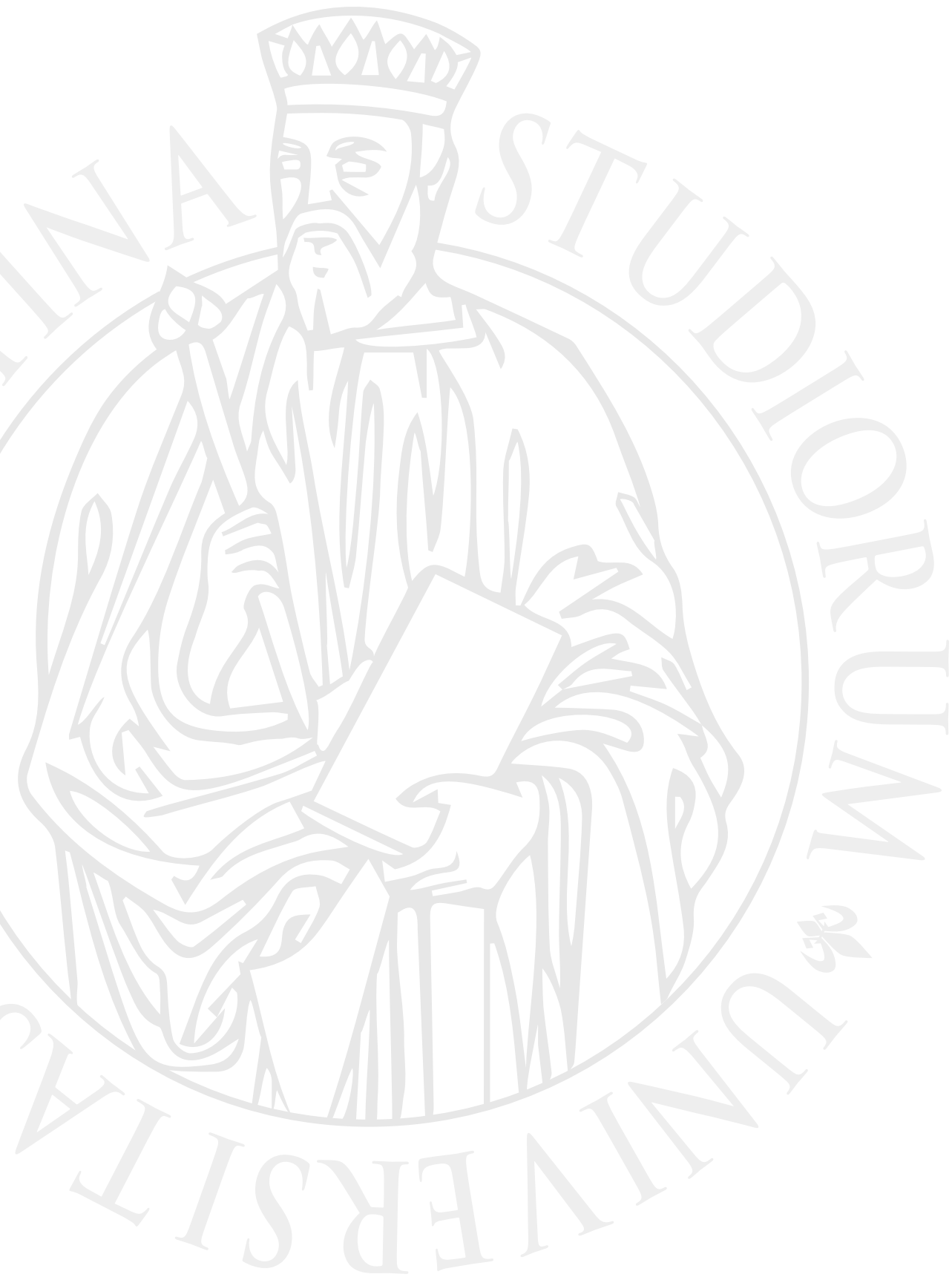




UNIVERSITÀ
DEGLI STUDI
FIRENZE

GPU programming basics

Prof. Marco Bertini





Data parallelism: GPU computing



2D convolution: 2-batch loading

Upload data in shared memory for convolution kernel

- The image is divided into tiles.
- These tiles after applying the convolution mask are the final output tiles whose size is $TILE_WIDTH * TILE_WIDTH$.
- For the pixels that belong to the border of the output tile the mask must borrow some pixels from the neighbor tile, when this tile belong to the borders of the image. Otherwise, these borrowed values are assigned to zero.



