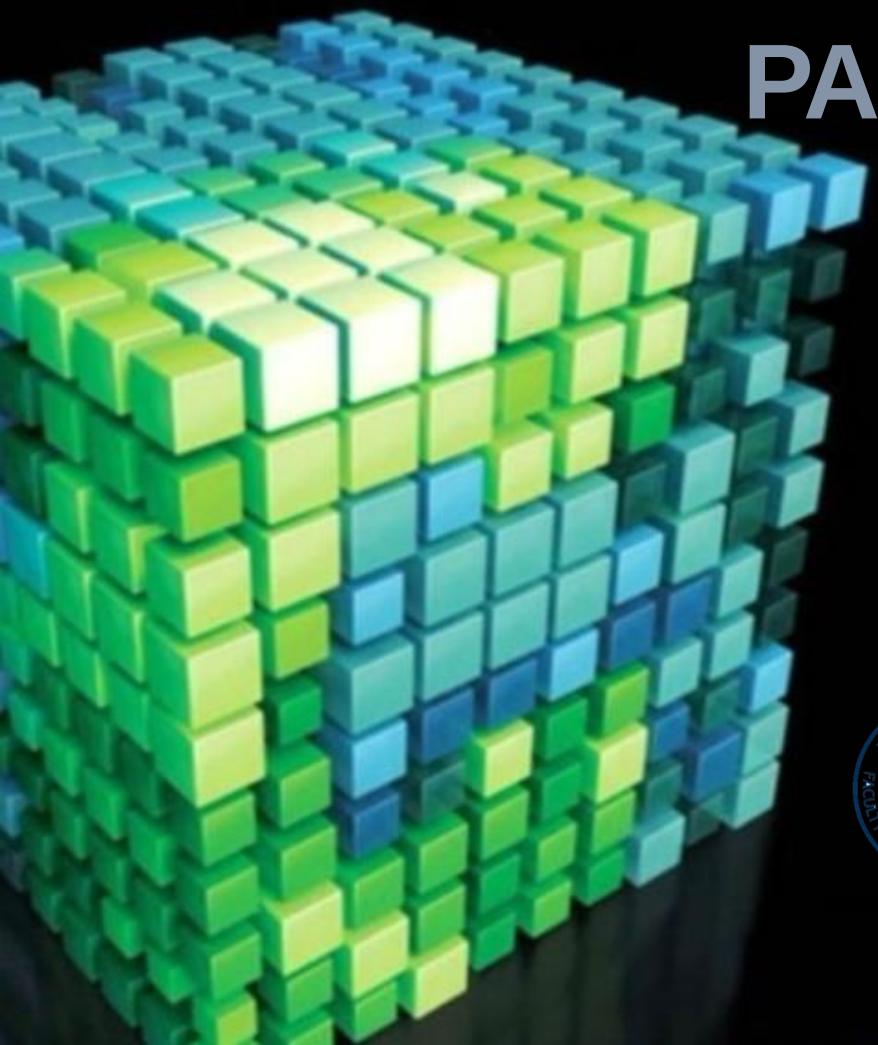
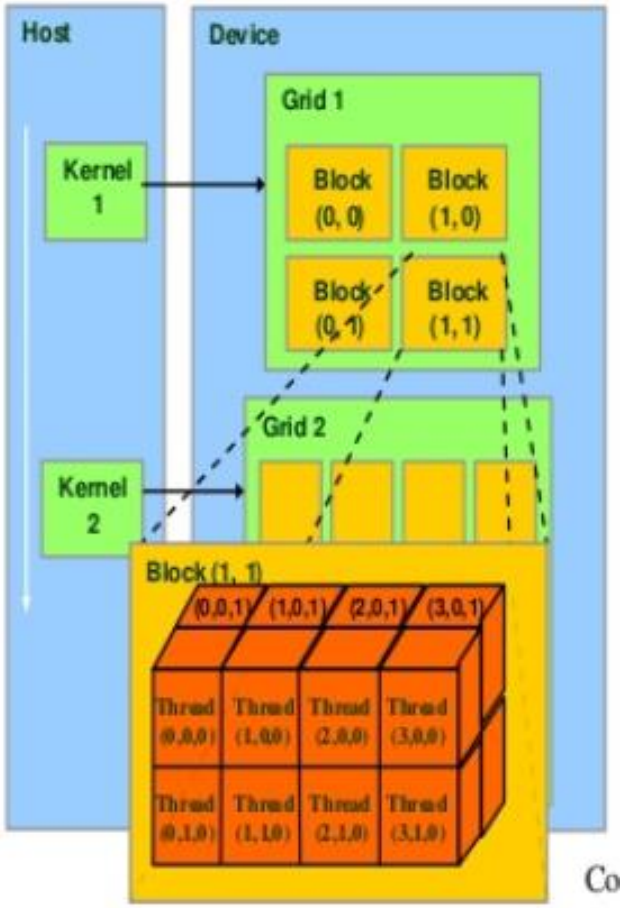


PARALELNI SISTEMI:
CUDA

MSc Aleksandra Stojnev
Prof. Dr. Natalija Stojanović

Izvršavanje niti

- Poziv CUDA kernela generiše grid niti koje su hijerarhijski podeljene u dva nivoa:
  - Na višem nivou – grid se sastoji od jednodimenzionalnog ili dvodimenzionalnog niza blokova.
  - Na nižem nivou, svaki od blokova se sastoji od jedno, dvo ili trodimenzionalnog niza niti.
- Ne postoje nikakve garancije za redosled izvršavanja blokova niti **(skalabilnost!)**.



