N-WAYS GPU BOOTCAMP

A QUICK GUIDE TO CUPY

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

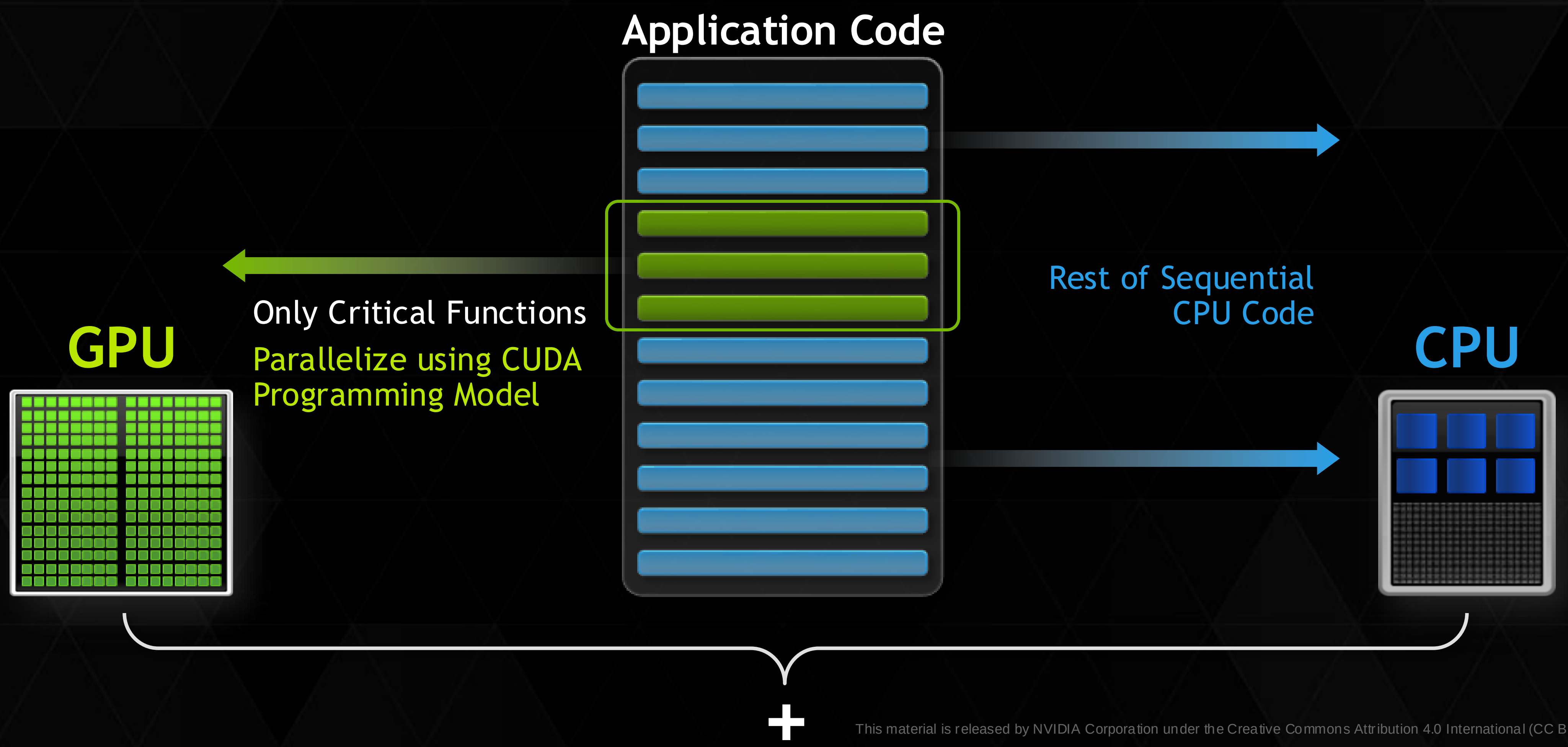
A QUICK GUIDE TO CUPY

What to expect?

- What is CuPy?
- Features of CuPy
- Installation Guide
- CuPy Fundamentals
- CUDA Kernels
- Summary

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

GPU COMPUTING



This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUDA ARCHITECTURE PROGRAMMING MODEL

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUDA KERNELS

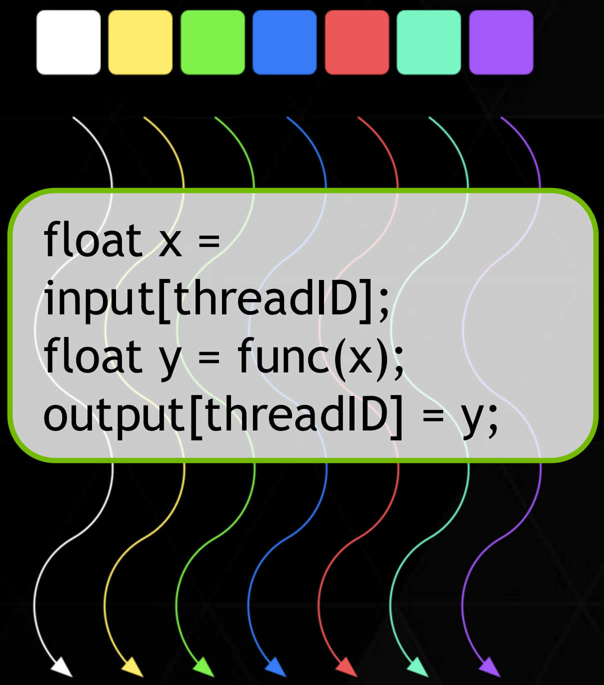
- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads
- CUDA threads:
  - Lightweight
  - Fast switching
  - Tens of thousands execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

