www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

CUDA Parallelism Model

Marc Jordà, Antonio J. Peña

Based on material from NVIDIA's GPU Teaching Kit

Montevideo, 21-25 October 2019

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);
}
```

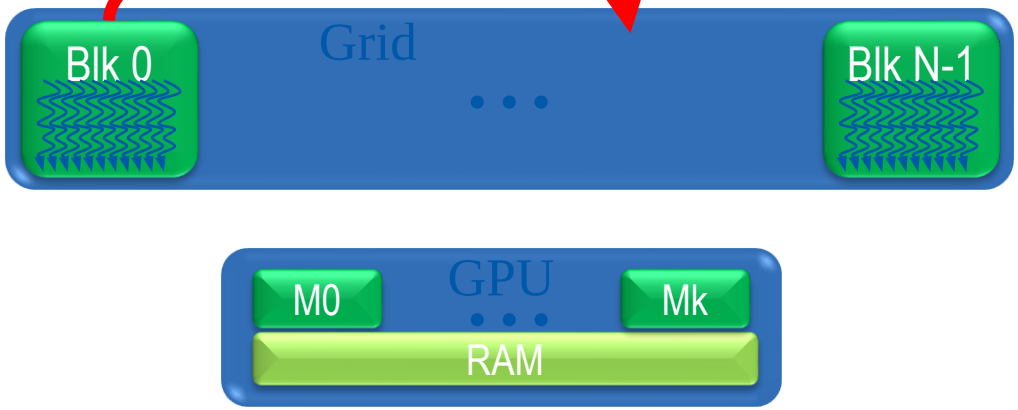
This is an equivalent way to express the ceiling function.

Kernel execution in a nutshell

