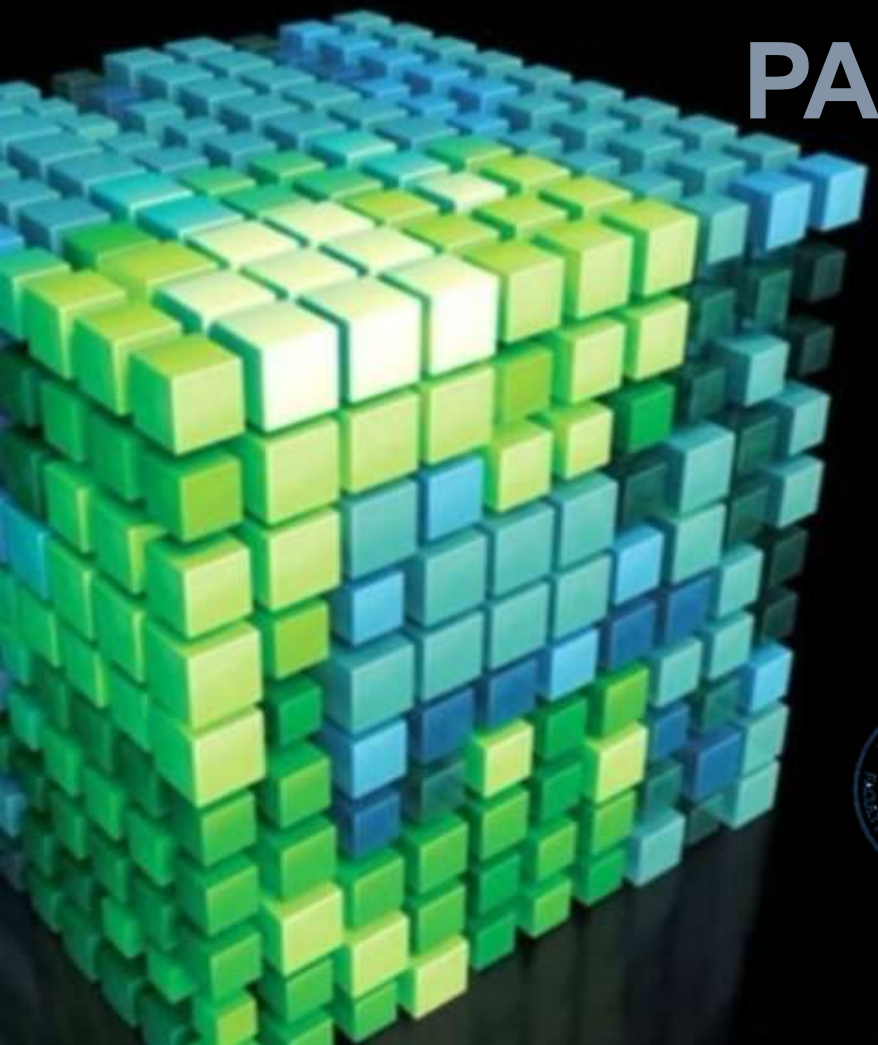
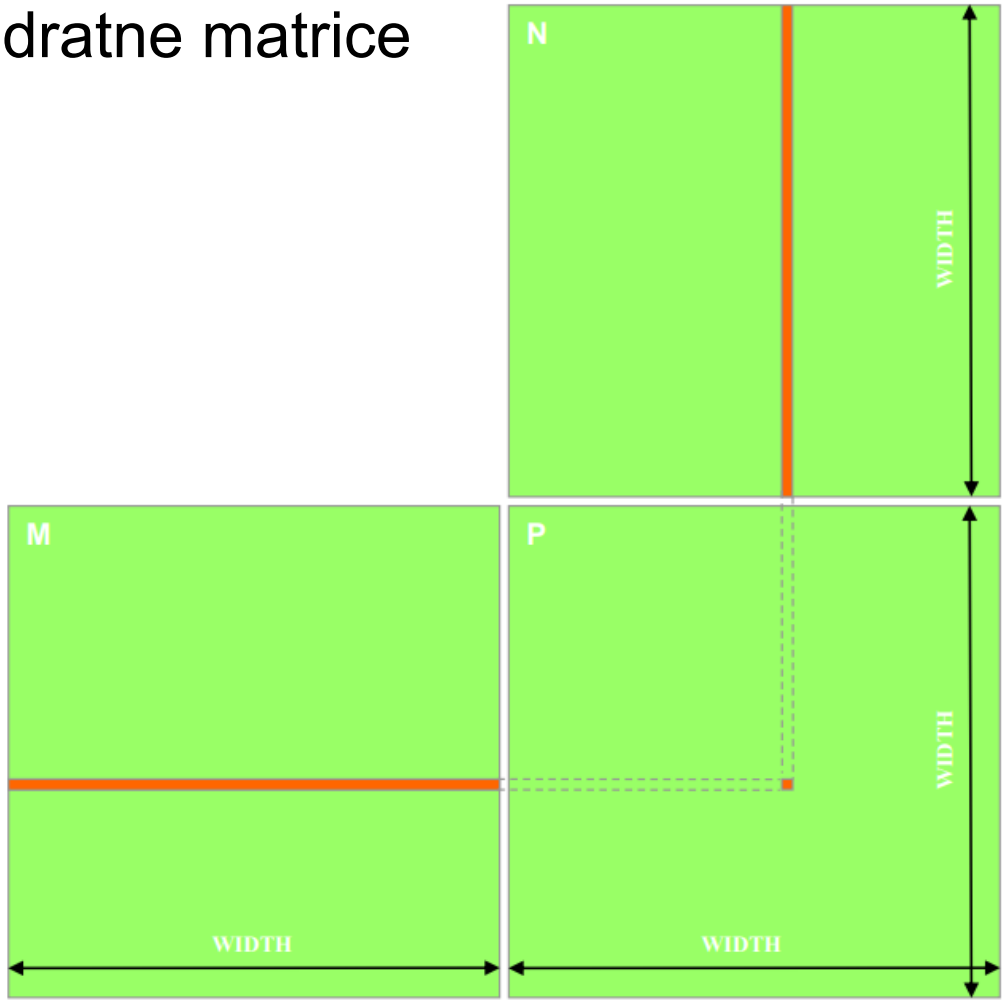


PARALELNI SISTEMI:
CUDA

MSc Aleksandra Stojnev
Prof. Dr. Natalija Stojanović

Primer: Množenje matrica

- Pomnožiti dve kvadratne matrice



Primer: Množenje matrica (2)

