CUDA Parallelism Model

UNIVERSITY OF CALIFORNIA, RIVERSIDE

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)

```
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);
}
```

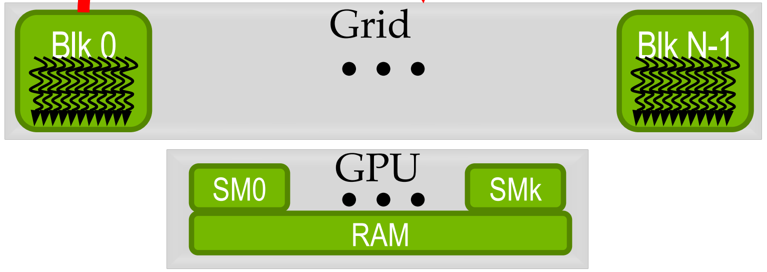
More on Kernel Launch (Host Code)

```
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);
}
```

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];
}
```



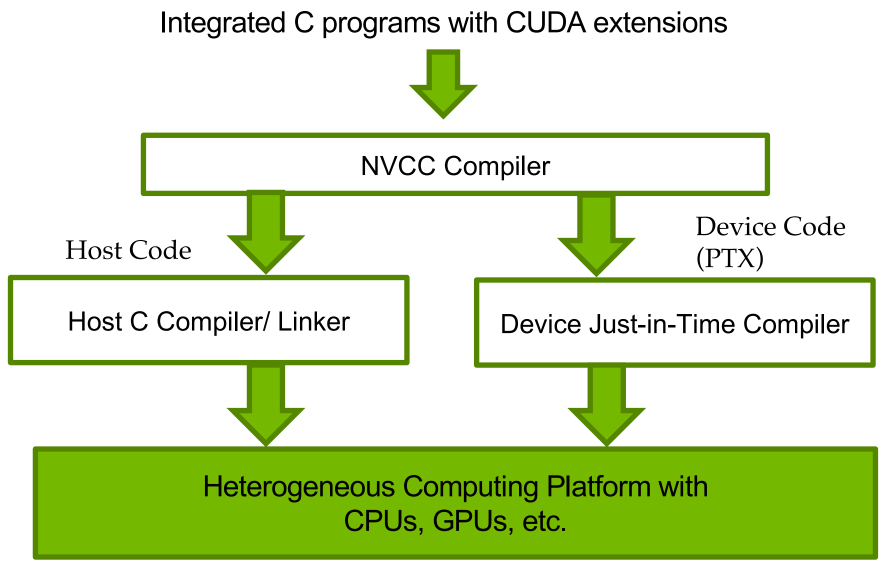
More on CUDA Function Declarations

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

Device = GPU
Host = CPU

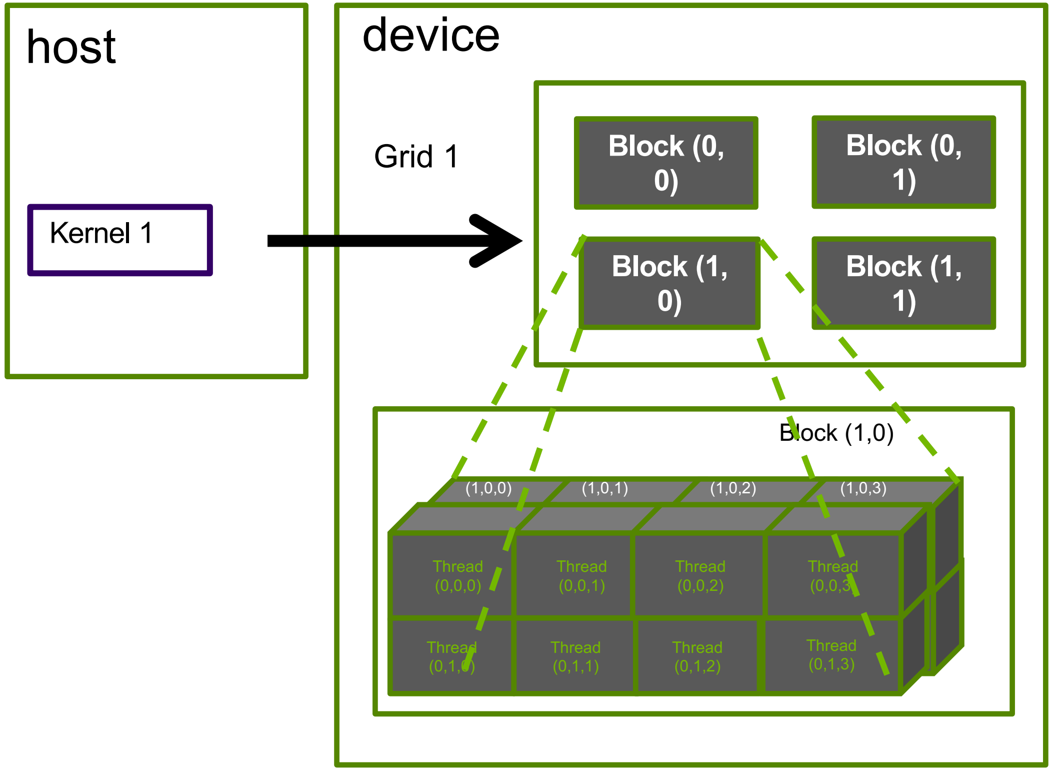
- **`__global__`** defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return **`void`**
