CHAPTER 3

Multidimensional grids and data

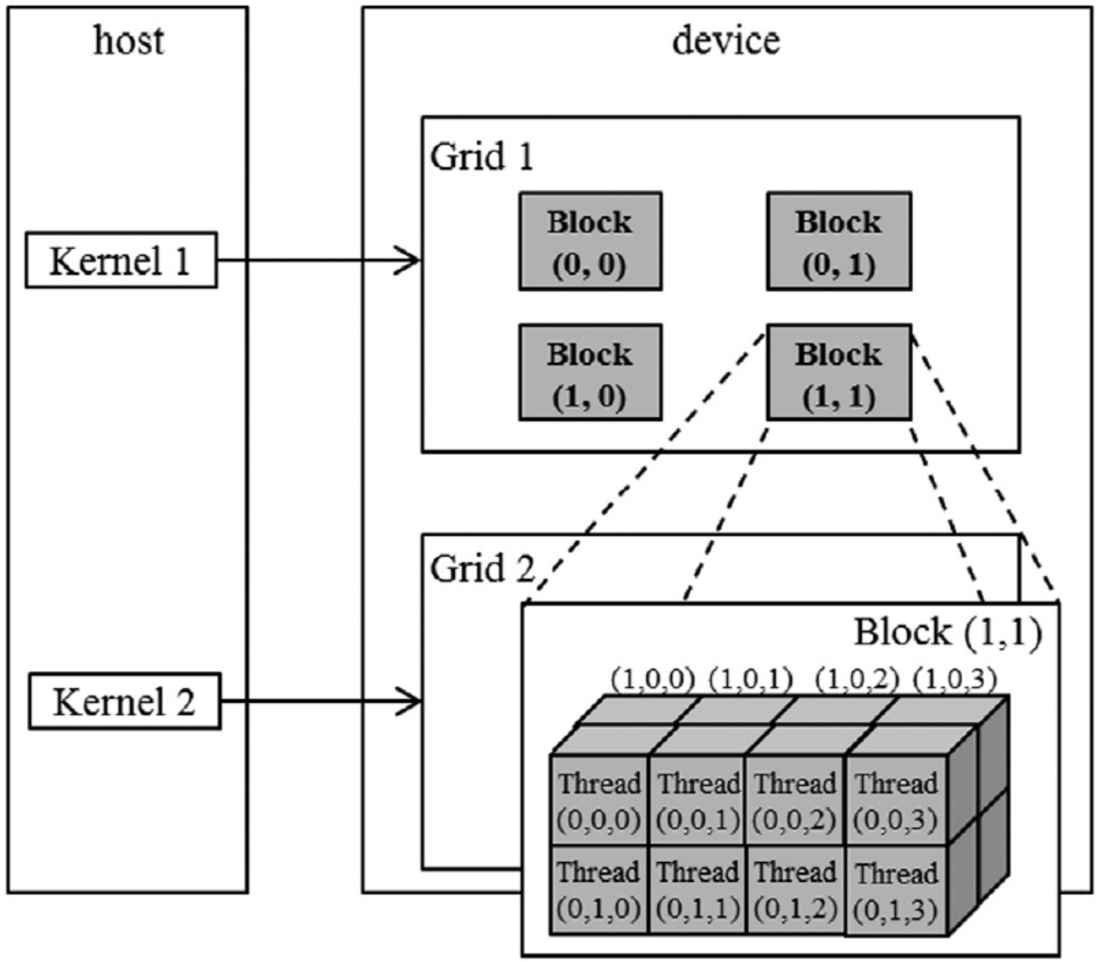

FIGURE 3.1

A multidimensional example of CUDA grid organization.

