Arhitektura GPU i CUDA

Uvod u GPGPU

- Razvoj 3D grafike i rastuća industrija video igara izvršio je veliki pritisak na razvoj grafičkih procesora, pa su vremenom isti evoluirali u paralelne i visokoprogramabilne procesore.
- Grafički procesori (GPU) su specijalizovani za računski intenzivna, visoko paralelna izračunavanja, i inicijalno su bili namenjeni za obradu grafike.
- Danas se koriste za računanja opšte namene (General-Purpose computation on GPU - GPGPU).

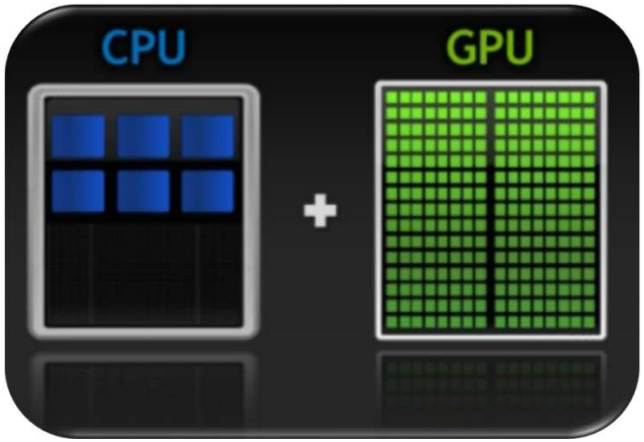
Primene GPGPU

Širok spektar primena:

- Fizičke simulacije (computational physics)
- Hemijske simulacije (computational chemistry)
- Biološke simulacije (life sciences)
- Finansijska izračunavanja (computational finance)
- Računarski vid (computer vision)
- Obrada signala
- Geometrija i matematika
- Baze podataka
- Treniranje i testiranje neuronskih mreža

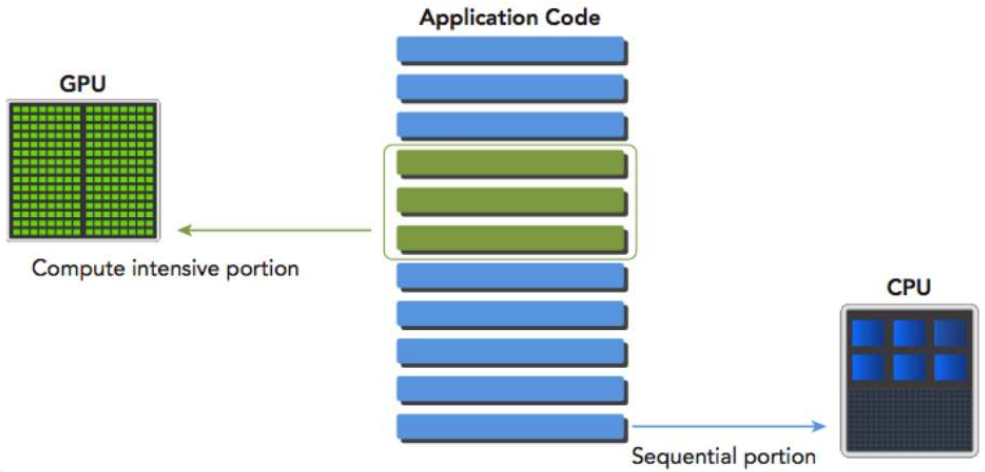
Uvod u GPGPU

- Heterogeno računarstvo
  - Korišćenje računskih resursa koji najbolje odgovaraju poslu
- CPU i GPU se najbolje koriste u režimu koprocesiranja:
  - CPU se koristi za sekvencijalni deo aplikacije gde je bitno kašnjenje (ulaz, izlaz, priprema podataka...)
  - GPU se koristi za delove koda koji troše najviše vremena (obrada velike količine podataka)



Heterogena aplikacija

- Heterogena aplikacija se sastoji iz dva dela:
  - Host kod (kod domaćina)
  - Device kod (kod uređaja)
  - Host kod se izvršava na CPU a device kod se izvršava na GPU



Heterogena aplikacija

- Izazov za GPU programera nije jednostavno postizanje dobrih performansi na GPU, već i raspoređivanje izračunavanja na sistemski procesor (host) i GPU, kao i prenos podataka između memorije host-a i GPU memorije.
- Kao što ćemo videti kasnije, kod GPU je pristutan višestruki paralelizam: višenitnost (multithreading), MIMD, SIMD i paralelizam na nivou instrukcija.

Višenitnost

- CPU niti uglavnom imaju veliki režijski trošak prilikom stvaranja (heavyweight)
  - Operativni sistem mora da uključuje i isključuje niti da bi obezbedio višenitnost u radu
  - Promene konteksta (Context switches) su spore i skupe
  - Koristi se mali broj niti (nekoliko desetina) i to zavisno od procesora
- GPU niti su veoma lake (lightweight)
  - Imaju veoma mali režijski trošak prilikom stvaranja
  - U tipičnom GPU sistemu pokrenuto je desetine hiljada niti

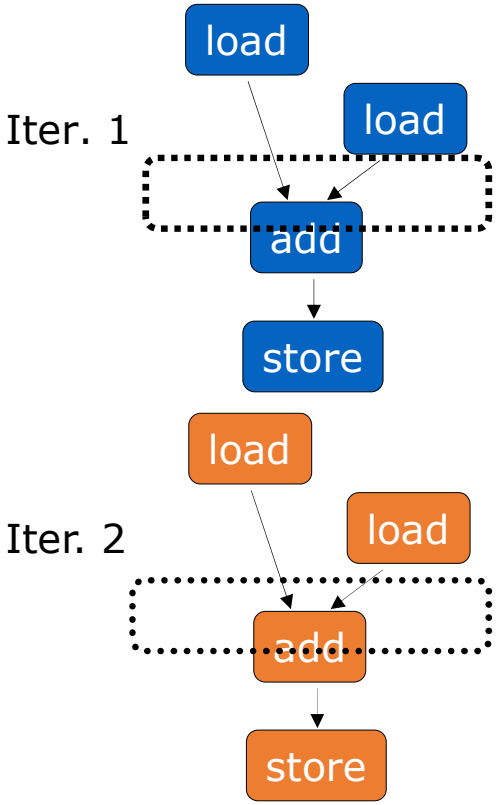
Programski model vs. Izvršni model

- Programski model se odnosi na način na koji programer kreira kod
  - E.g., Sekvencijalni, Data Parallel (paralelan po podacima), višenitni…
- Izvršni model se odnosi na način na koji se kod izvršava na hardveru
  - E.g., Izvršenje van redosleda pribavljanja, vektorski procesor, procesorska polja, multiprocesor, …
- Izvršni model se može razlikovati od programskog modela
  - Primer, SPMD programski model implementiran korišćenjem SIMD procesora kod GPU

Kako možemo iskoristiti paralelizam koji postoji u kodu?

Scalar Sequential Code

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

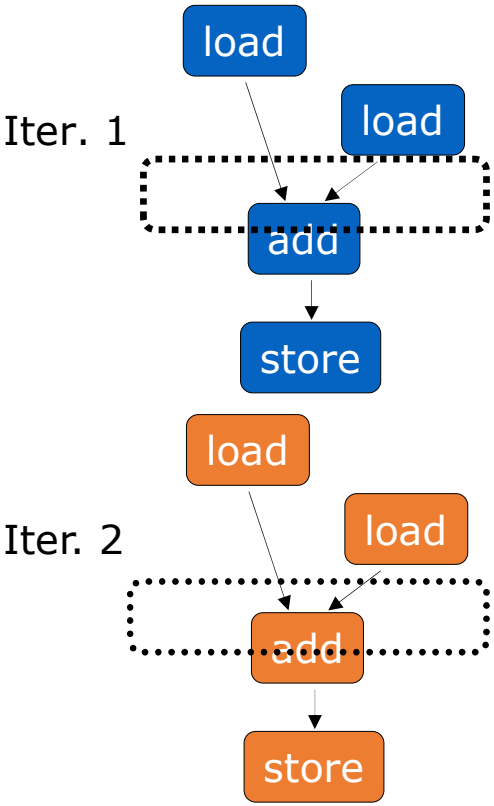


Postoje različiti programski modeli koji koriste paralelizam na nivou instrukcija koji postoji u ovom sekvencijalnom kodu

Prog. model 1: Sekvencijalni

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

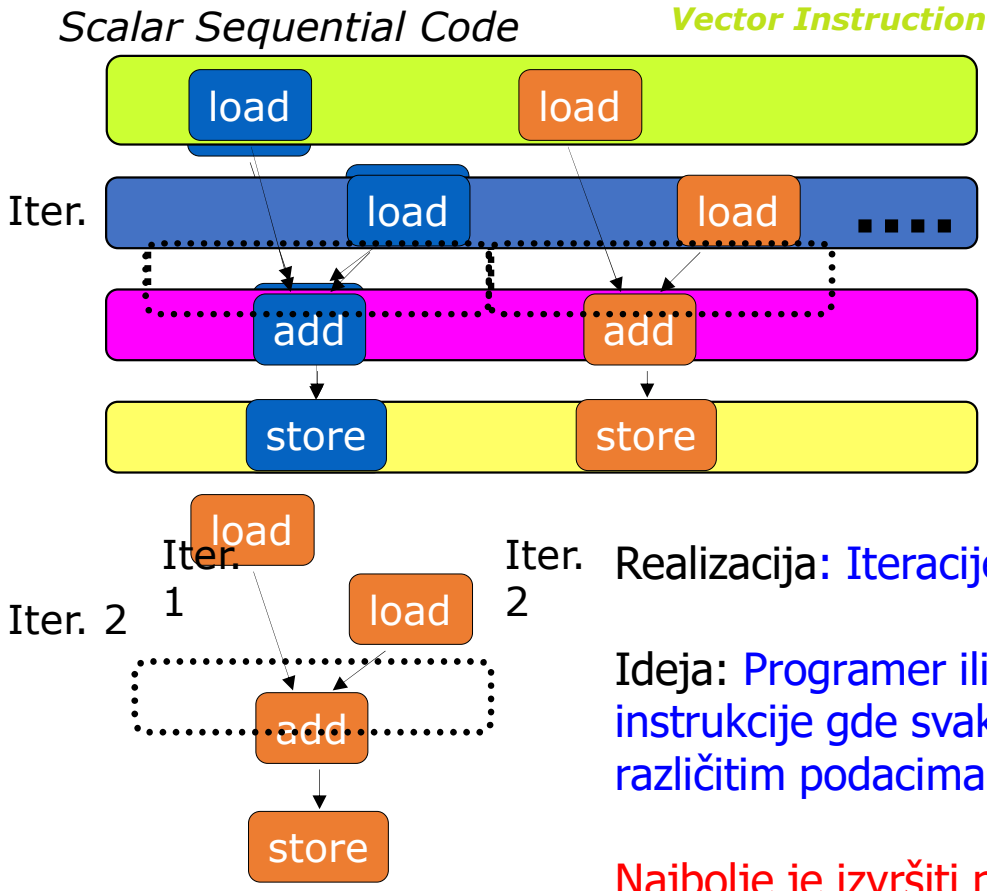
Scalar Sequential Code



- Ovaj kod može biti izvršen na:
- Protočnom procesoru
- Protočnom procesoru sa izvršenjem van redosleda pribavljanja
- Superskalarnom ili VLIW procesoru

Prog. model 2: Paralelan po podacima - Data Parallel

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Vectorized Code

VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

VST V3 → C

Realizacija: Iteracije su međusobno nezavisne

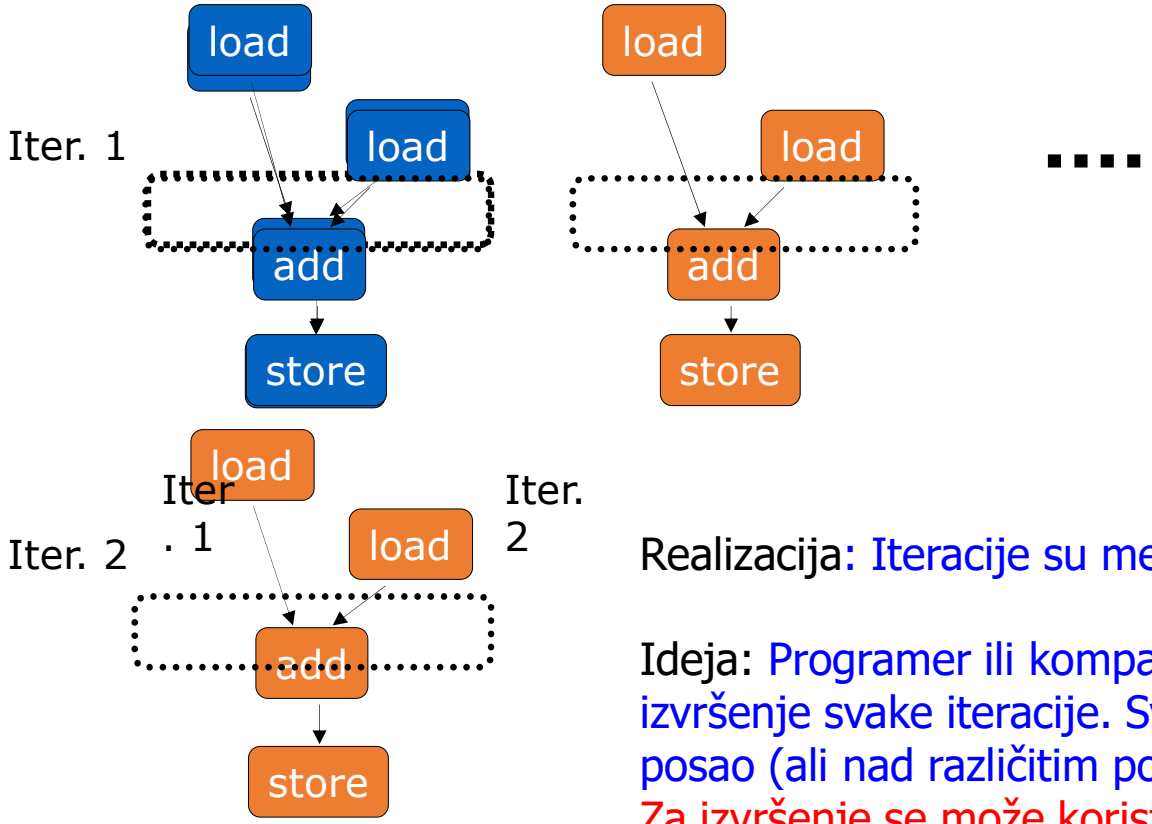
Ideja: Programer ili kompajler generiše SIMD instrukcije gde svaka obavlja istu operaciju nad različitim podacima

Najbolje je izvršiti na SIMD procesoru (vektorskom procesoru, procesorskom polju)

Prog. model 3: Višenitni-Multithreaded

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Scalar Sequential Code

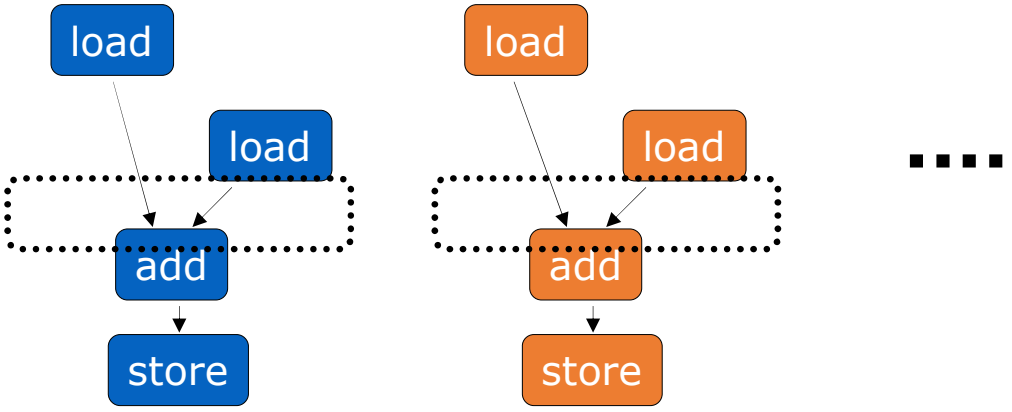


Realizacija: Iteracije su međusobno nezavisne

Ideja: Programer ili kompajler generiše po nit za izvršenje svake iteracije. Svaka nit obavlja isti posao (ali nad različitim podacima)
Za izvršenje se može koristiti MIMD mašina

Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Iter. 1

Iter. 2

Realization: Iteracije su međusobno nezavisne

Ovaj konkretni model se može nazvati:

SPMD: Single Program Multiple Data

Za izvršenje se može koristiti SIMT mašina

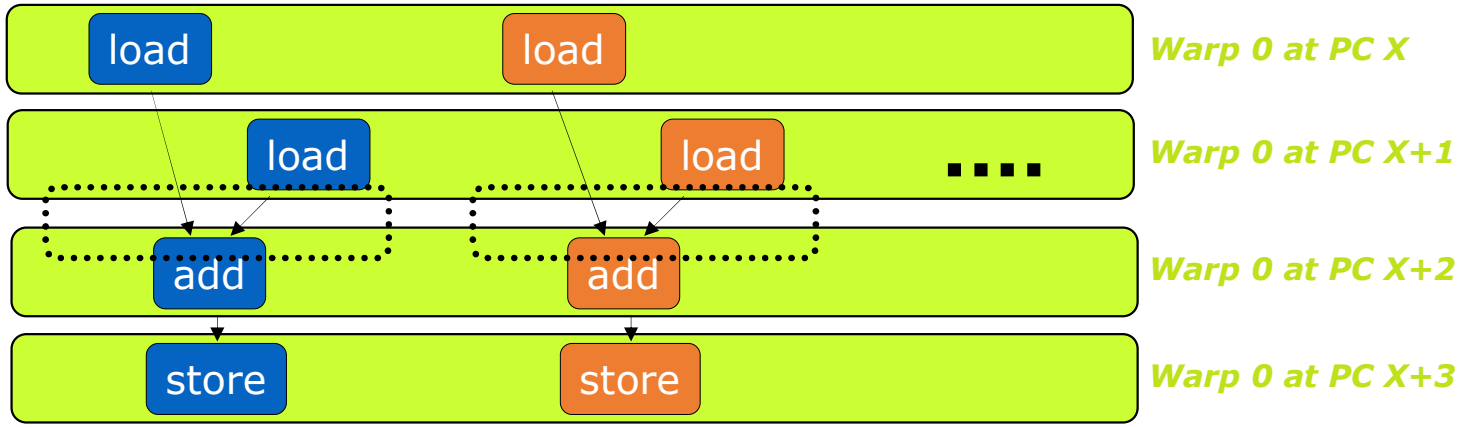
Single Instruction Multiple Thread

GPU je SIMD (SIMT) mašina

- GPU je SIMD (SIMT) mašina u osnovi, osim što GPU nije programiran korišćenjem SIMD instrukcija
- Programski model koji se koristi kod GPU su niti (SPMD model-Single Program Multiple Data)
  - Svaka nit izvršava isti kod ali nad različitim podacima
  - Svaka nit ima sopstveni kontekst (tj. može se tretirati/ponovno pokrenuti/izvršiti nezavisno)
- Skup niti koje izvršavaju istu instrukciju se dinamički grupišu od strane hardvera
- U suštini SIMD operacija biva formirana od strane hardvera!

SPMD na SIMT mašini

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Iter. 1

Iter. 2

Warp Skup niti koje izvršavaju istu instrukciju (tj., sa istom vrednošću PC)

Ovaj konkretni model se može nazvati : SPMD: Single Program Multiple Data

Izvršenje na GPU korišćenjem SIMT modela: Single Instruction Multiple Thread

SIMD vs. SIMT Execution Model

- SIMD: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN
- SIMT: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads
- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently (on any type of scalar pipeline) → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

SIMT

SIMT model uključuje tri ključne osobine koje ne poseduje SIMD:

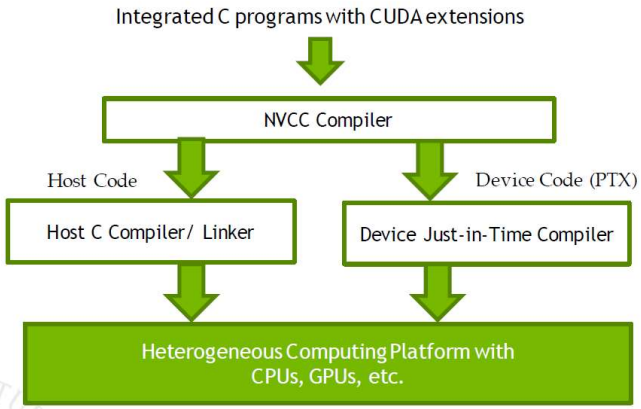
- Svaka nit ima svoj brojač instrukcija
- Svaka nit ima svoje sopstveno stanje registara
- Svaka nit može imati nezavisan tok izvršenja

NVIDIA GPU

- Kako bi omogućila programerima da lakše koriste grafičke procesore za potrebe razvoja aplikacija za paralelizovana izračunavanja, kompanija NVIDIA je razvila okruženje za razvoj programa koji se izvršavaju na njihovom hardveru.
- NVIDIA je odlučila da razvije jezik i programsko okruženje nalik C-u što bi poboljšalo produktivnost GPU programera.
- Ime njihovog sistema je CUDA, tj. Compute Unified Device Architecture.
- CUDA programer generiše C/C++ kod za sistemski procesor (host) i C/C++ dijalekt kod za GPU (uređaj(device), dakle D u CUDA).

CUDA program i prevođenje

- Host kod je napisan korišćenjem ANSI C, a kod za uređaj korišćenjem CUDA C.
- Izvorni kod sadrži CUDA ekstenzije kojima se specificira koji delovi koda se kako i gde izvršavaju
- NVIDIA C Compiler (nvcc) se koristi da generiše izvršni kod i za host (CPU) procesor i za uređaj (GPU)
  - PTX (Parallel Thread eXecution) kod
    - Predstavlja neku vrstu međukoda za GPU

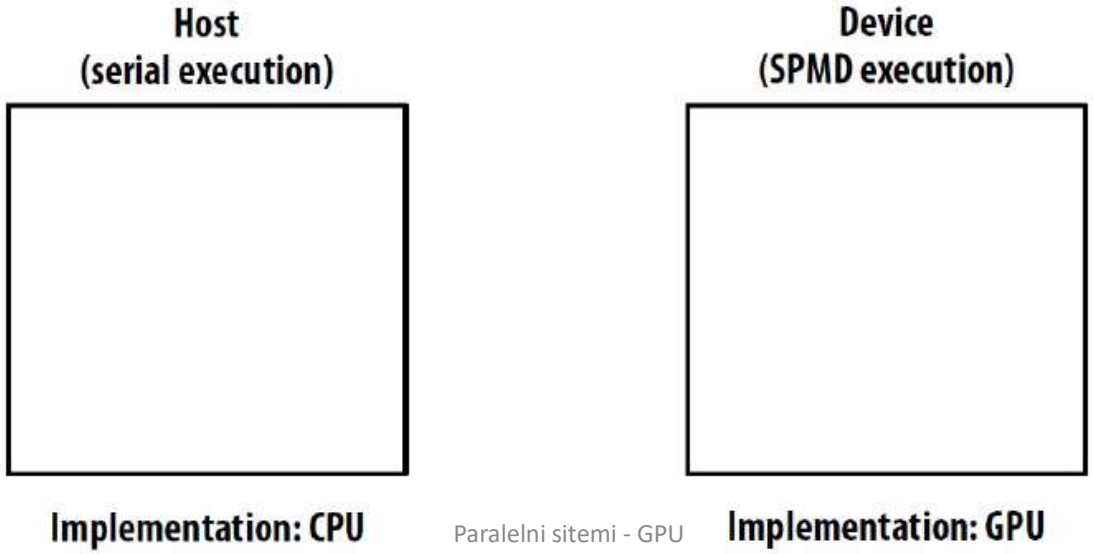


CUDA program i prevođenje

- Kod treba da bude kompajliran kompajlerom koji prepoznaje i razume ove dodatne deklaracije. Koristi se CUDA C kompajler kompanije NVIDIA, nazvan NVCC (NVIDIA C Compiler).
- NVCC obrađuje CUDA program koristeći CUDA ključne reči za razdvajanje host koda i device koda.
- Host kod je običan ANSI C kod, koji se dalje kompajlira standardnim C/C++ kompajlerima hosta i pokreće kao tradicionalni proces na CPU-u.
- Device kod je označen CUDA ključnim rečima radi označavanja funkcija koje obavljaju paralelnu obradu podataka, nazvanih kerneli, i njihovih pripadajućih podataka.
- PTX kod je nezavistan od specifične arhitekture i može se dalje optimizovati, kako bi se poboljšale performanse ili smanjila potrošnja resursa.
- Optimizovani PTX kod se kompajlira u mašinski kod specifičan za ciljanu GPU arhitekturu. Ovaj izvršni kod je ono što se izvršava na GPU-u.

Izvršenje CUDA programa

- Razdvajanje koda na "host" i "device” se vrši od strane programera
- "Device" kod - izvodi se na spoljašnjem uređaju (GPU), SPMD izvođenje
- "Host" kod- izvodi se na računaru hostu (domaćinu), serijsko izvođenje



CUDA

- Uvedena 2007. godine s NVIDIA Tesla arhitekturom

- Napomena: otvorena "verzija" CUDA-e je OpenCL
  - CUDA se izvršava samo na NVIDIA-inim GPU-ovima
  - OpenCL se izvršava na CPU-ovima i GPU-ovima mnogih proizvodača

CUDA niti

- Osnovna programska primitiva koja se definiše u CUDA-i je CUDA nit (CUDA Thread).
- Program se sastoji iz velikog broja ovakvih niti koje su grupisane u blokove.
- Blokovi su grupisani u grid
- Sve niti se generišu za vreme poziva kernela (kernel funkcije) i zajedno se zovu (čine) grid niti.
- Poziv kernel funkcije definiše koliko se blokova kreira u gridu i koliko svaki od blokova ima niti

CUDA kernel

- Najjednostavnija forma kernela (kernel funkcije):

  kernel_routine<<<gridDim, blockDim>>>(lista parametara);

- *gridDim – broj blokova u gridu (veličina "grida")*
- *blockDim – broj niti unutar svakog bloka (veličina bloka)*
- *args – limitirani broj argumenata, najčešće pokazivača na nizove na GPU, i skalarni podaci koje se kopiraju po vrednosti*
- Generalnija forma dozvoljava da *gridDim* i *blockDim* budu 2D ili 3D kako bi se pojednostavilo pisanje programa

- Da bi se u programu napravila razlika između funkcija za GPU (uređaj) i funkcija za sistemski procesor (host), CUDA koristi __global__ za prvi i __host__ za drugi.

CUDA promenljive

- Kada pokrenete program, host kod se izvršava na CPU-u i poziva kernel funkcije, koje se izvršavaju na GPU-u.
- Kernel (jezgro) se konfiguriše prilikom svakog poziva i na taj način se specificira koliko blokova se generiše - **gridDim**, i koliko niti ima svaki od blokova - **blockDim**
- Sve niti u bloku dele isti identifikator bloka - promenljivu **blockIdx**. Svaka nit u okviru bloka ima indeks niti kome se pristupa preko promenljive **threadIdx**.
- **Grupe niti simultano izvršavaju iste instrukcije, ali nad različitim podacima.**
- Upotreba promenljivih **blockIdx i threadIdx** nitima omogućava da odluče nad kojim podacima da rade.

Primer sabiranja dva vektora

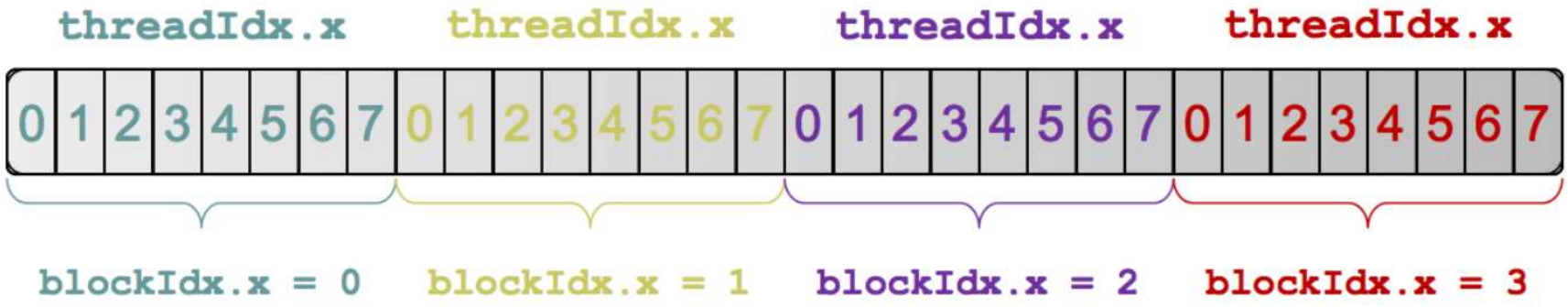
Neka je dat C kod za koji ćemo kreirati CUDA verziju:

```
void add(int *a, int *b, int *c, int n)
{
for (int i = 0; i < n; ++i)
        c[i] = a[i] + b[i];
}
```

Primer sabiranja dva vektora

- Razmotrimo indeksiranje unutar polja, jedna nit obrađuje jedan element
  - Pretpostavimo da se pokreće M=8 niti po bloku i da je polje veličine 32 elementa
  - Ako imamo M niti/bloku jedinstven indeks svake niti je dat sa:

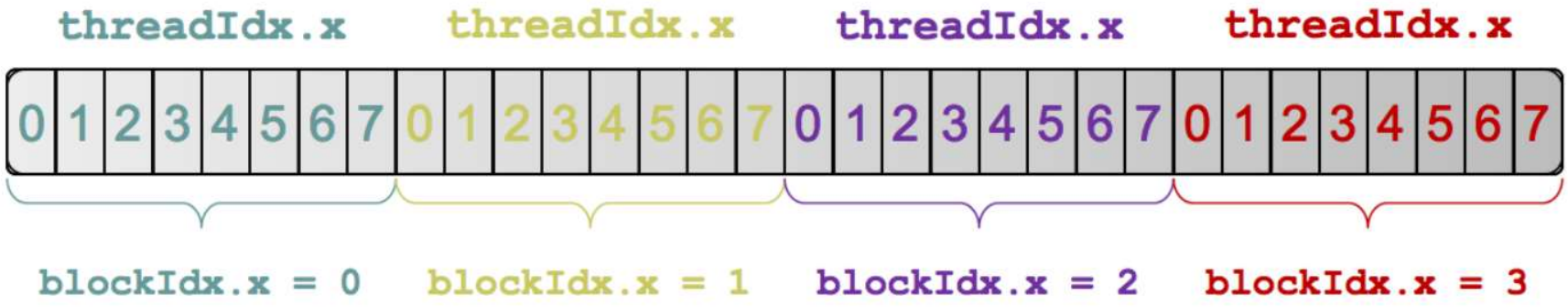
int index = ?



