GPU Computing

Lab2

ID di grid e block

Gerarchia thread

Kernel: indici e dimensioni

Kernel function: uso delle vars builtin

Dimensionare blocchi e griglia… Identificatore = variabile libera

La CPU non aspetta… prosegue

Runtime: distruzione contesto associato a processo

```
#include <stdio.h>

/*
 * Mostra DIMs e IDs di grid, block e thread
 */
__global__ void checkIndex(void) {
    printf("threadIdx:(%d, %d, %d) blockIdx:(%d, %d, %d) "
        "blockDim:(%d, %d, %d) gridDim:(%d, %d, %d)\n",
        threadIdx.x, threadIdx.y, threadIdx.z,
        blockIdx.x, blockIdx.y, blockIdx.z,
        blockDim.x, blockDim.y, blockDim.z,
        gridDim.x,gridDim.y,gridDim.z);
}

int main(int argc, char **argv) {
    // definisce grid e struttura dei blocchi
    dim3 block(4);
    dim3 grid(3);
    // controlla dim. dal lato host
    printf("grid.x %d grid.y %d grid.z %d\n", grid.x, grid.y, grid.z);
    printf("block.x %d block.y %d block.z %d\n", block.x, block.y, block.z);
    // controlla dim. dal lato device
    checkIndex<<<grid, block>>>();
    // reset device
    cudaDeviceReset();
    return(0);
}
```

Esecuzione

Compilazione

```
$ nvcc -arch=sm_75 grid1D.cu -o grid1D
```

Esecuzione con parametri **gridDim = 3** e **blockDim = 4**

```
$ ./ grid1D
grid.x = 3   grid.y = 1   grid.z = 1
block.x = 4   block.y = 1   block.z 1
threadIdx:(0, 0, 0) blockIdx:(1, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(1, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(1, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(3, 0, 0) blockIdx:(1, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(0, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(0, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(0, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(3, 0, 0) blockIdx:(0, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(2, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(2, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(2, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(3, 0, 0) blockIdx:(2, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
```

Completare il kernel grid2D

- ✓ “Filtrare” gli indici di un kernel CUDA basato su grid 2D che soddisfino il seguente requisito:
  - Attiva solo i thread con coordinate… (vedi notebook)

```
#include <stdio.h>

/*
 * Show DIMs & IDs for grid, block and thread
 */
__global__ void grid2D(void) {

    // TODO
}

int main(int argc, char **argv) {

      // grid and block structure
      dim3 block(7,6);
      dim3 grid(2,2);

      // check for host
      printf("CHECK for host:\n");
      printf("grid.x = %d\t grid.y = %d\t grid.z = %d\n", grid.x, grid.y, grid.z);
      printf("block.x = %d\t block.y = %d\t block.z %d\n", block.x, block.y, block.z);

      // check for device
      printf("CHECK for device:\n");
      checkIndex<<<grid, block>>>();

      // reset device
      cudaDeviceReset();
      return (0);
}
```

Risultato grid2D

```
$ ./ grid2D
grid.x = 2 grid.y = 2 grid.z = 1
block.x = 8 block.y = 7 block.z 1
threadIdx:(1, 4, 0) blockIdx:(1, 0, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(6, 4, 0) blockIdx:(1, 0, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(0, 5, 0) blockIdx:(1, 0, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(5, 5, 0) blockIdx:(1, 0, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(4, 6, 0) blockIdx:(1, 0, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(1, 4, 0) blockIdx:(0, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(6, 4, 0) blockIdx:(0, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(0, 5, 0) blockIdx:(0, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(5, 5, 0) blockIdx:(0, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(4, 6, 0) blockIdx:(0, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(1, 4, 0) blockIdx:(1, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(6, 4, 0) blockIdx:(1, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(0, 5, 0) blockIdx:(1, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
threadIdx:(5, 5, 0) blockIdx:(1, 1, 0) blockDim:(8, 7, 1) gridDim:(2, 2, 1)
. . .
```

Flip di un’immagine

Multithreading su pixel

Esempio: flip (V/H) di un’immagine

flip orizzontale

flip verticale

Formato bitmap (PPM)

- ✓ **PPM**: immagine bitmap a colori **lossless** di dimensioni $W \times H$
- ✓ Lunghezza in **byte** $N = 3 * W * H$ (size dell'immagne)

