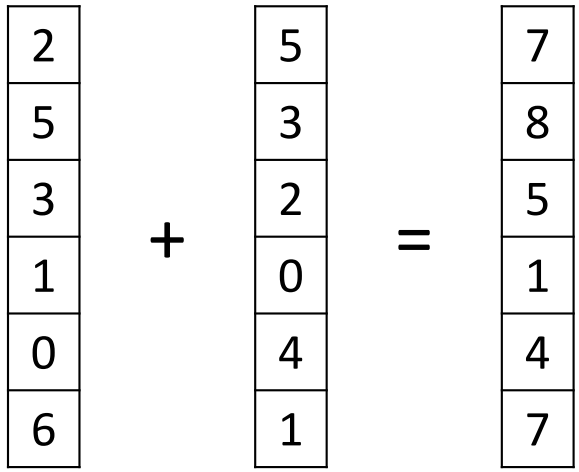
More CUDA

2/25/26

Recall: Adding vectors using CUDA

(Not actually fast…)

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



Recall: Allocating and freeing device memory

```
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

Recall: Moving data back and forth

```
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

Calling the kernel

```
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

The kernel itself

```
__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);
```

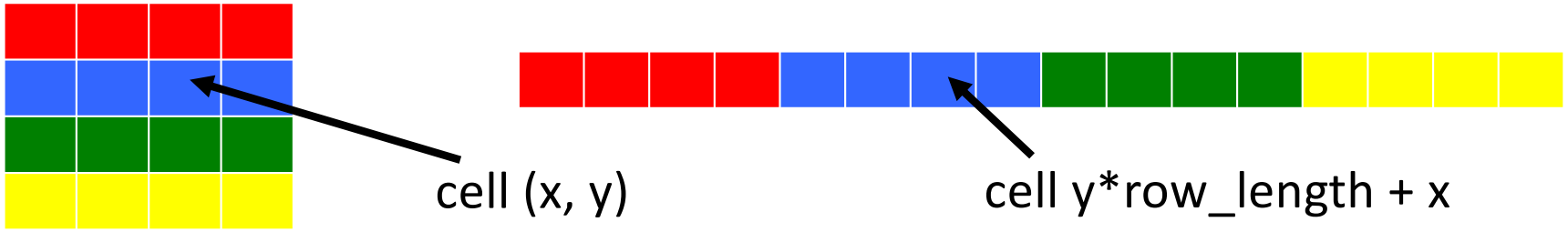
- Why use more than a single block?
  - Limited number of threads per block (depends on card being used)
- Why not use N blocks?
  - Not as fast: blocks are split into warps, which run simultaneously
  - Threads in block share variables (__shared__) and have barrier (__syncthreads())
  - Also, technically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

Linearizing multi-dimensional arrays

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

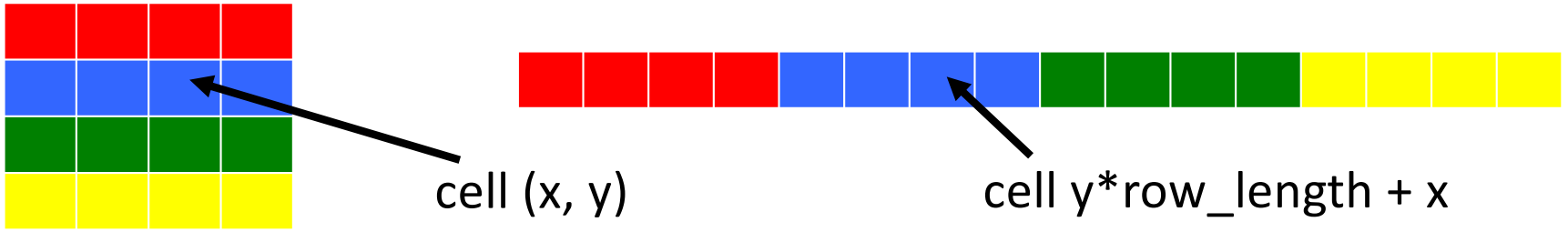
Linearizing multi-dimensional arrays

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



Linearizing multi-dimensional arrays

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



