



GPU Teaching Kit
Accelerated Computing



Lecture 3 - CUDA Parallelism Model

Next Objective

- To provide you with a framework for further studies on
 - Thinking about the problems of parallel programming
 - Discussing your work with others
 - Approaching complex parallel programming problems
 - Using or building useful tools and environments

Fundamentals of Parallel Computing

- Parallel computing requires that
 - The problem can be decomposed into sub-problems that can be safely solved at the same time
 - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
 - To solve problems in less time (strong scaling), and/or
 - To solve bigger problems (weak scaling), and/or
 - To achieve better solutions (advancing science)

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

Shared Memory vs. Message Passing

- We are focusing on shared memory parallel programming
 - This is what CUDA (and OpenMP, OpenCL) is based on
 - Future massively parallel microprocessors are expected to support shared memory at the chip level
- The programming considerations of message passing model is quite different!
 - However, you will find parallels for almost every technique you learned in this course
 - Need to be aware of space-time constraints

Data Sharing

- Data sharing can be a double-edged sword
 - Excessive data sharing drastically reduces advantage of parallel execution
 - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
 - Efficient use of on-chip, shared storage and datapaths
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization
- **Many:Many**, **One:Many**, **Many:One**, **One:One**

Synchronization

- Synchronization == Control Sharing
- Barriers make threads wait until all threads catch up
- Waiting is lost opportunity for work
- Atomic operations may reduce waiting
 - Watch out for serialization

- Important: be aware of which items of work are truly independent

Parallel Programming Coding Styles – Program and Data Models

Program Models

SPMD

Master/Worker

Loop Parallelism

Fork/Join

Data Models

Shared Data

Shared Queue

Distributed Array

These are not
necessarily mutually
exclusive.