2-batch loading

- Let us assume that a block is made of $TILE_WIDTH * TILE_WIDTH$ threads
- Since the shared memory area $(TILE_WIDTH + Mask_width - 1) * (TILE_WIDTH + Mask_width - 1)$ is larger than the block size $TILE_WIDTH * TILE_WIDTH$ and assuming it is smaller than $2 * TILE_WIDTH * TILE_WIDTH$, then each thread should move at most two elements from global memory to shared memory.
- It is convenient to split this loads in a two-stages process.

First step

- Load $TILE_WIDT * TILE_WIDTH$ elements:

```
dest = threadIdx.y * TILE_WIDTH +  
      threadIdx.x;
```

- flattens the 2D coordinates of the generic thread while

```
destX = dest % w;
```

```
destY = dest / w;
```

- makes the inverse operation, calculating the 2D coordinates of the generic thread with respect to the shared memory area.

First step

```
srcY = blockIdx.y * TILE_WIDTH + destY -  
      Mask_radius;
```

```
srcX = blockIdx.x * TILE_WIDTH + destX -  
      Mask_radius;
```

- $(\text{blockIdx.x} * \text{TILE_WIDTH}, \text{blockIdx.y} * \text{TILE_WIDTH})$ would be the coordinates of the global memory location if the block size and the shared memory size were the same.
- Since you are "borrowing" memory values also from neighbor tiles, then you have to shift the above coordinates by $(\text{destX} - \text{Mask_radius}, \text{destY} - \text{Mask_radius})$.

First step

```
// First batch loading
int dest = threadIdx.y * TILE_WIDTH + threadIdx.x;
int destY = dest / w;
int destX = dest % w;
int srcY = blockIdx.y * TILE_WIDTH + destY -
           Mask_radius;
int srcX = blockIdx.x * TILE_WIDTH + destX -
           Mask_radius;
int src = (srcY * width + srcX) * channels + k;
if (srcY >= 0 && srcY < height &&
    srcX >= 0 && srcX < width) {
    N_ds[destY][destX] = I[src];
} else {
    N_ds[destY][destX] = 0;
}
```

First step

```
// First batch loading
int dest = threadIdx.y * TILE_WIDTH + threadIdx.x;
int destY = dest / w;
int destX = dest % w;
int srcY = blockIdx.y * TILE_WIDTH + destY -
           Mask_radius;
int srcX = blockIdx.x * TILE_WIDTH + destX -
           Mask_radius;
int src = (srcY * width + srcX) * channels + k;
if (srcY >= 0 && srcY < height &&
    srcX >= 0 && srcX < width) {
    N_ds[destY][destX] = I[src];
} else {
    N_ds[destY][destX] = 0;
}
```

