

Multiprocesorski sistemi

GPU računarstvo (GPGPU i CUDA)

Marko Mišić

13S114MUPS, 13E114MUPS, 13M114MUPS

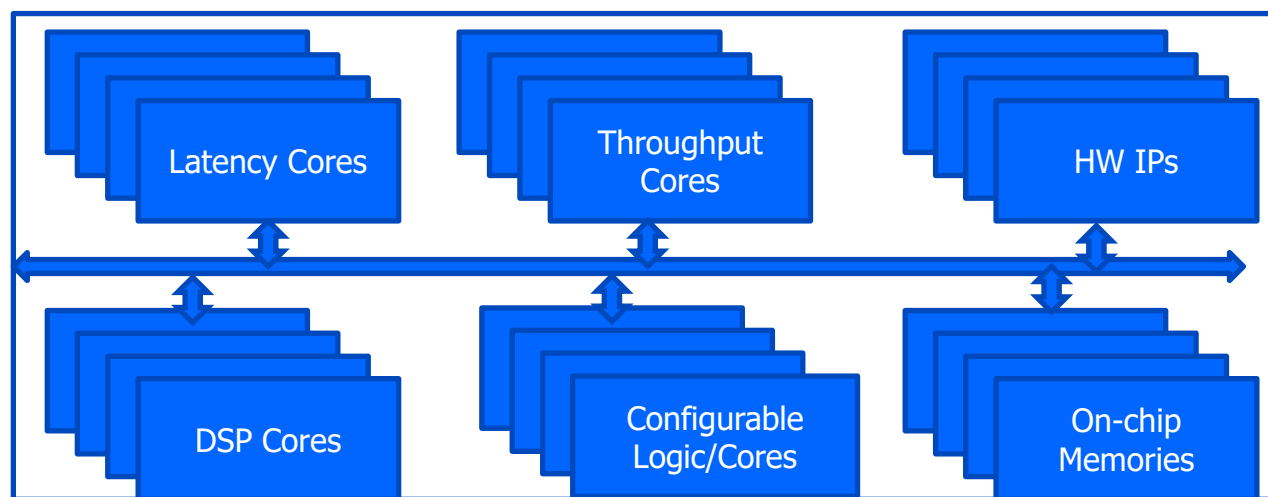
2025/2026.

Uvod u GPU računarstvo (1)

- Grafički procesori (*Graphics Processing Unit*, GPU) su prvobitno bili namenjeni za obradu grafike
 - Specijalizovani za računski intenzivne, grafičke algoritme
- Narastajuća industrija video igara, kao i potreba u komercijalnim aplikacijama je izvršila veliki pritisak na razvoj grafičkih procesora
 - 3D grafika (početkom '90ih)
- Vremenom su ovi procesori evoluirali u paralelne i viskoprogramabilne procesore
 - Orijentisani su ka obradi velike količine podataka

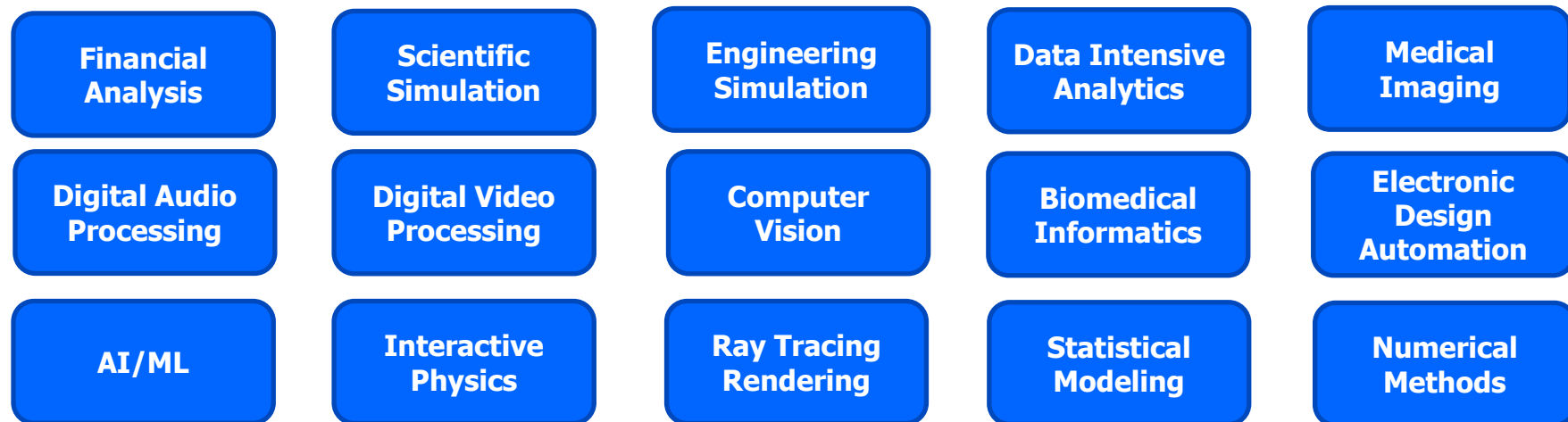
Uvod u GPU računarstvo (2)

- Grafički procesori se koriste za računanja opšte namene u poslednjih 20+ godina
- Taj trend se zove računanje opšte namene korišćenjem grafičkih procesorskih jedinica
 - *General-Purpose computation on GPUs* (GPGPU)
- Heterogeno računarstvo
 - Korišćenje računskih resursa koji najbolje odgovaraju poslu

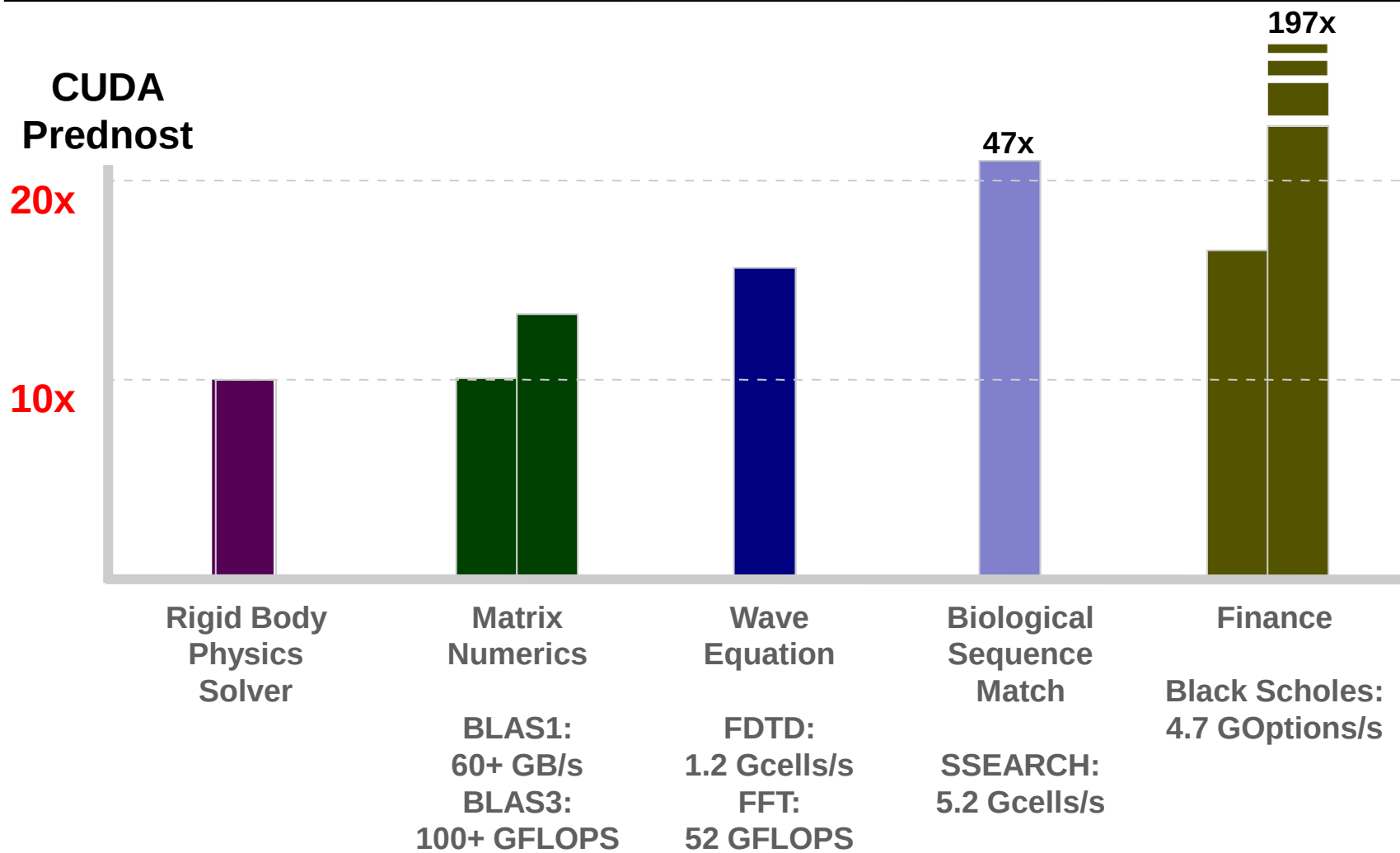


Uvod u GPU računarstvo (3)

- 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
 - Baze podataka
 - Mašinsko učenje i veštačka inteligencija (dominantno!)

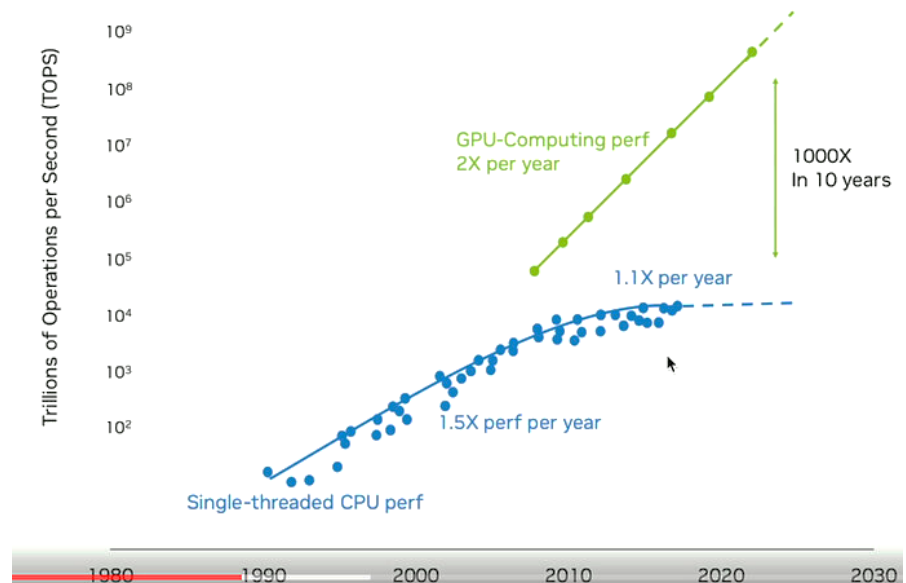
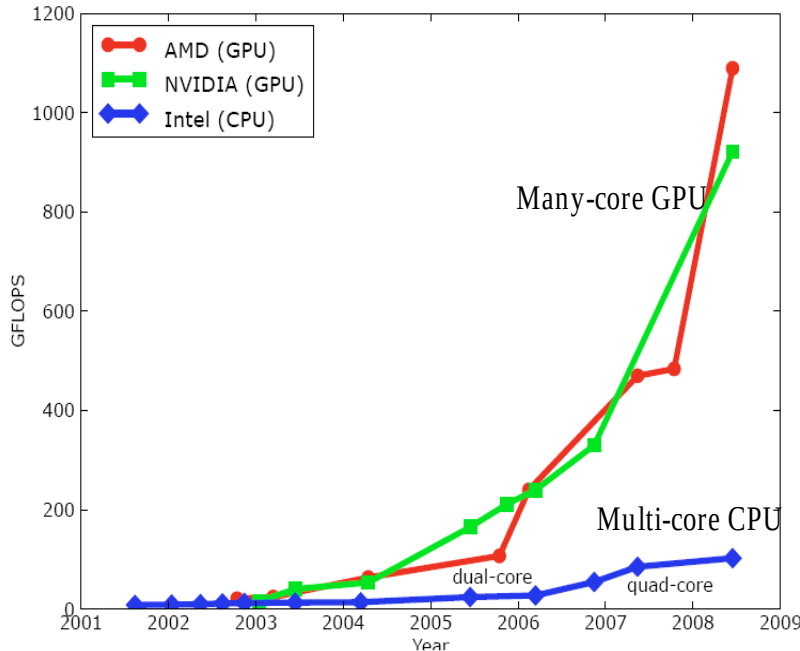


CUDA performanse za različite aplikacije



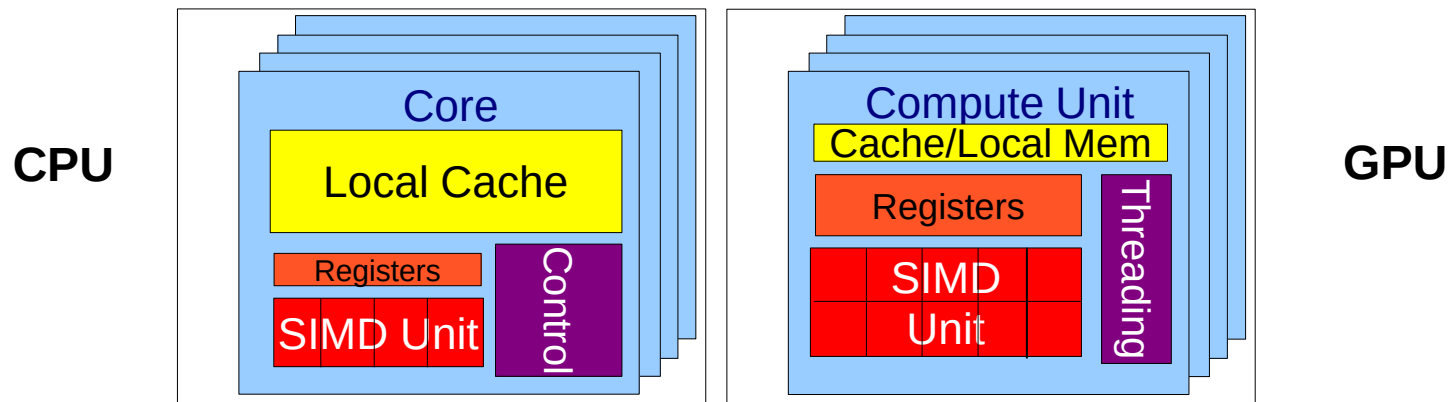
Zašto koristiti grafičke procesore?

- Grafički procesori su postali vrlo fleksibilni i dostupni
 - Računska snaga: 1 TFLOPS vs. 100 GFLOPS
 - Propusni opseg: $\sim 10x$ veći
 - Nalaze se u gotovo svakom računaru



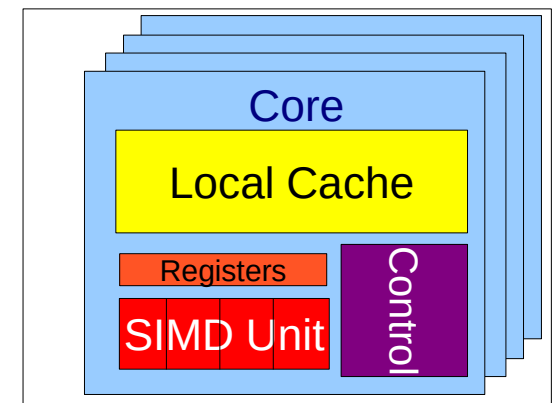
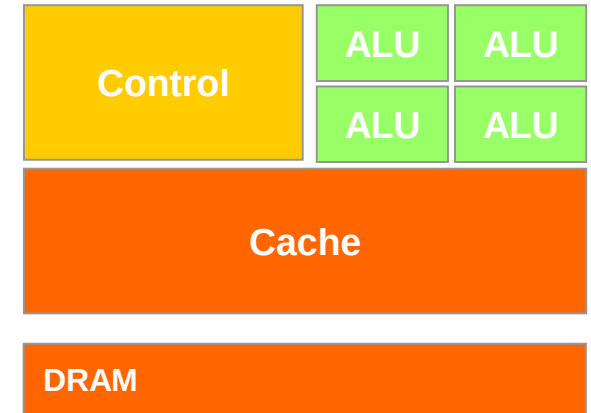
CPU vs. GPU (1)

- Fundamentalna razlika između centralnog i grafičkog procesora je u njihovom dizajnu
 - CPU je orijentisan ka tradicionalnom izvršenju poslova
 - GPU je orijentisan ka obradi podataka
 - Mnogo više tranzistora je namenjeno obradi podataka nego keširanju i kontroli toka



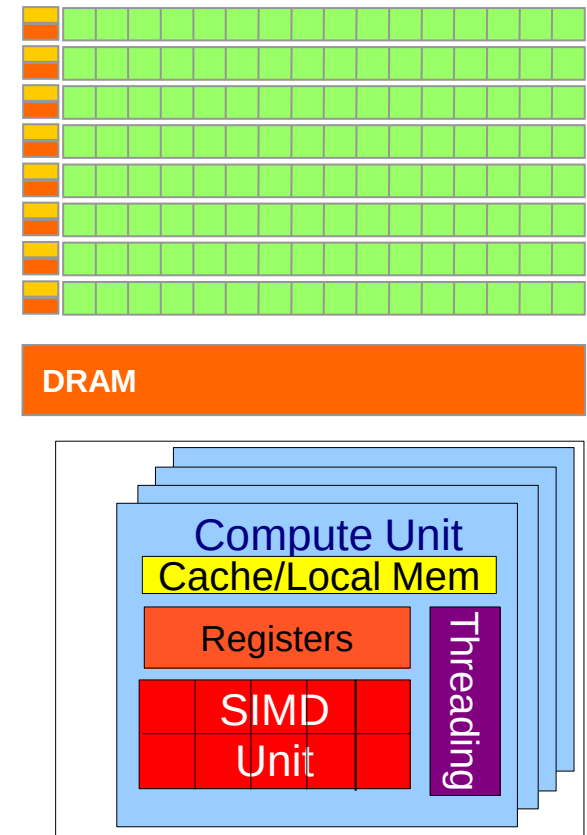
CPU vs. GPU (2)

- Centralni procesor je orijentisan ka smanjenju kašnjenja (*latency*)
 - Hijerarhija keš memorija
 - Veliki keševi koji konvertuju spore pristupe memoriji u brže pristupe keš memorijama
 - Hijerarhija nivoa - L1, L2, L3...
 - Sofisticirana kontrola toka
 - Predviđanje skokova
 - Prosleđivanje podataka
 - Manji broj procesora
 - Moćne ALU jedinice
 - Operacije se izvršavaju veoma brzo
 - Pipeline veličine 20 - 30 faza



CPU vs. GPU (3)

- Grafički procesori su orijentisani ka povećanju propusnog opsega (*throughput*)
 - Veliki broj procesnih jedinica
 - Energetski efikasnije ALU jedinice
 - Veoma male keš memorije
 - Instrukcijski keš
 - Povećavanje propusnog opsega memorije
 - Jednostavna kontrola toka
 - Nema predviđanja skokova
 - Nema prosleđivanja podataka
- Potreban veliki broj niti da bi se sakrila kašnjenja!



Za šta je GPU pogodan? (1)

- Grafički procesor je specijalizovan za računski intenzivna, paralelna izračunavanja
 - Pogodan za računanja *data-parallel* tipa
 - Isti skup instrukcija se izvršava nad velikim brojem podataka istovremeno
 - Smanjena je potreba za sofisticiranom kontrolom toka
 - Veliki broj izračunavanja se odigrava u odnosu na jedan pristup memoriji
 - Sva kašnjenja prilikom pristupa memoriji se mogu sakriti intenzivnim izračunavanjem umesto velikim keševima podataka

Za šta je GPU pogodan? (2)

- Centralni i grafički procesor se najbolje koriste u režimu koprociranja
- Centralni procesor treba koristiti za sekvencijalni deo aplikacije, gde je bitno kašnjenje
 - CPU je najmanje red veličine brži od GPU prilikom izvršavanja sekvencijalnog koda (10x+ puta)
 - Ulaz, izlaz, priprema podataka...
- Grafički procesor treba koristiti za delove koda koji troše najviše vremena
 - Paralelni deo aplikacije
 - Tipično za ubrzanje kritičnih operacija koje obrađuju veliku količinu podataka (10x+ puta)

Istorijat GPU programiranja (1)

- Grafički procesori su postali programabilni početkom 2000-ih sa pojavom programabilnih *shader-a*
- Programiranje je vršeno kroz grafičke API-je
 - OpenGL, DirectX...
 - Mnoga ograničenja
 - Hardverska i softverska (pristup memoriji, *API overhead...*)
- Jezik Brook (Stanford, 2004) je prvi doneo GPGPU programiranje
 - Ograničeni dometi
 - Zavisnost od grafičkih API-ja

Istorijat GPU programiranja (2)

- NVIDIA CUDA (2007)
 - Najzreliji standard, prisutan do danas, CUDA 13.1 (2026)
- AMD/ATI pokušaji
 - Brook+, FireStream, Close-To-Metal, ROCm 7.1 (2025)
- Microsoft DirectCompute (DirectX 10/DirectX 11)
- OpenCompute Language, OpenCL (2009)
 - Otvoreni standard koji je podržala grupa kompanija
 - OpenCL 3.0 (2025)
- Podrška za paralelizaciju direktivama
 - OpenACC 1.0 (2011), 3.4 (2025)
 - OpenMP 4.0 (2013), 5.0 (2018), 6.0 (2024)
- Intel OneAPI i SYCL/Data Parallel C++ (2019)
- Programiranje kroz biblioteke i namenske radne okvire

GPU programiranje danas (1)

- Tri načina za ubrzavanje rada aplikacija na GPU
 - Korišćenjem biblioteka i radnih okvira
 - cuBLAS, cuFFT, Magma, OpenCV, Thrust (CUDA STL), kokkos
 - TensorFlow/TensorRT, cuDNN
 - Korišćenjem prevodilačkih direktiva
 - OpenACC, OpenMP
 - Korišćenjem proširenja programskih jezika
 - CUDA, OpenCL, OneAPI, ROCm/HIP



GPU programiranje danas (2)

- Dostupno u velikom broju programskih jezika
 - Različiti nivoi i načini programiranja
 - *High-level* vs. *low-level* pristupi

Numerical analytics ▶

MATLAB,, Mathematica, LabVIEW

Python ▶

PyCUDA, Numba

Fortran ▶

CUDA Fortran, OpenACC

C ▶

CUDA C, OpenACC

C++ ▶

CUDA C++, Thrust

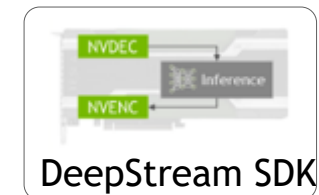
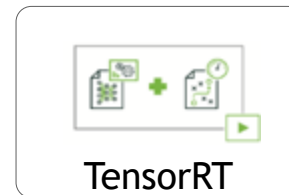
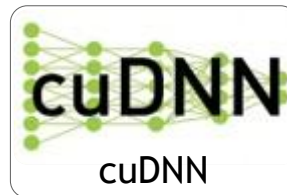
C# ▶

Hybridizer

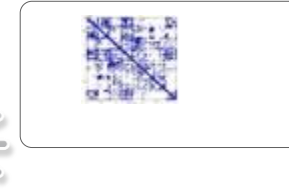
GPU programiranje danas (3)

- Veliki broj biblioteka sa GPU akceleracijom

DEEP LEARNING



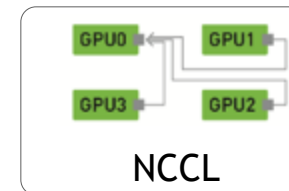
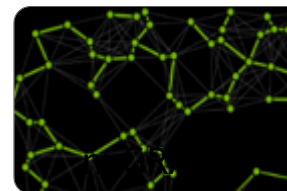
LINEAR ALGEBRA



SIGNAL, IMAGE, VIDEO



PARALLEL ALGORITHMS

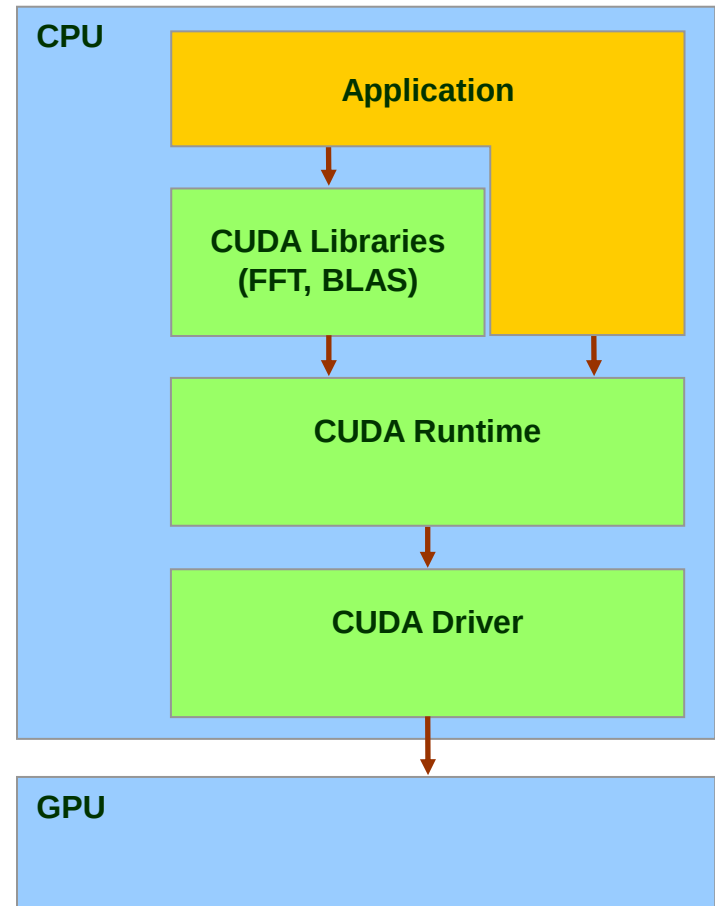


CUDA pregled (1)

- *Compute Unified Device Architecture* (CUDA)
 - Hardverska i softverska arhitektura za upravljanje izračunavanjem opšte namene na grafičkim procesorskim jedinicama
 - Dostupna na NVIDIA grafičkim procesorima
 - API dostupan i na drugim akceleratorima kroz *3rd party* podršku
- Opštenamenski programski model
 - SIMD / SPMD
 - Korisnik pokreće grupe niti na grafičkom procesoru
 - Programer eksplicitno izražava *data-parallel* model kroz izvršavanje pomoću niti (DLP *via* TLP)

CUDA pregled (2)

- Prati je odgovarajuća softverska podrška
 - Drajver i odgovarajući API
 - Ekstenzija jezika C za lakše programiranje
 - Alati
 - Prevodilac, debager, profajler
 - Gotov softver i biblioteke
 - GPU Computing SDK
 - CUFFT, CUBLAS...
 - Nezavisni proizvođači



CUDA pregled (3)

- Aktuelne arhitekture NVIDIA GPU
 - Definiše skup mogućnosti koje podržava hardver
 - Tesla, Fermi, Kepler, Maxwell, Pascal, Volta, Turing, Ampere, Ada Lovelace, Hopper, Blackwell, Rubin
 - *Major* i *minor* revizije arhitekture koje definišu karakteristike hardvera i instrukcijskog seta
 - *Compute capability* (najnovije 12.1)
- Aktuelna verzija CUDA Toolkit-a je 13.1
 - Definiše nivo softverske podrške
 - Prevodilac, alati, biblioteke
 - Definiše mogućnosti na nivou proširenja jezika
 - Upravljanje memorijom, ugrađene funkcije, itd.