Primer sabiranja dva vektora

- Razmotrimo indeksiranje unutar polja, jedna nit obrađuje jedan element
  - Pretpostavimo da se pokreće M=8 niti po bloku i da je polje veličine 32 elementa
  - Ako imamo M niti/bloku jedinstven indeks svake niti je dat sa:

int index = threadIdx.x + blockIdx.x * M; gde je M veličina bloka; tj, blockDim.x



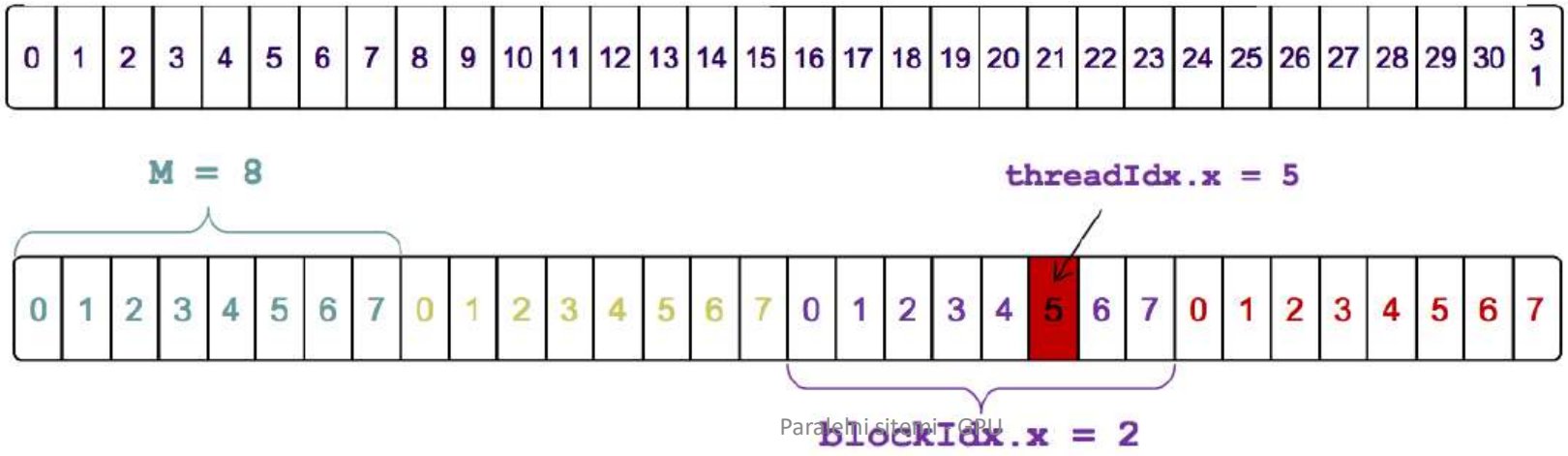
Primer sabiranja dva vektora

- Koji element niza će obrađivati nit sa indeksom 5 u bloku sa indeksom 2?

int index = threadIdx.x + blockIdx.x * blockDim.x;

= 5 + 2 * 8

=21



Primer sabiranja dva vektora

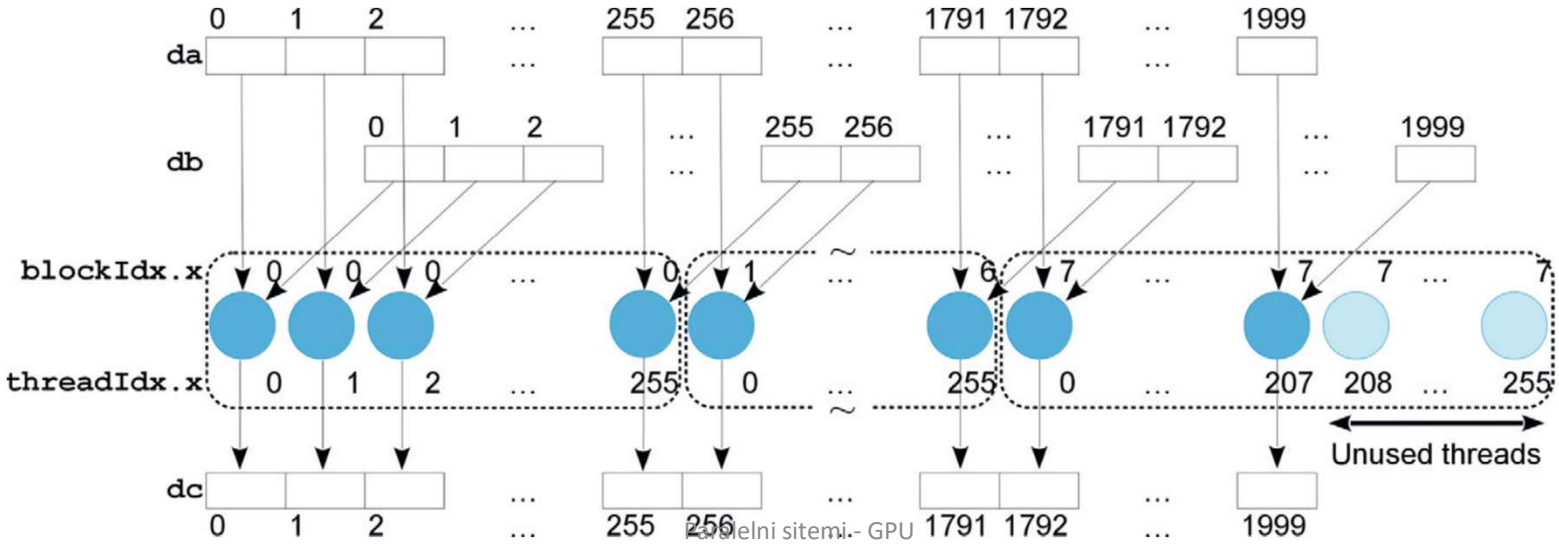
- Za dati indeks bloka i indeks niti unutar bloka svaka nit izračunava nad kojim delom podataka radi
- U nekim situacijama to ne funkcioniše
- Primer. ? bloka sa 512 niti/bloku obrađuju 1000 elemenata niza
- U toj situaciji poslednje ? niti ne rade ništa
- Primer. ? blokova sa 256 niti/bloku obrađuju 2000 elemenata niza
- Tada poslednjih ? niti ne rade ništa

Primer sabiranja dva vektora

- Za dati indeks bloka i indeks niti unutar bloka svaka nit izračunava nad kojim delom podataka radi
- U nekim situacijama to ne funkcioniše
- Primer. Dva bloka sa 512 niti/bloku obrađuju 1000 elemenata niza
- U toj situaciji poslednje 24 niti ne rade ništa
- Primer. 8 blokova sa 256 niti/bloku obrađuju 2000 elemenata niza
- Tada poslednjih 48 niti ne rade ništa

Primer sabiranja dva vektora

- GPU kernel funkcija počinje izračunavanjem odgovarajućeg indeksa niti *index* i to na osnovu identifikatora bloka, broj niti po bloku i identifikatora niti u bloku.
- Sve dok je indeks *index* unutar niza (index < n), vrši se sabiranje dva elementa niza *da* i *db*.



Primer sabiranja dva vektora

```
__global__ void add(int *a, int *b, int *c, int n)
{       int index = threadIdx.x + blockIdx.x * blockDim.x;
        if(index < n)
                c[index] = a[index] + b[index];
}
```

Poziv kernela:

add<<<(N+?)/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c, N);

- Neka je pokrenuto toliko niti da jedna nit obrađuje po jedan element niza, sa 256 CUDA niti po bloku.
- Svaka nit je vezana za odgovarajući indeks na osnovu koga može da pristupi odgovarajućem elementu niza

Primer sabiranja dva vektora

```
__global__ void add(int *a, int *b, int *c, int n)
{       int index = threadIdx.x + blockIdx.x * blockDim.x;
        if(index < n)
                c[index] = a[index] + b[index];
}
```

Poziv kernela:

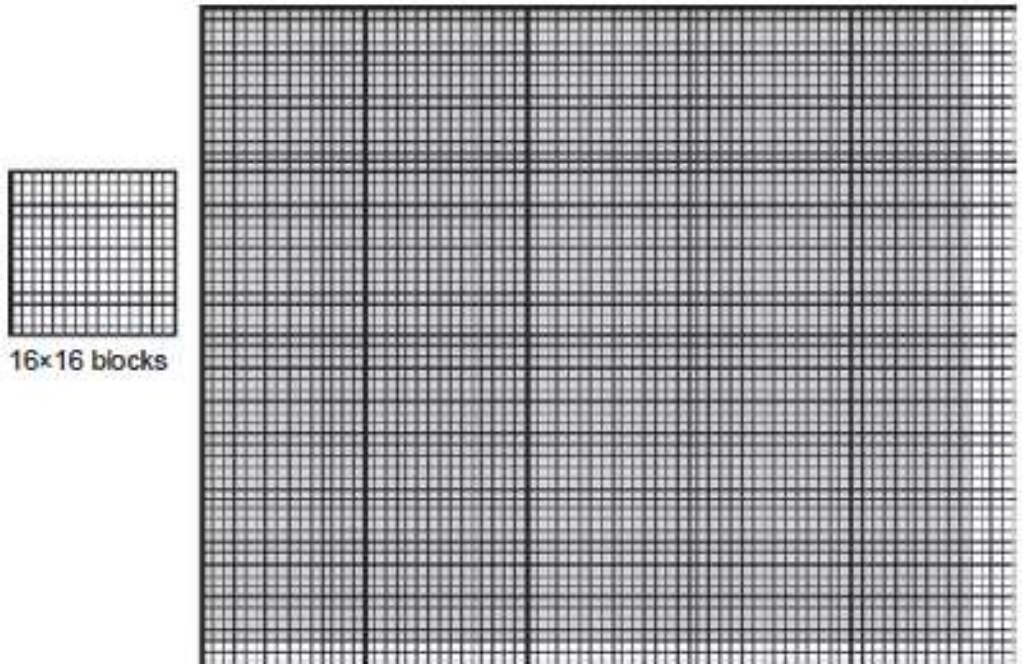
```
add<<<(N+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c, N);
```

- Neka je pokrenuto toliko niti da jedna nit obrađuje po jedan element niza, sa 256 CUDA niti po bloku.
- Svaka nit je vezana za odgovarajući indeks na osnovu koga može da pristupi odgovarajućem elementu niza

Primer obrade slike

Razmotrimo sliku veličine 76×62 piksela i želimo je obraditi sa blokom niti veličine 16×16.

Potrebno je 5×4 bloka i neki od blokova će imati neiskorišćene niti.




```
dim3 dimGrid(ceil(im_width/16.0), ceil(im_height/16.0), 1);
dim3 dimBlock(16, 16, 1);
pictureKernel<<<dimGrid,dimBlock>>>(d_Pin,d_Pout,im_width, im_height);
```

CUDA niti i GPU hardver

- Upoređujući C i CUDA kodove, vidimo zajednički obrazac kreiranje data-parallel CUDA koda.
- C verzija ima petlju u kojoj je svaka iteracija nezavisna od ostalih, omogućavajući da se petlja direktno transformiše u paralelni kod gde svaka iteracija petlje postaje zasebna nit.
- Programer određuje paralelizam u CUDA eksplicitno određujući dimenzije grida i broj niti u bloku niti.
- GPU hardver upravlja paralelnim izvršavanjem i upravljanjem nitima; to ne rade ni aplikacija ni operativni sistem.
- Da bi se pojednostavilo upravljanje pomoću hardvera, CUDA blokovi niti mogu da se izvršavaju nezavisno i to u bilo kom redosledu. Međutim, različiti blokovi niti ne mogu direktno komunicirati.

CUDA niti i GPU hardver

- Za planiranje, izvršavanje i upravljanje nitima zadužen je GPU hardver. Kako bi planiranje niti bilo dovoljno jednostavno, rezultat izvršenja programa ne sme da zavisi od redosleda izvršenja niti.
- Niti iz istog bloka ne mogu da komuniciraju sa nitima u drugim blokovima direktno – jedino je moguće koordinisati njihovo izvršenje korišćenjem globalne memorije grafičkog procesora i atomičnih operacija, međutim, ovakav pristup treba izbegavati ukoliko je to moguće, zbog znatnog narušavanja performansi.
- Paralelizam se u CUDA-i postiže tako što će svaka nit koja se kreira, da bude zadužena za obradu jedne (njoj pridružene) grupe podataka.
- Niti se izvršavaju u grupama od po 32 niti.

Upravo ovakav programski model predstavlja izazov programerima – iako izgleda kao MIMD, potrebno je razmišljati o SIMD modelu, zbog toga što sve niti iz grupe u jednom trenutku izvršavaju istu instrukciju. Zbog toga je potrebno voditi računa o mnogim hardverskim konceptima, koji će biti istaknuti u narednim slajdovima.

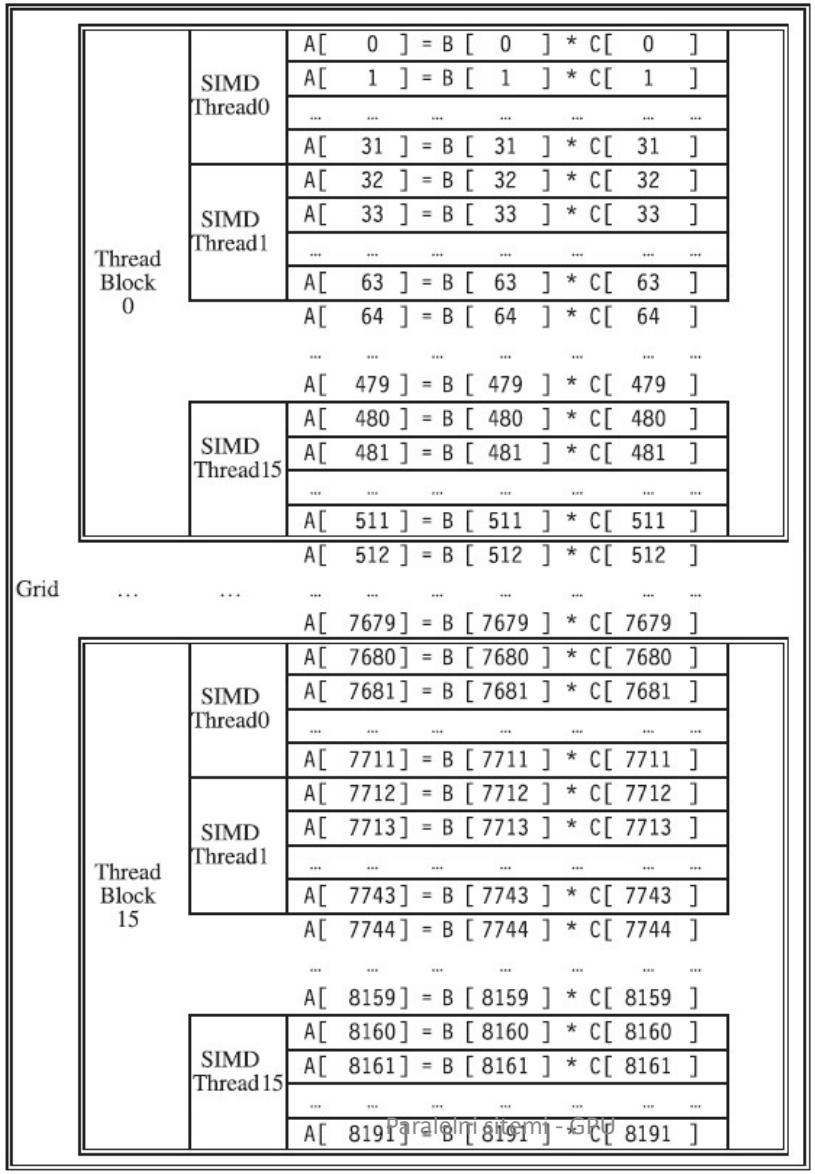
CUDA niti i GPU hardver

Da bismo razumeli arhitekturu grafičkih procesora, razmotrićemo jedan primer.

Npr. recimo da želimo da pomnožimo dva vektora dužine 8192 i da želimo da jedan blok niti množi 512 elementa, onda bismo imali 8192/512 = 16 blokova niti.

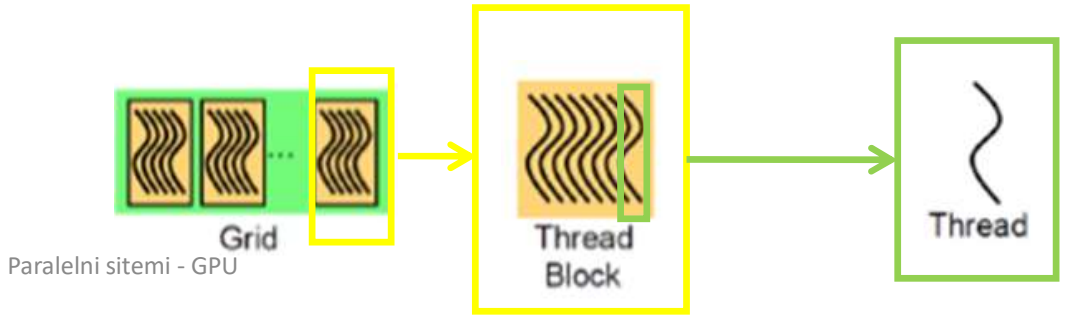
Rešetka (eng. Grid) – skup blokova niti (eng. Thread Block) koje izvršavaju napisani kod (kernel funkciju).

Blokovi niti služe za dekompoziciju problema na manje celine koje su pogodnije za obavljanje željenih izračunavanja.



