CUDA

GPGPU programming

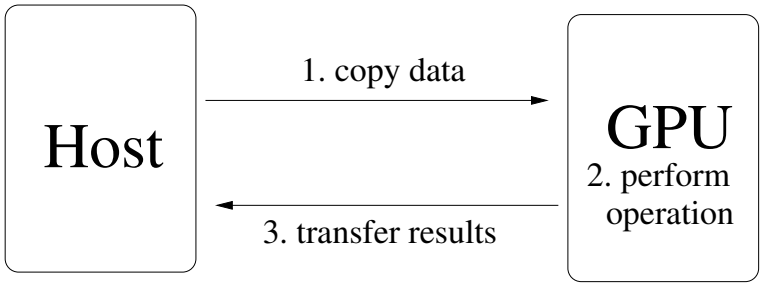
(General-Purpose Graphics Processing Unit)

- Graphics processing requires many similar operations in “graphics pipeline”
  - Triangles going through rotation and scaling, shading, and texturing
- Graphics Processing Units (GPUs) develop to meet this need and then get converted for general purpose programs
- CUDA (Compute Unified Device Architecture) is a GPU design and extension of C (et al) to support GPGPU programming developed by Nvidia
  - Market share leader; leading open alternative is OpenCL

Programming model: Memory

- Program mainly runs on “Host” (= CPU), but can call functions on “Device” (= GPU)

- Host and Device have separate address spaces (at least historically)
  - Memory must be explicitly transferred



Programming model: Processing

- GPU can run many threads simultaneously, but not independently
  - Device threads connected in groups called warps
  - All members of a warp perform the same operation
    - SIMD = Single Instruction, Multiple Data

- Programmer writes function to run on device (kernel)
- Invokes it with a number of blocks and threads (per block)
- All these threads run the function
  - Use implicit arguments blockIdx and threadIdx to identify itself

Logistics: Compiling and running CUDA programs

- Compiling:

  nvcc -o hello hello.cu

Name of the source code. CUDA files use the extension .cu

Compiler; really a wrapper for gcc or another C compiler

-o flag: Used to specify the executable to create

- Running:

  ./hello

“Hello World” for CUDA

```
#include <stdio.h>

__global__ void hello() {
        int id = threadIdx.x + blockIdx.x * blockDim.x;
        printf("Hello from thread %d (%d of block %d)\n", id, threadIdx.x, blockIdx.x);
}

int main() {
        hello<<<3,4>>>();                   //launch 3 blocks of 4 threads each

        cudaDeviceSynchronize();            //make sure kernel completes
}
```

“Hello World” for CUDA

```
#include <stdio.h>

__global__ void hello() {
        int id = threadIdx.x + blockIdx.x * blockDim.x;
        printf("Hello from thread %d (%d of block %d)\n", id, threadIdx.x, blockIdx.x);
}

int main() {
        hello<<<3,4>>>();                  //launch 3 blocks of 4 threads each

        cudaDeviceSynchronize();           //make sure kernel completes
}
```

Device code (runs on GPU)

Possible output

Hello from thread 0 (0 of block 0)
Hello from thread 1 (1 of block 0)
Hello from thread 2 (2 of block 0)
Hello from thread 3 (3 of block 0)
Hello from thread 8 (0 of block 2)
Hello from thread 9 (1 of block 2)
Hello from thread 10 (2 of block 2)
Hello from thread 11 (3 of block 2)
Hello from thread 4 (0 of block 1)
Hello from thread 5 (1 of block 1)
Hello from thread 6 (2 of block 1)
Hello from thread 7 (3 of block 1)

