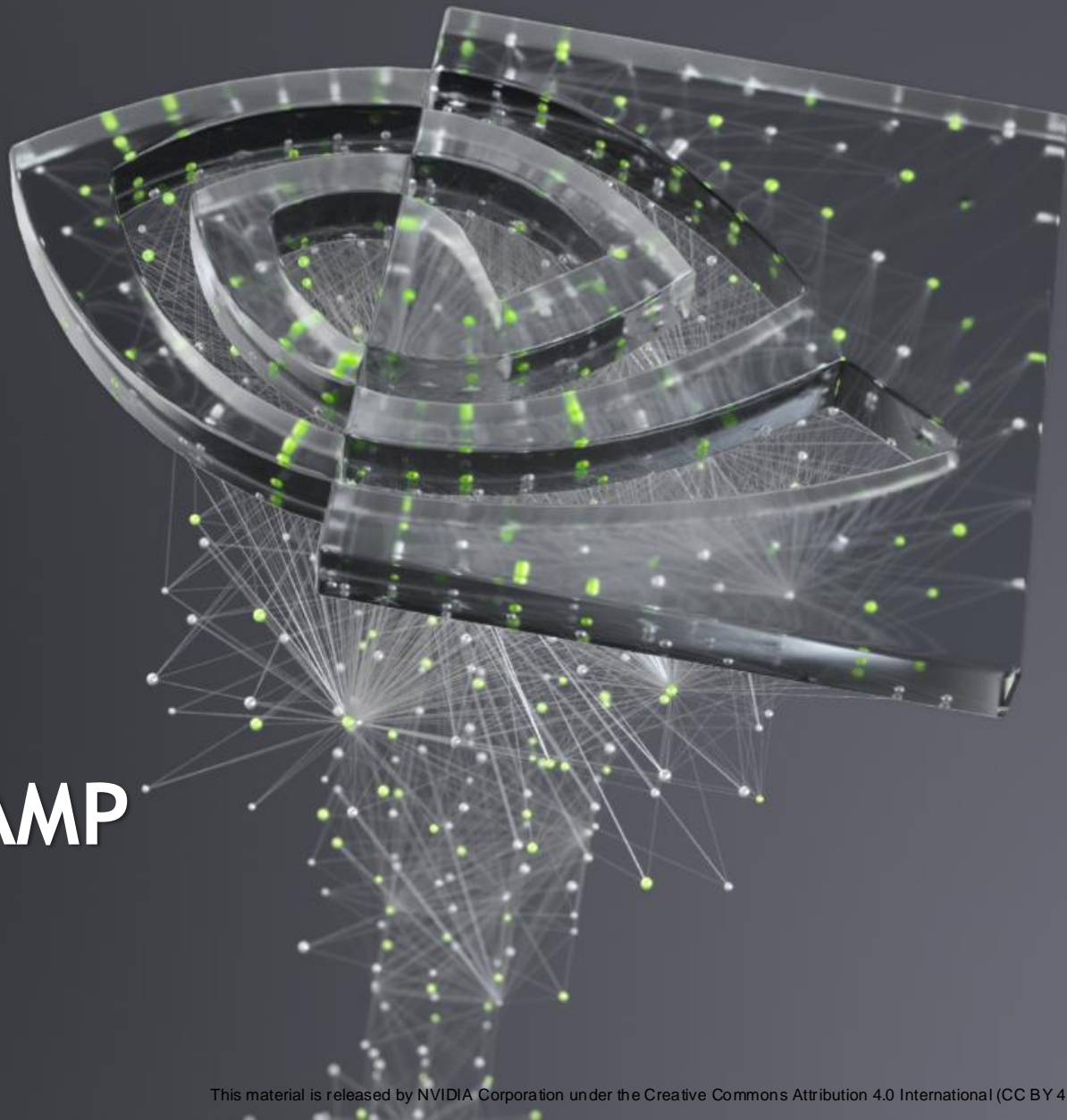




# N-WAYS GPU BOOTCAMP

## NUMBA FOR CUDA GPU

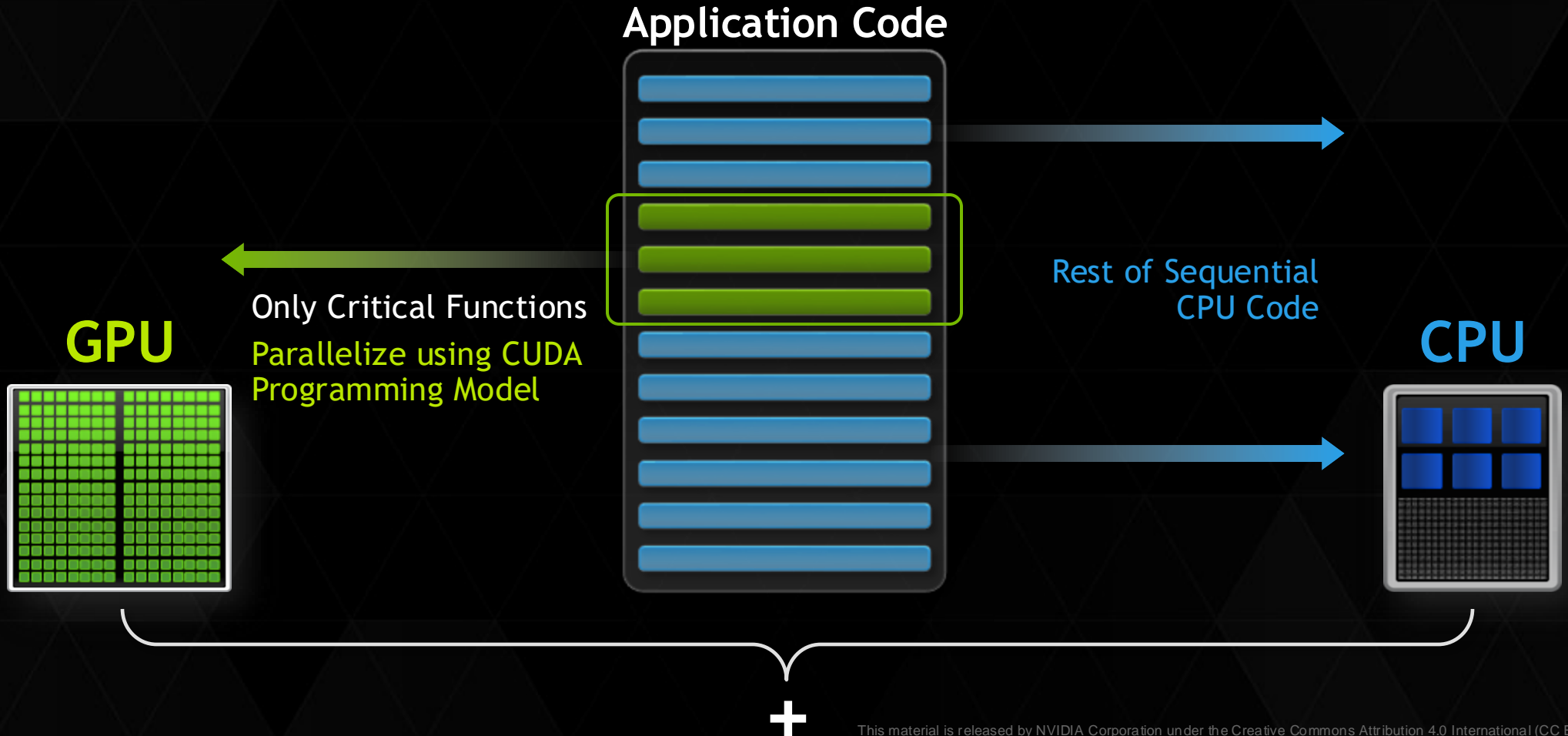


# NUMBA FOR CUDA GPU

## What to expect?

- Basic introduction to GPU Architecture
- GPU Memory and Programming Model
- Overview of Numba
- CUDA Kernels
- Memory Management
- Atomic Operation
- CUDA Ufuncs
- Summary

# GPU COMPUTING





# CUDA ARCHITECTURE PROGRAMMING MODEL

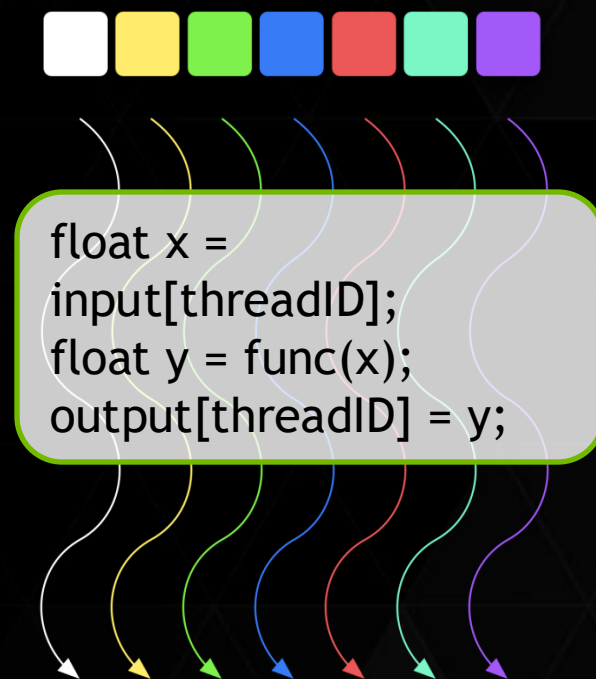
# CUDA KERNELS

- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads
- CUDA threads:
  - Lightweight
  - Fast switching
  - Tens of thousands execute simultaneously