Running inside a
for (k = 0; k < channels; k++) { ... }

Second step

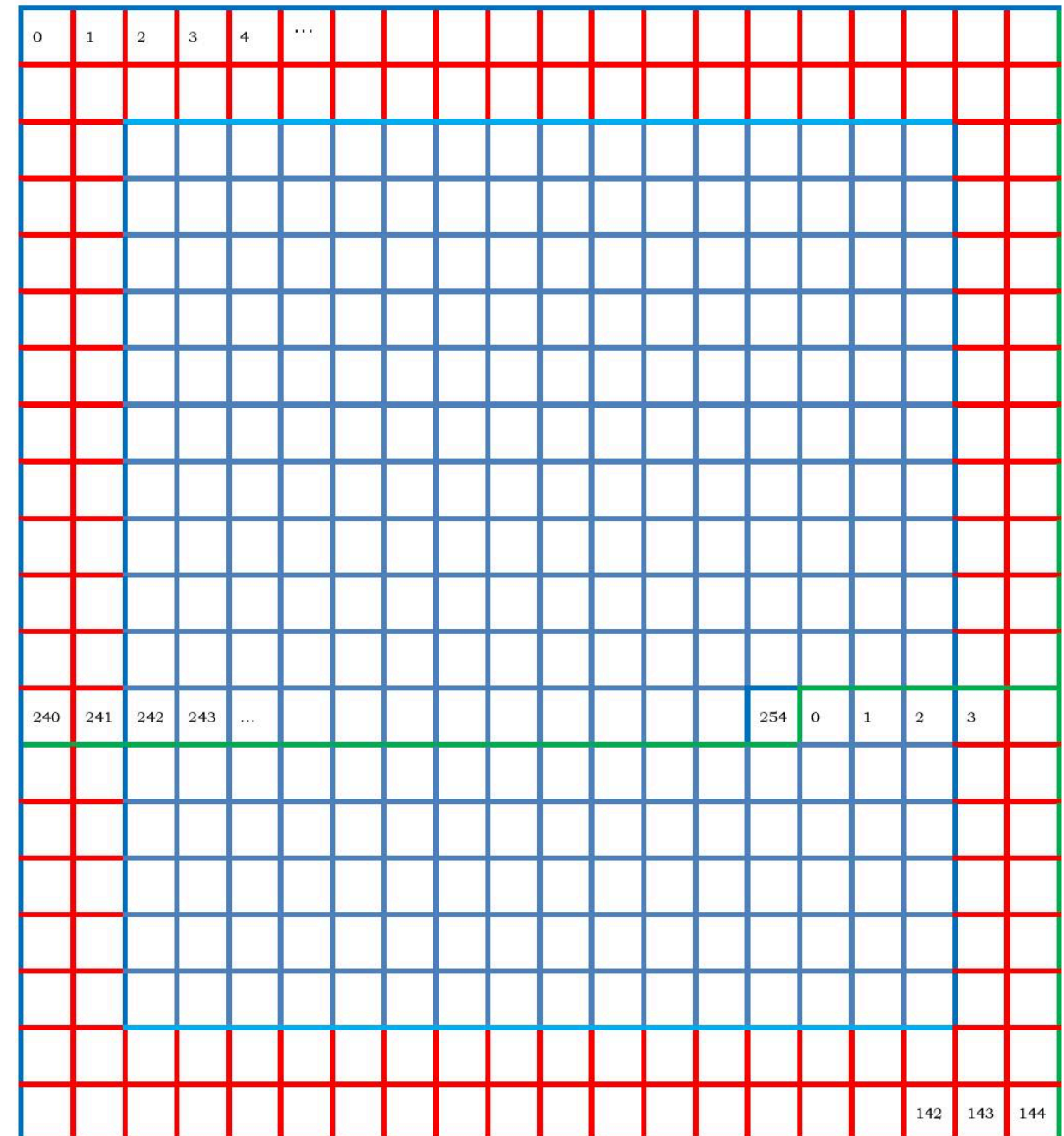
- Load the data outside the $TILE_WIDTH * TILE_WIDTH$
- Similar to first batch but now offset with $TILE_WIDTH * TILE_WIDTH$

```
dest = threadIdx.y * TILE_WIDTH +  
        threadIdx.x +  
        TILE_WIDTH * TILE_WIDTH;
```

- destY, destX, srcY and srcX use the same formulas

Second step

- The picture illustrates the correspondence between the flattened thread index `dest` and the shared memory locations.
- In the picture, the **blue** boxes represent the elements of the generic tile while the **red** boxes the elements of the neighbor tiles. The union of the blue and red boxes correspond to the overall shared memory locations. With `TILE_WIDTH=16`, all the 256 threads of a thread block are involved in filling the upper part of the shared memory above the **green** line, while 145 are involved in filling the lower part of the shared memory below the green line. So these threads participate to the second batch loading (with `TILE_WIDTH x TILE_WIDTH` offset).
- Notice that we have at most 2 memory loads per thread due to the particular choice of parameters. For example, using `TILE_WIDTH = 8`, results in number of threads per block of 64, while the shared memory size is $12 \times 12 = 144$, which means that each thread is in charge to perform at least 2 shared memory writes since $144/64 = 2.25$.



