Heterogeneous Computing for AI - Lecture ~06

Introduction to CUDA programming with PyCuda

Raghava Mukkamala
Associate Professor, Director for Centre for Business Data Analytics (bda.cbs.dk)
Department of Digitalization, Copenhagen Business School, Denmark
Email: rrm.digi@cbs.dk

Associate Professor, Department of Technology,
Kristiania University College,Oslo, Norway

Many slides are taken from the following authors with due respect to their contributions.
Applied Parallel Programming course (ECE408 / CS483 / CSE408)
https://wiki.illinois.edu/wiki/display/ECE408/ECE408+Home

Outline

- Introduction to Numpy and Scientific Computing
- Heterogeneous Parallel Computing
- Introduction to Data Parallel Programming
- Logical Execution Model of CUDA
    - Blocks
    - Threads
- Sample CUDA program

Learning goals for today

Theoretical

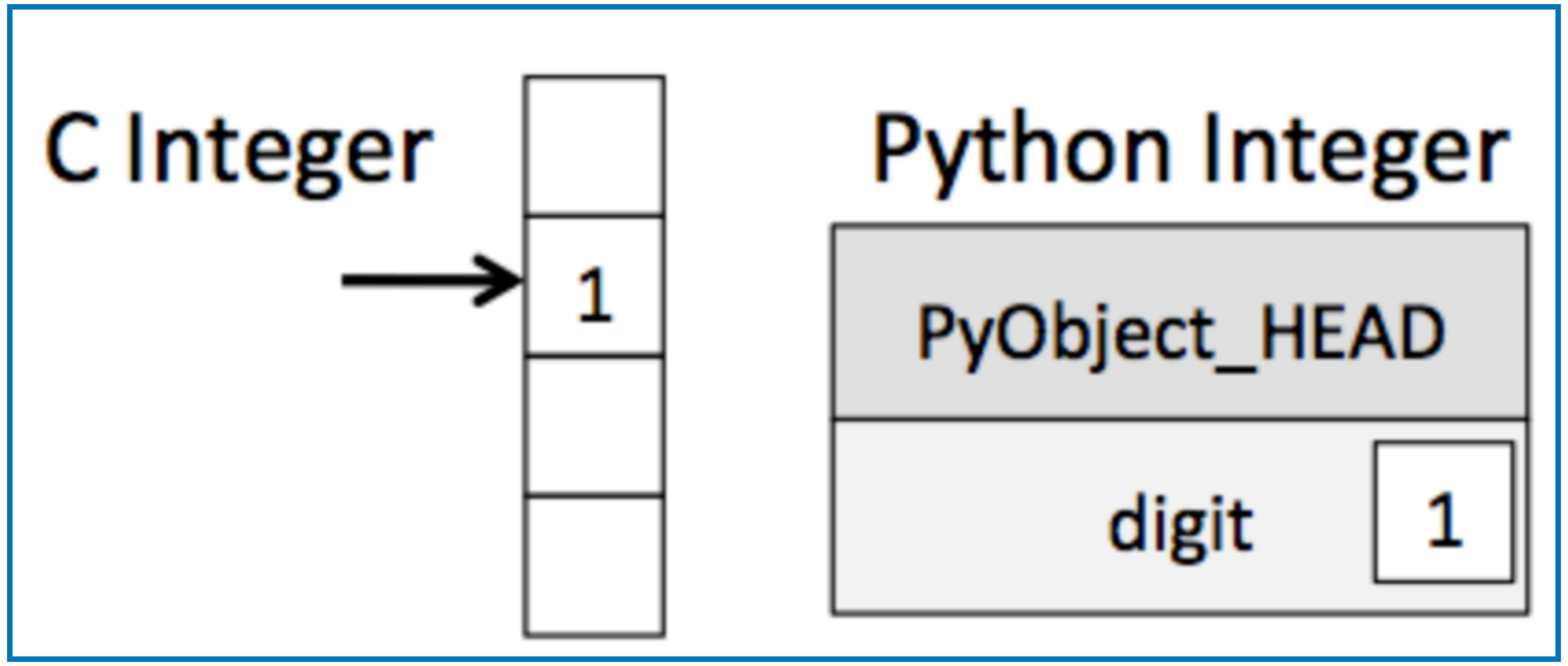
- To learn the basic concepts of data parallel computing
- To learn the basic features of the CUDA programming interface

Practical

- Be able to write programs with Numpy for data transformation and handling
- Understand the structure of programs that can run on CUDA GPU

Numpy and Scientific Computing

Motivation for Numpy

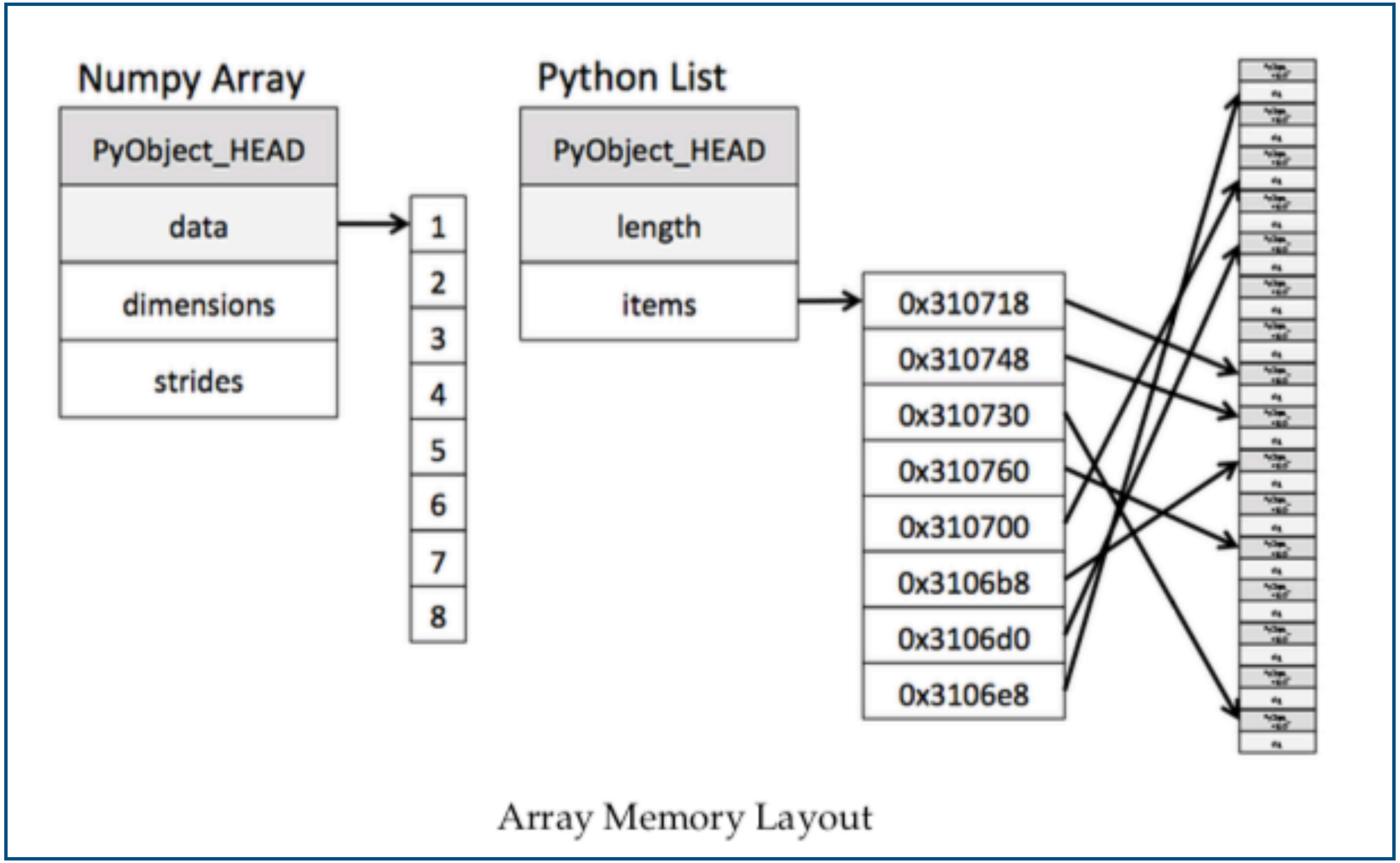


Python Integer Is More Than Just an Integer

- C integer is essentially a label for a position in memory whose bytes encode an integer value.
- A Python integer is a pointer to Python object information, including the bytes that contain the integer value and a lot of other information, such as type of variable, etc.

source: Python Data Science Handbook by Jake VanderPlas

A Python List Is More Than Just a List



source: Python Data Science Handbook by Jake VanderPlas

NumPy Motivation Example

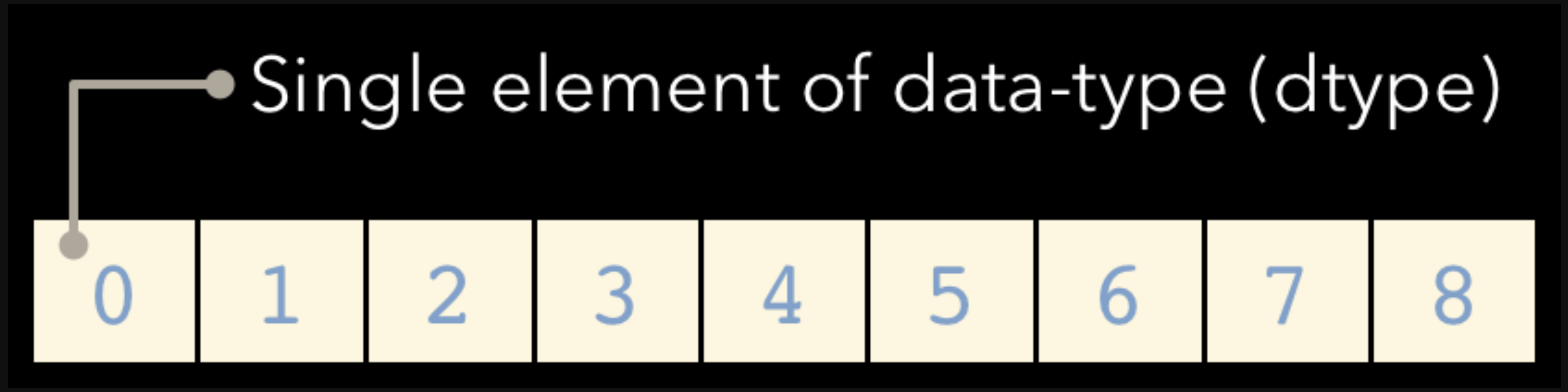
```python
height = [1.73, 1.68, 1.71, 1.89, 1.79]

weight = [65.4, 59.2, 63.6, 88.4, 68.7]

# You want to calculate BMI = weight / height ** 2
bmi = []

for i in range(len(weight)):
    bmi.append(weight[i] / height[i] ** 2)

print('bmi: ', bmi)
```

```python
import numpy as np

np_height = np.array(height)

np_weight = np.array(weight)

np_bmi = np_weight / np_height ** 2

print('np_bmi: ', np_bmi)
```

source: Source DataCamp

What is NumPy

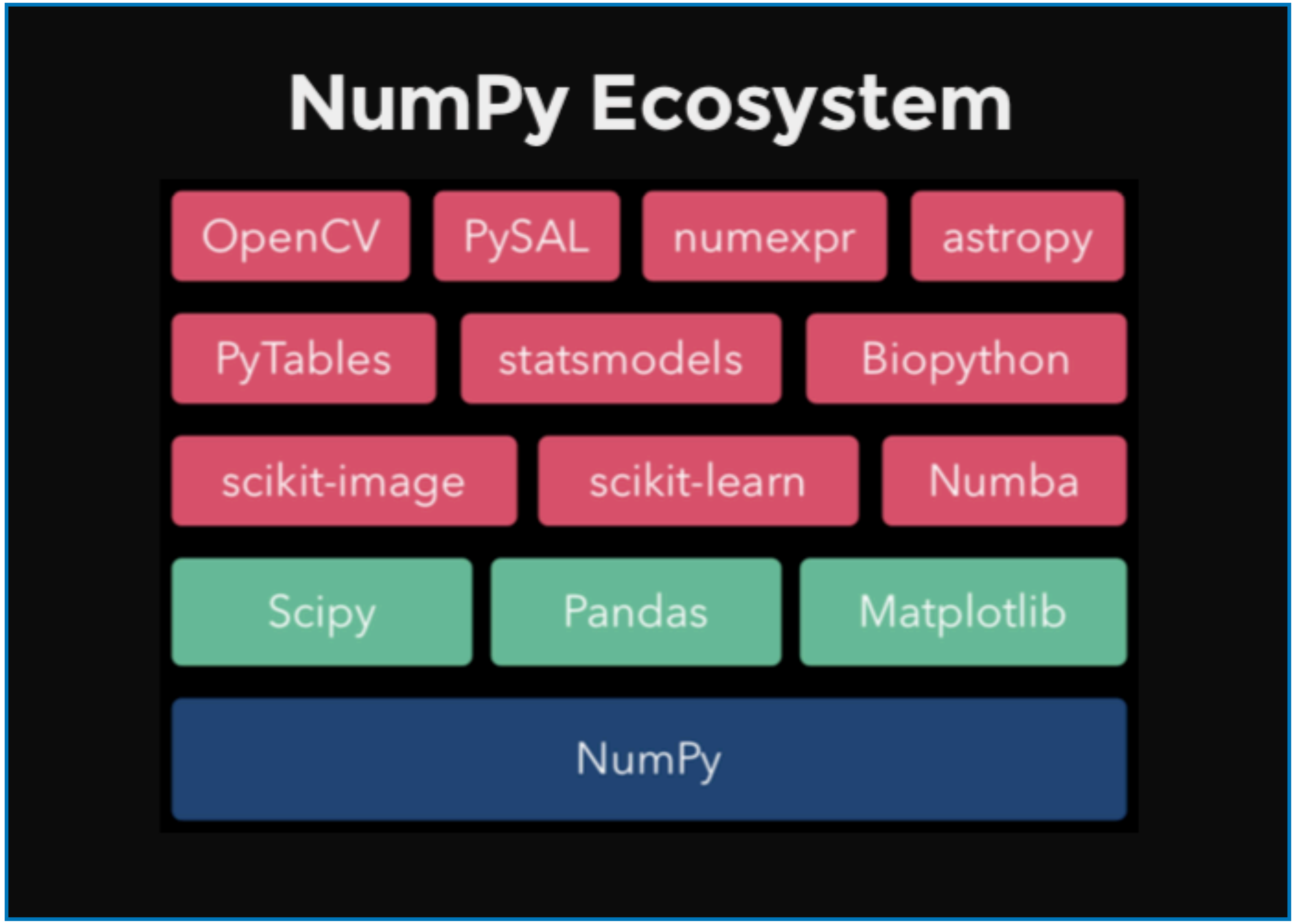
- NumPy is a Python C extension library for array-oriented computing
  - Efficient
  - In-memory
  - Contiguous (or Strided)
  - Homogeneous (but types can be algebraic)