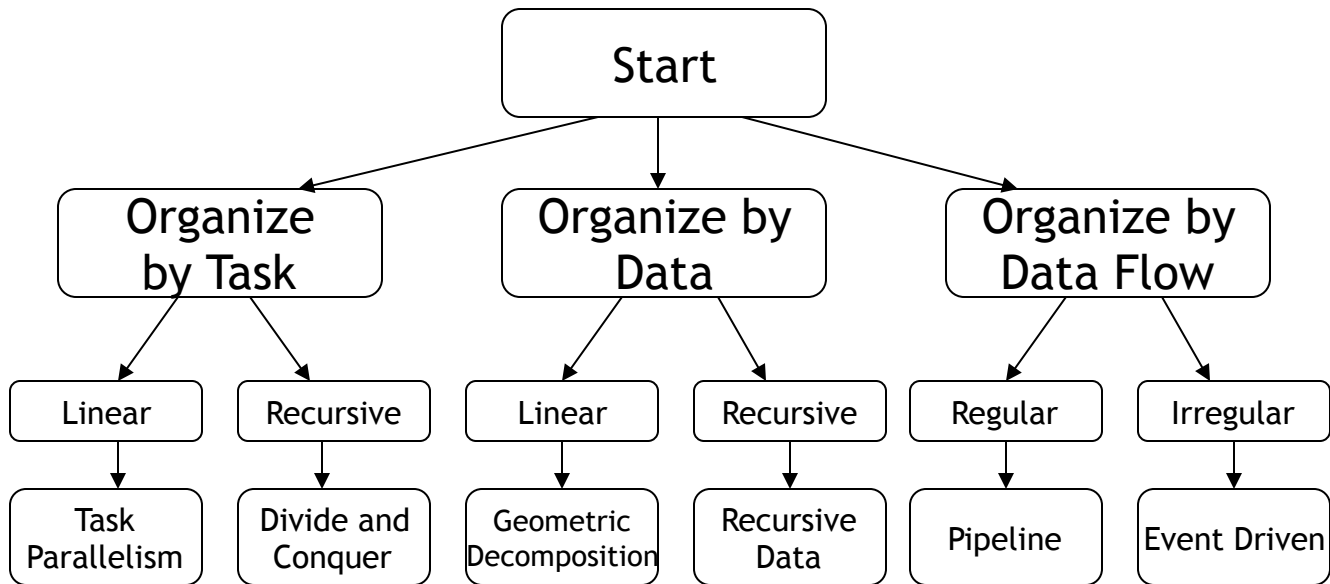
Program Models

- SPMD (Single Program, Multiple Data)
 - All PE's (Processor Elements) execute the same program in parallel, but has its own data
 - Each PE uses a unique ID to access its portion of data
 - Different PE can follow different paths through the same code
 - This is essentially the CUDA Grid model (also OpenCL, MPI)
 - SIMD is a special case – WARP used for efficiency
- Master/Worker
- Loop Parallelism
- Fork/Join

Program Models

- SPMD (Single Program, Multiple Data)
- Master/Worker (OpenMP, OpenACC, TBB)
 - A Master thread sets up a pool of worker threads and a bag of tasks
 - Workers execute concurrently, removing tasks until done
- Loop Parallelism (OpenMP, OpenACC, C++AMP)
 - Loop iterations execute in parallel
 - FORTRAN do-all (truly parallel), do-across (with dependence)
- Fork/Join (Posix p-threads)
 - Most general, generic way of creation of threads

Algorithm Structure



More on SPMD

- Dominant coding style of scalable parallel computing
 - MPI code is mostly developed in SPMD style
 - Many OpenMP code is also in SPMD (next to loop parallelism)
 - Particularly suitable for algorithms based on task parallelism and geometric decomposition.
- Main advantage
 - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

Typical SPMD Program Phases

- Initialize
 - Establish localized data structure and communication channels
- Obtain a unique identifier
 - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
 - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
 - Decompose global data into chunks and localize them, or
 - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
- Finalize
 - Reconcile global data structure, prepare for the next major iteration

Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
 - Use thread ID in loop index calculations to split loop iterations among threads
 - Potential for memory/data divergence
 - Use thread ID or conditions based on thread ID to branch to their specific actions
 - Potential for instruction/execution divergence

Both can have very different performance results and code complexity depending on the way they are done.

Next 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

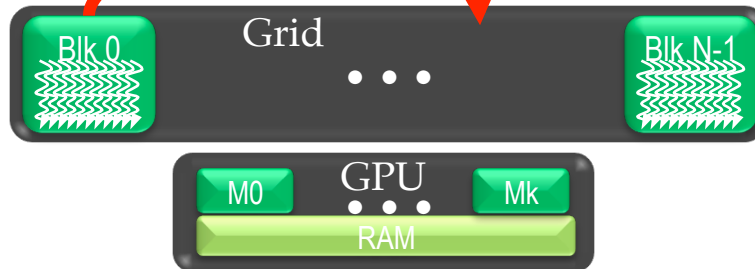
```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    dim3 DimGrid((n-1)/256 + 1, 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

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];  
}
```

