GPU Teaching Kit

Accelerated Computing

Module 3.1 - CUDA Parallelism Model

Kernel-Based SPMD Parallel Programming

Objective

- To learn the basic concepts involved in a simple CUDA kernel function
  - Declaration
  - Built-in variables
  - Thread index to data index mapping

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Example: Vector Addition Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and copies omitted
 // Run ceil(n/256.0) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

The ceiling function makes sure that there are enough threads to cover all elements.

More on Kernel Launch (Host Code)

Host Code

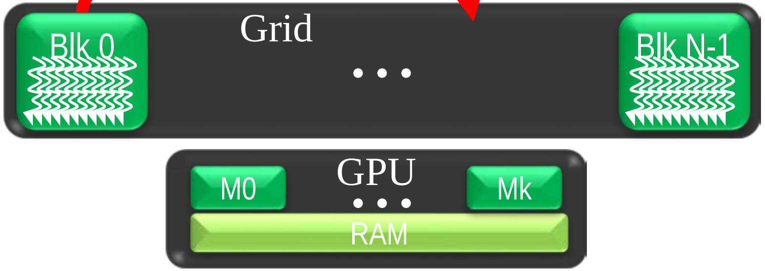
```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

This is an equivalent way to express the ceiling function.

Kernel execution in a nutshell

```
__host__
void vecAdd(…)
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);
vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B
,d_C,n);
}
```

```
__global__
void vecAddKernel(float *A,
     float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x
         + threadIdx.x;