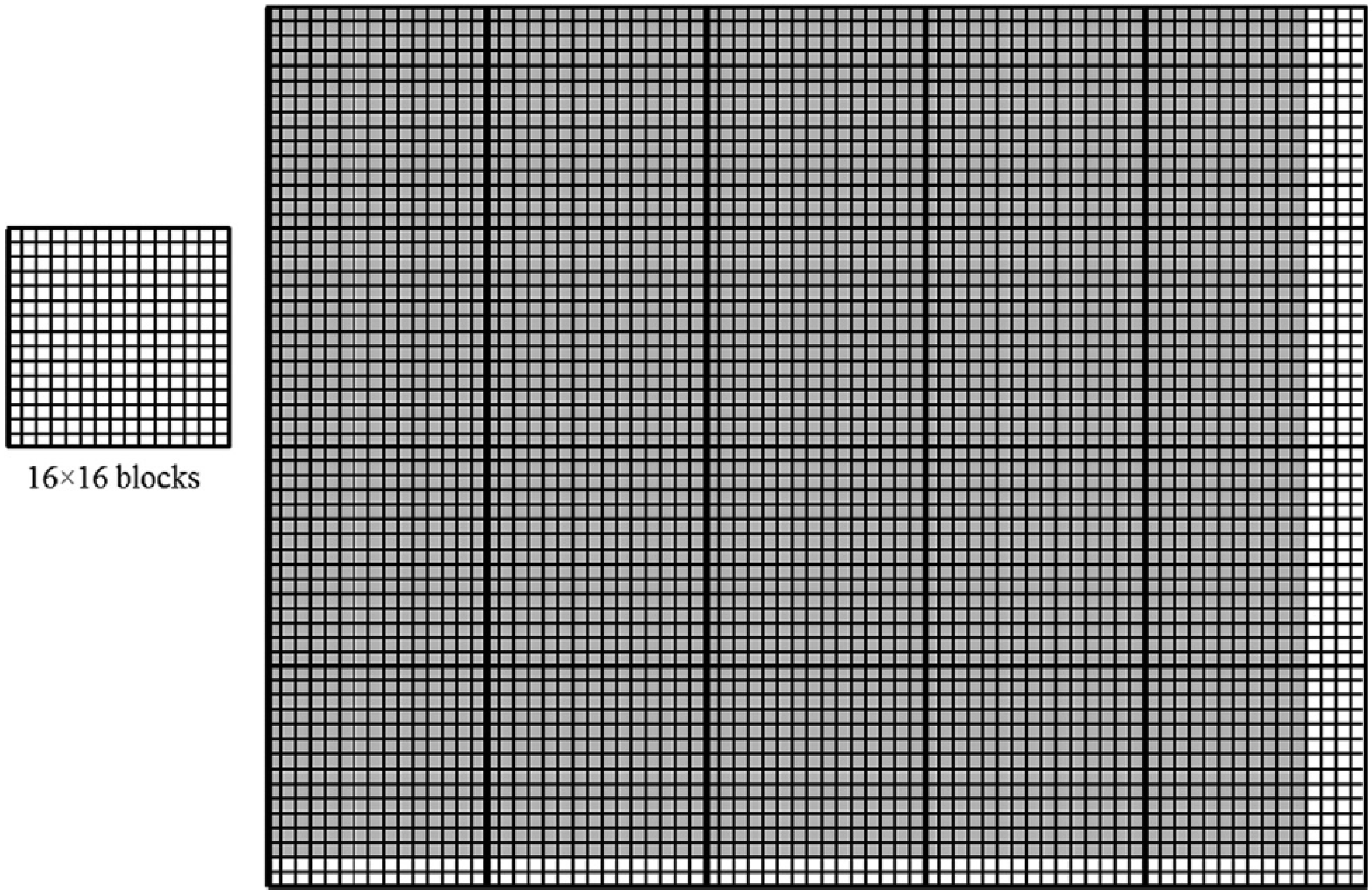

FIGURE 3.2

Using a 2D thread grid to process a 62×76 picture P.