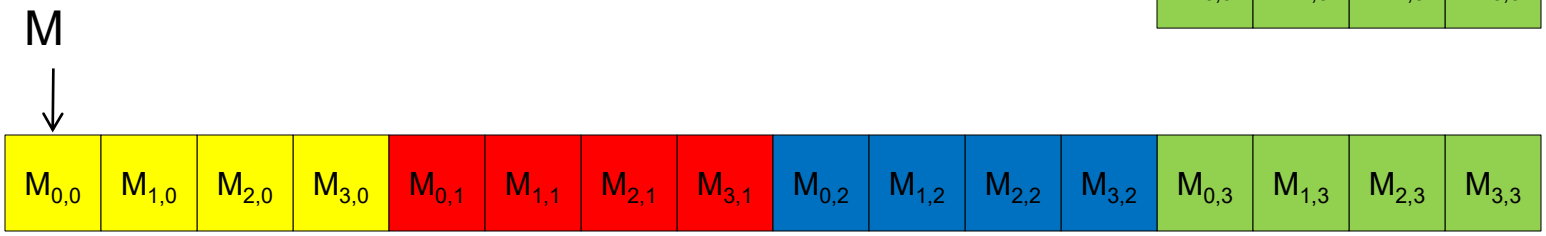
- Tradicionalni sekvencijalni kod:

```
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;
      }
}
```

Primer: Množenje matrica (3)

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

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$



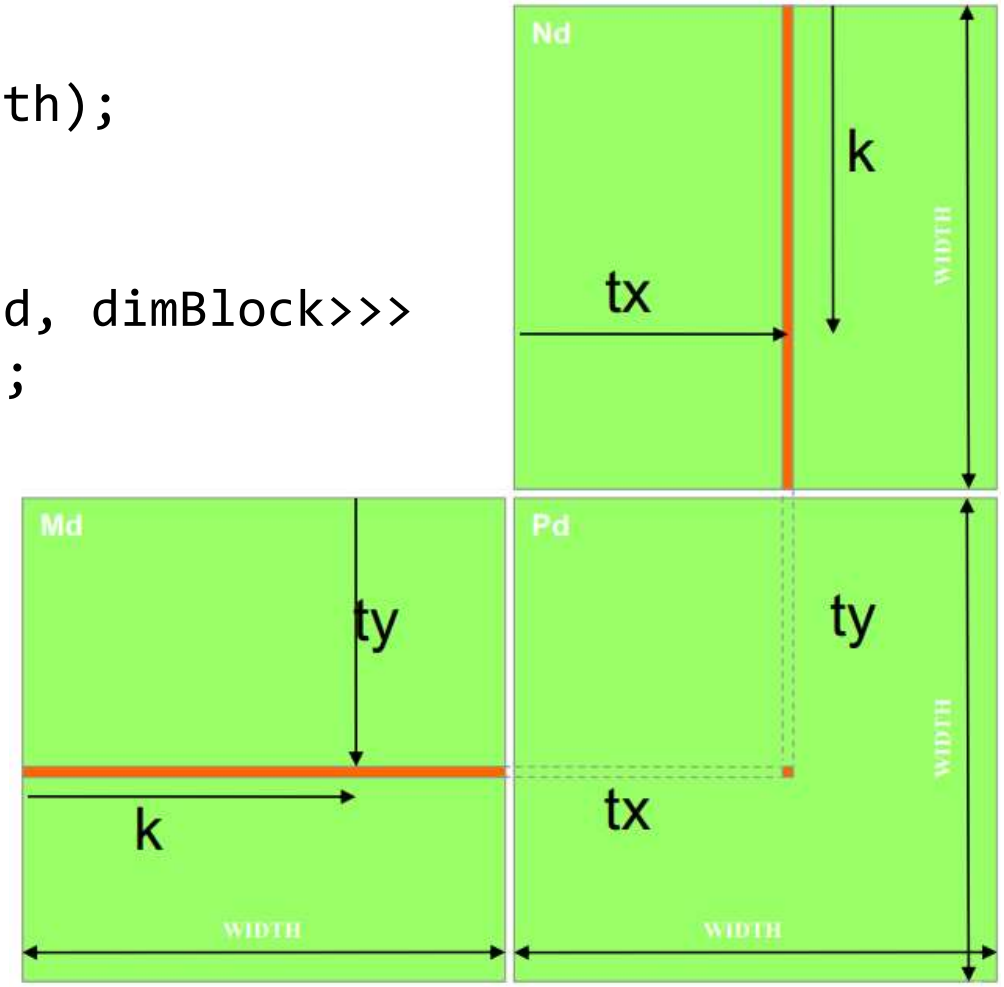
Primer: Množenje matrica (4)

HOST:

```
void MatrixMulOnDevice (float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md;
    float* Nd;
    float* Pd;

    // 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);

    ////////////////////////////////////////////////////
    // Kernel invocation code – to be shown later
    ////////////////////////////////////////////////////

    // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(Md);
    cudaFree(Nd);
    cudaFree (Pd);
  }
```

Primer: Množenje matrica (6)

```
// Konfiguracija kernela
  dim3 dimGrid(1, 1);
  dim3 dimBlock(Width, Width);

  // Pokretanje
  MatrixMulKernel<<<dimGrid, dimBlock>>>
        (Md, Nd, Pd, Width);
```



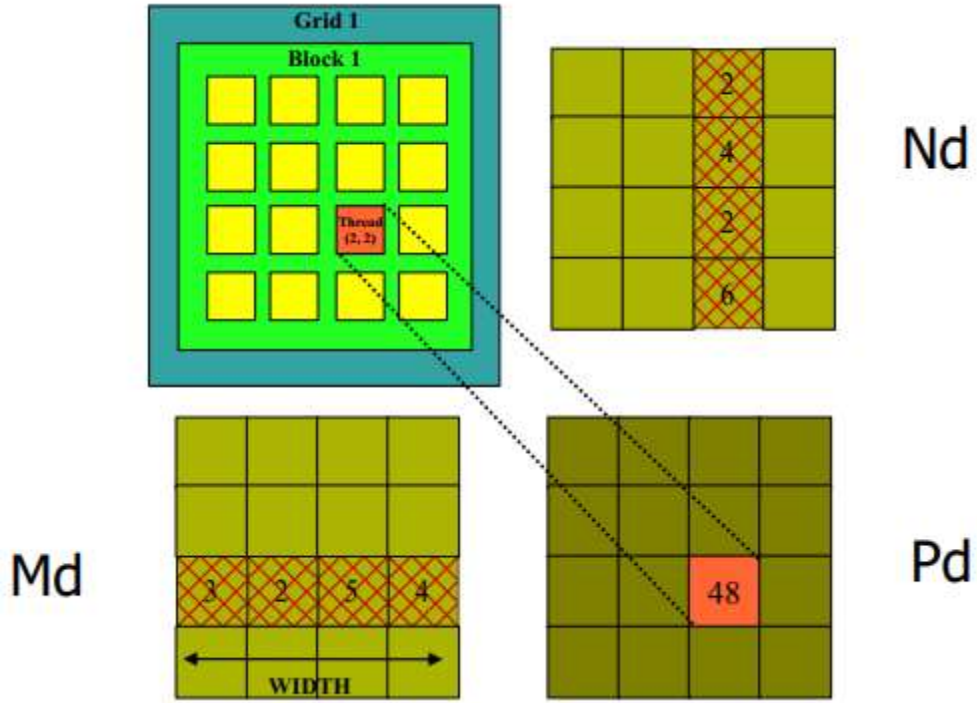
Primer: Množenje matrica (5)

