Chapter-3
Scalable parallel execution

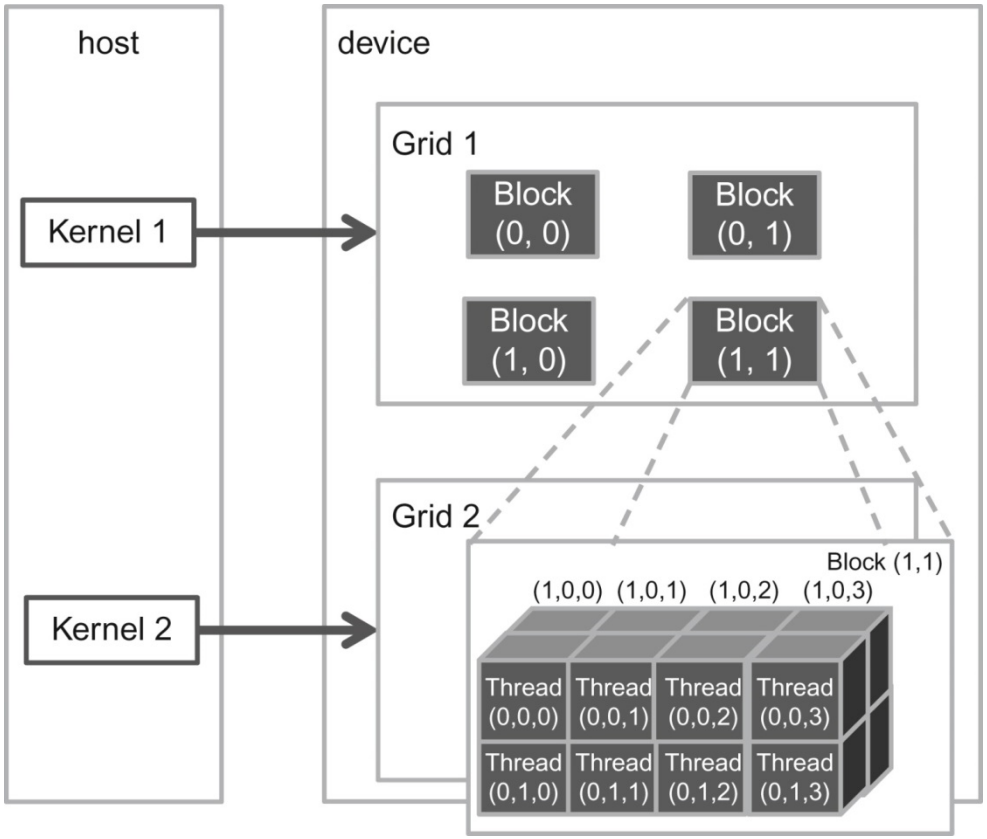

FIGURE 3.1: A multidimensional example of CUDA grid organization.

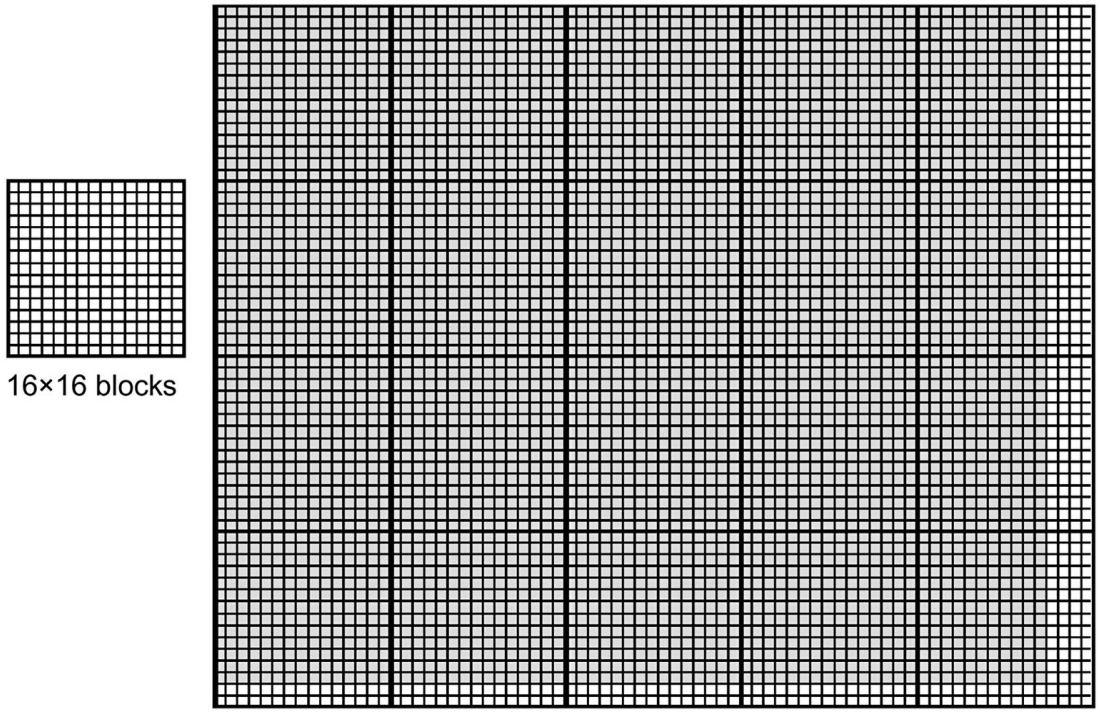

FIGURE 3.2: Using a 2D thread grid to process a 76 × 62 picture *P*.