- NumPy is suited to many applications
  - Image processing
  - Signal processing
  - Linear algebra
  - A plethora of others

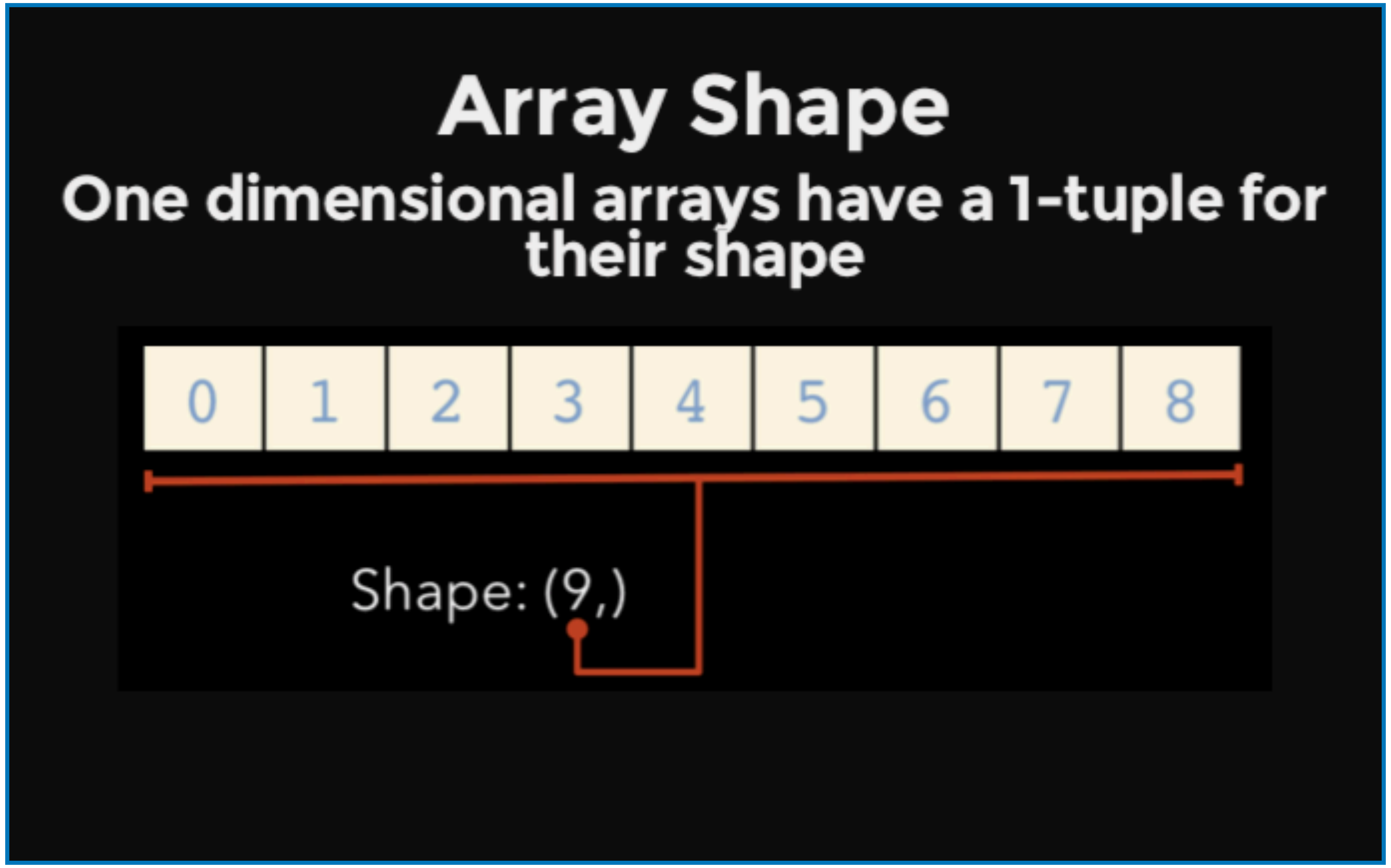
Bryan Van de Ven: Introduction to NumPy

Foundation of Scientific Stack

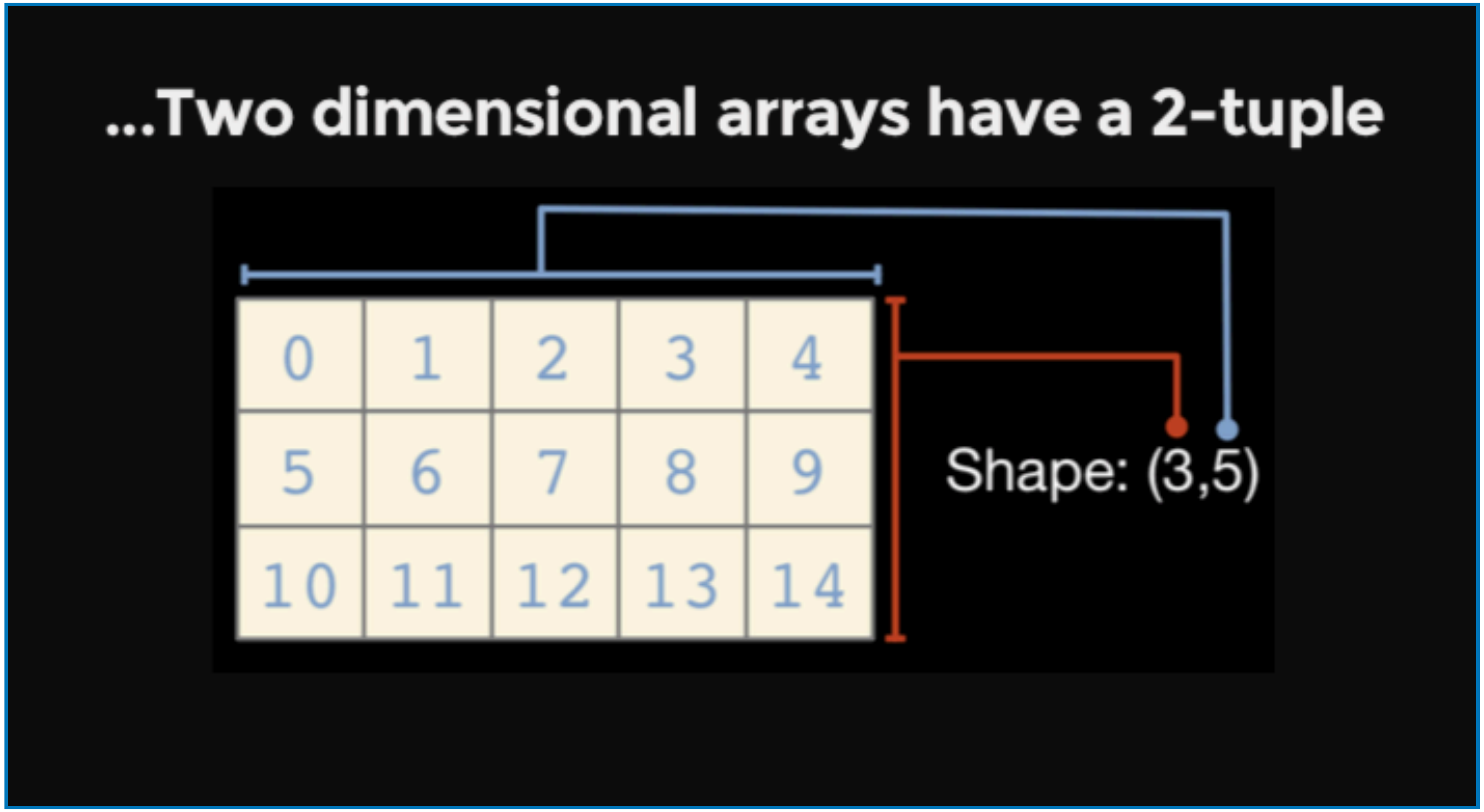


Bryan Van de Ven: Introduction to NumPy

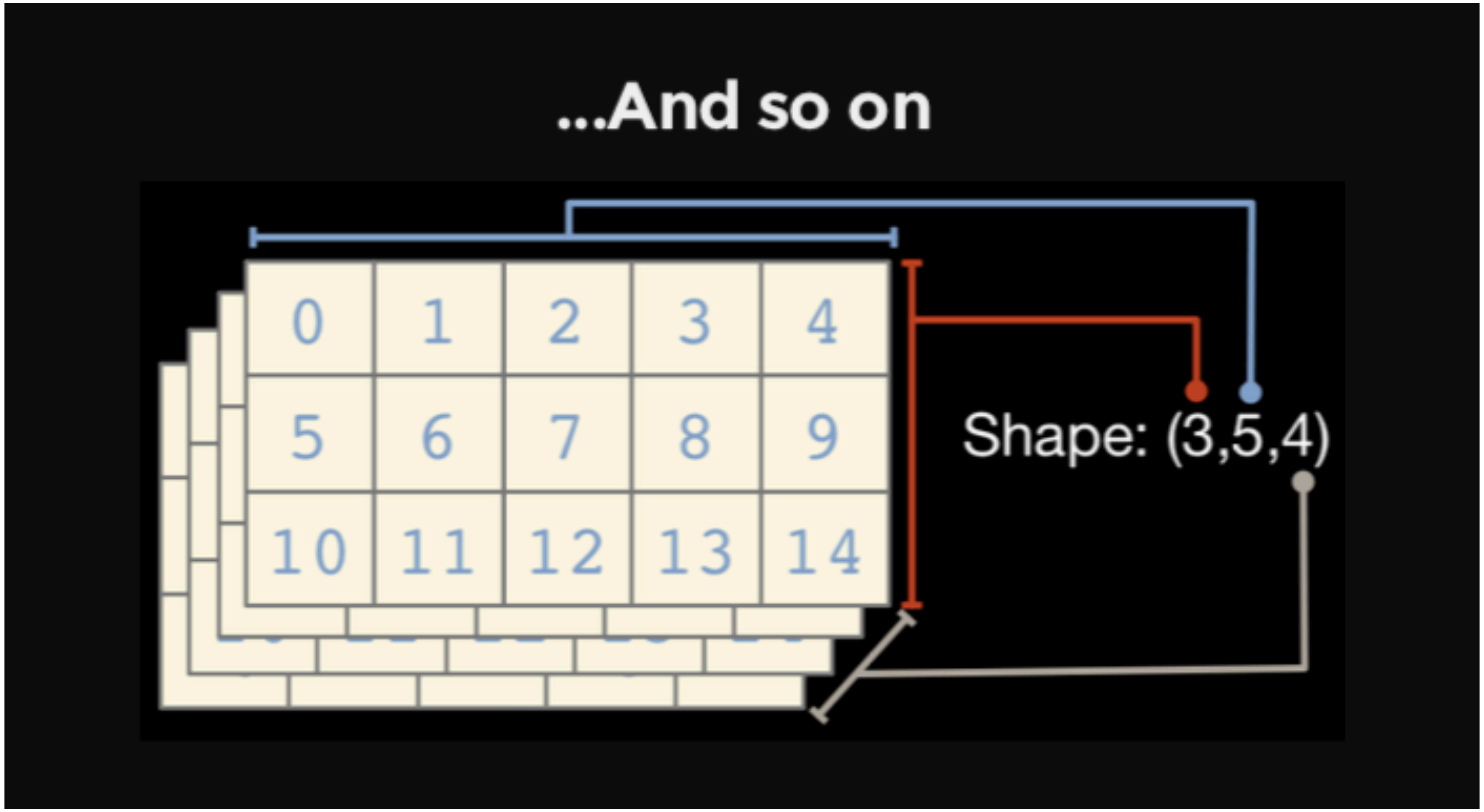
Array Shape and Dimension



Array Shape and Dimension



Array Shape and Dimension



Bryan Van de Ven: Introduction to NumPy

Data Types

Array Element Type (dtype)

- NumPy arrays comprise elements of a single data type
- The type object is accessible through the `.dtype` attribute

Here are a few of the most important attributes of dtype objects

- `dtype.byteorder` — big or little endian
- `dtype.itemsize` — element size of this dtype
- `dtype.name` — a name for this dtype object
- `dtype.type` — type object used to create scalars

There are many others...

