GPU Teaching Kit

# LECTURE 2 – CUDA PARALLELISM MODEL

Multidimensional Kernel Configuration

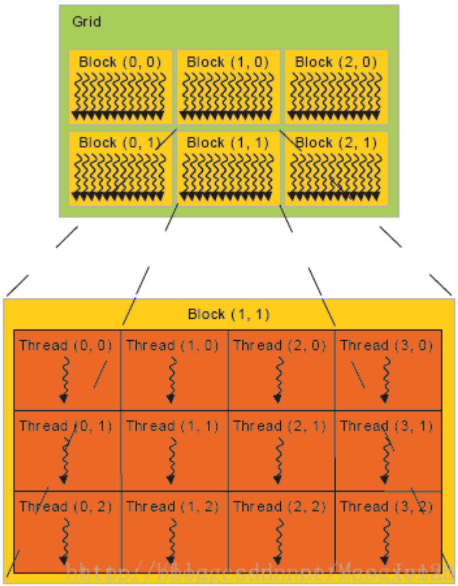
Multidimensional Kernel Configuration

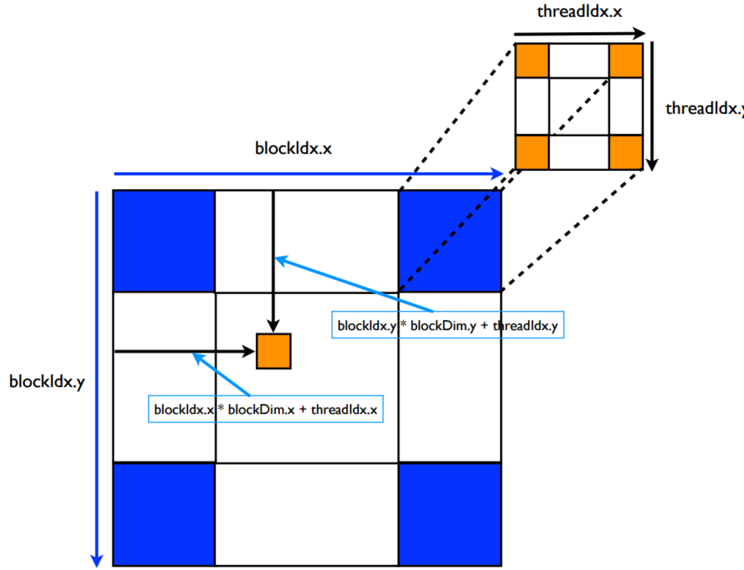
Color-to-Greyscale Image Processing Example

Blur Image Processing Example

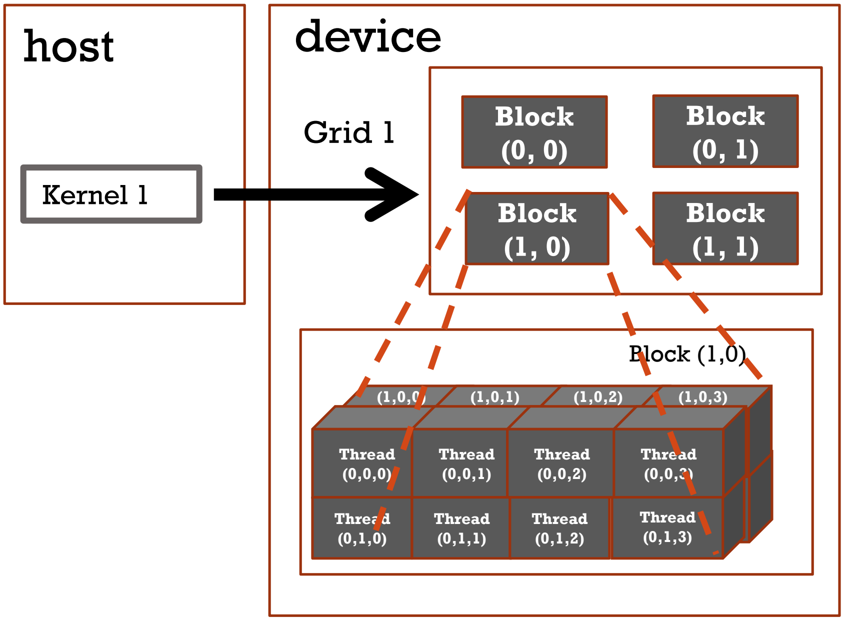
# OBJECTIVE

- **To understand multidimensional Grids**
  - Multi-dimensional block and thread indices
  - Mapping block/thread indices to data indices