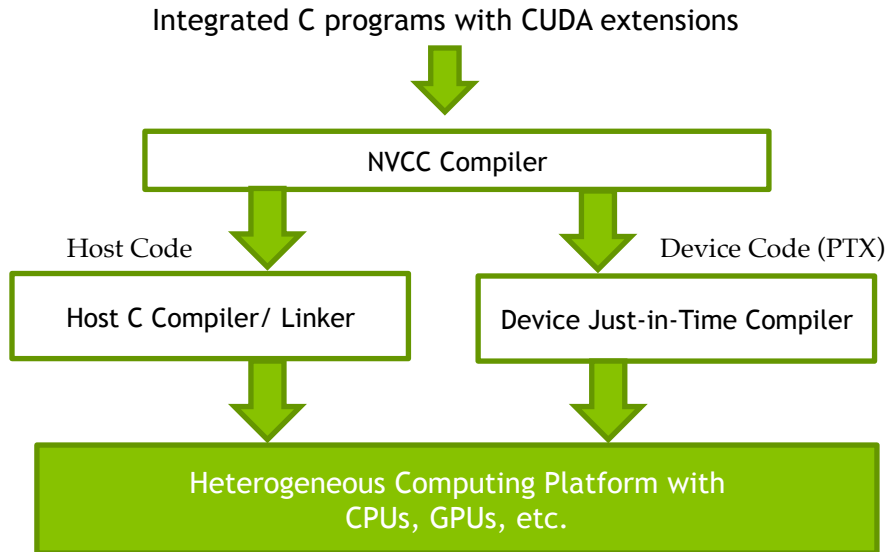


More on CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>device float DeviceFunc() ↵</code>	device	device
<code>__global__ void KernelFunc() ↵</code>	device	host
<code>__host__ float HostFunc() ↵</code>	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

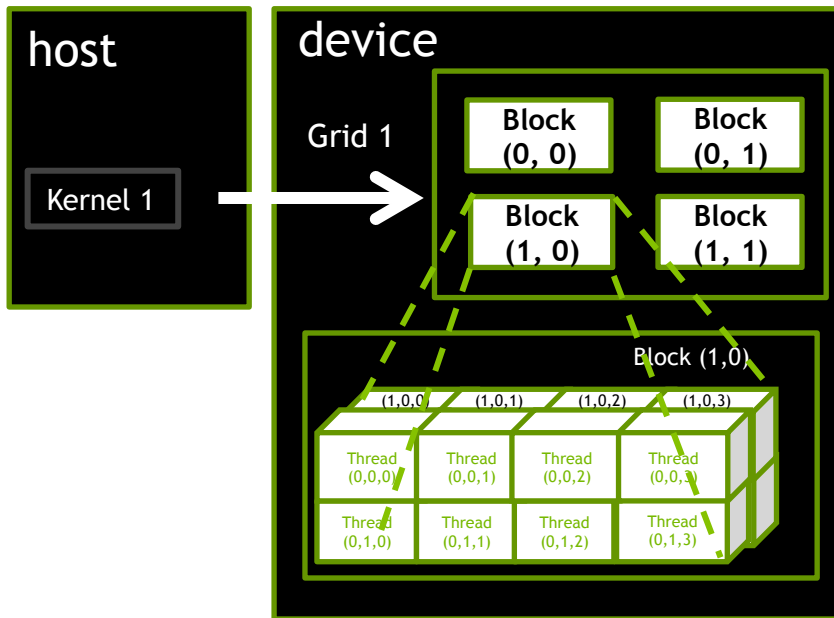
Compiling A CUDA Program



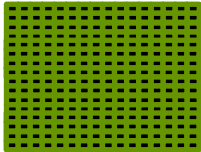
Next Objective

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

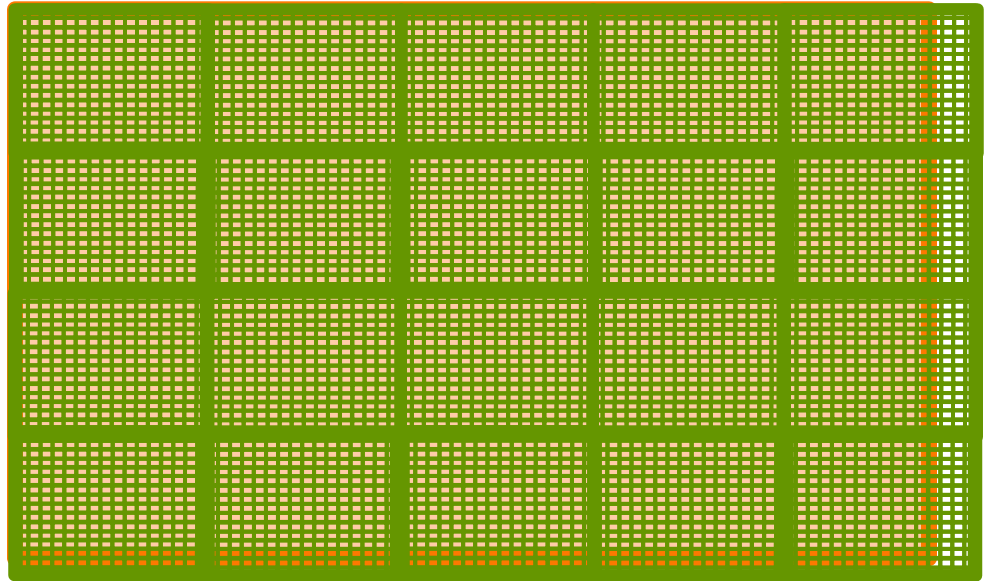
A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid

