Univerzitet u Beogradu
Matematički fakultet

Programiranje grafičkih uređaja: platforma CUDA

student: Stevan Prodanović
broj indeksa: 1024/2012
predmet: Metodologija stručnog i naučnog rada
školska godina: 2012/2013
nastavnik: dr Vladimir Filipović

Sadržaj

1. Uvod

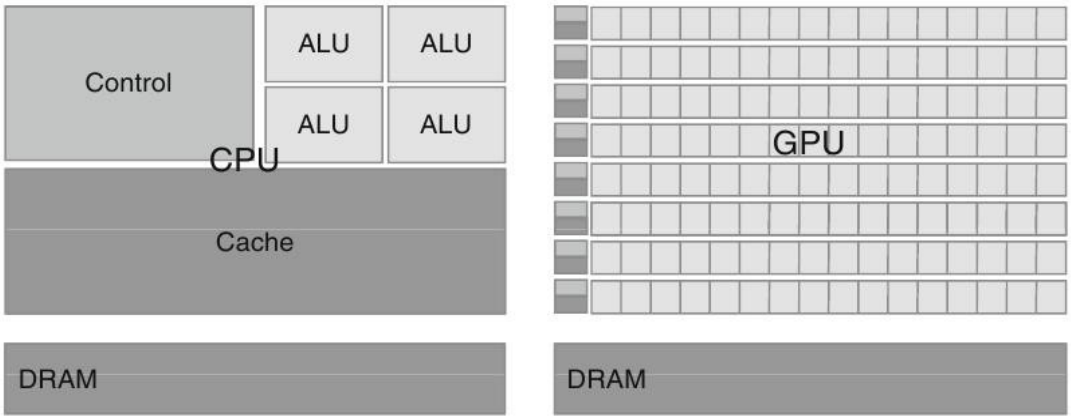
"Broj tranzistora koji se može smestiti na kvadratni inč silicijuma se duplira svakih 18-24 meseca."
Murov zakon, Gordon Mur

Počev od 1965. godine kada je Gordon Mur u časopisu *Electronic* objavio svoja predviđanja pa sve do početka 2000-tih poboljšanje performansi računara zasnivalo se mahom na procesu minimizacije tj. sabijanja sve većeg broj tranzistora po jedinici površine čipa. Međutim u jednom trenutku se došlo do granice fizičkih mogućnosti minijaturizacije te se, da bi se nastavilo sa daljim ubrzavanjem računara, morao promeniti pristup. Jedno od rešenja ovog problema je proizvodnja čipova sa više procesorskih jedinica. Razvoj ovakvih mikroprocesora kretao se u dva bitno različita smera:

1. **Multicore** arhitektura
2. **Manycore** arhitektura

Multicore se sastoji od više izvršnih jezgara (danas u svakodnevnoj upotrebi najčešće arhitekture sa 2, 4, 6 ili 8) na istom čipu koji su povezani lokalnom magistralom. Osnovni cilj je brže izvršavanje sekvecijalnih programa. Ovo se postiže pomoću kompleksne kontrolne logike koja omogućava da se instrukcije jedne niti izvršavaju paralelno ili čak van svog zadatog radosleda čuvajući pritom sekvencijalnu prirodu programa. Međutim problemi se javljaju pri dodavanju većeg broja jezgara, jer tada lako dolazi do zakrčenja na magistrali preko koje se obavlja sva komunikacija i tada cela priča polako počinje da gubi smisao.

Sa druge strane glavna ideja **manycore**-a podrazumeva broj jezgara dovoljno veliki da se tradicionalne multiprocesorske tehnike više ne mogu koristiti. Umesto da su sva međusobno povezana, kod manycore arhitekture jezgra su organizovana u grupe od po osam i ona dele zajedničku upravljačku jedinicu i instrukcioni keš. Glavna ideja kod ove arhitekture je da se poveća propusnost (broj instrukcija koje se mogu izvršiti u jedinici vremena) kako bi se omogućilo 'masivno' izvršavanje niti kod paralelnih aplikacija što se ostvaruje maksimizacijom površine čipa koja se koristi za izračunavanja. Jedna vrsta manycore-a su i grafički procesori (GPU – Graphics Processing Unit) koji su i tema ovog rada, odnosno jedan od primera ovakvih uređaja, a to su NVIDIA-ini uređaji koji podržavaju CUDA platfomu.



