

GPU Computing

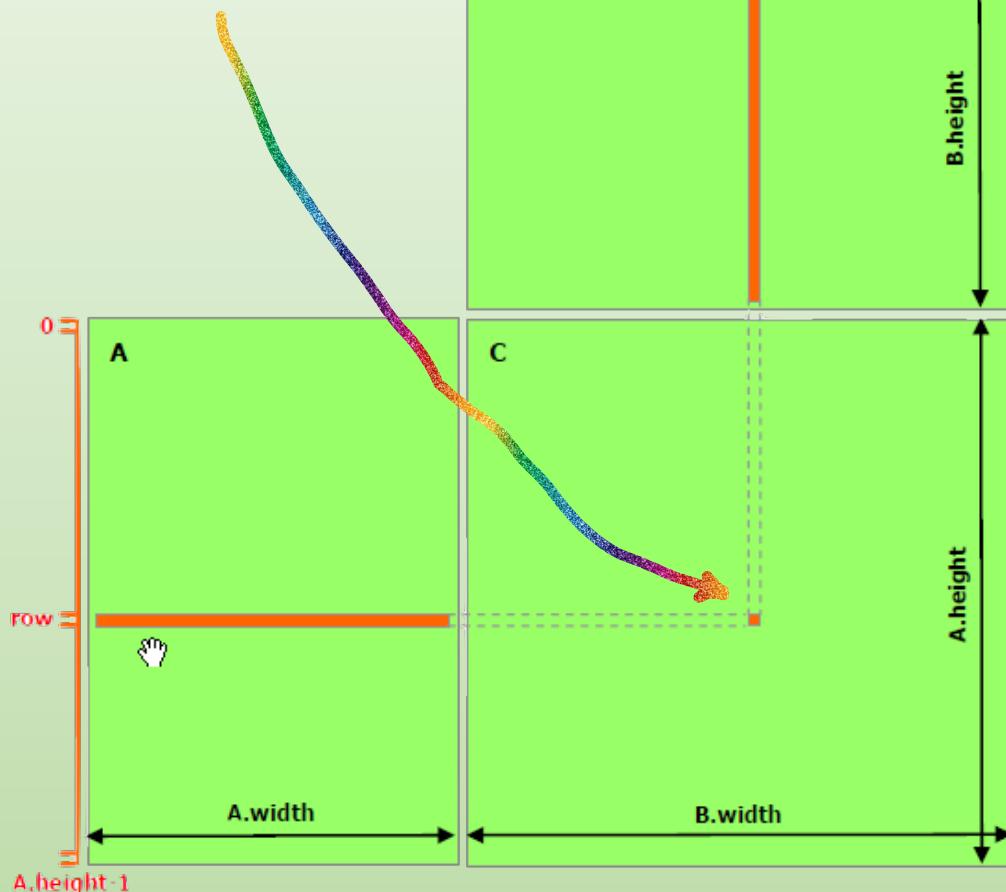
Lab 6

Prodotto di matrici

Con uso di shared memory (SMEM)

Prodotto matriciale (senza SMEM)

