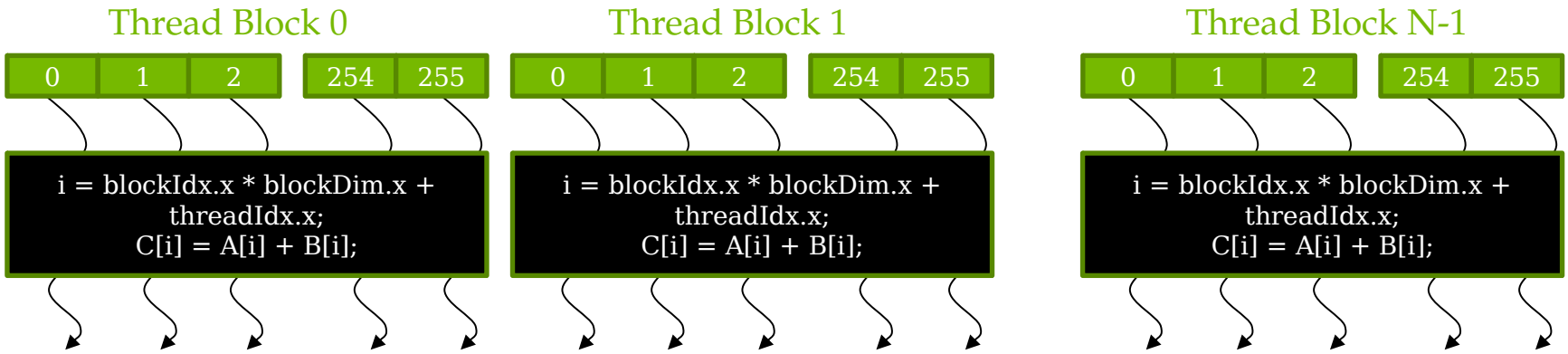
# CENG443
# Heterogeneous Parallel Programming

## CUDA Threads

# CUDA Thread Organization




## Divide thread array into multiple blocks

- Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
- Threads in different blocks do not interact

# Vector Addition with 8000 Elements

**Each thread calculates one output element**

**Thread block size is 1024 threads**

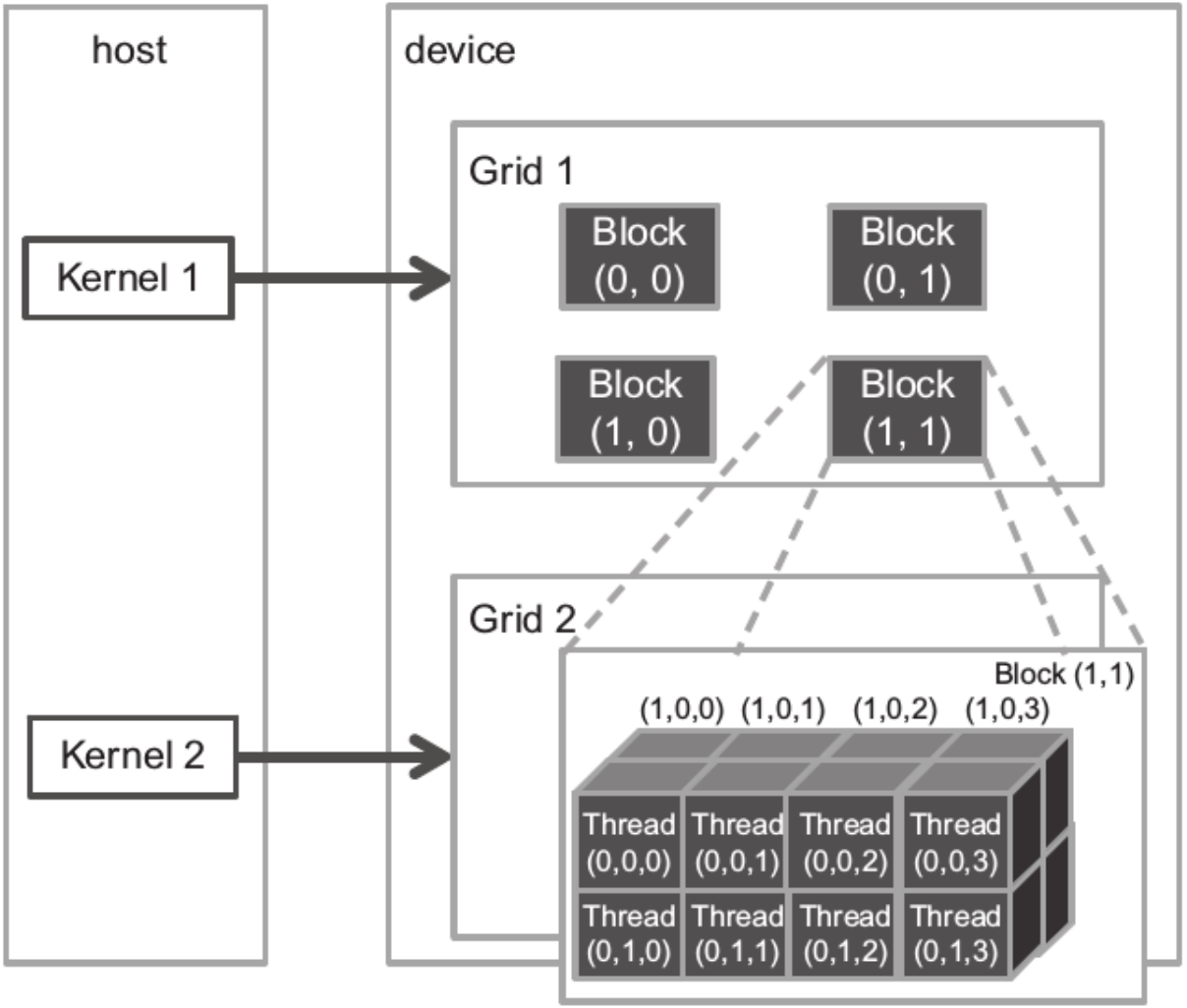
**The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements**

**How many threads will be in the grid?**

ceil(8000/1024) * 1024 = 8 * 1024 = 8192

(the minimal multiple of 1024 to cover 8000 is 1024*8 = 8192)