Header di 3 righe:
- formato
- W H
- max val

Header + sequenza di N byte

format

```
P6
1024 524
255
```

```
0000000   P   6  \n   1   0   2   4       5   2   4  \n   2   5   5  \n
0000020         x 227   ^     x 227   _     y 230         z 231   b   | 233   c
0000040   | 231   c   ~ 230   d   } 231   h   } 233   h   ~ 234   h 177
0000060 232   i   ~ 232   i   ~ 231   h 177 230   h   ~ 231   h 177 232
0000100   k 200 233   l 201 234   k 200 234   j 201 234   i 202 234   h
0000120 201 234   i 201 235   l 200 235   l 201 234   m 201 236   o 201
                                    . . .
6107700 276   ©  ** 275   é  ** 275   §  ** 275   §  ** 277   ç  ** 277
6107720   Ī  ** 301   Ī  ** 300   ê  ** 276   Ū  ** 276   ũ  ** 300   ũ
6107740  ** 277   Ī  ** 300   Ũ  ** 277   è  ** 277   ħ  ** 277   æ  **
6107760 277   å  ** 275 301 245 275 301 247 277   ç  ** 277   é  ** 277
6110000   ĩ  ** 276   ç  ** 273   ¨  ** 275   ¨  ** 275   è  ** 275 303
6110020
```

Rappresentazione e API C

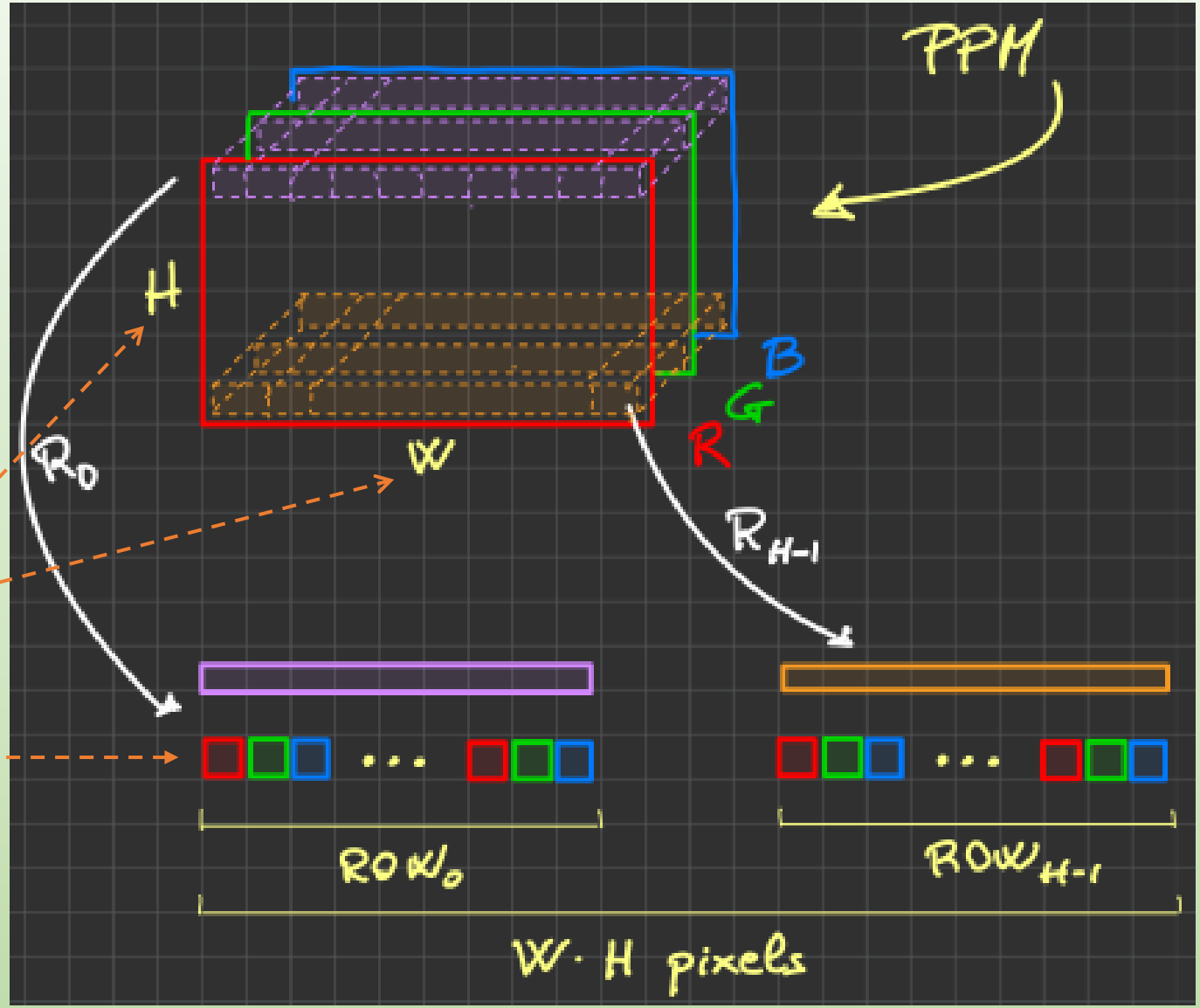
```c
#define color unsigned char
typedef struct {
    color r;
    color g;
    color b;
} pel;
typedef struct {
    int width, height, maxval;
    color *image;
} PPM;
```

```c
// Load ppm image from file
PPM *ppm_load(const char *filename);
// Write ppm image to file
void ppm_write(PPM *ppm, const char *filename);
// Get pel (pixel element) from ppm image
pel ppm_get(PPM *ppm, int x, int y);
// Set pel (pixel element) in ppm image
void ppm_set(PPM *ppm, int x, int y, pel c);
// Create a copy of ppm image
PPM *ppm_copy(PPM *ppm);
// Create a new ppm image (width x height) with all pixels set to c
PPM *ppm_make(int width, int height, pel c);
// Create a new ppm image (width x height) with random pixel values.
PPM *ppm_rand(int width, int height);
// Flip vertically in place by swapping row elements.
void ppm_flipH(PPM *ppm);
// Flip horizontally in place by swapping column elements.
void ppm_flipV(PPM *ppm);
// Flip vertically in place by swapping rows
void ppm_flipV_row(PPM *ppm);
// Compare two ppm images for equality
int ppm_equal(PPM *ppm1, PPM *ppm2);
// Blurring filter for ppm images
PPM *ppm_blur(PPM *ppm);
```