Out of order; blocks are executed separately and in any order

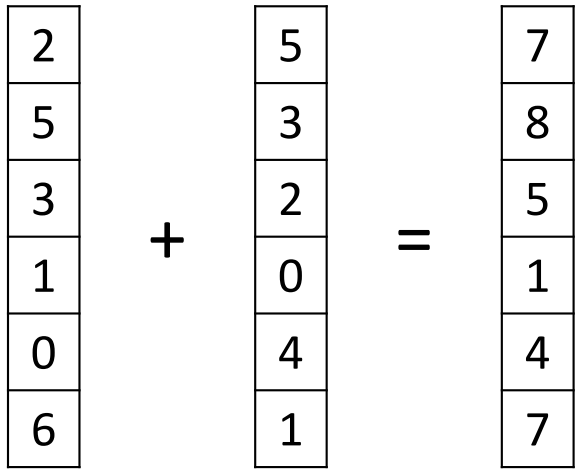
Overview of a CUDA program

- In host code:
  - Allocate memory on device
  - Copy data to device
  - Kernel call
  - Copy results to host
  - Free device memory
- In device code:
  - __global__
  - determine thread ID
  - bounds check

Adding vectors using CUDA

(Not actually fast…)

- Take 2 input arrays and add index-wise to produce output array



Adding vectors using CUDA

(Not actually fast…)

```
int main() {
        int* a;             //first input array (on host)
        int* a_dev;         //first input array (on device)

        a = (int*) malloc(N*sizeof(int));
        cudaMalloc((void**) &a_dev, N*sizeof(int));

        …                   //same for b and res

        free(a);
        cudaFree(a_dev);
}
```

- In host code:
  - **Allocate memory on device**
  - Copy data to device
  - Kernel call
  - Copy results to host
  - **Free device memory**
- In device code:
  - __global__
  - determine thread ID
  - bounds check

Adding vectors using CUDA

(Not actually fast…)

```
int main() {
        …
        cudaMemcpy(a_dev, a, N*sizeof(int),
                cudaMemcpyHostToDevice));

        …

        cudaMemcpy(res, res_dev, N*sizeof(int),
                cudaMemcpyDeviceToHost)

        …
}
```

- In host code:
  - Allocate memory on device
  - **Copy data to device**
  - Kernel call
  - **Copy results to host**
  - Free device memory
- In device code:
  - __global__
  - determine thread ID
  - bounds check

Adding vectors using CUDA

(Not actually fast…)

```
int main() {
        …
        int threads = 512;              //# threads per block
        int blocks  = (N+threads-1)/threads;
                                //# blocks (N/threads rounded up)

        kernel<<<blocks,threads>>>(res_dev, a_dev, b_dev);
        …
}
```

- In host code:
  - Allocate memory on device
  - Copy data to device
  - **Kernel call**
  - Copy results to host
  - Free device memory
- In device code:
  - __global__
  - determine thread ID
  - bounds check

Adding vectors using CUDA

(Not actually fast…)

```
__global__ void kernel(int* res, int* a, int* b) {
    //sets res[i] = a[i] + b[i]
    //each thread is responsible for one value of i

    int thread_id = threadIdx.x + blockIdx.x*blockDim.x;

    if(thread_id < N) {
        res[thread_id] = a[thread_id] + b[thread_id];
    }
}
```

- In host code:
  - Allocate memory on device
  - Copy data to device
  - Kernel call
  - Copy results to host
  - Free device memory
- In device code:
  - **__global__**
  - **determine thread ID**
  - **bounds check**

Threads and blocks

```
int threads = 512;                          //# threads per block
int blocks = (N+threads-1)/threads;     //# blocks (N/threads, rounded up)
kernel<<<blocks,threads>>>(res_dev, a_dev, b_dev);
```

- Why use more than a single block?
- Why not use N blocks?