Bryan Van de Ven: Introduction to NumPy

Data Types

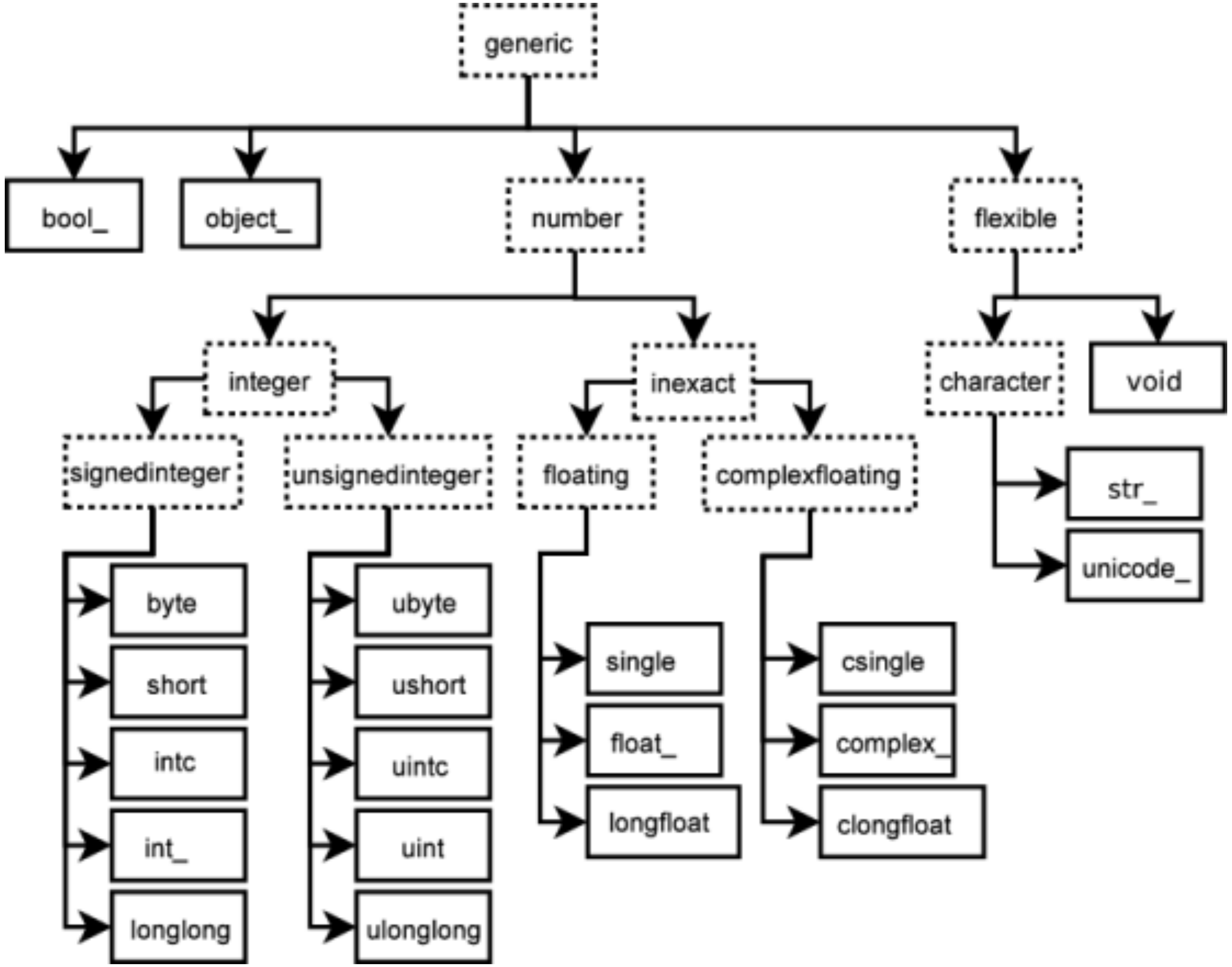
Array dtypes are usually inferred automatically

```
In [16]: a = np.array([1,2,3])

In [17]: a.dtype
Out[17]: dtype('int64')

In [18]: b = np.array([1,2,3,4.567])

In [19]: b.dtype
Out[19]: dtype('float64')
```

But can also be specified explicitly

```
In [20]: a = np.array([1,2,3], dtype=np.float32)

In [21]: a.dtype
Out[21]: dtype('int64')

In [22]: a
Out[22]: array([ 1.,  2.,  3.], dtype=float32)
```

NumPy Data Types



Bryan Van de Ven: Introduction to NumPy

Some Examples

Array Creation

Explicitly from a list of values

```
In [2]: np.array([1,2,3,4])
Out[2]: array([1, 2, 3, 4])
```

As a range of values

```
In [3]: np.arange(10)
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

By specifying the number of elements

```
In [4]: np.linspace(0, 1, 5)
Out[4]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

Zero-initialized

```
In [4]: np.zeros((2,2))
Out[4]:
array([[ 0.,  0.],
[ 0.,  0.]])
```

One-initialized

```
In [5]: np.ones((1,5))
Out[5]: array([[ 1.,  1.,  1.,  1.,  1.]])
```

Uninitialized

```
In [4]: np.empty((1,3))
Out[4]: array([[  2.12716633e-314,   2.12716633e-314,   2.15203762e-314]])
```

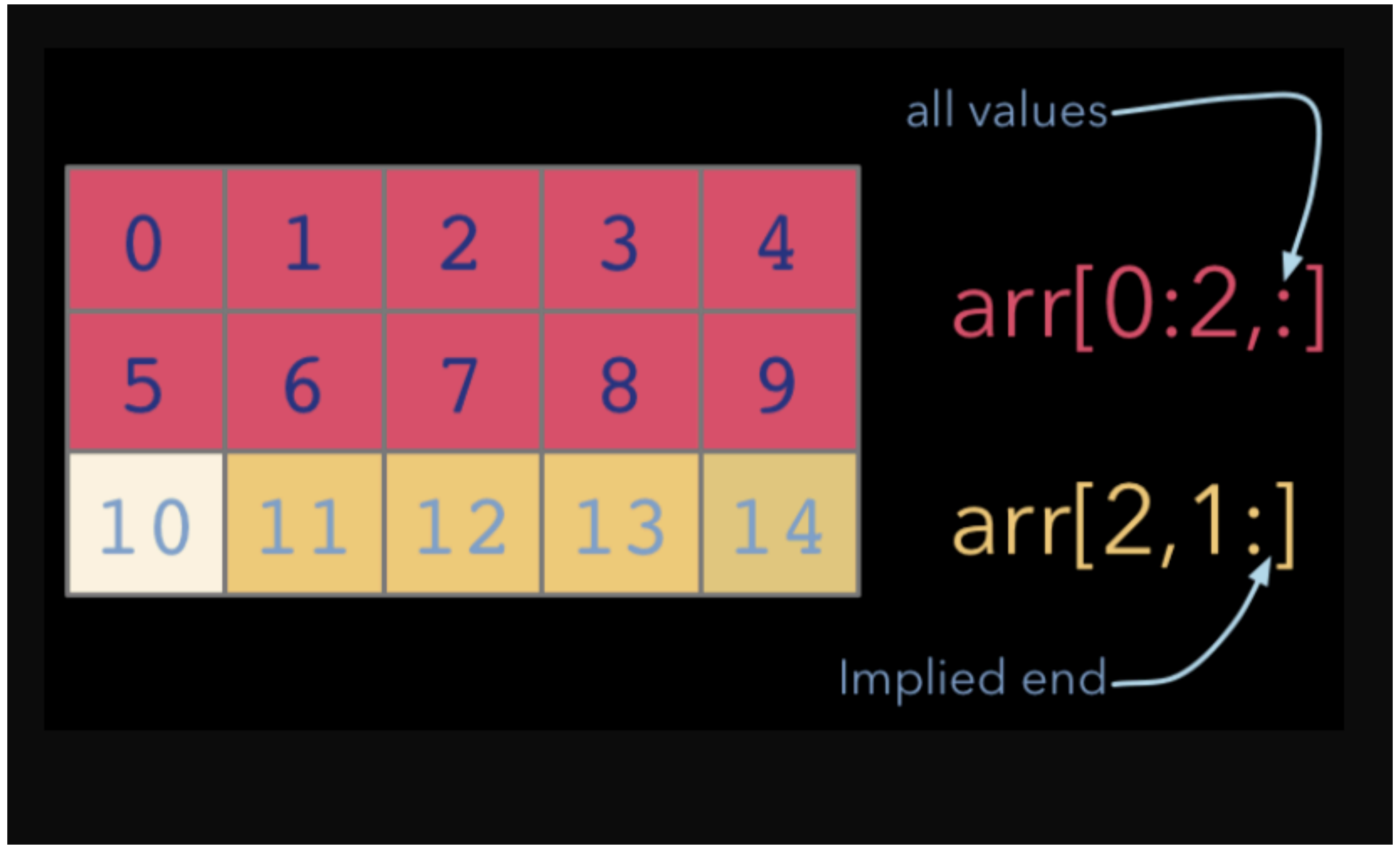
Constant diagonal value

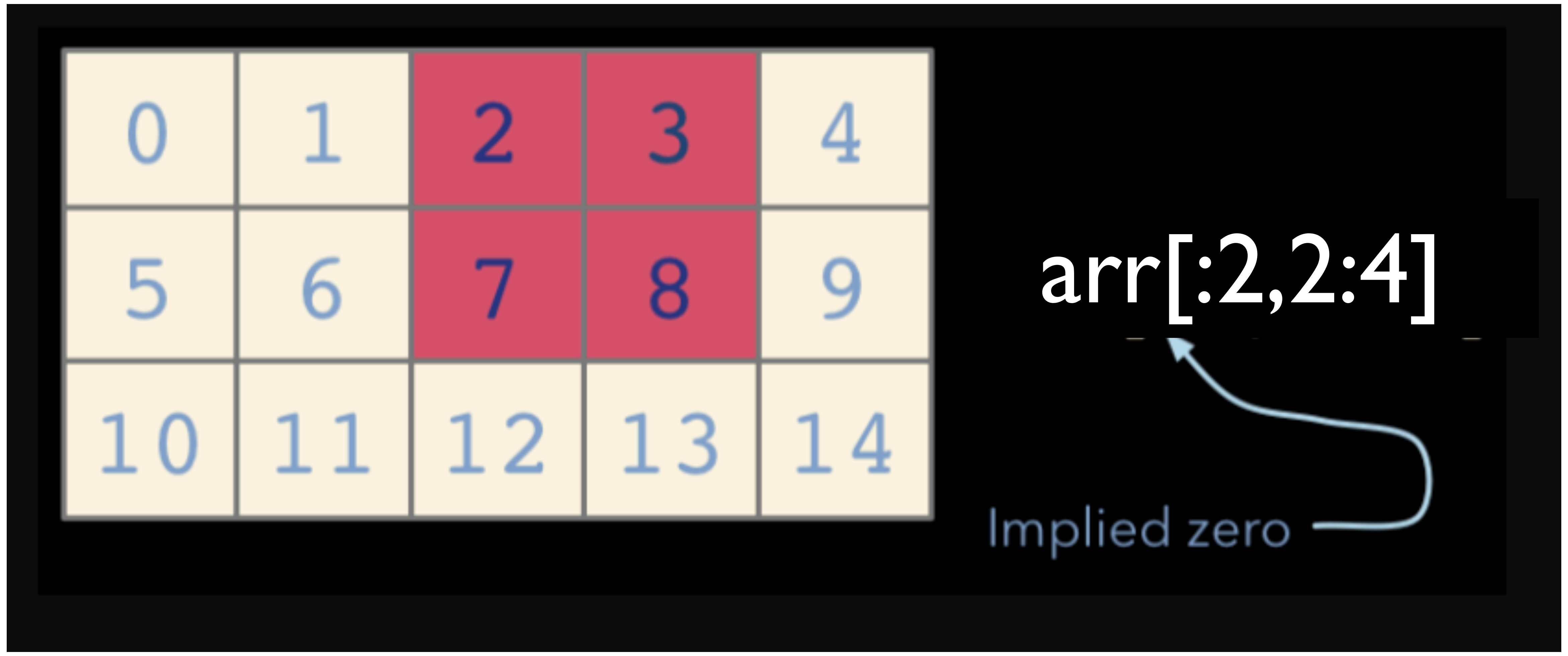
```
In [6]: np.eye(3)
Out[6]:
array([[ 1.,  0.,  0.],
[ 0.,  1.,  0.],
[ 0.,  0.,  1.]])
```

Multiple diagonal values

```
In [7]: np.diag([1,2,3,4])
Out[7]:
array([[1, 0, 0, 0],
[0, 2, 0, 0],
[0, 0, 3, 0],
[0, 0, 0, 4]])
```

