SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINARSKI RAD

Programska okruženja za programiranje na grafičkim procesorima

Ozana Živković

Voditelj: Doc.dr.sc. Siniša Šegvić

Zagreb, travanj 2010.

Sadržaj

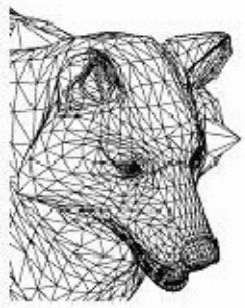
1. Uvod

Srce svih računalnih sustava predstavlja centralna procesorska jedinica. Generalna namjena procesora je obavljanje osnovnih zadataka, što znači da su prvenstveno namijenjeni izvođenju osnovnih matematičkih izračunavanja. Kompliciranizadaci s ogromnom količinom podataka kakvih je u posljednje vrijeme sve više zahtijevaju mnogo procesorske snage. Tako jedno računalo s jednim procesorom opće namjene nije više dovoljno za obradu takvih računski zahtjevnih grafičkih aplikacija. U cilju rješavanja tog problema kreće se sa paralelnom obradom podataka. Manji broj računala povezuje se u grozdove (engl. cluster). Računala su međusobno povezana brzom lokalnom mrežom i omogućuju paralelnu obradu podataka. Međutim, takvo povezivanje računala u grozdove pokazalo se vrlo kompliciranim i vrlo zahtjevnim. Zbog toga se kreće sa povećanjem broja jezgri procesora opće namjene. No, tada proizvođači sve više postaju svjesni potencijala koji nudi grafička procesorska jedinica. Kreće sa eksperimentiranjem izvođenja grafičkih algoritama na grafičkim procesorima. Zapravo, sve algoritme koje se željelo izvoditi na grafičkom procesoru, bilo je potrebno prethodno prevesti u elemente grafike. To je programerima uvelike otežavalo posao i zahtijevalo veliko iskustvo. Uskoro, proizvođači shvaćaju tešku poziciju programera i nastoje im olakšati posao nudeći jednostavnije općenamjensko sučelje. Danas općenamjensko programiranje grafičkih procesora čini posebnu granu u računarstvu. Za takvo uspješno programiranje neizbježno je kvalitetno okruženje. Prva okruženja za općenamjensko programiranje grafičkih procesora bila su napravljena od strane entuzijasta. Tada na tržište izlaze vodeći proizvođači nudeći kvalitetnija i daleko moćnija okruženja. U ovom radu velika pažnja posvećena je upravo jednom takvom (CUDA) okruženju tvrtke NVIDIA. Osnovne značajke CUDA okruženja predstavljene su u poglavlju **4.2**. Poglavlje **5.** posvećeno je programskoj realizaciji jednostavnih primjera u CUDA okruženju. Također, prikazano je dobiveno ubrzanje izvođenja u odnosu na procesore opće namjene.

2. Analiza grafičkih procesora i razlike u odnosu na procesore opće namjene

2.1. Osnovne značajke arhitekture modernih grafičkih procesora

Povijesno gledano, grafičko sklopovlje počinje se prvo koristiti za ubrzavanje dijelova grafičke protočne strukture. Osnovna namjena grafičke protočne strukture je pretvaranje virtualne scene u sliku. Kao što će biti prikazano u nastavku ovog poglavlja, princip rada grafičke protočne strukture analogan je proizvodnji na pokretnoj traci. Grupa poligona ulazi u sustav i nad njima se izvode potrebne operacije. Potom se ta grupa šalje u iduću fazu gdje se nastavlja s obradom, a na njezino mjesto ulazi nova grupa poligona. Poligonima se podrazumijevaju trokuti, eventualno četverokuti koji se koriste za prikazivanje predmeta s barem jednom zakrivljenom plohom. Na slici 1. prikazana je aproksimacija zakrivljene površine pomoću guste mreže poligona.



Slika 1. Vučja glava modelirana mrežom poligona

Grafička protočna struktura podijeljena je na iduće faze:

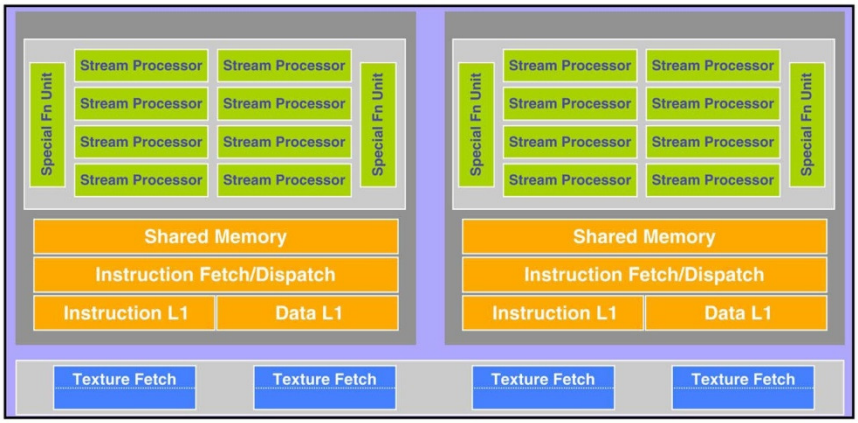
- Aplikacijska faza
- Geometrijska faza
- Faza rasteriziranja

U tom razdoblju grafička protočna struktura prelazi na grafičku procesorsku jedinicu (engl. Graphics Proccesing Unit, GPU). Nakon toga slijedi veliki trend - programirljivost.

Programirljivost je nastojanje da se od potpuno fiksnog sustava dođe do potpuno programirljivog grafičkog procesora s procesorskim elementima [11].

Od samog početka protočna struktura grafičkih procesora temelji se na drugačijem pristupu u odnosu na protočnu strukturu procesora opće namjene. Razmatra se protočnost zasnovana na zadacima. Svaki zadatak obrađuje velik broj ulaznih podataka, a izlaz iz zadatka prosljeđuje se kao ulaz u sljedeći zadatak. Paralelizam zadataka ostvaren je na način da se podaci unutar protočnih faza mogu obrađivati istovremeno. Podatkovnim paralelizmom smatra se istovremena obrada više podataka unutar svake faze. Kod procesora opće namjene u jednom vremenskom trenutku iskorišteni su svi resursi tog procesora. Svaki resurs se koristi u jednoj od faza protočne strukture. Svaka od faza u jednom vremenskom trenutku izvodi se nad različitom instrukcijom. Kod grafičkih procesora resursi se dijele na različite faze. Resurse grafičkog procesora predstavlja niz programirljivih *stream* procesora, a samim time svaki *stream* procesor može sudjelovati u različitim fazama tokom svog rada. Točnije, protočnost je razdijeljena u prostoru, a ne u vremenu kao što je to slučaj kod procesora opće namjene. Dio procesora (određen brojem *stream* procesora) radi na jednoj fazi, čiji se izlaz automatski preusmjerava kao ulaz u iduću fazu. Takva organizacija protočnosti je vrlo učinkovita kod grafičkih procesora koji su fokusirani na jednu funkciju. Učinkovitost proizlazi iz toga što se u bilo kojem trenutku obrađuje mnogo faza paralelnih zadataka, a unutar svake faze ostvarena je paralelna obrada podataka. Nadalje, svaka faza se može implementirati pomoću sklopa posebne namjene. Ukratko, rezultat je GPU protočna struktura s mnogo faza čija je funkcionalnost ubrzana korištenjem paralelnih sklopova posebne namjene i *stream* procesora. Također, izvođenje grafičkih operacija unutar grafičkog procesora može trajati i do tisuću ciklusa. Visokoj propusnosti pridonosi podatkovni paralelizam unutar faza i paralelizam zadataka između pojedinih faza. Glavni nedostatak je problem uravnotežavanja opterećenja i to što izvođenje unutar protočne strukture ovisi o izvođenju najsporije faze [7].

Strogi paralelizam zadataka unutar protočne strukture preslikava se u jedinstvene *shader* arhitekture. To su takve arhitekture gdje se sve programirljive jedinice protočne strukture sjedinjene i prikazuju prema van kao jedna sklopovska programirljiva jedinica.



Slika 2. Arhitektura grafičkog procesora GeForce 8800 GTX tvrtke NVIDIA

Na slici 2. prikazana je arhitektura grafičkog procesora *GeForce 8800 GTX* tvrtke NVIDIA. Radi se o *shader* arhitekturi, odnosno arhitekturi zasnovanoj na jedinstvenim programirljivim jedinicama. Ovaj grafički procesor sastoji se od šesnaest *stream* multiprocesora, a unutar svakog multiprocesora nalazi se osam *stream* procesora. Jedan takav par multiprocesora može se vidjeti na slici. Unutar svakog *stream* multiprocesora, osim osam *stream* procesora, nalaze se i zajednička pričuvna memorija za podatke i instrukcije, logička kontrolna jedinica, 16kB zajedničke memorije, te dvije specijalne jedinice za obavljanje posebnih funkcija.

2.2. Razlike između procesora opće namjene i grafičkih procesora

Fizička ograničenja u proizvodnji procesora opće namjene (engl. Central Processing Unit, CPU) zajedno s visokom potrošnjom energije sputavaju daljnji rast frekvencije procesora. Naime, Mooreov zakon kaže da se za postizanje smanjenja tehnologije izrade svake dvije godine broj tranzistora poveća dva puta. Trenutno je kanal u tranzistoru koji ne provodi električnu energiju širine 32nm, a njegovim smanjivanjem sve se više približava fizičkoj granici, tj. širini jednog atoma. To nije problem samo procesora opće namjene već svih mikroelektroničkih uređaja. Iz tog razloga, poboljšanje performansi procesora opće namjene ostvareno je povećanjem broja jezgri. Današnji procesori, namijenjeni za uporabu

od strane običnih korisnika, mogu sadržavati do četiri jezgre. Pretpostavlja se da daljnji rast jezgri neće biti tako brz jer zahtjeva rješavanje problema pristupa zajedničkoj memoriji. Procesori su dizajnirani na način da zajednički mogu rješavati neki računalni problem, dok cjelokupni procesor predstavlja MIMD (engl. Multiple Instruction, Multiple Dana Stream) model.