   if( i<n ) C[i] = A[i]+B[i];
}
```

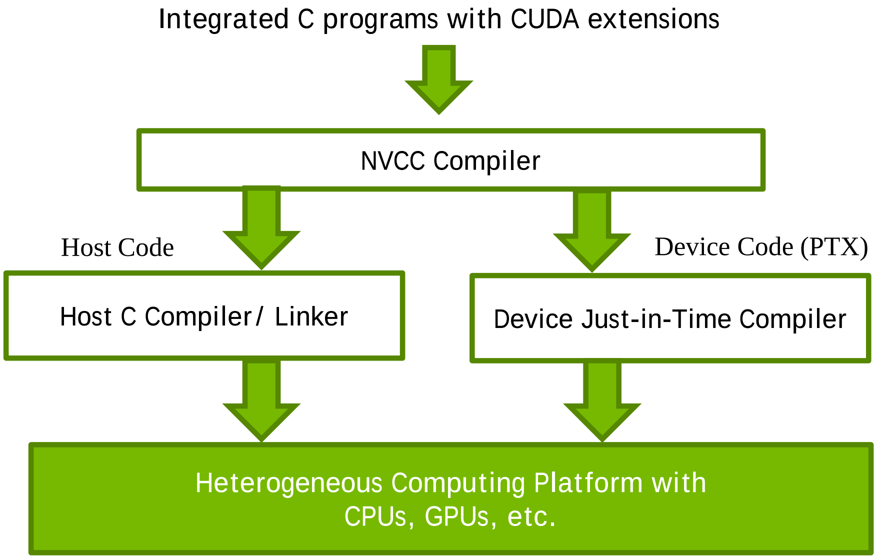


More on CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void  KernelFunc()`	device	host
`__host__   float HostFunc()`	host	host

- **`__global__`** defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return **`void`**
- **`__device__`** and **`__host__`** can be used together
- **`__host__`** is optional if used alone

Compiling A CUDA Program


Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C Compiler/ Linker
Device Just-in-Time Compiler
Heterogeneous Computing Platform with CPUs, GPUs, etc.

GPU Teaching Kit

Accelerated Computing

The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.

GPU Teaching Kit

