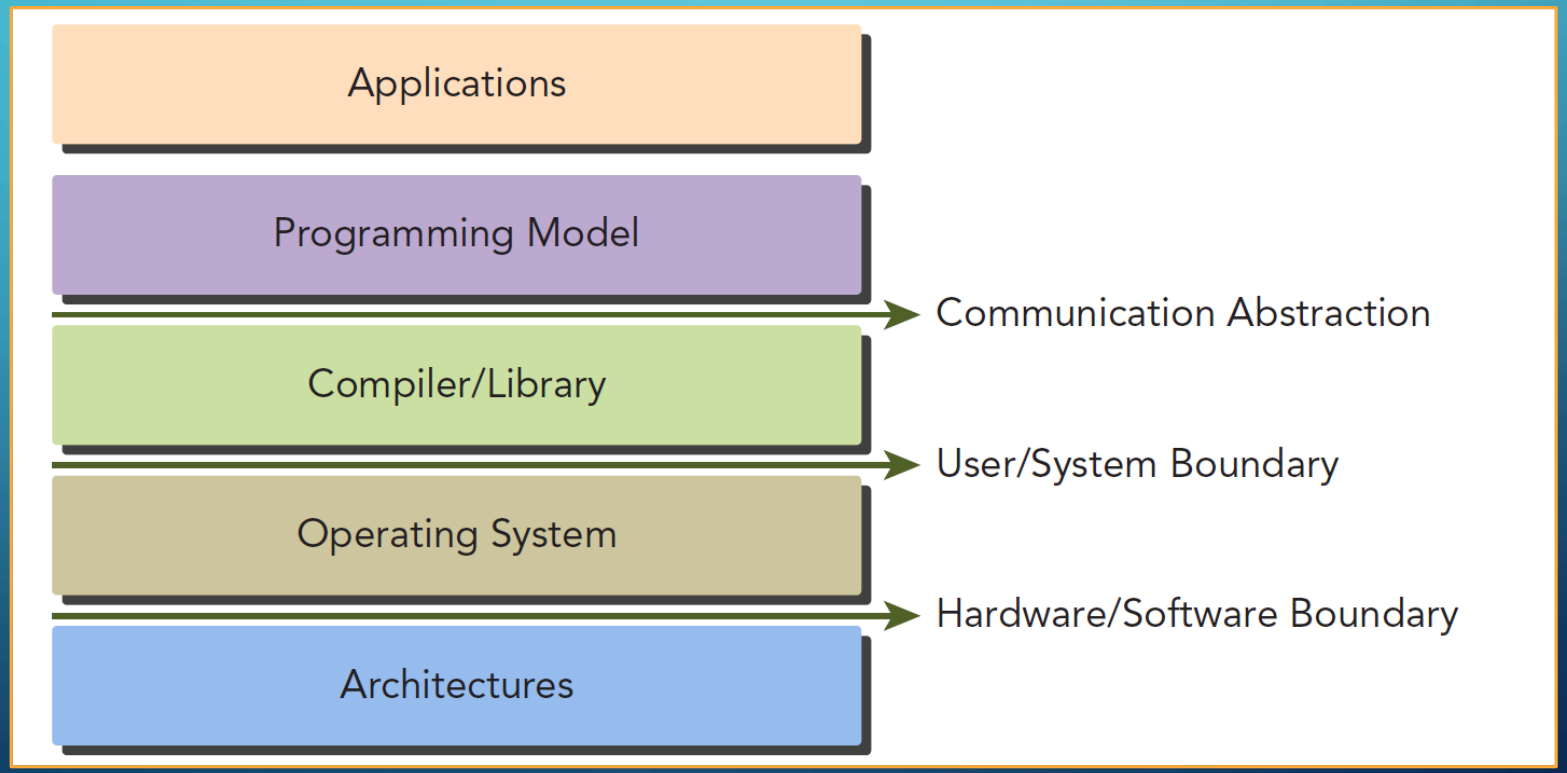
CUDA PROGRAMMING MODEL

CUDA PROGRAMMING STRUCTURE

- Programming models present an abstraction of computer architectures that act as a bridge between an application and its implementation on available hardware.



