



Esercizi su CUDA

Algoritmi e Calcolo Parallelo

- Implementare in cuda il prodotto scalare fra due array di float

$$c = \vec{a} \cdot \vec{b} = \sum_i a_i b_i$$

- ▶ a e b sono array di uguali dimensioni e con N elementi (N costante definita opportunamente o letta dall'utente)
- ▶ introdotto un vettore v come segue:

$$\vec{v} = \{a_1 b_1, a_2 b_2, \dots, a_n b_n\}$$

- ▶ abbiamo

$$c = \vec{a} \cdot \vec{b} = \sum_i v_i$$

```
__global__ void dotProdGPUv0(float *v, float *a,  
                             float *b, int N)  
{  
    // global index  
    int idx = threadIdx.x + blockIdx.x * blockDim.x;  
  
    // perform computation  
    if (idx < N)  
        v[idx] = a[idx] * b[idx];  
}
```

Come calcolare c a partire da v ?

In maniera seriale sull'host

Banale

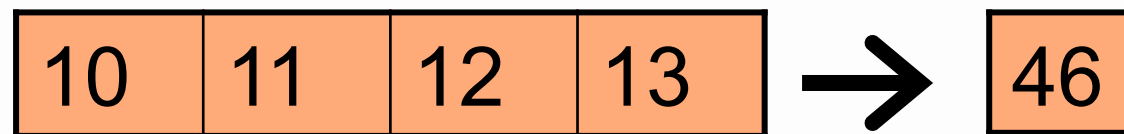
Richiede un il trasferimento di v

Sfruttando il parallelismo sulla GPU

Non è gestito automaticamente da CUDA

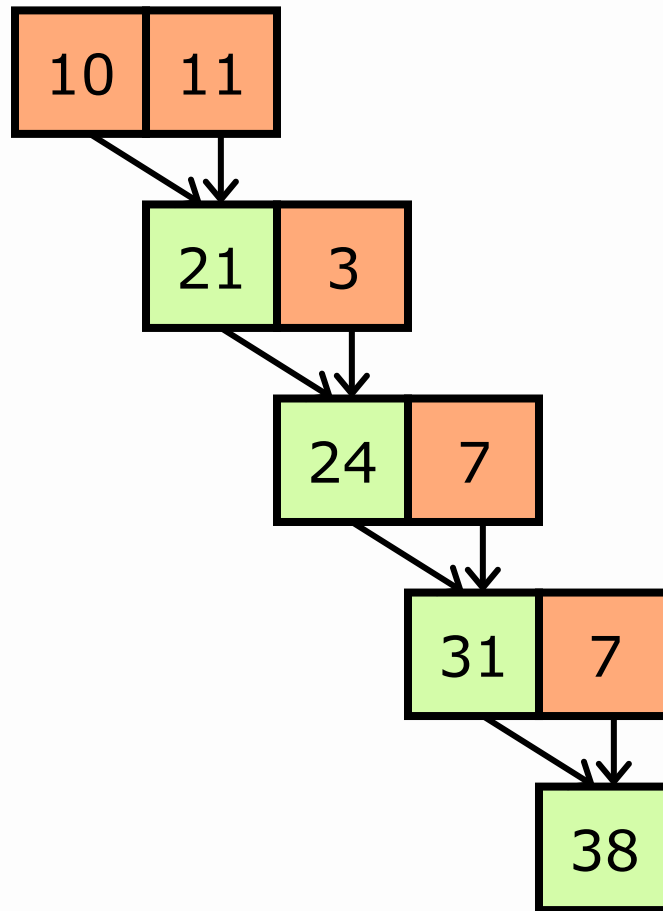
Evita trasferimenti di grandi quantità di dati

- ❑ Nel calcolo parallelo, capita frequentemente di ridurre più valori, calcolati in parallelo, in un singolo valore
- ❑ Possibili operazioni di riduzione sono:
 - ▶ ADD
 - ▶ MUL
 - ▶ AND
 - ▶ OR
- ❑ Ad esempio, dovendo sommare gli elementi di un vettore...



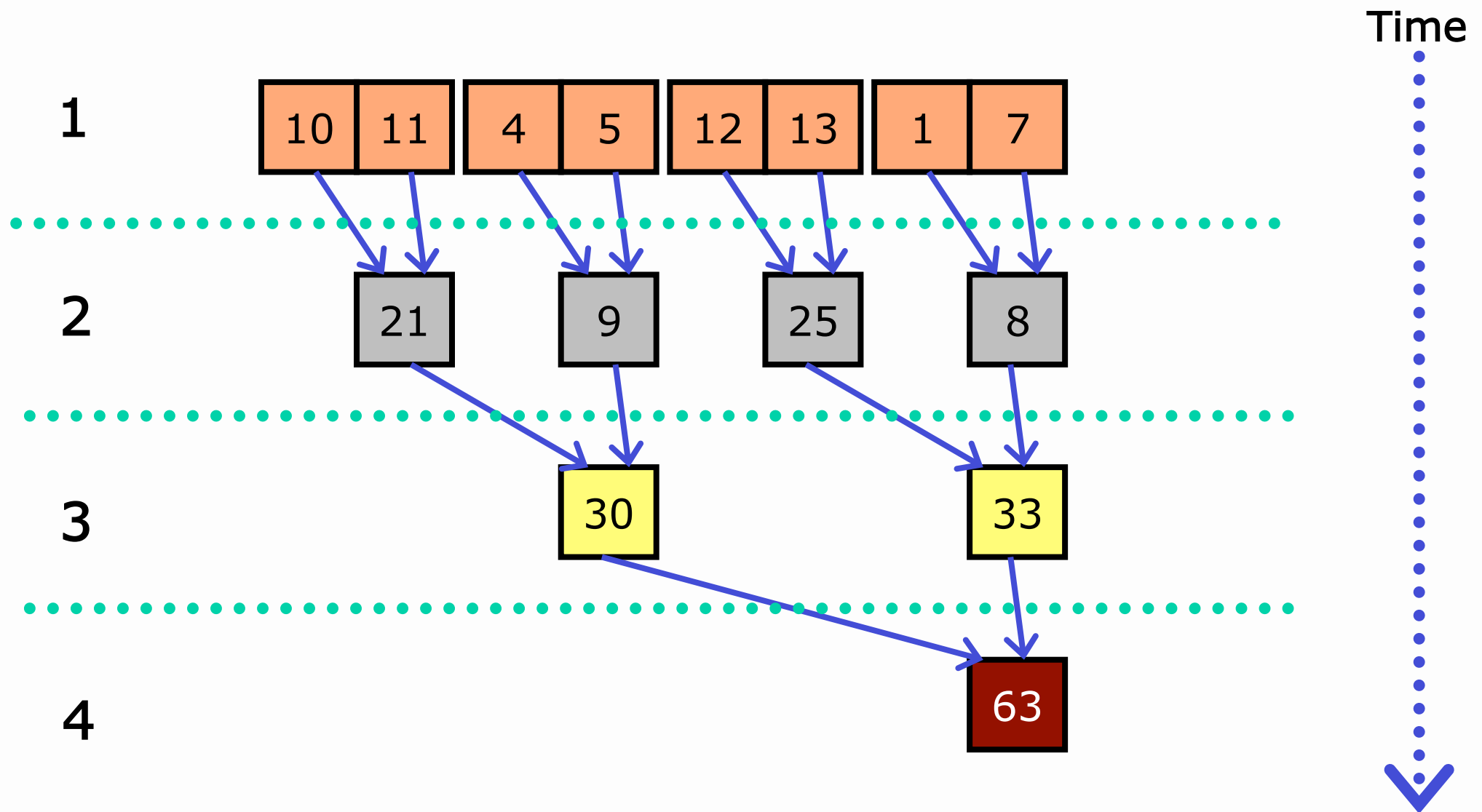
- ❑ ... analizziamo i passi che vengono effettuati e come questi possono essere parallelizzati

