

# 第二讲 – CUDA 并行化模型



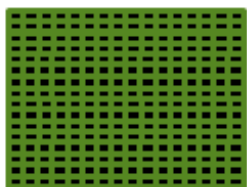
# 目录

- 多维内核配置
- 彩色图像灰度化处理
- 模糊图像处理
- 线程调度

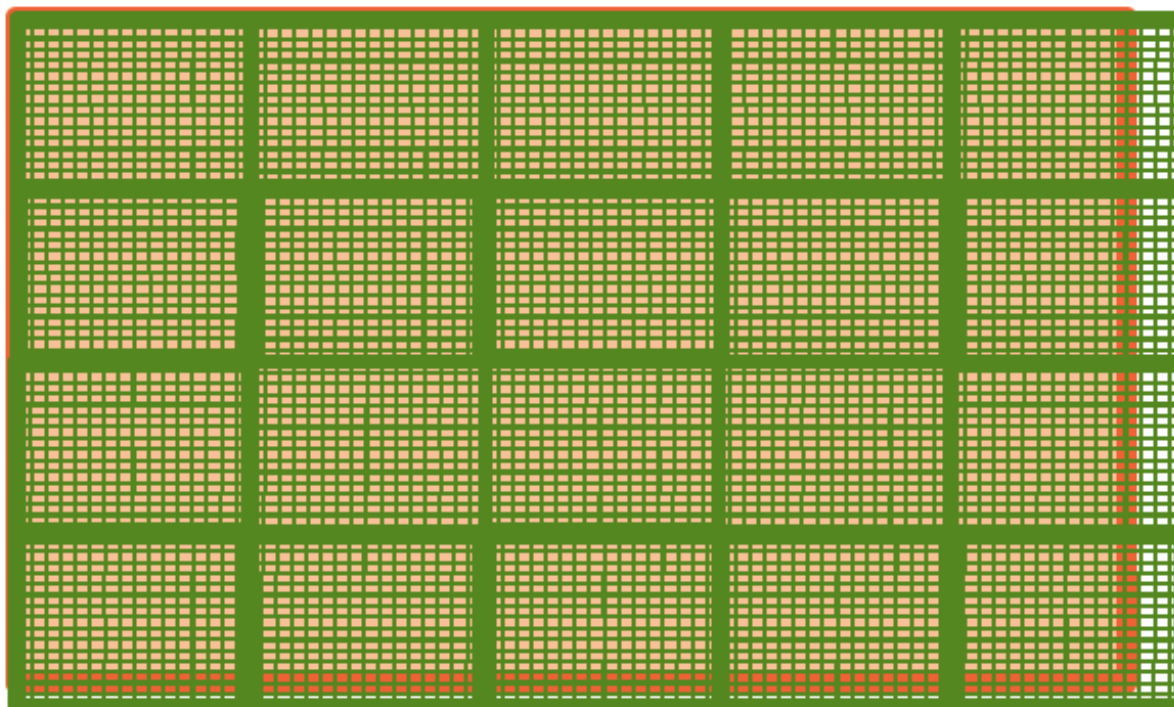
# 小节目标

- 了解多维线程网格
  - 多维线程块和线程索引
  - 将线程块/线程索引映射到数据索引

# 使用 二维线程网格处理图片



16×16 blocks



62×76 picture

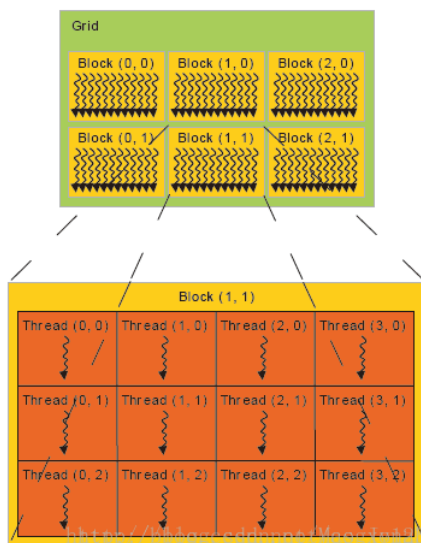


# 局部坐标与全局坐标映射

CUDA用四个内建变量标识线程、线程块的位置和大小。

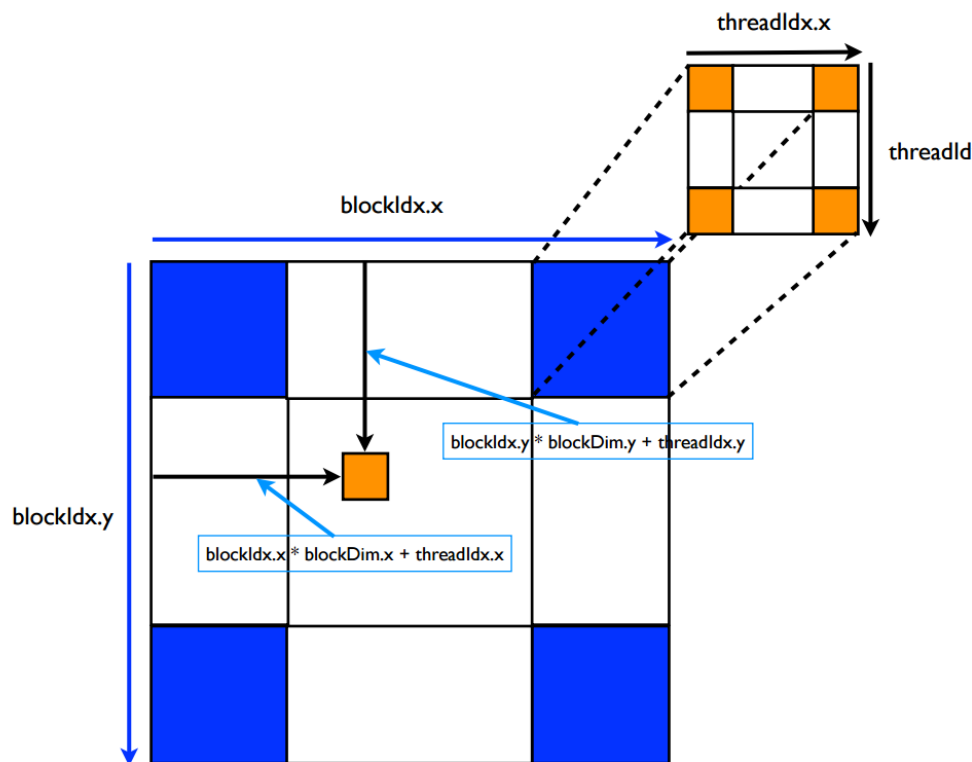
`threadIdx`, `blockIdx`

`blockDim`, `gridDim`



`gridDim.x=3`  
`gridDim.y=2`

`blockDim.x=4`  
`blockDim.y=3`



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

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



# C/C++ 中的行主序布局

M

$$\text{Row} * \text{Width} + \text{Col} = 2 * 4 + 1 = 9$$

M <sub>0</sub>	M <sub>1</sub>	M <sub>2</sub>	M <sub>3</sub>	M <sub>4</sub>	M <sub>5</sub>	M <sub>6</sub>	M <sub>7</sub>	M <sub>8</sub>	M <sub>9</sub>	M <sub>10</sub>	M <sub>11</sub>	M <sub>12</sub>	M <sub>13</sub>	M <sub>14</sub>	M <sub>15</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

M

M <sub>0,0</sub>	M <sub>0,1</sub>	M <sub>0,2</sub>	M <sub>0,3</sub>	M <sub>1,0</sub>	M <sub>1,1</sub>	M <sub>1,2</sub>	M <sub>1,3</sub>	M <sub>2,0</sub>	M <sub>2,1</sub>	M <sub>2,2</sub>	M <sub>2,3</sub>	M <sub>3,0</sub>	M <sub>3,1</sub>	M <sub>3,2</sub>	M <sub>3,3</sub>
------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------

M <sub>0,0</sub>	M <sub>0,1</sub>	M <sub>0,2</sub>	M <sub>0,3</sub>
M <sub>1,0</sub>	M <sub>1,1</sub>	M <sub>1,2</sub>	M <sub>1,3</sub>
M <sub>2,0</sub>	M <sub>2,1</sub>	M <sub>2,2</sub>	M <sub>2,3</sub>
M <sub>3,0</sub>	M <sub>3,1</sub>	M <sub>3,2</sub>	M <sub>3,3</sub>



# PictureKernel 源代码

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width)
{
    // Calculate the row # of the d_Pin and d_Pout element
    int Row = blockIdx.y*blockDim.y + threadIdx.y;