Slika 1: Multicore i manycore arhitektura

2. CUDA

Iako na prvi pogled grafički procesori deluju kao idealno rešenje za buduća ubrzanja apliakcija na tom putu su postojale određene prepreke i problemi.

Prvi problem je taj što su GPU uređaji namenjeni za numerička izračunavanja i kao takvi ne mogu biti konkurentni sa CPU po pitanju izvršavanja sekvencijalnih programa. Da bi se ovo rešilo potrebno bi bilo da možemo da pišemo programe koji će se delimično izvršavati na GPU, a delimično na CPU. Do skora je problem predstavljala i činjenica da su sva izračunavanja vršena sa jednostrukom preciznošću pa GPU uređaji nisu bili pogodni za aplikacije koje zahtevaju rad sa brojevima sa dvostrukom preciznošću. Ali noviji uređaji ne samo da podržavaju račun u dvostrukoj tačnosti već su po brzini ovih operacija prestigli CPU-ove.

Drugi problem je taj što je pisanje programa koji će raditi na grafičkim procesorima zahtevalo upotrebu funkcija grafičkih aplikativnih programskih interfejsa (kakav je npr. OpenGL) kako bi se pristupilo jezgrima GPU-a, te nalaženje načina da se algoritmi koje koristite predstave kroz geometrijske objekte sa kojima jedino umeju da rade ovakvi interfejsi. A to bi bilo kao da od umetnika tražite da vam izradi repliku impresionističke slike u vidu mozaika, ali koristeći samo alat iz automehaničarske radionice. Ovo je u velikoj meri ograničavalo spektar aplikacija koje su mogle biti pisane.

NVIDIA je prepoznala ove probleme i 2007. godine ponudila način da se oni prevaziđu u vidu CUDA (*Compute Unified Device Architecture*) programerskog modela, koji sa programerske strane predstavlja ništa drugo do proširenje viših programskih jezika (kao što su npr. C/C++, Fortran, Java, Python, Haskell i sl.). Ovo proširenje nam omogućava da deo koda koji će se izvršavati na CPU pišemo u odabranom višem programskom jeziku i prevodimo odabranim prevodiocem za taj jezik, a da za deo koda koji će se izvršavati na GPU, tzv. kernel funkcije, koristimo dodatne sintaksne strukture i prevodimo pomoću posebnog NVIDIA-inog kompajlera. Naravno CUDA ne predstavlja samo promenu u softveru već i u hardveru. CUDA programi više ne koriste grafički interfejs već na raspolaganju imaju interfejs za paralelno programiranje opšte namene koji ih uslužuje.

Još jedna jako bitna osobina CUDA-e je i skalbilnost. Na početku je rečeno da GPU može imati 256, 512, 1024 itd. izvršnih jezgara, međutim onda se postavlja pitanje da li CUDA program pisan za jednu konfiguraciju može da se pokrene i na ostalima. I odgovor je potvrdan, a u njegovoj osnovi leže tri ključne apstrakcije modela:

1. hijerarhijska organizacija niti,
2. deljena memorija i
3. barijerna sinhronizacija.

O svima će biti reči u nastavku.

3. Arhitektura i organizacija niti

Tipičan grafički procesor na kome je moguće izvršavati CUDA programe organizovan je kao niz tzv. Striming multiprocesora (u daljem tekstu **SM**). SM-ovi formiraju blokove, a njihov broj u jednom bloku varira od generacije do generacije CUDA uređaja. Svaki SM sastoji se od po 8 Striming procesora (**SP**), koji dele kontrolnu logiku i instrukcioni keš. Ovim se pristup kešu lokalizuje što za posledicu ima veću komunikacionu propusnost što se pokazuje kao bitan faktor u poboljšanju performansi. Svaki SP ima jedinicu za množenje i sabiranje (multiply-add -MAD) i dodatnu jedinicu za množenje kao i dodatne jedinice za specijalne funkcije sa brojevima u pokretnom zarezu kao što su kvadratni koren ili transcedentne funkcije. Svaki SP može pokrenuti hiljade niti po aplikaciji. Primera radi uređaj sa oznakom G80 ima 128 SP-ova (16 SM-ova, svaki po 8 SP-ova) i podržava 768 niti po SM-u. Ovo sve zajedno čini preko 12000 niti po čipu.