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 su potrebne hiljade niti za punu efikasnost
 - Višejezgarnom procesoru je potrebno samo nekoliko

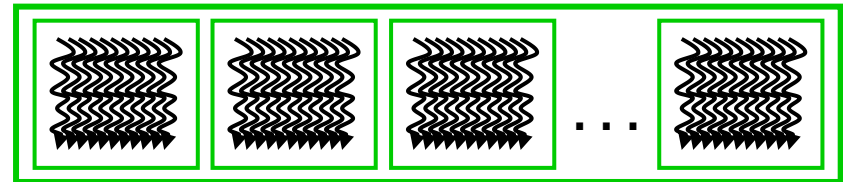
Programski model (2)

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

Serial Code (host)

Parallel Kernel (device)

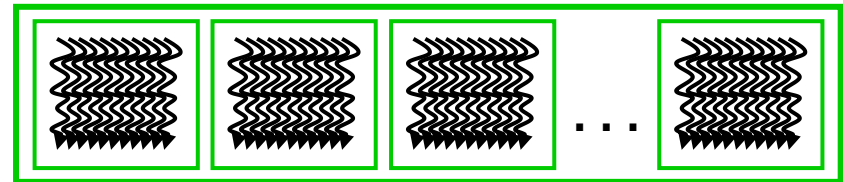
`KernelA<<< nBlk, nTid >>>(args);`



Serial Code (host)

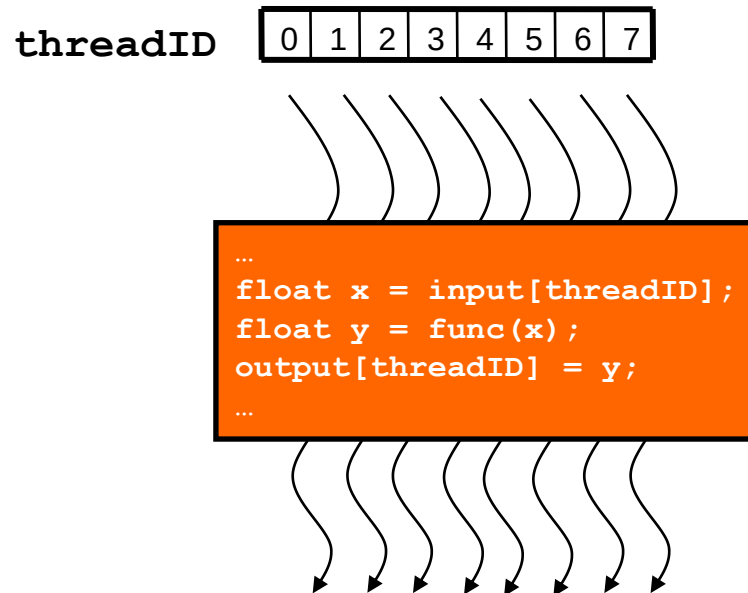
Parallel Kernel (device)

`KernelB<<< nBlk, nTid >>>(args);`



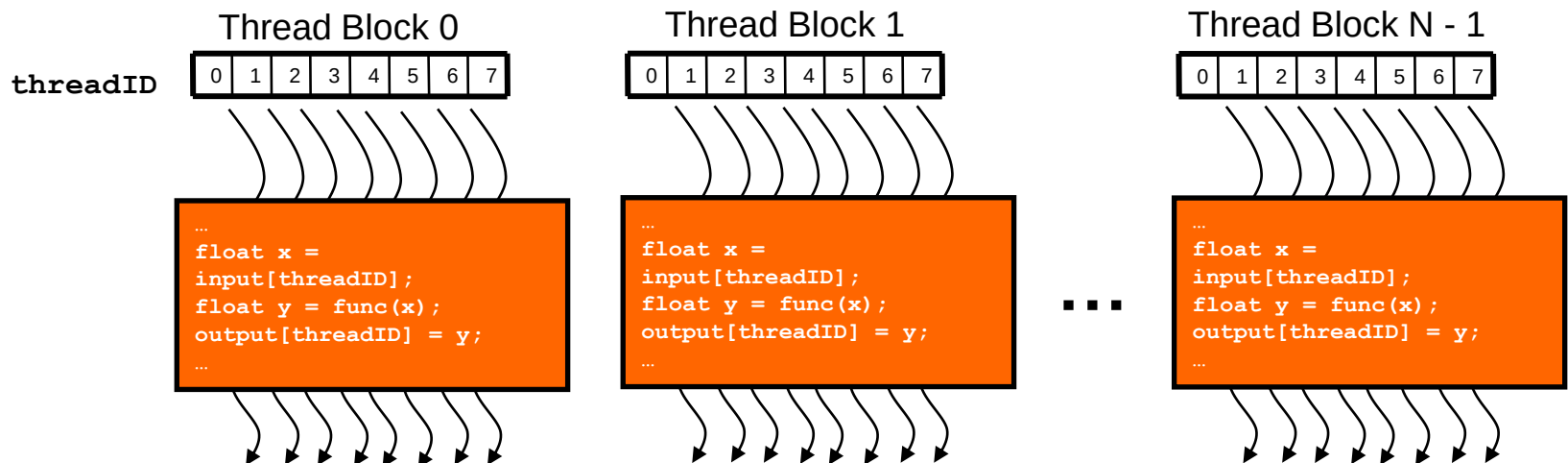
Izvršni model (1)

- 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



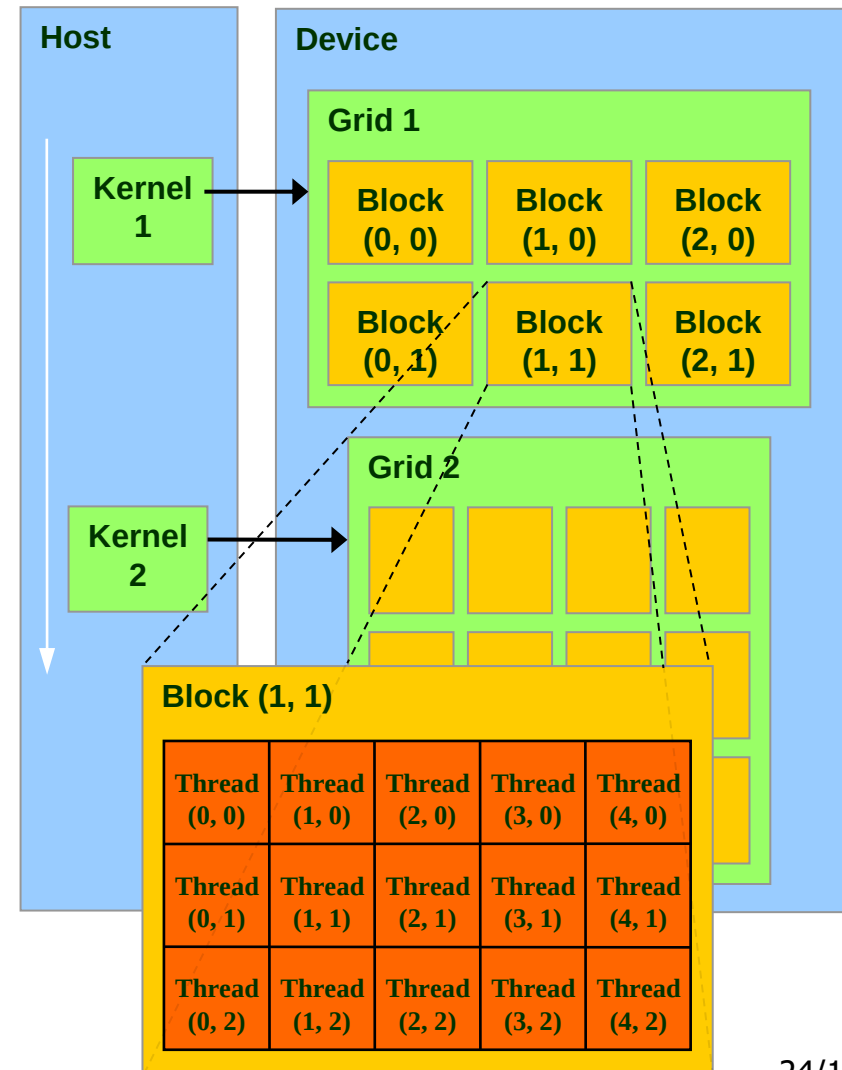
Izvršni model (2)

- 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
 - Omogućava skalabilnost izvršavanja na različitom hardveru



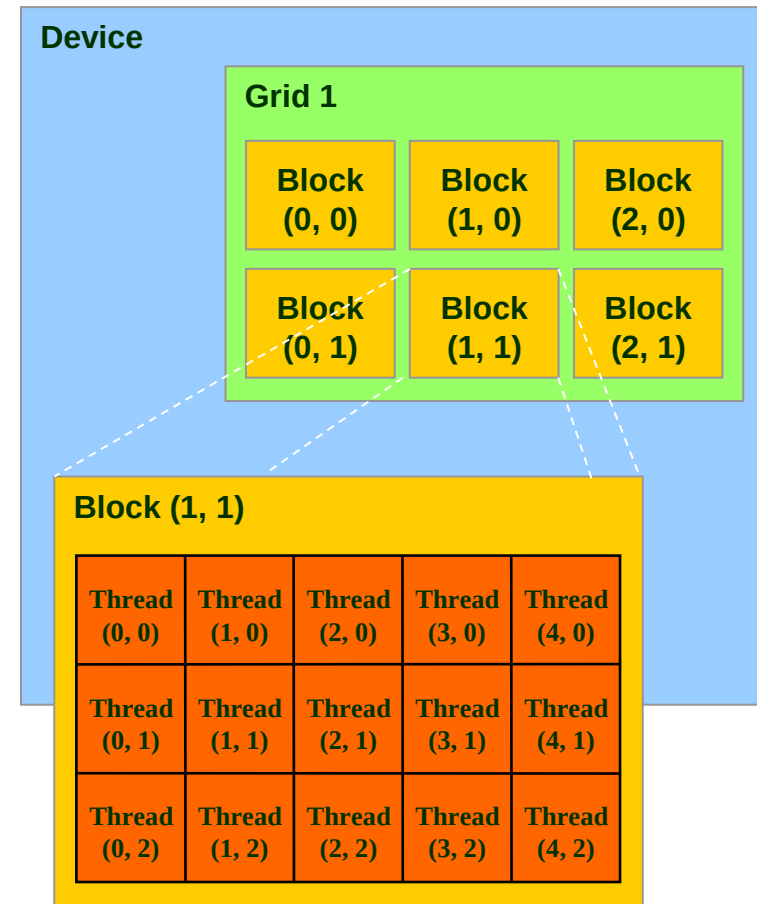
Izvršni model (3)

- 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.



Primer sabiranja dva vektora (1)

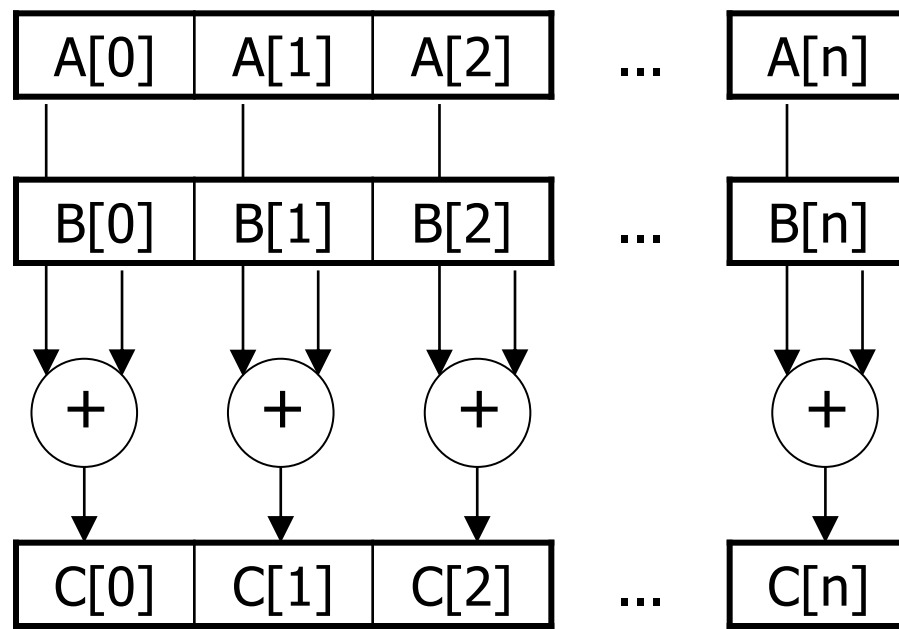
- Tradicionalni sekvencijalni C kod:

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n) {
    for (i = 0, i < n, i++)
        C[i] = A[i] + B[i];
}

int main() {
    // Memory allocation for h_A, h_B, and h_c
    // I/O to read h_A and _B
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

Primer sabiranja dva vektora (2)

- Na grafičkom procesoru, svaka nit će biti zadužena za izračunavanje jednog elementa rezultujućeg vektora



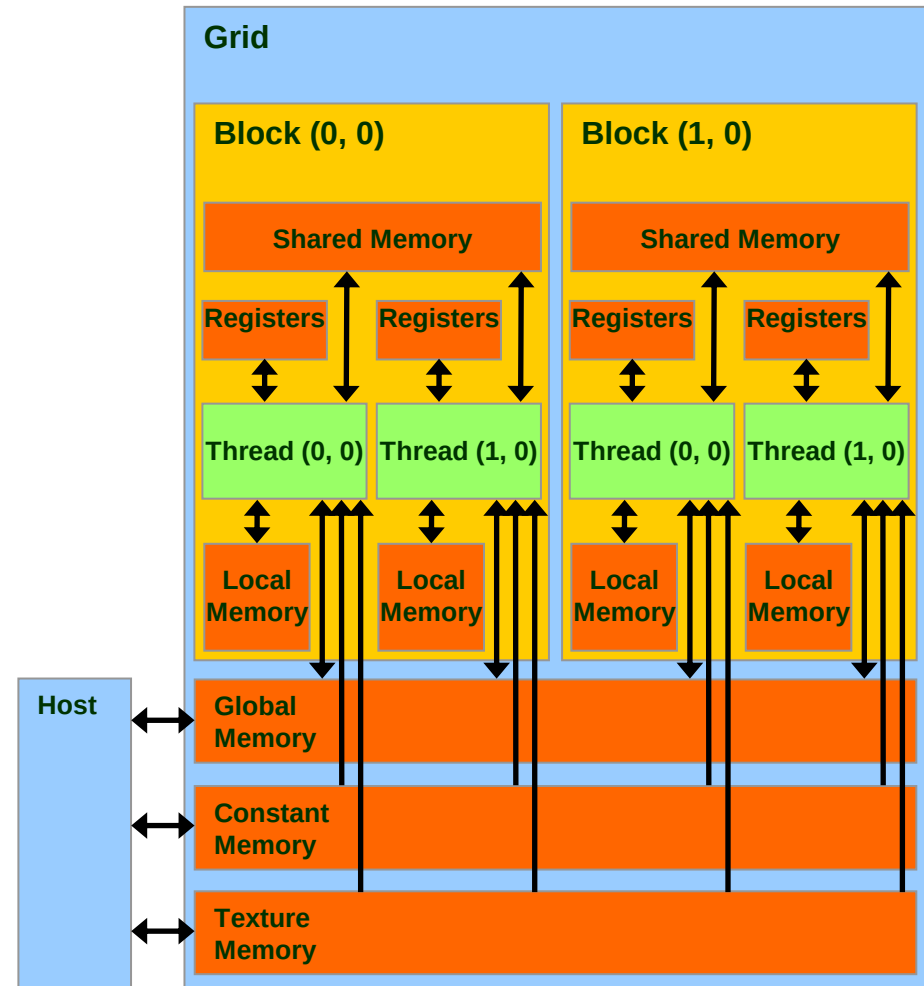
Primer sabiranja dva vektora (3)

- Kod koji će izvršavati centralni procesor mora biti restrukturiran:

```
void vecAdd(float* A, float* B, float* C, int n) {  
    intsize = n* sizeof(float);  
    float* devA, devB, devC;  
  
    ...  
    1. // Allocate device memory for A, B, and C  
    2. // copy A and B to device memory  
    3. // Kernel launch code - to have the device  
        // to perform the actual vector addition  
    4. // Copy vector C from the device memory  
    5. // Free device vectors  
}
```

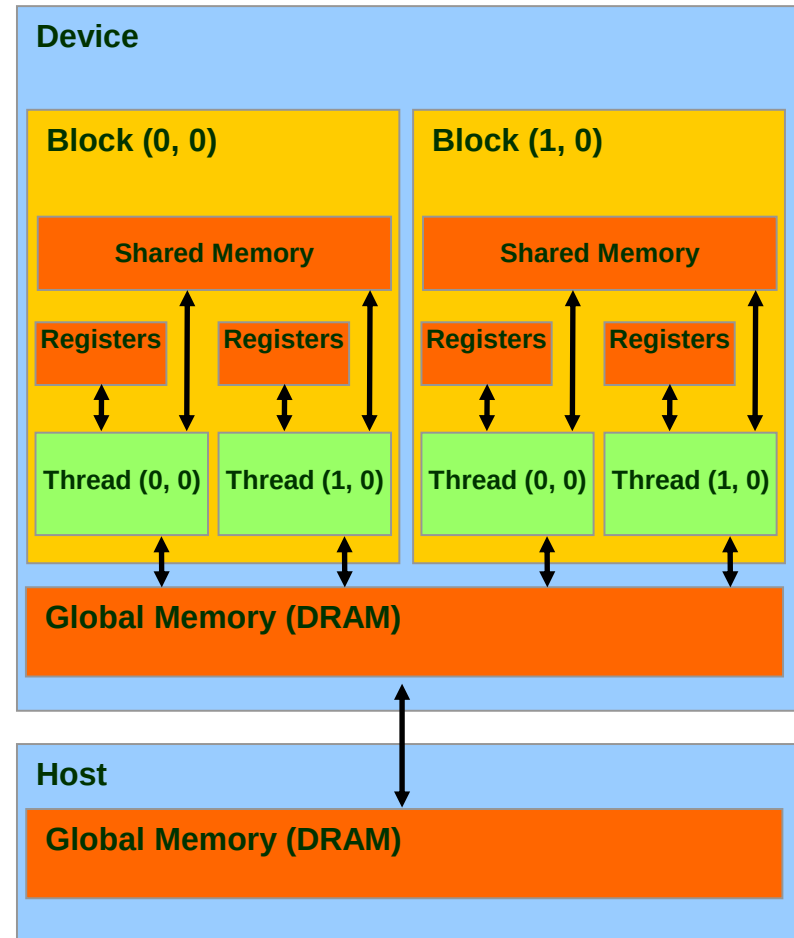
Memorijski model (1)

- Svaka nit može da:
 - Čita/piše po registrima dodeljenim na nivou niti
 - Čita/piše po lokalnoj (privatnoj) memoriji na nivou niti
 - Čita/piše po deljenoj memoriji na nivou bloka
 - Čita/piše po globalnoj memoriji na nivou uređaja
 - Čita konstantnu memoriju na nivou uređaja
 - Čita memoriju za texture na nivou uređaja



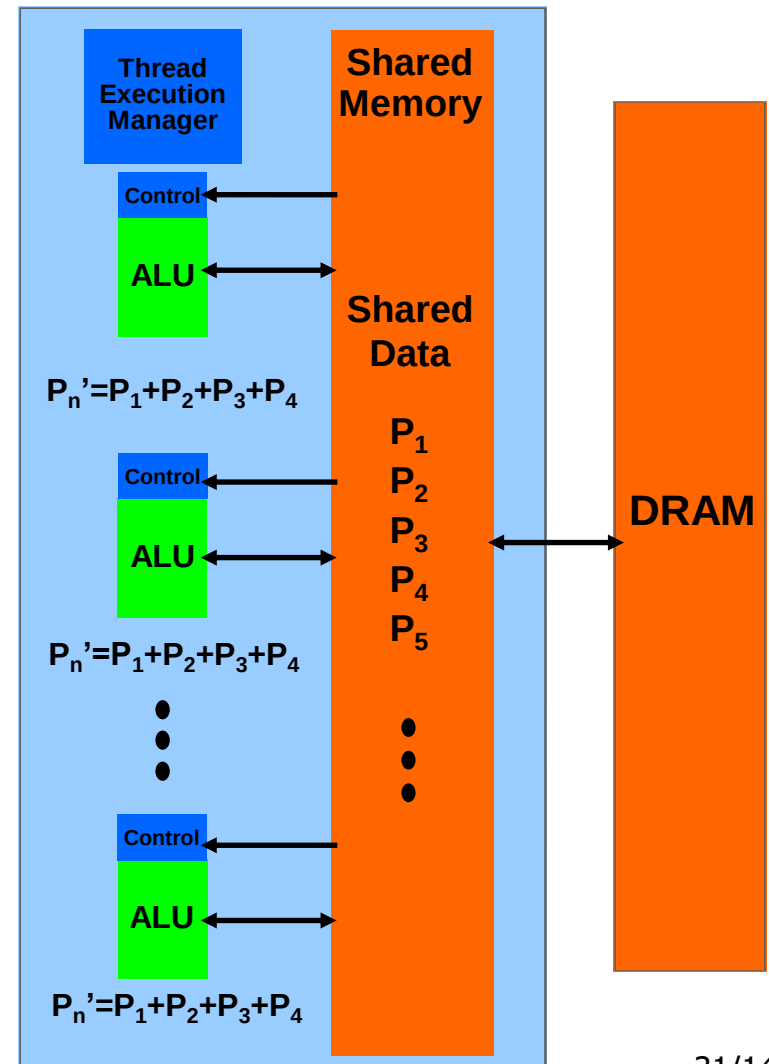
Memorijski model (2)

- Centralni procesor može da čita/piše po globalnoj, konstantnoj i memoriji za texture grafičkog procesora
 - Sve su smeštene u DRAM
- Sadržaj globalne memorije je dostupan svim nitima
 - Pristup ima veliko kašnjenje



Memorijski model (3)

- Deljena memorija se može koristiti na nivou bloka niti
 - Male veličine (16-96 KB)
 - Za red veličine brži pristup od globalne memorije
 - Niti su zadužene da eksplicitno učitaju podatke
- Deljena memorija omogućava da podaci budu bliži ALU jedinicama
 - Smanjuje potrebu za pristup globalnoj memoriji
 - Smeštanje međurezultata sa malim kašnjenjem
 - Povećava intenzitet računanja time što su podaci bliže procesorima
 - Povećava memorijski propusni opseg



Upravljanje memorijom

- Centralni i grafički procesor poseduju odvojene memorijske prostore
- CUDA podržava dva režima upravljanja memorijom
 - Implicitni i eksplicitni
- U eksplicitnom režimu programer vrši alokaciju memorije i odgovarajuće memorijske transfere
 - Alokacija memorije na strani domaćina se vrši statički ili standardnim C pozivima
- U implicitnom režimu programer vrši samo alokaciju memorije
 - Na strani domaćina i uređaja
 - Zahteva podršku CUDA *Unified memory*

Alokacija memorije (1)

- Vrší se putem odgovarajućih poziva API funkcija
 - Postoje različite varijante, u zavisnosti od načina smeštanja podataka u memoriju
- **cudaMalloc()**
 - Alocira objekat u globalnoj memoriji uređaja
 - Zahteva dva parametra
 - Adresu pointera na alocirani objekat
 - Veličinu alociranog objekta u bajtovima
- **cudaFree()**
 - Oslobađa objekat iz memorije uređaja
 - Zahteva pointer na objekat

Alokacija memorije (2)

- **cudaMallocManaged()**
 - Podrška za *Unified memory*
 - Alocira objekat na domaćinu i u globalnoj memoriji uređaja
 - Zahteva dva parametra
 - Adresu pointera na alocirani objekat
 - Veličinu alociranog objekta u bajtovima
- Memorijski transferi se dešavaju u pozadini
 - Izvršno okruženje ih samo vrši
 - Zahteva upotrebu sinhronizacije pre upotrebe rezultujućeg objekta

Alokacija memorije (2)

- Alokacija memorije na strani domaćina se vrši statički ili standardnim C pozivima
- Primer alokacije memorije za sabiranje dva vektora:

```
#define N 256
...
int A[N], B[N], C[N];
int size = N*sizeof(int);
int *devA, *devB, *devC;
...
cudaMalloc( (void**)&devA, size );
cudaMalloc( (void**)&devB, size );
cudaMalloc( (void**)&devC, size );
...
// Memory transfers and kernel launch
...
cudaFree(devA);
cudaFree(devB);
cudaFree(devC);
```

Memorijski transferi (1)

- Za prenos podataka između domaćina i uređaja, kao i unutar samog uređaja postoje odgovarajući pozivi
 - Različite varijante, u zavisnosti od organizacije podataka
 - Sinhroni/asinhroni i blokirajući/neblokirajući transferi
- **cudaMemcpy ()**
 - Obavlja memorijske transfere
 - Zahteva četiri parametra
 - Pokazivač na odredište
 - Pokazivač na izvor
 - Veličinu podataka koji se prenose u bajtovima
 - Tip transfera
 - Tipovi transfera
 - Host to Host (**cudaMemcpyHostToHost**)
 - Host to Device (**cudaMemcpyHostToDevice**)
 - Device to Host (**cudaMemcpyDeviceToHost**)
 - Device to Device (**cudaMemcpyDeviceToDevice**)

Memorijski transferi (2)

- Primer memorijskih transfera prilikom sabiranja dva vektora:

```
#define N 256
...
int A[N], B[N], C[N];
int size = N*sizeof(int);
int *devA, *devB, *devC;
...
// Device memory allocation
...
cudaMemcpy( devA, A, size, cudaMemcpyHostToDevice);
cudaMemcpy( devB, B, size, cudaMemcpyHostToDevice);
...
// Kernel launch
...
cudaMemcpy( C, devC, size, cudaMemcpyDeviceToHost);
...
// Free device memory
```

Deklaracija CUDA jezgra

- Kod koji se izvršava na grafičkom procesoru se piše u vidu odgovarajuće funkcije – jezgra

`__global__`

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

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

Pozivanje CUDA jezgra (1)

- Jezgro mora biti pozvano 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 rešetke 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

Pozivanje CUDA jezgra (2)

- Primer poziva jezgra:
 - Dvodimenzionalna rešetka 64x128
 - Dvodimenzionalni blok 32x8

```
__global__ void KernelFunc(...);  
dim3    DimGrid(64, 128);    // 8192 thread blocks  
dim3    DimBlock(32, 8);    // 256 threads per block  
KernelFunc<<< DimGrid, DimBlock >>>(...);
```