Posljednjih nekoliko godina ubrzano raste broj računski zahtjevnijih aplikacija, te se u procesore opće namjene uvode posebna vektorska proširenja (MMX, SSE, SSE2, SSE3). Međutim, područje njihove primjene je vrlo usko i nije dovoljno za postizanje visokih, vrhunskih performansi u svijetu trodimenzionalne grafike. Za učinkovito obavljanje takvih zadatka koristi se grafička procesorska jedinica.

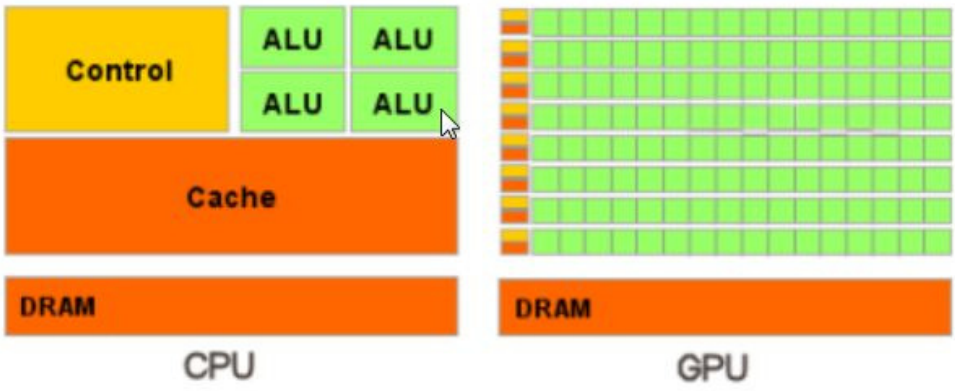
Kao što je već ranije spomenuto, prije nekoliko godina GPU procesor je bio fokusiran na jednu funkciju i izgrađen oko grafičke protočne strukture s primjenom u trodimenzionalnoj grafici. S vremenom, takvi grafički procesori evoluiraju u moćne programirljive procesore s aplikacijskim sučeljem i sklopom koji se sve više fokusira na programirljiv aspekt. Kao rezultat toga, današnji grafički procesori posjeduju ogromni aritmetički kapacitet i visoku memorijsku propusnost. Time nadaleko nadilaze mogućnosti najbržih procesora opće namjene, te se koriste za skup aplikacija sa slijedećim karakteristikama:

- Zahtjevi za ogromnom količinom računanja unutar stvarnog vremena
- Zahtjevi gdje paralelizam donosi veliki napredak u brzini
- Zahtjevi gdje je propusnost važnija od brzine jednog računanja

Kako su grafički procesori dizajnirani za drugačije tipove aplikacija u odnosu na procesore opće namjene, tako se i njihova arhitektura razvija u drugačijem smjeru. U prethodnom poglavlju prikazane su glavne značajke te arhitekture. Tako se multiprocesori (kako bi se lakše provela usporedba s procesorom opće namjene u daljnjem tekstu multiprocesori će biti jezgre GPUa) temelje na SIMD (Single Instruction, Multiple Data) modelu. Znači, jezgre GPUa obrađuju istovremeno više podataka. Iz toga slijedi bitna razlika između procesora opće namjene i grafičkih procesora. Jezgre procesora opće namjene od samog početka dizajnirane su da slijedno izvode niz instrukcija maksimalnom brzinom. Za razliku od toga, grafički procesor omogućava paralelno izvođenje više instrukcija. Također, unutar jednog ciklusa moguće je izvesti više od jedne instrukcije. Razlike između GPUa i CPUa proizlaze i iz

različitih načela pristupa memoriji. Budući da su grafički procesori namijenjeni obradi grafike gdje je izuzetno važan redoslijed podataka (slikovnih elemenata), pristupanje uzastopnim memorijskim lokacijama (bilo da se radi o čitanju ili pisanju) zahtjeva manje vremena. Za razliku od toga, kod procesora opće namjene pristup memoriji zahtjeva jednako vrijeme, neovisno da li se radi o uzastopnim ili raštrkanim memorijskim lokacijama. Što se tiče memorijskih upravljača, svi grafički procesori sadrže njih nekoliko, dok isto ne vrijedi za sve procesore opće namjene. Nadalje, paralelno izvođenje velikog toka podataka traži i veliku memorijsku propusnost, koja je jedna od glavnih značajki grafičkih procesora. Procesori opće namjene koriste veliku količinu pričuvne memorije kako bi smanjili latenciju pristupanja memoriji i time povećali performanse. Nasuprot tome, grafički procesori koriste manju pričuvnu memoriju i time postižu veću memorijsku propusnost. Latenciju pristupa memoriji reduciraju izvođenjem više od tisuću dretvi istovremeno. Znači kad jedna dretva čeka na podatke iz memorije, bez latencije se istovremeno izvodi druga dretva.

Sve prethodno navedene razlike između procesora opće namjene i grafičkih procesora proizlaze kao rezultat razlika u temeljnom dizajnu ovih dvaju procesora. Na slici 3., može se vidjeti taj različit pristup u dizajniranju arhitekture [5].



Slika 3. CPU i GPU posjeduju temeljne razlike u dizajnu

3. Općenamjensko programiranje na grafičkim procesorima

Kao što je već ranije spomenuto, izvođenje računski zahtjevnih operacija nije učinkovito na procesorima opće namjene. Grafički procesori su od samog početka dizajnirani za obrađivanje drugačijih aplikacija i njihova arhitektura razvija se u drugačijem smjeru. S vremenom od procesora fokusiranih na izvođenje jedne funkcije evoluiraju u moćne programirljive procesore. Sve to potiče proizvođače i programere da pokušaju iskoristit ogromnu memorijsku propusnost i računalnu snagu grafičkog procesora za izvođenje računski zahtjevnijih operacija. Posebno veliki interes za općenamjensko programiranje grafičkih procesora (engl. *General-Purpose computation on Graphics Processing Units*, GPGPU) javlja se prije nekoliko godina.

Grafički procesor prima na ulazu grupu poligona nad kojima izvodi potrebne grafičke operacije, a na svojem izlazu daje sliku. Iz tog razloga, na samom početku sve što se htjelo izvoditi na grafičkom procesoru bilo je nužno prevesti u elemente grafike. Potrebno je veliko programersko iskustvo kako bi se algoritmi koji se žele izvoditi na grafičkim procesorima prikazali kao grafički algoritmi. U to vrijeme program je bio strukturiran u odnosu na grafičku protočnu strukturu. Također, programirljive jedinice su bile dostupne jedino kao međukoraci unutar protočne strukture. Tadašnja želja programera je bila mogućnost izravnog pristupanja jednoj programirljivoj sklopovskoj jedinici (engl. Shader architecture) bez potrebe dijeljenja posla na više programirljivih jedinica. S vremenom, proizvođači shvaćaju potrebe programera i izlaze im u susret nudeći sučelje koje nije grafičko i jednu programirljivu jedinicu. Posao se današnjeg programera modernih grafičkih procesora svodi samo na definiranje računalne domene u obliku strukturirane mreže dretvi (engl. grid). Vrijednost svake od dretvi izračunava se općenamjenskim programom, koji omogućava izvođenje instrukcija na više podataka. Dobivene vrijednosti rezultat su matematičkih operacija na podacima globalne memorije. Kod pristupanja globalnoj memoriji (bilo da se radi o čitanju ili pisanju) koristi se isti međuspremik. Rezultat koji ostane unutar međuspremika može poslužiti kao ulaz u idućem izračunavanju. Ovakvim pristupom izbjegnuta je kompleksnost prethodnog načina gdje su programeri bili zaduženi za prilagođavanje grafičkih aplikacija.

Rezultat svega toga su programi implementirani u jednostavnijim i popularnijim programskim jezicima, čime je omogućena jednostavnija implementacija i provjera. Trenutno je najavljen dolazak nove arhitekture grafičkog procesora pod nazivom „Fermi“ od tvrtke NVIDIA koji će u potpunosti omogućiti normalan rad grafičkog procesora [15].

4. Relevantna okruženja za programiranje na grafičkim procesorima

Pokušaji iskorištavanja snage grafičkog procesora za zahtjevnije računske aplikacije s mnogo podatkovnog paralelizma započinju prije nekoliko godina. Takvi prvi pokušaji su bili vrlo jednostavni i svodili su se na korištenje sklopovskih funkcija kao što su na primjer rasterizacija i Z-međuspremik. Pojavnom *shader* arhitekture započinje se s ubrzavanjem matričnih izračunavanja. U to vrijeme programeri grafičkih procesora na raspolaganju imaju Direct3D ili OpenGL. Radi se o aplikacijama za iscrtavanje trodimenzionalne grafike koje su od programera tražile poznavanje tekstura. No, tada se pojavljuje prevoditelj BrookGPU koji je kreiran za aplikacije koje nisu grafičke Na taj način se olakšava posao programera. BrookGPU prevodi C++ kod zajedno s dodanim proširenjima, te se zatim povezuje s bibliotekama koje podržavaju DirectX i OpenGL. Brook imao je veliki broj nedostataka, ali istovremeno daje proizvođačima do znanja kako je budućnost općenamjensko paralelno programiranje grafičkih procesora. Tako tvrtka NVIDIA nešto kasnije na tržište izlazi sa CUDA-om. Mnogo detaljnije o CUDA okruženju za programiranje grafičkih procesora može se pročitati u odjeljku 4.2. Isto tako, kako bi održao korak na tržištu, AMD započinje sa razvojem okoline za GPGPU aplikacije. Danas najpoznatije okruženje koje ATI/AMD primarno preporuča je OpenCL.

4.1. Programsko okruženje OpenCL

Prva važnija razvojna okolina za GPGPU programiranje jest OpenCL (engl. *Open Computing Language*). Iako se u velikoj mjeri koristi u GPGPU grani računarstva, to nije primarna namjena OpenCL-a. OpenCL je zapravo okruženje za razvoj aplikacija koje se izvršavaju na velikom broju heterogenih platformi. U te platforme se ubrajaju i razne inačice centralnih procesnih jedinica, kao i različiti grafički procesori. Upravo mogućnost razvoja aplikacija za grafičke procesore je razlog što je OpenCL veoma popularan u tom području.

