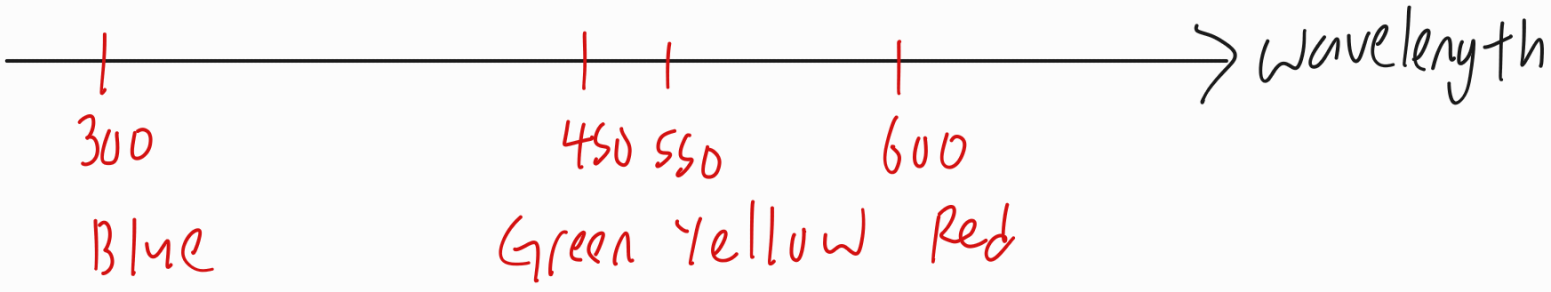
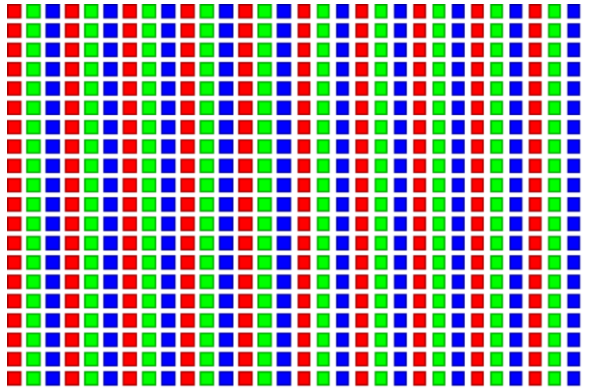
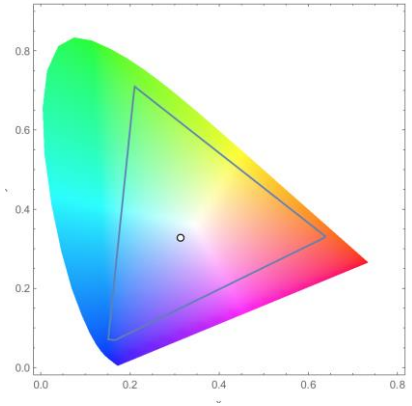


We can use RGB to represent any color

RGB COLOR IMAGE REPRESENTATION



- Each pixel in an image is an RGB value
- The format of an image's row is
(r g b) (r g b) … (r g b)
- RGB ranges are not distributed uniformly
- Many different color spaces, here we show the constants to convert to AdbobeRGB color space
  - The vertical axis (y value) and horizontal axis (x value) show the fraction of the pixel intensity that should be allocated to G and B. The remaining fraction (1-y–x) of the pixel intensity that should be assigned to R
  - The triangle contains all the representable colors in this color space