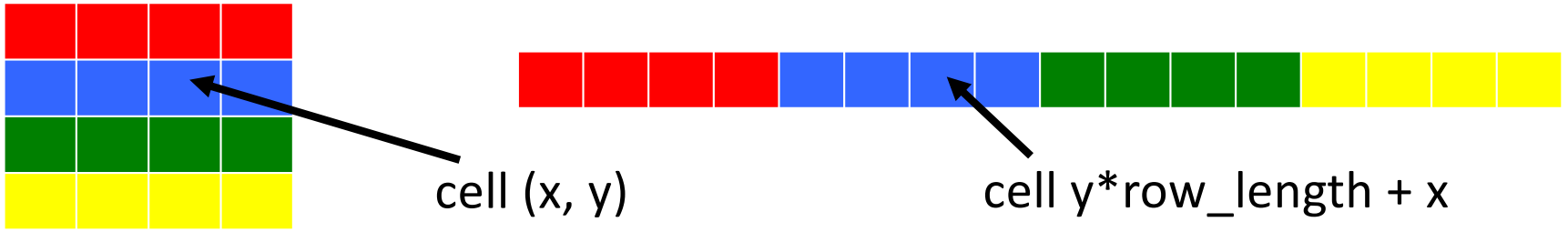
What is the 1D index of the cell below the cell with 1D index i?

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:



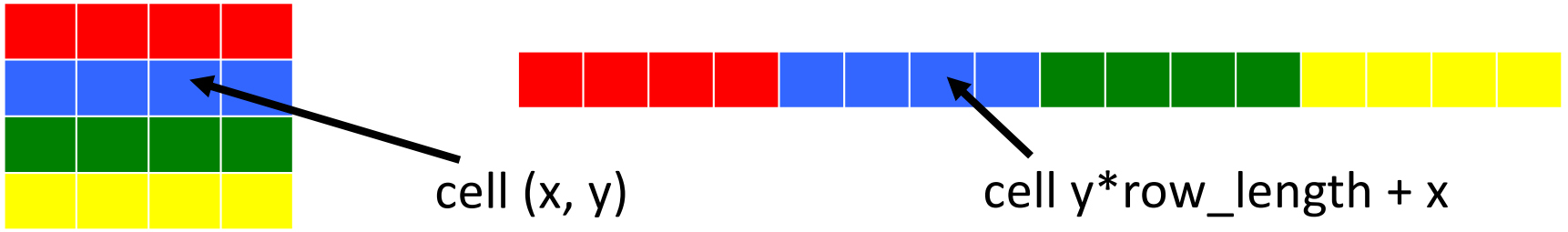
What is the 1D index of the cell below the cell with 1D index i?

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

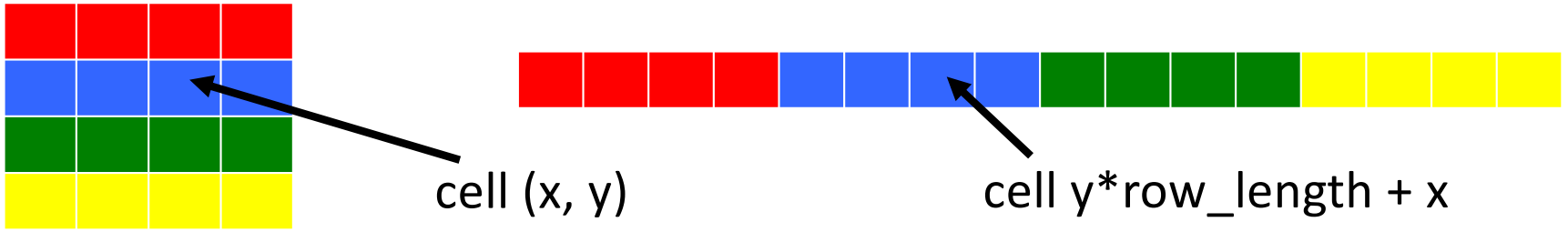
C. i + row_length >= row_length * col_length

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

Kernel from the lab

```
__global__ void kernel(int width, int height, unsigned char *d_input, unsigned char* d_output) {

	//coordinates of pixel for which this call is responsible
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	int j = blockIdx.y * blockDim.y + threadIdx.y;

	int offset;  //index in array corresponding to a pixel

	if(i >=0 && i < width && j >=0 && j < height) {
		offset = (j * width + i) * 3 + 0;		//0 is red channel
		d_output[offset] = 0;

		offset = (j * width + i) * 3 + 1;		//1 is green channel
		d_output[offset] = d_input[offset];

		offset = (j * width + i) * 3 + 2;		//2 is blue channel
		d_output[offset] = d_input[offset];
	}
}
```

Another application: Voronoi diagrams

- Given set of points, divide plane into sets closest to each of them