



# SYCL

---

Developed by : Benson Muite, [benson\\_muite@emailplus.org](mailto:benson_muite@emailplus.org)  
Work supported by NSF grant OAC-1829644

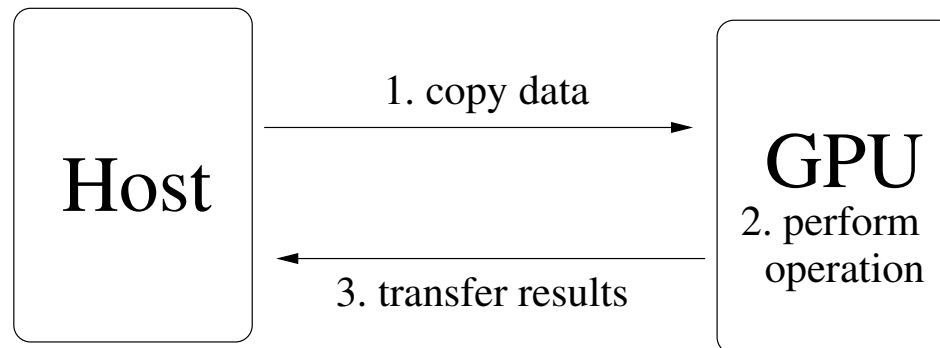
# GPGPU programming

(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
- SYCL is an XPU (GPU, CPU and FPGA) design and extension of C++ to support Heterogeneous programming developed by the Khronos group
  - Alternatives for GPUs include OpenCL, CUDA, ROCm

# Programming model: Memory

- Program mainly runs on “Host” (= CPU), but can call functions on “Devices” (= GPU, FPGA, CPU)
- Host and Device can have separate address spaces (at least historically)
  - Memory may need to be explicitly transferred
  - SYCL can automate some of the transfers



# Programming model: Processing

- XPU can run many threads simultaneously, but not independently
  - Performance benefits obtained from SIMD = Single Instruction, Multiple Data
  - SYCL abstracts away SIMD parallelism to enable the same code to run on different compute devices
- Programmer writes function to run on device (kernel)
- All the threads run the function

“Hello World” for SYCL

# Overview of a SYCL program

- In host code:
  - Setup problem call kernel
- In device code:
  - Allocate memory on device
  - Copy data from host
  - Kernel call
  - Copy results to host (can be done implicitly on expiry of scope)
  - Free device memory (can be done implicitly on expiry of scope)
  - `q.submit([&](cl::sycl::handler& cgh)`
  - determine thread ID
  - bounds check

---

# Adding vectors and Hello World using SYCL

(Not actually fast...)

# Recall: Calling a SYCL kernel

```
cgh.parallel_for<class vector_add>(work_items,  
    [=] (cl::sycl::id<1> tid)
```

or

```
cgh.parallel_for<class hello>(cl::sycl::nd_range<1>{a, b},  
    [=] (cl::sycl::nd_item<1> item)
```



# Recall: Calling a SYCL kernel

```
cgh.parallel_for<class vector_add>(work_items,  
                                   [=] (cl::sycl::id<1> tid)
```

or

```
cgh.parallel_for<class hello>(cl::sycl::nd_range<1>{a, b},  
                              [=] (cl::sycl::nd_item<1> item)
```

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

# Recall: Calling a SYCL kernel

```
cgh.parallel_for<class vector_add>(work_items,  
                                  [=] (cl::sycl::id<1> tid)
```

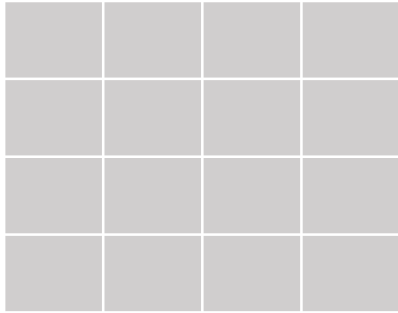
or

```
cgh.parallel_for<class hello>(cl::sycl::nd_range<1>{a, b},  
                              [=] (cl::sycl::nd_item<1> item)
```