Softverski pogled na GPU

- Niti su organizovane u dva nivoa hijerarhije. Grid (koji izvršava kernel) se sastoji od jednog ili više blokova, dok se svaki od tih blokova sastoji od jedne ili više niti.
- **Grid** (Rešetka) = Skup blokova niti koje izvršavaju kernel
- **Blok niti** = Grupa niti koje izvršavaju kernel i mogu komunicirati preko deljive memorije
- Kernel se izvršava kao grid blokova niti

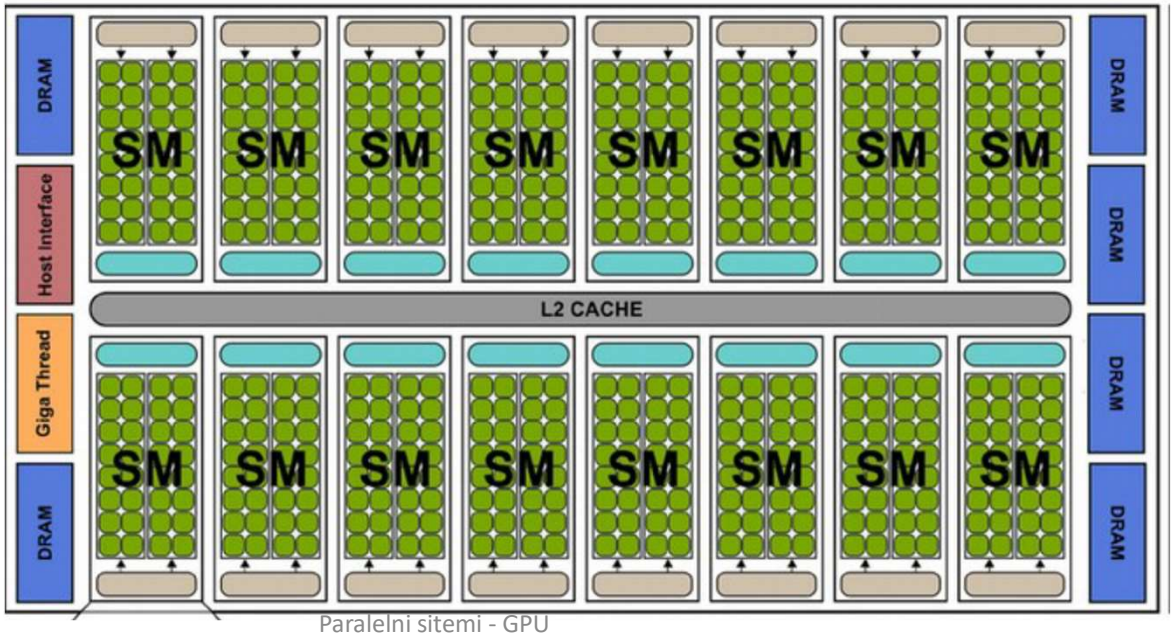


SW->HW

- Svaki blok niti se dodeljuje i izvršava na nekom od više postojećih višenitnih SIMD procesora (eng. Multithreaded SIMD processor). Ovi procesori se još i nazivaju streaming multiprocessors (SM).
- Za dodelu blokova SM procesorima je zadužen **planer blokova niti** (eng. Thread Block Scheduler), koji je hardverski implementiran.
- Broj blokova i broj niti po bloku definiše programer u samom kodu.
- U našem primeru (množenje vektora) planer blokova niti će poslati 16 blokova niti odgovarajućim SM-ima koji bi izračunavali 8192 elementa niza.
- Blokovi se dodeljuju SM-ovima. Moguće je da više blokova istovremeno radi na istom SM-u.
- Blok može da se izvršava na SM-u ako postoji dovoljno resursa (registara i deljive memorije) u okviru SM-a za njegovo izvršenje.

Arhitektura GPU

Na slici je prikazana uprošćena arhitektura grafičkog procesora sa 16 višenitnih SIMD procesora (16 SM-a).



Arhitektura GPU

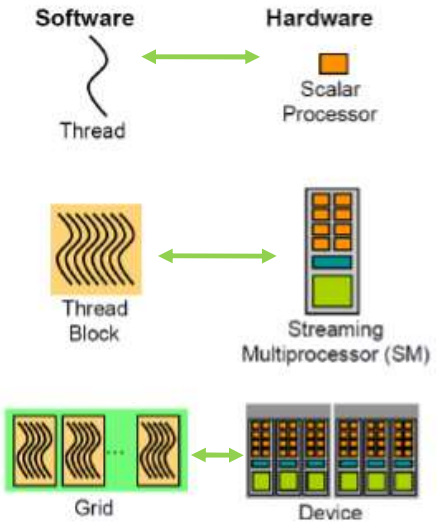
- Broj SM procesora može da varira od jedne do druge GPU
- Svaki SM se sastoji iz više **skalarnih procesora(SP)**
- Svaki SM je višenitni procesor sa zasebnim programskim brojačima.
- Upravo zahvaljujući činjenici da zapravo predstavljaju multiprocesore koji se sastoje od višenitnih procesora(SM-a), grafički procesori (GPU) dosta podsećaju na MIMD sisteme.

Number of SMs x SPs across generations

- Tesla (2007): 30 x 8
- Fermi (2010): 16 x 32
- Kepler (2012): 15 x 192
- Maxwell (2014): 24 x 128
- Pascal (2016): 56 x 64
- Volta (2017): 80 x 64

SW->HW

- **Grid → GPU**: Ceo grid je opslužen od strane GPU čipa
- **Blok → SM**: Svaki SM je odgovoran za opsluživanje jednog ili više blokova. Blok niti se nikada ne deli između različitih SM-a
- **Nit → SP**: Svaki SM je podeljen na više SP-a, gde je svaki od njih odgovoran za izvršenje pojedinačne niti iz bloka



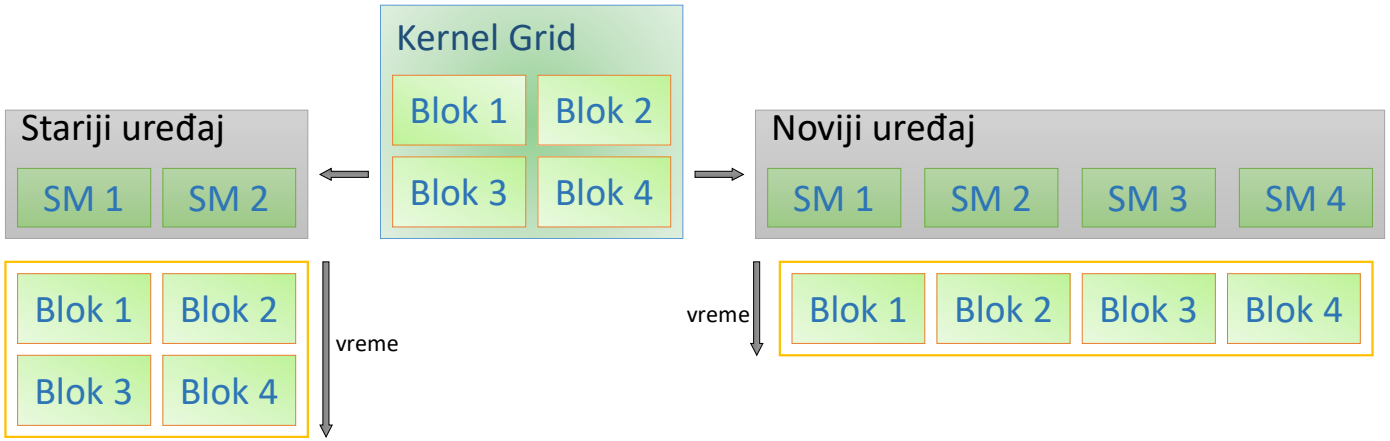
SW->HW

- Blokovi grida (rešetke) se označavaju i distribuiraju multiprocesorima (SM)
- Niti unutar jednog bloka niti se izvršavaju konkurentno na jednom multiprocesoru
- Više blokova niti se mogu izvršavati istovremeno na jednom multiprocesoru
- Kako se blokovi niti završavaju, novi blokovi se pokreću na oslobođenim multiprocesorima
- **SM je dizajniran da konkurentno izvršava na hiljade niti**
- Svaka nit se izvršava na skalarnom procesoru (SP)
- Instrukcije se izvršavaju redom

Skalabilnost

- Blokovi se mogu izvršiti bilo kojim redom jer ne moraju da čekaju jedni na druge tokom izvršenja.
- Novi uređaj sa više SM-a može izvršiti više blokova paralelno obezbeđujući bolje performanse.
- Hardver je slobodan da dodeljuje blokove bilo kom SM-u u bilo kom trenutku.
  - Kernel se prilagođava bilo kom broju SM-a.
- Ova fleksibilnost omogućava skalabilnu implementaciju.

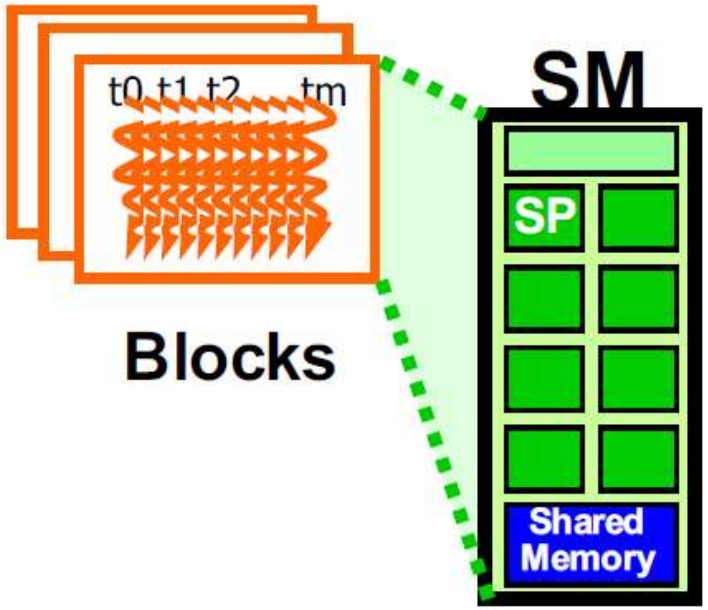
Skalabilnost



- Vreme napreduje od vrha ka dnu. Uređaj sa leve strane podržava izvršenje malog broja blokova odjednom (samo 2), dok noviji uređaj (desno) sa više resursa za izvršavanje(SM-a) istovremeno može izvršiti veći broj blokova (4). Posledica je različita brzina izvršenja.

Izvršenje blokova niti

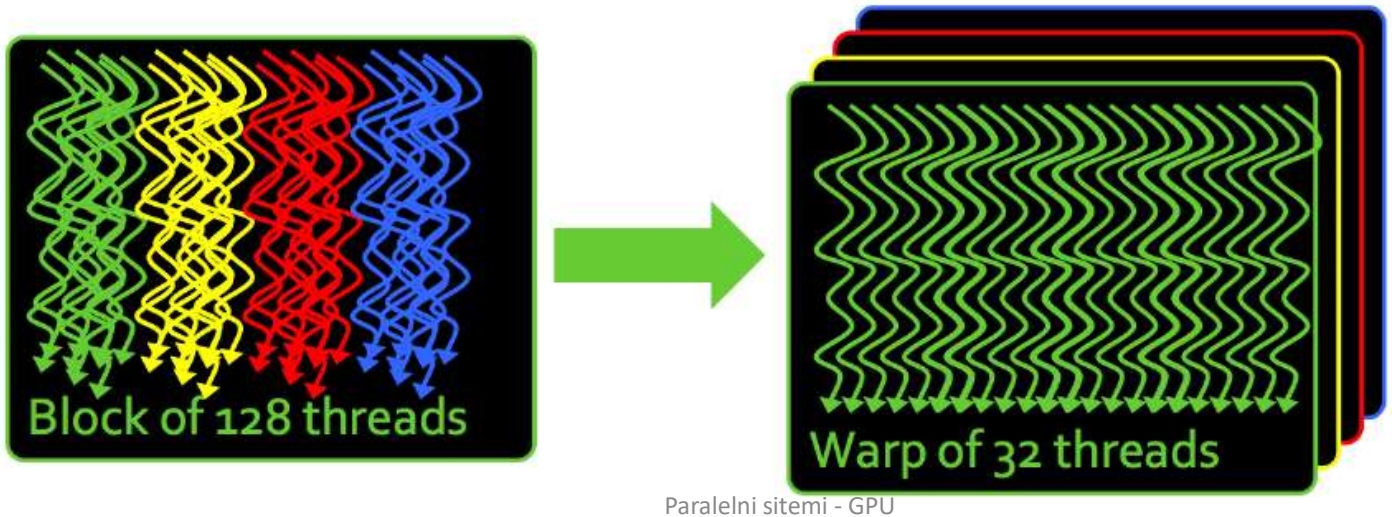
- Niti se dodeljuju Streaming Multiprocesorima (SM) u blokovima.
- Kod Fermi arhitekture do 8 blokova može biti dodeljeno svakom SM-u, ukoliko resursi to dozvoljavaju.
- Fermi SM može izvršiti do 1536 niti po SM.
- To može biti 256 (niti po bloku) * 6 blokova Ili 512 (niti po bloku) * 3 bloka, itd.
- Očigledno je da 12 blokova od po 128 niti nije izvodljiva opcija.
- SM upravlja/raspoređuje izvršenje niti.



Warp

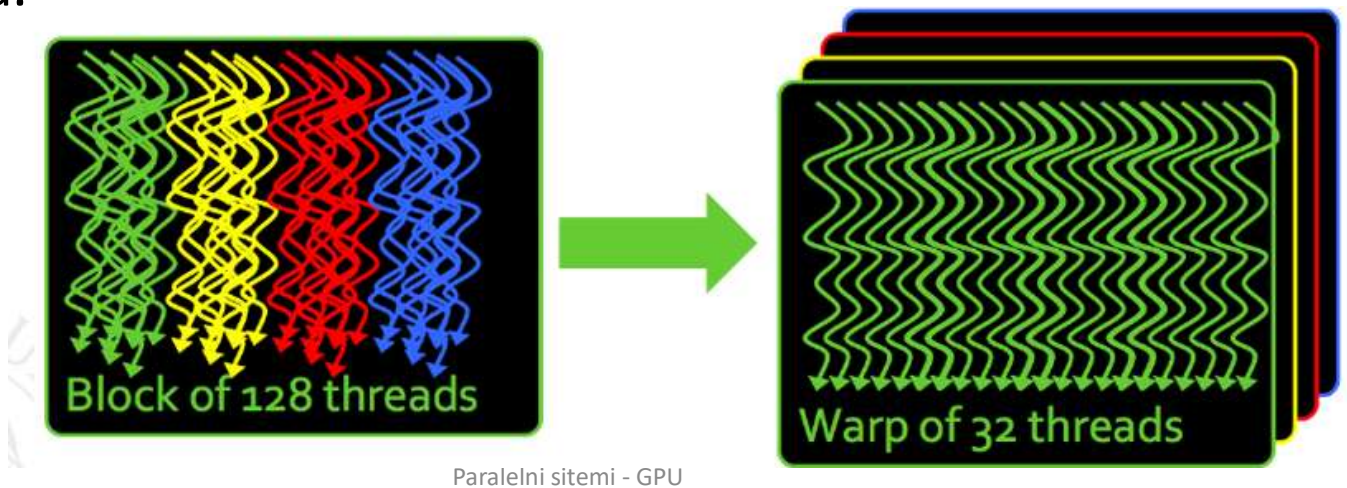
Do sada su bile definisane programske primitive: grid, blok niti, nit.

Na nižem nivou, mašinski objekat koji kreira i koji izvršava hardver jeste nit SIMD instrukcija - WARP.



Planer warpova

- Warp je skup paralelnih CUDA niti (obično 32) koje izvršavaju zajedno istu instrukciju na višenitnom SIMD procesoru (SM-u)
- Za planiranje izvršenja ovih niti se koristi druga hardverska komponenta – **planer SIMD niti (eng. Warp scheduler)** koji se nalazi u okviru SM-a.

