

GPU programming: CUDA

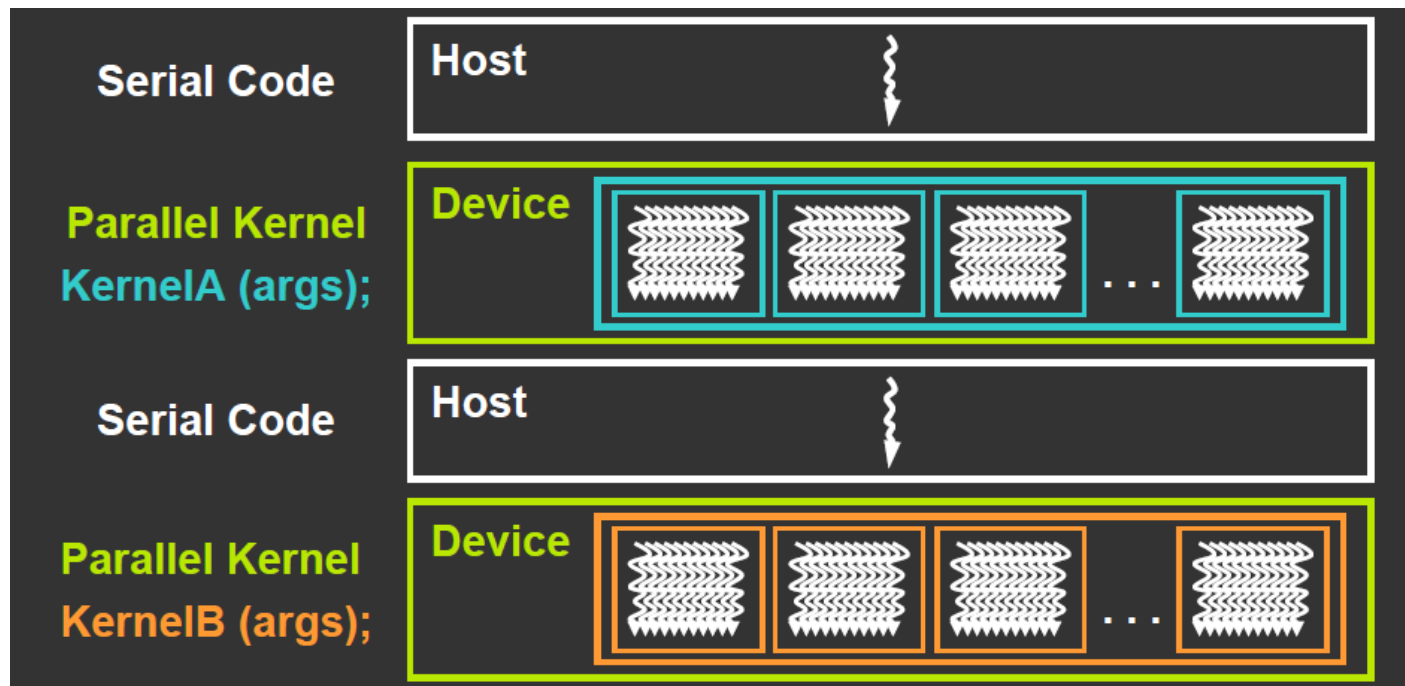
- Acknowledgement: the lecture materials are based on the materials in NVIDIA teaching center CUDA course materials, including materials from Wisconsin (Negrut), North Carolina Charlotte (Wikinson/Li) and NCSA (Kindratenko).