Kako se u okviru jednog bloka izvršavaju niti?

Izvršavanje niti

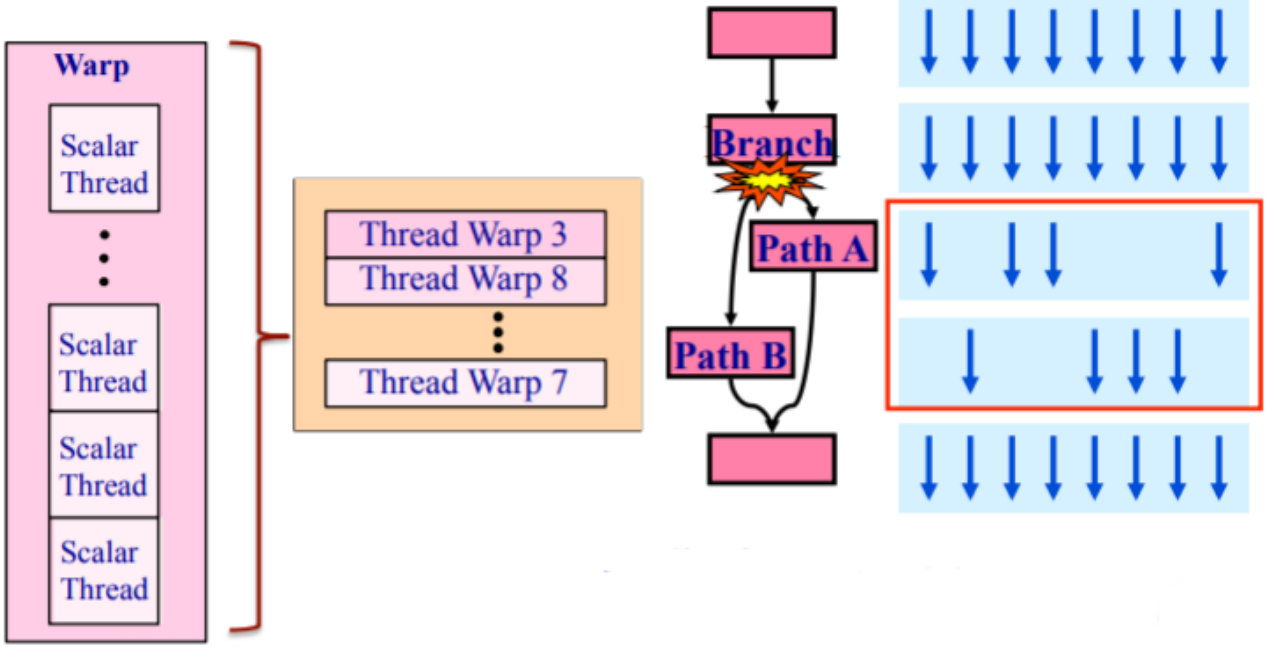
- Logično je da se niti u okviru jednog bloka mogu izvršavati u bilo kom redosledu.
  - Barijere se za osiguravanje da su sve niti završile zajedničku fazu u izvršenju pre nego što krenu u narednu.
  - Tačnost izvršenja jezgra ne bi trebalo da zavisi od cinjenice da će možda neke niti biti izvršavane u istom trenutnku.
- Zbog različitih hardverskih ograničenja i optimizacija, aktuelne generacije CUDA kartica grupišu niti za simultano izvršavanje - **warp**.
- Takva implementacija nameće neka ograničenja za pojedine tipove konstrukcija u kernelu
  - Ponekad je za poboljšanje performansi dovoljno zameniti takve konstrukcije drugima koje su ekvivalentne, ali imaju bolje performanse.

Podela u warpove

- Vrši se na osnovi indeksa niti:
  - Ako je blok organizovan kao jednodimenzionalni niz particionisanje je redom: 0-31 prvi, 32-63 drugi i tako dalje.
    - Za blokove čija veličina nije deljiva sa 32, poslednji warp se dopunjuje nitima do 32.
  - Za blokove koji imaju više dimenzija, iste se linearno projektuju:
    - Za dvodimenzionalne prvo idu sve niti koje imaju threadidx.y 0, pa 1 i td. Za trodimenzionalne blokove, prioritet ima threadidx.z (niže ka višem), pa se primenjuje pravilo za 2D.
- Hardver izvršava istu instrukciju za sve niti u okviru jednog warpa (**Single Instruction Multiple Thread**).
  - amortizacija troškova dobavljanja i procesiranja instrukcije

Warps

- **1 warp – 32 niti**
- Sve niti u okviru jednog warpa izvršavaju istu instrukciju
- Različite putanje u okviru koda se serijalizuju



Warp divergence

- Kada niti u okviru istog warpa prolaze različitom putanjom u kodu, kažemo da divergiraju u izvršavanju.

