

# 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)

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)**

```
/*
 * 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;
}

. . .

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 **0,2,4,...,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 Pascal P100

```
Data size 100.000.000

Execution Configure (block 1024 grid 976563)
Kernels:
    evenOdd_DIV<<<976563, 1024>>> elapsed time 0.006727 sec
    evenOdd_NO_DIV<<<976563, 1024>>> elapsed time 0.000175 sec
```

- ✓ Esecuzione su GPU Tesla M2090 (lagrange)

```
Data size 10.000.000

Execution Configure (block 256 grid 39063)
Kernels:
    evenOdd_DIV<<<39063, 256>>> elapsed time 0.000344 sec
    evenOdd_NO_DIV<<<39063, 256>>> elapsed time 0.000054 sec
```

# Branch efficiency

- ✓ Misurare l'efficienza in termini di **warp divergence**:

```
$ nvprof --metrics branch_efficiency ./divergence
```

- ✓ Escludere l'**ottimizzazione** del compilatore con i param di debug **-g** e **-G**:

```
$ nvcc -g -G -arch=sm_20 divergence.cu -o divergence
```

| Invocations                       | Metric Name       | Metric Description | Min     | Max     | Avg     |
|-----------------------------------|-------------------|--------------------|---------|---------|---------|
| Device "Tesla M2090 (0)"          |                   |                    |         |         |         |
| Kernel: evenOdd_NO_DIV(int*, int) |                   |                    |         |         |         |
| 1                                 | branch_efficiency | Branch Efficiency  | 100.00% | 100.00% | 100.00% |
| Kernel: evenOdd_DIV(int*, int)    |                   |                    |         |         |         |
| 1                                 | branch_efficiency | Branch Efficiency  | 87.50%  | 87.50%  | 87.50%  |

- ✓ Con **ottimizzazione** (automatica) del compilatore param **-O3** :

| Invocations                       | Metric Name       | Metric Description | Min     | Max     | Avg     |
|-----------------------------------|-------------------|--------------------|---------|---------|---------|
| Device "Tesla M2090 (0)"          |                   |                    |         |         |         |
| Kernel: evenOdd_NO_DIV(int*, int) |                   |                    |         |         |         |
| 1                                 | branch_efficiency | Branch Efficiency  | 100.00% | 100.00% | 100.00% |
| Kernel: evenOdd_DIV(int*, int)    |                   |                    |         |         |         |
| 1                                 | branch_efficiency | Branch Efficiency  | 100.00% | 100.00% | 100.00% |

# Branch efficiency: numero

- ✓ Il **compilatore** interviene per **eliminare la divergenza** nei branch
- ✓ Usa fare sostituzioni di **path condizionali** a patto che il corpo del branch sia piccolo

$$\#Branch\ Efficiency = 100 \times \left[ \frac{\#Branches - \#DivergentBranches}{\#Branches} \right]$$