CUDA

- CUDA is Nvidia's scalable parallel programming model and a software environment for parallel computing
 - Language: CUDA C, minor extension to C/C++
 - Let the programmer focus on parallel algorithms not parallel programming mechanisms.
 - A heterogeneous serial-parallel programming model
 - Designed to program heterogeneous CPU+GPU systems
 - CPU and GPU are separate devices with separate memory

Heterogeneous programming with CUDA

- Fork-join model: CUDA program = serial code + parallel **kernels** (all in CUDA C)
 - Serial C code executes in a **host** thread (**CPU** thread)
 - Parallel kernel code executes in many **device** threads (**GPU** threads)

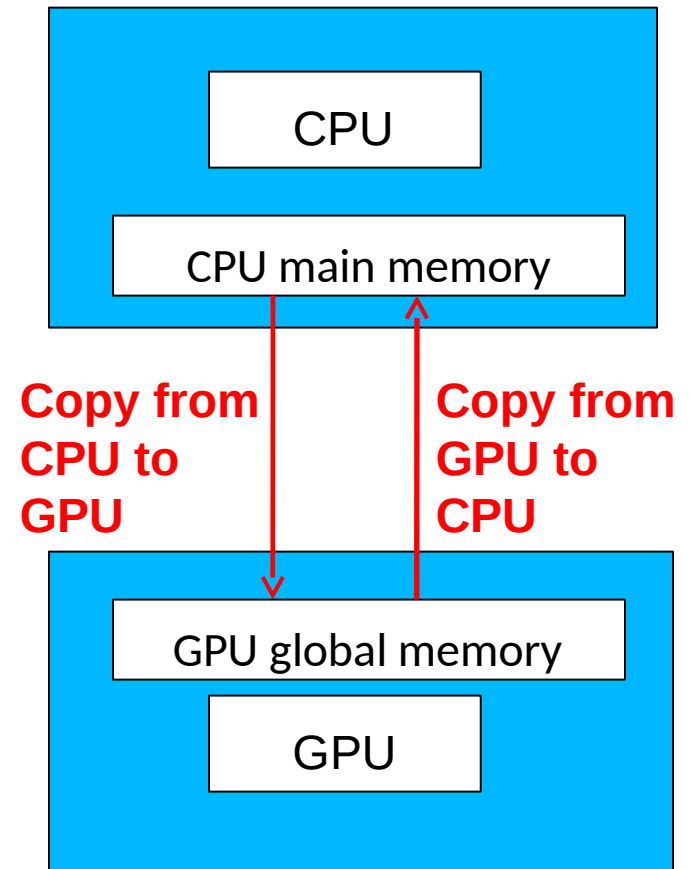


CUDA kernel

- Kernel code is regular C code except that it will use **thread ID** (CUDA built-in variables) to make different threads operate on different data
 - Also variables for the total number of threads
- When a kernel is reached in the code for the first time, it is launched onto GPU.

CPU and GPU memory

- CPU and GPU have different memories:
 - CPU memory is called host memory
 - GPU memory is called device memory
- Implication:
 - Explicitly transfer data from CPU to GPU for GPU computation, and
 - Explicitly transfer results in GPU memory copied back to CPU memory



Basic CUDA program structure

```
int main (int argc, char **argv ) {
```

1. Allocate memory space in device (GPU) for data
2. Allocate memory space in host (CPU) for data
3. Copy data to GPU
4. Call “kernel” routine to execute on GPU
(with CUDA syntax that defines no of threads and their physical structure)
5. Transfer results from GPU to CPU
6. Free memory space in device (GPU)
7. Free memory space in host (CPU)

```
return;
```

```
}
```

1. Allocating memory in GPU (device)

- The cudaMalloc routine:

```
int size = N * sizeof( int);    // space for N integers
int *devA, *devB, *devC;    // devA, devB, devC ptrs
cudaMalloc( (void**)&devA, size );
cudaMalloc( (void**)&devB, size );
cudaMalloc( (void**)&devC, size );
```

- 2. Allocating memory in host (CPU)?
 - The regular malloc routine

3. Transferring data from/to host (CPU) to/from device (GPU)

- CUDA routine `cudaMemcpy`

`cudaMemcpy(devA, &A, size, cudaMemcpyHostToDevice);`


`cudaMemcpy(devB, &B, size, cudaMemcpyHostToDevice);`

`DevA` and `devB` are pointers to destination in device (return from `cudaMalloc` and `A` and `B` are pointers to host data