OpenCL se sastoji od vlastitog jezika u kojem se pišu aplikacije, a zasnovan na C99 verziji programskog jezika C, te sučelja koja služe za upravljanje sklopovima koji izvode aplikacije. U ovom slučaju je riječ o sučelju prema grafičkom procesoru. OpenCL se zasniva na dvije vrste paralelizma:

- paralelizam zadataka
- paralelizam podataka

O paralelizmu zadataka je bilo riječi u prijašnjim poglavljima prilikom opisivanja grafičkih procesora i usporedbe grafičkih procesora i procesora opće namjene. Dovoljno je reći kako se moderni grafički procesori zasnivaju na paralelizmu zadataka. Osnovna ideja OpenCL-a jest omogućiti pristup grafičkom procesoru i memoriji grafičkog procesora operacijskom sustavu i programeru koji ih želi iskoristiti. Pristup memoriji grafičkog procesora je važan iz činjenice što grafički procesor dohvaća podatke za obradu upravo iz te memorije. Pošto operacijski sustav nema pristup toj memoriji potrebno je prvo podatke koji se obrađuju prenijeti u memoriju grafičkog procesora. Iako to oduzima dodatno vrijeme, na tom principu rade sva okruženja za izradu GPGPU aplikacija.

Razvoj na OpenCL-u je započeo Apple početkom 2008. godine. Kako su i ostale velike tvrtke bile zainteresirane za projekt, stvorena je nezavisna grupa nazvana *Khronos Group* koja je trebala nastaviti razvoj OpenCL-a. Prva inačica je izašla krajem 2008. godine i uključena je u tada trenutnu verziju MacOS operativnog sustava [9].

Popularnost i razvoj OpenCL-a uvjetovana je interesom ATI-ja, odnosno AMD-a. Nakon što je NVIDIA započela razvoj svoje CUDA platforme ATI/AMD je morao tržištu ponuditi vlastitu inačicu iste tehnologije. Paralelno sa FireStream, ATI je počeo razvijati i platformu za programiranje grafičkih procesora. Nazvana je *Stream SDK* i pandan je NVIDIA CUDA-i. U početku je AMD koristio *Close to Metal* kao okolinu za razvoj GPGPU aplikacija za svoje grafičke procesore, no nakon izdavanja novih DirectX 11 grafičkih procesora se odlučio za prelazak na OpenCL. Samim time, OpenCL je dobio na popularnosti te je postao dio Stream SDK-a.

Iako je OpenCL većinom poznat kao okruženje koje AMD primarno preporuča, niti ostali proizvođači ne izostaju sa podrškom. NVIDIA, IBM, VIA, Intel i neki drugi su odlučili omogućiti podršku za OpenCL u posljednjim verzijama programskih upravljača (engl. driver*)* za svoje grafičke kartice.

4.2. Programsko okruženje CUDA

Prije nekoliko godina tvrtka NVIDIA predstavlja novu arhitekturu za paralelno programiranje pod imenom CUDA (engl. *Compute Unified Device Architecture*). Radi se općenamjenskoj paralelnoj arhitekturi s novim paralelnim programskim modelom i skupom instrukcija, koja daleko pojednostavljuje paralelno programiranje na grafičkim procesorima. Time je omogućeno učinkovitije rješavanje mnogih složenijih računalnih problema na grafičkim procesorima nego što to omogućavaju procesori opće namjene.

4.2.1. Povijesni razvoj

U studenom 2006. godine tvrtka NVIDIA objavljuje CUDA projekt zajedno s G80 (engl. *GeForce 8 Series*) grafičkim procesorom. Javna Beta 1.0 verzija programskog razvojnog alata pojavljuje se u veljači iduće godine. Krajem 2007. godine na tržište izlazi verzija Beta 1.1 s proširenim skupom mogućnosti, ali ona nije bila toliko popularna kao prva verzija. Pokretanje CUDA programa zahtijeva NVIDIJINE grafičke kartice od verzije G80 na dalje, te upravljačke programe (engl. drivers) od verzije 169.xx na dalje. To omogućava CUDA programiranje na bilo kojem računalu s prethodim karakteristikama i uvodi programere grafičkih procesora u novo, naprednije razdoblje. Omogućen je asinkroni prijenos podataka u video memoriju, podržane su operacije za direktan pristup memoriji i izvođenje programa na 64-bitnim *Windows* operacijskim sustavima. Mogućnost spajanja dviju ili više grafičkih kartica koje zajedno obrađuju elemente grafike. Radi se o aplikacijama za paralelnu obradu računske grafike [14].

Trenutna verzija Beta 2.0 izlazi zajedno s *GeForce GTX200* u proljeće 2008. godine. Glavne prednosti ove verzije su: obrada podataka dvostruke preciznosti, podrška za operacijske sustave *Windows Vistu* i *Mac OS X*, optimiziran prijenos podataka, te podrška za 3D teksture. Kako je arhitektura prvotno dizajnirana za čitanje 32-bitnih podataka iz memorije i registara, obradom 64-bitnih podataka ne postižu se značajno bolje performanse. Isto tako grafičke aplikacije ne zahtijevaju obradu podataka dvostruke preciznosti. Međutim,

to je ipak vrlo korisno u projektima gdje se uz podatke jednostruke preciznosti pojavljuju i podaci dvostruke preciznosti. Prije nego što NVIDA uvodi mogućnost obrade takvih podataka, koristi se tehnika gdje se međurezultati jednostruke preciznosti grafičkog procesora šalju na daljnju obradu procesoru opće namjene. Iduća prednost Beta 2.0 verzije proizlazi iz mogućnosti izvođenja bilo kojeg CUDA program na procesorima opće namjene. No, zbog posve drugačije arhitekture, izvođenje CUDA koda na CPU je daleko sporije od njegovog izvođenja na grafičkim procesorima i time se gubi glavni smisao nastanka CUDE.

4.2.2. Značajke i prednosti CUDA okruženja

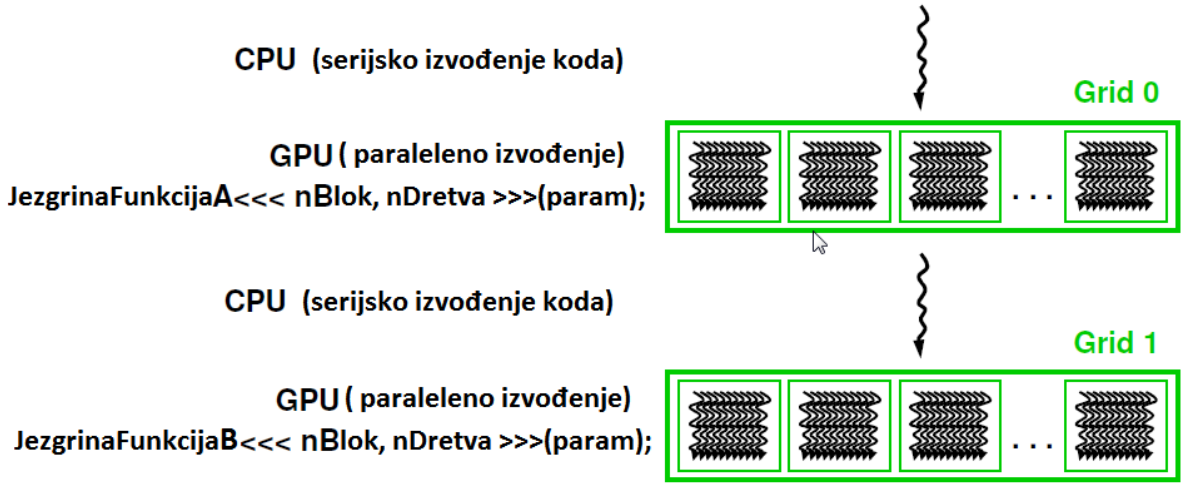
Programiranje CUDA tehnologijom je poprilično zahtjevno jer traži od programera potpuno razumijevanje organizacije memorije. Organizaciji memorije posvećen je odjeljak 4.2.5. Ipak, daleko je lakše od prethodnih GPGPU metoda jer algoritme nije potrebno prethodno prikazivati kao grafičke algoritme, odnosno strukturirati ih u odnosu na protočnu grafičku strukturu. Programi se dijele između nekoliko multiprocesora, što programiranje u CUDI povezuje s programiranjem MPI (engl. *Message Passing Interface*) i OpenMP (engl. Open Multi-Processing). Bitna prednost je to što CUDA dolazi zajedno s programskim okruženjem koje omogućava korištenje standardnog programskog jezika C sa minimalnim NVIDIA proširenjima za pristup resursima na grafičkim procesorima. Za prevođenje koda, CUDA koristi vlastiti prevoditelj Open64 C. Ne ovisi o grafičkim aplikacijama i posjeduje bitne osobine za općenamjensko programiranje, npr. izvođenje aritmetičkih operacija. Osim toga, CUDA uvodi neke sklopovske značajke kao što je zajednička memorija. To je vrlo mala količina memorije, oko 16KB za svaki multiprocesor, odnosno jezgru. Više o tome u odjeljku 4.2.5. Sljedeća bitna značajka je prikladniji pristup memoriji. Rezultat programa u grafičkim aplikacijama je 32-bitna vrijednost jednostruke preciznosti koja se pohranjuje na unaprijed definirano područje. Programsko okruženje CUDA omogućuje pohranu podataka na bilo koju memorijsku lokaciju, što omogućuje izvođenje algoritama koji se ranije nisu mogli izvesti dostupnim GPGPU metodama. Osim pisanja podataka, podaci se mogu i čitati sa bilo koje adrese. Također, programerima pruža mogućnost programiranja na nižoj razini, odnosno u asembleru.

Osim prednosti koje pruža programsko okruženje CUDA, postoji i nekoliko nedostataka. Glavni nedostatak je to što izvođenje CUDA programa zahtjeva integriranu NVIDIjINU grafičku karticu u računalu. Sljedeći nedostatak je što svaki od blokova unutar mreže dretvi mora sadržavati minimalno 32 dretve. Više o tome može se saznati u poglavlju **4.2.4**. Još jedan nedostatak CUDA okruženja je i to što ne podržava rekurzivne funkcije.

Potrebno je spomenuti da CUDA posjeduje aplikacije visoke razine (*Runtime API*) i aplikacije niske razine (*Driver API*). Te aplikacije ne mogu biti istovremeno korištene od strane jednog programa. Svi pozivi visoke razine prosljeđuju se i obrađuju u aplikacijama niske razine. Iznad aplikacija visoke razine postoji još viša razina s dvije biblioteke. Jedna od njih je CUBLAS (engl. CUDA *Basic Linear Algebra Subprograms*) za standardne algoritme linearne algebre. Druga biblioteka je CUFF (engl. CUDA *Fast Fourier Transform*), koja je vrlo bitna za analizu, filtriranje i obradu signala.

