



# ハンズオン Lab 1



# ハンズオンについて

- この講義は以下の教材をもとにしています

[https://github.com/openhackathons-org/gpubootcamp/blob/58e1329572bebc508ba7489a9f9415d7e0592ab8/hpc/nways/nways\\_labs/nways\\_MD/English/Python/jupyter\\_notebook/cupy/cupy\\_guide.ipynb](https://github.com/openhackathons-org/gpubootcamp/blob/58e1329572bebc508ba7489a9f9415d7e0592ab8/hpc/nways/nways_labs/nways_MD/English/Python/jupyter_notebook/cupy/cupy_guide.ipynb)

- Lab 1
  - CuPy による GPU コンピューティングの講義
  - ハンズオン
    - Exercise 1-4 を実施
- Lab 2
  - ハンズオンで使うコードの概要説明
  - Nsight Systems の概要説明
  - ハンズオン
    - Lab Task を実施

# ハンズオンについて

- この講義は以下の教材をもとにしています

[https://github.com/openhackathons-org/gpubootcamp/blob/58e1329572bebc508ba7489a9f9415d7e0592ab8/hpc/nways/nways\\_labs/nways\\_MD/English/Python/jupyter\\_notebook/cupy/cupy\\_guide.ipynb](https://github.com/openhackathons-org/gpubootcamp/blob/58e1329572bebc508ba7489a9f9415d7e0592ab8/hpc/nways/nways_labs/nways_MD/English/Python/jupyter_notebook/cupy/cupy_guide.ipynb)

- **Lab 1**

- **CuPy による GPU コンピューティングの講義**

- ハンズオン

- Exercise 1-4 を実施

- Lab 2

- ハンズオンで使うコードの概要説明

- Nsight Systems の概要説明

- ハンズオン

- Lab Task を実施

# Introduction

CuPy

- GPU を使って NumPy 互換の機能を提供するライブラリ



```
import numpy as np
X_cpu = np.zeros((10,))
W_cpu = np.zeros((10, 5))
y_cpu = np.dot(x_cpu, W_cpu)
```

```
y_cpu = cp.asnumpy(y_gpu)
```



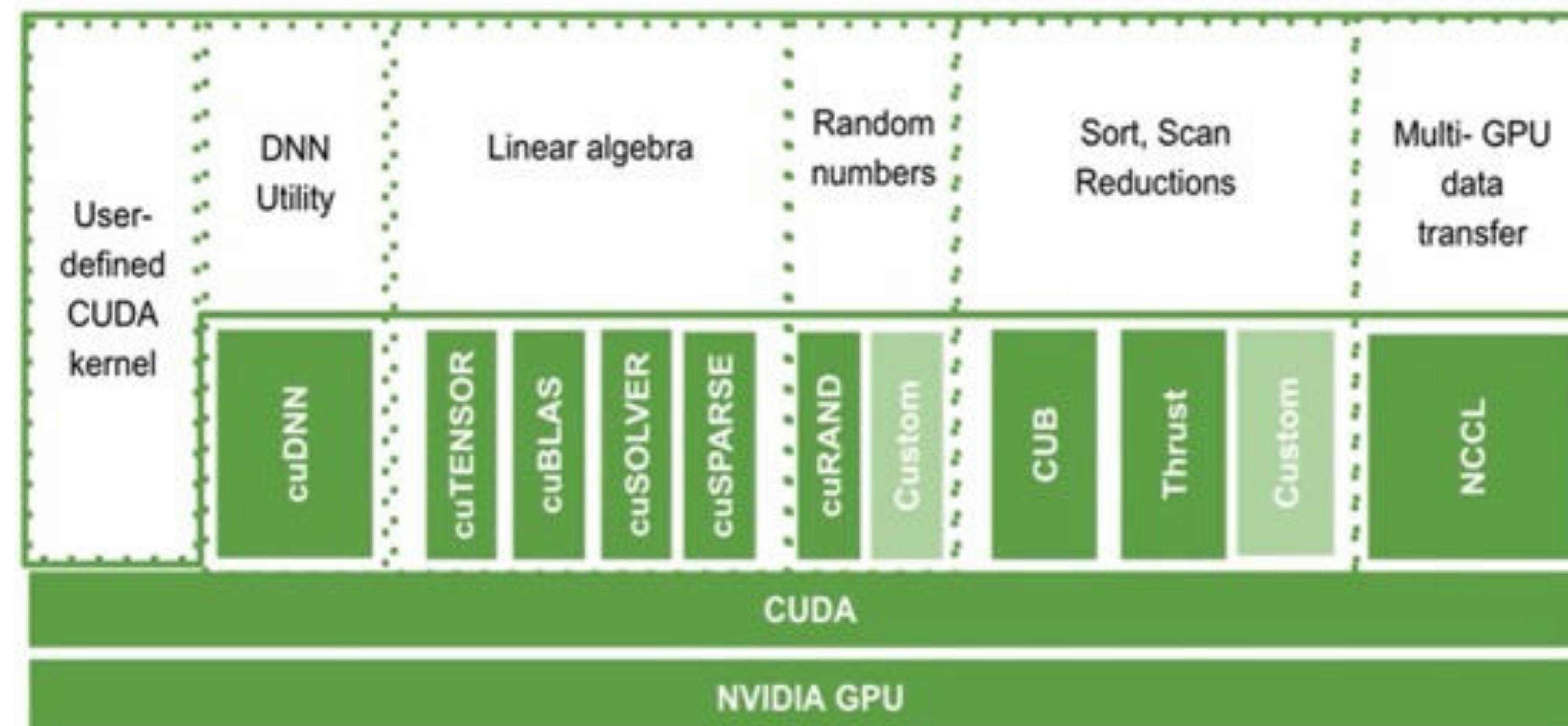
```
import cupy as cp
x_gpu = cp.zeros((10,))
W_gpu = cp.zeros((10, 5))
y_gpu = cp.dot(x_gpu, W_gpu)
```

```
y_gpu = cp.asarray(y_cpu)
```



