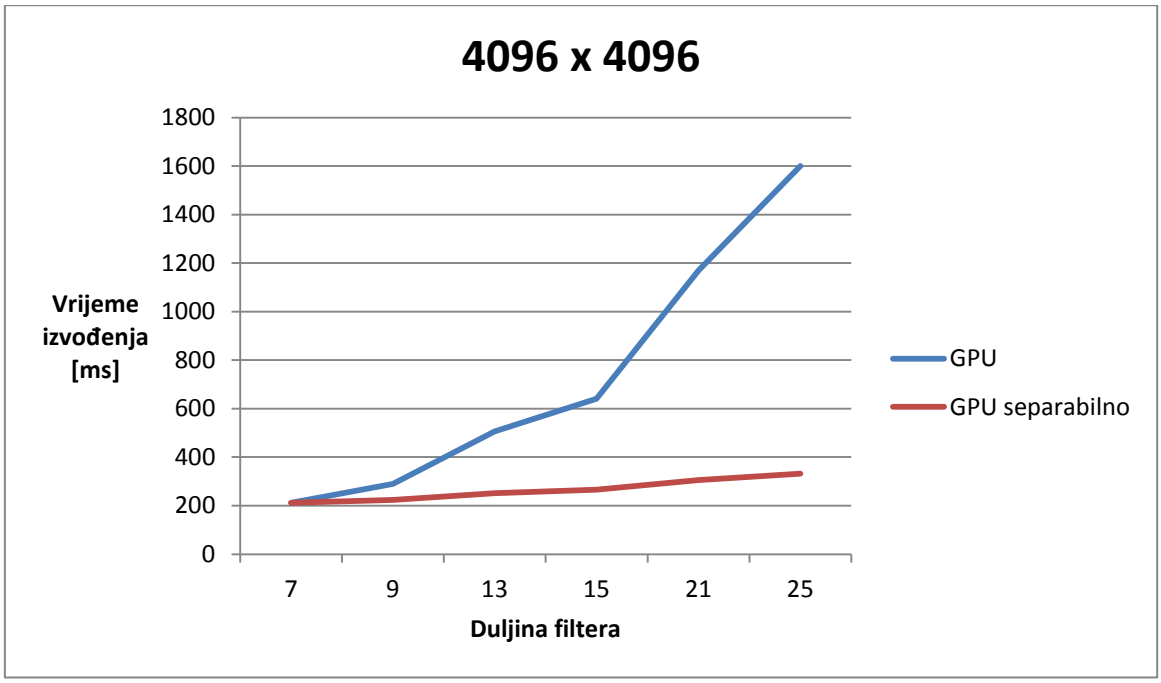


Slika 5.3 Rezultati izvođenja za rezoluciju 4096 x 4096

Ono što se nedvojbeno može zaključiti iz grafova jest značajno brže izvođenje na grafičkom procesoru. Najmanja zabilježena razlika je pri najnižoj rezoluciji i najmanjem filteru. U tom slučaju izvođenje na grafičkom procesoru jest 15 puta brže nego izvođenje na centralnom procesoru. Razlika se povećava sa porastom veličine filtera, tako da za filtere duljine 25 elemenata, ta razlika se

povećava na 27 puta brže vrijeme izvođenja. Razlog tome je povećanje omjera broja operacija i broja pristupa globalnoj memoriji. Što je omjer veći, manje se performansi gubi na pristupu memoriji. Zanimljivo je i primijetiti da veličina slike ne utječe na odnos performansi. Vrijeme izvođenja prati linearno porast broja piksela koji se trebaju obraditi. Sa svakim porastom rezolucije, čime se povećava 4 puta broj piksela koji se moraju obraditi, vrijeme izvođenja povećava se također u istoj mjeri, neovisno da li se radilo o izvođenju na centralnom ili grafičkom procesoru.

Testirana implementacija programskog koda prilagođenog za izvođenje na grafičkom procesoru nije uključivala zadnju optimizaciju, separabilne filtere. Razlog tome jest želja da se testiraju rješenja koja imaju potpuno jednake funkcionalnosti. Budući da neke efekte nije moguće ostvariti pomoću separabilnih filtera, rezultati testova su isključeni iz gornjih grafova. Ipak, kako bi se stekao dojam koliko ubrzanje se postiže i sa tom optimizacijom napravljen je sljedeći graf.



Slika 5.4 Rezultati izvođenja sa separabilnim filterima

Kao što se može vidjeti iz priloženog grafa, razlika u brzini izvođenja postoji i povećava se porastom veličine filtera. Razlika se kreće od skoro zanemarive za filtere veličine 7 elemenata, pa sve do skoro 5 puta za filtere duljine 25 elemenata.

Ako usporedimo zadnji slučaj, za rezoluciju 4096 x 4096 i filter duljine 25, elemenata sa istim rezultatom izvođenja na centralnom procesoru, dolazimo do ubrzanja od **130x** puta.