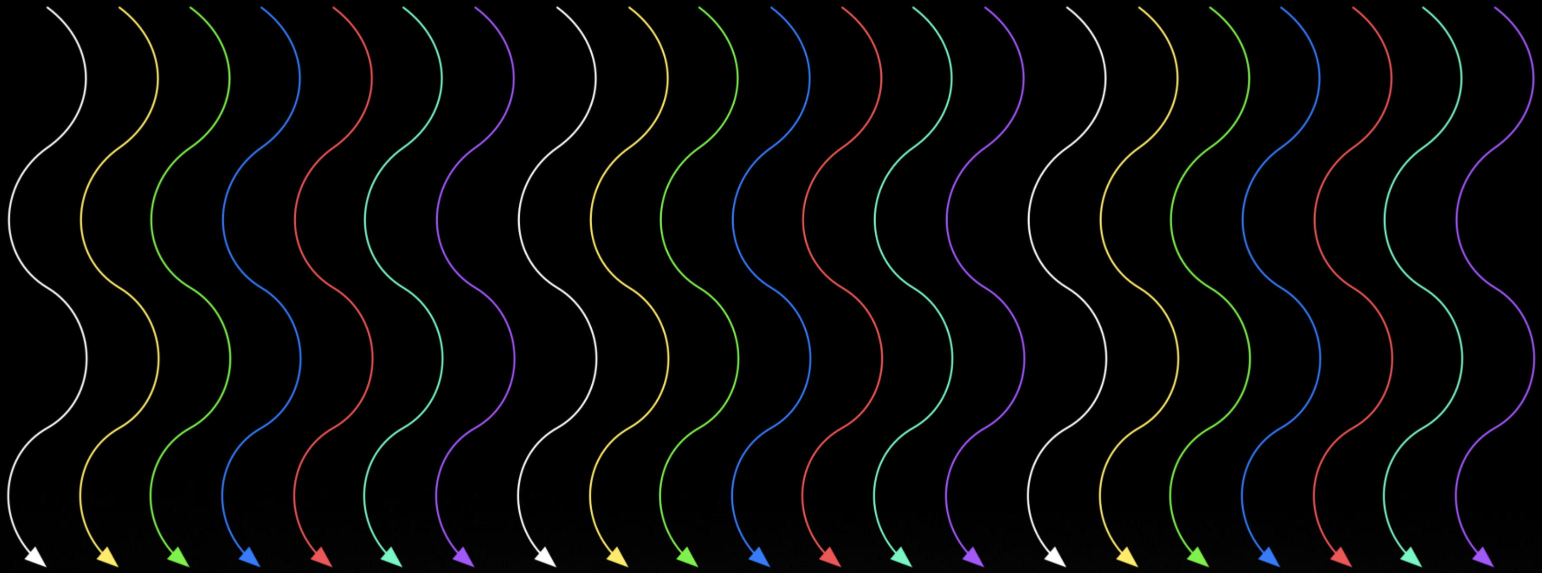
CUDA KERNELS: PARALLEL THREADS

- 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



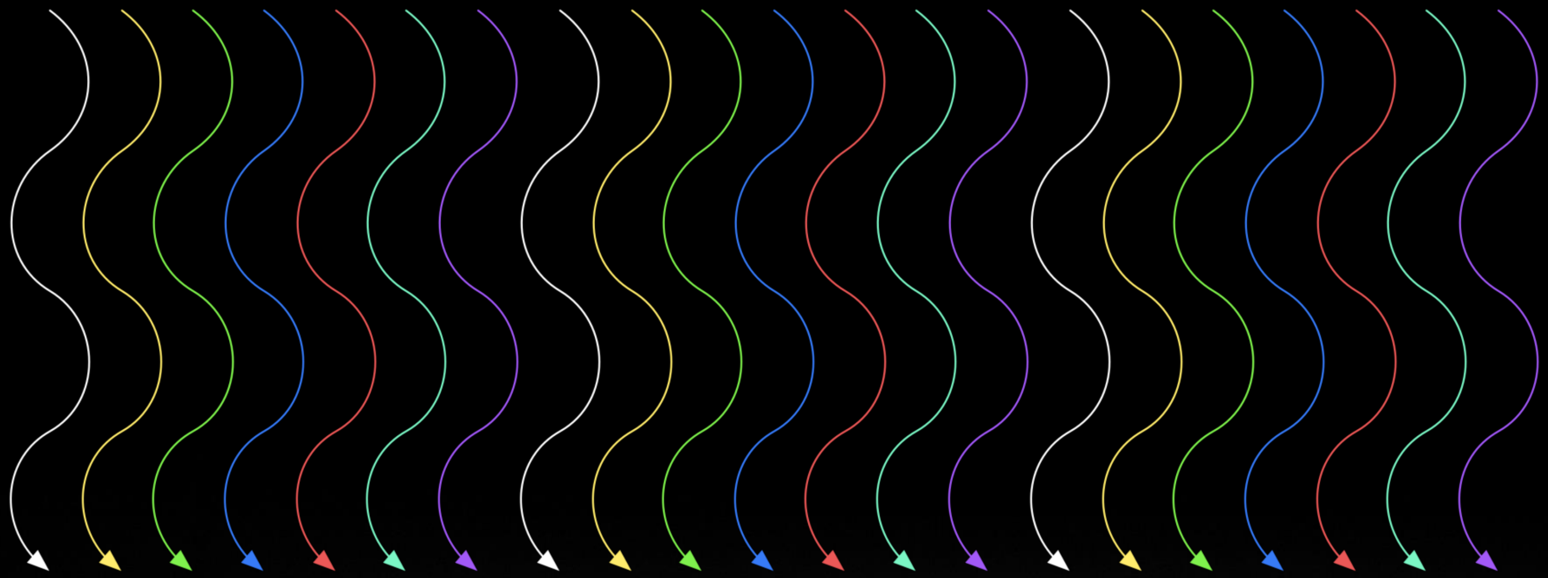
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUDA KERNELS: SUBDIVIDE INTO BLOCKS



This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

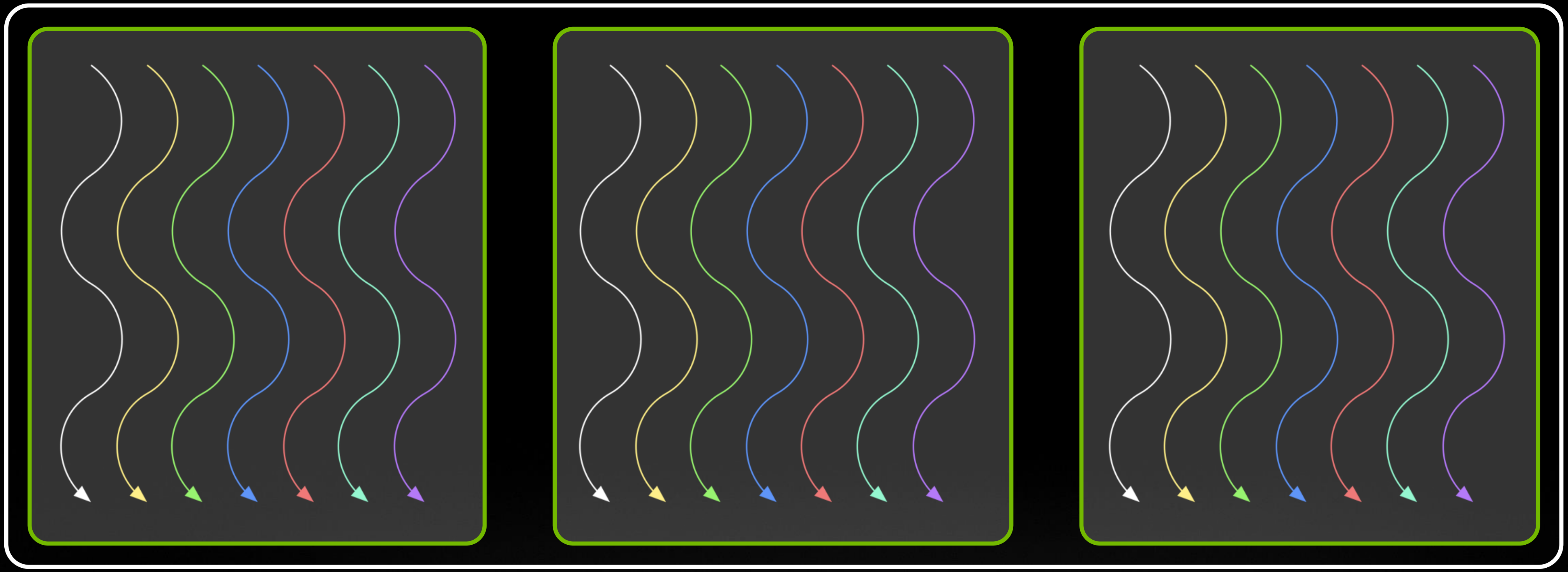
CUDA KERNELS: SUBDIVIDE INTO BLOCKS



- Threads are grouped into blocks

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUDA KERNELS: SUBDIVIDE INTO BLOCKS