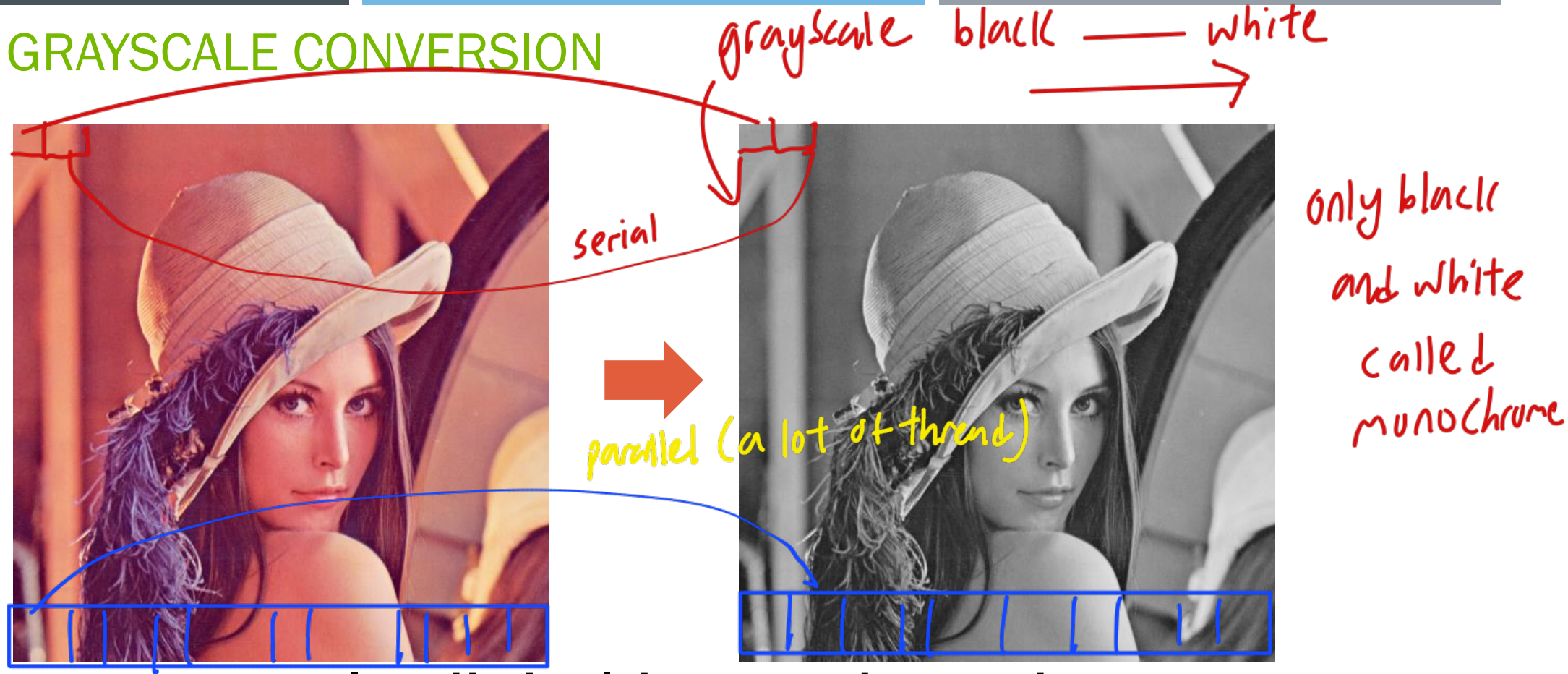


01_ToGrayScale with Cuda

```cpp
__global__ void to_grayscale_kernel(uint8_t *gray_img, uint8_t *bgr_img,
                                    int width, int height, int channels) {
    // Add your code here to turn the color image into grayscale image
    // Note: gray = 0.07 * blue + 0.71 * green + 0.21 * red
    // uint8_t = unsigned 8-bit integer type (Range: 0 - 255)
    // static_cast<uint8_t>(value) -> // Explicitly cast to uint8_t

    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    if (x < width && y < height){
      int gray_idx = y * width + x;
      int color_idx = gray_idx * channels;

      // Get BGR pixel
      uint8_t blue = bgr_img[color_idx];
      uint8_t green = bgr_img[color_idx + 1];
      uint8_t red = bgr_img[color_idx + 2];

      gray_img[gray_idx] = static_cast<uint8_t>(0.07 * blue + 0.71 * green + 0.21 * red);
    }
}
```

```cpp
const int thread_per_blk = 32;      // Use 32 threads per block for x and y
void cuda_color_to_grayscale(uint8_t *gray_img, uint8_t *color_img,
                             int width, int height, int channels) {
    uint8_t* cData;       // Color image data for device
    uint8_t* gData;       // Grayscale image data for device

    // Compute the image data size for color and grayscale in bytes. Note that
    // the gray scale image has only one channel (one color), so the size is
    // 3 times smaller.
    const size_t grayDataSize = sizeof(uint8_t) * width * height;
    const size_t colorDataSize = grayDataSize * channels;

    // Allocate memory for gData and cData with the size of grayDataSize
    // and colorDataSize, respectively
    cudaMalloc((void**)&cData, colorDataSize);
    cudaMalloc((void**)&gData, grayDataSize);

    // Transfer BGR image from host (color_img) to device (cData). The size
    // should be colorDataSize
    cudaMemcpy(cData, color_img, colorDataSize, cudaMemcpyHostToDevice);

    // Calculate blocksize and gridsize, use the recommended 'thread_per_blk'
    dim3 blockSize(thread_per_blk, thread_per_blk, 1);
    dim3 gridSize((width + thread_per_blk - 1) / thread_per_blk,
                  (height + thread_per_blk - 1) / thread_per_blk);

    // Call the GPU kernel
    to_grayscale_kernel <<<gridSize, blockSize>>> (gData, cData, width, height, channels);

    // Transfer Grayscale image from device (gData) to host (gray_img). The
    // size should be colorDataSize
    cudaMemcpy(gray_img, gData, grayDataSize, cudaMemcpyDeviceToHost);

    // Free the allocated device memory
    cudaFree(cData);
    cudaFree(gData);
}
```

RGB TO GRAYSCALE CONVERSION



A grayscale digital image is an image in which the value of each pixel carries only intensity information.