Second step

```
// Second batch loading
dest = threadIdx.y * TILE_WIDTH + threadIdx.x +
      TILE_WIDTH * TILE_WIDTH;
destY = dest / w;
destX = dest % w;
srcY = blockIdx.y * TILE_WIDTH + destY - Mask_radius;
srcX = blockIdx.x * TILE_WIDTH + destX - Mask_radius;
src = (srcY * width + srcX) * channels + k;
if (destY < w) {
    if (srcY >= 0 && srcY < height &&
        srcX >= 0 && srcX < width) {
        N_ds[destY][destX] = I[src];
    } else {
        N_ds[destY][destX] = 0;
    }
}
```

Second step

```
// Second batch loading
dest = threadIdx.y * TILE_WIDTH + threadIdx.x +
      TILE_WIDTH * TILE_WIDTH;
destY = dest / w;
destX = dest % w;
srcY = blockIdx.y * TILE_WIDTH + destY - Mask_radius;
srcX = blockIdx.x * TILE_WIDTH + destX - Mask_radius;
src = (srcY * width + srcX) * channels + k;
if (destY < w) {
    if (srcY >= 0 && srcY < height &&
        srcX >= 0 && srcX < width) {
        N_ds[destY][destX] = I[src];
    } else {
        N_ds[destY][destX] = 0;
    }
}
```

Running inside a
for (k = 0; k < channels; k++) { ... }



2D convolution: tile boundaries

2D Image Matrix with Automated Padding

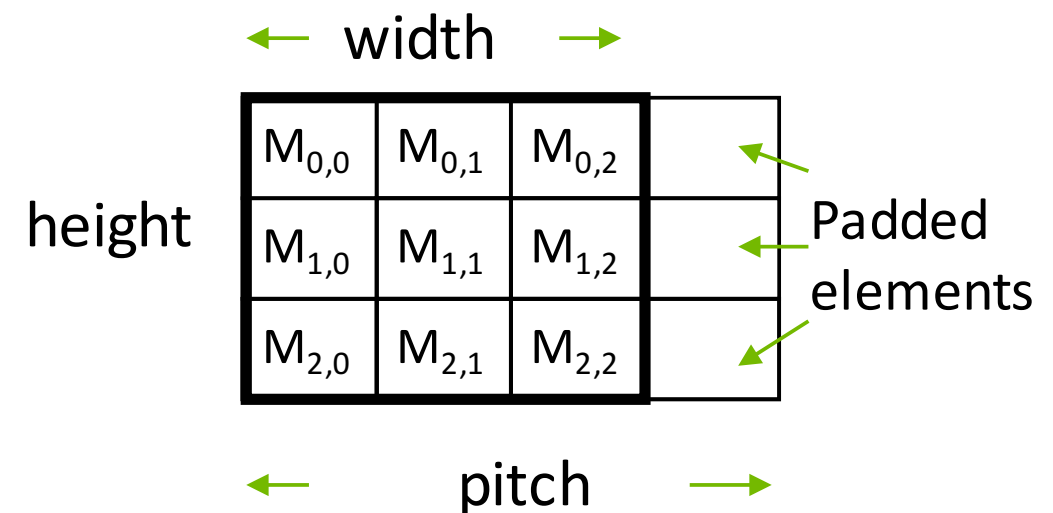
- It is sometimes desirable to pad each row of a 2D matrix to multiples of DRAM bursts
 - So each row starts at the DRAM burst boundary
 - Effectively adding columns
 - This is usually done automatically by matrix allocation function
 - Pitch can be different for different hardware
- Example: a 3X3 matrix padded into a 3X4 matrix