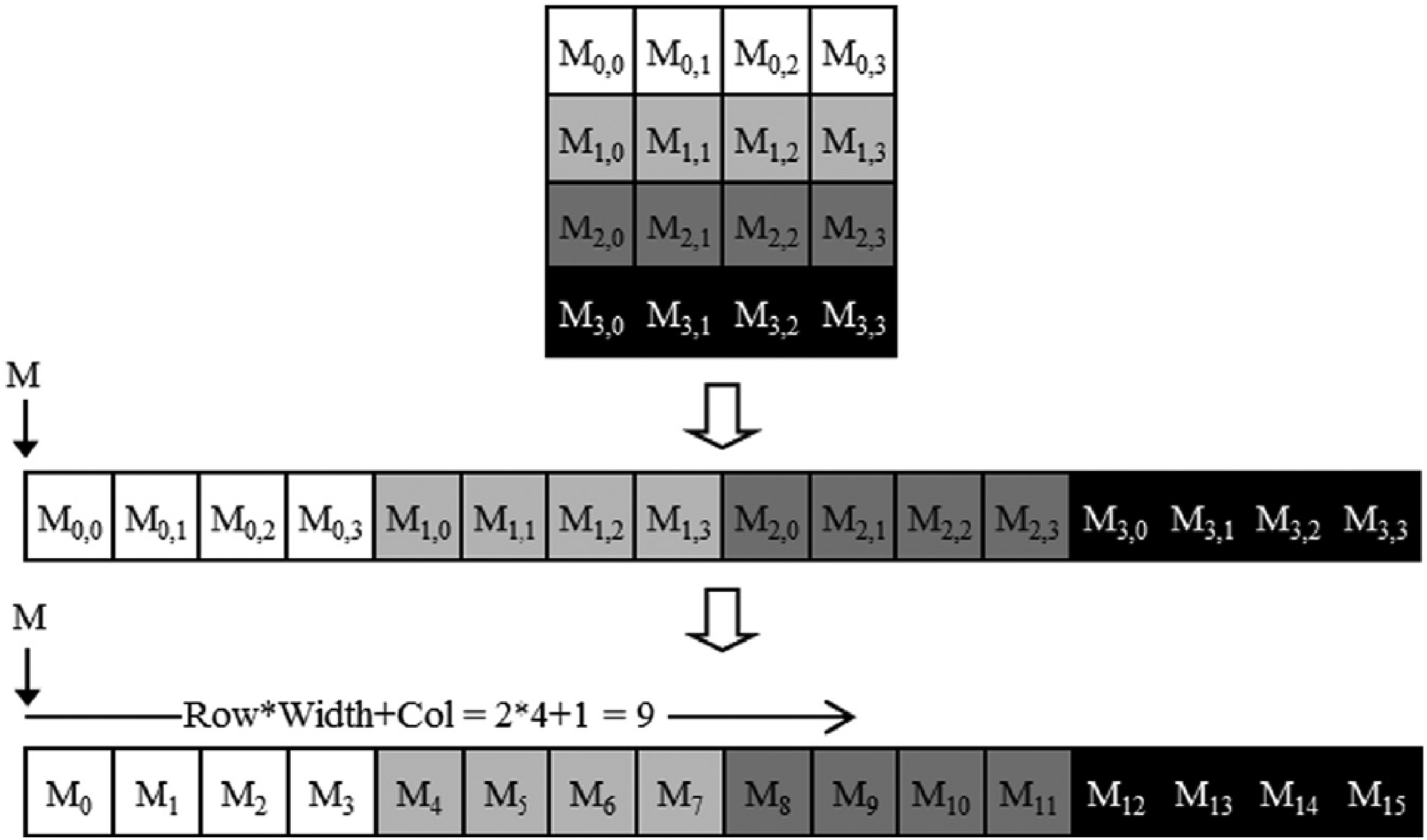

FIGURE 3.3

Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression j*Width+i for an element that is in the jth row and ith column of an array of Width elements in each row.

```
// The input image is encoded as unsigned chars [0, 255]
// Each pixel is 3 consecutive chars for the 3 channels (RGB)
__global__
void colortoGrayscaleConvertion(unsigned char * Pout,
                unsigned char * Pin, int width, int height) {