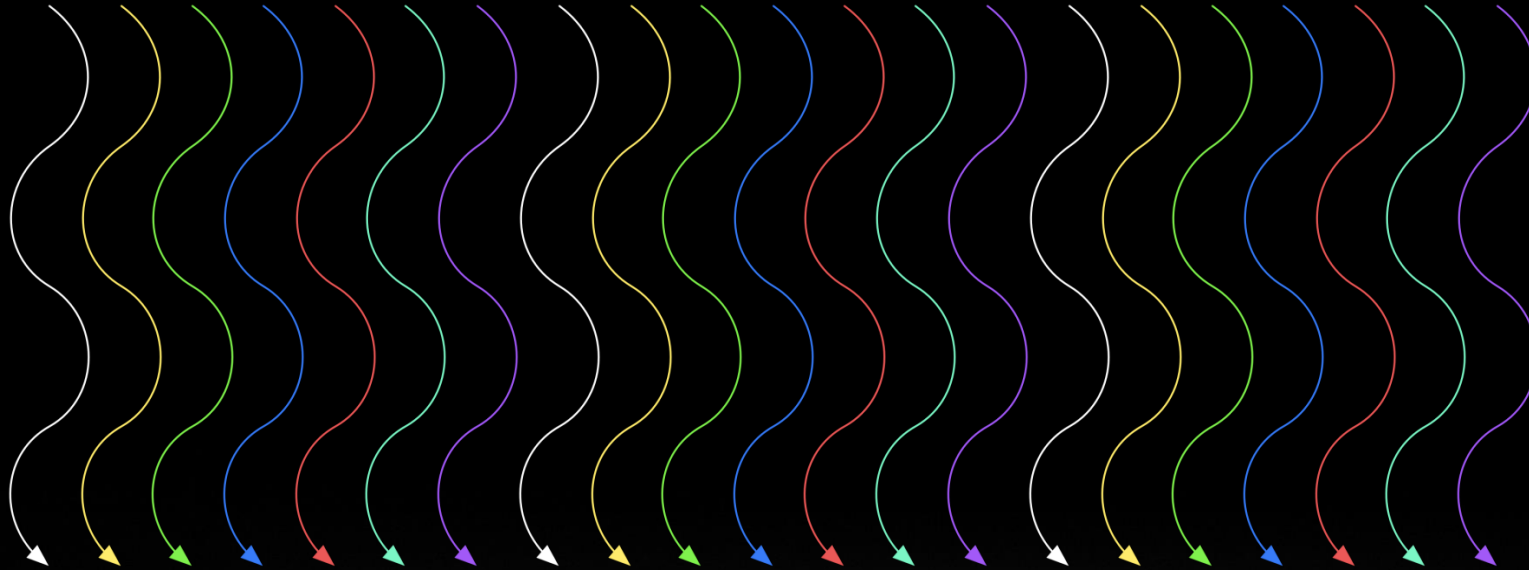
CPU	Host	Executes functions
GPU	Device	Executes kernels

# CUDA KERNELS: PARALLEL THREADS

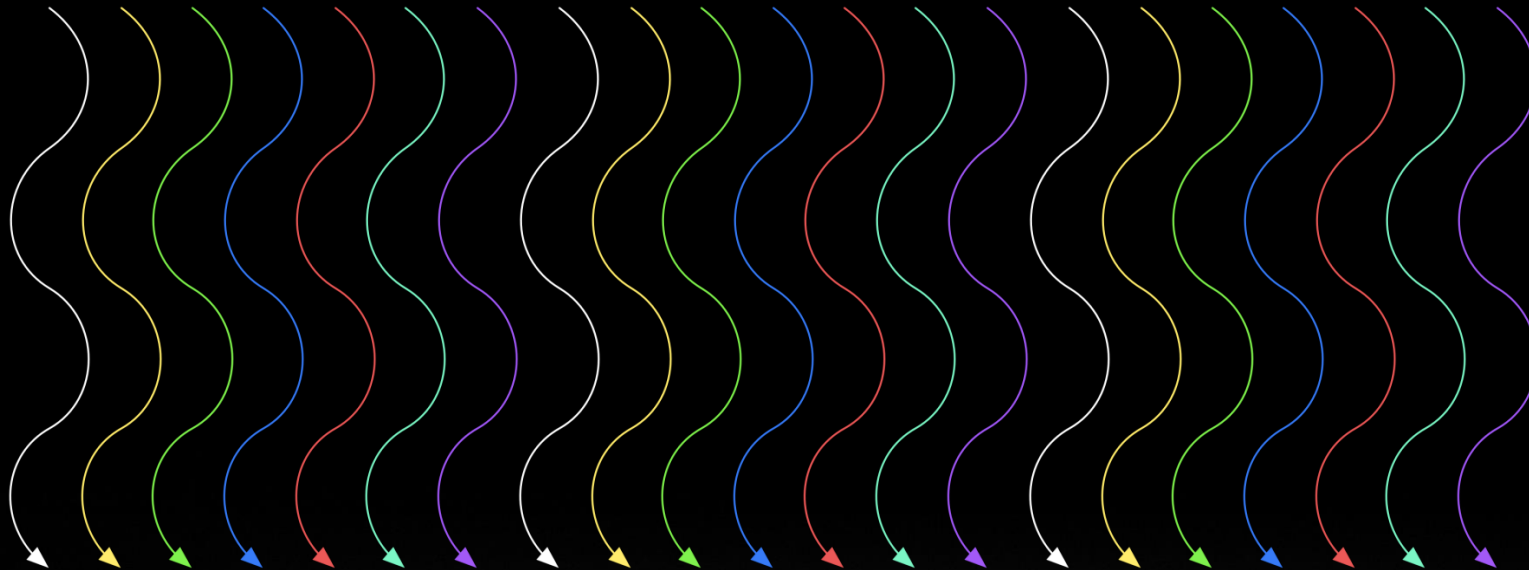
- A **kernel** is a function executed on the GPU
  - Array of threads, in parallel
- All threads execute the same code, can take different paths
  - Each thread has an ID
  - Select input/output data
  - Control decisions



# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



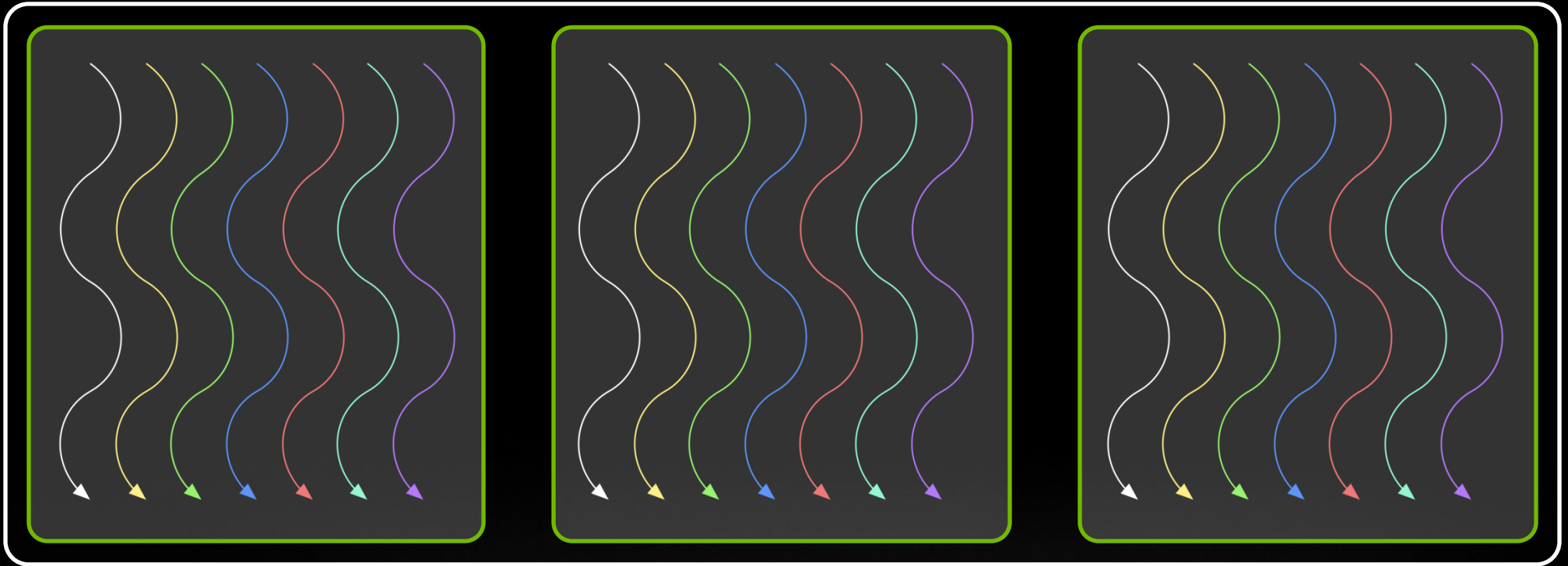
# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