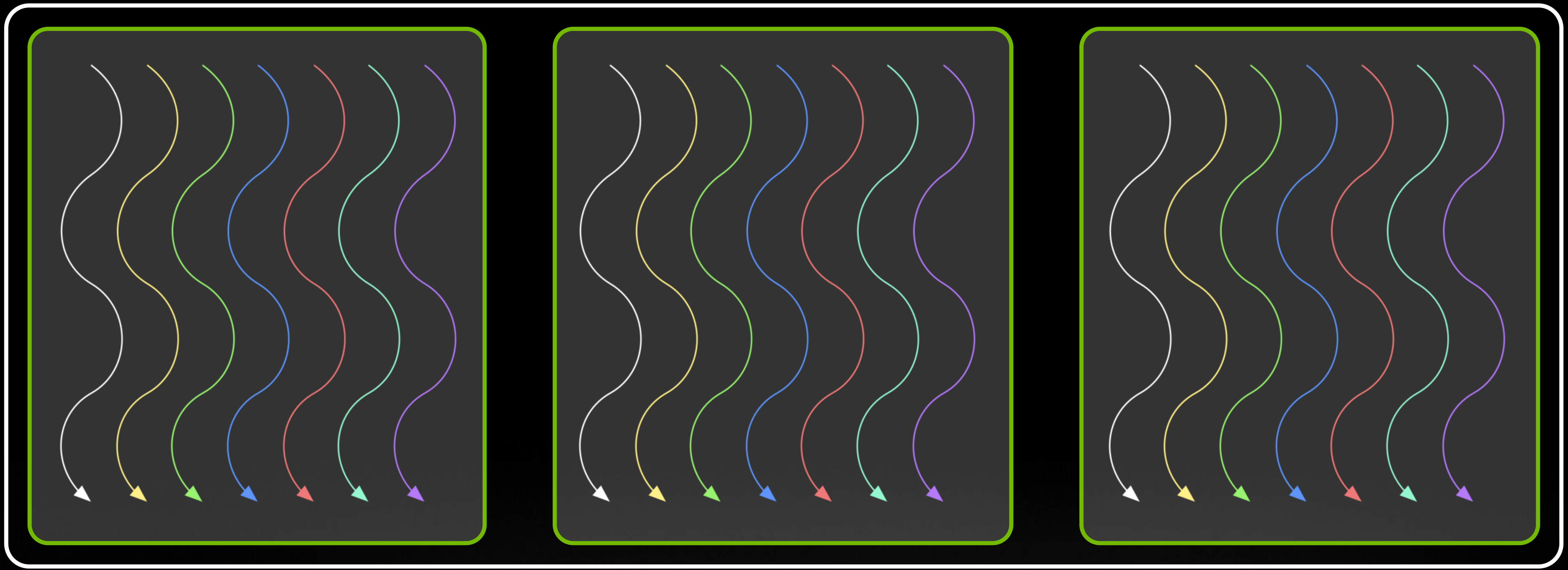


- Threads are grouped into blocks
- Blocks are grouped into a grid

NVIDIA

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

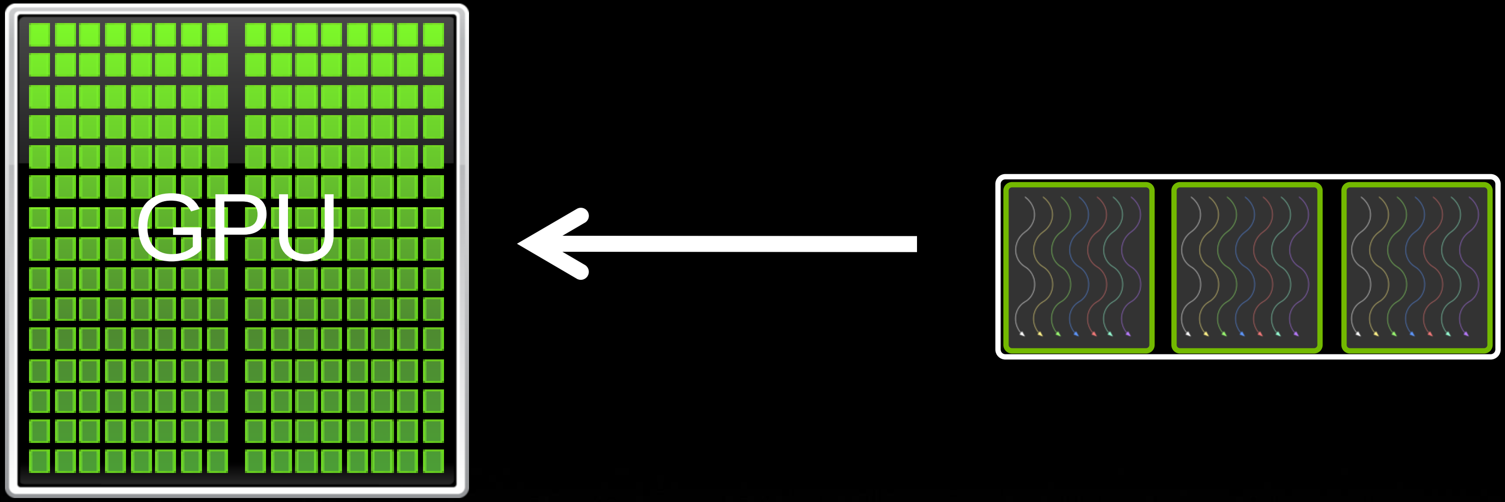
CUDA KERNELS: SUBDIVIDE INTO BLOCKS



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

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

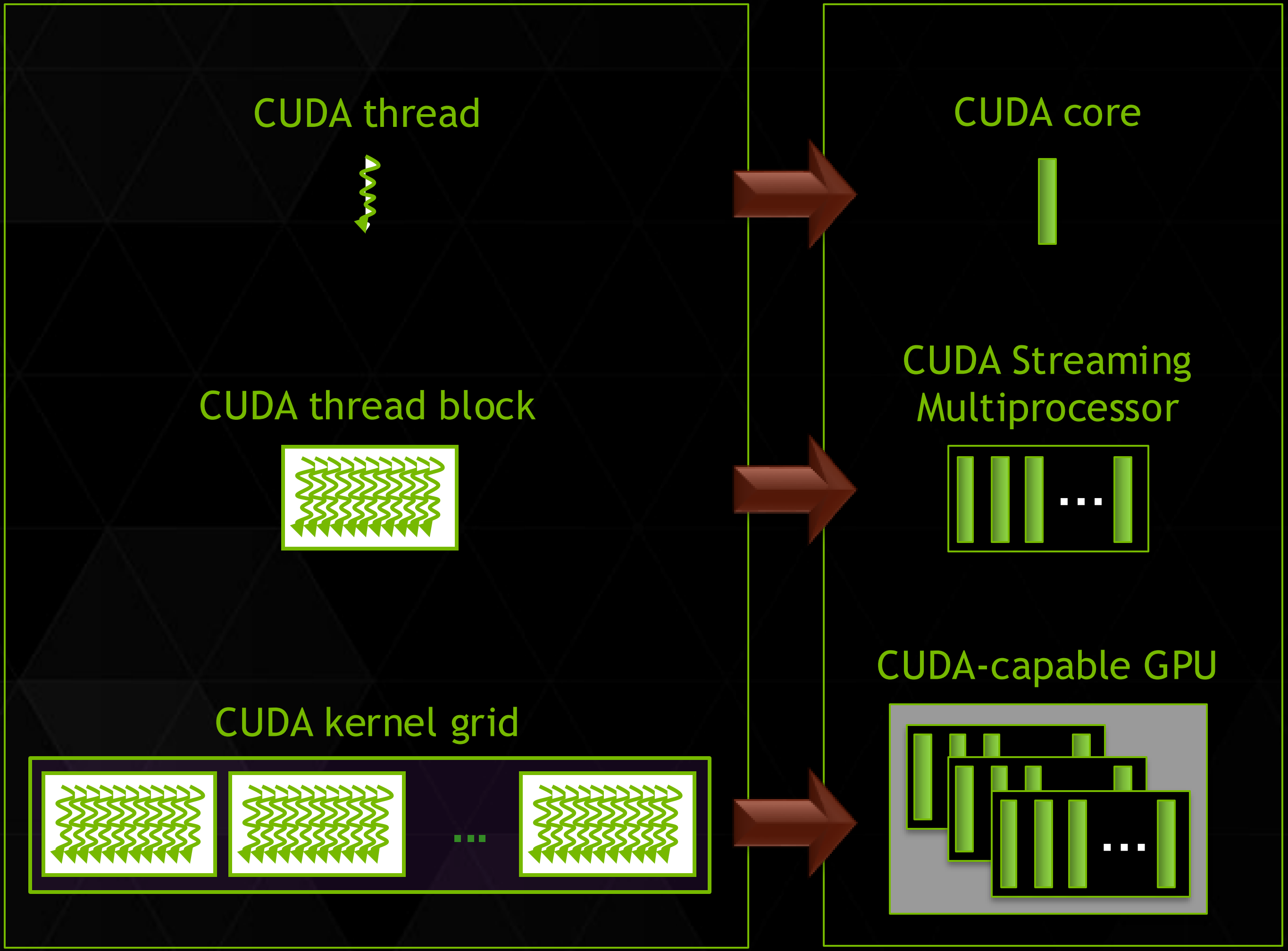
CUDA KERNELS: SUBDIVIDE INTO BLOCKS



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

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

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

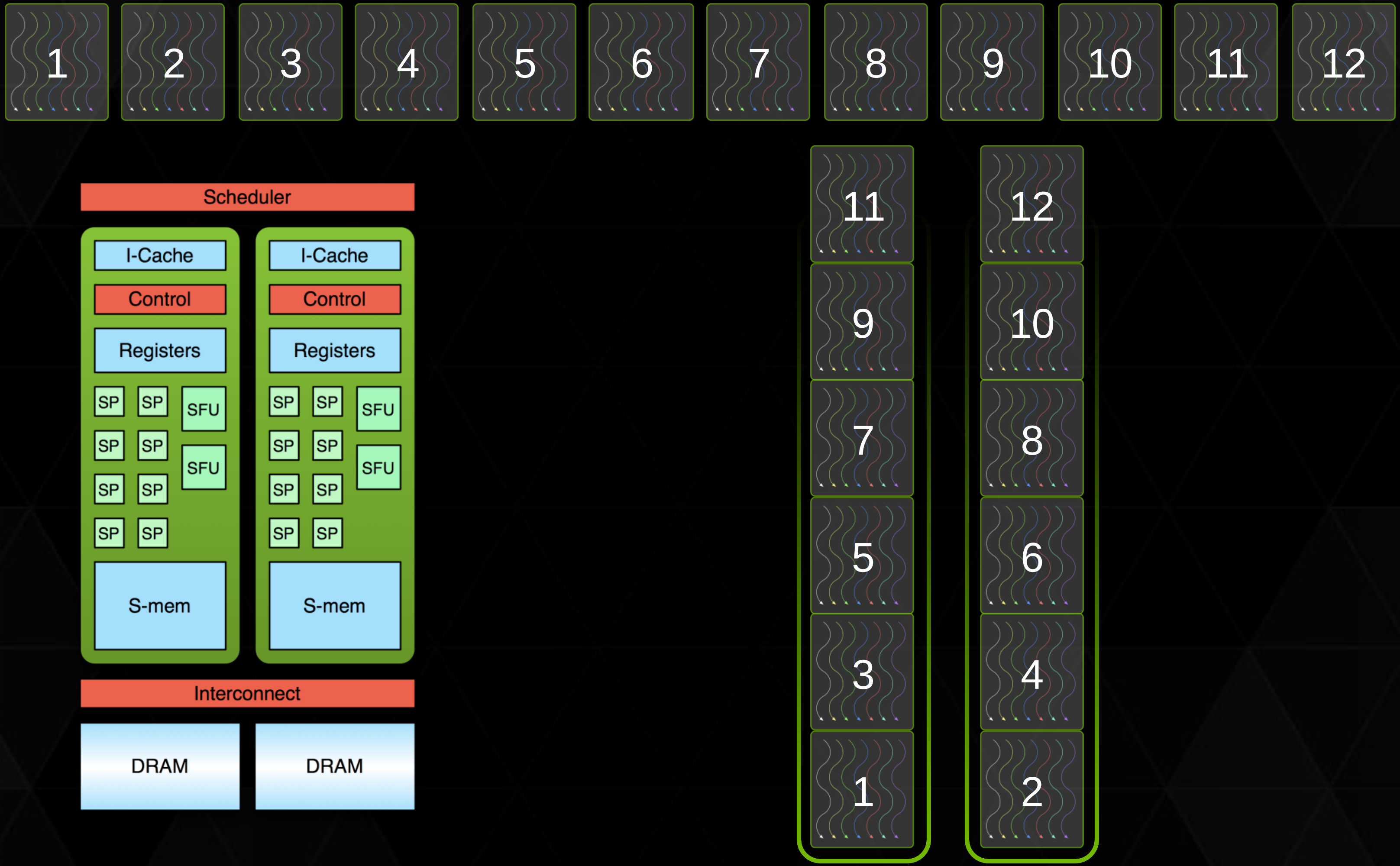
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