- Why use more than a single block?
  - Limited number of threads per processing element (depends on card being used)
- Why not use N blocks?
  - Threads in a processing unit share variables and have synchronization barriers
  - So, technically limited

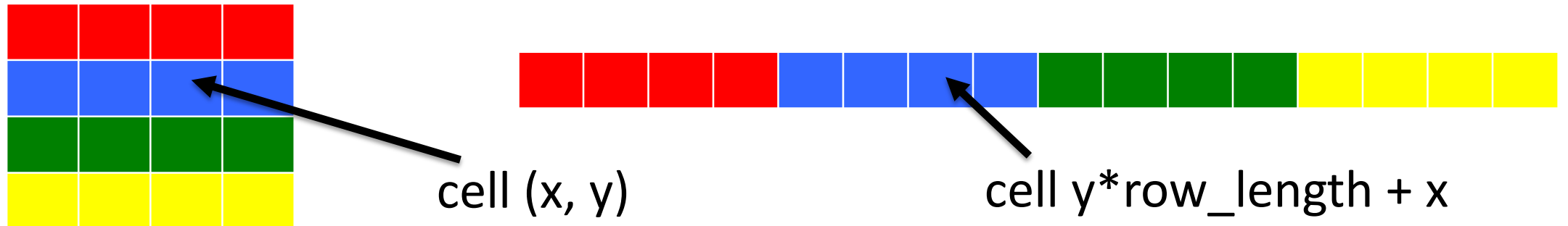
# Linearizing multi-dimensional arrays

- Memory is typically mapped to 1D arrays
- Need to represent 2D array:



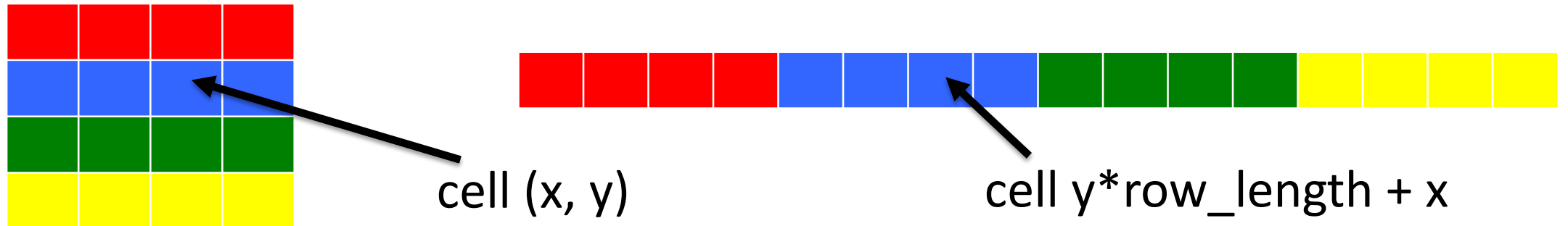
# Linearizing multi-dimensional arrays

- Memory is typically mapped to 1D arrays
- need to represent 2D array:



# Linearizing multi-dimensional arrays

- Memory is typically mapped to 1D arrays
- need to represent 2D array:

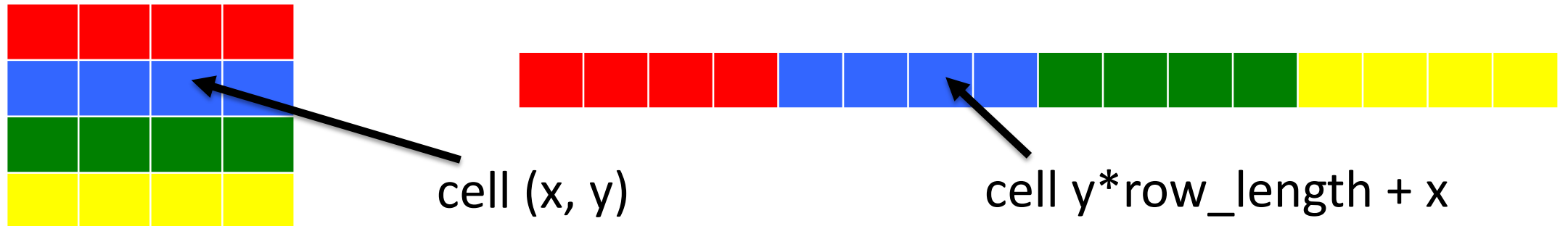


What is the 1D index of the cell below the cell with 1D index  $i$ ?