# Multidimensional Grid Organization



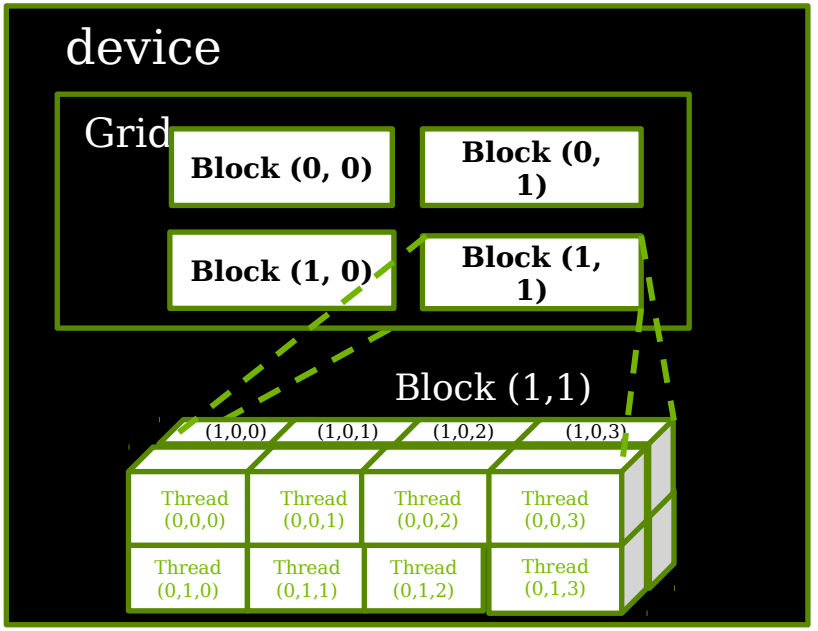
# Indexing

**Each thread uses indices to decide what data to work on**

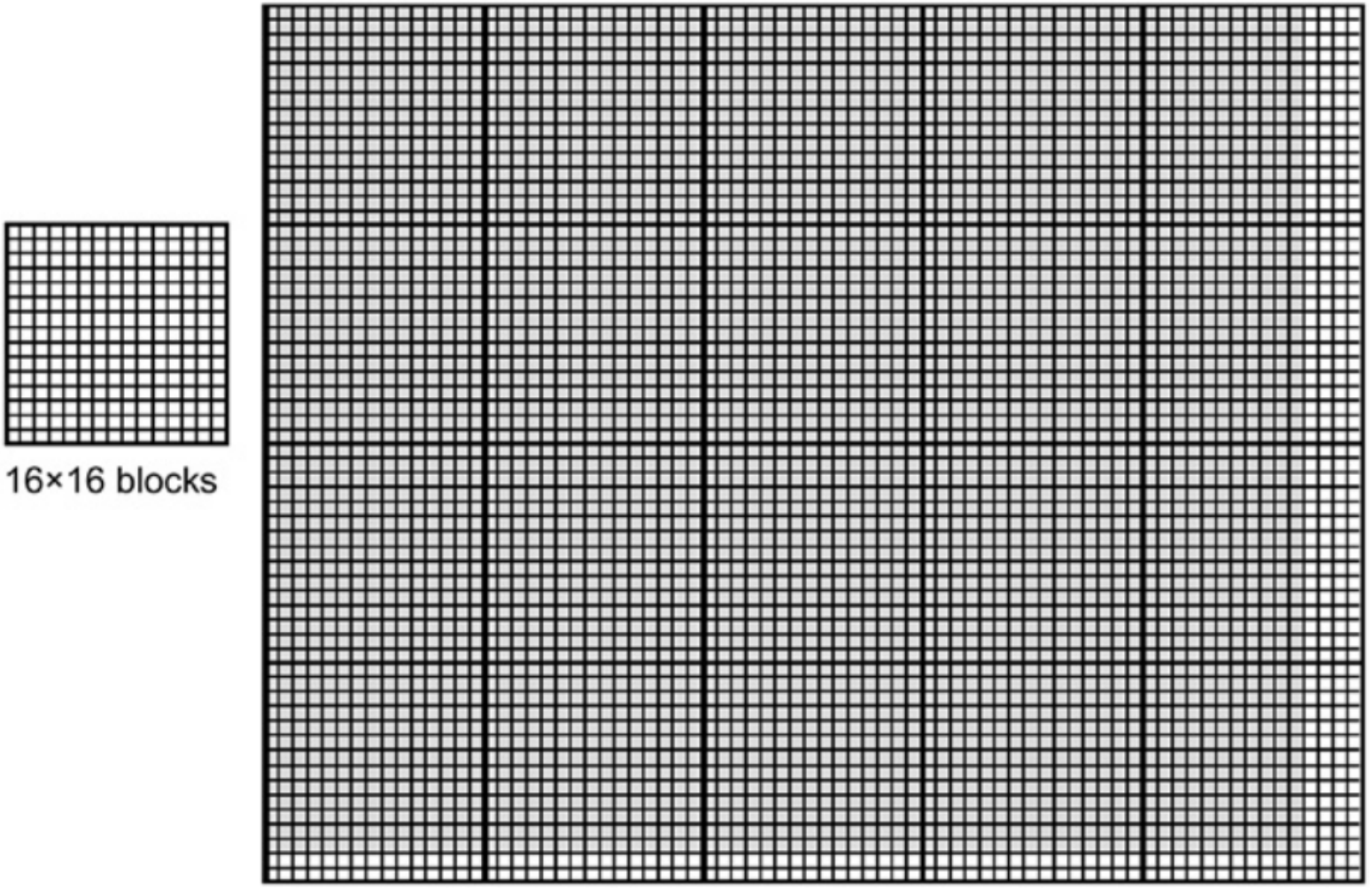
- blockIdx: 1D, 2D, or 3D (CUDA 4.0)
- threadIdx: 1D, 2D, or 3D

**Simplifies memory addressing when processing multidimensional data**

- Image processing
- Solving PDEs on volumes
- …

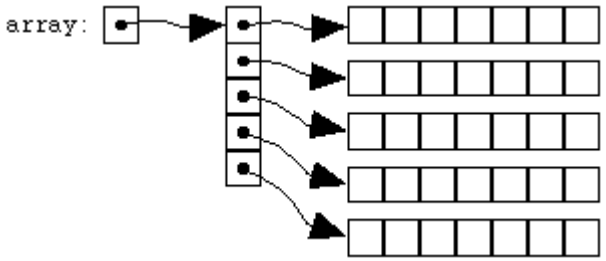


# Mapping Threads to Multidimensional Data



# Dynamically Allocated 2D Arrays in C – Pointer to a Pointer

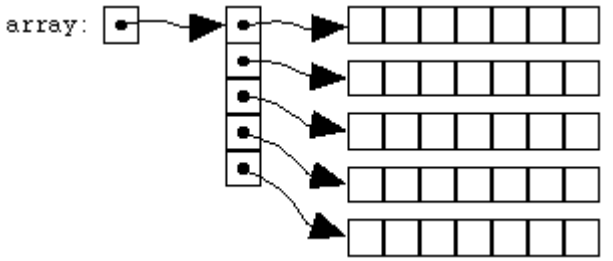
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int r = 3, c = 4, i, j, count;

   int **arr = (int **)malloc(r * sizeof(int *));
   for (i=0; i<r; i++)
       arr[i] = (int *)malloc(c * sizeof(int));

   // Note that arr[i][j] is same as *(*(arr+i)+j)
   count = 0;
   for (i = 0; i <  r; i++)
     for (j = 0; j < c; j++)
        arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count

   for (i = 0; i <  r; i++)
     for (j = 0; j < c; j++)
        printf("%d ", arr[i][j]);

