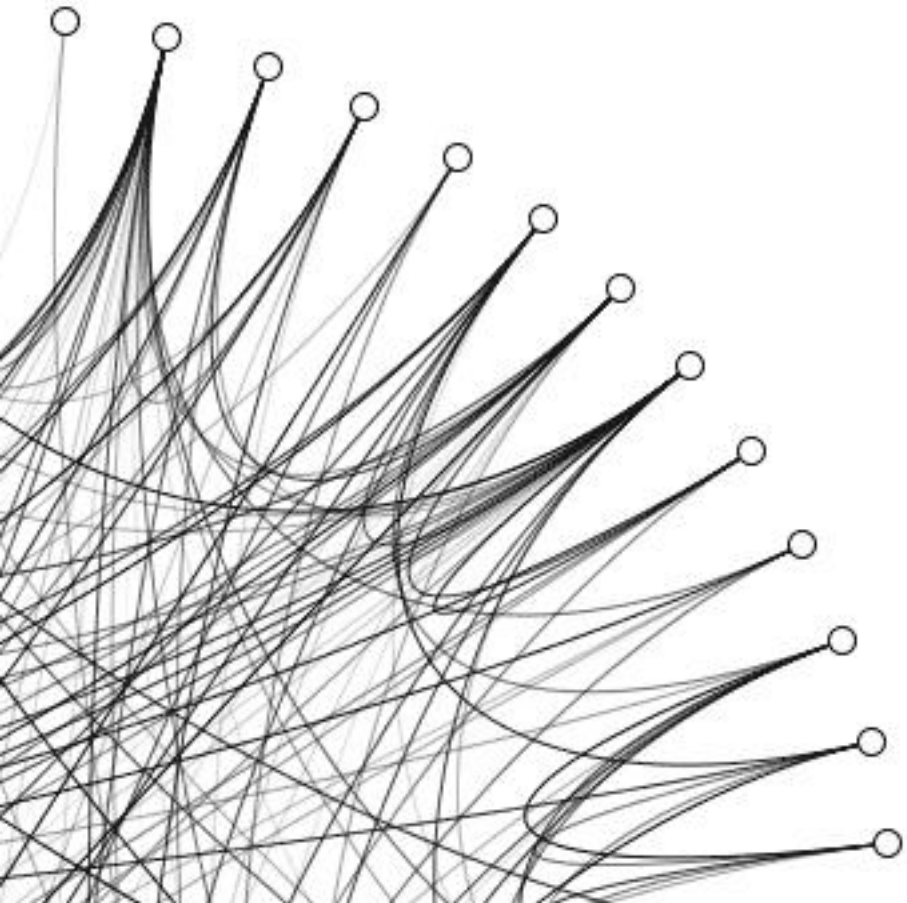
SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Programiranje na GPU procesorima uz pomoć CUDA arhitekture

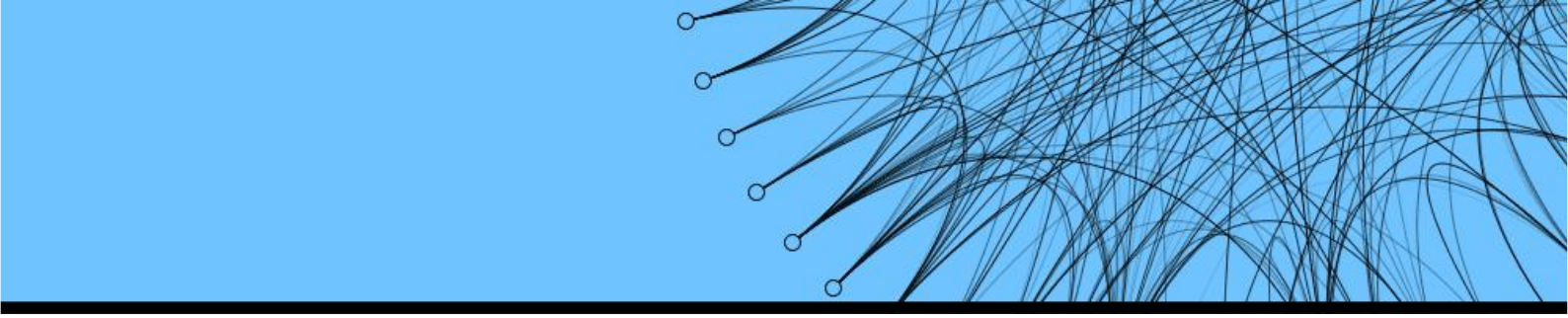
Matija Piškorec

mentor: prof.dr.sc. Branko Jeren



lipanj 2009.

Sadržaj



Uvod

U potrazi za sve snažnijim računalima i više računalne moći od posebnog interesa su sustavi koji nude mogućnost paralelnog rada. Riječ je o sustavima koji djeluju bilo lokalno - u vidu jednog superračunala s više procesorskih jedinica, ili distribuirano na veći broj običnih računala na Internetu. Kao primjeri potonjeg mogu se navesti projekti SETI@Home i Folding@Home koji preko Interneta uspješno distribuiraju dijelove proračuna koji se onda izvode na računalima dobrovoljaca i koji pri završetku vraćaju rezultate centrali koja ih je i poslala. Ovisno o popularnosti projekta takvi sustavi po proračunskoj snazi mogu višestruko nadmašiti i najjača trenutno dostupna superračunala. Tako prema podatcima od 19. travnja 2009. Folding@Home ima snagu od 8.7 petaflopsa[1] što je oko osam puta više od superračunala IBM Roadrunner koje slovi za trenutno najjače superračunalo. Detaljnija statistika za navedeni projekt može se naći na [1]. Većinu te snage dopridonose upravo GPU (eng. *Graphics Processing Unit*) procesori tj. grafičke kartice na računalima korisnika.

Izniman komercijalni interes pratio je (i često podupirao) razvoj računalne grafike tako da je ona danas sveprisutna u širokom rasponu aplikacija. Usporedo s njom razvijali su se i specijalizirani procesori koji su posebno optimizirani za rad s grafičkim proračunima. Priroda takvih proračuna izrazito je paralelna i ta karakteristika se odrazila i na arhitekturi takvih procesora. S vremenom se razvila ideja kako tu karakteristiku grafičkih procesora iskoristiti i za proračune općenite naravi - onih koji su nevezani za grafičke aplikacije. Tako nastaje koncept GPGPU (eng. *General-purpose computing on Graphics Processing Units*) koji u zadnje vrijeme dobiva sve više pažnje.

Programiranje na GPU procesorima nekada je značilo pisanje programa u jeziku samog procesora. Danas su na raspolaganju arhitekture koje omogućuju programiranje aplikacija u poznatom okruženju koristeći široko prihvaćene programske jezike. Jedna od njih je i CUDA (eng. Compute Unified Device Architecture) razvijena od tvrtke NVIDIA i posebno prilagođena za njihove grafičke procesore.

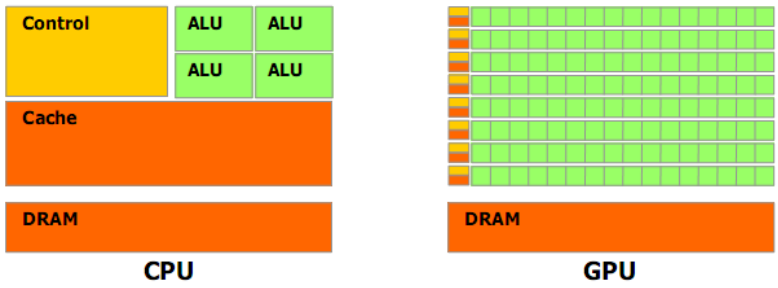
Svrha ovog seminara je da pruži osnovne informacije o razvoju paralelnih aplikacija u CUDA programskom modelu i da posluži kao informativni uvod svima onima koji se žele okušati u programiranju aplikacija na GPU-ovima. Za potpunije i iscrpnije informacije preporuča se službena dokumentacija za programiranje u CUDA okruženju (na engleskom jeziku) koja se može naći na [2].

[1] petaflops je (10^{15}) flopsa (eng. floating points per second)

Poglavlje 2.
Sklopovska implementacija CUDA modela

Kako bismo razumijeli zašto su GPU procesori tako pogodni za složene proračune općenite naravi korisno je pogledati sliku 1 koja shematski prikazuje udio pojedinih tranzistora na procesoru.

GPU procesori su dizajnirano tako da je veći udio tranzistora namjenjen procesuiranju podataka nego privremenom spremanju (engl. *data caching*) i kontroli toka (engl. *flow control*) - funkcije koje su bitne za uspješno izvršavanje operacijskog sustava i svakodnevnih aplikacija. Takav dizajn omogućuje paralelno izvršavanje puno većeg broja instrukcija no postavlja ograničenja prilikom kontrole programskog toka. To znači, između ostalog, da se maksimalna efikasnost postiže ako su instrukcije jednakog tipa. Svako odstupanje od tog zahtjeva utječe na performanse sustava. Srećom, taj uvjet je najčešće moguće ostvariti prilikom grafičkih proračuna gdje je potrebno primjeniti jednu vrstu instrukcije na veliku količinu podataka. Također, moguće su i brojne primjene na probleme općenite naravi koji nisu vezani za računalnu grafiku.



Slika 1. GPU procesor više tranzistora posvećuje procesuiranju podataka

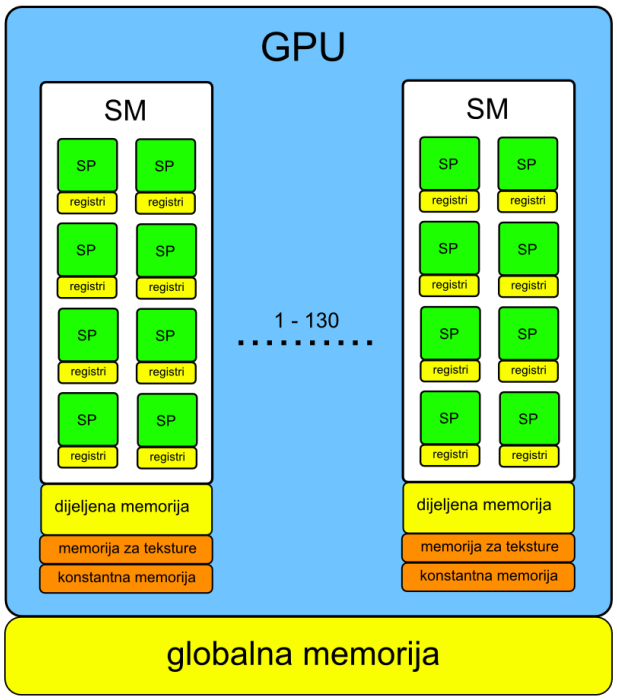
CUDA model omogućuje programerima razvoj paralelnih aplikacija za NVIDIA-ine grafičke procesore. On uključuje programski model koji definira apstrakcije koje su na raspolaganju programerima i sklopovsku implementaciju na NVIDIA-inim grafičkim procesorima, a koja omogućuje paralelno izvršavanje velikog broja dretvi (eng. *threads*). O CUDA programskom modelu biti će više riječi u sljedećem poglavlju.

U nastavku poglavlja opisuje se arhitektura NVIDIA-inih grafičkih procesora i način na koji oni upravljaju višedretvenim okruženjem.

2.1. Arhitektura NVIDIA-inih GPU procesora

Okosnica NVIDIA-inih GPU procesora u CUDA modelu je skup višedretvenih multiprocesora (eng. *multithreaded Streaming Processors*, skraćeno SM). Ovisno o modelu grafičkog procesora može ih biti i do 130, no uglavnom ih je oko 30. Svaki multiprocesor sastoji se, između ostalog, od osam skalarnih procesora (eng. *Scalar Processors*, skraćeno SP) i dijeljene memorije. Svi multiprocesori imaju pristup globalnoj memoriji na grafičkom procesoru.

Shematski prikaz NVIDIA-inih grafičkih procesora prikazan je na slici 2.



Slika 2. Arhitektura NVIDIA grafičkih procesora koji podržavaju CUDA programski model. Bitni čimbenik koji utječe na performanse je broj multiprocesora kojih, ovisno o modelu grafičkog procesora, može biti i do 130.