punto di vista
di un thread
associato a C



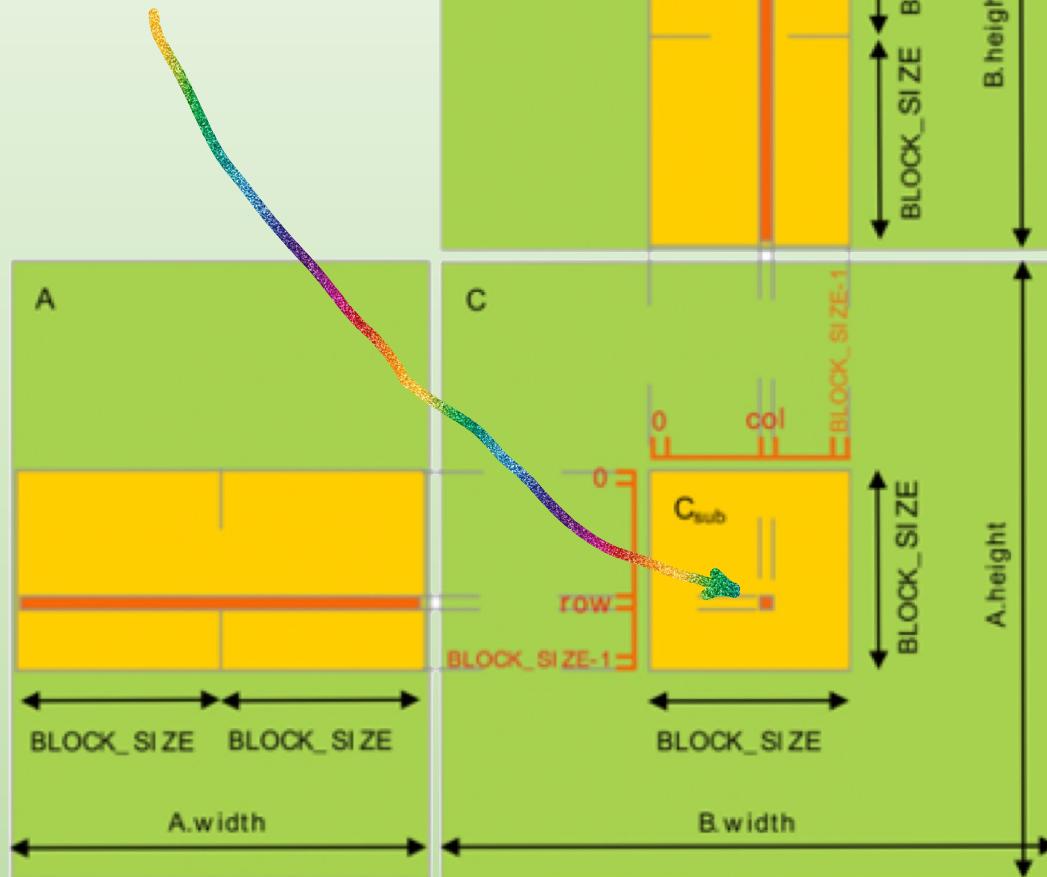
```
/*
 * Kernel for standard (naive) matrix product
 */

__global__ void matProd(mqdb A, mqdb B, mqdb C, int n) {
    // row & col indexes
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // each thread computes an entry of the product matrix
    if ((row < n) && (col < n)) {
        float val = 0;
        for (int k = 0; k < n; k++)
            val += A.elem[row * n + k] * B.elem[k * n + col];
        C.elem[row * n + col] = val;
    }
}
```

Prodotto matriciale (con SMEM)

punto di vista
di un thread
associato a C



1. loop su **row block** di **A** e **col block** di **B**
2. carica **A_{sub}** e **B_{sub}** nella SMEM del thread di **C_{sub}**
3. calcola **prodotto riga-col** in SMEM
4. somma il **contributo parziale** a tutte le entry di **C_{sub}** in **C**

Esercitazione

Prodotto di matrici con SMEM:

Scrivere un programma CUDA per prodotto matrici che usi la SMEM e riduca così il ‘traffico’ in global mem

passi:

1. Definire la SMEM per ogni blocco di C
2. Svolgere un ciclo sui blocchi per caricare la SMEM da global mem
3. Sincronizzare -1-
4. Nel ciclo effettuare localmente all’interno di ogni blocco il calcolo del prodotto riga-colonna e caricare sul registro
5. sincronizzare -2-
6. Scrivere il risultato finale su matrice prodotto in global mem

```
__global__ void mat_prod_shared(float* A, float* B, float* C) {  
    // indexes  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    // block shared memory  
    __shared__ float . . .  
    . . .  
    // uso del registro di accumulo  
    float sum = 0.0;  
    // per il numero di blocchi che si hanno per colonna do A  
    for (ciclo sui blocchi per col/row) {  
  
        // copia del blocco nella shared memory  
        . . .  
        __syncthreads();  
        // calcolo del prodotto in shared memory  
        . . .  
        __syncthreads();  
    }  
    // tutto è fatto: copiare nella matrice C  
    if (row < N && col < M)  
        C[row * M + col] = sum;
```

Algoritmo

Allocazioni preliminari di memoria shared... Il thread di indici **row** e **col** si occupa di calcolare il valore finale **C[row,col]**

