



PARALELNI SISTEMI: **CUDA**



MSc Aleksandra Stojnev
Prof. Dr. Natalija Stojanović

Literatura

- [1] David Kirk, Wen-mei Hwu: *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann
 - [2] Jason Sanders, Edward Kandrot: *CUDA by example: an introduction to general-purpose GPU programming*, Addison-Wesley Professional.
- ❖ Obratite pažnju na beleške ispod slajdova! U njima se nalaze dodatna detaljnija objašnjenja slajdova i/ili reference na poglavlja iz [1] i [2], u cilju boljeg razumevanja.


Uvod u GPGPU (1)

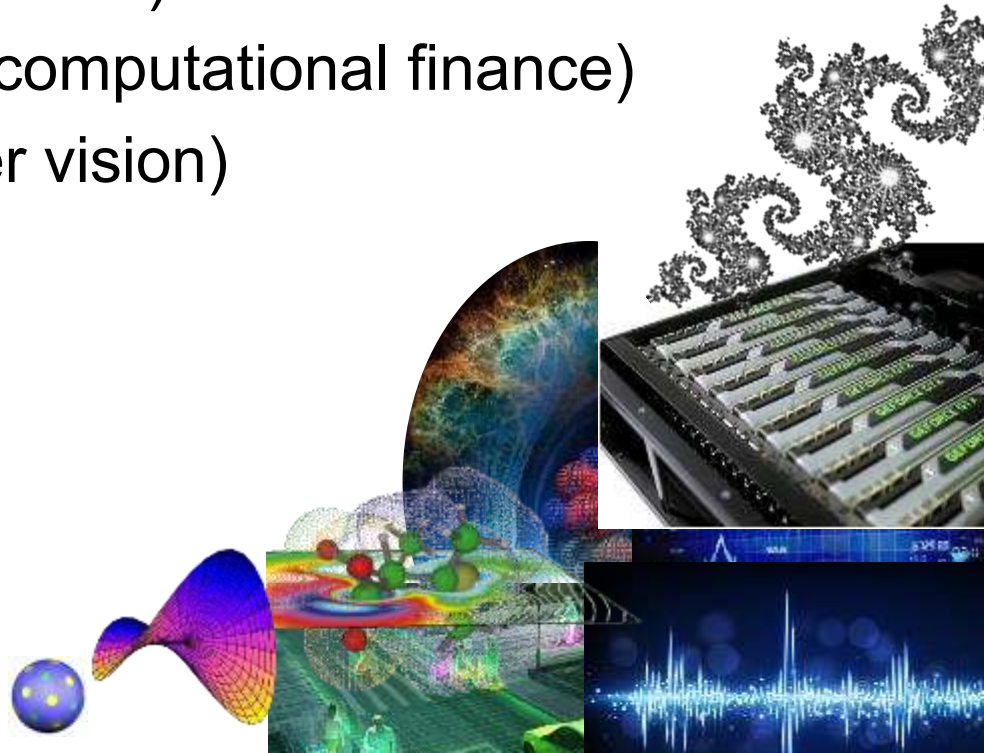
- Razvoj 3D grafike (i narastajuća industrija video igara) izvršio je veliki pritisak na razvoj grafičkih procesora, pa su vremenom isti evoluirali u paralelne i visokoprogramabilne procesore.
- Grafički procesori (GPU) su specijalizovani za računski intenzivna, visoko paralelna izračunavanja, i inicijalno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



Uvod u GPGPU (2)

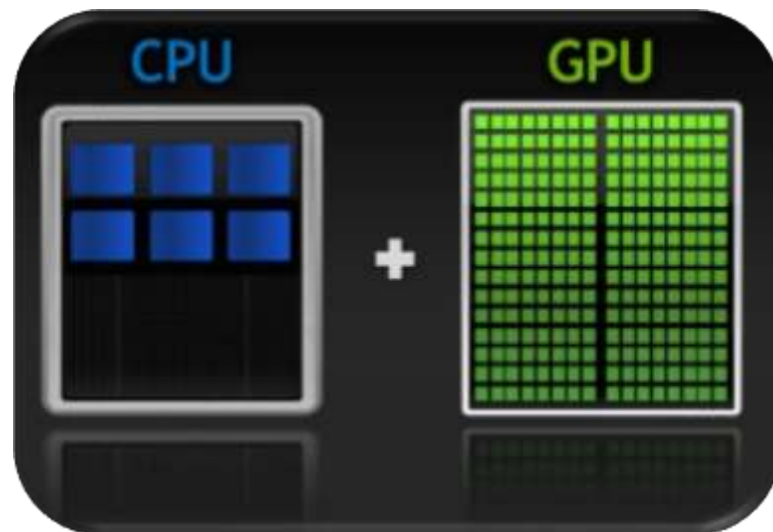
Širok spektar primena:

- Fizičke simulacije (computational physics)
 - Hemijske simulacije (computational chemistry)
 - Biološke simulacije (life sciences)
 - Finansijska izračunavanja (computational finance)
 - Računarska vizija (computer vision)
 - Obrada signala
 - Geometrija i matematika
 - Baze podataka
- 
- A decorative graphic in the bottom right corner featuring a vibrant, colorful image of a galaxy or nebula, transitioning from blue and green to orange and red, next to a partial view of a black server rack.