Memorija na multiprocesoru podijeljena je u četiri osnovna tipa:

1. lokalni 32-bitni registri (ima ih 8192 ili 16384, ovisno o modelu procesora)
2. dijeljena memorija (eng. *shared memory*) koju dijele svi SPovi
3. konstantna privremena memorija (eng. *constant cache*) koju dijele svi SPovi ali iz koje se može samo čitati (eng. *read-only*)
4. privremena memorija za teksture (eng. *texture cache*) iz koje se također može samo čitati

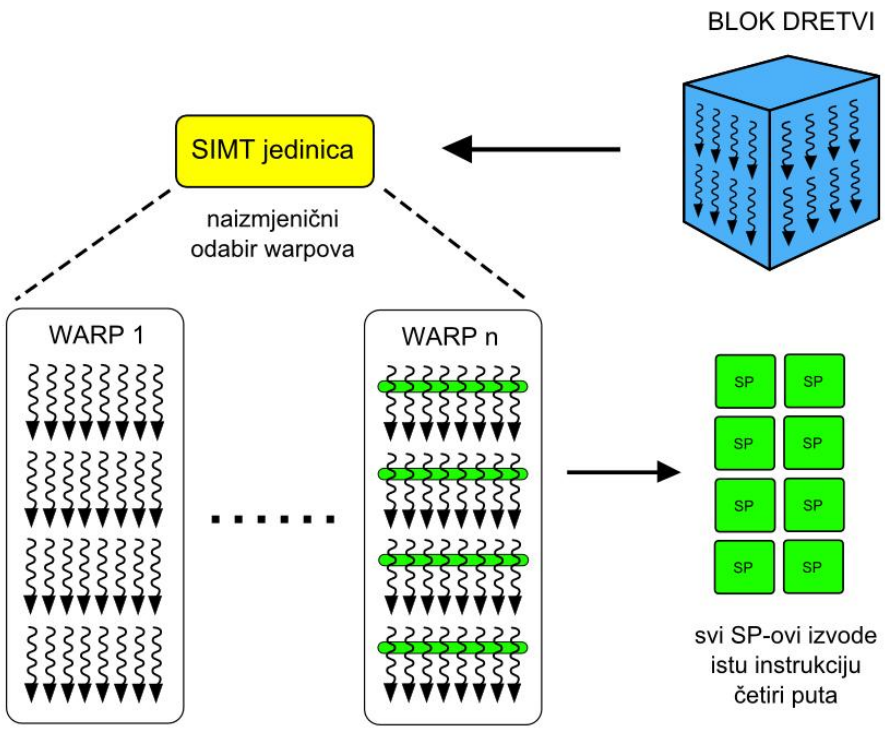
U slučaju konstantne memorije samo CPU ima isključivo pravo pisanja dok GPU može samo čitati. Sve navedene memorije podjednako su brze i preporuča se njihovo korištenje umjesto puno sporije globalne memorije.

2.2. SIMT arhitektura

CUDA model oslanja se na SIMT (eng. *Single Instruction Multiple Threads*) arhitekturu koja definira ponašanje skupine dretvi. Slično kao i SIMD (eng. *Single Instruction Multiple Data*) arhitektura također naglasak stavlja na to da jedna instrukcija kontrolira ponašanje više procesnih elemenata. Razlika je u tome što kod SIMT-a programer ne treba dijeliti podatke u vektore i što su dretve u mogućnosti da se granaju neovisno jedna o drugoj. To pojednostavljuje razvoj aplikacija no može dovesti do nepotrebnih usporavanja i svakako je jedna od bitnijih stvari na koje treba misliti prilikom pisanja algoritama.

Kako bi se progamerima olakšao razvoj paralelnih aplikacija u sklopu CUDA programskog modela razvijene su apstrakcije blokova i grida koje definiraju podjelu dretvi po multiprocesorima. O njima će biti više riječi u idućem poglavlju - zasada je dovoljno znati da je programer u mogućnosti podijeliti dretve po multiprocesorima u skupine koje se nazivaju blokovi koji se mogu izvršavati neovisno jedni o drugima.

SIMT jedinica unutar multiprocesora zadužena je za raspodjelu dretvi po njegovim skalarnim procesorima. Svaka dretva pridružuje se jednom skalarnom procesoru na kojem se neovisno izvršava s vlastitim skupom registara i instrukcijskim adresama. Kada SIMT jedinica dobije jedan ili više blokova koje multiprocesor mora obraditi ona prvo svaki blok podijeli na skupine od 32 dretve koje se zovu warpovi.



Slika 3. Shema SIMT arhitekture - svaki blok dijeli se na warpove od 32 dretve koji se naizmjenično izvršavaju. Nad svim dretvama u istom warpu izvode se iste instrukcije za što je potrebno četiri ciklusa sata (eng. clock cycle) po instrukciji.

Opravdavajući svoj naziv (SIMT - jedna instrukcija za više dretvi), nad svim dretvama unutar jednog warpa izvršavaju se identične instrukcije. To višetruko ubrzava izvršavanje dretvi pod uvjetom da dretve unutar jednog warpa ne divergiraju - primjerice, ako se neka od uvjetnih naredbi različito evaluira za različite dretve. U tom slučaju gubi se paralelizam i dretve se izvršavaju slijedno, jedna po jedna, dok se ponovno ne dostignu instrukcije koje su za sve jednake. SIMT jedinica naizmjence odabire aktivne warpove pri čemu za svaki izvrši po jednu instrukciju. Na taj način se maksimalno iskorištava procesorsko vrijeme jer dretve koje su na čekanju - primjerice, zbog pristupa memoriji, ne usporavaju izvršavanje dretvi u ostalim warpovima.

Redoslijed kojim se izvršavaju warpovi unutar bloka pa čak i redoslijed izvršavanja blokova nije definiran. Ukaže li se potreba za sinkronizacijom dretvi može se koristiti funkcija `__syncthreads()` nakon čega se garantira da su sve dretve unutar bloka izvršile sve prethodne naredbe i da su sva čitanja i pisanja u dijeljenu memoriju završena. No sa sinkronizacijom na taj način treba biti oprezan. Primjerice, ako se navedena funkcija stavi u tijelo uvjetne naredbe koja se različito evaluira za neke dretve iz istog bloka dogodit će se zastoj.

IProgramerima je ponašanje SIMT jedinice uglavnom nevažno jer raspodijelu dretvi rade na apstraktnijoj razini (prema CUDA programskom modelu, na blokove) no teži li se maksimalno efikasnom iskorištavanju mogućnosti grafičkih procesora korisno je u obzir uzeti i te stvari. O metodama optimizacije i kako SIMT arhitektura i podjela na warpove utječe na njih bit će rijeći u jednom od idućih poglavlja.

2.3. Revizije CUDA modela

Razvojem sve naprednijih i moćnijih NVIDIA-inih grafičkih procesora i CUDA model je trebao proći kroz nekoliko revizija (eng. *compute capability*) kako bi u potpunosti iskoristio sve mogućnosti novih tehnologija. Revizije se prate preko glavnog (eng. *major*) i sporednog (eng. *minor*) revizijskog broja. Trenutno svi NVIDIA-ini procesori imaju glavni revizijski broj 1 što znači da se oslanjaju na istu arhitekturu. Sporedni revizijski brojevi predstavljaju inkrementalna poboljšanja osnovne arhitekture, najčešće dodavanjem novih funkcionalnosti.

	1.0	**1.1**	**1.2**	**1.3**
maksimalni broj dretvi u bloku	512			
maksimalne dimenzije bloka	(512,512,64)			
maksimalna dimenzija grida	(65535, 65535)			
veličina warpa	32 dretve			
broj registara po SM-u	8192		16384	
dijeljena memorija po SM-u	16KB			
konstantna memorija	64KB			
aktivni blokovi po SM-u	8			
aktivni warpovi po SM-u	24		32	
aktivne dretve po SM-u	768		1024	
podrška za atomske funkcije		DA		
podrška za warp-vote funkcije			DA	
podrška za dvostruku preciznost				DA

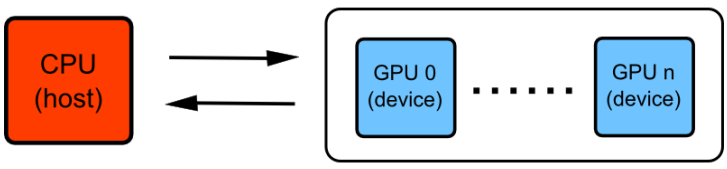
Poglavlje 3.
CUDA programski model

Osnovni zadatak CUDA programskog modela je da omogući efikasnu i transparentnu komunikaciju s paralelnim sklopovljem. Ključni koncepti kojima se to postiže su hijerarhija dretvi, dijeljene memorije i sinkronizacija barijerama. Oni su programeru predstavljeni kao minimalne nadogradnje C programskog jezika pa je upoznavanje i privikavanje na CUDA programski model olakšano. Nadalje, model potiče programera da poštuje osnovne principe razvoja paralelnih programa. Problem se prvo podijeli na jednostavnije podprobleme koji se mogu rješavati nezavisno, potom se ti podproblemi još finije podijele i izvršavaju kooperativno u paraleli. Za ovaj drugi dio brine se hijerarhija dretvi koja obavlja stvarnu raspodijelu podproblema po fizičkim procesorima. Pri tome važnu ulogu ima i sinkronizacija barijerama koja se obavlja nakon svake instrukcije i koja omogućuje efikasnu i brzu komunikaciju između dretvi. Može se reći da su programi razvijeni za CUDAu izrazito sitnozrnati (eng. *granularity* - omjer između vremena potrošenog za komunikaciju između procesa i vremena potrošenog za samo računanje). Time je zadovoljeno svojstvo skalabilnosti jer se tako razvijeni programi mogu izvršavati na bilo kojem broju procesora.

U ovom poglavlju predstavit će se glavni koncepti na kojima se zasniva razvoj aplikacija za CUDA arhitekturu. Riječ je o kernel funkcijama kojima se pristupa samim GPU procesorima, hijerarhiji dretvi i hijerahiji memorije. No prije toga par riječi o komunikaciji CPU procesora s grafičkim procesorom.

3.1. Komunikacija s CPU-om

Na slici 1. ilustrirana je osnovna pretpostavka CUDA programskog modela - dretve namijenjene grafičkom procesoru mogu se izvoditi na proizvoljnom broju nezavisnih GPU procesora (eng. device) dok se glavna dretva izvodi na CPU-u (eng. host). Pri tome je važno napomenuti da svaka dretva na CPU-u može istovremeno pristupiti samo jednom GPU-u pa je za paralelni rad više grafičkih procesora potrebno pokrenuti i više dretvi koje vrte glavni program. CPU raspodijelu posla na GPU izvršava pozivom kernel funkcija koje su opisane u sljedećem poglavlju.



Slika 4. Jedan klasični procesor (CPU) može kontrolirati više grafičkih procesora (GPU).

3.2. Kernel funkcije

CUDA razlikuje tri vrste funkcija s obzirom na to gdje se izvršavaju i tko ih ima pravo pozivati. U tom slučaju razlikujemo klasični CPU procesor (eng. *host*) i jedan ili više GPU procesora (eng. *device*). Pozivi kernel funkcija su asinkroni što znači da se odmah nakon poziva glavni program na CPU-u nastavlja izvršavati. Zato je uputno pozive kernel funkcija stavljati što ranije a nakon njih dio koda koji se izvršava na CPU-u i koji je neovisan o podatcima s GPU-a. Ipak, prije nego se kernel funkcija uistinu i počne izvršavati na GPU-u potrebno je da sve kernel funkcije koje su prethodno pozvane završe sa svojim radom. Ovo se može kontrolirati uz pomoć streamova koji omogućuju paralelno izvršavanje više kernel funkcija istovremeno. Streamovi su opisani u kasnijim poglavljima.

Kao što je već spomenuto u poglavlju 2, kernel funkcije imaju neka ograničenja koja ih razlikuju od običnih funcija koje se izvršavaju na CPU-u. Primjerice, u CUDA programskom modelu kernel funkcije ne mogu:

1. biti rekurzivne
2. deklarirati statične varijable unutar tijela funkcije
3. imati varijabilan broj argumenata
4. vraćati vrijednost (tj. moraju biti tipa void)
5. pozivati funkcije višeg tipa (npr. funkcije za ispis na ekran)

Naravno, i neki uobičajeni konstrukti viših programskih jezika - objekti, klase i naslijeđivanje u objektno-orijentiranom modelu primjerice, nisu podržani u kernel funkcijama. Ipak, navedena ograničenja neće bitno utjecati na funkcionalnosti koje se mogu postići ovakvim modelom. Kodovi za CPU i GPU procesor se zasebno prevode što znači da su za dio koda koji se izvodi na CPUu i dalje dostupne sve funkcionalnosti iz viših programskih jezika.

Kernel funkcije označavaju se dodavanjem odgovarajućih prefiksa ispred njihova imena:

__device__
Funkcija se izvršava na GPUu i poziva isključivo iz GPUa.

__host__
Funkcija se izvršava na CPUu i poziva se isključivo iz CPUa. U slučaju da funkcija nema definirani prefiks prefiks **__host__** se podrazumijeva (funkcionira kao standardna C funkcija).

__global__
Punokrvna kernel funkcija jer se izvršava na GPUu i poziva isključivo s CPUa.

Primjerice, definicija kernel funkcije tipa __global__ izgleda ovako:

```
__global__ void ZbrojiVektore(float *A, float *B, float *C);
```

Primjećujemo također da je funkcija tipa void što je jedno od ograničenja spomenutih u poglavlju 2.1. Prilikom poziva kernel funkcija potrebno je navesti konfiguraciju izvršavanja (eng. *execution configuration*) koja definira broj dretvi koje će izvršavati danu funkciju. Više o tome u idućem poglavlju.