4.2.3. Programski model

CUDA program se sastoji od jednog ili više dijelova koji se izvode na procesoru opće namjene ili na grafičkom procesoru. Dio programa s vrlo malo podatkovnog paralelizma ili onaj dio gdje ga uopće nema, implementira se standardnim jezikom C. Prevodi se i izvodi se na procesoru opće namjene kao i svaki drugi običan program. S druge strane, dio koda koji posjeduje veliku količinu podatkovnog paralelizma se implementira jezikom C i njegovim proširenjima za označavanje paralelnih funkcija i pridružene podatkovne strukture. Funkcija za paralelnu obradu podataka u daljem tekstu nazivat će se funkcija jezgre. Takav dio koda s velikom količinom podatkovnog paralelizma se izvodi na grafičkom procesoru. Izvođenje programa započinje na procesoru opće namjene, a pozivom funkcija jezgre izvođenje se nastavlja na grafičkom procesoru (vidi sliku 4) [2].



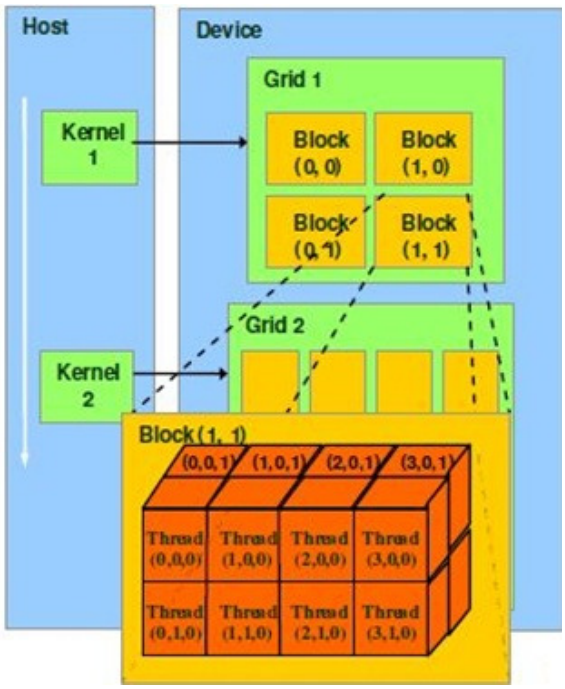
Slika 4. Izvođenje CUDA programa

Kako bi se iskoristila velika količina podatkovnog paralelizma, prilikom poziva jezgrine funkcije generira se ogroman broj dretvi. Jezgrina funkcija zapravo predstavlja dio koda koji će paralelno izvoditi sve dretve. Generirane dretve sačinjavaju mrežu dretvi (engl. grid). Više o njihovoj organizaciji može se pročitati u odjeljku 4.2.4. Kad sve dretve generirane prilikom poziva jezgrine funkcije završe, izvođenje se nastavlja na procesoru opće namjene. Svaki grafički procesor posjeduje svoju vlastitu memoriju. Kako bi se pozivom jezgrine funkcije omogućio nastavak izvođenja na grafičkom procesoru, programer je dužan zauzeti potrebnu količinu memorije na grafičkom procesoru. Nakon toga, programer je dužan u nju prebaciti podatke potrebne za izvođenje jezgrine funkcije iz memorije hosta. Ovdje se hostom smatra cjelina unutar koje se nalazi CPU. Nakon završetka izvođenja, potrebno je rezultat prebaciti natrag u memoriju hosta, te osloboditi prethodno zauzetu memoriju. Mnogo detaljnije o memoriji može se pročitati u odjeljku 4.2.5.

4.2.4. Organizacija dretvi

Kao što je već ranije spomenuto, pozivom jezgrine funkcije generiraju se mreže dretvi, a sve dretve unutar jedne mreže izvodit će isti odsječak koda. Dretve unutar mreže su organizirane u dvije razine hijerarhije [3]. Svaka mreža sadrži veliki broj dretvi koje su podijeljene unutar blokova. Naime, svaka mreža sastoji se od jednog ili više blokova unutar

kojih se nalazi jednak broj dretvi organiziranih na isti način. Blok dretvi organiziran je kao trodimenzionalno polje, a maksimalan broj dretvi unutar bloka iznosi 512. Ne koriste sve aplikacije sve tri moguće dimenzije bloka, tako da blokovi mogu biti i jednodimenzionalni ili dvodimenzionalni. Broj blokova u svakoj dimenziji i broj dretvi unutar jednog bloka određeni su ugrađenim varijablama. Varijable se inicijaliziraju prilikom izvođenja sustava pomoću vrijednosti koje je programer dužan navesti kod poziva jezgrine funkcije. Njihova glavna namjena je omogućiti da svaki blok i svaka dretva unutar tog bloka posjeduje jedinstvene koordinate pomoću kojih se pojedina dretva razlikuje od svih ostalih. Osim njihovog međusobnog raspoznavanja, varijable se koriste i za određivanje pozicije podatka koji će pojedina dretva obrađivati. Na slici 5. prikazana je organizacija dretvi unutar blokova. Kako svi blokovi sadrže jednak broj dretvi organiziranih na isti način, dovoljno je prikazati dretve unutar jednog bloka. Prikazani blok organiziran je kao trodimenzionalno 4×2×2 polje dretvi. Znači, unutar svakog bloka se nalazi 16 dretvi, a kako se svaka mreža sastoji od 4 bloka zaključujemo da se unutar jedne mreže nalazi ukupno 64 dretve. U stvarnosti je taj ukupan broj dretvi daleko veći.



Slika 5. Organizacija dretvi

Potrebno je osigurati da sve dretve unutar bloka završe s izvođenjem jedne faze prije nego što nastave s izvođenjem iduće faze. To se osigurava posebnom funkcijom za sinkronizaciju. Po pozivu te funkcije od strane jezgre, sve dretve zaustavljaju sa svojim izvođenjem i čekaju da preostale dretve unutar bloka dođu do tog mjesta. Sinkronizacija dretvi između različitih blokova nije moguća. Također, dretve unutar bloka se izvode u obliku malih skupina (engl. wraps). Broj dretvi unutar te skupine ovisi o implementaciji. Njihova glavna namjena je učinkovito izvođenje operacija s velikom latencijom, kao što je na primjer pristupanje memoriji. Ukoliko jedna skupina dretvi čeka na podatke, istovremeno se izvodi druga skupina. U bilo koje vrijeme unutar jednog bloka omogućeno je izvođenje samo jedne takve skupine, a za to se brine mehanizam zasnovan na prioritetima. Maksimalna učinkovitost se postiže paralelnim izvođenjem blokova, odnosno paralelnim izvođenjem jedne skupine dretvi iz svih blokova. Međutim, ukoliko grafički procesor ne posjeduje dovoljno resursa (multiprocesora), blokovi se mogu izvodit jedan po jedan. Isto tako, izvođenje svih blokova ne mora se odvijati istom brzinom. Svojstvo izvođenja iste aplikacije različitim brzinama naziva se transparentna skalabilnost i omogućava programerima lakše programiranje, te povećava iskoristivost aplikacija.

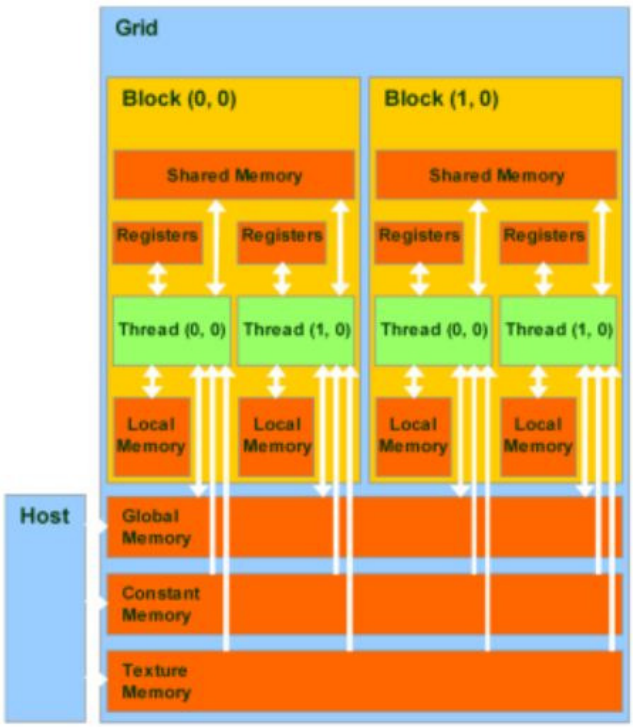
4.2.5. Memorija

U odjeljku **4.2.2** već su spomenute neke bitne značajke pristupanja memoriji. Ovdje će biti dan detaljan prikaz memorije. Kao što se može vidjeti na slici 6., postoji nekoliko tipova memorije.

Globalna memorija predstavlja najveću količinu memorije koja je dostupna svim stream multprocesorima. Količina globalne memorije varira između 256MB i 1.5GB, koliko nalazimo na najjačim grafičkim karticama (GTX295). Pruža veliku propusnost (do 100GB/s), ali operacije pristupa toj memoriji posjeduju veliku latenciju (oko stotinu ciklusa). Kako je vrijeme pristupa memoriji vrlo sporo, host će pohraniti potrebne podatke u tu memoriju, a za njihovu obradu potrebno ih je dohvatiti u zajedničku memoriju. Osim pisanja i čitanja globalne memorije, host također ima mogućnost pristupanja memoriji za konstante i memoriji teksture. Potrebno je napomenuti da lokalna, globalna, memorija za konstante i memorija teksture fizički čine jednu memoriju, ali s različitim mogućnostima pristupa.

Podatke iz memorije teksture i memorije za konstante mogu čitati svi multiprocesori grafičkog procesora. Količina memorije za konstante iznosi 64KB. Pristupanje je ovim memorijama izrazito sporo.

Na slici 6. može se vidjeti da unutar svakog bloka postoje: zajednička memorija, lokalna memorija i određen broj registara. Pristupanje registrima je izuzetno brzo kao i pristupanje zajedničkoj memoriji. Svaka dretva može pristupiti samo jednom dodijeljenom registru. Unutar registra najčešće se spremaju privatne varijable dretve. Lokalna memorija također je vrlo mala, a pristupiti joj može samo jedan stream procesor unutar multiprocesora. Što se tiče zajedničke memorije, ovdje se radi o 16kB brze memorije kojoj mogu pristupati svi stream procesori unutar jednog multiprocesora. Najčešće se koristi za interakciju dretvi tijekom izvođenja [4].