Accelerated Computing

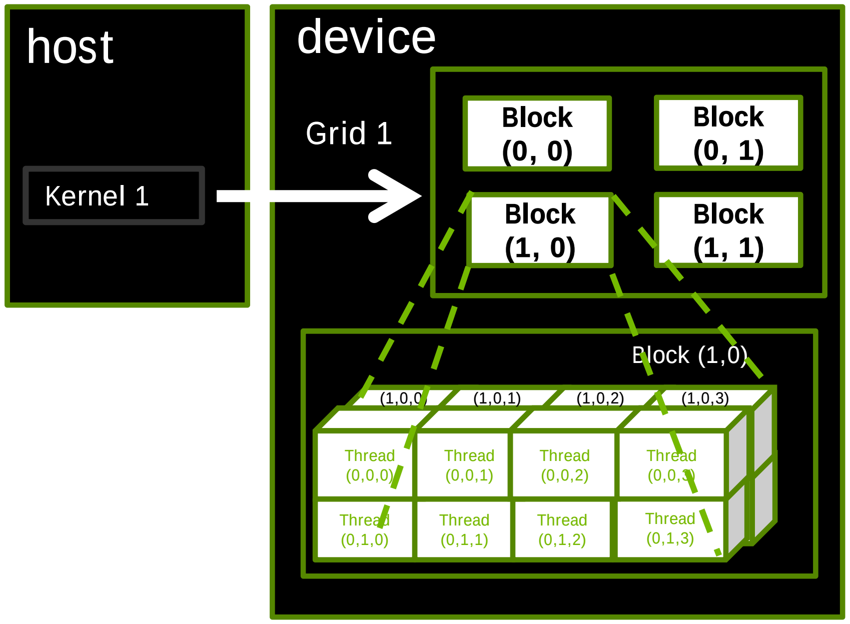
Lecture 3.2 – CUDA Parallelism Model

Multidimensional Kernel Configuration

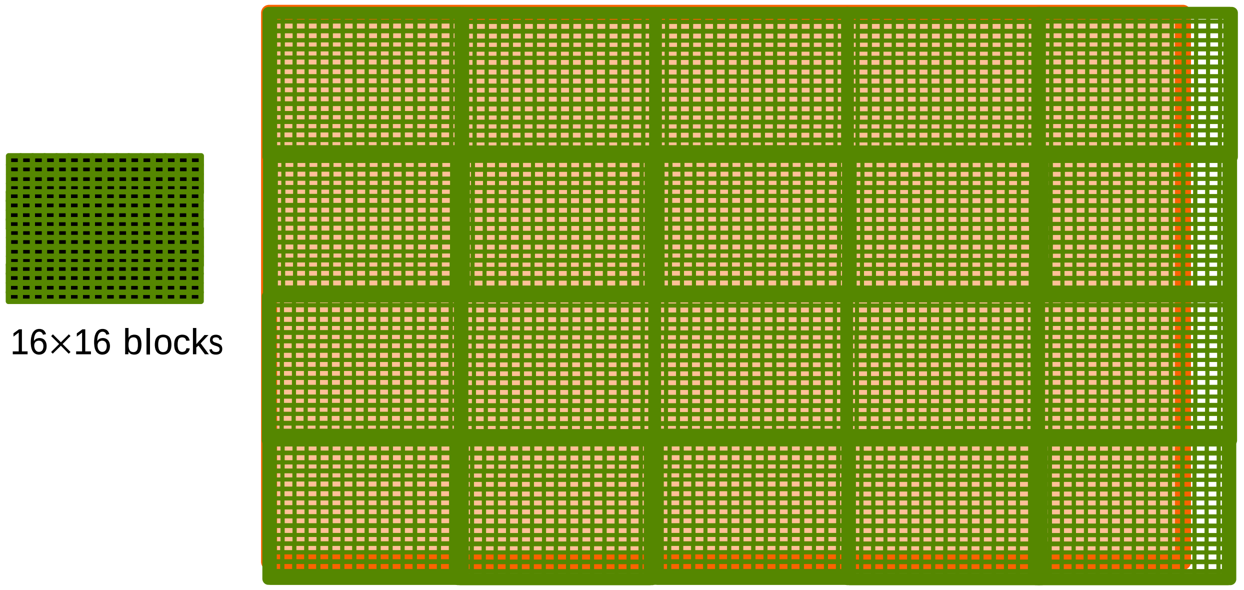
Objective

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

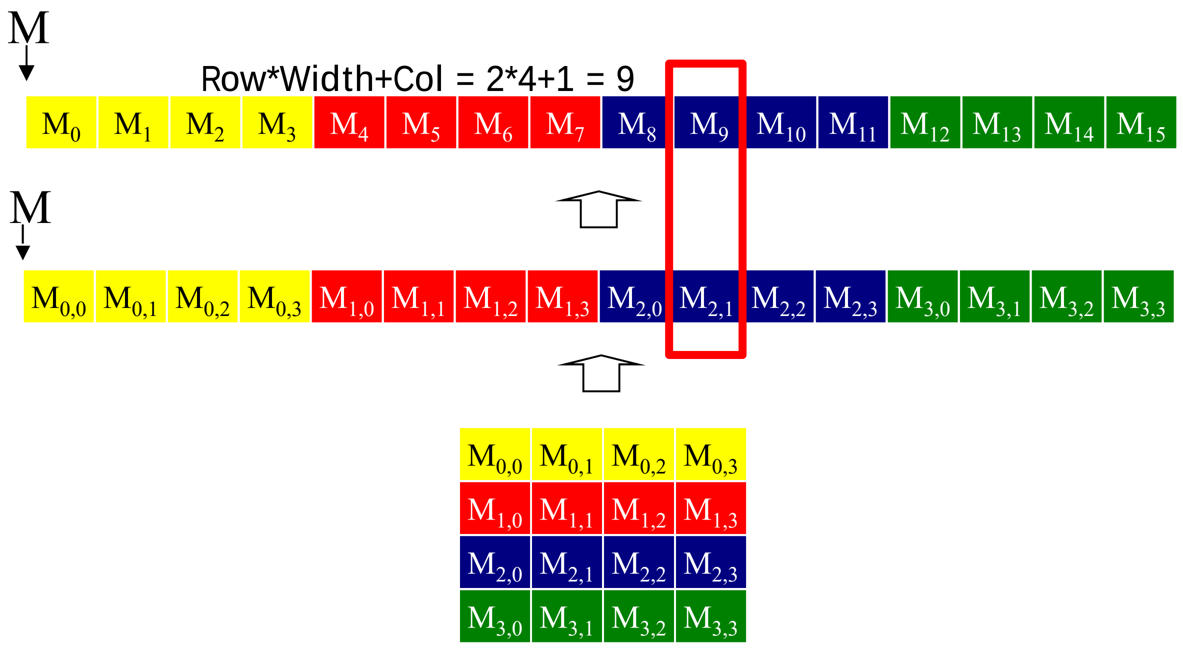
A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid



Row-Major Layout in C/C++


M
Row*Width+Col = 2*4+1 = 9
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

Source Code of a PictureKernel

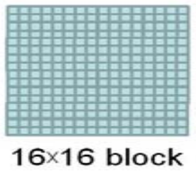
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                                   int height, int width)
{

  // 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];
  }
}
```