```
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];
}
```



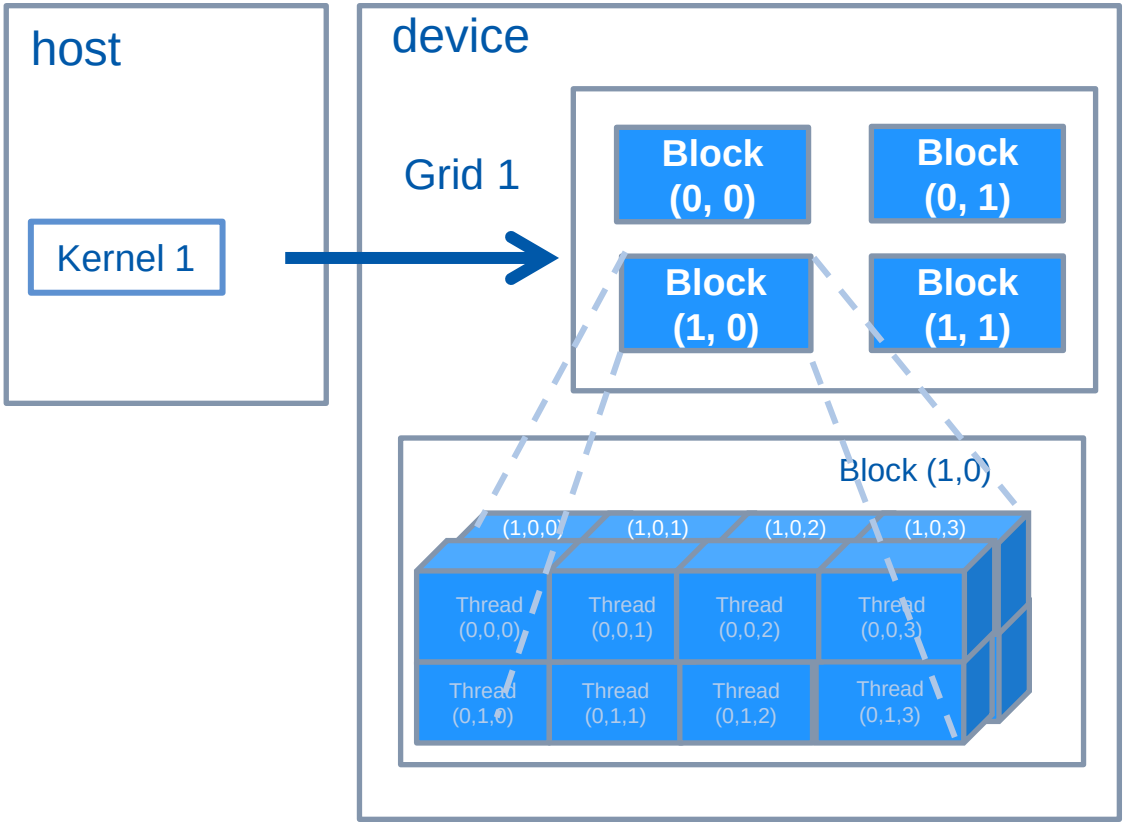
More on CUDA Function Declarations

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

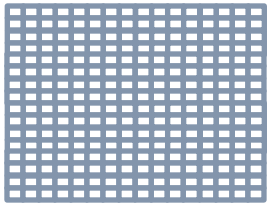
- `__global__` defines a kernel function
  - Each "__" consists of **two underscore** characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

MULTIDIMENSIONAL KERNEL CONFIGURATION

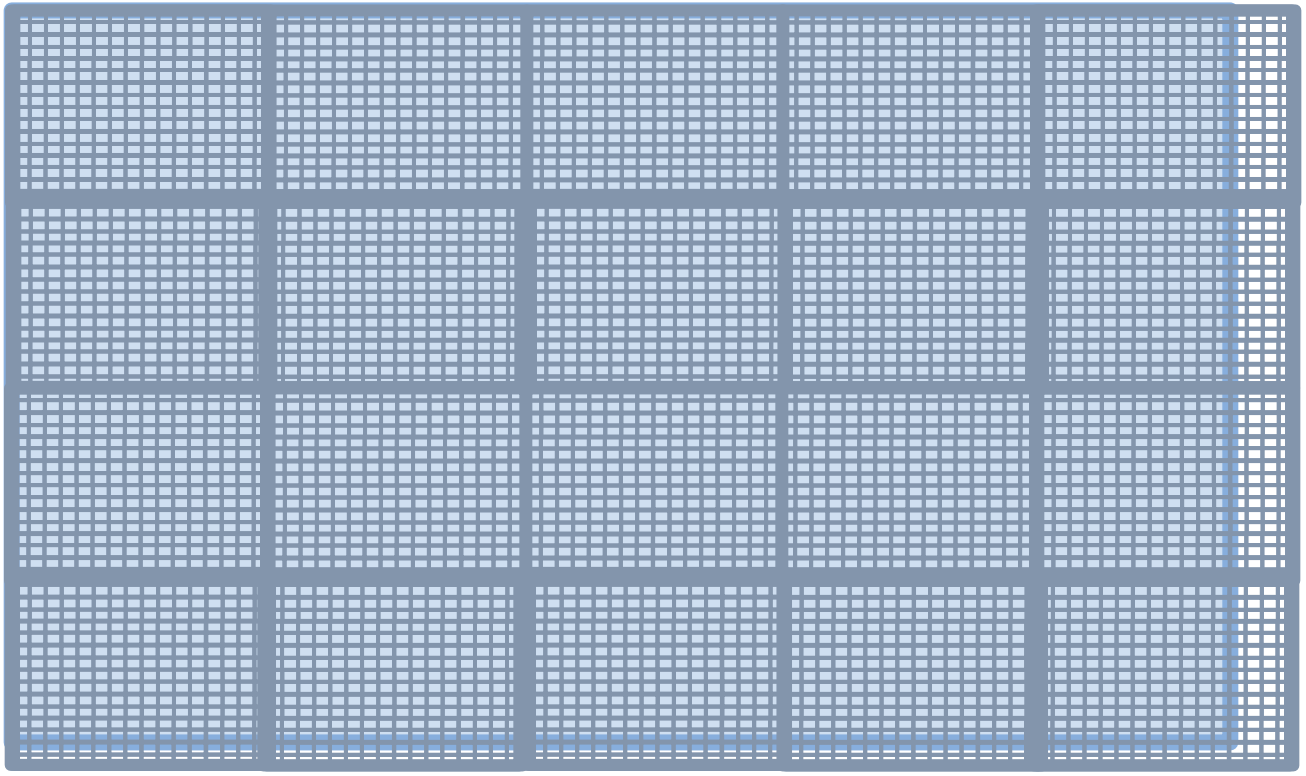
A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid



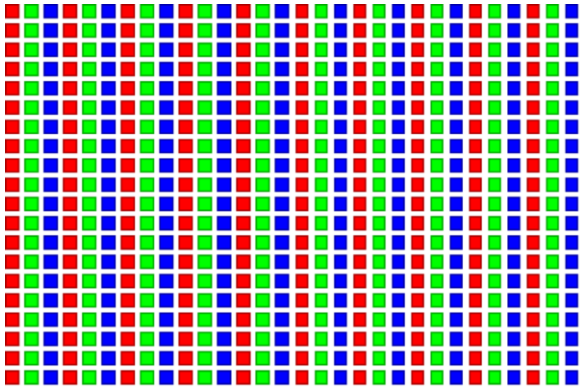
16×16 blocks



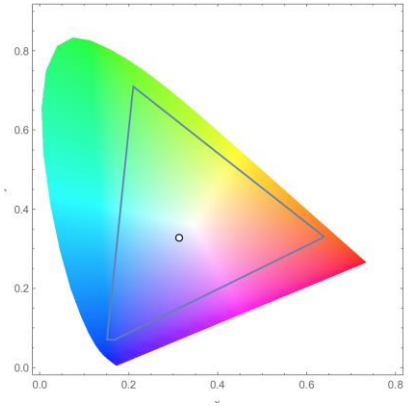
62×76 picture

RGB Color Image Representation

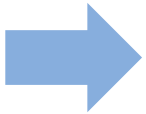
- Each pixel in an image is an RGB value



- 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



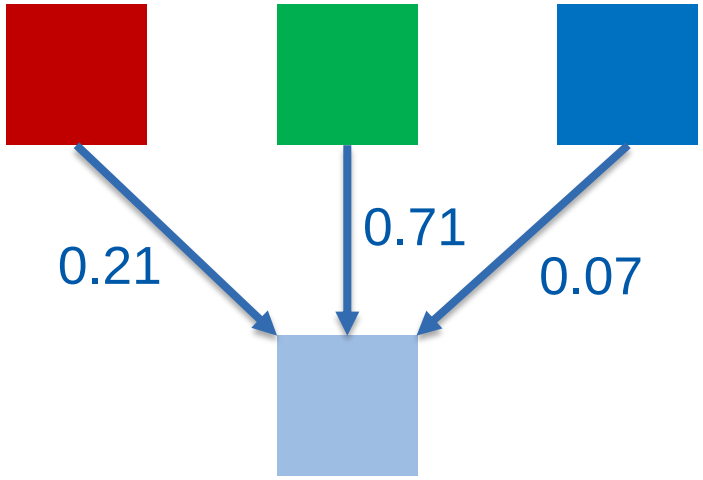
A grayscale digital image is an image in which the value of each pixel carries only intensity information.

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


 }
}
```