Slika 6. GetForce 8800GTX implementacija CUDA memorije

5. Primjeri programa za CUDA-u i ocjena njihove performanse

U ovom poglavlju prikazano je nekoliko jednostavnih primjera. Naglasak je na tome da se pokaže korištenje osnovnih CUDA API funkcija i sama struktura programa. Na kraju, u odjeljku 5.4 prikazani su i analizirani rezultati izvođenja [5].

5.1. Kvadriranje elemenata polja

Ovdje će biti prikazan jednostavan primjer CUDA programa s naglaskom da se prikaže sama struktura programa. Zadatak je izračunati vrijednosti elemenata jednodimenzionalnog polja duljine deset. Vrijednost svakog elementa će bit jednaka kvadratu njegove pozicije.

Program se sastoji od dva dijela, glavnog programa i jezgrine funkcije koja će se izvoditi na grafičkom procesoru. Glavni program koji se izvodi na procesoru opće namjene ima sljedeće zadatke:

- Inicijalizacija podatkovnog polja
- Kopiranje podatkovnog polja iz memorije hosta u CUDA memoriju grafičkog procesora
- Sinkronizacija dretvi i početak mjerenja vremena potrebnog za izvođenje jezgrine funkcije
- Poziv jezgrine funkcije koja izračunava vrijednosti elemenata polja
- Sinkronizacija dretvi i zaustavljanje mjerenja vremena
- Kopiranje podatkovnog polja iz memorije grafičkog procesora natrag u memoriju hosta
- Prikaz rezultata

Znači, prvo se unutar glavnog programa stvaraju dva pokazivača. Prvi pokazivač `a_h` pokazuje na polje spremljeno u memoriji hosta. To se polje alocira standardnom malloc() rutinom. Drugi pokazivač `a_g` pokazuje na podatkovno polje alocirano u CUDA memoriji grafičkog procesora. Za njegovu alokaciju koristi se CUDA API funkcija `cudaMalloc()`. Parametri koji se predaju funkciji `cudaMalloc()` su pokazivač na zauzetu memoriju i broj bajtova koje je potrebno zauzeti. Sljedeći korak je inicijalizacija polja spremljenog u memoriji

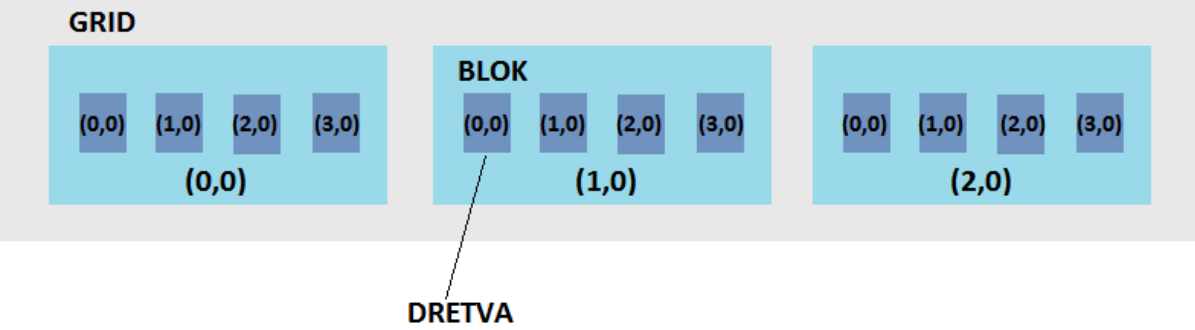
hosta. Svaki element se inicijalizira na vrijednost njegove pozicije. Slijedi kopiranje podatka iz memorije hosta u prethodno zauzetu CUDA memoriju grafičkog procesora. Podaci se kopiraju pomoću CUDA API funkcije `cudaMemcpy()`. Ova funkcija zahtijeva četiri parametra. Prvi parametar pokazuje na odredišnu lokaciju gdje će podaci biti kopirani. Drugi parametar pokazuje na podatke koje će se kopirati. Treći parametar određuje broj bajtova koje je potrebno kopirati. Posljednji, četvrti parametar određuje tip memorija na koje se odnosi kopiranje. Ovisno o smjeru, razlikuju se četiri vrste kopiranja:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

U ovoj situaciji podatkovno polje iz memorije hosta kopira se u CUDA memoriju, pa se kao četvrti parametar koristi `cudaMemcpyHostToDevice`. Nakon kopiranja podatkovnog polja može se pozvati jezgrina funkcija `square_array` koja će ih dalje obrađivati. Znači, sada jezgrina funkcija sadrži zasebno polje. Svaki element toga polja obrađuje jedna dretva. Dretve se grupiraju u skupine i na taj se način formiraju blokovi dretvi. Potreban broj blokova čini grid. U ovom primjeru polje se sastoji od deset elemenata, a kako svaka dretva obrađuje jedan element, broj dretvi unutar bloka postavljen je na četiri. Znači, da bi svaka dretva mogla obrađivati jedan element potrebna su tri takva bloka dretvi. Nakon određivanja broja blokova unutra grida i broja dretvi unutar bloka, potrebno je započeti s mjerenjem vremena. Izmjereno vrijeme potrebno za izvođenje jezgrine funkcije u nastavku će se usporediti sa vremenom obrade procesora opće namjene. U ovom primjeru mjerenje vremena pokreće se i zaustavlja tijekom izvođenja na procesoru opće namjene. Takav pristup zahtjeva sinkronizaciju dretvi procesora opće namjene sa grafičkim procesorom. Većina CUDA API funkcija je asinkrona. Odnosno, vraćaju kontrolu procesoru opće namjene prije nego što završe s poslom. Pozivanjem funkcije `cudaThreadSynchronize()` blokiraju se pozivi dretvi procesora opće namjene sve dok s poslom ne završe sve dretve u CUDI. Nakon početka mjerenja slijedi pozivanje jezgrine funkcije. Poziv jezgrine funkcije razlikuje se od uobičajenih poziva funkcija u standardnom jeziku C i izgleda ovako:

```
KernelFunction <<< dimGrid, dimBlock >>>(parametri)
```

Prvo se navodi sam naziv jezgrine funkcije, zatim se unutar `<<< … >>>` zadaje prethodno izračunat broj blokova koji određuju dimenziju grida i broj dretvi unutar bloka koji određuju dimenziju bloka. Po pozivu jezgrine funkcije u CUDA memoriji formira se grid prikazan na slici 7. Unutar zagrada se navodi niz parametara koji se predaju funkciji. U ovom primjeru to je pokazivač na polje podataka unutar CUDA memorije i broj elemenata polja.



Slika 7. Formiran grid unutar CUDA memorije

Ispred jezgrine funkcije `square_array` se nalazi kvantifikator `__global__`. Na taj način je označeno da se radi o jezgrinoj funkciji koja se izvodi na grafičkom procesoru. Funkcija prima broj elemenata polja i pokazivač na to polje. Svaka dretva unutar grid-a izvoditi će isti kod jezgrine funkcije i odrađivat će samo jedan element polja. Kao što je opisano u odjeljku 4.2.4 svaka se dretva razlikuje od ostalih dretvi unutar grida na temelju svojih koordinata. U ovom primjeru, svakoj dretvi dodjeljuje se jedan element polja na temelju njezinih koordinata. Indeks elementa polja pronalazi se množenjem indeksa bloka dretvi (`blockIdx.x`) sa brojem dretvi unutar bloka (`blockDim.x`) i dodavanjem indeksa dretve unutar bloka (`threadIdx.x`). Ukoliko je izračunati indeks unutar granica polja (manji od deset), vrijednost se pripadnog elementa kvadrira. Po povratku iz jezgrine funkcije obavlja se sinkronizacija dretvi i zaustavlja se mjerenje vremena. Podatkovne polje iz CUDA memorije grafičkog procesora kopira se memoriju hosta i ispisuje se rezultat. Na kraju oslobađa se prethodno zauzeta CUDA memorija i memorija hosta. U nastavku može se vidjeti opisana implementacija.

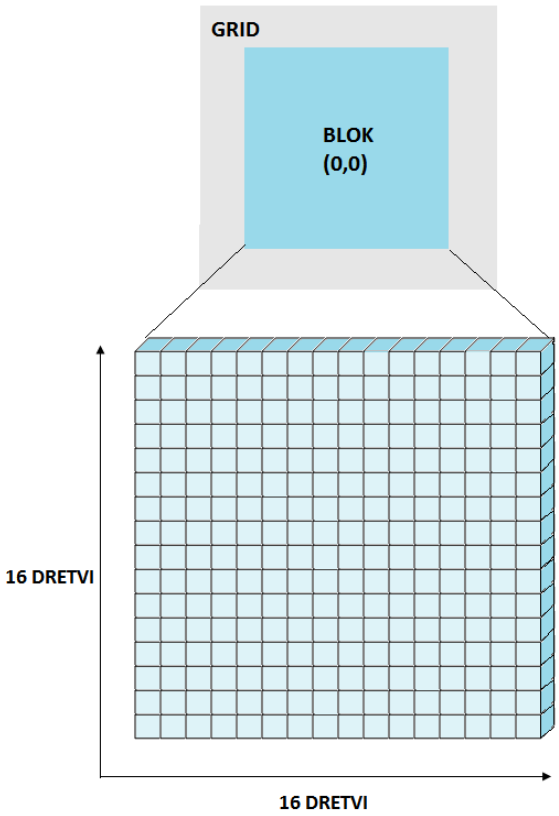
```
__global__ void square_array(float *a, int N)
{
       int idx = blockIdx.x * blockDim.x + threadIdx.x;
       if (idx<N) a[idx] = a[idx] * a[idx];
}

int main(void)
{

        float *a_h, *a_g;
         const int N = 10;
         int i;
         unsigned int timer=0;
         size_t velicina = N * sizeof(float);

         a_h = (float *)malloc(velicina);
         cudaMalloc((void **) &a_g, velicina);

         for (i=0; i<N; i++)
              a_h[i] = (float)i;

         cudaMemcpy(a_g, a_h, velicina, cudaMemcpyHostToDevice);

         int blok_velicina = 4;

         int n_blokova = N/blok_velicina + (N%blok_velicina == 0 ? 0:1);

         cutCreateTimer( &timer );
         cudaThreadSynchronize();
         cutStartTimer( timer );

         square_array <<< n_blokova, blok_velicina >>> (a_g, N);

         cudaThreadSynchronize();
         cutStopTimer( timer );

         cudaMemcpy(a_h, a_g, sizeof(float)*N, cudaMemcpyDeviceToHost);

         for (int i=0; i<N; i++)
              printf("%d %f\n", i, a_h[i]);

         printf("CUDA vrijeme izvodenja = %f ms\n",cutGetTimerValue( timer ));

         free(a_h);
         cudaFree(a_g);
}
```