  /* Code for further processing and free the
     dynamically allocated memory */

  return 0;
}
```




Not contiguous!

# Allocate Each Row Separately

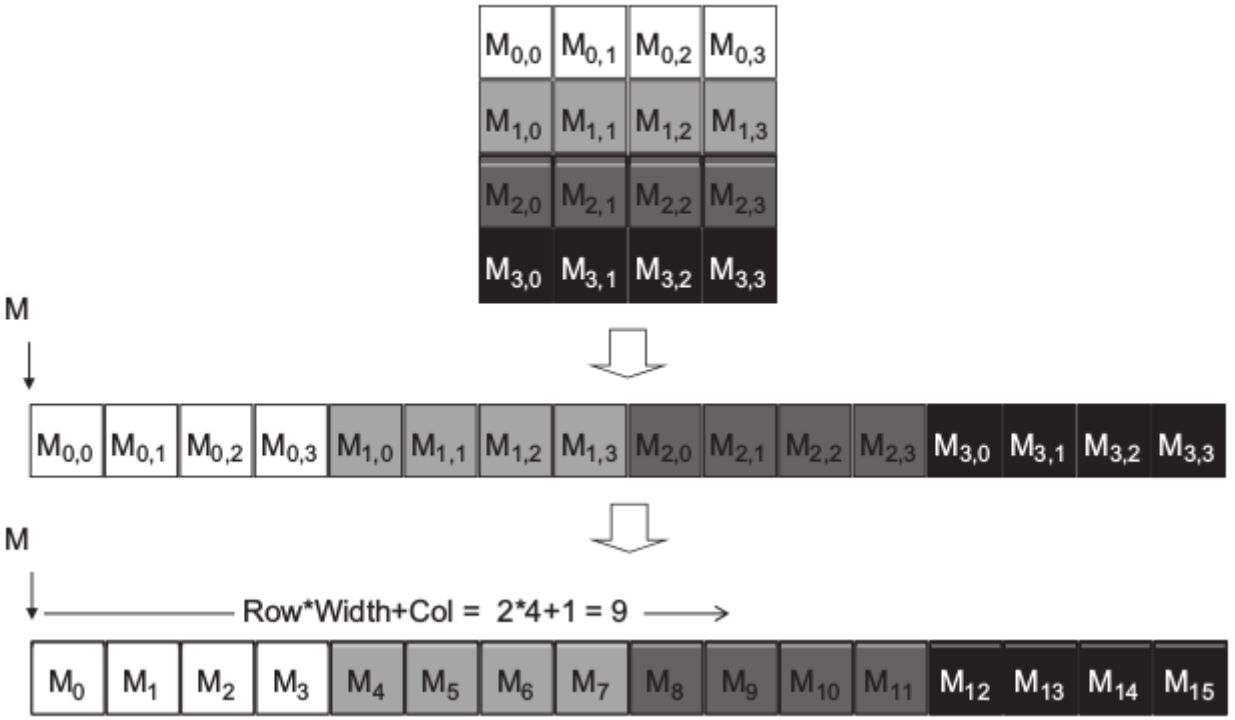
```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_B, size);
     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
     cudaMalloc((void **) &d_C, size);

     // Kernel invocation code – to be shown later

     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```



# Dynamically Allocated 2D Arrays in C – A single pointer

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int r = 3, c = 4;
   int *arr = (int *)malloc(r * c * sizeof(int));

   int i, j, count = 0;
   for (i = 0; i <  r; i++)
     for (j = 0; j < c; j++)
        *(arr + i*c + j) = ++count;

   for (i = 0; i <  r; i++)
     for (j = 0; j < c; j++)
        printf("%d ", *(arr + i*c + j));

  /* Code for further processing and free the
     dynamically allocated memory */

  return 0;
}
```

# Row-Major Layout in C



# Flatten 2D Matrix

**2D matrix to 1D array**

**n x m, which are the number of rows and columns also called the height and the width**

**a(i,j) can be flatten to 1D array b(k) where k= i*m + j**

```
for (int i=0; i < n; i++)
{
        for (int j =0; j< m; j++)
                b[i*m+j] = a[i][j];
 }
```

# 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];
  }
}
```

Scale every pixel value by 2.0

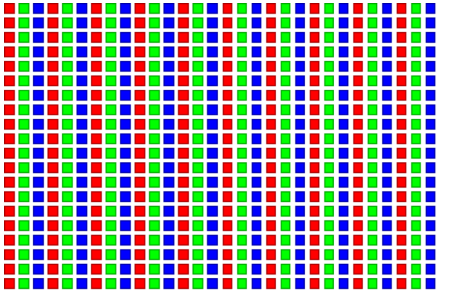
# Host Code for Launching PictureKernel

```
// assume that the picture is mn,
// 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(ceil(n/16), ceil(m/16), 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
...
```

# Alternative Way

```
cudaError_t cudaMallocPitch ( void **  devPtr,
                              size_t * pitch,
                              size_t   width,
                              size_t   height
                            )

cudaError_t cudaMemcpy2D ( void *   dst,
                           size_t   dpitch,
                           const void *  src,
                           size_t   spitch,
                           size_t   width,
                           size_t   height,
                           enum cudaMemcpyKind  kind
                         )
```

# RGB Color Image Representation

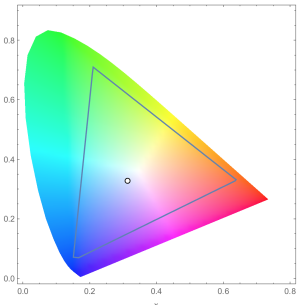


**Each pixel in an image is an RGB value**

**The format of an image's row is**

**(r g b) (r g b) … (r g b)**

**RGB ranges are not distributed uniformly**



**Many different color spaces, here we show the constants to convert to AdobeRGB color space**

The vertical axis (y value) and horizontal axis (x value) show the fraction of the pixel intensity that should be allocated to G and B. The remaining fraction (1-y–x) of the pixel intensity that should be assigned to R

The triangle contains all the representable colors in this color space

# RGB to Grayscale Conversion

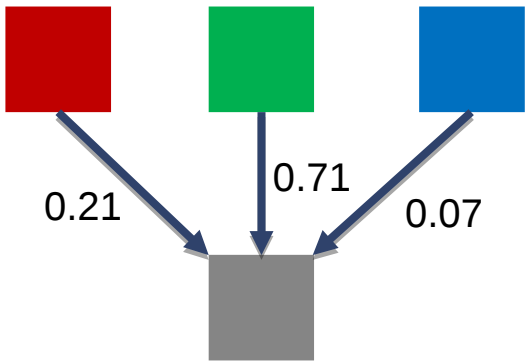
A grayscale digital image is an image in which the value of each pixel carries only intensity information.