Grid
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)

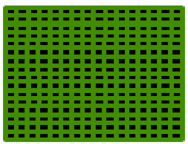

threadIdx.x
threadIdx.y
blockIdx.x
blockIdx.y
blockIdx.y * blockDim.y + threadIdx.y
blockIdx.x * blockDim.x + threadIdx.x

# A MULTI-DIMENSIONAL GRID EXAMPLE

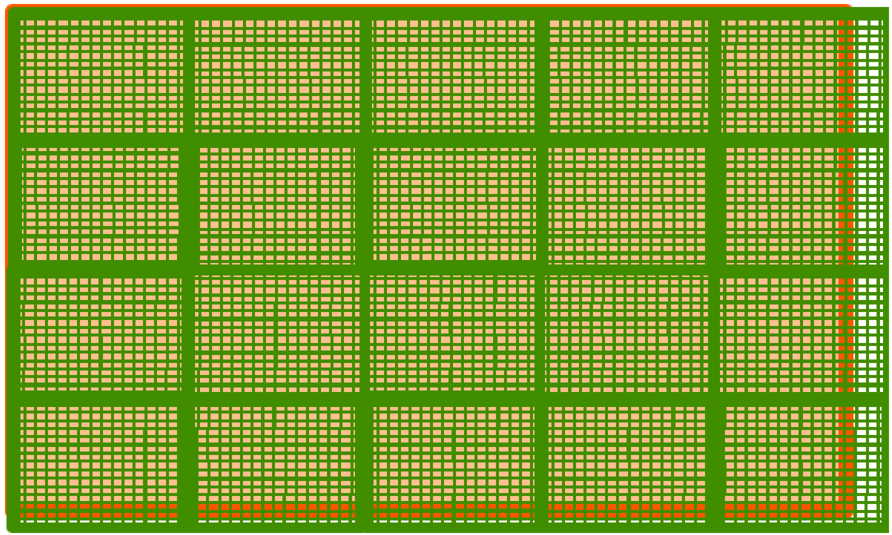



电子科技大学
University of Electronic Science and Technology of China

# PROCESSING A PICTURE WITH A 2D GRID

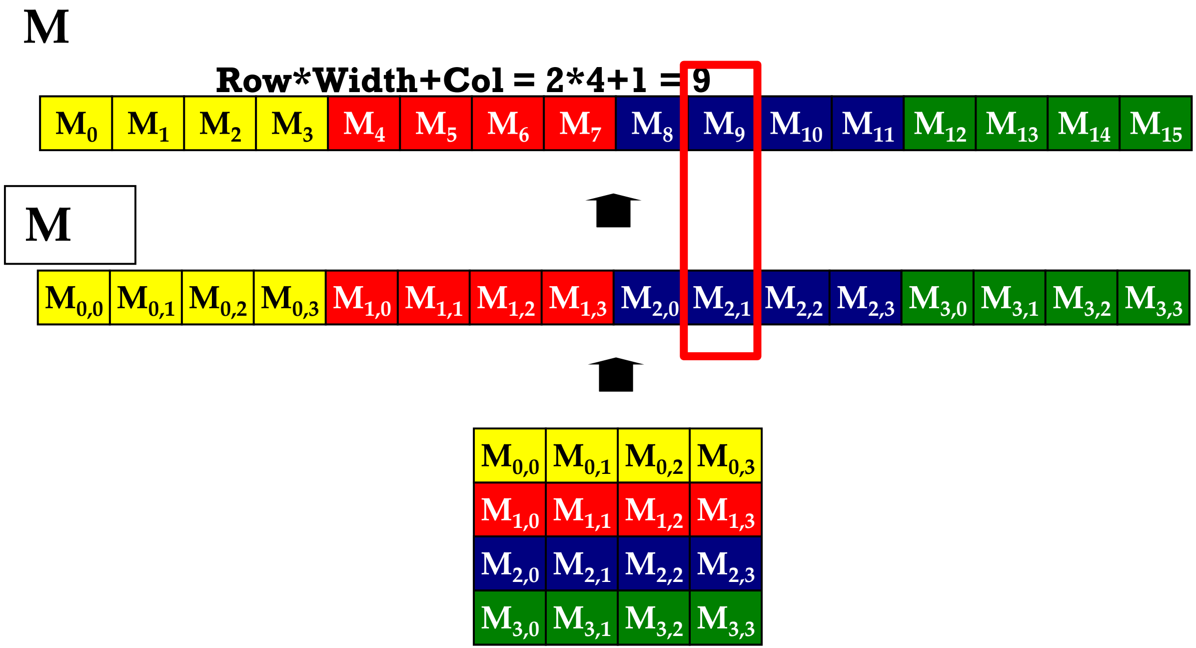


16×16 blocks



62×76 picture

# ROW-MAJOR LAYOUT IN C/C++



# SOURCE CODE OF A PICTUREKERNEL

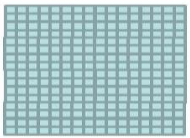
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width)
{

  // 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];
  }
}
```

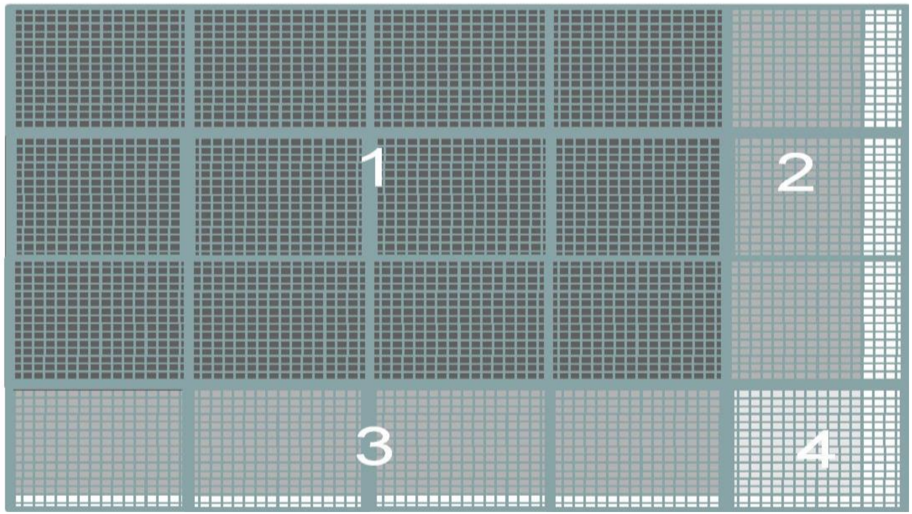
电子科技大学
University of Electronic Science and Technology of China

# HOST CODE FOR LAUNCHING PICTUREKERNEL

```
// 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);
…
```

