GPU Computing

Lab 3

Warp divergency

Inefficienza nei branch

Esempio: divergenza nei warp

Warp divergence:
16 thread su branch (a = 2)
16 thread su branch (b = 1)

```
/*
 * Kernel con divergenza dei warp
 */
__global__ void evenOdd_DIV(int *c) {
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        int a = 0, b = 0;

        if (tid % 2 == 0)
           a = 2;       // thread pari
        else
           b = 1;       // thread dispari
        c[tid] = a + b;
}
```

Soluzione:

- ragionare in termini di warp!
- dimenticare l'indicizzazione diretta thread-dato (del vettore)
- creare una nuova indicizzazione che tenga conto dei warp
- usare la var builtin **warpSize (=32)**

```
. . .
int wid = tid / warpSize; // warp index wid = 0,1,2,3,...
if (!(wid % 2))
  . . .
```

Esempio: eliminazione della divergenza

APPROCCIO: usare granularità dei warp e non quella dei thread!
Tutto si svolge a livello di BLOCCO!

wid è l'indice di warp all'interno del blocco

Uso i warp a coppie (64 thread) per calcolare l'indice **i** del vettore c:

- Il primo warp calcola gli indici pari **o,2,4,…,62**
- Il secondo warp calcola gli indici dispari **1,3,5,..,63**
- Il pari e dispari della coppia si ha con **wid%2**
- La formula che rende questo è **2*(tid%warpSize)**
- L'offset viene calcolato modulo 64 (2 warpSize): **(tid/64)*64**

```
/*
 * Kernel con divergenza dei warp risolta (solo se N > 64)
 */
__global__ void evenOdd_NO_DIV(int *c, int N) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int a = 0, b = 0, i;

  int wid = tid / warpSize; // indice dei warp wid = 0,1,2,3,...
  if (!(wid % 2))
    a = 2;          // Branch1: thread tid = 0-31, 64-95,...
  else
    b = 1;          // Branch2: thread tid = 32-63, 96-127,...

      // right index
      if (!(wid%2))  // even
         i = 2*(tid%32)  + (tid/64)*64;
      else           // odd
          i = 2*(tid%32)+1 + (tid/64)*64;

   if (i<N)
     c[i] = a + b;
}
```

Tempi di esecuzione

Esecuzione su GPU T4

```
Data size: 2,000,000,000 -- Data size (bytes): 8000 MB
Execution conf (block 1024, grid 1953125) Kernels:
 - evenOddDIV<<<1953125, 1024>>>   elapsed time 0.033627 sec
 - evenOddNODIV<<<1953125, 1024>>> elapsed time 0.000062 sec
```

Kernel DIV

Kernel NO DIV

Esecuzione su GPU H100

```
Data size: 2000000000 -- Data size (bytes): 8000 MB
Execution conf (block 1024, grid 1953125) Kernels:
 - evenOddDIV<<<1953125, 1024>>>   elapsed time 0.026766 sec
 - evenOddNODIV<<<1953125, 1024>>> elapsed time 0.000065 sec
```

Kernel DIV

Kernel NO DIV

Parallel reduction

Somma parallela

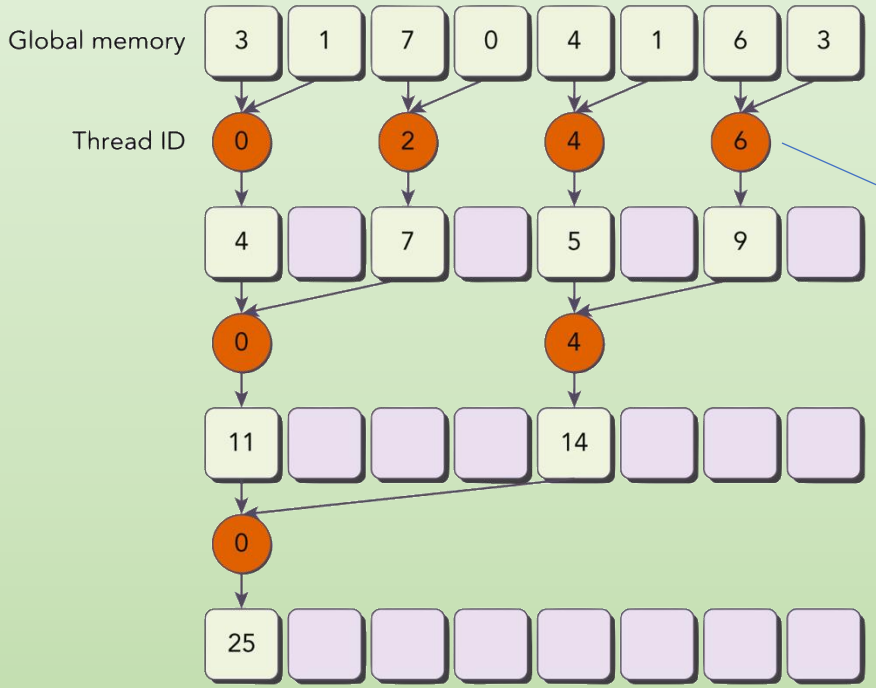
Somma parallela

Progettare un kernel per la somma di elementi di vettori di grandi dimensioni:

- Usare lo schema che suddivide in blocchi (richiesta sincronizzazione)
- Sommare su ogni blocco con parallel reduction (somma parziale)
- Utilizzare uno schema interlacciato
- Evitare la divergenza nella parallel reduction
- Unire le somme parziali dei blocchi

Divergenza in parallel reduction

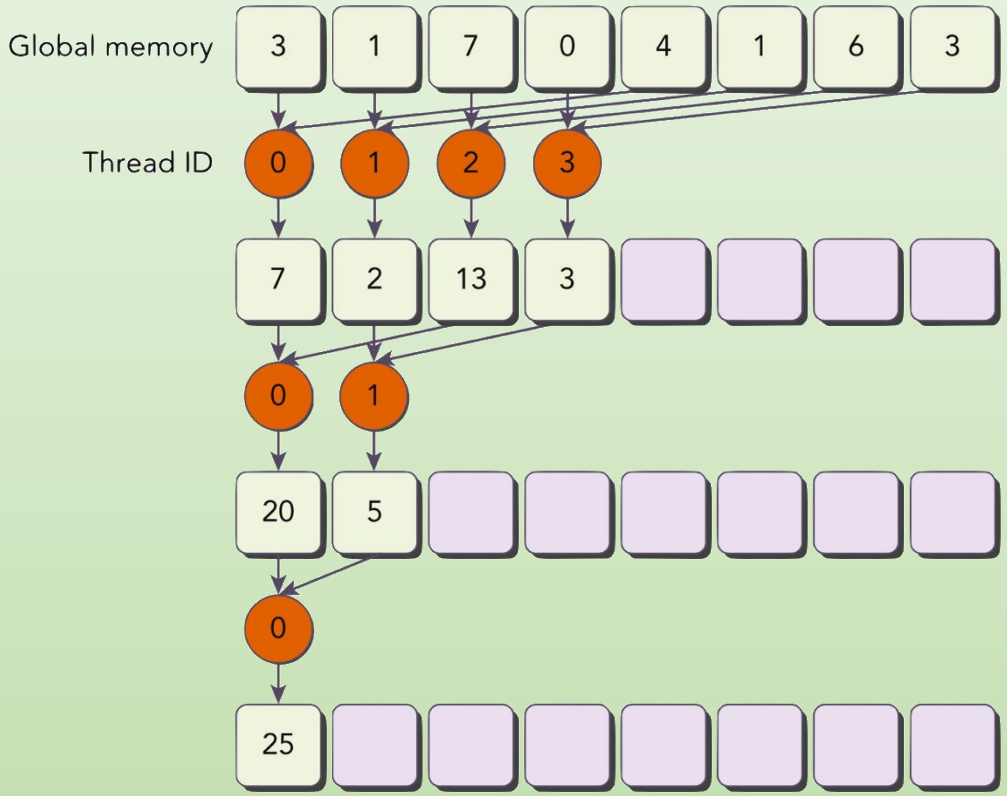
Schema inefficiente x troppa divergenza




```
__global__ void blockParReduce1(int *in, int *out, ulong n) {
   uint tid = threadIdx.x;
   ulong idx = blockIdx.x * blockDim.x + threadIdx.x;

   // boundary check
   if (idx >= n)
      return;

   // convert global data pointer to the local pointer of this block
   int *thisBlock = in + blockIdx.x * blockDim.x;

   // convert global data pointer to the local pointer of this block
   int *thisBlock = in + blockIdx.x * blockDim.x;

   // in-place reduction in global memory
   for (int stride = 1; stride < blockDim.x; stride *= 2) {
      if ((tid % (2 * stride)) == 0)
         thisBlock[tid] += thisBlock[tid + stride];

      // synchronize within threadblock
      __syncthreads();
   }

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

Eliminazione della divergenza

Schema efficiente perché senza divergenza… efficiente anche accesso in memoria!

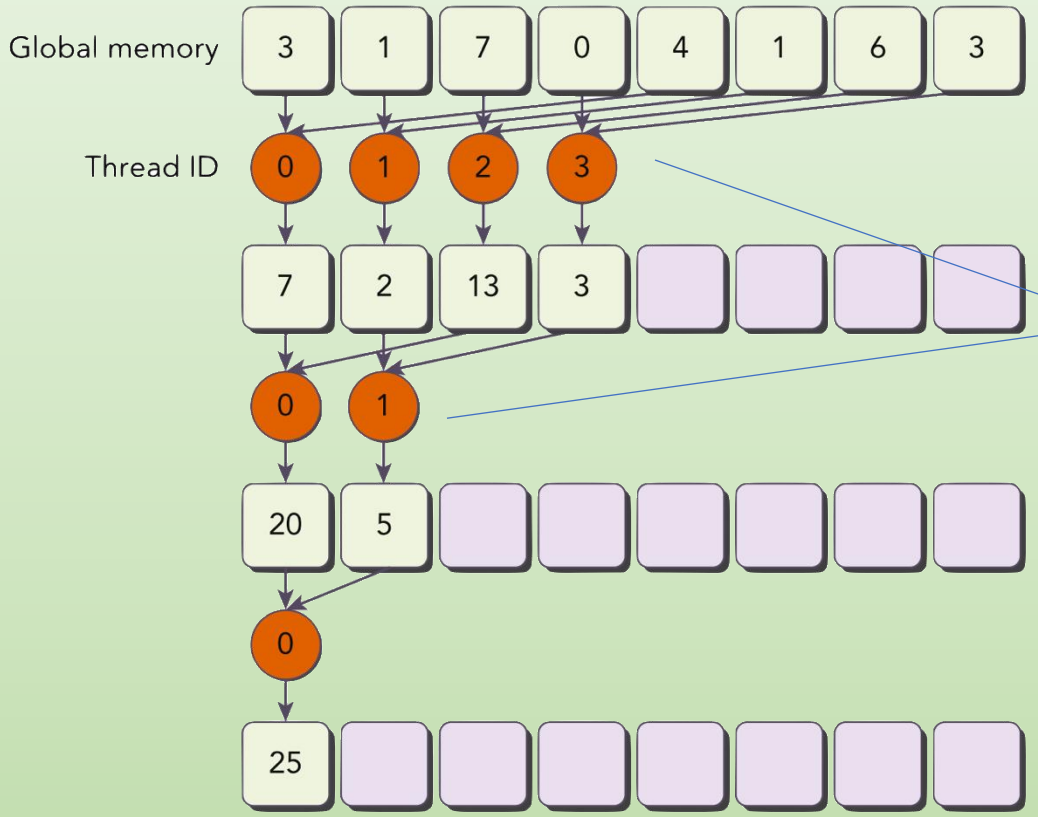