- **`__device__`** and **`__host__`** can be used together
- **`__host__`** is optional if used alone

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.

MULTI-DIMENSIONAL KERNEL CONFIGURATION

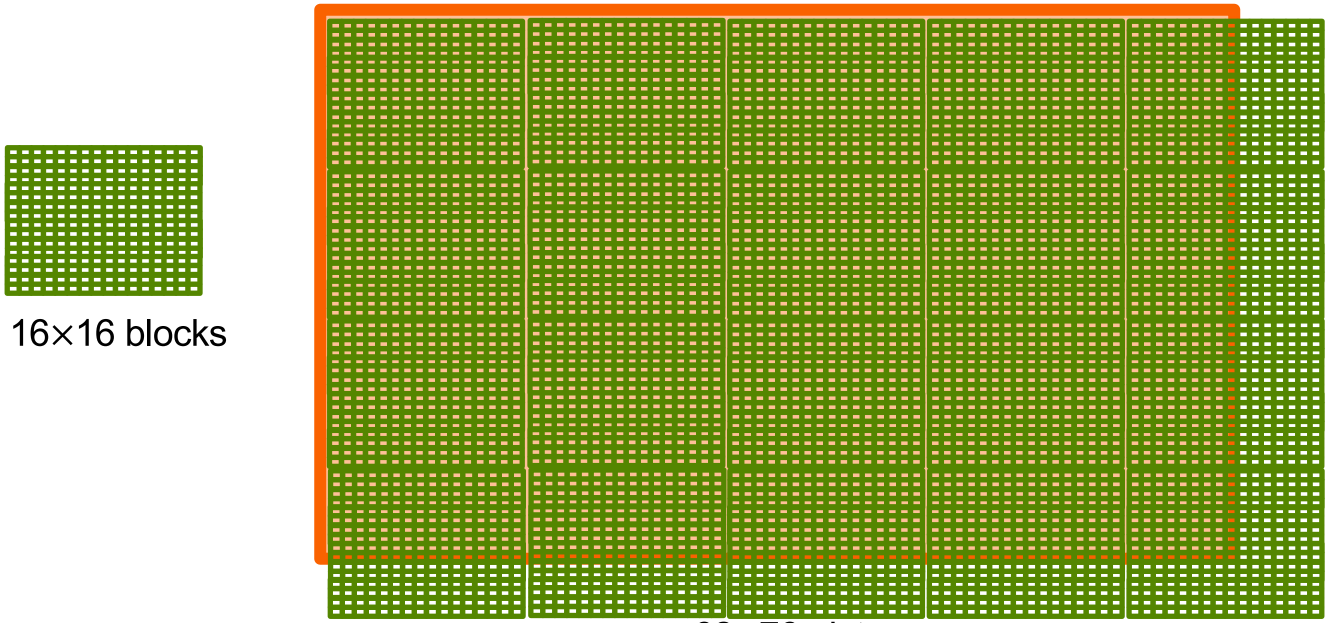
A Multi-Dimensional Grid Example



2D Grid

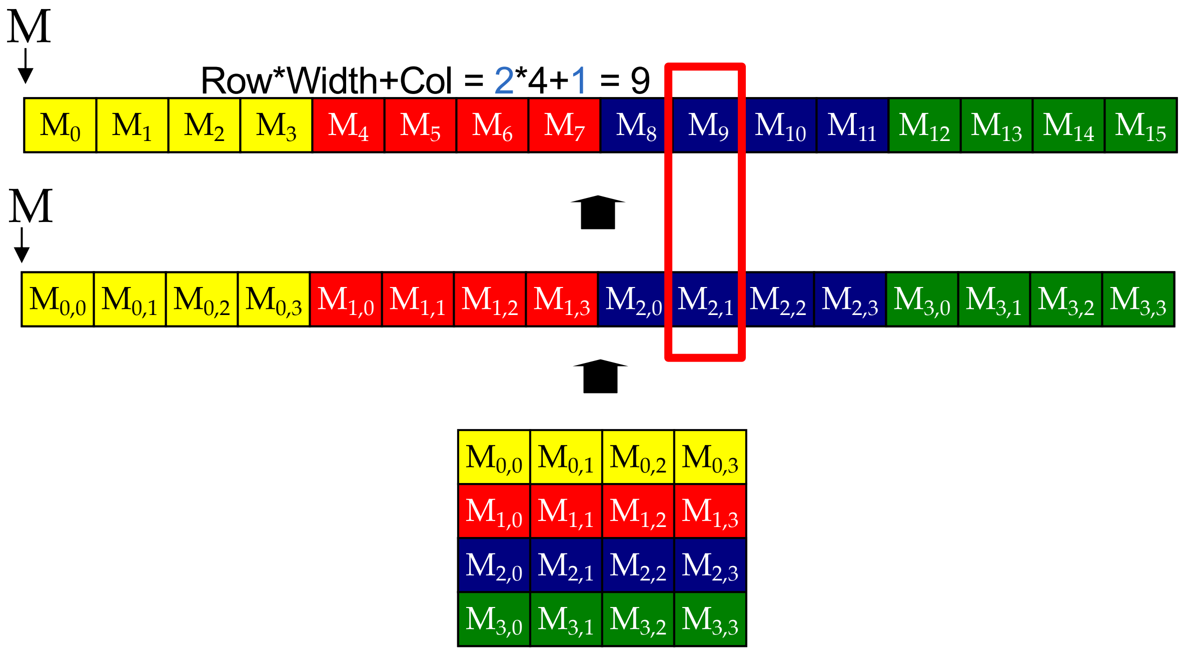
3D Blocks

Processing a Picture with a 2D Grid



62×76 picture

Review: Row-Major Memory 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
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

Example Kernel Code of a PictureKernel

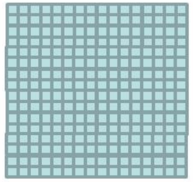
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width)
{

  // 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];
  }
}
```