# COVERING A 62×76 PICTURE WITH 16×16 BLOCKS




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

Multidimensional Kernel Configuration

Color-to-Greyscale Image Processing Example

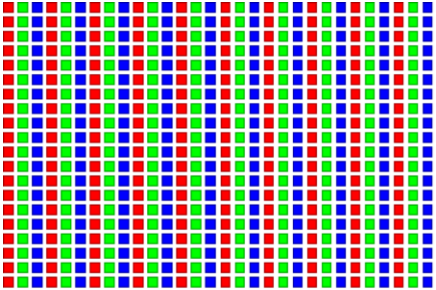
Blur Image Processing Example

# OBJECTIVE

- **To gain deeper understanding of multi-dimensional grid kernel configurations through a real-world use case**

# 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**

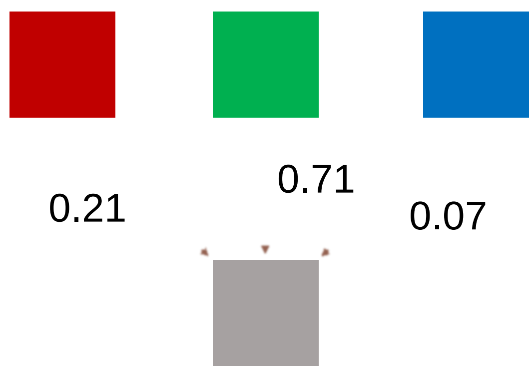


# RGB TO GRAYSCALE CONVERSION

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

# COLOR CALCULATING FORMULA

- **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**



# 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;
 }
}
```

电子科技大学
University of Electronic Science and Technology of China

```
#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;
 }
}
```

```
                    // 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;
 }
}
```