Riduzione sequenziale



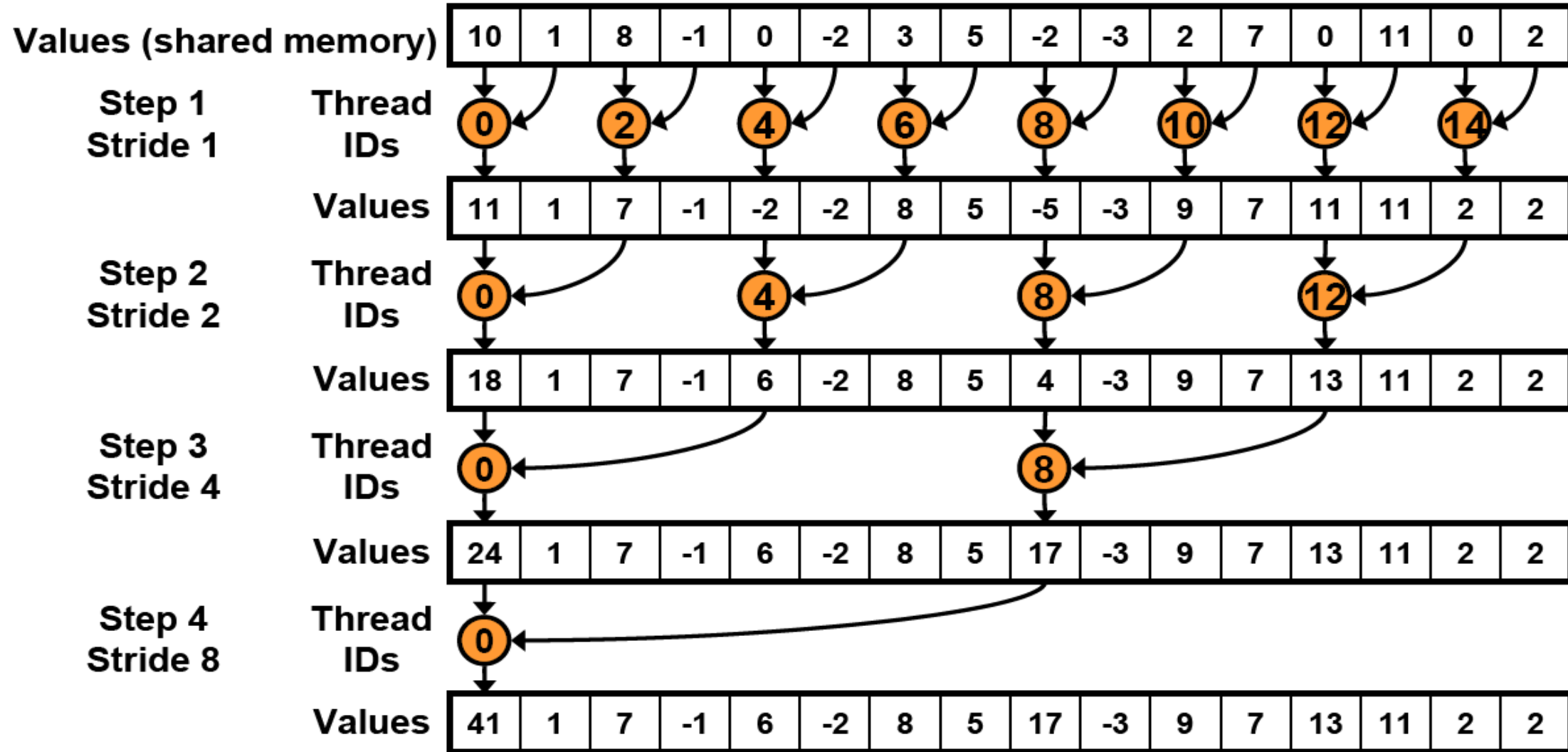
$O(N)$

Parallel Reduction



$O(\log N)$

Schema di riduzione in CUDA: Interleaved Access



Interleaved Accesses: Codice

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to 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=1; s < blockDim.x; s *= 2) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step
            // participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