5.2. Primjer množenja kvadratnih 16×16 matrica

Ovdje će biti prikazan jednostavan CUDA program za množenje kvadratnih matrica A i B dimenzija 16×16. Kao i u prethodnom primjeru, program je strukturiran u dva dijela. Jedan dio koji se izvodi na procesoru opće namjene, a drugi dio se izvodi na grafičkom procesoru. Prvi dio čini funkcija `mnozenjeMatrica()`. Funkcija prima pokazivače na matrice A, B i C prethodno zauzete u memoriji hosta. Kako su matrice A i B kvadratne dimenzija 16×16 i matrica rezultata C biti će istih dimenzija. Unutar funkcije `mnozenjeMatrica()` zauzima se ista količina CUDA memorije grafičkog procesora za sve tri matrice. Za zauzimanje CUDA memorije kao i u prethodnom primjeru 5.1 koristi se CUDA API funkcija `cudaMalloc()`. Također, kopiranje elemenata matrica A i B iz memorije hosta u prethodno zauzetu CUDA memoriju grafičkog procesora omogućava CUDA API funkcija `cudaMemcpy()`. Prije poziva jezgrine funkcije potrebno je odrediti dimenzije grida i bloka dretvi.

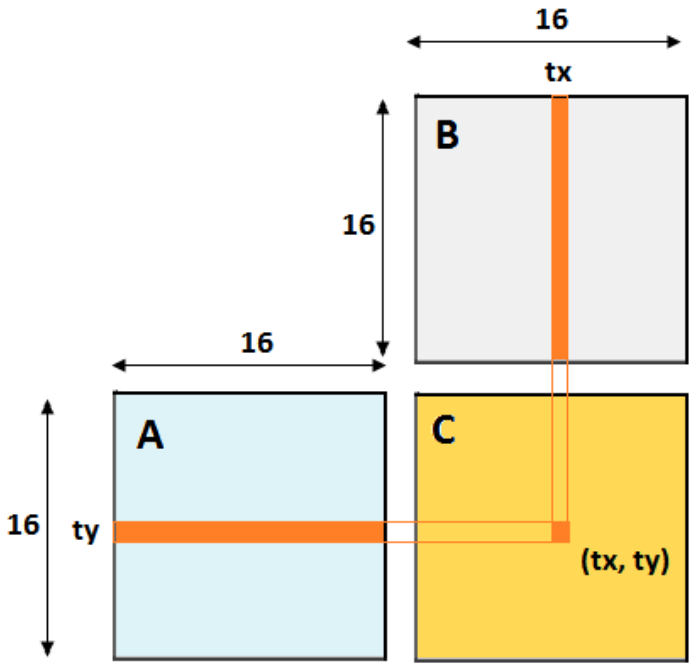
Za izračunavanje jednog elementa (od njih 256) matrice C zadužena je jedna dretva. Zbog toga je za izračun cijele matrice C potrebno 256 dretvi. Svih 256 dretvi će se nalaziti unutar jednog bloka. Kao što se vidi sa slike 8., unutar bloka u smjeru osi x i u smjeru osi y nalazi se točno 16 dretvi. U odjeljku 4.2.4 navedene su moguće dimenzije bloka dretvi. Kako se u ovom primjeru radi se dvodimenzionalnom bloku, prilikom postavljanja dimenzije bloka treća dimenzija biti će jednaka jedinici. Također, sve tri dimenzije grida postavljene su na vrijednost jedan. Razlog tome je to što se sve dretve nalaze unutar točno jednog bloka koji čini grid. Pozivanjem jezgrine funkcije u CUDA memoriji formira se grid prikazan na slici 8.

Prije pozivanja jezgrine funkcije `mnozi`, potrebno je obaviti sinkronizaciju dretvi procesora opće namjene i grafičkog procesora. Detaljniji razlog sinkronizacije objašnjen je u prethodnom primjeru 5.2. Također, da bi se odredilo postignuto ubrzanje izvođenjem na grafičkom procesoru, potrebno je započeti mjerenje vremena. Idući korak je pozivanje jezgrine funkcije `mnozi`. Time se izvođenje privremeno prekida na procesoru opće namjene i započinje se sa izvođenjem na grafičkom procesoru.



Slika 8. Formiran grid unutar CUDA memorije

Kao i u prethodnom primjeru sve dretve paralelno izvode jezgrinu funkciju, odnosno svaka dretva zadužena je za izračunavanje jednog elementa matrice C. Zbog toga je potreban mehanizam pomoću kojeg će se dretve međusobno razlikovati. Koriste se ugrađene varijable `threadIdx.x` i `threadIdx.y` kojima se identificira pojedina dretva. Varijabla `threadIdx.x` sadrži indeks dretve unutar bloka dretvi u smjeru osi x, a `threadIdx.y` označava indeks dretve unutar bloka u smjeru osi y. Radi pojednostavljenog zapisa, varijable će se označavati sa `tx` i `ty`. Tako će dretva s kordinatama `tx` i `ty` odrediti vrijednost elementa matrice C koji se nalazi u retku `ty` i stupcu `tx` (vidi Sliku 9.). Za njegovo izračunavanje dretva dohvaća redom elemente retka `ty` matrice A i elemente stupca `tx` matrice B. Dohvaćeni elementi se množe i pribrajaju rezultatu prethodne iteracije. Nakon što se prođe kroz svih 16 elemenata retka, odnosno stupca u varijabli `rezultatC` pohranjena je vrijednost elemenata matrice C s koordinatama `(tx, ty)`. Prije povratka iz jezgrine funkcije, izračunati rezultat pridružuje se odgovarajućem elementu matrice C pohranjene u CUDA memoriji grafičkog procesora.



Slika 9. Princip izračunavanje elementa matrice C

Nakon povratka iz jezgrine funkcije, završilo je izvođenje na grafičkom procesoru i nastavlja se sa izvođenjem na CPU. Po povratku potrebno je prvo natrag obaviti sinkronizaciju dretvi. Nakon što sigurno sve dretve završe s poslom zaustavlja se mjerenje vremena. Idući korak je kopiranje rezultata, odnosno matrice C iz CUDA memorije u memoriju hosta. Na kraju potrebno je osloboditi prethodno zauzetu CUDA memoriju za sve tri matrice. Implementacija ovog primjera može se vidjeti niže.

Dio programa koji se izvodi na procesoru opće namjene.

```
void mnozenjeMatrica(  float *A,  float *B, float *C, int dimenzija){

	float *AG, *BG, *CG;
	unsigned int timer=0;
	int velicina = dimenzija*dimenzija*sizeof(float);

	dim3 dimBlock(BLOK,BLOK);
	dim3 dimGrid(1, 1);


	cudaMalloc((void **) &AG, velicina);
	cudaMalloc((void **) &BG, velicina);
```

```
	cudaMemcpy(AG, A, velicina, cudaMemcpyHostToDevice);
	cudaMemcpy(BG, B, velicina, cudaMemcpyHostToDevice);

	cudaMalloc((void **) &CG, velicina);

	cutCreateTimer( &timer );
	cudaThreadSynchronize();
	cutStartTimer( timer );

	mnozi <<< dimGrid, dimBlock >>>(AG, BG, CG, dimenzija);

	cudaThreadSynchronize();
	cutStopTimer(timer);

	cudaMemcpy(C, CG, velicina, cudaMemcpyDeviceToHost);
	printf("CUDA vrijeme izvodenja = %f ms\n",cutGetTimerValue( timer ));

	cudaFree(AG);
	cudaFree(BG);
	cudaFree(BG);
}
```

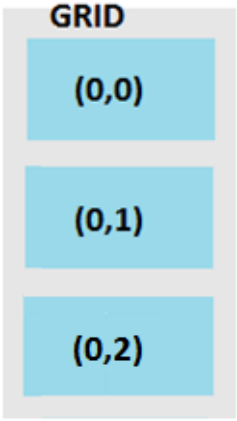
Dio programa koji se izvodi na grafičkom procesoru, takozvana jezgrina funkcija.

```
__global__ void mnozi(float *AG, float *BG, float *CG, int dimenzija){

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

	int k;

	float rezultatC = 0;

	for(k=0; k<dimenzija; k++){
		float Aelement=AG[ty*dimenzija + k];
		float Belement=BG[k*dimenzija + tx];
		rezultatC += Aelement* Belement;
	}

	CG[ty*dimenzija +tx]= rezultatC;

}
```

5.3. Primjer množenja pravokutnih matrica

Množenje kvadratnih matrica iz prethodnog primjera 5.2 ovdje će biti prošireno. U smislu da će se prikazati implementacija CUDA programa koji omogućuje množenje pravokutnih matrica. Jedini preduvjet izvođenja ovog programa je to da širine i visine matrica A i B moraju biti višekratnici broja 16. Također, širina matrice A mora odgovarati visini matrice B. Kao i u prethodnom primjeru program se dijeli na dva dijela. Dio programa koji se izvodi na procesoru opće namjene gotovo je identičan funkciji `mnozenjeMatrica()` iz primjera 5.2, te se ovdje neće ponovno razmatrati. Razlika je u tome što su sada matrice različitih dimenzija, te je potrebno pojedinačno izračunati veličinu matrica A, B i C. Ovisno o izračunatim veličinama zauzima se CUDA memorija grafičkog procesora. Iduću razliku predstavlja poziv jezgrine funkcije.