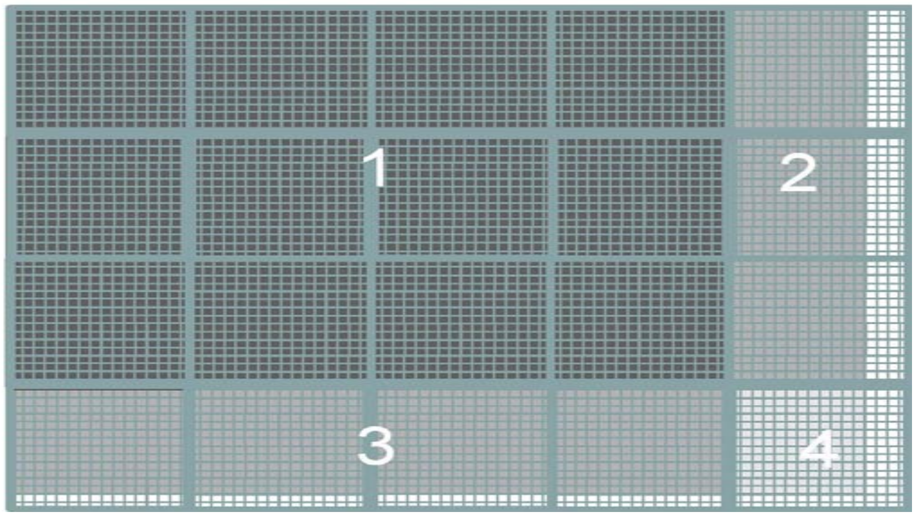
Scale every pixel value by 2.0

Host Code for Launching PictureKernel

```
// assume that the picture is 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.

GPU Teaching Kit

Accelerated Computing

The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.

GPU Teaching Kit

Accelerated Computing

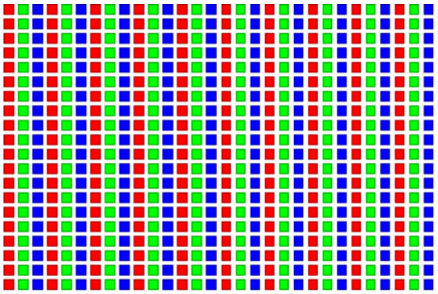
Lecture 3.3 – CUDA Parallelism Model

Color-to-Grayscale Image Processing Example

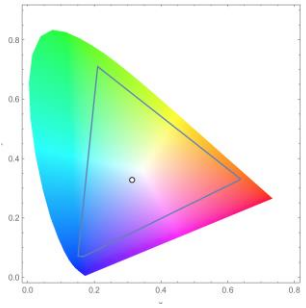
Objective

- 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
- 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

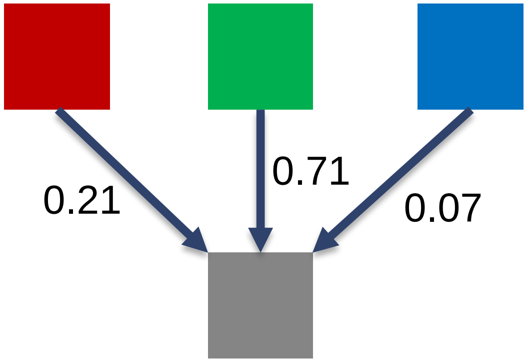


RGB to Grayscale Conversion

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

Color 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