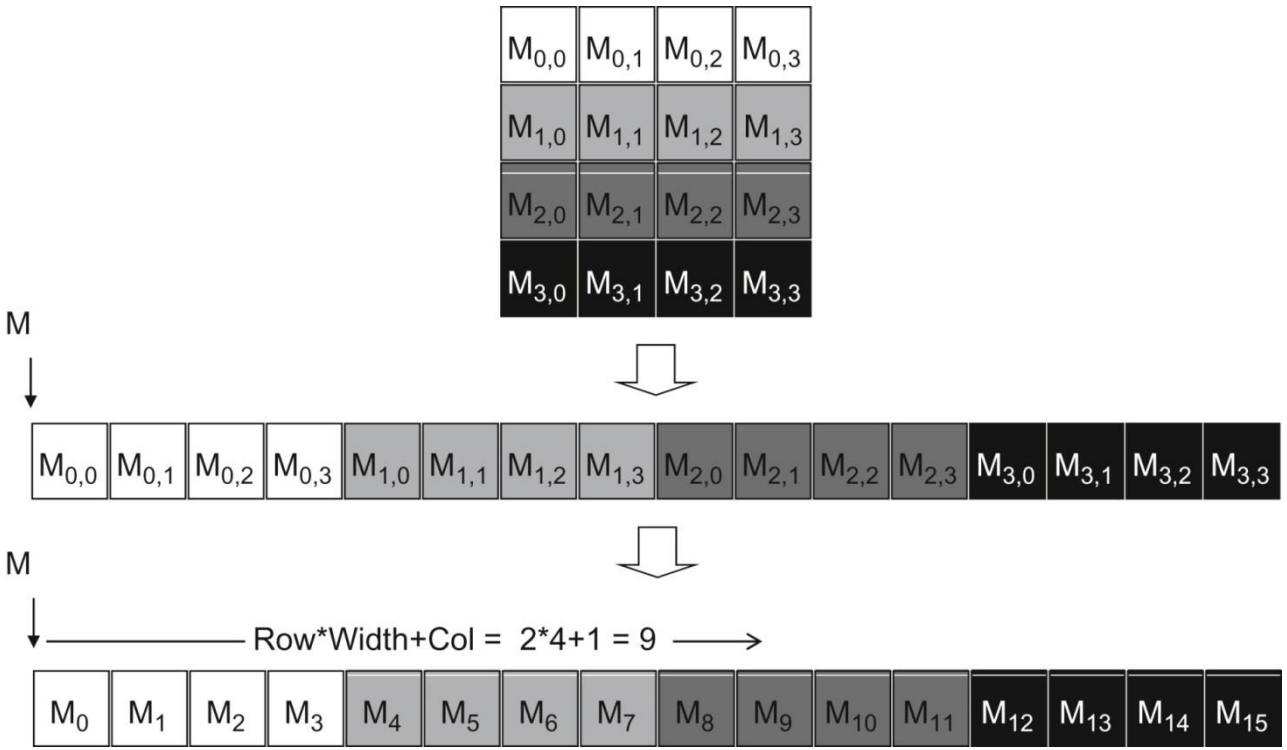

FIGURE 3.3: Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression *j**Width+ *i* for an element that is in the *j* th row and *i* th column of an array of Width elements in each row.

