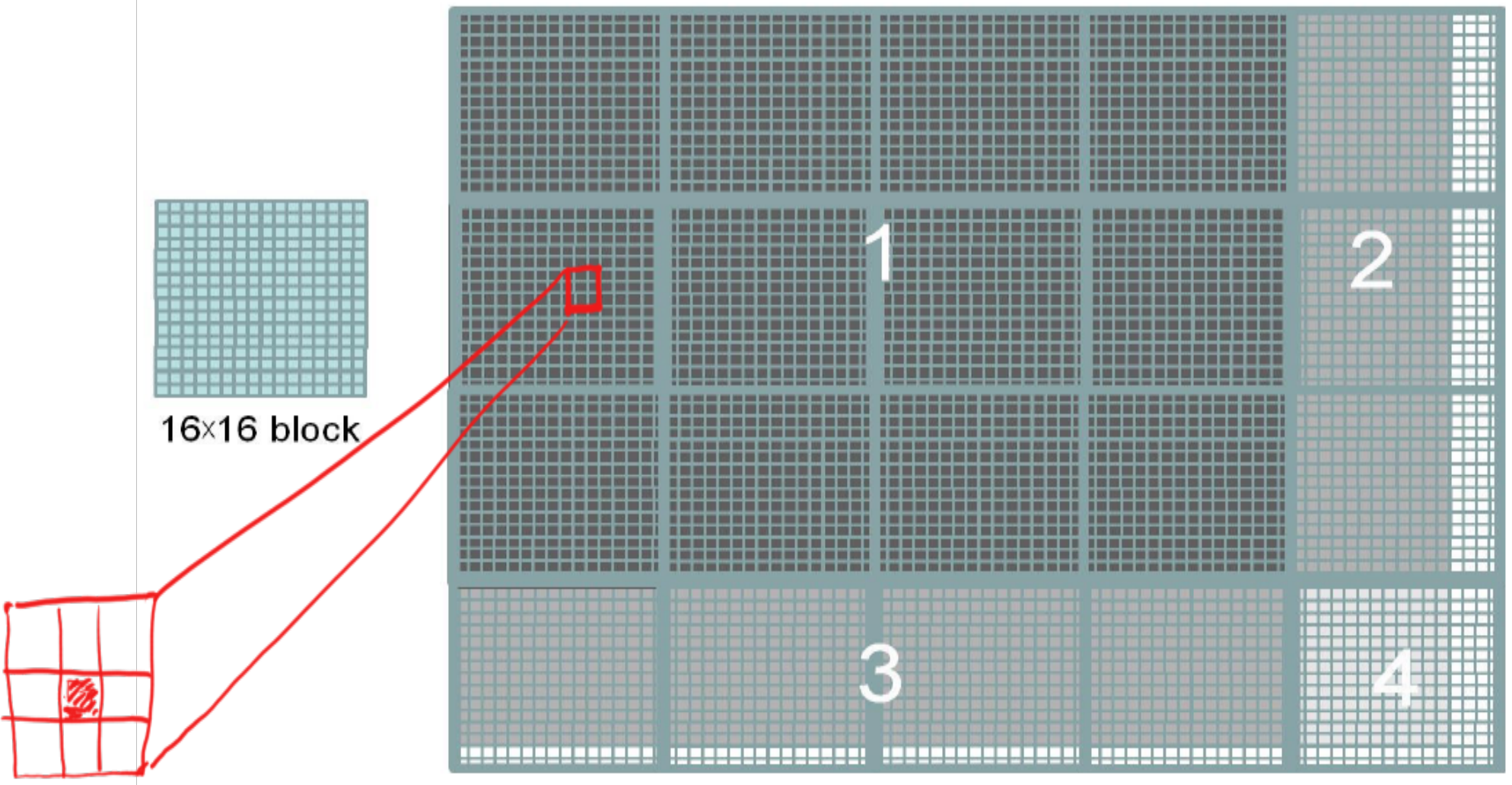
GPU programming with OpenCV Samples
By Mohit Kumar

CUDA:GPU:MultiDim:ex-3(Blur)

Image blurring

CUDA:GPU:MultiDim:ex-3(Blur)



Not all threads in a Block will follow the same control flow path.

CUDA:GPU:MultiDim:ex-3(Blur)

9	1	3	4	8
10	4	6	0	10
2	4	1	2	7
2	3	6		1
8	0	0	3	9

Avg Intensity Value =

$\frac{1}{9}(4 + 10 + 4 + 6 + 2 + 1 + 6 + 2 + 3)$

$(w_1 \cdot 4) + (w_2 \cdot 10) + (w_3 \cdot 4) + (w_4 \cdot 6) \ldots$

- weighted sum generates a far smoother image than just taking an average.

Weighted

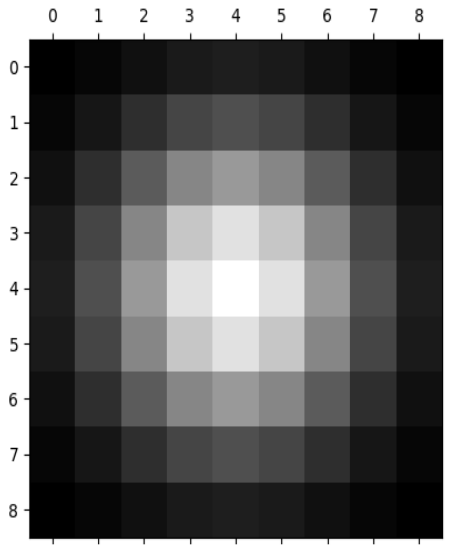
Unweighted.

CUDA:GPU:MultiDim:ex-3(Blur)

Visualizing the filter

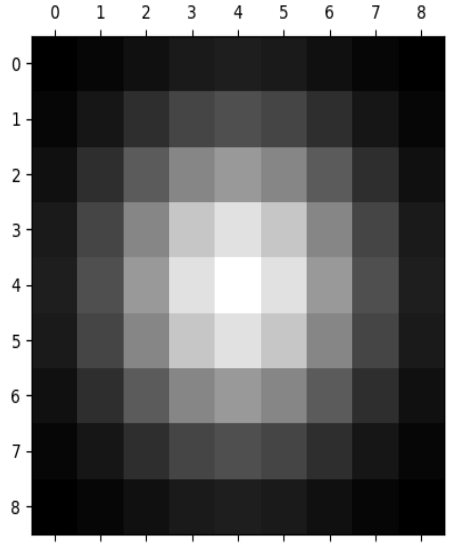
```
state=np.matrix(
      [ [0.018316, 0.043937, 0.082085, 0.119433, 0.135335, 0.119433, 0.082085, 0.043937, 0.018316],
        [0.043937, 0.105399, 0.196912, 0.286505, 0.324652, 0.286505, 0.196912, 0.105399, 0.043937],
        [0.082085, 0.196912, 0.367879, 0.535261, 0.606531, 0.535261, 0.367879, 0.196912, 0.082085],
        [0.119433, 0.286505, 0.535261, 0.778801, 0.882497, 0.778801, 0.535261, 0.286505, 0.119433],
        [0.135335, 0.324652, 0.606531, 0.882497, 1.000000, 0.882497, 0.606531, 0.324652, 0.135335],
        [0.119433, 0.286505, 0.535261, 0.778801, 0.882497, 0.778801, 0.535261, 0.286505, 0.119433],
        [0.082085, 0.196912, 0.367879, 0.535261, 0.606531, 0.535261, 0.367879, 0.196912, 0.082085],
        [0.043937, 0.105399, 0.196912, 0.286505, 0.324652, 0.286505, 0.196912, 0.105399, 0.043937],
        [0.018316, 0.043937, 0.082085, 0.119433, 0.135335, 0.119433, 0.082085, 0.043937, 0.018316]
])
plt.matshow(state, cmap=plt.cm.gray)
plt.show()
```