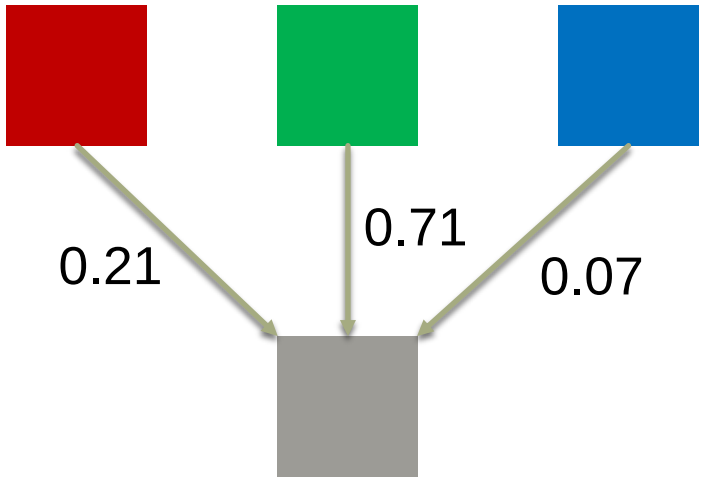
COLOR CALCULATING FORMULA

grayscale

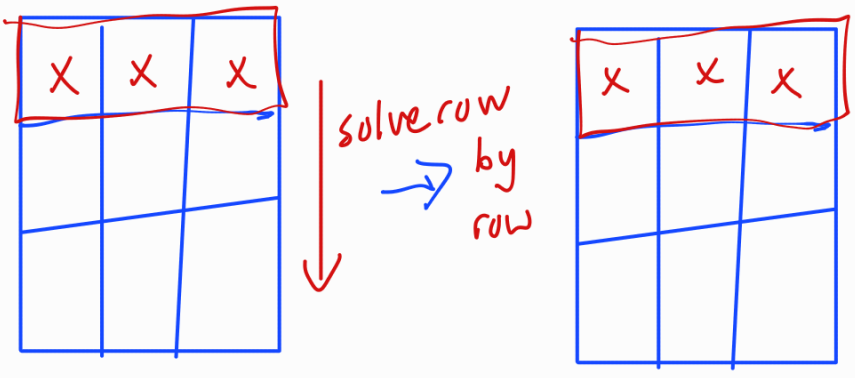
- For each pixel (r g b) at (I, J) do:

grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b

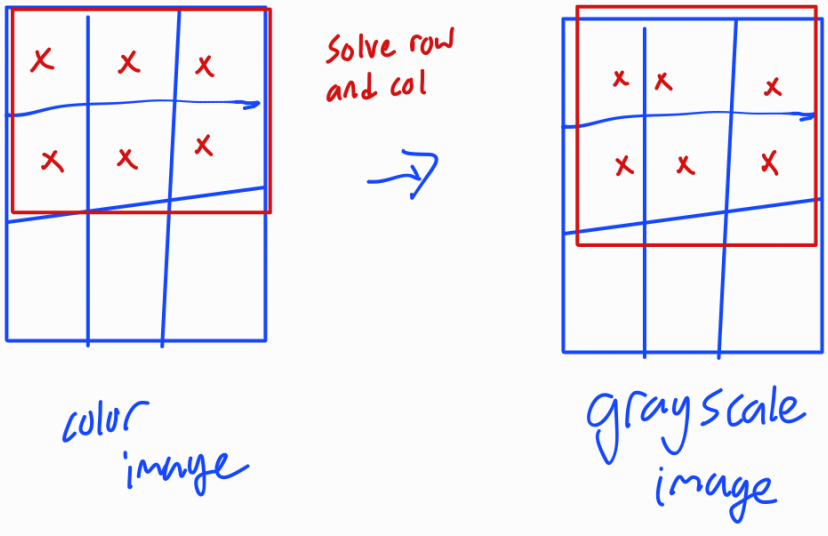
- This is just a dot product <[r,g,b],[0.21,0.71,0.07]> with the constants being specific to input RGB space