    // Calculate the column # of the d_Pin and d_Pout element
    int Col = blockIdx.x*blockDim.x + threadIdx.x;

    // each thread computes one element of d_Pout if in range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

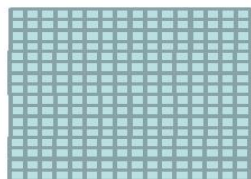


# PictureKernel 主机代码

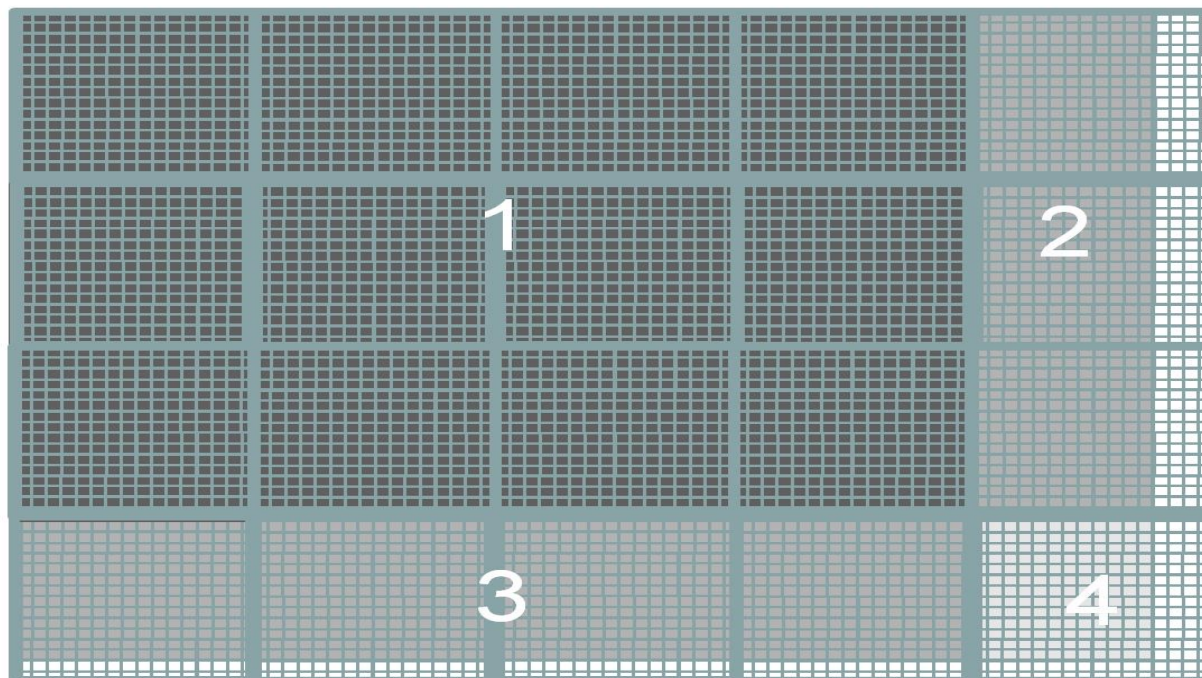
```
// assume that the picture is m × n,  
// m pixels in y dimension and n pixels in x dimension  
// input d_Pin has been allocated on and copied to device  
// output d_Pout has been allocated on device  
...  
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);  
dim3 DimBlock(16, 16, 1);  
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);  
...
```