(Non Normalized)




```
state=np.matrix(
      [ [0.000763, 0.001831, 0.003422, 0.004978, 0.005641, 0.004978, 0.003422, 0.001831, 0.000763],
        [0.001831, 0.004393, 0.008208, 0.011942, 0.013532, 0.011942, 0.008208, 0.004393, 0.001831],
        [0.003422, 0.008208, 0.015334, 0.022311, 0.025282, 0.022311, 0.015334, 0.008208, 0.003422],
        [0.004978, 0.011942, 0.022311, 0.032463, 0.036785, 0.032463, 0.022311, 0.011942, 0.004978],
        [0.005641, 0.013532, 0.025282, 0.036785, 0.041683, 0.036785, 0.025282, 0.013532, 0.005641],
        [0.004978, 0.011942, 0.022311, 0.032463, 0.036785, 0.032463, 0.022311, 0.011942, 0.004978],
        [0.003422, 0.008208, 0.015334, 0.022311, 0.025282, 0.022311, 0.015334, 0.008208, 0.003422],
        [0.001831, 0.004393, 0.008208, 0.011942, 0.013532, 0.011942, 0.008208, 0.004393, 0.001831],
        [0.000763, 0.001831, 0.003422, 0.004978, 0.005641, 0.004978, 0.003422, 0.001831, 0.000763]
])
plt.matshow(state, cmap=plt.cm.gray)
plt.show()
```

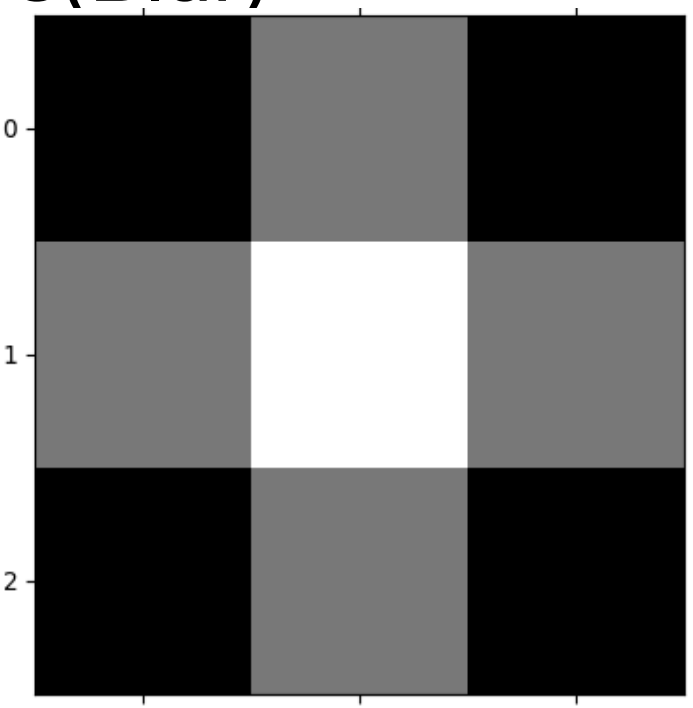
(Normalized)

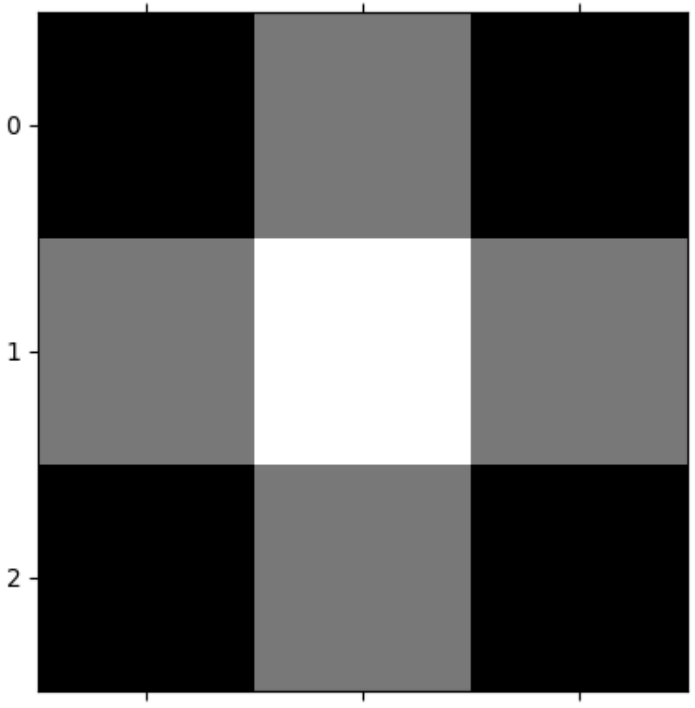


- The tactic with the filter results in a smoother blur than with the average
- Intuitively this happens because the pixel in question has a greater "say" on the resulting pixel than the surrounding pixel.

CUDA:GPU:MultiDim:ex-3(Blur)

```
state=np.matrix(
      [ [0.778801, 0.882497, 0.778801],
        [0.882497, 1.000000, 0.882497],
        [0.778801, 0.882497, 0.778801]
])

plt.matshow(state, cmap=plt.cm.gray)
plt.show()

state=np.matrix(
      [ [0.101868, 0.115432, 0.101868],
        [0.115432, 0.130801, 0.115432],
        [0.101868, 0.115432, 0.101868]

])
plt.matshow(state, cmap=plt.cm.gray)
plt.show()
```





- The filter for the slide is (3x3) for easier drawings

CUDA:GPU:MultiDim:ex-3(Blur)

Blur (Logical steps)

1. Seperate the RGB channels.
2. Apply the blurring filter to each channel seperately
3. Recombine the channel and reconstruct the image.

CUDA:GPU:MultiDim:ex-3(Blur)

Blur (Actual Steps)

1. Allocate memory for the input and the output-image on the device
   (a) copy image to device
   (b) memset '0' to output image on device
2. Allocate memory for each channel seperately (output) on the device
   (a) memset all to '0'
3. Allocate memory for each channel seperately (input) on the device and for the filter
   (a) memset RGB memory to '0'
   (b) copy filter to the device

CUDA:GPU:MultiDim:ex-3(Blur)