1D parallel

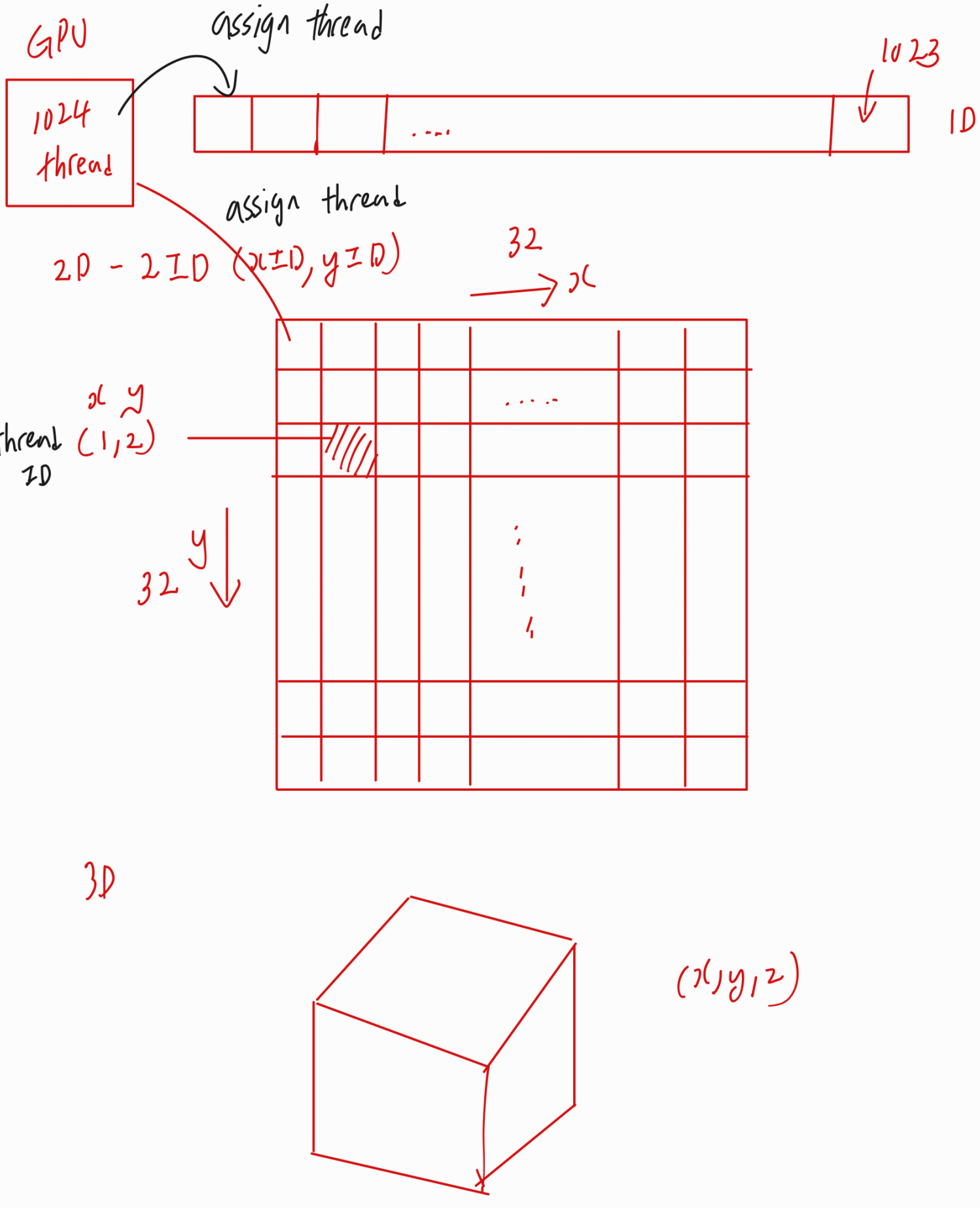


2D parallel



GPU

1D - 1ID (xID)