- A.  $i + 1$       B.  $i + 4$       C.  $i + \text{row\_length}$       D.  $i * \text{row\_length} - 1$   
E. Insufficient information to determine it

# Linearizing multi-dimensional arrays

- Memory is typically mapped to 1D arrays
- Need to represent 2D array:

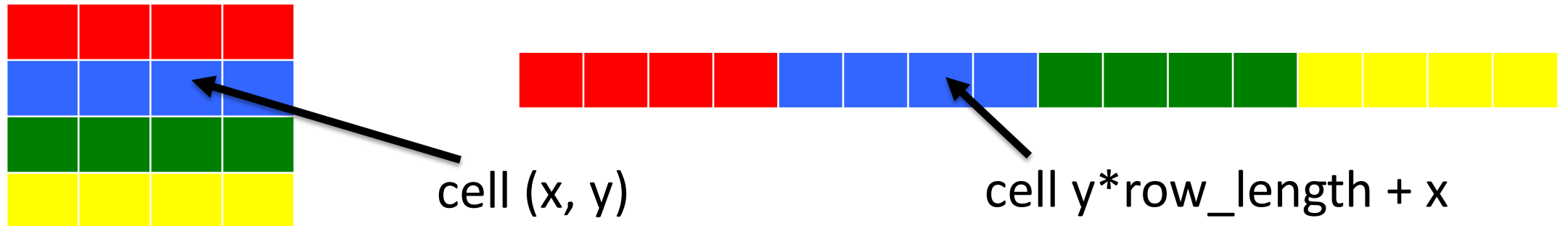


What is the 1D index of the cell below the cell with 1D index  $i$ ?

- A.  $i + 1$       B.  $i + 4$       C.  $i + \text{row\_length}$       D.  $i * \text{row\_length} - 1$   
E. Insufficient information to determine it

# Linearizing multi-dimensional arrays

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

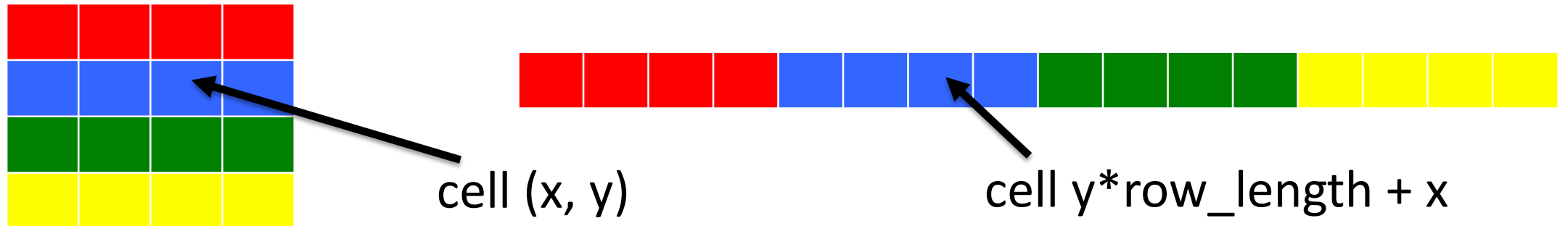


Which test determines if the cell with 1D index  $i$  is on the right edge (of the 2D matrix)?

- A.  $i \% \text{row\_length} == 0$
- B.  $i \% \text{col\_length} == 0$
- C.  $i + \text{row\_length} \geq \text{row\_length} * \text{col\_length}$
- D.  $i \% \text{row\_length} == \text{row\_length} - 1$
- 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.  $i \% \text{row\_length} == 0$
- B.  $i \% \text{col\_length} == 0$
- C.  $i + \text{row\_length} \geq \text{row\_length} * \text{col\_length}$
- D.  $i \% \text{row\_length} == \text{row\_length} - 1$
- E. Not exactly one of the above