```
#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) {











 }
}
```

RGB to Grayscale Conversion Code

```
#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



 }
}
```

RGB to Grayscale Conversion Code

```
#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;
 }
}
```

NVIDIA

GPU Teaching Kit

Accelerated Computing

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.

GPU Teaching Kit

Accelerated Computing

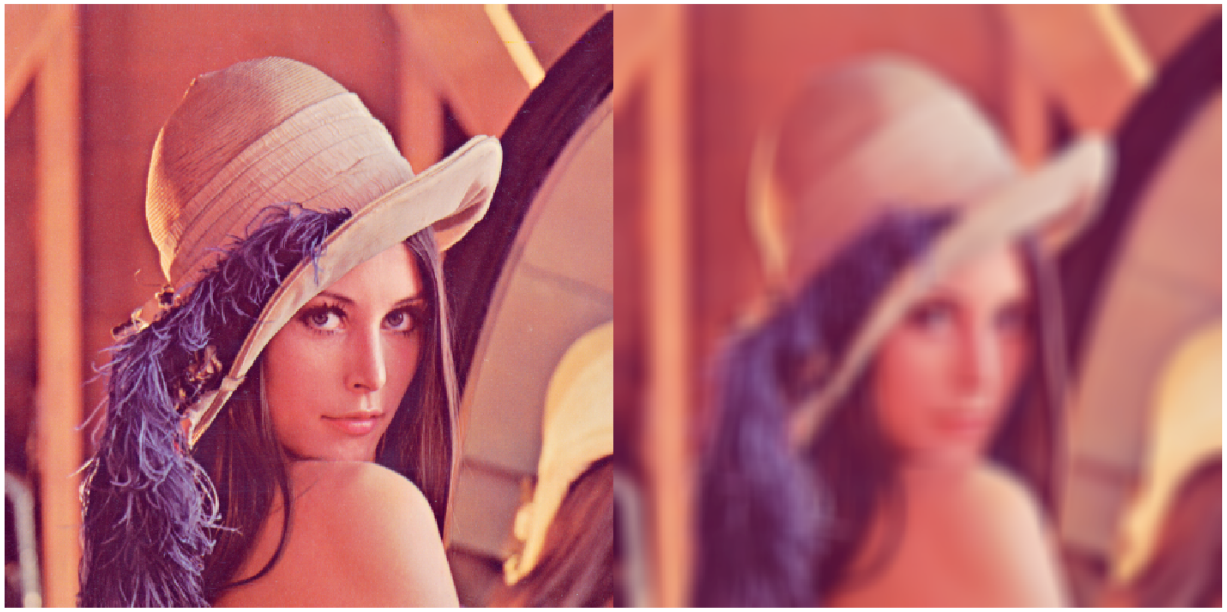
Lecture 3.4 – CUDA Parallelism Model

Image Blur Example

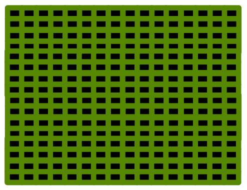
Objective

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

Image Blurring



Blurring Box



Pixels processed by a thread block

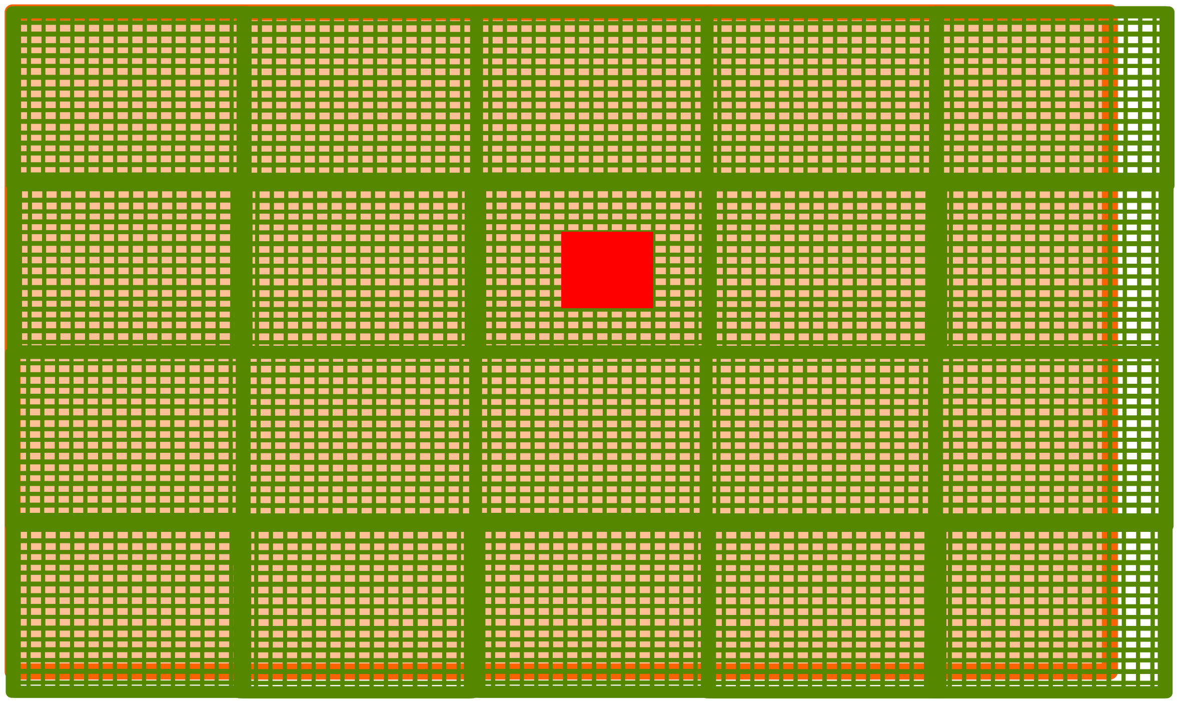


Image Blur as a 2D Kernel