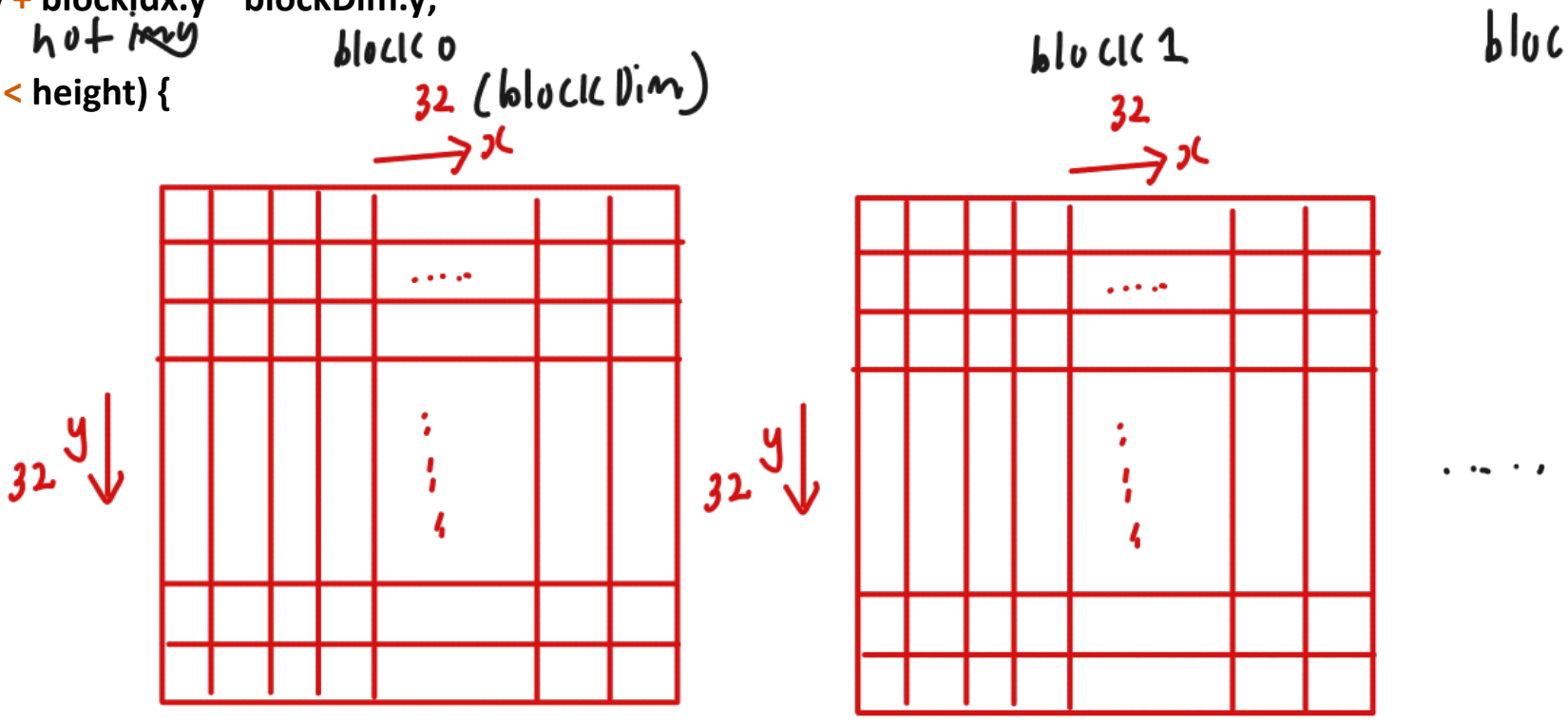
3D

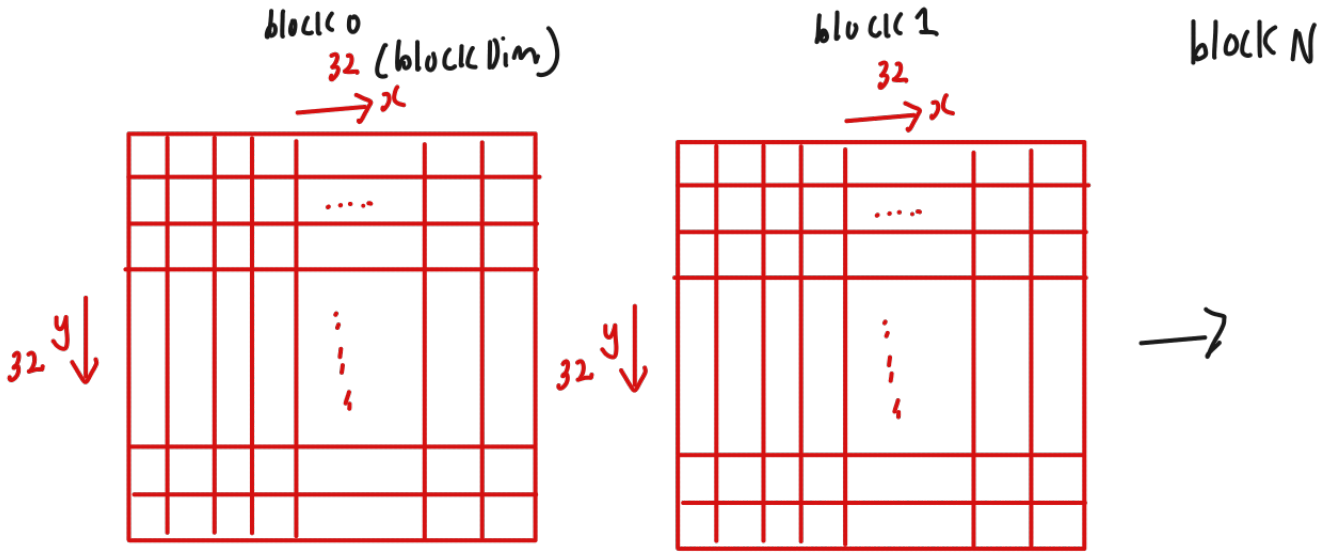
(x, y, z)

RGB TO GRAYSCALE CONVERSION CODE

```
                                    // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                                                    unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;
       wot img    hot img
 if (x < width && y < height) {
 }
}
```

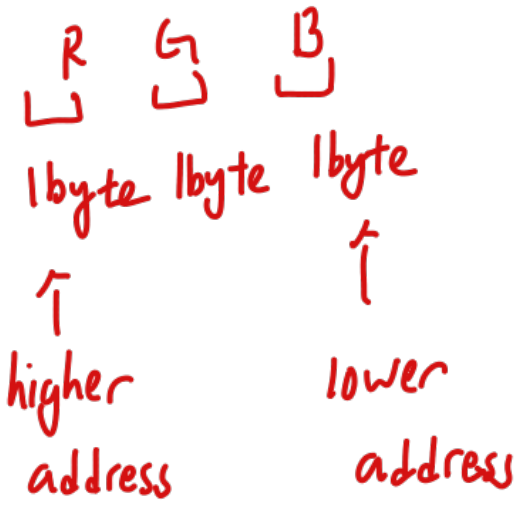
only process thread inside the img, thread outside the img don't run it.



