

# LECTURE 2 — CUDA PARALLELISM MODEL

Multidimensional Kernel Configuration



# Multidimensional Kernel Configuration

## Color-to-Greyscale Image Processing Example

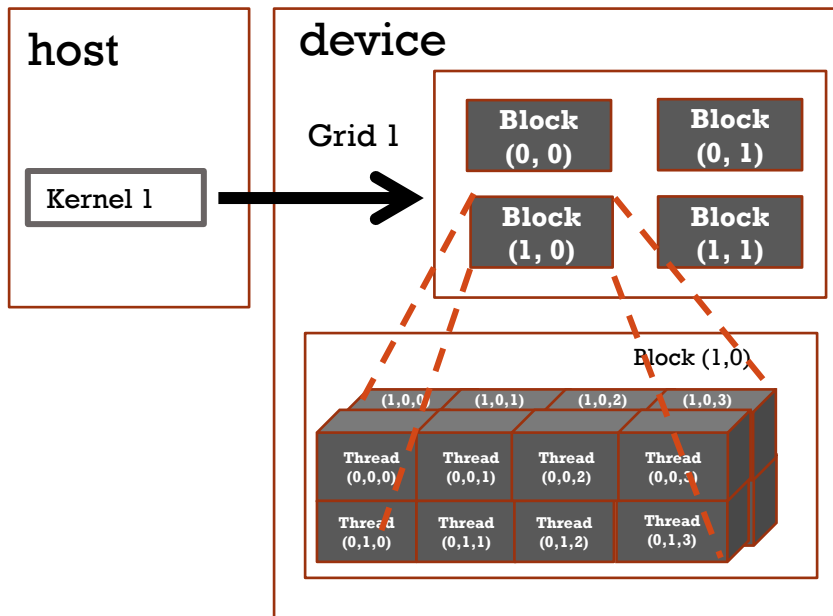
## Blur Image Processing Example

# OBJECTIVE

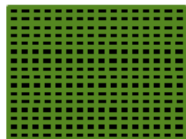
- **To understand multidimensional Grids**
  - Multi-dimensional block and thread indices
  - Mapping block/thread indices to data indices



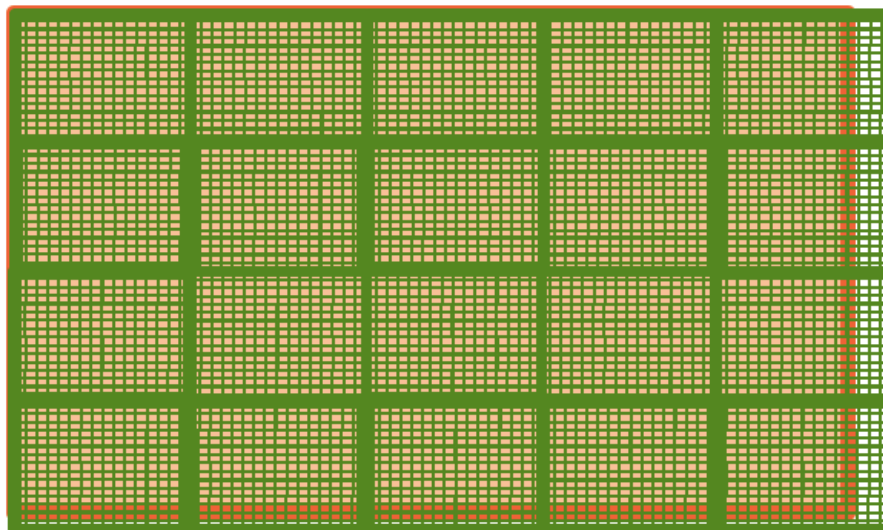
# A MULTI-DIMENSIONAL GRID EXAMPLE



# PROCESSING A PICTURE WITH A 2D GRID



16×16 blocks



62×76 picture



# ROW-MAJOR LAYOUT IN C/C++

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>



# 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];  
    }  
}
```



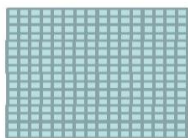
# HOST CODE FOR LAUNCHING PICTUREKERNEL