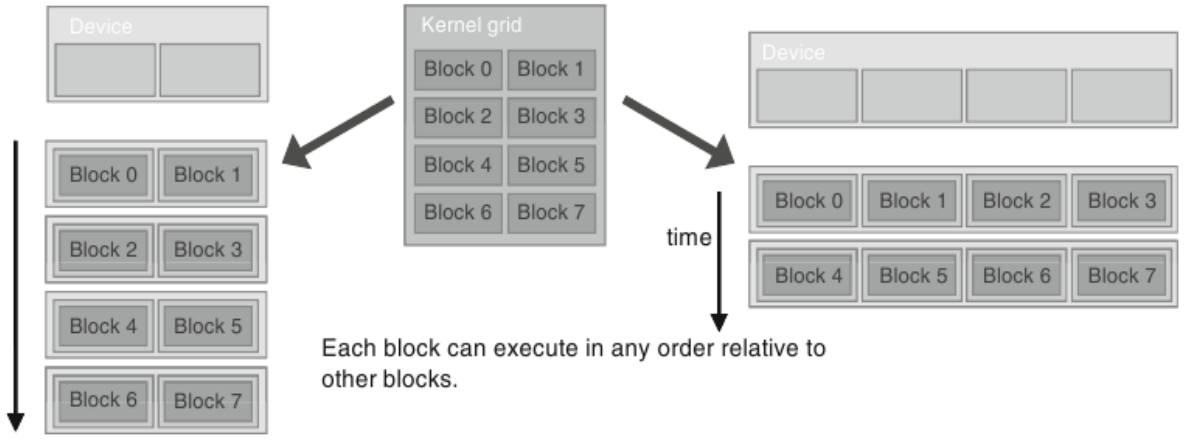
Kao što je već rečeno niti su **hijerarhijski organizovane**. Najviši nivo hijerarhije je mreža niti koja predstavlja trodimenzioni niz blokova niti koji su organizovani kao trodimenzioni niz niti. Kako bi se niti mogle međusobno razlikovati i identifikovati svaka nit je jedinstveno određena svojim koordinatama u okviru bloka, odnosno mreže niti. Ove koordinate čuvaju se u ugrađenim promenljivima **blockIdx** (indeks bloka u okviru mreže) i **threadIdx** (indeks niti u okviru odgovarajućeg bloka) i njima se može pristupiti iz kernel funkcije. Kao parametri kernel funkcije moraju se zadati dimenzije mreže i bloka. One se čuvaju u ugrađenim promenljivima **gridDim** i **blockDim** (trenutna arhitektura podržava blokove veličine 1024 niti) čije se vrednosti ne mogu promeniti nakon pokretanja kernel funkcije. Kada su nam poznate sve ove vrednosti lako možemo jedinstveno identifikovati svaku nit. Uzmimo za primer da su nam blokovi i mreže jednodimenzioni. Tada je:

threadID = blockIdx.x * blockidx.x + threadIdx.x

Svi ovi parametri su tipa **dim3** – koji je, primera radi, u C predstavljen kao struktura sa tri unsigned integer vrednosti x, y i z. Razlog zbog koga nam je važno da postoji mogućnost jedinstvenog identifikovanja svake niti ponaosob je taj da može da se odredi skup podskup skupa podataka nad kojim treba da operiše konkretna nit.

Ranije je pomenut pojam **barijerne sinhronizacije**. Šta je zapravo u pitanju i koja je svrha toga? CUDA omogućava nitima iz istog bloka da koordinišu svoje aktivnosti koristeći funkciju za sinhronizaciju **_syncthreads()**. Kada kernel funkcija pozove _syncthreads(), niti koje izvršavaju datu kernel funkciju biće zadržane na mestu poziva _syncthreads() - barijeri, sve dok svaka nit unutar bloka ne dostigne tu lokaciju. Ovo omogućava da sve niti u bloku završe prvu fazu izvršavanja kernela, pre nego što se prebace na sledeću. Poenta barijerne sinhronizacije je da se blokovi niti mogu izvršavati u bilo kom redosledu, jer nije dozvoljeno nitima iz različitih blokova da se međusobno sinhronišu.