- Divergentnost se javlja kada je predikat funkcija koja zavisi od threadID
- Nema divergencije ako sve niti idu istom putanjom
- U okviru jednog bloka može biti više putanja za izvršavanje

IF-THEN-ELSE

- Optimalno kada sve niti prolaze ili kroz *then* ili kroz *else* granu.
- Kada niti ne prolaze kroz iste putanje u kodu – SIMT model više ne funkcioniše.
- U takvim situacijama, izvršavanje warpova zahteva višestruke prolaze kroz program, različitim putanjama.
  - U slucaju if-then-else jedan prolaz ide za then granu, jedan za else. Ovi prolazi su sekvencijalni, pa doprinose uvećanju vremena izvšenja.

FOR

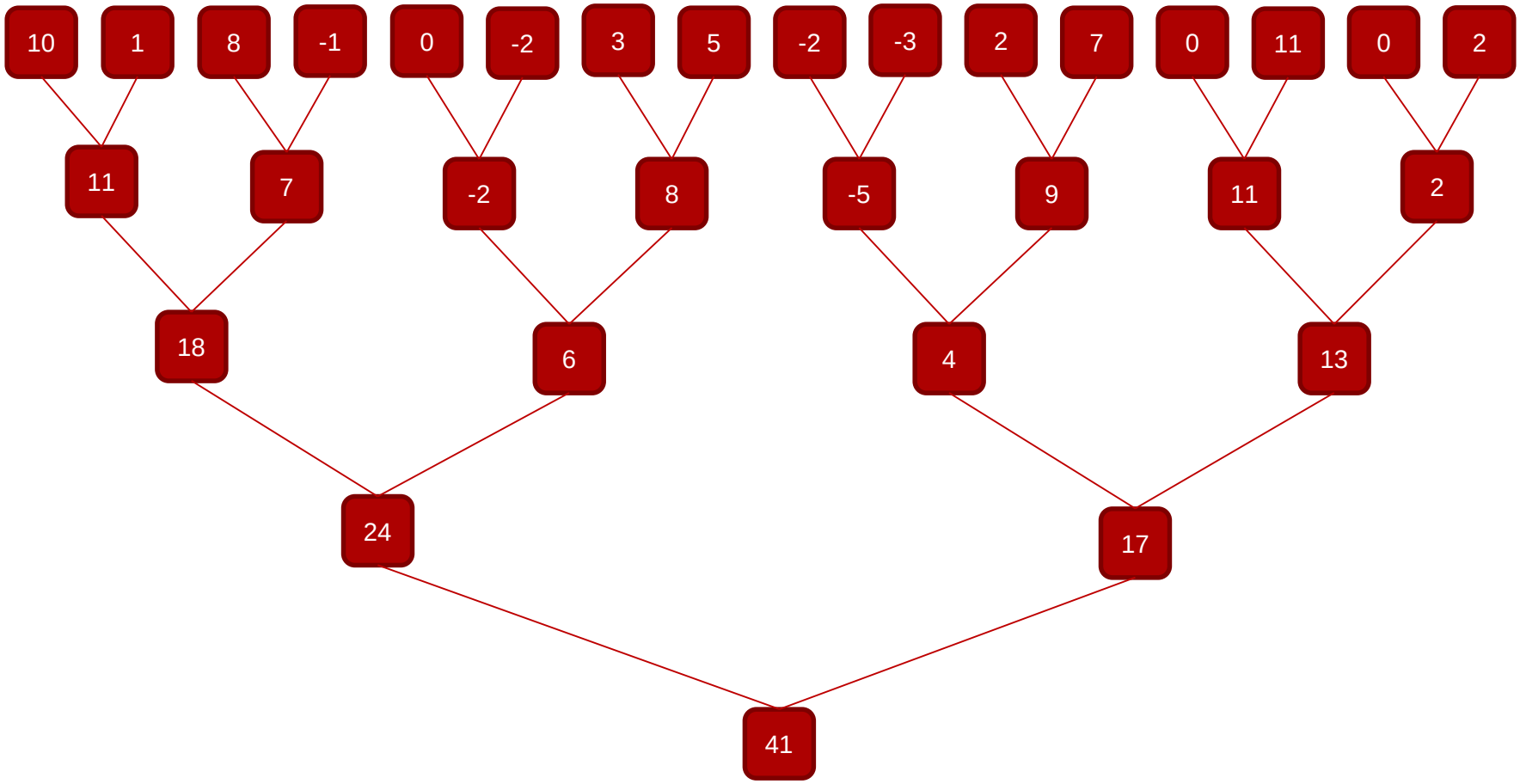
- Niti u okviru warpa izvršavaju for petlju koja može da iterira 6,7 ili 8 puta za različite niti.
  - Sve niti će prvih šest iteracija završiti zajedno.
  - Dva puta se izvršava 7 iteracija: jedan put za one koji izvršavaju, jedan put za one koje ne.
  - Isto važi i za iteraciju 8.