```
// assume that the picture is  $m \times n$ ,  
// 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);
```

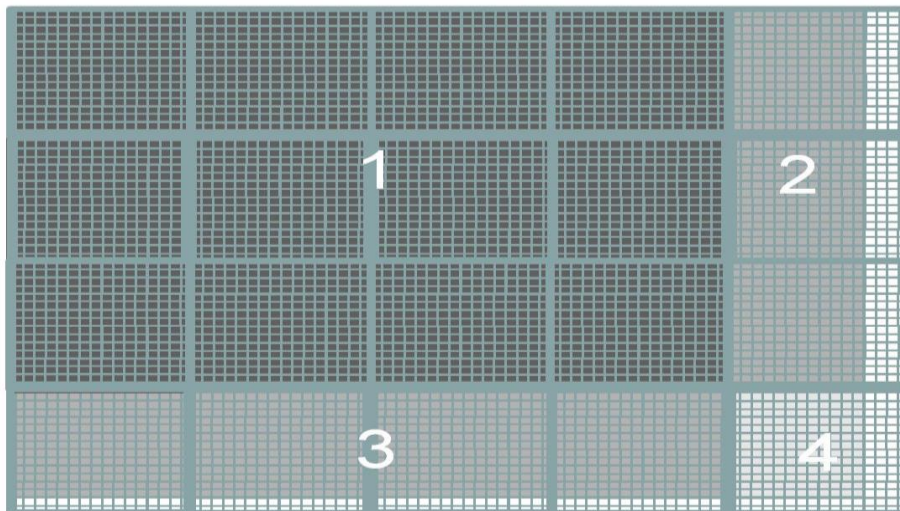




# COVERING A $62 \times 76$ PICTURE WITH $16 \times 16$ BLOCKS



$16 \times 16$  block



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

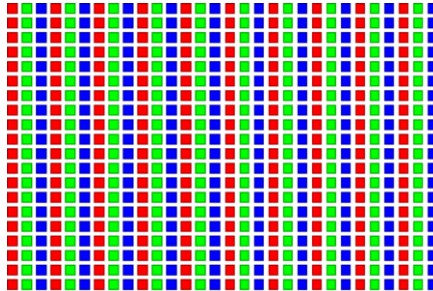
# OBJECTIVE

- **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:  
$$\text{grayPixel}[I,J] = 0.21*r + 0.71*g + 0.07*b$$
- This is just a dot product  $\langle [r,g,b], [0.21,0.71,0.07] \rangle$  with the constants being specific to input RGB space



0.21

0.71

0.07



# 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;  
    }
```



```

#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

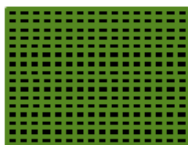
- **To learn a 2D kernel with more complex computation and memory access patterns**



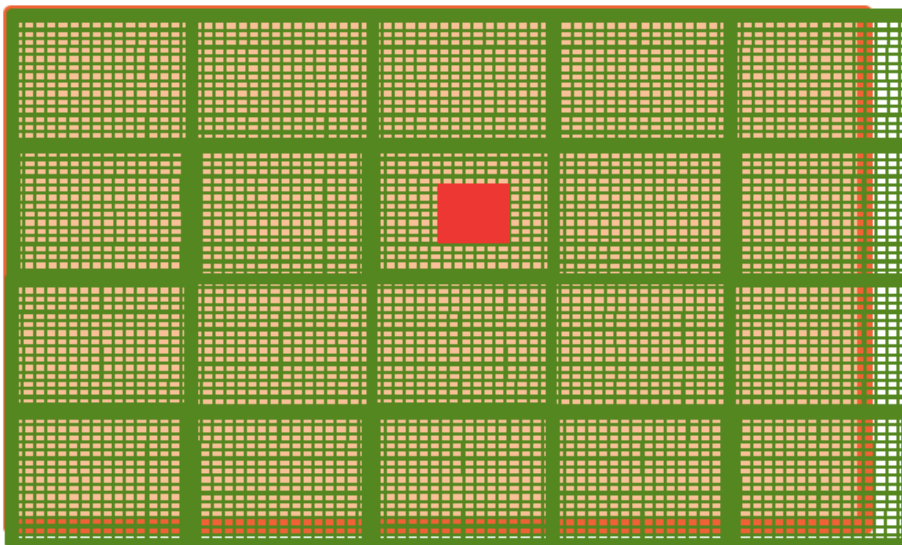
# IMAGE BLURRING



# Blurring Box

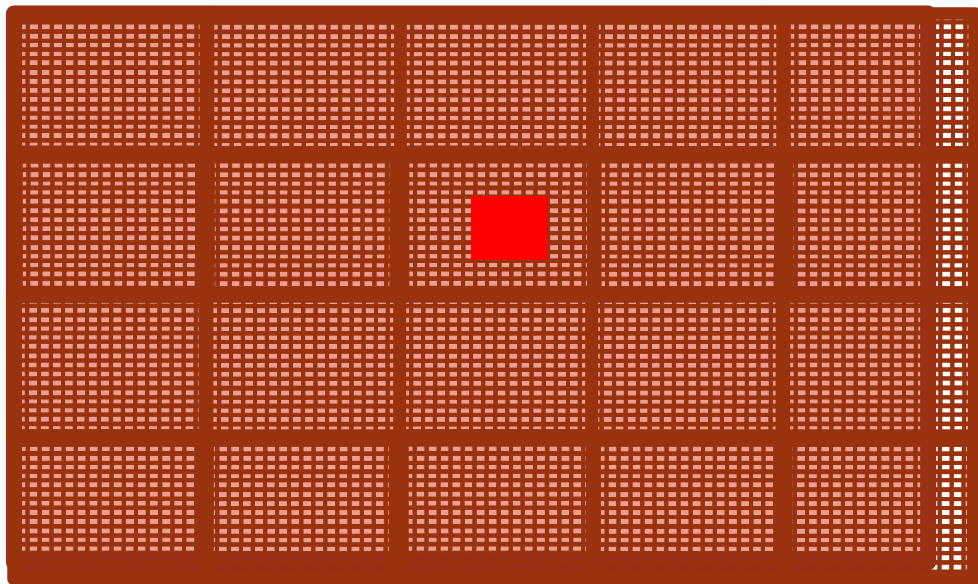
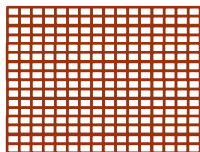


Pixels  
processed  
by a  
thread  
block



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





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



# 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
    }
}
```



