

Multiprocesorski sistemi

Paralelni algoritmi na CUDA

Matija Dodović, Marko Mišić

13S114MUPS, 13E114MUPS

2025/2026.

Paralelni algoritmi na grafičkom procesoru (1)

- Grafički procesor je *data-parallel* orijentisan
 - Vise hiljada niti se izvršava u paraleli
 - Vise hiljada elemenata se obrađuje
 - Sve podatke obrađuje isti program
 - SPMD model izvršavanja
 - Kontrast u odnosu na *task-parallel* model i paralelizam na nivou instrukcije (ILP)
- Način razmišljanja mora biti drugačiji i orijentisan ka podacima:
 - Algoritmi se moraju dizajnirati za paralelizam na nivou podataka
 - Potrebno je koristiti *data-parallel* primitive kao gradivne elemente za efikasno programiranje
 - Potrebno je razumeti složenost paralelnih algoritama

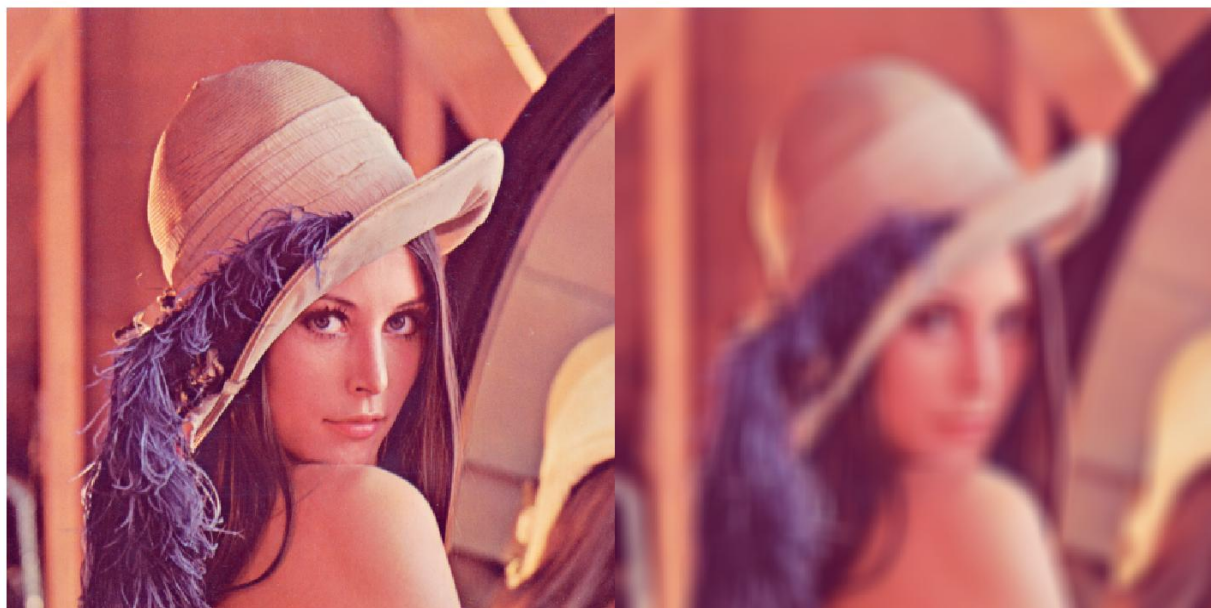
Paralelni algoritmi na grafičkom procesoru (2)

- Ključni koraci u paralelnom programiranju:
 - Pronaći konkurentnost u problemu
 - Strukturirati algoritam
tako da se konkurentnost prevede u performanse
 - Implementirati algoritam u pogodnom programskom okruženju
 - Izvršiti program i podesiti performanse koda
na stvarnom paralelnom sistemu
- Na žalost, ovi koraci nisu podeljeni u nivoe apstrakcije kojima bi mogli da se bavimo nezavisno jednim od drugih

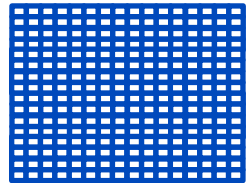
Problemi paralelnih algoritama

- Performanse mogu biti drastično umanjeno zbog velikog broja faktora:
 - Režijskog vremena potrebnog za sprovođenje paralelnog procesiranja
 - Disbalansa u opterećenju među procesnim elementima
 - Neefikasnim obrascima deljenja podataka
 - Zasićenja kritičnih resursa
 - Kao što je memorijski propusni opseg
- Pronalaženje i iskorišćavanje paralelizma često zahteva razmatranje problema iz ugla koji nije očigledan na prvi pogled
 - *Computational thinking*

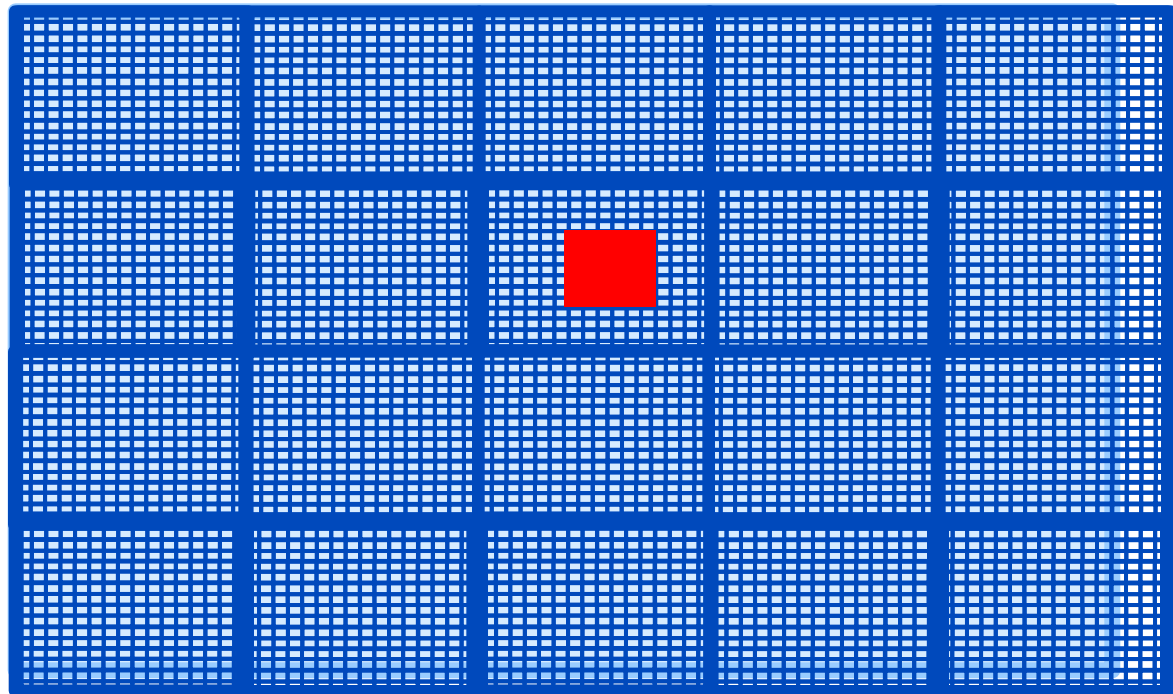
Blurovanje slike



Blurovanje slike



Box:
pikseli
koje
obrađuje
jedan blok
niti



Blurovanje slike

- 2D kernel

```
__global__ void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        ... // obrata jednog box-a
    }
}
```

Blurovanje slike

- Kod za jedan box (**BLUR_SIZE** – konstanta)

```
int pixVal = 0;
```

```
int pixels = 0;
```

```
// AVG susednih 2xBLUR_SIZE x 2xBLUR_SIZE piksela
```

```
for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {  
    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {
```

```
        int curRow = Row + blurRow;
```

```
        int curCol = Col + blurCol;
```

```
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
```

```
            pixVal += in[curRow * w + curCol];
```

```
            pixels++; // Akumulacija
```

```
        }
```

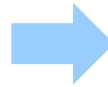
```
    }
```

```
}
```

```
// out je usrednjeni piskel
```

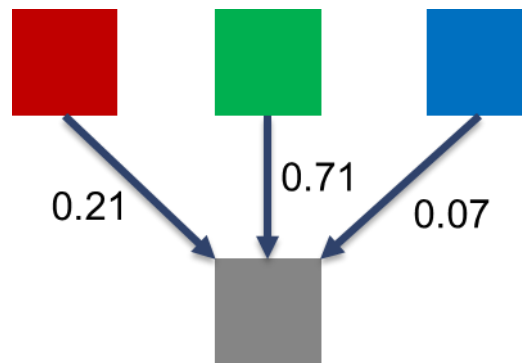
```
out[Row * w + Col] = (unsigned char)(pixVal / pixels);
```