LOOP: si carica in memoria shared i dati relativi ad ogni **blocco row** e **blocco col** prima di svolgere le somme e i prodotti relativi al blocco

```
__global__ void mat_prod_shared(float* A, float* B, float* C) {  
    // indexes  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    // block shared memory  
    __shared__ float A_s[BLOCK_SIZE][BLOCK_SIZE];  
    __shared__ float B_s[BLOCK_SIZE][BLOCK_SIZE];  
    . . .  
    . . .  
    // copy block from matrix to SMEM, m = block index  
    int c = m * BLOCK_SIZE + threadIdx.x;  
    int r = m * BLOCK_SIZE + threadIdx.y;  
    A_s[threadIdx.y][threadIdx.x] = A[IDX(row, c, P)];  
    B_s[threadIdx.y][threadIdx.x] = B[IDX(r, col, M)];  
    __syncthreads();
```

Convoluzione

Con uso di shared memory (SMEM)

Convoluzione parallela

```
__global__ void convolution1D(float *result, float *data, int n) {

    // shared memory size = TILE + MASK
    __shared__ float tile[TILE_SIZE];

    // edges
    unsigned int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int left = blockIdx.x * blockDim.x - MASK_RADIUS;
    int right = (blockIdx.x + 1) * blockDim.x;

    if (threadIdx.x < MASK_RADIUS)                                // left
        tile[threadIdx.x] = left < 0 ? 0 : data[left + threadIdx.x];
    else if (threadIdx.x >= blockDim.x - MASK_RADIUS) // right
        tile[threadIdx.x + MASK_SIZE - 1] = right >= n ? 0 :
            data[right + threadIdx.x - blockDim.x + MASK_RADIUS];
    . . .
}
```