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

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

TRANSPARENT SCALABILITY

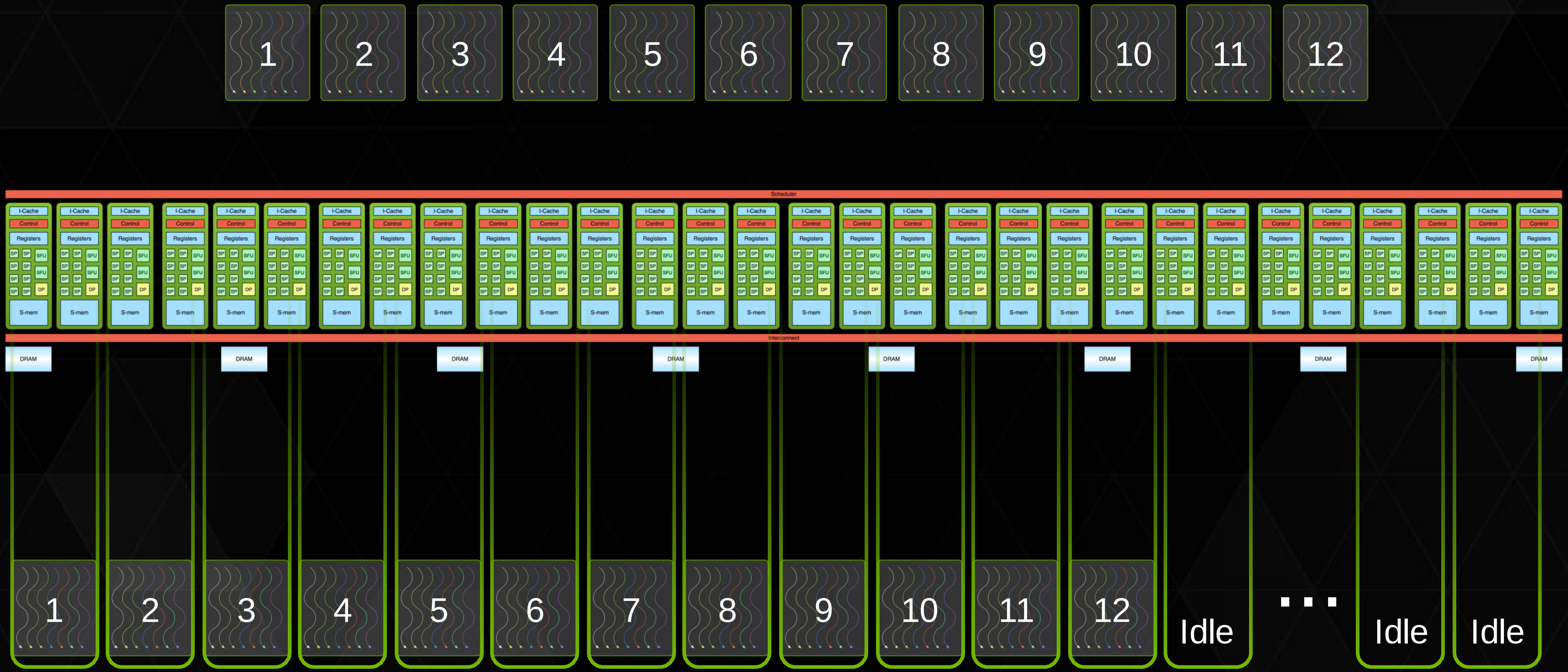


This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

TRANSPARENT SCALABILITY

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

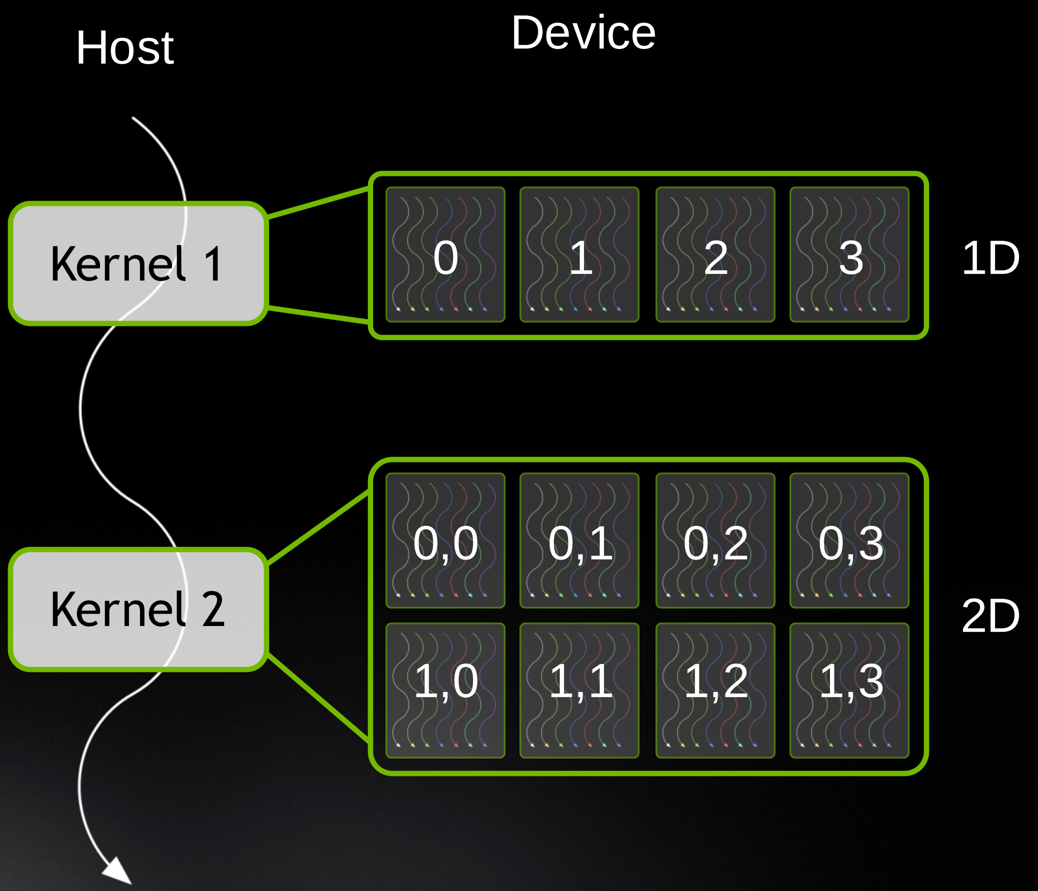
TRANSPARENT SCALABILITY –



This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

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



This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUDA ARCHITECTURE MEMORY MODEL

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

GPU ARCHITECTURE

Two Main components

Global memory

Analogous to RAM in a CPU server

Accessible by both GPU and CPU

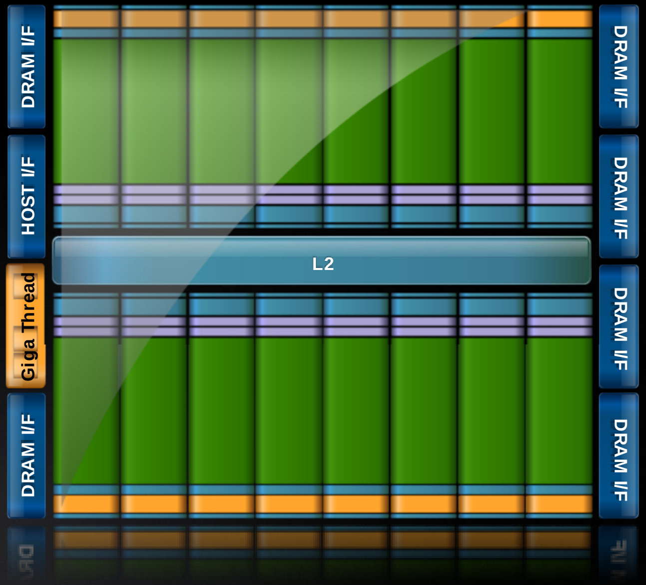
48GB with bandwidth currently up to 1 TB/s

Streaming Multiprocessors (SMs)

SMs perform the actual computations

Each SM has its own:

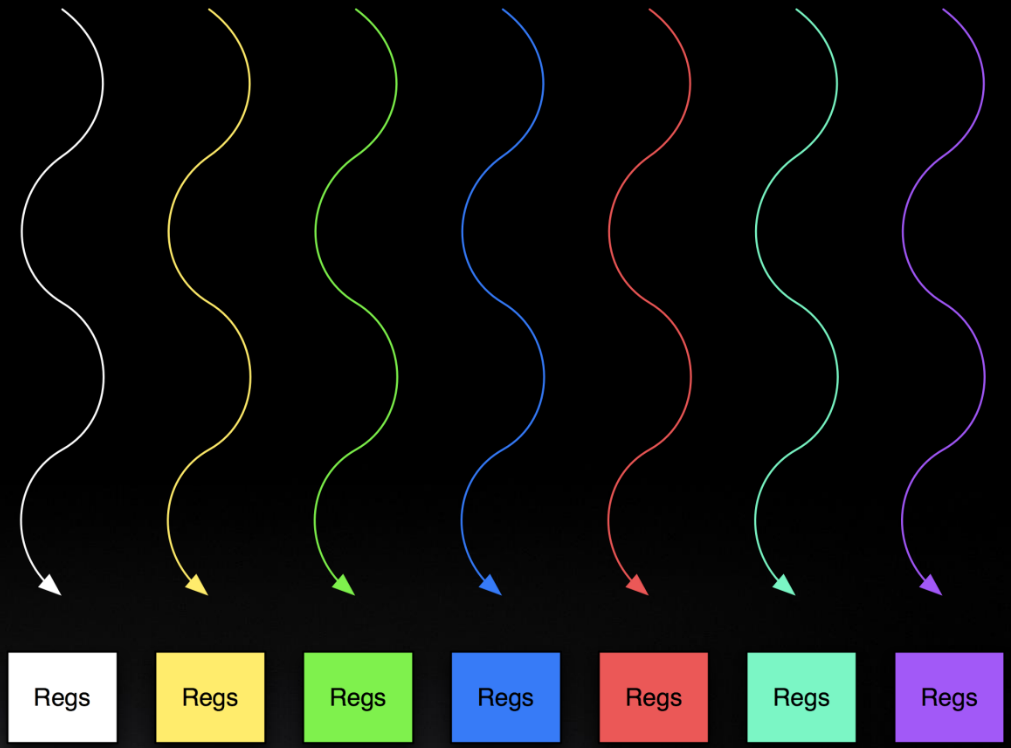
Control units, registers, execution pipelines, caches



