

GPU Computing

Lab 2 – image flipping

Esempio: flipping (V/H) di un'immagine



flip orizzontale



flip verticale



Formato BitMap (BMP)

- ✓ Il formato di file Windows bitmap **lossless** di dimensioni **$W \times H$**
- ✓ Le immagini bitmap possono avere una **profondità** di 1, 4, 8, 16, 24 o 32 **bit x pixel**
- ✓ ogni riga ha una lunghezza **N** in byte **multipla di 4**, dove $N = (3 * W + 3) \& (~3)$
- ✓ Le bitmap con 1, 4 e 8 bit contengono una **tavolozza** per la **conversione** dei (rispettivamente 2, 16 e 256) possibili indici numerici nei rispettivi colori
- ✓ Nelle immagini con profondità più alta il **colore non è indicizzato** bensì codificato direttamente nelle sue componenti cromatiche **RGB**.
- ✓ La versione 3 del formato (in assoluto la più comune) non supporta il canale alfa né i metadati
- ✓ I **dati raw** sono così strutturati: **54 byte** occupati **dall'header** (width, height, inizio-fine tavolozza, etc.) poi seguono **3 byte per ogni pixel** (spazio RGB) della matrice di pixel che forma l'immagine

Rappresentazione

- ✓ **Struct** di definizione di un'immagine e funzioni lettura e scrittura

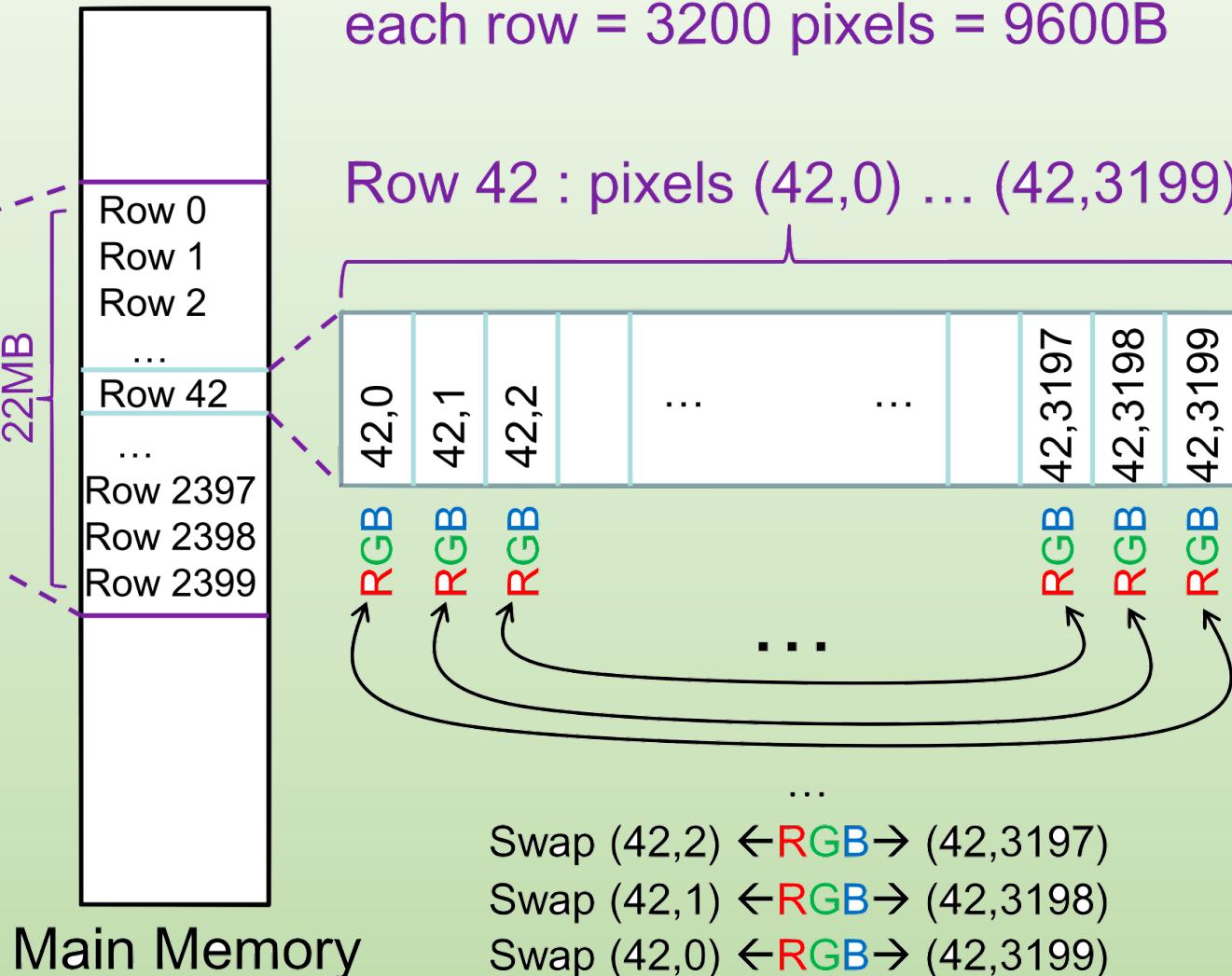
```
struct ImgProp {
    int Hpixels;
    int Vpixels;
    unsigned char HeaderInfo[54];
    unsigned long int Hbytes;
};

struct Pixel {
    unsigned char R;
    unsigned char G;
    unsigned char B;
};

typedef unsigned char pel;      // pixel element

pel** ReadBMP(char*);          // Load a BMP image
void WriteBMP(pel**, char*);   // Store a BMP image
```