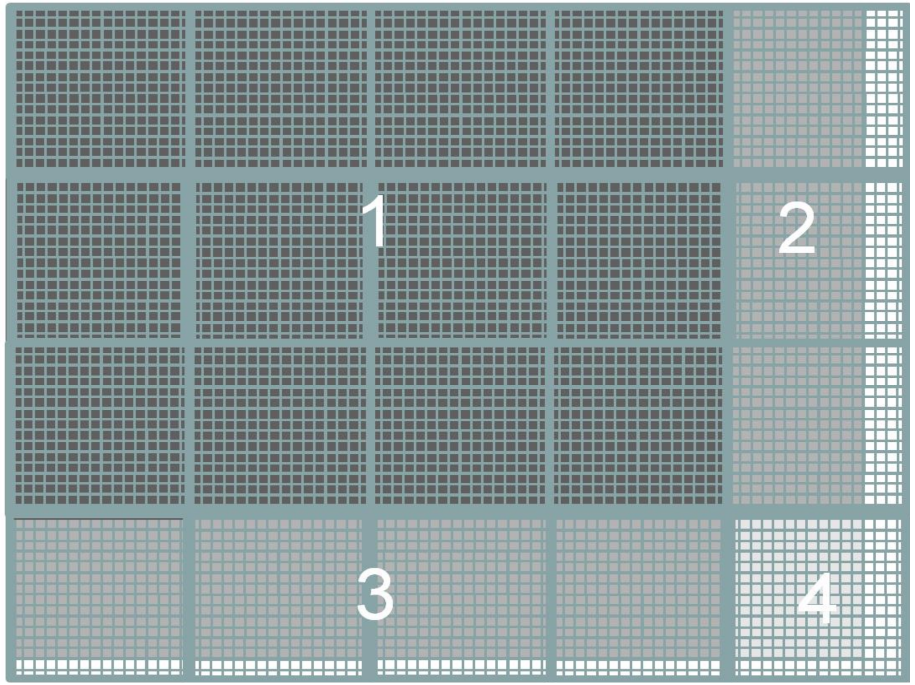
Scale every pixel value by 2.0

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


16×16 block

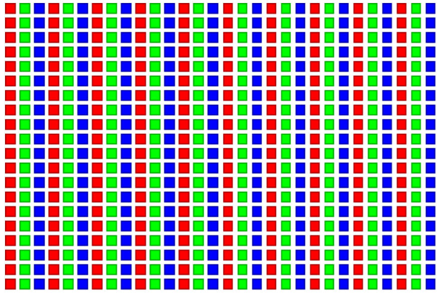


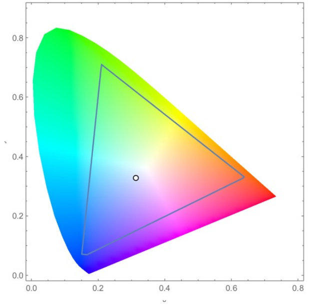
Note: Not all threads in a Block will follow the same control flow path.

COLOR-TO-GRAYSCALE IMAGE PROCESSING EXAMPLE

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 AdbobeRGB 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

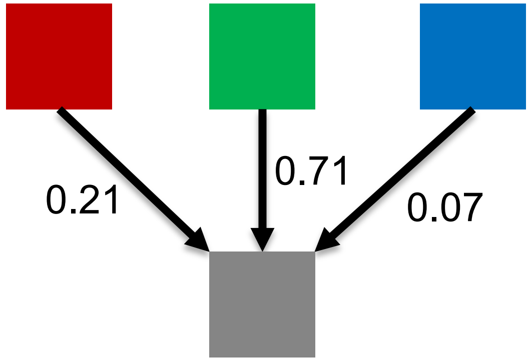
A grayscale digital image is an image in which the value of each pixel carries only intensity information.