# Color Calculation

**For each pixel (r g b) at (i,j)**

**gray[i,j] = 0,21 * r + 0,71 * g + 0,07 * b**

**Dot product with the constants being specific to input RGB space**



# RGB to Grayscale Conversion Kernel

```
#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;
 }
}
```

# A More Complex Kernel

**In vecAdd and colorConvert kernels, each thread performs only a small number of arithmetic operations on one array element**

**In most CUDA C programs, threads often perform complex algorithms on their data and need to cooperate with one another**

# Image Blurring

# Image Blurring

**Image processing: to reduce the impact of noise and granular rendering effects in an image by correcting problematic pixel values with the clean surrounding pixel values**

**Computer vision: to allow edge detection and object recognition algorithms to focus on thematic objects rather than being impeded by a massive quantity of fine-grained objects**

**Displays: to highlight a particular part of the image by blurring the rest of the image**

# Image Blurring

**Mathematically, an image blurring function calculates the value of an output image pixel as a weighted sum of a patch of pixels encompassing the pixel in the input image**

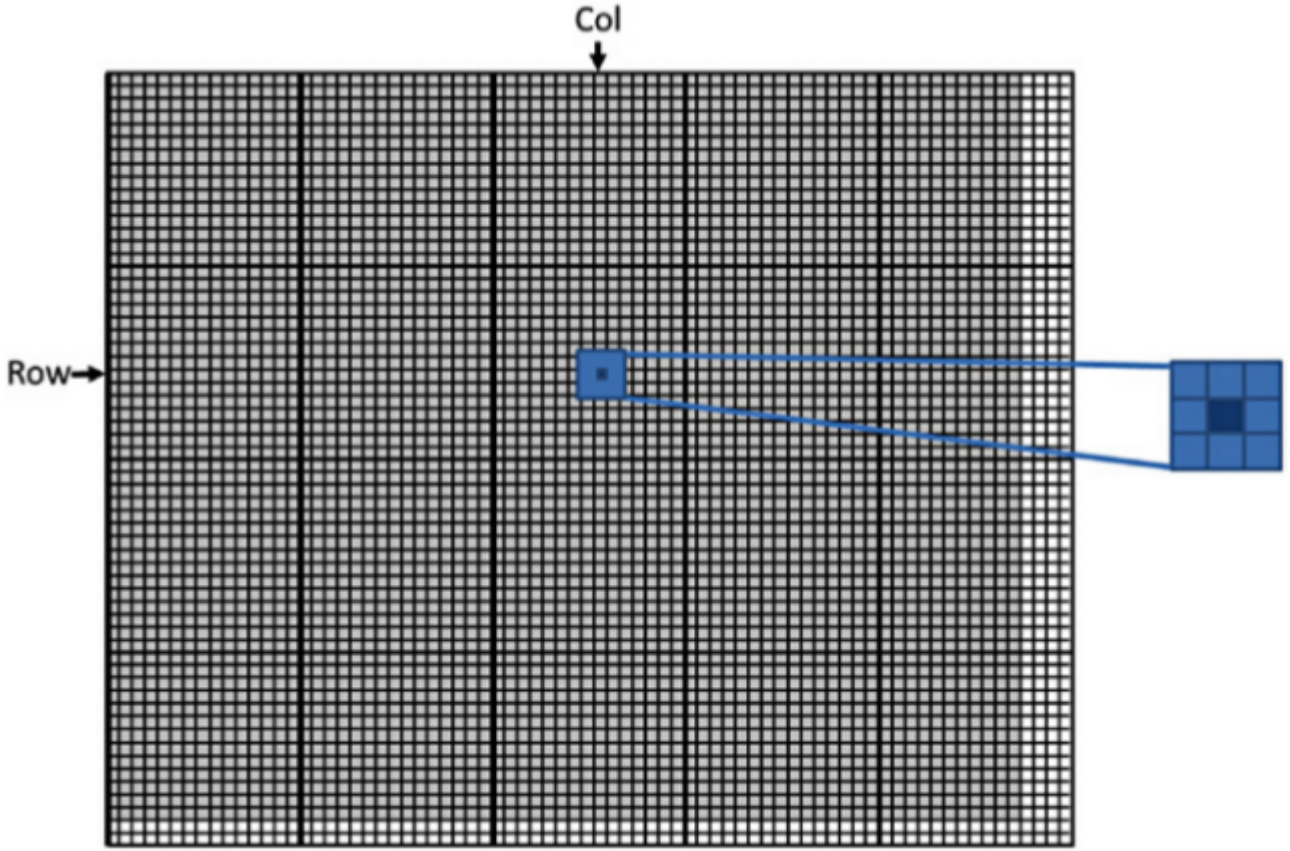
**The computation of such weighted sums belongs to the *Convolution* pattern**

**Taking a simple average value of the N×N patch of pixels surrounding, and including, our target pixel**

# Blurring Box: 3x3 patch

**Three rows ( Row-1, Row, Row+1 ) and three columns ( Col-1, Col, Col+1 )**

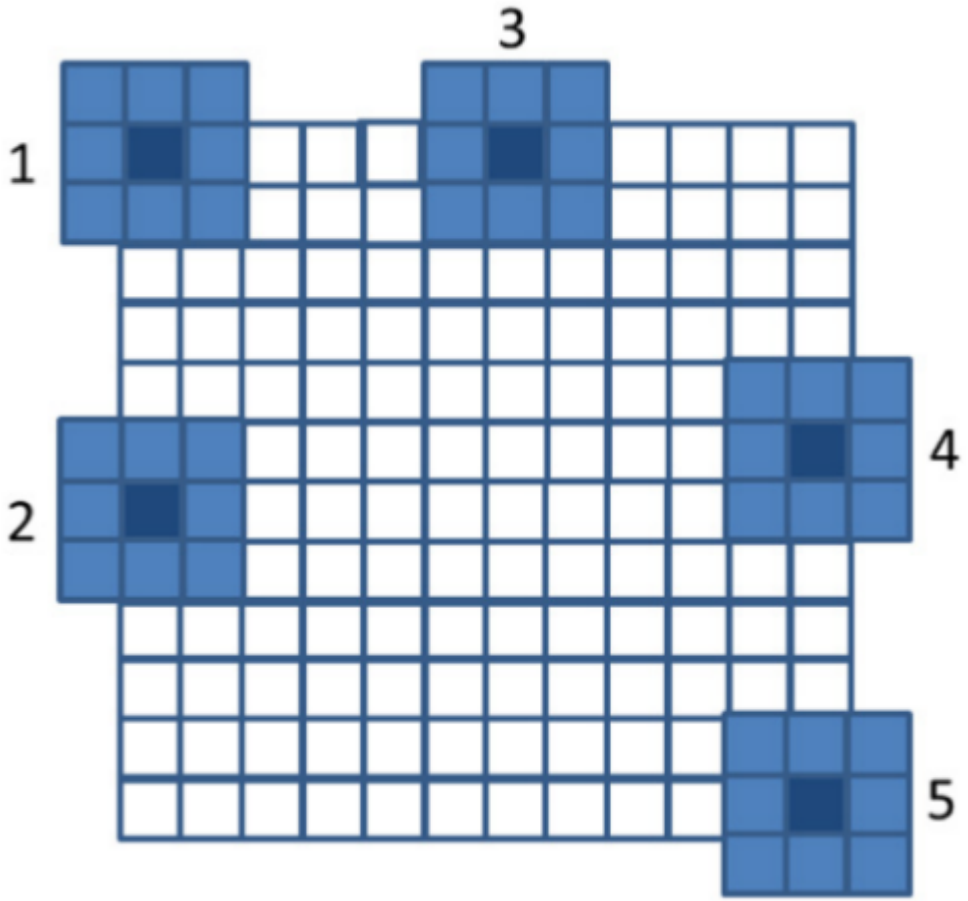
**for calculating the output pixel at (25, 50) : (24, 49), (24, 50), (24, 51), (25, 49), (25, 50), (25, 51), (26, 49), (26, 50), and (26, 51)**