# 使用 $16 \times 16$ 线程块覆盖 $62 \times 76$ 的图片



$16 \times 16$  block



# 目录

- 多维内核配置
- **彩色到灰度图像处理示例**
- 模糊图像处理示例
- 线程调度



# 小节目标

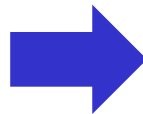
- 更深入地了解多维线程网格的内核配置
  - 通过真实的使用案例



# 目录

- 多维内核配置
- **彩色图像灰度化处理**
- 模糊图像处理
- 线程调度

# RGB图像到灰度图像的转换

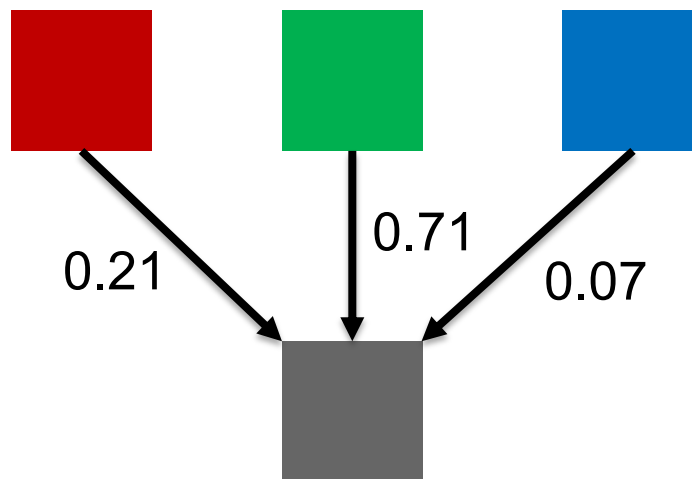


灰度数字图像是其中每个像素的值仅携带强度信息的图像。



# 颜色计算公式

- 对于 (I, J) 位置上的像素点 (r g b) :  
$$\text{grayPixel}[I,J] = 0.21*r + 0.71*g + 0.07*b$$
  - 对<[r,g,b],[0.21,0.71,0.07]>进行点乘操作



# RGB图像到灰度图像的转换代码

```
#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]; // 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;  
    }  
}
```

```

#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]; // red value for pixel
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
        unsigned char b = rgbImage[rgbOffset + 2]; // 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;
    }
}

```

```

#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]; // 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;
    }
}