Schema di swap



Uso di pthread

Multithreading su CPU

Pthreads library

POSIX-compliant operating system (Standard ANSI/IEEE POSIX 1003.1 - 1990)

Utilizzo

- ✓ Includere l'header della libreria `#include <pthread.h>`
- ✓ Compilare specificando la libreria: `gcc <opzioni> -pthread`

API function

- ✓ `pthread_create()` permette di **creare** un nuovo **thread** e ne restituisce il suo **ID** (unico)
- ✓ `pthread_join()` mette in **attesa** il chiamante fino alla **terminazione** del **thread** (si deve sempre specificare il thread di cui si vuole attendere la terminazione, il suo uso è “obbligato” per garantire la terminazione coerente dei thread quando termina il processo che ha creati i thread)
- ✓ `pthread_attr()` consente di **inizializzare** gli **attributi** dei thread
- ✓ `pthread_attr_setdetachstate()` **assegna/modifica** gli **attributi** appena inizializzati ai thread

Inizializzazione dei thread

✓ Attributo **detachstate** di un thread

- specifica se si può invocare o no la funzione join su un certo thread
- un thread è joinable per default

```
 . . .
#define MAXTHREADS    128
int NumThreads;                      // Total number of threads working in parallel
int ThParam[MAXTHREADS];             // Thread parameters ...
pthread_t ThHandle[MAXTHREADS];      // Thread handles
pthread_attr_t ThAttr;                // Pthread attributes
void (*FlipFunc)(pel** img);         // Function pointer to flip the image
void* (*MTFlipFunc)(void *arg);       // Function pointer to flip the image, multi-th version
. . .
if (NumThreads > 1) {
    pthread_attr_init(&ThAttr);
    pthread_attr_setdetachstate(&ThAttr, PTHREAD_CREATE_JOINABLE);
```

Creazione e lancio di thread

- ✓ **pthread_create()** inizializza e attiva l'esecuzione all'atto della creazione stessa
 - Il terzo argomento **MTFlipFunc** dice al thread di eseguire la **funzione**
 - Il quarto argomento **ThParam[i]** fornisce il ‘**puntatore**’ ai **dati** su cui applicare la funzione

```
for (i = 0; i < NumThreads; i++) {  
  
    ThParam[i] = i;  
  
    ThErr = pthread_create(&ThHandle[i], &ThAttr, MTFlipFunc, (void *)&ThParam[i]);  
  
    if (ThErr != 0) {  
  
        printf("\nThread Creation Error %d. Exiting abruptly... \n", ThErr);  
  
        exit(EXIT_FAILURE);  
    }  
}
```