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

Interleaved Accesses: Problem

```
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

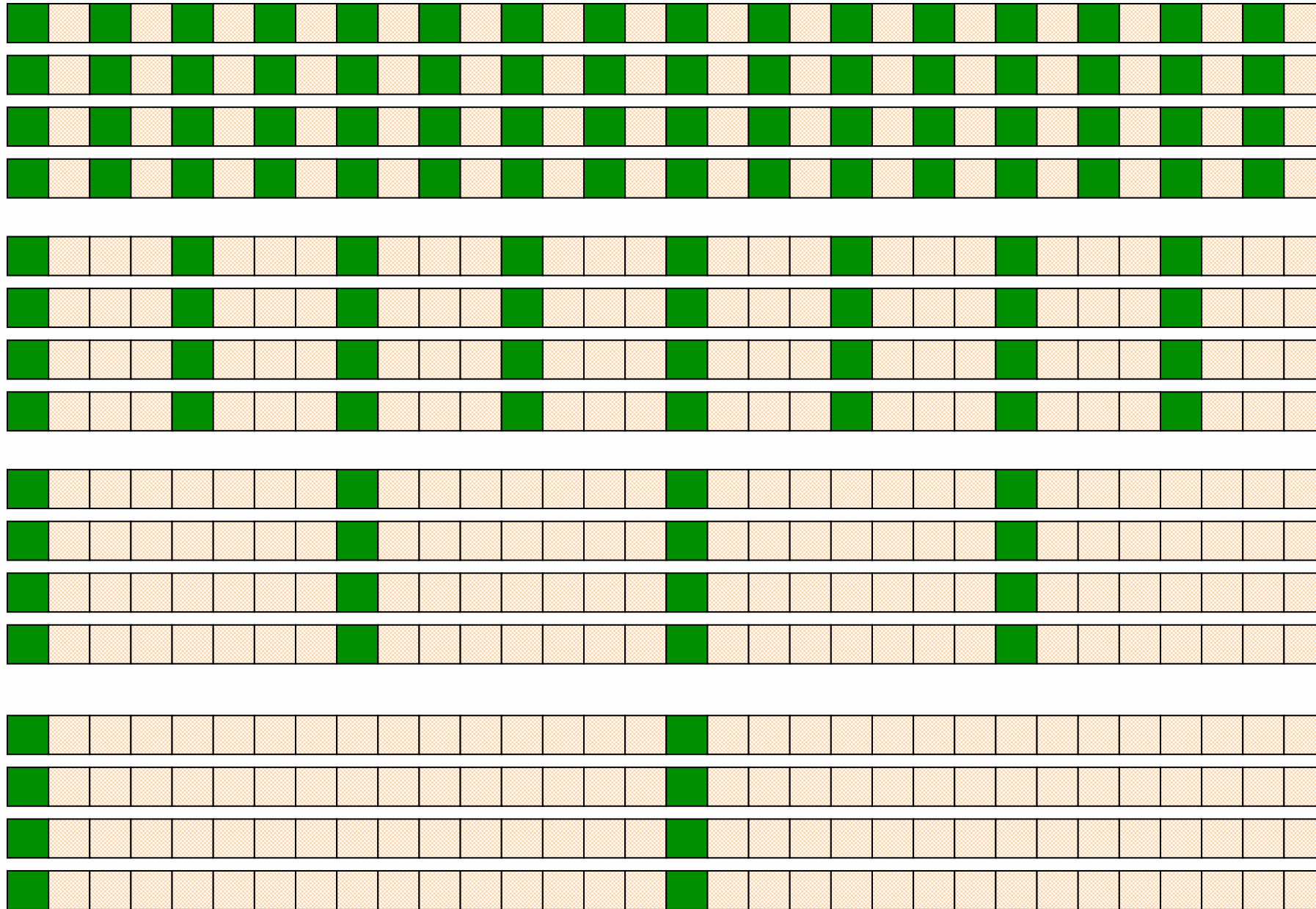
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step
            // participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