Convoluzione parallela 1D

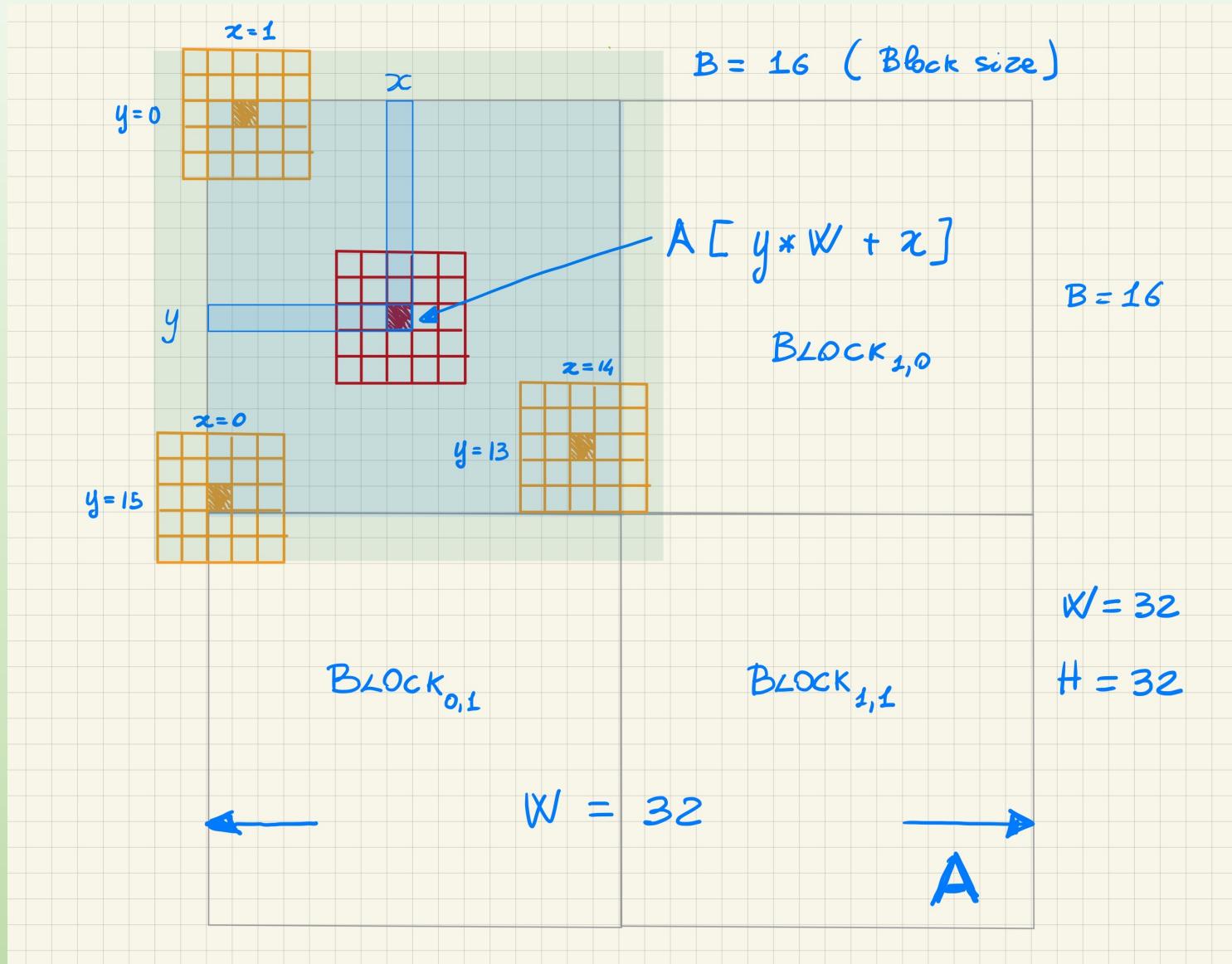
```
. . .
    // center
    tile[threadIdx.x + MASK_RADIUS] = data[idx];

    __syncthreads();

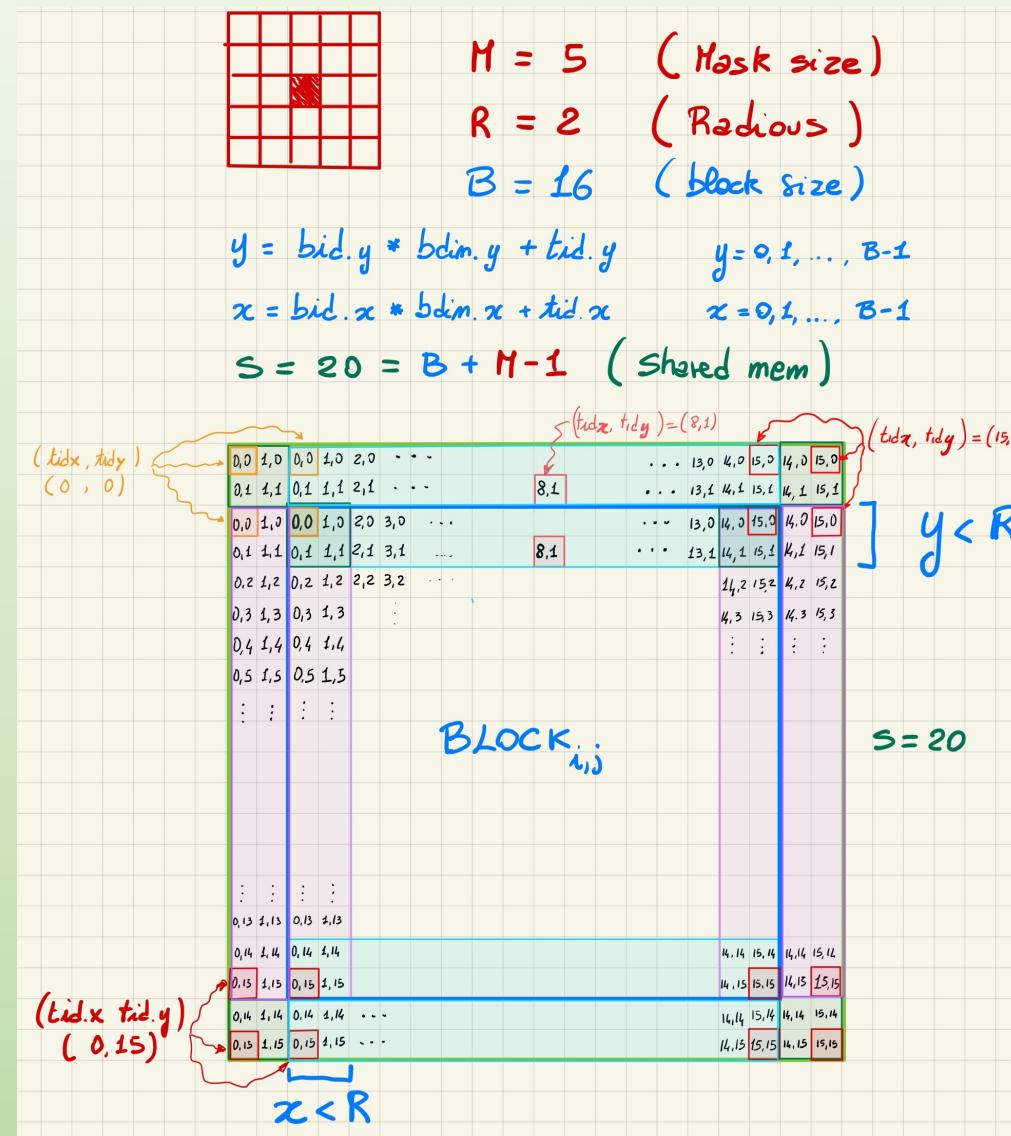
    // convoluzione: tile * mask
    float sum = 0;
    for (int i = -MASK_RADIUS; i <= MASK_RADIUS; i++)
        sum += tile[threadIdx.x + MASK_RADIUS + i] * d_mask[i + MASK_RADIUS];

    // store conv result
    result[idx] = sum;
}
```