Height is 3

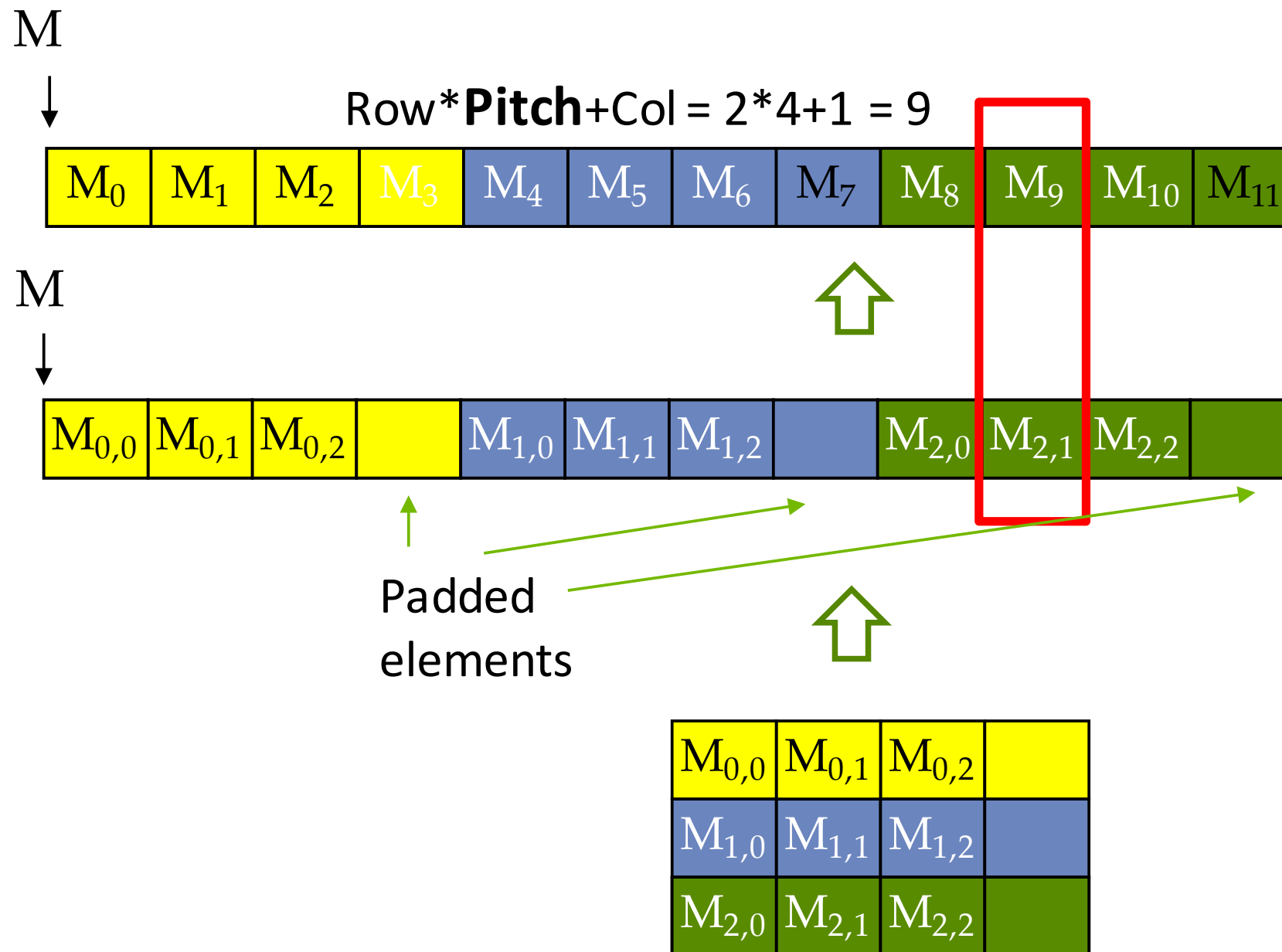
Width is 3

Channels is 1 (e.g. gray level image)

Pitch is 4



Row-Major Layout with Pitch



Sample image struct

```
// Image Matrix Structure declaration
```

```
//
```

```
typedef struct {
```

```
    int width;
```

```
    int height;
```

```
    int pitch;
```

```
    int channels;
```

```
    float* data;
```

```
} Image_t;
```

Setting Block Size

```
#define O_TILE_WIDTH 12
```

```
#define BLOCK_WIDTH (O_TILE_WIDTH + 4)
```

```
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);
```

```
dim3 dimGrid((Image_Width-1)/O_TILE_WIDTH+1,  
(Image_Height-1)/O_TILE_WIDTH+1, 1)
```