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) {
    // 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;
 }
}
```

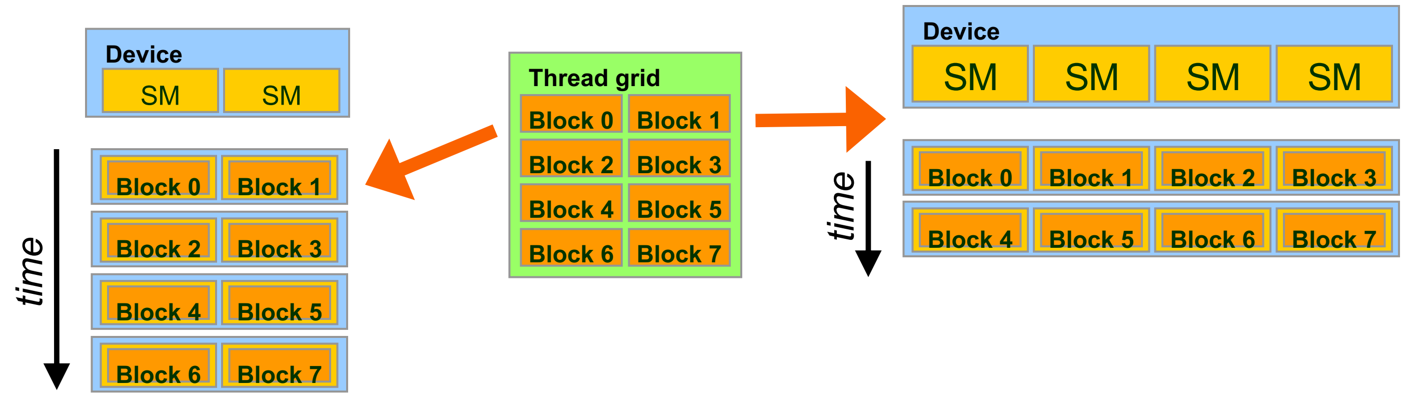
Every thread mapped to output pixel

THREAD BLOCK SCHEDULING

Objective

- To learn how a CUDA kernel utilizes hardware execution resources
  - Assigning thread blocks to execution resources
  - Capacity constrains of execution resources (hardware)
  - “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

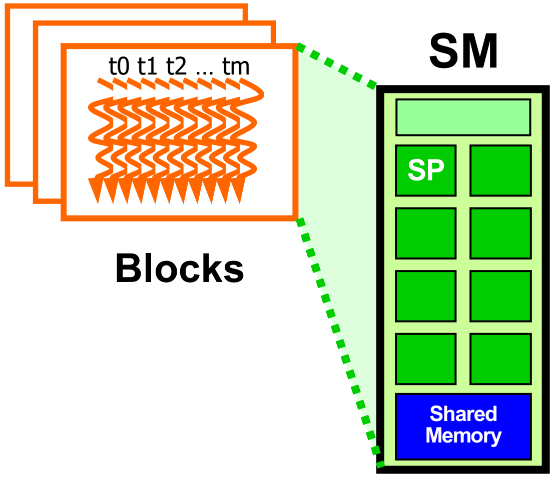
Example: Executing Thread Blocks

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

- Example with Fermi architecture:
  - 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



What Block Size Should I Use??

- For 2D algorithms using multiple blocks, *should I use 8X8, 16X16 or 32X32 blocks for Fermi?*
  - Limitations: 1536 threads/SM & 8 blocks / SM

Block size	# Threads / Block	# Blocks *that can fit in* SM's threads	Max blocks / SM	# Blocks in SM	# of threads utilized in SM
8x8					/ 1536
16x16					/ 1536
32x32					/ 1536

What Block Size Should I Use??

- For 2D algorithms using multiple blocks, *should I use 8X8, 16X16 or 32X32 blocks for Fermi?*
  - Limitations: 1536 threads/SM & 8 blocks / SM