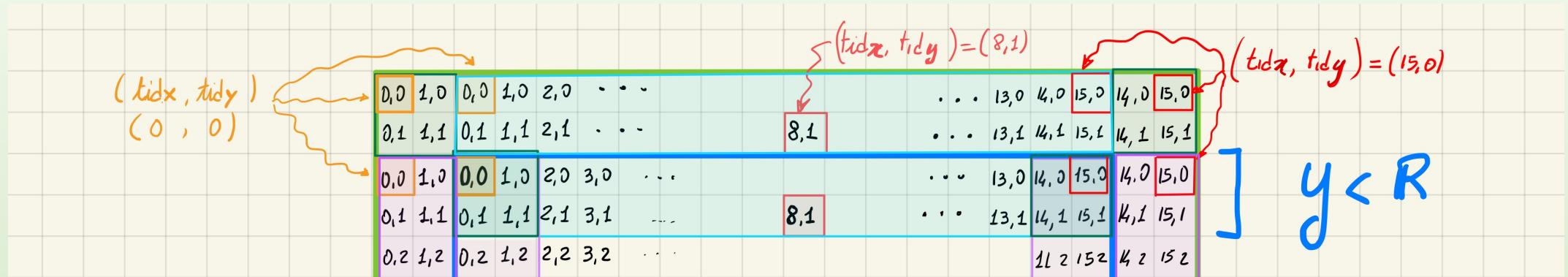
Conv 2D: schema con SMEM



Blocco e SMEM



Upper halo



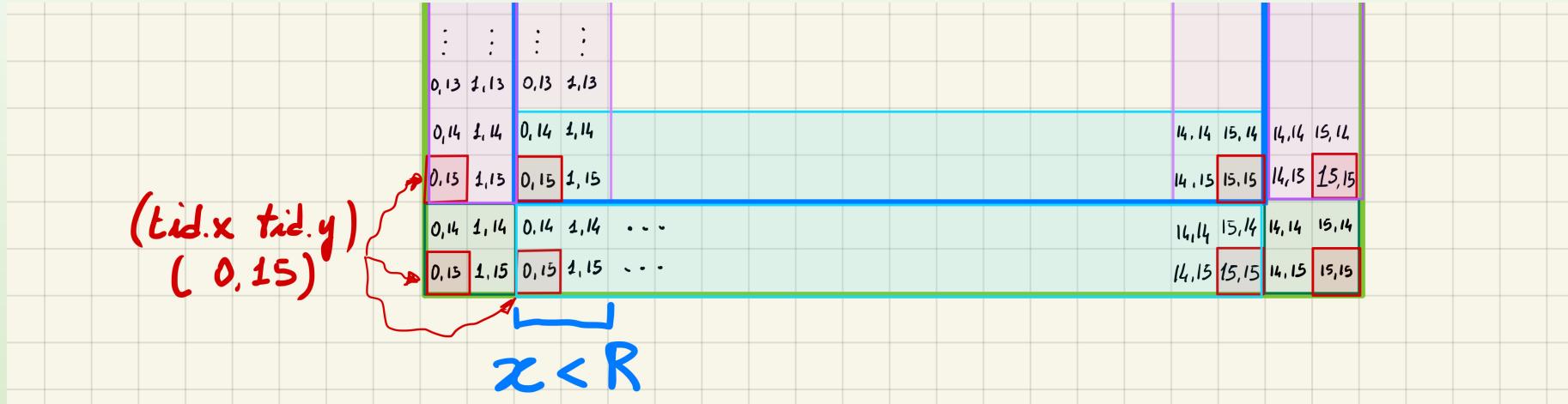
```
// 1. copy the tile upper halo
if ((threadIdx.y < RAD) ) {

    // left corner
    if (threadIdx.x < RAD && (x-RAD) >= 0 && (y-RAD) >= 0)
        A_s[threadIdx.y][threadIdx.x] = A[(y-RAD) * W + x - RAD];

    // right corner
    if (threadIdx.x >= BmR && (x+RAD) < W && (y-RAD) >= 0)
        A_s[threadIdx.y][threadIdx.x + 2*RAD] = A[(y-RAD) * W + x + RAD];

    // edge
    if ((y-RAD) >= 0)
        A_s[threadIdx.y][threadIdx.x + RAD] = A[(y-RAD) * W + x ];
}
```