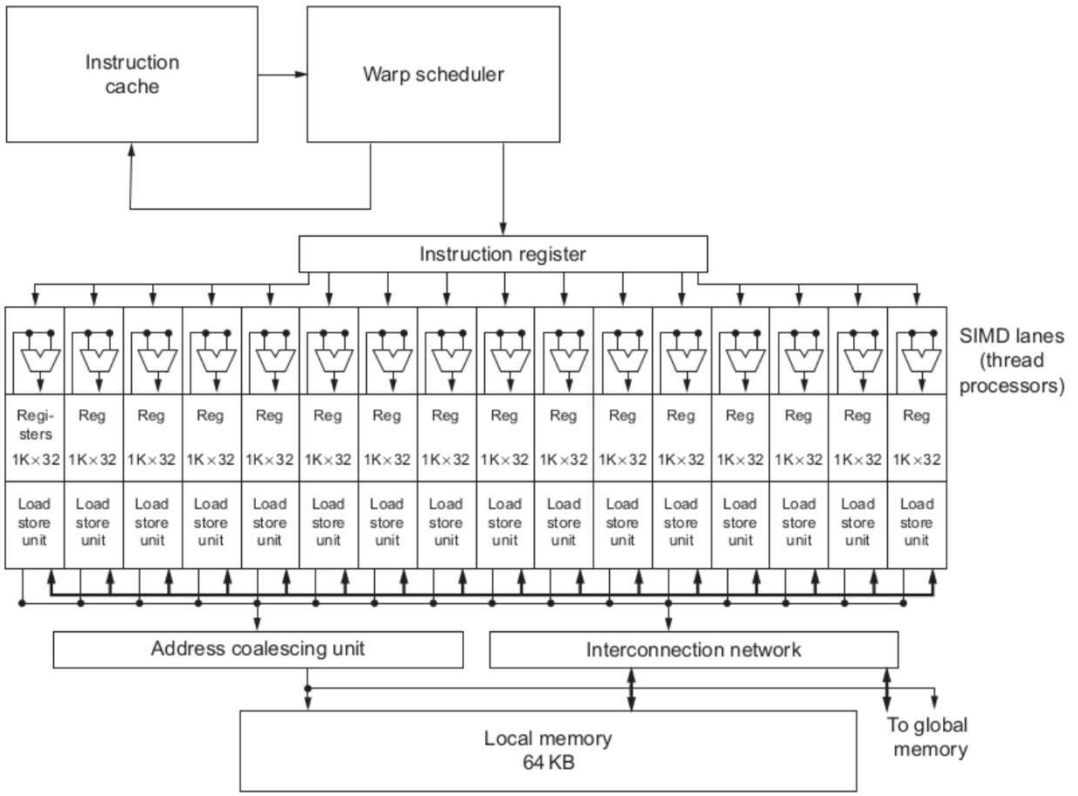


Planer warpova

- Warp Scheduler zna koji warpovi su spremni za izvršenje
  - Varpovi čija sledeća instrukcija ima svoje operande spremne
- Warp uglavnom sadrži 32 CUDA niti (u budućnosti se možda promeni) i sve one obavljaju istu instrukciju u bilo kom trenutku.
- U našem primeru imamo 512/32 = 16 warp-ova po bloku.(slika sa slajda)
- Kako warp sadrži SIMD instrukcije, SM mora da poseduje više paralelnih funkcionalnih jedinica koje zovemo SIMD staze (eng.lane).
- Svaka CUDA nit se izvršava na jednom od skalarnih procesora (jezgara), tj. na jednoj stazi streaming multiprocesora.

Blok dijagram SM

Na slici je prikazan je blok dijagram jednog streaming multiprocesora sa 16 staza, tj. 16 jezgara.



Planiranje izvršenja warpova

- Ukoliko imamo 16 SIMD staza, a 32 CUDA niti u warp-u, onda SIMD instrukciju izvršava 16 niti, a nakon toga sledećih 16 niti.
- Kao kratak rezime videćemo kako izgleda izvršavanje programa na grafičkom procesoru.
- Program se pokreće sa definisanim brojem blokova (dimenzija grida) i definisanim brojem niti po bloku (dimenzija bloka).
- Planer blokova niti raspoređuje blokove svim dostupnim višenitnim SIMD procesorima (SM-ovima).
- Svaki SM izvršava grupu od 32 niti koja predstavlja warp.
- Koji će se warp sledeći izvršavati određuje Warp scheduler (planer izvršenja SIMD niti).

Planiranje izvršenja warpova

- Kada se warp-u dodeli vreme za izvršenje, instrukcija toga warpa koja je na redu za izvršavanje se prosleđuje svim SIMD stazama (tj. jezgrima).
- Hardver izvršava istu instrukciju za sve niti u okviru jednog warpa (Single Instruction Multiple Thread).
- Niti u istom warp-u u jednom trenutku izvršavaju istu instrukciju, ili su neaktivne.
- Situacija da su neke niti neaktivne je moguća ukoliko u kodu postoji bilo kakvo uslovno grananje, nakon kojeg niti u warp-u imaju različite tokove izvršenja – niti divergiraju.

Divergentnost niti

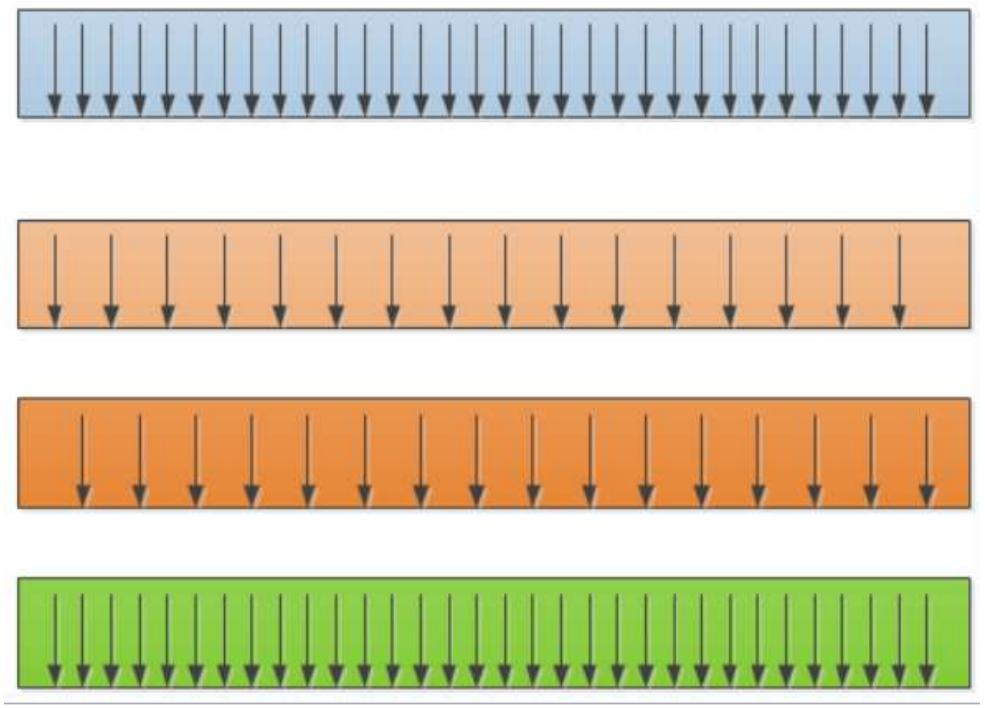
```
__global__ void odd_even(int n, int* x) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i % 2 == 0 ) {
    x[i] = x[i] + 1;
  } else {
    x[i] = x[i] + 2;
  }
}
```

Divergentnost niti

U tom slučaju će prvo biti aktivne samo niti koje ispunjavaju uslov, tj. izvršiće se then deo koda, dok će ostale niti biti neaktivne, a zatim će se biti aktivne samo one koje ne ispunjavaju uslov tj. izvršavaće se else deo koda

Serijalizacija izvršenja uzrokuje pad performansi

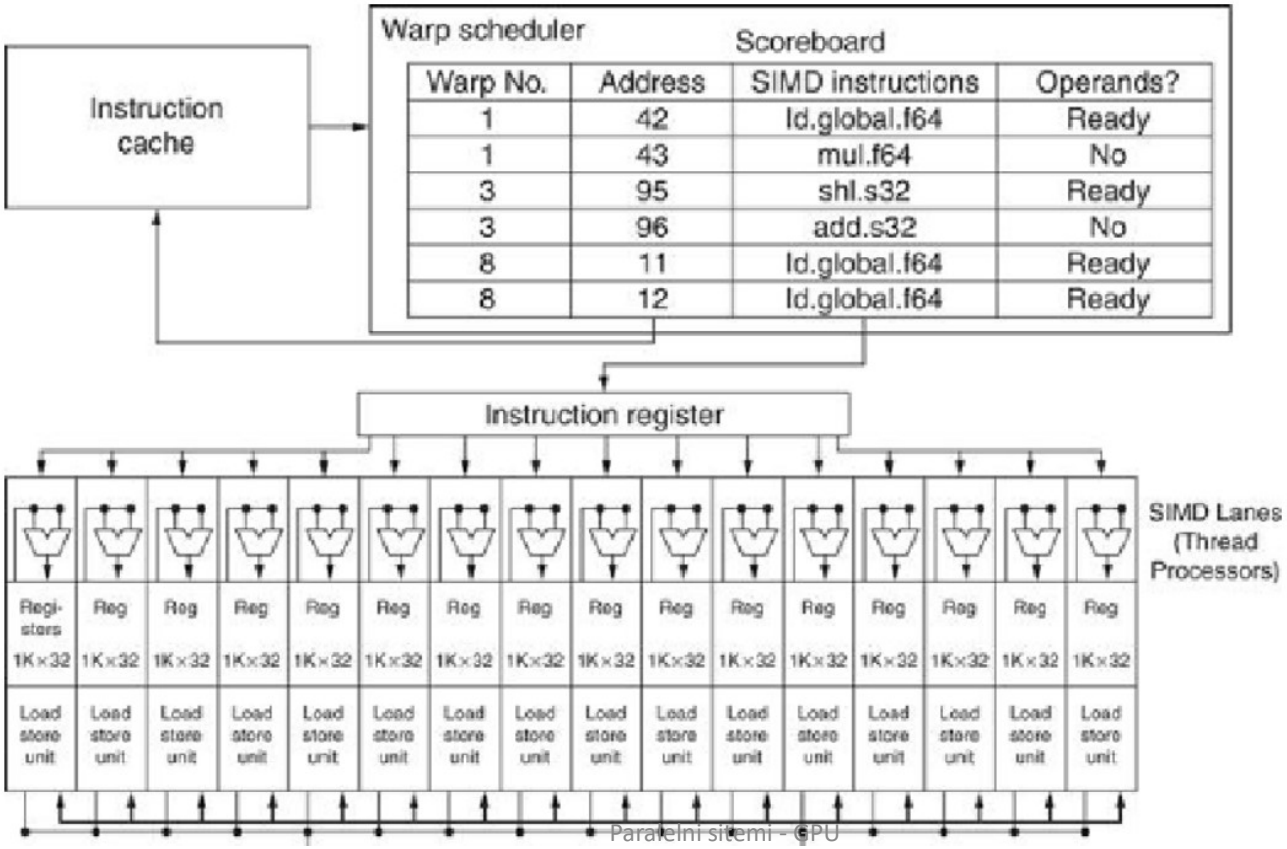
Ovaj koncept je potrebno imati u vidu prilikom pisanja CUDA programa.



Planiranje izvršenja warpova

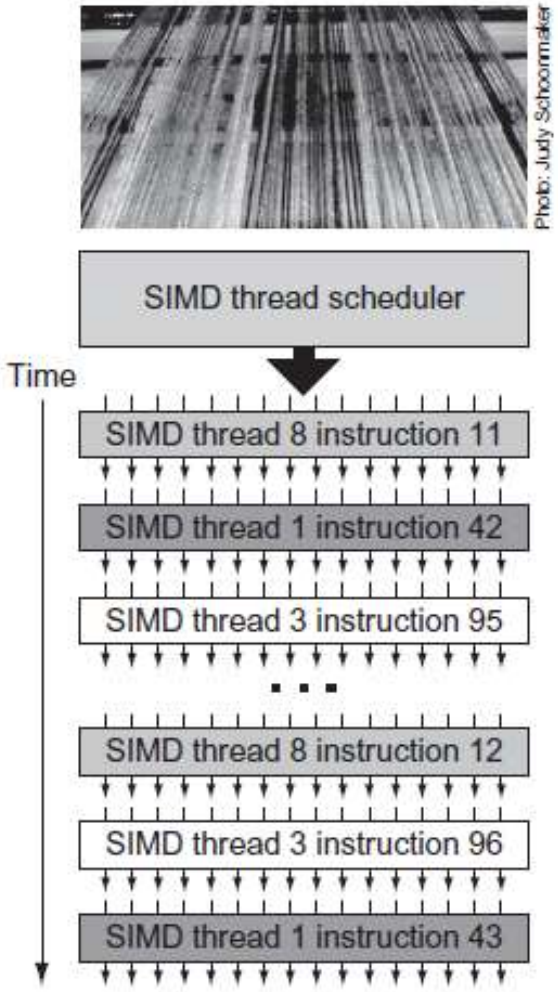
- Svaki put u trenutku izdavanja, Warp scheduler selektuje warp koji je spreman za izvršenje svoje sledeće instrukcije, i zatim izdaje tu instrukciju svim stazama.
- Pošto su po definiciji warpovi nezavisni, Warp Scheduler može izabrati bilo koji warp koji je spreman.
- Warp Scheduler uključuje Scoreboard za praćenje warpova da bi utvrdio koja je SIMD instrukcija spremna za izdavanje (izvršenje)

Planiranje izvršenja warpova



Planiranje izvršenja warpova

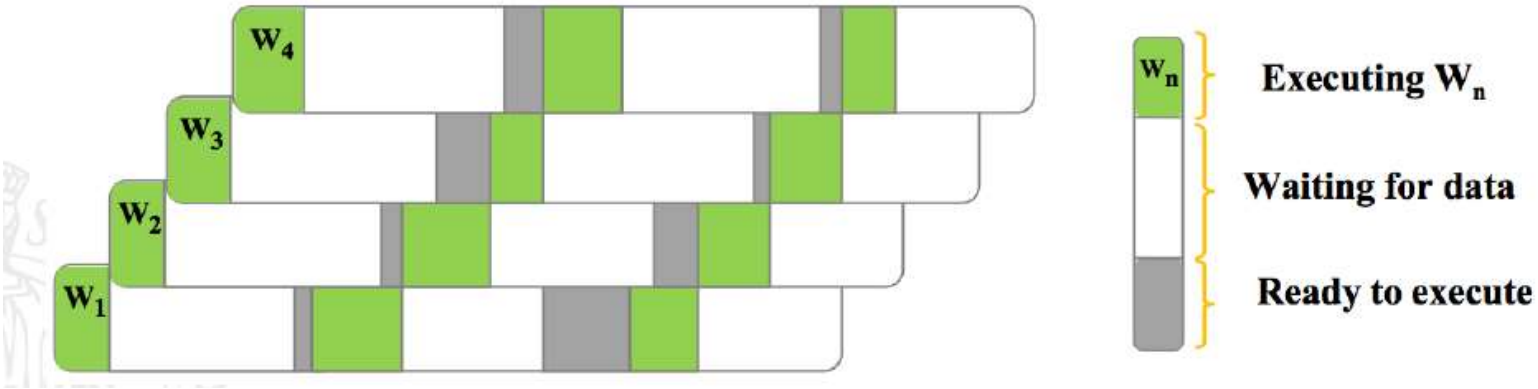
- Latencija memorijskih instrukcija može biti velika, pa je stoga i zahtev Scoreboard-a da utvrdi koje su instrukcije spremne za izvršenje.
- U svakom klok ciklusu, Warp scheduler odlučuje koji warp će da se izvrši sledeći, birajući od onih koji ne čekaju:
  - za podatke koji dolaze iz memorije uređaja (kašnjenje memorije)
  - završetak ranijih instrukcija (pipeline delay)



- Slika prikazuje kako Warp scheduler bira različite warpove koje izdaje tokom vremena.
- Ako sledeća instrukcija u warpu nije spremna za izdavanje (npr. postoji zavisnost u odnosu na prethodnu instrukciju), warp se zaustavlja-instrukcije u istom warpu se izdaju po redu.
- Jedan od warpova koji nije zaustavljen se selektuje u svakom ciklusu i njegova instrukcija se izdaje. Npr. u ciklusu 0 WS izdaje instrukciju iz warpa 8, u ciklusu 1 iz warpa 1 itd.

