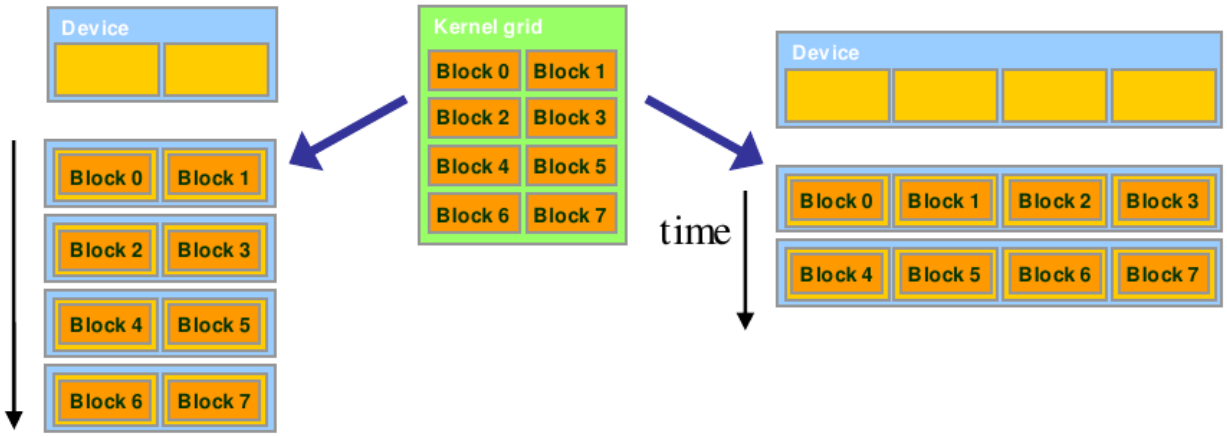
CUDA: organizzazione dei threads

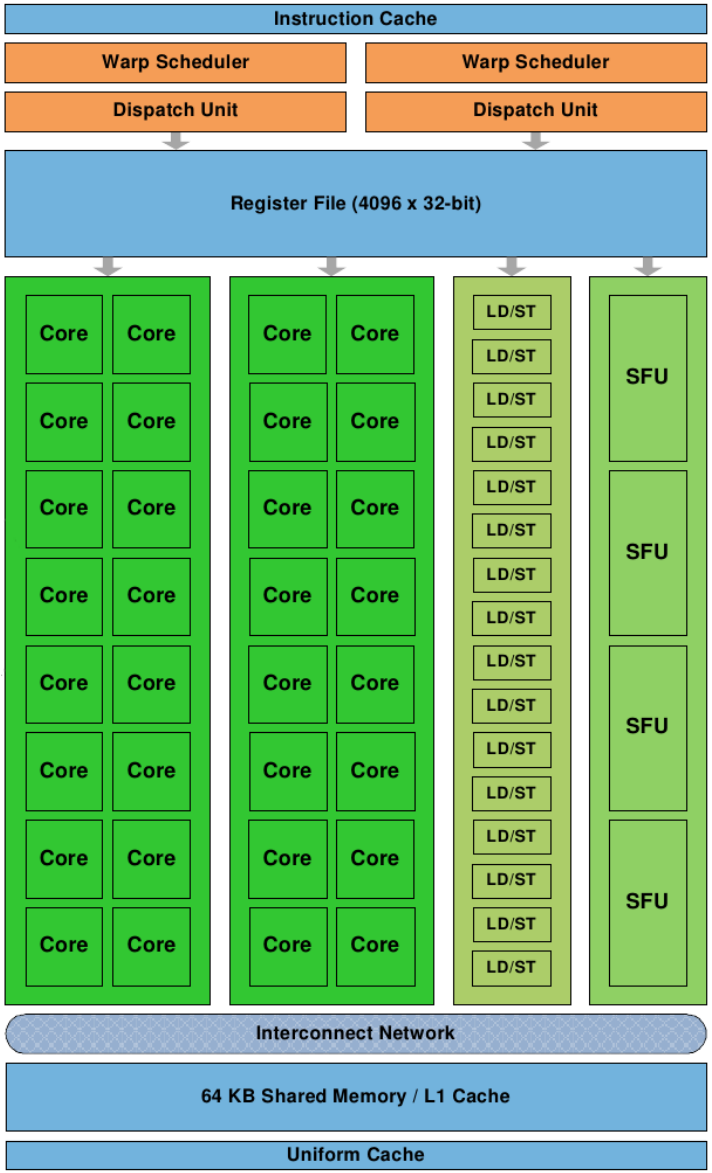
Argomenti

Thread-block e SM

- Un blocco di thread è assegnato ad un unico streaming multiprocessor (SM): questo significa gestire un gruppo di threads come una unità logica, fornendo ai threads dello stesso blocco la possibilità di utilizzare direttive di sincronizzazione (`__syncthreads()`) e memoria condivisa (`__shared__`).
- La scelta di non fornire direttive di sincronizzazione globali (tra threads di diversi blocchi) permette di gestire i blocchi di threads in modo indipendente: l'ordine di esecuzione dei blocchi di threads non è fissato a priori.
- Questa flessibilità di esecuzione permette di sviluppare implementazioni altamente scalabili su diverse architetture:



Scheduling dei threads (I)



Scheduling dei threads (II)

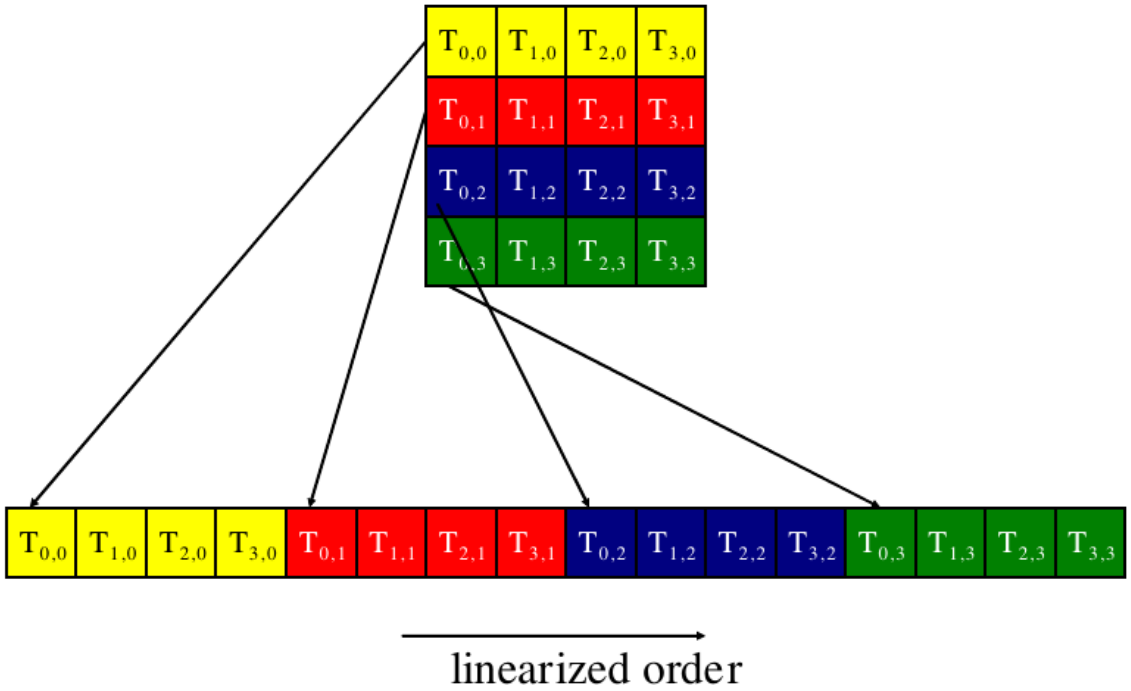
- Lo scheduling dei threads è direttamente legato alla particolare architettura della GPU e il suo intrinseco parallelismo.
- Un blocco di threads viene assegnato ad un singolo SM: i threads vengono ulteriormente suddivisi in gruppi, detti warp.
- La dimensione del warp dipende dall'architettura sotto esame, per esempio il chip GT200 utilizza warp di 32 threads.
- I thread appartenenti allo stesso warp vengono gestiti dall'unità di controllo (warp scheduler) in modo congiunto.
- Per sfruttare al massimo il parallelismo intrinseco del SM, i thread dello stesso warp devono eseguire la stessa istruzione: se questa condizione non si verifica si parla di divergenza dei threads.
- Se i thread dello stesso warp stanno eseguendo istruzioni diverse l'unità di controllo non può gestire tutto il warp: deve seguire le successioni di istruzioni per ogni singolo thread (o per sottoinsiemi omogenei di threads) in modo seriale.