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned char * Pout,  unsigned
             char * Pin, int width, int height) {,
int Col = threadIdx.x + blockIdx.x * blockDim.x;
int Row = threadIdx.y + blockIdx.y * blockDim.y;
if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r = Pin[rgbOffset    ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 2]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 3]; // blue value for pixel
   // 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 colorToGreyscaleConversion showing 2D thread mapping to data.

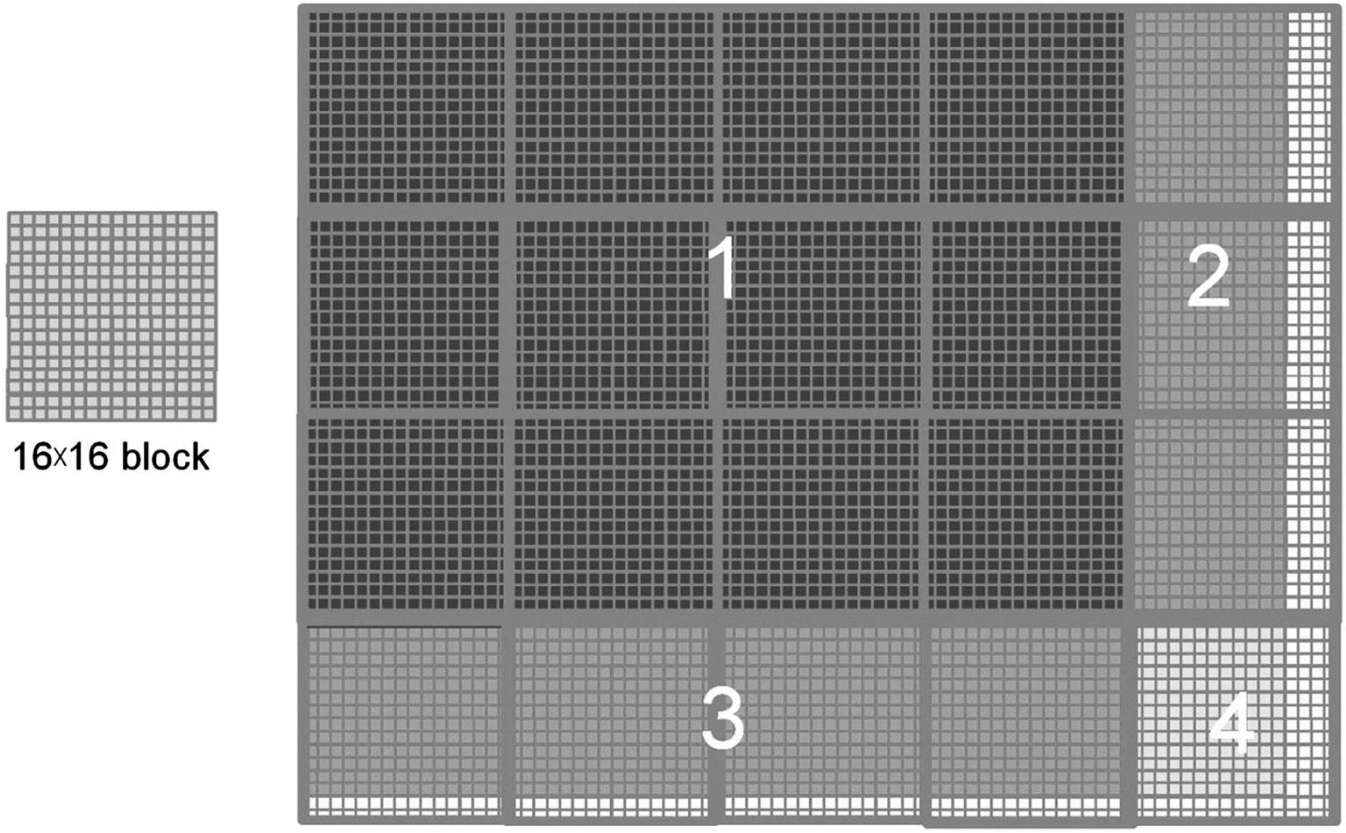

FIGURE 3.5: Covering a 76 × 62 picture with 16 × 16 blocks.