- Threads are grouped into **blocks**

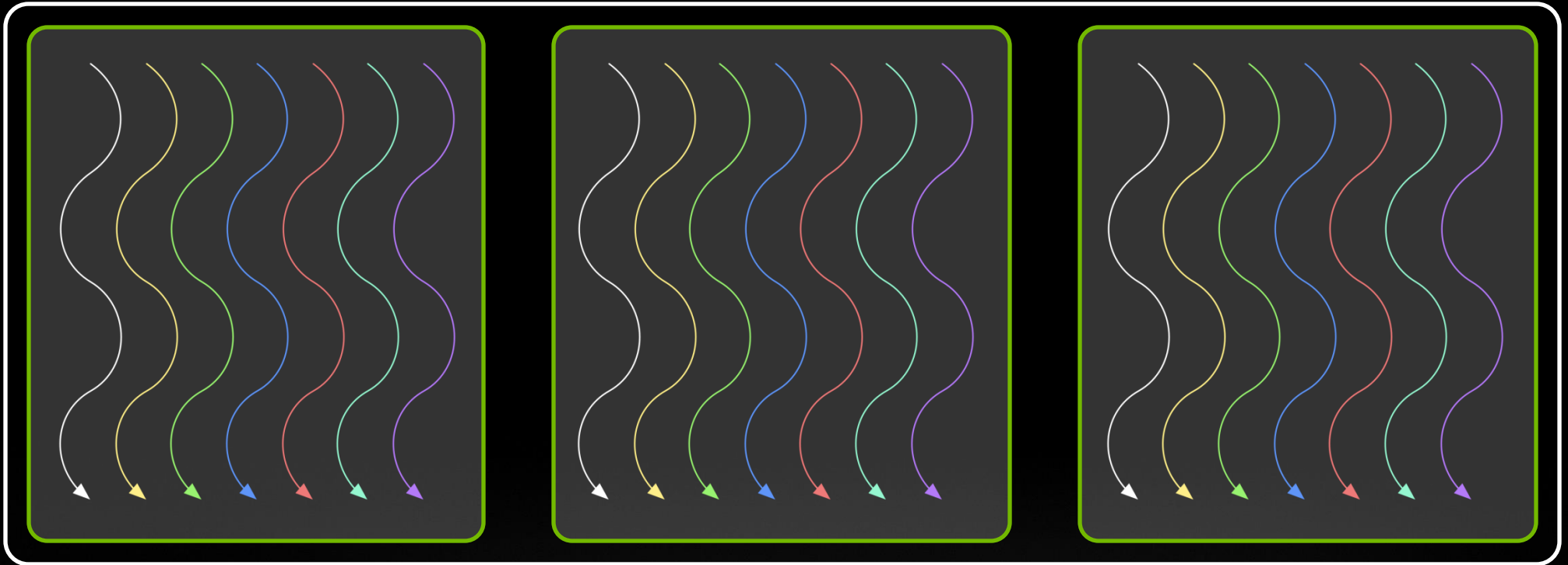


# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



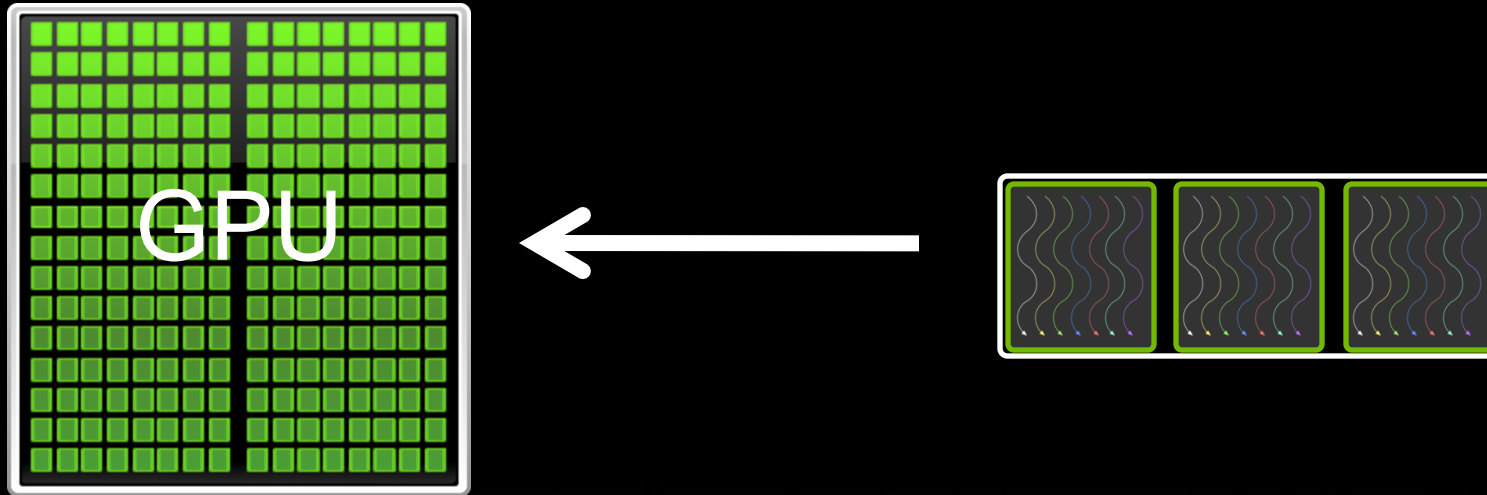
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



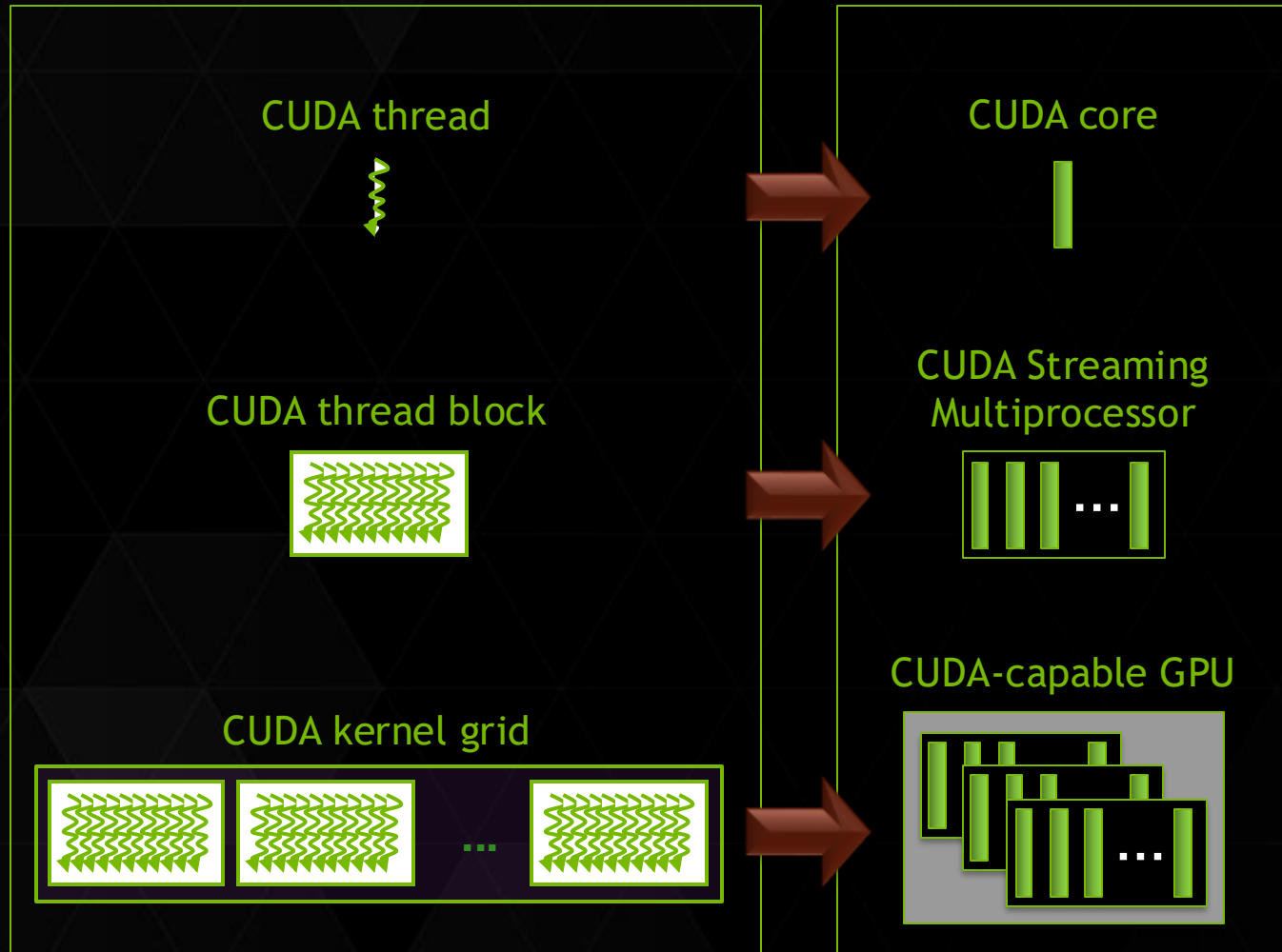
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**
- A **kernel** is executed as a **grid** of **blocks** of **threads**