4. launch the kernel to seperate the RGB

5. seperateChannel kernel executes

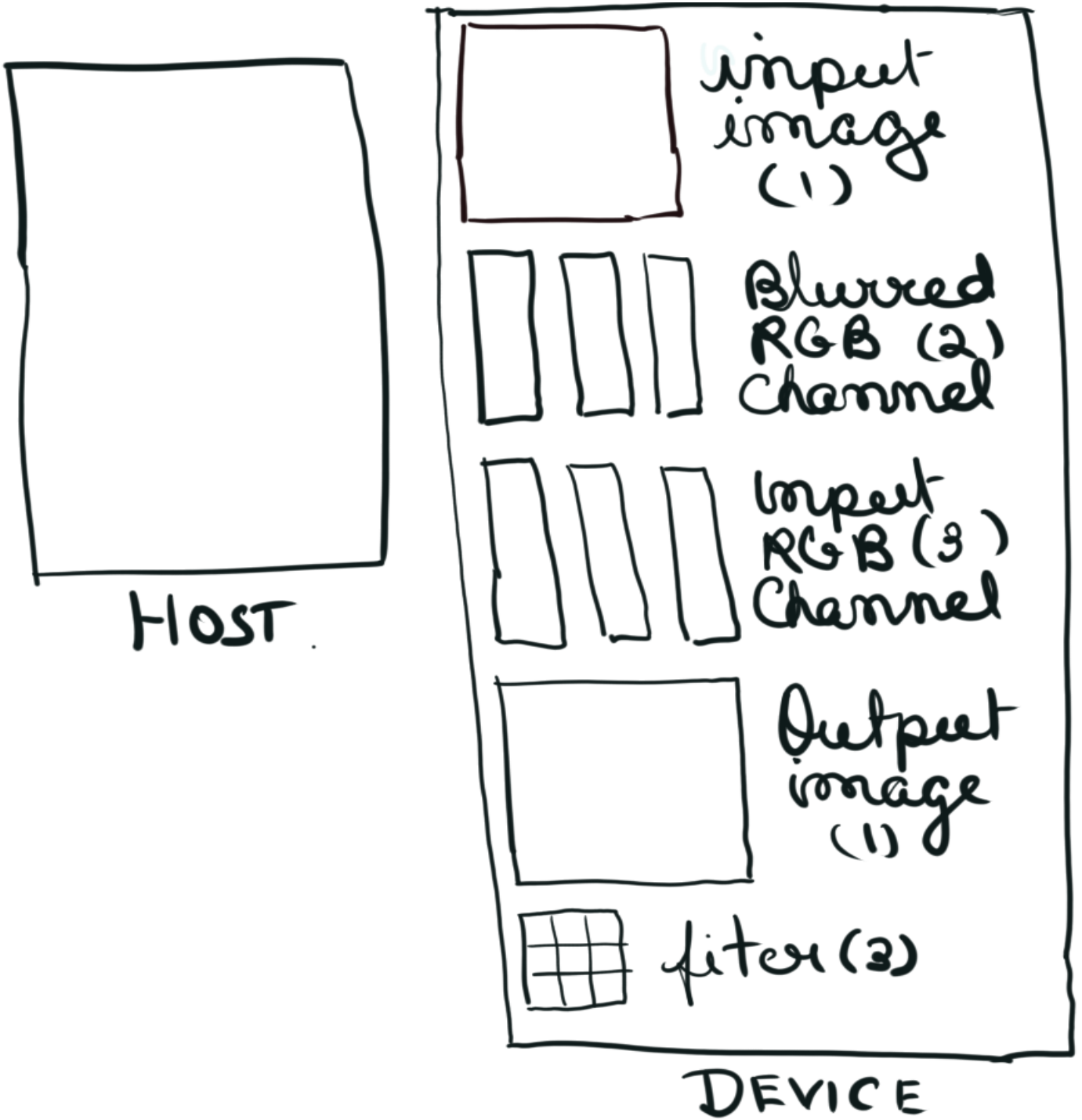
6. launch the kernel to bleur the channels seperately

6a, 6, c gaussianBleur excutes for all the channels seperately

7. Lanch the recoombine channel

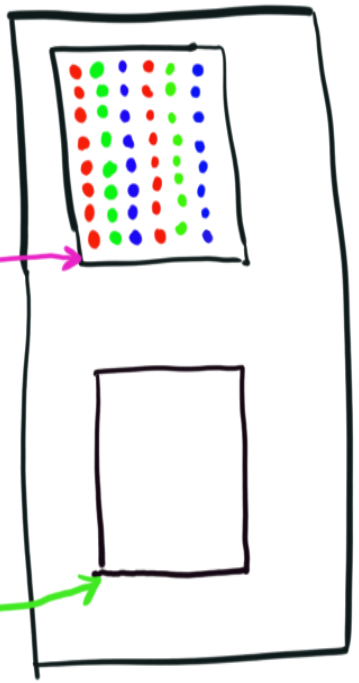
7a. recoombine kernel excutes

CUDA:GPU:MultiDim:ex-3(Blur)

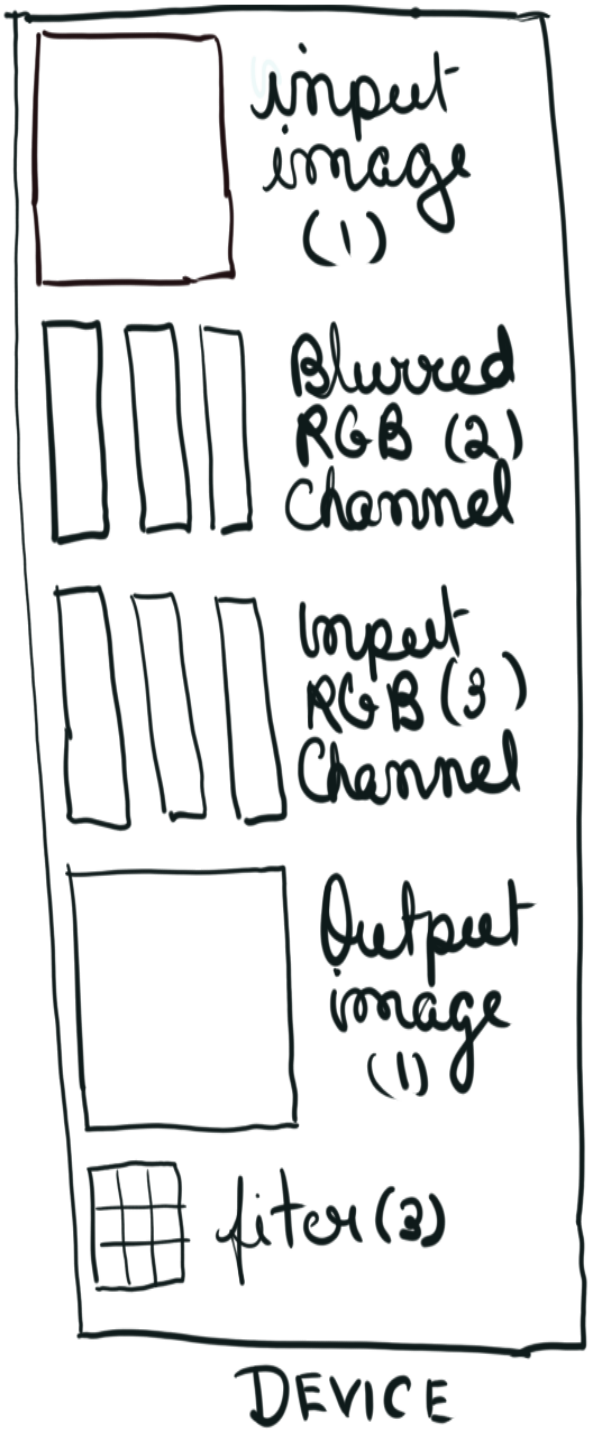


CUDA:GPU:MultiDim:ex-3(Blur)

```
cv::Mat image = cv::imread(filename.c_str(), CV_LOAD_IMAGE_COLOR);
if (image.empty()) {
    std::cerr << "Couldn't open file: " << filename << std::endl;
    exit(1);
}

cv::cvtColor(image, imageInputRGBA, CV_BGR2RGBA);

//allocate memory for the output
imageOutputRGBA.create(image.rows, image.cols, CV_8UC4);


*h_inputImageRGBA = (uchar4 *) imageInputRGBA.ptr<unsigned char>(0);
*h_outputImageRGBA = (uchar4 *) imageOutputRGBA.ptr<unsigned char>(0);
```