```

# 目录

- 多维内核配置
- 彩色图像灰度化处理
- **模糊图像处理**
- 线程调度



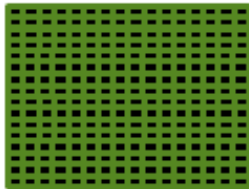
# 小节目标

- 学习具有更复杂计算和内存访问模式的二维内核

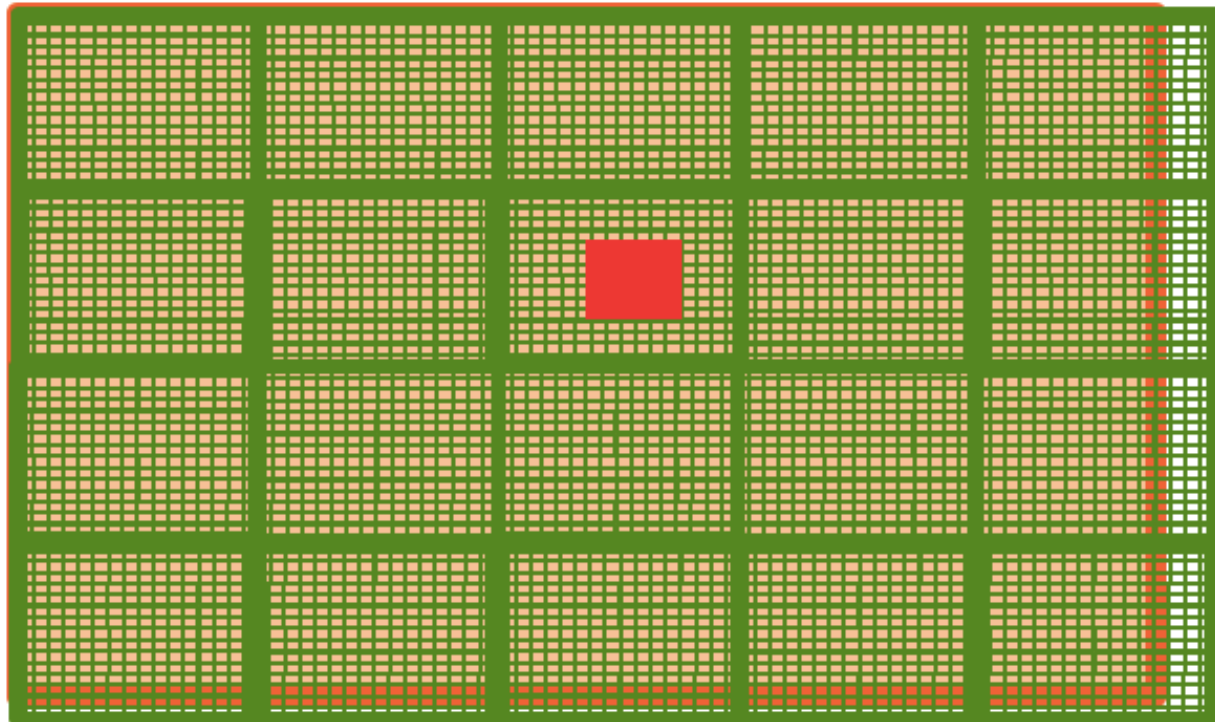
# 图像模糊



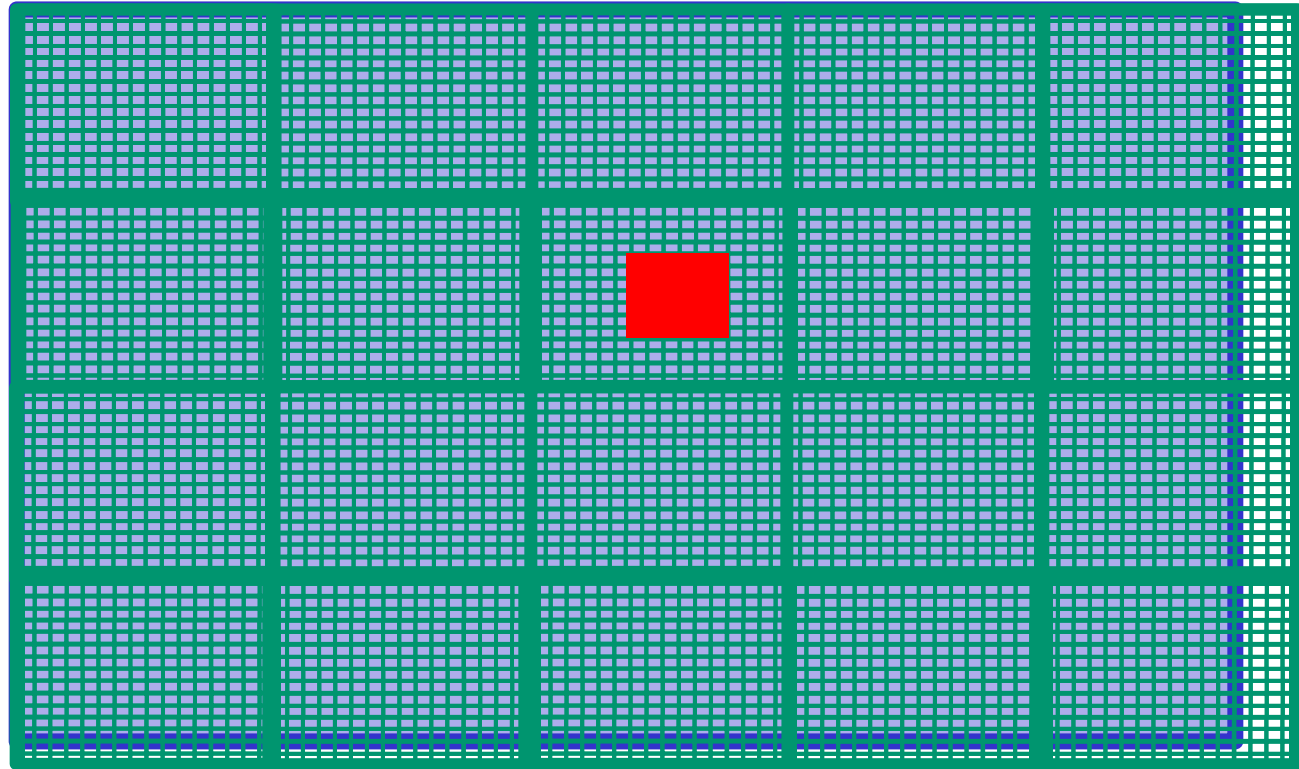
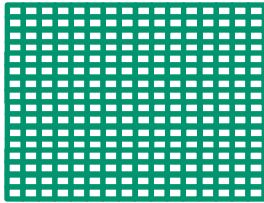
# Blurring Box