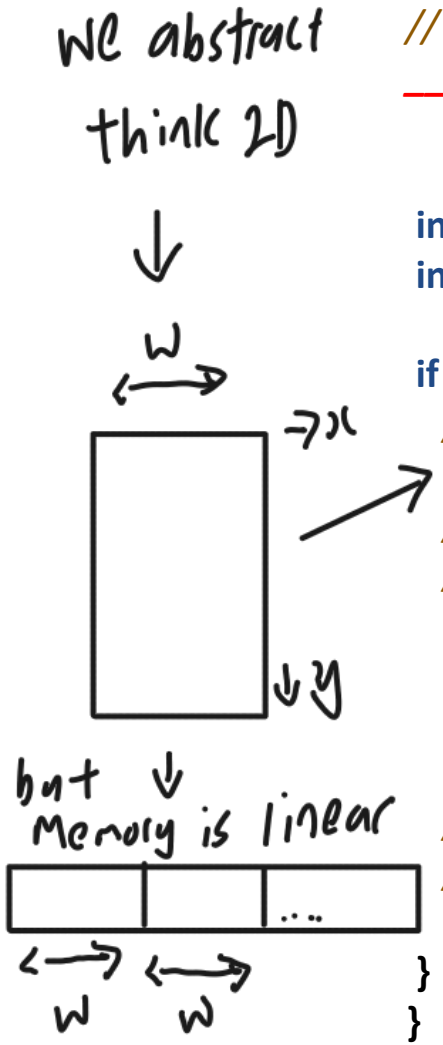

block 0
32 (blockDim)
x
32
y
block 1
32
x
32
y
block N

RGB TO GRAYSCALE CONVERSION CODE

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 if (x < width && y < height) {
   // get 1D coordinate for the grayscale image
   int grayOffset = y*width + x;
   // one can think of the RGB image having
   // CHANNEL times columns than the gray scale image
   int rgbOffset = grayOffset*CHANNELS;
   unsigned char r =  rgbImage[rgbOffset +2 ]; // red value for pixel
   unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
   unsigned char b = rgbImage[rgbOffset +2]; // blue value for pixel



 }
}
```



RGB TO GRAYSCALE CONVERSION CODE




```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;

if (x < width && y < height) {
  // get 1D coordinate for the grayscale image
  int grayOffset = y*width + x;
  // one can think of the RGB image having
  // CHANNEL times columns than the gray scale image
  int rgbOffset = grayOffset*CHANNELS;
  unsigned char r =  rgbImage[rgbOffset ]; // red value for pixel
  unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
  unsigned char b = rgbImage[rgbOffset + 3]; // blue value for pixel
  // perform the rescaling and store it
  // We multiply by floating point constants
  grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
}
}
```

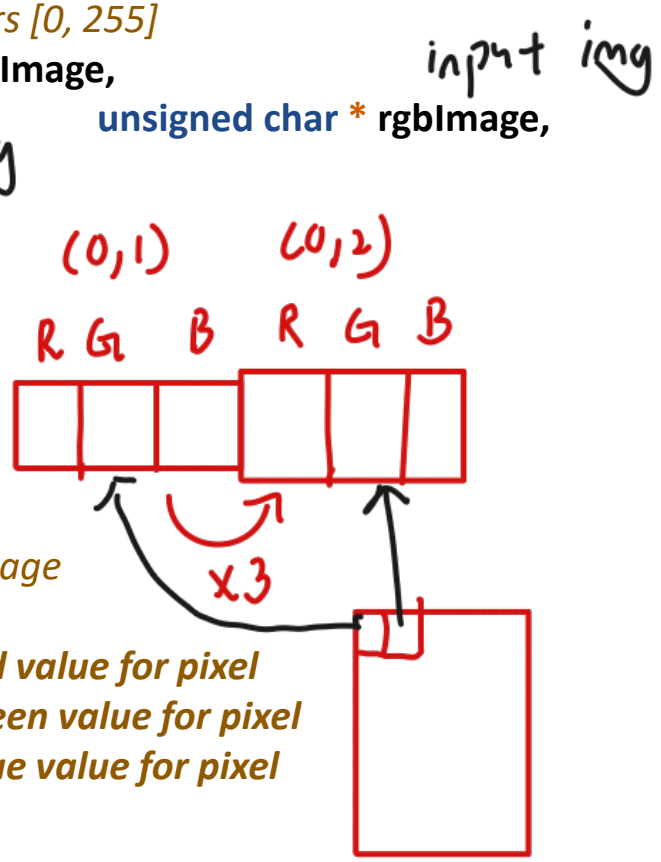


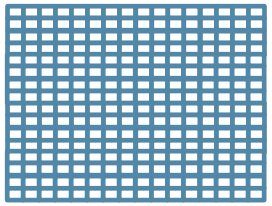
IMAGE BLURRING

02_ImageBlurrerWithCuda

```cpp
/**
 * The kernel implementating the blur_image processing in CUDA
 */
__global__ void blur_image_kernel(uint8_t *blur_img, uint8_t *bgr_img,
                                  int width, int height, int blur_size) {
    // Compute the linear index values for x and y
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    // Blur the image. Note that each pixel is handled by one thread
    if (x < width && y < height){
      int pixel_idx = (y * width + x) * 3; // Each pixel has 3 components (BGR)

      int sum_b = 0, sum_g = 0, sum_r = 0;
      int num_neighbors = 0;

      for (int i = -blur_size; i <= blur_size; i++){
        for (int j = -blur_size; j <= blur_size; j++){
          int neighbor_x = x + i;
          int neighbor_y = y + j;

          if (neighbor_x >= 0 && neighbor_x < width && neighbor_y >= 0 && neighbor_y < height){
            int neighbor_index = (neighbor_y * width + neighbor_x) * 3;

            sum_b += bgr_img[neighbor_index];
            sum_g += bgr_img[neighbor_index + 1];
            sum_r += bgr_img[neighbor_index + 2];

            num_neighbors++;
          }
        }
      }

      blur_img[pixel_idx] = sum_b / num_neighbors;
      blur_img[pixel_idx + 1] = sum_g / num_neighbors;
      blur_img[pixel_idx + 2] = sum_r / num_neighbors;
    }
}
```