HOST



DEVICE

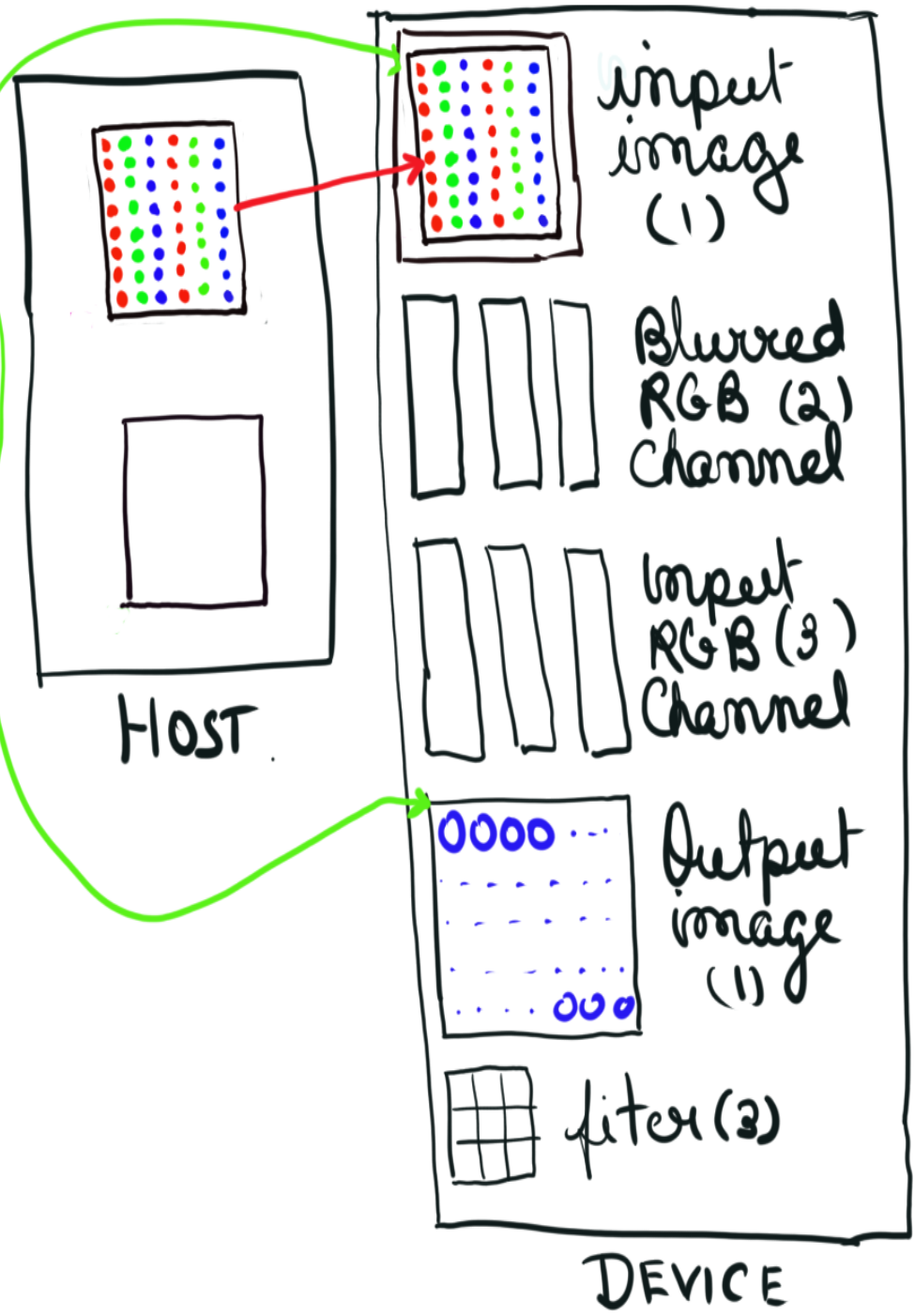
0. Open the image on the host memory

CUDA:GPU:MultiDim:ex-3(Blur)

```
const size_t numPixels = numRows() * numCols();
//allocate memory on the device for both input and output
checkCudaErrors(cudaMalloc(d_inputImageRGBA, sizeof(uchar4) * numPixels));
checkCudaErrors(cudaMalloc(d_outputImageRGBA, sizeof(uchar4) * numPixels));
checkCudaErrors(
        cudaMemset(*d_outputImageRGBA, 0, numPixels * sizeof(uchar4))); //m

//copy input array to the GPU
checkCudaErrors(
        cudaMemcpy(*d_inputImageRGBA, *h_inputImageRGBA,
                sizeof(uchar4) * numPixels, cudaMemcpyHostToDevice));
```