3. Defining/invoking kernel routine

- Define: CUDA specifier **__global__**


```
#define N 256
```

```
__global__ void vecAdd(int *A, int *B, int *C) { // Kernel definition  
    int i = threadIdx.x;   
    C[i] = A[i] + B[i];  
}
```

Each thread performs one pair-wise addition:

```
int main() {  
    // allocate device memory &  
    // copy data to device  
    // device mem. ptrs devA,devB,devC  
    vecAdd<<<1, N>>>(devA,devB,devC);   
    ...  
}
```

Thread 0: devC[0] = devA[0] + devB[0];

 Thread 1: devC[1] = devA[1] + devB[1];

Thread 2: devC[2] = devA[2] + devB[2];

This is the fork-join statement in Cuda
Notice the devA/B/C are device memory pointer

CUDA kernel invocation

- <<<...>>> syntax (addition to C) for kernel calls:
myKernel<<< n, m >>>(arg1, ...);
- <<< ... >>> contains thread organization for this particular kernel call in two parameters, **n** and **m**:
 - **vecAdd<<<1, N>>>(devA,devB,devC): 1 dimension block with N threads**
 - Threads execute very efficiently on GPU: we can have fine-grain threads (a few statements)
 - More thread organization later
- **arg1, ... , --** arguments to routine **myKernel** typically pointers to device memory obtained previously from **cudaMalloc**.

5. Transferring data from device (GPU) to host (CPU)

- CUDA routine `cudaMemcpy`

```
cudaMemcpy( &C, dev_C, size, cudaMemcpyDeviceToHost);
```

- `dev_C` is a pointer in device memory and `C` is a pointer in host memory.

6. Free memory space

- In “device” (GPU) -- Use CUDA cudaFree routine:

```
cudaFree( dev_a);  
cudaFree( dev_b);  
cudaFree( dev_c);
```

- In (CPU) host (if CPU memory allocated with malloc) -- Use regular C free routine:

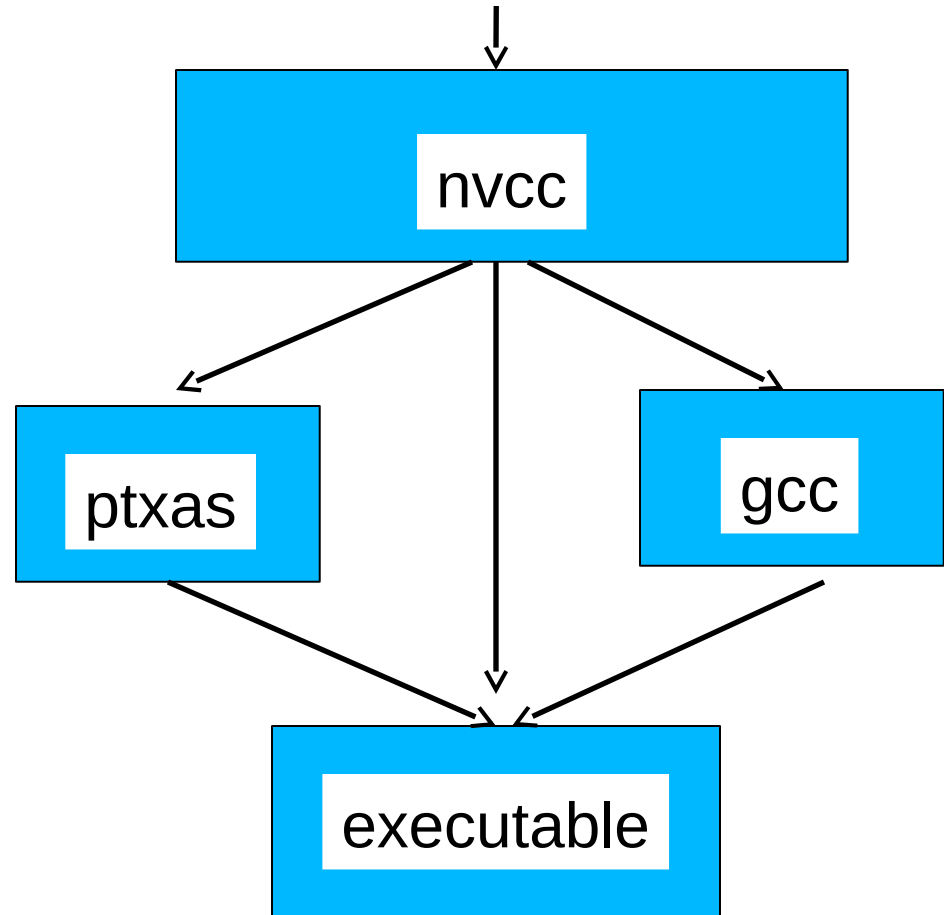
```
free( a );  
free( b );  
free( c );
```