KERNEL:

```
__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;
}
```

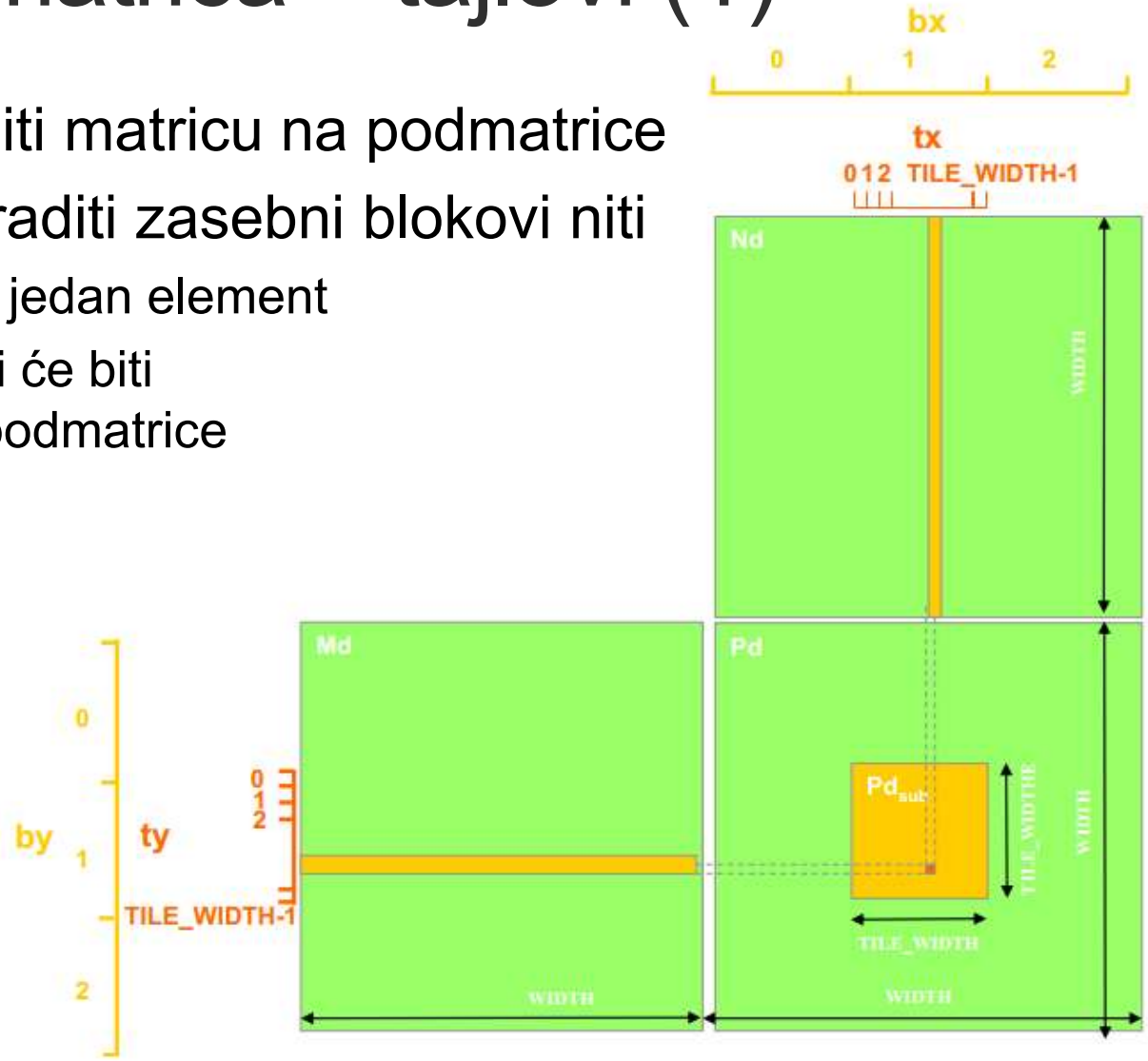
Primer: Množenje matrica - Nedostaci (1)