FIGURE 3.6: An original image and a blurred version.

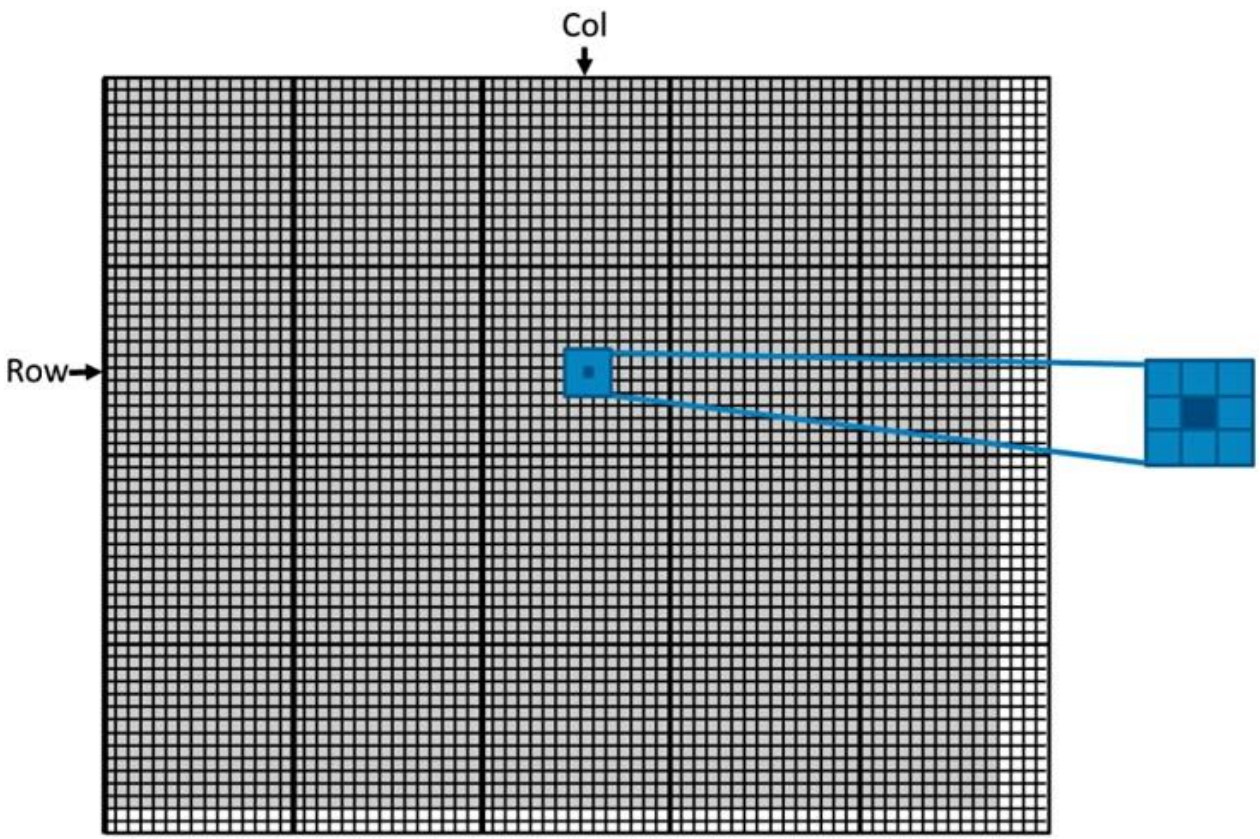

FIGURE 3.7: Each output pixel is the average of a patch of pixels in the input image.