1b

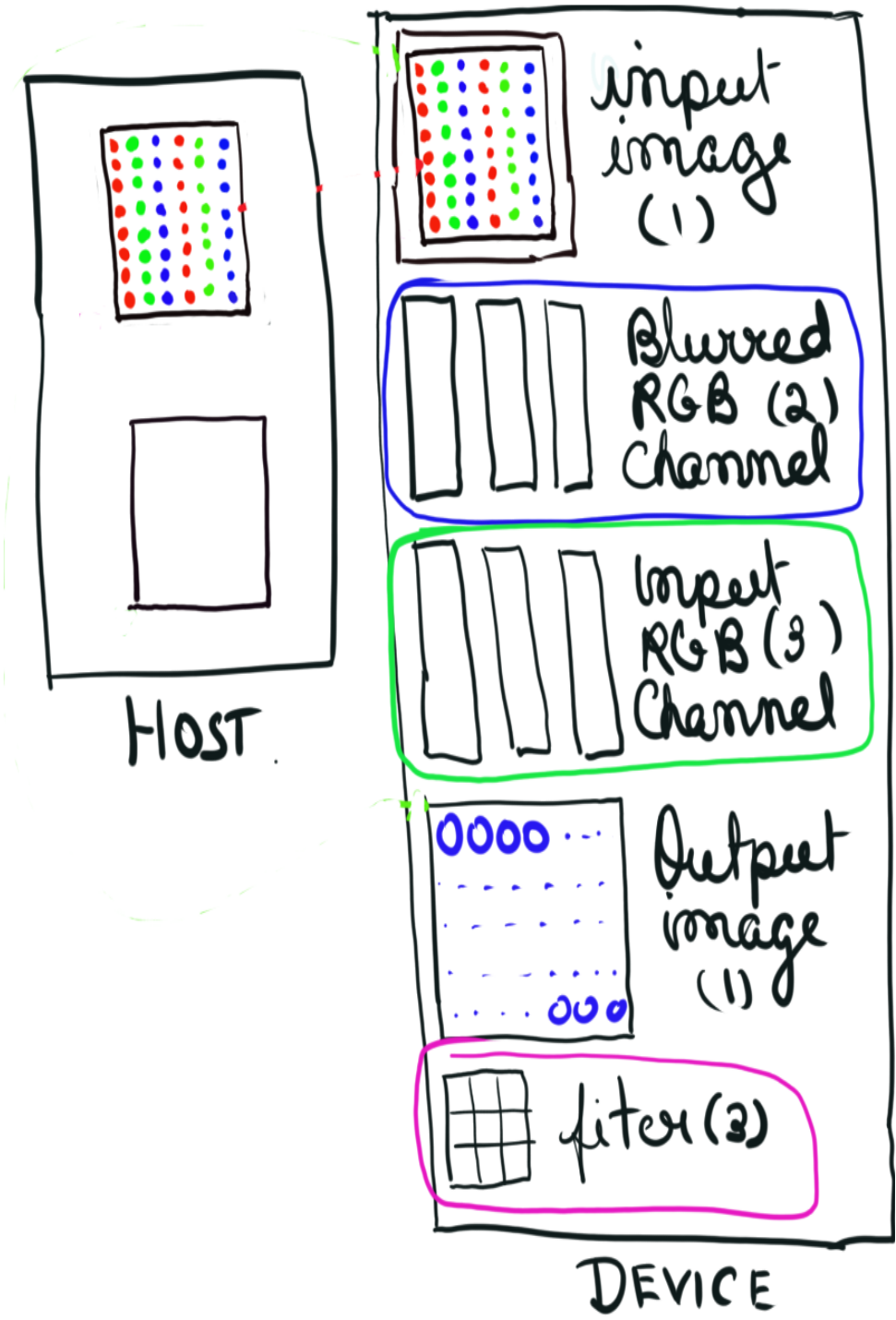
1a



CUDA:GPU:MultiDim:ex-3(Blur)

```
//blurred
checkCudaErrors(
        cudaMalloc(d_redBlurred, sizeof(unsigned char) * numPixels));
checkCudaErrors(
        cudaMalloc(d_greenBlurred, sizeof(unsigned char) * numPixels));
checkCudaErrors(
        cudaMalloc(d_blueBlurred, sizeof(unsigned char) * numPixels));
checkCudaErrors(
        cudaMemset(*d_redBlurred, 0, sizeof(unsigned char) * numPixels));
checkCudaErrors(
        cudaMemset(*d_greenBlurred, 0, sizeof(unsigned char) * numPixels));
checkCudaErrors(
        cudaMemset(*d_blueBlurred, 0, sizeof(unsigned char) * numPixels));
```

```
//allocate memory for the three different channels
//original
checkCudaErrors(
        cudaMalloc(&d_red,
                sizeof(unsigned char) * numRowsImage * numColsImage));
checkCudaErrors(
        cudaMalloc(&d_green,
                sizeof(unsigned char) * numRowsImage * numColsImage));
checkCudaErrors(
        cudaMalloc(&d_blue,
                sizeof(unsigned char) * numRowsImage * numColsImage));

size_t filtersize = sizeof(float) * filterWidth * filterWidth;
checkCudaErrors(cudaMalloc(&d_filter, filtersize));

checkCudaErrors(
        cudaMemcpy(d_filter, h_filter, filtersize, cudaMemcpyHostToDevice));
```



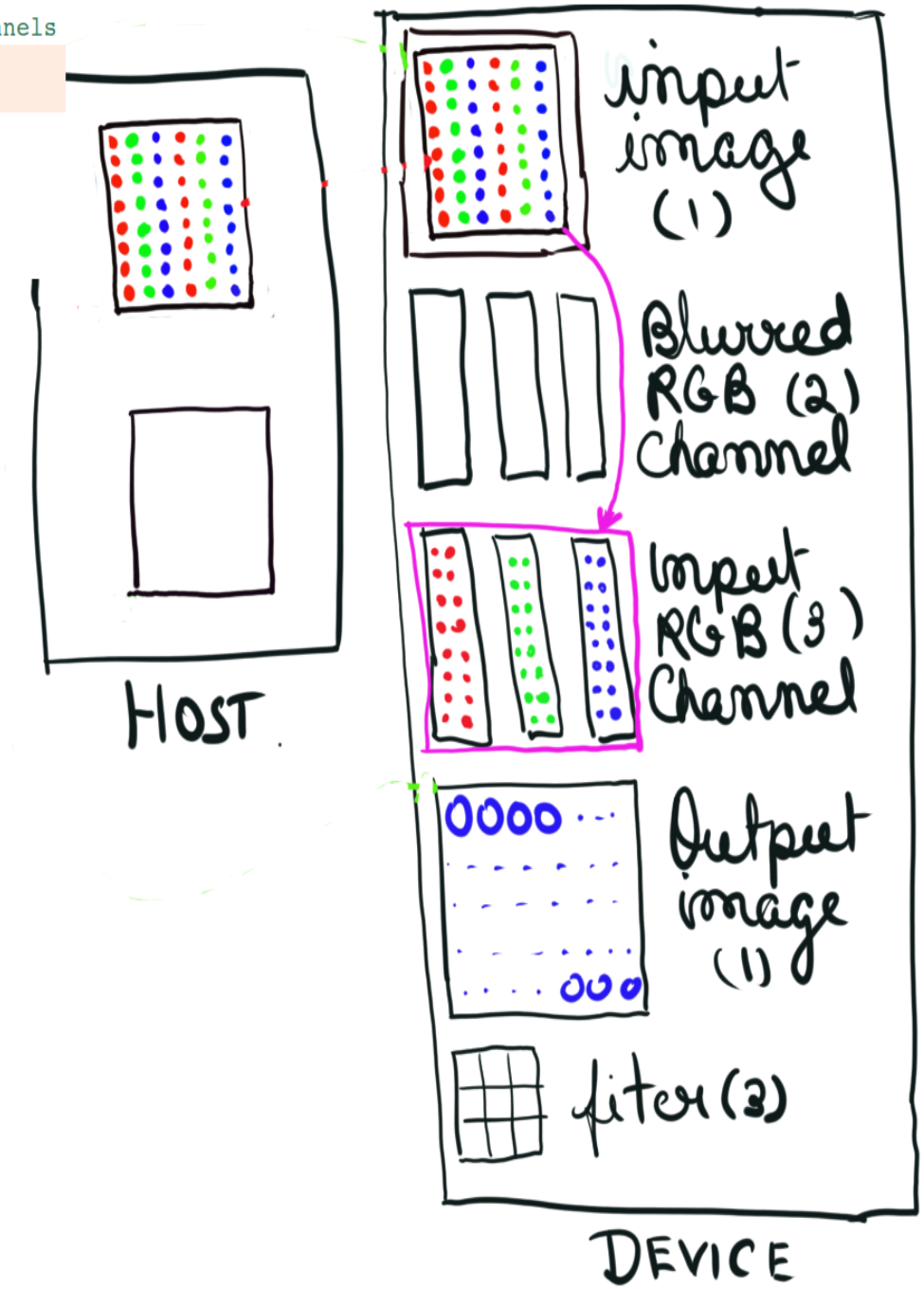
CUDA:GPU:MultiDim:ex-3(Blur)

```
const dim3 blockSize(16, 16, 1);  //TODO
const dim3 gridSize(ceil(numCols / 16.0), ceil(numRows / 16.0), 1);  //TODO
//TODO: Launch a kernel for separating the RGBA image into different color channels
separateChannels<<<gridSize, blockSize>>>(d_inputImageRGBA, numRows,
        numCols, d_red, d_green, d_blue);

// Call cudaDeviceSynchronize(), then call checkCudaErrors() immediately after
// launching your kernel to make sure that you didn't make any mistakes.
cudaDeviceSynchronize();
checkCudaErrors(cudaGetLastError());
```

4.

```
__global__
void separateChannels(const uchar4* const inputImageRGBA, int numRows,
        int numCols, unsigned char* const redChannel,
        unsigned char* const greenChannel, unsigned char* const blueChannel) {

    int Col = threadIdx.x + blockIdx.x * blockDim.x;
    int Row = threadIdx.y + blockIdx.y * blockDim.y;
    int i = Row * numCols + Col;
    if (Col < numCols && Row < numRows) {
        uchar4 rgba = inputImageRGBA[i];
        redChannel[i] = rgba.x;
        greenChannel[i] = rgba.y;
        blueChannel[i] = rgba.z;
    }
}
```

5.