3.3. Hijerarhija dretvi

Kernel funkcije funkcioniraju kao predložak za niz instrukcija koje izvršava skupina dretvi. Prilikom pozivanja kernel funkcije potrebno je definirati su dimenzije grida i bloka koji će izvršavati određeni broj dretvi. To se radi pomoću konfiguracije izvršavanja koja se označava pomoću '<<' i '>>' nakon imena funkcije:

```
ZbrojiVektore<<Dg, Db, Ns, S>>(A, B, C);
```

Opcije koje je tako moguće definirati su:

<< Dg, Db, Ns, S >>
Dg je tipa dim3 i definira dimenzije i veličinu grida. **Dg.x * Dg.y** je broj blokova koji se izvršavaju, **Dg.z** mora biti jednako 1.
Db je tipa dim3 i definira dimenzije i veličinu svakog bloka. Db.x * Db.y * Db.z je jednako broju dretvi koje svaki blok izvršava.
Ns je tipa size_t i definira količinu dodatne dijeljene memorije u bajtovima koja se dinamički alocira po bloku uz statički definiranu memoriju. Standardna vrijednost je 0.
S je tipa cudaStream i definira pripadni stream. Standardna vrijednost je 0.

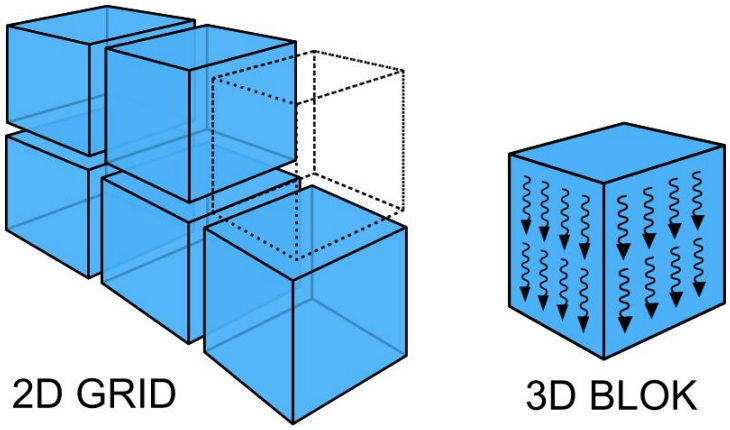
Primjerice, kernel funkcija koja zbraja dvije matrice tako da posao zbrajanja pojedinačnih elemenata raspodijeli na odgovarajući broj dretvi može se definirati i pozvati na sljedeći način:

```
__global__ void ZbrojiMatrice(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    dim3 dimBlok(N, N); // definiranje dimenzija bloka pomoću dim3 tipa
    ZbrojiMatrice<<<1, dimBlok>>>(A,B,C); // pozivanje kernel funkcije
}
```

Naravno, nema smisla da svaka dretva u bloku izvršava u potpunosti identičan niz instrukcija. Podjela posla se najčešće vrši tako da svaka dretva pristupa određenom dijelu zajedničke memorijske strukture. U takvim slučajevima praktično je kao osnovicu za indeksiranje koristiti varijable koje su specifične za svaku dretvu - u gornjem primjeru to su varijable **threadIdx.x** i **threadIdx.y** koje definiraju položaj dretve unutar bloka. Zapravo, dretve su unutar bloka definirane s tri varijable - uz gore spomenute može se koristiti i varijabla **threadId.z** pa se tako dobiva trodimenzionalno indeksiranje. U slučaju da ona nije definirana pretpostavljena vrijednost je 1. S druge strane, blokovi se unutar grida identificiraju s dvije varijable pa je njegova struktura dvodimenzionalna.

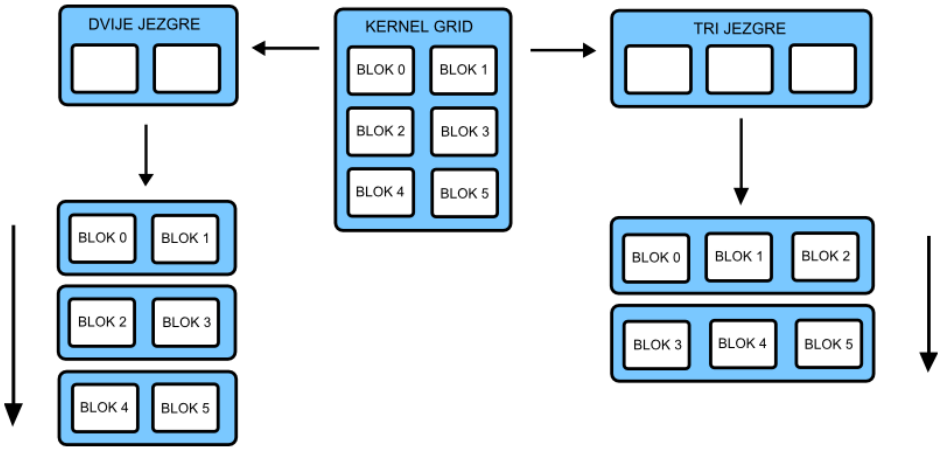
Shematski prikaz podjele dretvi po blokovima i blokova po gridu može se vidjeti na slici 5:



Slika 5. Hijerarhija dretvi, blokova i grida. Dretve su u blokovima indeksirane s tri koordinate a blokovi u gridu s dvije.

Zadatci koji su dodijeljeni različitim blokovima (tj. dretvama koje čine blok) moraju biti neovisni - redoslijed izvođenja blokova nije garantiran i ovisi, između ostalog, o konkretnom modelu grafičkog procesora na kojem se program izvodi, prije svega o broju dostupnih multiprocesora. CUDA programski model garantira da će se svaki blok izvoditi isključivo na jednom multiprocesoru što znači da će dretvama unutar svakog bloka biti dostupni resursi spremljeni u lokalnoj memoriji multiprocesora, primjerice, u dijeljenoj memoriji. Želi li se dijeliti podatke između dretvi iz različitih blokova potrebno je koristiti globalnu memoriju na grafičkom procesoru.

Takvom podjelom olakšan je razvoj aplikacija za CUDAu jer programer ne mora uzimati u obzir model grafičkog procesora (prije svega broj multiprocesora) na kojem će se njegov program izvršavati. Jednom prevedeni programi su skalabilni u smislu da više nije važno na kojem broju multiprocesora se izvode pa nema potrebe za ponovnim prevođenjem programskog koda. Svojstvo je ilustrirano na slici 6:



Slika 6. Apstrakcija grida omogućuje da se blokovi efikasno rasporede na slobodne jezgre (tj. multiprocesore, pošto je svaki multiprocesor zadužen za izvršavanje jednog bloka). Tako se isti program može izvoditi na različitim arhitekturama grafičkih procesora bez potrebe za ponovnim prevođenjem programskog koda.

3.4. Hijerarhija memorije

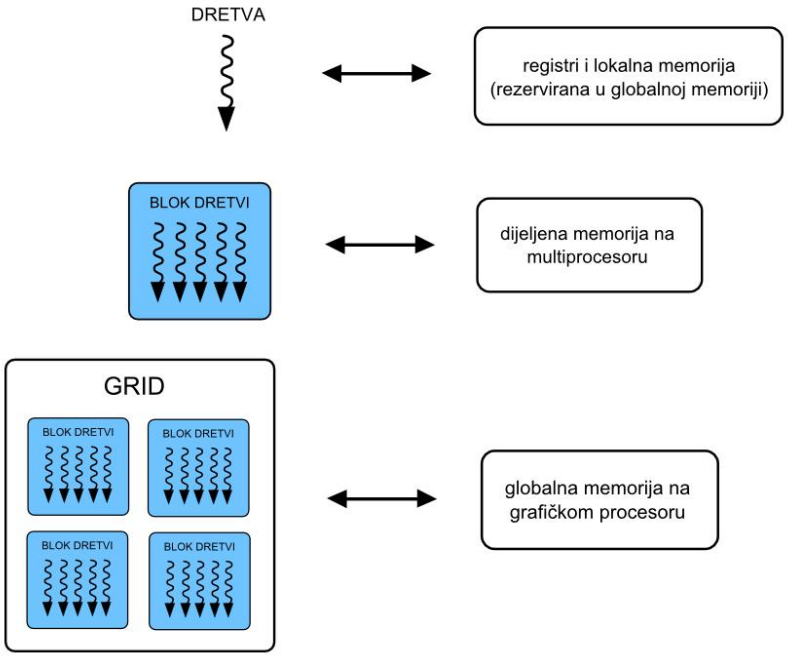
CUDA programski model točno definira koji memorijski prostori su dostupni pojedinim programskim konstruktima - dretvama, blokovima i gridu. Performanse izvršnog prgrama uvelike ovise o tome koja se memorija i na koji način koristi. Slijedi kratki prikaz svih vrsta memorija:

Registri
Najbrža vrsta memorije na multiprocesoru. Dostupni su isključivo pojednim dretvama i trajanjem su vezani za njih.

Dijeljena memorija
Uz registre, najbrža vrsta memorije dostupna svim dretvama u bloku. Ukoliko joj se pristupa na točno određeni način (tako da nema bank konflikata, o čemu će biti riječ u idućim poglavljima) brzina joj je usporediva s registrima. Količina dostupne dijeljene memorije definira se prilikom poziva kernel funkcije. Traje koliko i blokovi. Uz nju blokovima je dostupna i memorija za teksture i konstantna memorija. Međutim, pravo pisanja u nju imaju samo dretve koje se izvršavaju na CPUu dok dretve na GPUu mogu iz nje samo čitati.

Globalna memorija
Dostupna svim dretvama i svim blokovima kao i CPUu i trajanjem je vezana za aplikaciju. Može biti i do 150 puta sporija od registara ili dijeljene memorije.

Lokalna memorija
Dostupna isključivo pojedinim dretvama i trajanjem je vezana za njih no fizički je smještena u globalnoj memoriji. Zbog toga može biti i do 150 puta sporija od registara ili dijeljene memorije. U nju se obično smještaju varijable koje ne stanu u dijeljenu memoriju ili registre, i to obično radi prevoditelj automatski pa na to treba obratiti pozornost.



Slika 1. Hijerarhija memorije u CUDA programskom modelu definira koji memorisjki prostori su dostupni dretvama, blokovima i gridu.

Poglavlje 4.
Programiranje za CUDA-u

Cilj CUDA programskog modela je da programerima omogući jednostavni pristup paralelnim mogućnostima grafičkih procesora. U tu svrhu razvijen je određeni skup proširenja za programski jezik C. Ovo poglavlje je ustrojeno tako da pruži osnovne smjernice za programiranje u CUDA programskom modelu. Poglavlja su navedena redom kojim se otprilike pokušava pratiti uobičejeni razvoj CUDA aplikacije - prvo nešto o inicijalizaciji GPU-a i alokaciji memorije (radnje koje se izvršavaju na CPU-u) a potom predstavljanje osnovnih koncepata za razvo kernel funkcija (koje se izvršavaju na GPU-u). Za iscrpniji pregled navedenih koncepata preporuča se službeni priručnik za programiranje - "NVIDIA CUDA Programming Guide".

4.1. Inicijalizacija GPU-a

Ne postoji eksplicitna funkcija za pokretanje API-a. Umjesto toga, API se inicijalizira prvi put kad se pokrene neka CUDA funkcija. Po potrebi se pomoću funkcija **`cudaGetDeviceCount()`** i **`cudaGetDeviceProperties()`** može provjeriti koliko je GPU-a dostupno sustavu i koje su njihove karakteristike.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
}
```

Sada su u strukturi deviceProp spremljene informacije o GPU-u kojem glavna programska dretva može pristupiti. Neka njezina bitnija polja su:

`char name[256];`
ASCII string u koji je pohranjeno ime GPU-a.

`size_t totalGlobalMem;`
Količina globalne memorije na GPU-u u bajtovima.

`size_t sharedMemPerBlock;`
Maksimalna količina dijeljene memorije dostupna bloku dretvi u bajtovima.

`int regsPerBlock;`
Maksimalni broj 32-bitnih registara dostupnih jednom bloku dretvi.

`int warpSize;`
Broj dretvi u warpu.

`int maxThreadsPerBlock;`
Maksimalni broj dretvi u jednom bloku.

`int maxThreadsDim[3]; int maxGridSize[3];`
Maksimalna veličina svake dimenzije bloka i grida.

`size_t totalConstMem;`
Količina konstantne memorije na GPU-u u bajtovima.

`int major; int minor;`
Glavna i sporedna verzija revizije kojoj GPU pripada. Primjerice, ako je verzija revizije 1.3 onda je glavna verzija revizije 1 a sporedna 3.

`int clockRate;`
Frekvencija sata u kilohercima.

Željeni GPU može se odabrati pomoću funkcije **`cudaSetDevice()`**

```
cudaSetDevice(device);
```

To je bitno ako se u sustavu nalazi više grafičkih procesora i želimo svakoj dretvi glavnog programa pridružiti jedan. Pretpostavljena vrijednost je 0 pa ako je sustavu dostupan samo jedan GPU ova se funkcija može i izostaviti.

4.2. Alokacija memorije

Alokacija linearne memorije (u globalnom memorijskom prostoru GPU-a) vrši se pomoću funkcija **`cudaMalloc()`** ili **`cudaMallocPitch()`**. Oslobađanje se vrši pomoću **`cudaFree()`**.

Primjerice, alokacija polja od 256 elementata tipa float izgleda ovako:

```
float * array;
cudaMalloc( (void**)&array, 256 * sizeof(float) );
```

Za dvodimenzionalna polja optimalnije je koristiti funkciju **`cudaMallocPitch()`** koja u obzir uzima poseban način organizacije memorije koji minimizira usporavanja zbog eventualnih konflikata prilikom istovremenog čitanja slijednih memorijskih lokacija.