- Tip `dim3` je ugrađeni CUDA tip
- Unutar kernela svaka nit određuje podatke nad kojima će raditi pomoću ugrađenih promenljivih `threadIdx` i `blockIdx`

Primer sabiranje dva vektora (4)

- Kompletan program

```
#define N 1024
void vecAdd(float* A, float* B, float* C, int n);
int main (int argc, char **argv ) {
    int size = N *sizeof( int);
    int A[N], B[N], C[N];
    // Load arrays
    vecAdd(A, B, C, N);
    // Process results
    return 0;
}
```

Primer sabiranje dva vektora (5)

```
void vecAdd(float* A, float* B, float* C, int n){
    int size = n * sizeof(float);
    float *devA, *devB, *devC;
    cudaMalloc((void **) &devA, size);
    cudaMemcpy(devA, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &devB, size);
    cudaMemcpy(devB, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &devC, size);

    // Run ceil(N/256) blocks of 256 threads each
    vecAddKernel<<<ceil(N/256), 256>>>(devA, devB, devC, n);

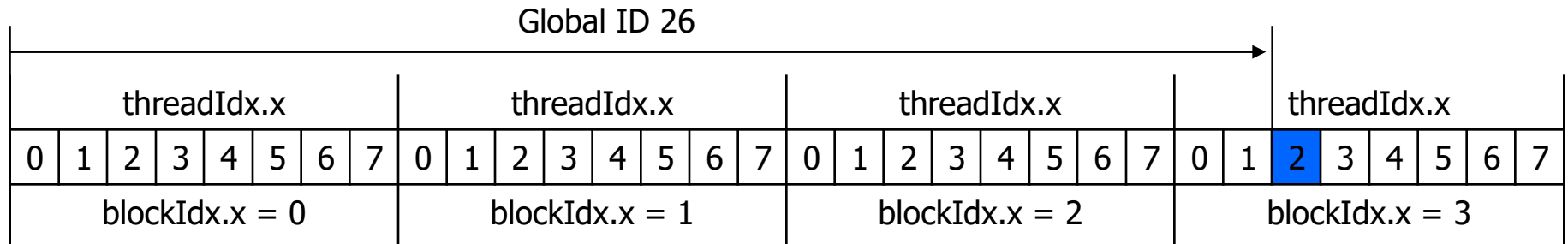
    cudaMemcpy(C, devC, size, cudaMemcpyDeviceToHost);
    cudaFree(devA); cudaFree(devB); cudaFree(devC);
}
```

Primer sabiranje dva vektora (6)

```
__global__ void vecAddKernel  
(float* devA, float* devB, float* devC, int n){  
    int idx =  
        blockIdx.x * blockDim.x + threadIdx.x ;  
    if (idx < n)  
        devC[idx] = devA[idx] + devB[idx];  
}
```

- Svaka nit računa indeks rezultujućeg elementa koji treba da izračuna
 - Na osnovu `blockDim.x` i `blockIdx.x` određuje pomeraj bloka u odnosu na početak niza
 - Na osnovu `threadIdx.x` određuje "globalni ID" konkretnog elementa koji treba da obradi

Primer sabiranje dva vektora (7)



- Primer sabiranja dva niza od 32 elementa:

- Jednodimenzionalni blok i rešetka

`gridDim(4, 1, 1)`

`blockDim(8, 1, 1)`

- Indeks konkretnog elementa koji svaka nit treba da obradi se dobija kao:

`idx = blockDim.x * blockIdx.x + threadIdx.x`

- Za konkretnu nit 2 u bloku 3:

`idx = 3*8 + 2 = 26`

CUDA API

- CUDA aplikativni programabilni interfejs (API) je ekstenzija ANSI standarda jezika C
 - Ekstenzija se sastoji od proširenja jezika C i izvršne biblioteke (*runtime*)
 - Proširenja su načinjena za delove koda namenjene izvršavanju na grafičkom procesoru
 - Za određene funkcionalnosti postoji podrška u hardveru
- Izvršna biblioteka implementira:
 - Podskup C funkcija koje mogu da se izvršavaju i na sistemu domaćinu i na uređaju
 - Skup funkcija za upravljanje i kontrolu uređaja
 - Skup funkcija specifičnih za uređaj (*intrinsic*s)

CUDA funkcije (1)

○ CUDA deklaracije funkcija

- Jezgra moraju imati kvalifikator `__global__`
- Kvalifikator `__host__` označava funkcije koje se izvršavaju samo na strani domaćina
- Kvalifikator `__device__` označava funkcije koje se izvršavaju samo na strani uređaja
- Kvalifikatori `__host__` i `__device__` se mogu koristiti zajedno

	Izvršava:	Poziva:
<code>__device__ float deviceFunc()</code>	uređaj	uređaj
<code>__global__ void kernelFunc()</code>	uređaj	domaćin
<code>__host__ float hostFunc()</code>	domaćin	domaćin

CUDA funkcije (2)

- Ograničenja CUDA funkcija
 - `__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(...)`

CUDA kvalifikatori promenljivih

- Automatske promenljive bez kvalifikatora se smeštaju u registre
 - Osim velikih struktura i statičkih nizova koji se smeštaju u lokalnu memoriju
- Pokazivači mogu da pokazuju samo na objekte iz globalne memorije:
 - Alocirane na strani domaćina i prosledene jezgru
`__global__ void KernelFunc(float* ptr);`
 - Statički deklarisanе objekte na strani uređaja
`float* ptr = &globalVar;`
- Kvalifikator `__device__` je opcion
ako su navedeni kvalifikatori `__shared__` ili `__constant__`

	Memorija	Opseg	Životni vek
<code>__device__ __shared__ int SharedVar;</code>	deljena	blok	blok
<code>__device__ int GlobalVar;</code>	globalna	grid	aplikacija
<code>__device__ __constant__ int ConstantVar;</code>	konstantna	grid	aplikacija

Ugrađeni tipovi

- Izvršna biblioteka obezbeđuje određene ugrađene tipove podataka
 - `[u]char[1..4]`, `[u]short[1..4]`,
`[u]int[1..4]`, `[u]long[1..4]`, `float[1..4]`
- To su strukture koje imaju x, y, z, w polja:

```
uint4 param;  
int y = param.y;
```
- Ugrađeni tip `dim3`
 - Zasnovan na `uint3`
 - Koristi se za zadavanje dimenzija

Ugrađene promenljive

- **dim3 gridDim;**
 - Dimenzije rešetke u broju blokova (1D, 2D ili 3D)
 - Maskimalne dimenzije 2147483647 x 65535 x 65535
- **dim3 blockDim;**
 - Dimenzije bloka u broju niti (1D, 2D ili 3D)
 - Maksimalne dimenzije (blok i rešetka)
 - Tesla arhitektura 512 niti (512 x 512 x 64)
 - Kasnije arhitekture 1024 niti (1024 x 1024 x 64)
- **dim3 blockIdx;**
 - Indeks bloka unutar rešetke
- **dim3 threadIdx;**
 - Indeks niti unutar bloka
- Maksimalne vrednosti pojedinih parametara su hardverski zavisne
 - Mogu biti podložne promenama u budućnosti

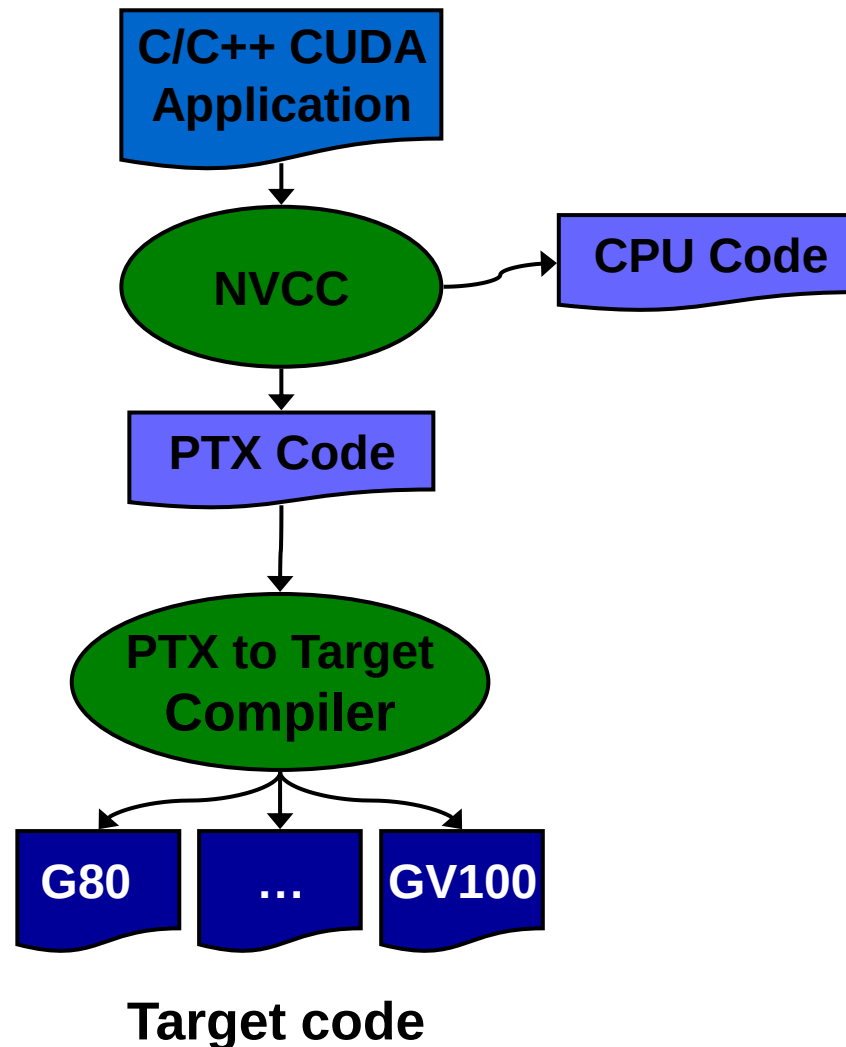
Ugrađene matematičke funkcije

- Izvršna biblioteka obezbeđuje određeni skup matematičkih funkcija
 - `pow`, `sqrt`, `cbrt`, `hypot`
 - `exp`, `exp2`, `expm1`
 - `log`, `log2`, `log10`, `log1p`
 - `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
 - `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
 - `ceil`, `floor`, `trunc`, `round`
- Kada se izvršavaju na strani domaćina, koriste se implementacije iz standardne C biblioteke
 - Podržane samo za skalarne tipove
 - Varijante koje imaju slovo `f` u nastavku imena, poput `sinf` rade sa podacima jednostruke preciznosti (`float`)
 - Postoje i brže, ali manje precizne varijante ovih funkcija, koje se mogu izvršavati samo na strani uređaja
 - `__sin`, `__cos`, `__tan`

Prevođenje CUDA programa (1)

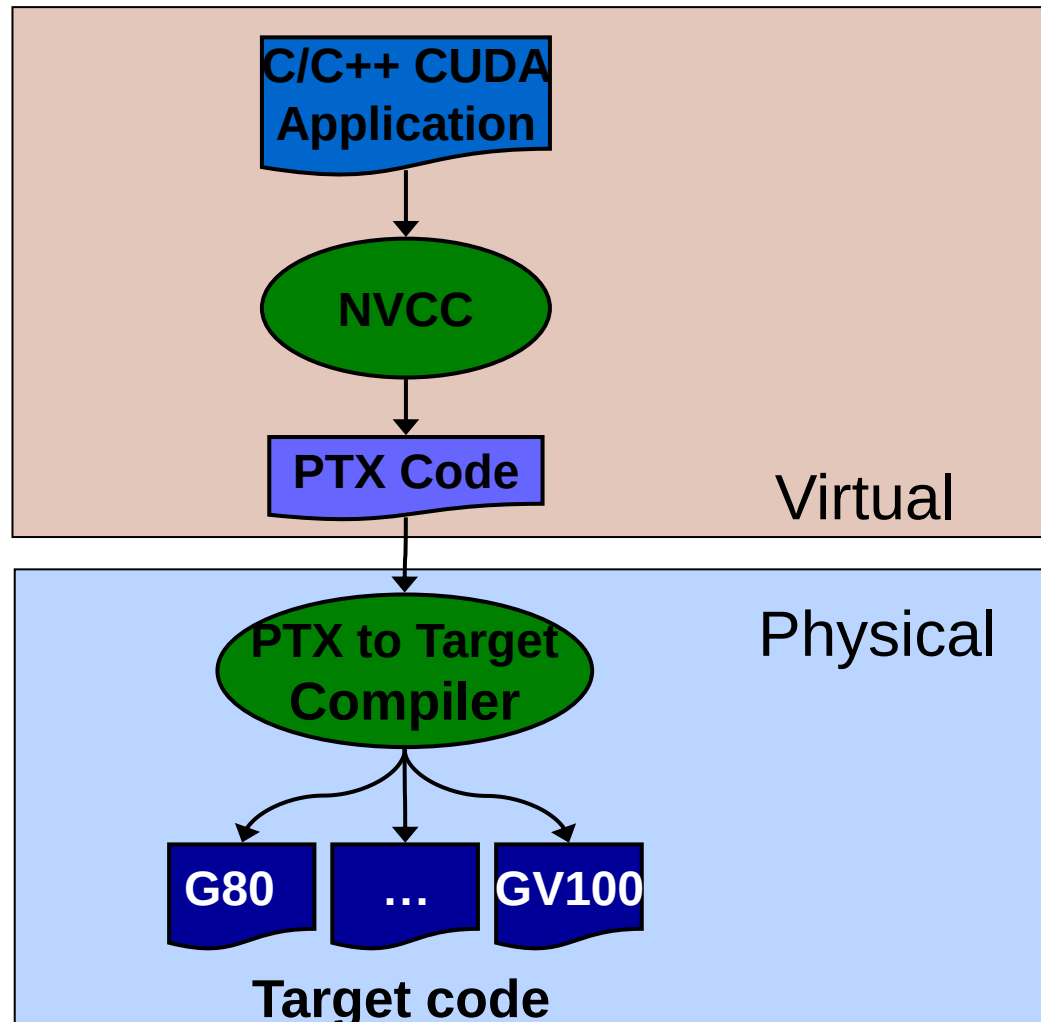
- Bilo koji izvorni kod koji sadrži CUDA ekstenzije mora se prevesti pomoću `nvcc` prevodioca
- NVCC je prevodilac-omotač (*compiler driver*)
 - Radi tako što poziva sve nepohodne alate i prevodioce
 - `cudacc`, `g++`, `cl`, ...
- Izlazi NVCC prevodioca su:
 - C kod koji se izvršava na strani domaćina (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)

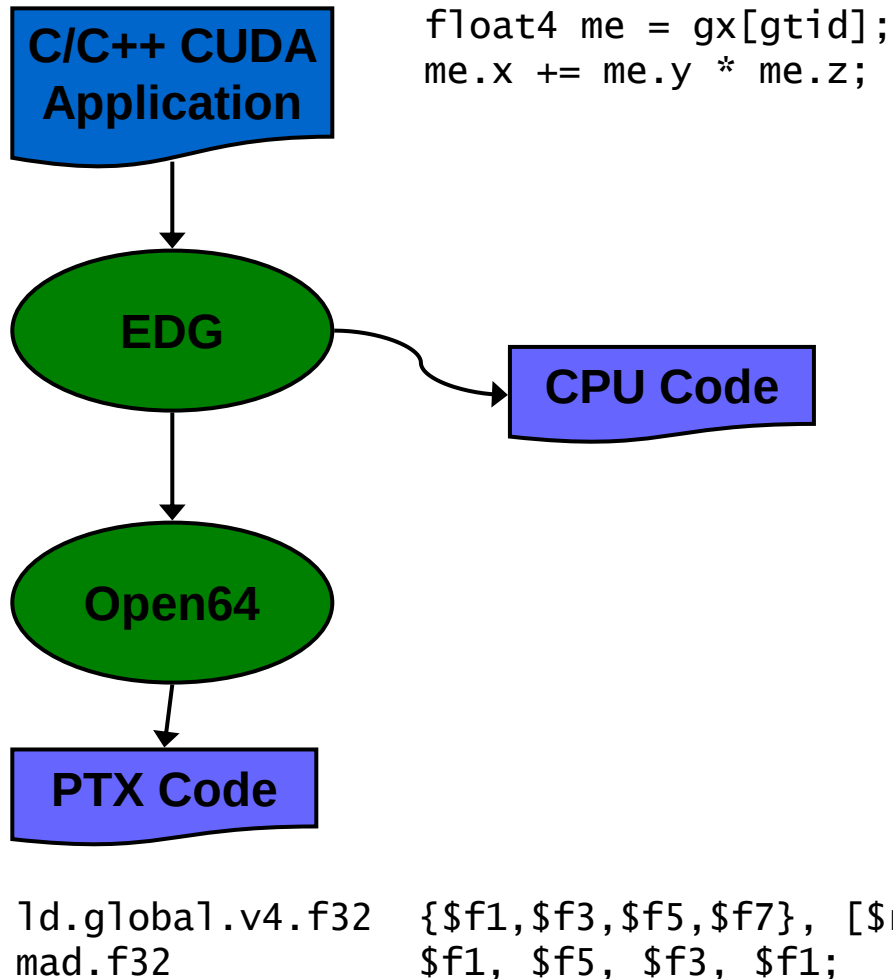


Target code

Prevođenje CUDA programa (3)



NVCC i PTX virtualna mašina



- EDG preprocesor
 - Razdvaja GPU od CPU koda
- Open64
 - Generiše GPU PTX asemblerski kod
- PTX kod
 - *Just-in-time* prevođenje
 - Omogućava izvršavanje na različitim ISA (*Instruction Set Architecture*)

Debagovanje CUDA programa

- Provera korektnosti koda kroz NVIDIA Compute Sanitizer skup alata
 - *memcheck*
 - Detektuje pristupe memoriji van opsega, curenja memorije i sl.
 - *racecheck*
 - Detektuje utrkivanja u deljenoj memoriji
 - *synccheck*
 - Detektuje nekorektnu sinhronizaciju na nivou bloka
 - *initcheck*
 - Detektuje neinicijalizovane pristupe globalnoj memoriji
- Interaktivno debugovanje kroz cuda-gdb
 - Pod Linux OS-om
 - *Nsight Visual Studio Code Edition*
 - Intergracija sa VS Code, omogućava *remote debugging*

Profaliranje CUDA programa

- Paket *Nsight*

- *Nsight Systems*

- Početna tačka za analizu performansi
 - Vremenska linija za CPU i GPU na nivou sistema
 - Detektuje velika uska grla kao što su spori memorijski transferi, periode kada GPU ne radi, probleme u multi-GPU komunikaciji i sl.

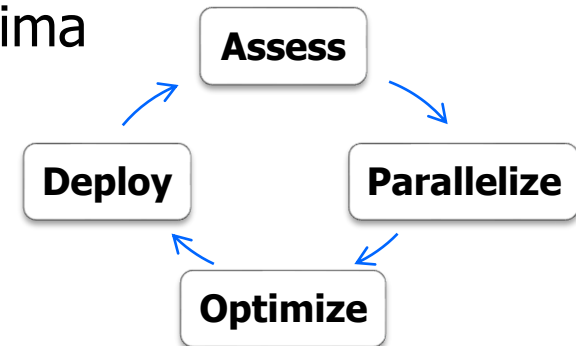
- *Nsight Compute*

- Informacije na nivou pojedinačnih jezgara
 - Analiza da li je jezgro *compute-bound* ili *memory-bound*

- Moguće profajliranje na udaljenom računaru

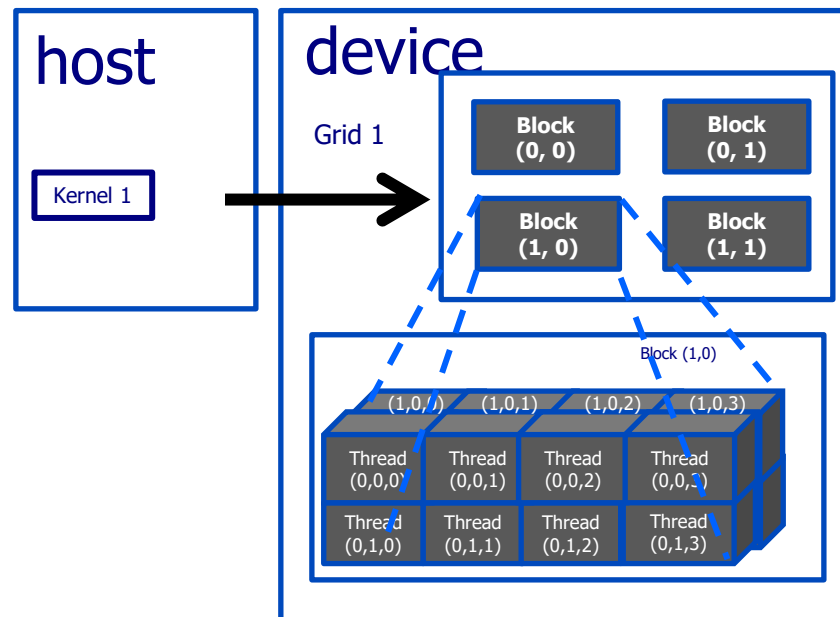
- Prikupljanje informacija *command line* alatima
 - Analiza kroz GUI na lokalnom računaru

- *Assess, Parallelize, Optimize, Deploy* (APOD) ciklus



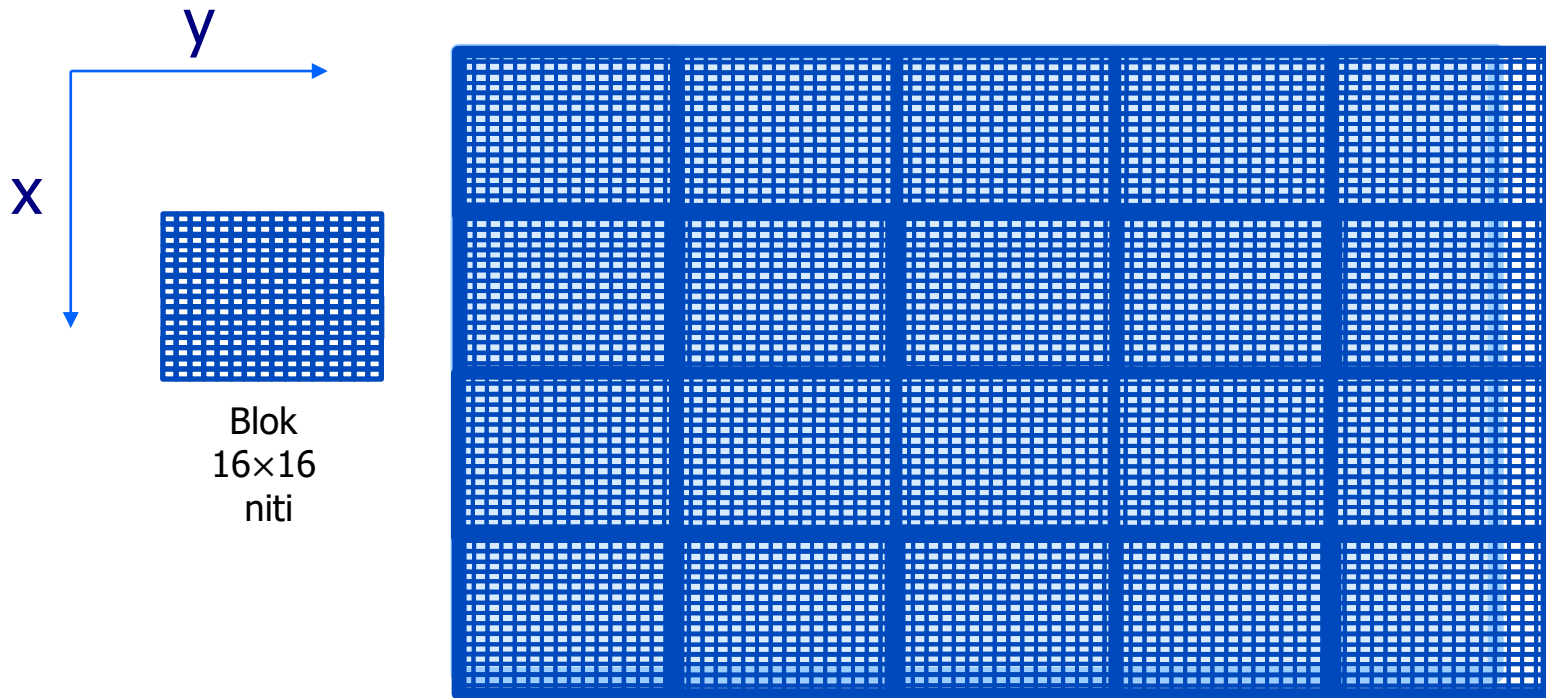
Višedimenzionalne rešetke (1)

- Jezgro se može izvršavati i u okviru višedimenzionalne rešetke
 - Olakšava pristup memoriji pri obradi višedimenzionalnih struktura
 - Matrice, slike i sl.
- Proizvoljna organizacija na nivou bloka i rešetke
 - 1D, 2D i 3D
- Tehnika podele problema na pločice (*tiles*)
 - *Tiling* tehnika



Višedimenzionalne rešetke (2)

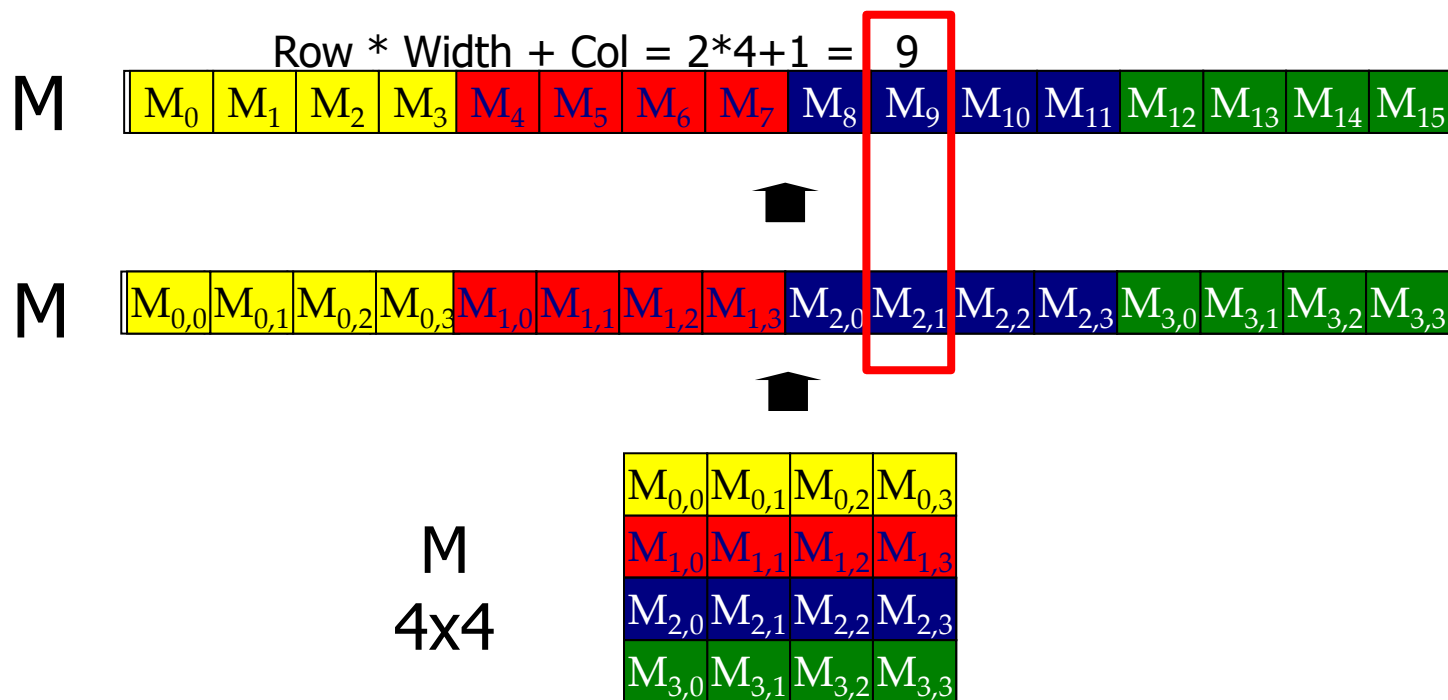
- Primer obrade slike korišćenjem 2D rešetke
 - Jedna nit obrađuje jedan piksel slike (matrice)