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    if (col < width && row < height) {
        // Get 1D offset for the grayscale image
        int grayOffset = row*width + col;
        // One can think of the RGB image having CHANNEL
        // times more columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = Pin[rgbOffset    ]; // Red value
        unsigned char g = Pin[rgbOffset + 1]; // Green value
        unsigned char b = Pin[rgbOffset + 2]; // Blue value
        // Perform the rescaling and store it
        // We multiply by floating point constants
        Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

FIGURE 3.4

Source code of `colorToGrayscaleConversion` with 2D thread mapping to data.

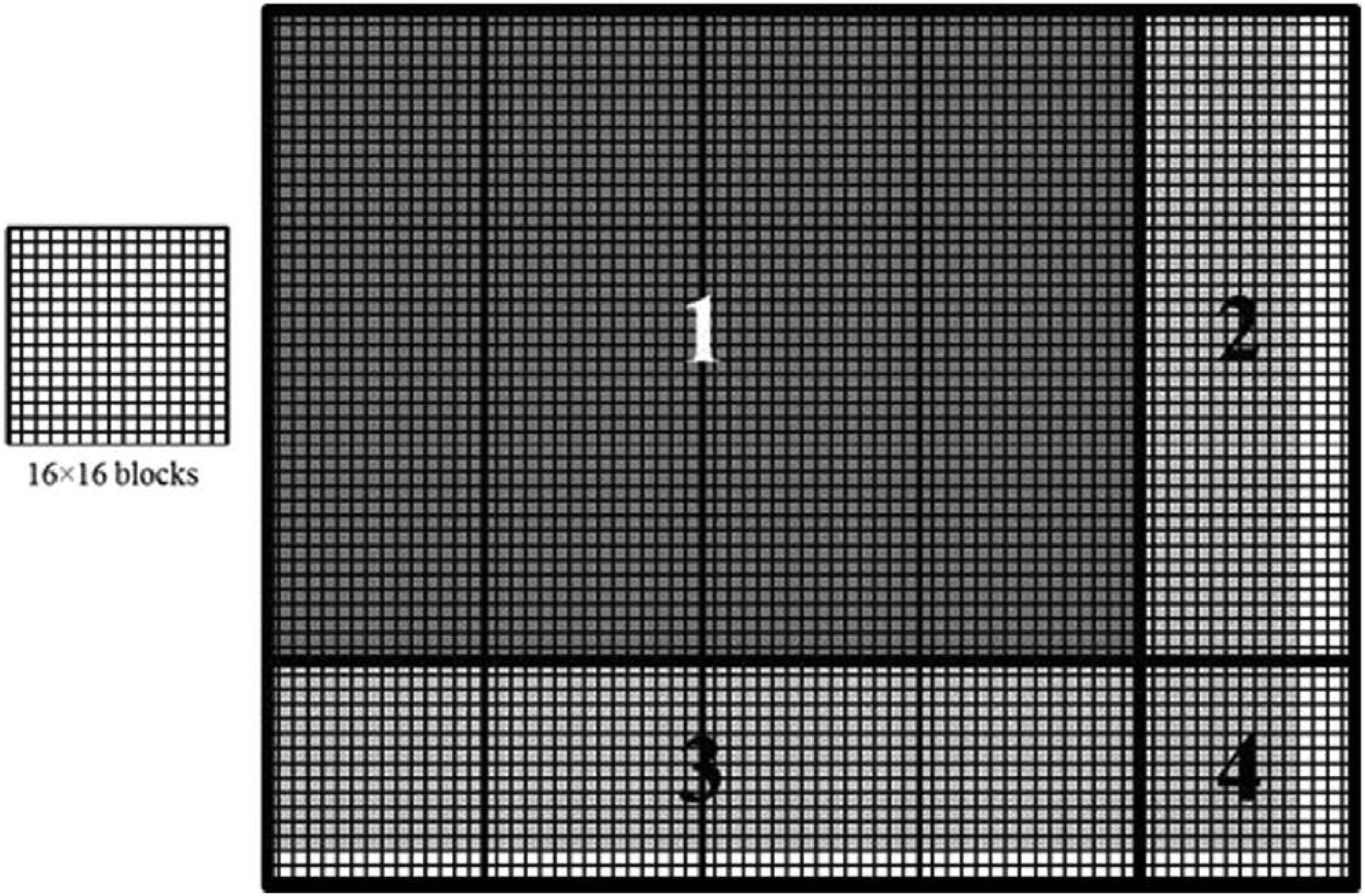

FIGURE 3.5

Covering a 76×62 picture with 16×16 blocks.

FIGURE 3.6

An original image (*left*) and a blurred version (*right*).

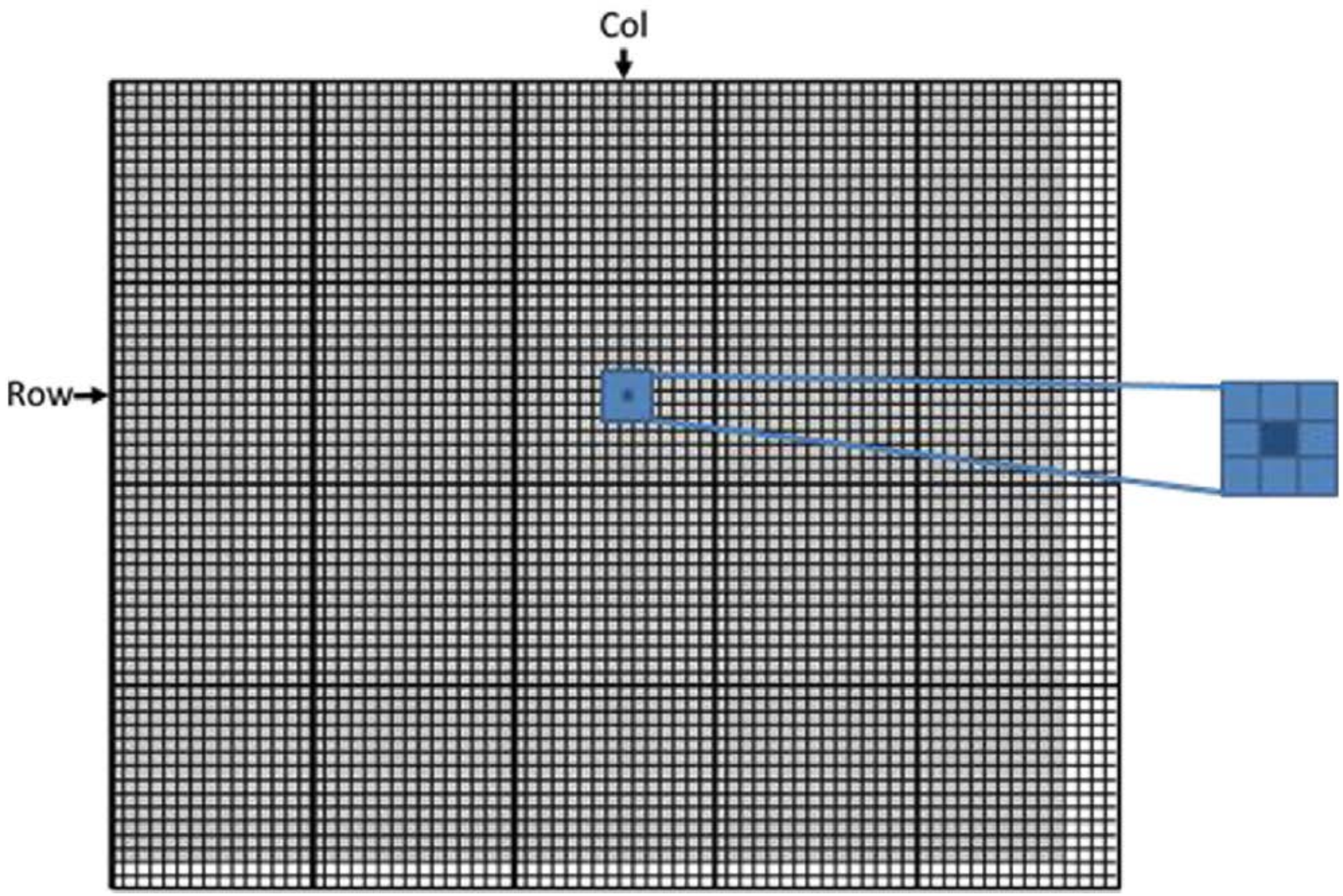

FIGURE 3.7

Each output pixel is the average of a patch of surrounding pixels and itself in the input image.