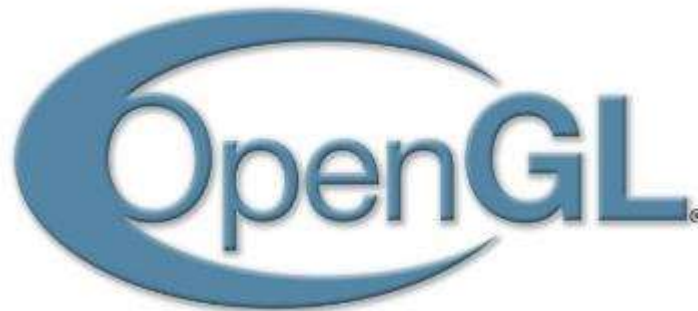
Uvod u GPGPU (3)

- 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 (obrađa velike količine podataka)



Legacy GPGPU Model: HW & SW

- Programiranje samo korišćenjem grafičkog API-ja
 - Nepovoljna kriva učenja
 - Restrikcije u pristupu memoriji
 - Limitirani DRAM bandwidth



CUDA - Compute Unified Device Architecture

- Arhitektura za upravljanje izračunavanjem opšte namene na grafičkim procesorskim jedinicama, dostupna na NVIDIA grafičkim karticama serije 8000 i novije
- Dolazi sa softverskim okruženjem koje omogućava developerima da koriste C kao viši programski jezik.
- Podržani su i drugi jezici, API i “directives-based” pristupi



CUDA komponente

- Driver
 - Low-level softver koji kontroliše grafičku karticu
- Toolkit
 - Nvcc CUDA kompajler
 - Nsight IDE plugin za Eclipse ili Visual Studio
 - Alati za profajliranje i debugging
 - Različite biblioteke
- SDK
 - Mnogo primera
 - Error-checking
 - Zvanično nije podržan od NVIDIA
 - Skoro pa bez dokumentacije

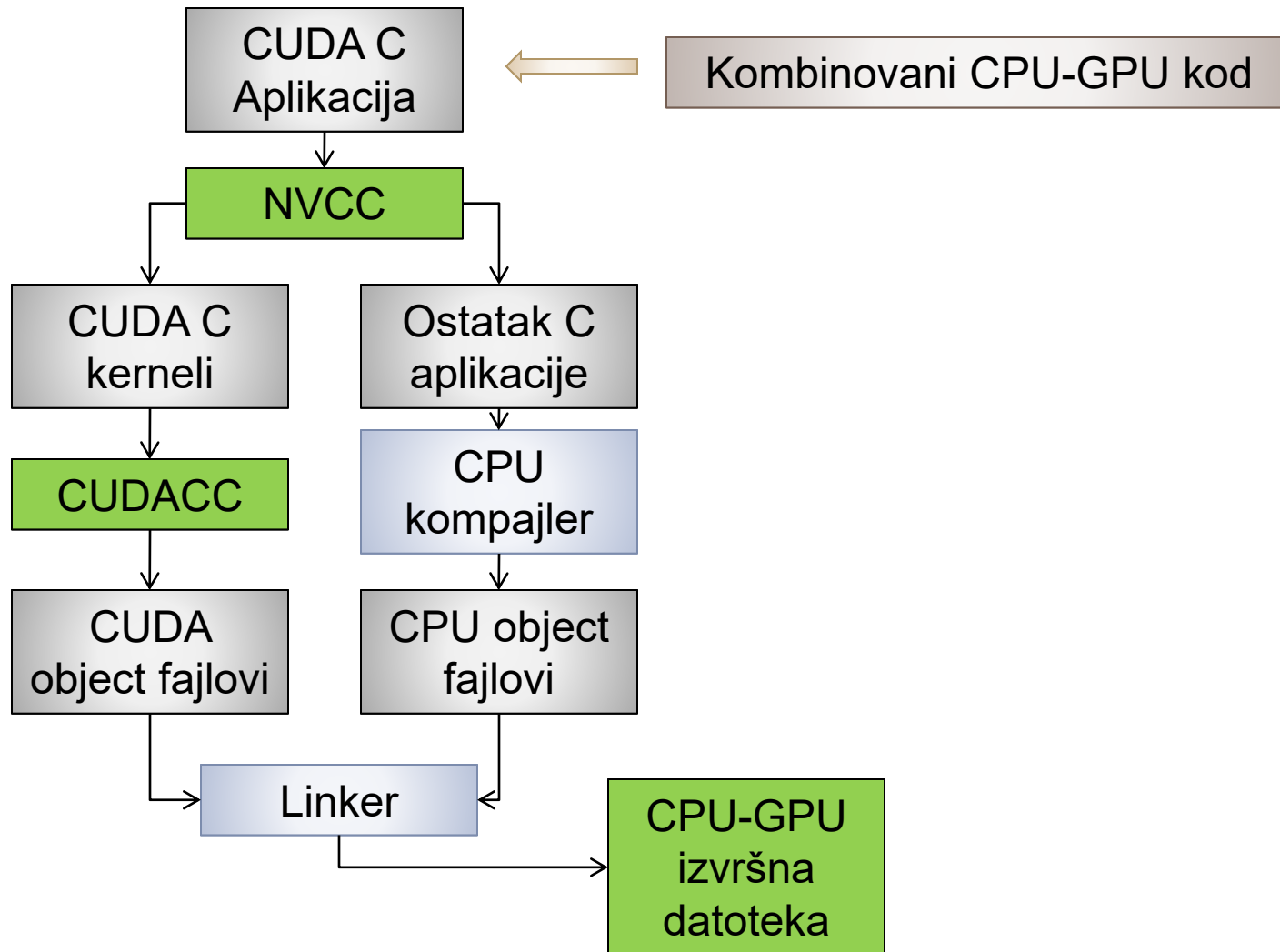
CUDA program

- CUDA program čine integrisani delovi koda za centralni i grafički procesor
 - Izvorni kod sadrži CUDA ekstenzije kojima se specificira koji delovi koda se kako i gde izvršavaju
- Prevođenje CUDA programa zahteva i prevodioca koji je u stanju da generiše kod koji se izvršava na centralnom procesoru i kod koji se izvršava na grafičkom procesoru