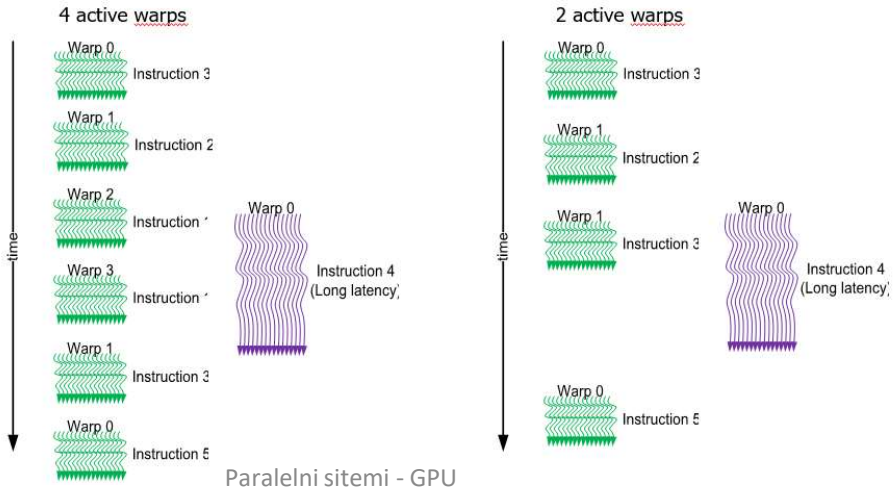
Planiranje izvršenja warpova

- SM može da se prebacuje između Warpova bez vidljvog overhead-a
- Warpovi sa instrukcijama čiji su operandi dostupni su spremni za izvršenje i biće razmatrani prilikom planiranja koji će sledeći warp biti selektovan za izvršenje
- Kad je warp selektovan za izvršenje, sve njegove niti **izvršavaju istu instrukciju paralelno nad različitim podacima**



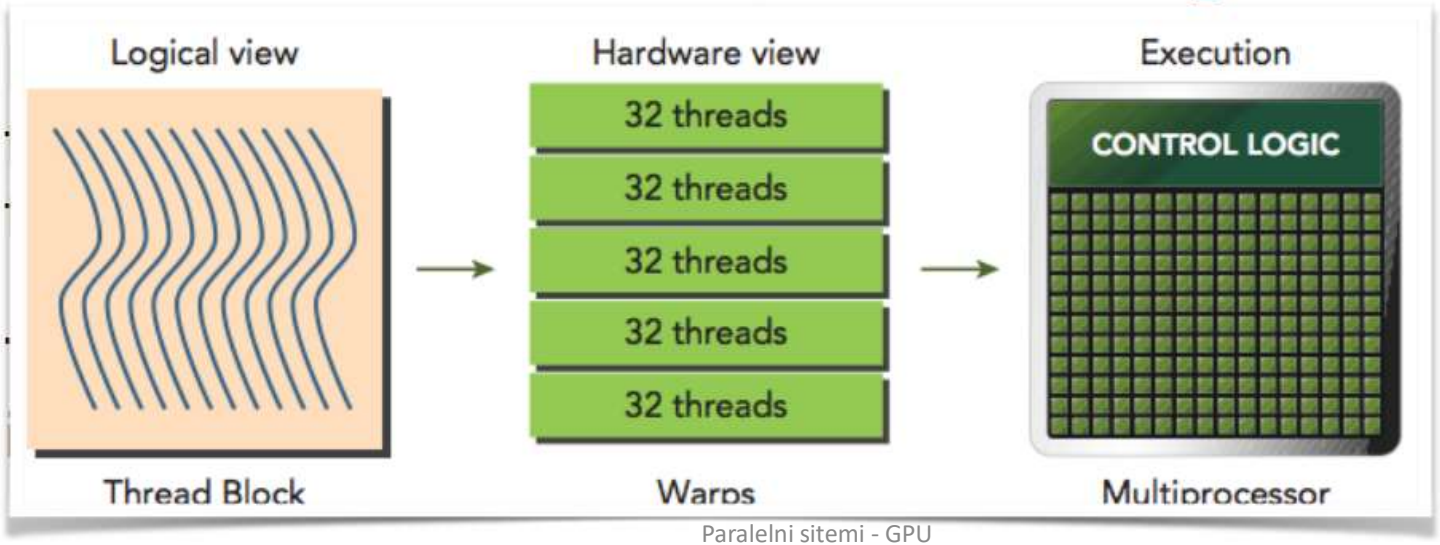
Planiranje izvršenja warpova

- Pretpostavka GPU arhitekata je da GPU aplikacije imaju toliko warpova da višenitnost može sakriti i latenciju prilikom pristupa memoriji i povećati iskorišćenost SM-a
- Nema potrebe za OS promenom konteksta, stanje niti se nalazi i ostaje u registrima (veliki registarski fajl) za svaku nit dok god se izvršava.



Niti i warpovi

- Podela niti u warpove vrši se na osnovu indeksa niti
- Ako je blok organizovan kao jednodimenzionalni niz, particionisanje bloka se obavlja redom : 0-31 nit-prvi warp, 32-63 nit-drugi warp itd.
- Za blokove čija veličina nije deljiva sa 32, poslednji warp se dopunjuje nitima do 32.



Planiranje izvršenja warpova

- **Preferirati veličine blokova niti koje rezultiraju uglavnom punim warpovima:**
- **Bad**: kernel<<<N, 1>>> (...)
- **Okay**: kernel<<<(N+31) / 32, 32>>>(...)
- **Better**: kernel<<<(N+127) / 128, 128>>>(...)
- **Preferirati dovoljan broj niti po bloku kako bi se hardveru pružilo mnogo warpova koji mogu da se prebacuju (sakriva latenciju memorije).**

Izbor veličine bloka niti

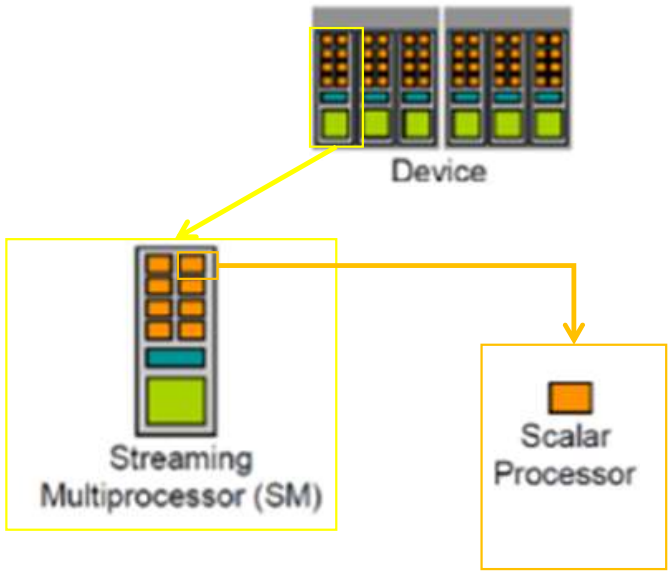
- Pretpostavimo da CUDA uređaj dozvoljava do 8 blokova po SM i 1024 niti po SM-u, pri čemu se primenjuje ograničenje koje prvo postane ograničavajuće.
- Takođe, dozvoljava se maksimalno 512 niti u svakom bloku.
- Da li koristiti blokove niti od 8 × 8, 16 × 16 ili 32 × 32?
- Ako koristimo blokove niti od **8 × 8**, svaki blok će imati samo 64 niti. Potrebno će nam biti 1024/64 = **12 blokova** da potpuno zauzmu jedan SM.
- Međutim, svaki SM može dozvoliti samo do 8 blokova; stoga ćemo završiti sa samo 64 × 8 = **512 niti u** svakom **SM**-u.
- Ovaj ograničeni niti broj implicira da će resursi izvršenja SM-a verovatno biti nedovoljno iskorišćeni jer će biti manje warpova dostupnih za raspoređivanje kod operacija sa dugim latencijama.

Izbor veličine bloka niti

- Blokovi od **16 × 16** rezultiraju sa 256 niti po bloku, što znači da svaki SM može prihvatiti 1024/256 = **4 bloka/SM**.
- Ovaj broj je unutar ograničenja od 8 blokova i dobra je konfiguracija jer će nam omogućiti pun kapacitet niti u svakom SM-u i maksimalan broj warpa za raspoređivanje kod operacija sa dugim latencijama.
- Blokovi od **32 × 32** bi dali 1024 niti u svakom bloku, što premašuje ograničenje od 512 niti po bloku ovog uređaja. Samo blokovi od 16 × 16 dozvoljavaju maksimalan broj niti dodeljenih svakom SM-u.
- Upotreba drugih resursa poput registara i deljene memorije takođe mora biti uzeta u obzir prilikom određivanja najprikladnijih dimenzija bloka niti.

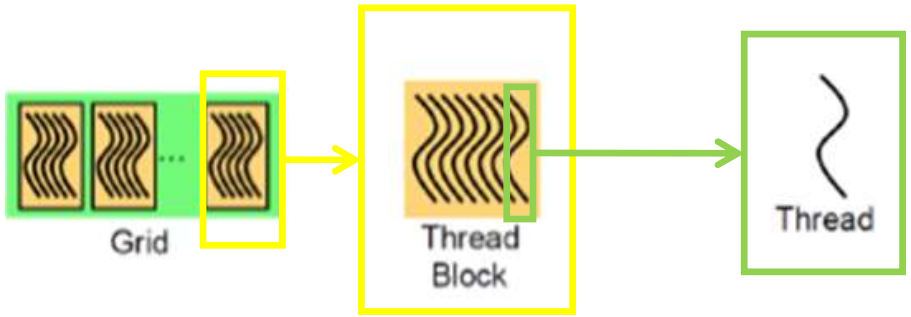
Hardverski pogled na GPU

- **Device** = GPU = skup multiprocesora
- **Multiprocesor** (streaming multiprocessor - SM) = skup procesora & deljive memorije



Softverski pogled na GPU

- **Kernel** = **Program** koji se izvršava na GPU
- **Grid** (Rešetka) = Skup blokova niti koje izvršavaju kernel
- **Blok niti** = Grupa SIMD niti koje izvršavaju kernel i mogu komunicirati preko deljive memorije
- Kernel se izvršava kao grid blokova niti
- Sve niti koje se generišu prilikom poziva kernela čine grid



Programski model (2)

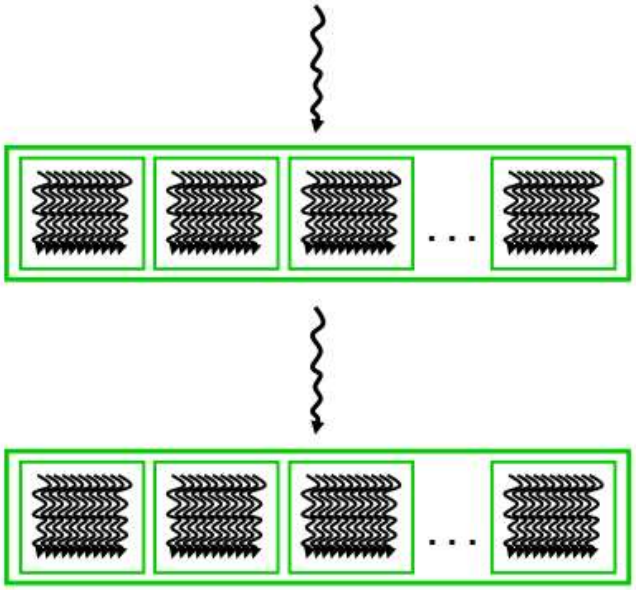
- CUDA program čine integrisani delovi koda za centralni i grafički procesor

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

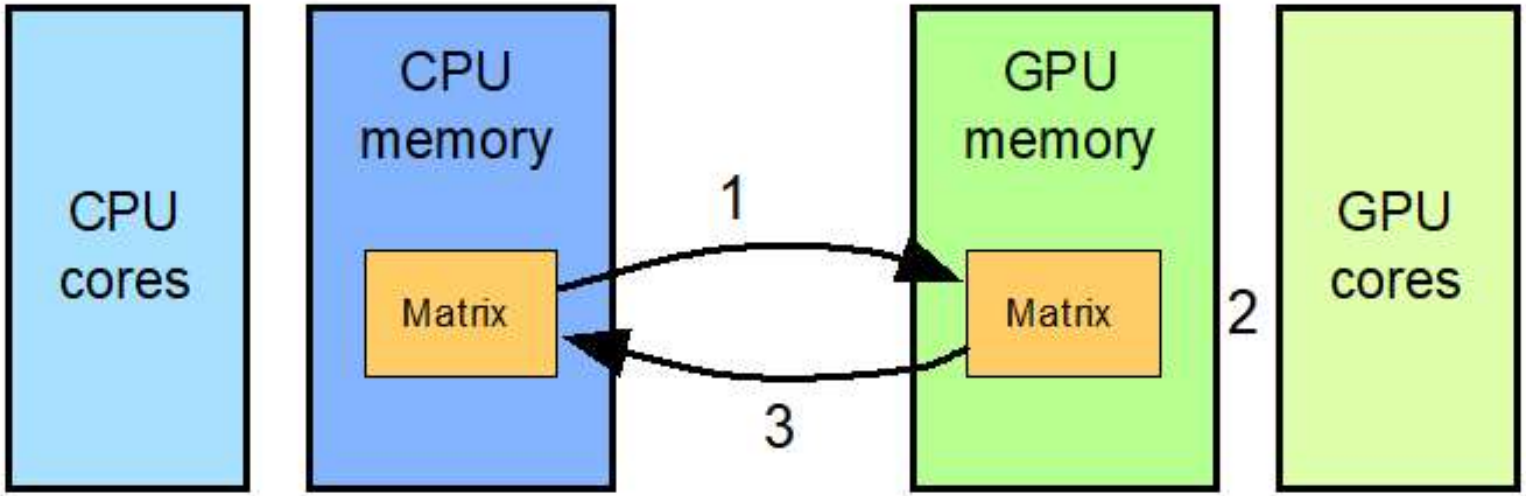


Struktura CUDA programa

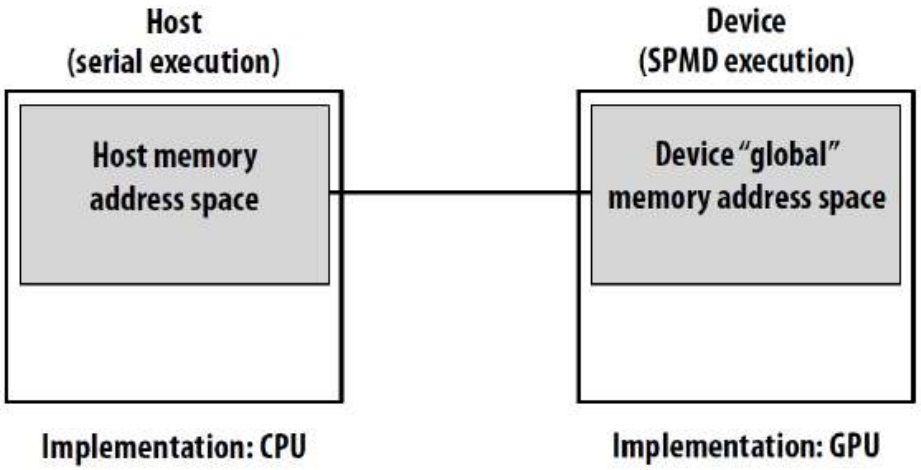
- Tipičan CUDA program se sastoji iz sledećih koraka:

1. Alokacija memorije na hostu i GPU i inicijalizacija podataka na hostu
2. Kopiranje podataka iz memorije hosta u memoriju GPU-a
3. Poziv CUDA funkcije (kernel) da obavi odgovarajuća izračunavanja na GPU
4. Kopiranje podataka iz memorije GPU-a u memoriju hosta
5. Ponavljanje koraka 2-4 po potrebi
6. Oslobađanje memorije i završavanje programa