```
__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) {
1.     int pixVal = 0;
2.     int pixels = 0;

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

5.         int curRow = Row + blurRow;
6.         int curCol = Col + blurCol;
         // Verify we have a valid image pixel
7.         if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {

8.           pixVal += in[curRow * w + curCol];
9.           pixels++; // Keep track of number of pixels in the avg
           }
         }
       }

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

FIGURE 3.8: An image blur kernel.

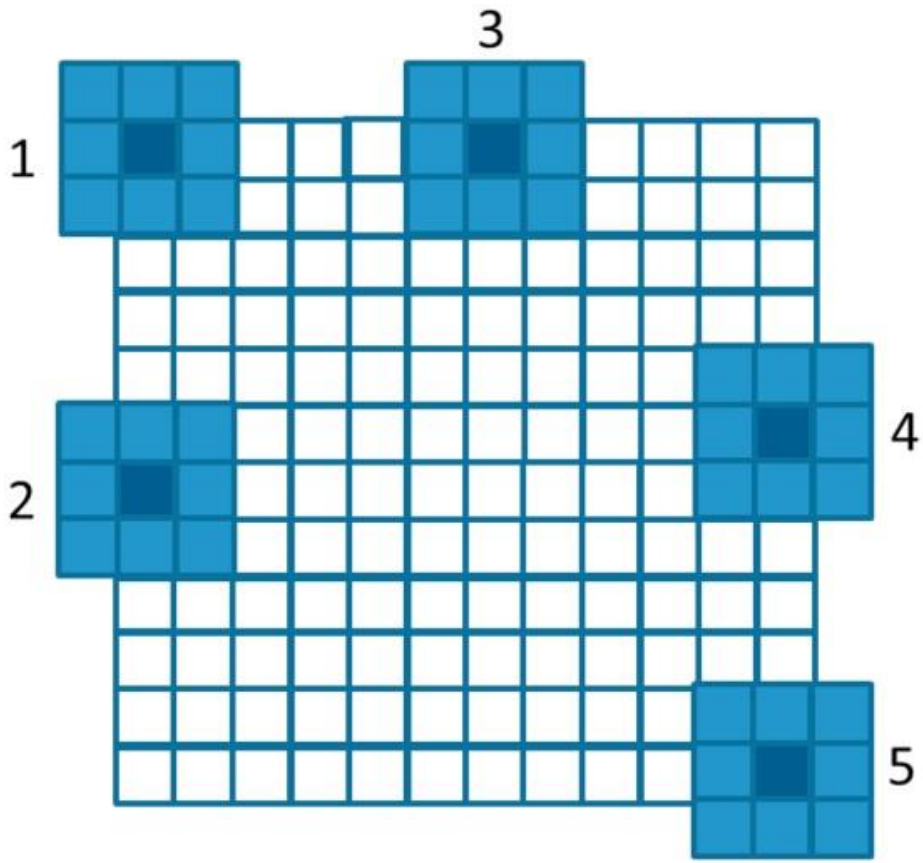

FIGURE 3.9: Handling boundary conditions for pixels near the edges of the image.

