NVIDIA

N-WAYS GPU BOOTCAMP

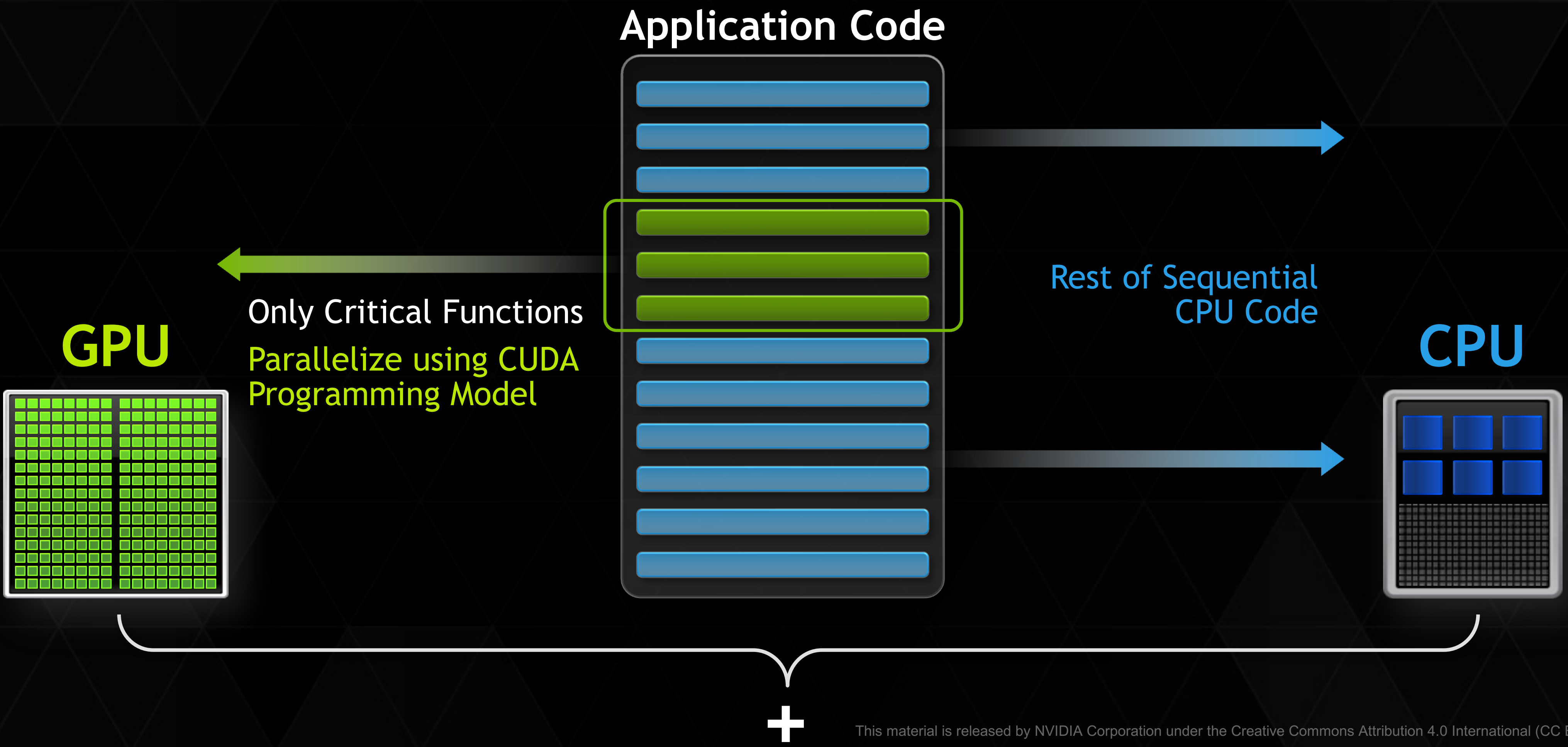
A QUICK GUIDE TO CUPY

A QUICK GUIDE TO CUPY

What to expect?

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

GPU COMPUTING



WHAT IS CUPY?