Block size	# Threads / Block	# Blocks *that can fit in* SM's threads	Max blocks / SM	# Blocks in SM	# of threads utilized in SM
8x8	64	24	8	8	512 / 1536
16x16					/ 1536
32x32					/ 1536

Limited by number of blocks per SM

What Block Size Should I Use??

- For 2D algorithms using multiple blocks, *should I use 8X8, 16X16 or 32X32 blocks for Fermi?*
  - Limitations: 1536 threads/SM & 8 blocks / SM

Block size	# Threads / Block	# Blocks *that can fit in* SM's threads	Max blocks / SM	# Blocks in SM	# of threads utilized in SM
8x8	64	24	8	8	512 / 1536
16x16	256	6	8	6	1536 / 1536
32x32					/ 1536

Fully utilize all hardware threads!

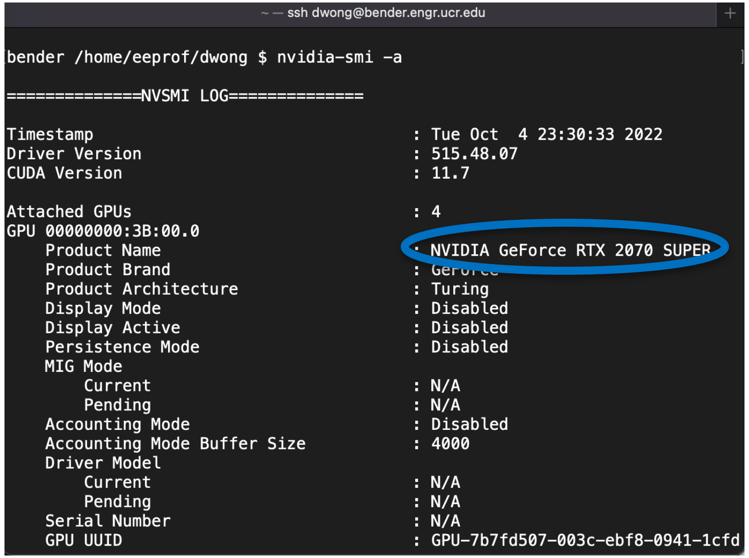
What Block Size Should I Use??

- For 2D algorithms using multiple blocks, *should I use 8X8, 16X16 or 32X32 blocks for Fermi?*
  - Limitations: 1536 threads/SM & 8 blocks / SM

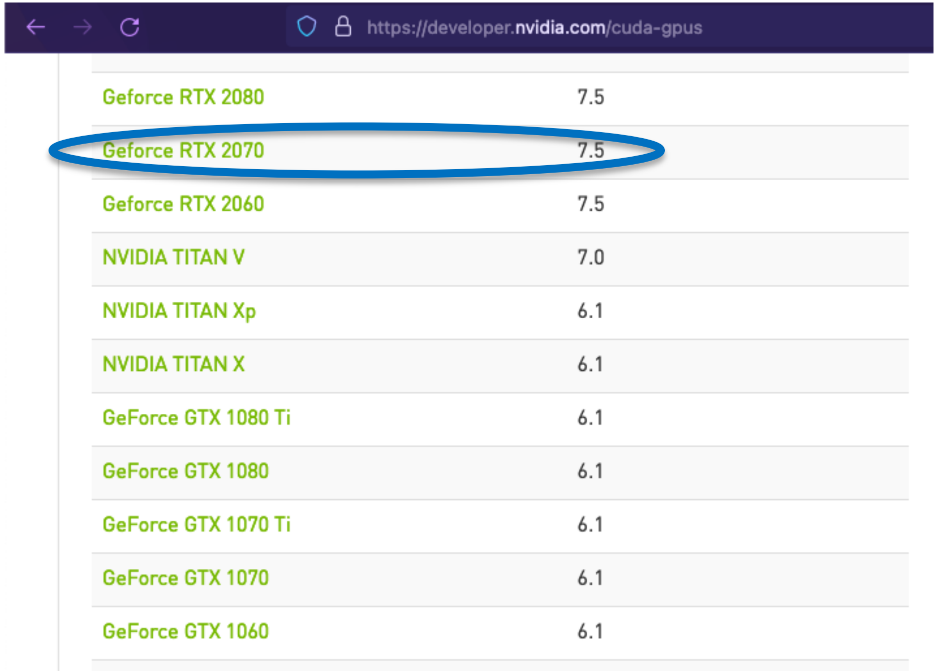
Block size	# Threads / Block	# Blocks *that can fit in* SM's threads	Max blocks / SM	# Blocks in SM	# of threads utilized in SM
8x8	64	24	8	8	512 / 1536
16x16	256	6	8	6	1536 / 1536
32x32	1024	1	8	1	1024 / 1536

Limited by number of threads per SM

What are your GPU's hardware thread limitations?