Join di thread

- ✓ Forma elementare di **sincronizzazione**: un thread si mette in **attesa** di un **altro thread**
- ✓ il thread che effettua il join si **blocca** finché uno specifico **thread non termina**
- ✓ Primo parametro **ThHandle[i]** è l'**id del thread** da aspettare
- ✓ **pthread_join(x)** significa: *wait until thread with the handle number x is done*
- ✓ La funzione **restituisce 0** in caso di **successo**, un valore diverso da zero in caso contrario

```
    . . .
    pthread_attr_destroy(&ThAttr);
    for (i = 0; i < NumThreads; i++) {
        pthread_join(ThHandle[i], NULL);
    }
    . . .
```

Esecuzione dei task (con thread)

- ✓ IL `main()` crea i thread e **assegna** un unico `tid` a ognuno a **runtime** (`ThHandle[i]`) nel codice sotto) e invoca una **funzione** per ogni thread (usando il puntatore `MTFlipFunc` con parametro `ThParam[i]`)

```
 . . .
    ThParam[i] = i;
    ThErr = pthread_create(&ThHandle[i], &ThAttr, MTFlipFunc, (void *)&ThParam[i]);
```

- ✓ Il `main()` deve anche **comunicare** al SO quali thread sono stati creati
- ✓ Il SO deve **decidere** se **creare** il thread (per es. se ci sono le risorse necessarie)
- ✓ Se un thread viene creato allora il SO gli **assegna** un handle in `ThHandle[i]`)
- ✓ alla fine il `main()` deve attendere che ogni thread termini – sincronizzazione `join` (dice all'OS di deallocate le risorse)

Flip verticale: divisione dei dati

- ✓ Se abbiamo un'immagine di dimensione 640 x 480 la versione sequenziale deve fare mirroring tra righe a partire da quelle in posizione estreme:

```
Row [0]: [0][0]↔[479][0], [0][1]↔[479][1] ... [0][639]↔[479][639]
Row [1]: [1][0]↔[478][0], [1][1]↔[478][1] ... [1][639]↔[478][639]
Row [2]: [2][0]↔[477][0], [2][1]↔[477][1] ... [2][639]↔[477][639]
Row [3]: [3][0]↔[476][0], [3][1]↔[476][1] ... [3][639]↔[476][639]
...
Row [239]: [239][0]↔[240][0], [239][1]↔[240][1] ... [239][639]↔[240][639]
```

- ✓ nella versione parallela devo ripartire il compito tra i vari thread... per es. se ho 4 threads, tid = 0,1,2,3

tid = 0 : Pixels [0...159]	Hbytes [0...477]
tid = 1 : Pixels [160...319]	Hbytes [480...959]
tid = 2 : Pixels [320...479]	Hbytes [960...1439]
tid = 3 : Pixels [480...639]	Hbytes [1440...1919]

```
void *MTFlipV(void* tid) {
    struct Pixel pix; //temp swap pixel
    int row, col;
    long ts = *((int *) tid);           // My thread ID is stored here
    ts *= ip.Hbytes / NumThreads; // start index
    long te = ts + ip.Hbytes / NumThreads - 1; // end index
    for (col = ts; col <= te; col += 3) {
        ...
    }
}
```