Multidimensional Kernel Configuration

Color-to-Greyscale Image Processing Example

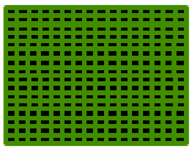
Blur Image Processing Example

# OBJECTIVE

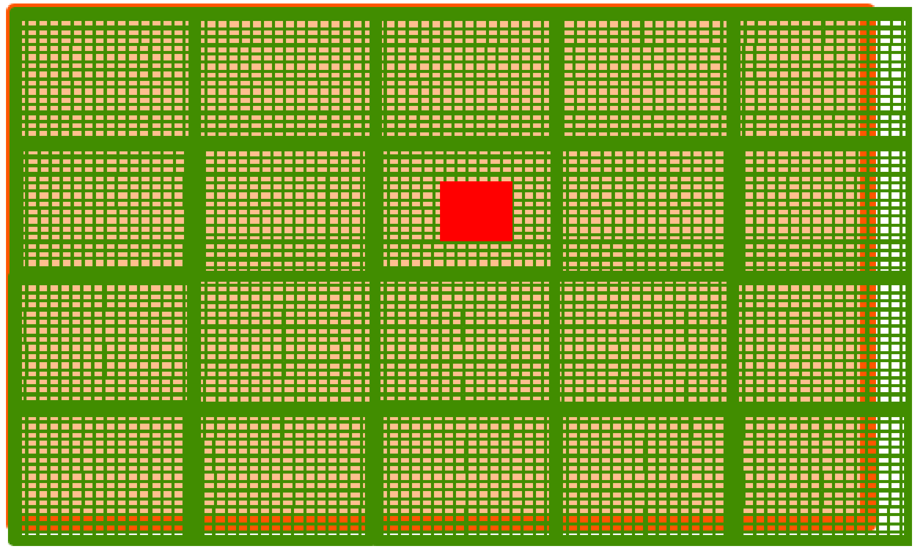
- **To learn a 2D kernel with more complex computation and memory access patterns**