Warp divergence

- Divergentnost niti može pouzrokovati ozbiljan pad performansi
- Nekad je korisno koristiti dva kernela – jedan opšti i jedan za granične slučajeve

PARALELNA REDUKCIJA

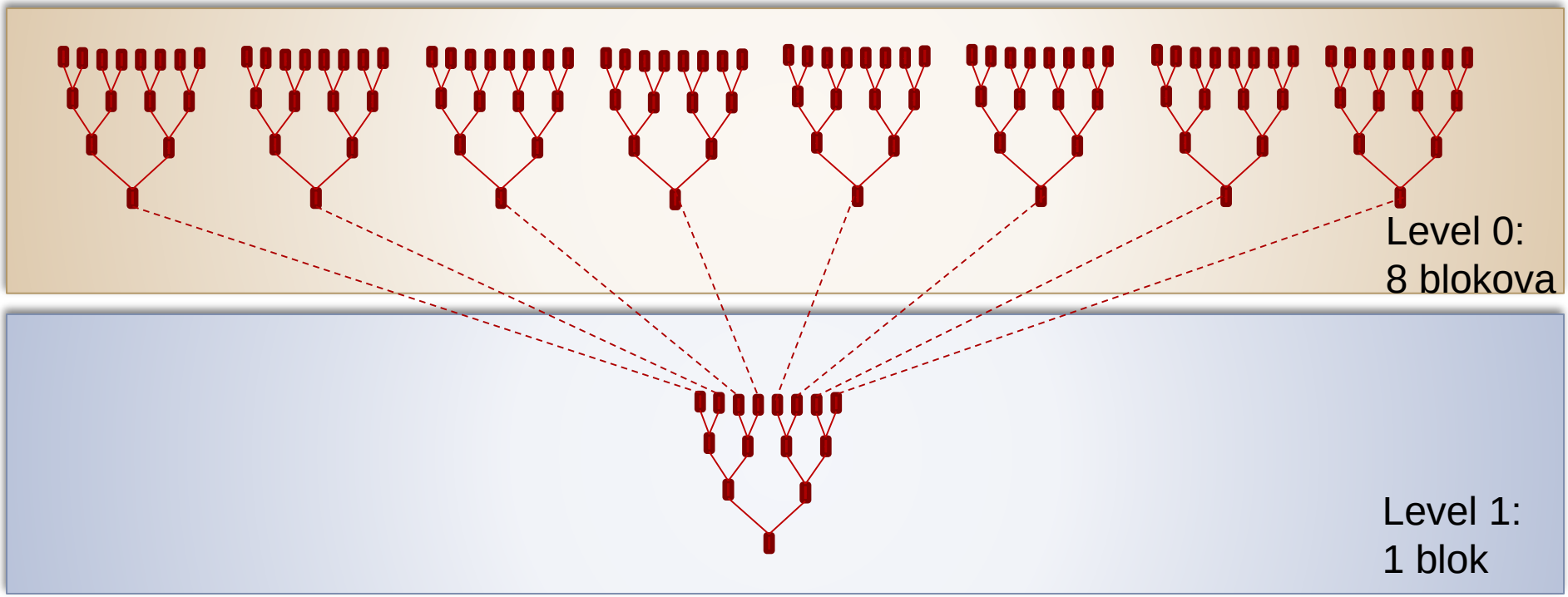
Paralelna redukcija



Paralelna redukcija

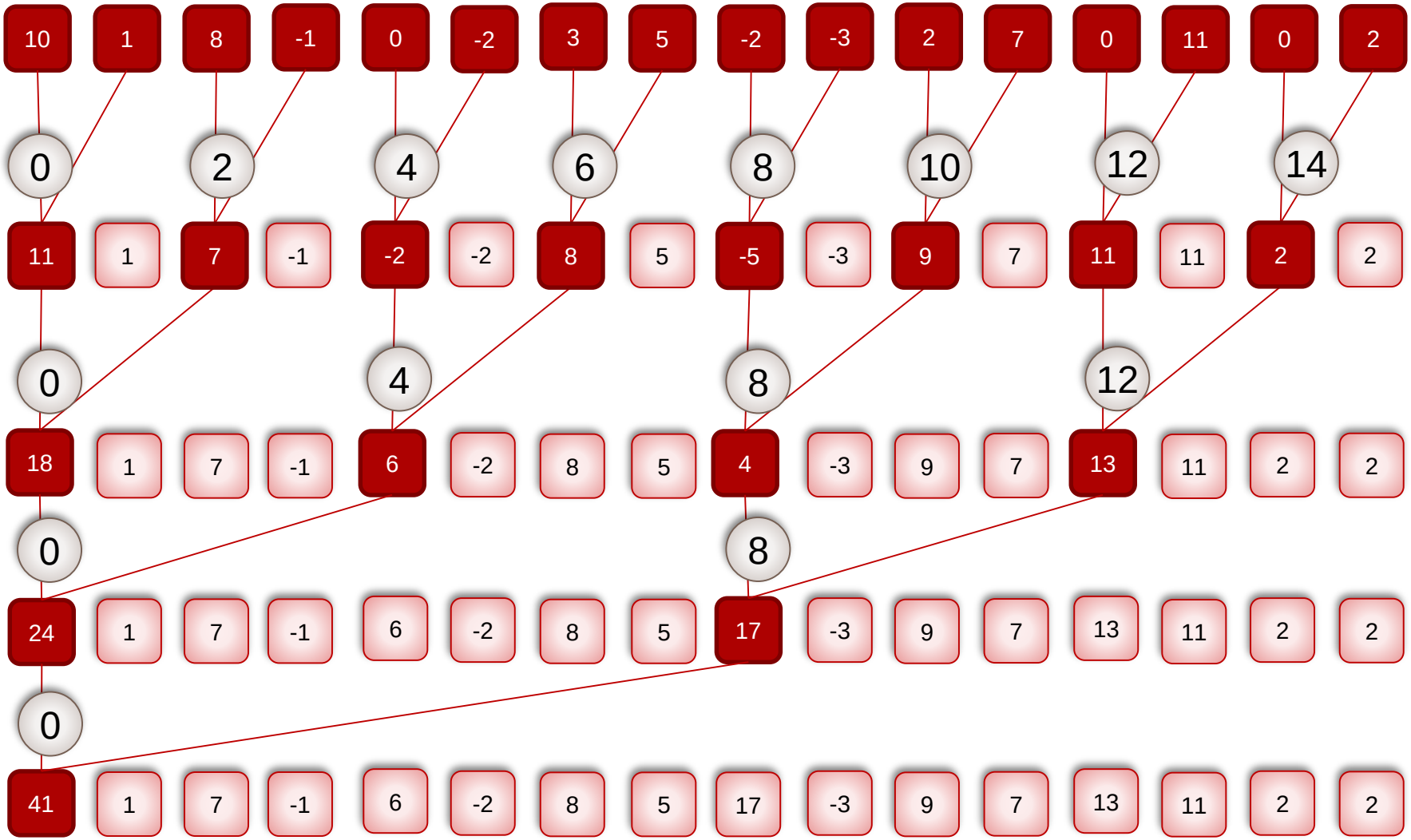
- Više blokova, više niti
  - Za procesiranje velikih nizova
  - Za upošljavanje svih SM na GPU
  - Svakom bloku dodeljuje se deo niza
- Problem: CUDA nema globalnu sinhronizaciju
  - Razbiti proces u više kernela

Paralelna redukcija



Paralelna redukcija - 1

Svaki korak – nova parcijalna suma



Paralelna redukcija – implementacija 1

- Ukupan broj niti jednak broju elemenata niza
- Svaki blok niti učitava svoj deo niza u deljivu memoriju
- Vektor inicijalizujemo tako da svi elementi budu 1
  - Zbog testiranja – ocekivana suma jednaka je ukupnom broju elemenata u vektoru
- Računanje sume vrši se u dva koraka
  - Oba koraka koriste isti kernel:
    - Prvi pokrece kernel sa vise blokova da se dobiju parcijalne sume za delove niza
    - Drugi pokrece kernel sa jednim blokom da se izračuna konačni rezultat.
- Rezultat smeštamo na poziciji 0 rezultujućeg niza

Paralelna redukcija – main 1

```