```
__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 average of the surrounding BLUR_SIZE x BLUR_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>=0 && curRow<h && curCol>=0 && curCol<w) {
               pixVal += in[curRow*w + curCol];
               ++pixels; // Keep track of number of pixels in the avg
             }
           }
        }
        // Write our new pixel value out
        out[row*w + col] = (unsigned char)(pixVal/pixels);
    }
}
```

FIGURE 3.8

An image blur kernel.

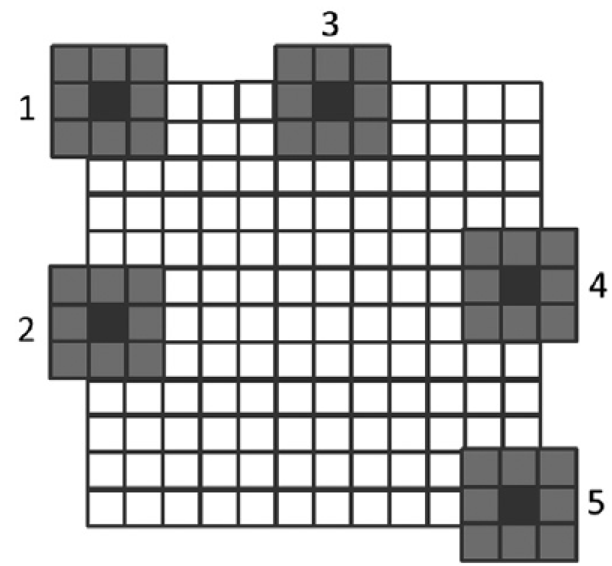

FIGURE 3.9

Handling boundary conditions for pixels near the edges of the image.

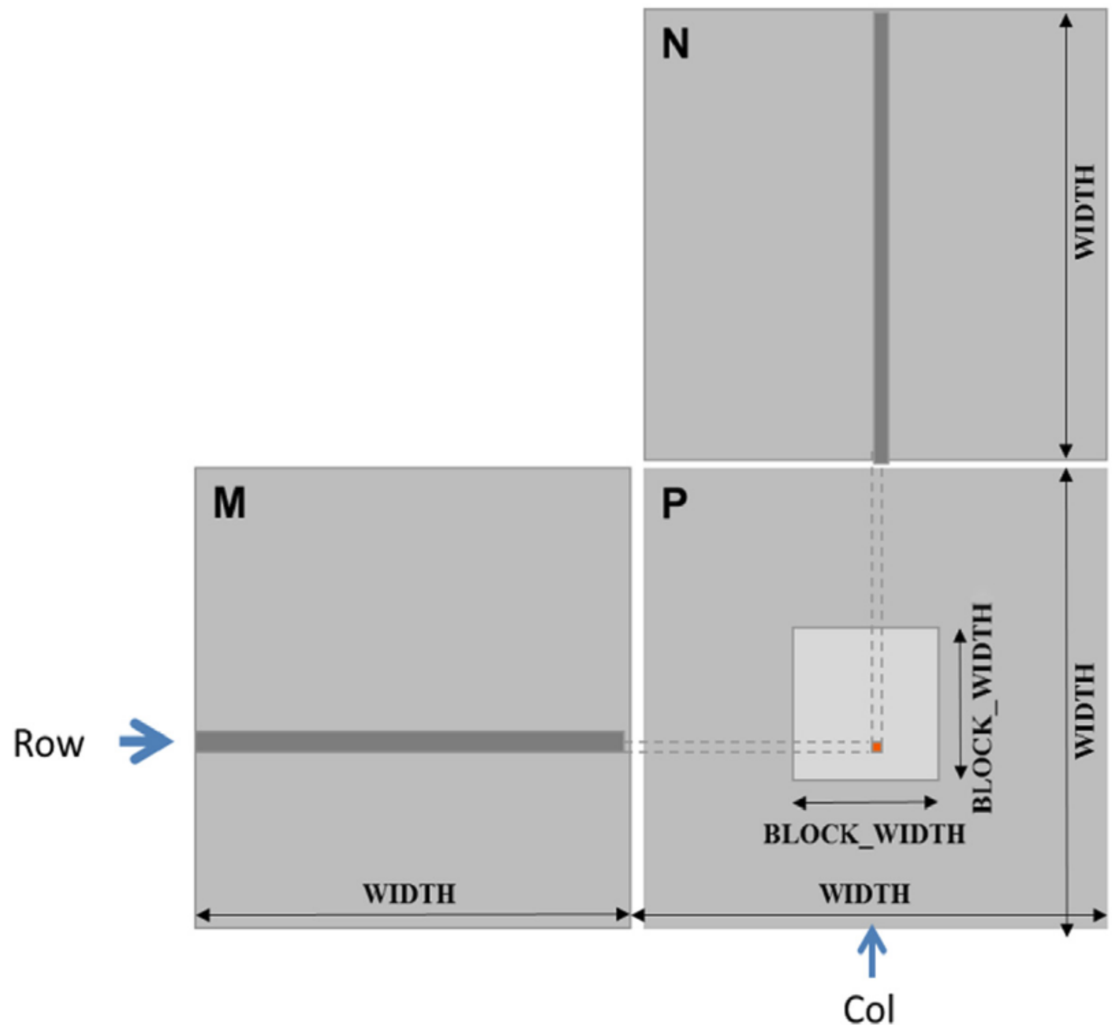

FIGURE 3.10

Matrix multiplication using multiple blocks by tiling P.