Ovakva implementacija omogućava skalabilnost. Naime, kako se izvršni resursi dodeljuju svim nitima u bloku zajedno to ako postoji nedostatak bilo kog resursa CUDA izvršno okruženje može automatski da redukuje broj blokova predviđenih za dodeljivanje svakom SM-u i tako bez problema nastavi izvršavanje i na slabijem procesoru.



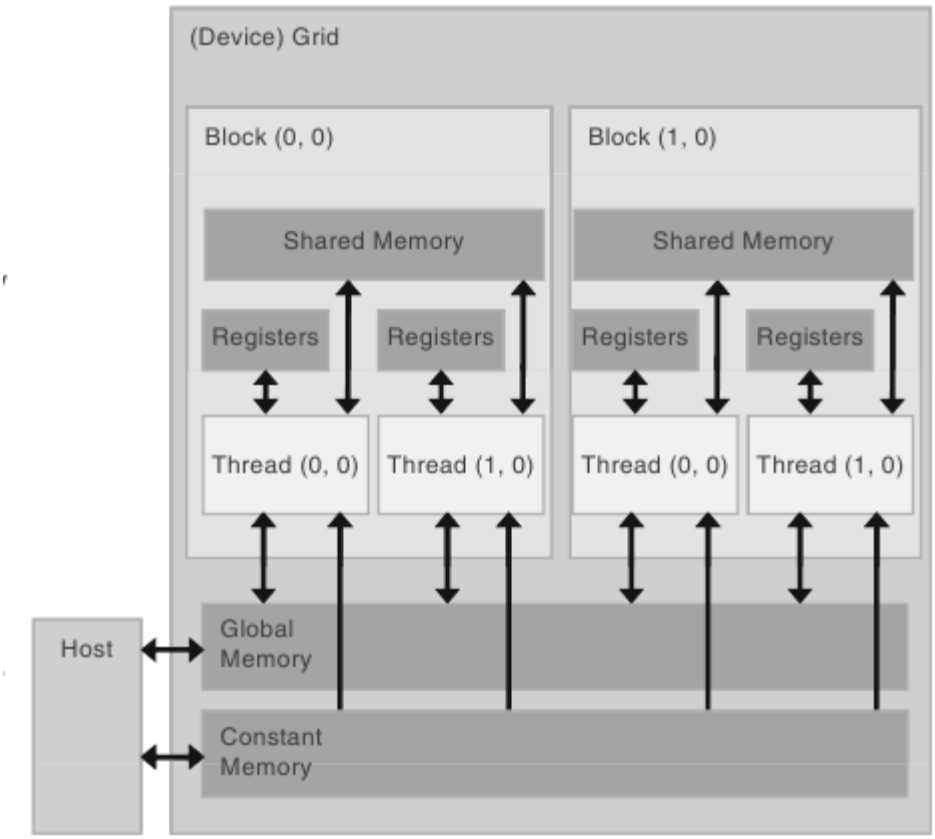
Slika 2: Različit redosled izvršavanja blokova na različitim procesorima

4. Memorije

Dakle, kako se zapravo odvija komunikacija između host programa i kernel funkcija. Svi ulazni podaci moraju se čitati iz host koda. Zatim se oni prebacuju na memoriju grafičkog uređaja, tamo se obrađuju i zatim se rezultati vraćaju nazad na host. Memorija na GPU koja se korisit u ovom procesu označena je kao **globalna memorija**. Međutim ako se u toku izračunavanja koristi samo ova memorija onda to može predstavljati "usko grlo". Uvedimo za početak pojam CGMA odnosa. CGMA (Compute to Global Memory Access) odnos predstavlja odnos broja pristupa globalnoj memoriji u odnosu na broj računskih operacija sa brojevima u pokretnom zarezu koje se izvode. Uzmimo opet za primer NVIDIA G80 čip koji podržava 86.4 GB/s propusnosti globalne memorije. Sa 4 bajta po podatku (float), može se očekivati učitavanje ne više od 21.6 (86.4/4) giga podataka jednostruke preciznosti po sekundi. Ukoliko koristimo samo globalnu memoriju i imamo CGMA odnos 1:1 to znači da će kernel postići brzinu ne veću od 21.6 Gflops-a, što na prvi pogled zvuči veoma dobro, ali je ipak veoma daleko od 367 Gflops-a koliko maksimalno može da postigne G80 uređaj. Dakle kako bi se poboljšale performanse kernela mora se poboljšati CGMA odnos.