```
$ nvcc -g -G -arch=sm_20 divergence.cu -o divergence
```

```
$ nvprof --events branch,divergent_branch ./divergence 256 1000000
```

| Invocations                       | Event Name       | Min    | Max    | Avg    | Total  |
|-----------------------------------|------------------|--------|--------|--------|--------|
| Device "Tesla M2090 (0)"          |                  |        |        |        |        |
| Kernel: evenOdd_NO_DIV(int*, int) |                  |        |        |        |        |
| 1                                 | branch           | 250042 | 250042 | 250042 | 250042 |
| 1                                 | divergent_branch | 0      | 0      | 0      | 0      |
| Kernel: evenOdd_DIV(int*, int)    |                  |        |        |        |        |
| 1                                 | branch           | 250042 | 250042 | 250042 | 250042 |
| 1                                 | divergent_branch | 31256  | 31256  | 31256  | 31256  |

#branch Efficiency =  $100 \times (250042 - 31256) / 250042 = 87.5\%$

# Parallel reduction

Somma parallela

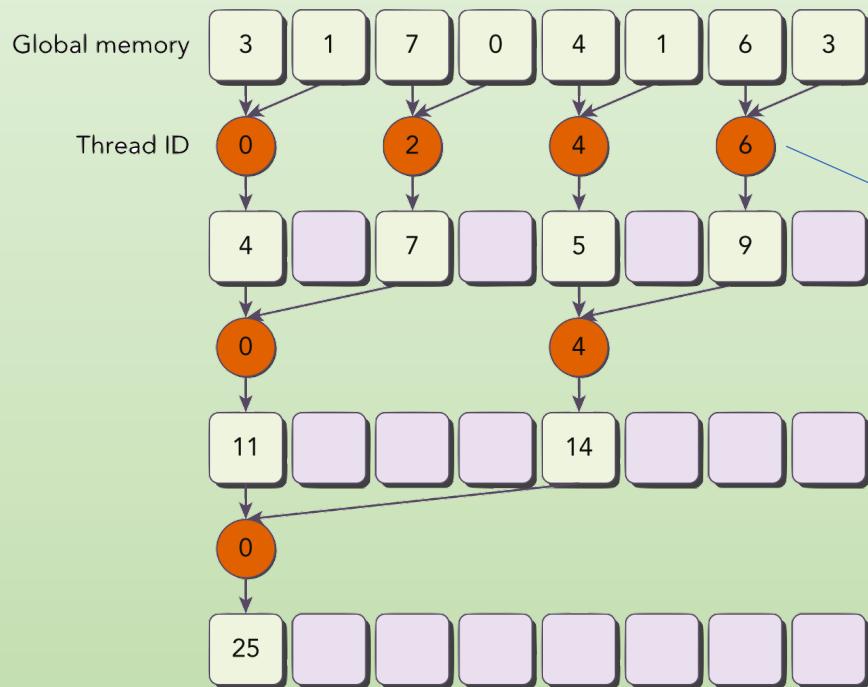
# Somma parallela

**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;

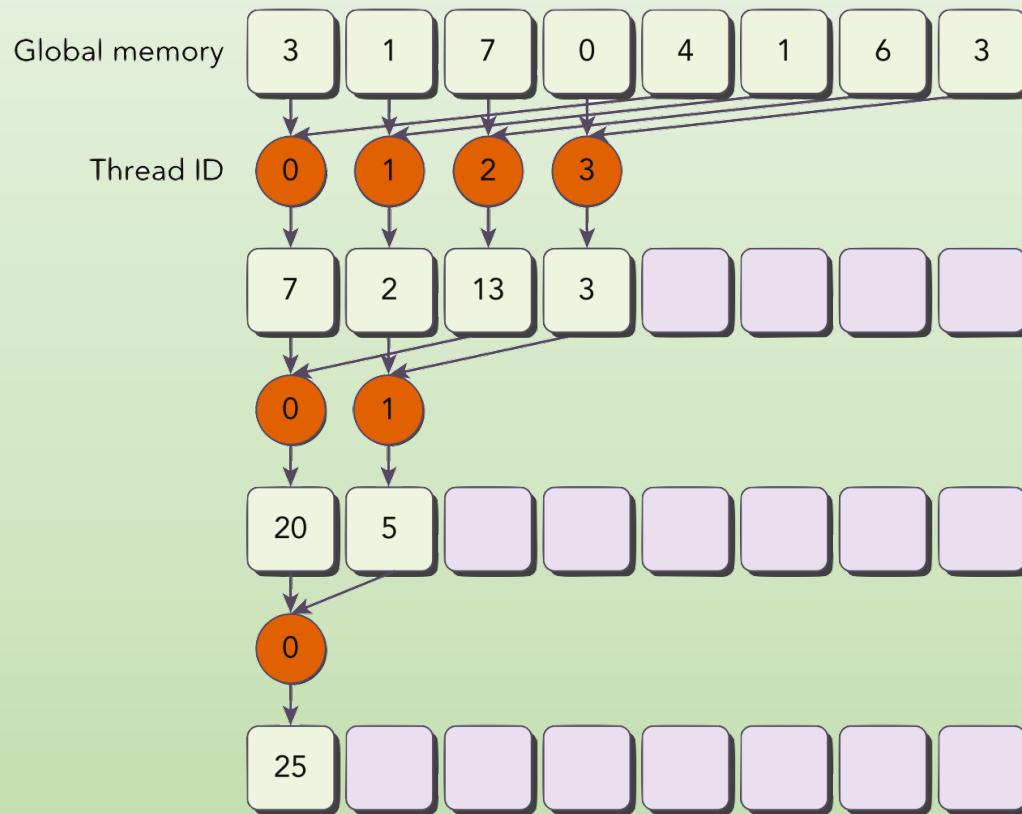
    // in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2) {
        if ((tid % (2 * stride)) == 0)
            thisBlock[tid] += thisBlock[tid + stride];

        // 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

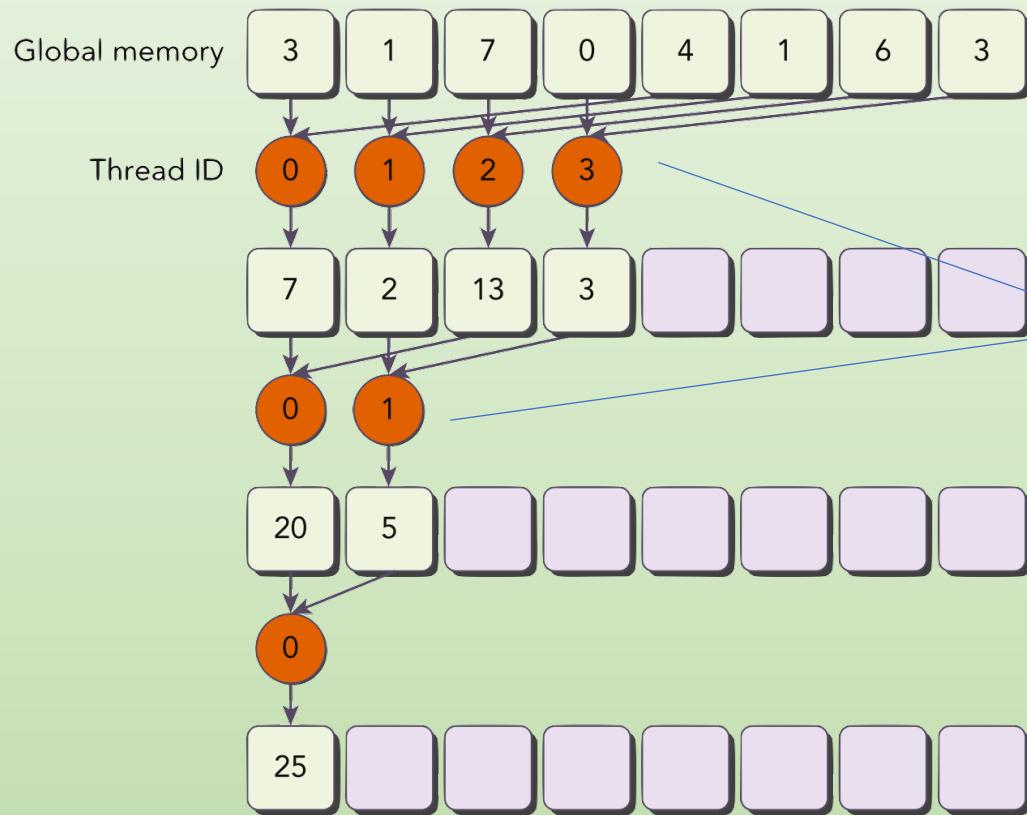
CPU procedure...
    Elapsed time: 1.578515 (sec)

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();
    }

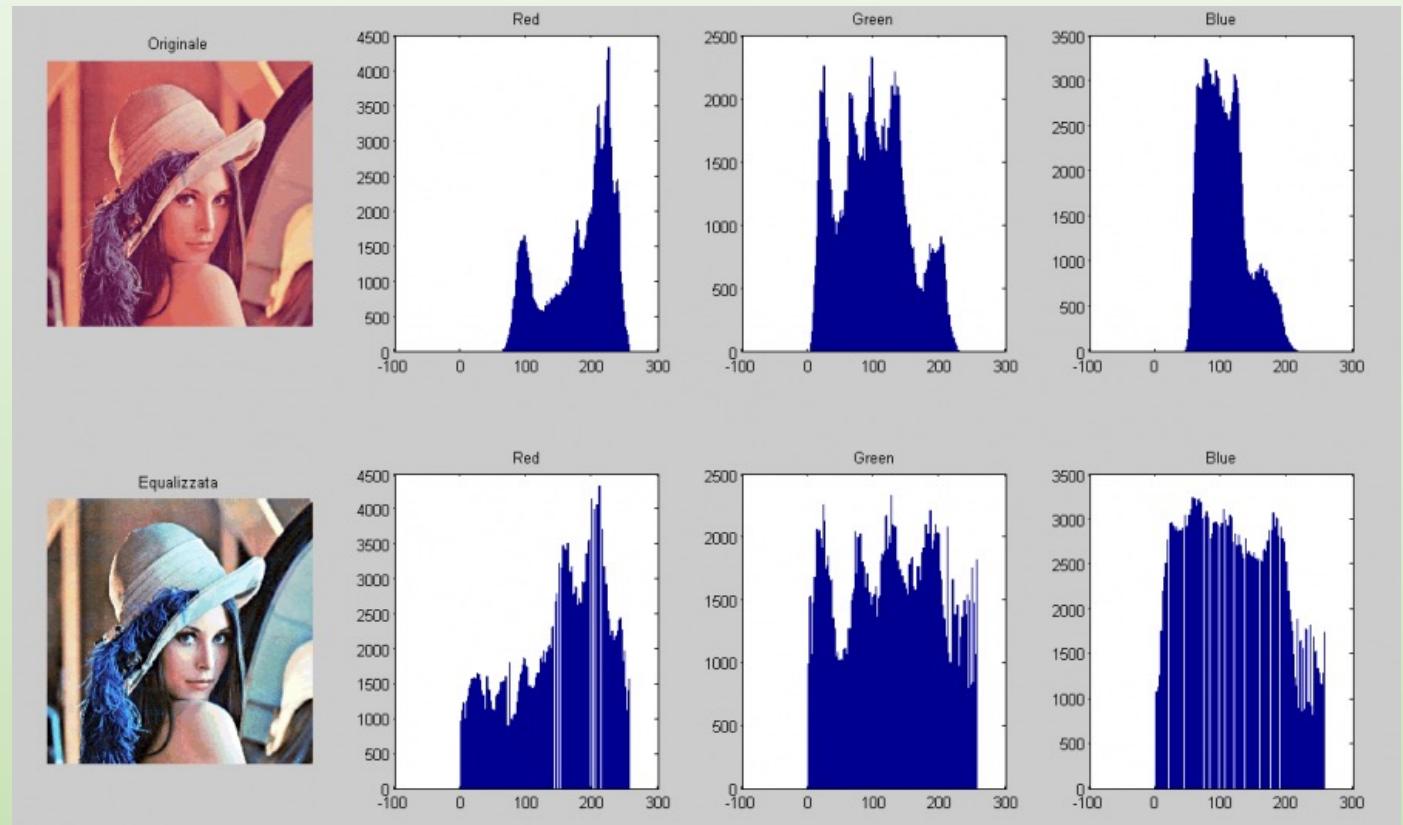
    // write result for this block to global mem
    if (tid == 0)
        out[blockIdx.x] = thisBlock[0];
}
```