```
__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);
    }
}
```

NVIDIA

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GPU Teaching Kit

Accelerated Computing

The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.

GPU Teaching Kit

Accelerated Computing

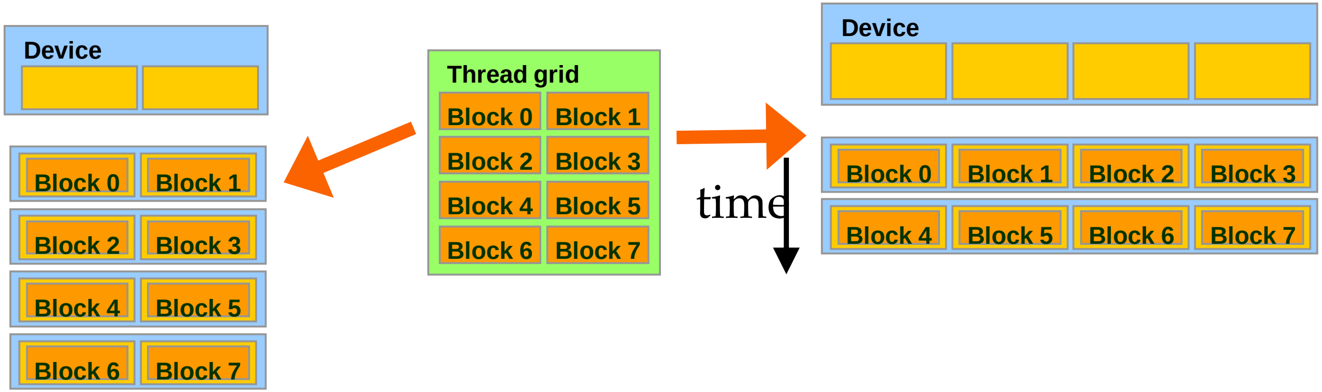
Lecture 3.5 – CUDA Parallelism Model

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

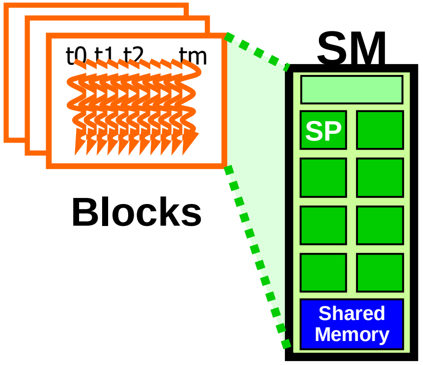
Transparent Scalability



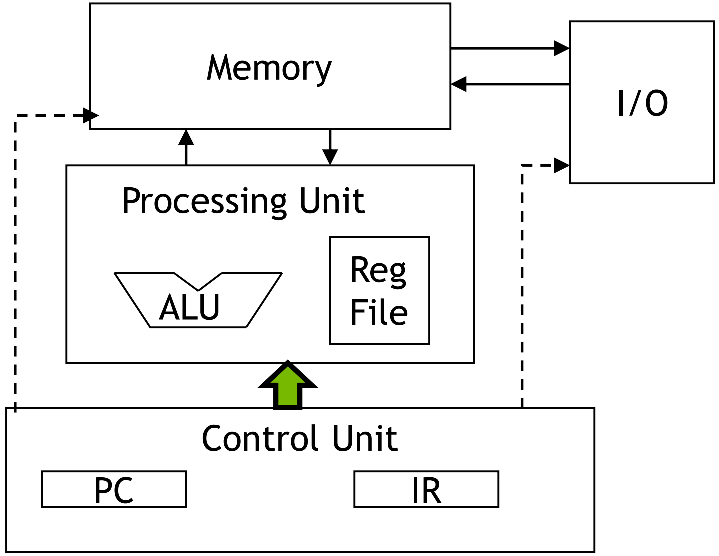
– Each block can execute in any order relative to others.
– Hardware is free to assign blocks to any processor at any time
  – A kernel scales to any number of parallel processors

Example: Executing Thread Blocks

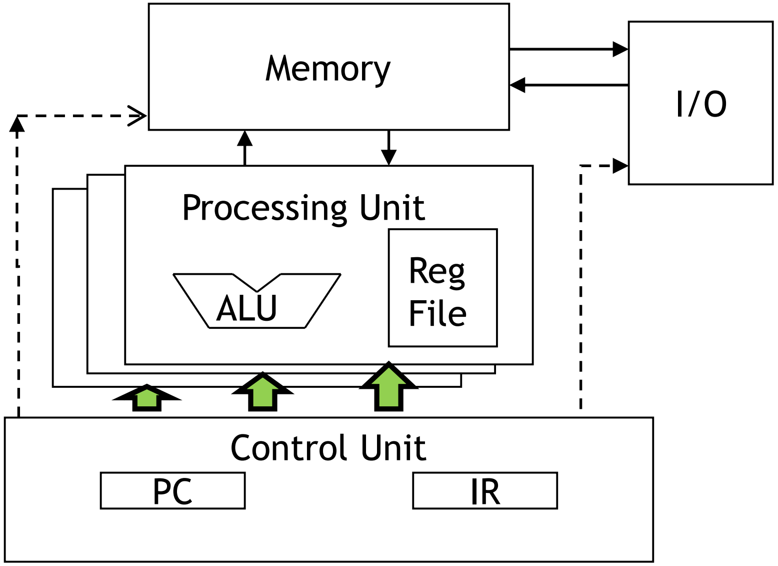
- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to 32 blocks to each SM as resource allows
  - Volta SM can take up to **2048** threads
    - Could be 256 (threads/block) * 8 blocks
    - Or 512 (threads/block) * 4 blocks, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