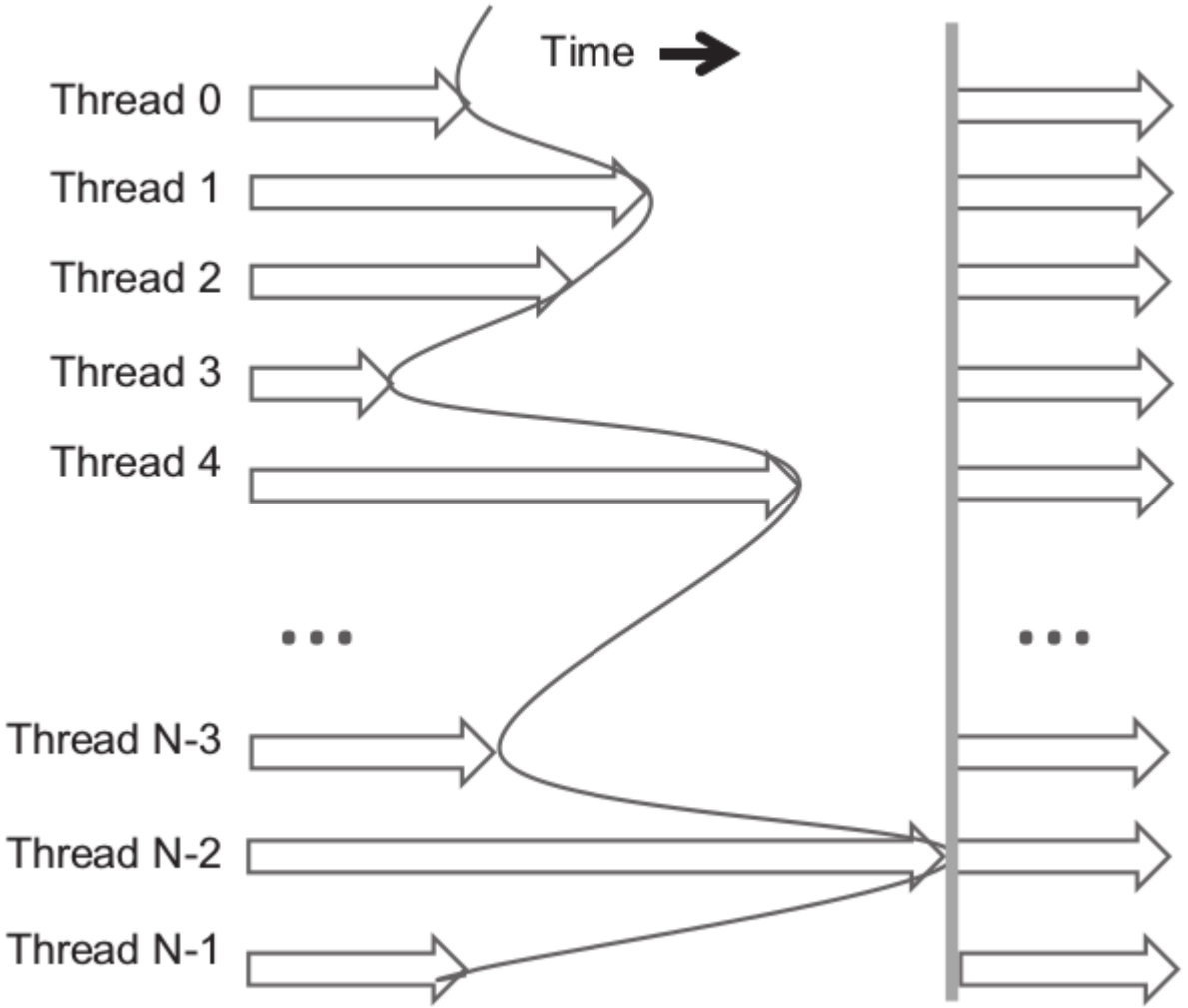
# Image Blurring Kernel

```
__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);
    }
 }
```