NVIDIA.
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

MEMORY HIERARCHY

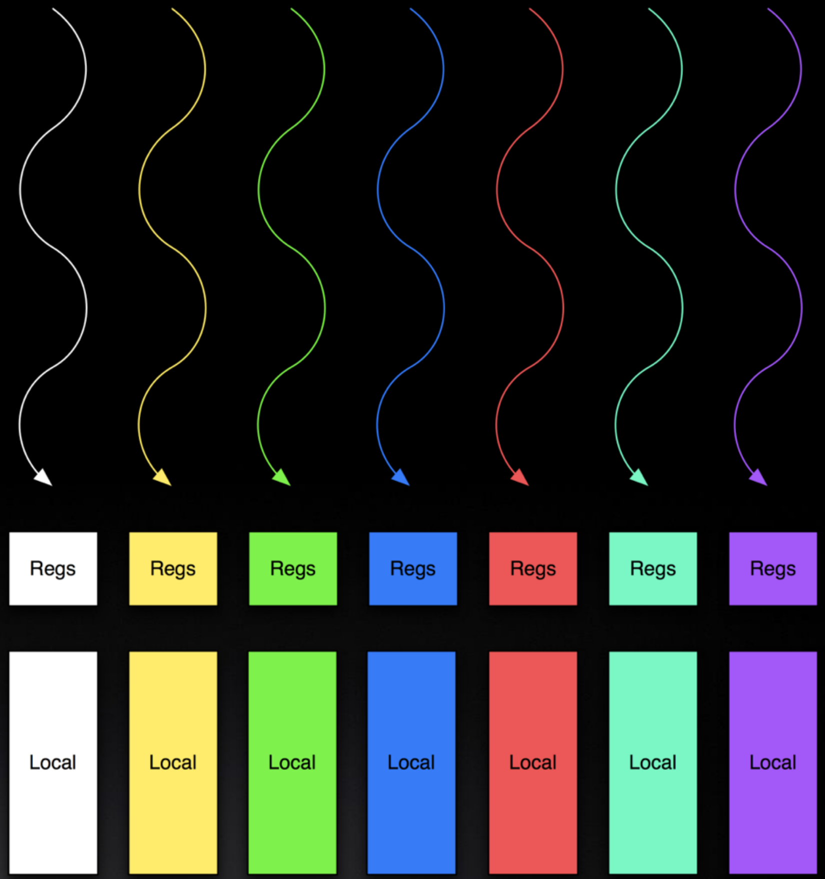
- Thread
  - Registers



This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

MEMORY HIERARCHY

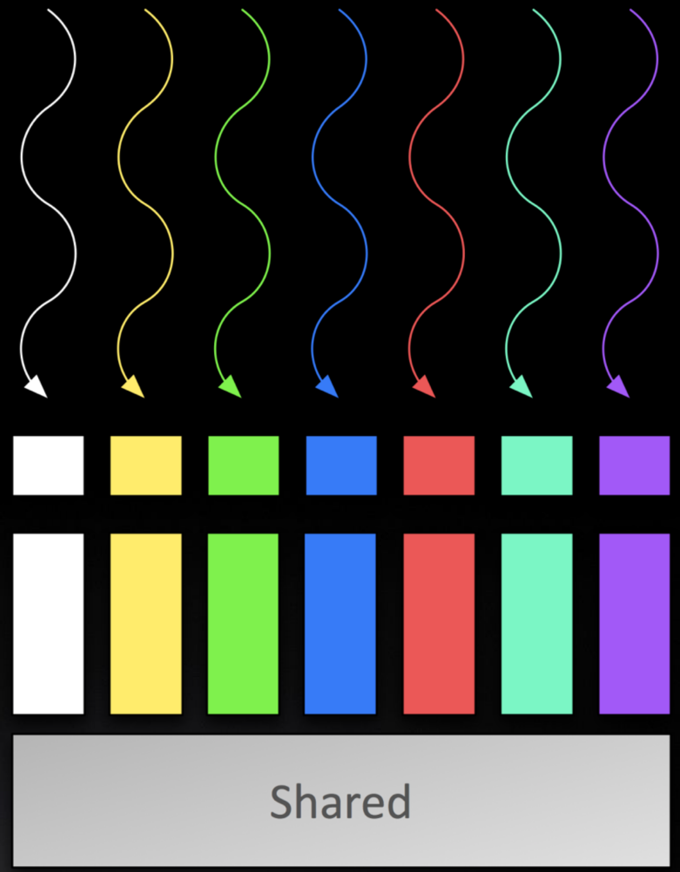
- Thread
  - Registers
- Thread
  - Local memory



This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

MEMORY HIERARCHY

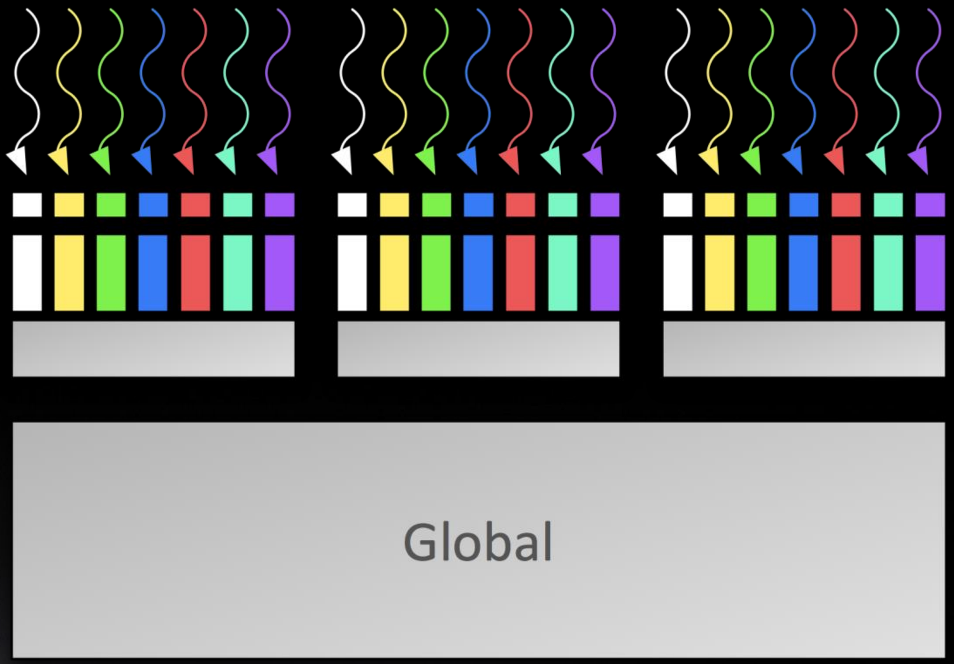
- Thread
  - Registers
- Thread
  - Local memory
- Block of threads
  - Shared memory



NVIDIA
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

MEMORY HIERARCHY

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



This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

WHAT IS CUPY?

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

OVERVIEW OF CUPY

- **CuPy** is an implementation of NumPy-compatible multi-dimensional array on CUDA
- **CuPy** consists of :

✓ cupy.ndarray

✓ the core multi-dimensional array class

✓ many functions

NVIDIA.
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

OVERVIEW OF CUPY

- **CuPy** supports a subset of numpy.ndarray interface which include:

✓ Basic & advance indexing, and Broadcasting

✓ Data types (int32, float32, uint64, complex64,...)

✓ Array manipulation routine (reshape)

✓ Linear Algebra functions (dot, matmul, etc)

✓ Reduction along axis (max, sum, argmax, etc)

For more details on broadcasting visit

(https://numpy.org/doc/stable/user/basics.broadcasting.html)

```
>>> import numpy as np
>>> X = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
#Basic indexing and slicing
>>> X[5:]
array([5, 6, 7, 8, 9])
>>> X[1:7:2]
array([1, 3, 5])

#Advance indexing
>>> X = np.array([[1, 2],[3, 4],[5, 6]])
>>> X[[0, 1, 2], [0, 1, 0]]
array([1, 4, 5])

#reduction and Linear Algebra function
>>> max(X)
9.0
>>> B = np.array([1,2,3,4], dtype=np.float32)
>>> C = np.array([5,6,7,8], dtype=np.float32)
>>> np.matmul(B, C)
70.0
#data type and array manipulation routine
>>> A =1j*np.arange(9, dtype=np.complex64).reshape(3,3)
[[0.+0.j 0.+1.j 0.+2.j]
 [0.+3.j 0.+4.j 0.+5.j]
 [0.+6.j 0.+7.j 0.+8.j]]
```


This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

FEATURES OF CUPY

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

FEATURES OF CUPY

- Features of CuPy includes:

✓ User-define elementwise CUDA kernels

✓ User-define reduction CUDA kernels

✓ Fusing CUDA kernels to optimize user-define calculation

✓ Customizable memory allocator and memory pool

✓ cuDNN utilities

- These features are developed to support performance.

- **CuPy uses on-the-fly kernel synthesis**: when a kernel call is required, it compiles a kernel code optimized for the shapes and dtypes of given arguments, sends it to the GPU device, and executes the kernel.

- **CuPy** also caches the kernel code sent to GPU device within the process, which reduces the kernel transfer time on further calls.

NVIDIA
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

INSTALLATION GUIDE

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

REQUIREMENTS

- **Recommended Linux distributions are Centos and Ubuntu**

✓ NVIDIA CUDA GPU with the Compute Capability 3.0 or larger

✓ CUDA Toolkit: v9.0 - v11.2

✓ Python: v3.5.1+ - v3.9.0+

- **Python Dependencies**

✓ NumPy/SciPy-compatible API in CuPy v8 is based on NumPy 1.19 and SciPy 1.5.

1. Wheels (precompiled binary package)

CUDA	Command
v9.0	`$ pip install cupy-cuda90`
v9.2	`$ pip install cupy-cuda92`
v10.0	`$ pip install cupy-cuda100`
v10.1	`$ pip install cupy-cuda101`
v10.2	`$ pip install cupy-cuda102`
v11.0	`$ pip install cupy-cuda110`
v11.1	`$ pip install cupy-cuda111`

2. Conda-Forge

✓ $ conda install -c conda-forge cupy

- **To install CuPy with the cuTENSOR support enabled, you can do:**

✓ $ conda install -c conda-forge cupy cutensor cudatoolkit=10.2

3. CuPy inside Docker

You can pull CuPy Docker images from:

https://hub.docker.com/r/cupy/cupy/.