Bottom halo



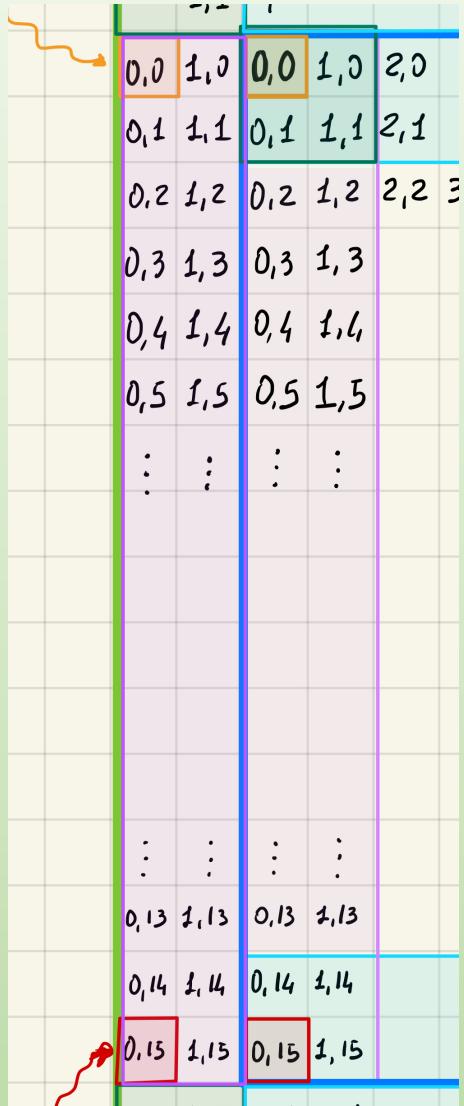
```
// 2. copy the tile bottom halo
if (threadIdx.y >= BmR) {

    // left corner
    if (threadIdx.x < RAD && (x-RAD) >= 0 && (y+RAD) < H)
        A_s[threadIdx.y + 2*RAD][threadIdx.x] = A[(y+RAD) * W + x - RAD];

    // right corner
    if (threadIdx.x >= BmR && (y+RAD) < H)
        A_s[threadIdx.y + 2*RAD][threadIdx.x + 2*RAD] = A[(y+RAD) * W + x + RAD];

    // edge
    if ((y+RAD) < H)
        A_s[threadIdx.y + 2*RAD][threadIdx.x + RAD] = A[(y+RAD) * W + x];
}
```