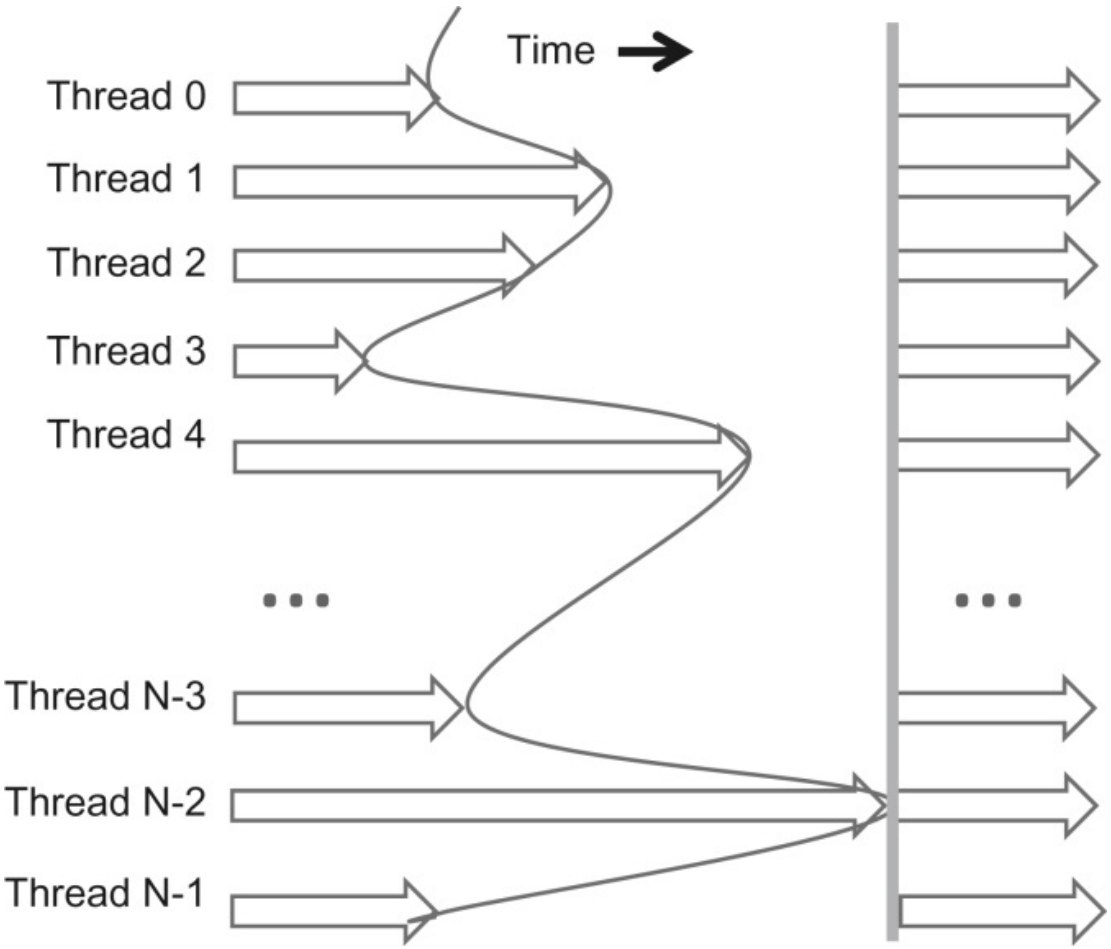

FIGURE 3.10: An example execution timing of barrier synchronization.

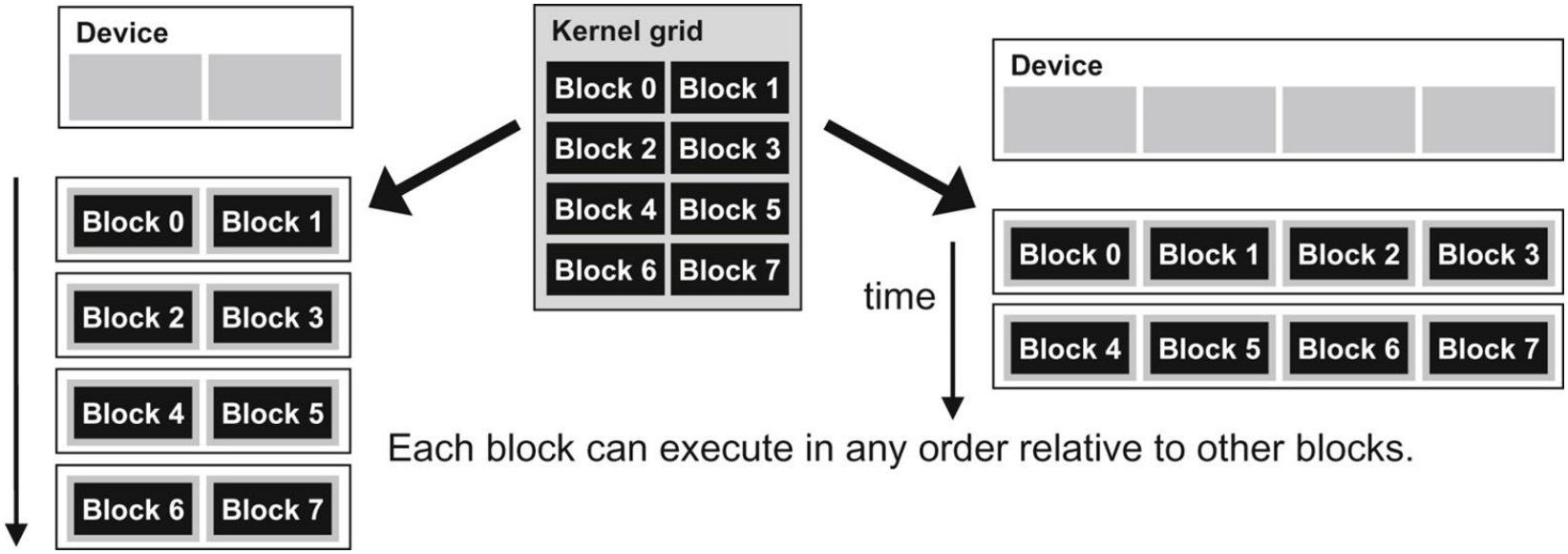