```
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch, width * sizeof(float), height);
```

Indeksiranje polja u kernel funkciji sada se provodi na sljedeći način:

```
float* element = (float*)((char*)devPtr + Redak * pitch) + Stupac;
```

Općenito, u slučaju višedimenzionalnih polja učitavanje iz memorije se može dodatno ubrzati ako se dijelovi polja iz globalne memorije prvo učitaju u dijeljenu (koja je brža) i potom indeksiraju pojedinačno. U donjem primjeru za indeksiranje svakog redka zadužena je jedna dretva u zasebnom bloku.

```
// CPU kod
float* devPtr;
int pitch;
int kolicinaDijeljeneMemorije = sirina * sizeof(float);
cudaMallocPitch((void**)&devPtr, &pitch, sirina * sizeof(float), visina);
myKernel<<<visina, 1, kolicinaDijeljeneMemorije>>>(devPtr, pitch);

// GPU kod
__global__ void myKernel(float* devPtr, int pitch)
{
    extern __shared__ float* redak_elemenata;
    redak_elemenata = (float*)((char*)devPtr + blockIdx.x * pitch);
    for (int i = 0; i < sirina; ++i)
    {
        float element = redak_elemenata[i];
    }
}
```

Ukaže li se potreba za kopiranjem dijela memorije iz CPU-a na GPU može se koristiti funkcija **`cudaMemcpy()`**. Prikazan je dio koda koji kopira polje data u globalnu memoriju GPU-a:

```
float data[256];
int size = sizeof(data);
float* devPtr;
cudaMalloc((void**)&devPtr, size);
cudaMemcpy(devPtr, data, size, cudaMemcpyHostToDevice);
```

Umjesto u globalnu memoriju podatke je moguće kopirati i u neku drugu vrstu memorije na GPU-u, primjerice u konstantnu pomoću funkcije **`cudaMemcpyToSymbol()`**. U slučaju konstatne memorije to je ujedno i jedini način upisivanja podataka u nju - GPU ima samo mogućnost čitanja.

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
```

Umjesto standardne alokacije memorije na CPU-u uz pomoću **`malloc()`** funkcije također je moguće alocirati i page-lock memoriju za što služi funkcija **`cuMemAllocHost()`**. Jedna od prednosti je što je prijenos podataka s CPU-a na GPU (ali ne i obratno!) brži ako je memorija na CPU-u alocirana kao page-lock memorija. Dodatna prednost je što se tako omogućava efikasno iskorištavanje streamova za istovremeno obavljanje prijenosa podataka i izvršavanje dretvi na GPU-u. Ipak, s alokacijom page-lock memorije ne treba pretjerati jer se time smanjuje količina fizičke memorije dostupne operacijskom sustavu pa se smanjuju i performanse cijelog sustava.

```
float* hostPtr;
cuMemAllocHost((void**)&hostPtr, 2 * size);
```

4.3. Tipovi varijabli

Vrste funkcija u CUDA programskom modelu i način njihova pozivanja već je demonstriran. Ovdje će se navesti posebni tipovi varijabli koji se mogu koristiti u kernel funkcijama. Dakle, unutar kernel funkcija mogu se koristiti svi jednostavni tipovi podataka dostupni u C programskom jeziku. Iznimka su brojevi dvostruke preciznosti (eng. *double*) koji su podržani jedino na grafičkim procesorima od verzije 1.3. Njima se prilikom deklaracije ispred imena mogu pridjeliti dodatne oznake koje definiraju u kojoj će memoriji biti spremljene.

`__device__`
Označava varijablu koja je spremljena u globalnoj memoriji GPU-a, dostupna svim dretvama unutar grida i vezana trajanjem za aplikaciju.

`__constant__`
Označava varijablu koja je pohranjena u konstantnoj memoriji GPU-a, dostupna svim dretvama u gridu i vezana trajanjem za aplikaciju. Samo CPU ima isključivo pravo pisanja u nju dok GPU može samo čitati.

`__shared__`
Označava varijablu koja je pohranjena u dijeljenoj memoriji na multiprocesoru i dostupna je samo dretvama unutar istog bloka. Vezana je trajanjem za blok. Tek nakon poziva funkcije **`__syncthreads()`** se garantira da će sve promjene u dijeljenoj memoriji biti vidljive svim ostalim dretvama u bloku. Inicijalizacija vrijednosti nije dopuštena odmah prilikom deklaracije.

U slučaju da varijabla koja se koristi unutar kernel funkcije nema pridijeljenu ni jednu od gore navedenih oznaka u pravilu će se smjestiti u registar. Ipak, prevoditelj će u nekim slučajevima, prilikom nedostatka memorijskog prostora primjerice, odlučiti spremiti varijablu u lokalnu memoriju (tj. u globalnu memoriju GPU-a) što može uvelike produžiti vrijeme pristupa.

Varijable u dijeljenoj memoriji imaju dodatnu mogućnost da se deklariraju kao externe i u tom slučaju memorija se alocira dinamički prilikom izvođenja programa.

```
extern __shared__ float Polje[];
```

Varijable definirane na ovaj način trebaju se eksplicitno deklarirati unutar kernel funkcija pomoću pomaka (eng. *offset*). Primjerice, želi li se deklarirati tri ovakva polja

```
short Polje0[128];
float Polje1[64];
int   Polje2[256];
```

u dinamički alociranoj dijeljenoj memoriji potrebno je to učiniti na ovakav način:

```
extern __shared__ char Polje[];
__device__ void funkcija()
{
    short* Polje0 = (short*)Polje;
    float* Polje1 = (float*)&Polje0[128];
    int*   Polje2 =   (int*)&Polje1[64];
}
```

Naravno, prilikom poziva navedene kernel funkcije potrebno je u konfiguraciji izvršavanja rezervirati potrebnu količinu dijeljene memorije za svaki blok.

4.4. Ugrađene varijable

Unutar svake kernel funkcije dostupne su ugrađene varijable kojima se može identificirati položaj dretve u bloku i bloka u gridu.

`gridDim, blockDIm`
Varijable tipa dim3 koje sadrže iznose svih tri dimenzija grida i bloka. Svakoj dimenziji pristupa se pomoću 'x', 'y' ili 'z' polja u strukturi dim3.

`blockIdx, threadIdx`
Varijable tipa uint3 koje sadrže indekse bloka u gridu i dretve u bloku. Svakoj dimenziji pristupa se pomoću 'x', 'y' ili 'z' polja u strukturi uint3.

`warpSize`
Varijabla tipa int koja sadrži broj dretvi u warpu.

Nijednoj od ugrađenih varijabli nije moguće uzeti adresu niti upisivati nešto u njih.

4.5. Ugrađeni vektorski tipovi

I CPU-u i GPU-u su u svako vrijeme dostupni ugrađeni vektorski tipovi koji se izvode iz osnovnih cjelobrojnih i float tipova. Ustrojeni su kao strukture i svakoj od komponenti se može pristupiti pomoću polja "x", "y", "z" i "w".

`char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4, double2`

Navedene tipove stvara se uz pomoć funkcije **`make_<ime_tipa>()`**, primjerice ovako:

```
int2 make_int2(int x, int y);
```

Uz njih, moguće je koristiti i tip **`dim3`** kojim se specificiraju dimenzije bloka i grida. To se radi funkcijama **`dimBlok()`** i **`dimGrid()`**. U slučaju da neka od tri kompnente nije inicijalizirana pretpostavljena je vrijednost 1. U donjem primjeru blok se definira kao dvodimenzionalno polje a grid kao jednodimenzionalno:

```
dim3 dimBlock(DIMENZIJA_BLOKA, DIMENZIJA_BLOKA);
dim3 dimGrid(BROJ_BLOKOVA);
```

4.6. Uporaba streamova

Već je spomenuto da su pozivi kernel funkcija asinkroni - glavni program se nastavlja izvršavati na CPU-u odmah nakon što je poziv kernel funkcije završen. Ponekad postoji potreba da se istovremeno pozove više kernel funkcija zajedno s dijelom koda koji definira kopiranje podataka u memoriju GPU-u ili natrag iz GPU-a na CPU. Kako bi se olakšala sinkronizacija dretvi koje pripadaju istom kernelu a ujedno i iskoristila mogućnost GPU-a da istovremeno vrši prijenos podataka i izvršava dretve na svojim procesorma uveden je koncept streama.

U donjem primjeru stvorena su dva strema od kojih svaki izvršava prijenos podataka iz memorije CPU-a na GPU, poziva kernel funkciju koja radi nešto s tim podatcima i na kraju obavlja prijenos podataka natrag na CPU.

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

Stvaranje dva streama ostavlja mogućnost da se memorijski transfer jednog streama poklopi s izvršavanjem kernela na drugom streamu. Ipak, da bi se to dogodilo potrebna su dva preduvjeta - za memorijski transfer treba koristiti funkciju **`cudaMemcpyAsync()`** koja izvršava asinkroni prijenos podataka i memorija na CPU-u mora biti alocirana kao page-lock memorija:

```
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
```

`cudaThreadSynchronize()` je pozvana na kraju kako bi se osigurali da su svi streamovi završili prije nego se nastavi dalje. Navedena funkcija može se koristiti i nakon jedinstvenog kernel poziva. **`cudaStreamSynchronize()`** može se koristiti za sinkroniziranje CPU-a s nekim specifičnim streamom. Streamovi se uništavaju pomoću **`cudaStreamDestroy()`**.

4.7. Matematičke funkcije

CUDA model podržava veliki broj standardnih matematičkih funkcija koje se mogu izvoditi na CPU-u ili GPU-u. Želi li se koristiti standardna preciznost koriste se standardna imena funkcija s dodatkom slova "f". Dupla preciznost se na procesorima s revizijom 1.3 postiže korištenjem imena funkcija bez dodatnog slova "f".

Kada se broj jednostruke preciznosti zaokružuje na cjeli broj treba koristiti funkciju **`rintf()`** a ne **`roundf()`** - prva se prevodi u samo jednu instrukciju GPU-a dok se druga prevodi u čak 8 instrukcija. Slično je i kod zaokruživanja brojeva dvostruke preciznosti gdje se opet preporuča korištenje **`rint()`** umjesto **`round()`**. Ostale funkcije za zaokruživanje - **`truncf(), ceilf(), floorf()`**, kao i njihove verzije za dvostruku preciznost - **`trunc(), ceil(), floor(),`** prevode se u jednu instrukciju.

Ako preciznost nije od velikog značenja u kernel funkcijama se mogu koristiti i brže matematičke funkcije smanjene preciznosti. Dobivaju se dodavanjem prefiksa "__" ispred imena - primjerice **`__sinf(x)`**. Želi li se jednostavno i brzo provjeriti utjecaj smanjene preciznosti na točnost proračuna prilikom prevođenja može se koristiti opcija **`-use_fast_math`** koja će sve matematičke funkcije prevesti u svoju manje preciznu verziju ako postoji.

Potpuni popis svih podržanih matematičkih funkcija može se naći u "NVIDIA CUDA Programming Guide" priručniku.

4.8. Atomske funkcije

Atomske funkcije izvode operaciju čitanja, modificiranja i pisanja iz globalne ili dijeljene memorije na način kojim se osigurava da nijedna druga dretva neće pisati ili čitati iz iste lokacije prije nego što je operacija završena. Rade isključivo sa cjelobrojnim varijablama s iznimkom **`atomicExch()`** koja podržava i realne vrijednosti jednostruke preciznosti. Atomske funkcije uvedene su od verzije 1.1 i u početku su podržavale samo atomske operacije s 32 bitnim riječima u globalnoj memoriji. Od verzije 1.2 podržavaju i atomske operacije s 32 bitnim riječima u dijeljenoj memoriji i 64 bitnim riječima u globalnoj memoriji. U nastavku su navedene sve atomske operacije:

`atomicAdd(), atomicSub(), atomicExch(), atomicMin(), atomicMax(), atomicInc(), atomicDec(), atomicCAS(), atomicAnd(), atomicOr(), atomicXor()`

Primjerice, atomska operacija **`atomicAdd()`** može se koristiti ovako:

```
int atomicAdd(int* adresa, int vrijednost);
```

Pri čemu se iz memorije učitava podatak s tražene adrese, dodaje mu se neka cjelobrojna vrijednost i rezultat se ponovno sprema na tu istu adresu.