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

REQUIREMENTS & ARCHITECTURE

4. Full RAPIDS Package

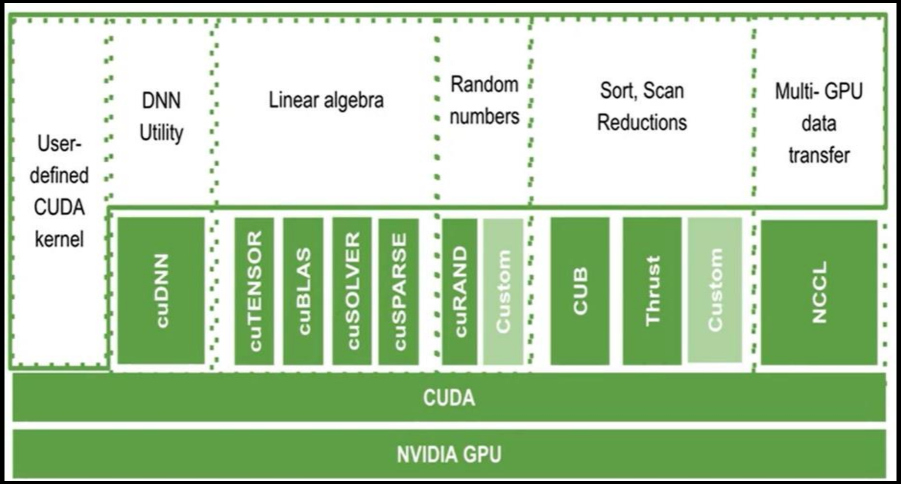
	Preferred		Advanced				
METHOD	Conda	Docker + Examples	Docker + Dev Env	Source			
RELEASE	Stable (0.18)		Nightly (0.19a)				
TYPE	RAPIDS and BlazingSQL		RAPIDS Core (w/o BlazingSQL)				
PACKAGES	All Packages	cuDF	cuML	cuGraph	cuSignal	cuSpatial	cuxfilter
LINUX	Ubuntu 16.04	Ubuntu 18.04	Ubuntu 20.04	CentOS 7	CentOS 8	RHEL 7&8	
PYTHON	Python 3.7		Python 3.8				
CUDA	CUDA 10.1.2	CUDA 10.2	CUDA 11.0				

NOTE: Ubuntu 16.04/18.04/20.04 & CentOS 7/8 use the same `conda create` commands.

COMMAND

```
conda create -n rapids-0.18 -c rapidsai -c nvidia -c conda-forge \
    -c defaults rapids-blazing=0.18 python=3.7 cudatoolkit=10.1
```

CuPy Architecture



NVIDIA
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

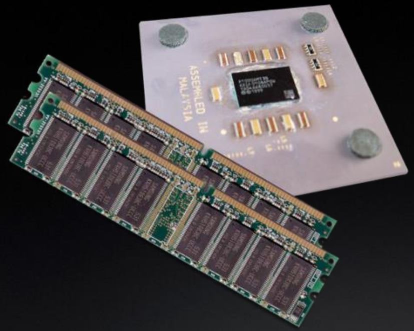
CUPY FUNDAMENTALS

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

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.

Host



Device

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUPY.NDARRAY

- CuPy is a GPU array backend that implements a subset of NumPy interface

CuPy	NumPy

`import cupy as cp`
 	
`X_gpu = cp.array([1, 2, 3, 4, 5])`
	`import numpy as np`
	
	`x = np.array([1, 2, 3, 4, 5])`

Current device (GPU ID: 0)

```
import cupy as cp

gpu_0 = cp.array([1, 2, 3, 4, 5])

# Switch device
cp.cuda.Device(1).use()
gpu_1 = cp.array([1, 2, 3, 4])
```

Switch GPU temporarily

```
import numpy as np
import cupy as cp

with cp.cuda.Device(1):
      gpu_1 = cp.array([1, 2, 3, 4])

# back to device id 0
gpu0 = cp.array([1, 2, 3, 4, 5])
```


NVIDIA.
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

DATA TRANSFER

Host →Device using **cupy.asarray**.

```
import cupy as cp
import numpy as np

X = np.array([1, 2, 3, 4, 5])
x_gpu = cp.asarray(x)
print(x_gpu)
Output:[1 2 3 4 5]
```

Device →Host using **cupy.asnumpy** or **cupy.ndarray.get()**

```
import cupy as cp
import numpy as np

X_gpu = cp.array([1, 2, 3, 4, 5])
# copy to Host
x_cpu = cp.asnumpy(x_gpu)
print(x_cpu)
[1 2 3 4 5]

#alternative option
x_cpu_alt = x_gpu.get()
x_cpu_alt
Output:[1 2 3 4 5]
```

Devices(GPU to GPU)

```
import cupy as cp
with cp.cuda.Device(0):
    x_gpu_0 = cp.ndarray([ 2, 3, 3])
x_gpu_0
[[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]]

with cp.cuda.Device(1):
    x_gpu_1 = cp.asarray(x_gpu_0)
x_gpu_1
[[[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]
  [0. 0. 0.]]]
```


This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

GPU & CPU AGNOSTIC CODE

- Using **cupy.get_array_module()**

```
>>> import cupy as cp
>>> import numpy as np
>>>
>>> #example: log(1 + exp(x))
>>> x_cpu = np.array([1, 2, 3, 4, 5])
>>> x_gpu = cp.get_array_module(x_cpu)
>>> result = x_gpu.maximum(0, x_cpu)+ x_gpu.log1p(x_gpu.exp(-abs(x_cpu))
>>> result
Output: [1.31326169 2.12692801 3.04858735 4.01814993 5.00671535]
>>>
>>>
>>> #An explicit conversion to host
>>> x_gpu = cp.array([6, 7, 8, 9, 10])
>>> result = cp.asnumpy(x_gpu) + x_cpu
>>> result
Output: [ 7  9 11 13 15]
>>> #An explicit conversion to device
>>> result = x_gpu + cp.asarray(x_cpu)
>>> result
Output: [ 7  9 11 13 15]
>>>
```


NVIDIA
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

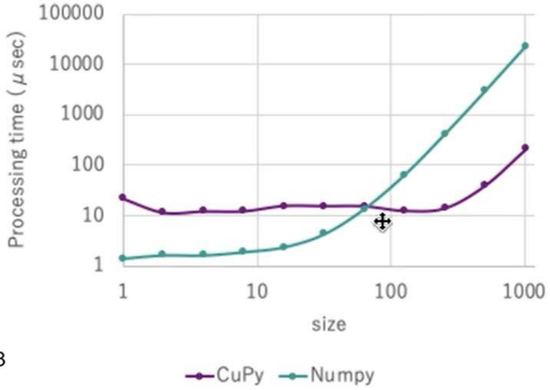
HOW MUCH FASTER IS CUPY THAN NUMPY ?

Dot product

```python
a = xp.ones((size, size), 'f')
b = xp.ones((size, size), 'f')

def f():
  xp.dot(a, b)
```

For a rough estimation, if the array size is larger than L1 cache of your CPU, CuPy gets faster than NumPy.

Try on Google Colab! http://bit.ly/cupywest2018

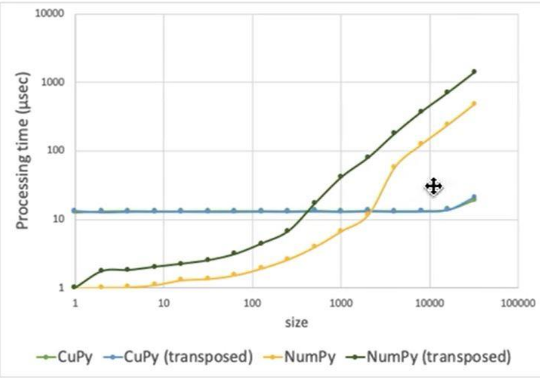


Add Function

```python
a = xp.ones((size, 32), 'f')
b = xp.ones((size, 32), 'f')

def f():
  a + b
```

```python
# Transpose
a = xp.ones((32, size), 'f').T
b = xp.ones((size, 32), 'f')

def f():
  a + b
```

Xeon Gold 6154 CPU @ 3.00GHz
Tesla V100-PCIE-16GB



NVIDIA.
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUDA KERNELS

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUPY CUDA KERNELS

- **CUDA Kernels can be defined in Cupy as follows:**

✓ Elementwise Kernels

✓ Reduction Kernels

✓ Raw Kernels

✓ Kernel Fusion

- These kernels are user-defined based.

NVIDIA.
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

ELEMENTWISE KERNEL

- The **ElementwiseKernel** class is used to define this type of kernel.
- This kernel consists of four parts which includes:

✓ a list of input argument

✓ a list of output argument

✓ a loop body code

✓ a kernel name

- Variable name starting with underscore “_” , “n”, and “i” are regarded as reserved keywords.

NVIDIA

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

ELEMENTWISE KERNEL

- Example : z = x*w + b

```
import cupy as cp                                          Data type
input_list = 'float32 x , float32 w, float32 b'  ←        Input argument list
output_list = 'float32 z' ←                               Output argument list
code_body  = 'z =  (x * w) + b' ←                          A loop body code
# elementwisekernel class defined
dnnLayerNode = cp.ElementwiseKernel(input_list, output_list, code_body,'dnnLayerNode')
x = cp.arange(9, dtype=cp.float32).reshape(3,3)
w = cp.arange(9, dtype=cp.float32).reshape(3,3)
b = cp.array([-0.5], dtype=cp.float32)
z = cp.empty((3,3), dtype=cp.float32)
# kernel call with argument passing
dnnLayerNode(x,w,b,z)
print(z)
#output
[[-0.5  0.5  3.5]
 [ 8.5 15.5 24.5]
 [35.5 48.5 63.5]]
```


This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

ELEMENTWISE KERNEL: GENERIC-TYPE KERNELS

- Example : z = x*w + b

Generic type placeholder

```
import cupy as cp
input_list = 'T x , T w, T b'
output_list = 'T z'
code_body  = 'z =  (x * w) + b'
# elementwisekernel class defined
dnnLayerNode = cp.ElementwiseKernel(input_list, output_list, code_body,'dnnLayerNode')
x = cp.arange(9, dtype=cp.float32).reshape(3,3)
w = cp.arange(9, dtype=cp.float32).reshape(3,3)
b = cp.array([-0.5], dtype=cp.float32)
z = cp.empty((3,3), dtype=cp.float32)
# kernel call with argument passing
dnnLayerNode(x,w,b,z)
print(z)
#output
[[-0.5  0.5  3.5]
 [ 8.5 15.5 24.5]
 [35.5 48.5 63.5]]
```

Multiple generic placeholder

```
import cupy as cp
input_list =  'T x , W w,   B b'
output_list = 'T z'
..............................
..............................
..............................
#output
..............................
..............................
```

Different types of placeholder

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

REDUCTION KERNEL

- Reduction kernel is implemented through the **ReductionKernel** class.