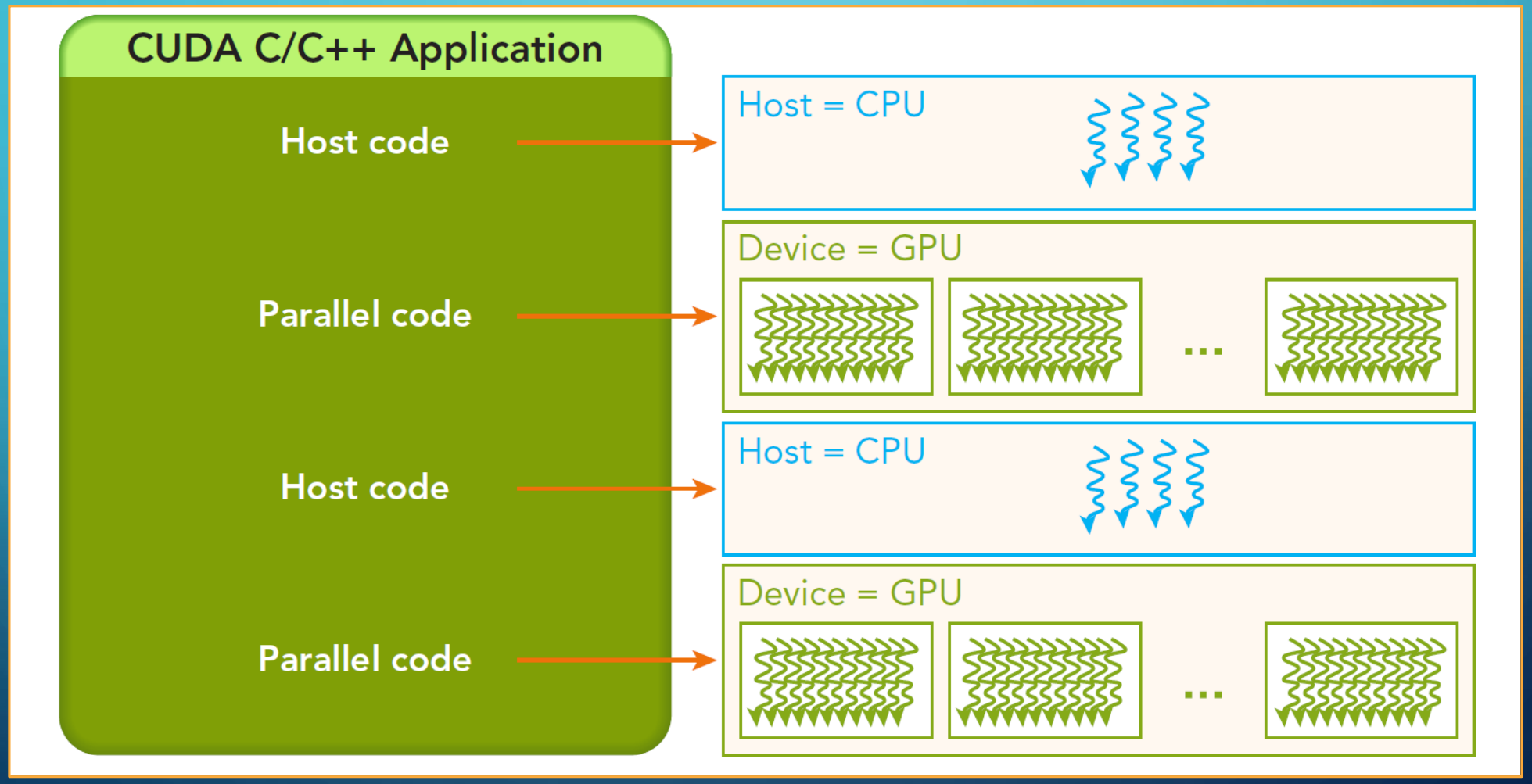
CUDA PROGRAMMING STRUCTURE

- The CUDA programming model provides the following special features to harness the computing power of GPU architectures.
  - A way to organize threads on the GPU through a hierarchy structure
  - A way to access memory on the GPU through a hierarchy structure
- A key component of the CUDA programming model is the kernel — the code that runs on the GPU device.
- As the developer, you can express a kernel as a sequential program.
- Behind the scenes, CUDA manages scheduling programmer-written kernels on GPU threads.

CUDA PROGRAMMING STRUCTURE

- The host can operate independently of the device for most operations.
- When a kernel has been launched, control is returned immediately to the host, freeing the CPU to perform additional tasks complemented by data parallel code running on the device.
- The CUDA programming model is primarily asynchronous so that GPU computation performed on the GPU can be overlapped with host-device communication.
- A typical CUDA program consists of serial code complemented by parallel code.
- A typical processing flow of a CUDA program follows this pattern:
  - Copy data from CPU memory to GPU memory.
  - Invoke kernels to operate on the data stored in GPU memory.
  - Copy data back from GPU memory to CPU memory.

CUDA PROGRAMMING STRUCTURE



MANAGING MEMORY

- The CUDA programming model assumes a system composed of a host and a device, each with its own separate memory.
- To allow you to have full control and achieve the best performance, the CUDA runtime provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory.

STANDARD C FUNCTIONS	CUDA C FUNCTIONS
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