```
~ — ssh dwong@bender.engr.ucr.edu

bender /home/eeprof/dwong $ nvidia-smi -a

==============NVSMI LOG==============

Timestamp                                 : Tue Oct  4 23:30:33 2022
Driver Version                            : 515.48.07
CUDA Version                              : 11.7

Attached GPUs                             : 4
GPU 00000000:3B:00.0
    Product Name                          : NVIDIA GeForce RTX 2070 SUPER
    Product Brand                         : GeForce
    Product Architecture                  : Turing
    Display Mode                          : Disabled
    Display Active                        : Disabled
    Persistence Mode                      : Disabled
    MIG Mode
        Current                           : N/A
        Pending                           : N/A
    Accounting Mode                       : Disabled
    Accounting Mode Buffer Size           : 4000
    Driver Model
        Current                           : N/A
        Pending                           : N/A
    Serial Number                         : N/A
    GPU UUID                              : GPU-7b7fd507-003c-ebf8-0941-1cfd
```



https://developer.nvidia.com/cuda-gpus

Geforce RTX 2080	7.5
Geforce RTX 2070	7.5
Geforce RTX 2060	7.5
NVIDIA TITAN V	7.0
NVIDIA TITAN Xp	6.1
NVIDIA TITAN X	6.1
GeForce GTX 1080 Ti	6.1
GeForce GTX 1080	6.1
GeForce GTX 1070 Ti	6.1
GeForce GTX 1070	6.1
GeForce GTX 1060	6.1

- Each Nvidia GPU has a "Compute capability"
  - Our RTX 2070 has a compute capability of 7.5

What are your GPU’s hardware thread limitations?

– https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

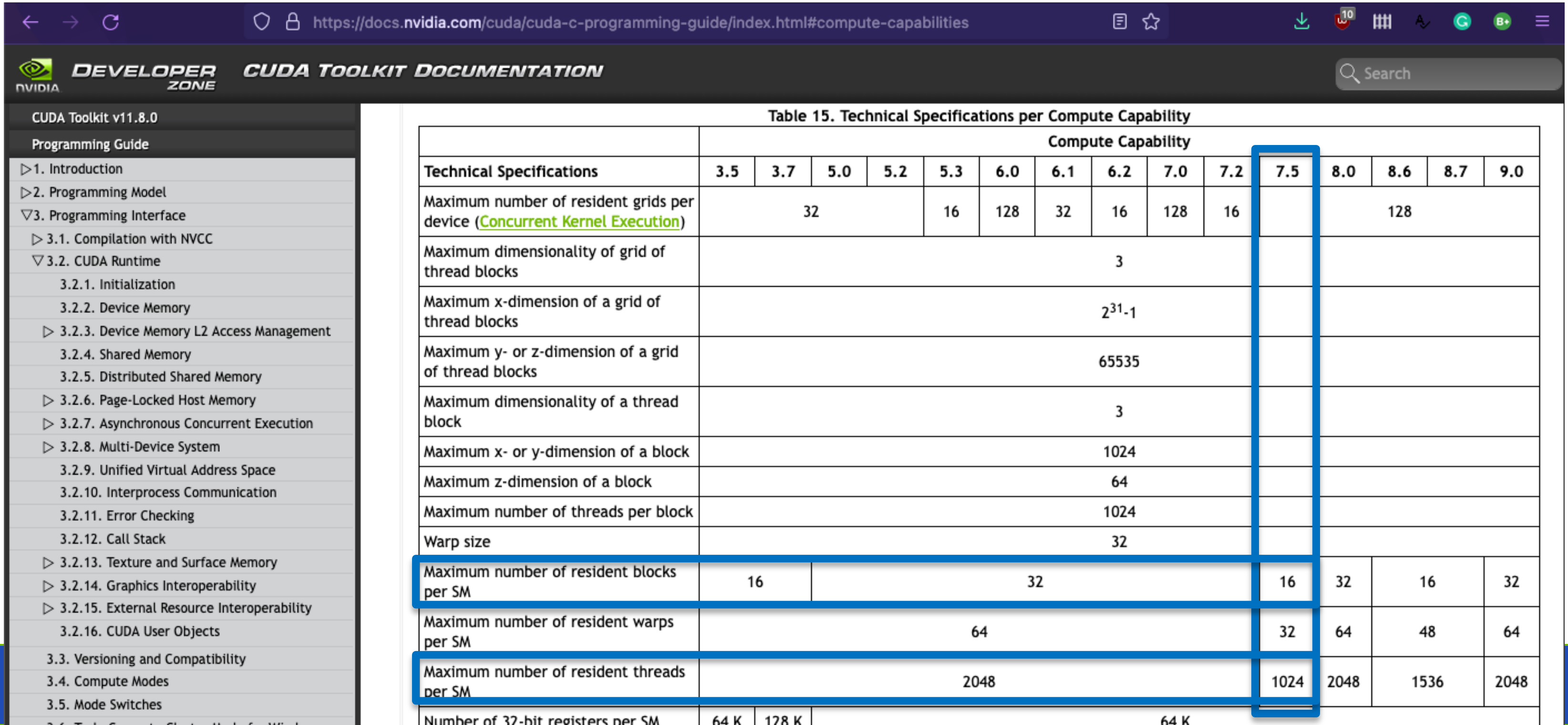
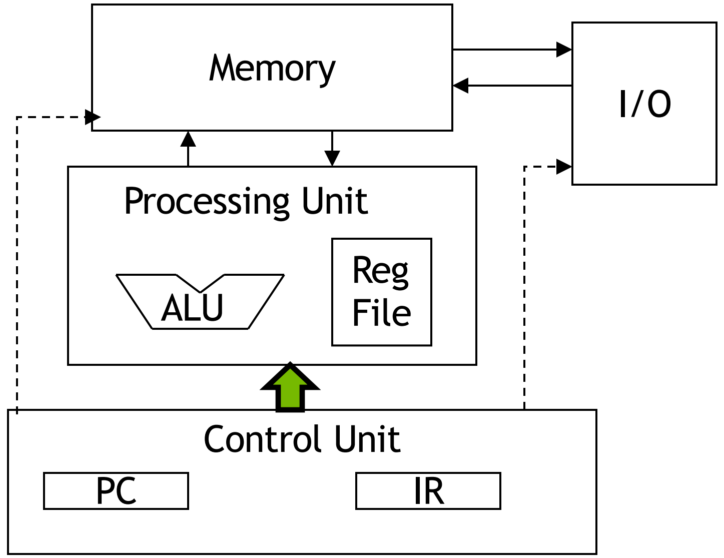