# CUDA KERNELS: SUBDIVIDE INTO BLOCKS



- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**
- A **kernel** is executed as a **grid** of **blocks** of **threads**

# KERNEL EXECUTION

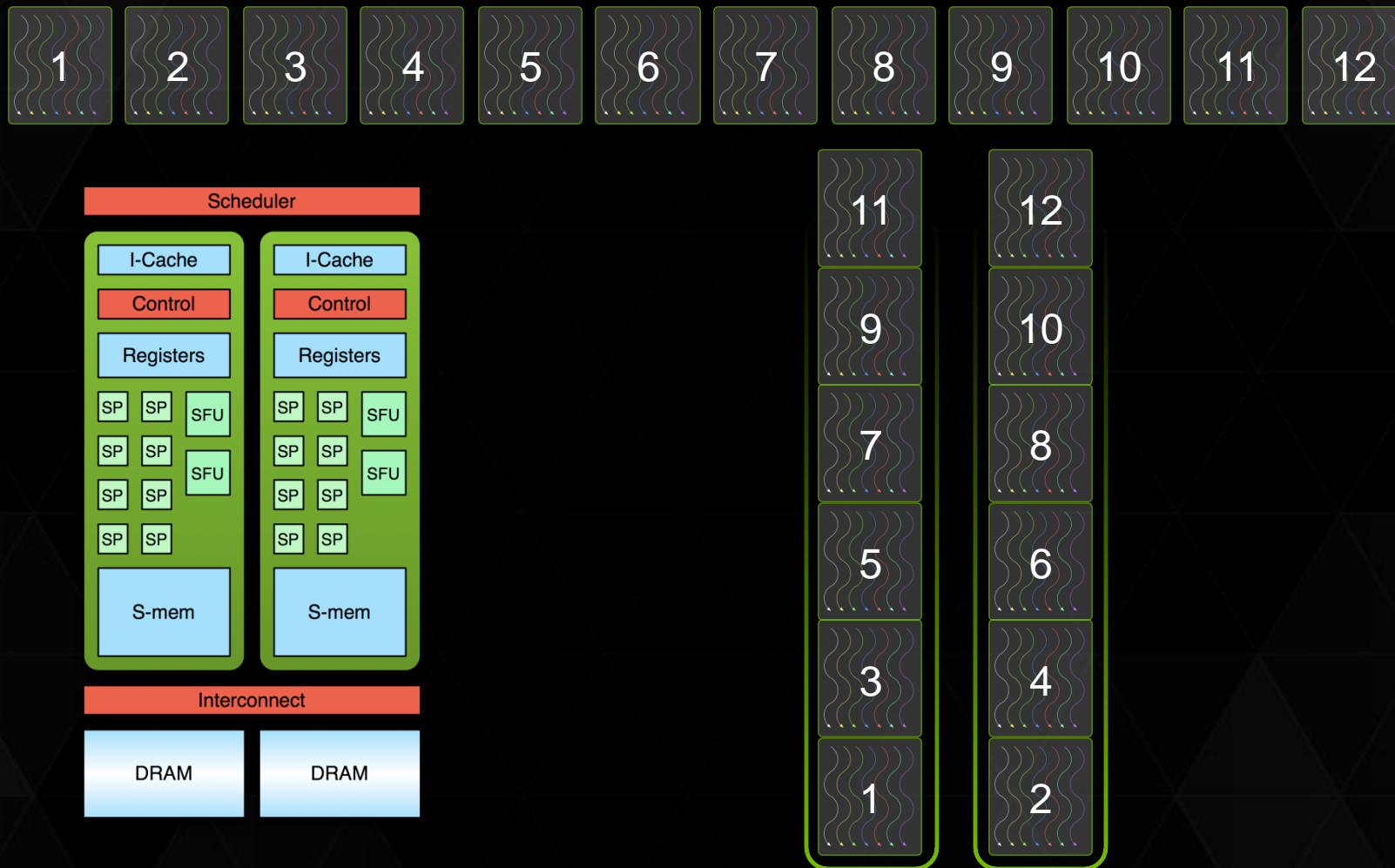


- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

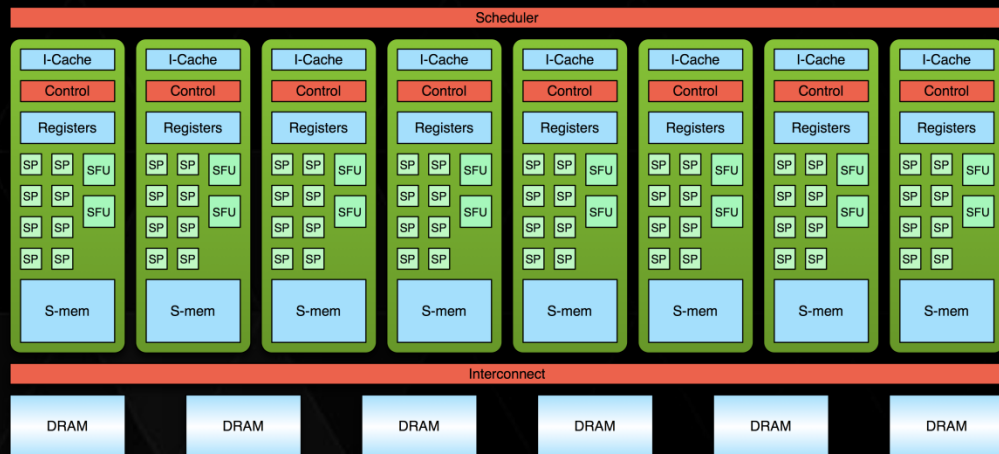
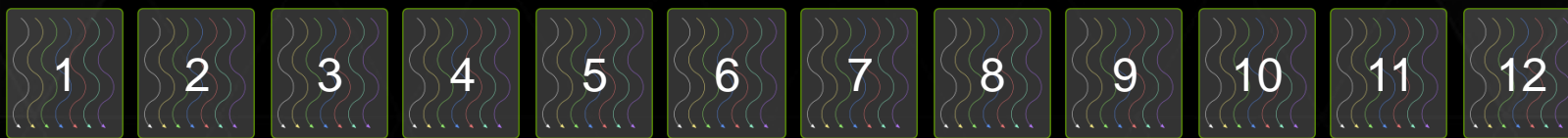
# COMMUNICATION WITHIN A BLOCK

- Threads may need to cooperate
  - Memory accesses
  - Share results
- Cooperate using **shared memory**
  - Accessible by all threads within a block
- Restriction to “within a block” permits scalability
  - Fast communication between  $N$  threads is not feasible when  $N$  large