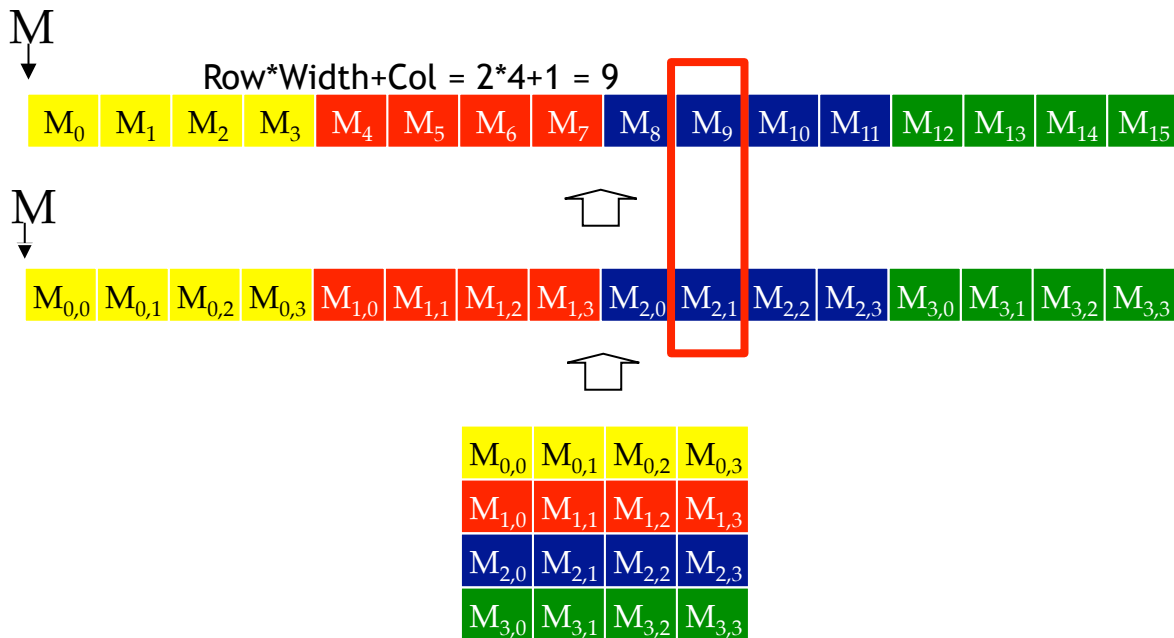


16x16 blocks



62x76 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;