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



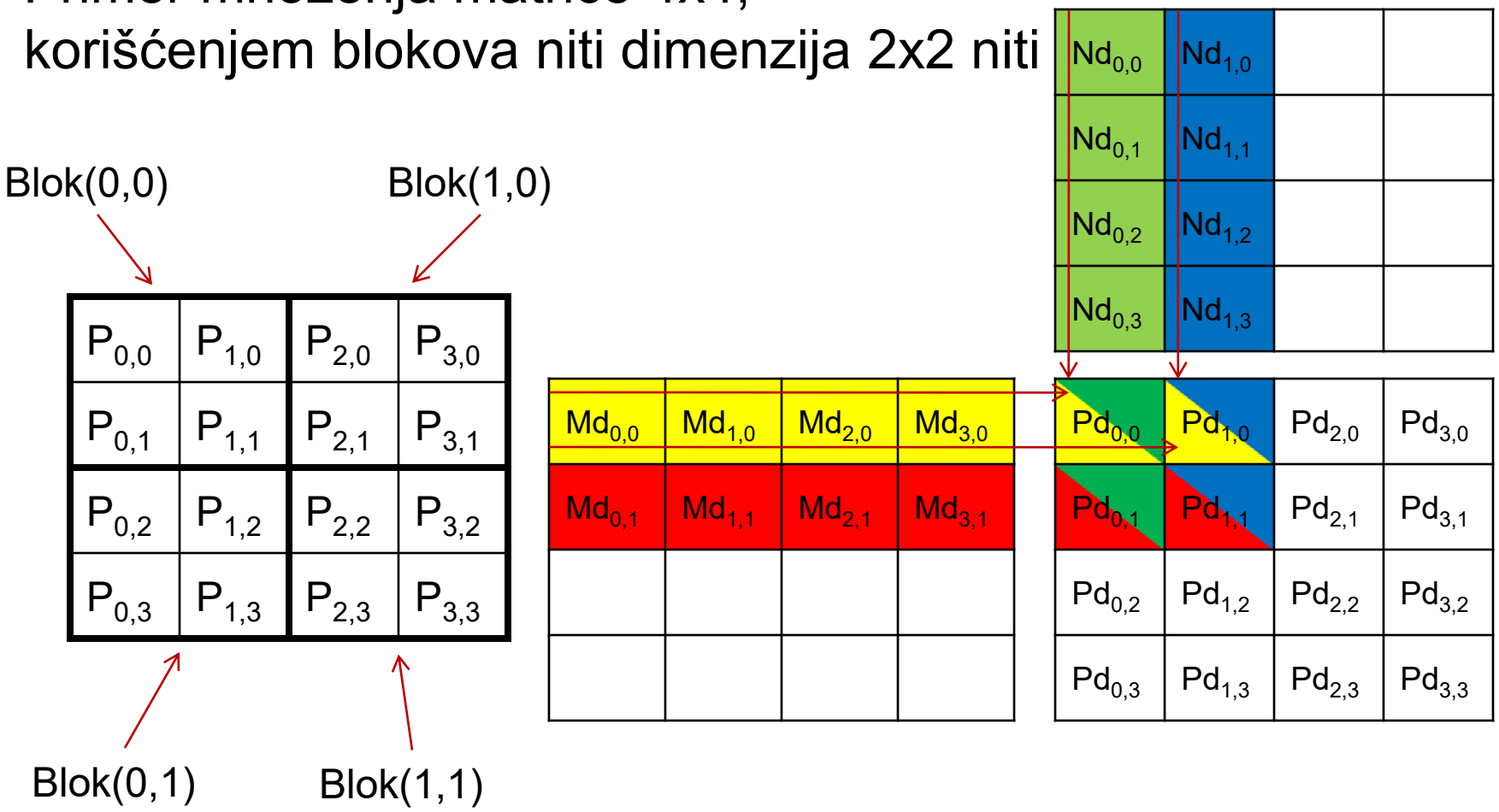
Množenje matrica – tajlovi (1)

- 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



Množenje matrica – tajlovi (2)

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



Množenje matrica – tajlovi (3)