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

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]]
```

FEATURES OF CUPY

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.

CUPY FUNDAMENTALS

CUPY.NDARRAY

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

CuPy

```
import cupy as cp

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

NumPy

```
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])
```

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.]]]
```

GPU & CPU AGNOSTIC CODE

- Using **cupy.get_array_module()**

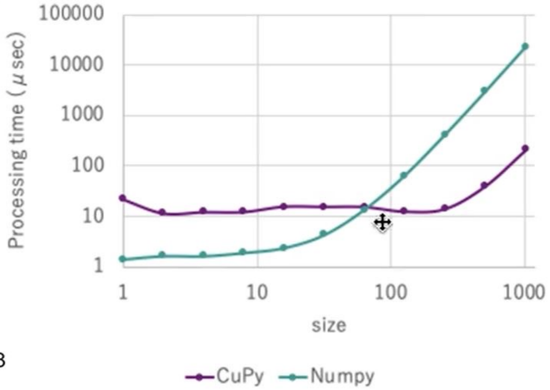
```
>>> import cupy as cp
>>> import numpy as np
>>>
>>> #example: log(1 + exp(x))
>>> x_cpu = np.array([1, 2, 3, 4, 5])
>>> x_gpu = cp.get_array_module(x_cpu)
>>> result = x_gpu.maximum(0, x_cpu)+ x_gpu.log1p(x_gpu.exp(-abs(x_cpu))
>>> result
Output: [1.31326169 2.12692801 3.04858735 4.01814993 5.00671535]
>>>
>>>
>>> #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]
>>>
```

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.



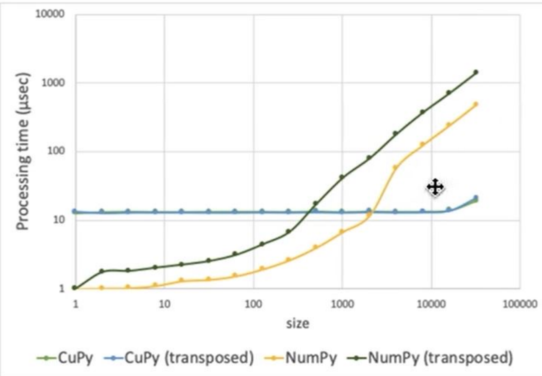
Try on Google Colab! http://bit.ly/cupywest2018

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



CUDA KERNELS

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.

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.

ELEMENTWISE KERNEL

- **Example : z = x*w + b**

```
import cupy as cp
input_list = 'float32 x , float32 w, float32 b'
output_list = 'float32 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]]
```

Data type
Input argument list
Output argument list
A loop body code

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

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.

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)     inputs
bias = -0.1
z = dnnLayer(x,w,bias)         kernel call
print(z)
#output
284.9
```

RAW KERNEL

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

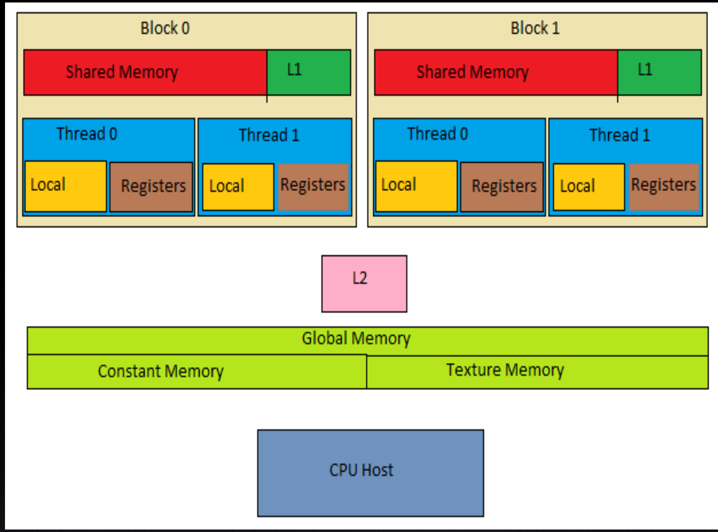
- *The **RawKernel*** object allows you to call the kernel with **CUDA's cuLaunchKernel interface**. In other words, you have control over:

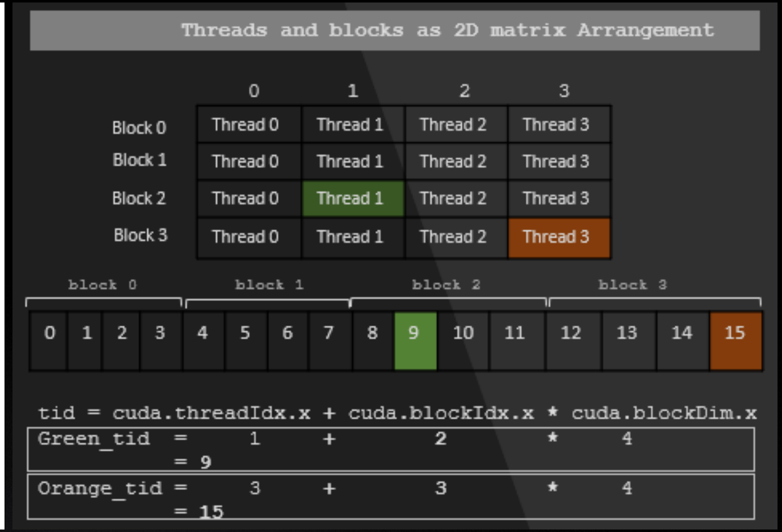
✓ grid size

✓ block size

✓ shared memory size

✓ and stream.