Prevođenje CUDA programa (1)

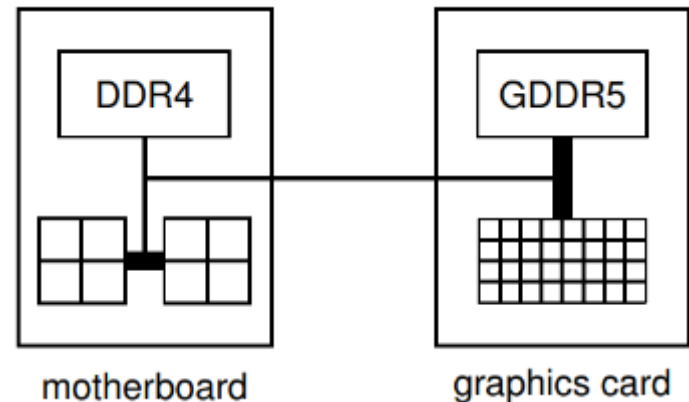
- Bilo koji izvorni kod koji sadrži CUDA ekstenzije mora se prevesti pomoću nvcc prevodioca
- NVCC je prevodilac-omotač (compiler driver)
 - Radi tako što poziva sve nepohodne alate i prevodioce
 - cudacc, g++, cl, ...
- Izlazi NVCC prevodioca su:
 - C kod koji se izvršava na strani domaćina (CPU kod) i koji se mora dalje prevesti odgovarajućim prevodiocem
 - PTX (Parallel Thread eXecution) kod
 - Predstavlja neku vrstu međukoda za grafički procesor
- Bilo koji program koji sadrži CUDA pozive, zahteva sledeće dve dinamičke biblioteke:
 - CUDA runtime biblioteku (**cuda**)
 - CUDA core biblioteku (**cuda**)

Prevođenje CUDA programa (2)



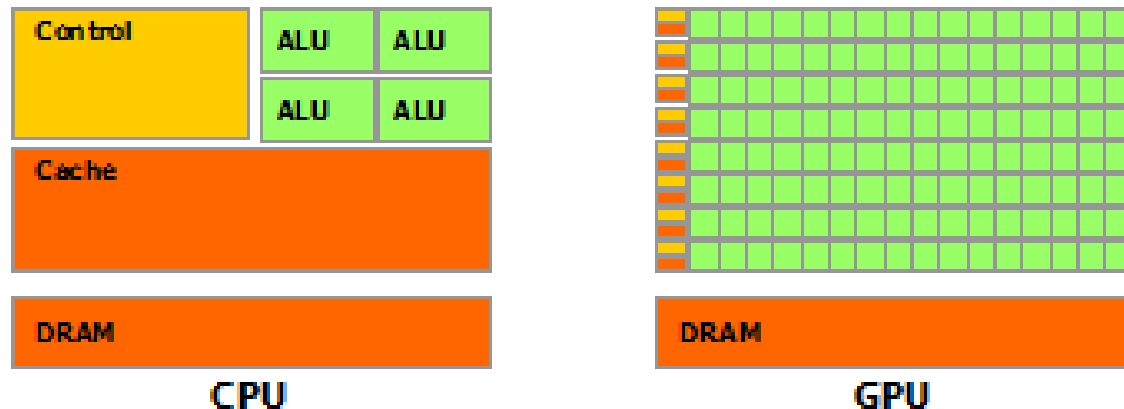
Hardverski pogled na GPU (1)

- *Massively-Parallel Many Core Architecture*
- Podrazumeva veliki broj *streaming* multiprocesora (SM) koji sami po sebi nisu preterano moćni, ali kombinovani sa dobro napisanim paralelnim kodom imaju izuzetno veliku moć računanja



CPU vs GPU (2)

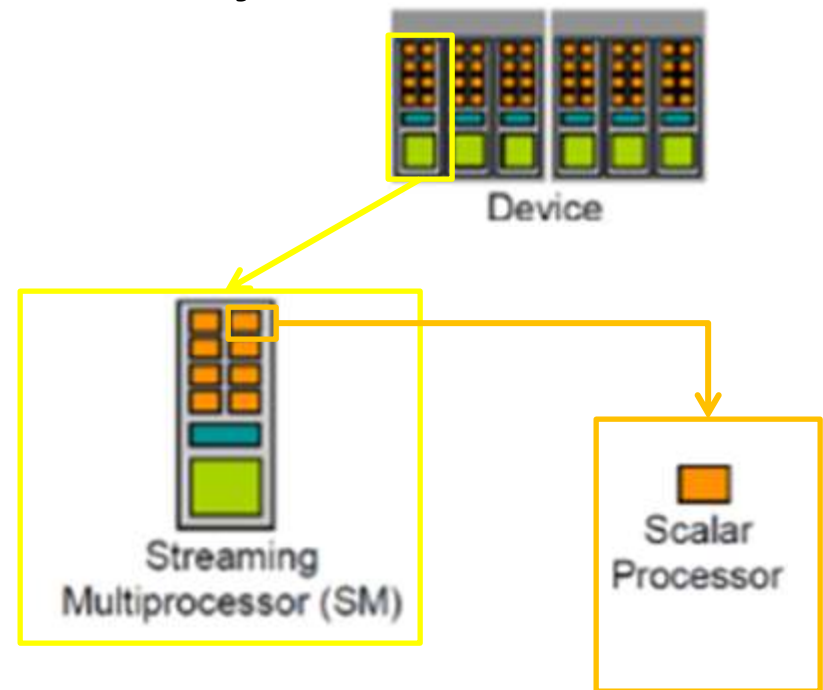
- Fundamentalna razlika između CPU i GPU je u njihovom dizajnu:



- CPU je orijentisan ka tradicionalnom izvršenju poslova, a GPU ka obradi podataka
- Kod GPU mnogo više tranzistora je namenjeno obradi podataka nego keširanju i kontroli toka

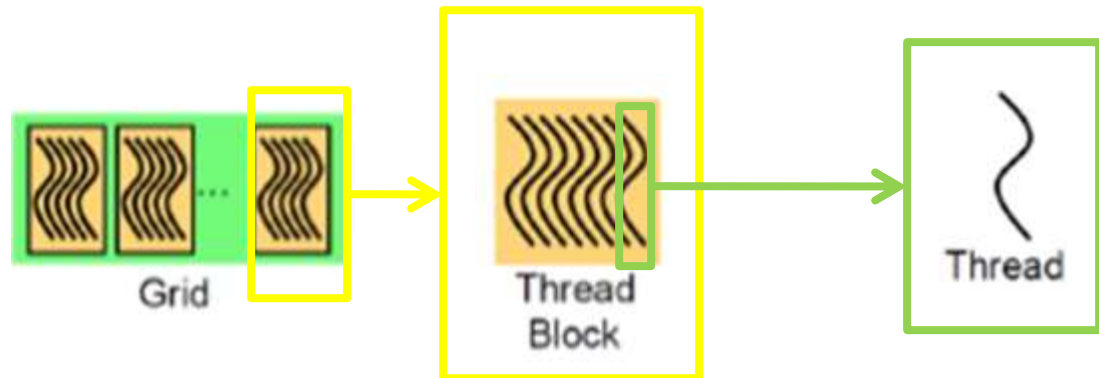
Hardverski pogled na GPU (2)

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



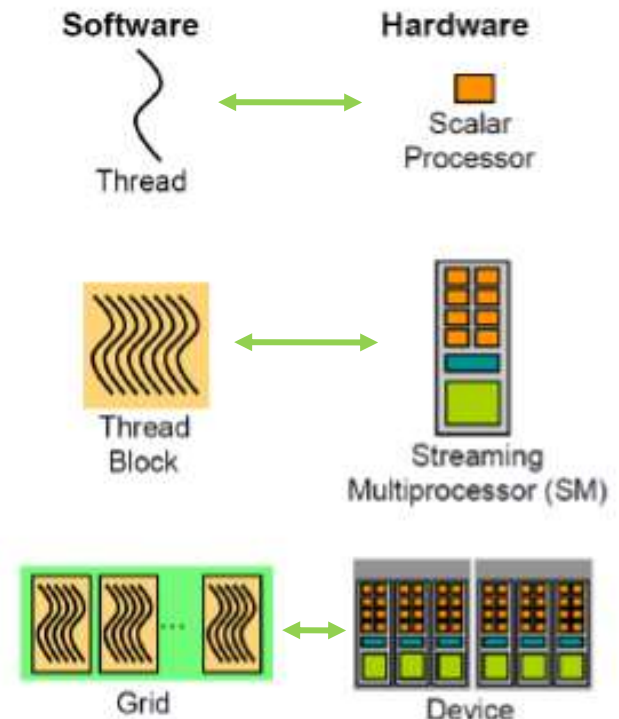
Softverski pogled na GPU (1)

- **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



Preslikavanje HW → SW

- **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, gde je svaki od njih odgovoran za pojedinačnu nit iz bloka



HW → SW

- Blokovi grida (rešetke) se enumerišu i distribuiraju multiprocessorima
- Niti unutar jednog bloka niti se izvršavaju konkurentno na jednom multiprocessoru
- Više blokova niti se mogu izvršavati istovremeno na jednom multiprocessoru
- Kako se blokovi niti završavaju, novi blokovi se pokreću na oslobođenim multiprocessorima
- **SM je dizajniran da konkurentno izvršava na hiljade niti**
- Svaka nit se izvršava na skalarnom procesoru (SP)
- Instrukcije se izvršavaju redom: nema predikcije grananja i nema spekulativnog izvršavanja

Softverski pogled (2)

- Na visokom nivou, master proces koji se izvršava na CPU vrši sledeće:
 1. Inicijalizacija kartice
 2. Alokacija memorije na hostu i device-u
 3. Kopiranje podataka iz memorije hosta u memoriju device-a
 4. Pokretanje većeg broja blokova na device-u
 5. Kopiranje podataka iz memorije device-a u memoriju hosta
 6. Ponavljanje 3-5 po potrebi
 7. Oslobađanje memorije i završavanje programa

Softverski pogled (3)

- Na nižem nivou (GPU):
 1. Svaki blok se izvršava na SM
 2. Ako je broj blokova veći od broja SM, na svakom SM će se izvršavati više od jednog bloka ako ima dovoljno registara i deljive memorije, dok će ostali čekati u redu i biti izvršeni kasnije
 3. Sve niti u okviru jednog bloka mogu pristupati lokalnoj deljivoj memoriji, ali ne mogu videti šta drugi blokovi rade (čak i kada su na istom SM)
 4. Ne postoje garancije za redosled kojim se blokovi izvršavaju

Programski model (1)