KERNEL:

```
__global__ void MatrixMulKernel
     (float* Md, float* Nd, float* Pd, int Width)
{
// Računanje indeksa za vrstu za Pd i M
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
// Računanje indeksa za kolonu za Pd i N
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;
    float Pvalue = 0;

// Svaka nit računa jedan element za podmatricu bloka
for (int k = 0; k < Width; ++k)
      Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

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

Množenje matrica – tajlovi (4)

- Poziv kernela?

Množenje matrica – tajlovi (4)

- Poziv kernela?

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


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

Množenje matrica - deljiva memorija (1)

- Jednostavni CUDA kerneli koji obezbeđuju da niti pristupaju određenim delovima podataka, koji su prethodno prebačeni na uređaj.
- Latenca globalne memorije i ograničen bandwidth za pristup dovode do zagušenja
- Rešenje: dodatni metodi za pristup memoriji (različiti tipovi memorije)

Množenje matrica - deljiva memorija (2)

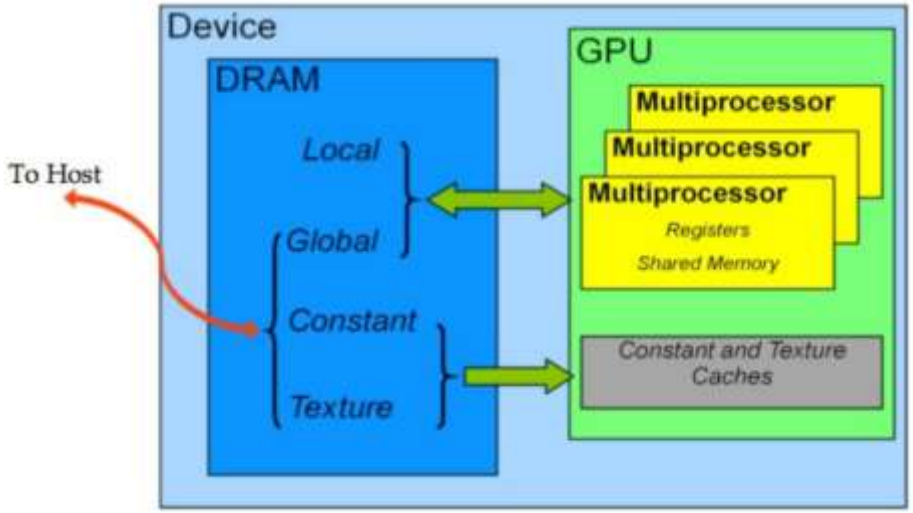
- Najbitniji deo kernela jeste for petlja koja se koristi da se računaju skalarni proizvodi.
  - U svakoj iteraciji ove petlje, imamo dva pristupa globalnoj memoriji, jedno FP množenje i jedno FP sabiranje.
  - Odnos FP operacija i pristupa globalnoj memoriji - 1:1.
    - **CGMA** odnos predstavlja broj FP operacija koji se izvršavaju za svaki pristup globalnoj memoriji u jednom delu (regionu) CUDA programa.

Množenje matrica - deljiva memorija (3)

- Sa CGMA odnosom koji je 1:1, kernel koji se koristi za množenje matrica će izvršavati ne više od 21.6 **biliona** FP operacija u sekundi (gigaflopsi) (kartica G80, global memory access bandwidth od 86.4GB/s)
- Svaka FP operacija zahteva jedan single-precision global memory datum.
- Maksimalni procenjeni broj operacija za istu karticu je 367gflops.

CUDA device memory types - podsetnik

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

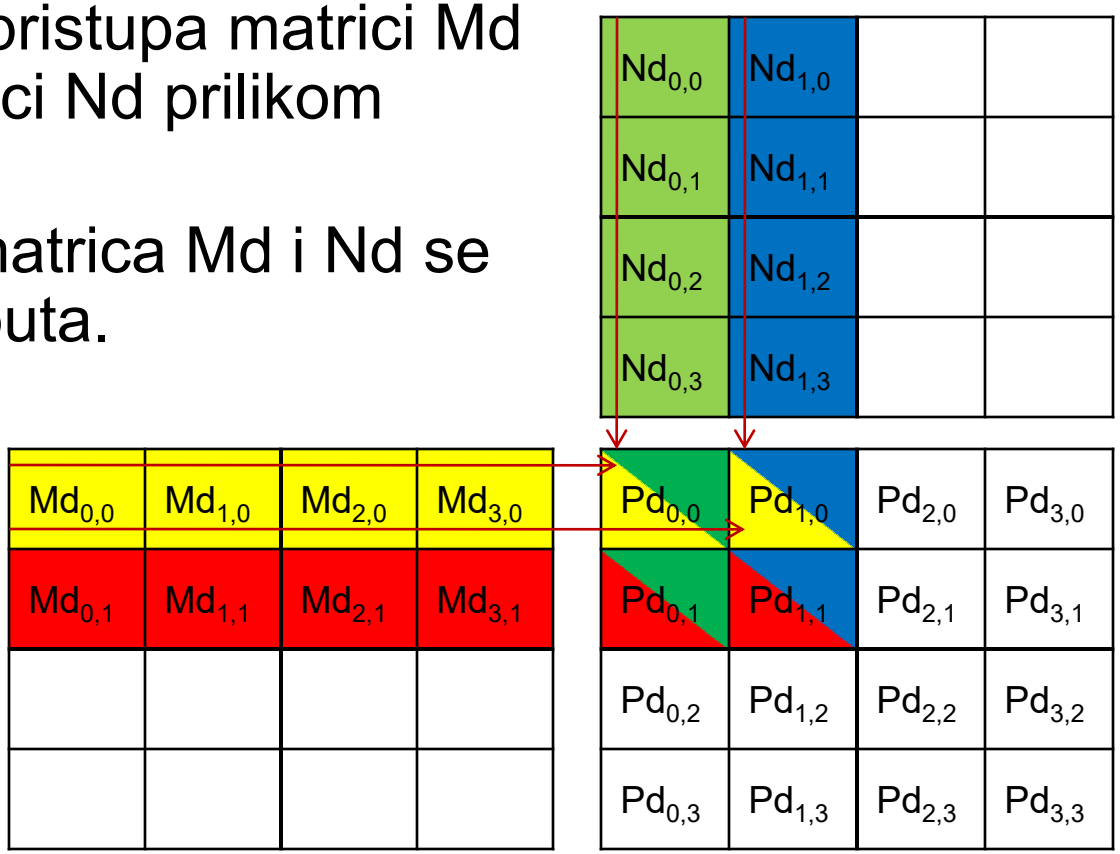


Strategija za redukciju broja pristupa globalnoj memoriji

- Particionisanje podataka u podskupove, ili tajlove (eng. *tiles*)
- Cilj je da svaki tajl može da se smesti u deljivu memoriju
- Globalna memorija (Global memory-GM) je velikog kapaciteta, ali spora, dok je deljiva memorija (Shared memory-SHM) mala i brza
- Problem: ne mogu sve strukture podataka da se podele u tajlove.

Množenje matrica - deljiva memorija (1)

- Svaka nit ima četiri pristupa matrici Md i četiri pristupa matrici Nd prilikom izvršenja.
- Svakom elementu matrica Md i Nd se pristupa tačno dva puta.

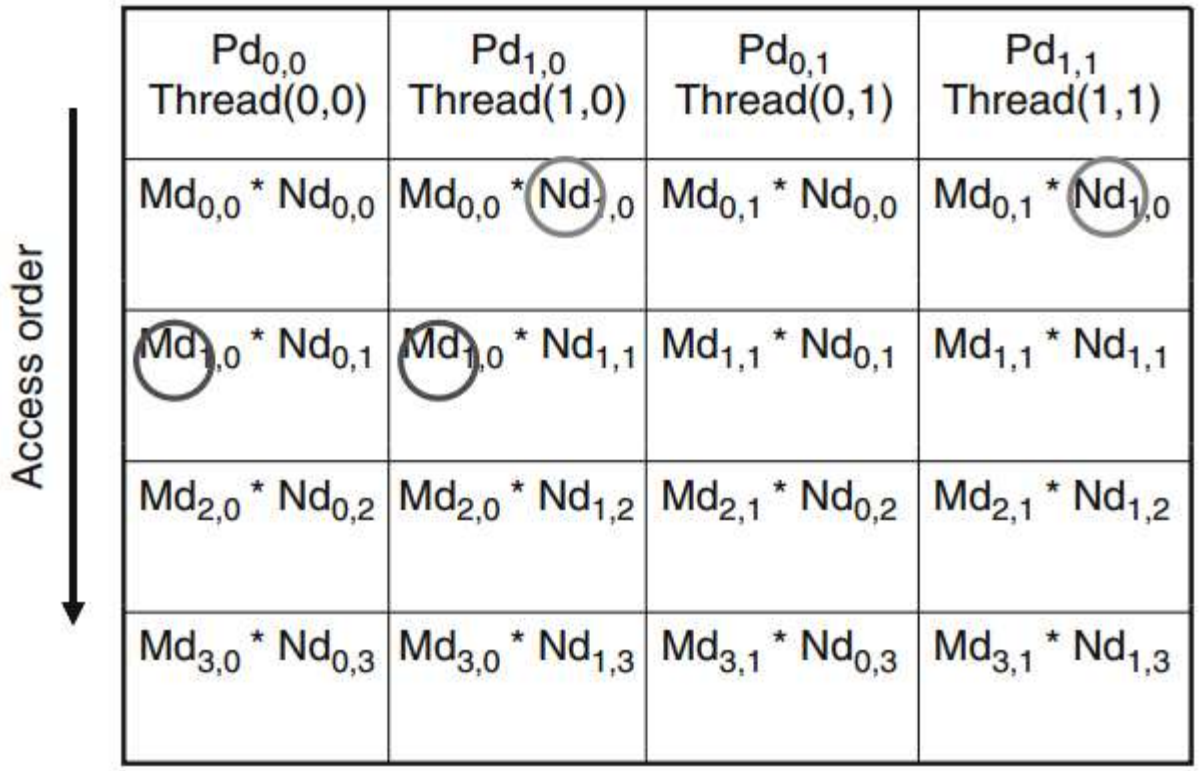


Množenje matrica - deljiva memorija (2)

- **Šta ako bi niti koje pristupaju istim elementima mogle međusobno da sarađuju?**
  - Elementi se mogu iz globalne memorije učitati samo jednom, što bi smanjilo ukupan broj pristupa globalnoj memoriji.
  - Svakom elementu matrica Md i Nd se pristupa tačno dva puta.
    - Potencijalno možemo smanjiti ukupan broj pristupa globalnoj memoriji
  - Redukcija je proporcionalna broju blokova: sa NxN blokova, potencijalna redukcija je N puta.

Množenje matrica - deljiva memorija (3)

PRISTUP GLOBALNOJ MEMORIJI – NITI IZ BLOKA (0,0)



Množenje matrica - deljiva memorija (4)

- **Kako niti mogu biti organizovane tako da se redukuje pristup memoriji?**