# Istogramma di un’immagine

Operazioni atomiche

# Iistogramma

- ✓ Considerare separatamente i canali **RGB** dell'immagine BPM
- ✓ Usare le operazioni **atomiche** nello sviluppo CUDA per calcolo frequenze
- ✓ Restituire tre **istogrammi** distinti (uno per canale) di **256** valori di intensità colore



# Hist: soluzione grid 1D

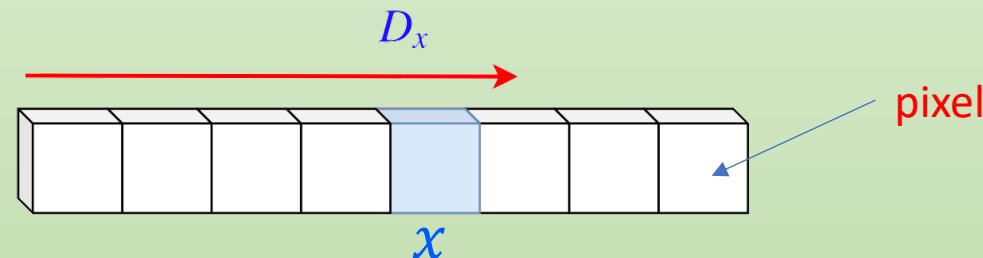
parametro  
libero

num blocchi  
dipendente da  
num pixel

```
// invoke kernels (define grid and block sizes)
dimBlock = 256;
uint nPixels = WIDTH*HEIGHT;
int dimGrid = (nPixels + dimBlock - 1) / dimBlock;
...
histogramBMP<<<dimGrid, dimBlock>>>(binsRGB, imgBMP, WIDTH);
```

- ✓ OSS: La griglia indicizza i pixel... non la terna di i byte dei valori RGB!