Organizzazione in memoria

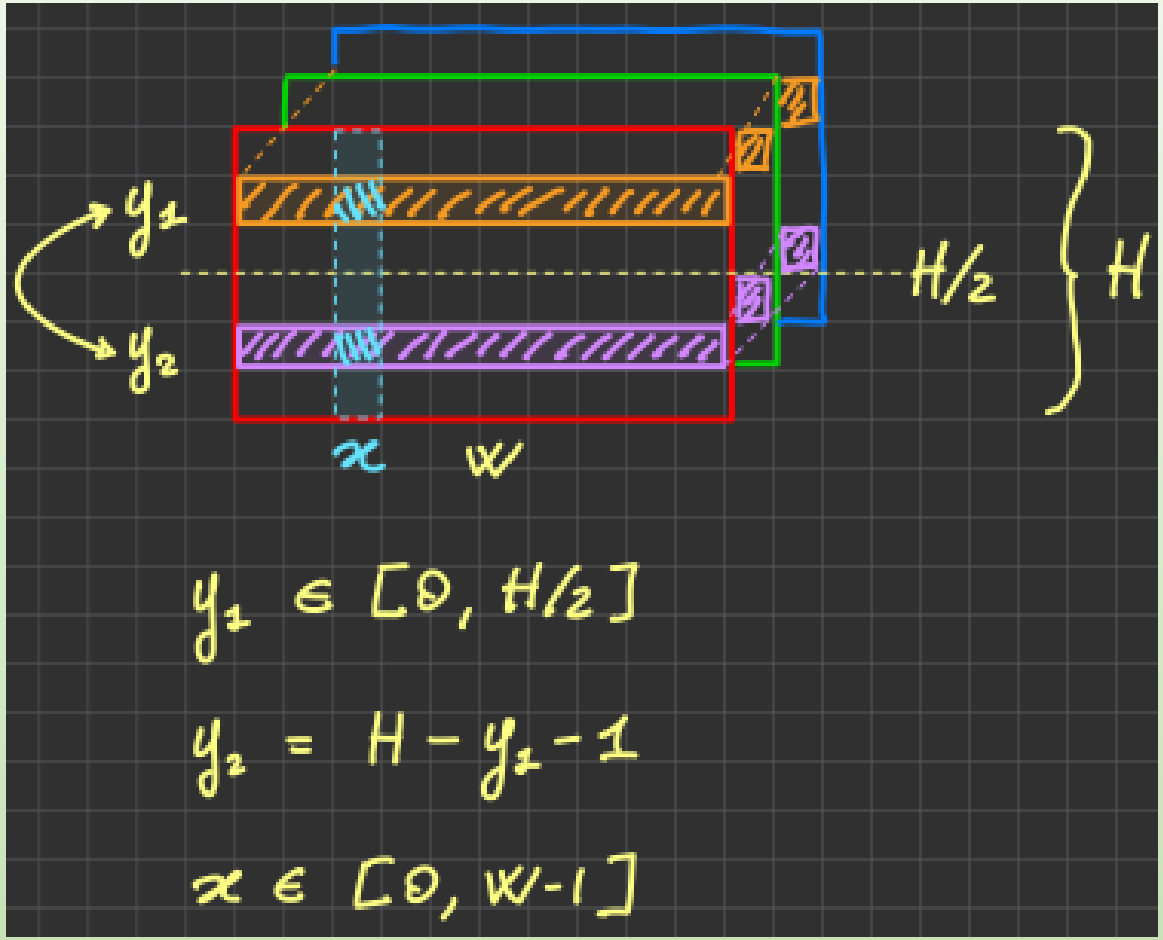
```
typedef struct {
     color r;
     color g;
     color b;
} pel;


typedef struct {
     int width, height;
     color *image;
} PPM;
```