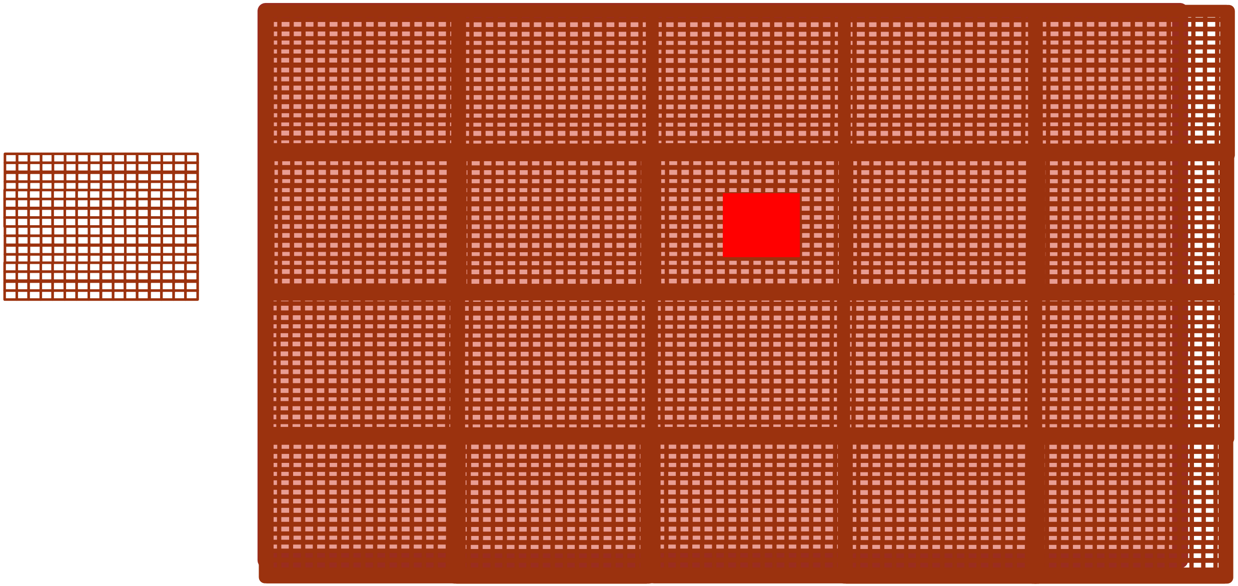
# IMAGE BLURRING

# 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);
    }
}
```

Multidimensional Kernel Configuration

Color-to-Greyscale Image Processing Example

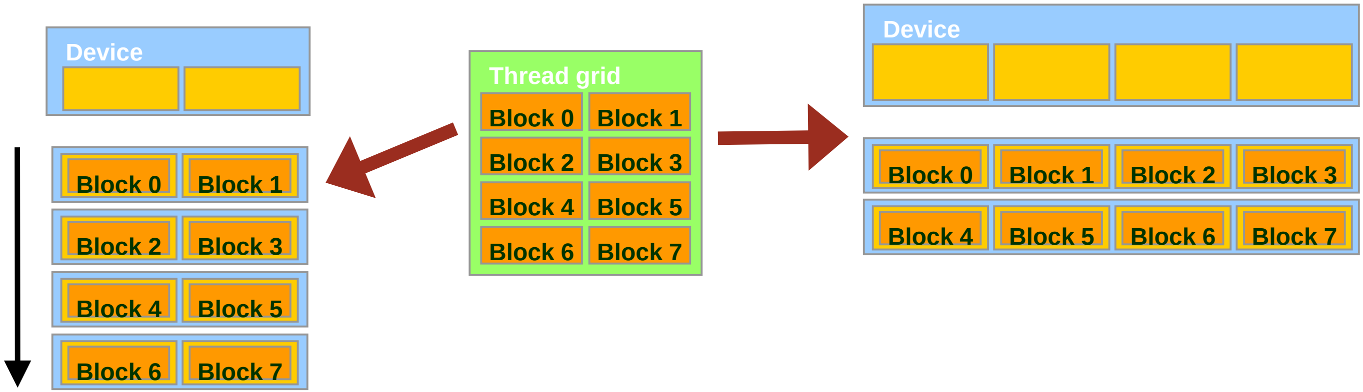
Blur Image Processing Example

Thread Scheduling

# OBJECTIVE

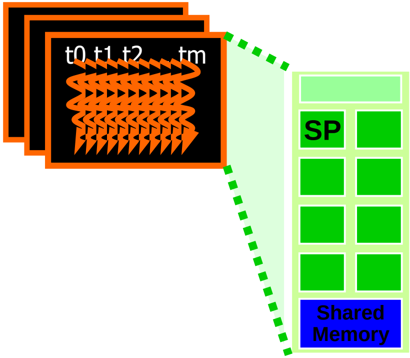
- **To learn how a CUDA kernel utilizes hardware execution resources**
  - Assigning thread blocks to execution resources
  - Capacity constrains of execution resources
  - Zero-overhead thread scheduling

# TRANSPARENT



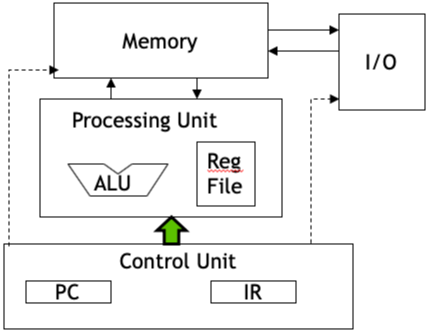

- **Each block can execute in any order relative to others.**
- **Hardware is free to assign blocks to any processor at any time**
  - A kernel scales to any number of parallel processors

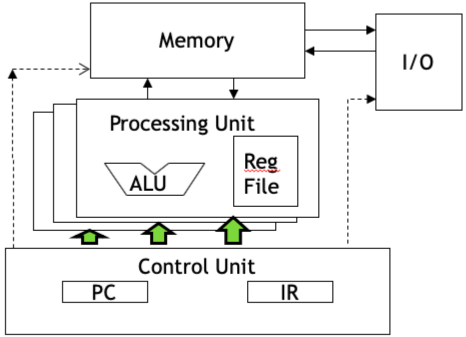
# EXAMPLE: EXECUTING THREAD BLOCKS




- **Threads are assigned to Streaming Multiprocessors (SM) in block granularity**
  - Up to **8** blocks to each SM as resource allows
  - Fermi SM can take up to **1536** threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
- **SM maintains thread/block idx #s**
- **SM manages/schedules thread execution**