- Sve niti kolaborativno učitavaju elemente Md i Nd matrice u deljivu memoriju pre nego što ih koriste u svojim izračunavanjima (računanje skalarnog proizvoda).
  - Potencijalni problem: veličina deljive memorije
    - Rešenje: Md i Nd matrice se dele u tajlove. Veličina tajlova određuje se količinom dostupne deljive memorije.
- Svaki skalarni proizvod se deli u faze
  - U svakoj fazi, sve niti u bloku kolaborativno učitavaju jedan tajl matrice Md i jedan tajl matrice Nd u deljivu memoriju **(Mds, Nds)**. Nakon učitavanja, ove se vrednosti koriste za skalarni proizvod.
- Ako je ulazna matrica dimenzije N, i veličina tajlova je TILE_WIDTH, skalarni proizvod se obavlja u N/TILE_WIDTH faza.

Množenje matrica - deljiva memorija (5)

PRVA FAZA, SVI BLOKOVI

$B_{0,0}$

Md 0,0	Md 1,0	Md 2,0	Md 3,0
Md 0,1	Md 1,1	Md 2,1	Md 3,1
Md 0,2	Md 1,2	Md 2,2	Md 3,2
Md 0,3	Md 1,3	Md 2,3	Md 3,3

Nd 0,0	Nd 1,0	Nd 2,0	Nd 3,0
Nd 0,1	Nd 1,1	Nd 2,1	Nd 3,1
Nd 0,2	Nd 1,2	Nd 2,2	Nd 3,2
Nd 0,3	Nd 1,3	Nd 2,3	Nd 3,3

$B_{0,1}$

Md 0,0	Md 1,0	Md 2,0	Md 3,0
Md 0,1	Md 1,1	Md 2,1	Md 3,1
Md 0,2	Md 1,2	Md 2,2	Md 3,2
Md 0,3	Md 1,3	Md 2,3	Md 3,3

Nd 0,0	Nd 1,0	Nd 2,0	Nd 3,0
Nd 0,1	Nd 1,1	Nd 2,1	Nd 3,1
Nd 0,2	Nd 1,2	Nd 2,2	Nd 3,2
Nd 0,3	Nd 1,3	Nd 2,3	Nd 3,3

$B_{1,0}$

Md 0,0	Md 1,0	Md 2,0	Md 3,0
Md 0,1	Md 1,1	Md 2,1	Md 3,1
Md 0,2	Md 1,2	Md 2,2	Md 3,2
Md 0,3	Md 1,3	Md 2,3	Md 3,3

Nd 0,0	Nd 1,0	Nd 2,0	Nd 3,0
Nd 0,1	Nd 1,1	Nd 2,1	Nd 3,1
Nd 0,2	Nd 1,2	Nd 2,2	Nd 3,2
Nd 0,3	Nd 1,3	Nd 2,3	Nd 3,3

$B_{1,1}$

Md 0,0	Md 1,0	Md 2,0	Md 3,0
Md 0,1	Md 1,1	Md 2,1	Md 3,1
Md 0,2	Md 1,2	Md 2,2	Md 3,2
Md 0,3	Md 1,3	Md 2,3	Md 3,3

Nd 0,0	Nd 1,0	Nd 2,0	Nd 3,0
Nd 0,1	Nd 1,1	Nd 2,1	Nd 3,1
Nd 0,2	Nd 1,2	Nd 2,2	Nd 3,2
Nd 0,3	Nd 1,3	Nd 2,3	Nd 3,3

Množenje matrica - deljiva memorija (6)

DRUGA FAZA, SVI BLOKOVI

$B_{0,0}$

$B_{0,1}$

$B_{1,0}$

$B_{1,1}$

Množenje matrica - deljiva memorija (7)

- **BLOK (0,0)**

	Phase 1			Phase 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_{11}*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1\cdot1}$*$Nds_{1,1}$

time →

Množenje matrica - deljiva memorija (8)

- Mds čuva elemente Md koji su u SHM, Nds čuva elemente Nd koji su u SM
- Na početku faze 1, 4 niti bloka B_{00} učitavaju elemente u Mds
- T_{00}: Mds_{00} <- Md_{00}
- T_{10}: Mds_{10} <- Md_{10}
- T_{01}: Mds_{01} <- Md_{01}
- T_{11}: Mds_{11} <- Md_{11}
- Na sličan način se učitavaju elementi matrice Nd u Nds
- Zatim se izračunavaju odgovarajući skalarni proizvodi

Množenje matrica - deljiva memorija (9)

- Svaka vrednost u SHM se koristi 2 puta
  - Npr. i T01 i T11 koriste vrednost Mds11
- Na taj način smanjuje se pristup memoriji dva puta
- Izračunavanje svakog vektorskog proizvoda se za ovaj primer odvija u 2 faze.
- U opštem slučaju, za matrice reda N i veličinu tajla TILE_WIDTH, potrebno je **N/TILE_WIDTH** faza
- U svakoj fazi, **isti** Mds i Nds se koriste da čuvaju delove Md i Nd matrica korišćenih u toj fazi
- Ovo omogućava da mala SHM služi za većinu pristupa za koje je korišćena GM

Množenje matrica - deljiva memorija (10)

- To je zato što svaka faza fokusirana na mali podskup elemenata ulazne matrice, što predstavlja lokalnost pri pristupu
- Lokalnost pristupa omogućava korišćenje malih brzih memorija koje opslužuju većinu pristupa i smanjuju pristup sporoj GM
- Lokalnost je važna za postizanje visokih performansi

Množenje matrica - deljiva memorija (11)

POZIV KERNELA