Complete CUDA examples

- See `vecadd.cu`
- Compare the speed of `vecadd.c` and `vecadd.cu`
- See also `vec_complex.c` and `vec_complex.cu`
- Compiling CUDA programs
 - Use the `gpu.cs.fsu.edu` (gpu1, gpu2, gpu3)
 - Naming convention `.cu` programs are CUDA programs
 - NVIDIA CUDA compiler driver: `nvcc`
 - To compile `vecadd.cu`: `nvcc -O3 vecadd.cu`

Compilation process

- nvcc “wrapper” divides code into host and device parts.
- Host part compiled by regular C compiler
- Device part compiled by NVIDIA “ptxas” assembler
- Two compiled parts combined into one executable



Executable file a “fat” binary” with both host and device code

CUDA C extensions

- Declaration specifiers to indicate where things live

`__global__ void mykernel(...) // kernel function on GPU`

`__device__ int globalVar; // variable in device`

`__shared__ int sharedVar; // in per block shared memory`

- Parallel kernel launch

`Mykernel<<<500,128>>> (...); // launch 500 blocks with 128 threads each`

- Special variables

– Dim3 `threadIdx`, `blockIdx`; // thread/block ID

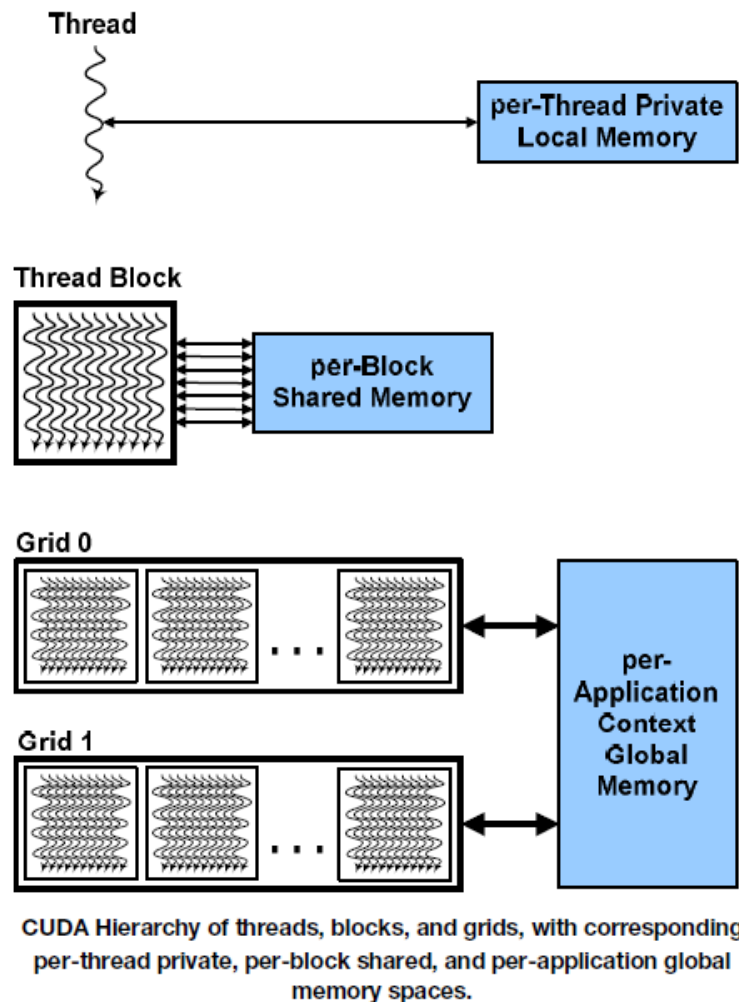
– Dim3 `blockDim`, `gridDim`; //thread/block size