电子科技大学

University of Electronic Science and Technology of China



\_\_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

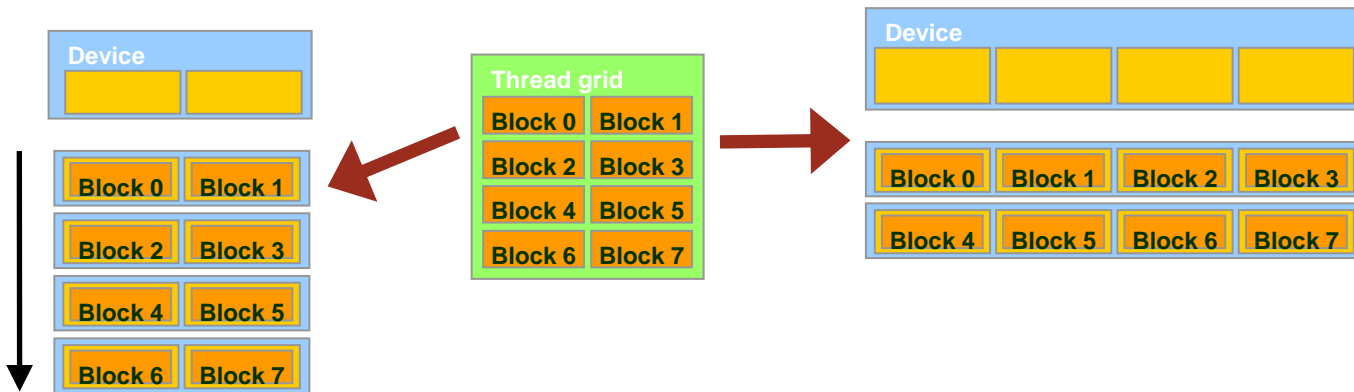


# OBJECTIVE

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



# TRANSPARENT

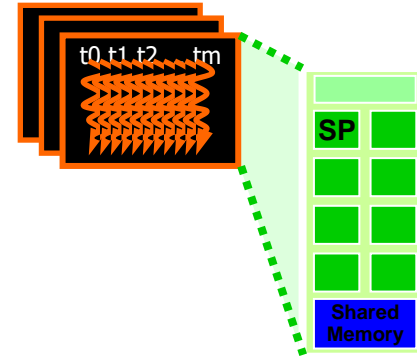


- **Each block can execute in any order relative to others.**
- **Hardware is free to assign blocks to any processor at any time**
  - A kernel scales to any number of parallel processors

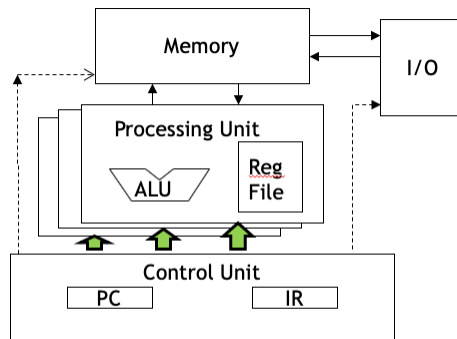
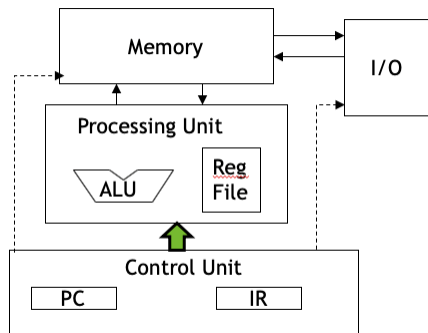