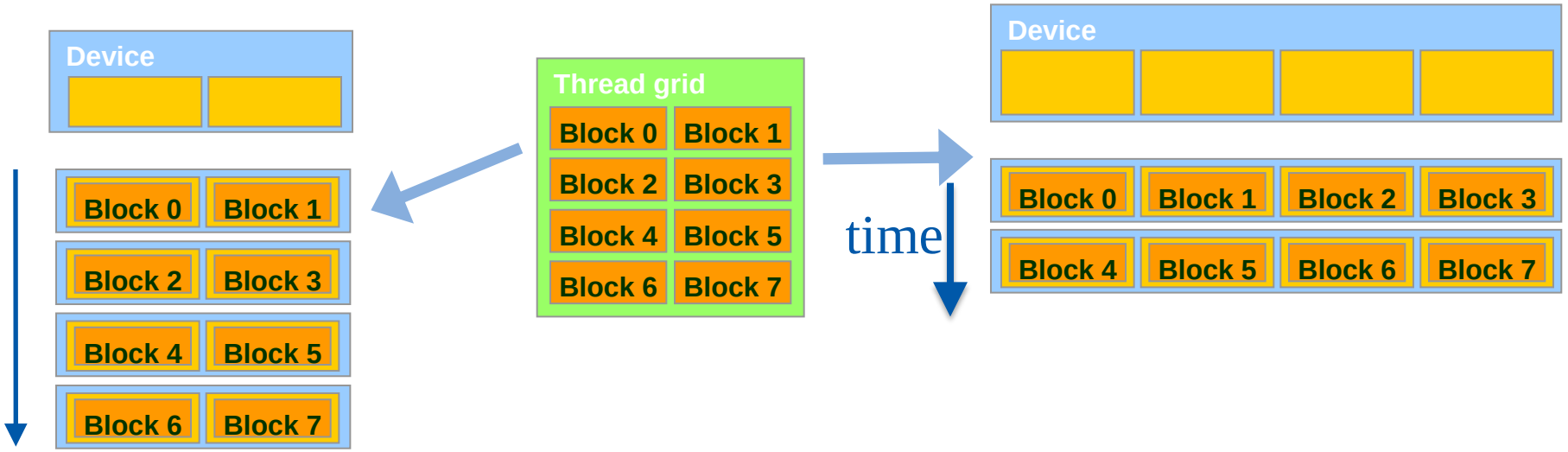
RGB to Grayscale Conversion Code

```
#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;
 }
}
```