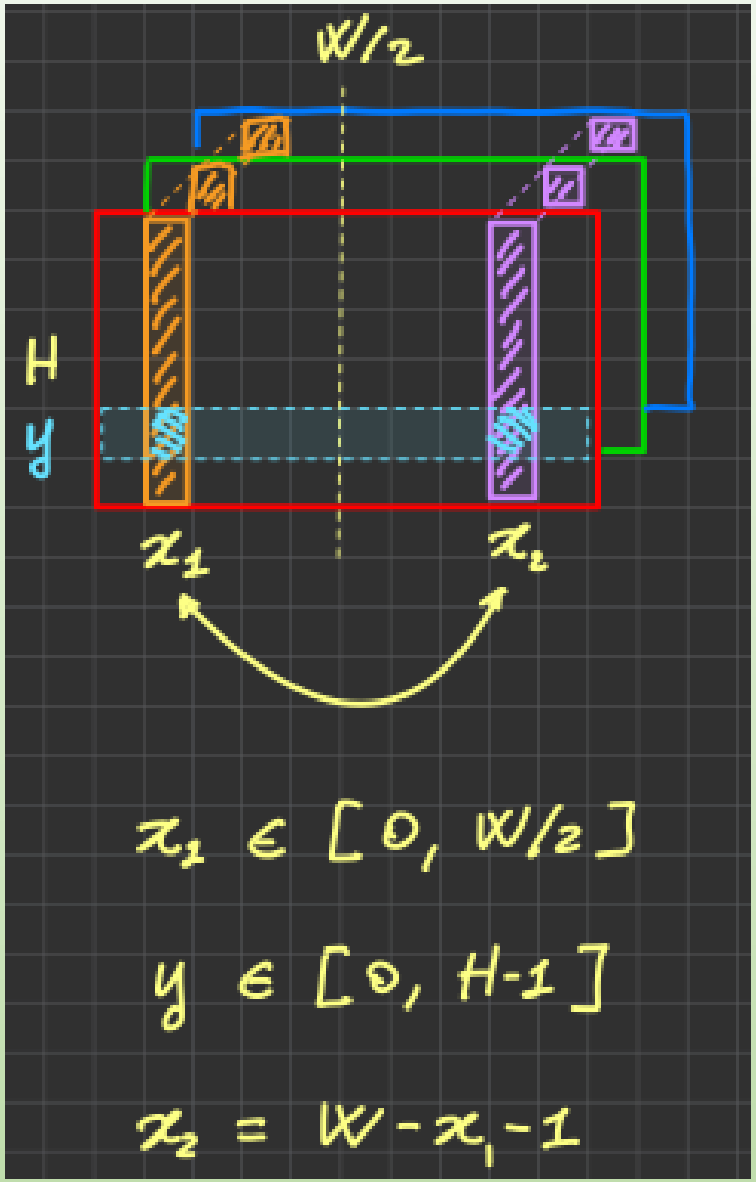
Schema di swap V (API C)

```c
/*
* Flip vertically in place by swapping row elements
*/
void ppm_flipV(PPM* ppm) {
  for (int y = 0; y < ppm->height/2; y++) {
    for (int x = 0; x < ppm->width; x++) {
       pel p1 = ppm_get(ppm, x, y);
       pel p2 = ppm_get(ppm, x, ppm->height-y-1);
       ppm_set(ppm, x, y, p2);
       ppm_set(ppm, x, ppm->height - y - 1, p1);
    }
  }
}
```



