CUDA

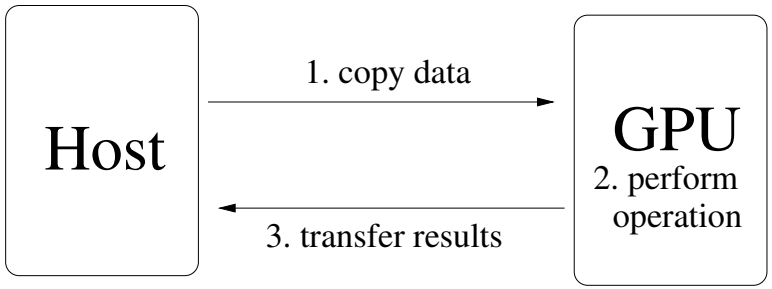
GPGPU programming
(General-Purpose Graphics Processing Unit)

- 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

“Hello World” for CUDA

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…)

Recall: Calling a CUDA kernel

```
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);
```

Recall: Calling a CUDA kernel

```
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?

Recall: Calling a CUDA kernel

```
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);
```

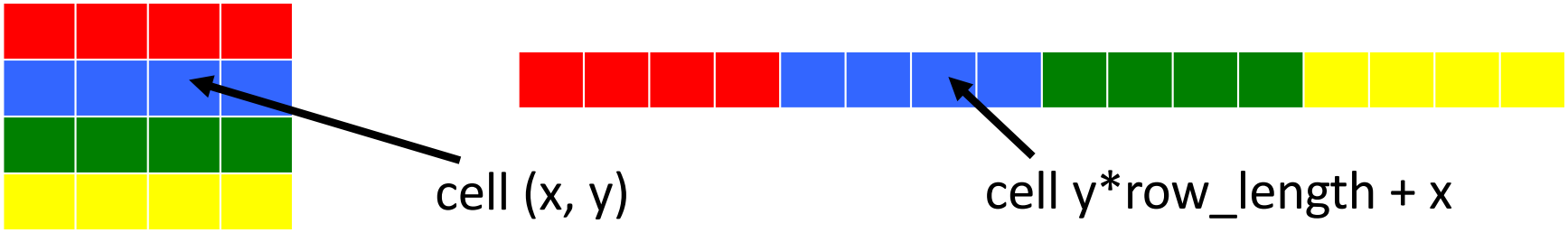
- Why use more than a single block?
  - Limited number of threads per block (depends on card being used)
- Why not use N blocks?
  - Threads in block share variables (__shared__) and have barrier (__syncthreads())
  - Also, technically limited (w/ newer cards, the limit is $2^{31} - 1$)