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

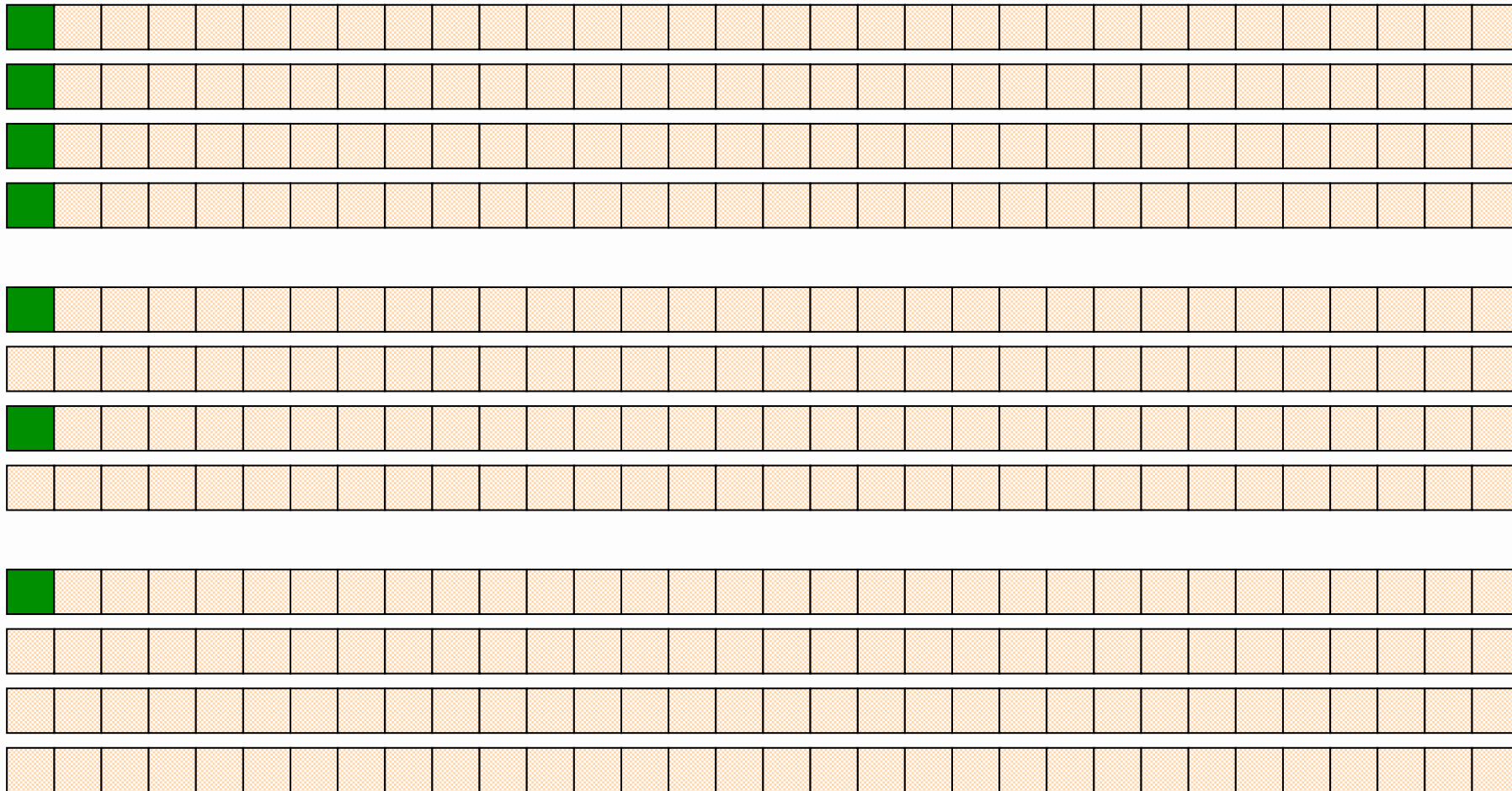
**Highly divergent code
leads to very poor
performance**

Thread Attivi



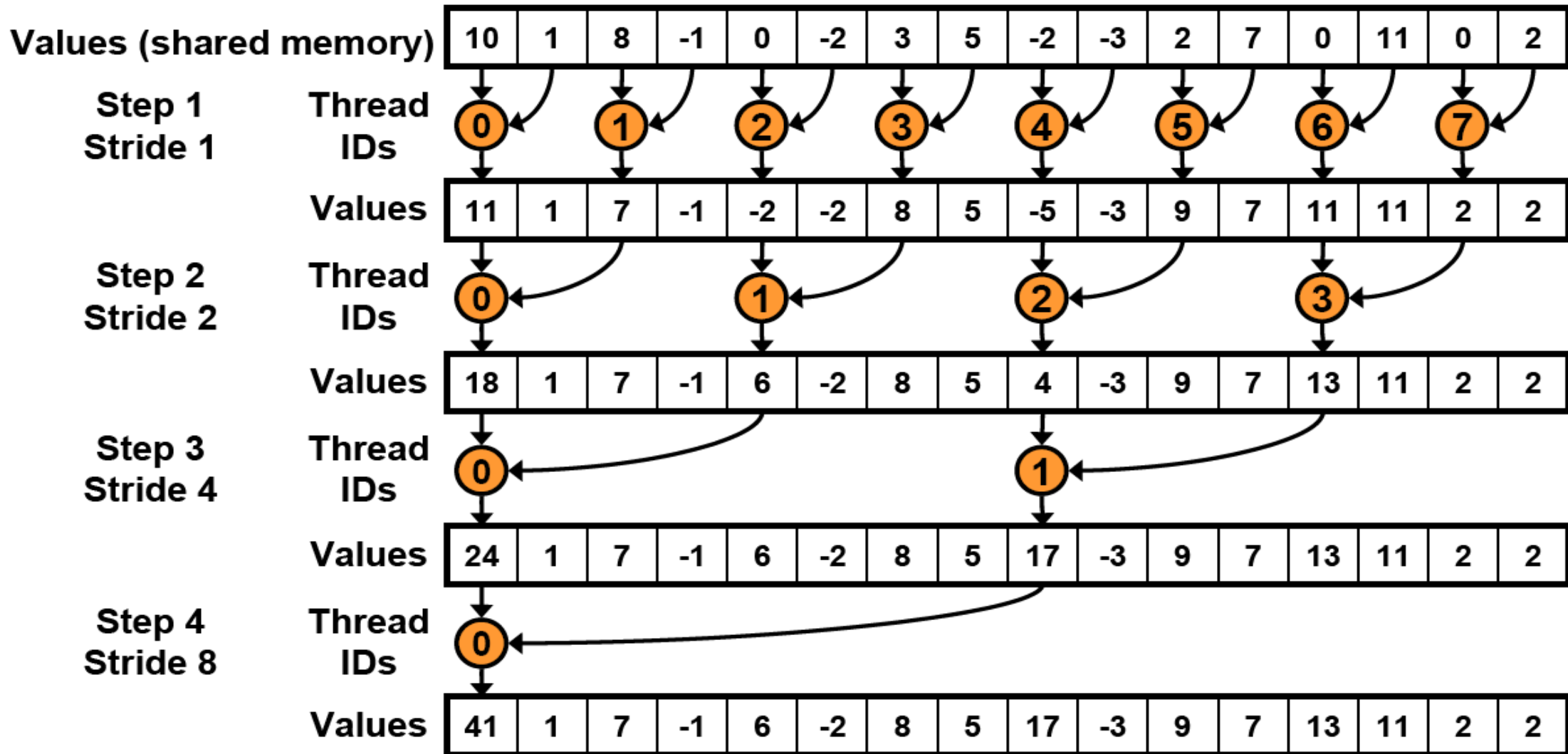
STEPS

Threads Attivi

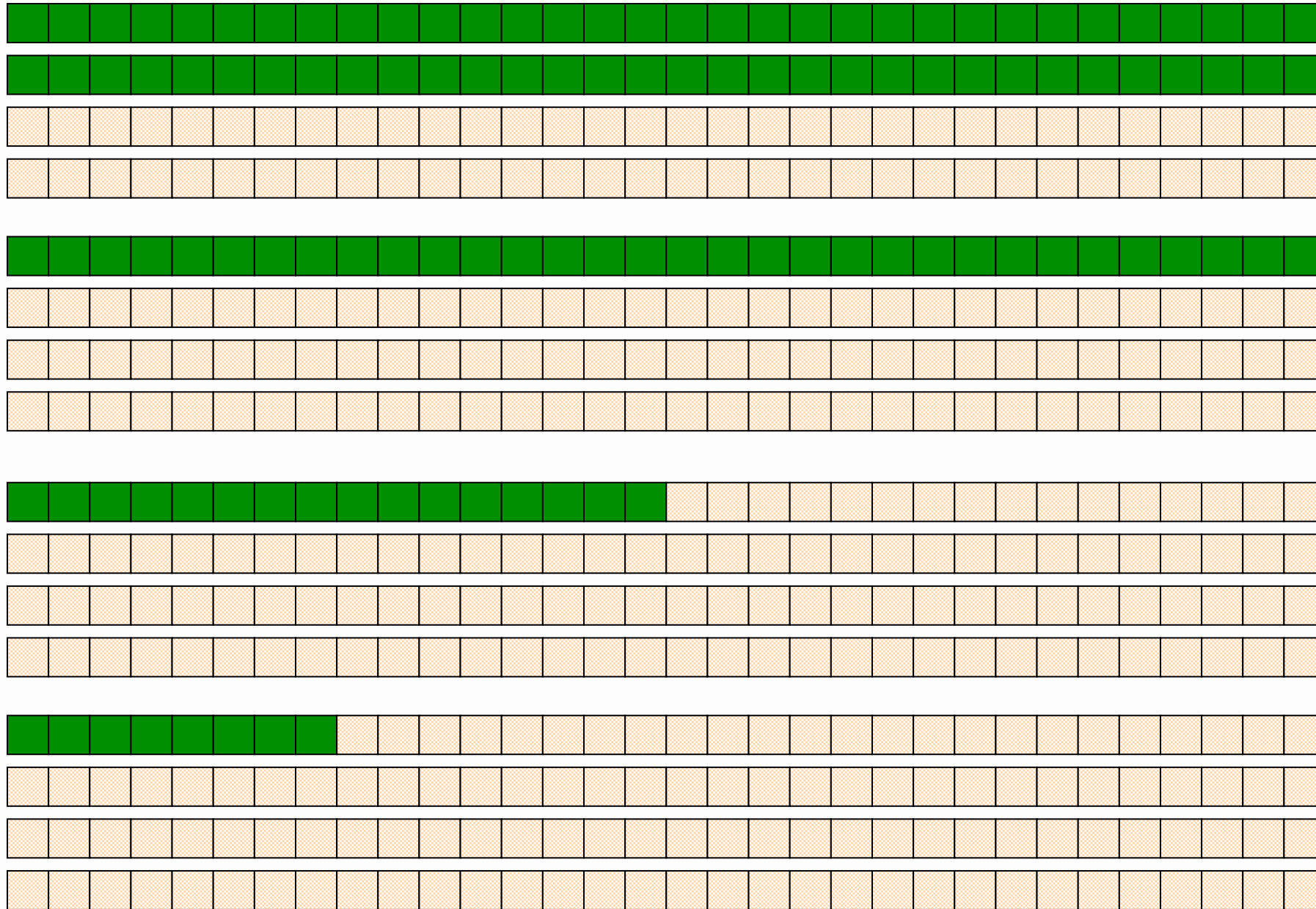


STEPS

Reduction #2: Access Pattern



Thread Attivi



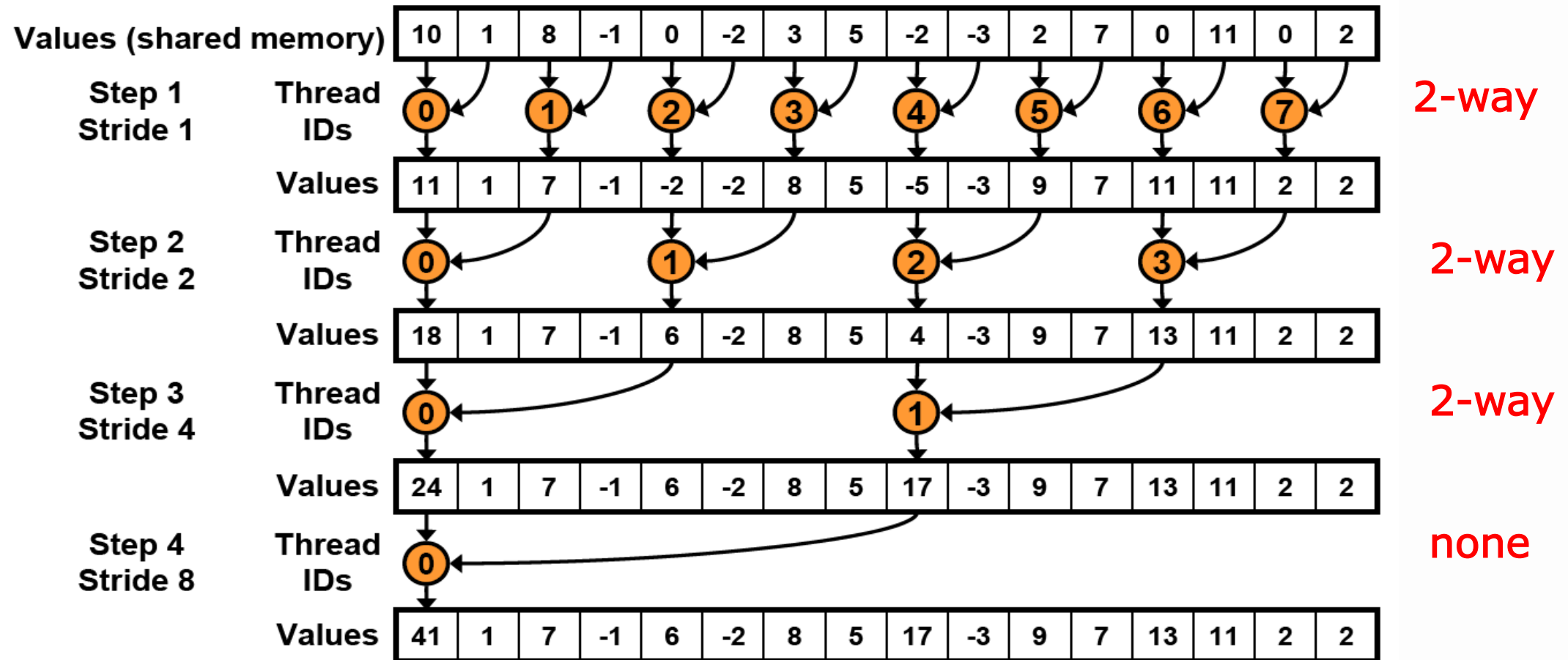