Slika dimenzija 62×76

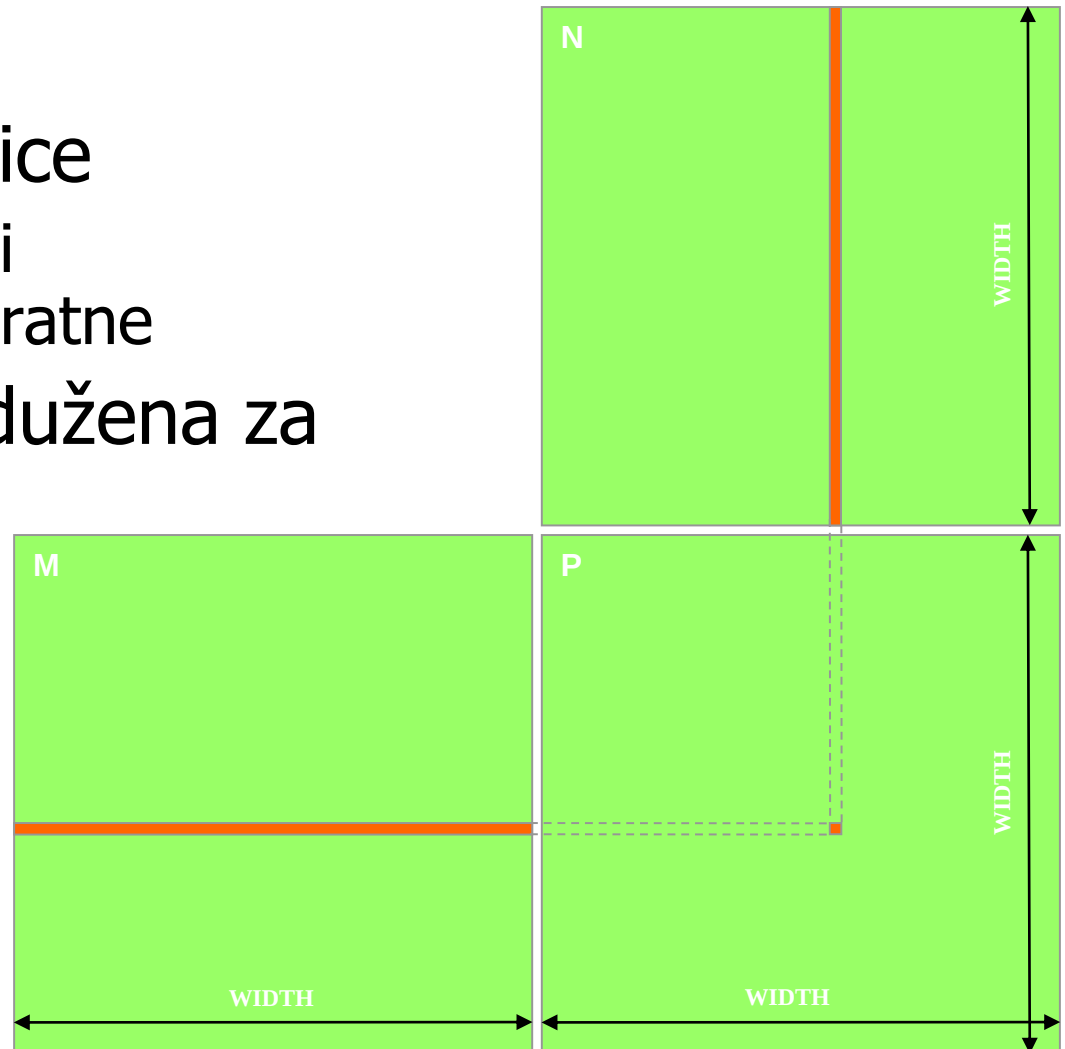
Višedimenzionalne rešetke (3)

- Adresiranje pojedinačnog elemenata matrice
 - Smeštanje elemenata matrice po vrstama
 - Tipično za C/C++
 - Jedna nit obrađuje jedan element matrice



Studija slučaja – množenje matrica (1)

- Potrebno je pomnožiti dve matrice
 - Zbog jednostavnosti pretpostavimo kvadratne
- Jedna nit će biti zadužena za računanje jednog elementa
 - Svaka nit će pristupati WIDTH puta elementima matrica M i N



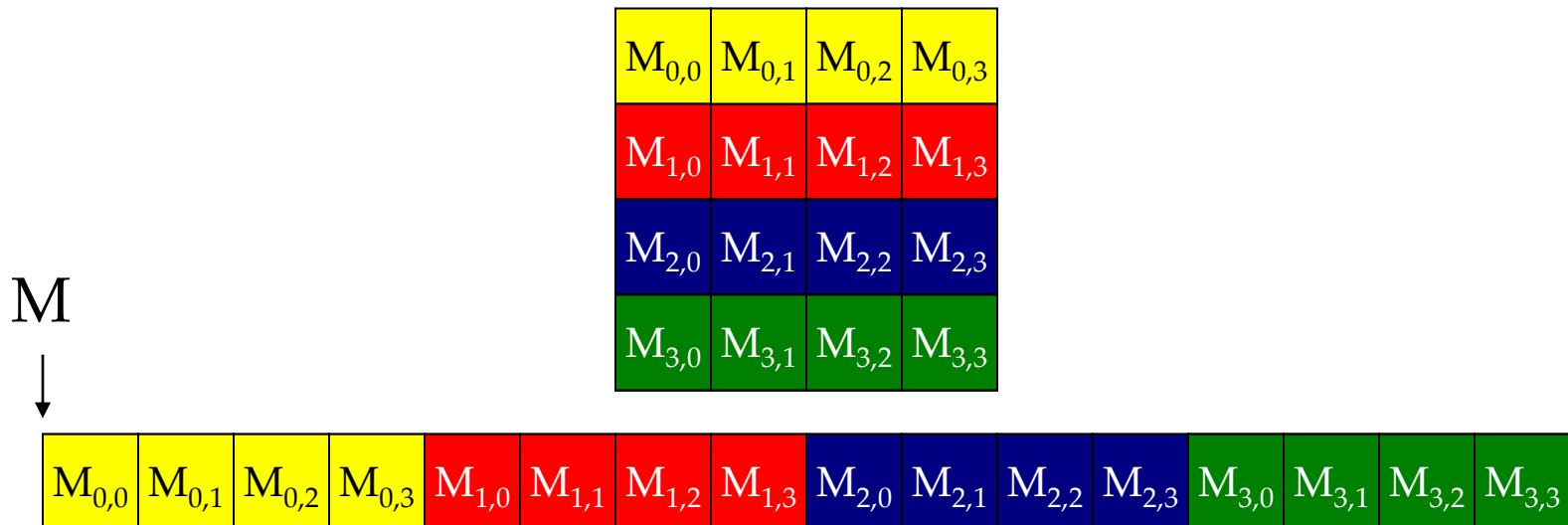
Studija slučaja – množenje matrica (2)

- Tradicionalni sekvencijalni kod:

```
void MatrixMulOnHost
(float* M, float* N, float* P, int Width) {
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

Smeštanje matrica u C-u

- Matrice se u C-u smeštaju po vrstama
 - Matrica će uređaju biti prenetu linearizovana
 - Svaka nit će proračunati adresu elementa kome treba da pristupi



Studija slučaja – množenje matrica (3)

- CUDA program na strani domaćina:

```
void MatrixMulOnDevice (float* M, float* N, float* P, int Width) {  
    int size = Width * Width * sizeof(float);  
    float *Md, *Nd, *Pd;  
1. // Allocate and Load M, N to device memory  
    cudaMalloc(&Md, size);  
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);  
    cudaMalloc(&Nd, size);  
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);  
    // Allocate P on the device  
    cudaMalloc(&Pd, size);  
2. //Kernel invocation code – to be shown later  
3. // Read P from the device  
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);  
    // Free device matrices  
    cudaFree (Md); cudaFree (Nd); cudaFree (Pd);  
}
```


Studija slučaja – množenje matrica (4)

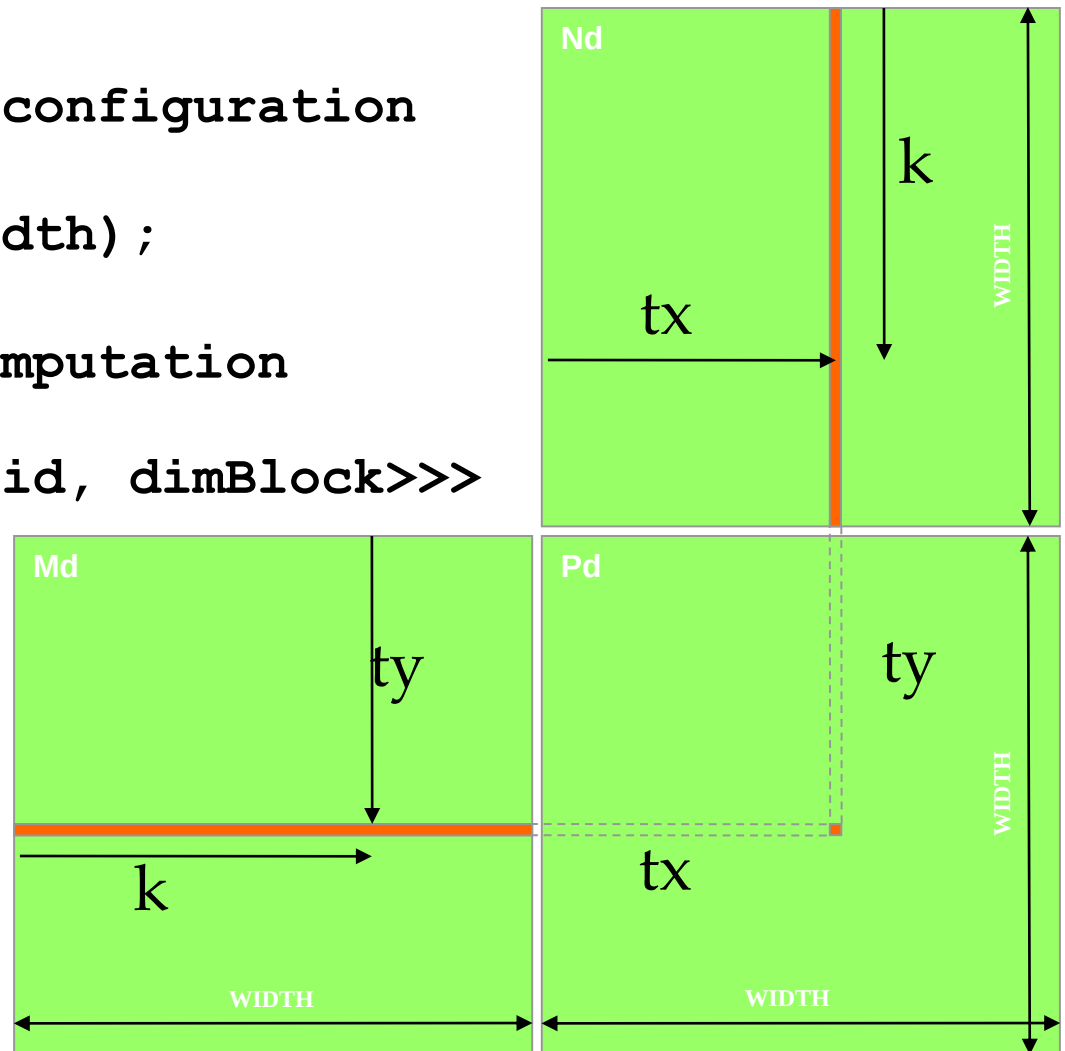
- Jezgro:

```
__global__ void MatrixMulKernel
(float* Md, float* Nd, float* Pd, int Width) {
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
        float Melement = Md[threadIdx.y * Width + k];
        float Nelement = Nd[k * Width + threadIdx.x];
        Pvalue += Melement * Nelement;
    }
    Pd[threadIdx.y * Width + threadIdx.x] = Pvalue;
}
```

Studija slučaja – množenje matrica (5)

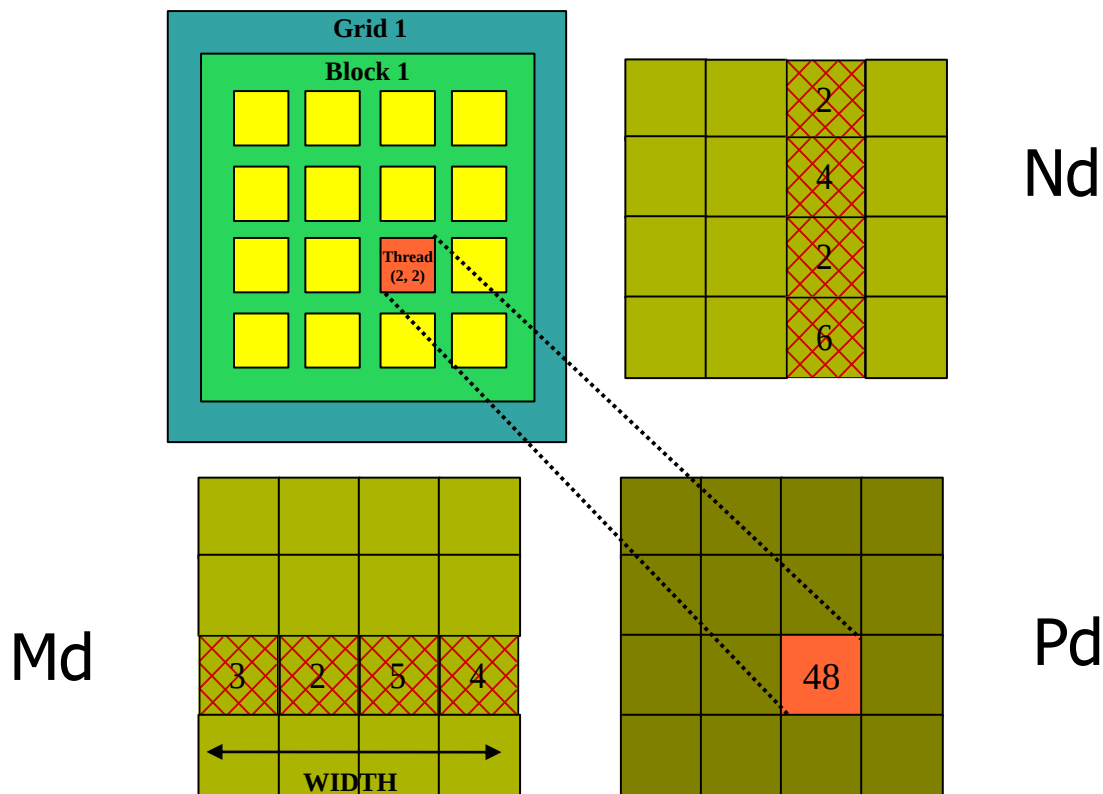
- Jezgro pokreće sledeći kod:

```
// Setup the execution configuration  
dim3 dimGrid(1, 1);  
dim3 dimBlock(Width, Width);  
  
// Launch the device computation  
threads!  
MatrixMulKernel<<<dimGrid, dimBlock>>>  
(Md, Nd, Pd, Width);
```



Nedostaci predloženog rešenja (1)

- Koristi se samo jedan blok niti
 - Matrice mogu biti samo ograničene veličine

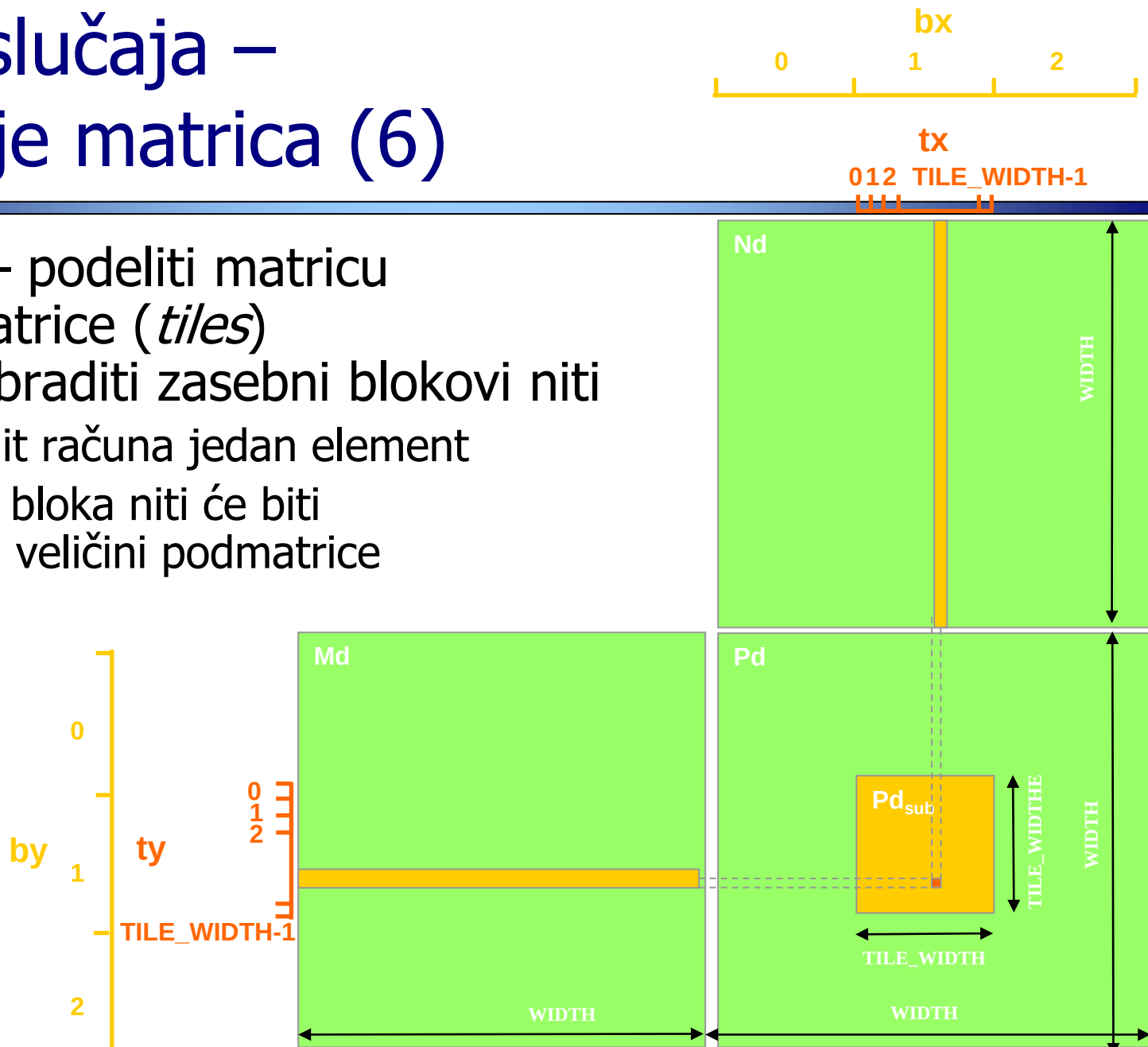


Nedostaci predloženog rešenja (2)

- Jedan blok niti računa matricu P_d
- Svaka nit računa jedan element P_d i pritom:
 - Učitava vrstu matrice M_d
 - Učitava kolonu matrice N_d
 - Izvršava jedno množenje i sabiranje za svaki par elementa iz matrica M_d i N_d
 - Odnos između računanja i pristupa (sporoj) globalnoj memoriji je mali (oko 1:1)
 - 2 operacije
 - 2 pristupa globalnoj memoriji
- Veličina matrice ograničena brojem niti dozvoljenom unutar jednog bloka niti
 - Ogranično arhitekturom (1024 niti)

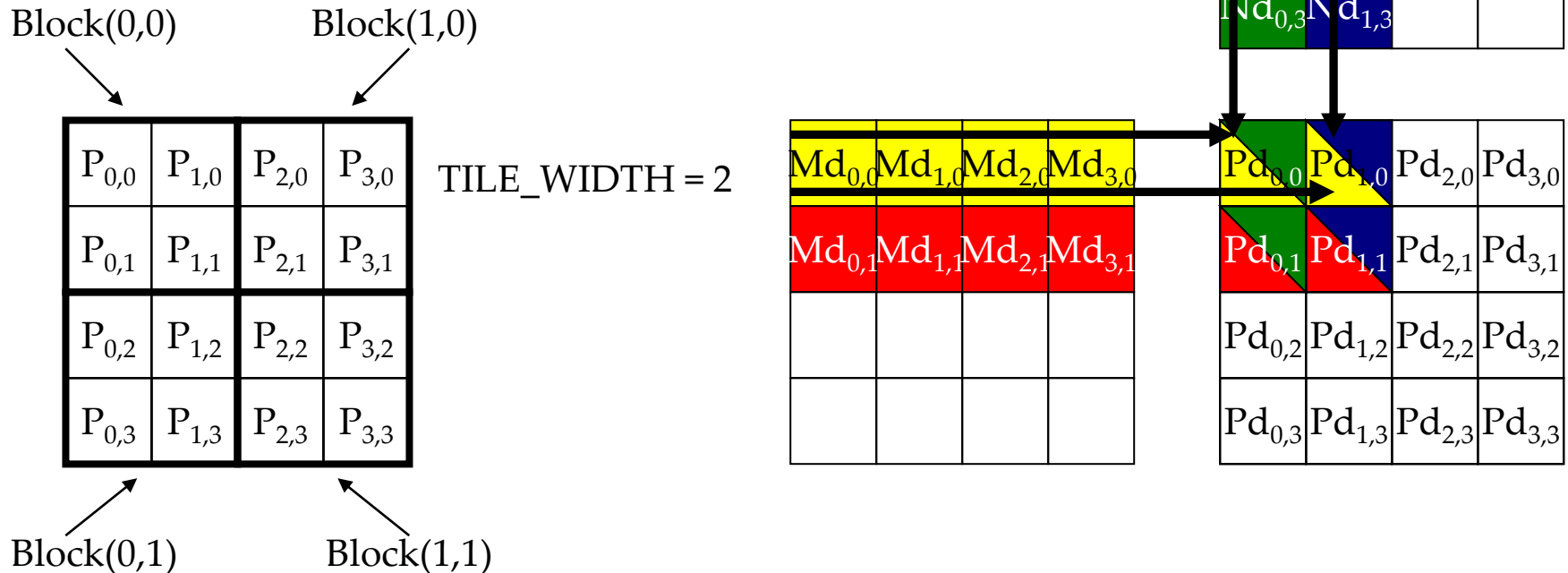
Studija slučaja – množenje matrica (6)

- Rešenje – podeliti matricu na podmatrice (*tiles*) koje će obraditi zasebni blokovi niti
 - Svaka nit računa jedan element
 - Veličina bloka niti će biti jednaka veličini podmatrice



Studija slučaja – množenje matrica (7)

- Primer množenja matrice 4x4, korišćenjem blokova niti dimenzija 2x2 niti



Studija slučaja – množenje matrica (8)

○ Jezgro:

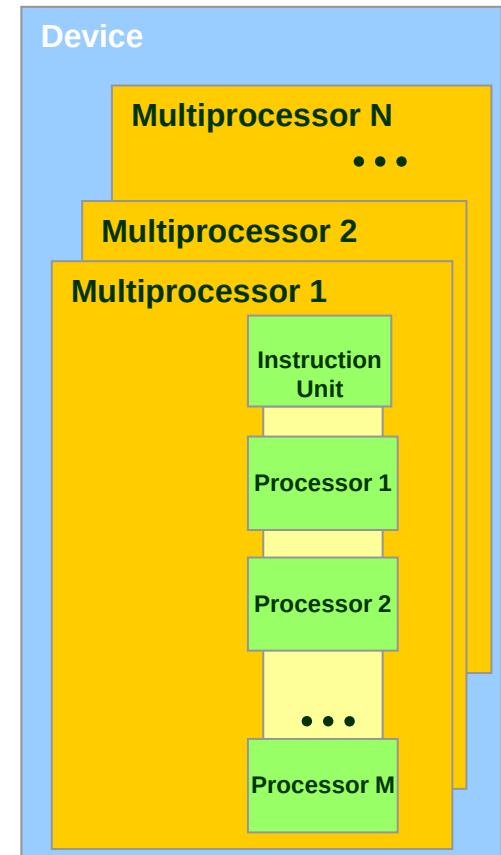
```
__global__ void MatrixMulKernel
(float* Md, float* Nd, float* Pd, int Width) {