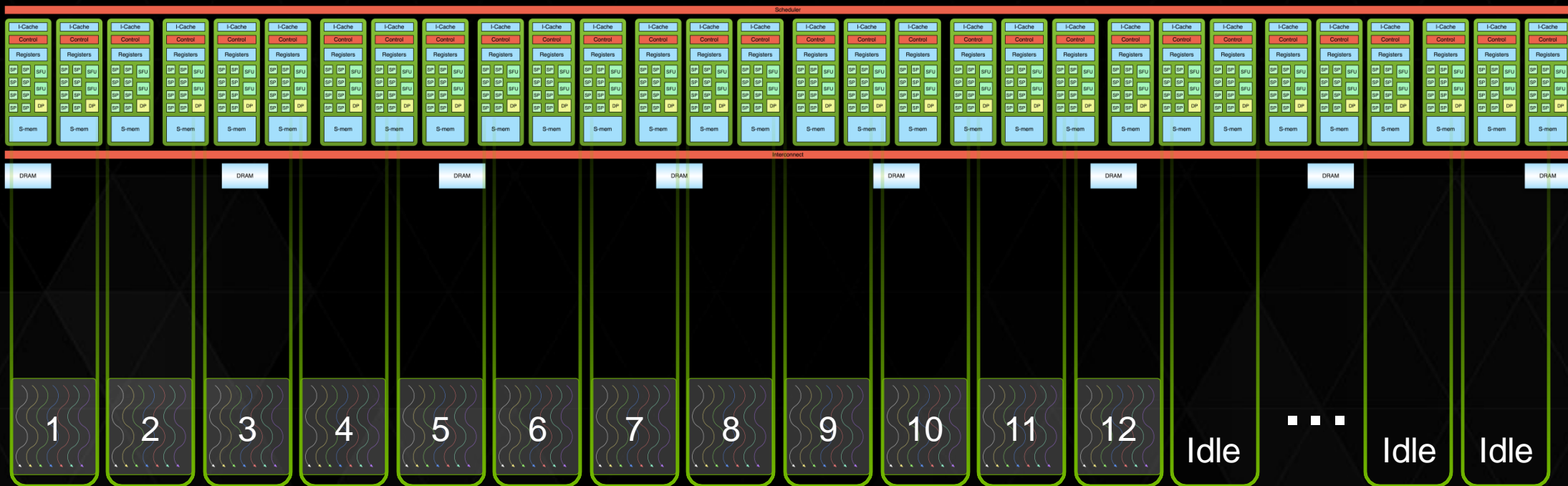
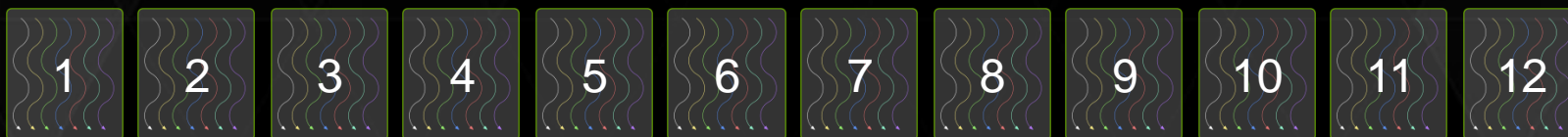
# TRANSPARENT SCALABILITY



# TRANSPARENT SCALABILITY



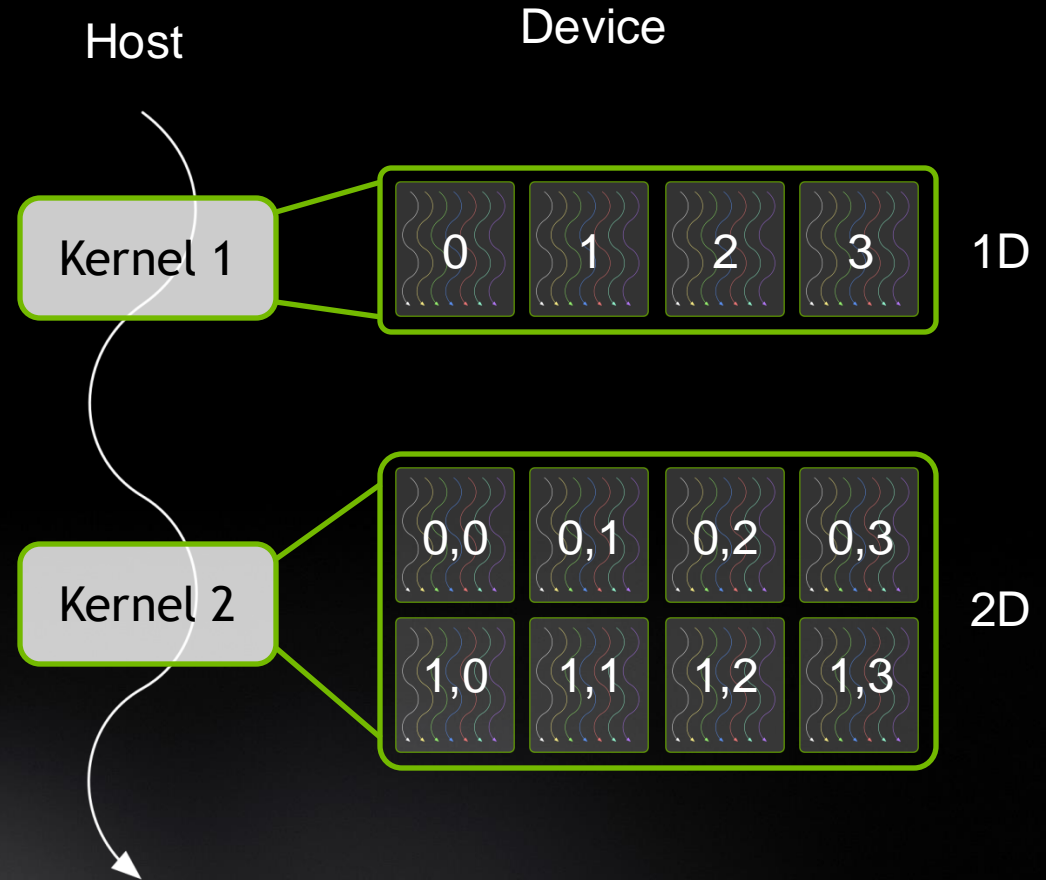
# TRANSPARENT SCALABILITY -





# CUDA PROGRAMMING MODEL - SUMMARY

- A kernel executes as a grid of thread blocks
- A block is a batch of threads
  - Communicate through shared memory
- Each block has a block ID
- Each thread has a thread ID





# CUDA ARCHITECTURE MEMORY MODEL

# GPU ARCHITECTURE

Two Main components

## Global memory

Analogous to RAM in a CPU server

Accessible by both GPU and CPU

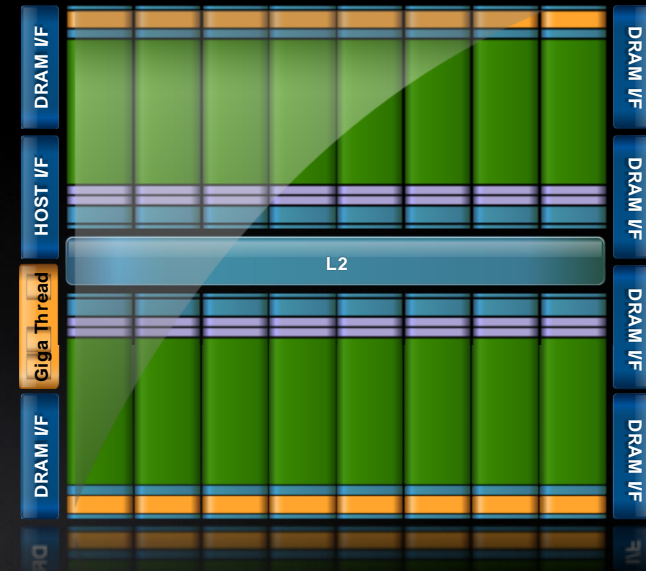
48GB with bandwidth currently up to 1 TB/s

## Streaming Multiprocessors (SMs)

SMs perform the actual computations

Each SM has its own:

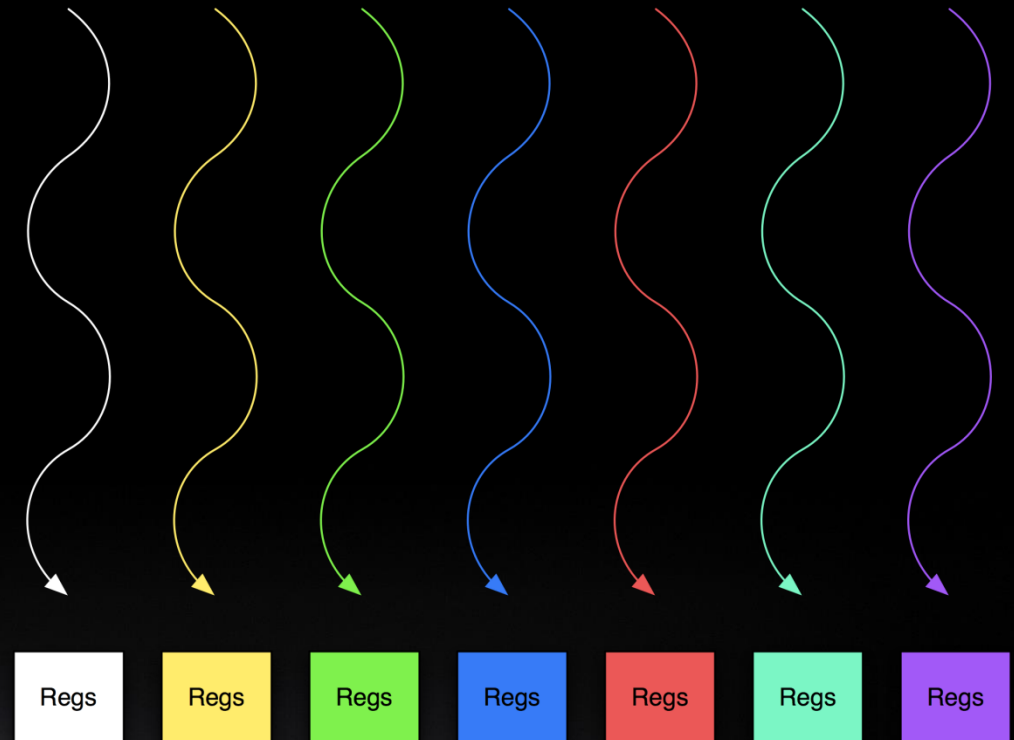
Control units, registers, execution pipelines, caches