- Intrinsic for specific operations in kernel

– `__syncthreads();` // barrier synchronization

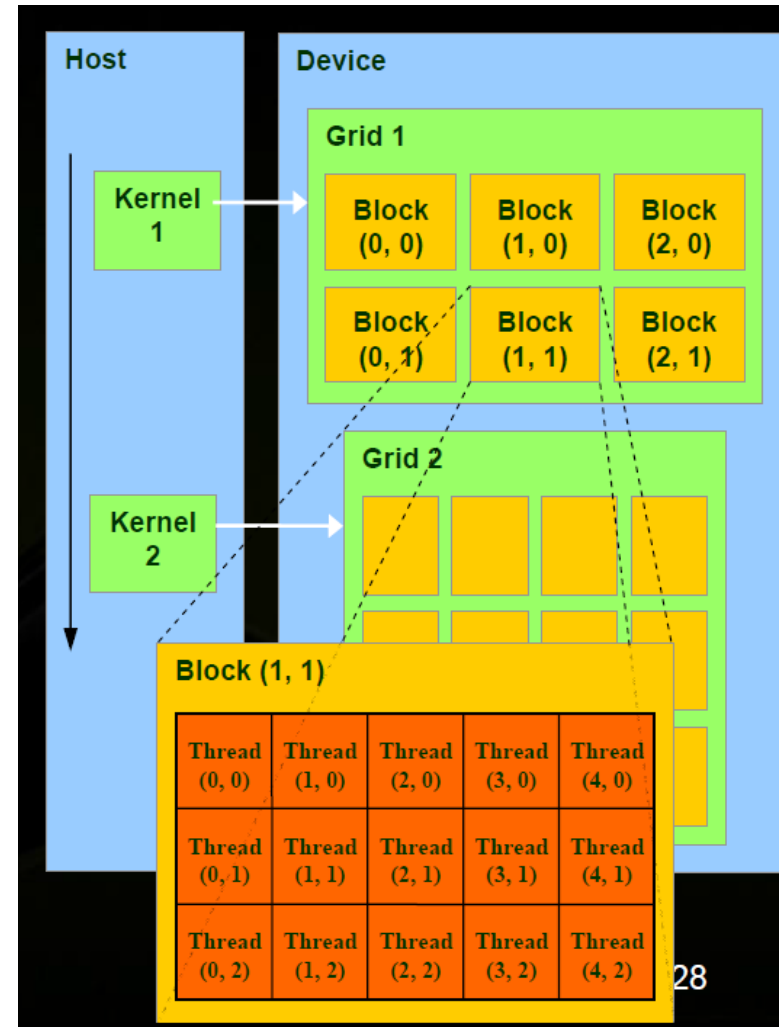
CUDA thread organization

- hierarchy of threads
 - Blocks of threads in 1 or 2 dimensions, the collection of block is called a **grid**.
 - Blocks can be 1D, 2D, or 3D.
 - Can easily deal with 1D, 2D, and 3D data arrays.



Cuda thread organization

- Threads and blocks have IDs
 - So each thread can decide what data to work on.
- Block ID (**blockIdx**):
1D or 2D
- Thread ID (**threadIdx**):
1D, 2D or 3D.