STEPS

Thread Attivi



STEPS

Reduction #2: Problem

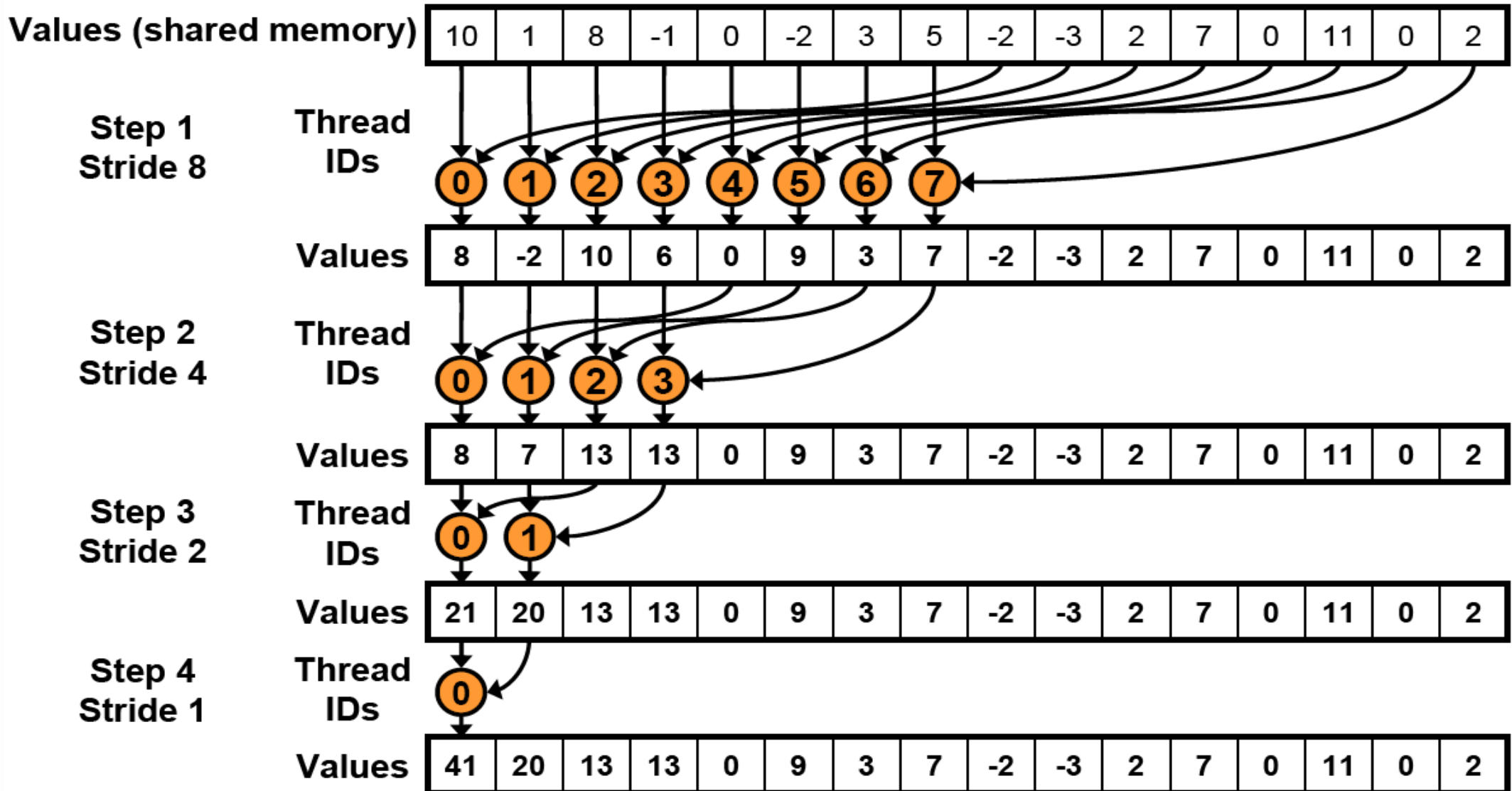


2-way bank conflicts at every step

Recall there are more than 16 threads

To see the conflicts see what happens with 128 threads

Reduction #3: Sequential Accesses



❑ Eliminates bank conflicts

Sequential Access- Codice