Lati e centro

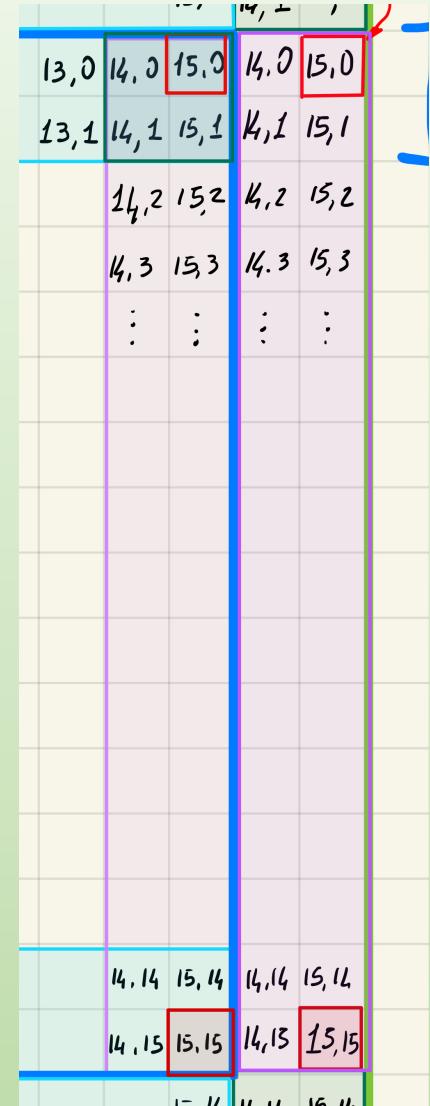


0,0	1,0	0,0	1,0	2,0
0,1	1,1	0,1	1,1	2,1
0,2	1,2	0,2	1,2	2,2
0,3	1,3	0,3	1,3	
0,4	1,4	0,4	1,4	
0,5	1,5	0,5	1,5	
:	:	:	:	
0,13	1,13	0,13	1,13	
0,14	1,14	0,14	1,14	
0,15	1,15	0,15	1,15	

```
// 3. copy the tile left-edge halo
if (threadIdx.x < RAD)
    // edge
    if ((x-RAD) >= 0)
        A_s[threadIdx.y + RAD][threadIdx.x] = A[y * W + x - RAD];

// 4. copy the tile right-edge halo
if (threadIdx.x >= BmR)
    // edge
    if ((x+RAD) < W)
        A_s[threadIdx.y + RAD][threadIdx.x + 2*RAD] = A[y * W + x + RAD];

// 5. copy the tile center <-> block
A_s[RAD + threadIdx.y][RAD + threadIdx.x] = A[y*W+x];
```



13,0	14,0	15,0	14,0	15,0
13,1	14,1	15,1	14,1	15,1
14,2	15,2	14,2	15,2	
14,3	15,3	14,3	15,3	
:	:	:	:	
14,14	15,14	14,14	15,14	
14,15	15,15	14,15	15,15	
15,15	15,15	15,15	15,15	

Blocchi e halo

Block 0,0

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	3	6	7	5	3	5	6	2	9	1
0	0	2	0	2	3	7	5	9	2	2	8
0	0	5	0	3	6	1	0	6	3	2	0
0	0	2	5	4	7	4	4	3	0	7	8
0	0	6	6	4	9	5	0	4	8	7	1
0	0	0	9	1	7	7	1	1	5	9	7
0	0	3	9	0	8	8	5	0	9	6	3
0	0	4	4	4	4	7	6	3	1	7	5
0	0	5	3	8	8	3	1	8	9	6	4
0	0	4	3	0	3	0	9	2	5	4	0

Block $_{1,1}$

0	9	6	3	8	5	6	1	1	5	9	8
3	1	7	5	9	6	2	1	7	8	5	7
8	9	6	4	3	3	3	8	6	0	4	8
2	5	4	0	5	9	4	6	9	2	2	4
0	2	1	0	5	1	1	0	8	5	0	6
0	6	2	9	9	0	8	1	3	1	1	0
0	5	6	6	4	0	0	4	6	2	6	7
2	7	6	1	3	2	1	5	9	9	1	4
2	9	6	1	0	4	2	2	2	0	5	5
9	6	2	5	4	4	9	9	3	6	0	5
6	9	4	2	2	6	4	1	2	8	8	9
0	4	7	6	8	9	0	6	8	7	9	0

Print matrix A...

3	6	7	5	3	5	6	2	9	1	2	7	0	9	3	6	0	6	2	6	1	8	7	9
2	0	2	3	7	5	9	2	2	8	9	7	3	6	1	2	9	3	1	9	4	7	8	4
5	0	3	6	1	0	6	3	2	0	6	1	5	5	4	7	6	5	6	9	3	7	4	5
2	5	4	7	4	4	3	0	7	8	6	8	8	4	3	1	4	9	2	0	6	8	9	2
6	6	4	9	5	0	4	8	7	1	7	2	7	2	2	6	1	0	6	1	5	9	4	9
0	9	1	7	7	1	1	5	9	7	7	6	7	3	6	5	6	3	9	4	8	1	2	9
3	9	0	8	8	5	0	9	6	3	8	5	6	1	1	5	9	8	4	8	1	0	3	0
4	4	4	4	7	6	3	1	7	5	9	6	2	1	7	8	5	7	4	1	8	5	9	7