Pixels  
processed  
by a  
thread  
block



# Blurring Box



# 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);
```

```
}
```



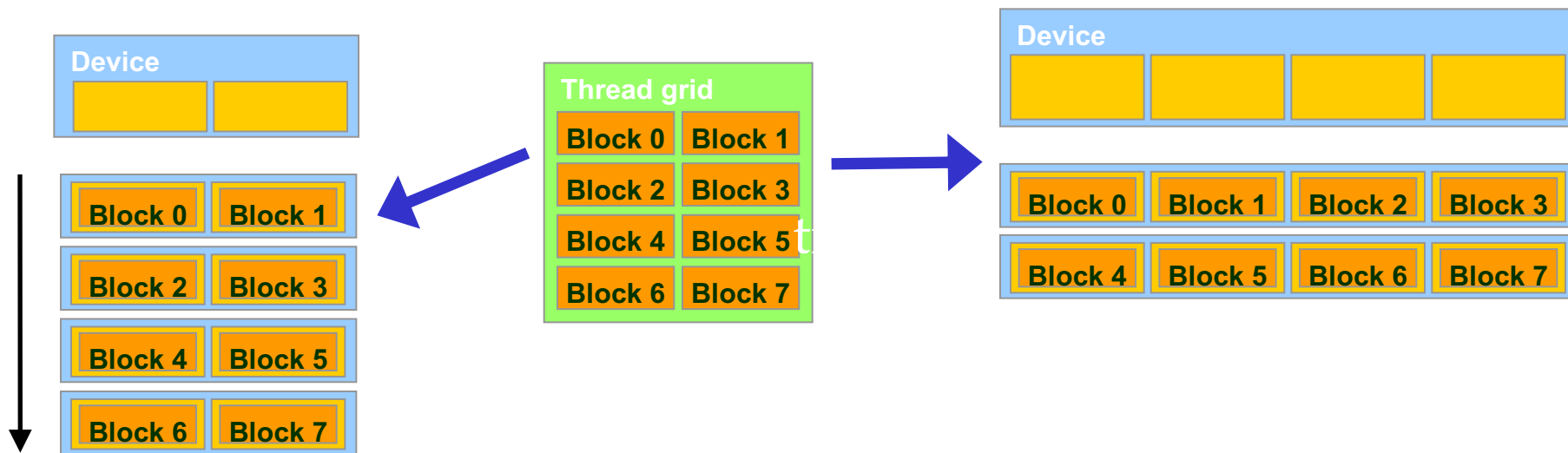
# 目录

- 多维内核配置
- 彩色到灰度图像处理示例
- 模糊图像处理示例
- **线程调度**

# 小节目标

- 了解 CUDA 内核如何利用硬件执行资源 ( hardware execution resources )
  - 将线程块分配给执行资源
  - 执行资源的容量限制
  - 零开销线程调度

# 硬件无关性

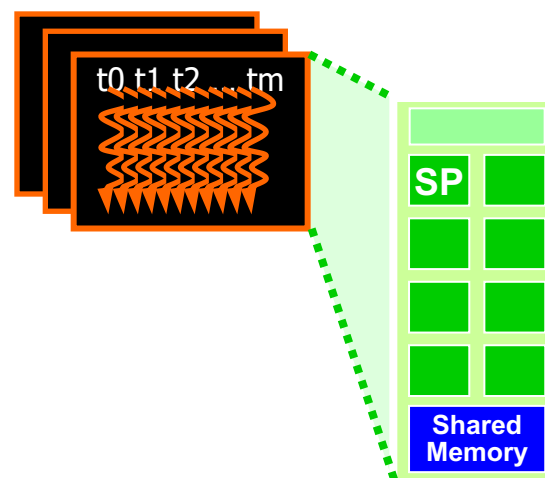


- 相对于其他线程块，每个线程块能以任何顺序执行
- 硬件可以随时自由地将线程块分配给任何处理器
  - 内核可扩展为任意数量的并行处理器

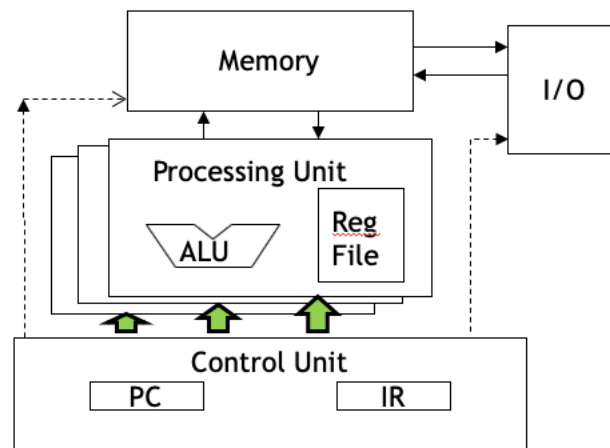
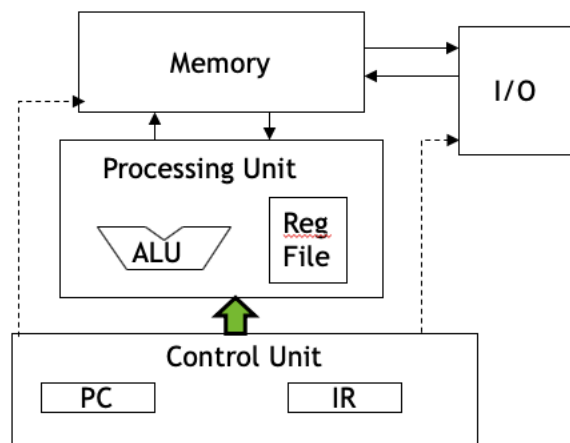