# MEMORY HIERARCHY

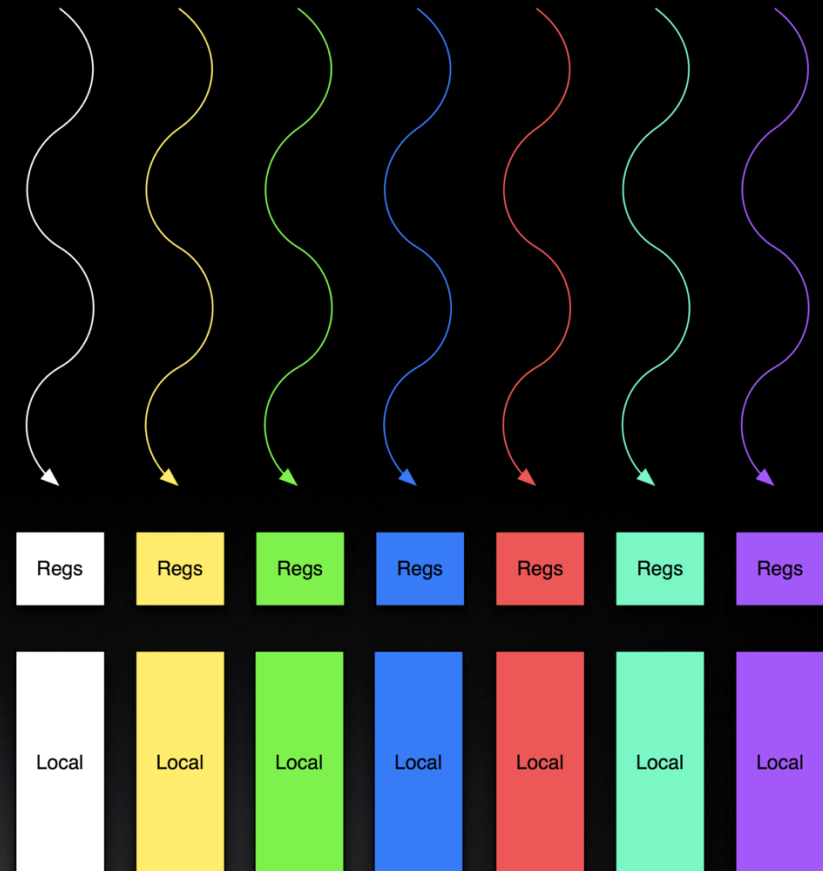
▸ Thread

▸ Registers



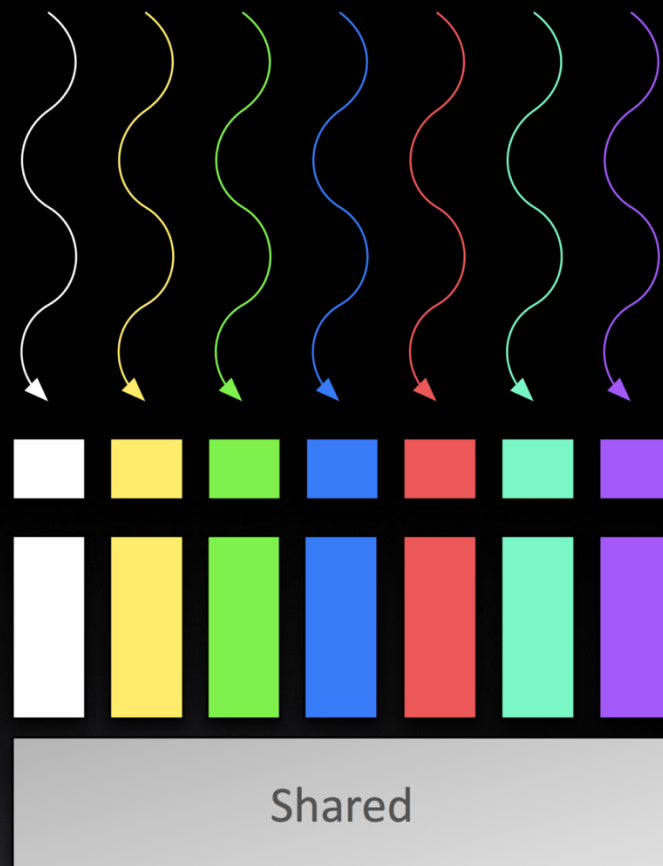
# MEMORY HIERARCHY

- Thread
  - Registers
- Thread
  - Local memory



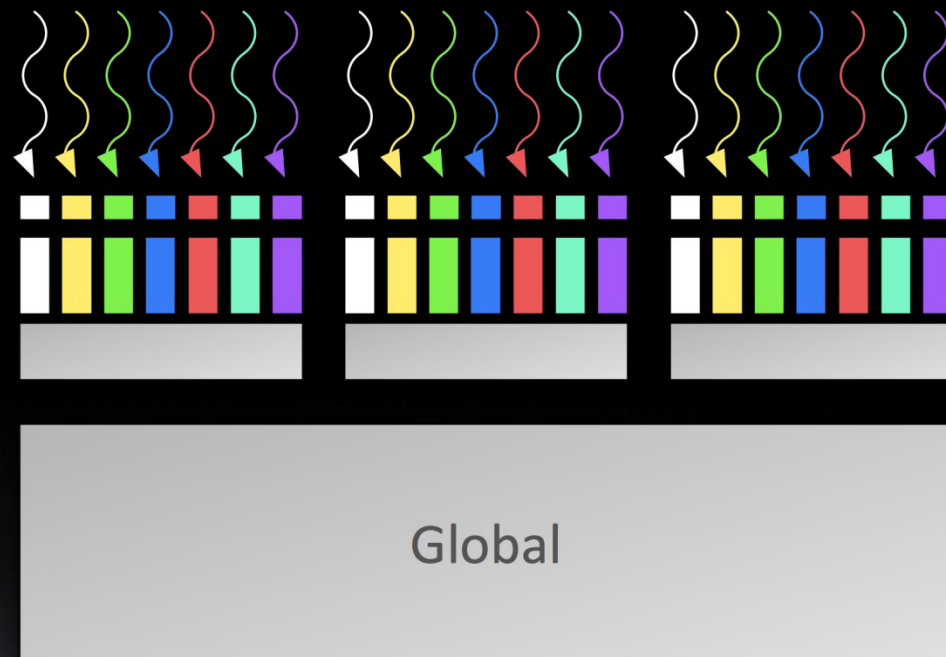
# MEMORY HIERARCHY

- Thread
  - Registers
- Thread
  - Local memory
- Block of threads
  - Shared memory



# MEMORY HIERARCHY

- Thread
  - Registers
- Thread
  - Local memory
- Block of threads
  - Shared memory
- All blocks
  - Global memory





# OVERVIEW OF NUMBA



# NUMBA

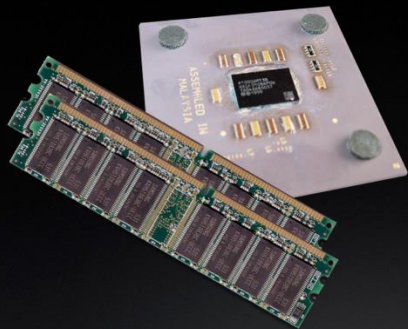
- **Numba** is a **just-in-time (jit)** compiler for Python that works best on code that uses NumPy arrays, functions, and loops.
- **Numba** has set of decorators that can be specified before user-defined functions to determine how they are compiled.
- A **decorated function** written in python is compiled into CUDA kernel hence, Numba supports CUDA GPU programming model.
- **Kernels** written in **Numba** automatically have direct access to NumPy arrays.

# NUMBA: THE BASICS

## ■ Terminology

- *Host* - The CPU and its memory (host memory)
- *Device* - The GPU and its memory (device memory)
- *Kernel* - A GPU function launched by the host and executed on the device.
- *Device Function* - A GPU function executed on the device and called from the device.