- In order to implement this kernel class, the following parts must be defined:

✓ Identity value: to initialize reduction value.

✓ Mapping expression: Used for the pre-processing of each element to be reduced.

✓ Reduction expression: It is an operator to reduce the multiple mapped values. The special variables **a** and **b** are used for its operands.

✓ Post mapping expression: It is used to transform the resulting reduced values. The special variable **a** is used as its input. Output should be written to the output parameter.

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

REDUCTION KERNEL

Example: $z = \sum_{i=1} x_i \, w_i + b$

```
import cupy as cp
dnnLayer = cp.ReductionKernel(
      'T x, T w, T bias’, ← input params. The bias represents b from the above equation.
      'T z’, ← output params
      'x * w’, ← map
      'a + b’, ← reduce
      'z = a + bias’, ← post-reduction map
      '0’, ← identity value
      'dnnLayer’ ← kernel name
         )
x = cp.arange(10, dtype=cp.float32).reshape(2,5)
w = cp.arange(10, dtype=cp.float32).reshape(2,5)
bias = -0.1
z = dnnLayer(x,w,bias) ← kernel call
print(z)
#output
284.9
```

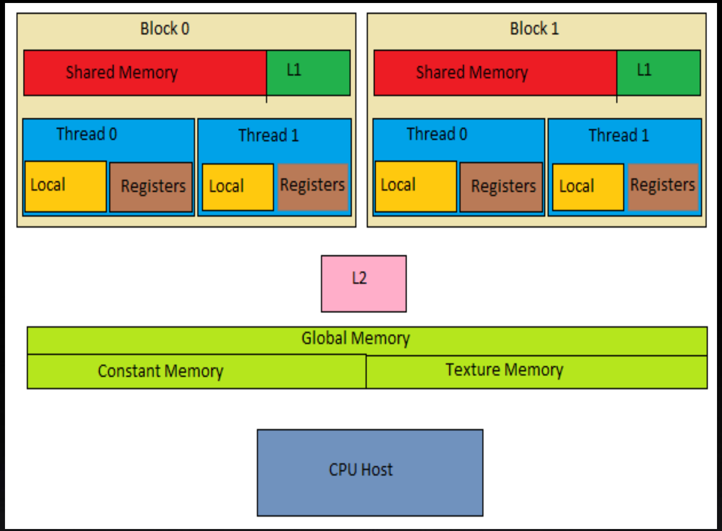
inputs

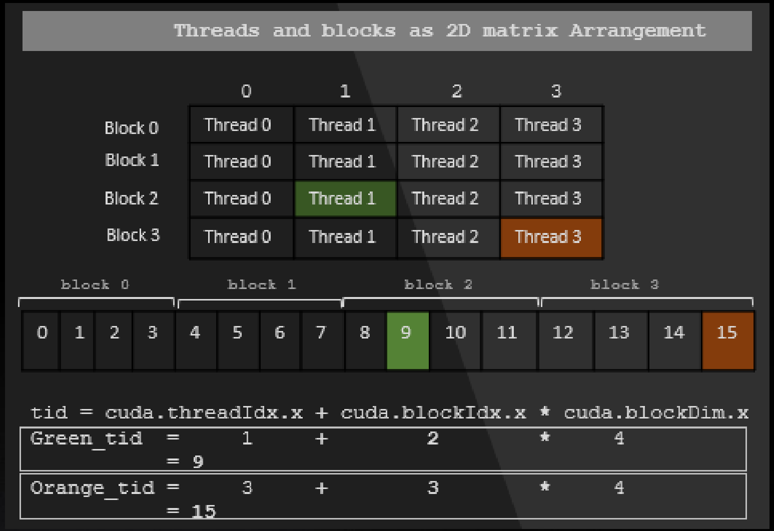
This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

RAW KERNEL

- Raw kernels enable the direct use of kernels from CUDA source, and it is defined through the ***RawKernel*** class.

- *The **RawKernel*** object allows you to call the kernel with **CUDA's cuLaunchKernel interface**. In other words, you have control over:





✓ grid size

✓ block size

✓ shared memory size

✓ and stream.

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

RAW KERNEL EXAMPLE

```
import cupy as cp
add_kernel = cp.RawKernel(r'''
extern "C" __global__
void add_func(const float* x1, const float* x2, float* y) {
int tid = blockDim.x * blockIdx.x + threadIdx.x;
y[tid] = x1[tid] + x2[tid];
}
''', 'add_func')
N = 100
shape = (10, 10)
x1 = cp.arange(N, dtype=cp.float32).reshape(shape)
x2 = cp.arange(N, dtype=cp.float32).reshape(shape)
y  = cp.zeros((shape), dtype=cp.float32)
add_kernel((10,), (10,), (x1, x2, y))
```

grid size   block size   arguments

```
#output
[[  0.   2.   4.   6.   8.  10.  12.  14.  16.  18.]
 [ 20.  22.  24.  26.  28.  30.  32.  34.  36.  38.]
 [ 40.  42.  44.  46.  48.  50.  52.  54.  56.  58.]
 [ 60.  62.  64.  66.  68.  70.  72.  74.  76.  78.]
 [ 80.  82.  84.  86.  88.  90.  92.  94.  96.  98.]
 [100. 102. 104. 106. 108. 110. 112. 114. 116. 118.]
 [120. 122. 124. 126. 128. 130. 132. 134. 136. 138.]
 [140. 142. 144. 146. 148. 150. 152. 154. 156. 158.]
 [160. 162. 164. 166. 168. 170. 172. 174. 176. 178.]
 [180. 182. 184. 186. 188. 190. 192. 194. 196. 198.]]
```

```
This also yield the same output:
add_kernel((1,), (100,), (x1, x2, y))
```


This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

RAW MODULES

- The **RawModule** class is used to defining a large raw CUDA C source or loading an existing CUDA binary.
- It is initialized by a CUDA C source code having several kernels (functions) such that needed kernels are retrieved by calling the ***get_function()*** method.

```
import cupy as cp
loaded_from_source = r'''
extern "C" {
__global__ void sum(const float* A, const float* B, float* C, int N)
 {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < N)
    {
      C[tid] = A[tid] + B[tid];
    }
 }
 __global__ void multiply(const float* A, const float* B, float* C, int N)
 {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < N)
    {
        C[tid] = A[tid] * B[tid];
    }
 }
}'''
```


This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

EXAMPLE OF RAW MODULE

```
import cupy as cp
loaded_from_source = r'''
extern "C" {
__global__ void sum_ker(const float* a, const float* b, float* c, int N)
 {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < N)
    {
      c[tid] = a[tid] + b[tid];
    }
 }
 __global__ void multiply_ker(const float* a, const float* b, float* c, int N )
 {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if(tid < N)
    {
        c[tid] = c[tid] * b[tid];
    }
 }
}'''
```

```
ker_sum((1,),(25,), (a, b, c))
print(y)
#output1
[[ 1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10.]
 [11. 12. 13. 14. 15.]
 [16. 17. 18. 19. 20.]
 [21. 22. 23. 24. 25.]]
```

```
ker_times((5,),(5,),(a, b, c))
print(y)
#output2
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.]
 [20. 21. 22. 23. 24.]]
```

```
Module    = cp.RawModule(code = load_raw_module)   ← Loading raw module
ker_sum   = module.get_function('sum_ker')         ← Accessing the sum_ker kernel
ker_times = module.get_function('multiply_ker')    ← Accessing the multiple_ker
```

```
a  = cp.arange(25,  dtype=cp.float32).reshape(5,5)
b  = cp.ones((5,5), dtype=cp.float32)
c   = cp.zeros((5,5),dtype=cp.float32)
```
← Creating arguments (input and output parameters)

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

KERNEL FUSION

- **Kernel fusion** is a decorator that fuses functions. It can be used to define an **elementwise** or **reduction** kernels easily.

```python
import cupy as cp

@cp.fuse(kernel_name='dnnlayerNode')
def dnnlayerNode(x, w, bias):
    return  (x * w) + bias


x = cp.arange(9, dtype=cp.float32).reshape(3,3)
w = cp.arange(9, dtype=cp.float32).reshape(3,3)
bias = cp.array([-0.5], dtype=cp.float32)
z = dnnlayerNode(x, w, bias)
print(z)
#output
[[-0.5   0.5   3.5]
 [ 8.5  15.5  24.5]
 [35.5  48.5  63.5]]
```

decorator

Function scope

```python
import cupy as cp

@cp.fuse
def sumlayer(x, w):
     return cp.sum(x * w, axis = -1)

x = cp.arange(10, dtype=cp.float32)
w = cp.arange(10, dtype=cp.float32)
z = sumLayer(x,w)
print(z)
#output
285.0
```