RGB u Grayscale konverzija



RGB u Grayscale konverzija

- $\text{grayPixel}[I,J] = 0.21*r + 0.71*g + 0.07*b$



RGB u Grayscale konverzija

```
#define CHANNELS 3 // imamo 3 kanala - po jedan za svaku boju
// Slika je kodovana tako da je svaki piksel u 3 boje intenziteta [0, 255]
__global__ void colorConvert(unsigned char * grayImage, unsigned char * rgbImage, int width, int height) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

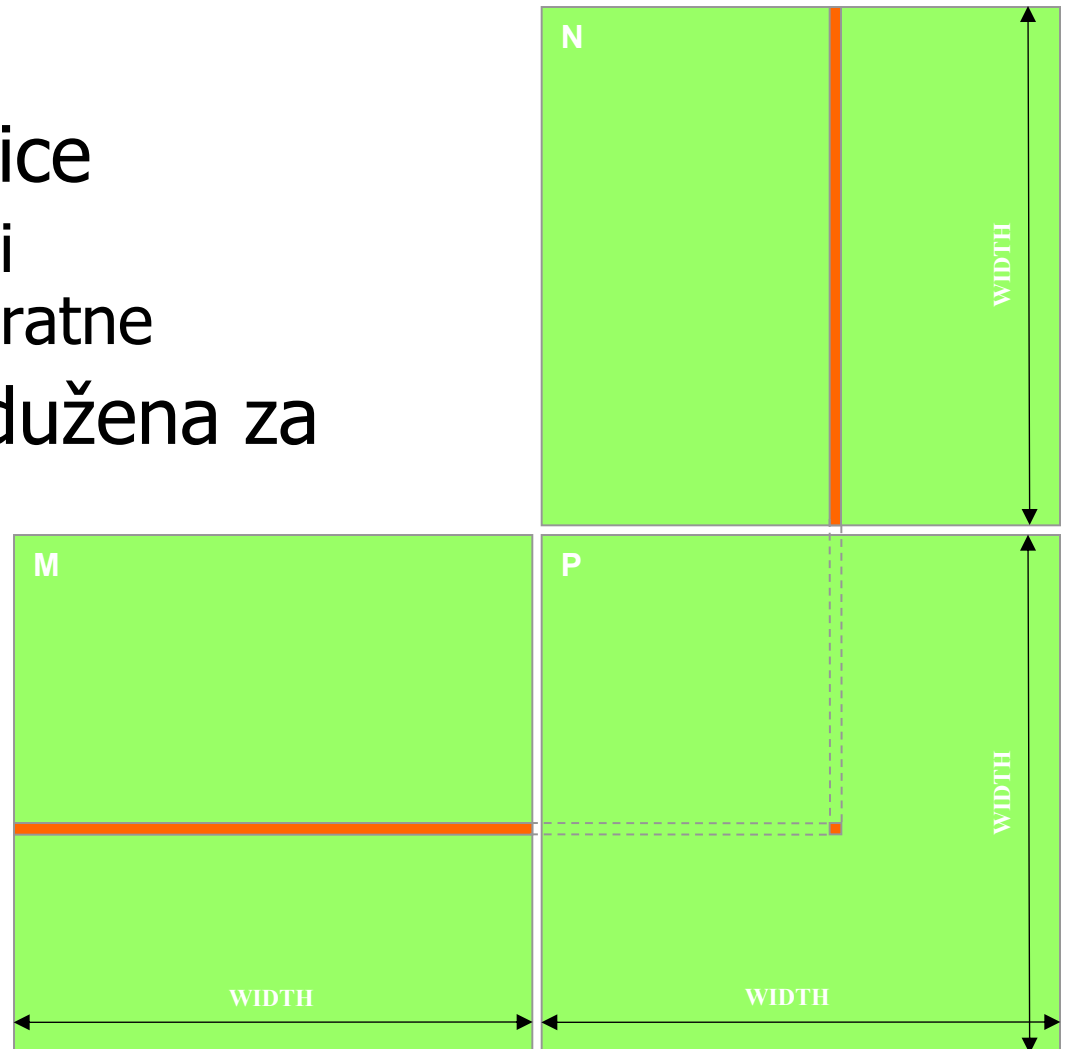
    if (x < width && y < height) {
        int grayOffset = y*width + x; // 1D koordinata za grayscale sliku

        // uzeti da RGB slika ima CHANNEL puta kolona nego grayscale slika
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = rgbImage[rgbOffset]; // red vrednost piksela
        unsigned char g = rgbImage[rgbOffset + 1]; // green vrednost piksela
        unsigned char b = rgbImage[rgbOffset + 2]; // blue vrednost piksela

        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

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



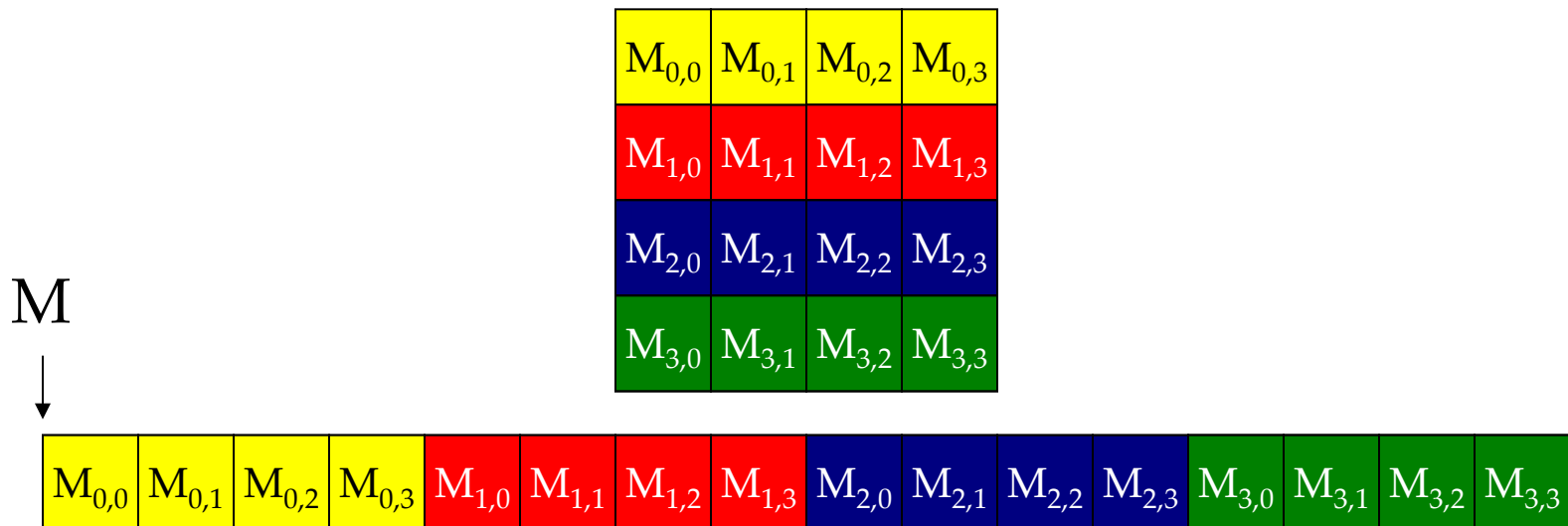
Studija slučaja – množenje matrica (2)

- 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:

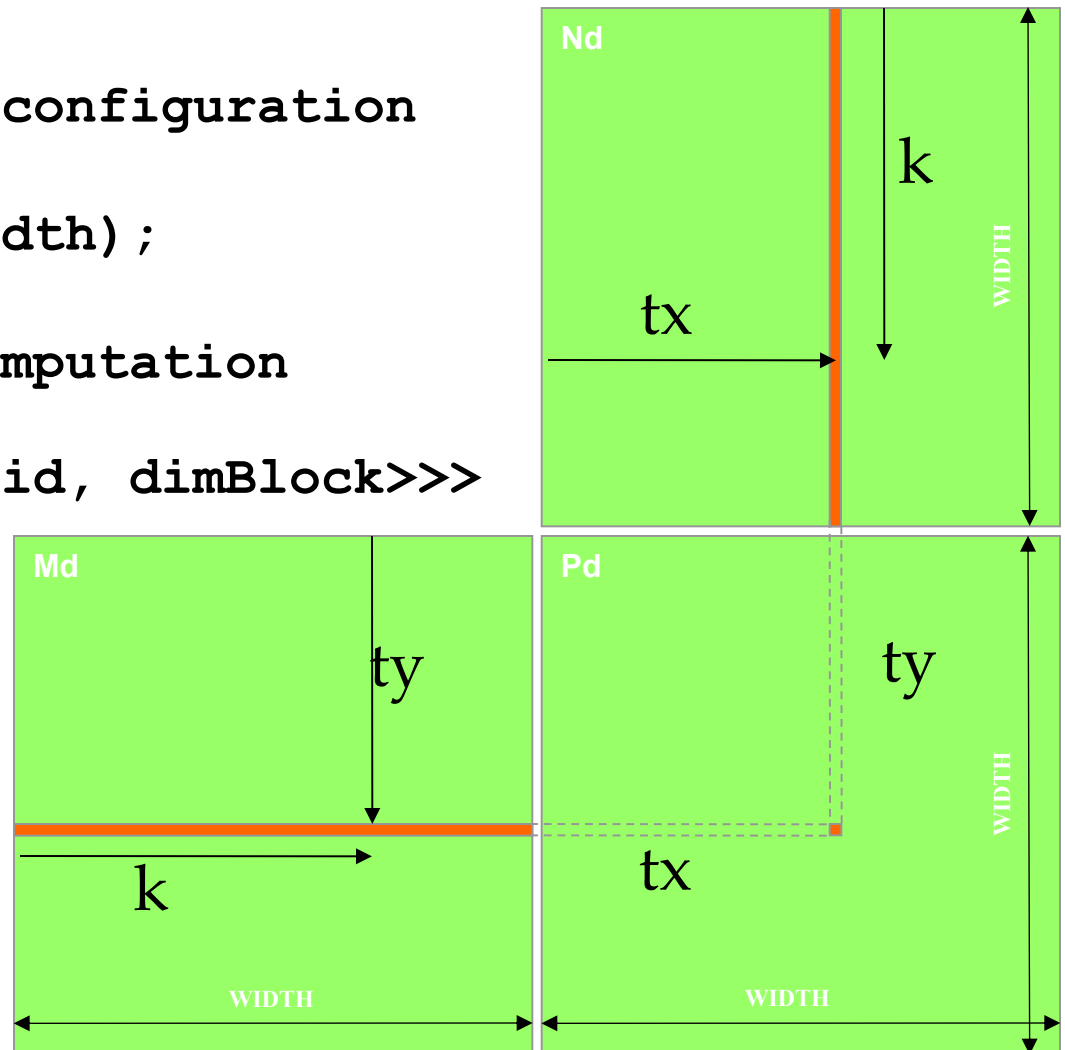
```
// Setup the execution configuration
```