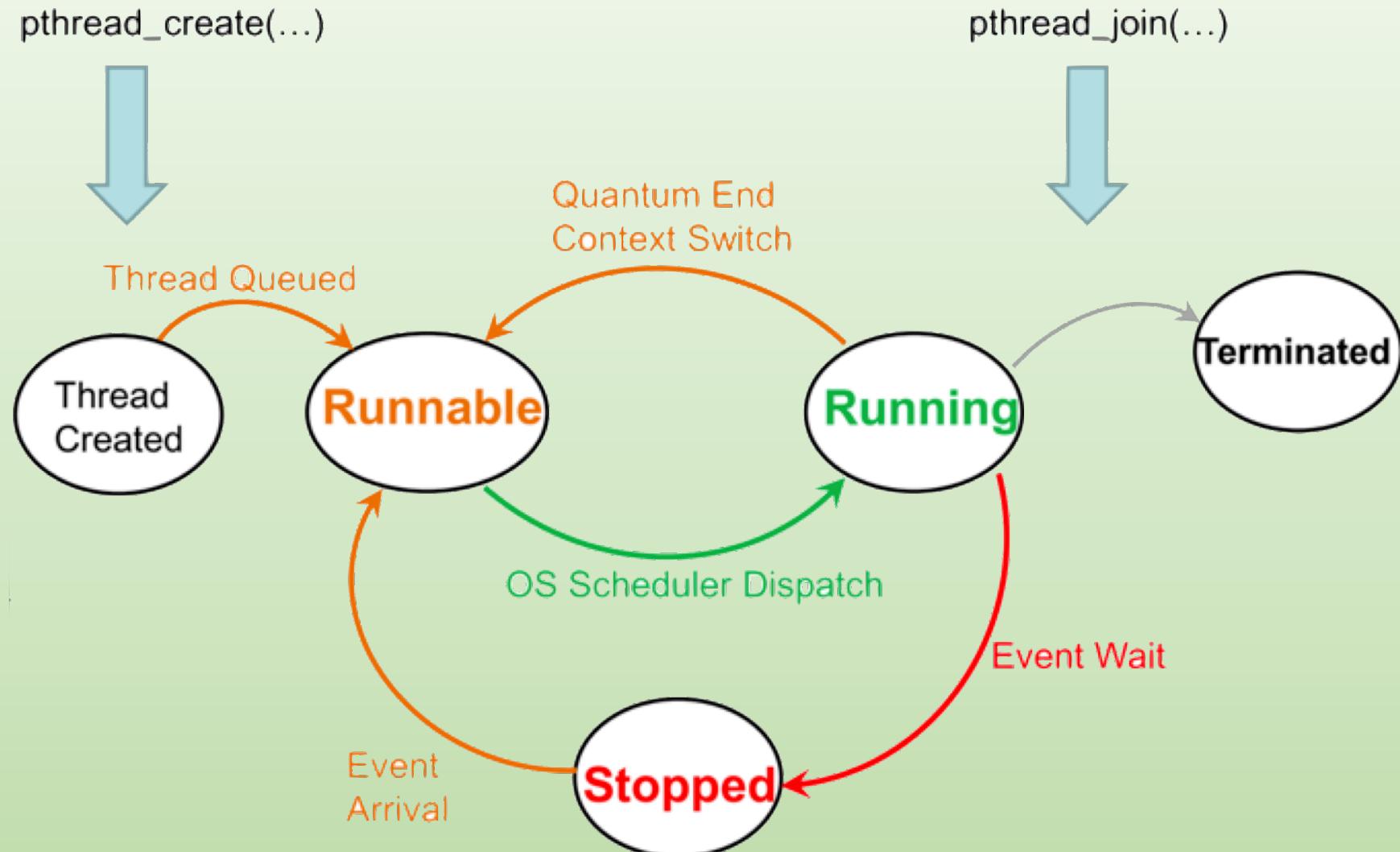
Prestazioni

- ✓ Da che cosa dipendono le differenze?
- ✓ ... velocità della CPU, numero di core, numero di thread?... gerarchie di memorie?
- ✓ Per es. su una CPU 4C/8T cosa succede se due thread eseguono sullo stesso core o in core differenti?
- ✓ come cambiano le prestazioni quando si lanciano 9 o più thread su CPU 4C/8T?
- ✓ politiche di scheduling?
- ✓ gestione di eccezioni: che accade se un lancio di thread fallisce per effetto del SO?