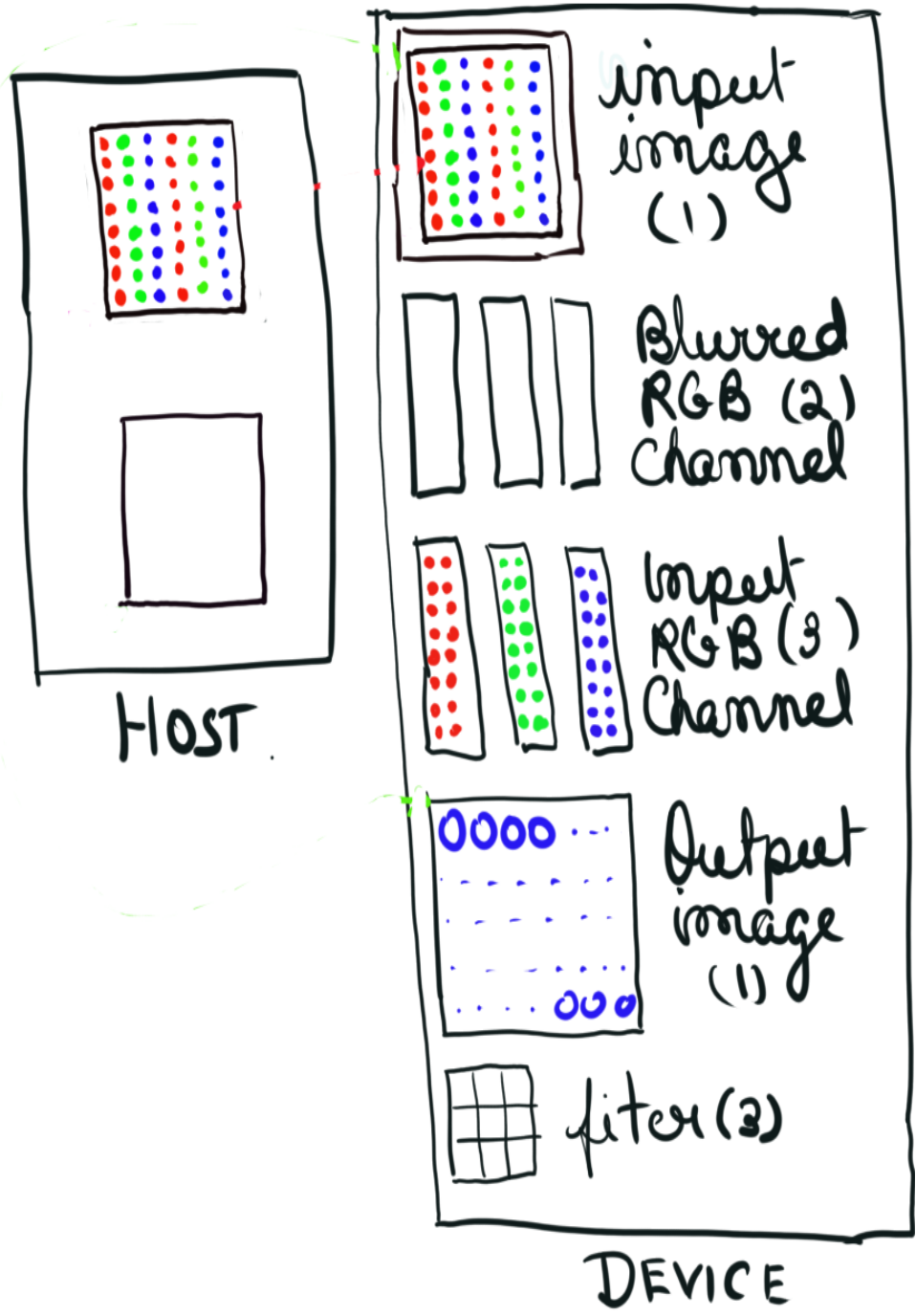
CUDA:GPU:MultiDim:ex-3(Blur)

```
gaussian_blur<<<gridSize, blockSize>>>(d_red, d_redBlurred, numRows,
        numCols, d_filter, filterWidth);

gaussian_blur<<<gridSize, blockSize>>>(d_green, d_greenBlurred, numRows,
        numCols, d_filter, filterWidth);

gaussian_blur<<<gridSize, blockSize>>>(d_blue, d_blueBlurred, numRows,
        numCols, d_filter, filterWidth);
```

6

(6a, 6b, 6c)

```
__global__
void gaussian_blur(const unsigned char* const inputChannel,
        unsigned char* const outputChannel, int numRows, int numCols,
        const float* const filter, const int filterWidth) {
        int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (col >= numCols || row >= numRows) {
        return;
    }
    float result = 0.0f;
    for (int row_delta = -filterWidth / 2; row_delta <= filterWidth / 2;
            row_delta++) {
        for (int col_delta = -filterWidth / 2; col_delta <= filterWidth / 2;
                col_delta++) {
            // Compute the coordinates of the value this coefficient applies
            // Apply clamping to image boundaries.
            int value_row = min(max(row + row_delta, 0), numRows - 1);
            int value_col = min(max(col + col_delta, 0), numCols - 1);

            // Compute the partial sum this value adds to the result when sc
            // the appropriate coefficient.
            float channel_value = static_cast<float>(inputChannel[value_row
                    * numCols + value_col]);
            float filter_coefficient = filter[(row_delta + filterWidth / 2)
                    * filterWidth + (col_delta + filterWidth / 2)];
            result += channel_value * filter_coefficient;
        }
    }
    outputChannel[row * numCols + col] = result;
}
```



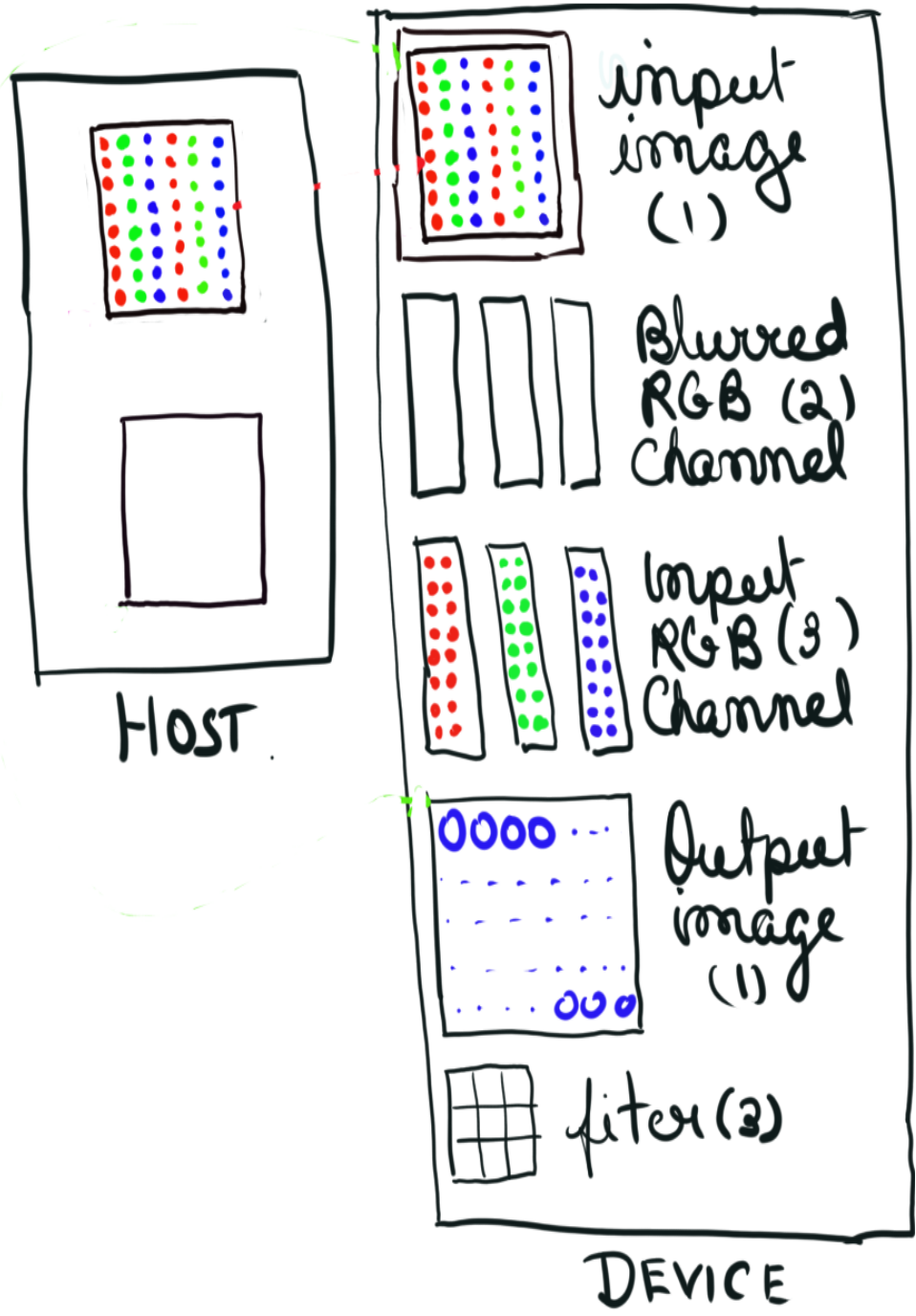
CUDA:GPU:MultiDim:ex-3(Blur)

```
gaussian_blur<<<gridSize, blockSize>>>(d_red, d_redBlurred, numRows,
        numCols, d_filter, filterWidth);

gaussian_blur<<<gridSize, blockSize>>>(d_green, d_greenBlurred, numRows,
        numCols, d_filter, filterWidth);

gaussian_blur<<<gridSize, blockSize>>>(d_blue, d_blueBlurred, numRows,
        numCols, d_filter, filterWidth);
```