- In general, BLOCK_WIDTH should be
- $O_TILE_WIDTH + (MASK_WIDTH-1)$

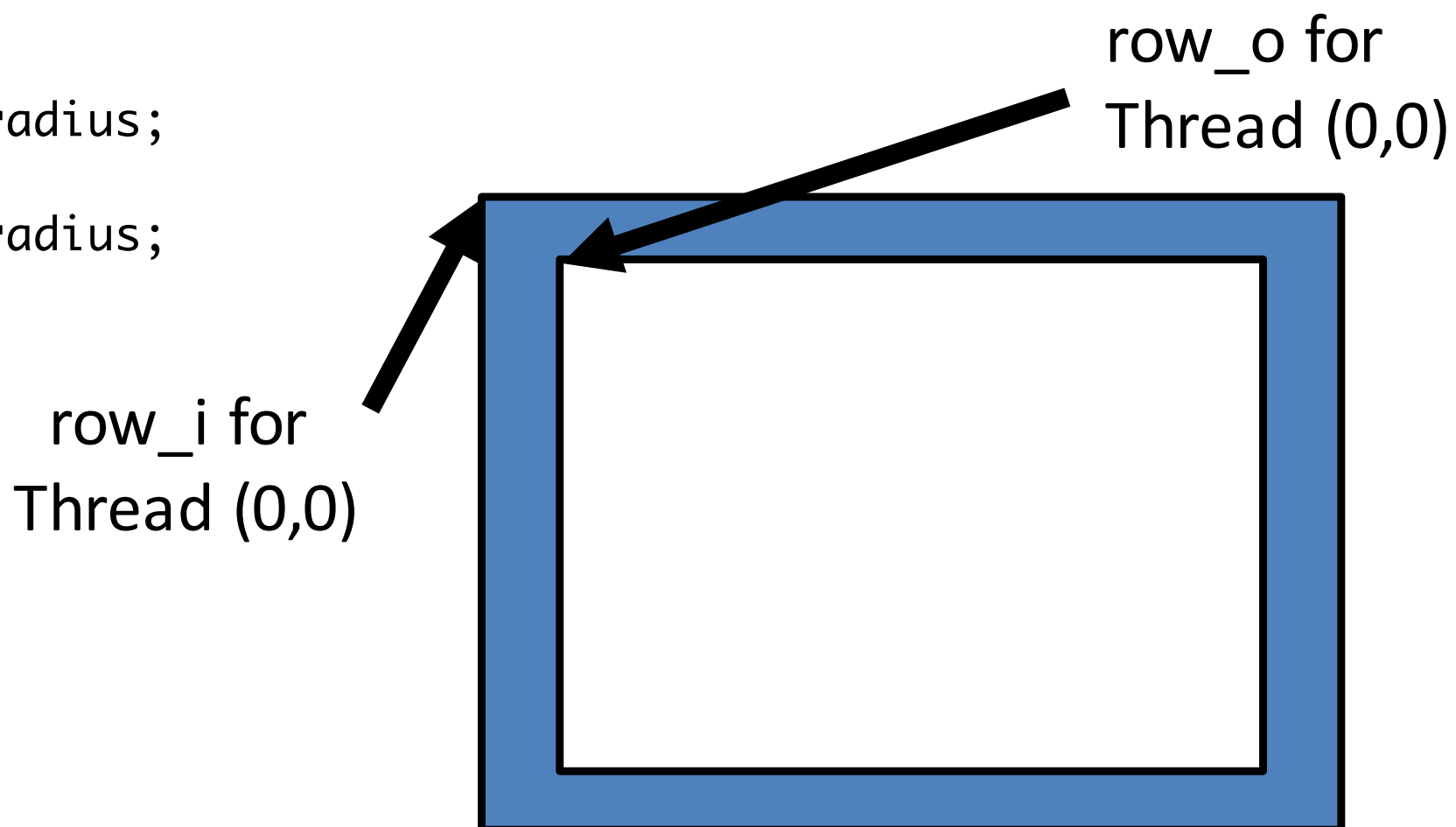
Using constant memory and caching for Mask

- Mask is used by all threads but not modified in the convolution kernel
 - All threads in a warp access the same locations at each point in time
- CUDA devices provide constant memory whose contents are aggressively cached
 - Cached values are broadcast to all threads in a warp
 - Effectively magnifies memory bandwidth without consuming shared memory
- Use of `const __restrict__` qualifiers for the mask parameter informs the compiler that it is eligible for constant caching, for example:

```
__global__ void convolution_2D_kernel(float *P, float *N, int height, int width, int channels, const float __restrict__ *M);
```