Argomenti

Suddivisione del blocco di thread

Osserviamo come viene diviso il blocco di thread in vari warp:

- I thread vengono suddivisi in base al valore del threadIdx.
- La struttura threadIdx è composta di 3 campi: i vari threadIdx appartenenti a diversi thread sono ordinati osservando threadIdx.z threadIdx.y threadIdx.x.

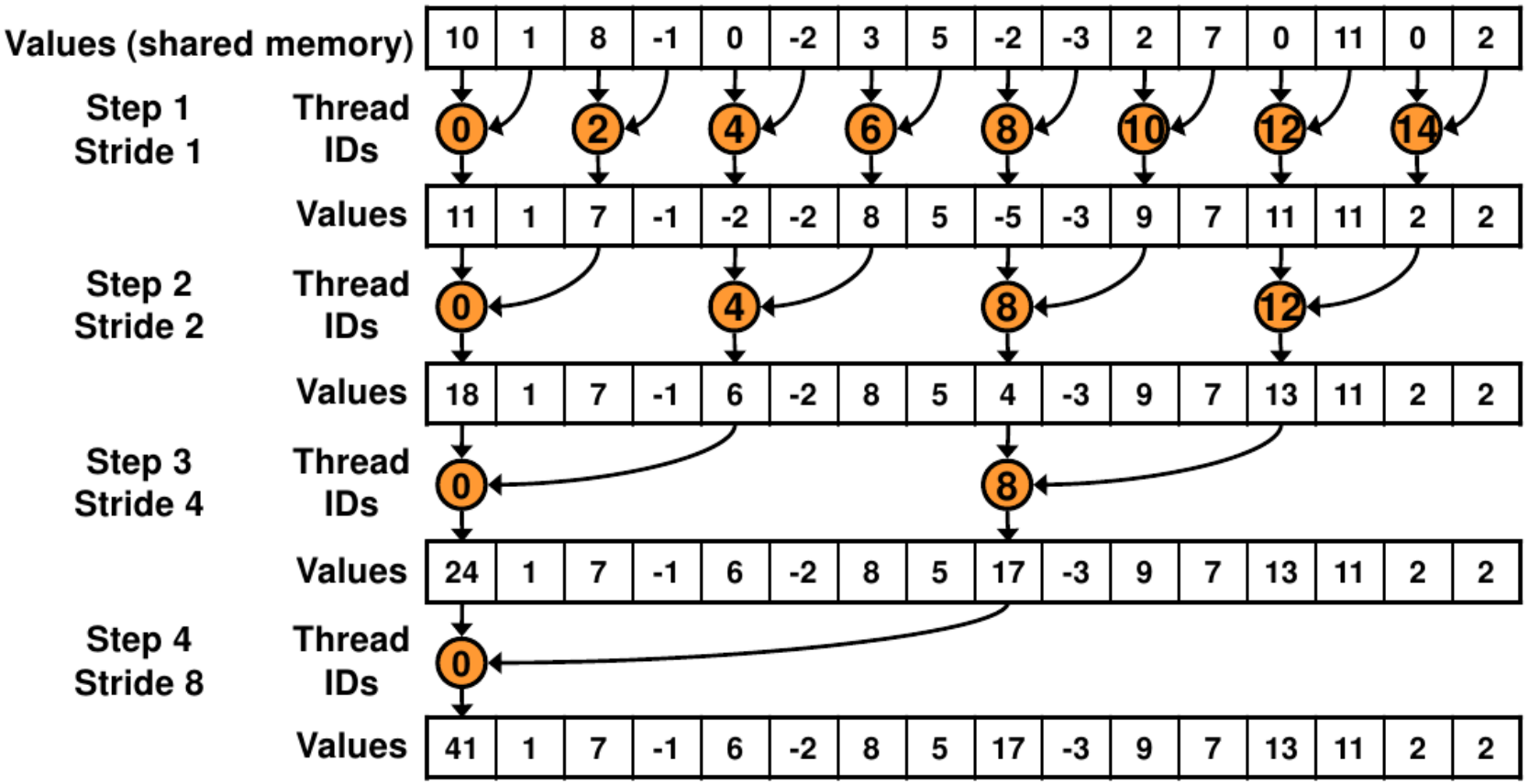


In figura: T(x,y), con x = threadIdx.x e y = threadIdx.y.

warp e divergenza

- Dopo aver linearizzato i valori di threadIdx, il vettore ottenuto viene suddiviso in gruppi di 32 threads ottenendo i warp.
- L'hardware esegue una istruzione per ogni thread nel warp, prima di passare alla istruzione successiva: questo stile di esecuzione viene chiamato Single Instruction Multiple Thread (SIMT).
- L'approccio funziona bene quando il thread segue un flusso di operazioni identico agli altri componenti il warp.
- Ad esempio: se nel kernel che abbiamo scritto è presente un costrutto di tipo `if-then-else`, l'esecuzione dei thread nel warp rimane parallela solo se tutti i thread eseguono il blocco `then` oppure il blocco `else`. Viceversa, se una parte di thread esegue il blocco `then` e i rimanenti eseguono il blocco `else`, l'approccio SIMT non porta più una esecuzione parallela. Lo scheduler è costretto a seguire solo un gruppo di thread (es. quelli che eseguono `then`) per poi considerare i rimenenti thread che eseguono `else`.