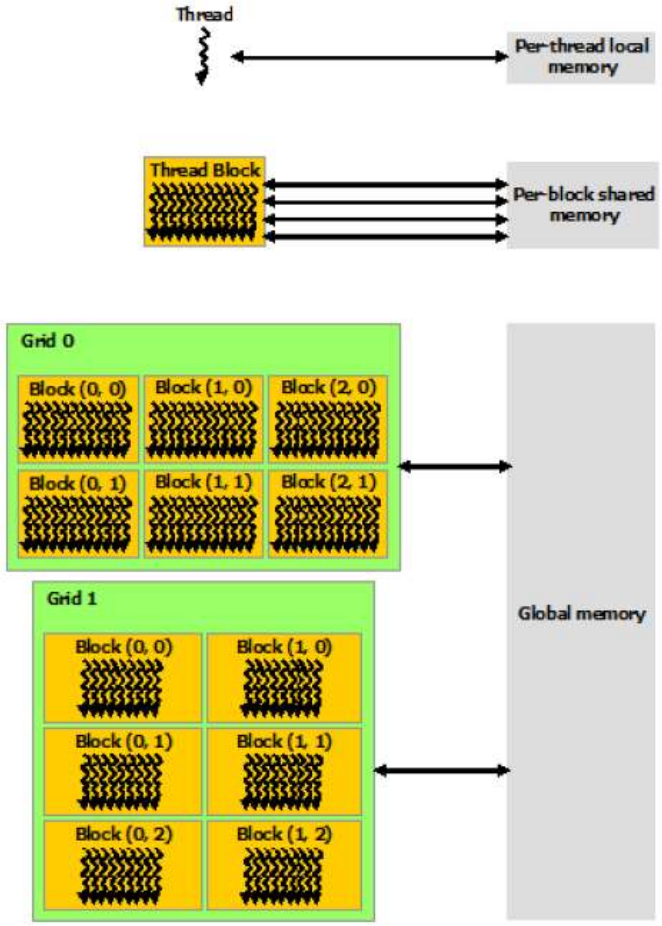
Struktura CUDA programa



Memorija hosta i memorija uređaja



Memorijska hijerarhija



- Ne samo da postoje odvojene memorije računara hosta i GPU uređaja, nego i na GPU uređaju postoji memorijska hijerarhija
- Tipovi memorije koji su vidljivi GPU kodu:
- registarska memorija dostupna samo tekućoj niti (R/W)
- lokalna memorija dostupna samo tekućoj niti (R/W)
- deljiva memorija dostupna svim nitima u bloku (R/W)
- globalna memorija dostupna svim nitima u gridu (R/W)
- konstantna memorija dostupna svim nitima u gridu (R)
- Teksturna memorija dostupna svim nitima u gridu (R)

Memorijska hijerarhija

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories

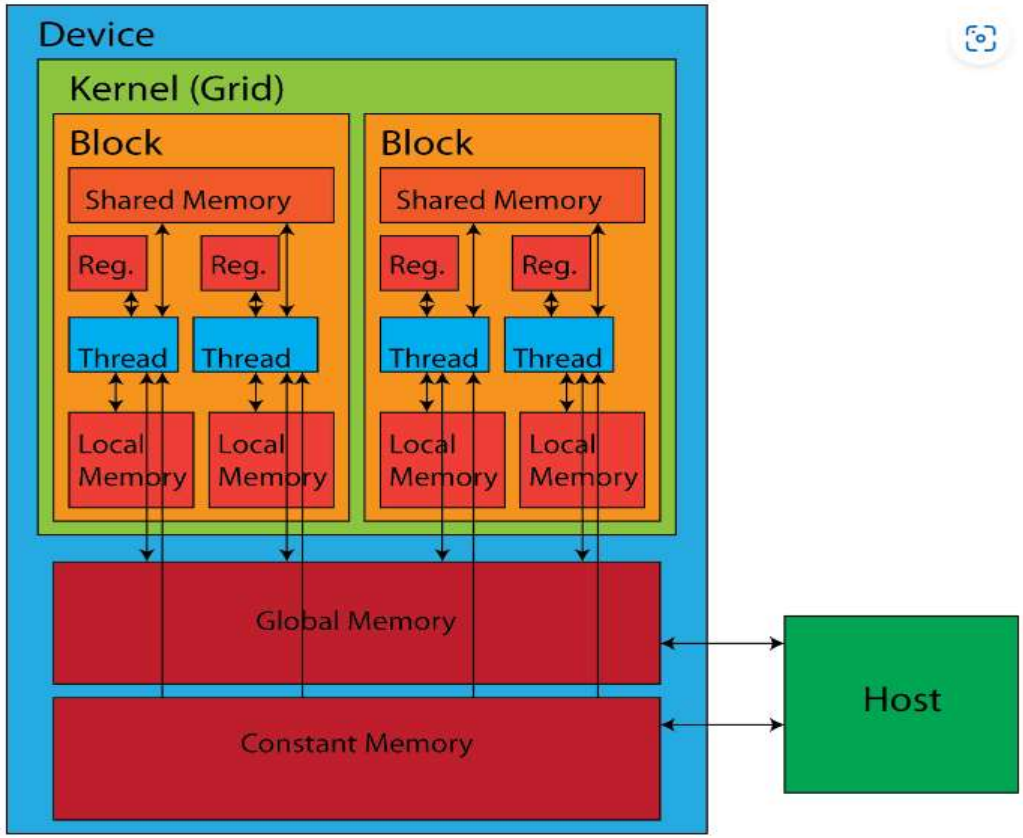


Figure 6

Planiranje izvršenja blokova

- Iako CUDA registri i deljena memorija mogu biti izuzetno efikasni u smanjenju broja pristupa globalnoj memoriji, treba paziti da se ostane unutar kapaciteta ovih memorija.
- Svaki CUDA uređaj nudi ograničene resurse, čime se ograničava broj niti koje istovremeno mogu boraviti u SM-u za određenu aplikaciju.
- Što više resursa svaka nit zahteva, manje niti može boraviti u svakom SM-u, a samim tim i manje niti može istovremeno raditi u celom uređaju.
- Da bismo ilustrovali interakciju između upotrebe registara i nivoa paralelizma koji uređaj može podržati, pretpostavimo da u uređaju aktuelne generacije, svaki SM izvršavati 1536 niti i ima 16.384(16K) registra.
- Iako je 16.384 veliki broj, svaka nit sme koristiti vrlo ograničen broj registara, uzimajući u obzir broj niti koje mogu boraviti u svakom SM-u.

Planiranje izvršenja blokova

- Da bi podržao 1536 niti, svaka nit može koristiti samo 16.384/1536 = 10 registara.
- Pretpostavimo da programer deklariše dodatne dve automatske promenljive u kernelu i poveća broj registara koje koristi svaka nit na 12.
- Ako imamo blokove veličine 16×16 (**256 niti**), **svaki blok** sada zahteva 12**256* = **3072 registra**.
- Pošto je 1536 niti/SM onda je 6 blokova po SM (1536/256)
- Broj registara potrebnih za 6 blokova sada je 6*3072=**18,432**, što premašuje ograničenje registara za dati uređaj.

Planiranje izvršenja blokova

- CUDA runtime sistem se nosi sa ovom situacijom smanjenjem broja blokova dodeljenih svakom SM-u za jedan (sada je broj blokova 5), smanjujući tako broj potrebnih registara na **15,360**. (18432-3072) **što je manje od 16.384**
- Međutim, to smanjuje **broj niti** koje se izvršavaju na SM-u sa 1536 na **1280/SM**. (1536-256)
- Drugim rečima, dodavanjem dve dodatne automatske promenljive, program vidi smanjenje paralelizma u SM-u za 1/6.
- Šta ako su veličine blokova 128 ili 512?

Planiranje izvršenja blokova

- Ako je veličina **bloka 128**, a 1536 niti/SM onda je 12 blokova po SM (1536/128). Ograničenje je 8 blokova/Sm pa imamo 8*128=**1024 niti/SM**
- Broj registara potrebnih za 1024 niti sada je 12,288(1024*12), što je unutar ograničenja registara za dati uređaj, **što je manje od 16.384**.
- Ako je veličina **bloka 512**, a 1536 niti/SM onda je 3 bloka po SM (1536/512). Ograničenje je 8 blokova/Sm pa imamo 3*128=1536 niti/SM
- Broj registara potrebnih za 1536 niti sada je 18432(1536*12), što premašuje ograničenje registara za dati uređaj.

Planiranje izvršenja blokova

- CUDA runtime sistem se nosi sa ovom situacijom smanjenjem broja blokova dodeljenih svakom SM-u za jedan(sada je broj blokova 2), smanjujući tako broj potrebnih registara na 12288. (2*512*12=12288) **što je unutar ograničenja registara za dati uređaj.**
- Međutim, to smanjuje **broj niti** koje se izvršavaju na SM-u sa 1536 na **1024/SM**. (1536-512)

Planiranje izvršenja blokova

- Neka svaki blok niti koristi 2K bajta deljene memorije.
  - U primeru matričnog množenja, svaki blok (veličine 16x16=256) zahteva 16 × 16 × 4 = 1K bajta prostora za Mds. (Svaki element tipa float, što je 4 bajta.) Još 1KB je potrebno za Nds. Dakle, svaki blok koristi 2K bajta deljene memorije.
- Deljena memorija od 16K bajta omogućava da istovremeno boravi 8 blokova u jednom SM-u.
- Budući da je ovo isto kao maksimum koji dozvoljava hardver za niti, deljena memorija nije ograničavajući faktor za ovu veličinu bloka.
- U ovom slučaju, prava ograničenja su ograničenja hardvera za niti koja dozvoljava samo 1536 niti u svakom SM-u.
- Ovo ograničenje ograničava broj blokova u svakom SM-u na šest (1536/256).
- Stoga će se koristiti samo 6*2KB = 12KB deljene memorije.

Planiranje izvršenja blokova

- Ovi limiti se menjaju od jednog uređaja do drugog, ali se mogu odrediti u vreme izvršavanja pomoću upita uređaja.
- Veličina deljene memorije u svakom SM-u takođe može varirati u zavisnosti od uređaja.
- Svaka generacija ili model uređaja može imati različite količine deljene memorije u svakom SM-u.
- Često je poželjno da kernel može koristiti različite količine deljene memorije u skladu sa raspoloživom količinom u hardveru.

Alokacija memorije i prenos podataka

- Alokacija memorije na strani hosta (domaćina) se vrši statički ili standardnim C pozivima
- Alokacija memorije na uređaju se vrši putem odgovarajućih poziva API funkcija
- **cudaMalloc()**
  - Alocira objekat u globalnoj memoriji uređaja
  - Zahteva dva parametra
    - Adresu pointera na alocirani objekat
    - Veličinu alociranog objekta u bajtovima
    - cudaMalloc((void **)&d_x, nbytes);
- **cudaFree()**
  - Oslobađa objekat iz memorije uređaja
  - Zahteva pokazivač na objekat

Memorijski transferi

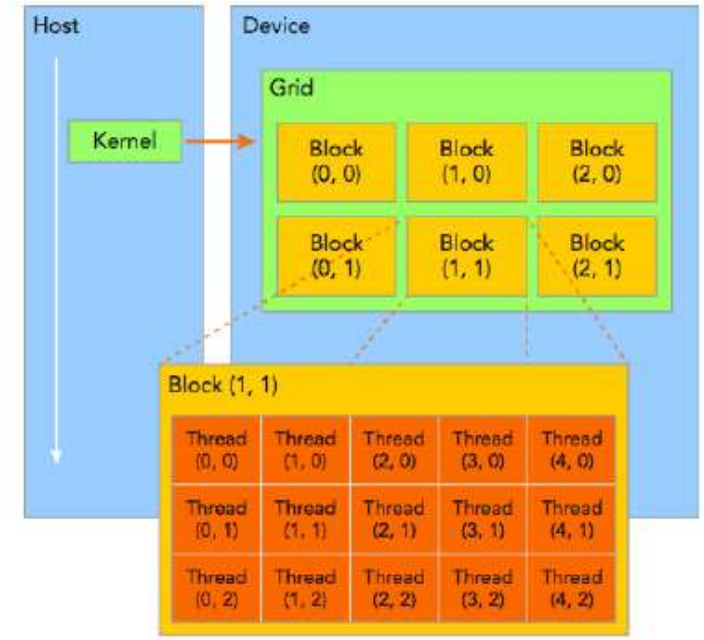
- Za prenos podataka između domaćina i uređaja, kao i unutar samog uređaja postoje odgovarajući pozivi
- **cudaMemcpy()**
  - Obavlja memorijske transfere
  - Zahteva četiri parametra:
    - Pokazivač na odredište
    - Pokazivač na izvor
    - Veličinu podataka koji se prenose u bajtovima
    - Tip transfera
    - cudaMemcpy(h_x, d_x, nbytes, cudaMemcpyDeviceToHost);
- Tipovi transfera
  - Host to Host(cudaMemcpyHostToHost)
  - Host to Device(cudaMemcpyHostToDevice)
  - Device to Host(cudaMemcpyDeviceToHost)
  - Device to Device(cudaMemcpyDeviceToDevice)
- Poziv cudaMemcpy() je sinhron
  - Kontrola se vraća CPU nakon što se kopiranje završi
  - Kopiranje startuje nakon što svi prethodni CUDA pozivi budu kompletirani

CUDA ekstenzije - funkcije

- CUDA program čine integrisani delovi koda za centralni i grafički procesor
- Kako prepoznati koja se funkcija gde izvršava?
  - Kvalifikator **__global__** označava kernel funkcije
  - Kvalifikator **__host__** označava funkcije koje se izvršavaju samo na strani domaćina
  - Kvalifikator **__device__** označava funkcije koje se izvršavaju samo na strani uređaja

	Izvšava:	**Poziva:**
__device__ float deviceFunc()	uređaj	uređaj
__global__ void kernelFunc()	uređaj	domaćin
__host__ float hostFunc()	domaćin	domaćin

Hijerarhija niti



Kada se funkcija **kernela** pokrene sa strane **hosta**, izvršenje je premešteno na uređaj(GPU) gde se generiše veliki broj niti i svaka nit izvršava naredbe navedene od strane kernel funkcije .

- CUDA obezbeđuje apstrakciju hijerarhije niti da bi vam omogućila da organizujete svoje niti. Ovo je hijerarhija niti na dva nivoa, razložene na blokove niti i mreže blokova

Programski model (4)

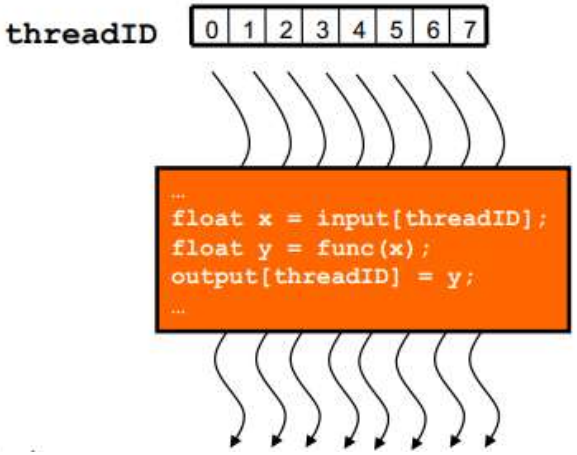
- Na nižem nivou, kada se pokrene jedan blok na SM, njega izvršava veći broj niti koje imaju pristup:
  - Promenljivama koje su prosleđene kao argumenti
  - Pokazivačima na nizove na device-u
  - Globalnim konstantama u memoriji device-a
  - Deljivoj memoriji i privatnim registrima/lokalnim promenljivama
  - Specijalnim promenljivama:
    - *gridDim* – veličina (dimenzije) grida blokova (broj blokova u gridu)
    - *blockDim* – veličina (dimenzije) svakog od blokova (broj niti u bloku)
    - *blockIdx* – indeks (ili 2D/3D indeksi) bloka
    - *threadIdx* – indeks (ili 2D/3D indeksi) niti
    - *warpSize* – uvek 32 za sada, ali može biti promenjen u budućnosti

Programski model (5)

- Kernel kod:
  - Napisan je iz ugla jedne niti
    - Razlikuje se od OpenMP multithreading-a
    - Sličan je modelu koji ima MPI gde se “rank” koristi za identifikaciju MPI procesa
    - Sve lokalne promenljive su privatne za nit
  - Potrebno je obratiti pažnju na to gde svaka od promenljivih “živi”:
    - Svaka operacija koja uključuje podatke koji se nalaze u memoriji uređaja zahteva da isti budu prebačeni u/iz GPU registara
    - Često je bolje kopirati vrednost u lokalnu registarsku promenljivu