Schema di swap H (API C)

```c
/*
* Flip horizontally in place by swapping column elem
*/
void ppm_flipH(PPM* ppm) {
   for (int x = 0; x < ppm->width/2; x++) {
      for (int y = 0; y < ppm->height; y++) {
         pel p1 = ppm_get(ppm, x, y);
         pel p2 = ppm_get(ppm, ppm->width-x-1,y);
         ppm_set(ppm, x, y, p2);
         ppm_set(ppm, ppm->width - x - 1, y, p1);
      }
   }
}
```



Flipping di un'immagine su GPU

Osservazioni:

- ✓ Decidere che cosa deve fare l'host e che cosa il device
- ✓ la memoria dell'immagine è linearizzata 1D… tenerne conto nel disegno del kernel
- ✓ stabilire la dimensione di blocco di thread
- ✓ provare diverse configurazioni per aumentare le prestazioni
- ✓ misurare le prestazioni
- ✓ di seguito alcuni suggerimenti…

Flipping con CUDA: Soluzione 1D

Numero thread x blocco

num blocchi totali

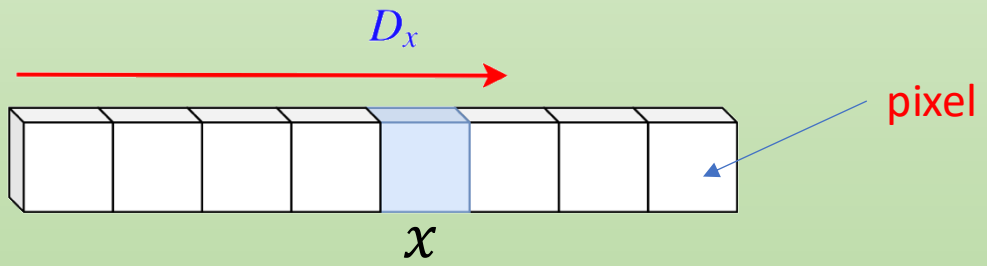
Flip H

Flip V

```
// invoke kernels (define grid and block sizes)
uint dimBlock = 256;
uint dimGrid = (WIDTH/2 * HEIGHT + dimBlock - 1) / dimBlock;
ppm_flipH_GPU <<<dimGrid, dimBlock>>> (ppm_d);
ppm_flipV_GPU <<<dimGrid, dimBlock>>> (ppm_d);
```

✓ **OSS**: La grid di thread indicizza I pixel… non la terna di byte dei valori RGB!

✓ **Grid 1D e block 1D**:



Mapping tids e pixels

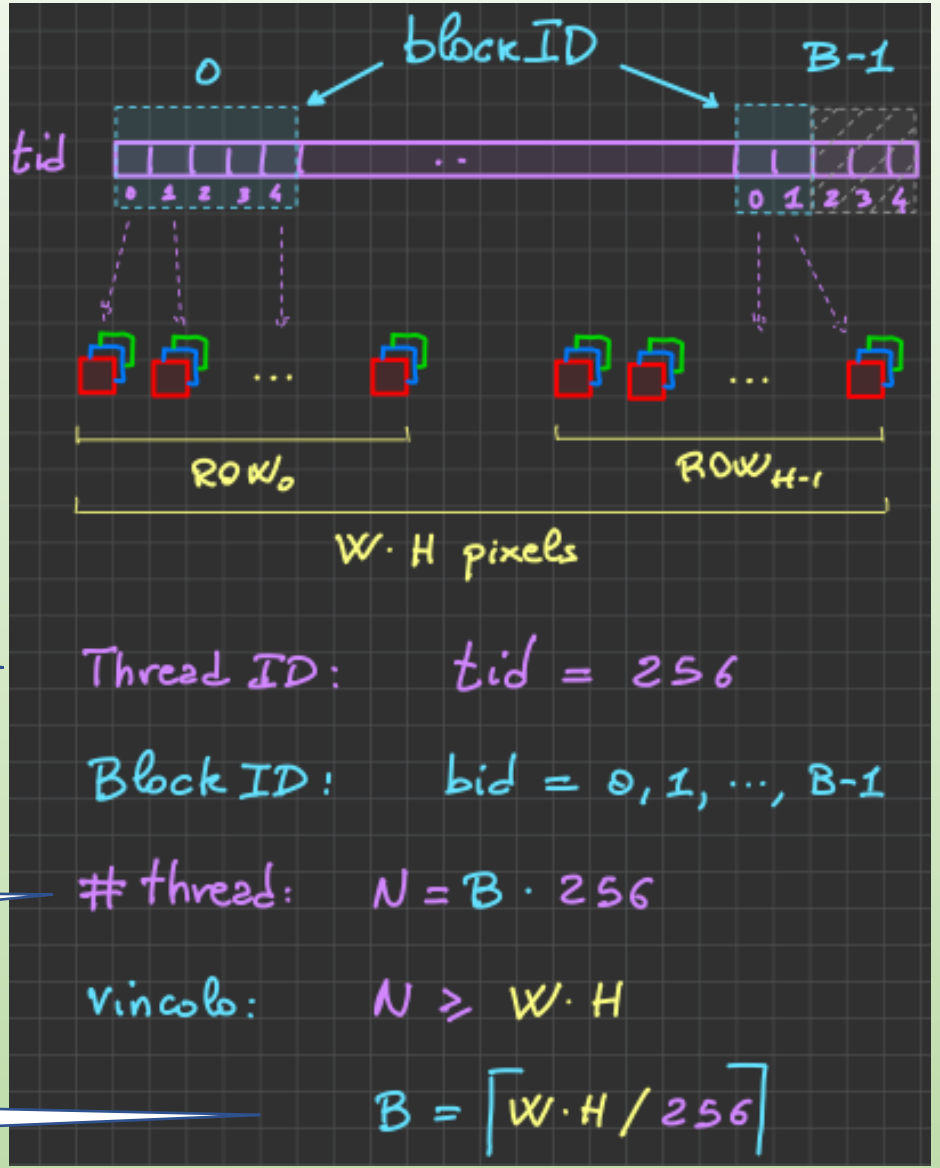
Numero thread disponibili con B blocchi e T thread x blocco