Feature	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6
Name	i5-4200M	i7-960	i7-4770K	i7-3820	i7-5930K	E5-2650
C/T	2C/4T	4C/8T	4C/8T	4C/8T	6C/12T	8C/16T
Speed:GHz	2.5-3.1	3.2-3.46	3.5-3.9	3.6-3.8	3.5-3.7	2.0-2.8
L1\$/C	64KB	64KB	64KB	64KB	64KB	64KB
L2\$/C	256KB	256KB	256KB	256KB	256KB	256KB
shared L3\$	3MB	8MB	8MB	10MB	15MB	20MB
Memory	8GB	12GB	32GB	32GB	64GB	16GB
BW:GB/s	25.6	25.6	25.6	51.2	68	51.2

#Threads	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	
Serial	V	109	131	159	117	181	185
2	V	93	70	50	58	104	95
3	V	78	46	33	43	75	64
4	V	78	67	49	59	54	49
5	V	93	55	40	52	35	57
6	V	78	51	35	55	35	48
8	V	78	52	37	53	26	37
9	V		47	34	52	25	49
10	V			40		23	45
12	V			35		28	38

Ciclo di vita di un thread



Uso di CUDA

Multithreading su GPU

Completere 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 (sia nella componente x sia nella y) abbiamo per somma un multiplo di 5

```
#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)
.
.
```

Esercitazione: flipping di un'immagine

Sviluppare un programma CUDA C che effettua il flipping (V/H) di un'immagine

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

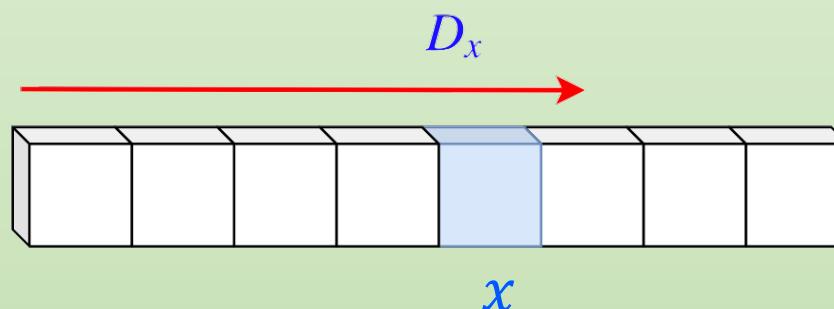
parametro
libero

num blocchi
per riga

num blocchi
totali

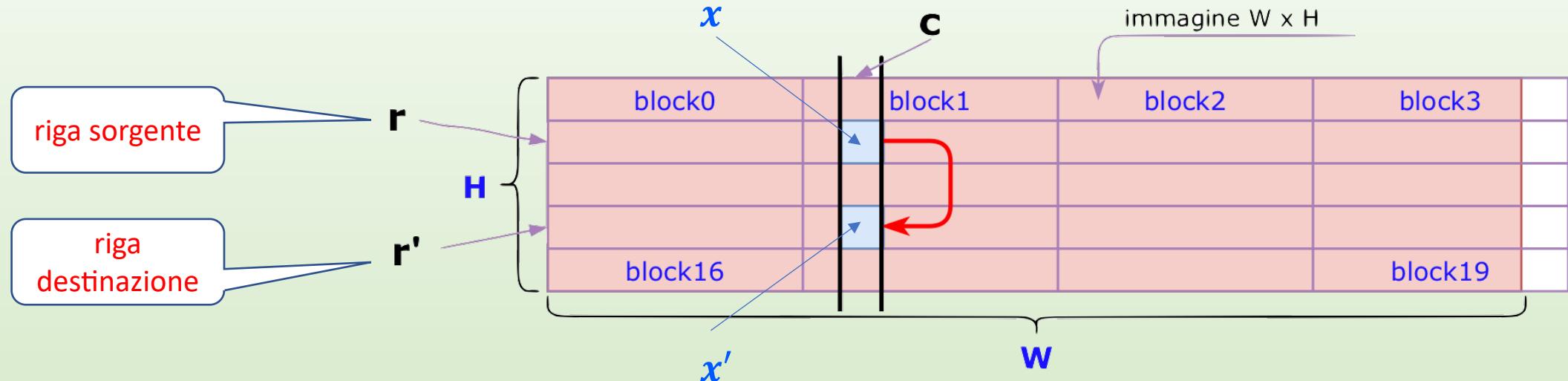
```
// invoke kernels (define grid and block sizes)
dimBlock = 256;
rowBlock = (WIDTH + dimBlock - 1) / dimBlock;
dimGrid = HEIGHT * rowBlock;
    ...
Hflip<<<dimGrid, dimBlock>>>(GPUcopyImg, GPUImg, WIDTH);
    ...
Vflip<<<dimGrid, dimBlock>>>(GPUcopyImg, GPUImg, WIDTH, HEIGHT);
```