# 示例：执行线程块

- 线程以线程块的粒度分配给流式多处理器
  - 在资源允许的情况下，每个 SM 最多 **8** 个块
  - Fermi SM 最多可占用 **1536** 个线程
    - 可以是  $256 \text{ (threads/block)} * 6 \text{ blocks}$
    - 或者  $512 \text{ (threads/block)} * 3 \text{ blocks}$  等
- SM 持续维护 thread/block idx 这一参数
- SM 管理/调度线程执行



# 带有 SIMD 单元的 Von-Neumann 模型

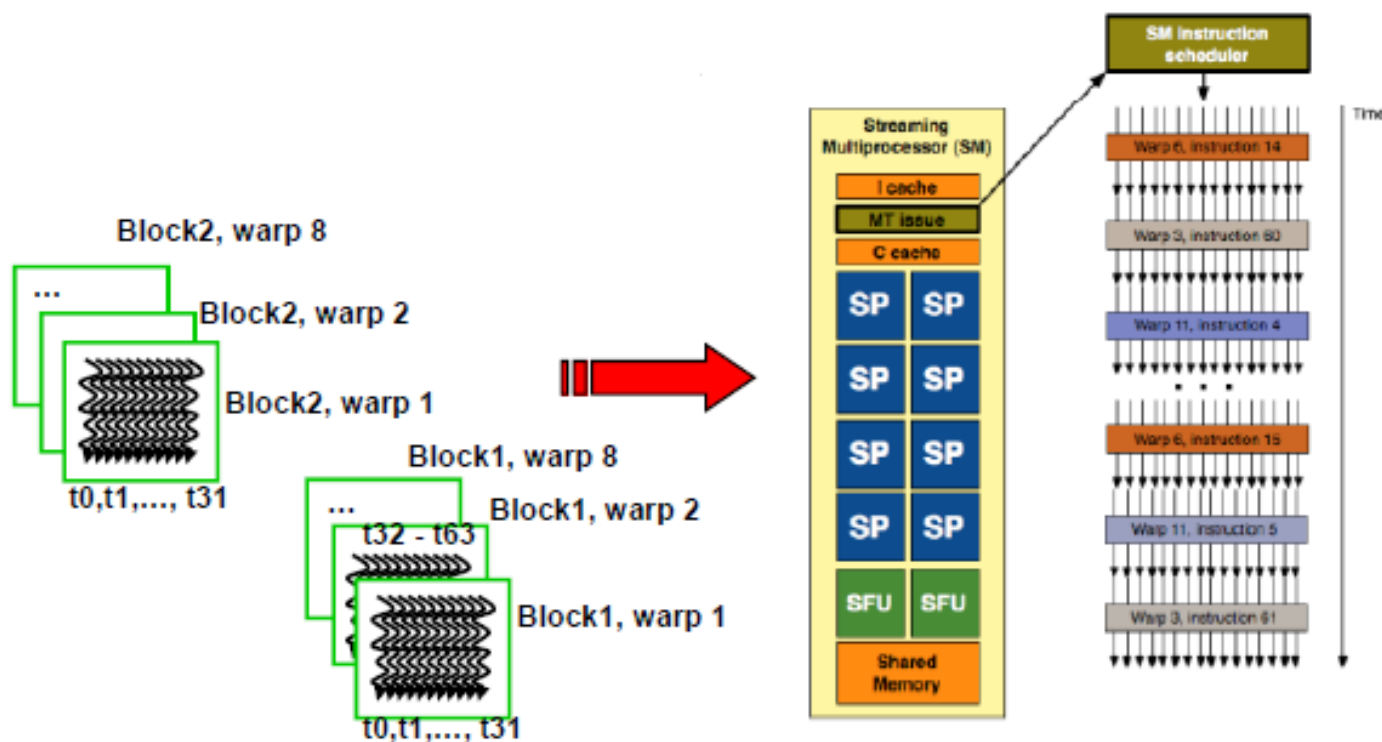


Single Instruction Multiple Data (SIMD)



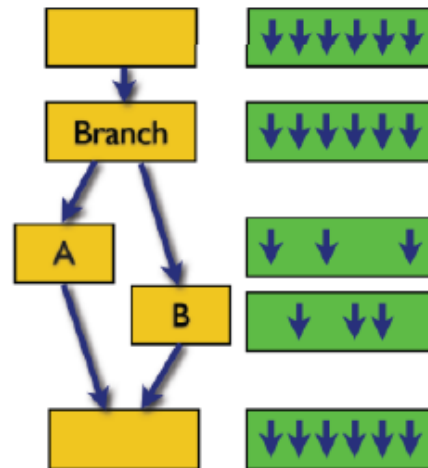
# 线程束 ( Warp ) 作为调度单元

- 每个线程块被分为多个包含 32 线程的线程束(warp)执行
  - Warp是一种硬件执行的策略，不是 CUDA 编程模型的一部分
  - 线程束是 SM 中的基本调度单元
  - 未来的 GPU 可能在每个线程束中有不同数量的线程