Block 2,0

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
3	6	0	6	2	6	1	8	7	9	0	0
1	2	9	3	1	9	4	7	8	4	0	0
4	7	6	5	6	9	3	7	4	5	0	0
3	1	4	9	2	0	6	8	9	2	0	0
2	6	1	0	6	1	5	9	4	9	0	0
6	5	6	3	9	4	8	1	2	9	0	0
1	5	9	8	4	8	1	0	3	0	0	0
7	8	5	7	4	1	8	5	9	7	0	0
6	0	4	8	8	8	9	7	7	6	4	3
9	2	2	4	7	7	5	4	8	1	2	8

Block 2,2

5	3	8	8	3	1	8	9	6	4	3	3	3	8	6	0	4	8	8	8	9	7	7	6
4	3	0	3	0	9	2	5	4	0	5	9	4	6	9	2	2	4	7	7	5	4	8	1
2	8	9	3	6	8	0	2	1	0	5	1	1	0	8	5	0	6	4	6	2	5	8	6
2	8	4	7	2	4	0	6	2	9	9	0	8	1	3	1	1	0	3	4	0	3	9	1
9	6	9	3	3	8	0	5	6	6	4	0	0	4	6	2	6	7	5	6	9	8	7	2
8	2	9	9	6	0	2	7	6	1	3	2	1	5	9	9	1	4	9	1	0	7	5	8
7	0	4	8	0	4	2	9	6	1	0	4	2	2	2	0	5	5	2	9	0	2	8	3
8	0	4	0	9	1	9	6	2	5	4	4	9	9	3	6	0	5	0	2	9	4	3	5
1	7	4	3	1	4	6	9	4	2	2	6	4	1	2	8	8	9	2	8	8	8	6	8
3	8	3	3	3	8	0	4	7	6	8	9	0	6	8	7	9	0	3	3	3	7	3	2
6	5	2	6	5	8	7	9	6	0	4	1	0	4	8	7	0	8	6	2	4	7	9	3
9	2	8	3	0	1	7	8	9	1	5	4	9	2	5	7	4	9	9	4	5	9	3	5
7	0	8	1	9	9	7	8	2	5	3	4	9	0	2	0	1	9	6	2	1	2	0	7
3	1	1	9	0	5	6	7	7	4	0	6	4	7	4	8	5	8	2	6	0	6	6	4
6	2	8	9	6	0	7	0	1	0	1	3	7	7	2	6	4	5	2	0	2	9	0	9
9	4	5	1	0	3	7	8	8	6	0	4	6	7	6	0	9	7	5	9	7	8	5	3

Esercitazione

Convoluzione 1D/2D con SMEM e constant mem:

Scrivere un programma CUDA per la convoluzione 1D/ 2D che usi la SMEM per i dati e la constant mem per la maschera del filtro
passi:

1. Definire la SMEM per ogni blocco di dimensione legata alla tile size
2. Caricare la constant memory con i coefficienti del filtro da host (pre kernel)
3. Nel kernel: caricare la SMEM da global mem
4. Sincronizzare -1-
5. Effettuare localmente il prodotto di convoluzione sfruttando i dati in cache
6. Scrivere il risultato finale su vettore/matrice in global mem

```
__global__ void convolution1D( . . . ) {  
  
    // shared memory size = TILE + MASK  
    __shared__ . . .  
  
    // gestione dei bordi  
    . . .  
  
    // caricamento della SMEM  
    . . .  
  
    __syncthreads();  
  
    // convoluzione: tile * mask  
    float sum += . . . // caricare ne registro  
  
    // risultato finale  
    result[idx] = sum;  
}
```

Convoluzione parallela 2D

```
/*
 * kernel for convolution 2D
 */
__global__ void conv2D(float *A, float *B) {

    const int TILE_SIZE = TILE_WIDTH + MASK_WIDTH - 1;

    __shared__ float A_s[TILE_SIZE][TILE_SIZE];

    // SHARED MEMORY BLOCK

    . . .

    __syncthreads();

    float Pvalue = 0;
    for (int i = 0; i < MASK_WIDTH; i++)
        for (int j = 0; j < MASK_WIDTH; j++)
            Pvalue += A_s[threadIdx.y + i][threadIdx.x + j] * M_dev[i][j];
    B[x * blockDim.x + y
}
```