FIGURE 3.11: Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

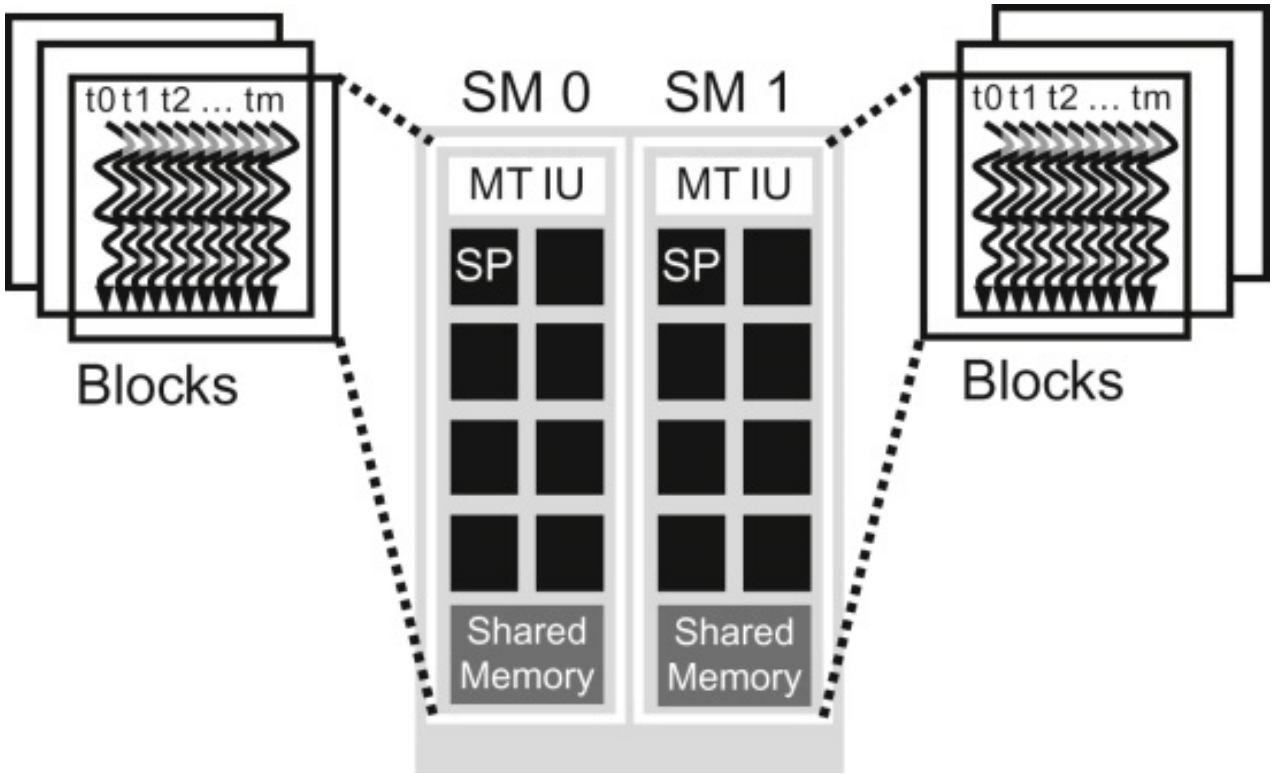

FIGURE 3.12: Thread block assignment to Streaming Multiprocessors (SMs).