Cilj ove implementacije je mogućnost množenja matrica različitih dimenzija i sa velikom količinom podataka. Princip izvođenja množenja na grafičkom procesoru ostaje isti, ali je sama izvedba nešto kompliciranija. Matrice sada sadrže mnogo više elemenata, te će biti potrebno i više dretvi. To povlači i postojanje više blokova unutar grida. Svaki blok (kao i u prethodnom primjeru 5.2) će se sastojat od 256 dretvi. U smjeru osi x nalazit će se 16 dretvi i u smjeru osi y također 16 dretvi (vidi sliku 8.). Dimenzija grida ovisi o visini matrice A i širini matrice B. Broj blokova unutar grida u smjeru osi x jednak je broju koji govori koliko se blokova širine 16 može staviti jedan do drugog sve dok se ne postigne širina matrice B. Isto tako, broj blokova unutar grida u smjeru osi y jednak je broju blokova visine 16 koji dodani jedan do drugog čine visinu matrice A. Na primjer, neka je matrica A dimenzija 48×32, a matrica B dimenzija 32×16. Tada će se u CUDA memoriji formirati grid dimenzija 1×3, prikazan na slici 10.



Slika 10. Formiran grid za dimenzije matrica A 48×32 i B 32×16

Parametri jezgrine funkcije su pokazivači na matrice A, B i C unutar globalne memorije, te širina matrice A i širina matrice B. Svaka dretva paralelno izvodi kod jezgrine funkcije. Zbog toga je potreban mehanizam pomoću kojeg će se dretve razlikovati. Kao i u prethodim primjerima, koriste se ugrađene varijable `threadIdx.x` i `threadIdx.y` kojima se identificira pojedina dretva. Uz indeks dretve potrebno je odrediti i indeks pojedinog bloka unutar grida. Ugrađene varijable `blockIdx.x` i `blockIdx.y` svakom bloku dretvi unutar grida dodjeljuju jedinstvene koordinate. Kao i kod dretvi, skraćeno se označavaju `bx` i `by`.

Svaka dretva zadužena je za izračunavanje jednog elementa matrice C. Kako bi obavila izračunavanje pojedinog elementa dretva mora pristupiti svim elementima određenog retka matice A i svima elementima određenog stupca matrice B, kao što je prikazano na slici 9. Znači dretva će čitati elemente matrice A iz globalne memorije toliko puta kolika je širina matrice B. Isto tako da bi pročitala potrebne elemente matice B, broj pristupa globalnoj memoriji jednak je visini matrice A. Potrebno je napomenuti da sve dretve unutra bloka s istom x koordinatom pristupaju istom retku matrice A. Također, sve dretve s istom y koordinatom pristupaju istom stupcu matrice B. Pristupanje globalnoj memoriji je vrlo sporo i zbog tog razloga traži se drugačije rješenje. Glavna ideja je dohvaćanje potrebnih elemenata matrica A i B u zajedničku memoriju. Pristupanje zajedničkoj memoriji je daleko brže od pristupanja globalnoj memoriji, ali s druge strane zajednička memorija je vrlo mala. U slučajevima kad su matrice A i B vrlo velikih dimenzija nije moguće dohvatiti sve potrebne elemente u zajedničku memoriju. Rješenje ovog problema predstavlja dohvaćanje manjih skupina elementa matice A i matrice B. Zbog pojednostavljivanja implementacije dimenzije

skupine elementa jednake su dimenzijama bloka. Tako će u ovom primjeru njihove dimenzije biti jednake 16×16 i takve skupine elemenata u daljnjem tekstu zvat ćemo kvadratnim podmaticama. Ovisno o dimenzijama matrica A i B, unutar širine matrice A i visine matrice B nalazit će se nekoliko kvadratnih podmatica. Dretva na temelju koordinate `blockIdx.x` određuje niz od 16 redaka matrice A koji će se redom u obliku kvadratnih podmatrica dohvaćati u zajedničku memoriju. Isto tako koordinata `blockIdx.y` određuje slijed od 16 stupaca matrice B koji će se u obliku kvadratnih matrica dohvaćati u zajedničku memoriju. Kvadratne podmatrice dohvaćaju se u prethodno alociranu zajedničku memoriju na sljedeći način. Svaka dretva unutar bloka dohvaća jedan element kvadratnih podmatrica A i B u zajedničku memoriju. Element se sprema unutar dvodimenzionalnog polja na poziciju određenu tx i ty koordinatama dretve. Nakon dohvaćanja svih elemenata potrebno je obaviti sinkronizaciju kako bi se osiguralo da su zaista dohvaćeni svi potrebni elementi matrica A i B. Idući korak je množenje elemenata retka podmatice A određenog koordinatom ty sa pripadnim elemetima stupca podmatice B. Stupac podmatrice B određen je tx koordinatom dretve. Znači, u svakom koraku dohvaća se jedan element retka podmatrice A i njemu odgovarajući element stupca podmatrice B. Dohvaćeni elementi se množe i pribrajaju rezultatu prethodne iteracije. Nakon prolaska po svim elementima retka, odnosno stupca kvadratne podmatice unutar varijable `rezultatC` pohranjena je međurezultat jednog elementa matrice C. Za izračun konačne vrijednosti pojedinog elementa matrice C potrebno je ponoviti postupak za sve podmatrice koje čine širinu matrice, odnosno visinu matrice B. Nakon prolaska po svim podmatricama unutar varijable `rezultatC` nalazi se konačna vrijednost elementa matrice C. O kojem se elementu matrice C radi, ovisi o koordinatama bloka i koordinatama dretve unutar tog bloka koja je obavila izračunavanje. Element se pohranjuje na točno određenu poziciju prethodno zauzetog jednodimenzionalnog polja unutar globalne memorije GPU-a. U nastavku je prikazana implementacija ovakve jezgrine funkcije.

```
__global__ void mnozi(float *AG, float *BG, float *CG, int sirinaA, int sirinaB){

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

      int k;
```

```
	int bx = blockIdx.x;
	int by = blockIdx.y;

	int pocetakA = sirinaA*BLOK*by;
	int korakA=BLOK;
	int krajA = pocetakA + sirinaA -1;

	int pocetakB = BLOK*bx;
	int korakB = BLOK*sirinaB;

	float rezultatC = 0.0;



	for( int i=pocetakA, j=pocetakB; i<krajA; i+=korakA, j+=korakB){
		__shared__ float AS[BLOK][BLOK];
		__shared__ float BS[BLOK][BLOK];

		AS[ty][tx]=AG[i + sirinaA*ty + tx];
		BS[ty][tx]=BG[j + sirinaB*ty + tx];

		__syncthreads();

		for(k=0; k<BLOK; ++k){
			rezultatC +=AS[ty][k]*BS[k][tx];
			}
		__syncthreads();
	}
	int c=sirinaB*BLOK*by + BLOK*bx;
	CG[c+sirinaB*ty+tx]=rezultatC;
}
```

5.4. Rezultati eksperimentalne evaluacije

U ovom poglavlju će biti prikazani rezultati za sva tri prethodno navedena CUDA programa. Analizirat će se dobiveni rezultati i objasnit zašto neki od njih ne daju očekivano ubrzanje. Mjerenje potrebnog vremena za izvođenje sva tri navedena CUDA primjera provedeno je na grafičkoj kartici GeForce GT 240 tvrtke NVIDIA. Vrijeme izvođenja programa C-u provedeno je na Dual CPU T2410.

5.4.1. Kvadriranje elemenata polja

Potrebno je naglasiti da prvenstvena namjena ovog jednostavnog programa bila je pokazati strukturu i način korištenje osnovnih CUDA API funkcija. Što znači da naglasak u ovom primjeru nije isključivo na ubrzanju izvođenja. Zbog tog razloga, dobivaju se rezultati izvođenja CUDA programa koji ne pokazuju ubrzanje izvođenja u odnosu na verziju C.

Tablica 1. Prosječno vrijeme izvođenja CUDA i C programa

	CUDA	C
PROSJEČNO	0,069 ms	0.019 ms

U tablici 1. prikazano je prosječno vrijeme dobiveno kroz pet mjerenja unutra 10 0000 poziva u Release načinu rada. Release način rada uključuje optimizacije. Kao što se vidi izvođenje CUDA programa je daleko sporije od izvođenje programa u C-u. Jedan od razloga je to što se program u C-u sastoji od dvije for petlje. Prolazak kroz dvije ugniježđene for petlje programa C je vrlo brz i kroz 100000 poziva iznosi u prosjeku samo 0.019ms. Idući razlog je sinkronizacija dretvi u CUDA programu. Kao što je opisano u poglavlju **4.2.4**, sinkronizacija dretvi je nužna kako bi se osiguralo da su završili svi zadaci prije povratka na nastavak izvođenja u procesoru opće namjene.

5.4.2. Množenje matrica 16×16

Ovdje će biti prikazani rezultati izvođenja CUDA programa kojim je implementirano množenje matrica dimenzija 16×16. Prikazano je prosječno vrijeme dobiveno tijekom pet mjerenja kroz 10 000 poziva.

Tablica 2. Prosječno vrijeme izvođenja CUDA i C programa

	CUDA	C
PROSJEČNO	520,18 ms	496,1 ms

Kao što se vidi iz tablice 2., množenje matrica u C-u je brže nego što to radi CUDA program. Postavlja se pitanje, zbog čega je izvođenje CUDA programa sporije od izvođenja programa u C-u. Razlog je mala količina podataka koja se obrađuje i nužna sinkronizacija dretvi. Tablica 3. prikazuje rezultate množenja matrica dimenzija 16×16 u CUDI bez korištenja sinkronizacije dretvi prije zaustavljanja mjerenja vremena. Rezultati predstavljaju prosječno vrijeme pet mjerenja tijekom 1000 i 10 000 poziva.

Tablica 3. Prosječno vrijeme izvođenja CUDA programa

	10 000 poziva	1000 poziva
PROSJEČNO	146,67 ms	17,34 ms

5.4.3. Množenje pravokutnih matrica

Implementiran primjer množenja matrica A i B (objašnjen u odjeljku 5.3) testiran je na matricama A dimenzije 48×32 i matrici B dimenzije 32×16. Za razliku od prethodnog primjera ni u ovom slučaju ne radi se o velikoj količini podataka. No, kao što se vidi iz tablice 3. Izvođenje CUDA programa u odnosu na prethodne primjere daje daleko bolje rezultate. Mjerenje se obavlja tijekom 1000 poziva.