6

(6a, 6b, 6c)

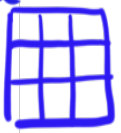
```
__global__
void gaussian_blur(const unsigned char* const inputChannel,
        unsigned char* const outputChannel, int numRows, int numCols,
        const float* const filter, const int filterWidth) {
        int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (col >= numCols || row >= numRows) {
        return;
    }
    float result = 0.0f;
    for (int row_delta = -filterWidth / 2; row_delta <= filterWidth / 2;
            row_delta++) {
        for (int col_delta = -filterWidth / 2; col_delta <= filterWidth / 2;
                col_delta++) {
            // Compute the coordinates of the value this coefficient applies
            // Apply clamping to image boundaries.
            int value_row = min(max(row + row_delta, 0), numRows - 1);
            int value_col = min(max(col + col_delta, 0), numCols - 1);

            // Compute the partial sum this value adds to the result when sc
            // the appropriate coefficient.
            float channel_value = static_cast<float>(inputChannel[value_row
                    * numCols + value_col]);
            float filter_coefficient = filter[(row_delta + filterWidth / 2)
                    * filterWidth + (col_delta + filterWidth / 2)];
            result += channel_value * filter_coefficient;
        }
    }
    outputChannel[row * numCols + col] = result;
}
```

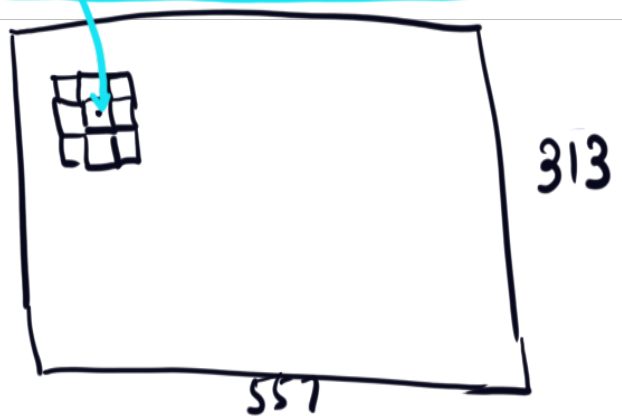


CUDA:GPU:MultiDim:ex-3(Blur)

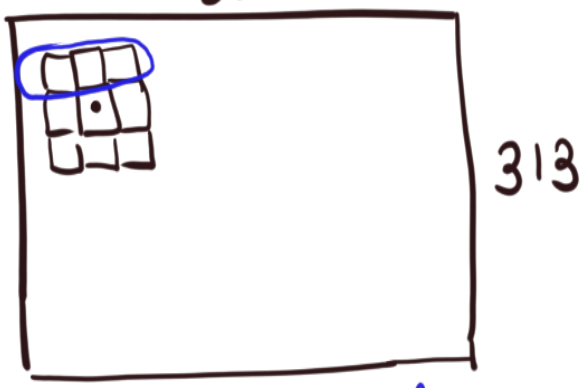
```
__global__
void gaussian_blur(const unsigned char* const inputChannel,
        unsigned char* const outputChannel, int numRows, int numCols,
        const float* const filter, const int filterWidth) {
        int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (col >= numCols || row >= numRows) {
        return;
    }
    float result = 0.0f;
    for (int row_delta = -filterWidth / 2; row_delta <= filterWidth / 2;
            row_delta++) {
        for (int col_delta = -filterWidth / 2; col_delta <= filterWidth / 2;
                col_delta++) {
            // Compute the coordinates of the value this coefficient applies
            // Apply clamping to image boundaries.
            int value_row = min(max(row + row_delta, 0), numRows - 1);
            int value_col = min(max(col + col_delta, 0), numCols - 1);

            // Compute the partial sum this value adds to the result when sc
            // the appropriate coefficient.
            float channel_value = static_cast<float>(inputChannel[value_row
                    * numCols + value_col]);
            float filter_coefficient = filter[(row_delta + filterWidth / 2)
                    * filterWidth + (col_delta + filterWidth / 2)];
            result += channel_value * filter_coefficient;
        }
    }
    outputChannel[row * numCols + col] = result;
}
```

3 for the picture





- for threadIdx.x = 3 and threadIdx.y = 3 and picture of 557 x 313 dimn.

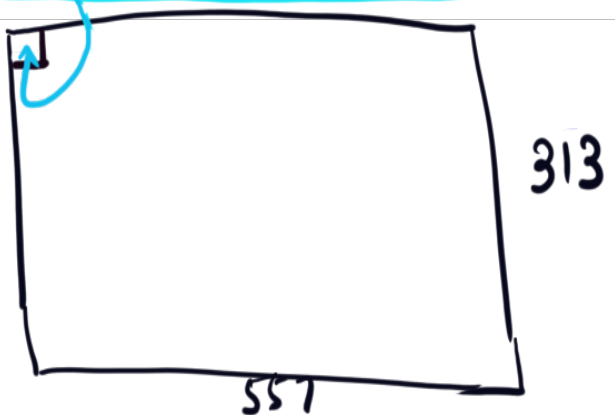


- channel value index is going to be (557+557+3-1) = 1116, 1117, 1118, 1673, 1674, 1675, ...
- Filter coffecient index is going to be 0 1 2 4 5 6 7 8
- After adding up writing the result at the same offset.