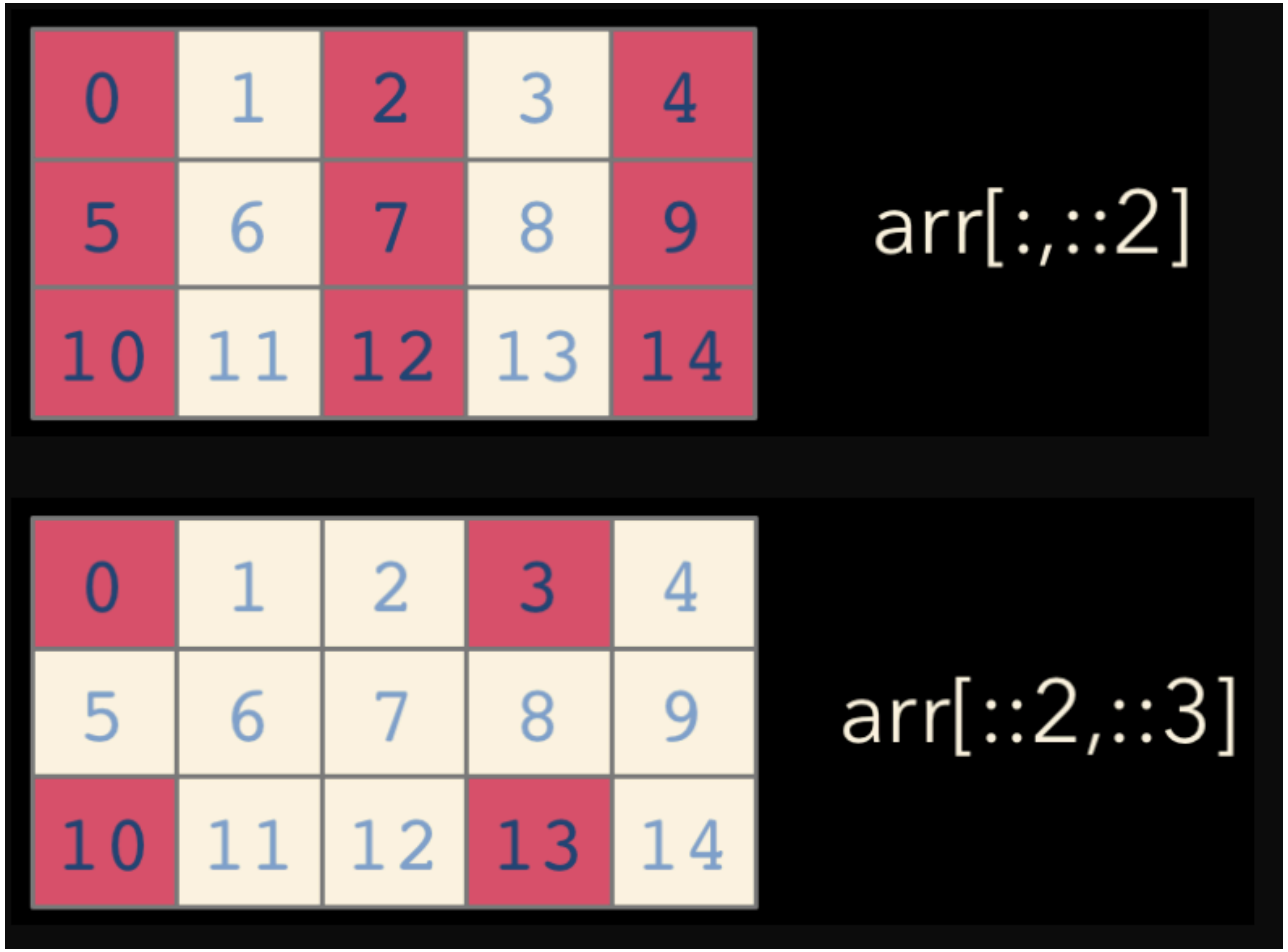
Indexing and Slicing





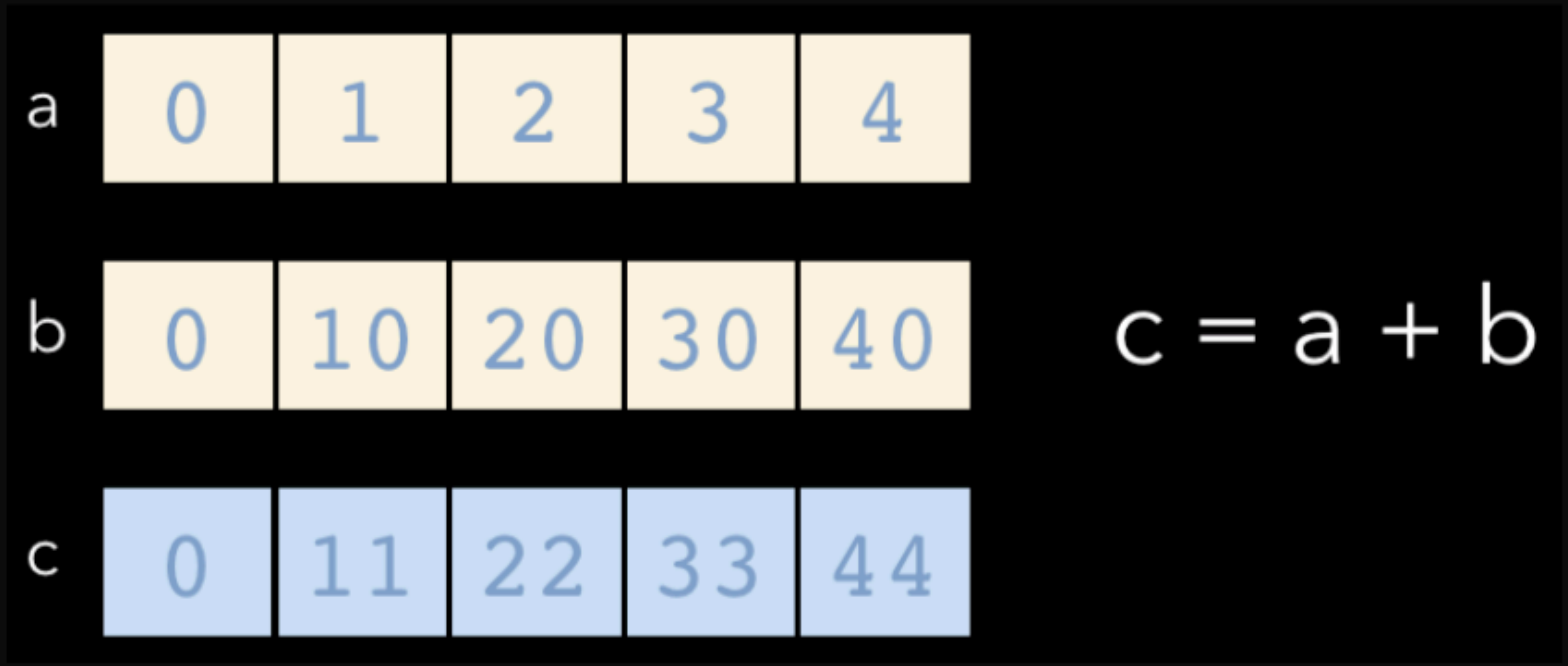
Bryan Van de Ven: Introduction to NumPy

Indexing and Slicing



Universal Functions (ufuncs)

NumPy ufuncs are functions that operate element-wise on one or more arrays



ufuncs dispatch to optimized C inner-loops based on array dtype

Bryan Van de Ven: Introduction to NumPy

NumPy has many built-in ufuncs

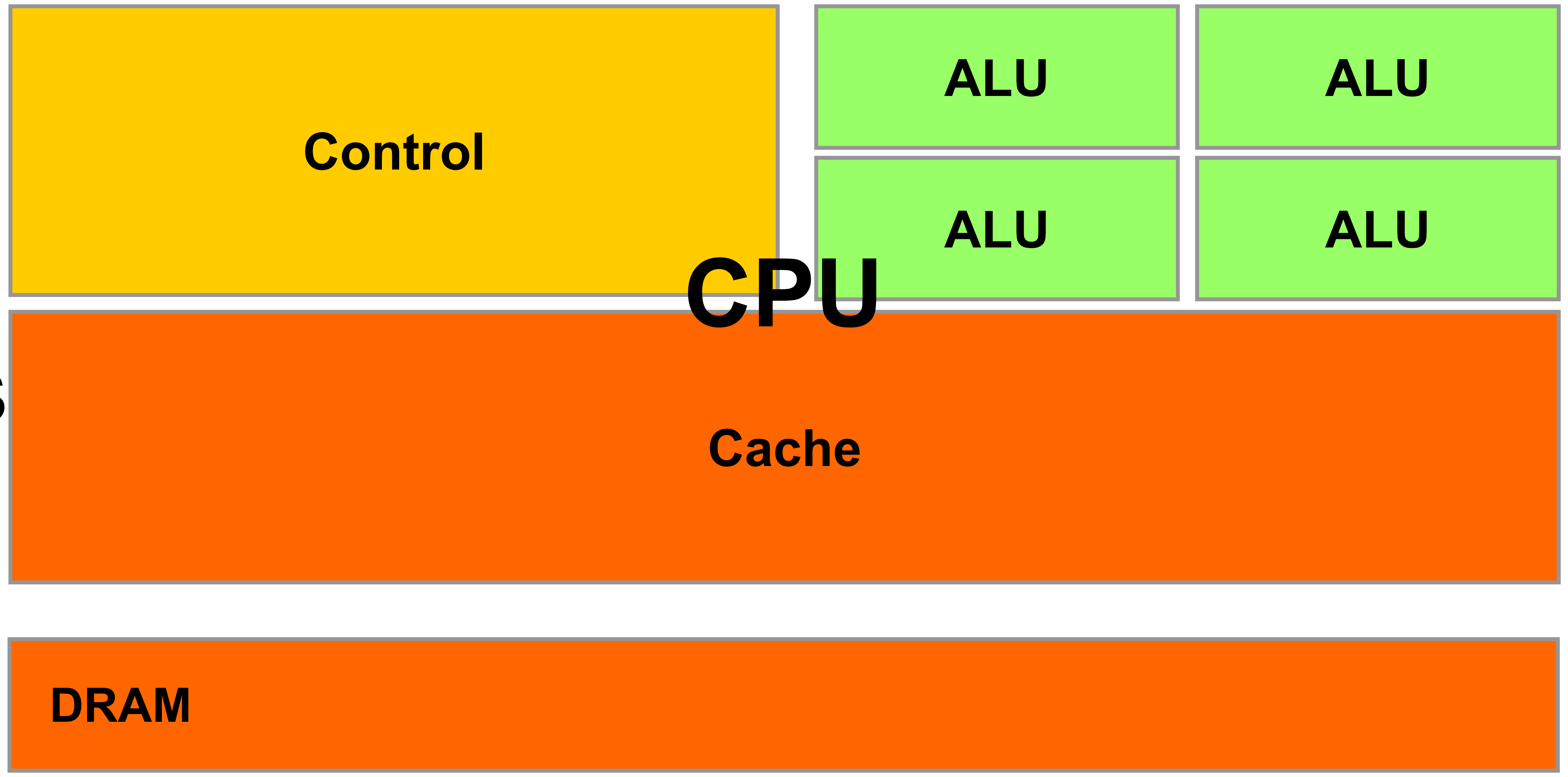
- comparison: <, <=, ==, !=, >=, >
- arithmetic: +, -, *, /, reciprocal, square
- exponential: exp, expm1, exp2, log, log10, log1p, log2, power, sqrt
- trigonometric: sin, cos, tan, acsin, arccos, atctan
- hyperbolic: sinh, cosh, tanh, acsinh, arccosh, atctanh
- bitwise operations: &, |, ~, ^, left_shift, right_shift
- logical operations: and, logical_xor, not, or
- predicates: isfinite, isinf, isnan, signbit
- other: abs, ceil, floor, mod, modf, round, sinc, sign, trunc

Bryan Van de Ven: Introduction to NumPy

Heterogeneous Parallel Computing

CPUs: Latency Oriented Design

- High clock frequency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
- Powerful ALUs
  - Reduced operation latency



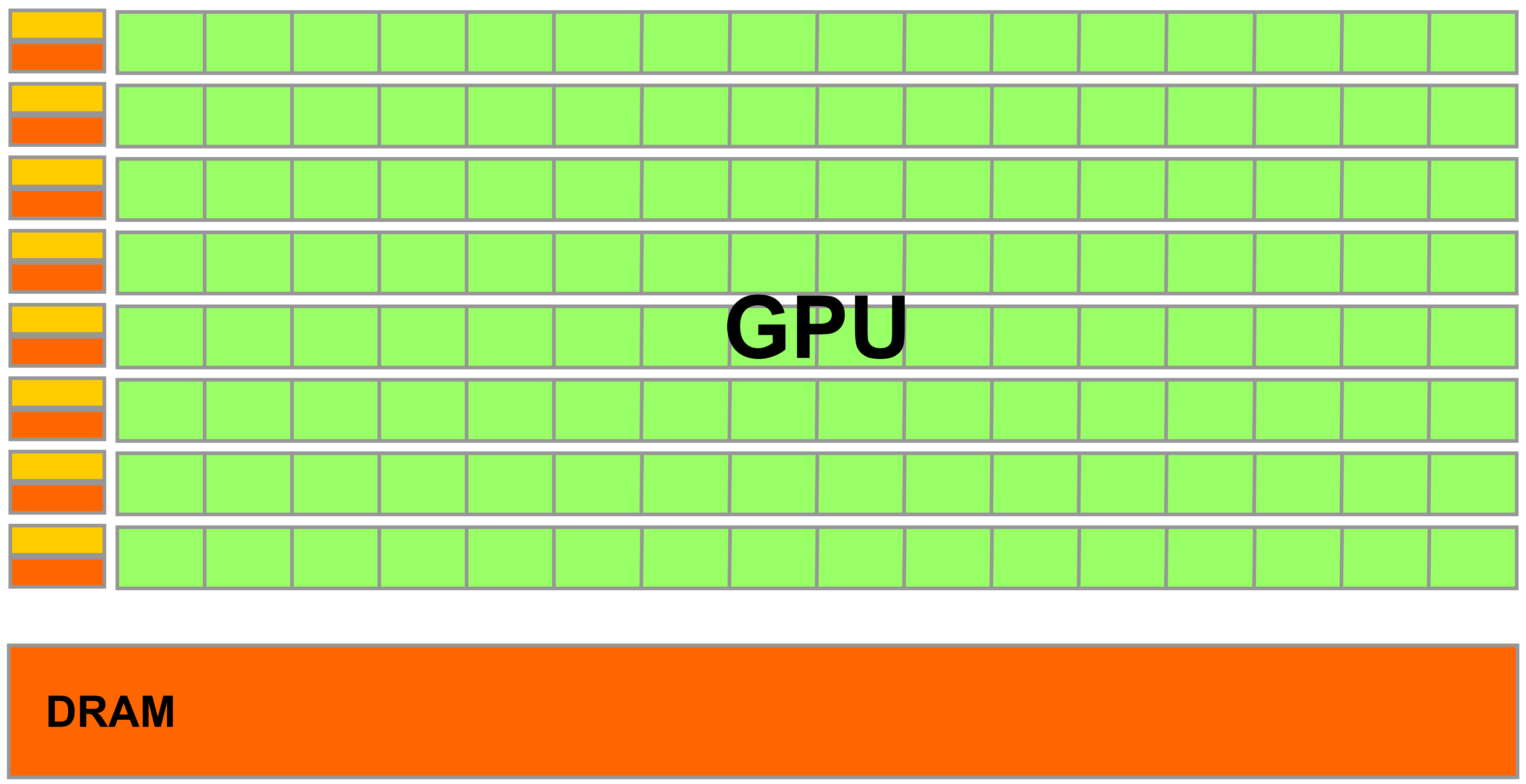
https://www.computerhope.com/jargon/b/branch-prediction.htm

https://en.wikipedia.org/wiki/Memory_latency

GPUs: Throughput Oriented Design

- Moderate clock frequency
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput

- Require massive number of threads to tolerate latencies



Applications Benefit from Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10+X faster than CPUs for parallel code

Winning Strategies Use Both CPU and GPU

- CPUs for sequential parts where latency hurts
  - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10+X faster than CPUs for parallel code