#include <iostream>
#include <vector>

using std::cout;
using std::vector;

#define SHMEM_SIZE 256
```

```
int main()
{
  int N = 65536; //(2^16)
  size_t bytes = N * sizeof(int);
  vector<int> h_v(N);
  vector<int> h_v_r(N);

  for (auto i = 0; i < N; h_v[i++] = 1);

  int *d_v, *d_v_r;
  cudaMalloc(&d_v, bytes);
  cudaMalloc(&d_v_r, bytes);
  cudaMemcpy(d_v, h_v.data(), bytes, cudaMemcpyHostToDevice);

  const int TB_SIZE = 256;
  int GRID_SIZE = N / TB_SIZE;

  sumReduction<<<GRID_SIZE, TB_SIZE>>>(d_v, d_v_r);
  sumReduction<<<1, TB_SIZE>>> (d_v_r, d_v_r);

  cudaMemcpy(h_v_r.data(), d_v_r, bytes, cudaMemcpyDeviceToHost);

  cout << "SUM> " << h_v_r[0] << std::endl;

  return 0;
}
```

Paralelna redukcija – kernel 1

```
__global__ void sumReduction(int *v, int *v_r)
{
  __shared__ int partial_sum[SHMEM_SIZE];
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  partial_sum[threadIdx.x] = v[tid];
  __syncthreads();

  for (int s = 1; s < blockDim.x; s *= 2)
 {
    if (threadIdx.x % (2 * s) == 0)
      partial_sum[threadIdx.x] += partial_sum[threadIdx.x + s];

    __syncthreads();
  }

  // Nit 0 ce upisati rezultat u memoriju
  if (threadIdx.x == 0)
    v_r[blockIdx.x] = partial_sum[0];
}
```