Tablica 3. Prosječno vrijeme izvođenja CUDA i C programa

	CUDA	C
PROSJEČNO	65,47 ms	338,7 ms

Na temelju ovih rezultata može se potvrditi kako su CUDA programi namijenjeni samo za obrađivanje velikog broja elemenata. Znači u slučajevima gdje podatkovni paralelizam dolazi do izražaja u punom smislu. Za male količine računanja kao što je to u primjeru kvadriranje polja, CUDA program nije efikasan s obzirom na brzinu. Izvođenje matematičkih operacija, kao što su u ovom slučaju množenje i zbrajanje mnogo je brže na grafičkim procesorima. No, cjelokupno vrijeme izvođenje uključuje i sinkronizaciju dretvi, a time se gubi na vremenu. Dokaz toga je tablica 4. u kojoj se može vidjeti vrijeme izvođenje CUDA programa bez sinkronizacije dretvi prije zaustavljanja mjerenja tijekom višestrukih poziva. Odnosno, sinkronizacija se obavlja tek nakon zadnjeg poziva. Prikazani su rezultati mjerenja vremena izvođenja tijekom 1000 i 10 000 poziva.

Tablica 3. Prosječno vrijeme izvođenja CUDA programa

	10 000 poziva	1000 poziva
PROSJEČNO	112,35 ms	12,9 ms

6. Budući razvoj i CPU+GPU implementacije

Trenutno se općenamjensko programiranje grafičkih procesora koristi u specijalne svrhe. Pri tome se misli na istraživanja u znanosti ili specijalno za neko područje, kao što je na primjer probijanje kriptografskih sustava. Potrebno je naglasiti da običan, prosječan korisnik nema baš previše koristi od ovakvog pristupa. S druge strane, proizvođači nastoje sve nove tehnologije dovesti do prosječnog korisnika. Danas to predstavlja najvažniji uvjet za uspjeh na tržištu. Tako se nastoji dovesti GPGPU običnom korisniku, a da on toga nije ni svjestan. Početak takvog pristupa danas već postoji - h264 filmovi visoke kvalitete dekodiraju se na grafičkom procesoru. Rezultat toga je minorno (jedva 10%) opterećenje procesora opće namjene. Međutim, teži se još boljem iskorištavanju snage grafičkog procesora. Trenutno je u razvoju Visual Studio 2010 i Internet Explorer 9 koji će se izvodit na grafičkom procesoru. Što će rezultirati brzim sučeljem, bez potrebe GPGPU okruženja kao što su npr. CUDA i OpenCL.

U današnjim računalima grafički procesor je odvojen od procesora opće namjene, odnosno čini zasebnu cjelinu. Zbog toga, mnogo ovisi o vrsti grafičkog procesora. Korisnik treba kupiti određenu grafičku karticu i baš za nju koristit predviđeno GPGPU okruženje ili biti siguran da ima zahtijevanu podršku kao što je npr. dekodiranje videa.

Prije svega tri godine, tvrtka AMD kupuje ATI. Dublji cilj te kupovine je dovesti na tržište integrirani sustav koji se sastoji od procesora opće namjene i grafičkog procesora. Nešto slično tome trenutno na tržištu nudi tvrtka Intel (Core i5 560). Na istom čipu nalazi se grafički procesor i procesor opće namjene. Time se postiže dodatna grafička snaga koja se može iskoristiti općenamjenskim programiranjem. Još uvijek se za to ne mogu naći programski razvoji alati. No, najavljen je izlazak upravljačkog programa (engl. driver) koji će omogućiti izvođenje aplikacija na grafičkim procesorima, kao npr. dekodiranje *h264* videa visoke rezolucije (1080p). Izlazak programskog razvojnog alata omogućit će korisniku programiranje aplikacija bez da razmišlja o vrsti grafičke kartice. Sama činjenica da se grafički procesor nalazi unutar procesora opće namjene omogućit će korisniku izravno iskorištavanje snage grafičkog procesora. Postavlja se pitanje što će biti sljedeći korak. Da li će to biti korištenje grafičkog sustava za izvođenje cjelokupnog operacijskog sustava? Možda u budućnosti, no

trenutno to ipak predstavlja samo nove ideje za što bolje iskorištavanje snage grafičkog procesora.

7. Zaključak

Izvođenje matematičkih zahtjevnijih operacija na velikom skupu podataka daleko je brže na grafičkim procesorima. Brže izvođenje rezultat je drugačijeg pristupa u dizajniranju arhitekture u odnosu na procesore opće namjene. Jezgre procesora opće namjene od samog početka dizajnirane su da slijedno izvode niz instrukcija maksimalnom brzinom. Za razliku od toga, jezgre grafičkog procesora istovremeno izvode različite instrukcije na različitim podacima. Time se postiže paralelna obrada podataka koja rezultira ubrzanjem izvođenja.

Prvih pokušaji programiranja grafičkih procesora zahtijevali su strukturiranje programa u odnosu na grafičku protočnu strukturu. Današnji pristup općenamjenskog programiranja omogućuje izravno pristupanja jednoj programirljivoj sklopovskoj jedinici.

Glavni dio seminarskog rada temelji se na analizi programskog okruženja CUDA. U sklopu tog okruženja implementirana su tri jednostavna primjera. Prvi primjer, kvadriranje elementa polja odabran je kako bi se pokazala osnovna struktura CUDA programa i korištenje osnovnih CUDA API funkcija. Zbog tog razloga, rezultati testiranja ovog primjera ne pokazuju ubrzanje izvođenja. Kako bi se postiglo željeno ubrzanje izvođenja što je i glavni smisao postojanja programskog okruženja CUDA, potrebno je obrađivati veću količinu podataka gdje podatkovni paralelizam dolazi do izražaja. U primjeru množenja pravokutnih matrica obrađuje se veća količina elementa i time dolazi do ubrzanja izvođenja. Vrlo mnogo vremena gubi se na sinkronizaciju dretvi koje zahtijeva programiranje u CUDI. Sinkronizacija dretvi osigurava da se završe svi zadaci koji se izvode na grafičkom procesoru prije nego se nastavi s izvođenjem na procesoru opće namjene.

Obzirom na performanse i potencijale koje nude moderni grafički procesori, općenamjensko programiranje trenutno se nalazi u rapidnom razvoju. Stoga se može reći da će upravo ovo područje imat ključnu ulogu u razvoju budućih računalnih sustava.

8. Literatura

[1] D. Kirk i W. Hwu, Chapter 1: Introduction, CUDA Textbook, IAP09 CUDA@MIT (6.963), 2006-2008.

http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook, 2.3.2010.

[2] D. Kirk i W. Hwu, Chapter 2: CUDA Programming Model, CUDA Textbook, IAP09 CUDA@MIT (6.963), 2006-2008.

http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook, 2.3.210.

[3] D. Kirk i W. Hwu, Chapter 3: CUDA Threading Model , CUDA Textbook, IAP09 CUDA@MIT (6.963), 2006-2008.

http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook, 10.3.210.

[4] D. Kirk i W. Hwu, Chapter 4: CUDA Memory Model , CUDA Textbook, IAP09 CUDA@MIT (6.963), 2006-2008.

http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook, 10.3.210.

[5] NVIDIA Corporation, CUDA Programming Guide Version 2.3,

http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf, 18.4.2010.

[6] GPGPU: Category: Research,

http://gpgpu.org/category/research, 10.3.2010.

[7] J. D. Owens, M. Houston, D. Luebke, S. Green,J. E. Stone, i J. C. Phillips, GPU Computing, University of California-Santa Barbara, 2008.

http://www.uweb.ucsb.edu/~yichuwang/ecv/paper/gpu_computing.pdf, 17.4.2010.

[8] H. Nguyen, GPU Gems 3, Part VI: GPU Computing, 2007.

[9] OpenCL, Wikipedia, the free encyclopedia

http://en.wikipedia.org/wiki/OpenCL , 16.4.2010.

[10] A. Berillo, Non-graphic computing with graphics processors, CUDA history, 2008. http://ixbtlabs.com/articles3/video/cuda-1-p4.html, 16.4.2010.

[11] I. Kulišić, Animacija tijela korištenjem programabilne grafičke kartice, Diplomski rad, Fakultet elektrotehnike i računarstva, 2007. http://hotlab.tel.fer.hr/student_projects/diplomski-Kulisic.pdf, 17.4.2010.

[12] S. N. Sinha, J. M Frahm, M. Pollefeys i Y. Genc, Feature tracking and matching in video using programmable graphics hardware, University of North Carolina at Chapel Hill, 2007.

[13] C. Wu, A GPU Implementation of Scale Invariant Feature Transform, University of North Carolina at Chapel Hill, 2010. http://www.cs.unc.edu/~ccwu/siftgpu/, 28.4.2010.

[14] Scalable Link Interface, Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Scalable_Link_Interface, 20.4.2010.

[15] NVIDIA, Next generation CUDA architecture, Fermi http://www.nvidia.com/object/fermi_architecture.html, 18.4.2010.

[16] H. Ribičić, Korištenje programirljivih grafičkih procesora u implementaciji fizikalno temeljenih modela, Seminar, Fakultet elektrotehnike i računarstva, 2009.

9. Sažetak

U ovom radu razmatraju se dostupne okoline za izvršavanje općenamjenskih računalnih operacija na grafičkim procesorima (engl. General-Purpose Computation on Graphics Hardware, GPGPU). Naglasak je na programskom okruženju CUDA.

Prikazano je nekoliko jednostavnih primjera programiranja u CUDI. Struktura svakog CUDA programa sastoji se od dva dijela. Prvi dio predstavlja izvođenje programa na procesoru opće namjene. Radi se o dijelu programa implementiranom u standardnom jeziku C. Drugi dio čini jezgrina funkcija, čijim se pozivom izvođenje nastavlja na grafičkom procesoru. Taj dio programa implementiran je djelom u jeziku C, a preostali dio čine CUDA API funkcije. Osnovne CUDA API funkcije koje mora posjedovati svaki CUDA program odnose se na prijenos potrebnih podataka za obradu iz memorije hosta u prethodno zauzetu memoriju grafičkog procesora. Zatim poziv jezgrine funkcije unutar koje se obavlja obrada podataka. Zadnji korak predstavlja spremanje rezultata iz memorije grafičkog procesora u memoriju hosta i oslobađanje prethodno zauzete memorije grafičkog procesora.

CUDA programi namijenjeni su za izvođenje složenijih matematičkih operacija nad velikim skupom podataka. Taj pristup ilustriran je primjerom množenja pravokutnih matrica. Matrice su podijeljene na odgovarajući broj podmatrica koje se obrađuju istovremeno. Na taj način postiže se ubrzanje obrade podataka.