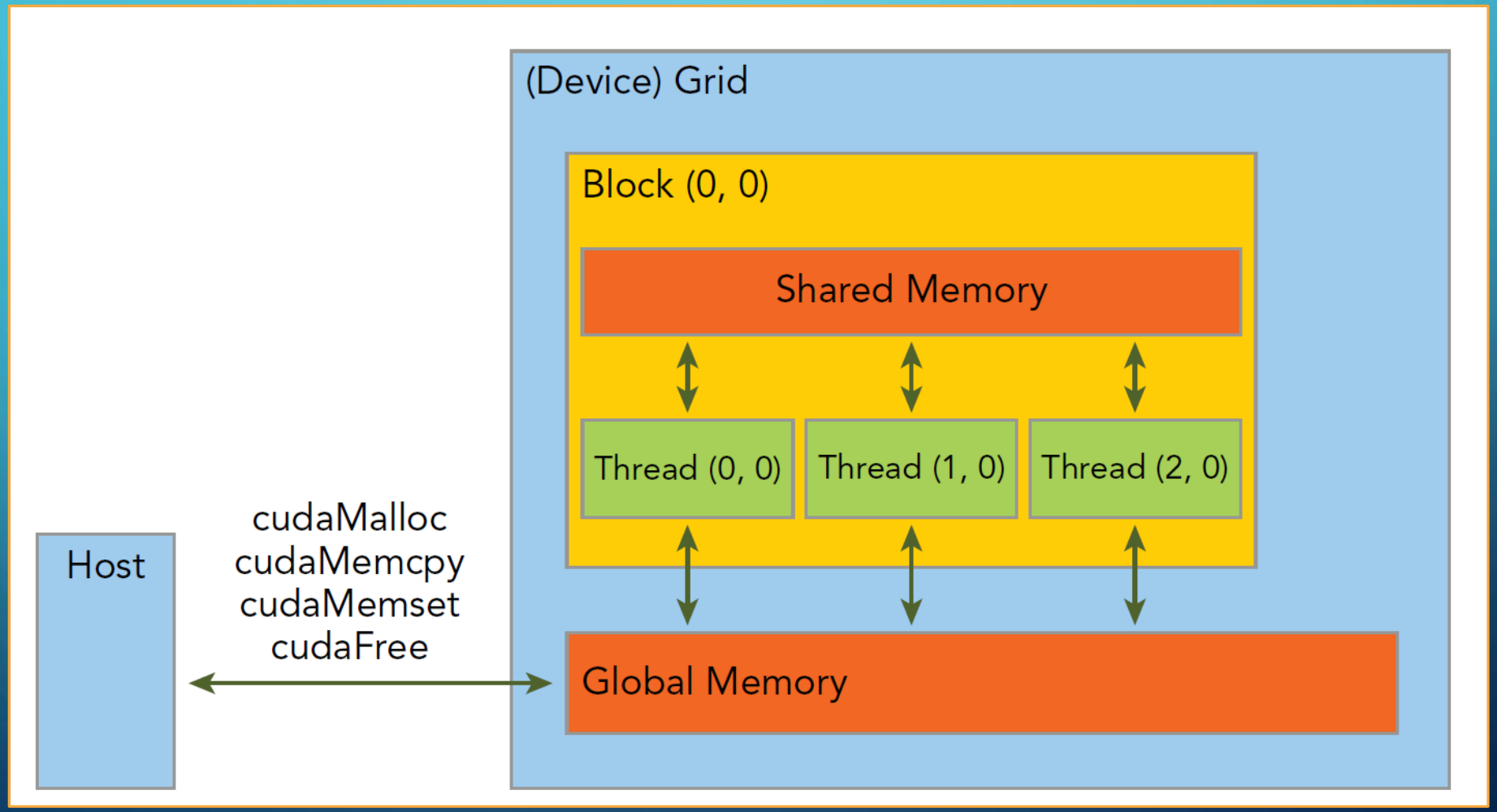
MANAGING MEMORY

- The function used to transfer data between the host and device is: cudaMemcpy, and its function signature is:

  cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)

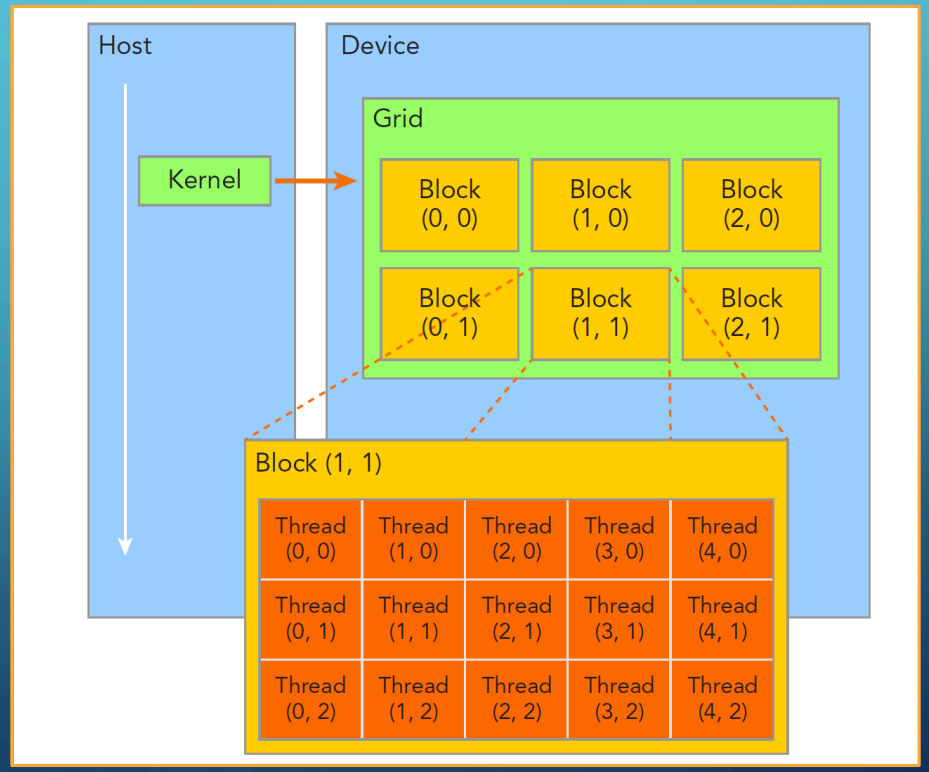
- This function copies the specified bytes from the source memory area, pointed to by **src**, to the destination memory area, pointed to by **dst**, with the direction specified by **kind**, where kind takes one of the following types:
  - **cudaMemcpyHostToHost**
  - **cudaMemcpyHostToDevice**
  - **cudaMemcpyDeviceToHost**
  - **cudaMemcpyDeviceToDevice**

MANAGING MEMORY



ORGANIZING THREADS

- When a kernel function is launched from the host side, execution is moved to a device where a large number of threads are generated and each thread executes the statements specified by the kernel function. CUDA exposes a thread hierarchy abstraction to enable you to organize your threads.
- This is a two-level thread hierarchy decomposed into blocks of threads and grids of blocks.