```cpp
const int thread_per_blk = 32;      // Use 32 threads per block for x and y
void blur_image_core(uint8_t *blur_img, uint8_t *bgr_img,
                                  int width, int height, int blur_size) {
    uint8_t* iData;                 // Input image data for device
    uint8_t* oData;                 // Output image data for device

    // Compute the image data size (hint: use width and height; note that
    // the pixel size is 1 byte, and remember that each pixel has 3 colors)
    const size_t dataSize = width * height * 3 * sizeof(uint8_t);          // Modify to the correct size

    // Allocate the device memory for iData and oData the size of `dataSize`
    cudaMalloc((void**)&iData, dataSize);
    cudaMalloc((void**)&oData, dataSize);

    // Transfer BGR image from host to device memory.
    cudaMemcpy(iData, bgr_img, dataSize, cudaMemcpyHostToDevice);

    // Calculate the blocksize and gridsize using the given `thread_per_blk`.
    dim3 blockSize(thread_per_blk, thread_per_blk);
    dim3 gridSize((width + thread_per_blk - 1) / thread_per_blk,
              (height + thread_per_blk - 1) / thread_per_blk);

    // Call the CUDA blur image kernel
    blur_image_kernel <<<gridSize, blockSize>>> (oData, iData, width, height, blur_size);

    // Transfer the output (processed) image from device to host memory.
    cudaMemcpy(blur_img, oData, dataSize, cudaMemcpyDeviceToHost);