CUDA:GPU:MultiDim:ex-3(Blur)

```
__global__
void gaussian_blur(const unsigned char* const inputChannel,
        unsigned char* const outputChannel, int numRows, int numCols,
        const float* const filter, const int filterWidth) {
        in  col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (col >= numCols || row >= numRows) {
        return;
    }
    float result = 0.0f;
    for (int row_delta = -filterWidth / 2; row_delta <= filterWidth / 2;
            row_delta++) {
        for (int col_delta = -filterWidth / 2; col_delta <= filterWidth / 2;
                col_delta++) {
            // Compute the coordinates of the value this coefficient applies
            // Apply clamping to image boundaries.
            int value_row = min(max(row + row_delta, 0), numRows - 1);
            int value_col = min(max(col + col_delta, 0), numCols - 1);

            // Compute the partial sum this value adds to the result when sc
            // the appropriate coefficient.
            float channel_value = static_cast<float>(inputChannel[value_row
                    * numCols + value_col]);
            float filter_coefficient = filter[(row_delta + filterWidth / 2)
                    * filterWidth + (col_delta + filterWidth / 2)];
            result += channel_value * filter_coefficient;
        }
    }
    outputChannel[row * numCols + col] = result;
}
```

3 for the picture

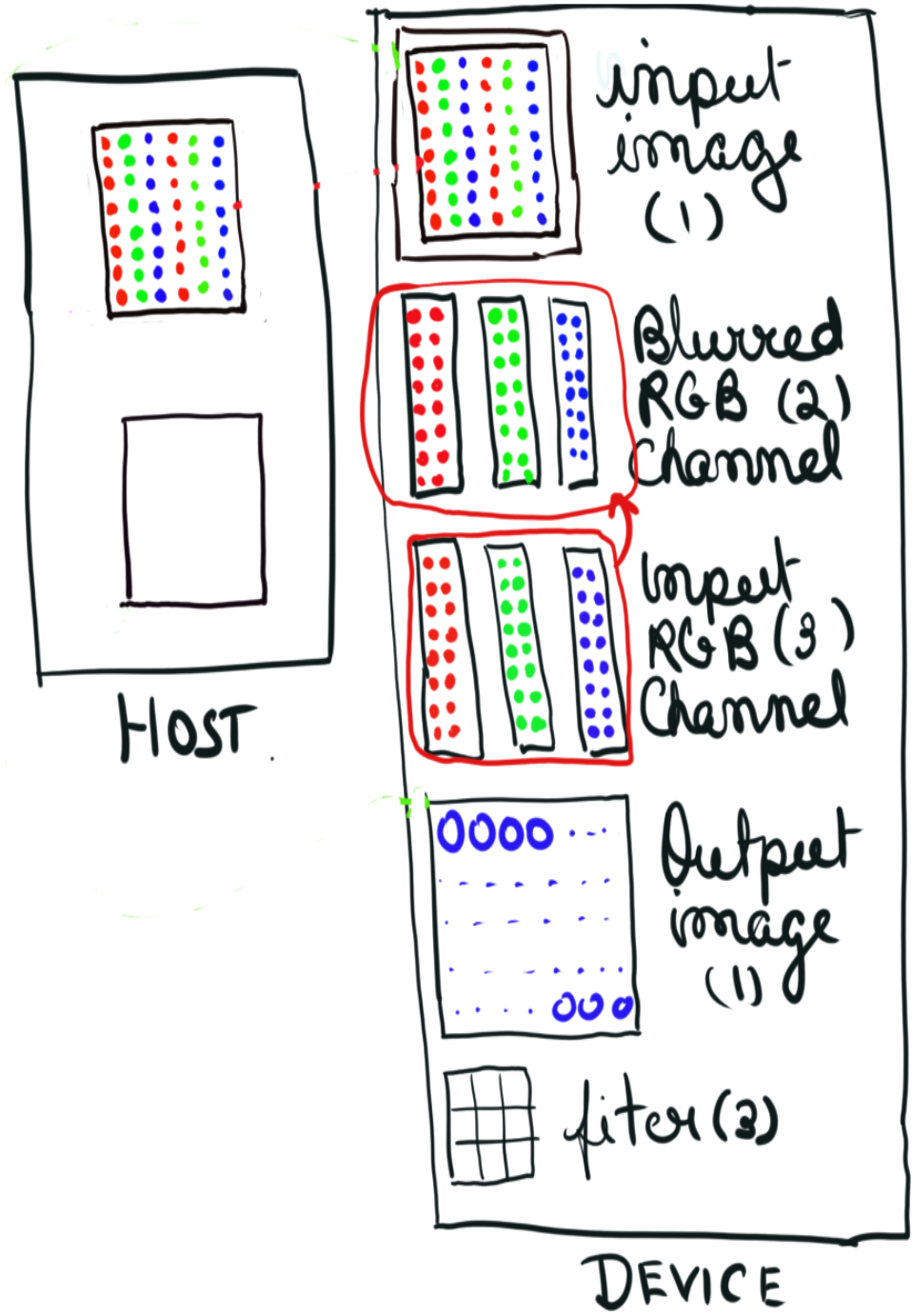


- for threadIdx.x = 0 and threadIdx.y = 0 and picture of 557 x 313 dim.

- channel value index is going to be ((0,0,1), (0,1,1), (557, 557, 558)).
- Pink values are copied because these are edge pixels

CUDA:GPU:MultiDim:ex-3(Blur)

```
__global__
void gaussian_blur(const unsigned char* const inputChannel,
        unsigned char* const outputChannel, int numRows, int numCols,
        const float* const filter, const int filterWidth) {
        int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (col >= numCols || row >= numRows) {
        return;
    }
    float result = 0.0f;
    for (int row_delta = -filterWidth / 2; row_delta <= filterWidth / 2;
            row_delta++) {
        for (int col_delta = -filterWidth / 2; col_delta <= filterWidth / 2;
                col_delta++) {
            // Compute the coordinates of the value this coefficient applies
            // Apply clamping to image boundaries.
            int value_row = min(max(row + row_delta, 0), numRows - 1);
            int value_col = min(max(col + col_delta, 0), numCols - 1);

            // Compute the partial sum this value adds to the result when sc
            // the appropriate coefficient.
            float channel_value = static_cast<float>(inputChannel[value_row
                    * numCols + value_col]);
            float filter_coefficient = filter[(row_delta + filterWidth / 2)
                    * filterWidth + (col_delta + filterWidth / 2)];
            result += channel_value * filter_coefficient;
        }
    }
    outputChannel[row * numCols + col] = result;
}
```



CUDA:GPU:MultiDim:ex-3(Blur)

```
recombineChannels<<<gridSize, blockSize>>>(d_redBlurred, d_greenBlurred,
        d_blueBlurred, d_outputImageRGBA, numRows, numCols);
```

7a

```
__global__
void recombineChannels(const unsigned char* const redChannel,
        const unsigned char* const greenChannel,
        const unsigned char* const blueChannel, uchar4* const outputImageRGBA,
        int numRows, int numCols) {
    const int2 thread_2D_pos = make_int2(blockIdx.x * blockDim.x + threadIdx.x,
            blockIdx.y * blockDim.y + threadIdx.y);
    const int thread_1D_pos = thread_2D_pos.y * numCols + thread_2D_pos.x;
    //make sure we don't try and access memory outside the image
    //by having any threads mapped there return early
    if (thread_2D_pos.x >= numCols || thread_2D_pos.y >= numRows)
        return;
    unsigned char red = redChannel[thread_1D_pos];
    unsigned char green = greenChannel[thread_1D_pos];
    unsigned char blue = blueChannel[thread_1D_pos];
    //Alpha should be 255 for no transparency
    uchar4 outputPixel = make_uchar4(red, green, blue, 255);
    outputImageRGBA[thread_1D_pos] = outputPixel;
}
```

recombine the seperately blurred RGB pixels.