ORGANIZING THREADS

- All threads spawned by a single kernel launch are collectively called a grid.
- All threads in a grid share the same global memory space.
- A grid is made up of many thread blocks.
- A thread block is a group of threads that can cooperate with each other.
- Threads from different blocks cannot cooperate.
- Threads rely on the following two unique coordinates to distinguish themselves from each other:
  - **blockIdx (block index within a grid)**
  - **threadIdx (thread index within a block)**
- These variables appear as built-in, pre-initialized variables that can be accessed within kernel functions.
- When a kernel function is executed, the coordinate variables blockIdx and threadIdx are assigned to each thread by the CUDA runtime.
- Based on the coordinates, you can assign portions of data to different threads.

ORGANIZING THREADS

- The coordinate variable is of type uint3, a CUDA built-in vector type, derived from the basic integer type. It is a structure containing three unsigned integers, and the 1st, 2nd, and 3rd components are accessible through the fields x, y, and z respectively.
  - **blockIdx.x**
  - **blockIdx.y**
  - **blockIdx.z**
  - **threadIdx.x**
  - **threadIdx.y**
  - **threadIdx.z**
- The dimensions of a grid and a block are specified by the following two built-in variables:
  - **blockDim (block dimension, measured in threads)**
  - **gridDim (grid dimension, measured in blocks)**

ORGANIZING THREADS

- These variables are of type dim3, an integer vector type based on uint3 that is used to specify dimensions.
- When defining a variable of type dim3, any component left unspecified is initialized to 1.
- Each component in a variable of type dim3 is accessible through its x, y, and z fields, respectively, as shown in the following example:
  - **blockDim.x**
  - **blockDim.y**
  - **blockDim.z**

ORGANIZING THREADS

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

__global__ void checkIndex(void) {
    printf("threadIdx:(%d, %d, %d)  blockIdx:(%d, %d, %d)  blockDim:(%d, %d, %d) "
        "gridDim:(%d, %d, %d)\n", threadIdx.x, threadIdx.y, threadIdx.z,
        blockIdx.x, blockIdx.y, blockIdx.z, blockDim.x, blockDim.y, blockDim.z,
        gridDim.x,gridDim.y,gridDim.z);
}

int main(int argc, char **argv) {
   // define total data element
   int nElem = 6;

   // define grid and block structure
   dim3 block (3);
   dim3 grid  ((nElem+block.x-1)/block.x);

   // check grid and block dimension from host side
   printf("grid.x %d grid.y %d grid.z %d\n",grid.x, grid.y, grid.z);
   printf("block.x %d block.y %d block.z %d\n",block.x, block.y, block.z);

   // check grid and block dimension from device side
   checkIndex <<<grid, block>>> ();

   // reset device before you leave
   cudaDeviceReset();

   return(0);
}
```

ORGANIZING THREADS

```
grid.x 2 grid.y 1 grid.z 1
block.x 3 block.y 1 block.z 1
threadIdx:(0, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(0, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

WRITING YOUR KERNEL

- A kernel function is the code to be executed on the device side.
- In a kernel function, you define the computation for a single thread, and the data access for that thread.
- When the kernel is called, many different CUDA threads perform the same computation in parallel.
- A kernel is defined using the __global__ declaration specification as shown:

  __global__ void kernel_name(argument list);

- A kernel function must have a void return type.
- Function type qualifiers specify whether a function executes on the host or on the device and whether it is callable from the host or from the device.

QUALIFIERS	EXECUTION	CALLABLE	NOTES
`__global__`	Executed on the device	Callable from the host Callable from the device for devices of compute capability 3	Must have a `void` return type
`__device__`	Executed on the device	Callable from the device only	
`__host__`	Executed on the host	Callable from the host only	Can be omitted

WRITING YOUR KERNEL

CUDA KERNELS ARE FUNCTIONS WITH RESTRICTIONS

The following restrictions apply for all kernels:

- Access to device memory only
- Must have `void` return type
- No support for a variable number of arguments
- No support for static variables
- No support for function pointers
- Exhibit an asynchronous behavior

WRITING YOUR KERNEL

- The C code for vector addition on the host is given below:

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
   for (int i = 0; i < N; i++)
      C[i] = A[i] + B[i];
}
```

- The C code for vector addition on the device is given below:

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C) {
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
}
```

- Supposing a vector with the length of 32 elements, you can invoke the kernel with 32 threads as follows:

```
sumArraysOnGPU<<<1,32>>>(float *A, float *B, float *C);
```

WRITING YOUR KERNEL

- For the general case, you can calculate the unique index of global data access for a given thread based on the 1D grid and block information as follows:

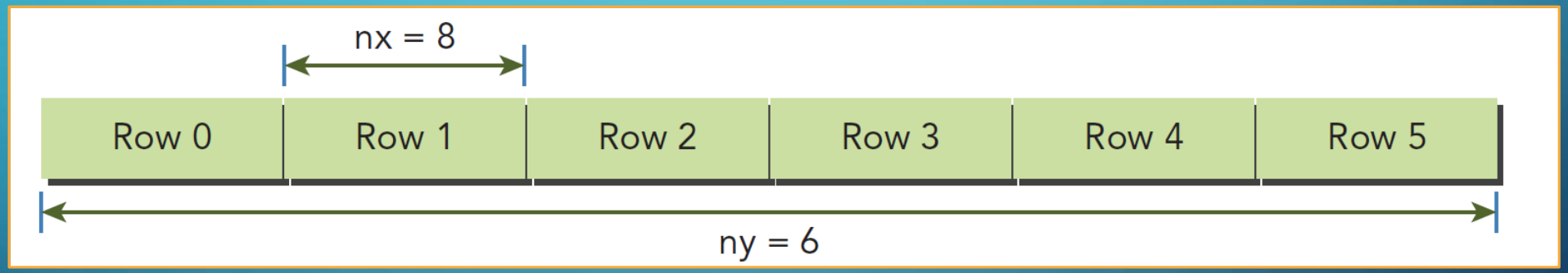
```
__global__ void sumArraysOnGPU(float *A, float *B, float *C) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        C[i] = A[i] + B[i];
}
```

ORGANIZING PARALLEL THREADS

- If threads are organized using the right grid and block size properly, it can make a big impact on kernel performance.
- For matrix operations, a natural approach is to use a layout that contains a 2D grid with 2D blocks to organize the threads in your kernel.
- You will see that a naive approach will not yield the best performance.
- You are going to learn more about grid and block heuristics using the following layouts for matrix addition:
  - **2D grid with 2D blocks**
  - **1D grid with 1D blocks**
  - **2D grid with 1D blocks**

INDEXING MATRICES WITH BLOCKS AND THREADS

- Typically, a matrix is stored linearly in global memory with a row-major approach.
- The figure illustrates a small case for an 8 x 6 matrix.



INDEXING MATRICES WITH BLOCKS AND THREADS

- In a matrix addition kernel, a thread is usually assigned one data element to process.
- Accessing the assigned data from global memory using block and thread index is the first issue you need to solve.
- In the first step, you can map the thread and block index to the coordinate of a matrix with the following formula:

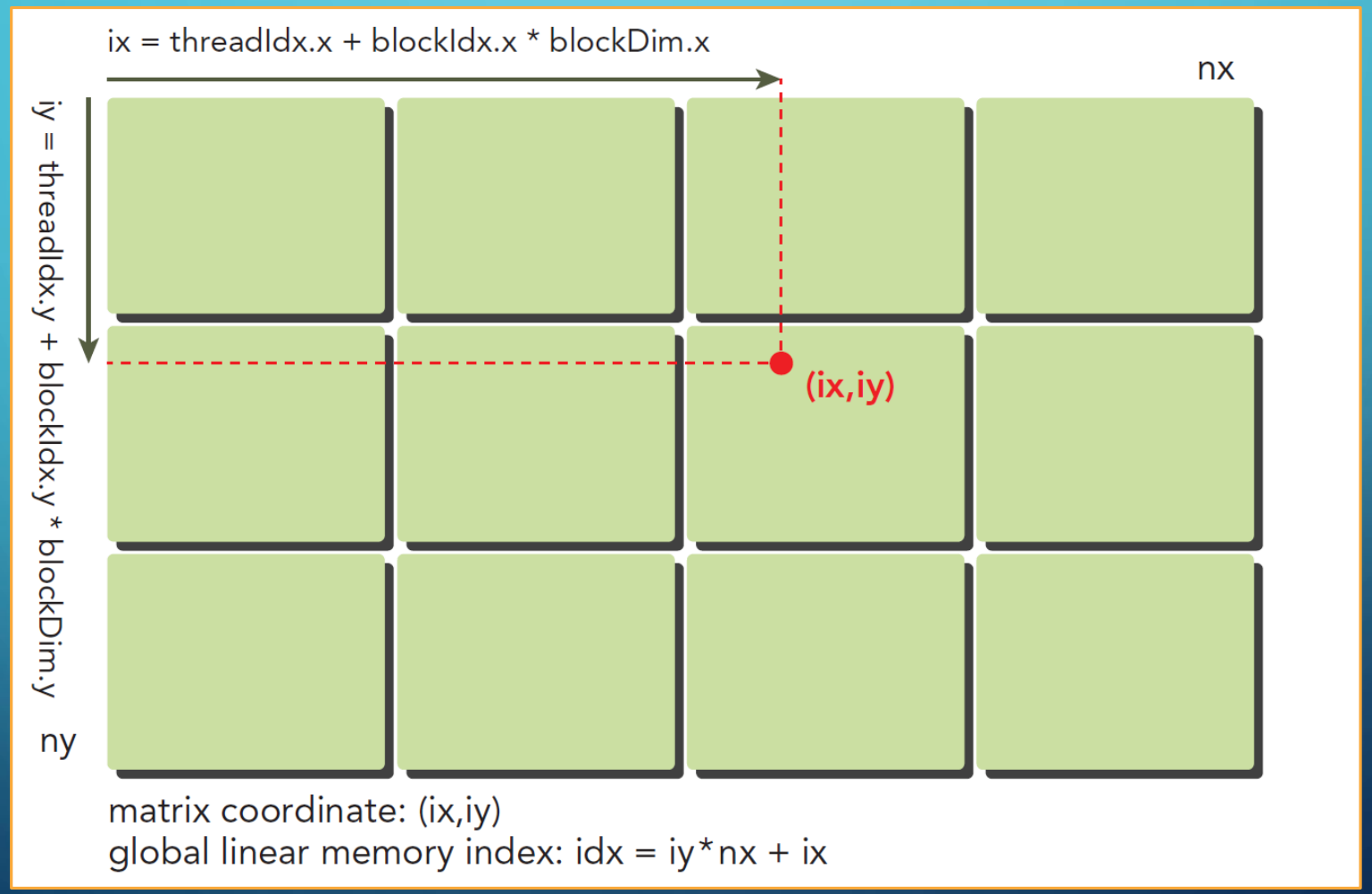
  ix = threadIdx.x + blockIdx.x * blockDim.x