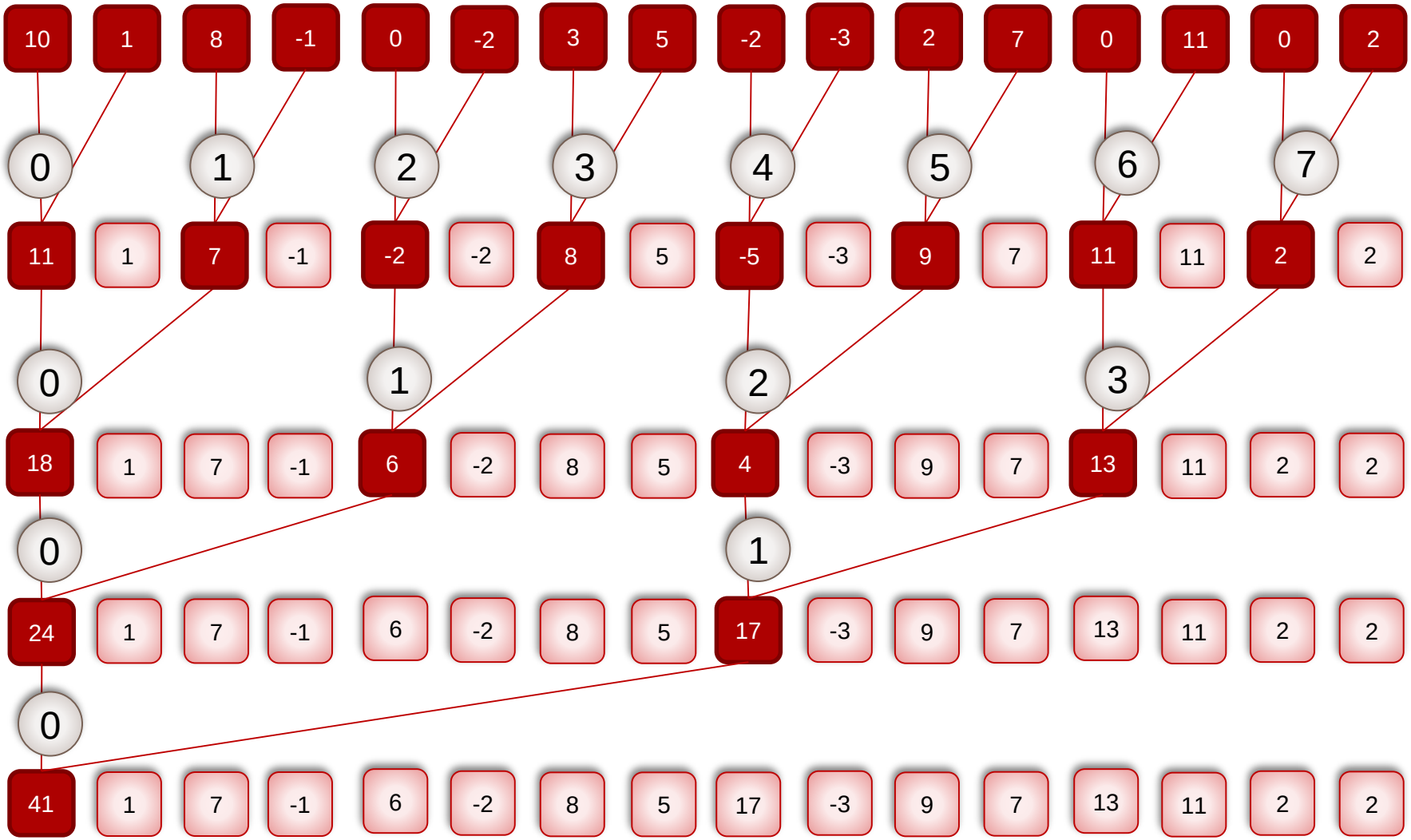
Paralelna redukcija 1

- Pola niti unutar istog warpa je aktivno, pola nije
  - Warp divergence
- Mod instrukcija unutar kernela

Rešenje:

- Drugačije indeksiranje unutar kernela
  - Nit ne pristupa uvek elementima koji odgovaraju njenom tid-u
- Aktivne niti imaju sukcesivne id-jeve
  - Smanjena divergentnost warpova
- Mod instrukcija zamenjena grananjem

Paralelna redukcija - 2



Paralelna redukcija – kernel 2

```
__global__ void sumReduction(int *v, int *v_r)
{
  __shared__ int partial_sum[SHMEM_SIZE];
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  partial_sum[threadIdx.x] = v[tid];
  __syncthreads();

  for (int s = 1; s < blockDim.x; s *= 2)
  {
    int index = 2 * s * threadIdx.x;

    if (index < blockDim.x)
      partial_sum[index] += partial_sum[index + s];

    __syncthreads();
  }

  if (threadIdx.x == 0)
    v_r[blockIdx.x] = partial_sum[0];
}
```