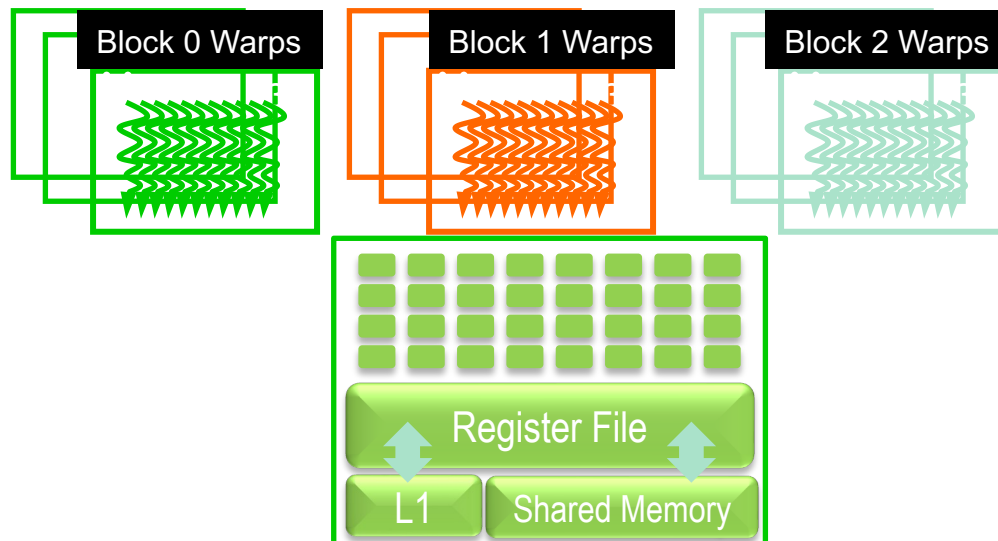
# 线程调度

- 线程束中的线程在 SIMD 中执行
  - 线程束被选中时，其中所有线程都执行相同的指令
  - N 条执行路径  $\rightarrow$  1/N 吞吐量
  - (应尽量避免在同一线程束内出现分支)
- SM 实现零开销的线程束调度
  - 下一条指令的操作数已准备好可供使用的线程束才有资格执行
  - 基于优先调度策略 ( prioritized scheduling policy ) 选择合格的线程束执行



# 关于线程束的例子

- 如果 3 个线程块分配给一个 SM，每个线程块有 256 个线程，那么一个 SM 中有多少个线程束？
  - 每个线程块被分为  $256/32 = 8$  个线程束
  - 一共有  $8 * 3 = 24$  个线程束



# 课堂问答：线程块粒度的考虑

— 对于使用多个线程块的矩阵乘法，哪种线程块配置更好？请解释原因。

（对于 Fermi 来说，每个 SM 最多可以占用 1536 个线程，最大管理8个线程块）

- A 8X8
- B 16X16
- C 32X32



**ANY QUESTIONS?**