    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // Each thread computes one element of the block
        submatrix
        for (int k = 0; k < Width; ++k)
            Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];
        Pd[Row * Width + Col] = Pvalue;
    }
}
```

Hardverska implementacija (1)

- Grafički procesor se može posmatrati kao skup multiprocatora
- Svaki multiprocesor je skup 32-bitnih skalarnih procesora SIMD arhitekture
- U svakom taktu, svaki procesor unutar multiprocatora izvršava istu instrukciju
 - Uključujući skokove (grananja)
- Ovakav način izvršavanja je skalabilan
 - Dodavanjem novih multiprocatora popravljaju se performanse



Hardverska implementacija (2)

- Niti se izvršavaju na skalarnim procesorima
- Blokovi niti se izvršavaju na pojedinačnim multiprocesorima
 - Blokovi ne mogu da migriraju
- Nekoliko blokova niti može da se izvršava na jednom multiprocesoru
 - U zavisnosti od potreba za resursima
 - Registrima
 - Deljenom memorijom

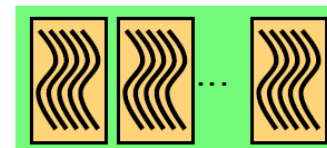
Software



Thread



Thread Block

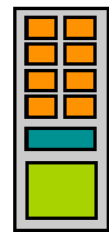


Grid

Hardware



Scalar Processor



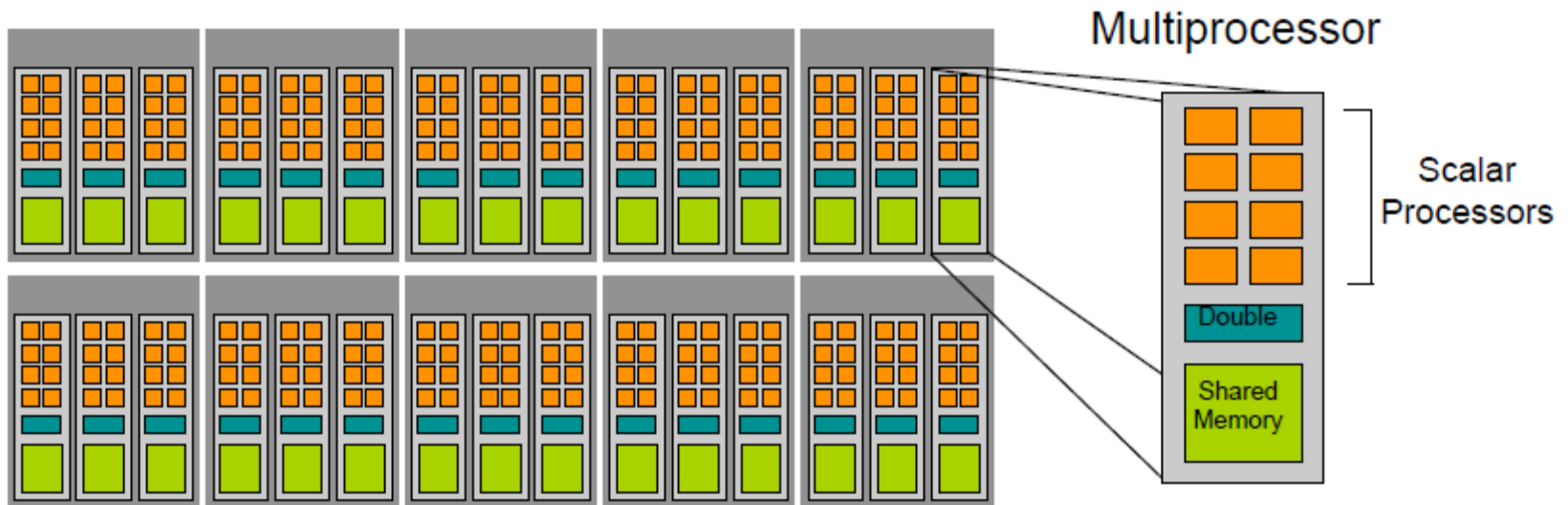
Streaming Multiprocessor (SM)



Device

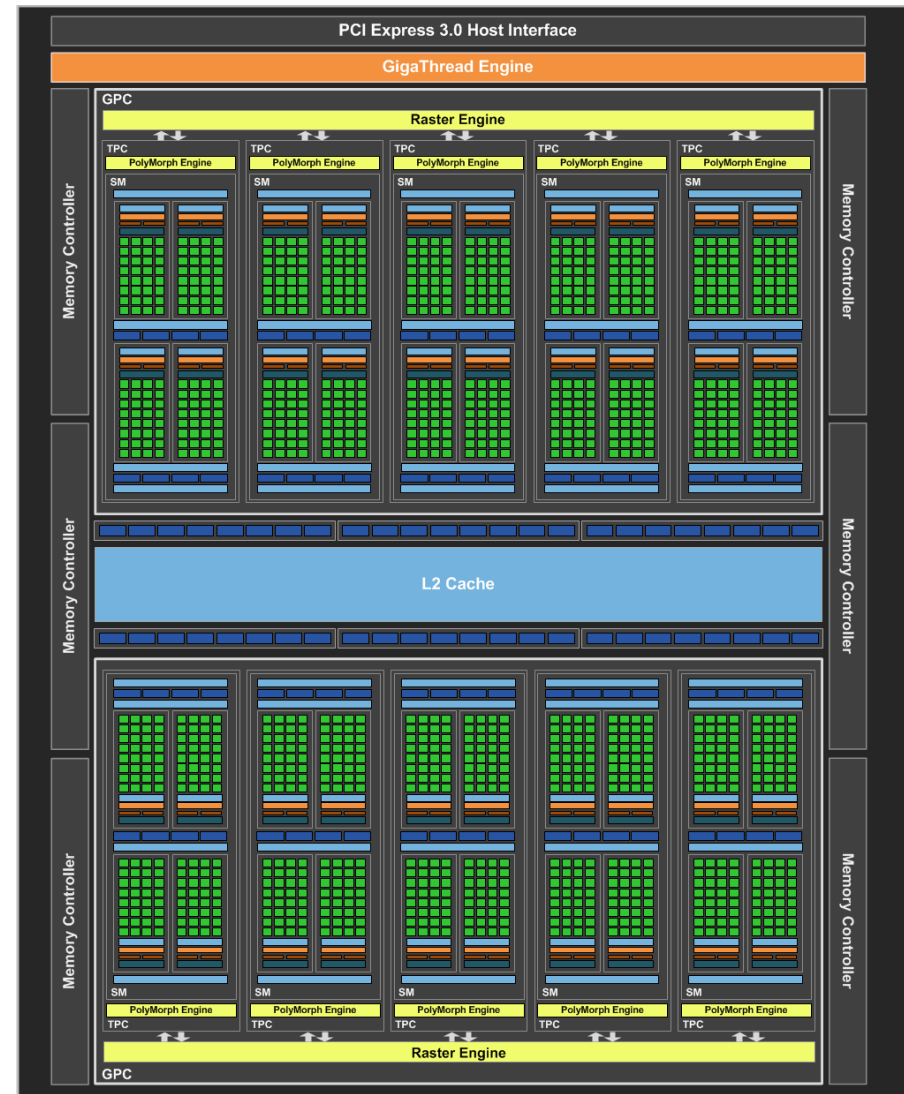
Hardverska implementacija (3)

- Primer arhitekture (stariji GT200 grafički procesor, *compute capability* 1.3)
 - 240 skalarni procesora (SP) izvršava niti jezgra
 - 30 *streaming* multiprocesora (SM) sadrži:
 - 8 skalarnih procesora
 - 16KB deljene memorije za saradnju na nivou bloka
 - 1 jedinicu za rad u pokretnom zarezu dvostruke preciznosti
 - 2 jedinice specijalne namene (SFU)
 - 64KB registarski fajl



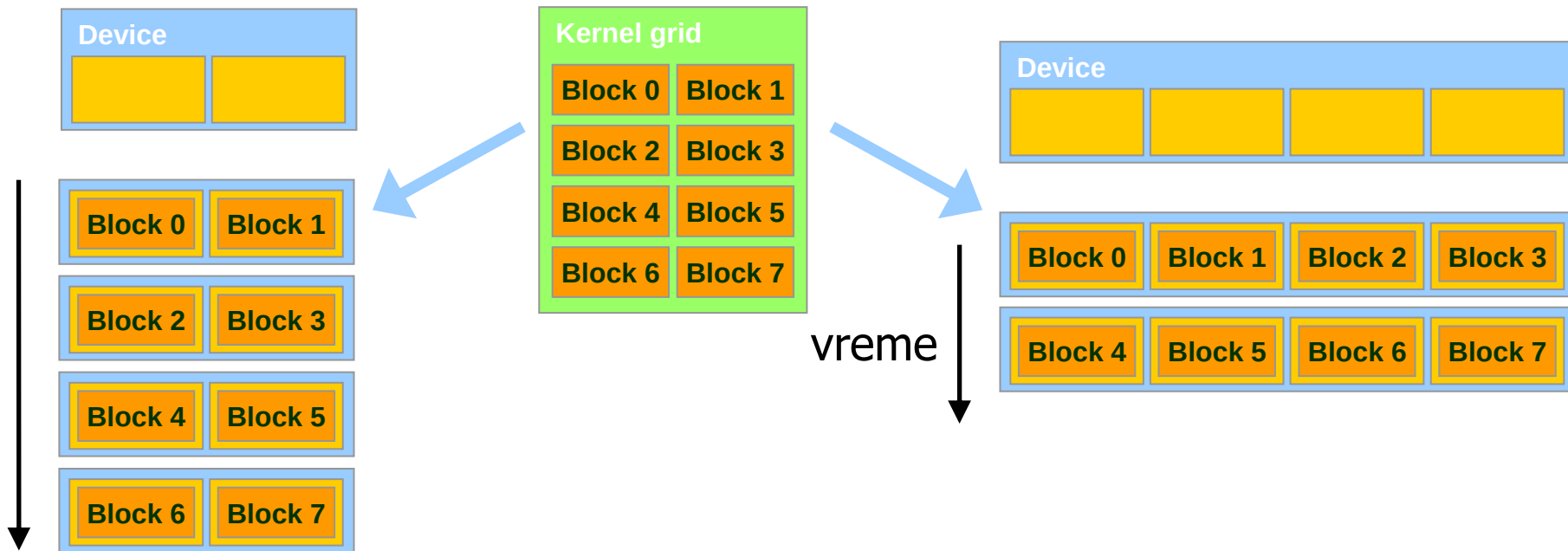
Hardverska implementacija (4)

- Primer arhitekture (noviji GP106-400-A1 GPU, *compute capability 7.0*)
 - 2 GPC (*Graphics Processing Clusters*)
 - 1280 CUDA jezgara (skalarnih procesora) 10 *streaming* multiprocesora (SM) sadrži:
 - 128 skalarnih procesora
 - 96KB deljenje memorije za saradnju na nivou bloka
 - 6 32-bitnih memorijskih kontrolera
 - 256KB registarski fajl
 - 1536KB L2 *cache*



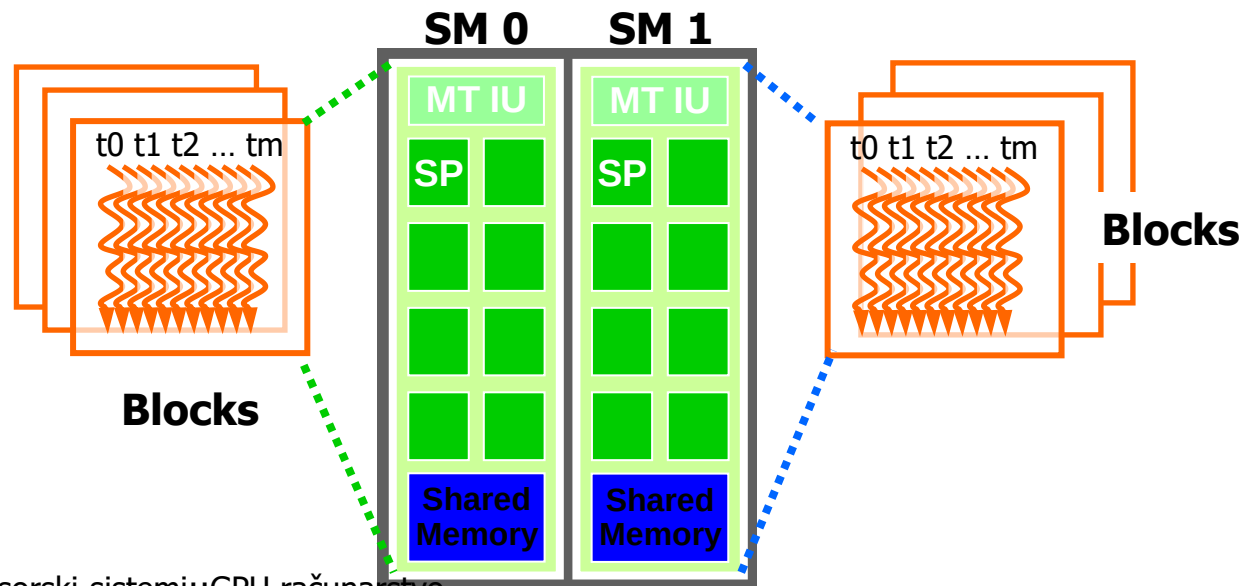
Izvršavanje blokova niti (1)

- Hardver raspoređuje blokove niti procesorima slobodno, bez ikakvih ograničenja
 - Jezgra su skalabilna nezavisno od broja paralelnih procesora na kojima se izvršavaju
 - Svaki blok se može izvršiti u bilo kome relativnom poretku u odnosu na druge blokove



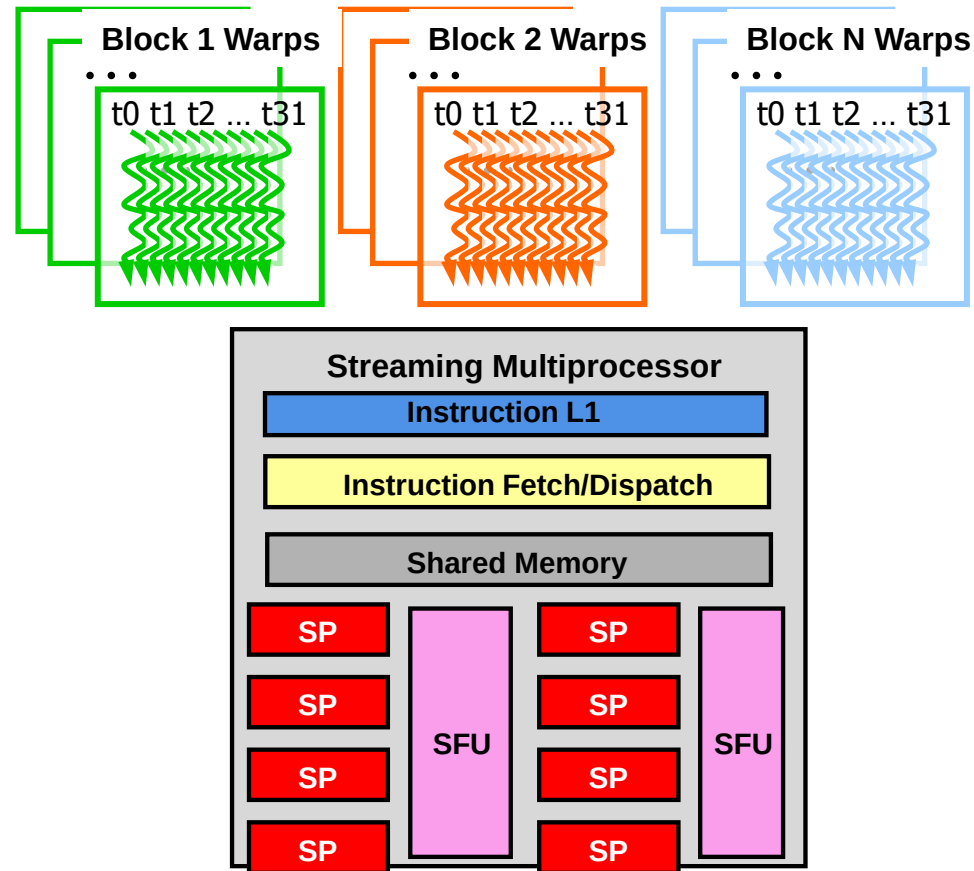
Izvršavanje blokova niti (2)

- Niti se multiprocesorima dodeljuju na nivou bloka
 - Najviše 8-32 blokova, zavisno od arhitekture, ukoliko drugi resursi dozvoljavaju
 - Npr. na Volta arhitekturi sa *compute capability* 7.0, najviše 2048 niti se može izvršavati na jednom multiprocesoru
 - Primer: 2 bloka po 1024 niti ili 4 bloka po 512 niti
 - Multiprocesor je zadužen za upravljanje i raspoređivanje niti za izvršavanje
 - Multiprocesor održava `blockIdx` i `threadIdx` za svaku nit



Raspoređivanje i izvršavanje niti (1)

- Niti iz bloka se na SM-u izvršavaju u jedinicama koje se nazivaju *warp*-ovi
 - SIMD izvršavanje
 - 1 *warp* = 32 niti
 - *Warp* je jedinica za raspoređivanje na SM-u
 - Implementaciona odluka
- Ako se na SM-u izvršava 2 bloka od po 1024 niti:
 - Svaki blok se deli na $1024 / 32 = 32$ *warp*-a
 - Na SM-u ukupno $32 * 2 = 64$ *warp*-a za izvršavanje

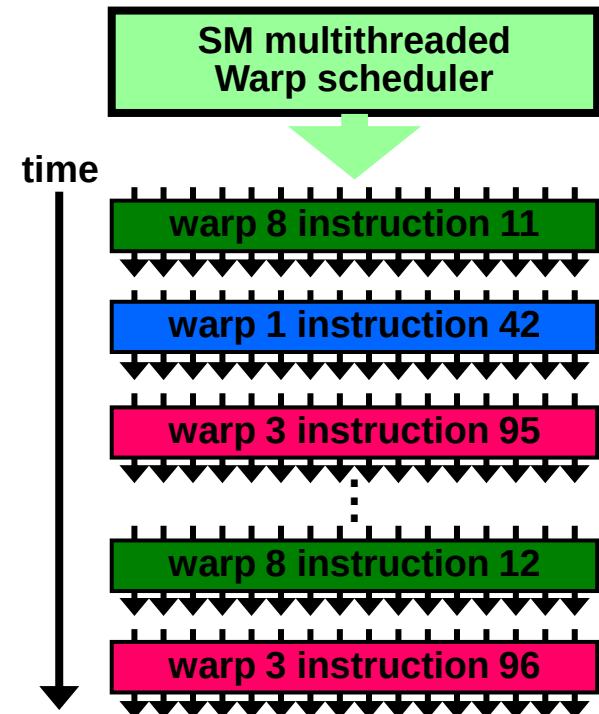


Raspoređivanje i izvršavanje niti (2)

- Blokovi niti se u *warp*-ove dele na sledeći način:
 - Niti u okviru bloka se linearizuju po vrstama
 - Indeksi niti unutar bloka su rastući i uzastopni
 - *Warp*-ovi se formiraju počevši od niti sa indeksom 0
 - Podela se uvek radi na isti način
 - To znanje se može iskoristiti kod kontrole toka
- Redosled izvršavanja *warp*-ova ne mora biti jednoznačan
 - Ukoliko postoje zavisnosti među nitima unutar jednog bloka, mora se vršiti sinhronizacija
 - Ugrađena funkcija `__syncthreads()` se koristi za sinhronizaciju niti na nivou bloka

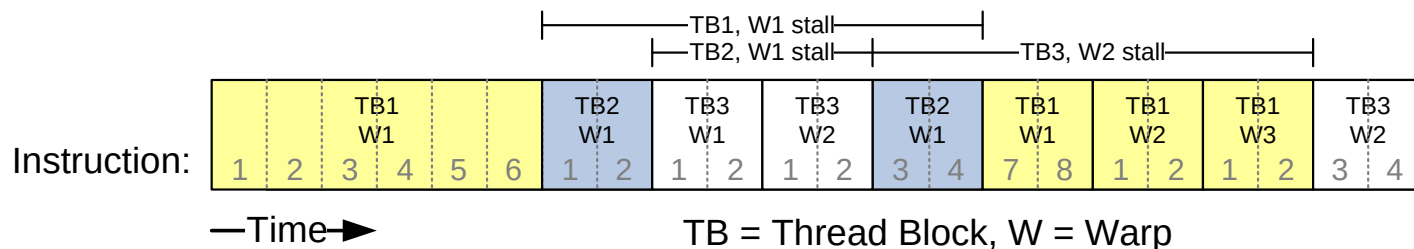
Raspoređivanje i izvršavanje niti (3)

- Na SM-ovima je implementirana *zero-overhead* politika raspoređivanja *warp*-ova
 - Broj *warp*-ova koji se izvršava na SM-u u jednom trenutku zavisi direktno od arhitekture SM-a
 - Raspoloživi za izvršavanje su samo oni *warp*-ovi kod kojih su svi operandi dostupni za izvršavanje u narednoj instrukciji
 - *Warp*-ovi se biraju za izvršavanje na osnovu prioriteta
 - *Round robin + aging*
 - Sve niti unutar *warp*-a izvršavaju istu instrukciju kada su izabrane



Raspoređivanje i izvršavanje niti (4)

- Prosečno vreme pristupa globalnoj memoriji je oko 200 ciklusa procesora
- Na novijim arhitekturama, 2 ciklusa su u proseku potrebna da bi se instrukcija izvršila u *warp*-u
 - Ako, u proseku, svaka četvrta instrukcija zahteva pristup memoriji:
 - Potrebno je najmanje 26 *warp*-ova na jednom SM-u da bi se u potpunosti tolerisalo kašnjenje
 - $2 * 4 * 26 = 208$



Kontrola toka (1)

- SM-ovi su procesori SIMD arhitekture
 - Dohvatanje i dekodovanje instrukcije se vrši zajednički za više procesnih jedinica
 - Sve niti moraju izvršavati isti kod u okviru *warp*-a
- Ovo se efikasno izvršava ukoliko sve niti u okviru *warp*-a slede istu putanju izvršavanja
 - Sva *if-else* uslovna grananja donose istu odluku
 - Sve petlje se izvršavaju isti broj iteracija
- Problem prilikom izvršavanja mogu biti *warp*-ovi kod kojih postoji divergencija u kontroli toka
 - *Branch (control) divergence* problem
 - Deo niti unutar *warp*-a izvršava jednu putanju koda, a deo niti izvršava drugu putanju (kod *if-else*)
 - Niti izvršavaju različit broj iteracija petlji

Kontrola toka (2)

- Grananja se izvršavaju tako što sve niti izvrše sve moguće putanje
 - Izvršavanje niti se serijalizuje
 - Izvršavaju se sve moguće putanje jedna po jedna
 - Alternativno, vrši se predikatsko (spekulativno) izvršavanje
 - Instrukcije pod predikatom mora da ubaci prevodilac
 - Zatim se u zavisnosti od uslova grananja, odbacuju/ne upisuju rezultati putanja koje nisu aktivne
- Ovakav način izvršavanja može dovesti do značajnih usporenja!
 - Broj putanja može biti veliki u zavisnosti od ugneždavanja kontrolnih struktura

Kontrola toka (3)

- Divergencija u kontroli toka nastaje kada je uslov grananja ili petlje zavisen od indeksa niti u okviru bloka
- Primer jezgra sa divergencijom kontrole toka:
 - `if (threadIdx.x > 10) { }`
 - U ovom slučaju postoje dve putanje izvršavanja niti unutar bloka
 - Prvih 10 niti izvršava jednu putanju, ostale niti drugu
 - Problem na nivou prvog *warp*-a u bloku
- Primer bez divergencije kontrole toka:
 - `if (blockIdx.x > 2) { }`
 - Odluka se donosi na nivou bloka
 - Svi *warp*-ovi u bloku izvršavaju iste putanje

Kontrola toka (4)

- Primer sabiranja dva vektora od 10000 elemenata
 - Pokreće se 10 blokova sa po 1024 niti
 - Ukupno 10240 niti, suvišne niti ne izvršavaju obradu
- Niti u blokovima 0-8 neće imati divergentne putanje
 - 288 *warp*-ova će se izvršavati na isti način
- U bloku 9 će biti divergentnih putanja:
 - Prva 24 *warp*-a u bloku će se izvršavati na isti način kao u prethodnim blokovima
 - *Warp* 25 u bloku će divergirati
 - Pola niti će izvršavati obradu, druga polovina neće
 - *Warp*-ovi nakon toga će sadržati niti koje ne izvršavaju obradu
- Minimalan efekat na performanse izvršavanja
 - Smanjuje je sa povećanjem veličine niza

Granularnost blokova (1)

- Dimenzije blokova niti treba pažljivo odabrati
- Primer množenja matrica
 - Na arhitekturi sa *compute capability* 7.0 (Volta) sa 2048 niti po SM-u
- Da li treba koristiti 8x8, 16x16 ili 32x32 blokove?
- Za blok dimenzija 8x8, izvršava se 64 niti po bloku
 - Maksimalno 2048 niti / 64 niti po bloku = 32 bloka niti
 - Svaki SM može da prihvati 32 blokova na izvršenje
 - Potpuna okupiranost
- Za blok dimenzija 16x16, izvršava se 256 niti po bloku
 - Maksimalno 2048 niti / 256 niti po bloku = 8 blokova niti
- Za blok dimenzija 32x32, ukupno bi se izvršavalo 1024 niti po bloku
 - Maksimalno 2048 niti / 1024 niti po bloku = 2 bloka niti

Granularnost blokova (2)

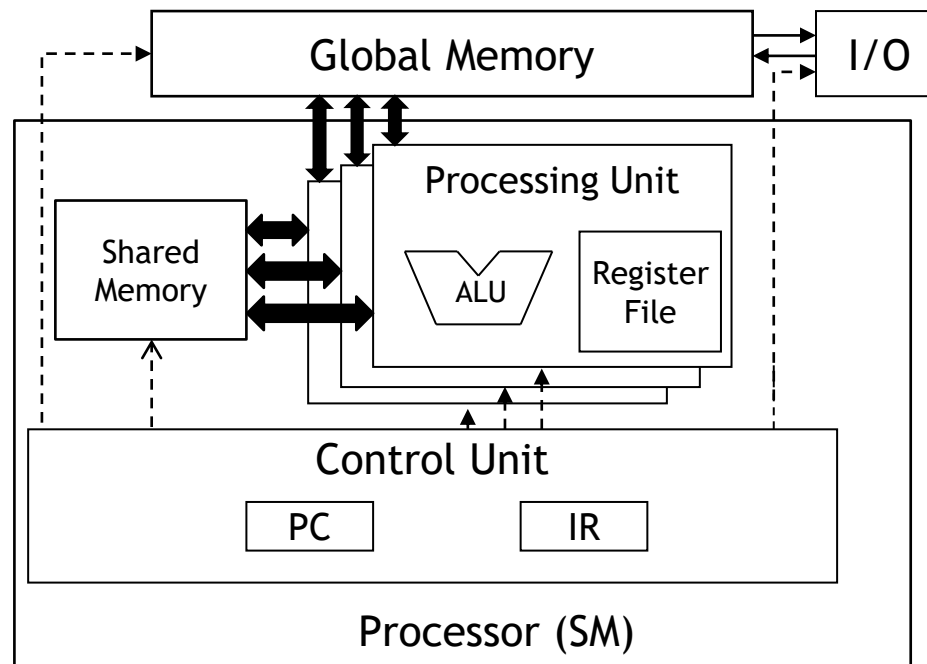
- Odluka o granularnosti blokova dosta zavisi od arhitekture grafičkog procesora
 - U sva tri slučaja iskorišćava se pun kapacitet pojedinačnog multiprocesora
 - Osim ako potreba za drugim resursima ne onemogućuje izvršavanje ovakve konfiguracije
 - Nameće korišćenje profajlera i sličnih alatki
- *Occupancy* metrika
 - Odnos broja aktivnih *warp*-ova i maksimalnog mogućeg broja *warp*-ova koji mogu teoretski da se izvršavaju na jednom SM-u simultano
- Na starijim generacijama ne mora biti ovako!
 - Proveriti parametre u okviru dokumentacije

Granularnost blokova (3)

- Primer starije, Fermi arhitekture (cc 2.0)
 - 1536 niti po SM-u, blok do 1024 niti, najviše 8 blokova po SM-u
- Za blok dimenzija 8x8, izvršava se 64 niti po bloku
 - Maksimalno 1536 niti / 64 niti po bloku = 36 blokova niti
 - Svaki SM može da prihvati samo 8 blokova na izvršenje
 - Samo 512 niti će se izvršavati na svakom SM-u!