- ✓ Grid 1D e block 1D:



- ✓ Indice di pixel progressivo da 1 a $W \times H$ $x = blkDim * blkID + thID$
($W = 3200$ $H = 2400$)

Indici x flip verticale



Indice del pixel sorgente:

$$x = \text{blk}_{\text{Dim}} * \text{blk}_{\text{ID}} + \text{th}_{\text{ID}}$$

$$x = |\text{blk}| * \text{blk}_i + \text{th}_j$$

$$x = b * i + j$$

Indice del pixel da sostituire:

$$x' = r' * m + c'$$

num blocchi per riga:

riga sorgente:

colonna sorgente:

riga destinazione:

colonna destinazione:

$$m = (w + b - 1)/b = \left\lceil \frac{w}{b} \right\rceil$$

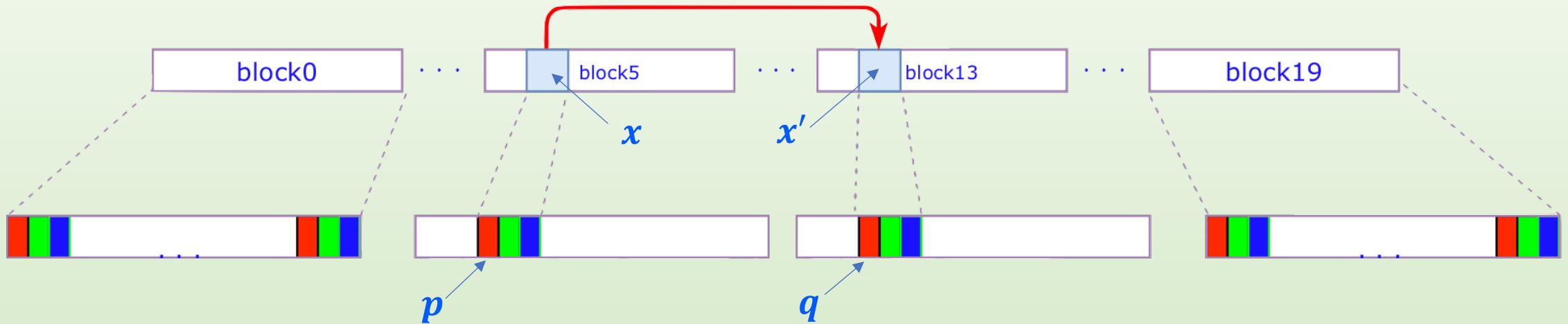
$$r = i/m$$

$$c = x - r * w$$

$$r' = h - 1 - r$$

$$c' = c$$

Accesso a byte in memoria lineare



multiplo di 3
byte corrisp. a
terna RGB del
pixel src in
row r e col c

lo stesso per
pixel di dst in
row r' e col c

```
uint s = (w * 3 + 3) & (~3);           // num bytes x row (mult. 4)
uint r1 = h - 1 - r;                   // dest. row (mirror)
// ** byte granularity **
uint p = s * r + 3*c;                 // src byte position of the pixel
uint q = s * r1 + 3*c;                // dst byte position of the pixel
// swap pixels RGB
imgDst[q] = imgSrc[p];               // R
imgDst[q + 1] = imgSrc[p + 1];        // G
imgDst[q + 2] = imgSrc[p + 2];        // B
```

Test: BMP 8000x6000

48M pixel = 144M + 54 byte