  iy = threadIdx.y + blockIdx.y * blockDim.y

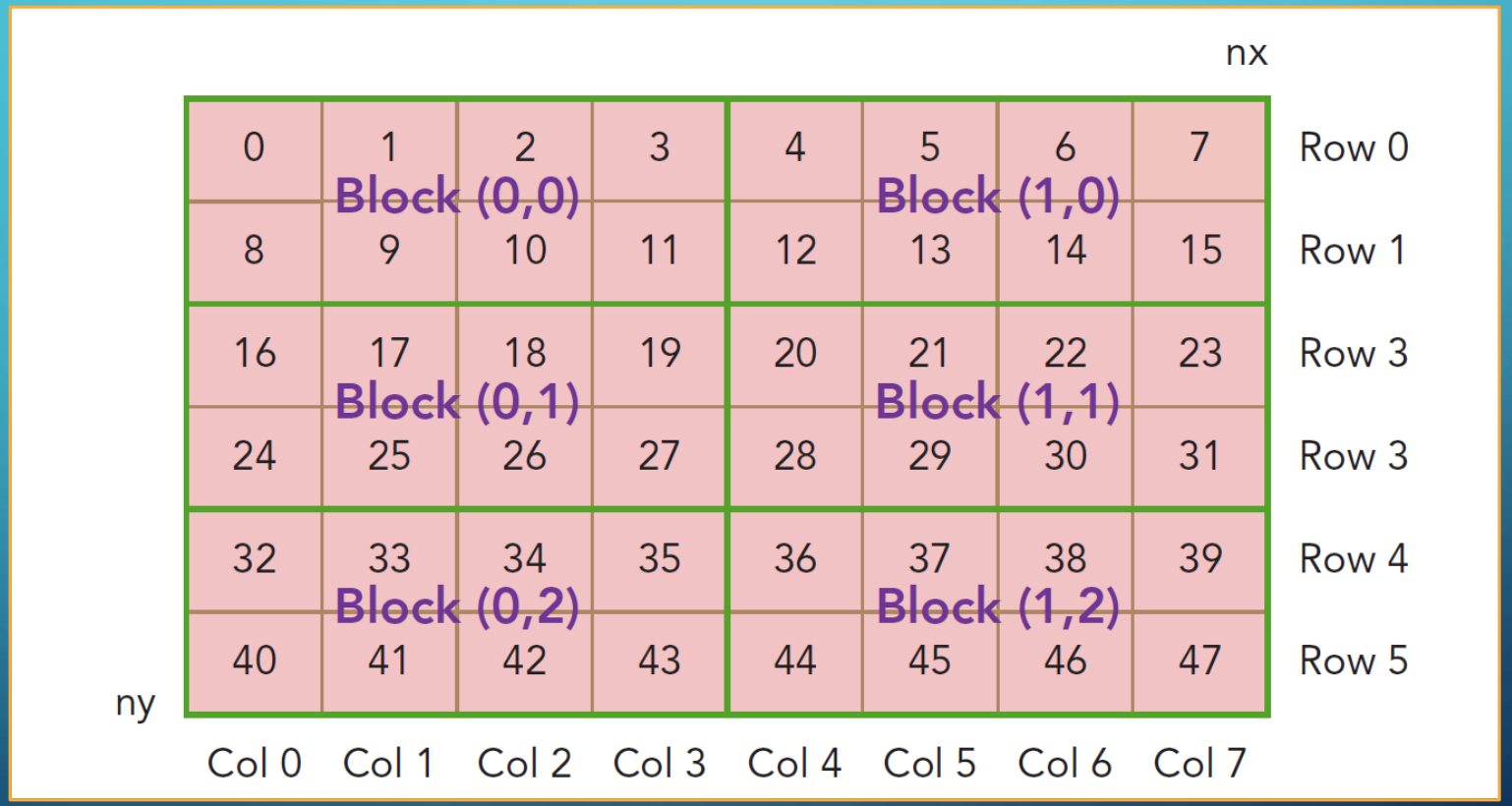
- In the second step, you can map a matrix coordinate to a global memory location/index with the following formula:

  idx = iy * nx + ix

SUMMING MATRICES WITH A 2D GRID AND 2D BLOCKS



SUMMING MATRICES WITH A 2D GRID AND 2D BLOCKS



SUMMING MATRICES WITH A 2D GRID AND 2D BLOCKS

- Kernel to sum the matrix with a 2D thread block:

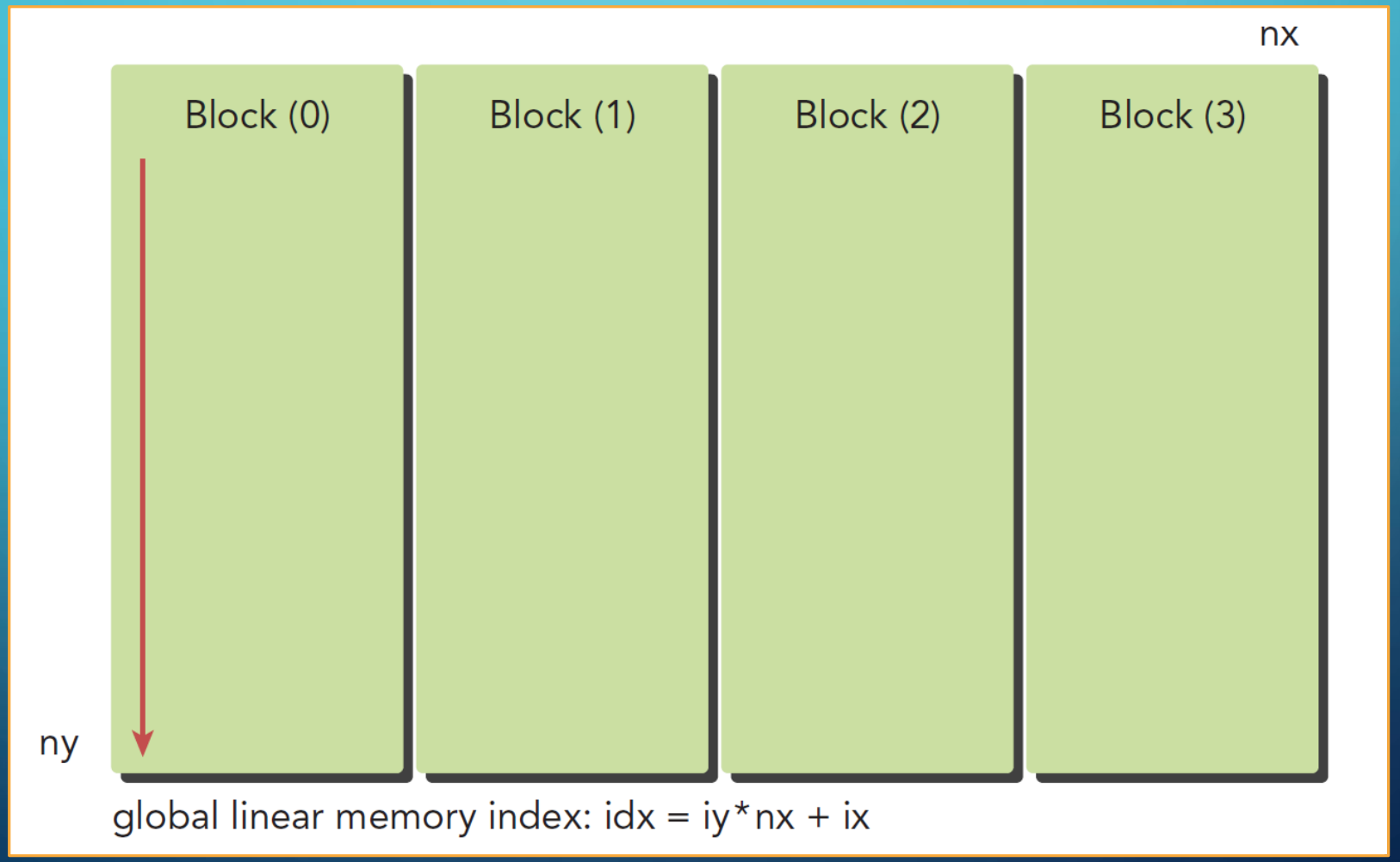
```
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC,
      int nx, int ny) {
   unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
   unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
   unsigned int idx = iy*nx + ix;

   if (ix < nx && iy < ny)
      MatC[idx] = MatA[idx] + MatB[idx];
}
```

SUMMING MATRICES WITH A 2D GRID AND 2D BLOCKS

KERNEL CONFIGURATION	KERNEL ELAPSED TIME	BLOCK NUMBER
(32,32)	0.060323 sec	512 x 512
(32,16)	0.038041 sec	512 x 1024
(16,16)	0.045535 sec	1024 x 1024