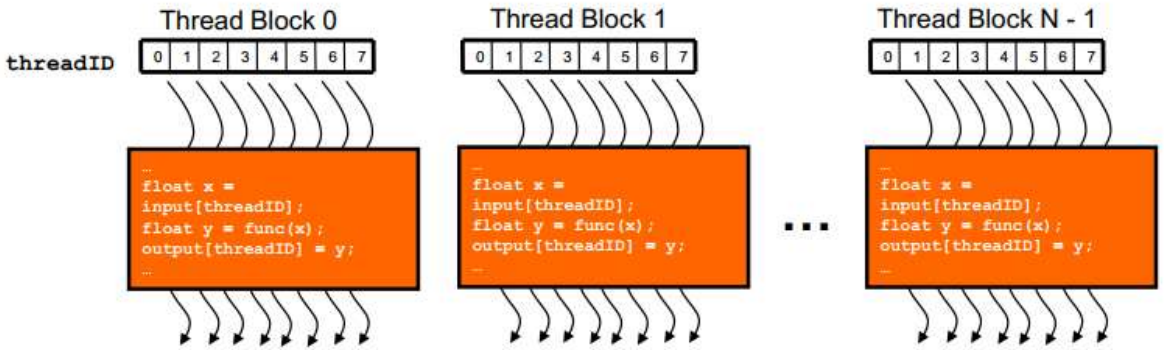
Izvršni model (1)

- CUDA kernel (jezgro) se izvršava pomoću niza niti raspoređenih u odgovarajuću rešetku (grid)
  - Sve niti izvršavaju isti kod
    - SIMD/SPMD/SIMT model izvršavanja
  - Svaka nit ima jedinstveni identifikator (indeks) koji koristi da bi vršila pristup memoriji i donosila odluke



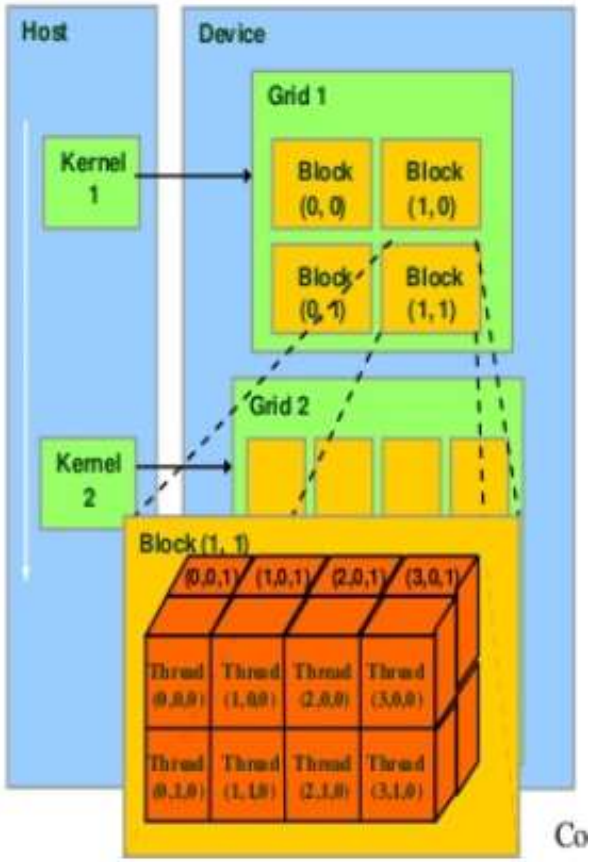
Niti u bloku

- Niti unutar grida(rešetke) su podeljene u nezavisne blokove
  - Svaki blok ima jedinstven identifikator unutar rešetke
  - Niti unutar istog bloka mogu da sarađuju
    - Koristeći sinhronizaciju, atomične operacije i deljenu memoriju
  - Niti iz različitih blokova ne mogu da sarađuju
  - Na ovaj način se omogućava transparentno skaliranje na bilo koji broj procesora



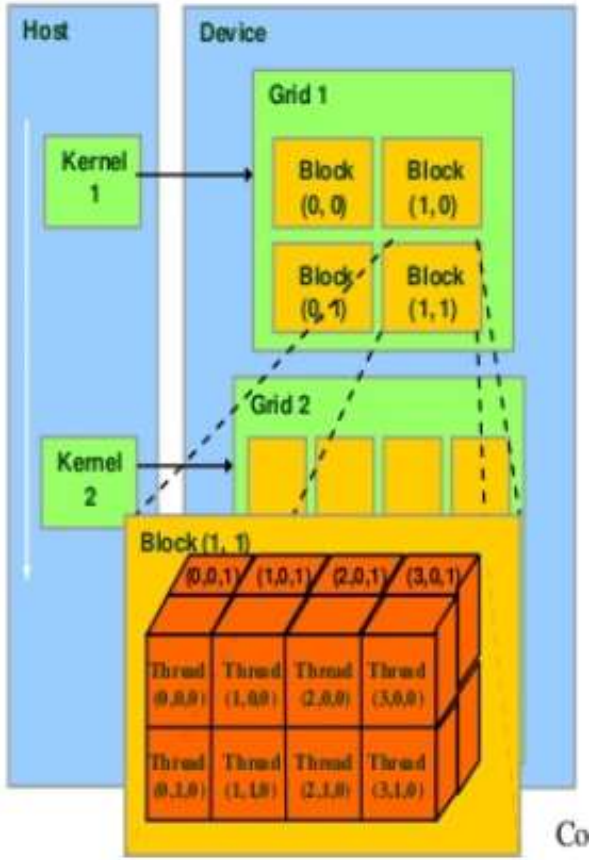
Hijerarhija niti

- Kernel se konfiguriše prilikom svakog poziva
  - Zadaju se dimenzije bloka i rešetke
  - Blok i rešetka mogu biti višedimenzionalni
    - 1D, 2D ili 3D
- Niti i blokovi imaju identifikatore (indekse)
  - Tako da mogu da odluče nad kojim podacima da rade



Hijerarhija niti

- Za svaki blok se može odrediti indeks unutar rešetke
  - Block ID: 1D, 2D, 3D
  - ***blockIdx*** promenljiva
- Za svaku nit se može odrediti indeks unutar bloka
  - Thread ID: 1D, 2D, 3D
  - ***threadIdx*** promenljiva
- Pojednostavljuje pristup memoriji pri obradi višedimenzionalnih struktura
  - Obrada slika i sl.



CUDA funkcije – kernel (1)

- Funkcije kernela imaju sledeće osobine
  - Definišu se kvalifikatorom **__global__**
  - Moraju biti **void funkcije**
  - Parametri jezgra mogu biti skalarni podaci ili pokazivači na podatke alocirane na uređaju

```
__global__
void vecAdd( int *devA, int *devB, int *devC, int n);
```

CUDA funkcije – kernel (2)

- Kernel mora biti pozvan pomoću odgovarajuće izvršne konfiguracije
  - Zadaje se pomoću sintaksne ekstenzije jezika C, pomoću trostrukih zagrada <<< i >>>

    myKernel<<< n, m >>>(arg1, …);
- Parametri n i m definišu organizaciju blokova niti na nivou grida i niti na nivou bloka
- Postoje još dva opciona parametra
  - Za eksplicitno rezervisanje deljene memorije na nivou bloka
  - Za upravljanje tokovima (streams)
- Svaki poziv kernela je asinhron
  - Kontrola se odmah vraća centralnom procesoru
  - Kernel se izvršava nakon što su svi prethodni CUDA pozivi kompletirani

CUDA funkcije - ograničenja

- **__device__** funkcijama se ne može uzeti adresa
  - One se najčešće implementiraju kao inline funkcije
- Za funkcije koje se izvršavaju na uređaju:
  - Ograničeno dozvoljena rekurzija
    - Hardversko ograničenje – stek u deljenoj memoriji
    - Od Fermi arhitekture GPU-ova
- Nije dozvoljeno deklarisanje statičkih promenljivih unutar funkcije
- Nisu dozvoljene funkcije sa varijabilnim brojem argumenata
  - Funkcije poput printf(...)

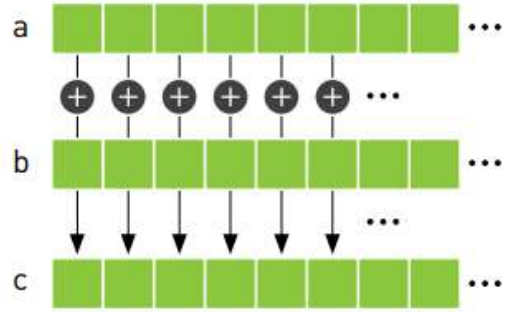
Sabiranje vektora – Tradicionalni C kod

```
#include "book.h“
#define N 10

void add(int* a, int* b, int* c);

int main( void )
{
    int a[N], b[N], c[N];

    // popunjavanje nizova
    for (int i=0; i<N; i++)
    {
         a[i] = -i;
         b[i] = i * i;
    }
    add( a, b, c );
    // štampanje rezultata
    for (int i=0; i<N; i++)
    {
         printf( "%d + %d = %d\n", a[i], b[i], c[i] );
```

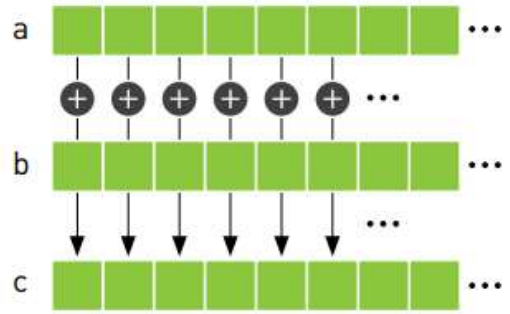


Sabiranje vektora – Tradicionalni C kod

```
#include "book.h"
#define N 10

void add(int* a, int* b, int* c)
{
   for (i=0; i < N; i++)
   {
         c[i] = a[i] + b[i];
   }
}
// šta ako na raspolaganju imamo više CPUova?
void add2(int* a, int* b, int* c)
{
    int tid = 0; // CPU 0

    while (tid < N)
    {
        c[tid] = a[tid] + b[tid];
        tid += BROJ_CPUova; // ako imamo samo jedan CPU - ide +1
```



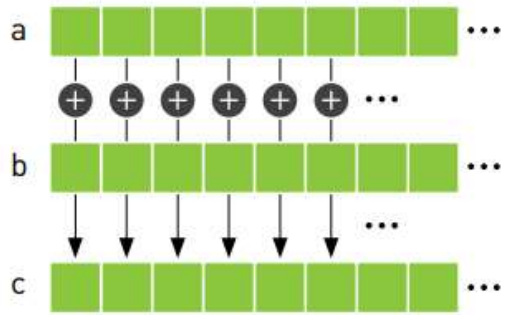
Sabiranje vektora – Tradicionalni C kod

CPU CORE 1

```
void add( int *a, int *b, int *c )
{
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

CPU CORE 2

```
void add( int *a, int *b, int *c )
{
    int tid = 1;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```



Sabiranje vektora – GPU kod (1)

```
#define N 10
int main( void )
{
  int a[N];
  int b[N];
  int c[N];
  int* dev_a;
  int* dev_b;
  int* dev_c;

  // alokacija memorije na GPU
  HANDLE_ERROR(cudaMalloc((void**)& dev_a, N * sizeof(int)));
  HANDLE_ERROR(cudaMalloc((void**)& dev_b, N * sizeof(int)));
  HANDLE_ERROR(cudaMalloc((void**)& dev_c, N * sizeof(int)));

  // inicijalizacija nizova a i b, na CPU
  for (int i=0; i<N; i++)
  {
      a[i] = -i;
      b[i] = i * i;
  }
  ...
```

Sabiranje vektora – GPU kod (2)

```
...
// kopiranje nizova a i b na GPU
HANDLE_ERROR(cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice));
HANDLE_ERROR(cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice));

add<<<N,1>>>(dev_a, dev_b, dev_c);

// kopiranje niza c sa GPU na CPU
HANDLE_ERROR(cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost));

// prikaz rezultata
for (int i=0; i<N; i++)
{
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}

// oslobađanje GPU memorije
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Sabiranje vektora – GPU kod (3)

```
__global__ void add( int *a, int *b, int *c )
{
   // Obrada podataka sa tid indeksom
   int tid = blockIdx.x;

   if (tid < N)
       c[tid] = a[tid] + b[tid];
}
```

- Izvršavanje kernela - broj
- Predefinisane promenljive

Sabiranje vektora – GPU kod (4)

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Sabiranje vektora – 1 blok, više niti (1)

Izmene:

- Blokovi i niti
  - Ranije smo kreirali N blokova, svaki sa 1 niti
    - add<<<N,1>>>(dev _ a, dev _ b, dev _ c);
  - Sad hoćemo 1 blok, svaki sa po N niti
    - add<<<1,N>>>(dev _ a, dev _ b, dev _ c);
- Indeksiranje podataka
  - Koristili smo *blockIdx*
    - **int** tid = blockIdx.x;
  - Sada je vrednost *blockIdx* ista za svaku nit, pa koristimo *threadIdx*
    - **int** tid = threadIdx.x;

Sabiranje vektora – 1 blok, više niti (2)

```
#include "../common/book.h“
#define N 10
int main( void )
{
  int a[N];
  int b[N];
  int c[N];
  int* dev_a;
  int* dev_b;
  int* dev_c;

  // alokacija memorije na GPU
   HANDLE_ERROR(cudaMalloc((void**)& dev_a, N * sizeof(int)));
   HANDLE_ERROR(cudaMalloc((void**)& dev_b, N * sizeof(int)));
   HANDLE_ERROR(cudaMalloc((void**)& dev_c, N * sizeof(int)));

  // inicijalizacija nizova a i b, na CPU
   for (int i=0; i<N; i++)
  {
       a[i] = -i;
       b[i] = i * i;
   }
  ...
```

Sabiranje vektora – 1 blok, više niti (3)

```
...
// kopiranje nizova a i b na GPU
HANDLE_ERROR(cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice));
HANDLE_ERROR(cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice));

add<<<1,N>>>(dev_a, dev_b, dev_c);

// kopiranje niza c sa GPU na CPU
HANDLE_ERROR(cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost));
// prikaz rezultata
for (int i=0; i<N; i++)
{
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}
// oslobađanje GPU memorije
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Sabiranje vektora – 1 blok, više niti (4)

```
__global__ void add( int *a, int *b, int *c )
{
   // Obrada podataka sa tid indeksom
   int tid = threadIdx.x;

   if (tid < N)
       c[tid] = a[tid] + b[tid];
}
```

- Izvršavanje kernela - broj
- Predefinisane promenljive

Sabiranje vektora – nedostaci

- Broj blokova koje je moguće pokrenuti u jednoj dimenziji u pozivu je hardverski ograničen na 65 535.
- Broj niti u bloku je takođe ograničen - 1024
- Kako sabrati vektore koji su mnogo veći?
  - Kombinacija blokova i niti

Sabiranje vektora – više blokova, više niti (4)

Izmene:

- Blokovi i niti
  - Sad imamo više blokova i više niti
  - I dalje nam treba N niti, ali ih je potrebno podeliti u više blokova
  - Proizvoljno, možemo uzeti 128 niti po bloku (bitno da bude manje od maksimalnog broja niti po bloku)
    - add<<<(N+127)/128,128>>>(dev _ a, dev _ b, dev _ c);
    - Ovakav poziv pokreće više niti nego što je potrebno
- Indeksiranje podataka
  - Koristili smo *blockIdx,* i *threadIdx*
  - Sada indeksiranje izgleda kao konverzija dvodimenzionalnog indeksa u jednodimenzionalni:
    - int tid = threadIdx.x + blockIdx.x * blockDim.x;
  - Varijabla *blockDim* čuva broj niti po svakoj dimenziji u bloku

Sabiranje vektora – nedostaci

- Šta kada je broj potrebnih niti veći od maksimalnog broja niti?
  - max_broj_niti = max_broj_blokova * max_broj_niti_u_bloku
- Rad sa toliko velikim nizovima nije neuobičajen: današnje kartice imaju dovoljno memorije
  - GeForce GTX 1080 ima 8GB

- Neophodna je izmena kernela

Sabiranje vektora – veliki vektori

```
__global__ void add(int* a, int* b, int* c)
{
    int tid = threadIdx.x +
                  blockIdx.x * blockDim.x;

    while (tid < N)
    {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

Sličnosti sa CPU implementacijom?

blockDim.x * gridDim.x – broj niti

Performanse i ograničenja

- Glavna uska grla (bottlenecks)
  - Pristup globalnoj memoriji
  - CPU<->GPU prenosi podataka
- Pristup memoriji
  - Sakrivanje latencije
  - Coalesced pristup memoriji
  - Ponovno korišćenje podataka (Data reuse)
    - Korist kod deljive memorije
- SIMD (Warp) iskorišćenost: Divergencija niti
- Atomične operacije: Serializacija izvršenja
- Prenos podataka između CPU i GPU
  - Preklapanja komunikacije i izračunavanja (upotreba tokova podataka (streams))