This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

KERNEL FUSION: MERITS & DEMERITS

- **Merits**

✓ Relax the bandwidth bottleneck

✓ Reduce memory consumption

✓ Speedup function calls

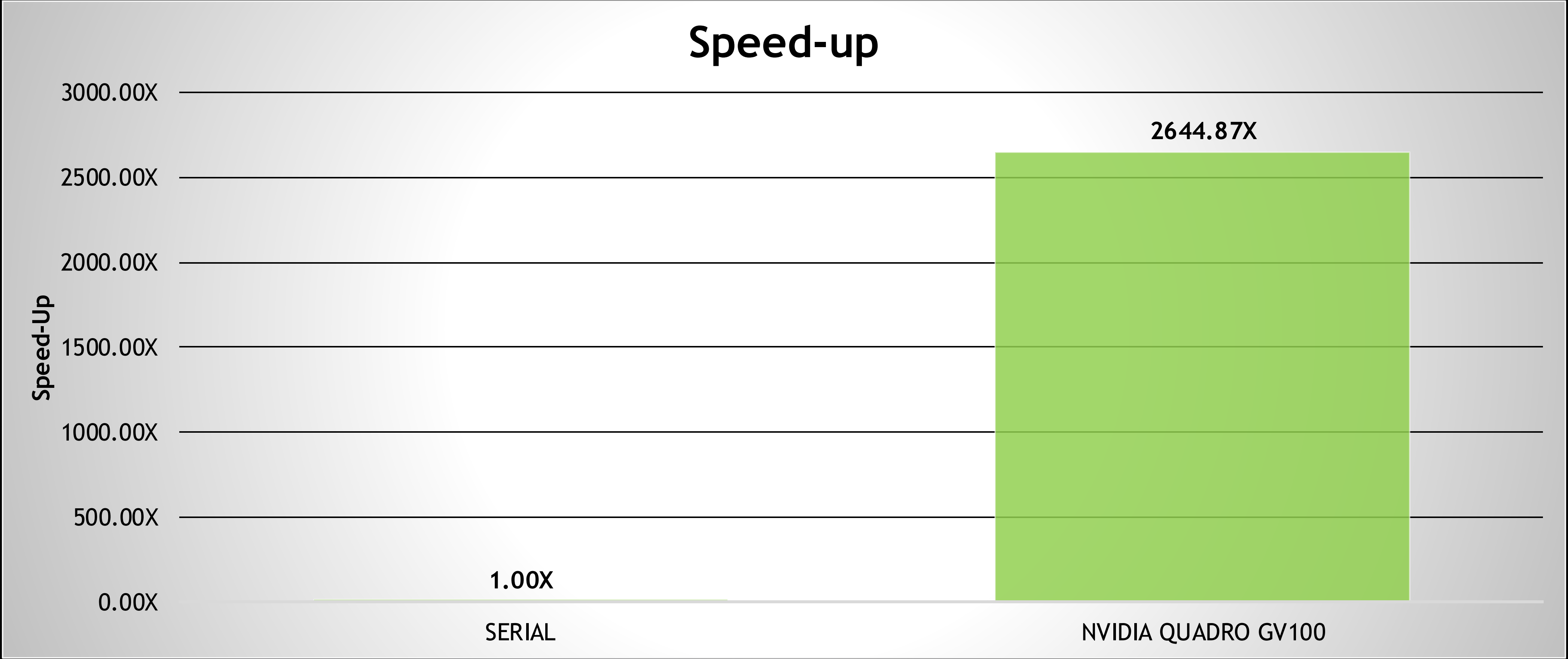
- **Demerits**

✓ No support for **cupy.matmul()** and **cupy.reshape()** operations

✓ Support only reduction and element-wise operations

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

CUPY SPEEDUP



HPC SDK 21.2, NVIDIA Quadro GV100

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

SUMMARY

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

SUMMARY

- **CuPy** is an implementation of NumPy-compatible multi-dimensional array on **CUDA**

Installation

✓ Wheels (precompiled binary package)

✓ Conda-Forge

✓ CuPy inside Docker

✓ Conda (full RAPIDS package)

Data Movement

✓ Host to Device (CPU →GPU) using cupy.asarray.

✓ Device to Host(GPU→CPU) using cupy.asnumpy or cupy.ndarray.get()

✓ between devices(GPU to GPU), cupy.ndarray is used.

Cupy User-defined Kernels

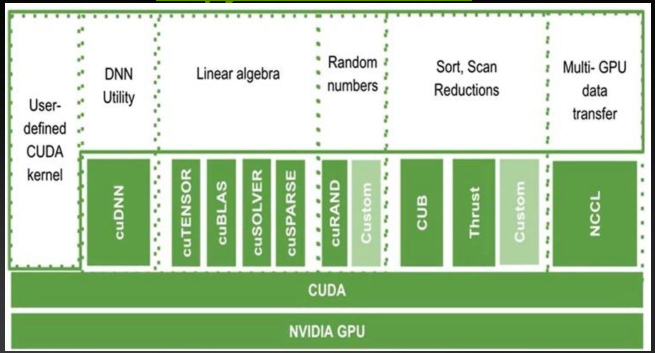
✓ Elementwise Kernels

✓ Reduction Kernels

✓ Raw Kernels

✓ Kernel Fusion

Cupy Architecture



You want to save GPU memory?

```
import cupy as cp
size = 32768
a = cp.ones((size, size)) # 8GB
b = cp.ones((size, size)) # 8GB
cp.dot(a, b)              # 8GB
```

```
Traceback (most recent call last):
  ...
cupy.cuda.memory.OutOfMemoryError: out of memory to
allocate 8589934592 bytes (total 17179869184 bytes)
```

Try Unified Memory! (Supported only on V100)

- Just edit 2 lines to enable **unified memory**

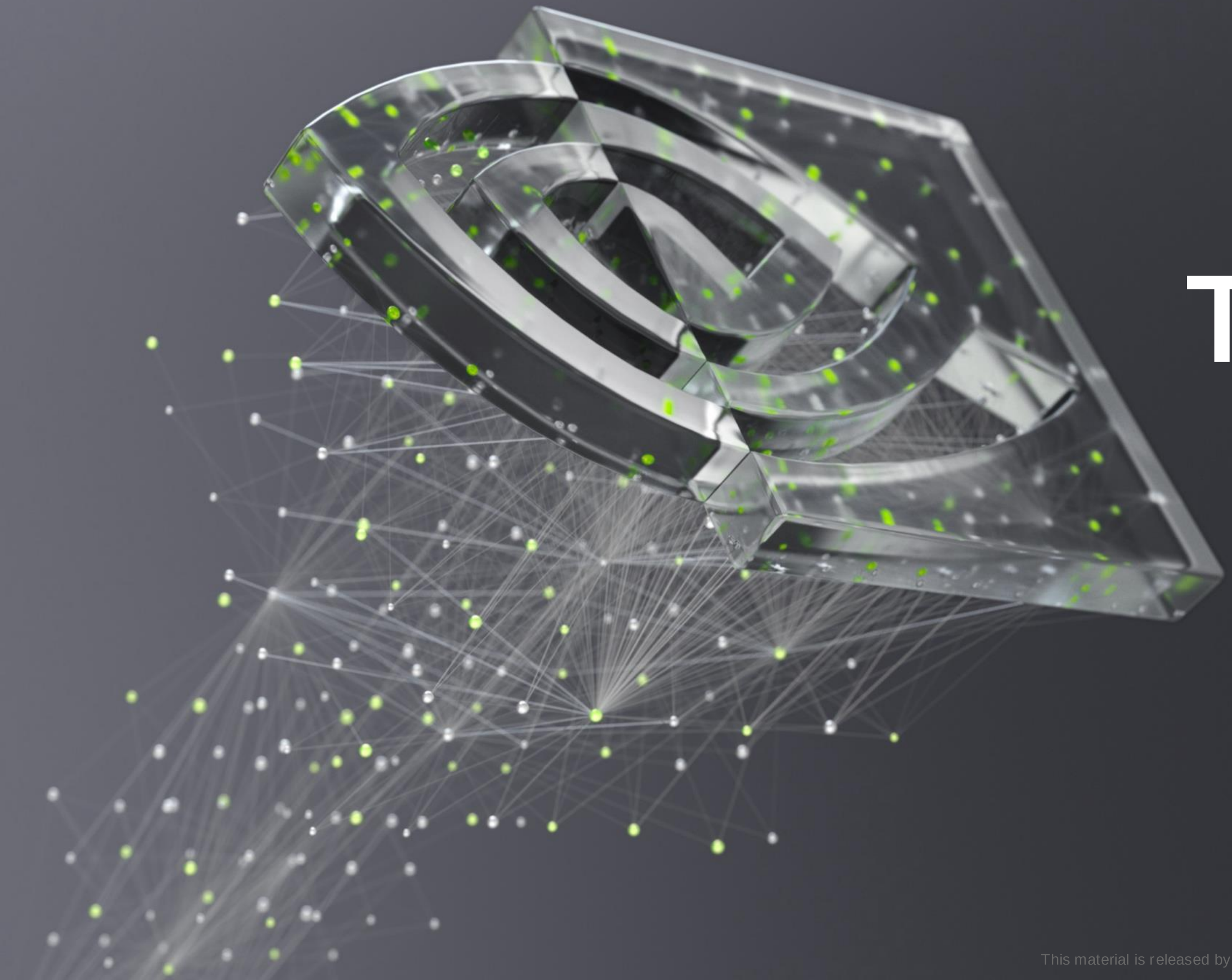
```
import cupy as cp

pool = cp.cuda.MemoryPool(cp.cuda.malloc_managed)
cp.cuda.set_allocator(pool.malloc)

size = 32768
a = cp.ones((size, size)) # 8GB
b = cp.ones((size, size)) # 8GB
cp.dot(a, b)              # 8GB
```


This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

REFERENCES

- https://developer.nvidia.com/hpc-sdk
- https://docs.cupy.dev/en/stable/
- https://cupy.dev/
- CuPy Documentation Release 8.5.0, Preferred Networks, inc. and Preferred Infrastructure inc., Feb 26, 2021.
- Bhaumik Vaidya, Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA, Packt Publishing, 2018.
- Crissman Loomis and Emilio Castillo, CuPy Overview: NumPy Syntax Computation with Advanced CUDA Features, GTC Digital March, March 2020.
- https://www.gpuhackathons.org/technical-resources
- https://rapids.ai/start.html

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)



THANK YOU

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)