- Grafički procesor se posmatra kao koprocesor (uređaj, compute device) u odnosu na centralni procesor (domaćin, host)
 - Izvršava računski intenzivan deo aplikacije
 - Izvršava jako veliki broj niti u paraleli
 - Posедуje svoju sopstvenu DRAM memoriju
- Deo aplikacije koji vrši obradu nad podacima izvršava se u vidu jezgra (kernel) koristeći veliki broj niti
 - GPU niti su lake (lightweight)
 - Imaju veoma mali režijski trošak prilikom stvaranja
 - GPU-u su potrebne hiljade niti za punu efikasnost
 - Višejezgarom procesoru je potrebno samo nekoliko

Programski model (2)

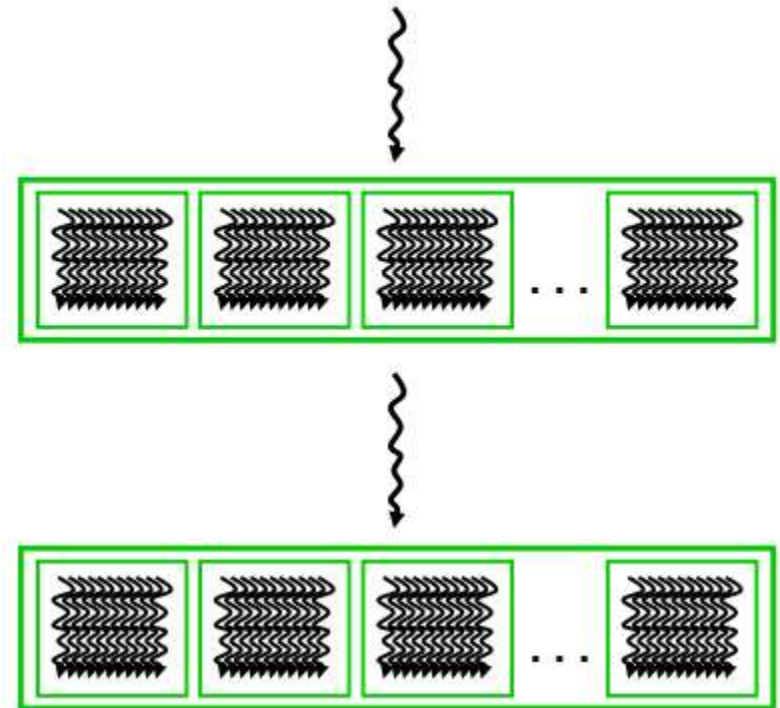
- CUDA program čine integrisani delovi koda za centralni i grafički procesor

Serial Code (host)

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

Serial Code (host)

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



Programski model (3)

- Najjednostavnija forma kernela:

```
kernel_routine<<<gridDim, blockDim>>>(args);
```

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

Programski model (4)

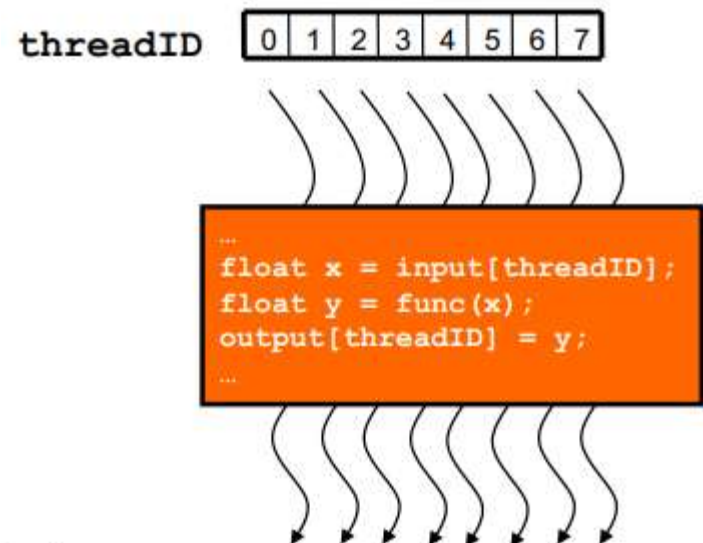
- Na nižem nivou, kada se pokrene jedan blok na SM, njega izvršava veći broj niti koje imaju pristup:
 - Promenljivama koje su prosleđene kao argumenti
 - Pokazivačima na nizove na device-u
 - Globalnim konstantama u memoriji device-a
 - Deljivoj memoriji i privatnim registrima/lokalnim promenljivama
 - 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

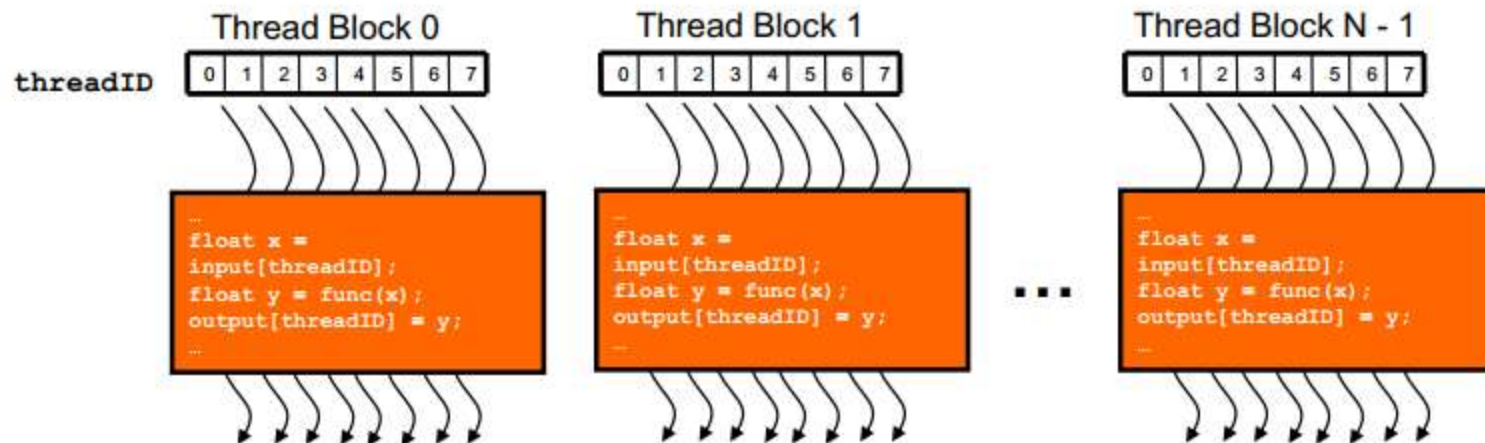
Izvršni model (1)