```
__global__ void reduce3(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to 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 /= 2) { // step = s x 2
        if (tid < s) { // only threadIDs divisible by the step
            // participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

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

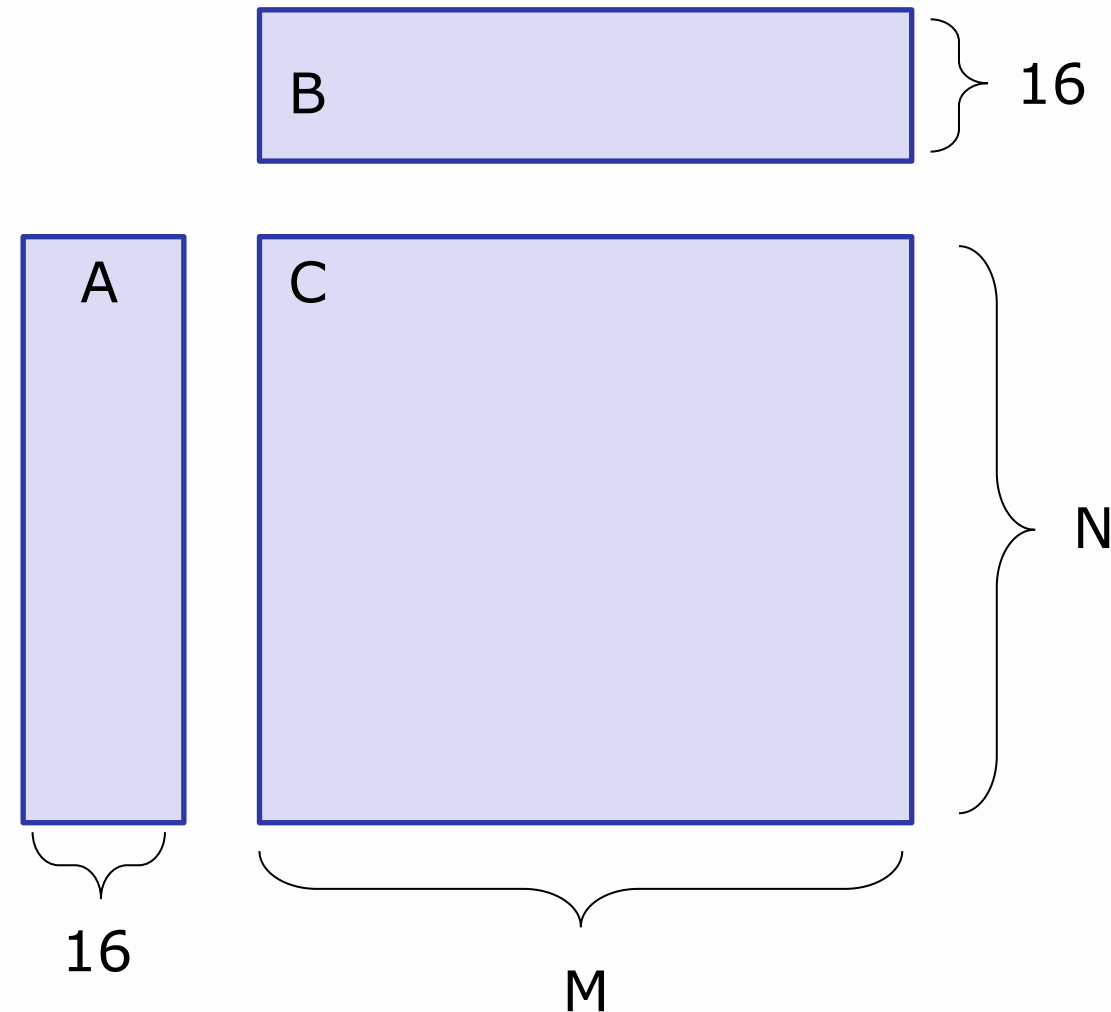
```
__global__ void dotProdGPU3(float *c, float *a, float *b, int N){
    // global index
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    // index within block
    int tid = threadIdx.x;
    // shared memory for reduction
    extern __shared__ float s_data[];
    // perform computation
    if (idx < N)
        s_data[tid] = a[idx] * b[idx];
    __syncthreads();
    // do reduction in shared mem
    for(unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s)
            s_data[tid] += s_data[tid + s];
        __syncthreads(); }
    // write result for this block to global mem
    if (tid == 0)
        c[blockIdx.x] = s_data[0];
}
```