```
__global__ void MatrixMulKernel(float* M, float* N,
                                float* P, int Width) {
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;
    if ((row < Width) && (col < Width)) {
        float Pvalue = 0;
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[row*Width+k]*N[k*Width+col];
        }
        P[row*Width+col] = Pvalue;
    }
}
```

FIGURE 3.11

A matrix multiplication kernel using one thread to compute one P element.

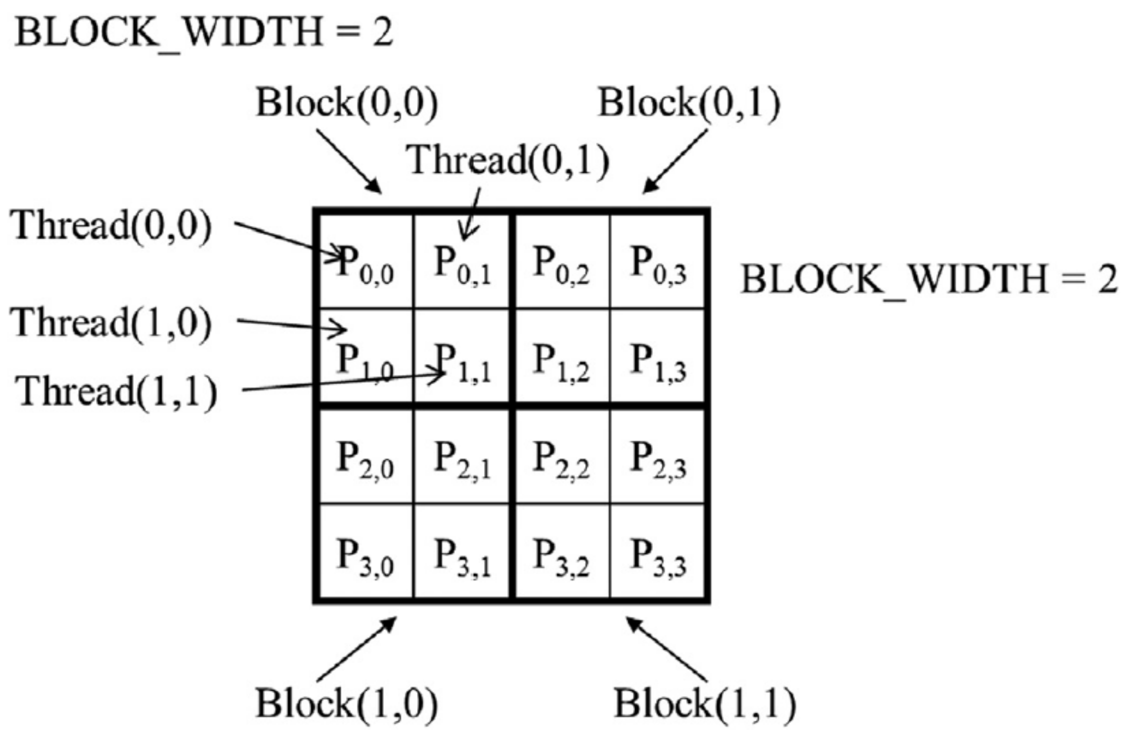

FIGURE 3.12

A small execution example of `matrixMulKernel`.

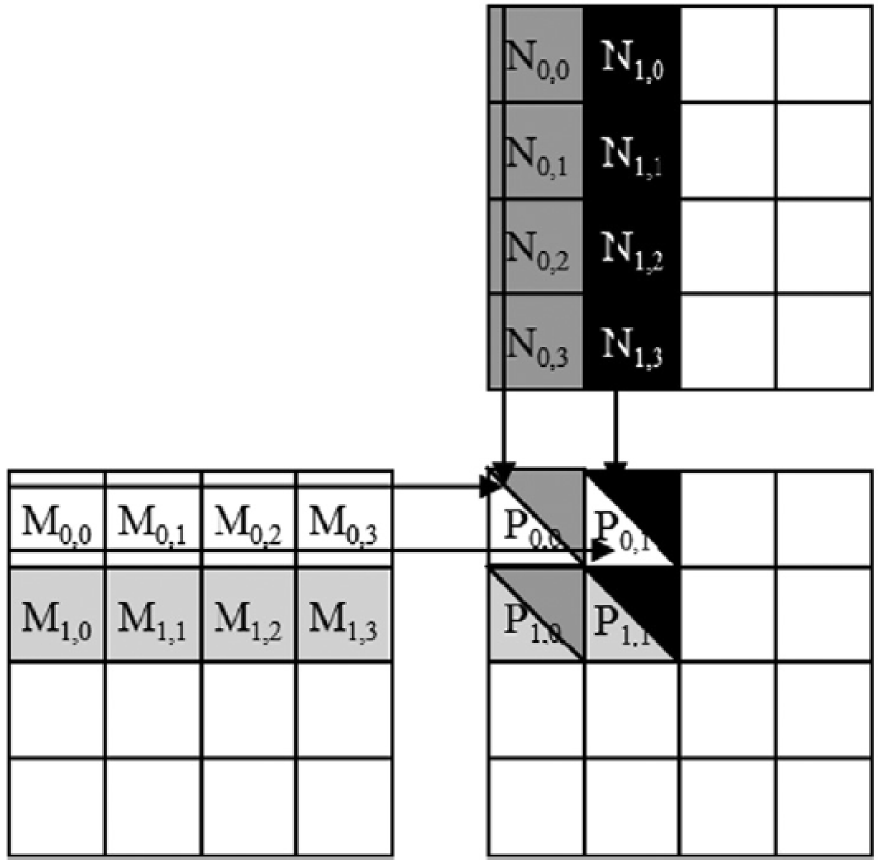

FIGURE 3.13

Matrix multiplication actions of one thread block.