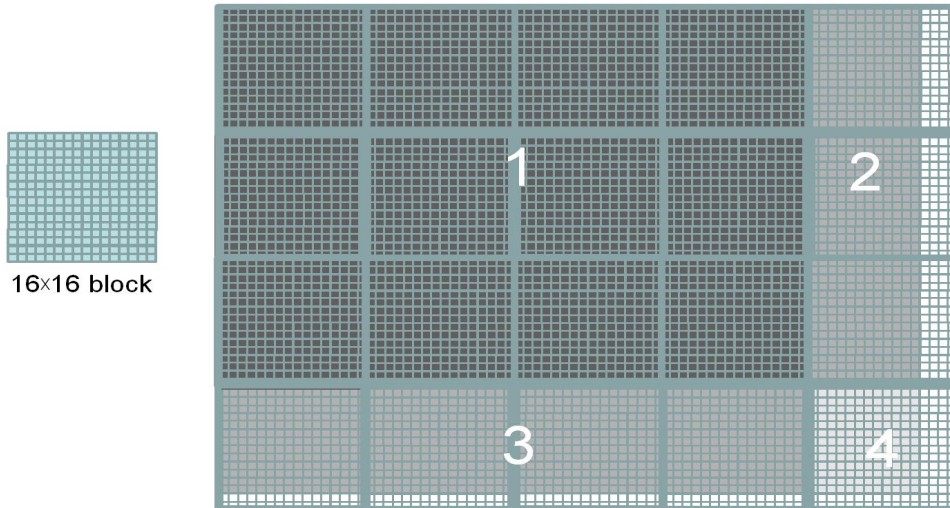
    // each thread computes one element of d_Pout if in
    range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

Scale every pixel value by 2.0

Host Code for Launching PictureKernel

```
// assume that the picture is m x 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 62x76 Picture with 16x16 Blocks



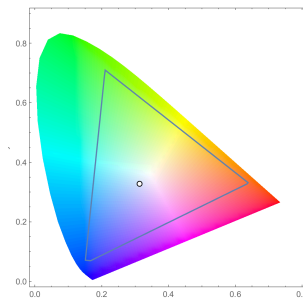
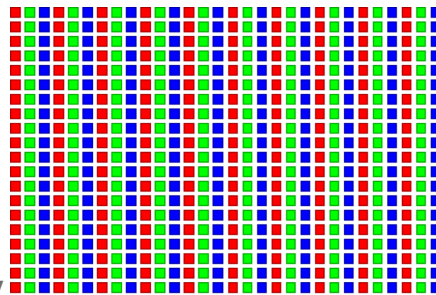
Not all threads in a Block will follow the same control flow path.

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



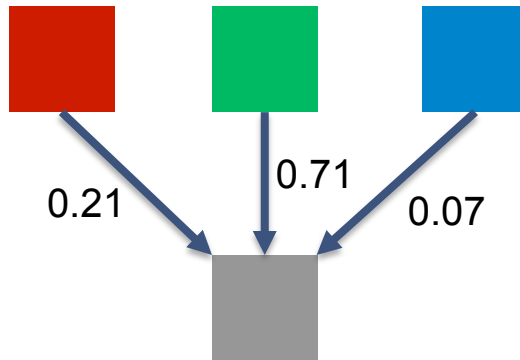
RGB to Grayscale Conversion



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:
 $\text{grayPixel}[I,J] = 0.21*r + 0.71*g + 0.07*b$
- 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



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;
    }
}
```