# Boundary Conditions

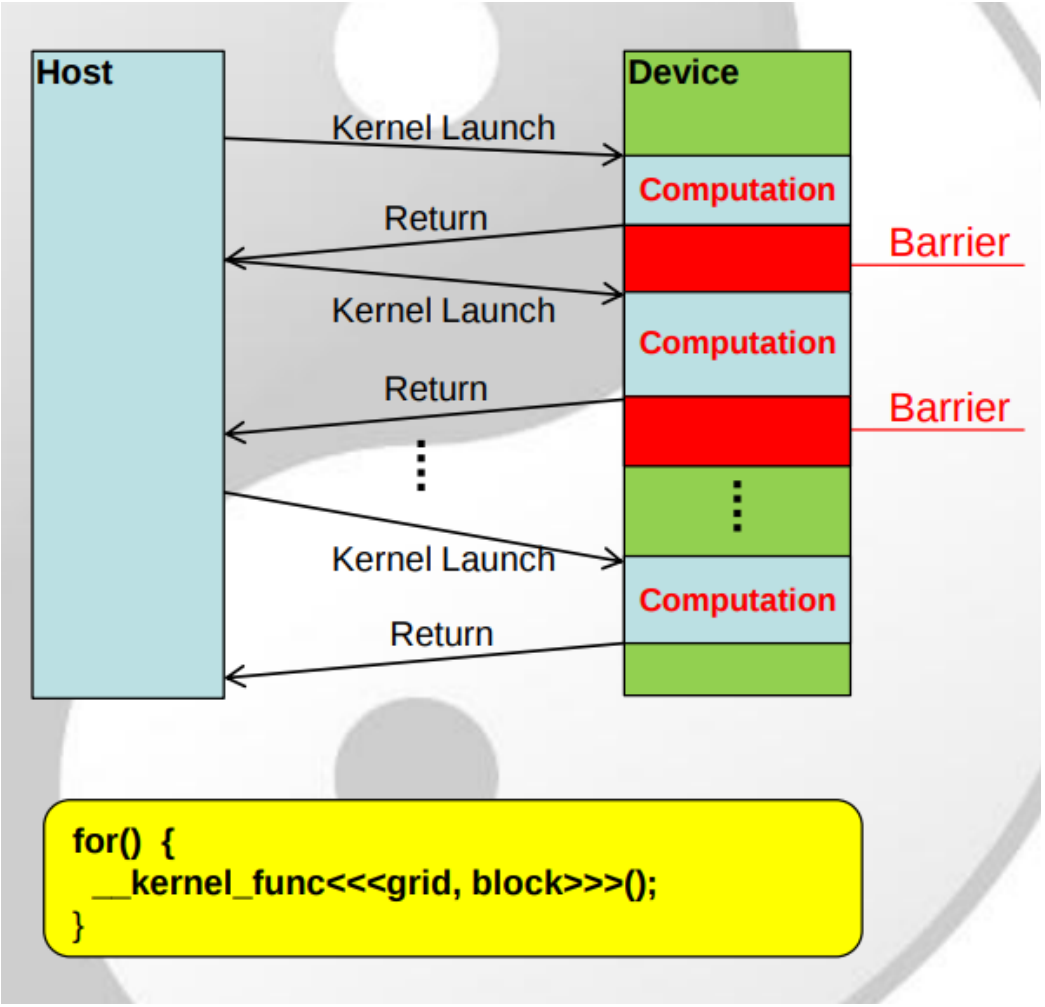


# Barrier Synchronization



# CUDA Synchronization

- **Inter-block sync is implemented via kernel launches**



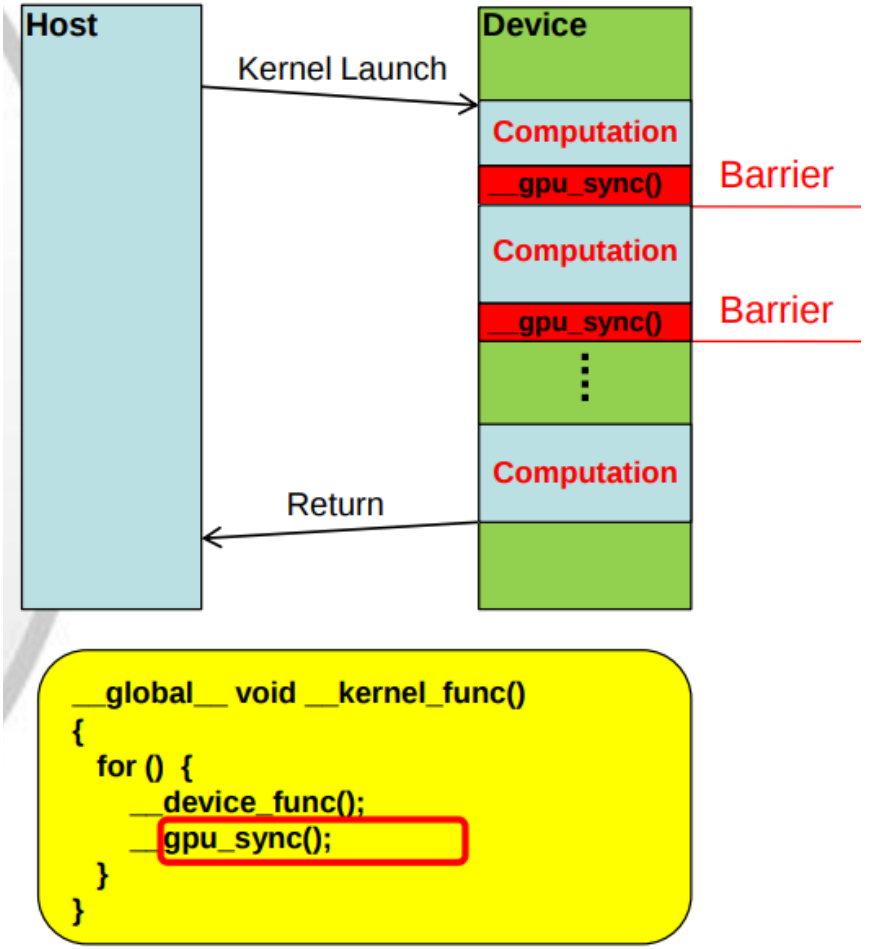
# CUDA Synchronization

- **Intra-block sync is implemented with __syncthreads()**

```
void __syncthreads();
```

**Synchronizes all threads within a block**

**All threads must reach the barrier (if!)**



# Synchronization

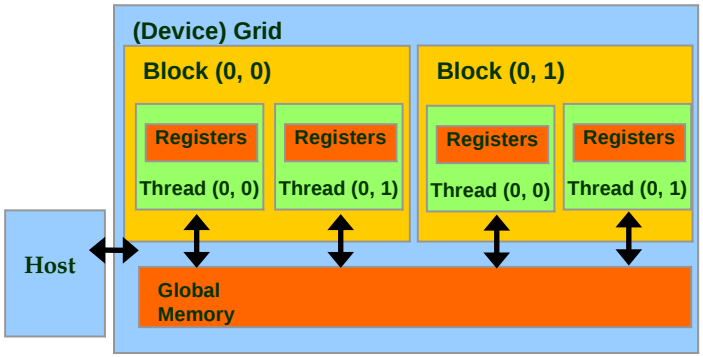
**The ability to synchronize also imposes execution constraints on threads within a block**

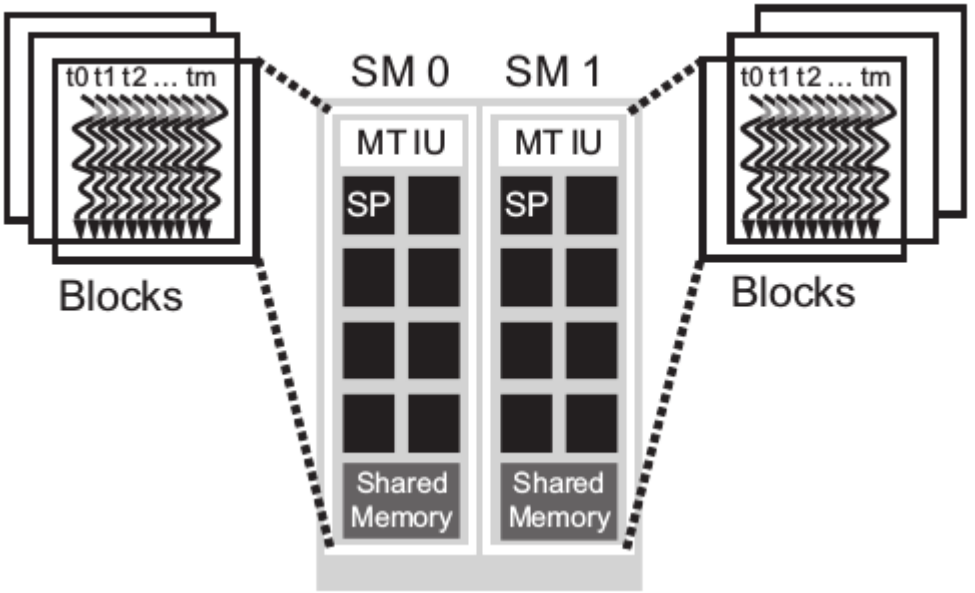
**These threads should execute in close temporal proximity with each other to avoid excessively long waiting times**