- CUDA 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



Izvršni model (2)

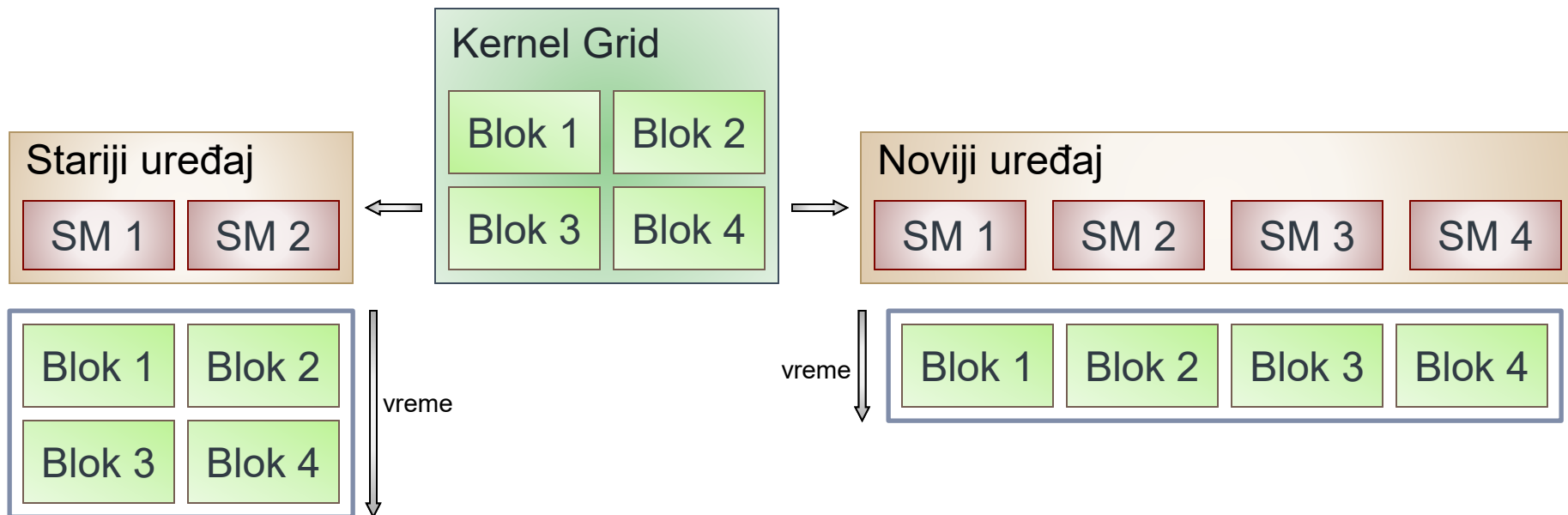
- Niti unutar rešetke su podeljene u nezavisne blokove
 - Svaki blok ima jedinstven identifikator unutar rešetke
 - Niti unutar istog bloka mogu da sarađuju
 - Koristeći sinhronizaciju, atomske operacije i deljenu memoriju
 - Niti iz različitih blokova ne mogu da sarađuju
- Na ovaj način se omogućava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



Skalabilnost

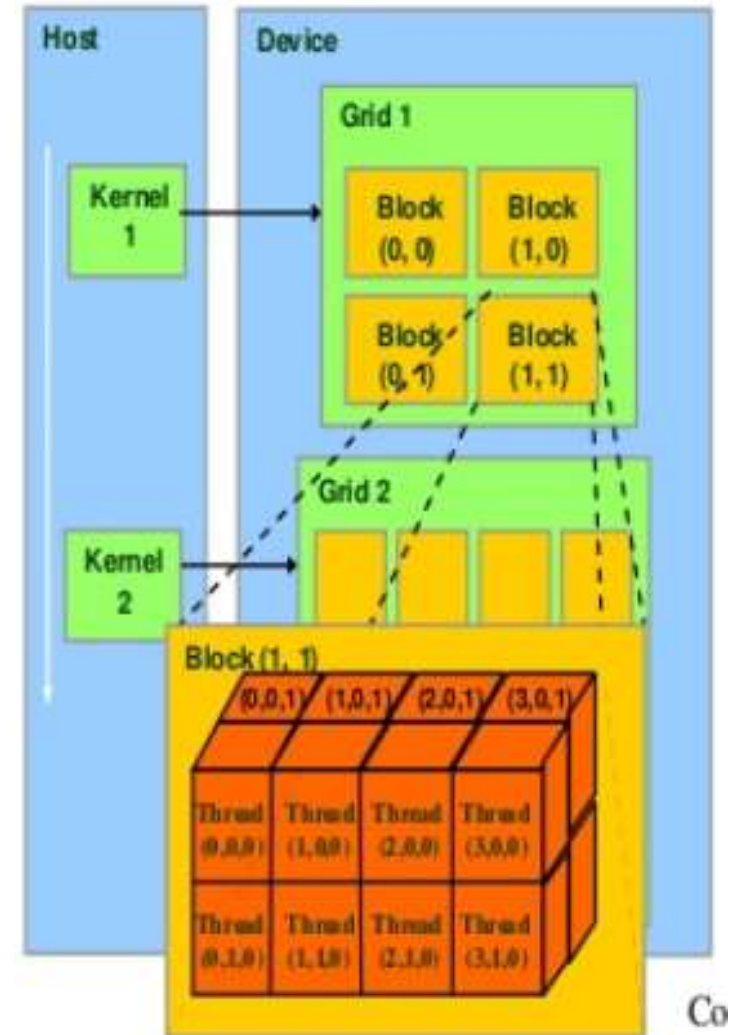
- Blokovi mogu izvršiti u bilo kojem redu u odnosu na druge blokove.