# THE VON-NEUMANN MODEL WITH SIMD UNITS

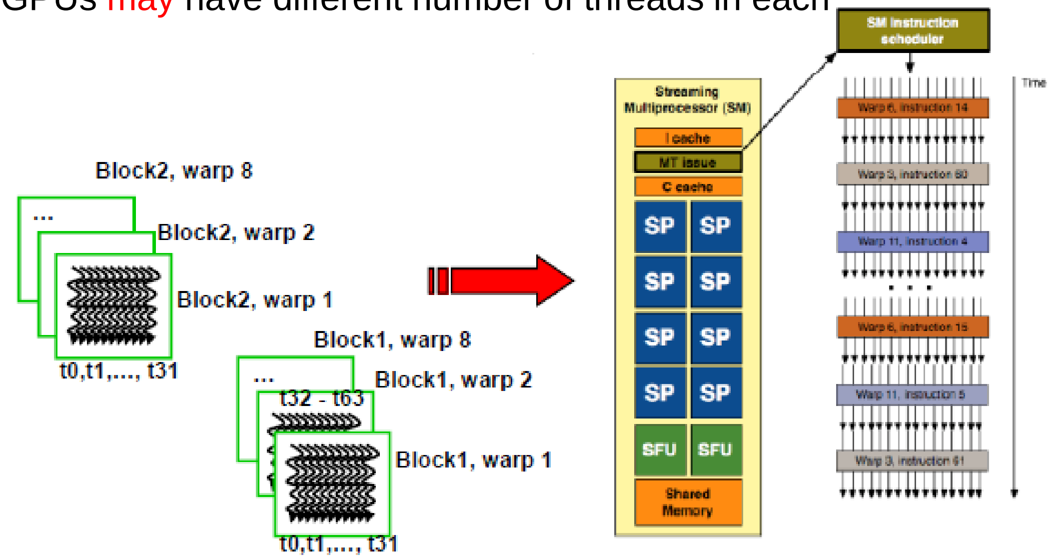



Single Instruction Multiple Data
(SIMD)

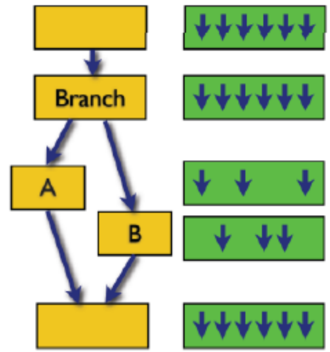
# WARPS AS SCHEDULING UNITS

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are basic scheduling units in SM
  - Future GPUs may have different number of threads in each warp



# THREAD SCHEDULING

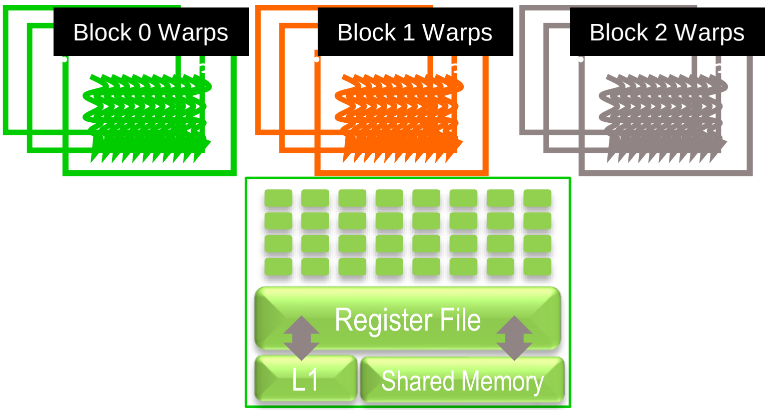
- Threads in a warp execute in SIMD
  - All threads in a warp execute the same instruction when selected
  - N way path→1/N throughput(应尽量避免在同一warp内出现分支)
- **SM implements zero-overhead warp scheduling**
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution based on a prioritized scheduling policy




电子科技大学
University of Electronic Science and Technology of China

# WARP EXAMPLE

- **If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?**
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps



# QUICK TEST
# BLOCK GRANULARITY CONSIDERATIONS

- **For Matrix Multiplication using multiple blocks, which block configurations is better？ Please explain the reason. （ For Fermi, each SM can take up to 1536 threads ）**
  - A 8X8,
  - B 16X16
  - C 32X32