Device characteristics – hardware limitations

- NVIDIA defined “compute capabilities” 1.0, 1.1, ... with limits and features
 - Give the limits of threads per block, total number of blocks, etc.
- Compute capability 1.0
 - Max number of threads per block = 512
 - Max sizes of x- and y-dimension of thread block = 512
 - Maximum size of each dimension of grid of thread blocks = 65535

Specifying Grid/Block structure

- The programmer need to provide each kernel call with:
 - Number of blocks in each dimension
 - Threads per block in each dimension
 - **myKernel<<< B, T >>>(arg1, ...);**
- **B** – a structure that defines the number of blocks in grid in each dimension (1D or 2D).
- **T** – a structure that defines the number of threads in a block in each dimension (1D, 2D, or 3D).
- B and T are of type **dim3 (uint3)**.

1-D grid and/or 1-D blocks

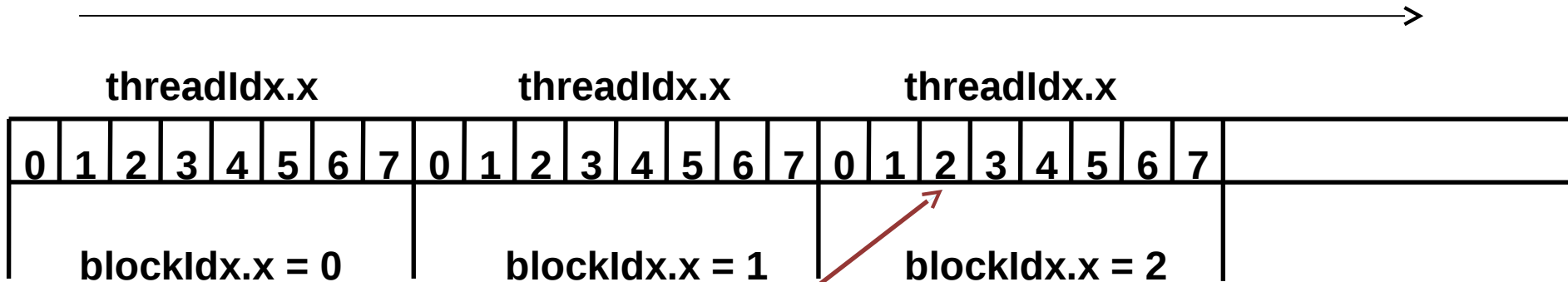
- For 1-D structure, one can use an integer for each of B and T in:
 - **myKernel<<< B, T >>>(arg1, ...);**
- **B** – *An integer would define a 1D grid of that size*
- **T** – *An integer would define a 1D block of that size*
 - **myKernel<<< 1, 100 >>>(arg1, ...);**
- Grids can be 2D and blocks can be 2D or 3D
 - struct **dim3** {x; y; z;} threadIdx, blockIdx;
- Grid/block size
 - Dim3 gridDim; size of grid dimension x, y (z not used)
 - Dim3 blockDim; - size of grid dimension,

Compute global 1-D thread ID

- dim3
- **threadIdx.x** -- “thread index” within block in “x” dimension
- **blockIdx.x** -- “block index” within grid in “x” dimension
- **blockDim.x** -- “block dimension” in “x” dimension (i.e. number of threads in a block in the x dimension)
- Full global thread ID in x dimension can be computed by:
$$x = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$$
- how to fix vecadd.cu to make it work for larger vectors? See vecadd1.cu. What is the right number of threads per block?

Compute global 1-D thread ID

Global ID 18



gridDim = 3 x 1

blockDim = 8 x 1

Global thread ID = **blockIdx.x** * **blockDim.x** + **threadIdx.x**

= 2 * 8 + 2 = thread 18 with linear global addressing

1D grid/block examples

```
__global__ void vecadd(float* A, float* B, float* C)
{ int i = threadIdx.x; // threadIdx is a CUDA built-in variable
  C[i] = A[i] + B[i];
}
Vecadd<<<1,n>>>( dev_A, dev_B, dev_C );
```

```
__global__ void vecadd(float* A, float* B, float* C)
{ int i = blockIdx.x * blockDim.x + threadIdx.x;
  C[i] = A[i] + B[i];
}
vecadd<<<32,n/32>>>( dev_A, dev_B, dev_C );
```