4.9. Warp Vote funkcije i sinkronizacija

Pomoću warp vote funkcija moguće je evaluirati neki uvjet za sve dretve u bloku. Podržani su od revizije 1.2. One se, primjerice, mogu iskoristiti za provjeru dibergencije tj. hoće li se neka uvjetna naredba različito evaluirati za neke dretve iz bloka i tako zbog serijaliziracije usporiti izvođenje svih dretvi u bloku.

Funkcija **`__all()`** vraća ne-nul vrijednost ako je navedeni uvjet istinit za sve dretve u bloku:

```
int __all(int uvjet);
```

S druge strane, funkcija **`__any()`** vraća ne-nul vrijednost ako je navedeni uvjet istinit za barem jednu dretvu u bloku:

```
int __any(int uvjet);
```

Sinkronizacija dretvi u jednom bloku može se provesti funkcijom **`__synchthreads()`** koja funkcionira kao neka vrsta barijere - svaka dretva mora pričekati dok sve ostale dretve unutar bloka nisu došle do navedene naredbe. Jedino nakon nje se garantira da će sva čitanja i pisanja u dijeljenu memoriju biti vidljiva svim dretvama u bloku pa ju treba obavezno pozvati prije čitanja memorijska lokacija koju je prethodno modificirala neka druga dretva.

4.10. Prevođenje s **nvcc** prevoditeljem

nvcc prevoditelj olakšava prevođenje CUDA programa jer automatski poziva sve alate koji su potrebni u raznim fazama prevođenja. GPU i CPU kod se zasebno prevode - GPU kod u asemblerski ptx kod ili u izvršni cubin objekt, a CPU u c kod koji se kasnije može zasebno prevesti ili odmah u izvršnu verziju pozivanjem gcc ili nekog drugog prevoditelja. Svi CUDA programi završavaju s ekstenzijom **`.cu`**.

Želi li se prevedeni program odmah i pokrenuti može se koristiti opcija **`-run`**:

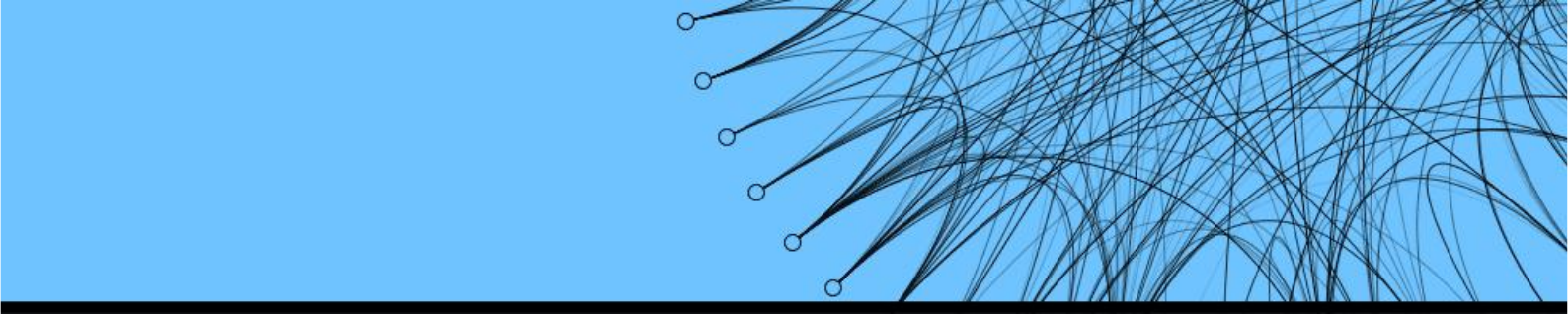
```
>> nvcc -run ime_programa.cu
```

Čak i u nedostatku NVIDIA-inog grafičkog procesora programi se mogu pokretati u emulacijskom modu pomoću opcije **`-deviceemu`** pri čemu se prevedeni program izvodi isključivo na CPU-u:

```
>> nvcc -deviceemu ime_programa.cu
```

Korištenje dvostruke preciznosti mora se posebno naglasiti prilikom prevođenja ili će prevoditelj automatski zaokružiti sve varijable u dvostrukoj preciznosti na jednostruku. To se radi definiranjem revizije kojoj grafički procesor pripada (dvostruka preciznost je podržana tek od revizije 1.3):

```
>> nvcc -arch sm_13 ime_programa.cu
```



Poglavlje 5.
Optimizacija

Dizajn efikasnih paralelnih programa jedna je stvar, a prilagodba odabranom programskom modelu nešto posve drugo. I kod razvoja aplikacija za CUDA-u stalno na umu treba imati specifičnosti (da baš ne kažemo ograničenja) grafičkih procesora za koje se kod prevodi. nvcc prevoditelj uglavnom će uspješno provesti optimiranje niske razine no većina važnijih odluka - odabir broja dretvi i količinu dijeljene memorije po bloku primjerice, ostavljena su na odgovornost programeru. Optimizacija performansi ugrubo se provodi pomoću tri osnovne strategije:

1. Maksimalno iskorištavanje paralelnosti u algoritmu
2. Optimizacija korištenja memorije
3. Optimizacija korištenja instrukcija

Iskorištavanje paralelnosti u algoritmu počinje s iskorištavanjem podatkovnog paralelizma (eng. *data paralelism*). To znači da je podatke na kojima se vrši obradba potrebno podijeliti na što veći broj dretvi. Kada neka od dretvi zatreba podatke koje je pripremila neka druga dretva potrebna je sinkronizacija koja se svodi na sljedeća dva slučaja:

1. **Dretve pripadaju istom bloku:**
   U tom slučaju potrebno je sinkronizirati dretve unutar bloka pomoću naredbe `__syncthreads()` nakon koje je garantirano da su sve promjene u dijeljenoj memoriji osvježene.

2. **Dretve pripadaju različitim blokovima:**
   U tom slučaju za razmjenu podataka je potrebna globalna memorija i sinkronizacija nije moguća unutar jednog poziva kernel funkcije već su potrebna dva - jedan za upisivanje i drugi za čitanje podataka. Pošto su pozivi kernel funkcija asinkroni potrebno je između dva poziva izvršiti funkciju `cudaThreadSynchronize()`.

Optimizacija korištenja memorije prije svega znači da je potrebno minimizirati transfer podataka s CPU-a na GPU, kao i korištenje globalne memorije GPU-a općenito u korist djeljene memorije na multiprocesoru. Ponekad je najbolja opcija izbjeći učitavanje iz memorije u potpunosti i jednostavno ponoviti proračun iznova kad god je potreban. U pravilu, ako je količina memorijskih operacija koje se provode relativno velika u odnosu na aritmetičke operacije dobra je ideja izbjeći računanje na GPU-u i umjesto toga cijeli proračun izvesti na CPU-u.

Optimizacija korištenja instrukcija provodi se minimiziranjem uporabe instrukcija koje zahtjevaju puno ciklusa sata da se izvrše, pažljiva uporaba uvjetnog grananja (if, while, switch, do, for), korištenje manje preciznih ali bržih funkcija u slučaju kada preciznost nije važna - funkcije jednostruke preciznosti i intrinzičnih umjesto regularnih funkcija.

5.1. Parametri dretvi i blokova

Svaki multiprocesor ima mogućnost pokretanja više blokova u isto vrijeme. Blokovi koji se trenutno izvršavaju na određenom multiprocesoru se nazivaju aktivni blokovi. Njihov broj ovisi o količini dijeljene memorije rezervirane za svaki blok i o broju registara po dretvi u bloku. Ako količina dostupne dijeljene memorije ili registara nije dovoljna za pokretanje barem jednog bloka dretvi poziv kernel funkcije neće uspijeti.

Vrlo korisni alat koji pomaže prilikom konfiguracije izvršavanja kernel funkcija je "CUDA GPU Occupancy Calculator" koji dolazi u vidu Excel tablice. Pomoću njega se može provjeriti iskorištenost svakog multiprocesora za odabrani broj dretvi po bloku, količini dijeljene memorije i broju registara po bloku. Od navedena tri parametra jedino se broj registara po bloku ne može eksplicitno definirati već se definira automatski u procesu prevođenja. Srećom, nvcc prevoditelj ima mogućnost ispisivanja informacija o broju registara i količini lokalne, dijeljene i konstantne memorije za svaku kernel funkciju. To se radi dodavanjem opcije **`-ptxas-options=-v`** prilikom prevođenja.

Dobiveni podatci potom se mogu unijeti u kalkulator kako bi se dobila procjena iskorištenosti svakog multiprocesora. U nastavku su navedene neke općenite smjernice prilikom definiranja konfiguracije izvršavanja:

1. Preporuča se da blokova bude barem koliko i multiprocesora na GPU-u. U protivnom multiprocesori kojima nije dodijeljen ni jedan blok ostaju nezaposleni.

2. Dapače, pošto prilikom izvršavanja pojedinog bloka uvijek postoje čekanja (zbog čekanja na pristup memoriji primjerice) uputno je da broj blokova bude barem dvostruko veći od broja multiprocesora. Tako će svaki multiprocesor moći naizmjence izvršavati blokove ovisno o tome koji od njih je trenutno na čekanju. To u praksi znači da je potrebno minimalno oko 100 blokova ako se želi da se aplikacija izvršava optimalno i na budućim generacijama procesora (koji će imati puno veći broj multiprocesora). Maksimalni broj aktivnih blokova po multiprocesoru je 8.

3. Broj dretvi po bloku trebao bi biti što veći i po mogućnosti višekratnik broja 64. Taj broj je praktičan jer warpovi u koje SIMT jedinica dijeli dretve na multiprocesoru broje po 32 dretve. Također, optimizacija pristupa registarskoj memoriji koju provodi prevoditelj najbolje funkcionira kad je broj dretvi višekratnik broja 64 i kad ih je više od 192. Zbog toga se najčešće i uzima 192 ili 256 dretvi po bloku. Maksimalni broj dretvi u bloku je 512. Osim toga, broj dretvi u bloku je efektivno ograničen samo brojem dostupnih registara po multiprocesoru pa treba pripaziti da je prilikom izvršavanja dostupno dovoljno registara za sve dretve.

Programerima se savjetuje da isprobaju više različitih konfiguracija izvršavanja (eng. execution configuration) prilikom pokretanja kernel funkcija kako bi se uvjerili da je njihov odabir optimalan.

5.2. Performanse instrukcija

Podsjetimo se ukratko što se točno događa prilikom izvršavanja svake instrukcije. Istovjetne instrukcije se u warpu izvršavaju paralelno na osam skalarnih procesora pri čemu je za svaku instrukciju potrebno:

1. učitati operande za svaku dretvu u warpu
2. izvršiti instrukciju
3. zapisati rezultat za svaku dretvu u warpu

Dakle, na vrijeme potrebno da se izvršavi svaka instrukcija uvelike utječu i faktori koji se ne tiču kompleksnosti same instrukcije - prije svega u u kojoj vrsti memorije se nalaze operandi i gdje je potrebno spremiti rezultat. Zbog toga je za maksimalne performanse potrebno:

1. koristiti što manje sporijih instrukcija (one koje zahtijevaju više od 4 ciklusa sata da se izvrše)
2. pobrinuti se da se operandi za instrukcije učitavaju i spremaju u što bržu memoriju (općenito pravilo je - prvo registri i dijeljena memorija a tek onda globalna memorija)
3. omogućiti sustavu da što više preklopi memorijske operacije s aritmetičkima i tako smanji utjecaj čekanja. To se postiže većim omjerom aritmetičkih naspram memorijskih operacija u programu i povećanjem broja dretvi u bloku kako je opisano u poglavlju 5.1.

5.2.1. Aritmetičke instrukcije

Kako bi izvršio jednu instrukciju po warpu svakom multiprocesoru treba:

4 ciklusa sata
Za zbrajanje, množenje i množenje-zbrajanje u jednostrukoj preciznosti.
Za cjelobrojno zbrajanje.
Za operacije nad bitovima, usporedbe i konverziju tipova.

16 ciklusa sata
Za recipročnu vrijednost, recipročno korjenovanje, **`__logf()`**

Dijeljenje i modulo operacije u cjelobrojnoj aritmetici su pogotovo spore i preporuča se njihovo izbjegavanje ili korištenje odgovarajućih operacija nad bitovima. Primjerice, ako je n potencija od 2 onda je **`(x/n)`** ekvivalentno s **`(x>>log2(n))`** a **`(x%n)`** je ekvivalentno s **`(x&(n-1))`**. Prevoditelj će u tim slučajevima automatsku provesti potrebnu konverziju samo ako je n literal.