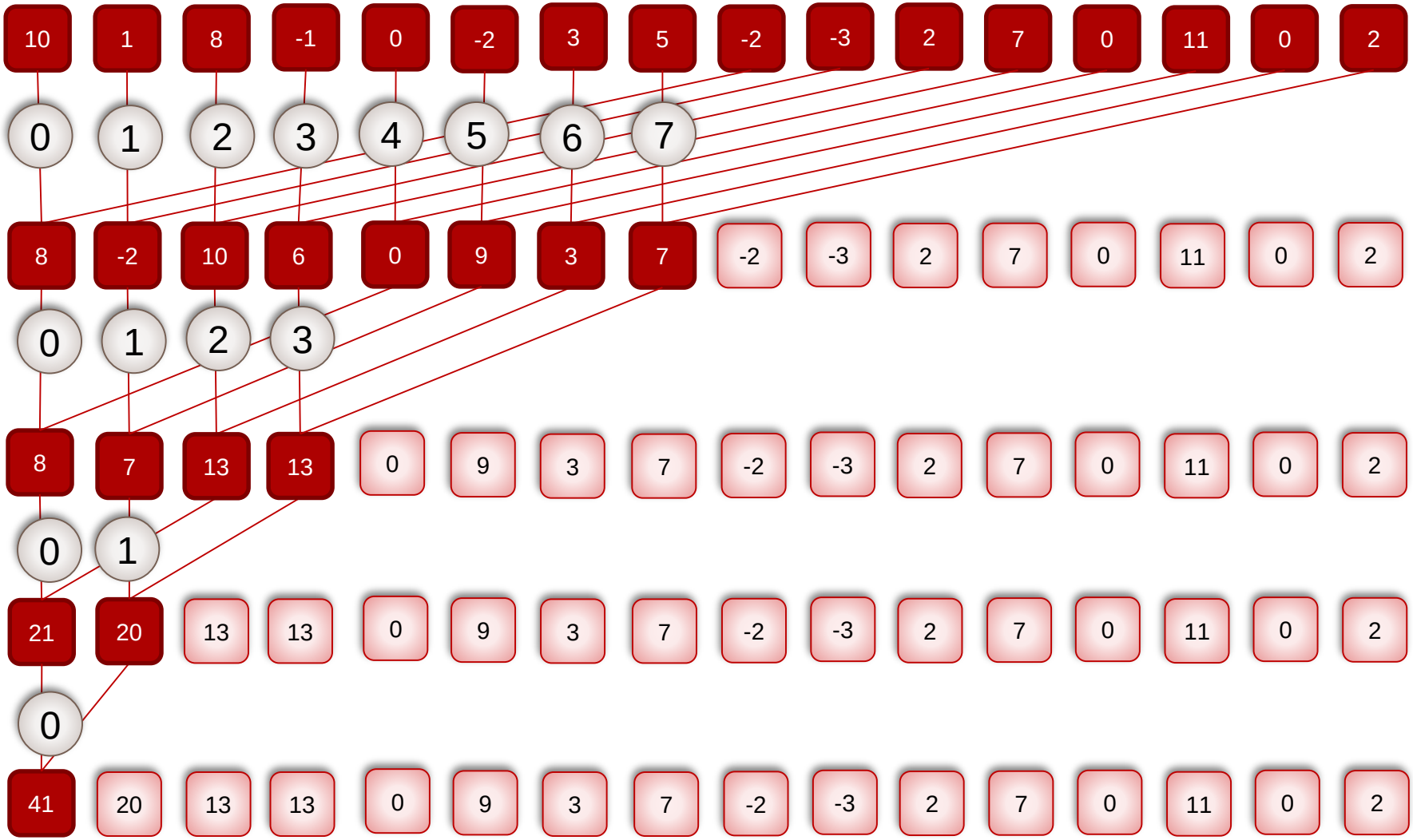
Paralelna redukcija 2

- Bank konflikti za deljenu memoriju

Rešenje:

- Susedne niti pristupaju susednim lokacijama
  - Stride je u svakoj iteraciji dvaput manji

Paralelna redukcija - 3



Paralelna redukcija – kernel 3

```
__global__ void sum_reduction(int *v, int *v_r)
{
  __shared__ int partial_sum[SHMEM_SIZE];
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  partial_sum[threadIdx.x] = v[tid];
  __syncthreads();

  for (int s = blockDim.x / 2; s > 0; s >>= 1)
 {
    if (threadIdx.x < s)
      partial_sum[threadIdx.x] += partial_sum[threadIdx.x + s];
    __syncthreads();
  }

  if (threadIdx.x == 0)
    v_r[blockIdx.x] = partial_sum[0];
}
```