Novi uređaj može izvršiti više blokova paralelno obezbeđujući bolje performanse.



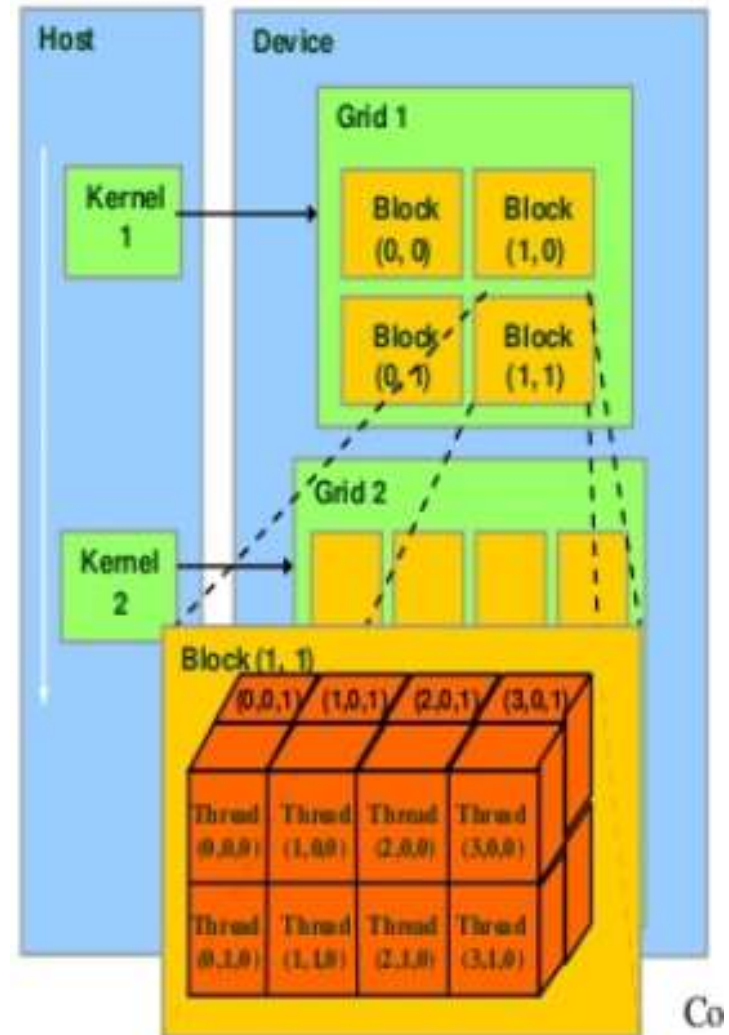
Izvršni model (3)

- Jezgro se konfigurira 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



Izvršni model (4)

- 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.



Razvoj CUDA aplikacija

Razvoj CUDA aplikacija

- Alokacija memorije i prenos podataka između GPU i CPU
 - Alokacija memorije
 - Memorijski transferi
- CUDA ekstenzije
 - Karakteristike funkcija koje se izvršavaju na hostu i uređaju
 - global, host & device funkcije
 - Izgled CUDA aplikacije
 - Osnovni koraci

Alokacija memorije i prenos podataka

- Alokacija memorije na strani 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 kompletiran

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

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

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 jezgru 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(...)

Say Hello to CUDA (1)

```
int main( void )  
{  
    printf( "Hello, World!\n" );  
    return 0;  
}
```

Say Hello to CUDA (2)

```
#include "book.h"

__global__ void kernel( void ) {}

int main( void )
{
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- Prazna funkcija nazvana kernel(), sa kvantifikatorom __global__
- Poziv ove funkcije, dekorisan sa <<<1,1>>>

Say Hello to CUDA (3)

Kernel funkcija sa parametrima

Alokacija

Poziv kernela

Pribavljanje i štampanje rezultata

Dealokacija

```
#include <iostream>
#include "book.h"

__global__ void add(int a, int b, int* c)
{
    *c = a + b;
}

int main( void )
{
    int c;
    int* dev_c;

    HANDLE_ERROR(cudaMalloc((void**)& dev_c, sizeof(int)));

    add<<<1,1>>>(2, 7, dev_c);

    HANDLE_ERROR(cudaMemcpy(&c, dev_c, sizeof(int),
                             cudaMemcpyDeviceToHost));

    printf("2 + 7 = %d\n", c);
    cudaFree( dev_c );
    return 0;
}
```


Say Hello to CUDA (4)

Kernel funkcija sa parametrima

Alokacija

Poziv kernela

Pribavljanje i štampanje rezultata

Dealokacija