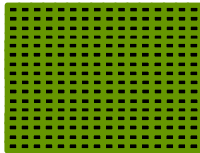
Next Objective

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

Image Blurring



Blurring Box



Pixels
processed
by a
thread
block

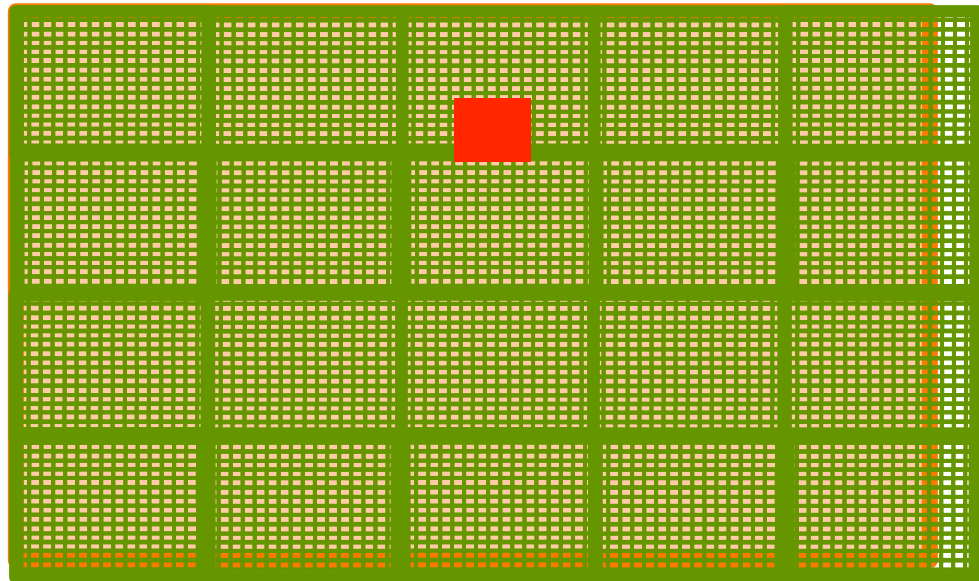


Image Blur as a 2D Kernel

```
__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  
    }  
}
```

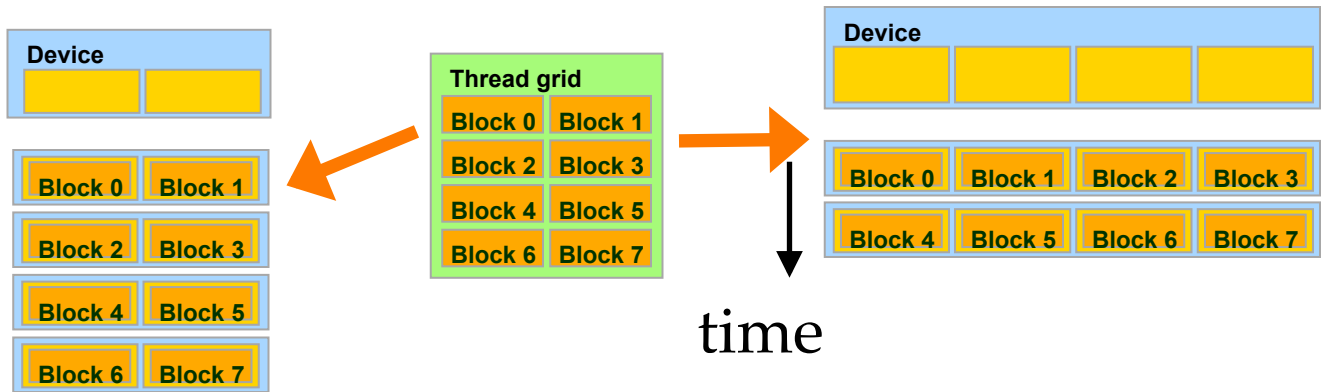
```
__global__
```

```
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);  
    }  
}
```