Numero pixel da elaborare: W x H

Numero thread fissati a priori: es. T=256

Numero thread necessari: N

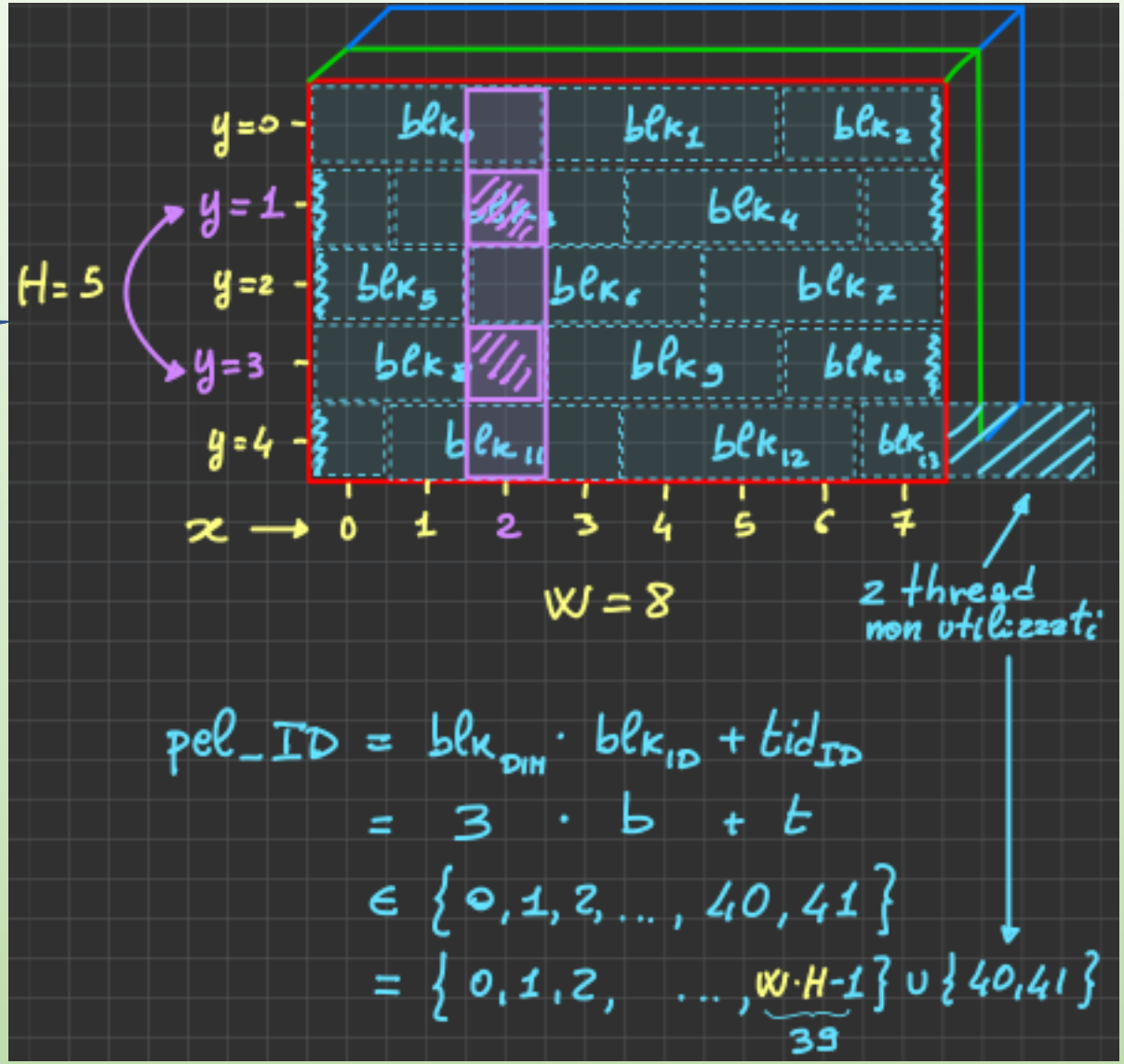
Numero blocchi calcolati: B



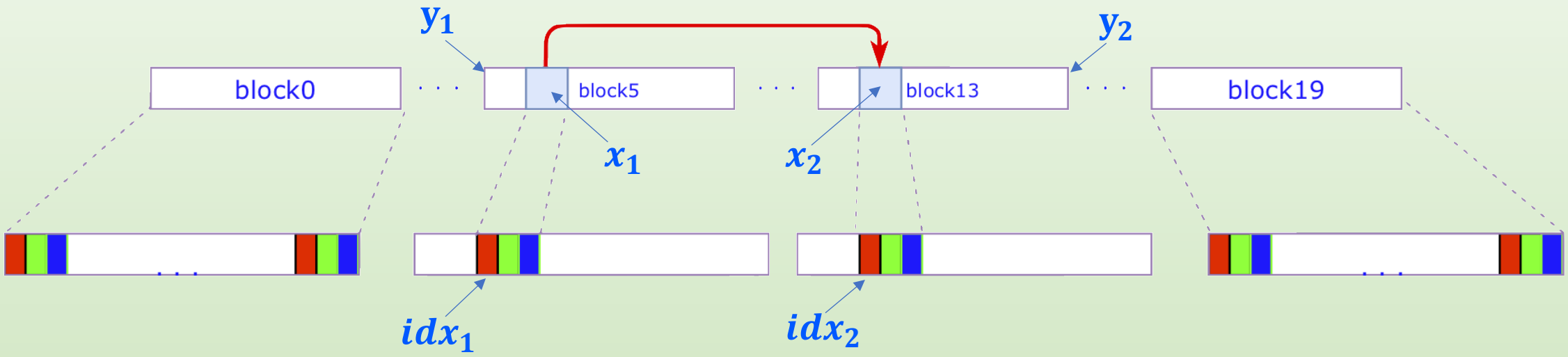
‘Copertura’ con blocchi di thread

Righe su cui effettuare lo swap e colonna unica

Spazio indici di thread per i pixel dopo aver fissato la dimensione di blocco di thread **3** e note **W** e **H**