Kako bi ovo bilo moguće CUDA model uvodi, pored globalne, još nekoliko vrsta memorije: registri, lokalna, deljena, konstanta i teksturna memorija. **Registri** predstavljaju najbrži vid memorije i do 150 puta je brža od globalne. Svaka nit ima posebnu registarsku memoriju kojoj samo ona može da pristupi. Sve automatske skalarne promenljive (osim nizova) deklarisane u kernelu su smeštene u registre. Kada je kernel funkcija deklariše, privatna kopija te promenljive se generiše za svaku nit koja izvršava kernel funkciju. Pristupanje ovim promenljivima je veoma brzo i paralelno, ali moramo biti pažljivi da ne pređemo ograničeni kapacitet registara, jer tada kompajler može prebaciti neke promenljive na lokalnu memoriju što dovodi do neočekivanog pada performansi. Automatski nizovi čuvaju se u **lokalnoj memoriji**, koja je zapravo deo globalne memorije, ali izdeljen na delove koji pripadaju pojedinačnim nitima koje jedine mogu da im pristupaju, te se tako za svaku nit kreira lokalna kopija niza. Kada nit završi sa izvršavanjem koda, sve njene automatske promenljive prestaju da postoje. Promenljive se mogu eksplicitno deklarisati kao lokalne upotrebom ključne reči **__local__** ispred deklaracije

promenljive. **Deljena memorija** je vidljiva za sve niti jednog bloka i moguće joj je pristupiti za vreme izvršavanja kernela i obično se koristi da sačuva deo podataka (sa globalne memorije) koji se često koriste u izvršnoj fazi kernela. Za deklarisanje neke promenljive kao deljene koristi se ključna reč **__shared__.** Deljene promenljive su efikasno sredstvo koje omogućava da niti unutar istog bloka sarađuju jedna sa drugom. Pristup je veoma brz i izuzetno paralelan. **Konstantna memorija** je deo globalne memorije, ali sa keširanim pristupom što je čini mnogo efikasnijom od ostatka globalne memorije. Sa stanovišta kernela u pitanju je memorija samo za čitanje. Prosotor za promenljivu se rezerviše u konstantnoj memoriji korišćenjem ključne reči **__constant__** s tim da deklaracija mora biti izvan bilo kog tela funkcije. Oblast važenja konstantne promenljive su sve mreže (sve niti u svim mrežama vide istu verziju konstantne promenljive) i može joj se pristupiti tokom celog trajanja aplikacije. Poslednja vrsta memorije jeste **teksturna memorija** koja takođe predstavlja deo globalne memorije samo za čitanje i koja se koristi kada treba mapirati određeni skup podataka.



Slika 3: CUDA memorije i oblasti važenja

Dakle, vidimo da je prilikom kreiranja CUDA programa važno je poznavati raspoložive tipove memorije. Globalna memorija je očigledno znatno sporija sa aspekta pristupa i zbog toga bi trebalo izbegavati njeno korišćenje. Tipičan postupak u CUDA programima treba da izgleda otprilike ovako:

- razložiti problem na potprobleme,
- podeliti ulazne podatke na skupove koji mogu da stanu u registarsku i deljenu memoriju,
- učitati jedan skup podataka iz globalne memorije u registre i/ili deljenu memoriju,

- obraditi skup podataka pomoću bloka niti,
- kopirati rezultate nazad u globalnu memoriju.

5. Zaključak

Kroz ovaj rad dat je kratak pregled CUDA platforme i programerskog modela. Masivno paralelno programiranje je trenutno jedan od najkonkretnijih načina za ubrzavanje aplikacija. Ali da bi se ono koristilo mora se veoma pažljivo i detaljno pristupati rešavanju problema i paralelizaciji algoritama kako bismo izvukli najbolje moguće performanse. Osim CUDA-e treba pomenuti, bez zalaženja u detalje, još dva modela za programiranje grafičkih uređaja, a to su OpenCL – kreiran i održavan od strane Kronos grupe kao i Majkrosoftov DirectCompute.

Literatura

[1] David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Elsevier Inc., 2010

[2] *NVIDIA CUDA C Programming Guide*, NVIDIA Corporation, 2012

[3] Slobodna encoklopedia, Wikipedia http://www.wikipedia.org