Heterogeneous Parallel Computing are used in Many Application Domains

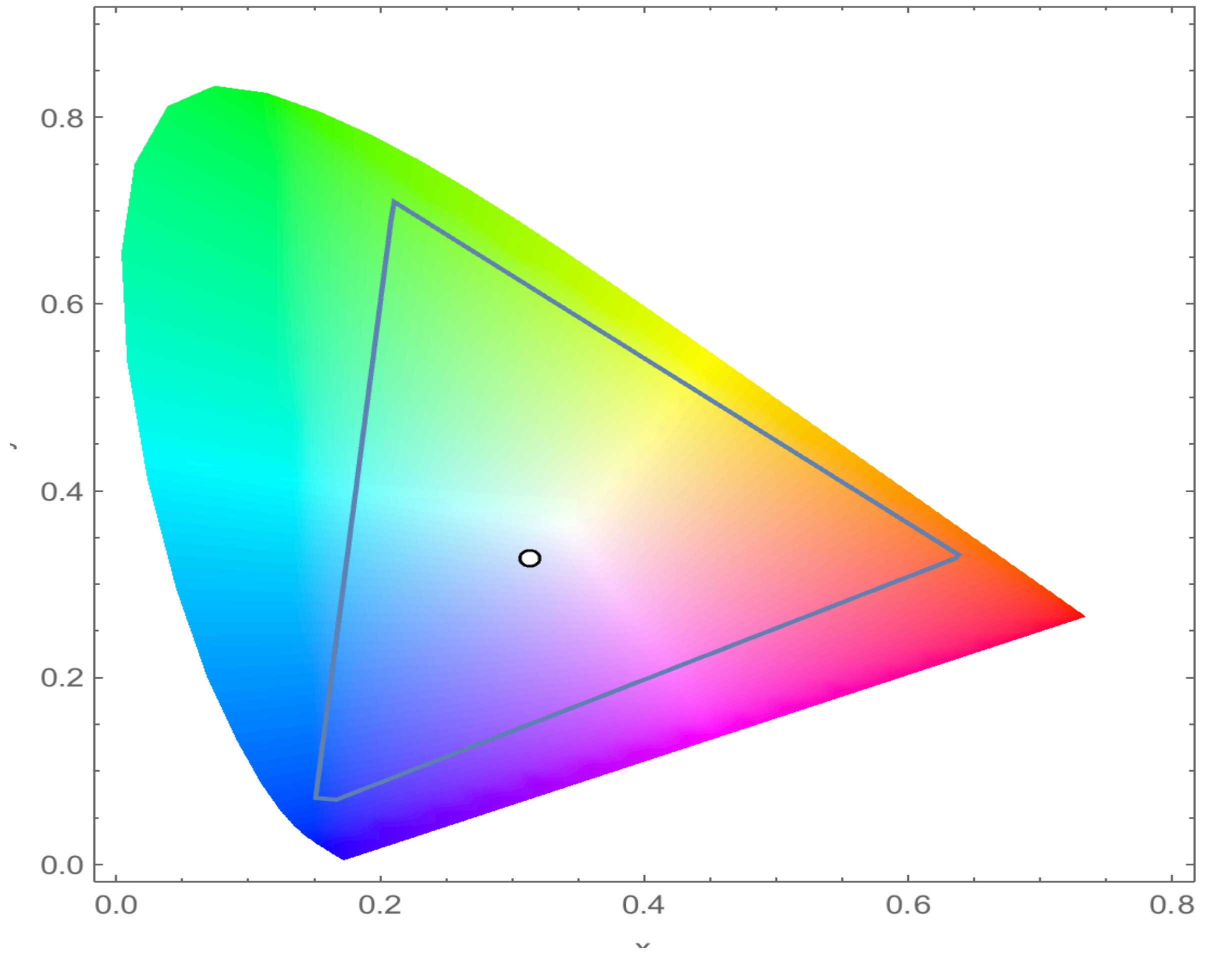
Financial Analysis	Scientific Simulation	Engineering Simulation	Data Intensive Analytics	Medical Imaging
Digital Audio Processing	Digital Video Processing	Computer Vision	Machine Learning	Electronic Design Automation
Biomedical Informatics	Statistical Modeling	Ray Tracing Rendering	Interactive Physics	Numerical Methods

Parallel Programming Work Flow

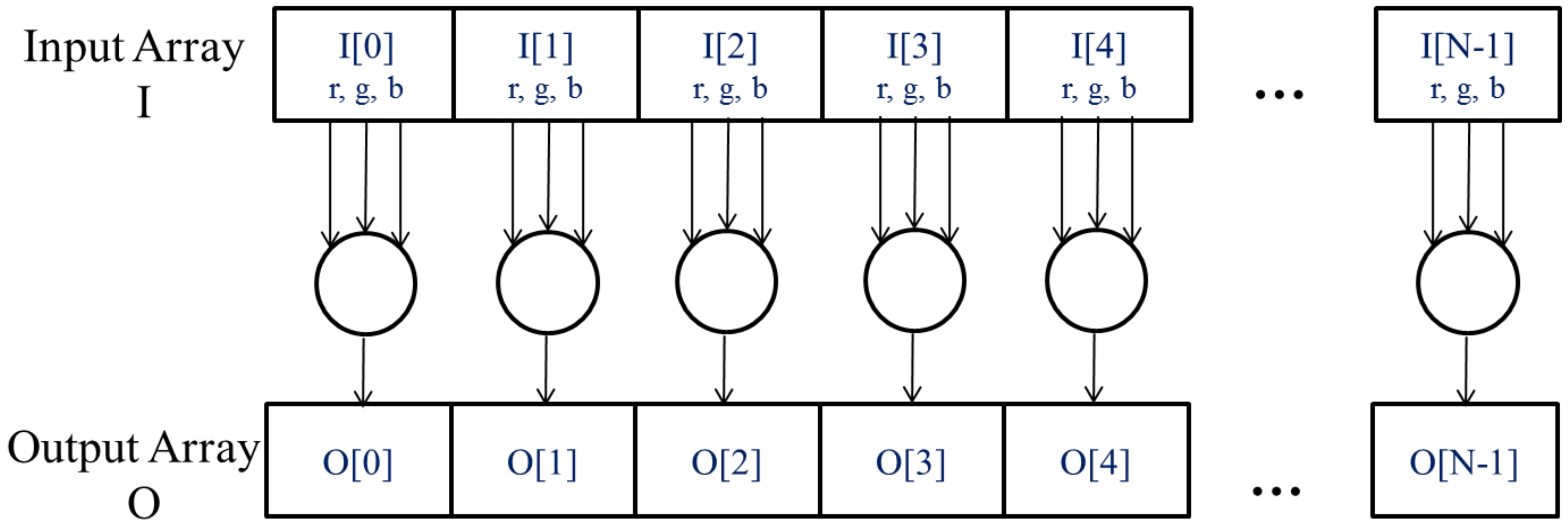
- Identify compute intensive parts of an application
- Adopt/create scalable algorithms
- Optimize data arrangements to maximize locality
- Performance Tuning
- Pay attention to code **portability**, **scalability**, and **maintainability**

Introduction to CUDA/PyCUDA

A Data Parallel Computation Example: Conversion of a color image to grey–scale image

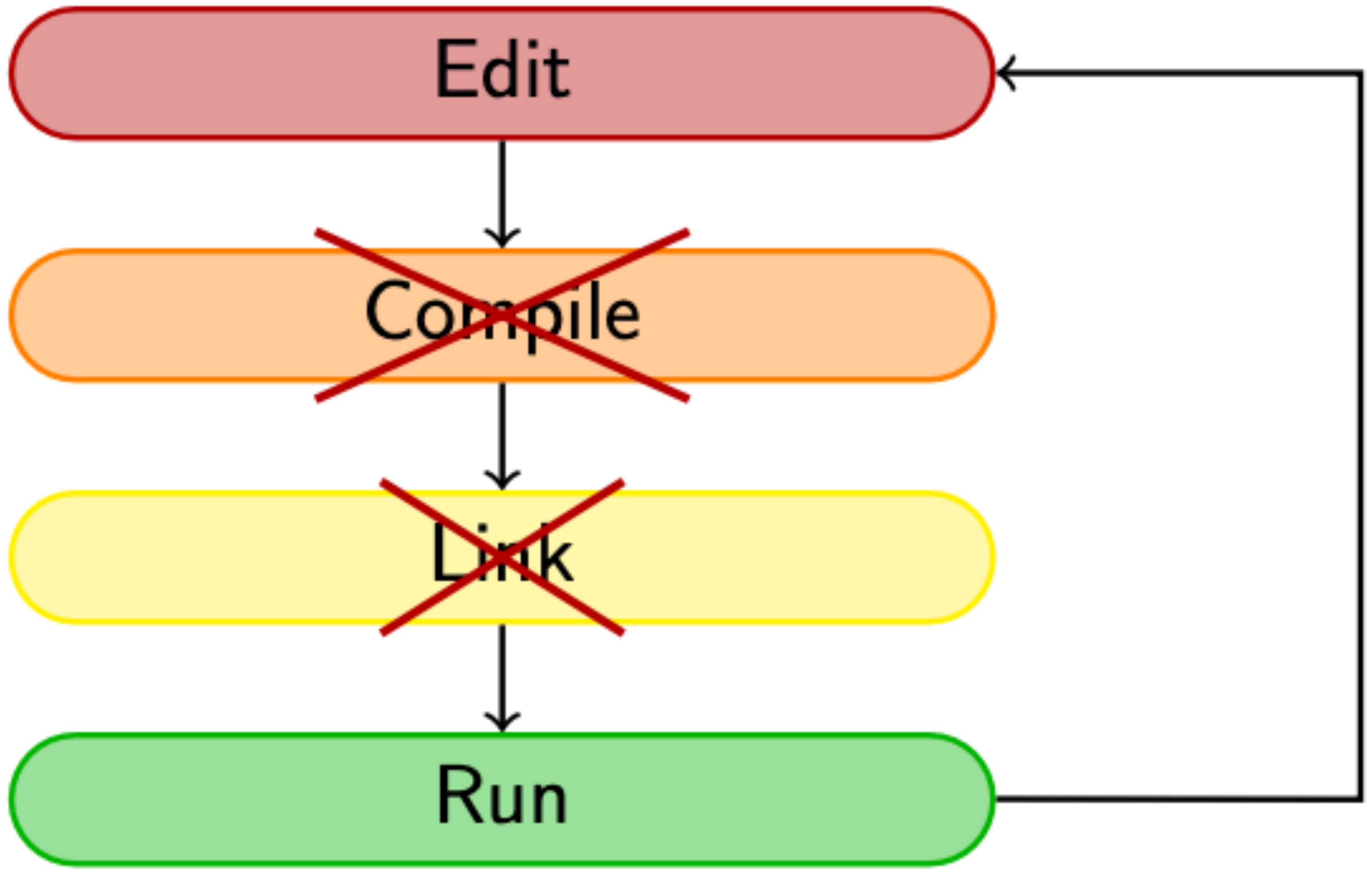


The pixels can be calculated independently of each other

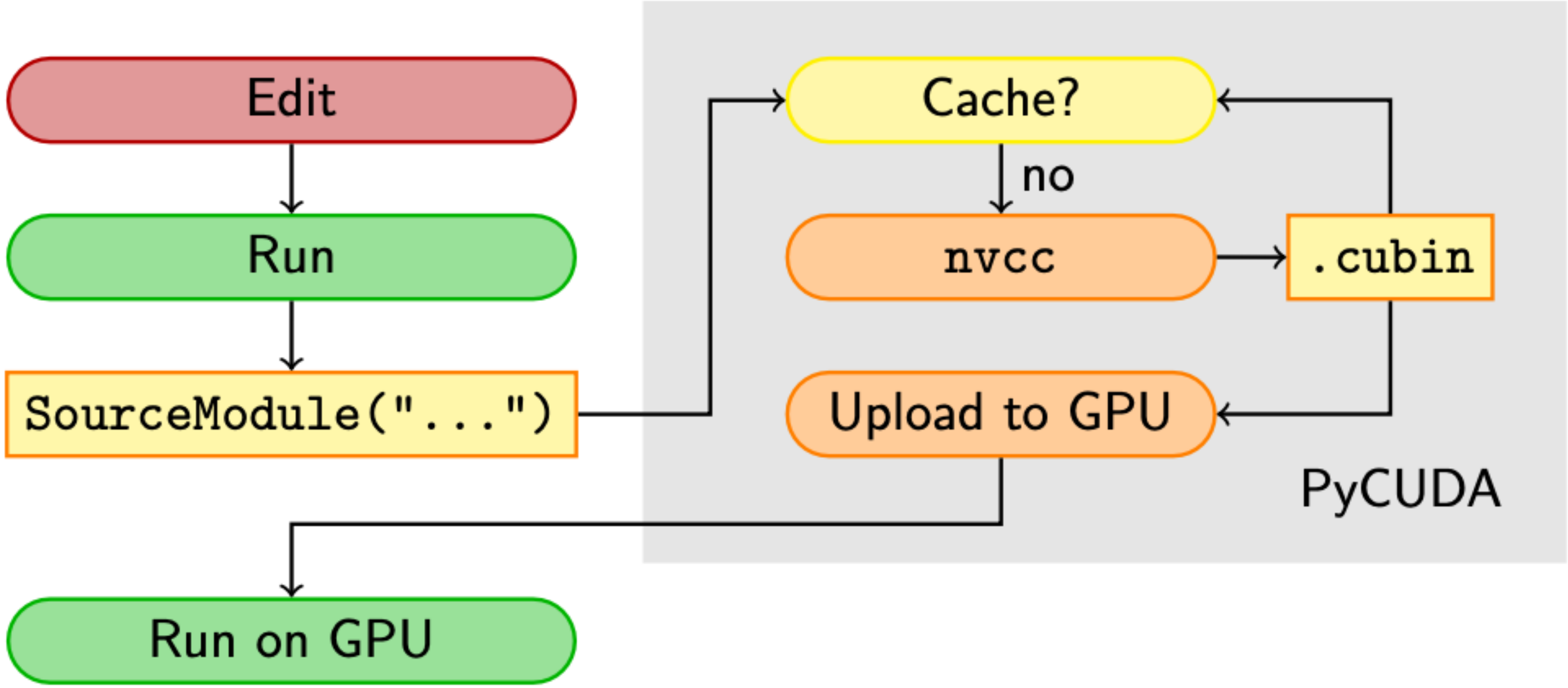


Scripting: Interpreted, not Compiled

Program creation workflow:



PyCUDA: Workflow



CUDA/OpenCL – Execution Model

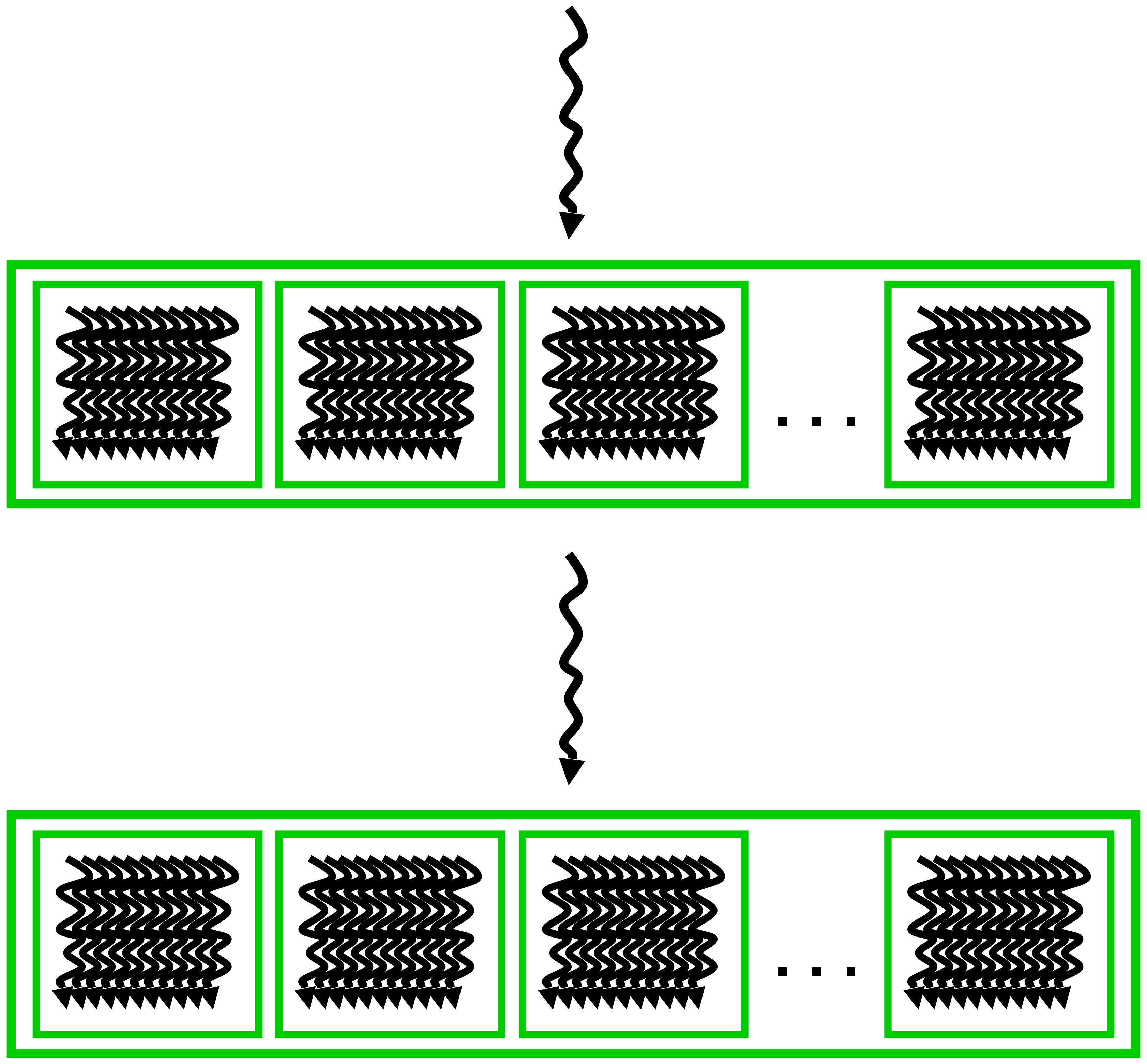
- Integrated host+device app C program
  - Serial or modestly parallel parts in **host C/Python** code
  - Highly parallel parts in **device** SIMD (Single Instruction Multiple Data) kernel C code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

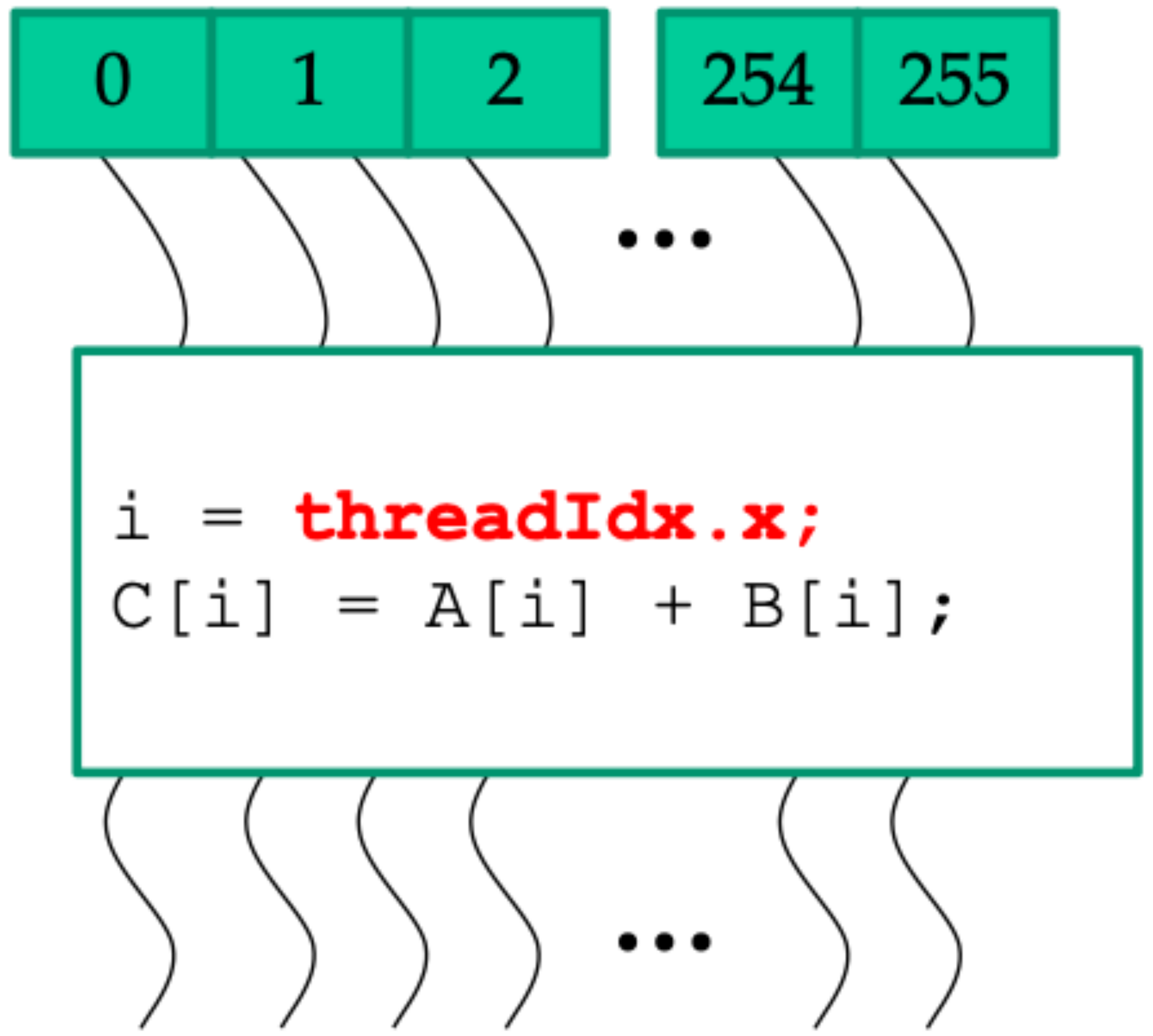
Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);



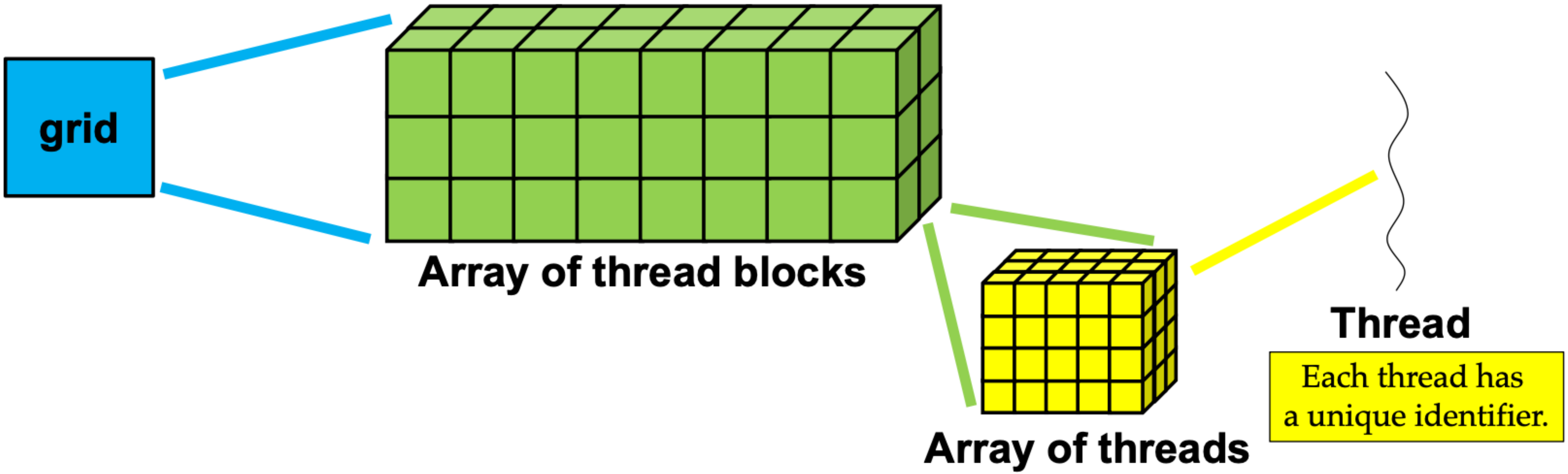
Arrays of Parallel Threads

- A CUDA kernel is executed as a grid (array) of threads
  - All threads in a grid run the same kernel code
  - Single Program Multiple Data (SPMD model)
  - Each thread has **a unique index** that it uses to compute memory addresses and make control decisions



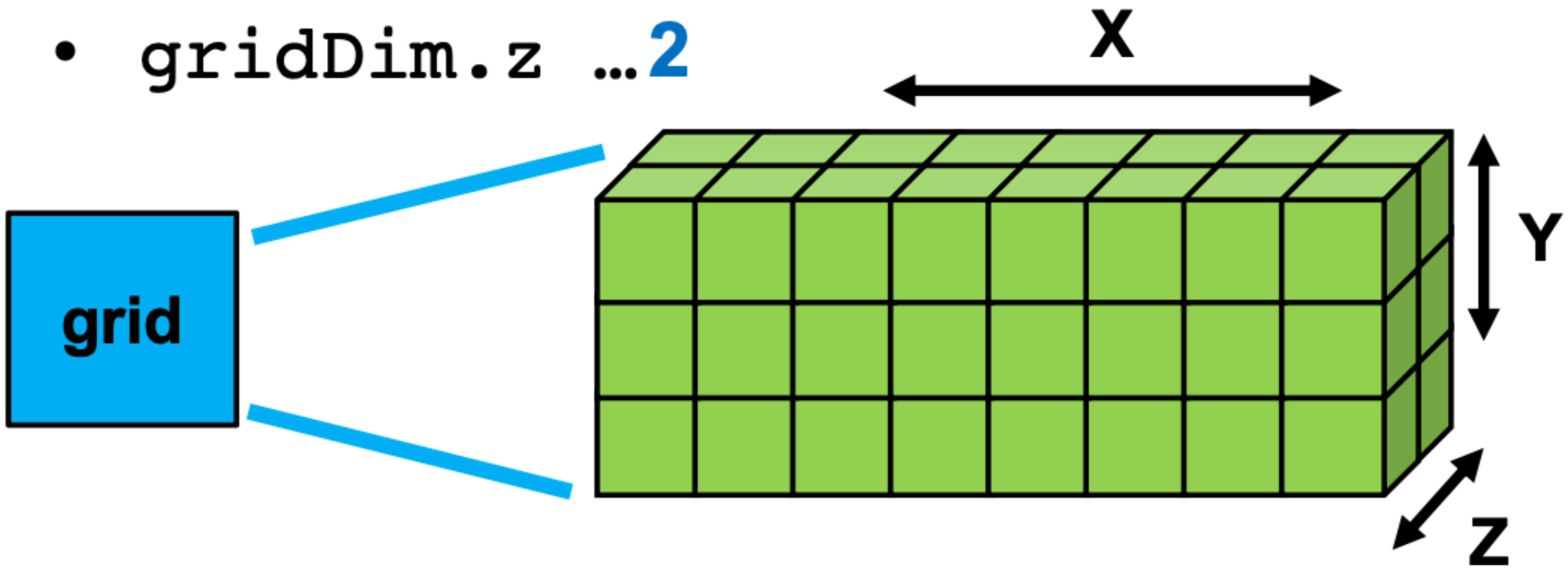
Logical Execution Model for CUDA

- Each CUDA kernel
  - is executed by a **grid**,
  - a 3D array of **thread blocks**, which are
  - 3D arrays of **threads**.