THREAD SCHEDULING

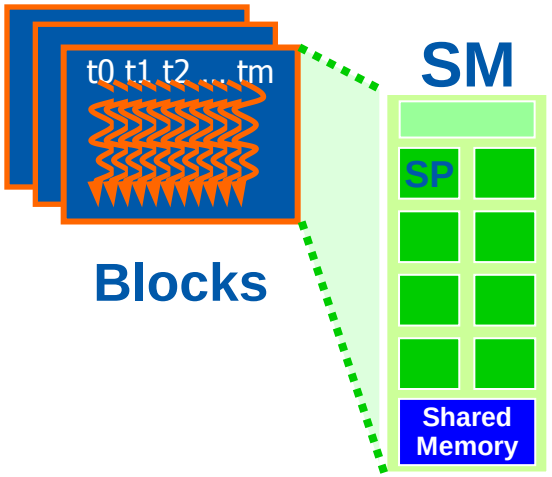
Transparent Scalability



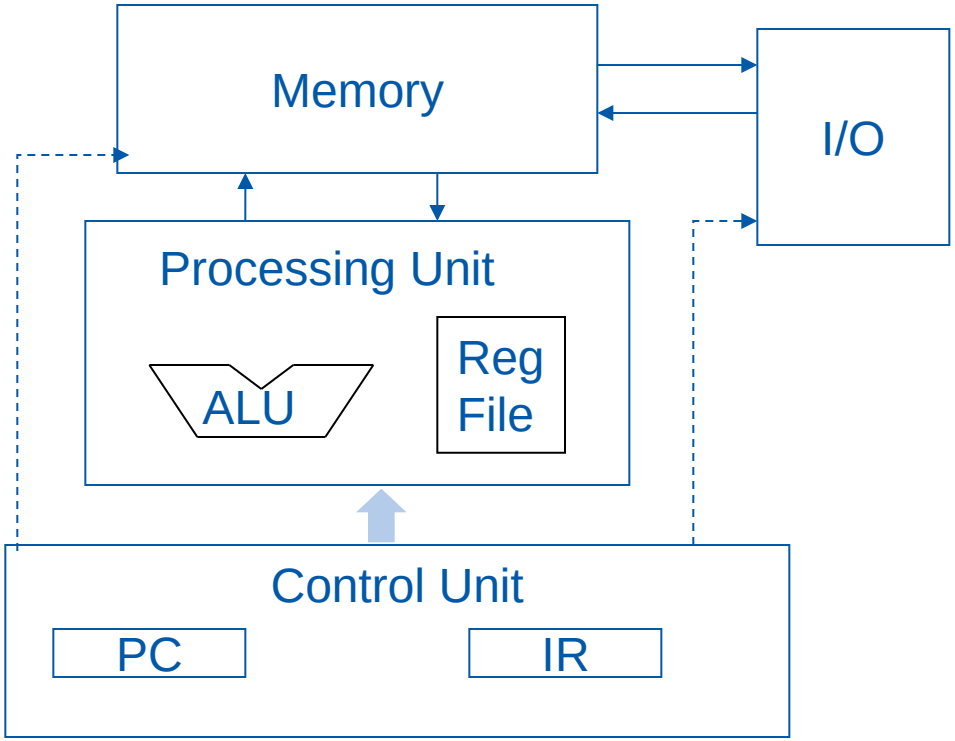
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors

Example: Executing Thread Blocks

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

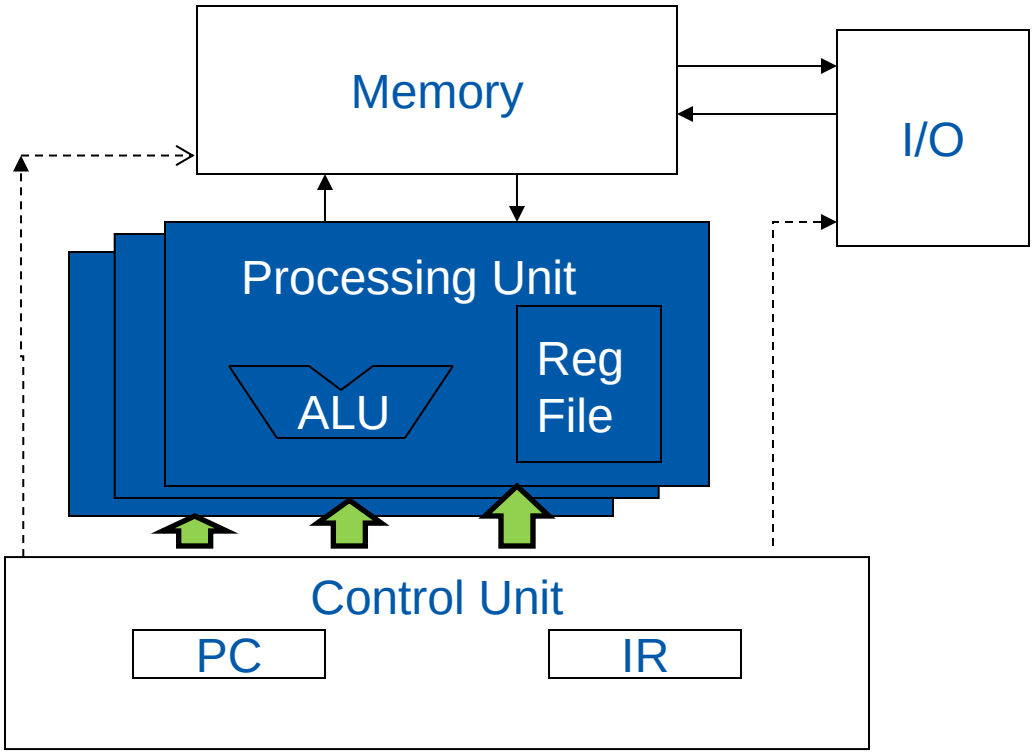


The Von-Neumann Model



Scalar arithmetic units

The Von-Neumann Model with SIMD units



Single Instruction Multiple Data (SIMD), aka vector arithmetic units

Warps as Scheduling Units

Each Block is executed as 32-thread **Warp**s

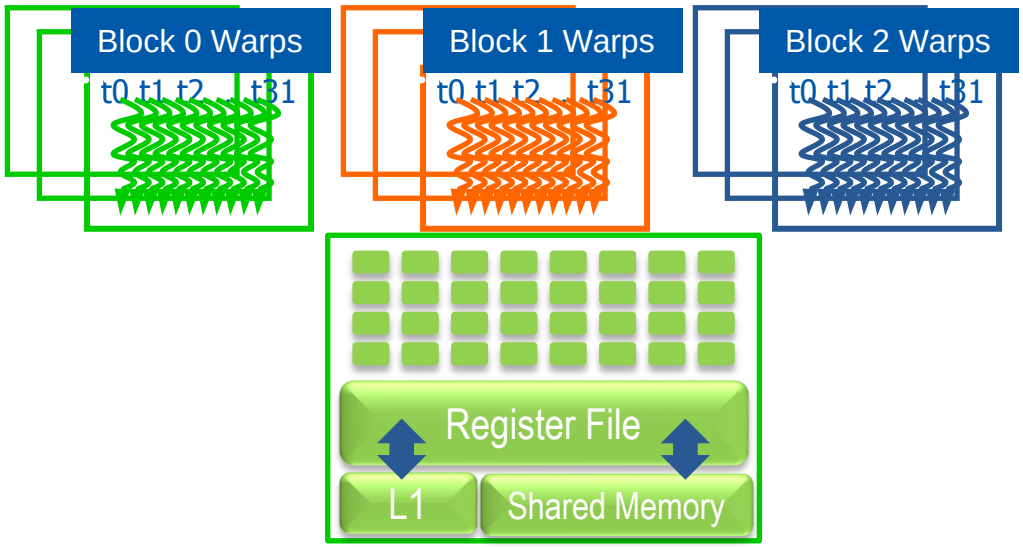
- An implementation decision, not part of the CUDA programming model
- Warps are scheduling units in SM
- Threads in a warp execute in SIMD
- Future GPUs may have different number of threads in each warp