Higher dimensional grids/blocks

- Grids can be 2D and blocks can be 2D or 3D
 - struct **dim3** {x; y; z;};
- Grid/block size
 - Dim3 gridDim size of grid dimension x, y (z not used)
 - Dim3 blockDim - size of grid dimension,

2D grid/blocks

- To set dimensions, use for example:

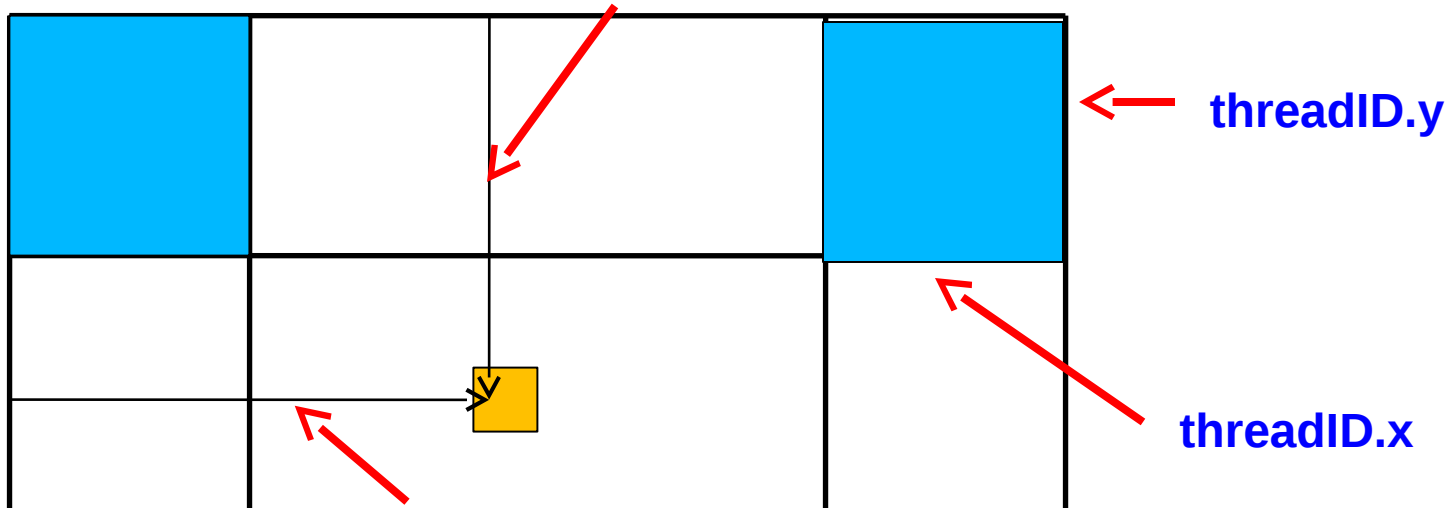
```
dim3 grid(16, 16);    // Grid -- 16 x 16 blocks  
dim3 block(32, 32);   // Block -- 32 x 32 threads  
myKernel<<<grid, block>>>(...);
```

- which sets:

```
gridDim.x = 16  
gridDim.y = 16  
blockDim.x = 32  
blockDim.y = 32  
blockDim.z = 1
```