Ostale instrukcije zahtijevaju puno više ciklusa sata za izvršavanje pošto su implementirane kao kombinacija gore navedenih instrukcija.

Dijeljenje u jednostrukoj preciznosti zahtijeva 36 ciklusa sata ali može se koristiti i **`__fdividef(x,y)`** koja obavlja posao u samo 20 ciklusa.

`__sinf(x), __cosf(x), expf(x)` traju 32 ciklusa sata. S druge strane, **`sinf(x), cosf(x), tanf(x), sincosf(x)`** su puno sporije, pogotovo u slučaju kad je apsolutna vrijednost x-a veća od 48039. Uz to, kako bi se izračunala

takva vrijednost potrebno je korištenje lokalne memorije što dodatno usporava stvar.

Ovome još treba ubrojati i slučajeve kad prevoditelj vrši automatsku pretvorbu tipova - primjerice, kada se koriste char i short operandi koji se uglavnom moraju prevesti u int tip. Drugi primjer je kada se u funkcijama za jednostruku preciznost koriste operandi s dvostrukom preciznošću bez eksplicitne pretvorbe pomoću "f" sufiksa (primjerice ovako - **`3.14159f`**).

5.2.2. Instrukcije kontrole toka

Bilo koja instrukcija kontrole toka (if, switch, do, for, while) može znatno utjecati na brzinu izvođena ako dođe do divergencije dretvi tj. ako dretve u istom warpu počnu izvršavati drugačiji niz instrukcija. U tom slučaju gubi se mogućnost paralelnog izvođenja instrukcija jer se svaka dretva mora izvršiti u seriji dok ponovno sve dretve u warpu ne dođu do istog dijela programskog koda.

Gore navedene naredbe kontrole toka trebale bi se koristiti tako da se maksimalno smanji mogućnost divergentnih dretvi u warpu. To ne bi trebalo biti teško jer je raspoređivanje dreti jednog bloka u warpove dobro poznato - dretve se grupiraju u warpove slijedno po indeksima tako da dretve s indeksom od 0 do 31 pripadnu u prvi warp, one s indeksom 32 do 63 u drugi i tako dalje. Zato je naredbe kontrole toka uputno pisati tako da ovise samo o nekom uvjetu koji je isti za sve dretve - primjerice, uvjet (**`threadIdx / WSIZE`**) gdje je **`WSIZE`** veličina warpa jednako se evaluira za sve dretve u istom warpu.

Ponekad će prevoditelj optimizirati petlju koristeći direktivu #pragma unroll o kojoj će više riječi biti u nekom od sljedećih poglavlja. Na taj način nijedna dretva ne može divergirati.

5.2.3. Memorijske instrukcije

Memorijske instrukcije uključuju instrukcije koje čitaju ili pišu u dijeljenju, lokalnu ili globalnu memoriju. Iako je za izvršavanje svake memorijske instrukcije potrebno samo 4 ciklusa sata, u slučaju lokalne ili globalne memorije potrebno je još dodatnih 400 do 600 ciklusa sata čekanja kako bi se obavio pristup globalnom adresnom prostoru.

Primjerice, operacija pridruživanja u sljedećem programskom odsječku:

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

uzima 4 ciklusa sata kako bi pokrenula čitanje iz globalne memorije, 4 ciklusa sata kako bi pokrenula pisanje u dijeljenu memoriju i dodatnih 400 do 600 ciklusa sata čekanja na obavljanje tražene operacije čitanja iz globalne memorije. Većina takvih zastoja može se uspješno sakriti ako postoji dovoljno aritmetičkih instrukcija koje će se izvršavati za vrijeme čekanja na memorijsku operaciju.

5.3. Tekstualno profiliranje

CUDA SDK uključuje vizualnu i tekstualnu verziju profilera pomoću kojih se mogu provjeriti okvirne performanse CUDA programa i identificirati kritični dijelovi koji dovode do usporavanja. Naglasak je na "okvirne" jer profiler mjeri performanse samo jednog multiprocesora i to tako da ključne događaje bilježi po warpovima. Primjerice, ako se pojavi jedno nekonzistentno čitanje u warpu to će se zabilježiti kao jedan događaj tako da će krajnji rezultat brojati sva nekonzistentna čitanja u svim warpovima na jednom multiprocesoru. Zato je bitno pokrenuti što više blokova (NVIDIA-ina preporuka je barem 100) kako bi se osiguralo da svaki procesor dobije podjednako opterećenje i kako bi vrijednosti koje izbaci profiler bile referentne. Profiliranje se izvodi na razini sklopovlja.

Tekstualno profiliranje se namješta pomoću sistemskih varijabli:

`CUDA_PROFILE:` Postavljanje na 0 ili 1 uključuje ili isključuje profiler.
`CUDA_PROFILE_LOG:` Ime datoteke u koju se sprema izvještaj (pretpostavljena vrijednost je **`./cuda_profile.log`**).
`CUDA_PROFILE_CONFIG:` Ime konfiguracijske datoteke u koju se navodi do četiri signala koja se žele mjeriti.

Zadnja varijabla je i najvažnija jer se njome definiraju signali koji se žele mjeriti. Istovremeno je moguće mjeriti samo četiri signala. Njihova imena navode se u odvojenim linijama u konfiguracijskoj datoteci.

`gld_incoherent:` Broj neprekloplјenih čitanja iz globalne memorije.
`gld_coherent:` Broj prekloplјenih čitanja iz globalne memorije.
`gsd_incoherent:` Broj neprekloplјenih pisanja u globalnu memoriju.
`gsd_coherent:` Broj prekloplјenih pisanja u globalnu memoriju.
`local_load:` Broj učitavanja iz lokalne memorije.
`local_store:` Broj pisanja u lokalnu memoriju.
`branch:` Broj grananja u dretvama.
`divergent_branch:` Broj divergentnih grananja unutar warpa.
`instructions:` Broj izvršenih instrukcija.
`warp_serialize:` Broj dretvi u warpu koje su se morale serijalizirati zbog adresnih konflikata u dijeljenoj ili konstantnoj memoriji.
`cta_launched:` Broj izvršenih blokova.

Primjerice, želimo pokrenuti tekstualno profiliranje:

```
>> export CUDA_PROFILE = 1
>> export CUDA_PROFILE_CONFIG = $HOME/.cuda_profile_config
```

U konfiguracijskoj datoteci odaberemo praćenje sljedeća dva signala - **`gld_coherent`**, **`gld_incoherent`**. Nakon pokretanja programa u izvještaju **`./cuda_profilr.log`** spremljeni su sljedeći podatci:

```
method,gputime,cputime,occupancy,gld_incoherent,gld_coherent,gst_incoherent,
gst,coherent
method=[ memcopy ] gputime=[ 438.432 ]
method=[ _Z17reverseArrayBlockPiS_ ] gputime=[ 267.520 ] cputime=[ 297.000 ]
occupancy=[ 1.00 ], gld_incoherent=[ 0 ], gld_coherent=[ 1952 ]
method=[ memcopy ] gputime=[ 349.344 ]
```

Od posebnog interesa je stavka **`cputime`** koja odražava vrijeme potrebno za izvršavanje određene kernel funkcije gledano iz pozicije CPU-a.

5.4. Performanse memorije

Performanse prilikom korištenja memorije uvelike ovise o vrsti memorije kojoj se pristupa. Ipak, za maksimalne performanse potrebno je voditi i računa o načinu na koji se pristupa memoriji kako ne bi došlo došlo do nekinzistentnih preklapanja i konflikata (eng. *bank conflicts*). Na novijim grafičkim procesorima - konkretno, GT200 ili nekoj novijoj seriji, većina tih stvari je automatizirana i programer ne treba o njim brinuti.

Tipični način za efikasno pristupanje memoriji je sljedeći:

1. Učitavanje podataka iz globalne memorije GPU-a u dijeljenu memoriju.
2. Sinkronizacija sa svim ostalim dretvama u bloku kako bi svaka dretva mogla sigurno čitati iz dijeljene memorijske lkacije u koju je upisivala neka druga dretva.
3. Procesuirajne podataka iz dijeljene memorije.
4. Ponovna sinkronizacija kako bi se osiguralo da su sve promjene vidljive svim ostalim dretvama.
5. Upisivanje rezultata ponovno u globalnu memoriju GPU-a.

5.4.1. Globalna memorija

Globalna memorija nema odgovarajuću cache memoriju pa u slučaju višestrukog pristupa istoj memorijskoj lokaciji svi pristupi traju jednako dugo. Zbog toga je važno pripaziti na koji način se vrše pristupi globalnoj memoriji.

Prvo, treba se voditi računa da se željena memorijska operacija prevede u jednu memorijsku instrukciju što će se dogoditi samo ako se učitavaju riječi duljine 32, 64 ili 128 bita. Ovo svojstvo se zove poravnatost (eng. alignment). Poravnatost je sigurno zadovoljena za ugrađene tipove (float2 i float4 primjerice). Koristi li se ručno napisana struktura s jednostavnim podatkovnim tipovima potrebno je osigurati se da se učitavanje provodi u što manjem broju instrukcija.

64-bitna instrukcija za učitavanje može se zatražiti pomoću **`__align__(8)`** :

```
struct __align__(8) {
    float a;
    float b;
};
```

U protivnom bi prevoditelj učitavanje pokušao postići s dvije 32-bitne instrukcije što bi bilo puno sporije. Isto tako se može zatražiti i 128-bitna instrukcija za učitavanje pomoću **`__align__(16)`**:

```
struct __align__(16) {
    float a;
    float b;
    float c;
};
```

Strukture veće od 128 bita uvijek je poželjno deklarirati s **`__align__(16)`** jer se tako osigurava da će se prilikom učitavanja koristiti najmanji mogući broj instrukcija. Svaka adresa varijable koja je smještena u globalnoj memoriji ili je vraćena alokacijom memorije je poravnata na barem 256 bita.

Drugo, propusnost globalne memorije iskorištava se najefikasnije ako se simultani pristupi memoriji dretvi u polu-warpu (prilikom izvršavanja jedne instrukcije čitanja ili pisanja iz globalne memorije) mogu spojiti (eng. *coalesce*) u jednu memorijsku transakciju od 32, 64 ili 128 bajta.

Da bi se dogodilo spajanje memorijskih transakcija za sve dretve u polu-warpu potrebno je zadovoljiti sljedeće uvjete (poravnatost varijabli se podrazumijeva):

Za revizije 1.0 i 1.1

1. Sve dretve moraju pristupati 32-bitnim riječima (spajanje u jednu 64-bajtnu transakciju) ili 64-bitnim rječima (spajanje u jednu 128-bajtnu transakciju) ili 128-bitnim rječima (spajanje u dvije 128-bajtne transakcije).
2. Svih 16 riječi kojima pristupaju dretve u polu-warpu moraju ležati u istom segmentu koji je veličine transakcije (ili dvostruko veći od veličine transakcije u slučaju 128-bitnih riječi).
3. Dretve moraju pristupati riječima slijedmo - k-ta dretva u polu-warpu mora pristupiti k-toj riječi u transakciji.

Za reviziju 1.2 i više

1. Sve dretve moraju pristupati 8-bitnim riječima (spajanje u jednu 32-bajtnu transakciju) ili 16-bitnim riječima (spajanje u jednu 64-bajtnu transakciju) ili 32-bitnim ili 64-bitnim riječima (spajanje u jednu 128-bajtnu transakciju). Redoslijed pristupa memoriji nije bitan i spajanje će se obaviti čak i ako više dretvi pristupa istoj memorijskoj lokaciji.

Spojene 32-bajtne transakcije pružaju najveću memorijsku propusnost u usporedbi s 64-bajtnim i 128-bajtnim transakcijama. Pristupi memoriji koji nisu spojeni uzrokuju i do za red veličine manju memorijsku propusnost u usporedbi sa spojenim pristupima. Zbog toga je bitno uvijek voditi računa o načinu na koji se vrši pristup globalnoj memoriji.

Neki uobičajeniji načini pristupa globalnoj memoriji su opisani u nastavku. Pristup u kojem svaka dretva s indeksom **`ID`** pristupa jednom elementu polja s početnom adresom **`pocetnaAdresa`** obično se obavlja ovako:

```
pocetnaAdresa + ID
```

Da bi se postiglo spajanje memorijskih transakcija potrebno je da je tip varijable koja se učitava zadovoljava uvjet poravnatosti i da nije veći od 128 bita (16 bajtova). Premašivanje veličine obično se događa ako je riječ o strukturi većoj od 16 bajtova. U tom slučaju pametno je rastaviti tu strukturu u više struktura koje zadovoljavaju gore navedene uvjete. Sada je samo potrebno umjesto jednom polju struktura pristupiti više njih kako bi se prikupili svi podatci.