```
dim3 dimGrid(1, 1);
```

```
dim3 dimBlock(Width, Width);
```

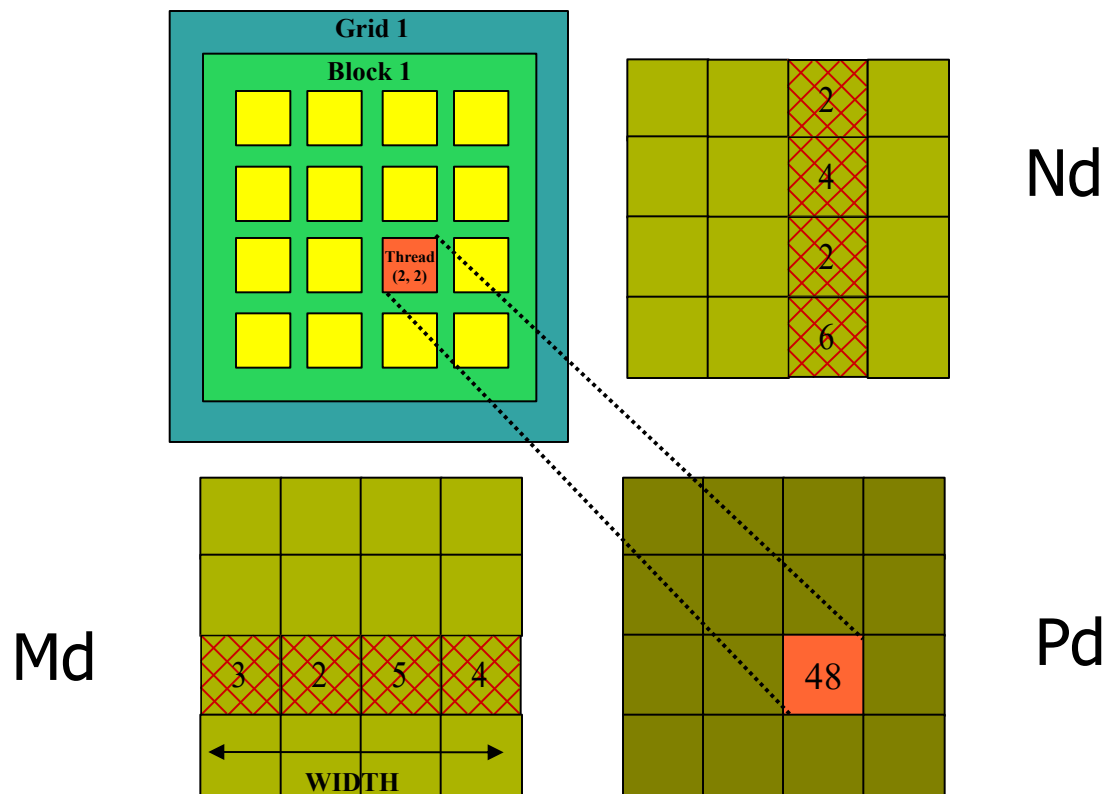
```
// Launch the device computation  
threads!
```

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



Nedostaci predloženog rešenja (1)

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

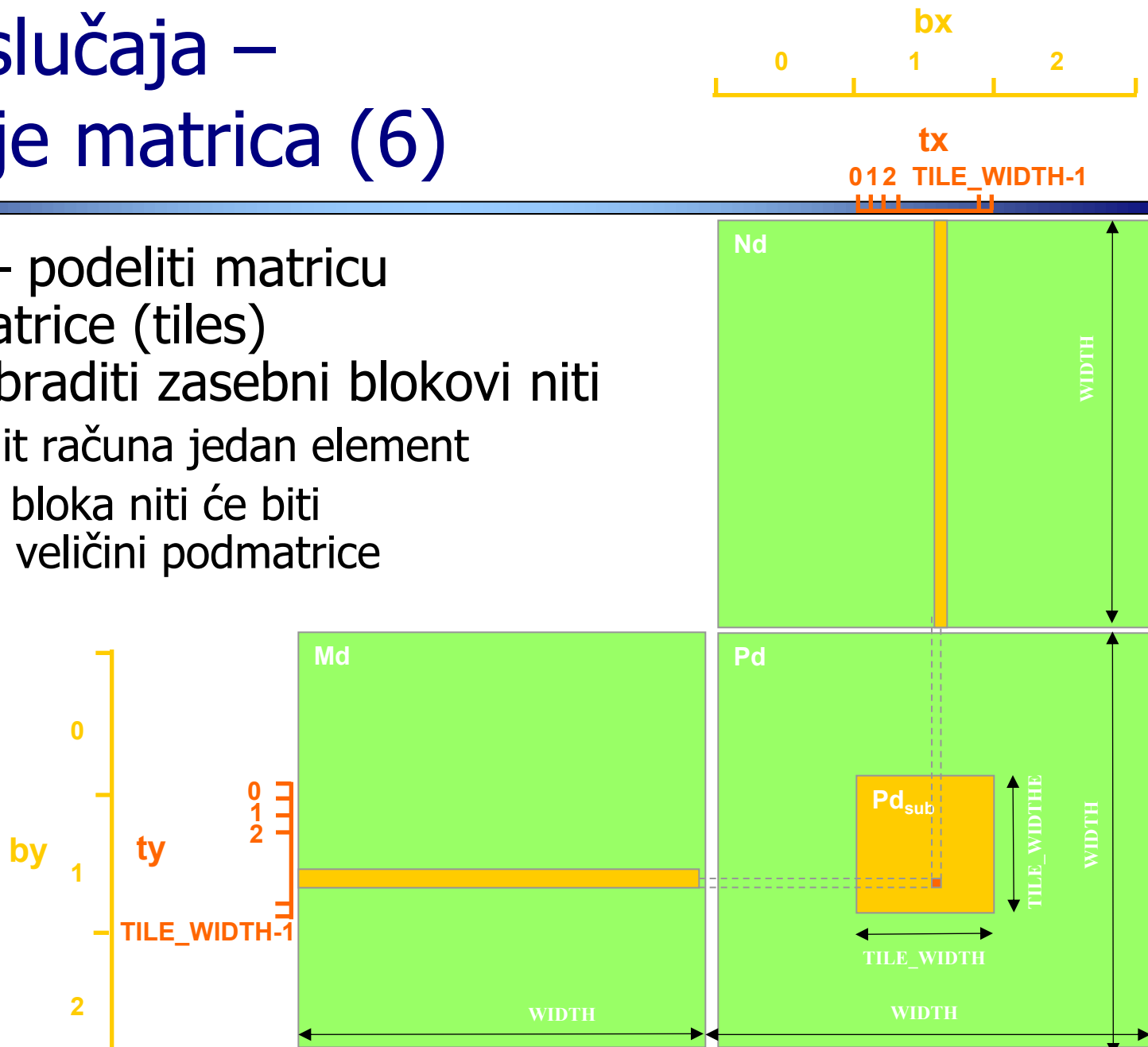


Nedostaci predloženog rešenja (2)

- 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 (512/1024 niti)

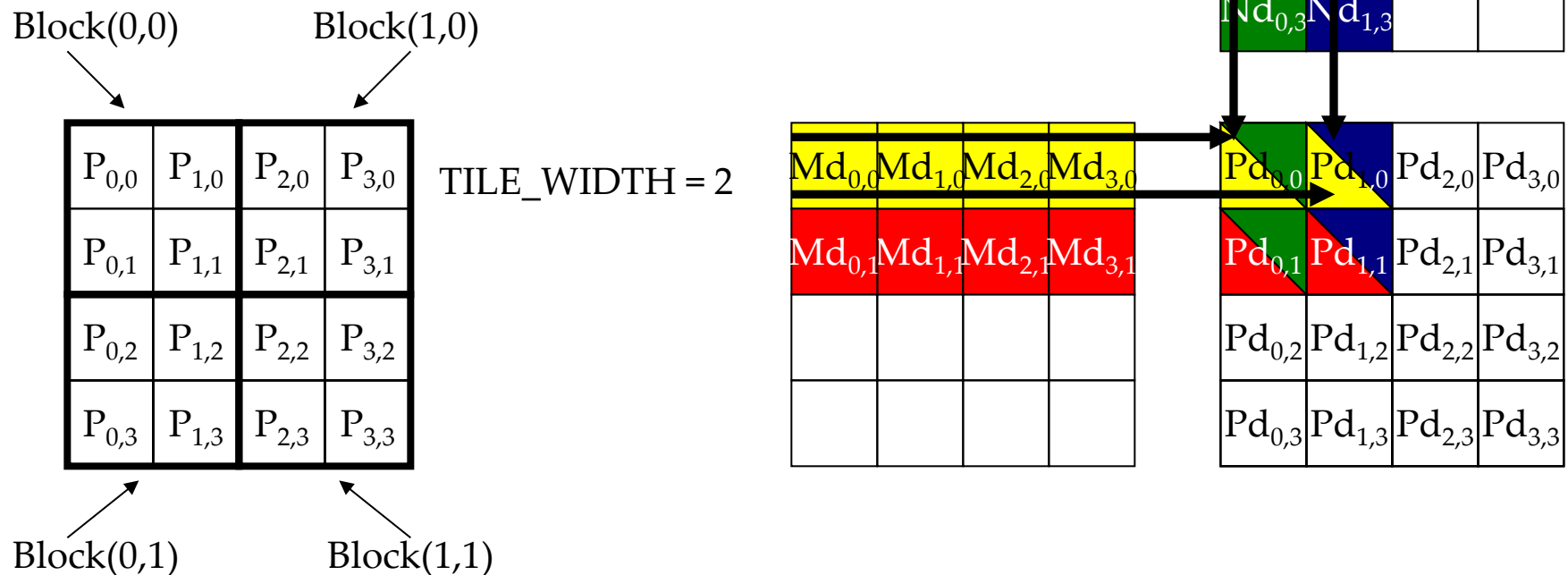
Studija slučaja – množenje matrica (6)

- 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;
    }
}
```

Podeli i sumiraj

- *Partition and Summarize*
 - Česta algoritamska strategija za procesiranje velikih skupova podataka na GPU
 - Nalikuje *MapReduce* radnom okviru za distribuiranu obradu
- Tehnika podrazumeva da:
 - Ne postoji neki zahtevani poredak obrade elemenata u skupu
 - Operacije koje se sprovode su asocijativne i komutativne
 - Skup podataka se deli na manje celine (*chunks*)
 - Jedna nit obrađuje jednu celinu
 - Redukciono stablo se koristi da se rezultati kombinuju u krajnji rezultat
- Poslednji korak je tipično najvažniji u efikasnoj implementaciji na GPU

Operacija redukcije (1)

- Tipična operacija koja se javlja u paralelnim algoritmima je operacija redukcije
 - Kao što su algoritmi sa logikom *Partition and Summarize*
- Redukcija prodrazumeva svođenje niza vrednosti na jednu skalarnu vrednost
 - Računanje zbira, proizvoda, maksimuma, minimuma, logičko I/ILI nad svim elementima strukture i sl.
 - Upotreba dobro definisane jedinične vrednosti
- Podrazumeva se da je operator koji se upotrebljava u redukciji asocijativan
 - Tehnički, to ne mora biti tačno za aritmetiku u pokretnom zarezu
 - U ovom slučaju treba biti pažljiv

Operacija redukcije (2)

- Redukcije se koriste u velikom broju primena, mada ne obavezno u paralelnoj formi
 - Kod množenja matrice, pojedinačna nit je vršila redukciju prilikom računanja skalarnog proizvoda vrste i kolone matrice
- Redukcija je potrebna zbog određenih transformacija koje se obavljaju prilikom paralelizacije koda
 - Tipičan primer je privatizacija izlazne lokacije, kako bi se smanjila potreba za sinhronizacijom
 - Umesto da više niti dodaje svoj rezultat u istu lokaciju, svaka nit dobija svoju privatnu lokaciju
 - Konačan rezultat se dobija kombinovanjem vrednosti privatnih lokacija putem redukcije