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities
NVIDIA
DEVELOPER ZONE
CUDA TOOLKIT DOCUMENTATION
Search

Table 15. Technical Specifications per Compute Capability

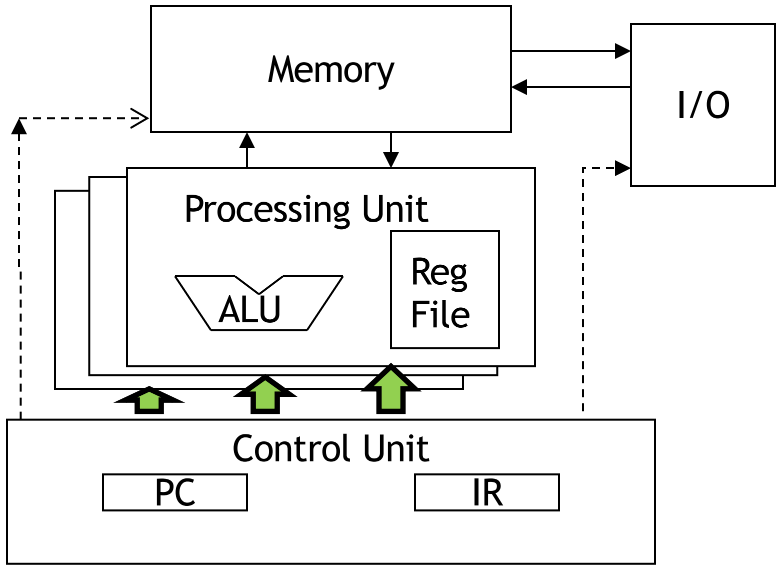
	Compute Capability														
Technical Specifications	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6	8.7	9.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16	128	16		128			
Maximum dimensionality of grid of thread blocks	3														
Maximum x-dimension of a grid of thread blocks	2^{31}-1														
Maximum y- or z-dimension of a grid of thread blocks	65535														
Maximum dimensionality of a thread block	3														
Maximum x- or y-dimension of a block	1024														
Maximum z-dimension of a block	64														
Maximum number of threads per block	1024														
Warp size	32														
Maximum number of resident blocks per SM	16		32								16	32	16		32
Maximum number of resident warps per SM	64										32	64	48		64
Maximum number of resident threads per SM	2048										1024	2048	1536		2048
Number of 32-bit registers per SM	64 K	128 K	64 K												

THREAD BLOCK SCHEDULING - WARPS

The Von-Neumann Model



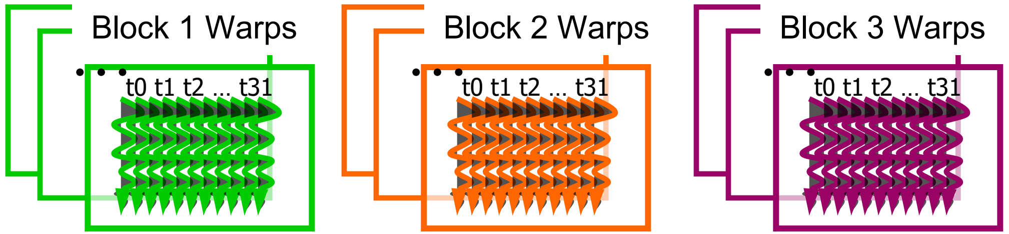
The Von-Neumann Model with SIMD units



Single Instruction Multiple Data (SIMD)

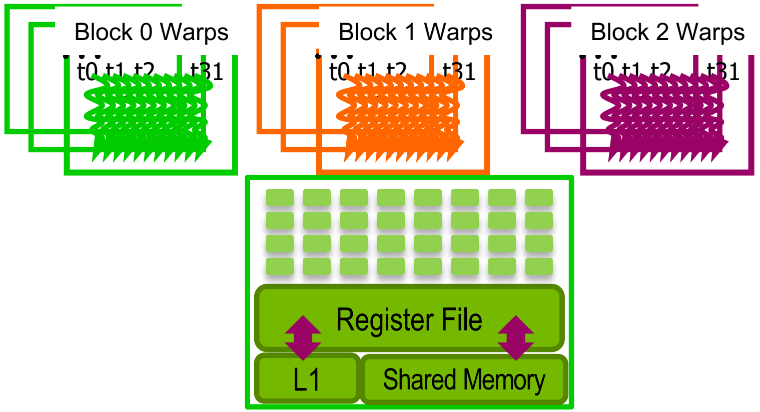
Warps as Scheduling Units

- Each Block is executed as *32-thread Warps*
  - An implementation decision, not part of the CUDA programming model
  - **Warps are scheduling units within SMs**
  - Threads in a warp execute in SIMD (More details later)
  - 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



Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order