Host



Device



# NOTE

- It is recommended to visit the NVIDIA official documentary web page and read **through CUDA C programming guide** (<https://docs.nvidia.com/cuda/cuda-c-programming-guide>)
- This is because most CUDA programming features exposed by Numba map directly to the CUDA C language offered by NVidia.
- Numba does not implement these features of CUDA:
  - ✓ **dynamic parallelism**
  - ✓ **texture memory**



# CUDA KERNELS

# KERNEL EXECUTION CONCEPT

- In CUDA, written code can be executed by hundreds or thousands of threads at a single run, hence, a solution is modeled after the following thread hierarchy:
  - ✓ **Grid**
  - ✓ **Thread Block**
  - ✓ **Thread**
- Numba exposes three kinds of GPU memory:
  - ✓ **global device memory**
  - ✓ **shared memory**
  - ✓ **local memory.**
- Memory access should be carefully considered in order to keep bandwidth contention at minimal.

# KERNEL DECLARATION

- A **kernel function** is a GPU function that is called from a CPU code by specifying the **number of block threads and threads per block** and can not explicitly return a value except through a passed array.
- A kernel can be called multiple times with varying number of blocks per grid and threads per block after its has been compiled once.

```
@cuda.jit
def arrayAdd(array_A, array_B, array_out):
    #...code body ...

N = 1000
threadspblock = 128
blockspgrid = ( N + (threadspblock - 1))// threadspblock

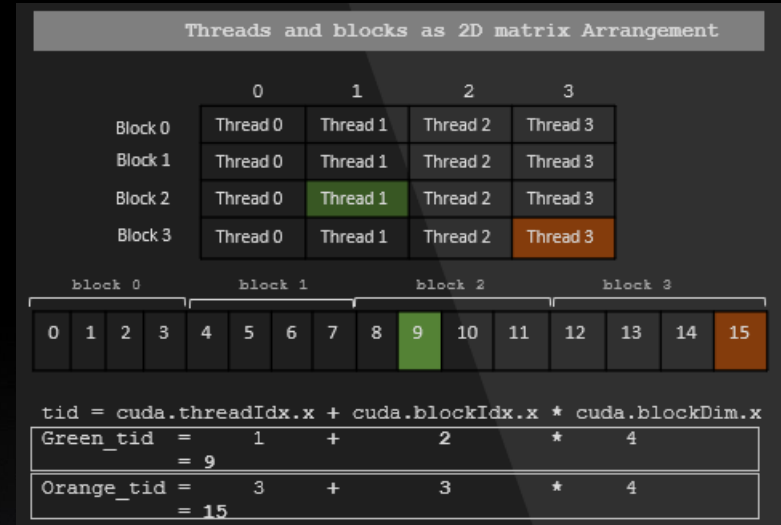
arrayAdd[blockspgrid, threadspblock](array_A, array_B, array_out)
```

# THREAD POSITIONING

- When running a kernel, the kernel function's code is executed by every thread once.

```
threadsperblock = 128
N = array_out.size

@cuda.jit
def arrayAdd(array_A, array_B, array_out):
    tid = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if tid < N: #Check array boundaries
        array_out[tid] = array_A[tid] + array_B[tid]
```



## Note

Unless you are sure the block size and grid size are a divisor of your array size, you **must** check boundaries as shown in the code block above.

# EXAMPLE 1: ADDITION ON ARRAYS

```
import numba.cuda as cuda
import numpy as np
```

```
N = 500000
threadsperblock = 1204
```

```
@cuda.jit
def arrayAdd(array_A, array_B, array_out):
    tid = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if tid < N:
        array_out[tid] = array_A[tid] + array_B[tid]
```

```
array_A = np.arange(N, dtype = np.int)
array_B = np.arange(N, dtype = np.int)
array_out = np.zeros(N, dtype = np.int)
```

```
blockpergrid = N + (threadsperblock - 1) // threadsperblock
```

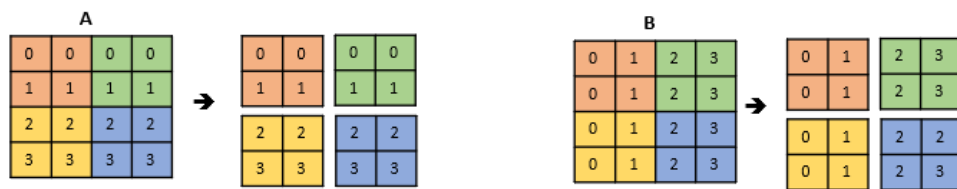
```
arrayAdd[blockpergrid, threadsperblock](array_A, array_B, array_out)
```



# MATRIX SLICE CONCEPT

$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix}$ 
 $B = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix}$ 
 $A \times B = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 4 & 8 & 12 \\ 0 & 8 & 16 & 24 \\ 0 & 12 & 24 & 36 \end{bmatrix}$ 
 $N = 4; \text{ Shape} = (N, N)$

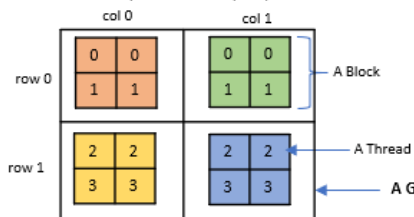
## Device Memory Data Fitting Logic



### Approach 1:

Block per Grid = (2,2)

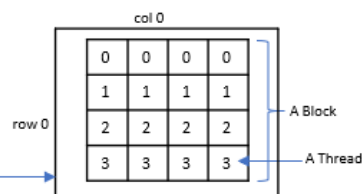
Thread per Block = (2,2)



### Approach 2:

Block per Grid = (1,1)

Thread per Block = (4,4)



## Sample computation on column 0

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} (0 \times 0 + 0 \times 0 + 0 \times 0 + 0 \times 0) & 0 & 0 & 0 \\ (1 \times 0 + 1 \times 0 + 1 \times 0 + 1 \times 0) & 4 & 8 & 12 \\ (2 \times 0 + 2 \times 0 + 2 \times 0 + 2 \times 0) & 8 & 16 & 24 \\ (3 \times 0 + 3 \times 0 + 3 \times 0 + 3 \times 0) & 12 & 24 & 36 \end{bmatrix}$$

A
B
C

$C[\text{row}][\text{col}] = A[\text{row}][k] * B[k][\text{col}]$   
 Where  $k = 1 \dots N$

## Note

Approach 2 is would not be possible if the matrix size exceed the maximum number of threads per block on the device, while Approach 1 would continue to execute. Most common GPUs have maximum of 1024 threads per thread block.

# THREAD REUSE

- It is possible to specify a few number of thread for a data size such that threads are reused to complete computation of the entire data.
- This statement is used in a while loop: `tid += cuda.blockDim.x * cuda.gridDim.x`

```
import numba.cuda as cuda
import numpy as np

N = 500000
threadsperblock = 1024

@cuda.jit
def arrayAdd(array_A, array_B, array_out):
    tid = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    while tid < N:
        array_out[tid] = array_A[tid] + array_B[tid]
        tid += cuda.blockDim.x * cuda.gridDim.x

#.....
#blockpergrid = N + (threadsperblock - 1) // threadsperblock

arrayAdd[1, threadsperblock](array_A, array_B, array_out)
```

This is one of the approach used when a data to be computed is larger than the maximum number of threads available in a device memory.



# MEMORY MANAGEMENT

# DATA TRANSFER

- Numba can automatically transfer NumPy arrays to the device and vice versa when a kernel is executed.
- In order to avoid the unnecessary transfer for read-only arrays, the following APIs can be used to manually control the transfer

## Copy host to device

```
import numba.cuda as cuda
import numpy as np

N = 500000
h_A = np.arange(N, dtype=np.int)
h_B = np.arange(N, dtype=np.int)
h_C = np.zeros(N, dtype=np.int)

d_A = cuda.to_device(h_A)
d_B = cuda.to_device(h_B)
d_C = cuda.to_device(h_C)
```

## enqueue a transfer to a stream

```
h_A = np.arange(N, dtype=np.int)
stream = cuda.stream()
d_A = cuda.to_device(h_A, stream=stream)
```

## Copy device to host / enqueue the transfer to a stream

```
h_C = d_C.copy_to_host()
h_C = d_C.copy_to_host(stream=stream)
```

# EXAMPLE 2: ADDITION TASK

```
import numba.cuda as cuda
import numpy as np
```

```
N = 200
threadsperblock = 25
```

```
@cuda.jit
def arrayAdd(d_A, d_B, d_C):
    tid = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if tid < N:
        d_C[tid] = d_A[tid] + d_B[tid]
```

```
h_A = np.arange(N, dtype=np.int)
h_B = np.arange(N, dtype=np.int)
h_C = np.zeros(N, dtype=np.int)
```

```
d_A = cuda.to_device(h_A)
d_B = cuda.to_device(h_B)
d_C = cuda.to_device(h_C)
```

```
blockpergrid = N + (threadsperblock - 1) // threadsperblock
arrayAdd[blockpergrid, threadsperblock](d_A, d_B, d_C)
```

```
h_C = d_C.copy_to_host()
print(h_C)
```