# EXAMPLE: EXECUTING THREAD BLOCKS

- **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  $256 \text{ (threads/block)} * 6 \text{ blocks}$
    - Or  $512 \text{ (threads/block)} * 3 \text{ blocks}$ , etc.
- **SM maintains thread/block idx #s**
- **SM manages/schedules thread execution**



# THE VON-NEUMANN MODEL WITH SIMD UNITS

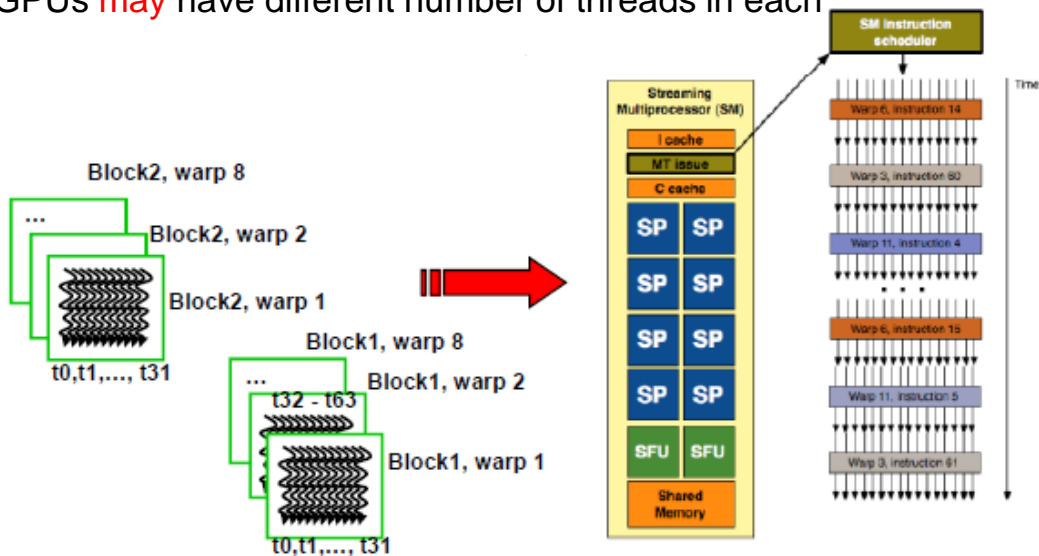


Single Instruction Multiple Data (SIMD)



# 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



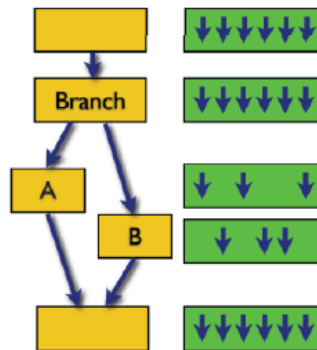
电子科技大学

University of Electronic Science and Technology of China



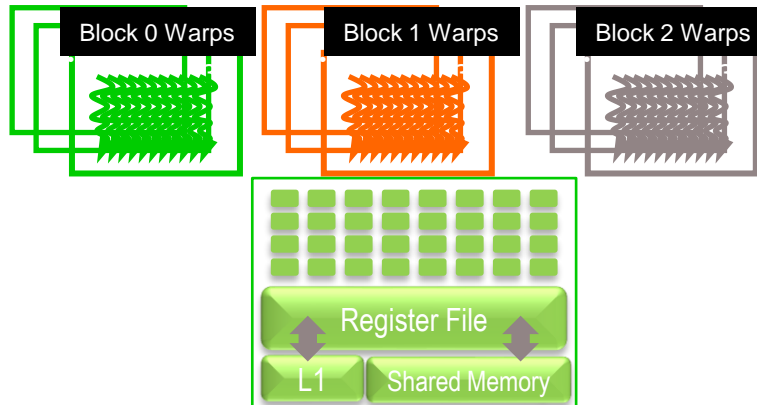
# THREAD SCHEDULING

- Threads in a warp execute in SIMD
  - All threads in a warp execute the same instruction when selected
  - N way path  $\rightarrow$   $1/N$  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



# 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





# BLOCK GRANULARITY CONSIDERATIONS

- **For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks for Fermi?**
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, which translates to 24 Blocks. However, each SM can only take up to 8 Blocks, **only 512 threads will go into each SM!**
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM.