`gridDim` Gives Number of Blocks

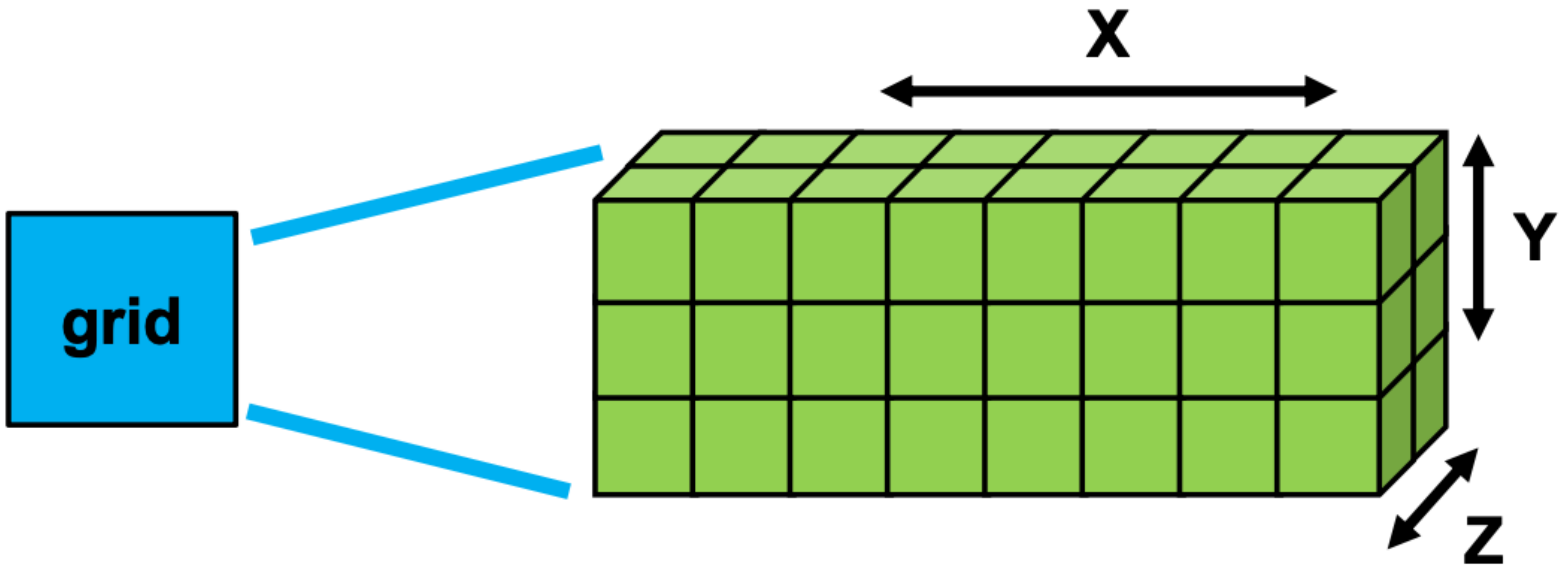
- Number of blocks in each dimension is
  - `gridDim.x` …8
  - `gridDim.y` …3
  - `gridDim.z` …2



For 1D (and 2D grids), simply use grid dimension 1 for Y (and Z).

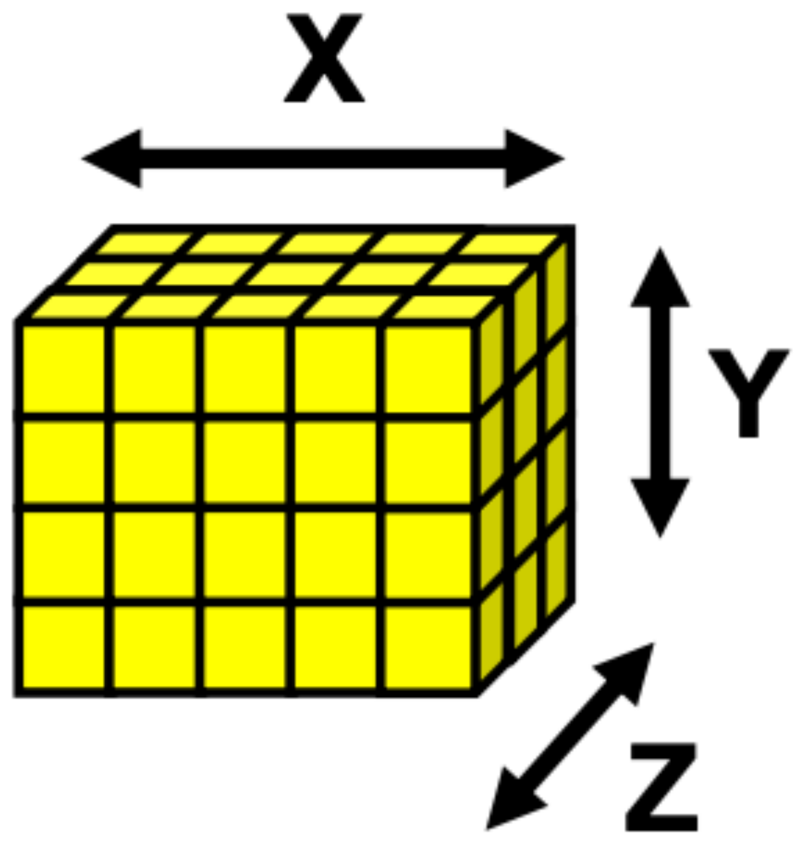
`blockIdx` is Unique for Each Block

- Each block has a unique index tuple
  - `blockIdx.x` (from 0 to (`gridDim.x` – 1))
  - `blockIdx.y` (from 0 to (`gridDim.y` – 1))
  - `blockIdx.z` (from 0 to (`gridDim.z` – 1))



`blockDim`: # of Threads per Block

- Number of blocks in each dimension is
  - `blockDim.x` … **5**
  - `blockDim.y` … **4**
  - `blockDim.z` … **3**



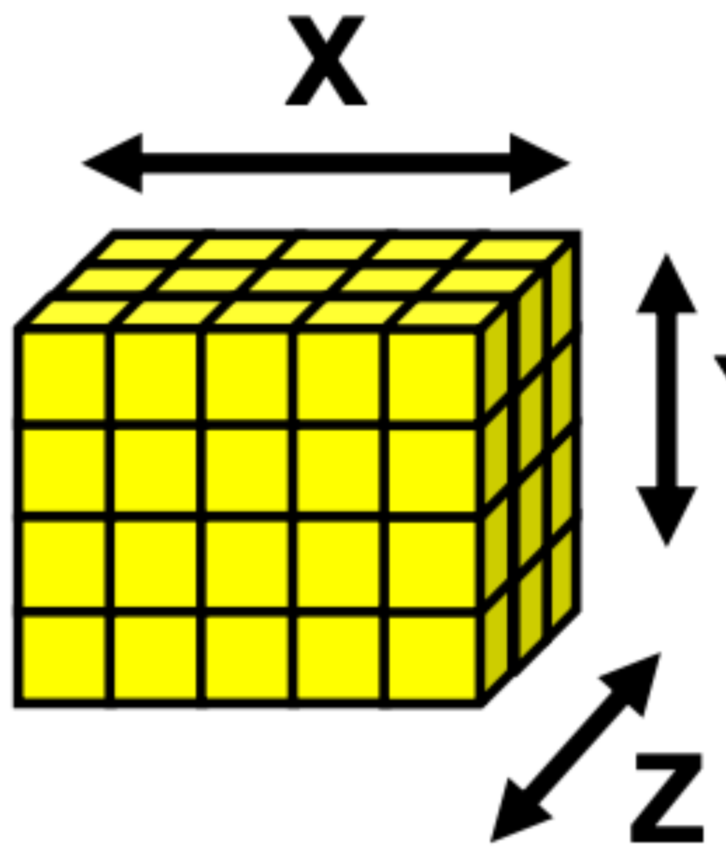
For 1D (and 2D blocks), simply use block dimension 1 for Y (and Z).

`threadIdx` Unique for Each Thread

- Each thread has a unique index tuple
  - `threadIdx.x` (from 0 to (`blockDim.x` – 1))
  - `threadIdx.y` (from 0 to (`blockDim.y` – 1))
  - `threadIdx.z` (from 0 to (`blockDim.z` – 1))

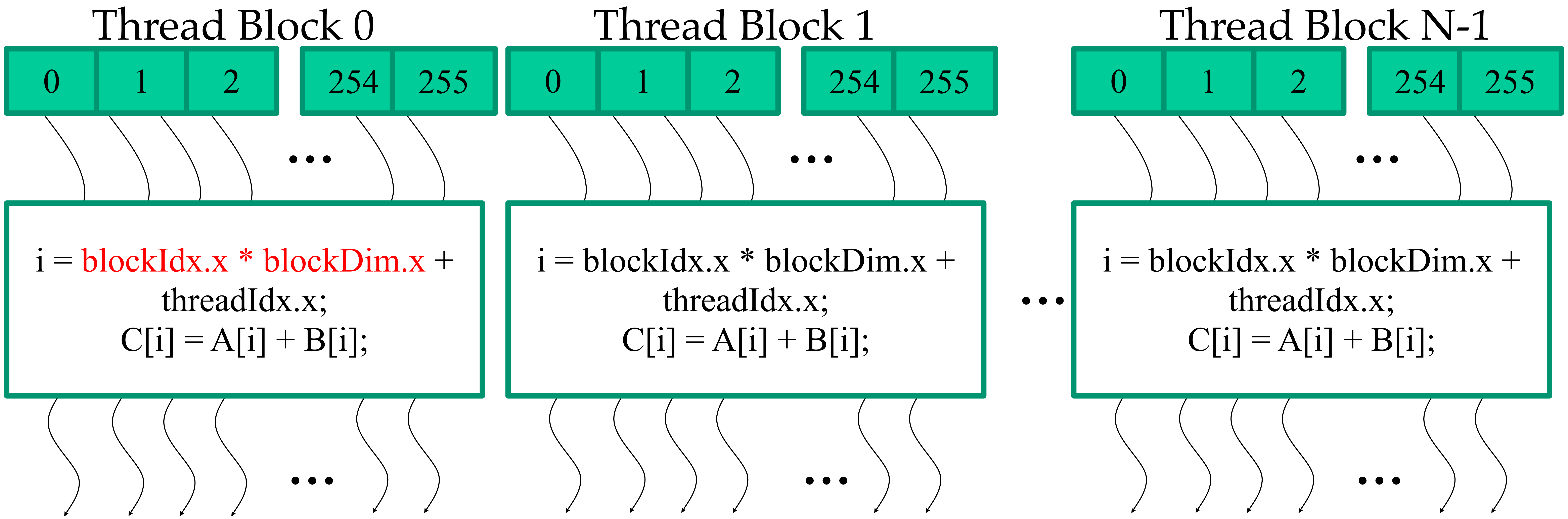
`threadIdx` tuple is unique to each thread
WITHIN A BLOCK.

`threadIdx` and `blockIdx` is unique to each thread
WITHIN A GRID.



Thread Blocks: Scalable Cooperation

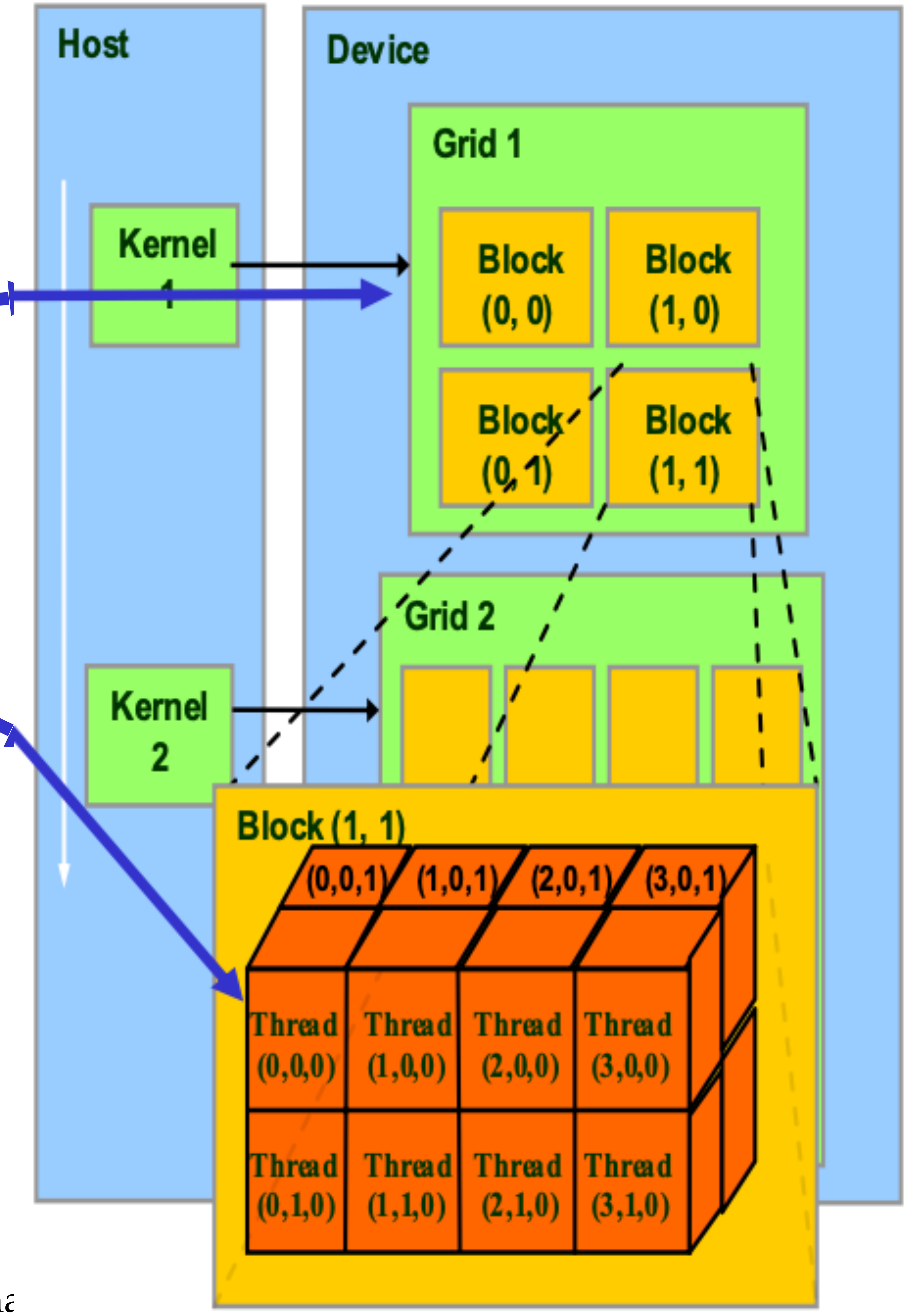
- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cooperate less (later)