Drugi često korišteni pristup je kada svaka dretva s indeksima **`(IDx,IDy)`** pristupa jednom elementu dvodimenzionalnog polja s početnom adresom **`pocetnaAdresa`** širine **`sirina`**:

```
pocetnaAdresa + sirina * IDy + IDx
```

U tom slučaju spajanje se događa ako je širina bloka dretvi višekratnik veličine polu-warpa (16 u trenutnim revizijama) i ako je sirina višekratnik od 16. To znači da će se polju čija širina nije višekratnik od 16 pristupati puno brže ako se njegova veličina proširi do najbližeg višekratnika od 16 i ostatak popuni s nekim vrijednostima. U tu svrhu služe funkcije **`cudaMallocPitc()`** i **`cuMemAllocPitch()`** koje omogućuju programerima da stvaraju i manipuliraju poljima koja se podvrgavaju tim uvjetima.

5.4.2. Lokalna memorija

Lokalna memorija je zapravo dio globalne memorije kojem pravo pristupa ima samo odgovarajuća dretva. Ona služi za automatsko spremanje svih varijabli koje bi inače trebale prebivati u registrima multiprocesora. Zbog toga što je riječ o globalnom memorijskom prostoru lokalna memorija je višestruko sporija od dijeljene memorije i registara koji se nalaze na multiprocesorima.

Programer nema direktnog utjecaja na to hoće li se varijabla spremiti u lokalnu memoriju - tu odluku donosi prevoditelj u trenutku prevođenja. Prevođenjem s opcijama **`-ptx`** ili **`-keep`** stvara se ptx asemblerski kod kojim je moguće provjeriti je li varijabla smještena u lokalnu memoriju. U tom slučaju ona je deklarirana s .local mnemonikom i pristupa joj se pomoću ld.local i st.local. Ako varijabla nije smještena u lokalnu memoriju to ne znači da ju kasnije faze prevođenja ipak neće smjestiti tamo. Ne postoji direktan način da se to provjeri za svaku pojedinu varijablu, no moguće je provjeriti ukupno zauzeće lokalne memorije za svaki poziv kernel funkcije (**`lmem`**) specificiranjem opcija **`--ptax-options=-v`** prilikom prevođenja.

Lokalna memorija praktična je iz razloga što programer ne mora razmišljati ima li na multiprocesoru dovoljno registarskog prostora za sve varijable u dretvama istog bloka. Ipak, ponekad je zbog optimizacije potrebno osigurati da varijabla neće biti spremljena u lokalnu memoriju. Nekoliko savjeta je:

1. Velike strukture ili polja za koja prevoditelj procijeni da bi zauzela previše memorijskog prostora obično se spremaju u lokalnu memoriju.
2. Polja koja su indeksirana konstantnim vrijednostima poznatim u trenutku prevođenja obično se spremaju u registre (ako nisu prevelika). S druge strane, polja indeksirana varijablama ne mogu nikada biti spremljena u registre. To se može rješiti odmotavanjem petlji (eng. *loop unrolling*) kojim se petlja rastavlja na niz naredbi pri čemu se varijable zamijenjuju konstantnim vrijednostima. Odmotavanje petlji se izvodi pomoću **`#pragma unroll`** direktive koja se stavlja neposredno prije petlje i koja se odnosi samo na tu petlju. Opcionalno se može definirati i broj kojim se definira koliko iteracija će petlja odmotati.

```
#pragma unroll 5
for ( int i = 0; i < n; i++ )
```

U ovom slučaju mora se pripaziti da n nije manji od 5 jer će to utjecati na točnost programa. Treba imati na umu da odmotavanje petlji može višestruko povećati broj korištenih registara što može rezultirati time da se varijable na kraju ipak spreme u lokalnu memoriju, čime se cijela poanta odmotavanja gubi. Moguće je koristiti opciju -maxregcount=vrijednost koja govori prevoditelju da koristi više registara (128 maksimalno). To će ograničiti broj dretvi u bloku koje se mogu istovremeno pokrenuti što može ograničiti mogućnost da se sakrije memorijska latencija izvođenjem dretvi koje ne čekaju pristup memoriji.

5.4.3. Dijeljena memorija

Zbog toga što se nalazi na samom multiprocesoru dijeljena memorija je puno brža od lokalne i globalne memorije. Zapravo, za dretve iz istog warpa pristup dijeljenoj memoriji je brz kao i pristup registrima pod uvjetom da nema bank konflikata (eng. *bank conflicts*) između dretvi.

Kako bi se postigla visoka memorijska propusnost dijeljena memorija je podijeljena u memorijske module jednake veličine koji se zovu banke (eng. *banks*) i kojima se može pristupati simultano. Primjerice, ako se istovremeno zahtjeva n pristupa dijeljenoj memoriji i svih n pristupa odgovaraju adresama koje spadaju u različite banke onda ih je moguće poslužiti simultano. S druge strane, ako se barem dvije adrese nalaze u istoj banki pristupi se moraju serijalizirati što znatno usporava cijeli memorijski transfer.

U slučaju dijeljene memorije banke su organizirane tako da slusjedne 32-bitne riječi pripadaju istoj banki i memorijska propusnost je 32 bita u dva ciklusa sata. U trenutnoj glavnoj reviziji CUDA modela (1.x) broj dretvi u warpu je 32 i veličina banke je 16 tako da se zahtjev za pristupom dijeljenoj memoriji kod dretvi iz istog bloka dijeli na dva zahtjeva - svaki za jednu polovicu warpa. Zbog toga ni ne može biti bank konflikata između dretvi koje pripadaju različitim polovicama warpa.

Česti je slučaj kada svaka dretva pristupa 32-bitnoj riječi iz polja indeksiranog s identifikatorom dretve ID i određenim pomakom s:

```
__shared__ float dijeljeno_polje[32];
float data = dijeljeno_polje[pocetniIndeks + s * ID];
```

Najjednostavniji slučaj je kad je pomak s jednak 1 - onda svaka dretva u poluwarpu pristupa svojoj banki i dujeljenoj memoriji koja odgovara njenom indeksu unutar poluwarpa. Veći pomak znači da će se svaka dretva pristupati svakoj s-toj banki pri čemu se indeksi banki cirkularno premotavaju. Tako će, primjerice, kod pomaka 3 dretva s indeksom 6 pristupati banki 1 umjesto banki 18. Pošto ima 16 banki jedini uvjet da ne dođe do konflikta je da je pomak neparan.

Komliciraniji slučaj je kad se pristupa elementu koji je manji ili veći od 32 bita. Primjerice, do konflikta dolazi ako se pristupa polju znakova na sljedeći način:

```
__shared__ char dijeljeno_polje[32];
char data = dijeljeno_polje[pocetniIndeks + ID];
```

Razlog tome je što se varijable **`dijeljena[0]`**, **`dijeljena[1]`**, **`dijeljena[2]`** i **`dijeljena[3]`** nalaze u istoj banki. S druge strane, pristupa li se na sljedeći način izbjegnut će se konflikti:

```
char data = dijeljeno_polje[pocetniIndeks + 4 * ID];
```

Jednako tako postoje konflikti i kod pristupa brojevima dvostruke preciznosti:

```
__shared__ double dijeljeno_polje[32];
double data = dijeljeno_polje[pocetniIndeks + ID];
```

To je zbog toga što se pristup brojevima dvostruke preciznosti rastavlja u dvije transakcije od 32 bita. Trenutno ne postoji jednostavni i pouzdani način da se to izbjegne.

Pristup strukturi se dijeli u koliko god je memorijskih transakcija potrebno kako bi se učitali svi članovi strukture. Primjerice, u sljedećem programskom odsječku

```
__shared__ struct type dijeljeno_polje[32];
struct type data = dijeljeno_polje[pocetniIndeks + ID];
```

broj memorijskih pristupa ovisi o tipu strukture. Tako možemo imati:

1. Tri odvojene memorijske transakcije bez konflikata ako je struktura definirana kao:

   ```
   struct type {
       float x, y, z;
   }
   ```

2. Dva odvojene memorijske transakcije s konfliktima ako je struktura definirana kao:

   ```
   struct type {
       float x, y;
   }
   ```

3. Dvije odvojene memorijske transakcije s konfliktima ako je struktura definirana kao:

   ```
   struct type {
       float x;
       char c;
   }
   ```

Iz navedenih razloga bitno je pažljivo planirati pristup dijeljenoj memoriji kako bi se postigle maksimalne performanse.

5.4.4. Registri

U općem slučaju pristup registrima ne dodaje dodatne cikluse sata prilikom izvršavanja instrukcija. Ipak, usporavanja se mogu dogoditi zbog čitaj-poslije-pisanja ovisnosti (eng. *read-after-write*) i memorijskih konflikata prilikom pristupa registrima.

Čitaj-poslije-pisanja ovisnosti se mogu ignorirati čim postoji barem 192 aktivne dretve po multiprocesoru koje ih mogu sakriti.

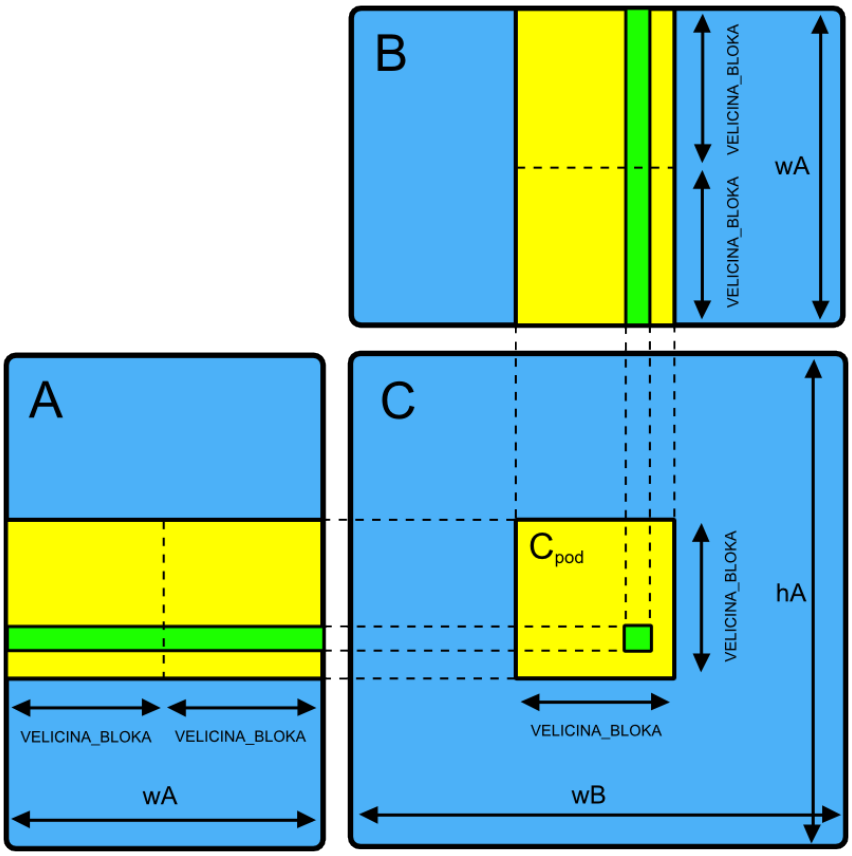
Memorijski konflikti prilkom pristupa registrima se automatski uklanjaju prilikom prevođenja i naknadno prilikom odabira dretvi za izvršavanje i programer nema direktnog utjecaja na njih. Zbog toga nije ni potrebno organizirati podatke u specijalizirane tipove podataka - **`float4`** ili **`int4`** primjerice. Ipak, najbolji rezultati u uklanjanju konflikata se postižu ako je broj dretvi u bloku višekratnik od 64.

Poglavlje 6.
Primjer - množenje matrica

Kao reprezentativni primjer opisuje se problem množenja matrica i njegova implementacija u CUDA modelu. Ovo nije najefikasnije moguće rješenje nego služi kao dobra ilustriracija brojnih specifičnosti prilikom programiranja na CUDA-i.

Problem nalaženja produkta C dviju matrica A i B s dimenzijama (wA, hA) i (wB, hB) podijeljen je na dretve na sljedeći način:

1. Svaki blok dretvi zadužen je za računanje jedne podmatrice C_{pod}
2. Svaka dretva unutar bloka zadužena je za računanje jednog elementa matrice C_{pod}. Velicina svake stranice bloka je 16 što znači da se u bloku



Slika 7. Primjer množenja matrica u CUDA programskom modelu - svaki blok izračunava jednu podmatricu C_{pod} pri čemu svaka dretva unutar bloka izračunava jedan element podmatrice.

nalazi 256 dretvi. Tako je broj dretvi u bloku višekratnik veličine warpa (32) a ujedno je i manji od maksimalnog broja dretvi u bloku (512).