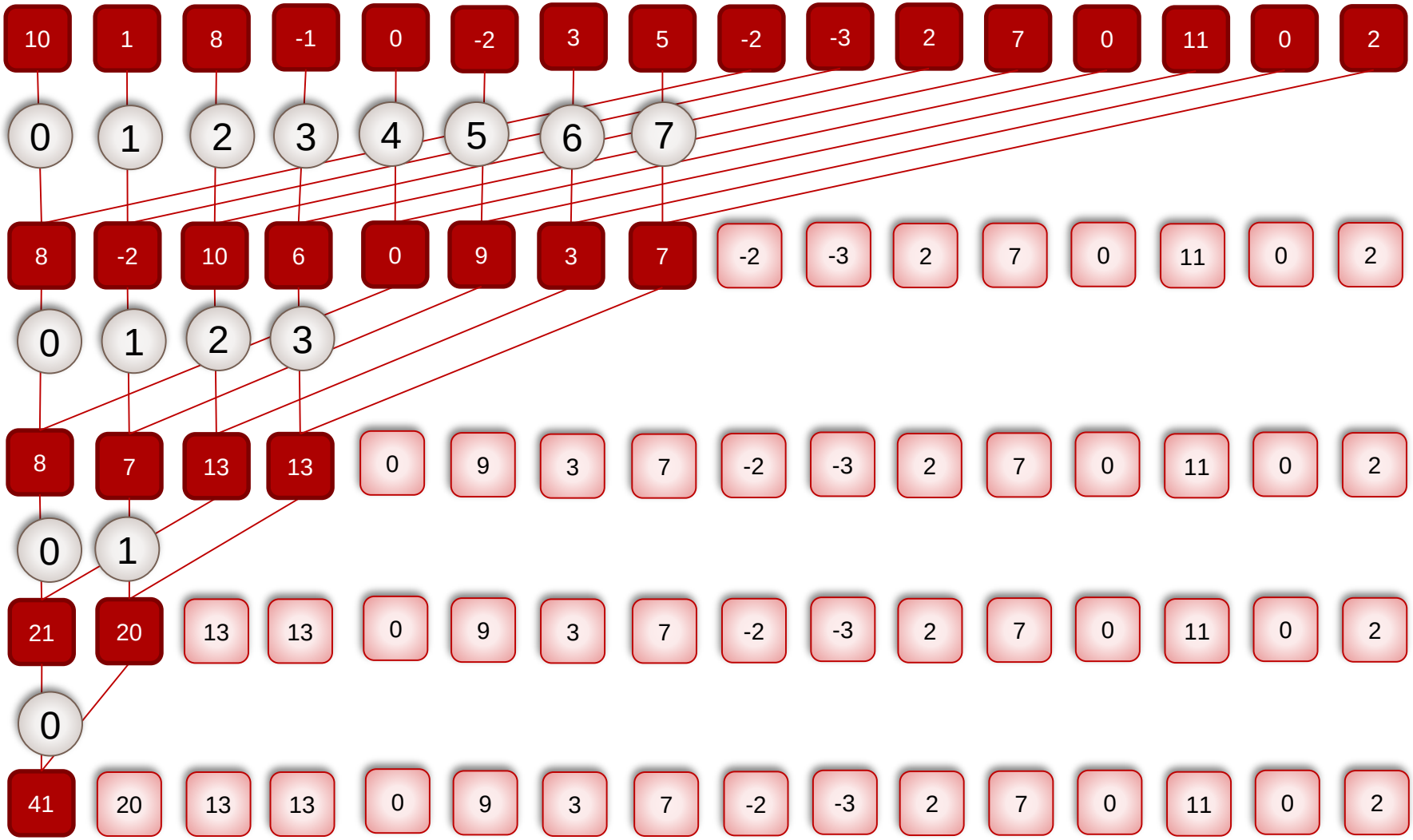
Paralelna redukcija 3

- Nepotrebno veliki broj niti
  - Neke niti samo učitavaju odgovarajući element ulaznog niza u deljenu memoriju

Rešenje:

- Prepoloviti broj blokova
- Zameniti učitavanje jednog elementa u deljenu memoriju sa dva učitavanja i jednom sumom

Paralelna redukcija - 4



Paralelna redukcija – main 4

```
#define SIZE 256
#define SHMEM_SIZE 256 * 4
```

```
int main()
{
  int N = 65536;
  size_t bytes = N * sizeof(int);
  vector<int> h_v(N);
  vector<int> h_v_r(N);
  int *d_v, *d_v_r;
  cudaMalloc(&d_v, bytes);
  cudaMalloc(&d_v_r, bytes);

  for (int i = 0; i < N; i++)
    h_v[i] = 1;//rand() % 10;

  cudaMemcpy(d_v, h_v.data(), bytes, cudaMemcpyHostToDevice);

  int TB_SIZE = SIZE;
  int GRID_SIZE = N / TB_SIZE / 2;

  sum_reduction << <GRID_SIZE, TB_SIZE >> > (d_v, d_v_r);
  sum_reduction << <1, TB_SIZE >> > (d_v_r, d_v_r);

  cudaMemcpy(h_v_r.data(), d_v_r, bytes, cudaMemcpyDeviceToHost);

  cout << "SUM> " << h_v_r[0] << std::endl;
  return 0;
}
```

Paralelna redukcija – kernel 4

```
__global__ void sum_reduction(int *v, int *v_r)
 {
  __shared__ int partial_sum[SHMEM_SIZE];
  int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;

  partial_sum[threadIdx.x] = v[i] + v[i + blockDim.x];
  __syncthreads();

  for (int s = blockDim.x / 2; s > 0; s >>= 1)
  {
    if (threadIdx.x < s)
      partial_sum[threadIdx.x] += partial_sum[threadIdx.x + s];

    __syncthreads();
  }

  if (threadIdx.x == 0)
    v_r[blockIdx.x] = partial_sum[0];
}
```

???