Next 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

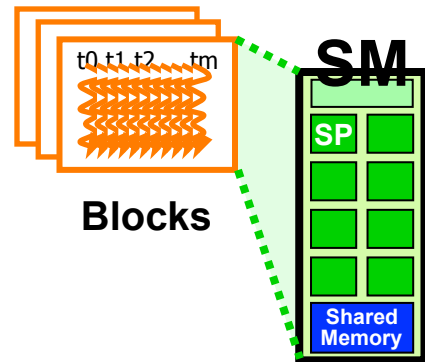
Transparent Scalability



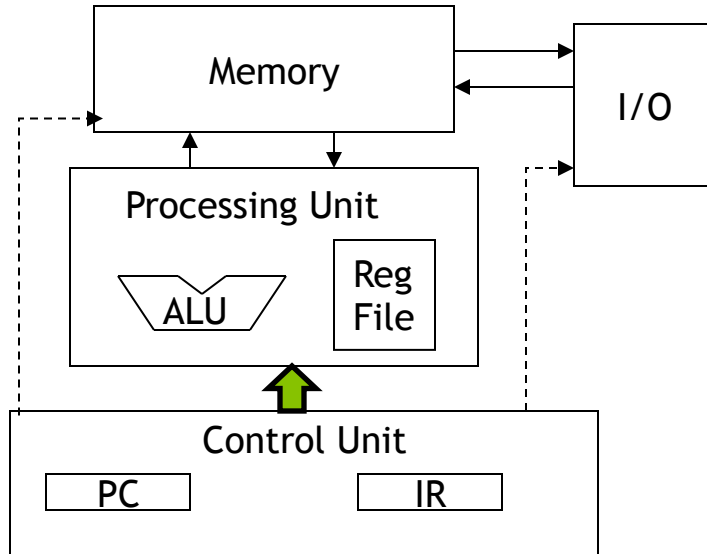
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
 - A kernel scales to any number of parallel processors

Example: Executing Thread Blocks

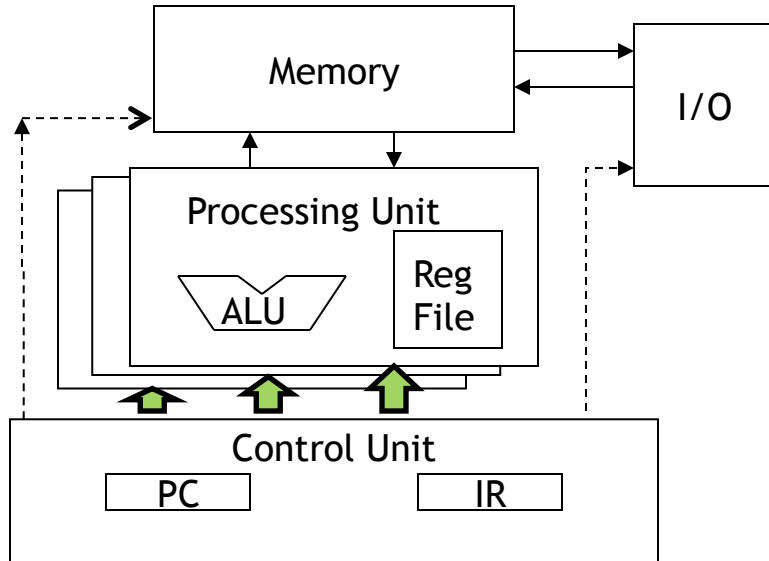
- Threads are assigned to **Streaming Multiprocessors (SM)** in block granularity
 - Up to 32 blocks to each SM as resource allows
 - Volta SM can take up to **2048** threads
 - Could be 256 (threads/block) * 8 blocks
 - Or 512 (threads/block) * 4 blocks, etc.
- SM maintains thread/block idx #s
- SM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



The Von-Neumann Model



The Von-Neumann Model with SIMD units



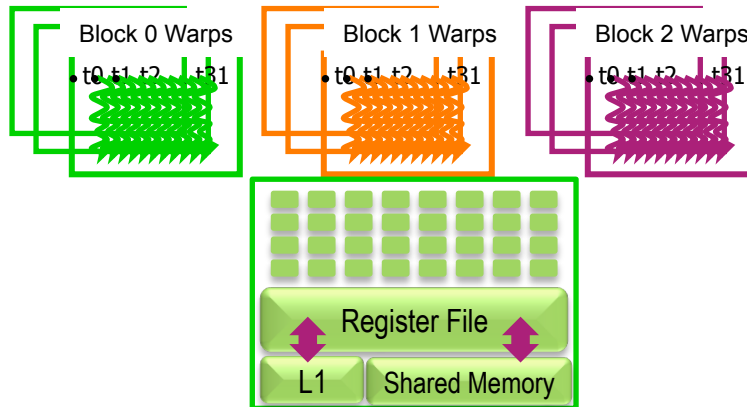
Single Instruction Multiple
Data
(SIMD)

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

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](https://creativecommons.org/licenses/by-nc/4.0/).