Sekvencijalna redukcija

- Inicijalizuje se rezultat jediničnom vrednošću redukcione operacije
 - 0 za zbir
 - 1 za proizvod
 - Najmanja ili najveća vrednost za max i min
- Iterira se kroz ulazni niz i sprovodi se redukciona operacija nad rezultatom i trenutnom vrednošću elementa na ulazu
 - Sprovodi se N redukcioni operacija za N ulaznih vrednosti
 - Složenost $O(n)$
 - Računski efikasan algoritam

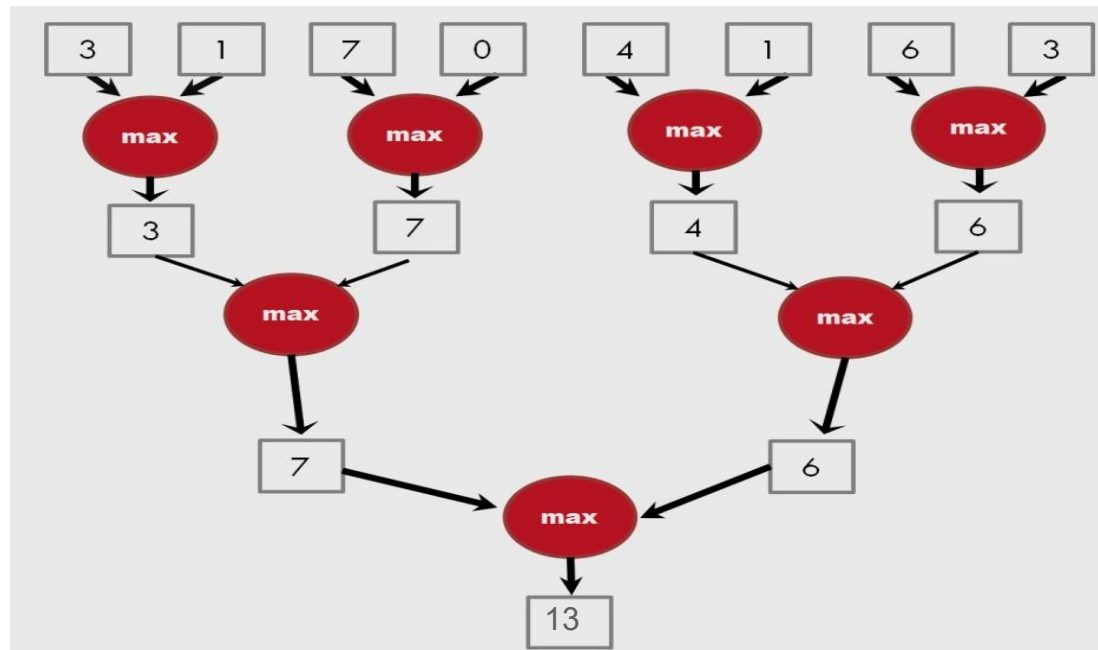
Paralelna redukcija (1)

- Neefikasan način da se ovo obavi na GPU predstavlja sledeći kod:
 - Nit 0 iz bloka sabira elemente sekvencijalno
 - Ostale niti ne rade ništa

```
if (threadID == 0) {  
    float sum = 0;  
    for (int t = 0; t < n; ++t)  
        sum += array[t];  
}
```

Paralelna redukcija (2)

- Da bismo izvršili redukciju N elemenata u paraleli, možemo koristiti princip stabla
 - Redukcija se vrši kroz $N-1$ operaciju u $\log(N)$ koraka



Složenost paralelne redukcije

- Redukcija se obavlja u $\log(N)$ paralelnih koraka
 - U svakom koraku S , izvršava se $N/2^S$ paralelnih operacija
- Za $N=2^D$, izvršava se $\sum_{S \in [1..D]} 2^{D-S} = N-1$ operacija
 - Algoritam je računski efikasan, jer ne izvršava više operacija nego sekvencijalna redukcija
- Sa P niti fizički u paraleli, vremenska složenost je $O(N/P + \log N)$
 - U poređenju sa $O(N)$ za sekvencijalnu redukciju

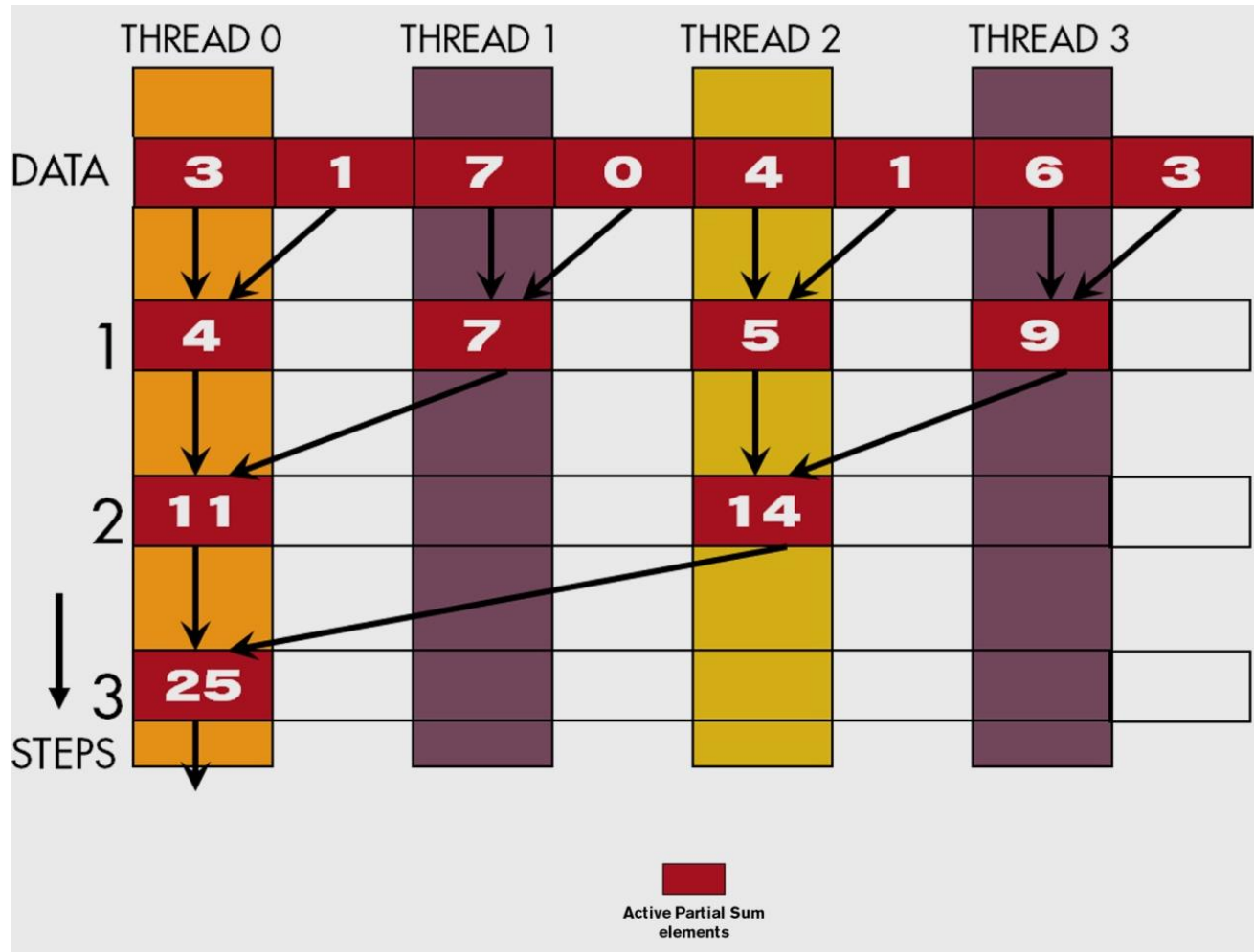
Paralelna redukcija na GPU (1)

- Primer sumiranja elemenata vektora
 - Vektor se nalazi u globalnoj memoriji uređaja
 - Dužina je deljiva veličinom bloka niti
- Prikazujemo redukciju na nivou bloka niti
 - Rekurzivno se deli broj niti na pola u svakom koraku
 - Sumiraju se dve vrednosti na nivou niti u svakom koraku
- Finalna redukcija se može uraditi ponovnim pozivanjem jezgra ili na domaćinu

Paralelna redukcija na GPU (2)

- Redukcija se obavlja *in-place* korišćenjem deljene memorije
 - Pretpostavimo da se vektor nalazi u globalnoj memoriji uređaja
 - Koristi se deljena memorija za smeštanje međurezultata (vektora parcijalnih suma)
 - U svakoj iteraciji dobija se vektor parcijalnih suma
 - U svakom koraku smo bliži konačnom rezultatu
 - Finalni rezultat se smešta u element 0 u deljenoj memoriji

Paralelna redukcija na GPU (3)



Paralelna redukcija na GPU (4)

- U primeru sa slike:
 - Svaka nit je odgovorna za jednu (izlaznu) lokaciju sa parnim indeksom u vektoru parcijalnih suma
 - Nakon svakog koraka redukcije, polovina niti više nije potrebna
 - Jedna od ulaznih lokacija za svaku aktivnu nit je uvek jedna od izlaznih lokacija iz prethodnog koraka
 - U svakom koraku se distanca između ulaznih elementa svake aktivne niti povećava
- Naivna implementacija redukcije