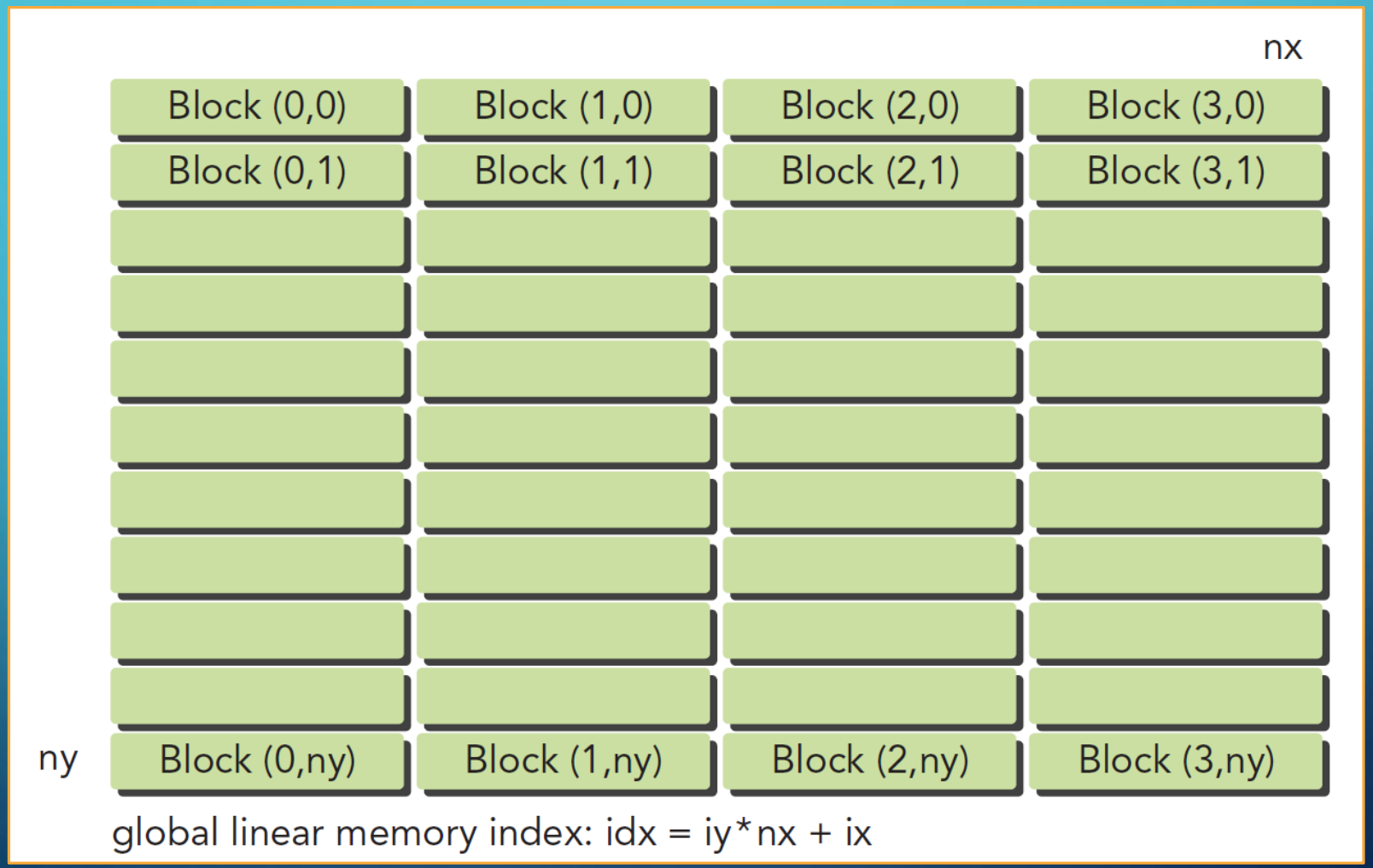
SUMMING MATRICES WITH A 1D GRID AND 1D BLOCKS



SUMMING MATRICES WITH A 1D GRID AND 1D BLOCKS

```
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC,
      int nx, int ny) {
   unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
   if (ix < nx ) {
      for (int iy=0; iy<ny; iy++) {
         int idx = iy*nx + ix;
         MatC[idx] = MatA[idx] + MatB[idx];
      }
   }
}
```

SUMMING MATRICES WITH A 2D GRID AND 1D BLOCKS



SUMMING MATRICES WITH A 2D GRID AND 1D BLOCKS

- This can be viewed as a special case of a 2D grid with 2D blocks, where the second dimension of the block is 1.
- Therefore, the mapping from block and thread index to the matrix coordinate becomes:

ix = threadIdx.x + blockIdx.x * blockDim.x;

iy = blockIdx.y;

```
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *MatC,
      int nx, int ny) {
   unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
   unsigned int iy = blockIdx.y;
   unsigned int idx = iy*nx + ix;

   if (ix < nx && iy < ny)
      MatC[idx] = MatA[idx] + MatB[idx];
}
```

RESULTS COMPARISON OF DIFFERENT KERNEL IMPLEMENTATIONS

KERNEL	EXECUTION CONFIGURE	TIME ELAPSED
`sumMatrixOnGPU2D`	(512,1024), (32,16)	0.038041
`sumMatrixOnGPU1D`	(128,1), (128,1)	0.044701
`sumMatrixOnGPUMix`	(64,16384), (256,1)	0.030765

SUMMING MATRICES WITH A 2D GRID AND 1D BLOCKS

- From the matrix addition examples, you can see several things:
  - Changing execution configurations affects performance.
  - A naive kernel implementation does not generally yield the best performance.
  - For a given kernel, trying different grid and block dimensions may yield better performance.

USING THE RUNTIME API TO QUERY GPU INFORMATION

```
$ nvcc checkDeviceInfor.cu -o checkDeviceInfor
$ ./checkDeviceInfor
```

```
./checkDeviceInfor Starting...
Detected 2 CUDA Capable device(s)
Device 0: "Tesla M2070"
  CUDA Driver Version / Runtime Version          5.5 / 5.5
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 5.25 MBytes (5636554752 bytes)
  GPU Clock rate:                                1147 MHz (1.15 GHz)
  Memory Clock rate:                             1566 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
```

REFERENCES

- Professional CUDA C Programming, by John Cheng, Max Grossman, Ty McKercher, Wrox