U tablici 4.2 dane su i numeričke vrijednosti izvođenja testova:

Tablica 4.2 Rezultati testiranja

Rezolucija	Duljina filtera	CPU [ms]	GPU [ms]	GPU separabilno [ms]
1024 x 1024	7	218	14	14
	9	351	19	14
	13	706	31	16
	15	940	40	17
	21	1850	70	19
	25	2610	100	21
2048 x 2048	7	870	53	53
	9	1430	72	57
	13	2840	127	63
	15	3780	160	67
	21	7690	290	76
	25	10770	400	82
4096 x 4096	7	3520	212	212
	9	5890	290	224
	13	11780	506	252
	15	15580	641	266
	21	30770	1170	306
	25	43200	1600	332

6. Zaključak

U ovome radu opisana je arhitektura modernih grafičkih procesora, te mogućnost njihove primjene za obavljanje poslove opće namjene. Kako bi se prikazalo u kojoj mjeri se može očekivati ubrzanje primjenom grafičkih procesora, implementirana je konvolucija koja predstavlja temelj za brojne operacije koje se vrše pri obradi slika.

Rezultati testiranja implementiranog rješenja ukazuju na minimalno ubrzanje od 15 puta u odnosu na klasičnu implementaciju na centralnom procesoru. Sa povećanjem broja operacija koje se izvršavaju pri obradi, ta razlika ima tendenciju rasta. U sklopu testiranja tako je pri najsloženijem slučaju zabilježeno i ubrzanje od 27 puta u odnosu na klasičnu implementaciju. Ako se u obzir uzme i modifikacija samog algoritma konvolucije, moguće je postići i ubrzanja od nekoliko stotina puta!

Iz ostvarenih rezultata može se lako uvidjeti zašto područje primjene grafičkog sklopovlja danas u znatnoj mjeri nadilazi samo poslove grafičke naravi. Svi poslovi koji zahtijevaju veliku moć izračunavanja mogu u znatnoj mjeri profitirati izvođenjem na grafičkim karticama. Danas vjerojatno najpoznatiji projekt koji koristi i grafičke kartice za izračunavanja jest Folding@home [4] koji se bavi istraživanjem raznih bolesti poput Alzheimerove, Parkinsonove i raznih oblika rakova... Također sve češće se danas grafičke kartice koriste kao zamjena za superračunala, jer omogućavaju ekvivalentne performanse, a sve to po daleko manjoj cijeni i potrošnji.

Nažalost, također postoje i nedostatci. Ono što se ne vidi iz rezultata testiranja je vrijeme utrošeno na optimiziranje programskog koda. Većina aplikacija se može relativno jednostavno prilagoditi izvođenju na grafičkom procesoru, ali za iskorištavanje punog potencijala situacija postaje znatno kompliciranija. Kako bi se postigle maksimalne performanse i zaobišla sva ograničenja potrebno je puno truda i vremena, kao i znanje o hardveru na kojem će se programsko ostvarenje izvoditi.

Nadolaskom budućih generacija grafičkih kartica, vrijeme potrebno za razvoj aplikacija trebalo bi biti znatno skraćeno, a samo programiranje olakšano.

7. Literatura

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

[2] Frederic Patin, *An Introduction To Digital Image Processing*, 28.3.2010

http://www.gamedev.net/reference/programming/features/imageproc/page2.asp

[3] Young, I.T., Gerbrands, J.J., van Vliet, L.J., *Image Processing Fundamentals*, 28.3.2010

http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip.html

[4] Wikipedia, *Folding*@home, 30.5.10.

http://en.wikipedia.org/wiki/Folding@home

[5] *NVIDIA CUDA Compute Unified Device Architectur*, 24.4.2007.

http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/0.8/NVIDIA_CUDA_Programming_Guide_0.8.2.pdf , 28.3.2010

[6] Lee, S., *CUDA Convolution*, 8.12.2008.

http://www.evl.uic.edu/sjames/cs525/final.html, 28.3.2010

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

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

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

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

[9] 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.

[10] Masood, J., *Nvidia CUDA,* 15.10.2009.,

http://www.hardwareinsight.com/nvidia-cuda/, 22.5.2010.

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

http://www.tomshardware.com/news/Nvidia-GPU-Huang-570x,8544.html, 20.4.2010.

8. Sažetak / Abstract

U ovome radu opisuje se arhitektura modernih grafičkih kartica, okolina potrebna za razvoj CUDA aplikacija, te potencijalna primjena grafičkih procesora za obradu slika. Također ukratko je navedena i neizbježna usporedba sa centralnim procesorima.

Temelj većine operacija obrade slika jest konvolucija, te je stoga i ista implementirana za izvođenje na centralnom procesoru, kao i na grafičkom. Izvedba prilagođena centralnom procesoru implementirana je u programskom jeziku C, dok izvedba za grafički procesor napisana je u CUDA jeziku. Ukratko je opisana osnovna CUDA implementacija kao i implementirane optimizacije.

Na kraju prikazani su i komentirani rezultati testiranja implementiranih rješenja. Iz rezultata testiranja se moglo vidjeti da je moguće postići i ubrzanja od nekoliko stotina puta.

Ključne riječi: računalni vid, obrada slika, konvolucija, CUDA, separabilni filteri.

Application of graphics hardware for image processing operations

This paper describes the architecture of a modern graphics card, necessary environment for development of CUDA applications and the potential application of graphics processor units for image processing. Also there's a brief comparison with central processor units.

The foundation for majority of image processing operations is convolution and is therefore implemented to perform on both central and graphical processors. Implementation adapted for central processors is written in C programming language while graphics processor implementation is written in CUDA. Basic CUDA implementation is briefly described as well as implemented optimizations.

In the end, results acquired by testing of implemented solution are displayed and commented on. One can see from the results that it is possible to archive acceleration of a few hundred times.

Keywords: computer vision, image processing, convolution, CUDA, separable filters