Naivna paralelna redukcija (1)

```
__global__ void reduce(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Load shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];

    // Do reduction in shared memory
    for (unsigned int stride = 1; s < blockDim.x; s *= 2) {
        __syncthreads();
        int index = 2 * stride * tid;
        if (index < blockDim.x) {
            sdata[index] += sdata[index + stride];
        }
    }

    // Thread 0 writes result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Memorijski
konflikt

Naivna paralelna redukcija (2)

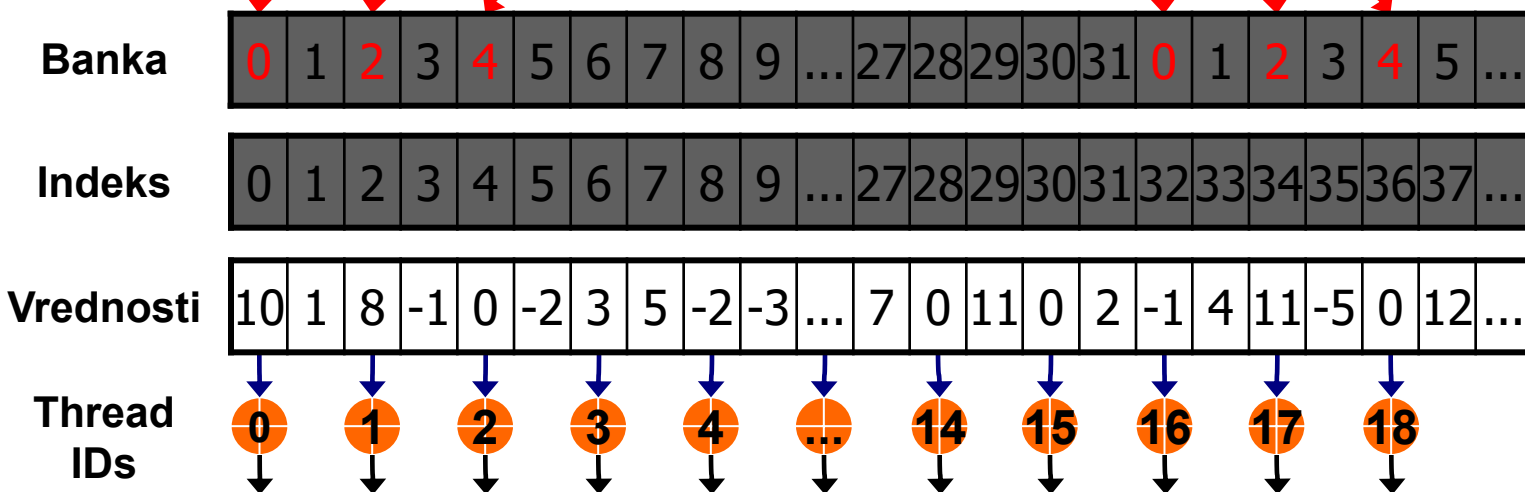
- Dešavaju se konflikti u deljenoj memoriji
 - Dešava se simultani pristup kod indeksiranja niza i to dva puta
 - Prvi pristup (`sdata[index]`)

Niti 0 i 16 pristupaju istoj memorijskoj banki

Niti 1 i 17 pristupaju istoj memorijskoj banki

Niti 2 i 18 pristupaju istoj memorijskoj banki, itd.

Korak 1



Naivna paralelna redukcija (3)

- Drugi simultani pristup (`sdata[index + stride]`)

Niti 0 i 8 pristupaju istoj memorijskoj banki

Niti 1 i 9 pristupaju istoj memorijskoj banki

Niti 2 i 10 pristupaju istoj memorijskoj banki, itd.

Korak 1

Banka

0	1	2	3	4	5	6	7	8	9	...	27	28	29	30	31	0	1	2	3	4	5	...
---	---	---	---	---	---	---	---	---	---	-----	----	----	----	----	----	---	---	---	---	---	---	-----

Indeksi

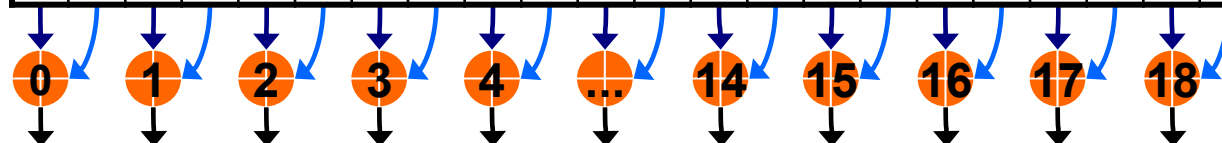
0	1	2	3	4	5	6	7	8	9	...	27	28	29	30	31	32	33	34	35	36	37	...
---	---	---	---	---	---	---	---	---	---	-----	----	----	----	----	----	----	----	----	----	----	----	-----

Vrednosti

10	1	8	-1	0	-2	3	5	-2	-3	...	7	0	11	0	2	-1	4	11	-5	0	12	...
----	---	---	----	---	----	---	---	----	----	-----	---	---	----	---	---	----	---	----	----	---	----	-----

Pomeraj 1

Nit ID



Naivna paralelna redukcija (4)

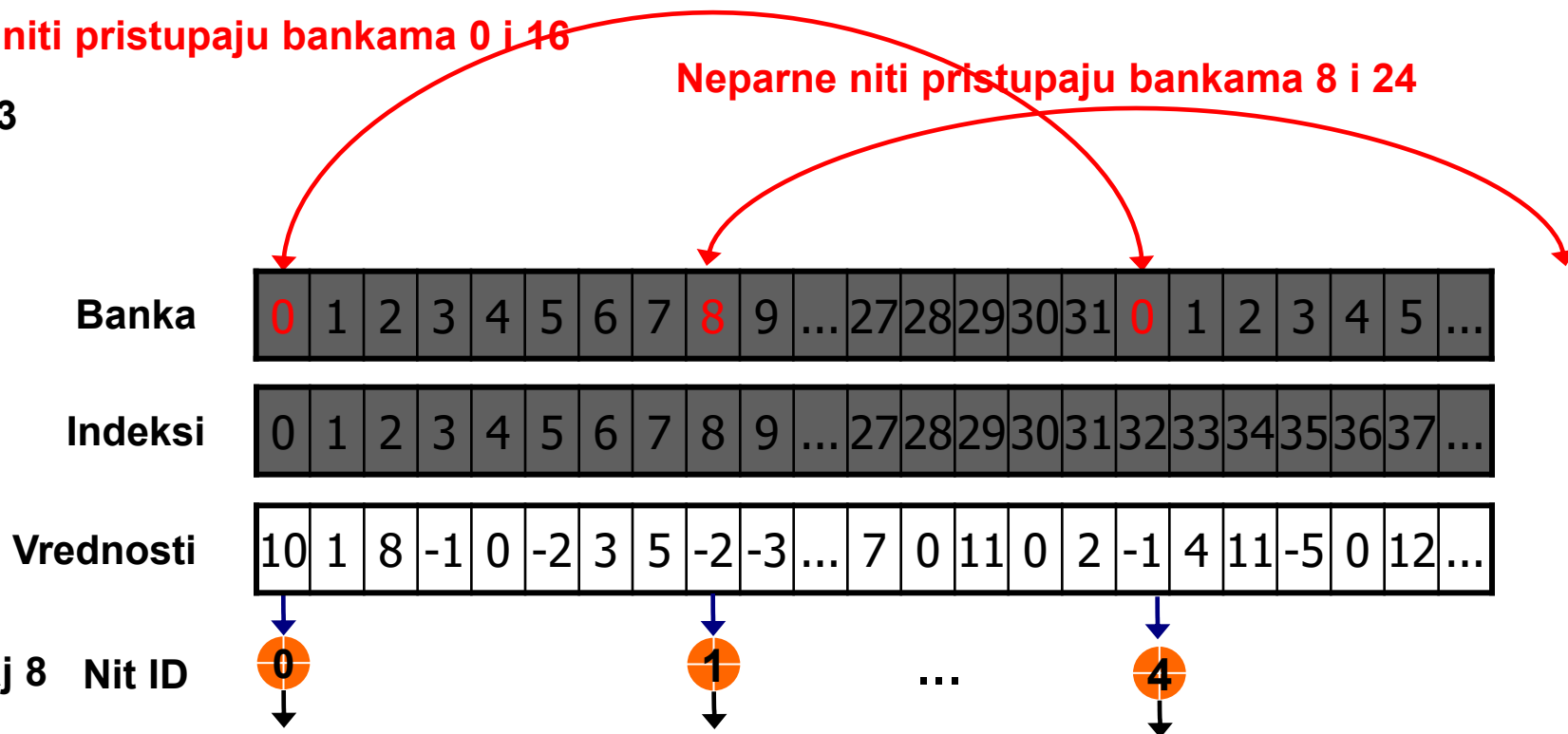
- Prvi simultani pristup (`sdata[index]`)

- 4-ostruki konflikt

Parne niti pristupaju bankama 0 i 16

Neparne niti pristupaju bankama 8 i 24

Korak 3



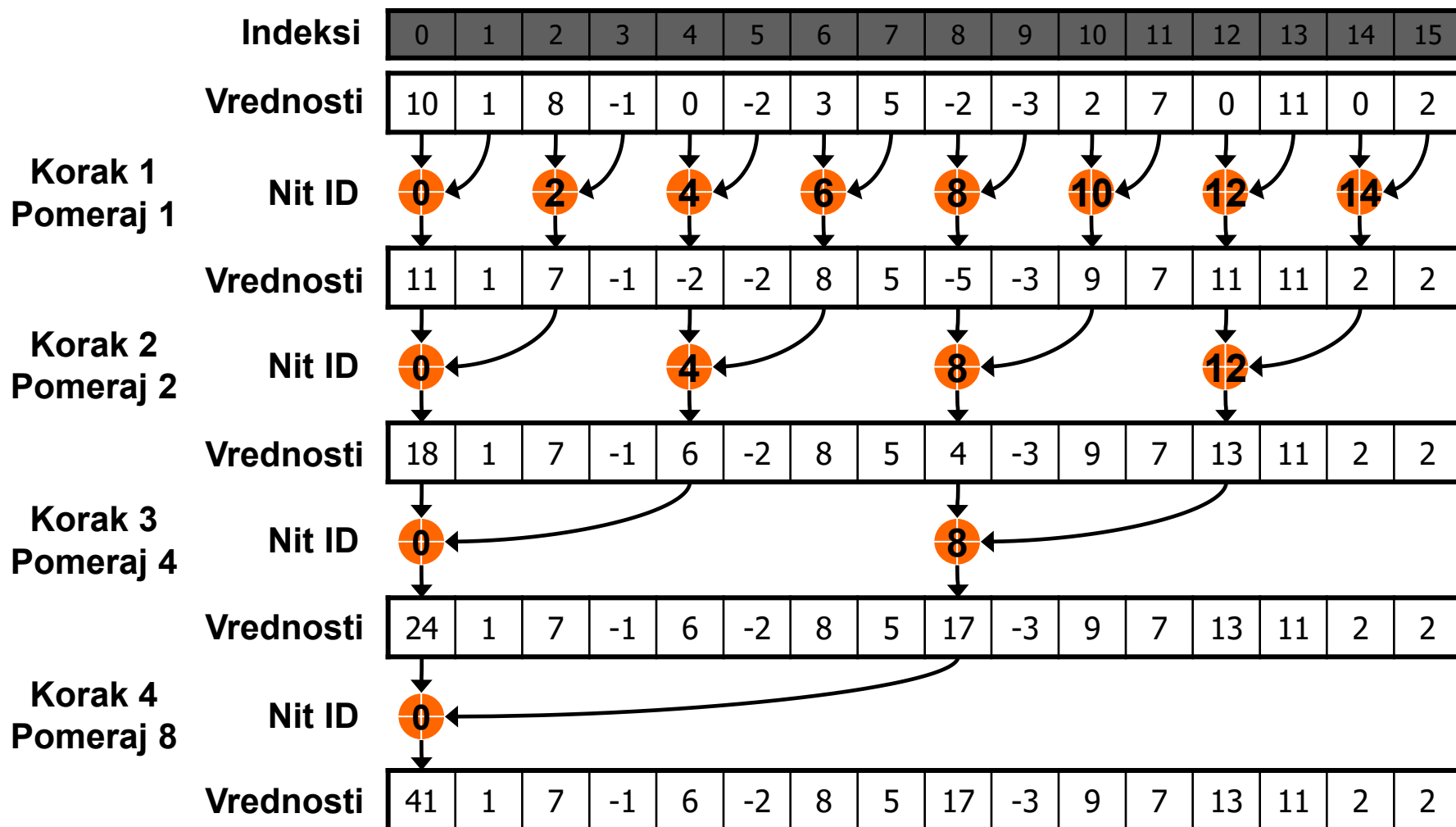
Redukcija bez konflikta (1)

- Korektna implementacija bez memorijskih konflikta:

```
for (int s = 1; s < blockDim.x; s *= 2) {  
    __syncthreads();  
    if (threadIdx.x % (2 * s) == 0)  
        sdata[threadID] += sdata[threadIdx.x + s];  
}
```

- Problem je veliki broj divergentnih *warp*-ova prilikom izvršavanja ovakvog koda
 - Polovina niti (parne niti) će biti aktivna, a polovina ne (neparne niti)
 - Ukoliko postoje grananja unutar *warp*-a, sve niti će izvršiti sve putanje, što dovodi do usporenja

Redukcija bez konflikta (2)



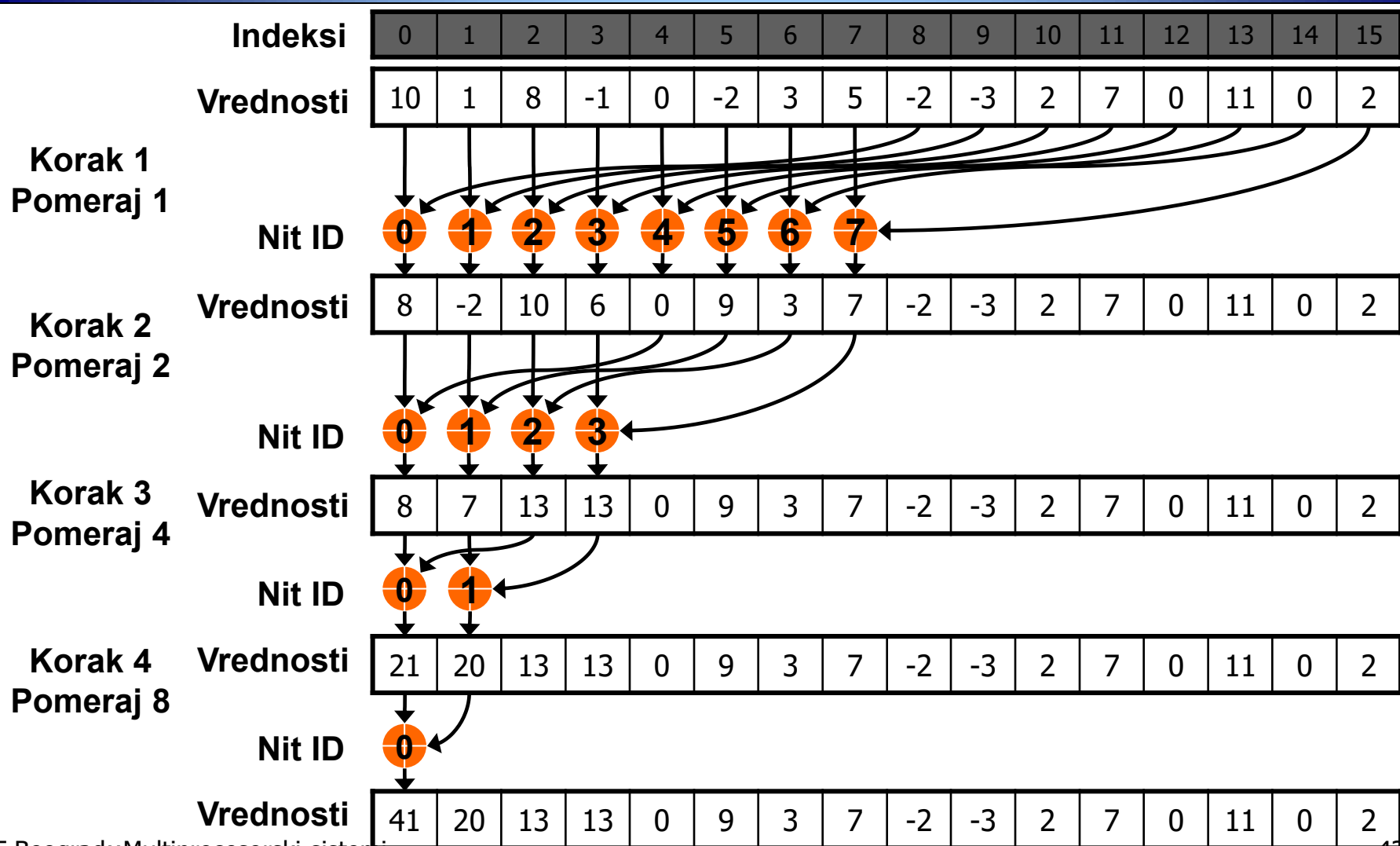
Bolje rešenje redukcije (1)

- Prethodna rešenja iskazuju sledeće probleme:
 - Memorijske konflikte (prvo rešenje)
 - Divergenciju u kontroli toka
 - Slabo iskorišćenje resursa
 - Većina niti u *warp*-u prestaje da bude aktivna posle nekoliko koraka
 - Polovina niti učestvuje samo u učitavanju podataka
 - Blok podataka redukuje efektivno $N/2$ niti

Bolje rešenje redukcije (2)

- Bolje rešenje redukcije se dobija primenom sledećih saveta:
 - Parcijalne sume treba održavati u početnim lokacijama niza
 - Aktivne niti treba da budu konsektivne u bloku
- Ideja je da niti pristupaju nesusednim elementima u bloku koji redukuju
 - Moguće s obzirom da su redukcionni operatori komutativni i asocijativni

Bolje rešenje redukcije (3)



Bolje rešenje redukcije (4)

```
__global__ void reduce(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // load shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] = sdata[tid] + sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Bolje rešenje redukcije (5)

- Treba primetiti da nakon učitavanja elementa u deljenu memoriju samo polovina niti radi redukciju:

```
for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {  
    if (tid < s)  
        sdata[tid] = sdata[tid] + sdata[tid + s];  
    __syncthreads();  
}
```
- To se može promeniti tako što će svaka nit raditi dva čitanja i prvo sumiranje u redukciji:
 - Broj blokova se tako smanjuje za pola

```
unsigned int tid = threadIdx.x;  
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;  
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];  
__syncthreads();
```