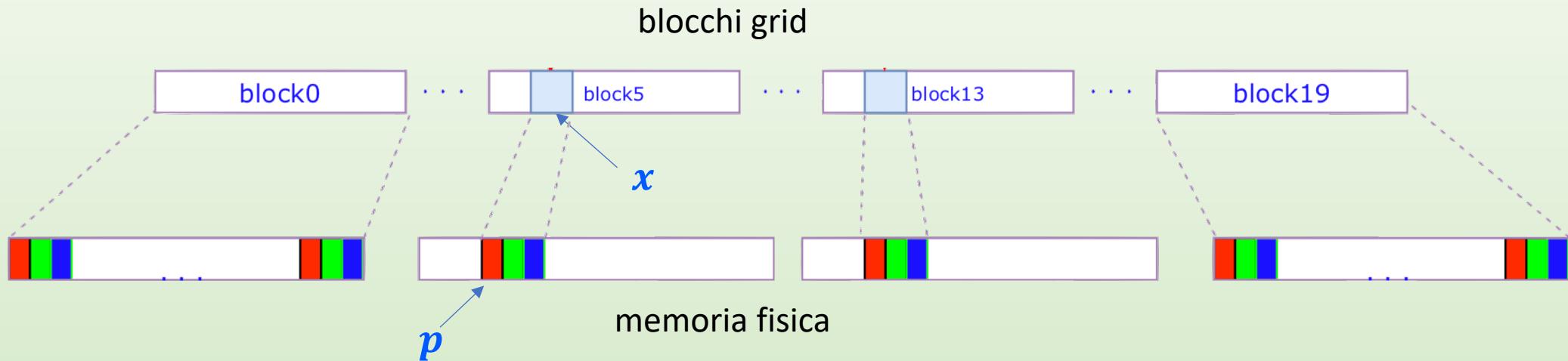
- ✓ Grid 1D e block 1D:



- ✓ Indice di pixel progressivo da 0 a  $W \times H - 1$   
(W = 3200 H = 2400)

$$x = blkDIM * blkID + thID$$

# Accesso a byte in memoria lineare



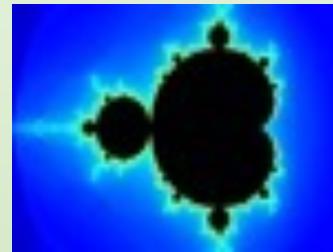
posizione della  
terna RGB del  
pixel src

usa  
espressione  
atomica x  
incremento  
dei bin

```
uint s = (w * 3 + 3) & (~3);           // num bytes x row (mult. 4)
. . .
// byte granularity
uint p = s * r + 3*off;      // src byte position of the pixel
pel R = imgSrc[p];
. . .
atomicAdd(&bins[R], 1);    // inc bin of pixel R
. . .
```

# Tempi di computazione

- ✓ Tempi (in sec) di esecuzione di diverse **GPU** vs la **CPU** (Intel Xeon CPU E5-2620 v4 @ 2.10GHz) e relativi **speedup** (rapporto tempi Host/Device)
- ✓ Immagini considerate:



**Mandelbrot** (W=8000, H=6000) = **144 MB**

**Dog** (W=1024, H=524) = **1.6 MB**

| CPU/GPU (blk size) | Dog     | Mandelbrot | Speedup GPU vs CPU |
|--------------------|---------|------------|--------------------|
| CPU                | 0.0027  | 0.24       | -                  |
| Tesla M2090        | «fill»  | «fill»     |                    |
| Tesla P100 (128)   | 0.00028 | 0.039      | ~ 10 * ~ 6         |
| Tesla P100 (512)   | 0.00033 | 0.040      | ~ 8 * ~ 6          |
| Tesla P100 (1024)  | 0.00043 | 0.039      | ~ 6 * ~ 6          |