Accesso a byte in memoria lineare



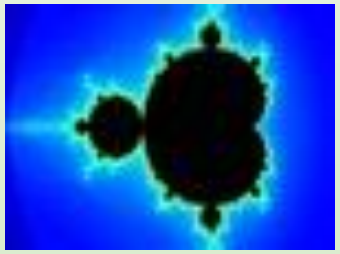
multiplo di 3 byte corrisponde a terna RGB del pixel src in row **y1** e col **x**

lo stesso per pixel di destinazione in row **y2** e col **x**

```
// ** byte granularity **
uint idx1 = 3 * (x + y1 * WIDTH);
. . .
uint idx2 = 3 * (x + y2 * WIDTH);
```

Flip verticale: colonne out of range

✓ Se si fissa la dimensione di blocco ($b = \text{blk}_{\text{Dim}} = |\text{blk}|$) la grid risultante ha un numero di colonne che potrebbe eccedere quello dell'immagine e quindi devono essere escluse



H=6000

W=8000

H=524

W=1024

$b = 64$

$8000/b = 125$

$num\ Blk = b * 125 = 8000$

$1024/b = 16$

$num\ Blk = b * 16 = 1024$

$b = 128$

$8000/b = 62.5$

$num\ Blk = b * 63 = 8064$

$1024/b = 8$

$num\ Blk = b * 8 = 1024$

$b = 256$

$8000/b = 31.2$

$num\ Blk = b * 32 = 8192$

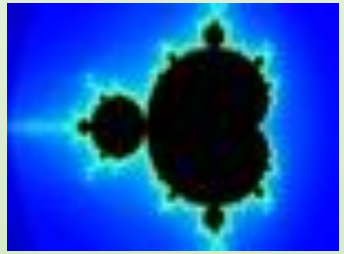
eccedenti!!

$1024/b = 4$

$num\ Blk = b * 4 = 1024$

Tempi di computazione

- ✓ Tempi (in sec) di esecuzione di diverse **GPU** vs la **CPU** (2,9 GHz Intel Core i7 quad-core) e relativi **speedup** (rapporto tra tempi Dev/Hots)
- ✓ Immagini considerate:



Mandelbrot (W=8000, H=6000) = **I44 MB**

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

CPU/GPU	Dog	Mandelbrot	Speedup GPU vs CPU
CPU	0.0026	0.3336	-
Tesla M2090	0.00017		~15.2 -
Tesla K40	0.00011	0.0038	~23.6 - ~87.8
Tesla P100	0.000084	0.0014	~30.9 - ~238.3

Filtro di blurring per un’immagine

Multithreading con schema 2D

API C

```c
/*
* Set pel (pixel element) in ppm image.
*/
void ppm_set(PPM* ppm, int x, int y, pel c) {

      int i = x + y*ppm->width;
      ppm->image[3*i] = c.r;
      ppm->image[3*i + 1] = c.g;
      ppm->image[3*i + 2] = c.b;
}
```

```c
/*
* Get pel (pixel element) from ppm image.
*/
pel ppm_get(PPM* ppm, int x, int y) {
      pel p;
      int i = x + y*ppm->width;
      p.r = ppm->image[3*i];
      p.g = ppm->image[3*i + 1];
      p.b = ppm->image[3*i + 2];
      return p;
}
```

```c
/*
* blur kernel
*/
pel ppm_blurKernel(PPM *ppm, int x, int y, int width, int height, int KERNEL_SIZE) {
   float R=0, G=0, B=0;
   int numPixels = 0;
   int RADIUS = KERNEL_SIZE/2;
   for(int r = -RADIUS; r < RADIUS; ++r) {
      for(int c = -RADIUS; c < RADIUS; ++c) {
         int row = y + r;
         int col = x + c;
         if(row > -1 && row < height && col > -1 && col < width) {
            pel p = ppm_get(ppm, col, row);
            R += p.r;
            G += p.g;
            B += p.b;
            numPixels++;
        }
      }
   }
   pel p_fil = {(color)(R/numPixels), (color)(G/numPixels), (color)(B/numPixels)};
   return p_fil;
}
```