Prodotto fra matrici

- Implementare in CUDA il prodotto fra due matrici

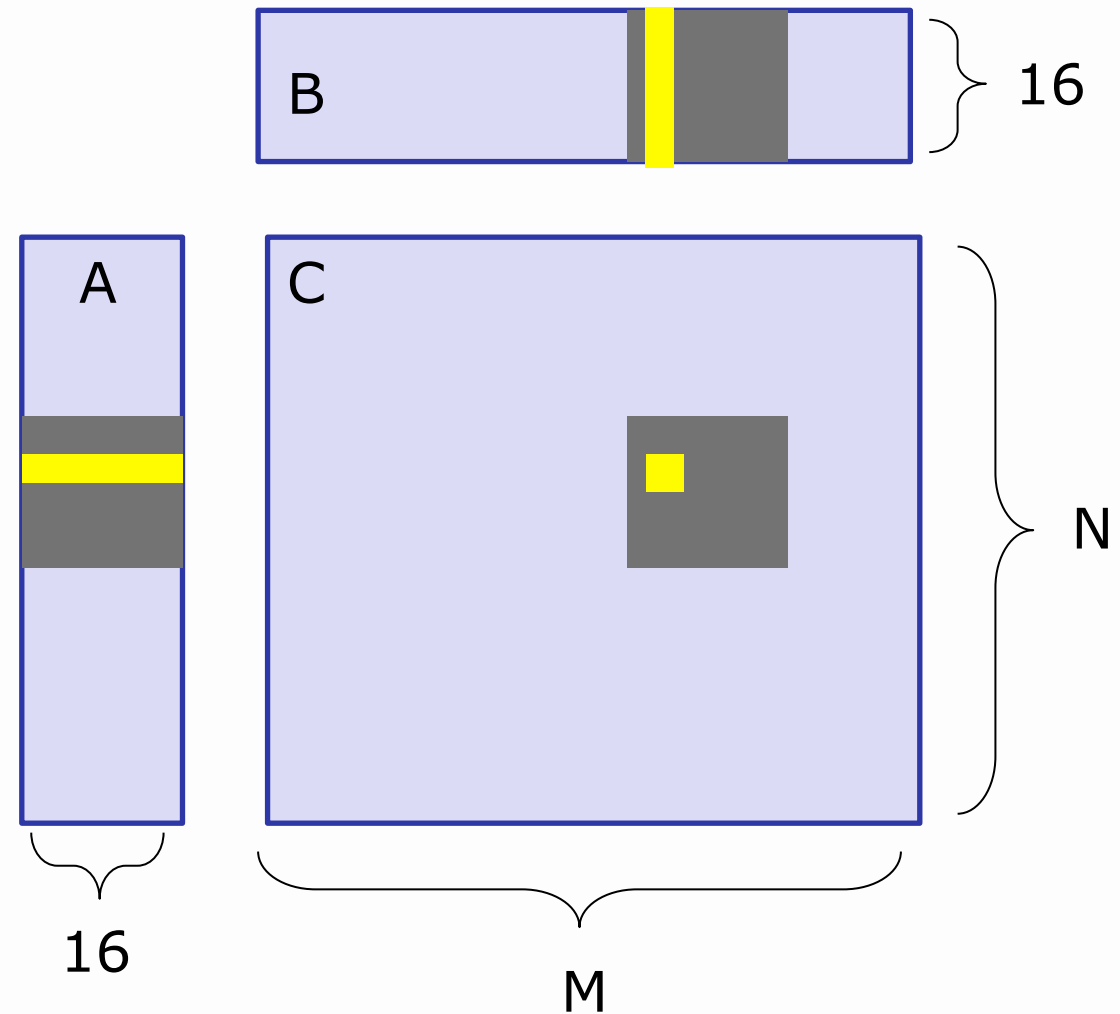
$$C = AB$$

- ▶ A è una matrice $N \times 16$
- ▶ B è una matrice $16 \times M$
- ▶ M ed N sono multipli interi di 16



Soluzione

- ❑ Le matrici A e B vengono divise in tile 16×16
- ❑ Ogni tile viene assegnata ad un blocco di thread
- ❑ Ogni thread calcola un elemento della matrice C