**One needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier**

**CUDA runtime systems satisfy this constraint by assigning execution resources to all threads in a block as a unit**

# Executing Thread Blocks



# Executing Thread Blocks

**Threads are assigned to Streaming Multiprocessors (SM) in block granularity**

**Fermi SM can take up to 8 blocks and 1536 threads**

- Could be 256 (threads/block) * 6 blocks
- Or 512 (threads/block) * 3 blocks, etc.

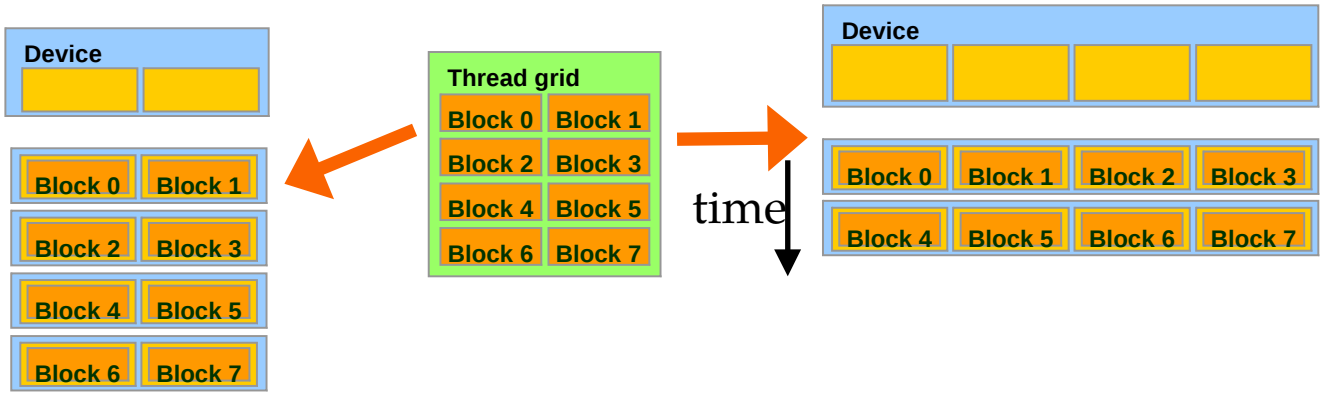
**Threads run concurrently**

- SM assigns/maintains thread id #s
- SM manages/schedules thread execution

# Transparent Scalability

**The ability to execute the same application code on hardware with different numbers of execution resources**

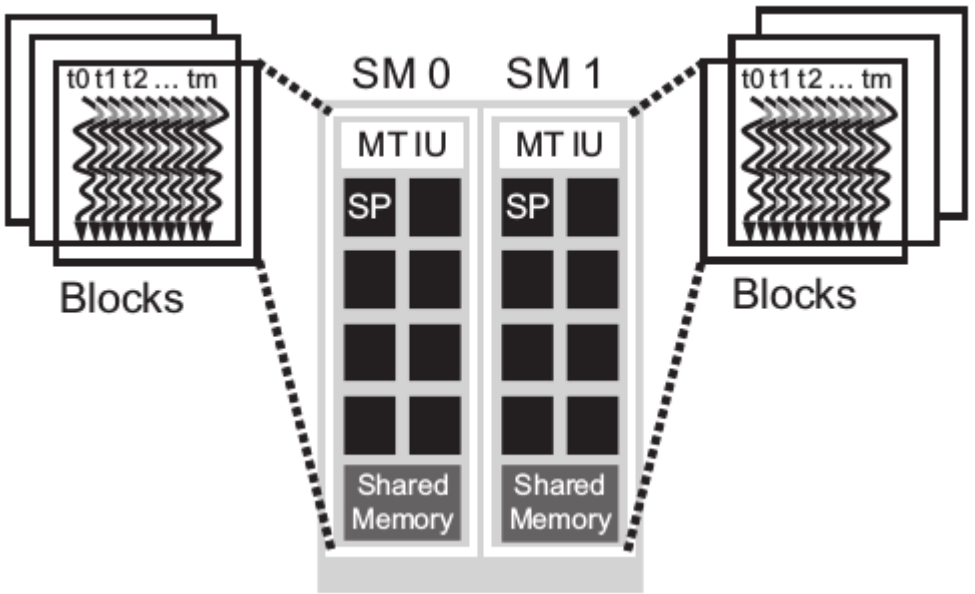
**Each block can execute in any order relative to others**



# Resource Assignment

**Multiple thread blocks can be assigned to each SM**

**Each device sets a limit on the number of blocks that can be assigned to each SM**



# Thread Scheduling

- Each Block is executed as 32-thread Warps (In the majority of implementations to date)
- The size of warps is implementation-specific (hardware implementation), not part of CUDA specification (`warpSize` field of the device query variable)
- Warps are scheduling units in SM
- Future GPUs may have different number of threads in each warp

# Warp Example

**3 blocks are assigned to an SM**

**Each block has**

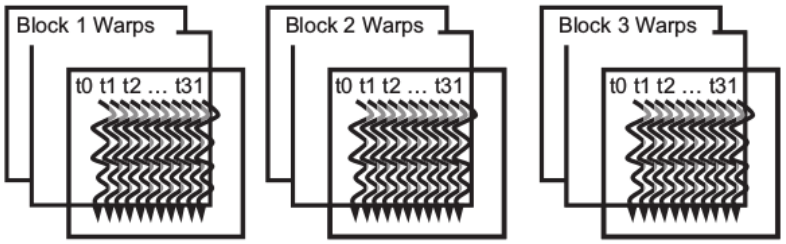
256 threads

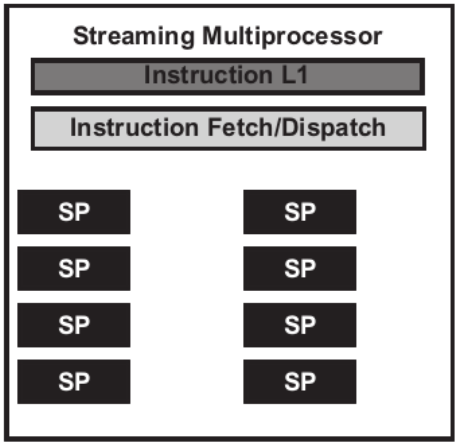
256/32=8 warps

**3 x 8 = 24 warps in each SM**

**SIMT model**

at any instant in time, one instruction is fetched and executed for all threads in the warp