```
// 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 - deljiva memorija (12)

KERNEL (1)

```
__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 - deljiva memorija (13)

KERNEL (2)

```
// Loop over the Md and Nd tiles required to compute
//the Pd element
 for (int m = 0; m < Width/TILE_WIDTH; ++m) {
  // Coolaborative 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 - deljiva memorija (14)

- *Mds* i *Nds* su promenljive u SHM i njihova oblast važenja su niti u okviru bloka
- Promenljive *bx*, *by*, *tx* i *ty* su automatske skalarne promenljive i smeštene su u registrima.
  - Njihova oblast važenja je nit i postoji privatna kopija za svaku nit
- Promenljive *Row* i *Col* su indeksi vrste i kolone rezultujućeg Pd elementa
- *Pvalue* je promenljiva u kojoj se čuvaju privremene vrednosti rezultujućeg elementa PD

Množenje matrica - deljiva memorija (15)

- U svakoj fazi izračunavanja, svaka nit u nekom bloku učitava po jedan element matrice Md u SHM
- $TILE\_WIDTH^2$ niti sarađuju u učitavanju $TILE\_WIDTH^2$ različitih vrednosti matrice Md u SM
- Slično važi i za elemente matrice Nd
- Da bi se obezbedilo da sve niti u okviru istog bloka matrice kompletiraju učitavanje vrednosti u Mds i Nds pre izvođenja izračunavanja dela skalarnog proizvoda koristi se poziv ***syncthreads()***

Množenje matrica - deljiva memorija (20)

- Nakon toga, moguće je otpočeti izračunavanje gde svaka nit u K petlji izvodi izračunavanje jednog dela svog skalarnog proizvoda
- Nakon izvršenja K petlje, neophodan je još jedan poziv ***syncthreads()*** funkcije koja sad obezbeđuje da su sve niti u okviru nekog bloka završile sa korišćenjem sadržaja Mds i Nds pre nego što započne sledeća faza i učitaju se novi elementi u Mds i Nds

Množenje matrica-performanse

- Prilikom opisanog množenja matrica, pristip GM je smanjen TILE_WIDTH puta
  - Ako se koriste 16x16 blokovi, pristup se smanjuje 16 puta
- Prednosti korišćenja deljene memorije:
  - Svaki blok niti treba da ima veliki broj niti
  - TILE_WIDTH ne bi trebalo da bude manje od 16
    - 16 x 16 = 256 niti po bloku
  - Treba da bude veliki broj blokova niti
    - Rezultujuća matrica veličine 1024 x 1024 daje 64 x 64 = 4096 blokova niti

Množenje matrica-performanse

- U takvom slučaju, svaki blok niti ima 2 x 256 = 512 pristupa (čitanja) iz globalne memorije
- Zatim se vrši 256 x (2 x 16) = 8192 operacija množenja i sabiranja
- U takvom slučaju memorijski propusni opseg više nije limitirajući faktor
- **Broj računskih operacija mnogo veći od broja pristupa memoriji**
  - 8192 >> 512

Atomične operacije

Atomične operacije

- x++

- Čitanje vrednosti x
- Inkrementiranje pročitane vrednosti
- Upis rezultata u x

read-modify-write operacija

Atomične operacije – dve niti

X = 7, Niti A i B inkremetiraju X

Korak	Primer
1. Nit A čita x	A čita vrednost 7
2. Nit A inkrementira pročitanu vrednost	A računa 8
3. A upisuje rezultat u x	x ← 8
4. Nit B čita x	B čita vrednost 8
5. Nit B inkrementira pročitanu vrednost	B računa 9
6. B upisuje rezultat u x	x ← 9

Korak	Primer
1. Nit A čita x	A čita vrednost 7
2. Nit B čita x	B čita vrednost 7
2. Nit A inkrementira pročitanu vrednost	A računa 8
5. Nit B inkrementira pročitanu vrednost	B računa 8
3. A upisuje rezultat u x	x ← 8
6. B upisuje rezultat u x	x ← 8

Compute capability

- Sve operacije koje smo dosad naveli su podržane od strane svake CUDA-capable grafičke kartice
- Kao što postoje različiti setovi instrukcija za CPU (MMX, SSE, SSE2), postoje različiti setovi instrukcija i za GPU
- NVIDIA funkcionalnosti i mogućnosti grafičkih kartica obeležava sa ***compute capability***
- Na primer, atomične operacije su podržane tek od *compute capability* 1.1 pa naviše

Compute capability - kompajliranje

- Da bi se kompajlirao CUDA kod, potrebno je naznačiti koji je minimalni compute capability.
- Navođenjem istog, kompajleru se daje mogućnost i za dodatne optimizacije.

```
nvcc –arch=sm_11
```

Race conditions

- Hazard koji se javlja kada rezultat izvršenja zavisi od trenutka u kom se dešavaju događaji nad kojima nemamo kontrolu
  - Redosled izvršenja niti
- Problemi nastaju kada se naruši SIMD paradigma
- Atomične operacije su *brute force* način da se ovi problemi zaobiđu
  - Atomics, locks, and mutex