    // Free all the device memory
    cudaFree(iData);
    cudaFree(oData);
}
```

BLURRING BOX



Pixels processed by a thread block

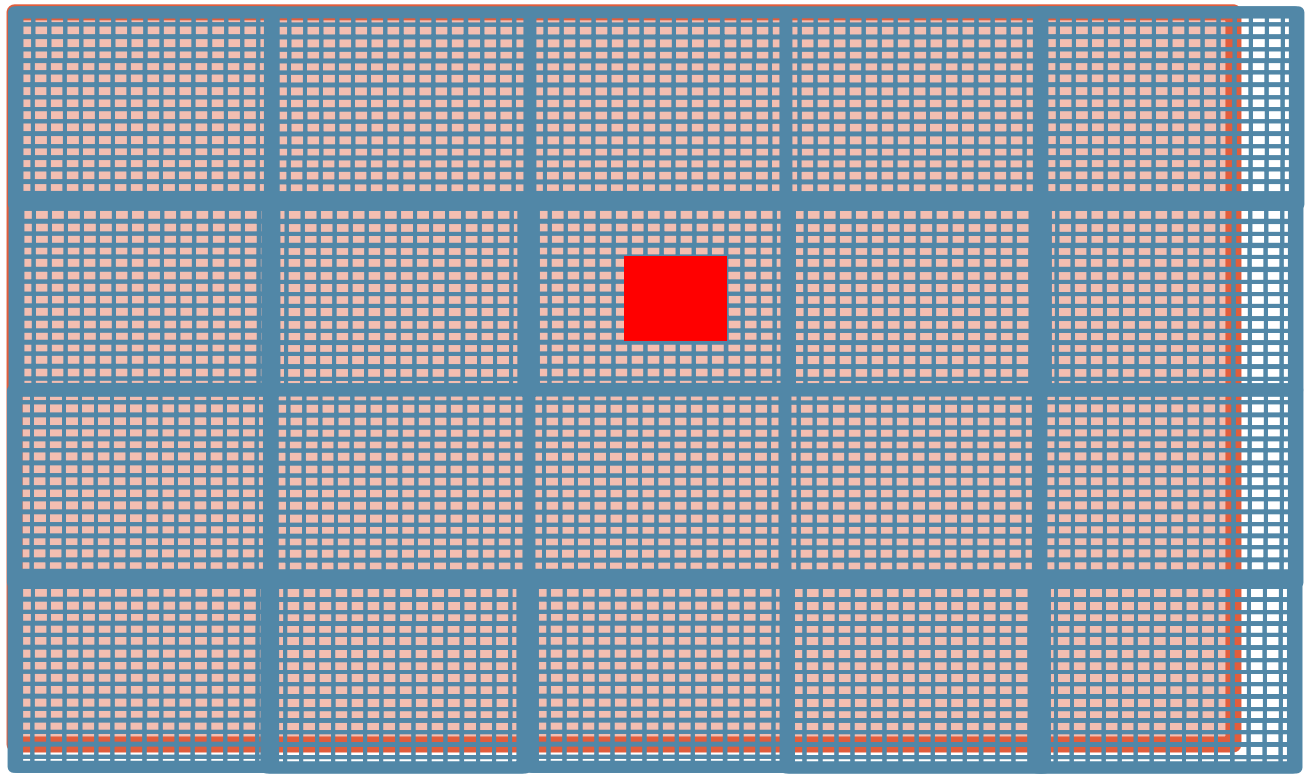


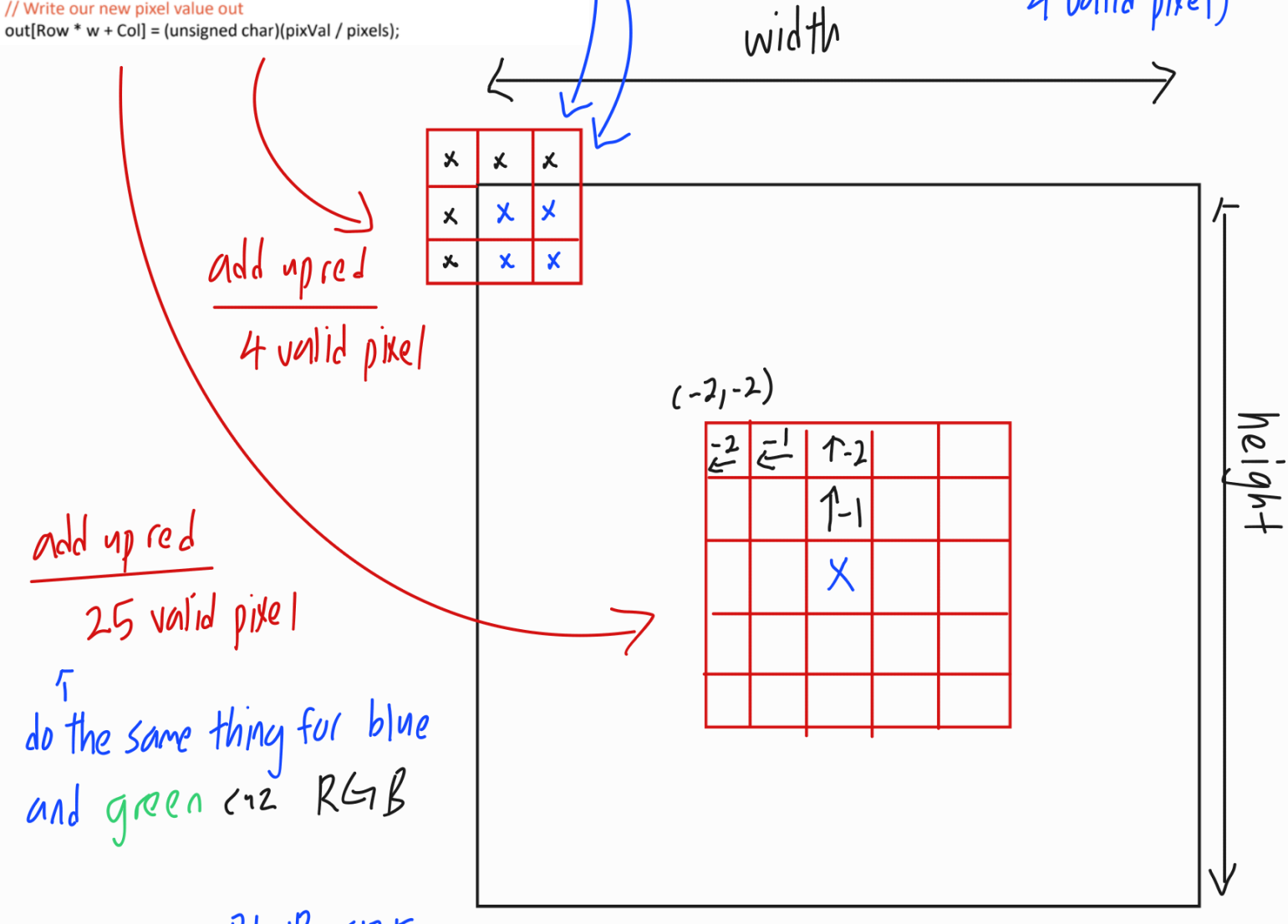
IMAGE BLUR AS A 2D KERNEL

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
  int Col  = blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
      ... // Rest of our kernel
  }
}
```

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col  = blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        int pixVal = 0;
        int pixels = 0;

        // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                // Verify we have a valid image pixel
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol];
                    pixels++; // Keep track of number of pixels in the accumulated total
                }
            }
        }

        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
```

```
    // Verify we have a valid image pixel
    if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
      pixVal += in[curRow * w + curCol];
      pixels++; // Keep track of number of pixels in the accumulated total
    }
  }
}

// Write our new pixel value out
out[Row * w + Col] = (unsigned char)(pixVal / pixels);
```

decide how many pixel needed to divide (in this case, 4 valid pixel)



assume BLUR_SIZE = 2

```
for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
  for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {
```

-2 ... 3 ... -2 ... 3

or

≤ BLUR_SIZE