Linearizing multi-dimensional arrays

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

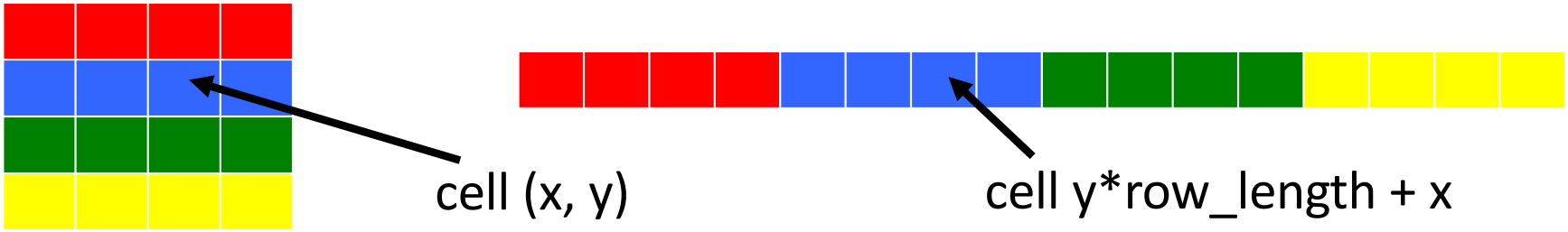
Linearizing multi-dimensional arrays

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



Linearizing multi-dimensional arrays

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



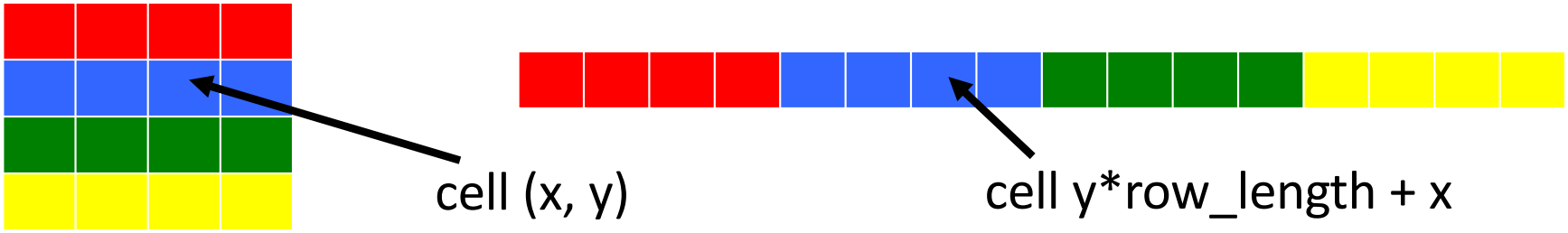
What is the 1D index of the cell below the cell with 1D index i?

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:



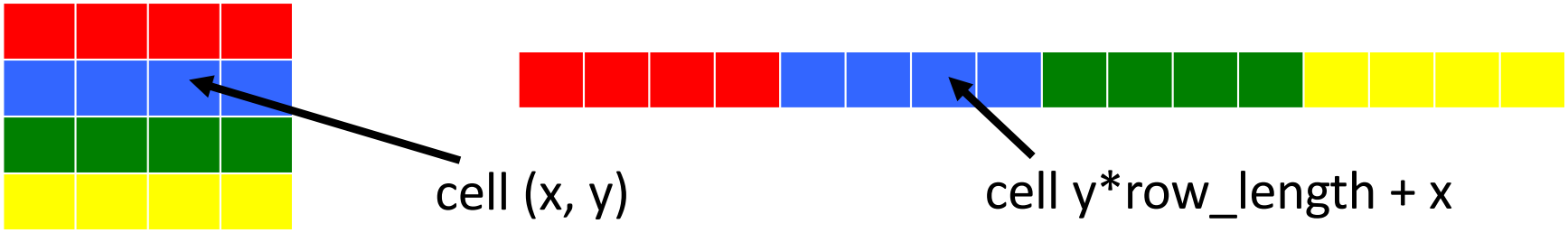
What is the 1D index of the cell below the cell with 1D index i?

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

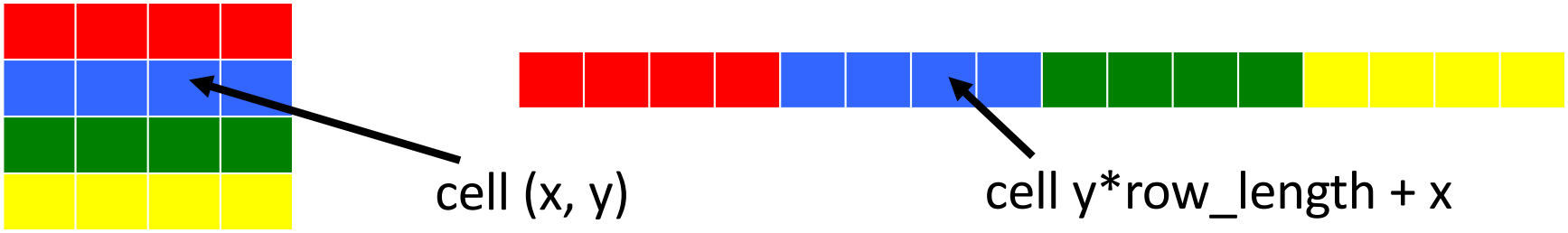
C. i + row_length >= row_length * col_length

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