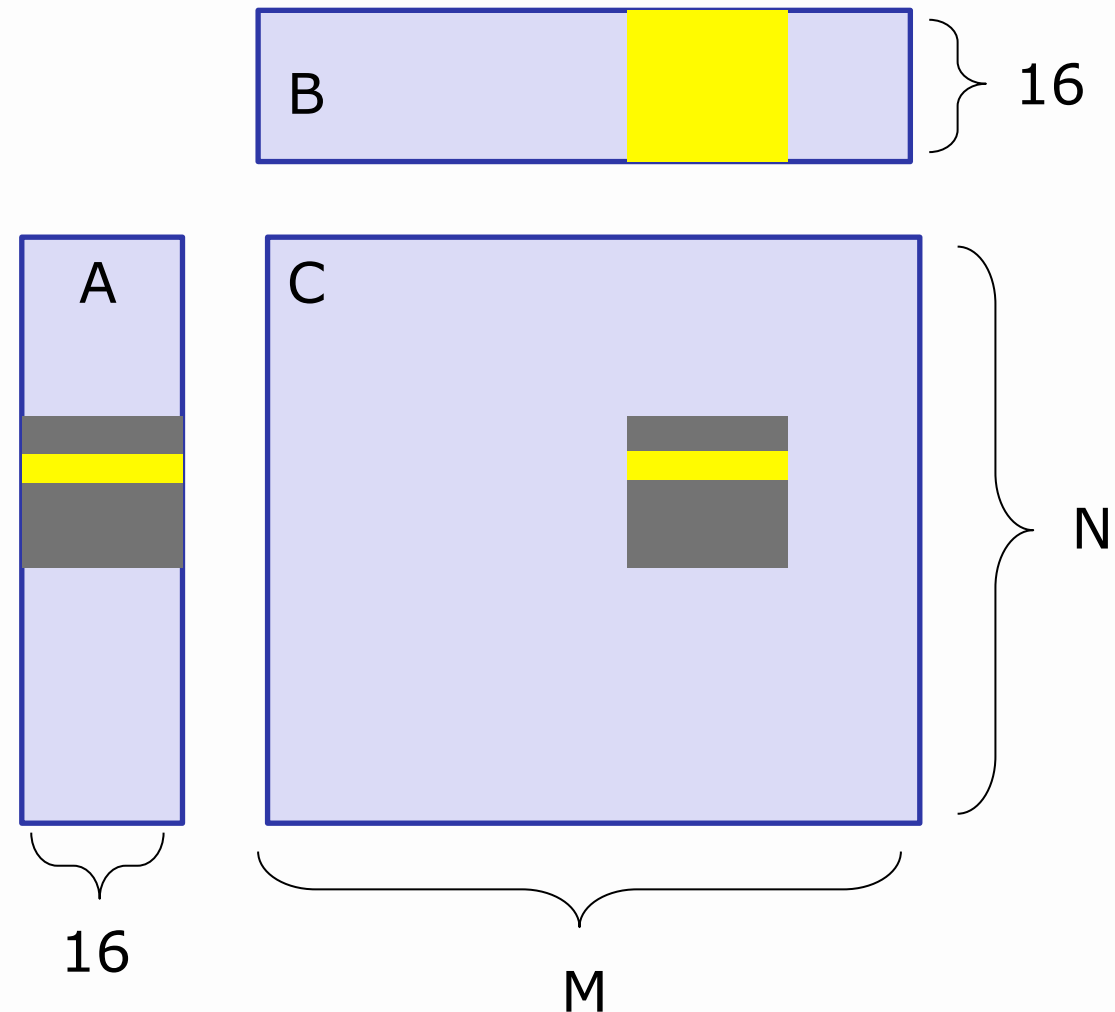


Soluzione: implementazione

```
__global__ void simpleMultiply(float *a, float* b,
                               float *c, int N){
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Soluzione: ottimizzazione

- Un half-warp calcola una riga della matrice C
 - ▶ ogni thread legge la stessa riga di A
 - 16 transazione su schede < 1.2
 - 1 transazione su schede ≥ 1.2
 - ▶ ad ogni iterazione, la lettura dello stesso elemento di A richiede
 - la lettura di A avviene più volte inutilmente
- L'uso della memoria shared può migliorare le prestazioni



Soluzione: implementazione (2)

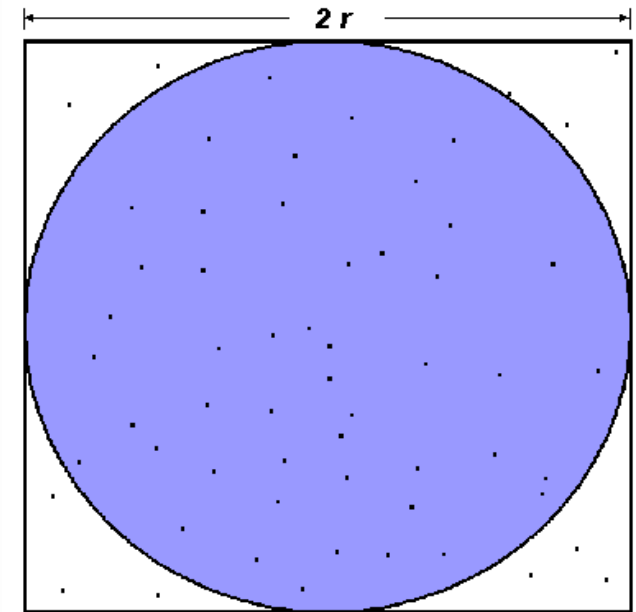
```
__global__ void coalescedMultiply(float *a, float*
                                   b, float *c, int N) {
    __shared__ float aTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] =
        a[row*TILE_DIM+threadIdx.x];
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i] * b[i*N+col]; }
    c[row*N+col] = sum;
}
```


Soluzione: implementazione (3)

```
__global__ void sharedABMultiply(float *a, float* b,
                                float *c, int N) {
    __shared__ float aTile[TILE_DIM][TILE_DIM],
                   bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] =
        a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] =
        b[threadIdx.y*N+col];
    __syncthreads();
    for (int i = 0; i < TILE_DIM; i++) {
        sum+=aTile[threadIdx.y][i]*bTile[i][threadIdx.x];
    }
    c[row*N+col] = sum;
}
```

- Implementare in CUDA una soluzione parallela del seguente algoritmo per approssimare PI

```
cin >> npoints; count = 0;
for(j=0, j<npoints; j++) {
    x = random();
    y = random();
    if (inCircle(x, y))
        count++;
}
PI = 4.0*count/npoints
```



$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4 \times \frac{A_C}{A_S}$$

- ▶ A che livello parallelizzereste?
- ▶ Pensare ad una implementazione “furba” di `inCircle(x, y)`
 - **Nota:** è possibile riformulare il problema

- ❑ Due possibili approcci
 - ▶ Generare i numeri random con la CPU e trasferirli nella memoria della GPU
 - ▶ Generare i numeri random tramite la GPU
- ❑ Per la generazione random su GPU è possibile usare la libreria **CURAND**
 - ▶ Manuale disponibile qui: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CURAND_Library.pdf
 - ▶ Consente la generazione di numeri random sia dall'host sia dal device

CURAND Host API: overview

1. Creazione di un generatore con `curandCreateGenerator()`
2. Definizione del seed del generatore con `curandSetPseudoRandomGeneratorSeed()` e/o altre opzioni
3. Allocare memoria sul device (`cudaMalloc()`)
4. Generazione numeri random con `curandGenerate()` o altre funzioni
5. Esecuzione di uno o più kernel che richiedono i numeri random generati
6. Deallocazione risorse tramite `curandDestroyGenerator()`

CURAND Host API: funzioni generazione

- ❑ `curandStatus_t curandGenerate(
 curandGenerator_t generator,
 unsigned int *outputPtr,
 size_t num)`
- ❑ `curandStatus_t CurandGenerateUniform(
 curandGenerator_t generator,
 float *outputPtr,
 size_t num)`
- ❑ `curandStatus_t curandGenerateNormal(
 curandGenerator_t generator,
 float *outputPtr,
 size_t n,
 float mean,
 float stddev)`

- ❑ Analoghe funzioni per generare numeri double

CURAND Host API: esempio

```
#include <curand.h>

...

size_t n = 100;
curandGenerator_t gen;
float *devData, *hostData;

/* Allocate n floats on host */
hostData = (float *)calloc(n, sizeof(float));

/* Allocate n floats on device */
cudaMalloc((void **)&devData, n * sizeof(float));
```

CURAND Host API: esempio (2)

```
...
/* Create pseudo-random number generator */
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
/* Set seed */
curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
/* Generate n floats on device */
curandGenerateUniform(gen, devData, n);
/* Use devData on device */
...
/* Cleanup */
curandDestroyGenerator(gen);
cudaFree(devData);
free(hostData);
...
```