2-D grids and 2-D blocks

$\text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$



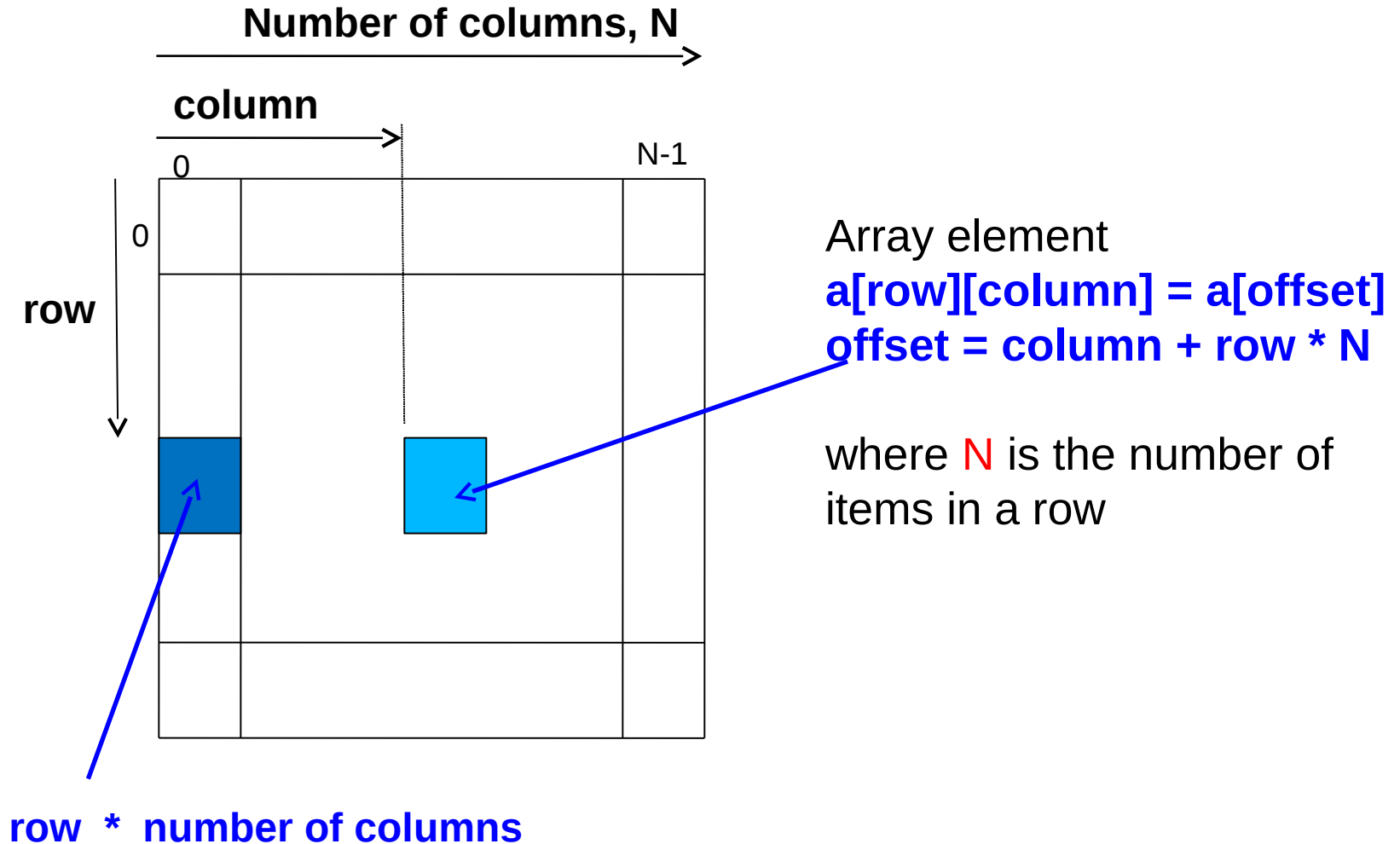
$\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$



Flatten 2 dimension array into linear memory

- Generally memory allocated dynamically on device (GPU) and we cannot not use two-dimensional indices (e.g. **A[row][column]**) to access array as we might otherwise.
- Need to know how array is laid out in memory and then compute distance from the beginning of the array.
- Row major and column major order storage of multi-dimensional arrays.

Flattening an array



2D grid/block example: matrix addition

- `#define N 2048 // size of arrays`
- `__global__ void addMatrix (int *a, int *b, int *c) {`
- `int col = blockIdx.x*blockDim.x+threadIdx.x;`
- `int row = blockIdx.y*blockDim.y+threadIdx.y;`
- `int index = col + row * N;`
- `if (col < N && row < N) c[index]= a[index] + b[index];`
- `}`
- `int main() {`
- `...`
- `dim3 dimBlock (16,16);`
- `dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);`
- `addMatrix<<<dimGrid, dimBlock>>>(devA, devB, devC);`
- `...`
- `}`