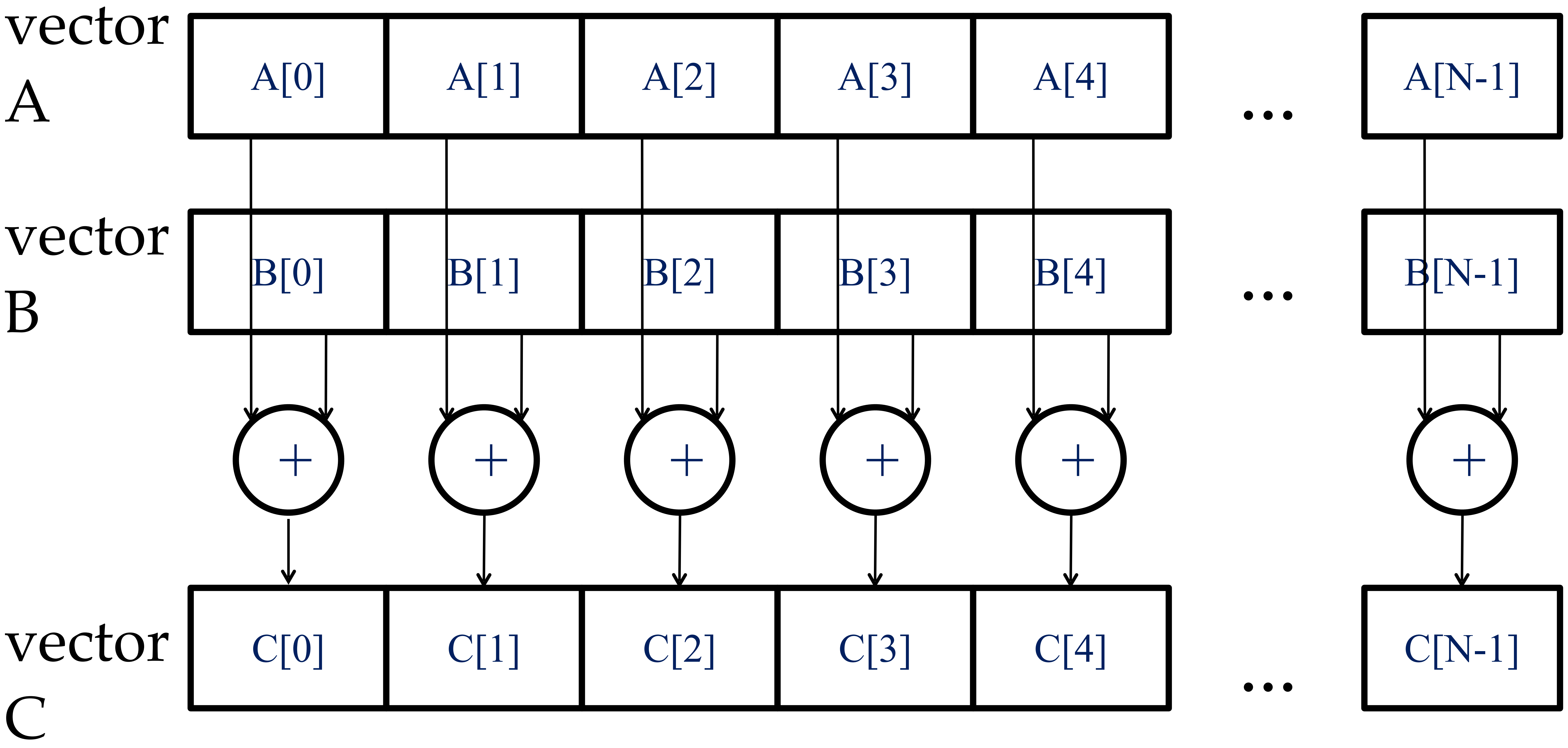
blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D
    threadIdx: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Vectors, matrices, tensors
  - Solving PDEs on volumes
  - …



Vector Addition – Conceptual View



Doubling the array

Whetting your appetite

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)
```

[This is `examples/demo.py` in the PyCUDA distribution.]

https://github.com/inducer/pycuda/blob/main/examples/demo.py

Doubling the array

Whetting your appetite

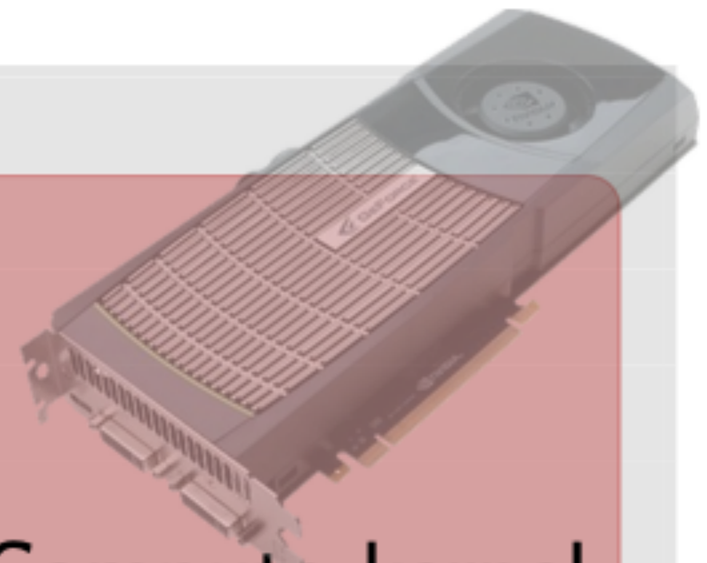
```python
mod = cuda.SourceModule("""
    __global__ void twice( float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```


https://github.com/inducer/pycuda/blob/main/examples/demo.py

Doubling the array

Whetting your appetite

```
mod = cuda.SourceModule("""
    __global__ void twice( float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```



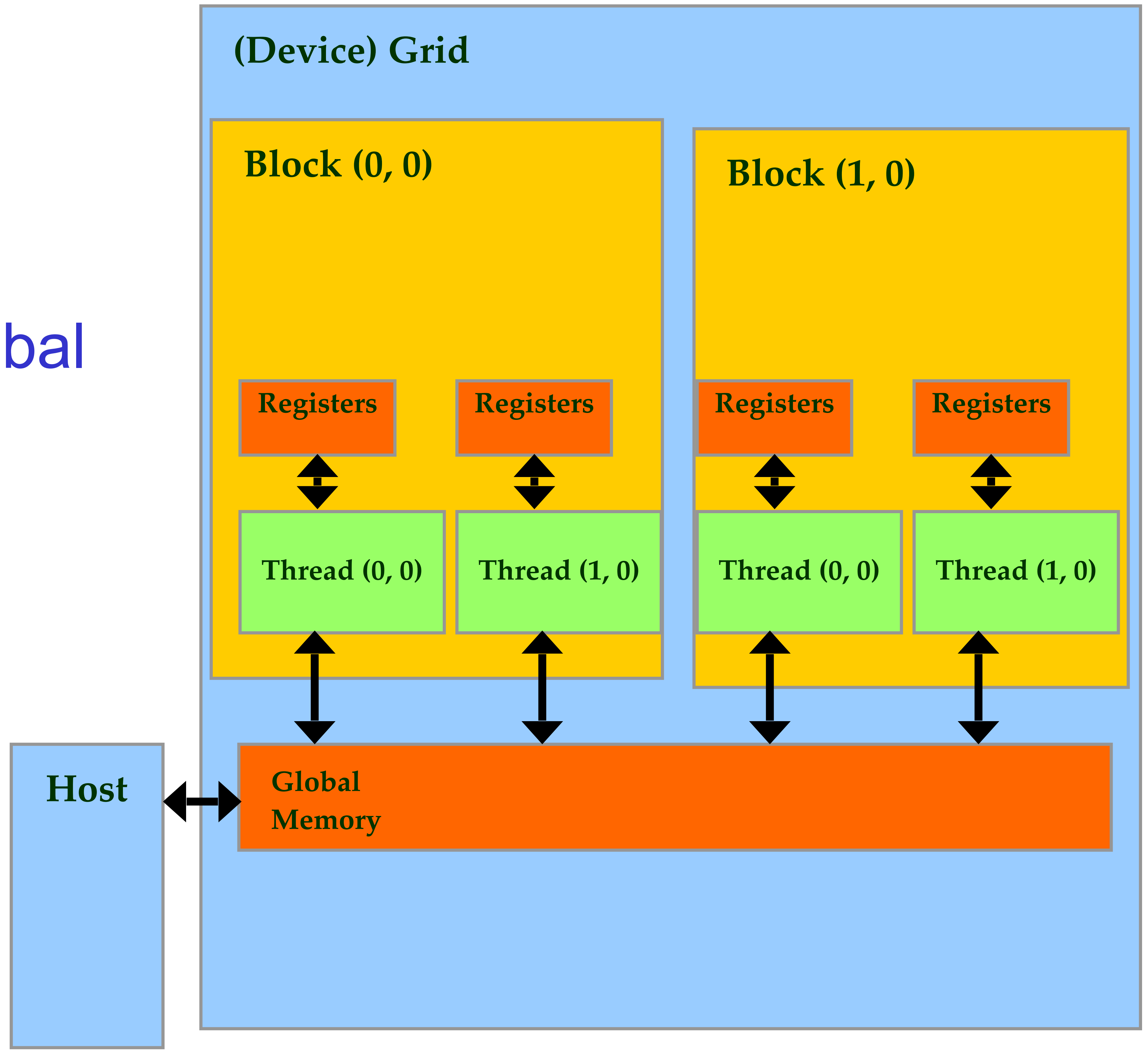
Compute kernel

https://github.com/inducer/pycuda/blob/main/examples/demo.py

Partial Overview of CUDA Memories

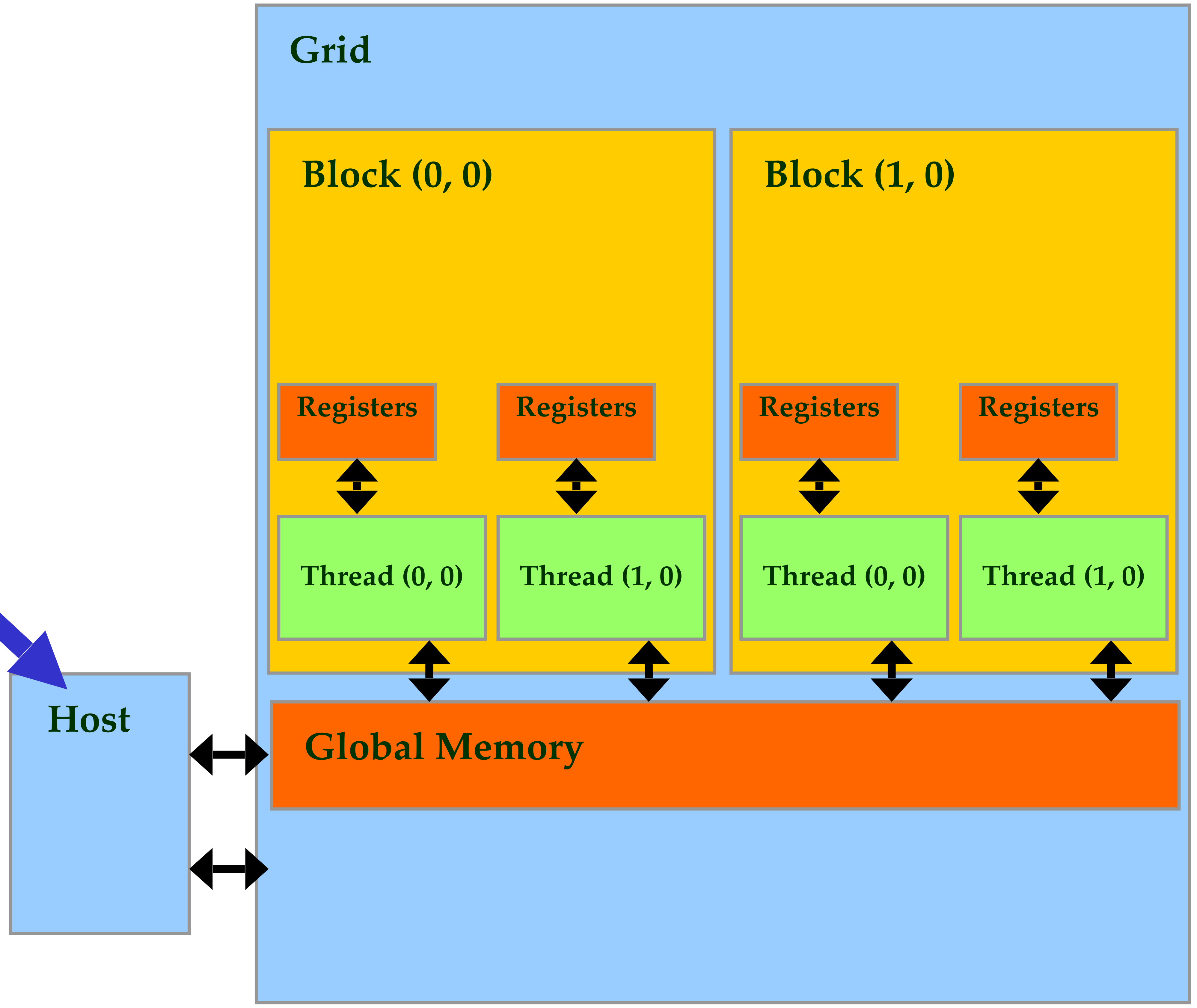
- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory
- Host code can
  - Transfer data to/from per grid global memory

We will cover more later.



CUDA Device Memory Management API functions

- cudaMalloc()
  - Allocates object in the device <u>global memory</u>
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** the allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - **Pointer** to freed object



The End