- Za blok dimenzija 16x16, izvršava se 256 niti po bloku
 - Maksimalno 1536 niti / 256 niti po bloku = 6 blokova niti
 - Iskorišćava se pun kapacitet pojedinačnog multiprocesora
 - Osim ako potreba za drugim resursima ne onemogućí izvršavanje ovakve konfiguracije
- Za blok dimenzija 32x32, izvršava se 1024 niti po bloku
 - Maksimalno 1536 niti / 1024 niti po bloku = 1 blok niti
 - Samo jedan blok se izvršava na SM-u
 - Redukcija paralelizma od 1/3!

Hardverski pogled na CUDA memorije

- Hijerarhija memorijskih prostora
 - Registarski fajl, deljena memorija, globalna memorija



Alokacija registara (1)

- Svaki multiprocesor poseduje registarski fajl
 - Registri se dinamički dodeljuju blokovima koji se izvršavaju na pojedinačnom multiprocesoru
 - Veličina registarskog fajla zavisi od arhitekture
 - 8-64K 32-bitnih registara na svakom SM-u
 - Niti iz drugih blokova ne mogu pristupati registrima dodeljenim jednom bloku niti
 - Svaka niti pristupa samo registrima koji su joj dodeljeni
 - Najviše 255 registara po niti
- Broj blokova koji se izvršava na jednom SM-u direktno zavisi od njihovih potreba za registrima
 - Što može dovesti do slabog iskorišćenja resursa

Alokacija registara (2)

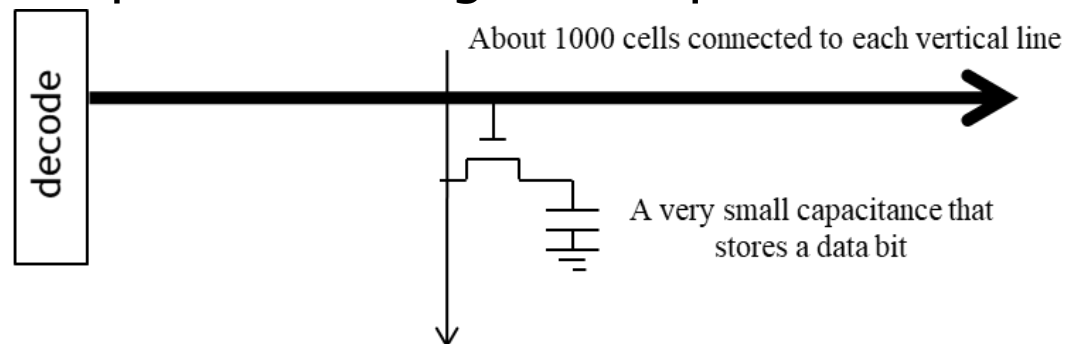
- Pretpostavimo sledeći scenario:
 - Na raspolaganju je 8K registarski fajl
 - Na primer, na staroj G80 arhitekturi (cc 1.1)
 - Jezgro se izvršava u blokovima veličine 16x16
 - Svaka nit koristi 10 registara
- Za izvršavanje svakog bloka je potrebno $256 * 10 = 2560$ registara
 - $2560 * 3 = 7680 < 8192$
 - Na SM-u se može izvršavati 3 bloka niti, što se tiče alokacije registara

Alokacija registara (3)

- Ukoliko se broj registara po svakoј niti poveća samo za jedan:
 - Za izvršavanje svakog bloka će biti potrebno $256 * 11 = 2816$ registara
 - $2816 * 3 = 8448 > 8192$
 - Samo dva bloka će moći da se izvrše na SM-u
 - $2816 * 2 = 5632 << 8192$
 - Redukcija paralelizma skoro 1/3!
- Nije kritično na novijim arhitekturama
 - Međutim, prevodilac svojim (ne)optimizacijama može da napravi problem
 - Kvalifikator `__launch_bounds__` se može iskoristiti da ograniči broj registara po niti

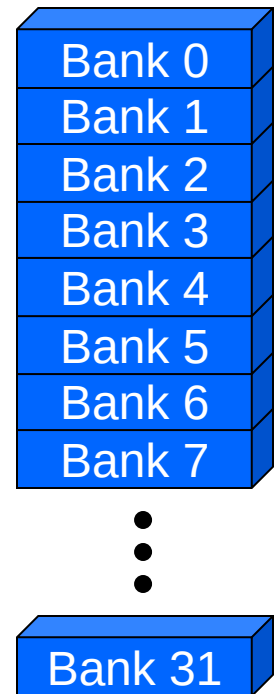
Paralelna memorijska arhitektura (1)

- Memorijski propusni opseg (*bandwidth*) predstavlja jedno od najvažnijih uskih grla modernih višejezgarnih i mnogojezgarnih procesora
 - Kod paralelne mašine, veliki broj niti pristupa memoriji
 - Zahteva se velika količina podataka za obradu
- Zahteva specifičnu organizaciju DRAM podsistema
 - Pristup u transakciji (DRAM *burst*), memorijske banke i kanali
- Značajan uticaj na performanse
 - Veoma izražen problem kod grafičkih procesora



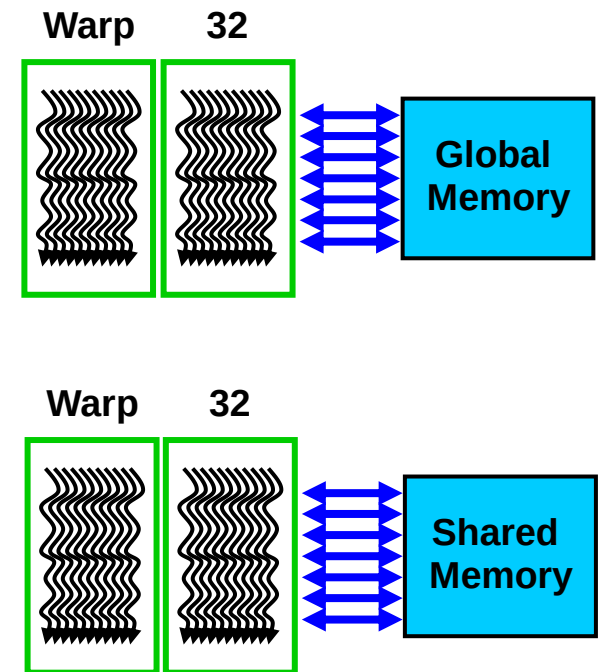
Paralelna memorijska arhitektura (2)

- Memorija je preklapljena i podeljena u memorijske banke na grafičkom procesoru
 - *Memory interleaving* tehnika
 - Globalna i deljena memorija
 - Vrlo bitno za postizanje velikog propusnog opsega
- Svaka memorijska banka može da usluži jedan zahtev u jednom ciklusu
 - Celokupna memorija može simultano da usluži onoliko pristupa koliko ima memorijskih banki
- Više simultanih pristupa istoj banki dovodi do konflikta
 - Konfliktne pristupi se serijalizuju



Paralelna memorijska arhitektura (3)

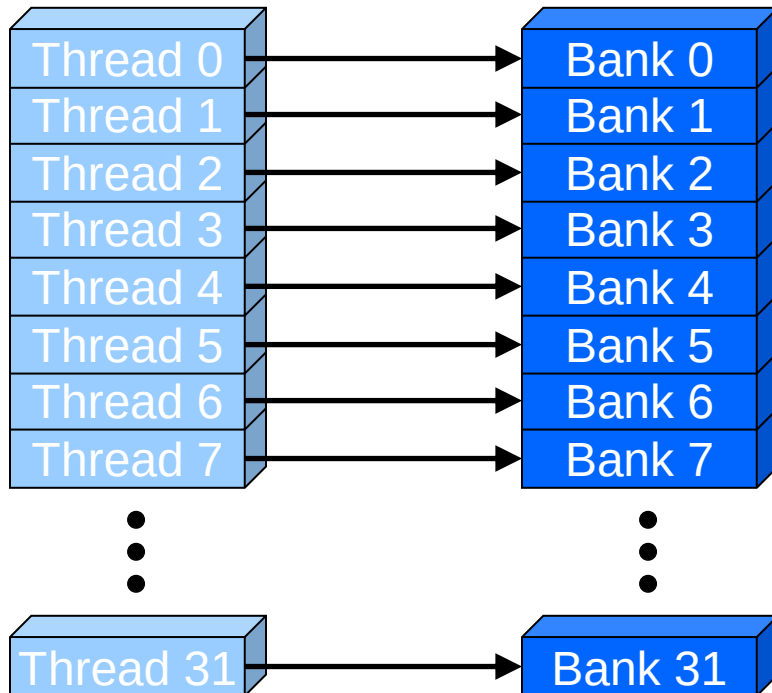
- Memorija je podeljena u 32 banke
 - Uzastopne 32-bitne reči se dodeljuju uzastopnim memorijskim bankama
- Pristup memoriji na CUDA se kombinuje u transakcije
 - Najbolje performanse se dobijaju kada sve niti unutar *warp*-a pristupaju uzastopnim memorijskim lokacijama
 - Tada nema konflikata
 - Konflikti su mogući jedino unutar *warp*-a



Primeri pristupa memoriji (1)

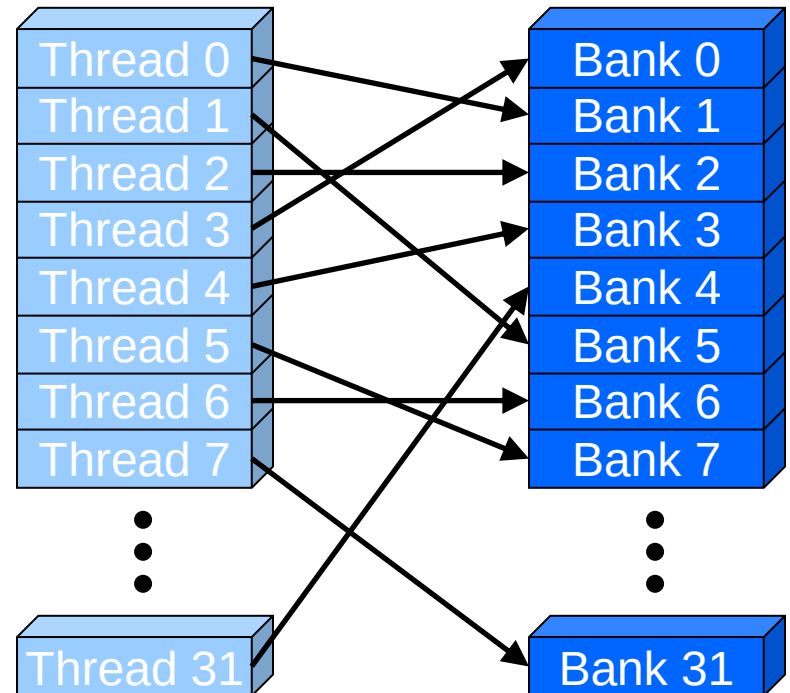
- Nema konflikata

- Linearno adresiranje
- $\text{stride} = 1$



- Nema konflikata

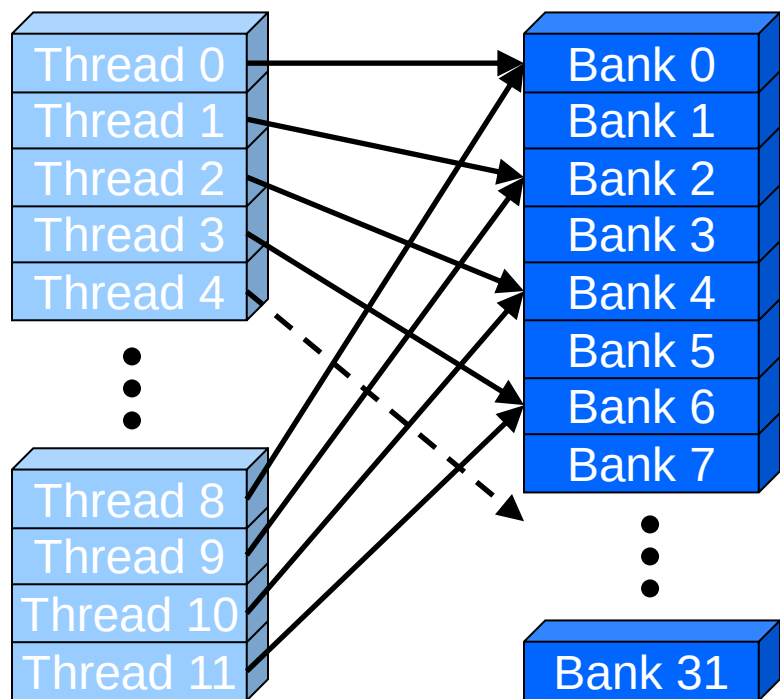
- Slučajan pristup memoriji



Primeri pristupa memoriji (2)

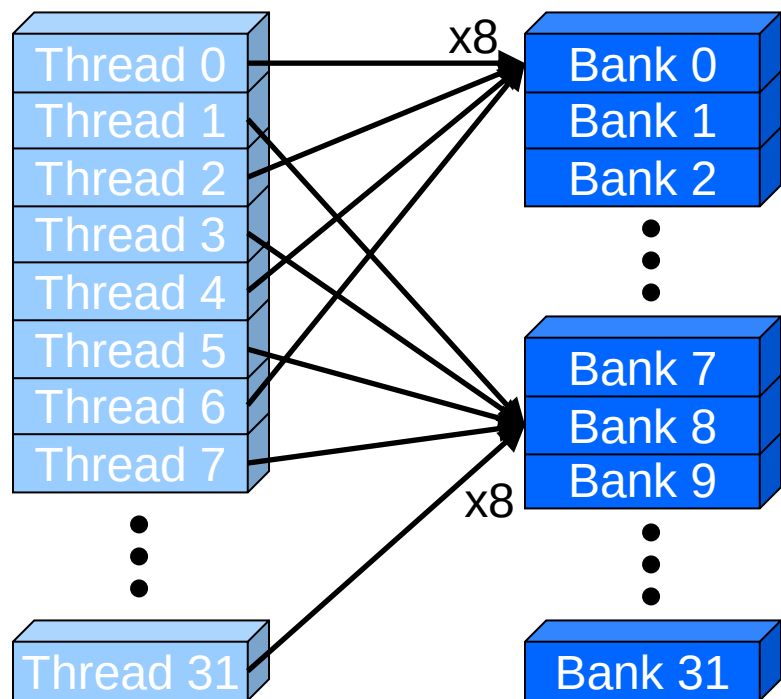
○ Dvostruki konflikt

- Linearno adresiranje
- stride = 2



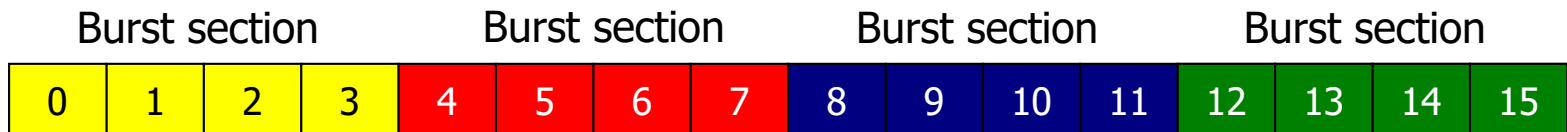
○ 8-struki konflikt

- Linearno adresiranje
- stride = 8



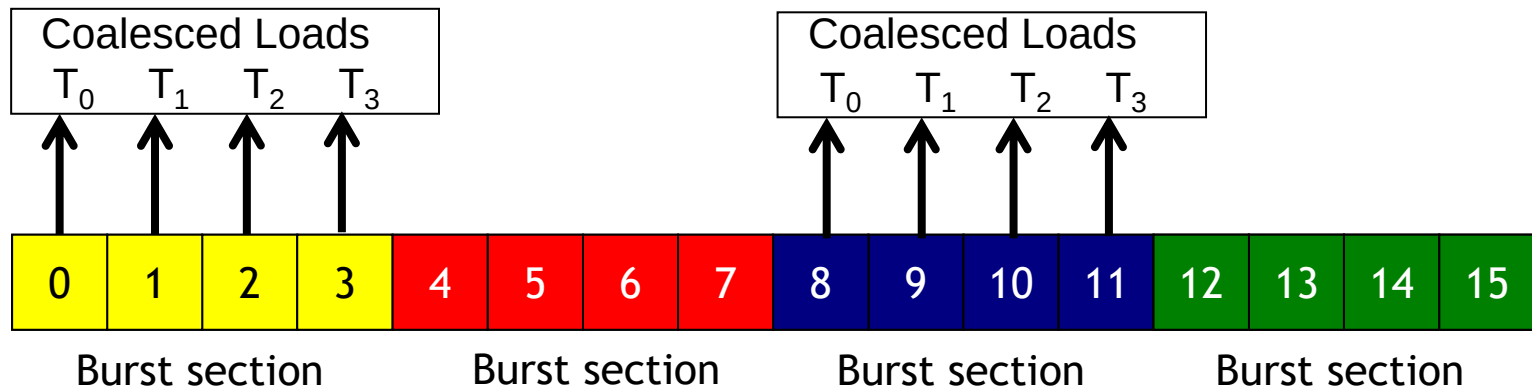
Pristup memoriji u transakcijama (1)

- Najbolje performanse se dobijaju kada sve niti unutar *warp*-a pristupaju uzastopnim memorijskim lokacijama
 - Spojeni/sjedinjeni pristup memoriji (*memory coalescing*)
 - Posledica pristupa podacima u *burst*-u (transakciji)
- Ceo adresni prostor je podeljen u *burst* sekcije
 - Kada god se pristupi lokaciji, sve ostale lokacije u okviru iste sekcije se takođe dostavljaju
 - U praksi, adresni prostor je reda veličine 4GB, a *burst* sekcije su veličine 128 bajtova ili više



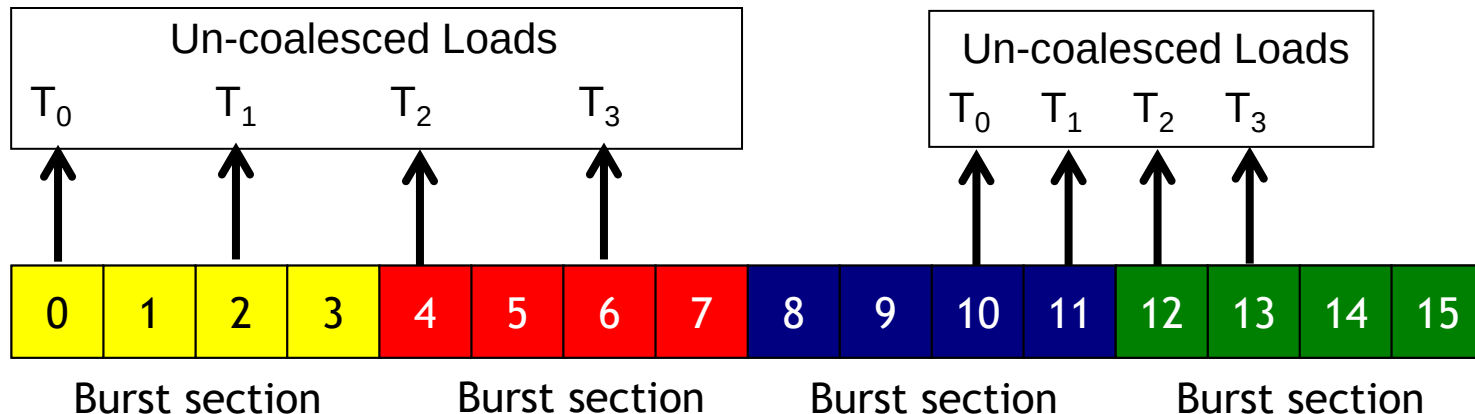
Pristup memoriji u transakcijama (2)

- Pristup u jednoj transakciji se dešava kada:
 - Sve niti u okviru *warp*-a izvršavaju *load* instrukciju
 - Ukoliko pristupi svim lokacijama upadaju u okviru iste *burst* sekcije
 - Tada će biti generisan samo jedan zahtev DRAM memoriji
 - Pristup će biti potpuno sjedinjen (*fully coalesced*)
- U suprotnom, biće generisano više transakcija



Pristup memoriji u transakcijama (3)

- Pristup u više transakciji se dešava kada:
 - Niti u okviru *warp*-a izvršavaju *load* instrukciju
 - Pristupi lokacijama se protežu preko granica *burst* sekcija
 - Biće generisano više zahteva DRAM memoriji
 - Pristup nije sjedinjen (*non-coalesced*)
- Neki bajtovi kojima je pristupljeno će biti odbačeni
 - Neće biti korišćeni od strane niti

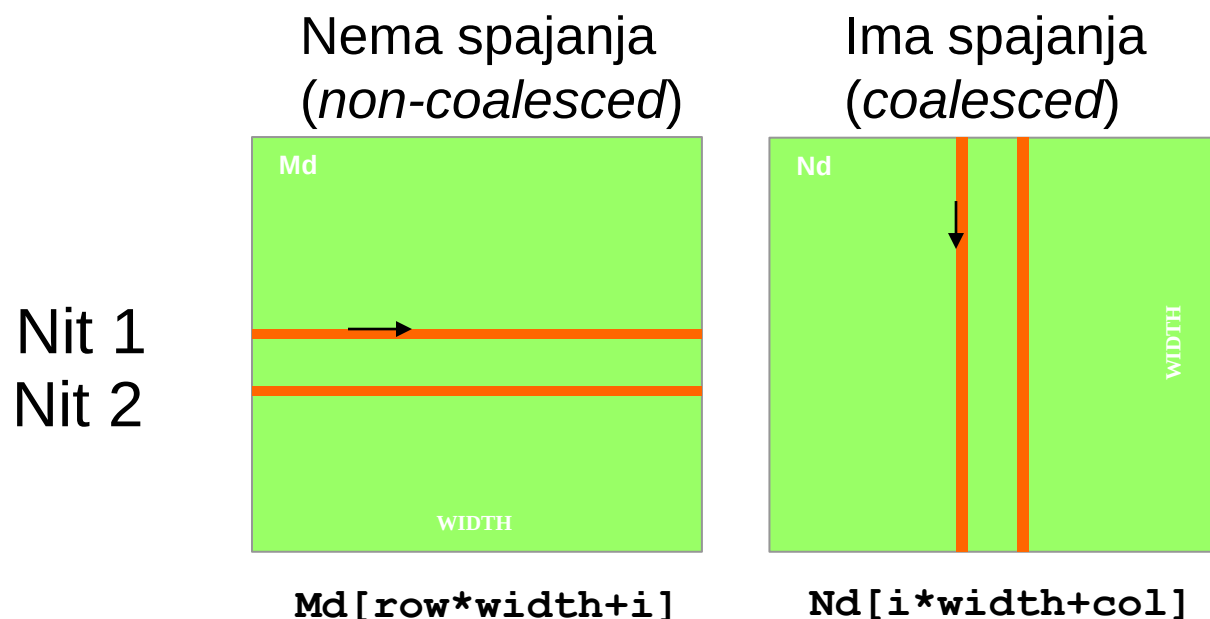


Pristup memoriji u transakcijama (4)

- Moguća okvirna provera da li je pristup sjedinjen u okviru *warp*-a
- Adresni izraz za pristup nizu **A** treba da bude oblika:
 - **A[*expr* + threadIdx.x]**
 - Gde je **expr** izraz sa članovima koji su nezavisni od **threadIdx.x**
- Tada niti pristupaju sukcesivnim lokacijama
 - Pristup će biti potpuno sjedinjen ukoliko niti pristupaju lokacijama iz istog DRAM *burst*-a

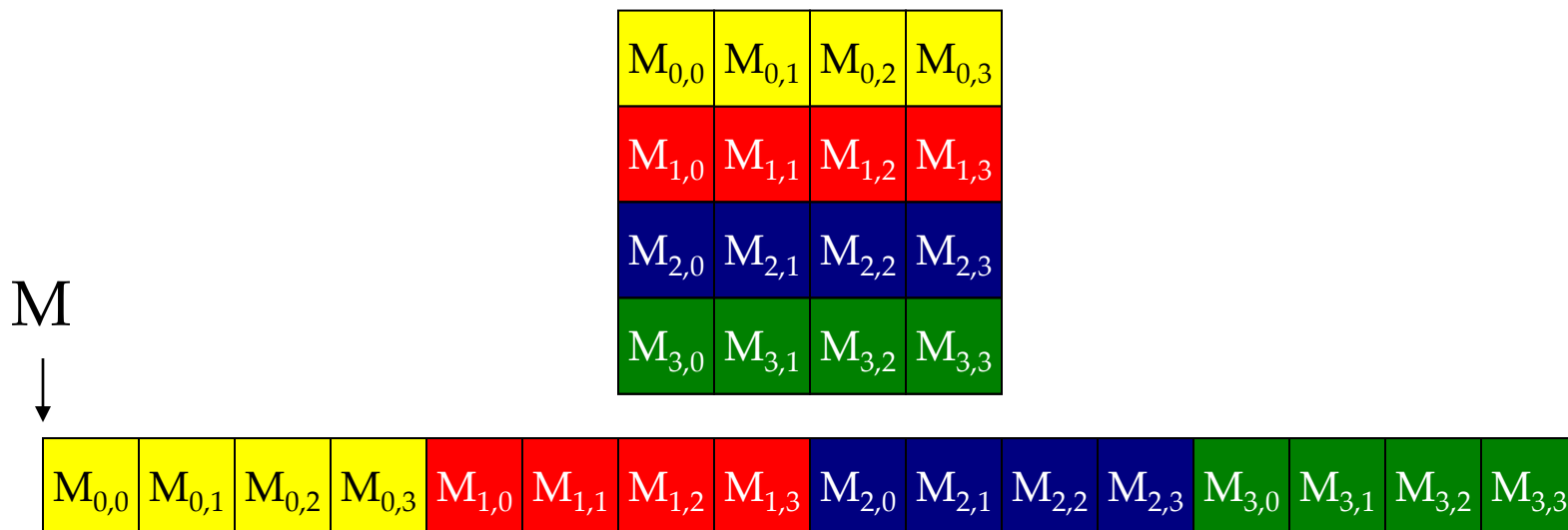
Pristup memoriji u transakcijama – množenje matrica (1)

- Primer množenja matrica prikazuje oba načina pristupa
 - Pristup vrstama matrice M od strane niti iz iste vrste
 - Niti pristupaju istom elementu u jednom trenutku
 - Pristup kolonama matrice N od strane niti iz iste vrste
 - Niti pristupaju susednim elementima u jednom trenutku



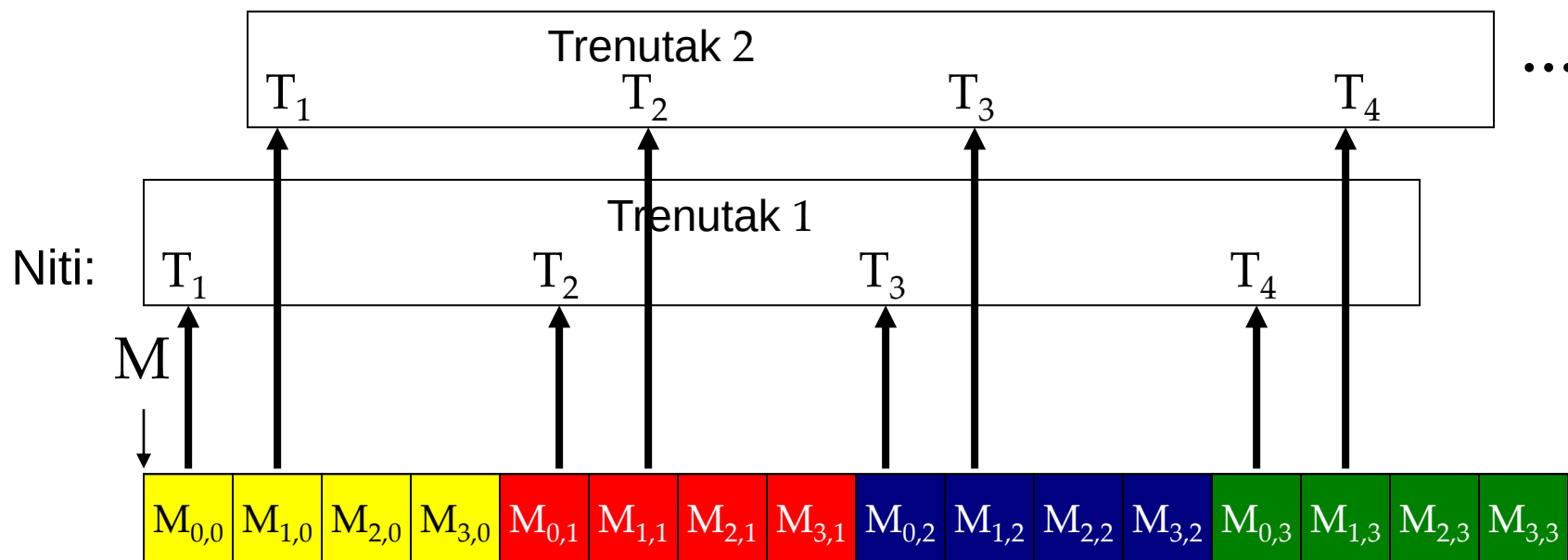
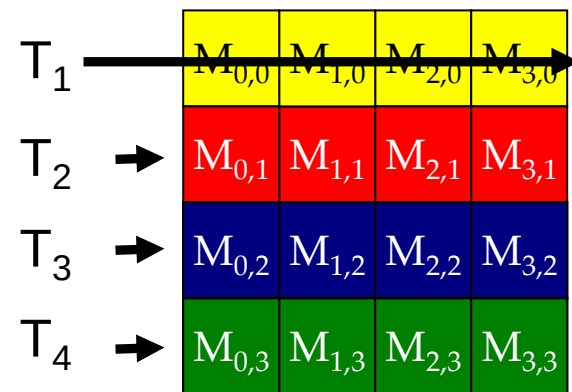
Smeštanje matrica u memoriju (podsetnik)

- Matrice se na programskom jeziku C podrazumevano smeštaju po vrstama
 - Matrica se na uređaju smešta linearizovana



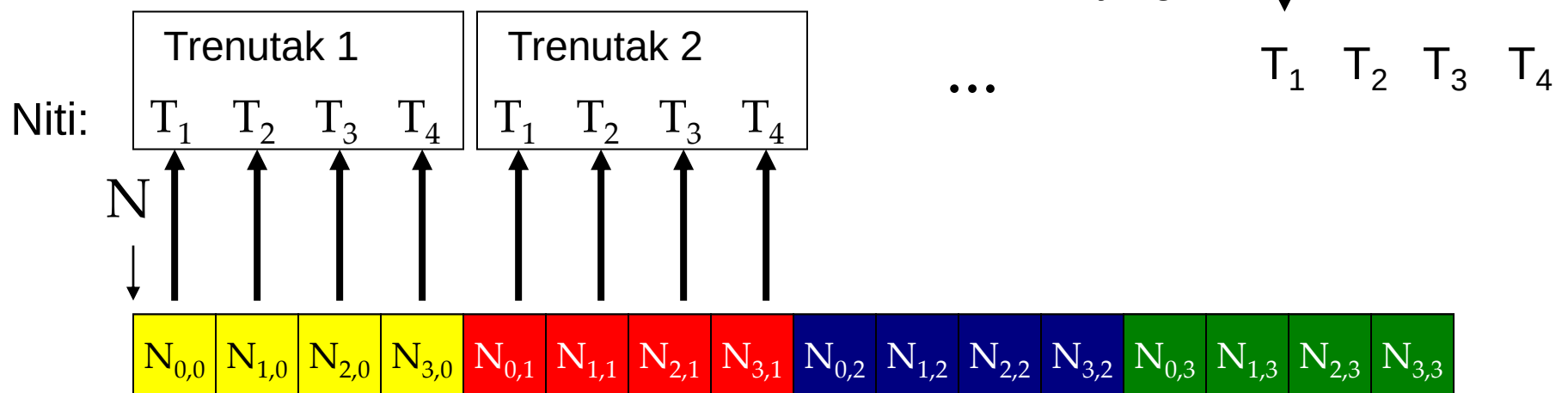
Pristup memoriji u transakcijama – množenje matrica (2)

- Pristup vrstama matrice M_d
- Ne postoji kombinovanje u transakciju:
 - Niti ne pristupaju uzastopnim lokacijama u jednom trenutku
 - *Non-coalesced access*



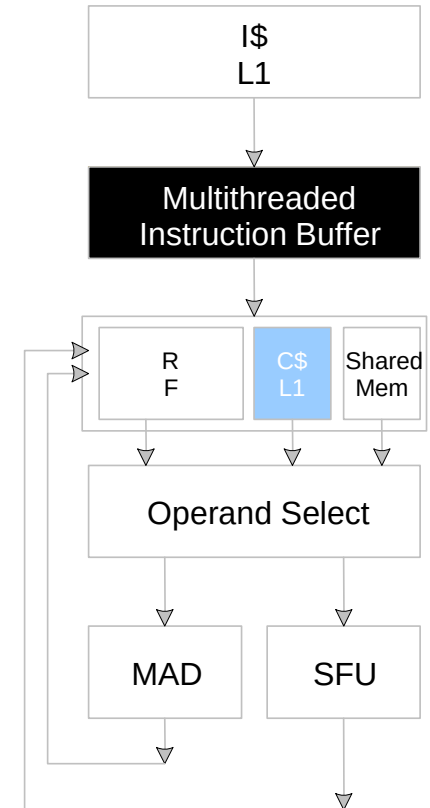
Pristup memoriji u transakcijama – množenje matrica (3)

- Pristup kolonama matrice N_d
- Postoji kombinovanje u transakciju:
 - Niti pristupaju uzastopnim lokacijama u svakom trenutku
 - *Coalesced access*



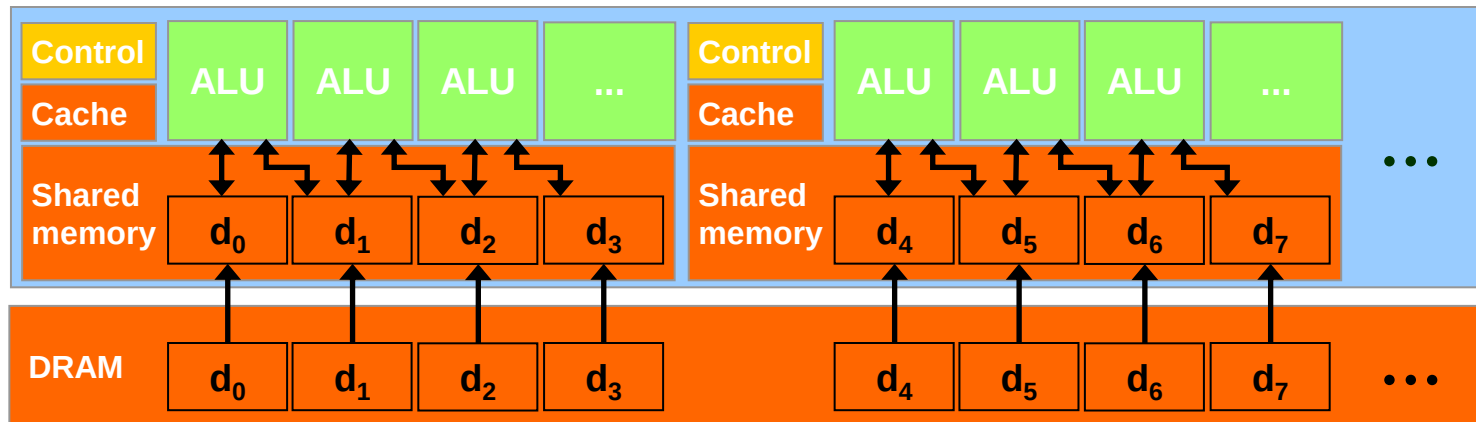
Konstantna memorija

- Region konstantne memorije se nalazi u DRAM-u (64KB veličine)
 - Međutim, pristup konstantnoj memoriji je keširan radi bržeg pristupa
 - Svaki SM ima svoj L1 keš
- Pojedinačna vrednost iz konstantne memorije može biti objavljena svim nitima unutar *warp-a*
 - *Broadcast* mehanizam
 - Efikasan način za pristup vrednosti koja je zajednička za sve niti unutar bloka
- Pogodno koristiti kada god pristup podacima podrazumeva samo čitanje



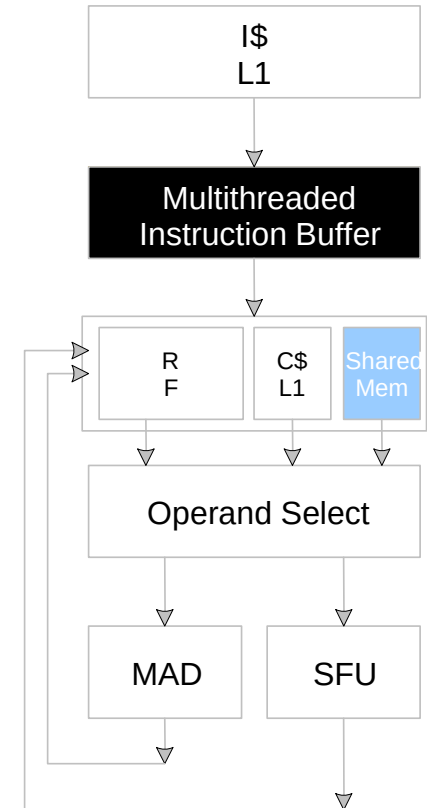
Deljena memorija (1)

- Za efikasno deljenje podataka na nivou bloka niti može se koristiti deljena memorija
 - Omogućava veliku uštedu memorijskog propusnog opsega
 - Nalazi se na čipu svakog SM-a (*on-chip memory*)
 - Kombinovani L1 keš i deljena memorija
 - Efikasan pristup, 3-4 ciklusa procesora



Deljena memorija (2)

- Deljena memorija se može smatrati kao neka vrsta keša upravljano od strane korisnika
 - *User managed cache/scratchpad*
- Svaki SM ima ograničenu veličinu deljene memorije
 - Kapacitet zavisan od arhitekture GPU
 - Starije arhitekture 16-48KB
 - Novije arhitekture 64-228KB
- Deljena memorija je podeljena u 32 banke sastavljenih od 32-bitnih reči
 - Niti iz istog bloka mogu slobodno da čitaju i pišu



Deljena memorija (3)

- Pristup deljenoj memoriji će biti brz skoro kao pristup registrima, ukoliko nema konflikata prilikom pristupa memorijskim bankama
- Brz pristup deljenoj memoriji:
 - Ako sve niti iz *warp*-a pristupaju različitim memorijskim bankama, nema konflikta
 - Ako sve niti iz *warp*-a pristupaju istoj adresi, nema konflikta
 - Koristi se *broadcast* mehanizam
- Spor pristup deljenoj memoriji:
 - Konflikt prilikom pristupa se dešava kada više niti iz *warp*-a pristupaju istoj memorijskoj banci
 - Ne nužno i istoj lokaciji
 - Pristupi se tada serijalizuju

Deljena memorija (4)

- Deljena memorija se unutar jezgra zadaje ključnom rečju **__shared__**

- Može se zadati eksplicitno (statički)