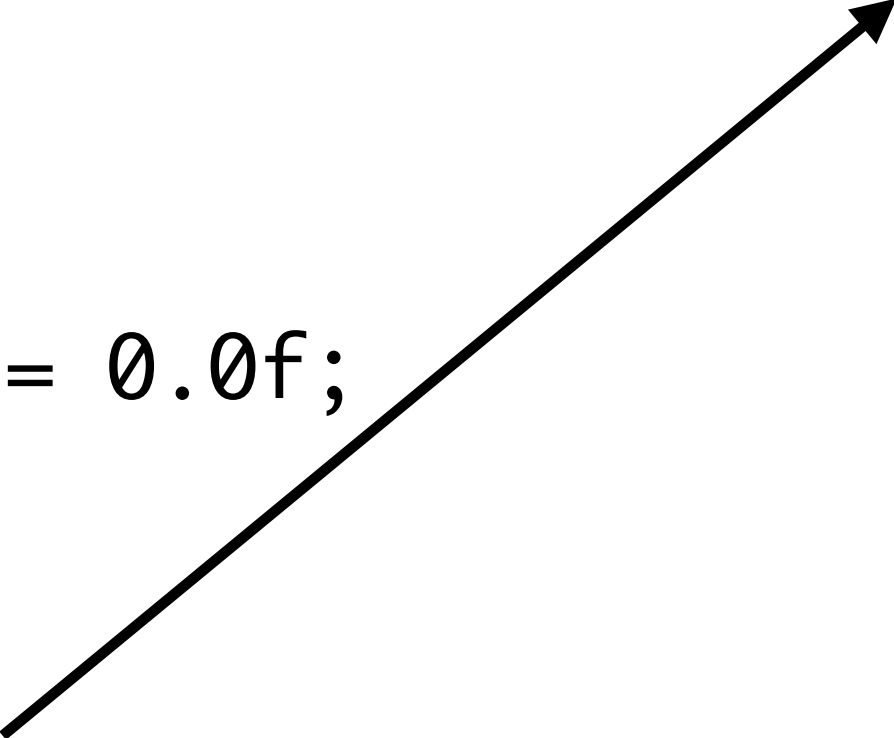
Shifting from output coordinates to input coordinate

```
int tx = threadIdx.x;  
int ty = threadIdx.y;  
int row_o = blockIdx.y*O_TILE_WIDTH + ty;  
int col_o = blockIdx.x*O_TILE_WIDTH + tx;  
  
int row_i = row_o - mask_radius;  
int col_i = col_o - mask_radius;
```



Taking Care of Boundaries (1 channel example)

```
if((row_i >= 0) && (row_i < height) &&
    (col_i >= 0) && (col_i < width)) {
    Ns[ty][tx] = data[row_i * width + col_i];
} else{
    Ns[ty][tx] = 0.0f;
}
```



- Use of width here is OK if pitch is set to width (no padding)

Calculating output

Some threads do not participate in calculating output

```
float output = 0.0f;
```

```
if(ty < 0_TILE_WIDTH && tx < 0_TILE_WIDTH){
```

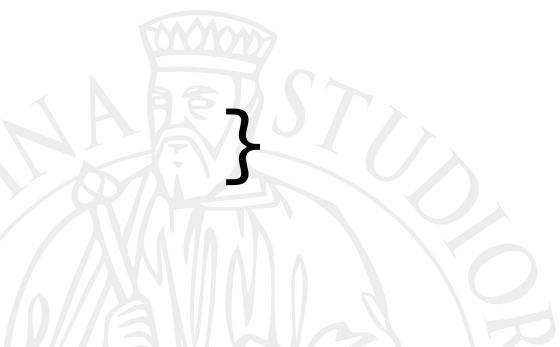
```
    for(i = 0; i < MASK_WIDTH; i++) {
```

```
        for(j = 0; j < MASK_WIDTH; j++) {
```

```
            output += M[i][j] * Ns[i+ty][j+tx];
```

```
        }
```

```
    }
```



Writing output

- Some threads do not write output (1 channel example)

```
if(row_o < height && col_o < width)
```

```
    data[row_o*width + col_o] =  
        output;
```




Credits

- These slides report material from:
 - NVIDIA GPU Teaching Kit



Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 2nd edition - Chapt. 8

or

Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 3rd edition - Chapt. 7