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))
```

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];
    }
 }
}'''
```

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)

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

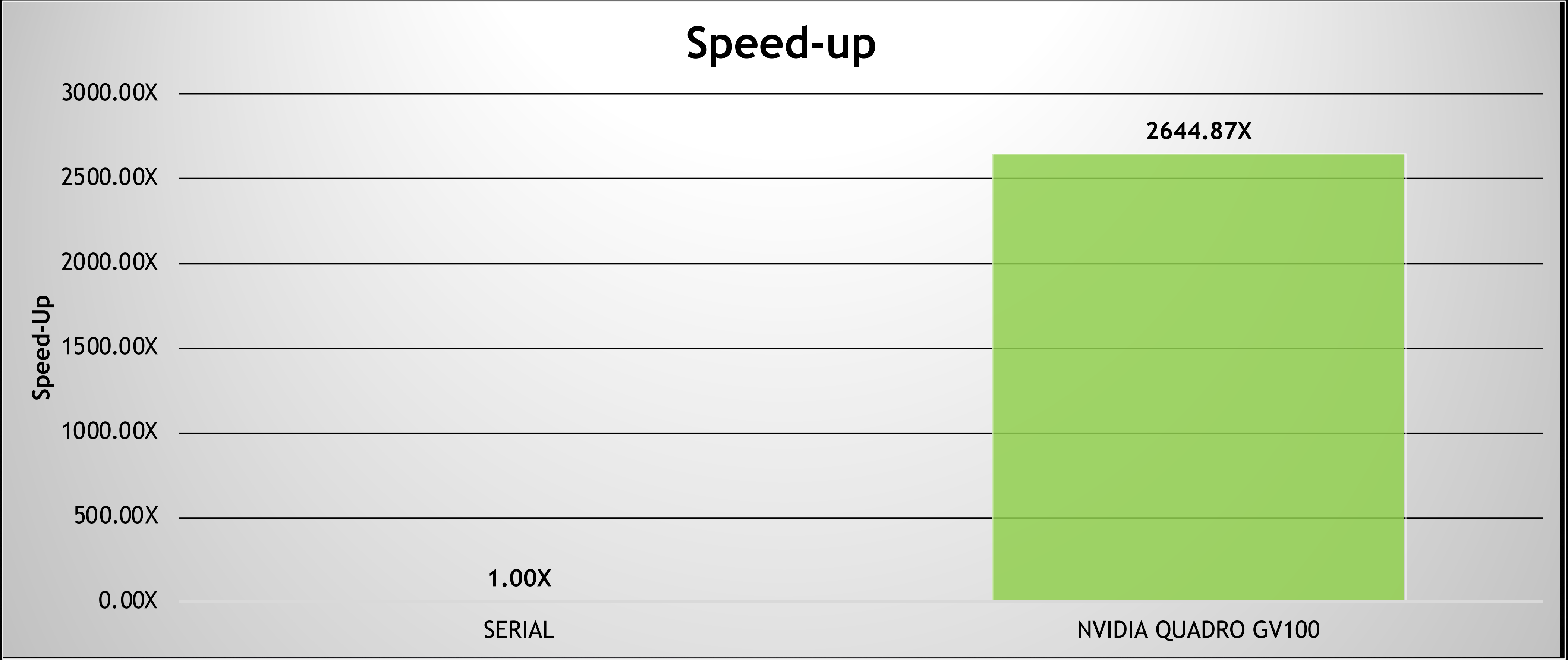
Function scope

```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
```

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

CUPY SPEEDUP



HPC SDK 21.2, NVIDIA Quadro GV100

SUMMARY

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)

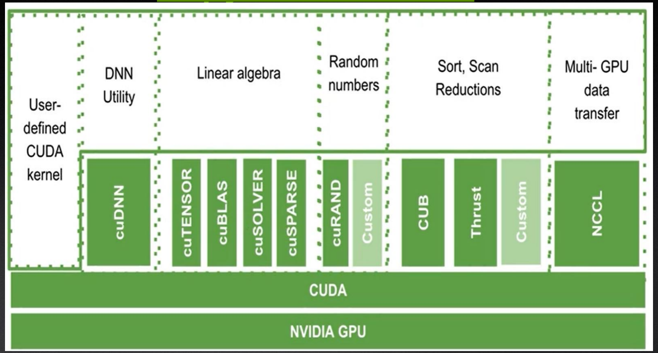
Data Movement

- ✓ 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
```

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

THANK YOU