```
#include <iostream>
#include "book.h"

__global__ void add(int a, int b, int *c)
{
    *c = a + b;
}

int main( void )
{
    int c;
    int* dev_c;

    HANDLE_ERROR(cudaMalloc((void**)& dev_c, sizeof(int)));

    add<<<1,1>>>(2, 7, dev_c);

    HANDLE_ERROR(cudaMemcpy(&c, dev_c, sizeof(int),
                             cudaMemcpyDeviceToHost));

    printf("2 + 7 = %d\n", c);
    cudaFree( dev_c );
    return 0;
}
```

Say Hello to CUDA (4)

Kernel funkcija sa parametrima

Alokacija

Poziv kernela

Pribavljanje i štampanje rezultata

Dealokacija

```
#include <iostream>
#include "book.h"

__global__ void add(int a, int b, int *c)
{
    *c = a + b;
}

int main( void )
{
    int c;
    int* dev_c;

    HANDLE_ERROR(cudaMalloc((void**)& dev_c, sizeof(int)));

    add<<<1,1>>>(2, 7, dev_c);

    HANDLE_ERROR(cudaMemcpy(&c, dev_c, sizeof(int),
                             cudaMemcpyDeviceToHost));

    printf("2 + 7 = %d\n", c);
    cudaFree( dev_c );
    return 0;
}
```

Sabiranje vektora – Tradicionalni C kod

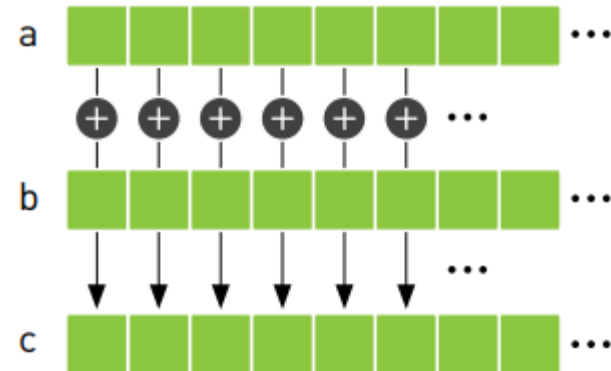
```
#include "book.h"
#define N 10

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] );
    }

    return 0;
}
```



Sabiranje vektora – Tradicionalni C kod

```
#include "book.h"
```

```
#define N 10
```

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

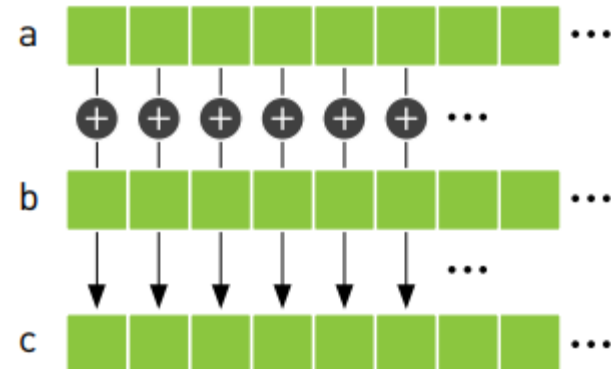
```
{
    for (i=0; i < N; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

// 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
    }
}
```



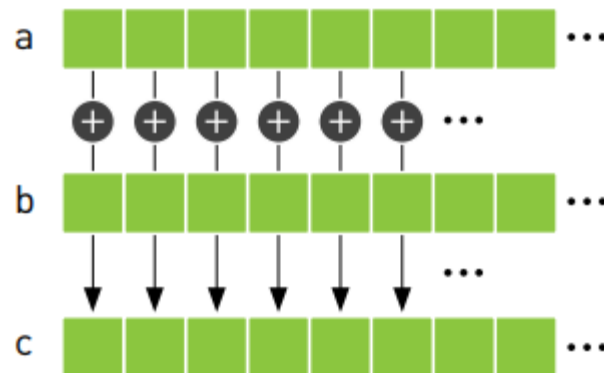
Sabiranje vektora – Tradicionalni C kod

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)

```
#include "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 – 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
 - Idalje 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?
 - $\text{max_broj_niti} = \text{max_broj_blokova} * \text{max_broj_niti_u_bloku}$
- 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?

$\text{blockDim.x} * \text{gridDim.x}$ – broj niti

Sabiranje vektora – veliki vektori

- Da bi se izbeglo pokretanje više blokova nego što je neophodno, potrebno je razumno ograničiti broj blokova

```
add<<<128,128>>>( dev _ a, dev _ b, dev _ c );
```

- Koji je sada limit za broj elemenata u vektoru?

Pribavljanje informacija o uređaju

Kako odabrati broj blokova i niti?

- Da bi razvili što optimalnije aplikacije, neophodno je da znamo karakteristike uređaja na kom se aplikacija izvršava
- U slučajevima kada postoji više uređaja, potreban je način za identifikaciju i selekciju

```
int count;
cudaGetDeviceCount(&count));
```

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
    int concurrentKernels;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int tccDriver;
}
```

Izbor uređaja

```
int main( void )
{
    cudaDeviceProp prop;
    int dev;
    HANDLE_ERROR(cudaGetDevice(&dev));
    printf("ID of current CUDA device: %d\n", dev);
    memset( &prop, 0, sizeof(cudaDeviceProp));
    prop.major = 1;
    prop.minor = 3;
    HANDLE_ERROR( cudaChooseDevice(&dev, &prop));
    printf("ID of CUDA device closest to revision 1.3:
                                                %d\n", dev);
    HANDLE_ERROR(cudaSetDevice(dev));
}
```