```
__shared__ float DynamicSharedMem[BLOCK_SIZE];
```

- Može se specificirati prilikom poziva jezgra unutar izvršne konfiguracije

- Takva memorija se alocira u promenljive deklarisanе kao

```
extern __shared__ float DynamicSharedMem[];
```

- Primer pozivanja takvog jezgra:

```
__global__ void KernelFunc(...);
```

```
...
```

```
size_t SharedMemBytes = 64; // 64 bytes of shared memory
```

```
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

Tipična strategija programiranja (1)

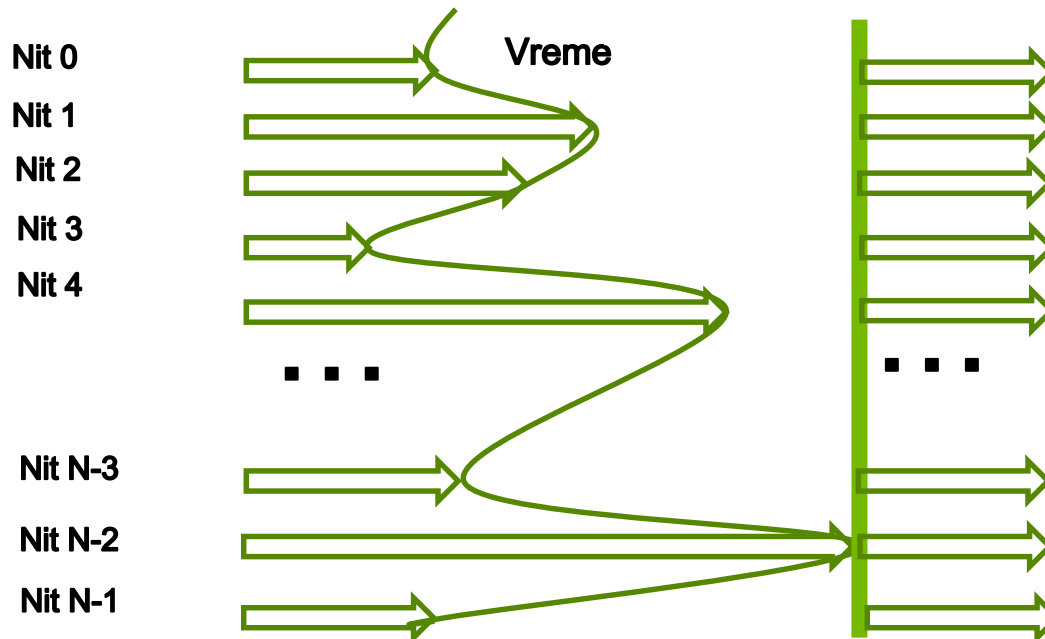
- Globalna memorija se nalazi u memoriji uređaja (DRAM)
 - Mnogo sporiji pristup nego kod deljene memorije
- Deljena memorija se upotrebljava kako bi se smanjili efekti memorijskog propusnog opsega na performanse
- Uobičajena strategija za sprovođenje izračunavanja podrazumeva podelu podataka na podblokove (*tiles*)
 - Fokus rada niti se prebacuje na manje podblokove podataka u jednom trenutku vremena

Tipična strategija programiranja (2)

- *Tiling* tehnika koristi se prednost brze deljene memorije
 - Podaci se dele na podblokove koji mogu da stanu u deljenu memoriju
 - Izvršavanje se deli na faze
- Svaki podblok se obrađuje jednim blokom niti
 - Podblok se učitava iz globalne u deljenu memoriju od strane niti iz bloka
 - Kako bi se omogućilo čitanje u transakciji
 - Vrš se obrada podbloka u deljenoj memoriji
 - Svaka nit može efikasno da pristupi svakom podatku iz podbloka
 - Rezultati se kopiraju nazad iz deljene memorije u globalnu
 - Prelazi se na obradu narednog podbloka

Tipična strategija programiranja (3)

- Bitno je uskladiti rad niti po fazama
 - Može zahtevati korišćenje sinhronizacije na barijeri
 - Koristi se API funkcija `__syncthreads()`



Sinhronizacija na barijeri

- Sinhronizacija na barijeri je moguća samo na nivou bloka niti
 - Može biti pozvana samo unutar jezgra
- Jednom kada sve niti dostignu sinhronizacionu tačku, izvršavanje se nastavlja normalno
- Poziv treba koristiti da bi se izbegli RAW / WAR / WAW hazardi pristupa deljenoj ili globalnoj memoriji
- Koristi se često u *tiled* algoritmima:
 - Obezbeđuje da svi elementi podbloka budu učitani
 - Obezbeđuje da svi elementi podbloka budu konzumirani pre naredne faze algoritma
- Mora se pažljivo koristiti unutar uslovnih grananja
 - Sve niti unutar bloka moraju izvršavati istu granu

Množenje matrica – podsetnik (1)

- Jezgro:

```
__global__ void MatrixMulKernel
(float* Md, float* Nd, float* Pd, int Width) {

    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

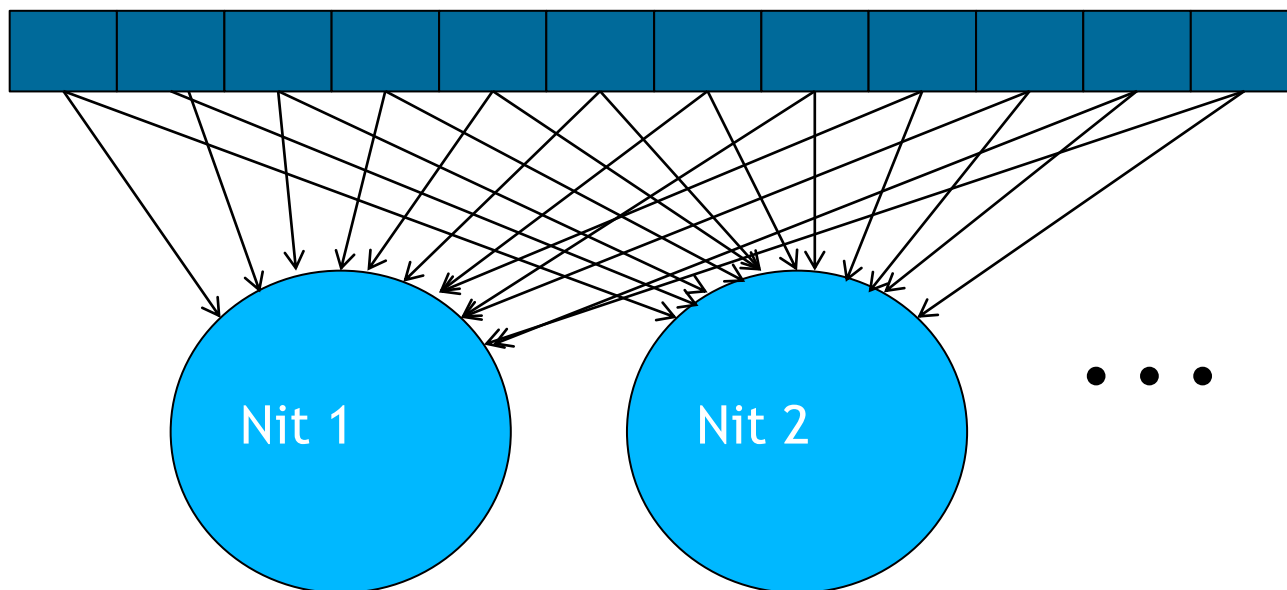
    float Pvalue = 0;
    // Each thread computes one element of the block submatrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row * Width + Col] = Pvalue;
}
```

Množenje matrica – podsetnik (2)

- Svaka nit iz iste vrste će pristupati WIDTH puta elementima matrica M i N
 - Postoje suvišni, redundantni pristupi elementima, kako na nivou vrste matrice M, tako i na nivou kolone matrice N

Globalna memorija



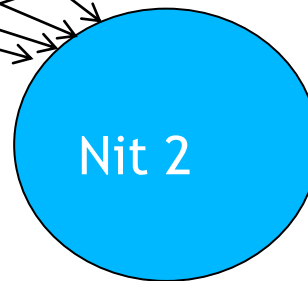
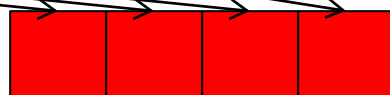
Množenje matrica – deljena memorija (1)

- Ideja je da se matrice podele na podblokove
 - Niti će učitavati jedan po jedan podblok i obrađivati ih

Globalna memorija



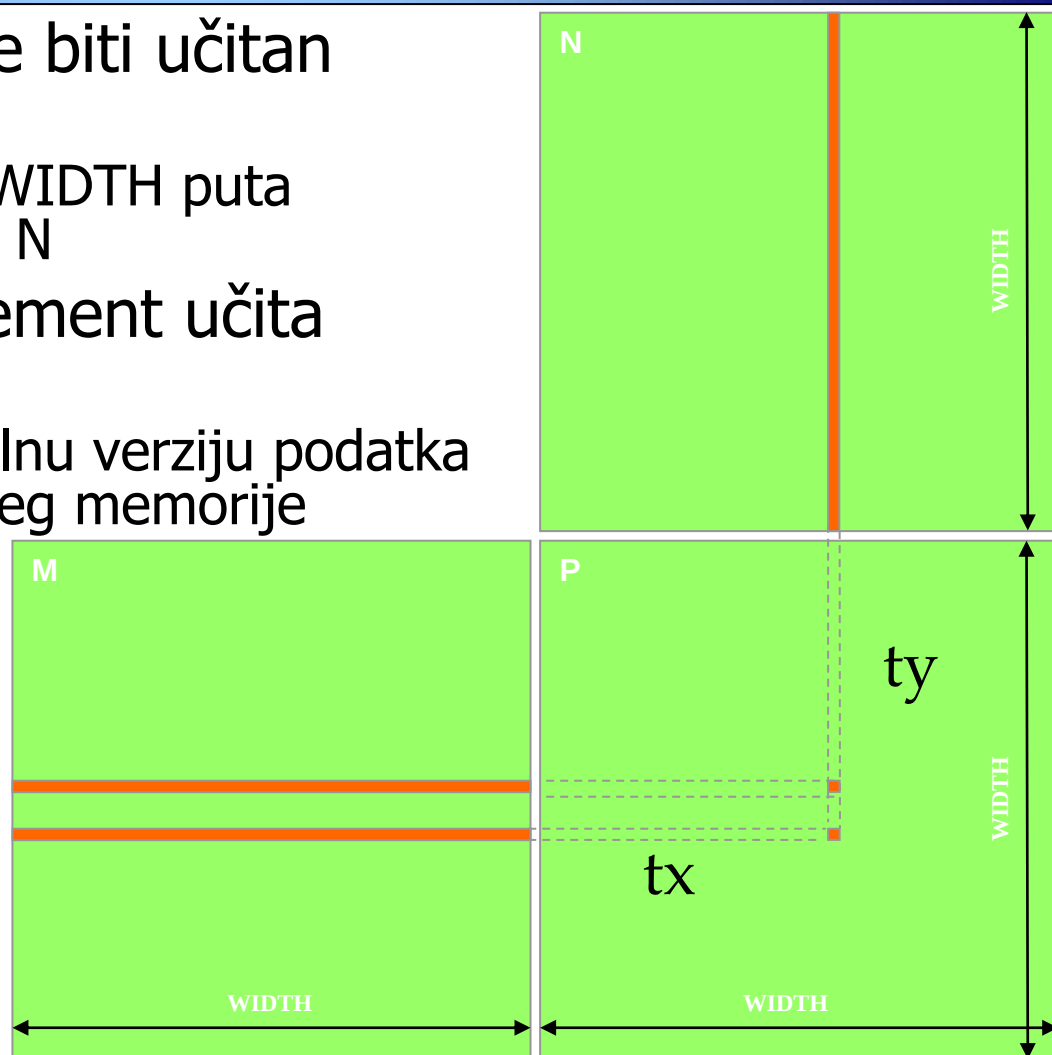
Deljena memorija



...