# ATOMIC OPERATION

# NUMBA ATOMIC OPERATION

- Atomic operation is required in a situation where multiple threads attempt to modify a common portion of the memory.
- Typical example includes simultaneous withdrawal from a bank account through an ATM machine or many threads modifying a particular index of an array based on certain condition(s).

```
import numba.cuda as cuda
cuda.atomic.add(array, index, value)
cuda.atomic.min(array, index, value)
cuda.atomic.max(array, index, value)
cuda.atomic.nanmax(array, index, value)
cuda.atomic.nanmin(array, index, value)
cuda.atomic.compare_and_swap(array, old_value, current_value)
cuda.atomic.sub(array, index, value)
```

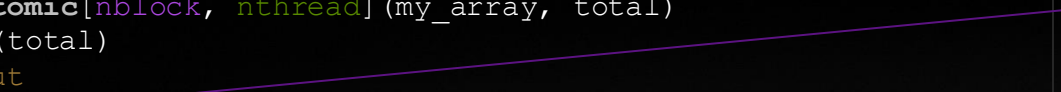
# NUMBA ATOMIC OPERATION

## Atomic

```
size = 10
nthread = 10

@cuda.jit()
def add_atomic(my_array, total):
    tid = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    cuda.atomic.add(total, 0, my_array[tid])

my_array = np.array([1,2,3,4,5,6,7,8,9,10], dtype=np.int32)
total = np.zeros(1, dtype=np.int32)
nblock = int(size/ nthread)
add_atomic[nblock, nthread](my_array, total)
print(total)
#output
[55]
```



## Non-Atomic

```
size = 10
nthread = 10

@cuda.jit()
def add_non_atomic(my_array, total):
    tid = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    total[0] += my_array[tid]

my_array = np.array([1,2,3,4,5,6,7,8,9,10], dtype=np.int32)
total = np.zeros(1, dtype=np.int32)
nblock = int(size/ nthread)
add_non_atomic[nblock, nthread](my_array, total)
print(total)

#output
[1]
```



An abstract background featuring a complex network graph. The graph consists of numerous nodes, represented by small circles in white and yellow-green, connected by thin, light gray lines. The nodes are distributed across the frame, with a higher density in the upper right quadrant, creating a sense of depth and connectivity.

# CUDA UFUNCS

# CUDA UFUNCS

- The CUDA ufunc supports passing intra-device arrays to reduce traffic over the PCI-express bus.
- It also supports asynchronous mode by using stream keyword.

```
from numba import vectorize
import numpy as np

@vectorize(['float32(float32, float32)'], target='cuda')
def compute(a, b):
    return (a - b) * (a + b)

N = 10000
A = np.arange(N, dtype=np.float32)
B = np.arange(N, dtype=np.float32)
C = compute(A, B)
```

```
@vectorize(['float32(float32, float32)'], target='cuda')
```

Ufuncs decorator

Result data type

arguments data type

target accelerator

# DEVICE FUNCTION

- The **CUDA device functions** can only be invoked from within the device and can return a value like normal functions.
- The device function is usually placed before the CUDA ufunc kernel otherwise a call to the device function may not be visible inside the ufunc kernel.

```
from numba import vectorize
import numba.cuda as cuda
import numpy as np
```

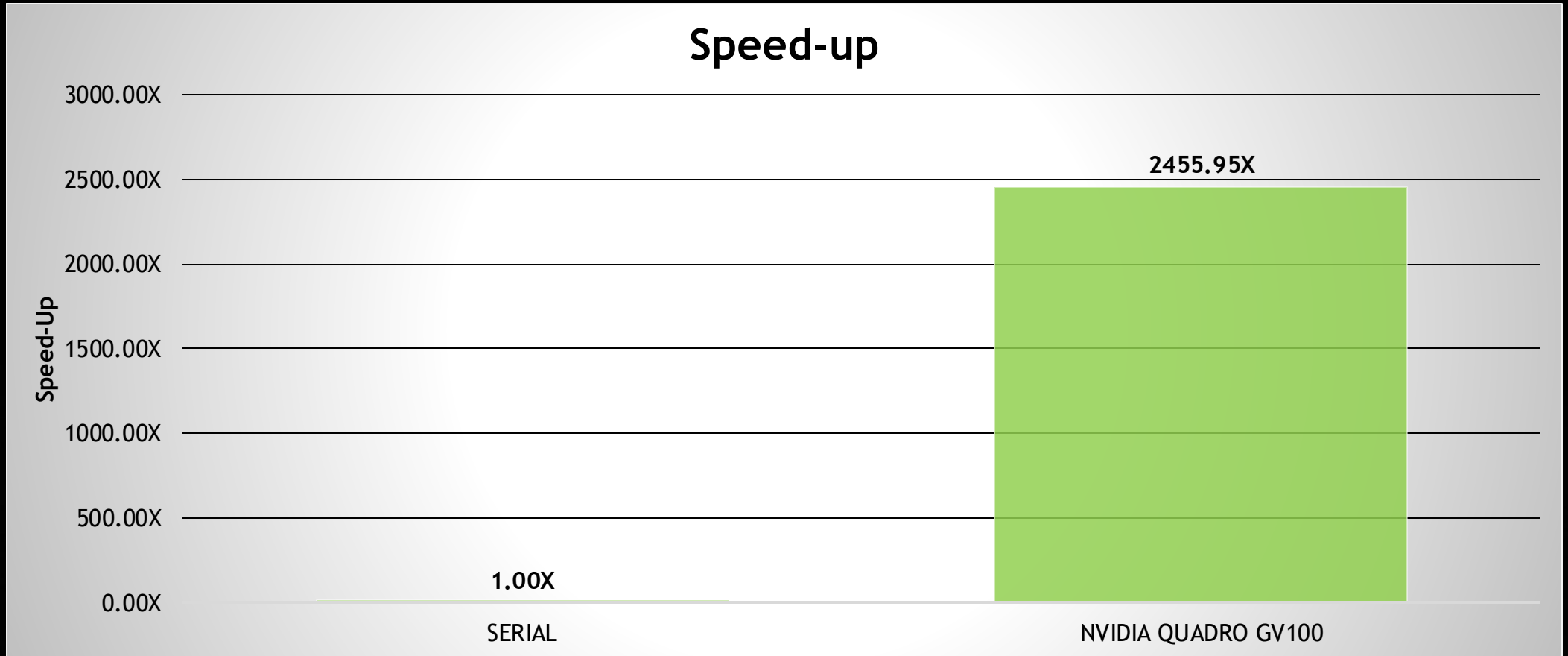
device function

```
@cuda.jit('float32(float32)', device=True, inline=True)
def device_ufunc(c):
    return math.sqrt(c)
```

```
@vectorize(['float32(float32, float32)'], target='cuda')
def compute(a, b):
    c = (a - b) * (a + b)
    return device_ufunc(c)
```

call to a device function

# NUMBA SPEEDUP

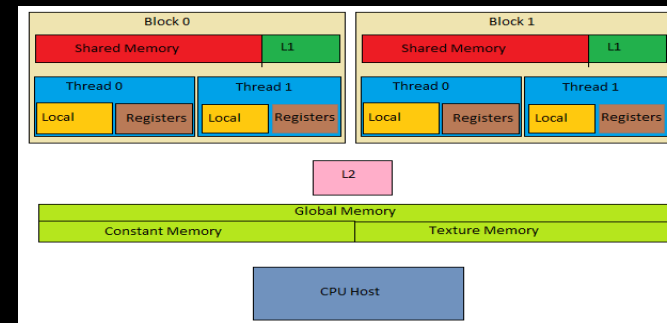




# SUMMARY

# SUMMARY

- Numba supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model.
- Data movement involves **host to device** and **device to host**
- NumPy arrays are transferred between the CPU and the GPU automatically.
- Atomic operation: `cuda.atomic.add(array, index, value)`



Ufuncs kernel

```
@vectorize(['float32(float32, float32)'], target='cuda')
def compute(a, b):
    c = (a - b) * (a + b)
    return device_ufunc(c)
```

device function

```
@cuda.jit('float32(float32)', device=True, inline=True)
def device_ufunc(c):
    return math.sqrt(c)
```

```
N = 200
threadsperblock = 25
```

CUDA JIT kernel

```
@cuda.jit
def arrayAdd(d_A, d_B, d_C):
    tid = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if tid < N:
        d_C[tid] = d_A[tid] + d_B[tid]
```

```
h_A = np.arange(N, dtype=np.int)
```

CPU → GPU

```
d_A = cuda.to_device(h_A)
```

```
blockspergrid = N + (threadsperblock - 1) // threadsperblock
```

Call to kernel

```
arrayAdd[blockspergrid, threadsperblock](d_A, d_B, d_C)
```

```
h_C = d_C.copy_to_host()
```

GPU → CPU

# REFERENCES

- <https://developer.nvidia.com/hpc-sdk>
- Numba Documentation, Release 0.52.0-py3.7-linux-x86\_64.egg, Anaconda, Nov 30, 2020.
- Bhaumik Vaidya, Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA, Packt Publishing, 2018.
- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>





# THANK YOU