Esempio di Reduction



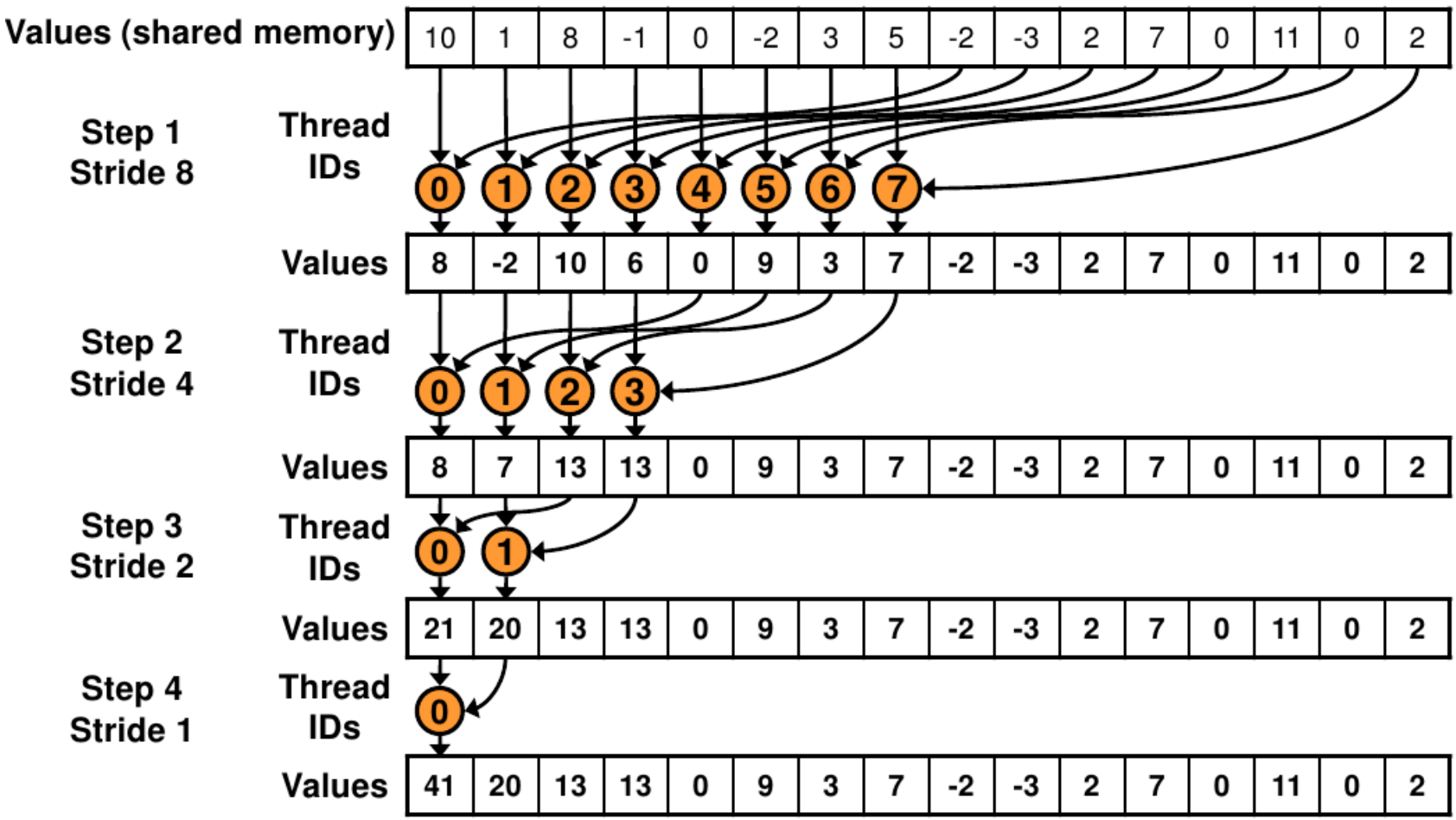
Reduction

kernel

```
__global__ void my_reduce_0 (double *idata, double *odata, unsigned int n) {
__shared__ double sdata[TILEWIDTH];

  unsigned int stride, idx = threadIdx.x;
  unsigned int ii  = (blockIdx.x * blockDim.x) + threadIdx.x;
  sdata[idx] = (ii < n) ? idata[ii] : 0.0;

  __syncthreads();

  for (stride = 1; stride < blockDim.x; stride <<= 1) {

    if (idx % (2*stride) == 0) sdata[idx] += sdata[idx+stride];

    __syncthreads();

  }
  if(idx == 0) odata[blockIdx.x] = sdata[0];
}
```

- I warp divergono già dalla prima iterazione del ciclo for: i thread vengono divisi in base a threadIdx.x pari o dispari
- Modifichiamo il ciclo for...

Reduction 1


Values (shared memory)
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1
Stride 8
Thread IDs
0 1 2 3 4 5 6 7
Values
8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 2
Stride 4
Thread IDs
0 1 2 3
Values
8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 3
Stride 2
Thread IDs
0 1
Values
21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 4
Stride 1
Thread IDs
0
Values
41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Reduction 1: kernel

kernel

```