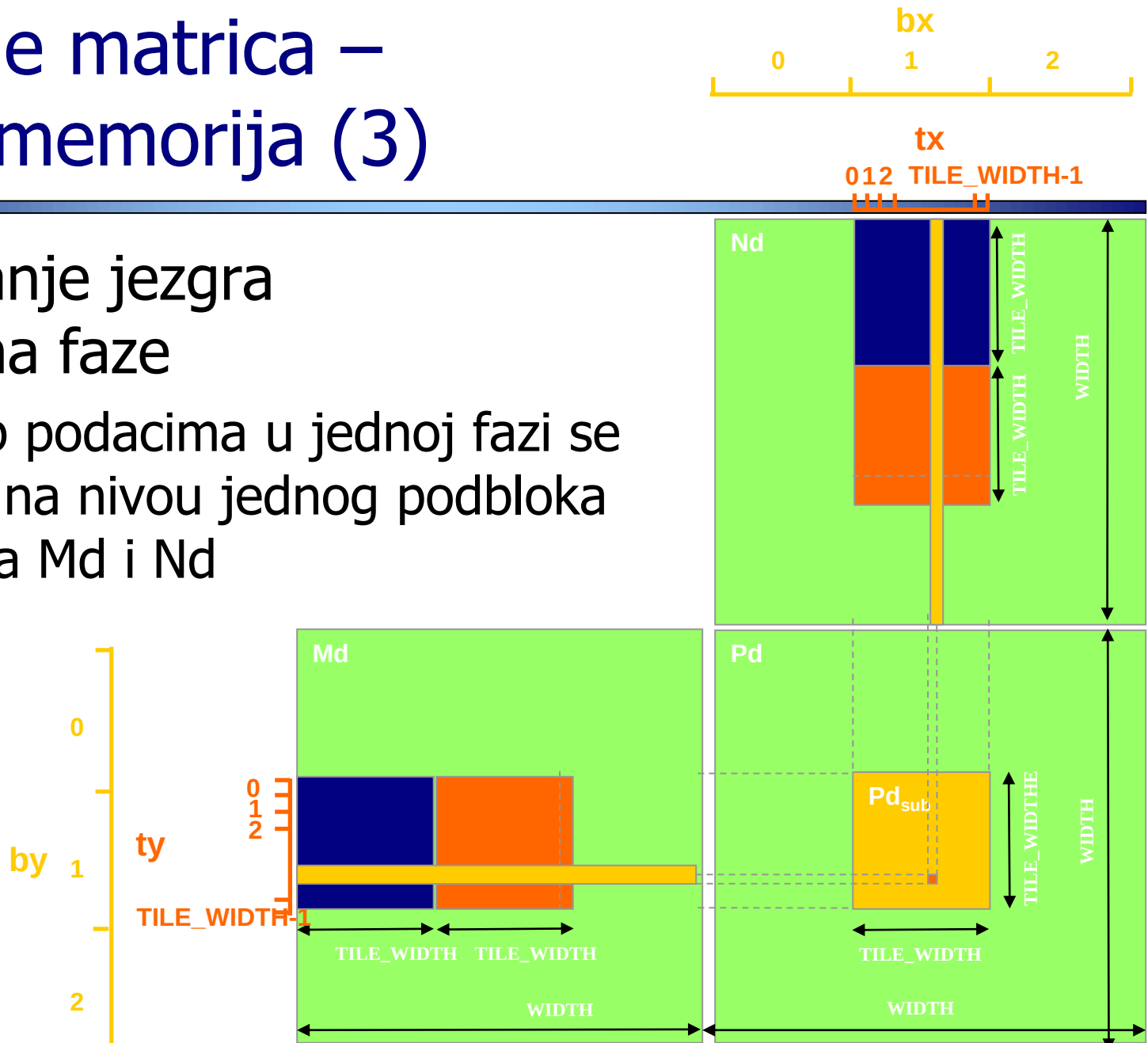
Množenje matrica – deljena memorija (2)

- Svaki ulazni element će biti učitán WIDTH puta
 - Svaka nit će pristupati WIDTH puta elementima matrica M i N
- Ideja je da se svaki element učitá u deljenu memoriju
 - Više niti će koristiti lokalnu verziju podatka da uštede propusni opseg memorije



Množenje matrica – deljena memorija (3)

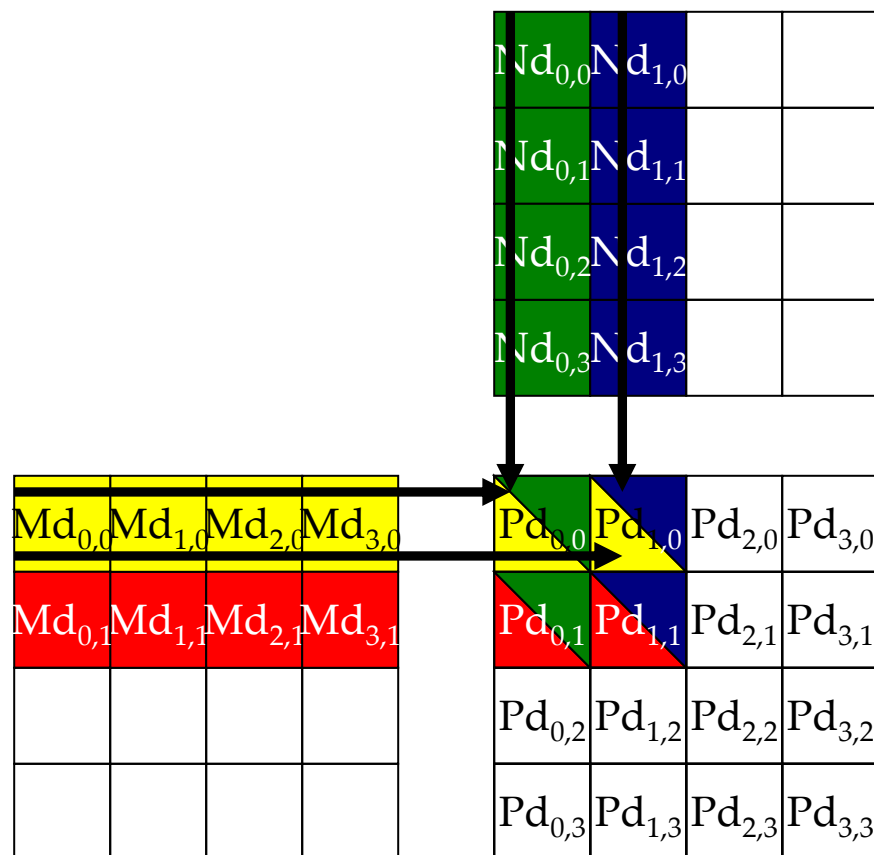
- Izvršavanje jezgra se deli na faze
 - Pristup podacima u jednoj fazi se odvija na nivou jednog podbloka matrica Md i Nd



Množenje matrica – deljena memorija (4)

- Primer izračunavanja jednog podbloka rezultujuće matrice

- Matrica dimenzija 4x4
- Podblokovi dimenzija 2x2



Množenje matrica – deljena memorija (5)

- Svaki element matrica M_d i N_d se koristi tačno dva puta prilikom izračunavanja podbloka matrice P_d

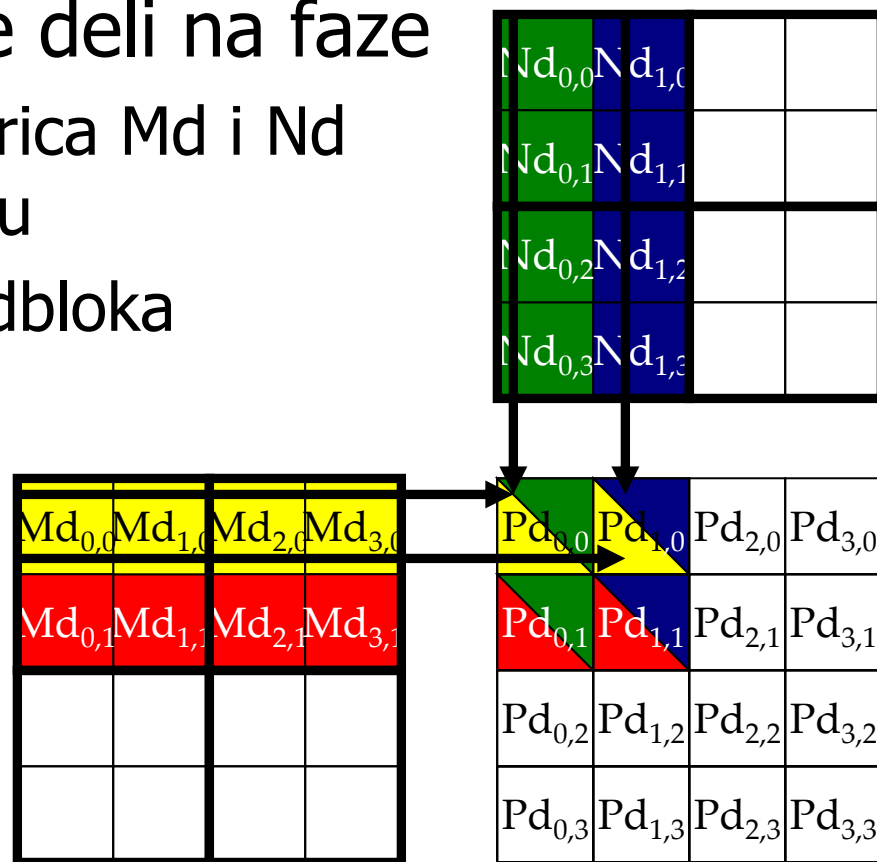
Redosled pristupa

$P_{0,0}$ thread _{0,0}	$P_{1,0}$ thread _{1,0}	$P_{0,1}$ thread _{0,1}	$P_{1,1}$ thread _{1,1}
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Množenje matrica – deljena memorija (6)

- Izračunavanje matrice se deli na faze

- Najpre se podblokovi matrica M_d i N_d učitaju u deljenu memoriju
- Zatim se izračuna deo podbloka matrice P_d
- Svaka faza koristi jedan deo



Množenje matrica – deljena memorija (7)

	Faza 1			Faza 2		
$T_{0,0}$	Md_{0,0} ↓ Mds _{0,0}	Nd_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1}	Md_{2,0} ↓ Mds _{0,0}	Nd_{0,2} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1}
$T_{1,0}$	Md_{1,0} ↓ Mds _{1,0}	Nd_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1}	Md_{3,0} ↓ Mds _{1,0}	Nd_{1,2} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1}
$T_{0,1}$	Md_{0,1} ↓ Mds _{0,1}	Nd_{0,1} ↓ Nds _{0,1}	PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1}	Md_{2,1} ↓ Mds _{0,1}	Nd_{0,3} ↓ Nds _{0,1}	PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1}
$T_{1,1}$	Md_{1,1} ↓ Mds _{1,1}	Nd_{1,1} ↓ Nds _{1,1}	PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1}	Md_{3,1} ↓ Mds _{1,1}	Nd_{1,3} ↓ Nds _{1,1}	PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1}

vreme



Množenje matrica – deljena memorija (8)

- Kompletan kod:

```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width / TILE_WIDTH, Width / TILE_WIDTH);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>> (Md, Nd, Pd, Width);
```

Množenje matrica – deljena memorija (9)

```
__global__ void MatrixMulKernel
(float* Md, float* Nd, float* Pd, int Width) {

    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    //Identify the row and column of the
    //Pd element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
```

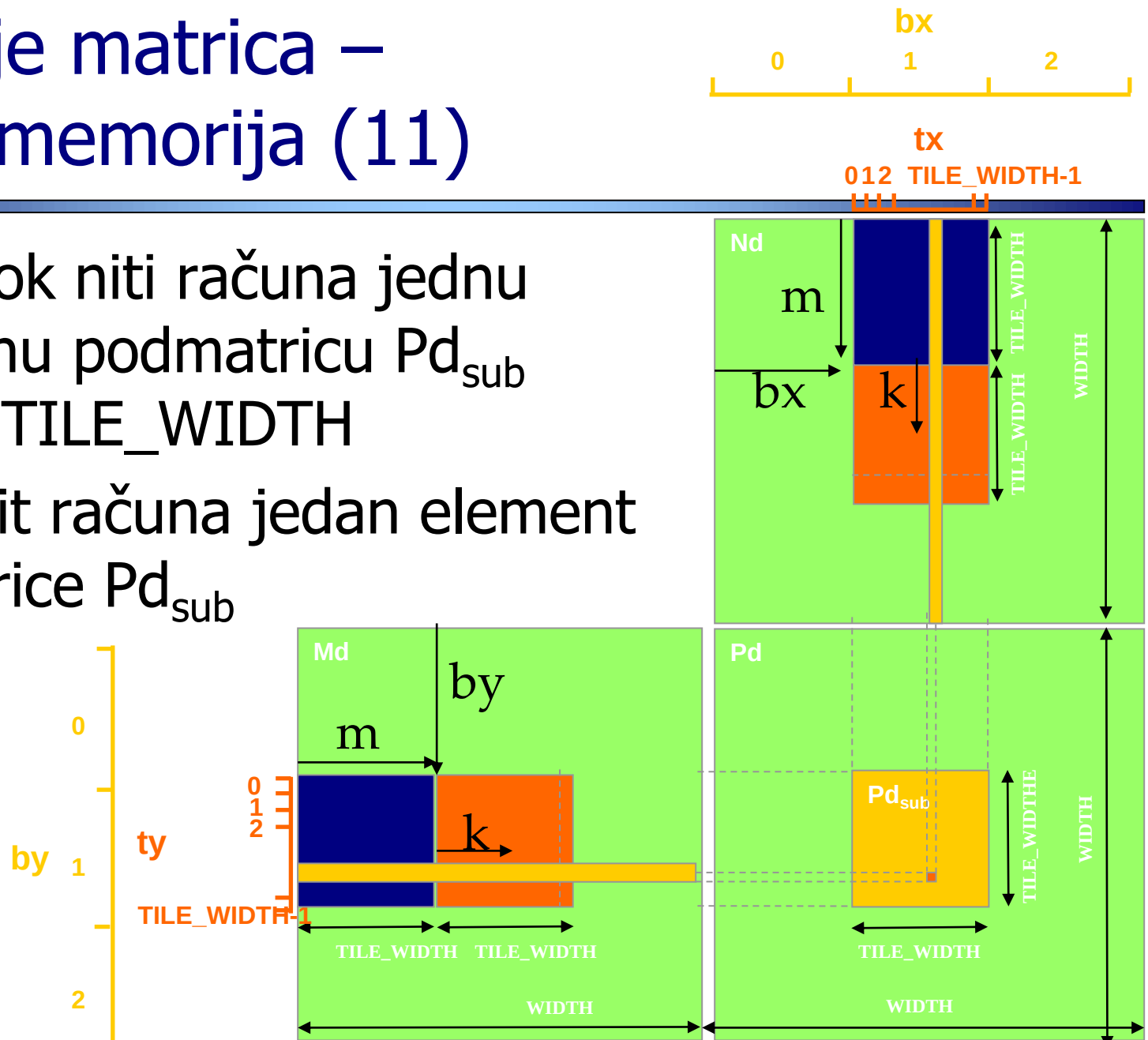
Množenje matrica – deljena memorija (10)

```
// Loop over the Md and Nd tiles required to compute
//the Pd element
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles
    // into shared memory
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
        Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
}
Pd[Row*Width+Col] = Pvalue;
}
```

Množenje matrica – deljena memorija (11)

- Svaki blok niti računa jednu kvadratnu podmatricu Pd_{sub} veličine $TILE_WIDTH$
- Svaka nit računa jedan element podmatrice Pd_{sub}



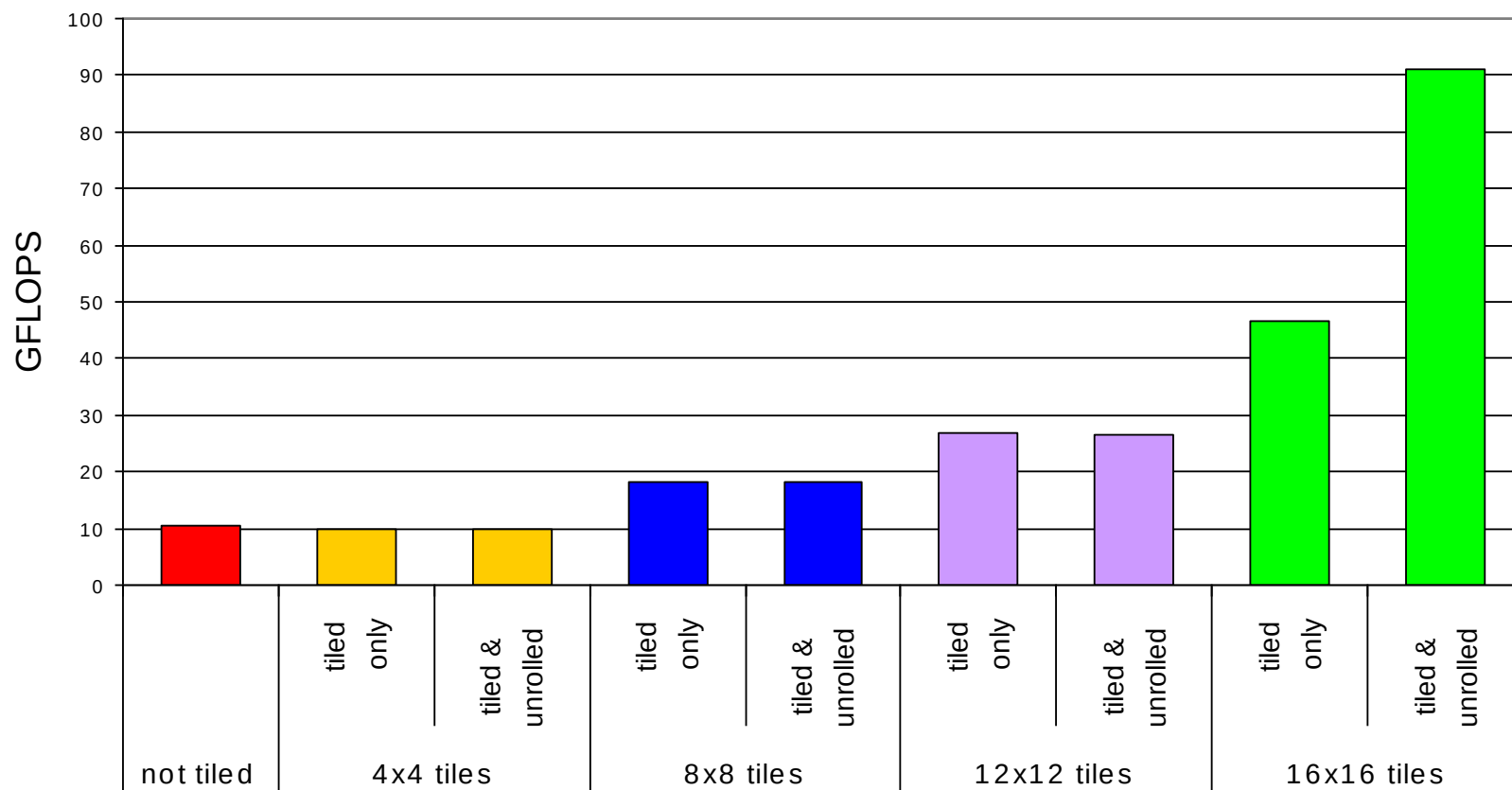
Množenje matrica – performanse (1)

- Svaki blok niti treba da ima veliki broj niti
 - Za $\text{TILE_WIDTH} = 16$, biće $16 \times 16 = 256$ niti po bloku
 - Za $\text{TILE_WIDTH} = 32$, biće $32 \times 32 = 1024$ niti po bloku
- Za veličinu bloka $\text{TILE_WIDTH} = 16$
 - Svaki blok niti radi
 $2 \times 256 = 512$ pristupa (čitanja) iz globalne memorije
 - Zatim se vrši $256 \times (2 \times 16) = 8192$ operacija
- Za veličinu bloka $\text{TILE_WIDTH} = 32$
 - Svaki blok niti radi
 $2 \times 1024 = 2048$ pristupa (čitanja) iz globalne memorije
 - Zatim se vrši $1024 \times (2 \times 32) = 65536$ operacija

Množenje matrica – performanse (2)

- Korišćenjem deljene memorije, memorijski propusni opseg više nije limitirajući faktor
 - Broja računskih operacija mnogo veći od broja pristupa memoriji
 - $8192 \gg 512$ TILE_WIDTH = 16
 - $65536 \gg 2048$ TILE_WIDTH = 32
- Kapacitet deljene memorije može biti ograničavajući faktor kod *tiled* algoritama
 - Za TILE_WIDTH = 16, svaki blok niti koristi $2 \times 256 \times 4B = 2KB$ deljene memorije
 - Za TILE_WIDTH = 32, svaki blok niti koristi $2 \times 1024 \times 4B = 8KB$ deljene memorije
 - Može ograničiti broj blokova koji se izvršavaju na SM-u

Množenje matrica – performanse (3)



Atomične operacije (1)

- Postoje situacije kod kojih je potrebno obezbediti atomičnost operacija nad globalnom memorijom
 - Kako bi se izbegli hazardi podataka (*data race*)
 - Npr. inkrementiranje globalnog brojača
- Atomične operacije omogućavaju izvršavanje *read-modify-write* operacije nad memorijskom lokacijom
 - Podržano od strane hardverskih instrukcija
- Hardverski se obezbeđuje da nijedna druga nit ne može da pristupi lokaciji dok se trenutna operacija ne završi
 - Druge niti koje pokušavaju atomičnu operaciju će biti blokirane u redu za čekanje
 - Atomične operacije se izvršavaju serijalizovano nad istom lokacijom

Atomične operacije (2)

- Atomične operacije se izvršavaju pozivanjem ugrađenih (intrinzičkih) funkcija (*intrinsics*) u okviru jezgra
 - Atomično sabiranje, oduzimanje
 - Inkrementiranje, dekrementiranje
 - Minimum, maksimum
 - Razmena vrednosti lokacija (*exchange*), *CAS (compare and swap)*
- Mogu se iskoristiti za ograničenu globalnu sinhronizaciju i zaštitu deljenih objekata
- Zavisno od arhitekture grafičkog procesora
 - *Compute capability* definiše dostupne operacije

Atomične operacije (3)

- Familija *atomic add* intrinzičkih funkcija

```
int atomicAdd(int* address, int val);
```

- Čita 32-bitnu reč sa stare lokacije `address` u globalnoj ili deljenoj memoriji
- Računa `old + val`
- Smešta rezultat u memoriju na istu adresu
- Funkcija vraća staru vrednost `old`

- Druge funkcije u familiji:

- Unsigned 32-bit integer atomic add

```
unsigned int atomicAdd(unsigned int*  
address, unsigned int val);
```

- Unsigned 64-bit integer atomic add

```
unsigned long long int atomicAdd(unsigned long  
long int* address, unsigned long long int val);
```

- Single-precision floating-point atomic add

```
float atomicAdd(float* address, float val);
```

Dodatne CUDA API funkcionalnosti

- CUDA izvršni (*runtime*) API pruža razne mogućnosti za upravljanje uređajem i izvršavanjem programa
- Postoje funkcije za:
 - Upravljanje uređajem
 - Sa podrškom za više grafičkih procesora na jednom sistemu (Multi-GPU)
 - Upravljanje memorijom
 - Upravljanje teksturama
 - Saradnju sa eksternim (grafičkim) API-jima
 - Upravljanje greškama
 - Upravljanje događajima
- CUDA sistem se automatski inicijalizuje kada se prvi put pozove neka funkcionalnost

Upravljanje uređajem

- Upravljanje uređajem je omogućeno putem odgovarajućih funkcija
 - Dohvatanje ukupnog broja uređaja u sistemu `cudaGetDeviceCount()`
 - Dohvatanje karakteristika uređaja `cudaGetDeviceProperties()`
- Izbor uređaja:
 - Eksplicitno postavljanje aktivnog uređaja `cudaSetDevice()`
 - Izbor uređaja koji najbolje zadovoljava zadate uslove `cudaChooseDevice()`

Upravljanje memorijom (1)

- Dva tipa memorijskih objekata
 - Linearna memorija
 - Pristupa joj se pomoću 32-bitnih pokazivača
 - CUDA nizovi
 - Specifični, netransparentni objekti koji se koriste za smeštanje i čitanje tekstura
- Alokacija memorije na uređaju
 - Linearna memorija (1D)
 - `cudaMalloc()`, `cudaFree()`
 - Linearna memorija sa padding-om (2D, 3D)
`cudaMallocPitch()`, `cudaMalloc3D`
 - Parametar `pitch` govori o načinu poravnanja nizova u memoriji kako bi se zadovoljili uslovi sa kombinovani/sjedinjeni pristup
 - Alokacija CUDA nizova
`cudaMallocArray()`, `cudaFreeArray()`

Upravljanje memorijom (2)

- Alokacija page-locked memorije
 - Brži pristup kada se ne dozvoljava zamena stranica
`cudaHostAlloc()`, `cudaMallocHost()`, `cudaFreeHost()`
- Memorijski transferi sa domaćina na uređaj, uređaja na domaćina i unutar samog uređaja
 - Sinhroni i asinhroni transferi
 - Asinhronne funkcije nastavak `Async` u imenu
 - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToArray()`, `cudaMemcpyFromArray()`, `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
- Dohvatanje adrese simbola
`cudaGetSymbolAddress()`

Upravljanje teksturama i površima

- 2D prostorni keševi
 - Teksture se samo čitaju, po površima može i da se piše
- Vezuju se za posebne objekte
 - Ti objekti se zatim koriste u okviru jezgra
 - CUDA nizove (optimizovan pristup)
 - 1D linearnu memoriju (uz ograničenja)
 - Funkcije za rad sa teksturama:
`cudaBindTexture()`, `cudaUnbindTexture()`
- Teksturama se pristupa pomoću posebne hardverske jedinice i ugrađenih funkcija jezgra
 - `tex1D()`, `tex2D()`, `tex3D()`
`float u, v; // Coordinates`
`float4 value = tex2D(myTexRef, u, v);`

Saradnja sa drugim API

- CUDA može da saraduje sa OpenGL, DirectX (Direct3D) i Vulkan API
- Baferi iz spoljnih API-ja se mogu direktno mapirati u CUDA adresni prostor
 - Različite ručke, tipično zavisne od OS-a i API-ja
 - Podaci se mogu čitati i obrađivati
 - Podaci se mogu upisati i proslediti dalje na obradu
 - Može se vršiti sinhronizacija nad ovim objektima
 - Funkcije kao što su:
`cudaExternalMemoryGetMappedBuffer()` ;
`cudaExternalMemoryGetMappedMipmappedArray()` ;

Upravljanje greškama

- Sve funkcije koje su deo CUDA runtime vraćaju strukturu `cudaError_t` sa opisom greške
 - Postoji oko 25 različitih kodova grešaka
- Poslednji kod greške koji je proizveo neki od poziva se može dobiti sa:
 - `cudaGetLastError()`
 - Greške asinhronih poziva se dohvataju ovom funkcijom ili prilikom poziva neke druge CUDA funkcije
- String koji opisuje odgovarajuću grešku se može dobiti pozivom:
 - `cudaGetErrorString()`

Upravljanje događajima

- CUDA podržava koncept događaja
 - Koriste se za praćenje napretka asinhronih događaja
 - Događaj se beleži onda kada se izvrše sve komande zadate tokom komandi
 - Događaji se predstavljaju tipom `cudaEvent_t`, a koriste se funkcije:
`cudaEventCreate()`, `cudaEventRecord()`,
`cudaEventSynchronize()`, `cudaEventElapsedTime()`,
`cudaEventDestroy()`
- Vreme na uređaju se može meriti pomoću funkcija koje rade sa CUDA događajima
 - Koriste se precizni GPU tajmeri

Konkurentno izvršavanje

- Kako bi se omogućilo konkurentno izvršavanje i na domaćinu i na uređaju, određeni broj poziva je asinhron
 - Pozivi jezgru
 - Memorijski transferi unutar uređaja
 - Memorijski transferi označeni **Async** funkcijama
 - Po potrebi, sinhronizacija se može obaviti funkcijom: **cudaDeviceSynchronize()**
- Takođe, konkurentnost se može postići:
 - Preklapanjem memorijskih transfera i izvršavanja jezgra korišćenjem koncepta tokova (*streams*)
 - Paralelnim izvršavanjem jezgara na uređajima koji to dozvoljavaju (*cc* $\geq$ 2.0)

Dinamički paralelizam

- Grafički procesori sa $cc \geq 3.5$ omogućavaju da se u okviru jednog jezgra omoguće pozivi drugim jezgrima
 - Dinamički paralelizam
 - Mogućnost da jezgro kreira novi posao direktno na GPU
- Dinamički paralelizam omogućava da se smanji potreba za transferom kontrole toka između domaćina i uređaja
 - Izvršna konfiguracija novog poziva nekom jezgru može da se zada na GPU
 - Omogućena je sinhronizacija između jezgara – roditelja i potomaka
- Pogodno za probleme koji iskazuju:
 - Ugneždeni i hijerarhijski paralelizam
 - Potrebu za rekurzijom
 - Iregularnu strukturu petlji

CUDA biblioteke (1)

- CUBLAS
 - CUDA *Basic Linear Algebra Subprograms* (BLAS)
 - Implementacija BLAS standarda na CUDA
 - Kompatibilna sa FORTRAN aplikacijama
 - Uključuje se zaglavljem `cublas.h`
- CUFFT
 - CUDA *Fast Fourier Transform* (FFT)
 - Uključuje implementaciju najvažnijih i najkorišćenijih CUDA algoritama
 - Uključuje se zaglavljem `cufft.h`
- CURAND
 - Implementira generisanje slučajnih brojeva na uređaju
 - Uključuje se zaglavljem `curand.h` i `curand_kernel.h`

CUDA biblioteke (2)

- NVIDIA Performance Primitives (NPP)
 - Implementira veliki broj gotovih algoritama za obradu slike i video signala
- CUSPARSE
 - Implementira algoritme za rad sa retkim matricama
- Thrust biblioteka
 - Thrust je biblioteka CUDA šablona za C++ bazirana na *Standard Template Library* (STL).
 - Thrust dozvoljavam programeru da implementira HPC aplikacije sa minimalnim programerskim naporom
 - Implementiran je API visokog nivoa
 - Memorijski transferi su sakriveni od korisnika i sl.

Literatura

- David Kirk, Wen-mei Hwu, Programming Massively Parallel Processors: A Hands on Approach, Morgan Kaufmann
- NVIDIA CUDA C Programming Guide 13.1, 2026.
- NVIDIA GPU Teaching Kit 2025
- Razni materijali i dokumentacija sa NVIDIA sajta
- <http://en.wikipedia.org/wiki/GPGPU>
- <http://en.wikipedia.org/wiki/CUDA>