Bolje rešenje redukcije (6)

```
__global__ void reduce(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // load shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();
    // do reduction in shared mem
    for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] = sdata[tid] + sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Uklanjanje petlji (1)

- Samo jedan *warp* unutar bloka je aktivan u poslednjih nekoliko koraka redukcije
 - Broj iteracija petlje se može smanjiti i time ukloniti suvišni pozivi `__syncthreads()`
- Može da se uradi razmotavanje petlje
 - Poznata *loop unrolling* tehnika koju sprovode prevodioci ili programer
 - Eliminišu se skokovi u petlji ponavljanjem naredbi iz tela petlje
 - Optimizacija vreme/prostor
 - Povećava dužinu binarnog koda

Uklanjanje petlji (2)

- Razmotavaju se sve iteracije petlje u kojima učestvuje samo jedan *warp* od 32 niti

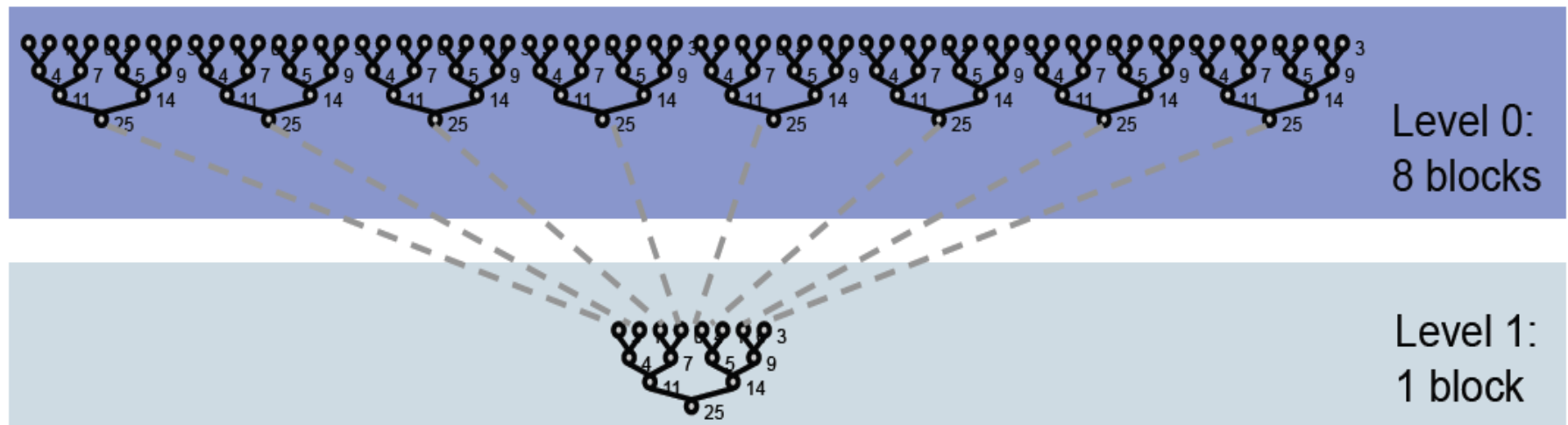
```
for (unsigned int s = blockDim.x/2; s > 32; s >>= 1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
if (tid < 32) sdata[tid] += sdata[tid + 32];
if (tid < 16) sdata[tid] += sdata[tid + 16];
if (tid < 8) sdata[tid] += sdata[tid + 8];
if (tid < 4) sdata[tid] += sdata[tid + 4];
if (tid < 2) sdata[tid] += sdata[tid + 2];
if (tid < 1) sdata[tid] += sdata[tid + 1];
```

Redukcija nad velikim nizovima (1)

- Do sada je razmatrana redukcija na nivou bloka podataka
 - Nezavisni blokovi niti redukuju jedan blok podataka iz globalne memorije
 - CUDA ne dozvoljava globalnu sinhronizaciju među blokovima
- Potreban način da niti iz različitih blokova međusobno sarađuju
 - Rešenje je iskoristiti višestruke pozive istog jezgra
 - Poziv jezgru se može posmatrati kao globalna sinhronizaciona tačka
 - Poziv jezgru troši zanemarljivo režijsko vreme
 - Nakon završetka redukcije na nivou pojedinačnih blokova, treba pozvati jezgro ponovo, sada samo sa jednim blokom
 - Alternativno, poslednji korak se može izvršiti na CPU

Redukcija nad velikim nizovima (2)

- Primer redukcije u više nivoa
 - *Tiled* algoritam
 - Svaki blok niti redukuje jedan podblok podataka iz ulaznog niza
 - Međurezultati se smeštaju u globalnu memoriju i koriste kao ulazni podaci za sledeći poziv jezgru
 - Postupak se ponavlja onoliko puta koliko je potrebno

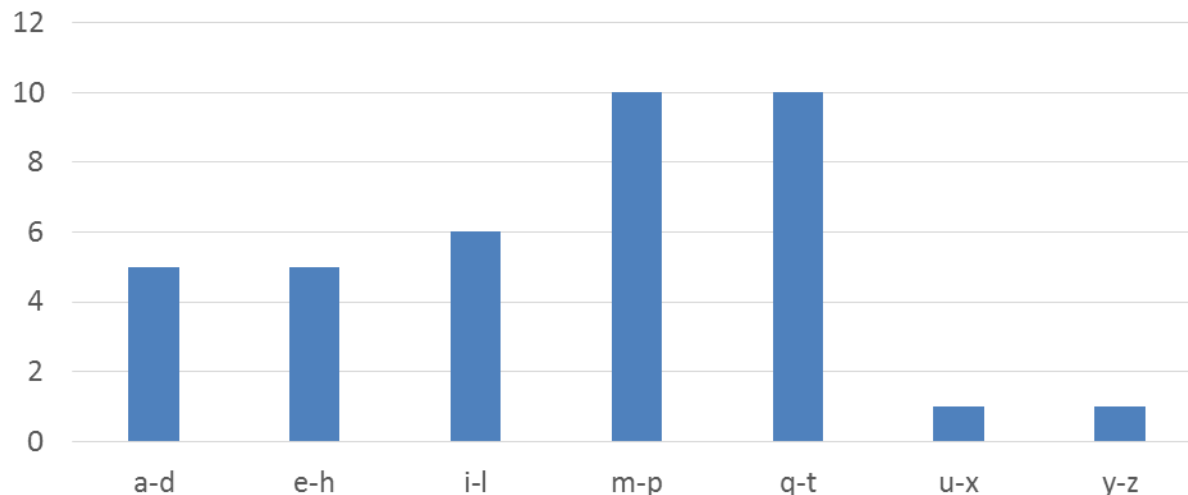


Određivanje histograma (1)

- Veoma često je potrebno odrediti statistiku pojavljivanja elemenata na skupu elemenata
 - Statistiku pojavljivanja pojedinačnih elemenata
 - Statistiku pojavljivanja elemenata klasifikovanih u određene podskupove ili opsege (*bins*)
- Bazično rešenje:
 - Koristiti niz brojača inicijalizovanih na 0
 - Za svaki element u skupu, iskoristiti vrednost da se identifikuje odgovarajući brojač koji treba inkrementirati
- Efikasno rešenje na grafičkom procesoru zahteva dosta drugačiji pristup od dosadašnjih
 - Zbog postojanja interferencije niti na izlazu

Određivanje histograma (2)

- Primer određivanja histograma stringa
Programming Massively Parallel Processors
 - Pretpostaviti da svaki brojač predstavlja četiri uzastopna slova engleskog alfabeta
 - *a-d, e-h, i-l, m-p, q-t, u-x, y-z*
 - Za svaki karakter ulaznog stringa treba inkrementirati odgovarajući brojač



Određivanje histograma (3)

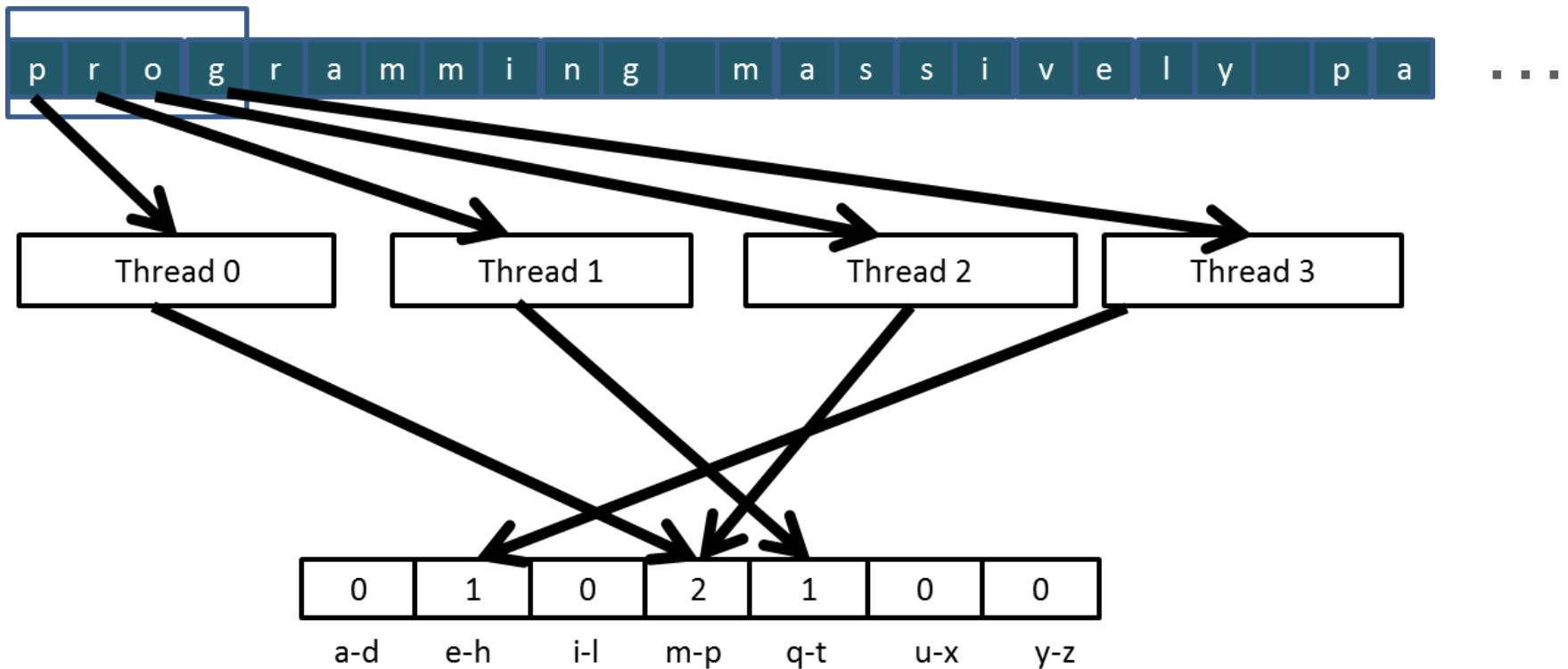
- Bazično rešenje na grafičkom procesoru
 - Podeliti ulazni niz na blokove (*tile*)
 - Jedna nit će biti zadužena za jedan element bloka
 - Nit će inkrementirati odgovarajući brojač
 - Uzastopne niti pristupaju uzastopnim elementima kako bi se obezbedio sjedinjeni pristup
 - Po potrebi, niti ponavljaju postupak



Interleaved partitioning

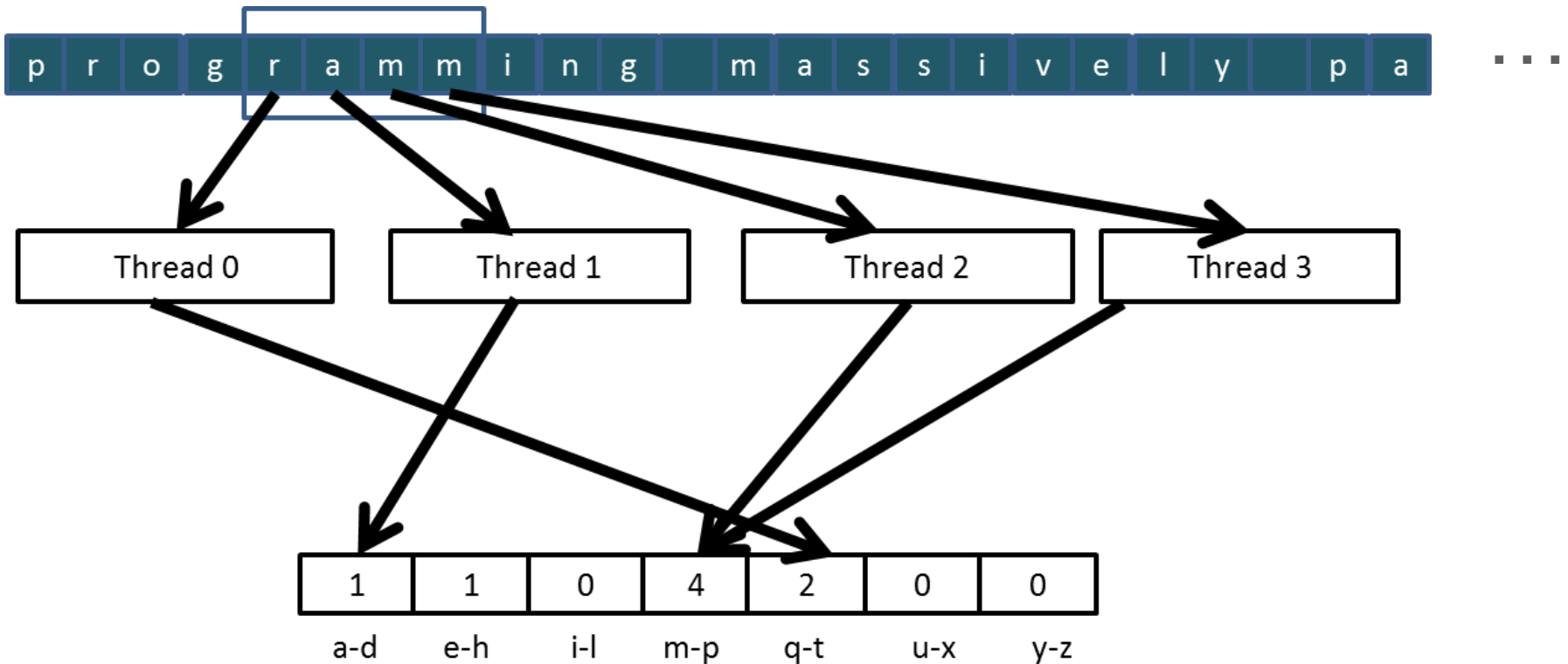
Određivanje histograma (4)

- Prvi blok niti



Određivanje histograma (5)

- Drugi blok niti
 - Primećuje se *race condition* problem kod pristupa brojačima!



Određivanje histograma (6)

- Način podele podataka po nitima je dobar
 - Dobro se koristi propusni opseg
- Međutim, postoji utrkivanje (hazard) prilikom pristupa brojačima
 - Dešava se *Read-Modify-Write* sekvenca operacija
 - Mora se sprovesti atomično
- Rešenje je u korišćenju atomičnih operacija dostupnih u okviru CUDA jezgra
 - Na hardveru koji ih podržava

Bazično histogram jezgro (1)

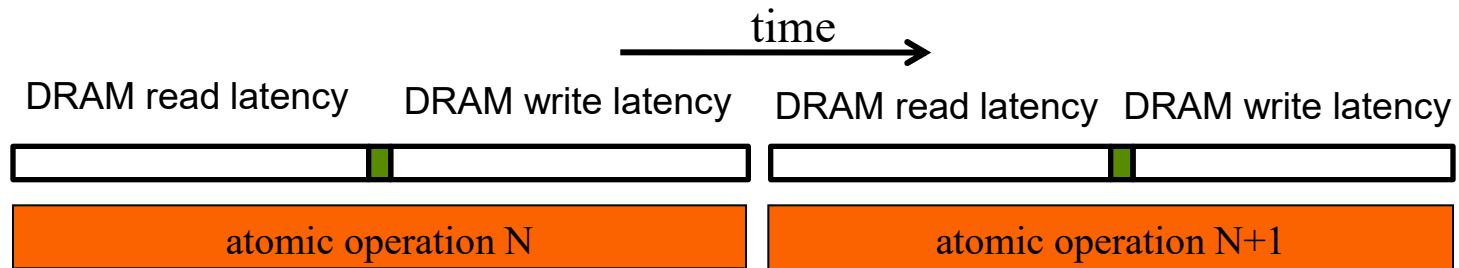
```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;