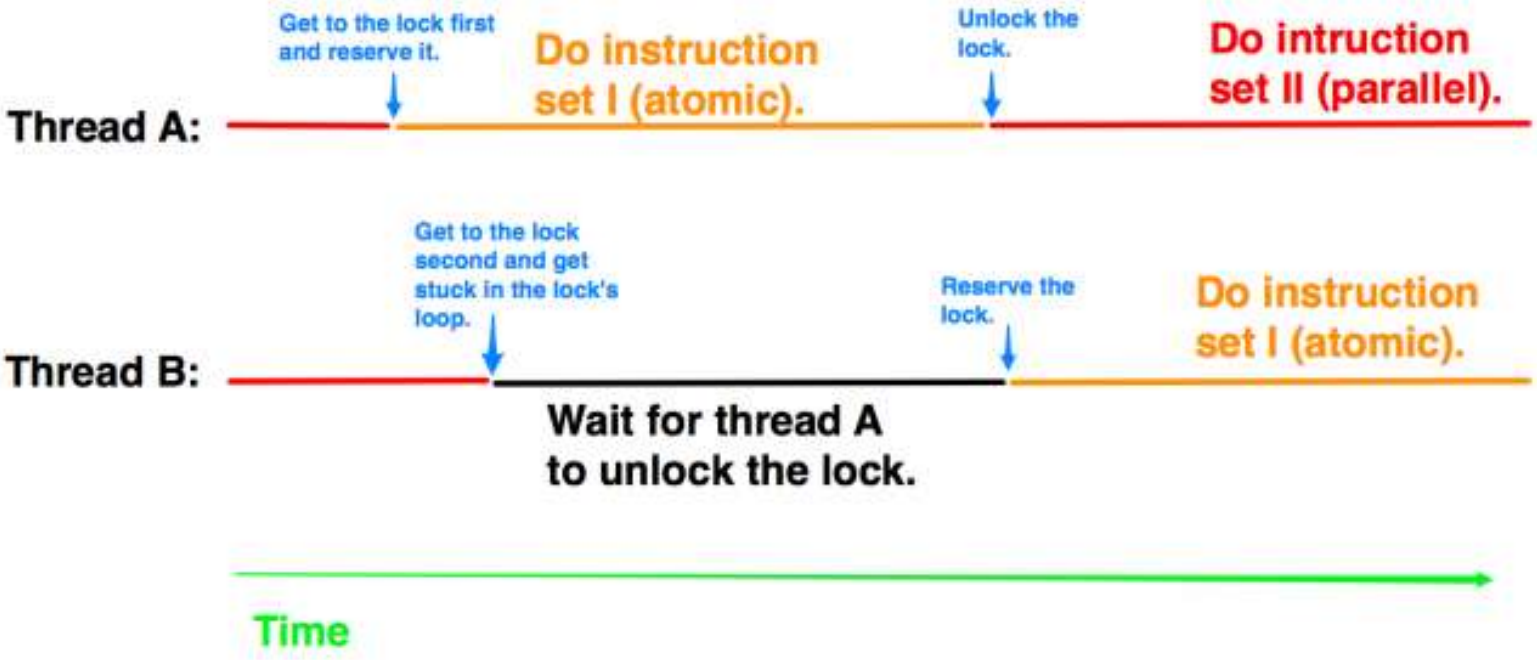
Atomici

- atomicAdd()
- atomicSub()
- atomicMin()
- atomicMax()
- atomicInc()
- atomicDec()
- atomicExch()
- atomicCAS()
- atomicAnd()
- atomicOr()
- atomicXor()

Locks and mutex

- Lock:
  - Mehanizam u paralelnom programiranju kojim se obezbeđuje da ceo segment koda bude izvršen atomično
- Mutex
  - - “mutual exclusion”, princip koji stoji iza lock-ova.
  - Dok jedna nit izvršava deo koda unutar lock-a, mutex se drži zaklučanim tako da nijedna druga nit ne može da ga otključa

Locks and mutex



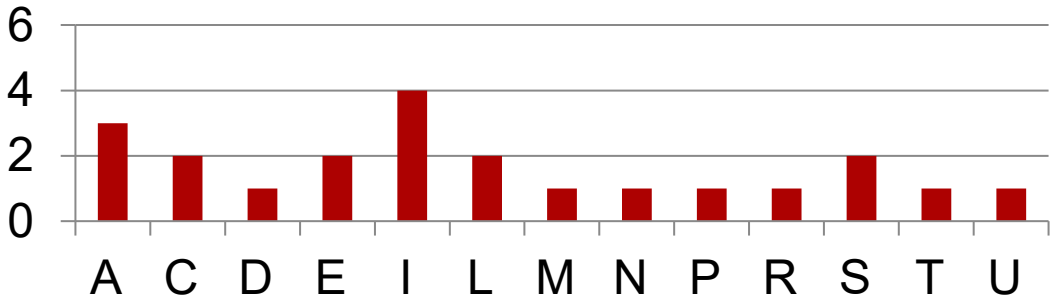
Primer: Računanje histograma

- Jedan od čestih zadataka jeste računanje histograma za dati set podataka
- Histogramom se predstavlja broj pojavljivanja svakog elementa u datom skupu
- Primer:

  Skup podataka: ***Paralelni sistemi i CUDA C***

  Histogram:

3	2	1	2	4	2	1	1	1	1	2	1	1
A	C	D	E	I	L	M	N	P	R	S	T	U



Računanje histograma – CPU (1)

```
#define SIZE (100*1024*1024)

int main(void)
{
    //kreiranje random niza
    unsigned char *buffer = (unsigned char*)
                                    big_random_block(SIZE);
    //kreiranje histograma i inicijalizacija na 0
    unsigned int histo[256];
    for(int i=0; i<256; i++)
        histo[i] = 0;

...
```

Računanje histograma – CPU (2)

```
...
// ideja je da kad god vidimo neku vrednost z u
// nizu, inkrementiramo vrednost bina z u histogramu. Na taj
// način brojimo broj ponavljanja elementa z u nizu
    for (int i=0; i<SIZE; i++)
        histo[buffer[i]]++;
// provera tačnosti
    long histoCount = 0;
    for (int i=0; i<256; i++)
    {
        histoCount += histo[i];
    }

    printf("Histogram Sum: %ld\n", histoCount);

    free(buffer);
     return 0;
 }
```

Računanje histograma – GPU (1)

```
#define SIZE (100*1024*1024)

int main(void)
{
    //kreiranje random niza
    unsigned char *buffer = (unsigned char*)
                        big_random_block(SIZE);

// alociranje memorije na GPU i prenos podataka
   unsigned char *dev_buffer;
   unsigned int *dev_histo;
   HANDLE_ERROR(cudaMalloc((void**)&dev_buffer, SIZE));
   HANDLE_ERROR(cudaMemcpy(dev_buffer,buffer,SIZE,cudaMemcpyHostToDevice));
   HANDLE_ERROR(cudaMalloc( (void**)&dev_histo, 256 * sizeof(long)));

   HANDLE_ERROR(cudaMemset(dev_histo, 0, 256 * sizeof(int)));
…
```

Računanje histograma – GPU (2)

```
…
  cudaDeviceProp prop;
  HANDLE_ERROR( cudaGetDeviceProperties( &prop, 0 ) );
  int blocks = prop.multiProcessorCount;
  histo_kernel<<<blocks*2,256>>>( dev_buffer, SIZE, dev_histo );

  unsigned int histo[256];
  HANDLE_ERROR( cudaMemcpy(histo, dev_histo, 256 *
  sizeof(int),cudaMemcpyDeviceToHost));

  long histoCount = 0;
  for (int i=0; i<256; i++)
          histoCount += histo[i];
  printf( "Histogram Sum: %ld\n", histoCount );
  //TODO> provera tačnosti
  cudaFree(dev_histo);
  cudaFree(dev_buffer);
  free(buffer);
  return 0;
}
```

Računanje histograma – GPU (2)

GLOBALNA MEMORIJA

```
#define SIZE (100*1024*1024)
__global__ void histo_kernel(unsigned char *buffer,
                             long size,
                             unsigned int *histo)
{
        int i = threadIdx.x + blockIdx.x * blockDim.x;
        int stride = blockDim.x * gridDim.x;

        while (i < size)
        {
                atomicAdd( &(histo[buffer[i]]), 1 );
                i += stride;
        }
}
```

Računanje histograma – GPU (2)

GLOBALNA I DELJIVA MEMORIJA

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size,
                             unsigned int *histo)
{
        __shared__ unsigned int temp[256];
        temp[threadIdx.x] = 0;
        __syncthreads(); // inicijalizacija na 0

        int i = threadIdx.x + blockIdx.x * blockDim.x;
        int offset = blockDim.x * gridDim.x;

        while (i < size)
        {
                atomicAdd( &temp[buffer[i]], 1);
                i += offset;
        }

        __syncthreads();
        atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
}
```