Threads and blocks

```
int threads = 512;                          //# threads per block
int blocks = (N+threads-1)/threads;     //# blocks (N/threads, rounded up)
kernel<<<blocks,threads>>>(res_dev, a_dev, b_dev);
```

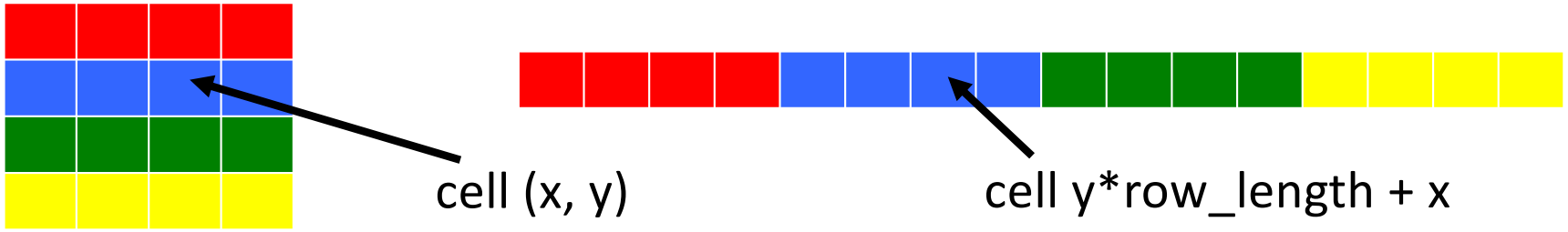
- Why use more than a single block?
  - Limited number of threads per block (depends on card being used)
- Why not use N blocks?
  - Not as fast: blocks are split into warps, which run simultaneously
  - Threads in block share variables (__shared__) and have barrier (__syncthreads())
  - Also, technically limited (w/ newer cards, the limit is $2^{31} – 1$)

Linearizing multi-dimensional arrays

- cudaMemcpy only transfers 1D arrays
- need to represent 2D array:

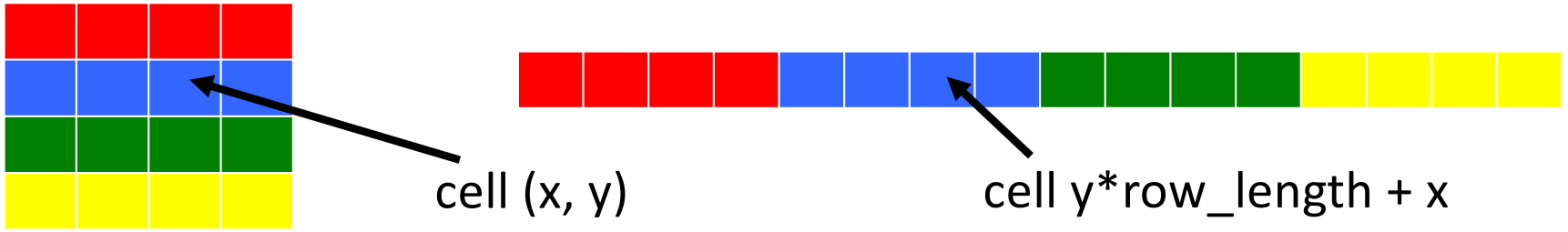
Linearizing multi-dimensional arrays

- cudaMemcpy only transfers 1D arrays
- need to represent 2D array:



Linearizing multi-dimensional arrays

- cudaMemcpy only transfers 1D arrays
- need to represent 2D array:



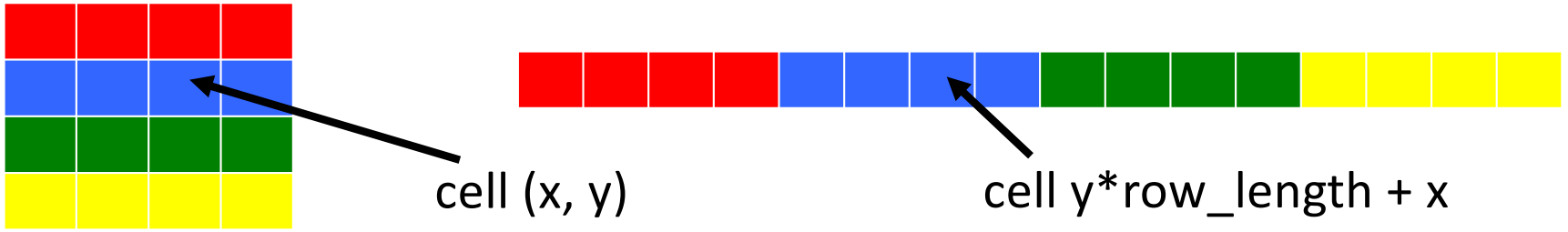
What is the 1D index of the cell below the cell with 1D index i?