    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        int alphabet_position = buffer[i] - "a";
        if (alphabet_position >= 0 && alphabet_position < 26)
            atomicAdd(&(histo[alphabet_position/4]), 1);
        i += stride;
    }
}
```


Bazično histogram jezgro (2)

- Predstavljeno rešenje je korektno
 - Međutim, postoji nekoliko važnih nedostataka
- Atomične operacije konzumiraju značajan propusni opseg
 - Atomična operacija nad DRAM lokaciom počenje sa čitanjem koje ima kašnjenju od nekoliko stotina ciklusa
 - Atomična operacija nad DRAM lokaciom završava se pisanjem koje ima kašnjenju od nekoliko stotina ciklusa
 - U međuvremenu, nijedna nit ne može da pristupi lokaciji
- Svaka *Read-Modify-Write* sekvenca operacija prouzrokuje dvostruko kašnjenje u pristupu memoriji
 - Više pristupa istoj lokaciji prouzrokuje serijalizaciju

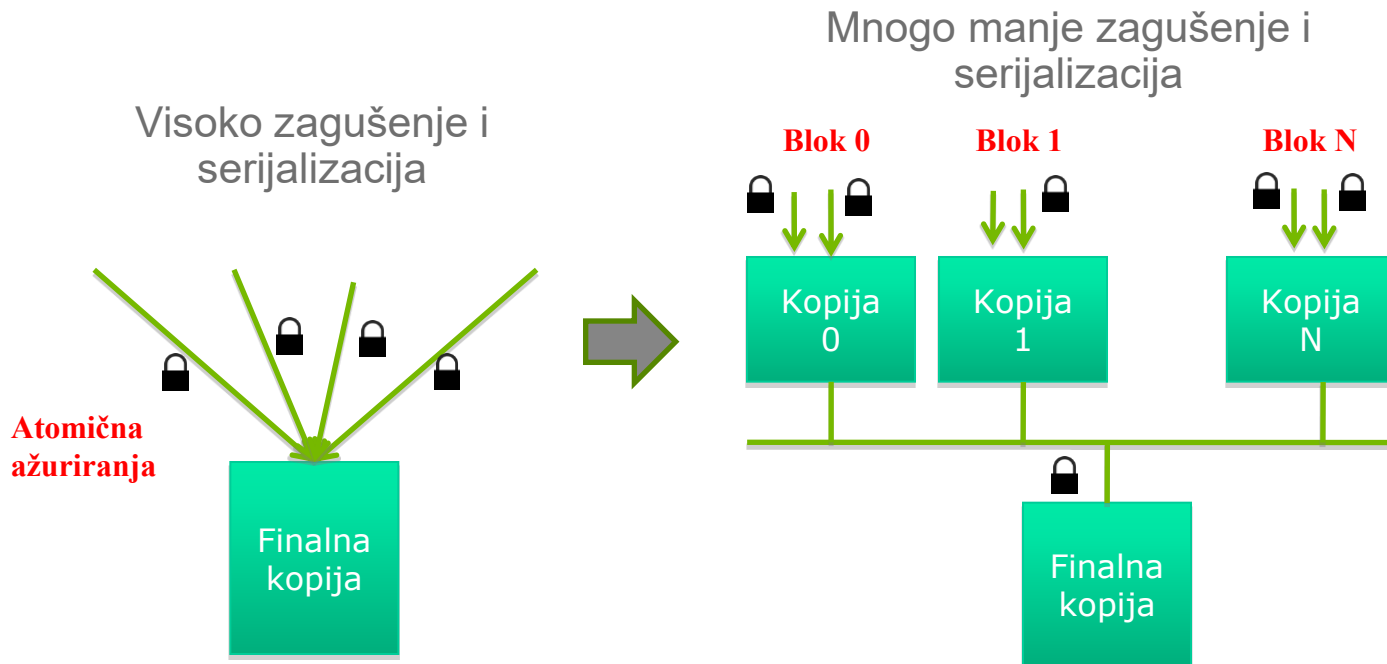


Propusni opseg atomičnih operacija

- Propusni opseg atomičnih operacija je limitiran
 - Zavisi od brzine kojom atomična operacija može da se sprovede
- Brzina sprovođenja atomične operacije je ograničena ukupnim kašnjenjem
Read-Modify-Write sekvence operacija
 - Tipično oko 1000 ciklusa za DRAM lokaciju globalne memorije
- Hardverska poboljšanja utiču na ovu brzinu
 - Atomične operacije se značajno brže (~ 10 puta) nad lokacijama u L2 kešu
 - L2 keš je deljen od strane svih blokova niti na SM-u
 - Još brže se sprovode nad deljenom memorijom, što uobičajeno zahteva algoritamske promene u kodu

Bolje histogram jezgro (1)

- Rešenje za poboljšanje performansi jezgra u slučaju određivanja histograma je privatizacija izlaza
 - Tehnika privatizacije omogućava smanjivanje kašnjenja, povećanje propusnog opsega i smanjivanje serijalizacije



Bolje histogram jezgro (2)

- Privatizacija podataka donosi prednosti, ali i režijske troškove
 - Na GPU se privatizacija radi korišćenjem deljene memorije
- Prednosti privatizacije
 - Mnogo manje zagušenje i serijalizacija kod pristupa privatnim kopijama i finalnoj kopiji rezultata
 - Samo niti iz istog bloka pristupaju privatnim brojačima
 - Performanse mogu da se poboljšaju i do 10x
- Režijski troškovi privatizacije
 - Trošak kreiranja i inicijalizacije privatnih kopija
 - Trošak akumuliranja sadržaja privatnih kopija na globalnu kopiju sa finalnim rezultatom

Bolje histogram jezgro (3)

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo) {
    __shared__ unsigned int histo_private[7];

    if (threadIdx.x < 7) histo_private[threadIdx.x] = 0;
    __syncthreads();

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &(amp;private_histo[buffer[i]/4]), 1);
        i += stride;
    }
    // wait for all other threads in the block to finish
    __syncthreads();
    if (threadIdx.x < 7)
        atomicAdd(amp;(histo[threadIdx.x]), private_histo[threadIdx.x] );
}
```

Bolje histogram jezgro (4)

- Privatizacija je moćna i često korišćena tehnika za paralelizaciju
 - Operacija koja se sprovodi mora da bude asocijativna i komutativna
 - Računanje histograma jeste takva operacija
- Ograničenja jezgra za računanje histograma postoje
 - Privatni histogrami moraju da budu dovoljno mali da bi stali u deljenu memoriju
 - Kod velikih histograma se može primeniti delimična privatizacija histograma u deljenoj memoriji
 - Ostatak se onda nalazi u globalnoj memoriji

Literatura

- David Kirk, Wen-mei Hwu, Programming Massively Parallel Processors: A Hands on Approach, Morgan Kaufmann
- NVIDIA CUDA C Programming Guide 10.2, 2020.
- NVIDIA GPU Teaching Kit 2017
- Razni materijali i dokumentacija sa NVIDIA sajta
- <http://en.wikipedia.org/wiki/GPGPU>
- <http://en.wikipedia.org/wiki/CUDA>