__global__ void my_reduce_1 (double *idata, double *odata, unsigned int n) {
__shared__ double sdata[TILEWIDTH];

  unsigned int stride, idx = threadIdx.x;
  unsigned int ii = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[idx] = (ii < n) ? idata[ii] : 0.0;

  __syncthreads();


  for (stride = blockDim.x/2; stride > 0; stride >>= 1) {

    if (idx < stride ) sdata[idx] += sdata[idx+stride];

    __syncthreads();

  }
  if(idx == 0) odata[blockIdx.x] = sdata[0];
}
```

- I warp non divergono
- Uso metà degli effettivi thread

Argomenti

Eventi in CUDA (I)

Il costo di esecuzione di una porzione di codice CUDA (es. un kernel) può essere misurata, in termini di tempo, utilizzando gli eventi.
Per creare un evento, si utilizza la funzione:

```
cudaError_t cudaEventCreate (cudaEvent_t * event)
```

Per distruggere un evento si utilizza:

```
  cudaError_t cudaEventDestroy (cudaEvent_t event)
```

Per registrare un evento:

```
cudaError_t cudaEventRecord (cudaEvent_t event,
                             cudaStream_t stream = 0)
```

Eventi in CUDA (II)

```
cudaEvent_t start,stop;

cudaEventRecord(start, 0);

   /* Codice CUDA (es. lancio di un kernel) */

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Nella variabile `elapsedTime` troviamo il tempo trascorso, espresso in millisecondi.