The Von-Neumann Model



The Von-Neumann Model with SIMD units



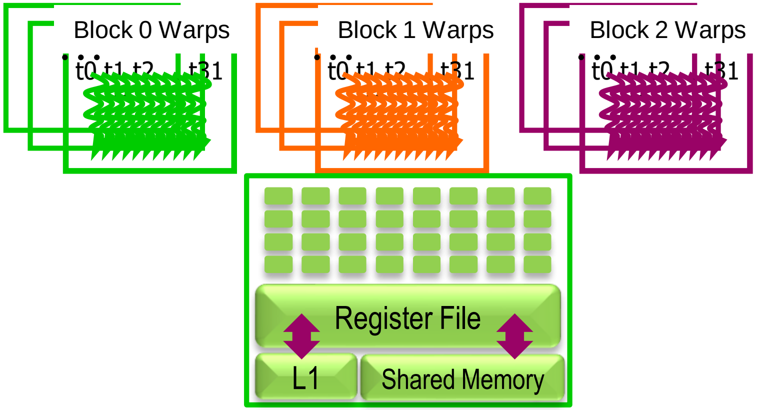
Single Instruction Multiple Data
(SIMD)

Warps as Scheduling Units

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

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



Example: Thread Scheduling (Cont.)

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

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?
  - For 4X4, we have 16 threads per Block. Each SM can take up to 2048 threads, which translates to 128 Blocks. However, each SM can only take up to 32 Blocks, so only 512 threads will go into each SM!
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 2048 threads, it can take up to 32 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 30X30, we would have 900 threads per Block. Only two blocks could fit into an SM for Volta, so only 1800/2048 of the SM thread capacity would be utilized.

GPU Teaching Kit

The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.