```
dim3 dimGrid(32, 1, 1);
dim3 dimBlock(128, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```

In-text figure 1

```
dim3 dog(32, 1, 1);
dim3 cat(128, 1, 1);
vecAddKernel<<<dog, cat>>>(...);
```

In-text figure 2

```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```

In-text figure 3

```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```

In-text figure 4

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

In-text figure 5

```
dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1);
dim3 dimBlock(16, 16, 1);
colorToGrayscaleConversion<<<dimGrid,dimBlock>>>
                              (Pin_d, Pout_d, m, n);
```

In-text figure 6

```
col = blockIdx.x*blockDim.x + threadIdx.x
```

In-text figure 7

```
int plane = blockIdx.z*blockDim.z + threadIdx.z
```

In-text figure 8

```
row = blockIdx.y*blockDim.y + threadIdx.y
```

In-text figure 9

```
col = blockIdx.x*blockDim.x + threadIdx.x
```

In-text figure 10

```
__global__ void foo_kernel(float* a, float* b, unsigned int M, unsigned int N) {
    unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
    if(row < M && col < N) {
        b[row*N + col] = a[row*N + col]/2.1f + 4.8f;
    }
}
void foo(float* a_d, float* b_d) {
    unsigned int M = 150;
    unsigned int N = 300;
    dim3 bd(16, 32);
    dim3 gd((N - 1)/16 + 1, (M - 1)/32 + 1);
    foo_kernel <<< gd, bd >>>(a_d, b_d, M, N);
}
```

In-text figure 11