# SPs vs Threads

**In general, there are fewer SPs than the threads assigned to each SM**

**Each SM has only enough hardware to execute instructions from a small subset of all threads assigned to the SM at any point in time**

**In early GPU designs, each SM can execute only one instruction for a single warp at any given instant**

**In recent designs, each SM can execute instructions for a small number of warps at any point in time**

# Why We Need Warps?

**Why we need to have so many warps in an SM if it can only execute a small subset of them at any instant?**

**This is how CUDA processors efficiently execute long-latency operations, such as global memory accesses: Latency tolerance!**

**When an instruction to be executed by a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution.**

**Instead, another resident warp that is no longer waiting for results will be selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution.**

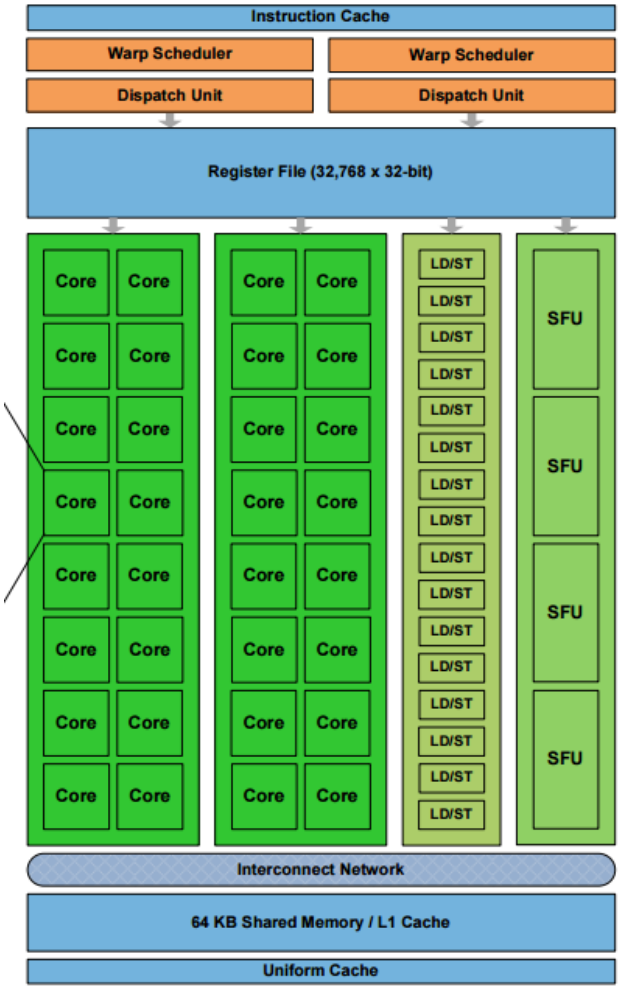
# Warp Scheduling

## 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
- All threads in a warp execute the same instruction when selected

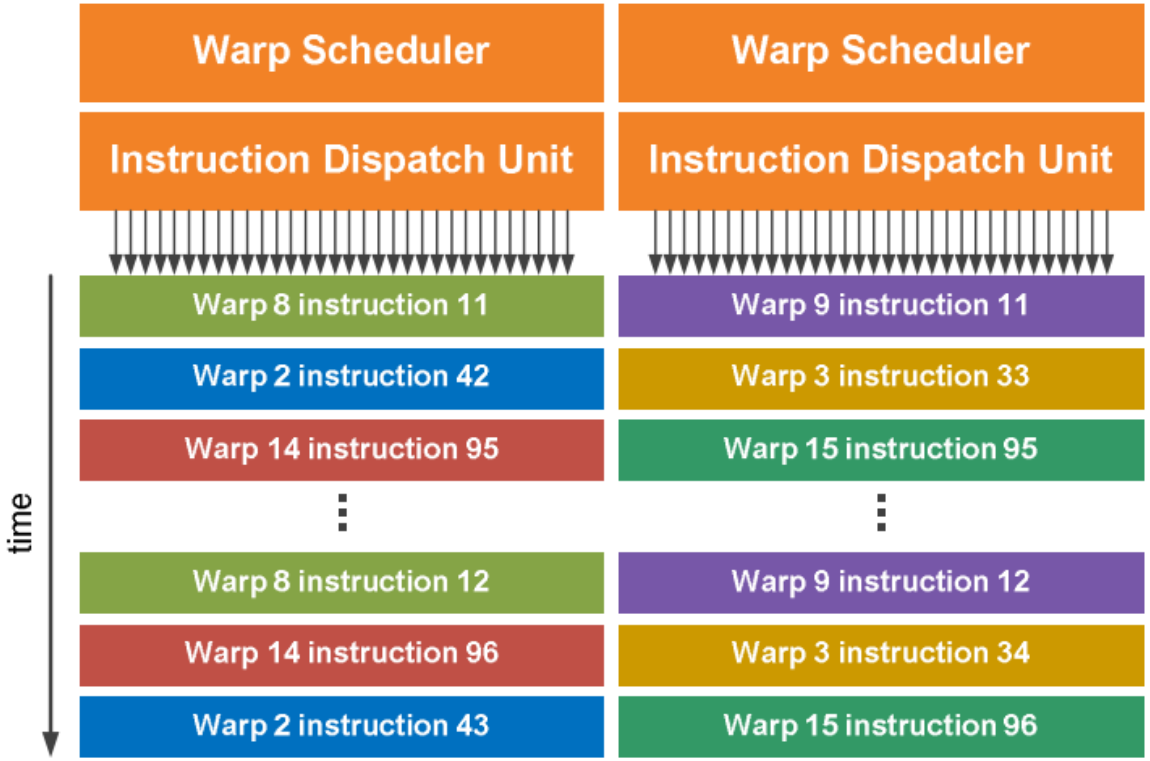
## The long waiting time of warp instructions is “hidden” by executing instructions from other warps

# Fermi Architecture-SM (16 total)



# Fermi Architecture-Warps

**Dual warp scheduler selects two warps, and issues one instruction from each warp to a group of sixteen cores, sixteen load/store units, or four SFUs**



# Block Granularity Considerations

**CUDA device: up to 8 blocks and 1024 threads per SM, up to 512 threads in each block**

**For image blur, should we use 8 × 8, 16 × 16, or 32 × 32 thread blocks?**

**8 × 8 blocks**

Each block = 64 threads

1024/64 = 12 blocks per SM > 8

Only 64 × 8 = 512 threads in each SM

**16 × 16 blocks**

Each block = 256 threads

1024/256 = 4 blocks per SM < 8

**32 × 32 blocks**

Each block = 1024 threads > 512