Having **Warp**s into account matters in terms of performance

- Threads in a warp “will wait” for other threads to finish their work because of the SIMD nature

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 I use 8X8, 16X16 or 32X32 blocks?

- 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. Using only 2/3 of the thread capacity of an SM.


BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
QUIZ

Question 1

- If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index:

a) i = threadIdx.x + threadIdx.y;
b) i = blockIdx.x + threadIdx.x;
c) i = blockIdx.x * blockDim.x + threadIdx.x;
d) i = blockIdx.x * threadIdx.x;

Question 1 - Answer

- If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index:

a) i = threadIdx.x + threadIdx.y;
b) i = blockIdx.x + threadIdx.x;
c) i = blockIdx.x * blockDim.x + threadIdx.x;
d) i = blockIdx.x * threadIdx.x;

Question 2

- We want to use each thread to calculate two (adjacent) output elements of a vector addition. Assume that variable *i* should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

a) i = blockIdx.x * blockDim.x + threadIdx.x + 2
b) i = blockIdx.x * threadIdx.x * 2
c) i = (blockIdx.x * blockDim.x + threadIdx.x) * 2
d) i = blockIdx.x * blockDim.x * 2 + threadIdx.x

Question 2 - Answer

- We want to use each thread to calculate two (adjacent) output elements of a vector addition. Assume that variable *i* should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

a) i = blockIdx.x * blockDim.x + threadIdx.x + 2
b) i = blockIdx.x * threadIdx.x * 2
c) i = (blockIdx.x * blockDim.x + threadIdx.x) * 2
d) i = blockIdx.x * blockDim.x * 2 + threadIdx.x

Explanation: Every thread covers two adjacent output elements. The starting data index is simply twice the global thread index. Another way to look at it is that all previous blocks cover (blockIdx.x*blockDim.x)*2. Within the block, each thread covers 2 elements so the beginning position for a thread is threadIdx.x*2.

Question 3

- We want to use each thread to calculate two output elements of a vector addition. Each thread block processes 2*blockDim.x consecutive elements that form two sections. All threads in each block will first process a section, each processing one element. They will then all move to the next section, again each processing one element. Assume that variable *i* should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

a) i = blockIdx.x * blockDim.x + threadIdx.x + 2
b) i = blockIdx.x * threadIdx.x * 2
c) i = (blockIdx.x * blockDim.x + threadIdx.x) * 2
d) i = blockIdx.x * blockDim.x * 2 + threadIdx.x

Question 3 - Answer

- We want to use each thread to calculate two output elements of a vector addition. Each thread block processes 2*blockDim.x consecutive elements that form two sections. All threads in each block will first process a section, each processing one element. They will then all move to the next section, again each processing one element. Assume that variable *i* should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

a) i = blockIdx.x * blockDim.x + threadIdx.x + 2
b) i = blockIdx.x * threadIdx.x * 2
c) i = (blockIdx.x * blockDim.x + threadIdx.x) * 2
d) i = blockIdx.x * blockDim.x * 2 + threadIdx.x

Explanation: Each previous block covers (blockIdx.x*blockDim.x)*2. The beginning elements of the threads are consecutive in this case so just add threadIdx.x to it.

Question 4

- For a vector addition, assume that the vector length is 8,000, each thread calculates one output element, and the thread block size is 1,024 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?

a) 8,000
b) 8,196
c) 8,192
d) 8,200

Question 4 - Answer

- For a vector addition, assume that the vector length is 8,000, each thread calculates one output element, and the thread block size is 1,024 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?

a) 8,000
b) 8,196
c) 8,192
d) 8,200

Explanation: ceil(8000/1024)*1024 = 8 * 1024 = 8192. Another way to look at it is the minimal multiple of 1,024 to cover 8,000 is 1024*8 = 8192.

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
marc.jorda@bsc.es, antonio.pena@bsc.es