```
__global__ void blockParReduce2([args]*) {

   uint tid = threadIdx.x;
   ulong idx = blockIdx.x * blockDim.x + threadIdx.x;

   // TODO

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

Esecuzione su Pascal P100

```
****  test on parallel reduction  ****
 Vector length: 3072.00 MB  sum: 3221225472

 CPU procedure...
  Elapsed time: 1.578515 (sec)

 GPU kernels (mem required 12884901888 bytes)

 Launch kernel: blockParReduce1...
  Elapsed time: 0.395506 (sec) - speedup 4.0

 Launch kernel: blockParReduce2...
  Elapsed time: 0.154076 (sec) - speedup 10.2
```

Eliminazione della divergenza




```
__global__ void blockParReduce2(int *in, int *out, ulong n) {

   uint tid = threadIdx.x;
   ulong idx = blockIdx.x * blockDim.x + threadIdx.x;

   // boundary check
   if (idx >= n)
      return;

   // convert global data ptr to the local ptr of this block
   int *thisBlock = in + blockIdx.x * blockDim.x;

   // in-place reduction in global memory
   for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
      if (tid < stride)
         thisBlock[tid] += thisBlock[tid + stride];

       // synchronize within threadblock
       __syncthreads();
   }

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

Istogramma di un’immagine

Operazioni atomiche

Istogramma (C API)

```c
/*
* Istogramma RGB dell'immagine PPM
*/
int *ppm_histogram(PPM *ppm) {
   int *histogram = (int *)malloc(3* 256 * sizeof(int));
   // initialize histogram
   for (int x = 0; x < 3 * 256; x++)
      histogram[x] = 0;

   // count the number of pixels for each color
   for (int x = 0; x < ppm->width * ppm->height; x++) {
      histogram[ppm->image[3*x]]++;
      histogram[ppm->image[3*x+1] + 256]++;
      histogram[ppm->image[3*x+2] + 512]++;
  }
  return histogram;
}
```