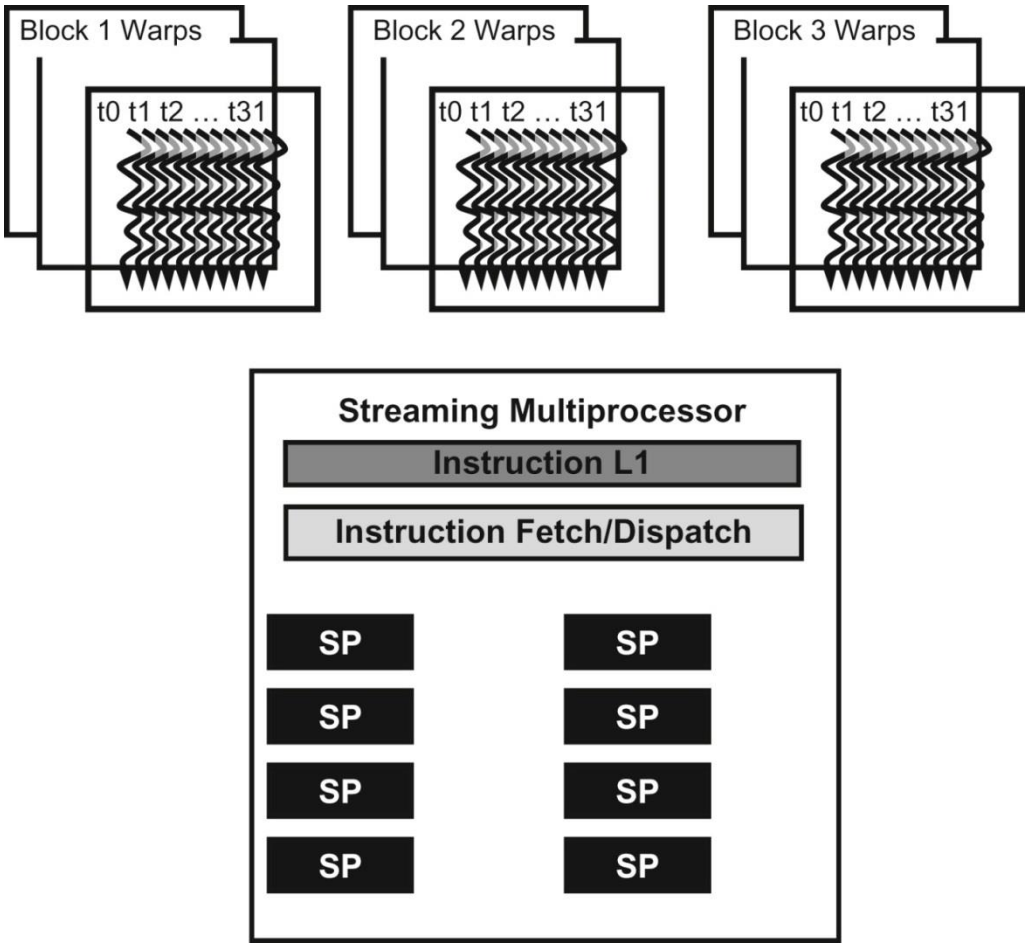

FIGURE 3.13: Blocks are partitioned into warps for thread scheduling.