  - In x-dimension first, y-dimension next, and z-dimension last

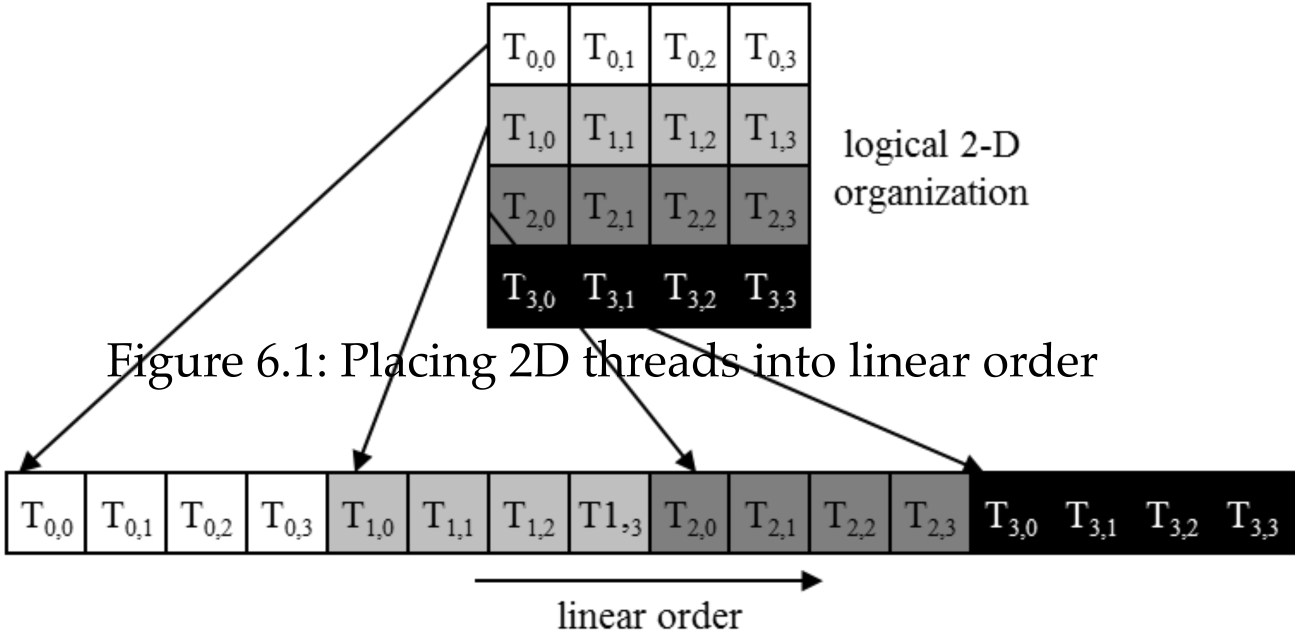

$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
logical 2-D organization
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$ $T_{1,0}$ $T_{1,1}$ $T_{1,2}$ T1,3 $T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$ $T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
linear order

Figure 6.1: Placing 2D threads into linear order

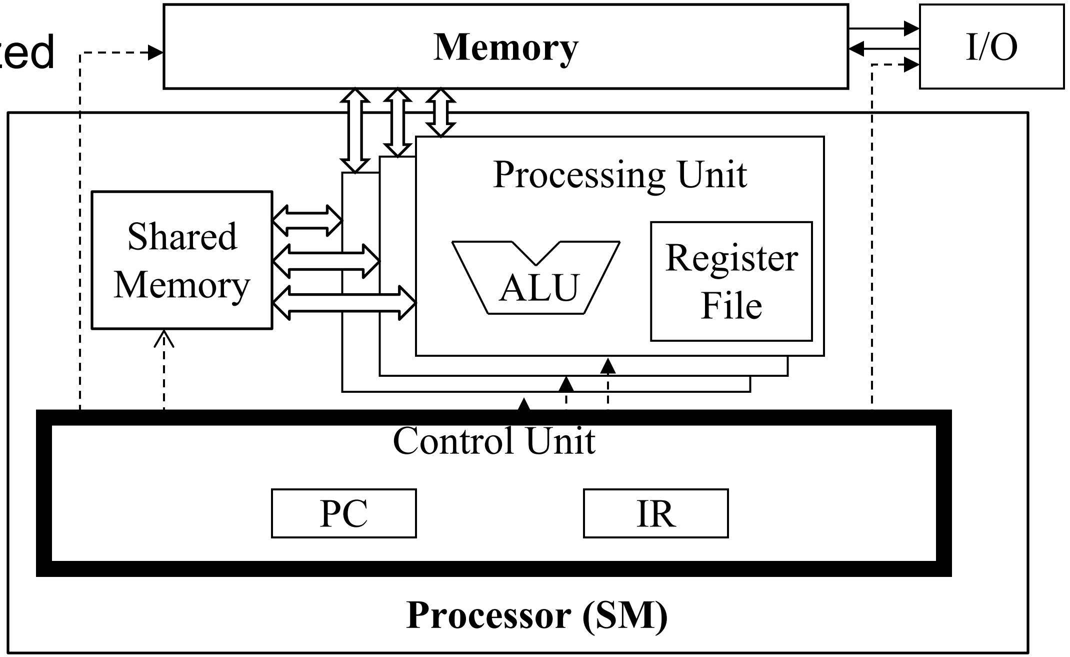
Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0, Warp 1 starts with Thread 32, etc.
- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).

WARP DIVERGENCE

SMs are SIMD Processors

- Control unit for instruction fetch, decode, and control is shared among multiple processing units
  - Control overhead is minimized



SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time
  - Also called SIMT execution model

- This works efficiently if all threads follow the same control flow path
  - All if-then-else statements make the same decision
  - All loops iterate the same number of times

Control Divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - Some take the then-path and others take the else-path of an if-statement
  - Some threads take different number of loop iterations than others

- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements

Control Divergence Examples

- Divergence can arise when branch or loop condition is a function of thread indices
- Example kernel statement with divergence:
  - if (threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - If (blockIdx.x > 2) { }
  - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path

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];
}
```