C_{pod} je jednaka umnošku dvije pravokutne matrice - podmatrice od A s dimenzijama (wA, VELICINA_BLOKA) i podmatrice od B s dimenzijama (VELICINA_BLOKA, wA). Svaka od tih podmatrica dijeli se u kvadratne podmatrice veličine VELICINA_BLOKA koje su dovoljno velike da se prepišu u dijeljenu memoriju na multiprocesoru. Izračun elemenata podmatrice potom se izvršava u onoliko koraka koliko ima takvih blokova i to na slijedeći način:

1. Svaka dretva unutar bloka prepisuje jedan element kvadratnih podmatrica A i B u dijeljenu memoriju.
2. Svaka dretva potom izračunava jedan element podmatrice C_{pod} tako da pomnoži odgovarajuće elemente iz svojeg retka matrice A i svojeg stupca matrice B.
3. Djelomični rezultat se zbraja s trenutnom vrijednošću odgovarajućeg elementa podmatrice C_{pod}.
4. Nakon što su elementi podmatrice C_{pod} u potpunosti izračunati rezultat se upisuje u globalnu memoriju.

6.1. Izvorni kod

U nastavku je izlistan izvorni kod množenja matrica na CUDA-i. Napisane su dvije funkcije - **`Mul()`** koja se izvodi na CPU-u i koja obavlja inicijalizaciju memorije i transfer podataka na GPU te kernel funkcija **`Muld()`** koja izvodi množenje matrica na GPU-u.

```
// velicina jedne dimenzije bloka dretvi
#define VELICINA_BLOKA 16

// deklaracija kernel funkcije za mnozenje na GPU-u
__global__ void Muld(float*, float*, int, int, float*);

// funkcija za mnozenje na CPU-u koja poziva kernel funkciju
// izracunaj C = A * B
//   hA je visina od A
//   wA je sirina od A
//   wB sirina od B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;

    // ucitaj A i B u globalnu memoriju GPU-a
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // alociraj dio globalne memorije GPU-a za C
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);
```

```
    // izracunaj konfiguraciju izvrsavanja pod pretpostavkom
    // da su dimenzije matrica visektarnici od VELICINA_BLOKA
    dim3 dimBlock(VELICINA_BLOKA, VELICINA_BLOKA);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // pokreni izracunavanje na GPU-u
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // posto je poziv kernel funkcije asinkron sada mozemo nastaviti
    // izvrsavati dio koda koji je neovisan o GPU-u

    // ucitaj C iz globalne memorije GPU-a
    // ovo je memorijski transfer izmedju CPU-a i GPU-a pa smo sigurni
    // da je kernel funkcija zavrsila sa svojim radom (i da je C izracunat)
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // oslobodi memoriju na GPU-u
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

```
// kernel funkcija za mnozenje matrica koju poziva Mul()
// izracunaj C = A * B
//   wA sirina od A
//   wB je sirina od B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // indeks bloka dretvi
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // index dretve u bloku
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // indeks prve podmatrice od A koju blok procesuira
    int aBegin = wA * VELICINA_BLOKA * by;

    // indeks zadnje podmatrice od A koju procesuira blok
    int aEnd   = aBegin + wA - 1;

    // velicina koraka kojim se iterira kroz podmatrice od A
    int aStep  = VELICINA_BLOKA;

    // indeks prve podmatrice od B koju blok procesuira
    int bBegin = VELICINA_BLOKA * bx;

    // velicina koraka kojim se iterira kroz podmatrice od B
    int bStep  = VELICINA_BLOKA * wB;

    // element podmatrice od C koji racuna trenutna dretva
    float Csub = 0;

    // iteriraj kroz sve podmatrice od A i B koje su potrebne za racunanje
    // podmatrice od C
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // dijeljena memorija za podmatricu od A
```

```
        __shared__ float As[VELICINA_BLOKA][VELICINA_BLOKA];

        // dijeljena memorija za podmatricu od B
        __shared__ float Bs[VELICINA_BLOKA][VELICINA_BLOKA];

        // ucitaj matrice iz globalne u dijeljenu memoriju
        // svaka dretva ucitava jedan element matrice
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // sinkroniziraj dretve kako bi osigurao da su svi elementi ucitani
        __syncthreads();

        // pomnozi dvije ucitane podmatrice
        // svaka dretva izracunava jedan element umnoska
        for (int k = 0; k < VELICINA_BLOKA; ++k)
        {
            Csub += As[ty][k] * Bs[k][tx];
        }

        // sinkroniziraj dretve u bloku kako bi osigurao da je
        // izracun proslog umnoska zavrsio prije nego se krene
        // na novi par podmatrica
        __syncthreads();
    }

    // upisi izracunatu podmatricu Cpod globalnu memoriju
    // svaka dretva upisuje jedan element
    int c = wB * VELICINA_BLOKA * by + VELICINA_BLOKA * bx;
    C[c + wB * ty + tx] = Csub;
}
```

6.2. Usporedba sa CPU-om

Kako bi se dobio dojam koliko je implementacija u CUDA-i brža od klasične serijske implementacije na CPU-u napisana je funkcija **`Mul_CPU()`** koja izvodi množenje matrica na CPU-u i uspoređena su vremena izvođenja za razne veličine matrica. Iako se ne koriste najefikasnije implementacije za problem množenja matrica ipak se može steći dojam koliko je implementacija na GPU-u nadmoćnije od one na CPU-u.

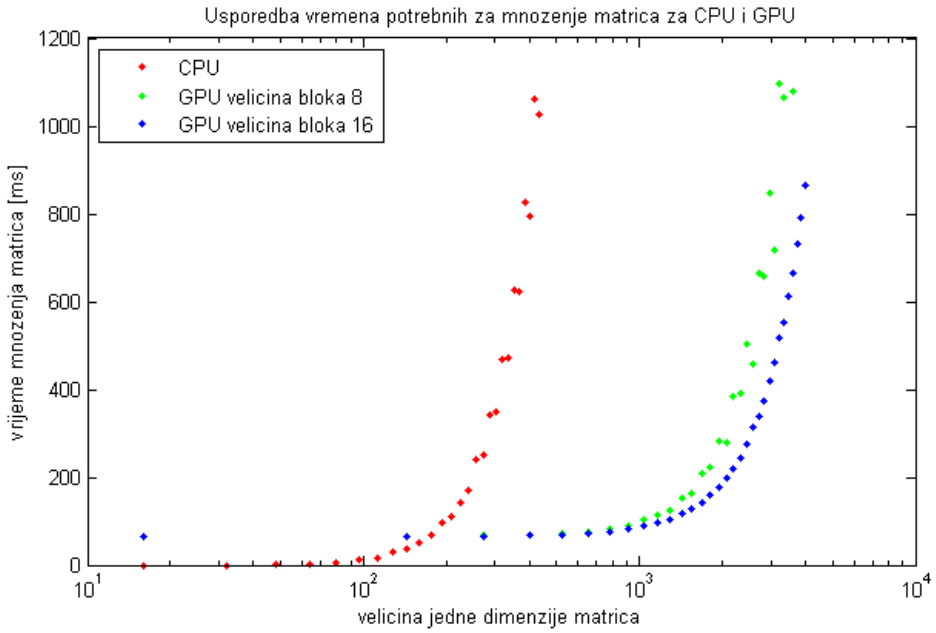
```
// mnozenje matrica na CPU-u
void Mul_CPU(const float* A, const float* B, int hA, int wA, int wB,
             float* C)
{
    int i, j, a, b;

    for ( int i = 0; i < hA; i++ ) {
        for ( int j = 0; j < wB; j++ ) {
            for ( a = 0, b = 0; a < wA; a++, b++ ) {
                C[ i * wB + j ] += A[ i * wA + a ] * B[ b * wB + j ] ;
            }
        }
    }
}
```

Mjerenja su provedena na eksperimentalnom GPU-CPU clusteru Hybrid koji se nalazi na Zavodu za fiziku Prirodoslovno-matematičkog fakulteta Sveučilišta u Splitu. Hybrid se sastoji od osam čvorova od kojih svaki raspolaže s dva CPU procesora Core 2 Quad Xeon s 8 Gb radne memorije i dva GPU procesora Tesla S1070. Pri mjerenju je korišten samo jedan čvor i jedan GPU procesor.

Množile su se isključivo kvadratne matrice dimenzija od 16 do 624 za CPU i od 16 do 3984 za GPU. Kod izvođenja na GPU-u uzete su veličine blokova dretvi od 8 i 16. Vremena su izražena u milisekundama. Rezultati su prikazani na slici 1.

Primjećuje se da je za matrice male veličine (dimenzija manjih od oko 170x170) izvođenje na CPU-u brže jer se kod GPU-a troši vrijeme na transfer podataka u globalnu memoriju. Troškovi memorijskih transfera dominiraju sve do matrica veličine od oko 1000x1000 nakon čega dolazi do zasićenja i vrijeme izvođenja se povećava eksponencijalno slično kao i kod CPU-a. Ipak, zbog velikog broja neovisnih multiprocesora (120 u slučaju Tesle S1070) izvođenje na GPU-u uz pomoć CUDA arhitekture omogućuje računanje umnoška matrica koje su za red veličine veće od standardnih.



Slika 8. Usporedba vremena potrebnih za množenje dviju matrica na CPU-u i GPU-u. U ovom slučaju CUDA omogućuje množenje matrica koje su za red veličine veće od standardnih. Također se zamjećuje da rad s većim blokovima daje bolje rezultate. Mjerenja su provedena na jednom čvoru Hybrid clustera koji raspolaže s dva CPU procesora Core 2 Quad Xeon i 8 Gb radne memorije te dva GPU procesora Tesla S1070.

Zaključak

Programiranje na grafičkim procesorima nekada je značilo programirati u jeziku grafičkog procesora na razini assemblera. Dostupnost arhitekture kao što je CUDA omogućuje da se puno veći broj potencijalnih korisnika okuša u razvoju efikasnih paralelnih aplikacija za grafičke procesore. Od studenog 2006. kada je NVIDIA predstavila CUDA-u razvijena je impresivna brojka projekata koji su njenim korištenjem uspijeli ostvariti ubrzanje za jedan do dva reda veličine.

Ovaj seminar zamišljen je kao kratki vidič za sve željne upustiti se u paralelno programiranje na grafičkim procesorima. U njemu se mogu naći sve bitnije informacije za razvoj efikasnog i optimiziranog koda. Za one koji se samostalno žele upustiti u programiranje na CUDA platformi preporuča se službeni vodič "NVIDIA CUDA Programming Guide" koji je bio i polazišna točka za seminar. Ipak, određeni dijelovi su prošireni i dodatno pojašnjeni - prije svega dio o tekstualnom profiliranju i dio o SIMT arhitekturi. Za te je dijelove jako dobra referenca bila serija web članaka od Rob Farbera "CUDA, Supercomputing for the Masses". Primjer množenja matrica na kraju seminara također je preuzet iz službenog vodiča no nadodana mu je usporedba brzine izvođenja sa CPU-om. Mjerenje je provedeno na GPU-CPU Hybrid clusteru koji se nalazi na Prirodoslovno-matematičkom fakultetu Sveučilišta u Splitu.

Literatura

[1] "Client Statistics by OS", http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats, 19. travnja, 2009.

[2] "NVIDIA CUDA Programming Guide v2.1", http://www.nvidia.com/object/cuda_develop.html, 10. travnja, 2009.

[3] "NVIDIA CudaReferenceManual_2.1", http://www.nvidia.com/object/cuda_develop.html, 10. travnja, 2009.

[4] R. Farber: "CUDA, Supercomputing for the Masses", http://www.ddj.com/architect/207200659, 20.travnja, 2009.

[5] "ASPLOS 2008 CUDA Tutorial", http://gpgpu.org/asplos2008, 25.travnja, 2009.

Sažetak

U sklopu ovog seminara predstavljene su osnovne smjernice za programiranje na NVIDIA-inim grafičkim procesorima uz pomoć CUDA programskog modela. Opisana je arhitektura NVIDIA-inih procesora i način pomoću kojeg ona implementira višedretveno paralelno okruženje u kojem se izvode programi, kao i programski model koji se zasniva na hijerarhiji dretvi koje su podjeljene u blokove. Predstavljena su osnovna proširenja C programskog jezika i dodatni set funkcija koje služe za pristup paralelnim mogućnostima grafičkog procesora. Jedno poglavlje posvećeno je savjetima za optimizaciju programskog koda koja se prije svega oslanja na efikasno korištenje dostupnih vrsta memorije, odabira pogodnih instrukcija koje se izvršavaju u što manje ciklusa sata i smanjuju mogućnost prisilne serijalizacije paralelnog programa te pažljive konfiguracije izvršavanja paralelnih funkcija na grafičkom procesoru. Na kraju je predstavljen primjer implementacije množenja matrica u CUDA modelu i uspoređena su vremena izvođenja s implementacijom na CPU-u.