A. i + 1 B. i + 4 C. i + row_length D. i * row_length – 1

E. Insufficient information to determine it

Linearizing multi-dimensional arrays

- cudaMemcpy only transfers 1D arrays
- need to represent 2D array:



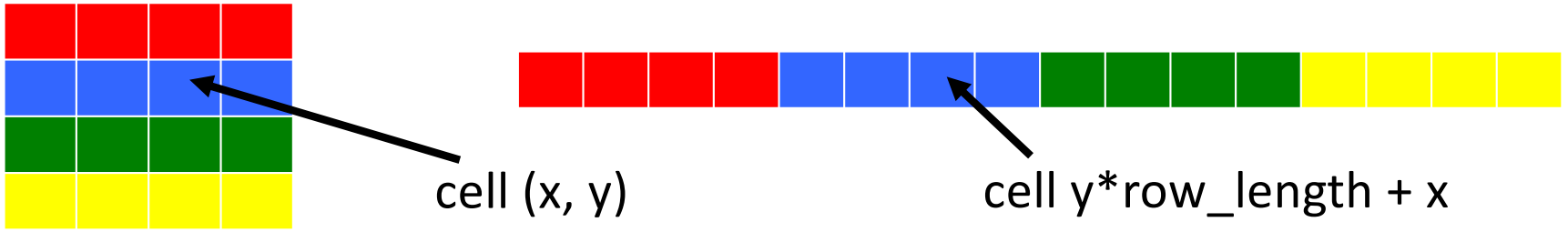
What is the 1D index of the cell below the cell with 1D index i?

A. i + 1 B. i + 4 C. i + row_length D. i * row_length – 1

E. Insufficient information to determine it

Linearizing multi-dimensional arrays

- cudaMemcpy only transfers 1D arrays
- need to represent 2D array:



Which test determines if the cell with 1D index i is on the right edge (of the 2D matrix)?

A. i % row_length == 0

B. i % col_length == 0

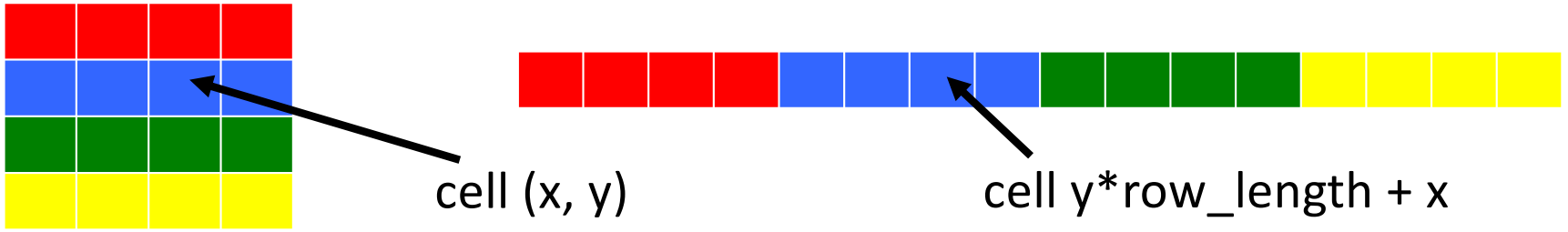
C. i + row_length >= row_length * col_length

D. i % row_length == row_length – 1

E. Not exactly one of the above

Linearizing multi-dimensional arrays

- cudaMemcpy only transfers 1D arrays
- need to represent 2D array:



Which test determines if the cell with 1D index i is on the right edge (of the 2D matrix)?

A. i % row_length == 0

B. i % col_length == 0

C. i + row_length >= row_length * col_length

D. i % row_length == row_length – 1

E. Not exactly one of the above