# CuPy Architecture

- CuPy API を通して以下を利用できる
  - ユーザーが独自に定義した CUDA カーネル
  - ディープニューラルネットワークのためのライブラリである cuDNN
  - cuTensor, cuBLAS, cuSOLVER, cuSPARSE が提供する線形代数のための機能
  - cuRAND が提供する乱数生成
  - CUB, Thrust が提供する sort, scan reduction などの基本アルゴリズム
  - 複数 GPU 間の通信のためのライブラリである NCCL が提供する機能





# CuPy Fundamentals (0)

- 数ある CuPy の機能の中から、3つの機能に絞って、基本的な使い方を紹介
  - Variable or data initialization
  - Data transfer
  - Device selection



# CuPy Fundamentals (1)

## Variable or data initialization

- CuPy ndarray (GPU 上のメモリ) を確保する
- CuPy ndarray にデータ タイプ、値をアサインする

```
import cupy as cp
X1 = cp.array([1,2,3,4,5,6,7,8,9,10], dtype=cp.int32) #array of 10 values
X2 = cp.arange(100, dtype=cp.float32) #generating array of 100 values
X3 = cp.empty((3,3), dtype=cp.float32) #initializing empty 2D array of 3X3 matrix
sizebin = 10000
X4 = cp.zeros(sizebin, dtype=cp.int64) #initializing array filled with 10,000 zeros
```



# CuPy Fundamentals (2)

## Data transfer

- Host (NumPy) <-> Device (CuPy) 間のデータ移動
- `cp.asarray()` : Host -> Device
- `cp.asnumpy()` : Host <- Device

```
import numpy as np
import cupy as cp

#copy data from Host to Device using cp.asarray()
h_X = np.arange(100, dtype=np.float32) #generating array of 100 values on the Host with NumPy
d_X = cp.asarray(h_X) # copy data to Device

#copy data from Device to Host using cp.asnumpy()
h_X = cp.asnumpy(d_X)
```



# CuPy Fundamentals (3)

## Device selection

- デフォルトの Device ID は 0
  - デフォルトのデバイスを使う

```
X1 = cp.array([1,2,3,4,5,6,7,8,9,10], dtype=cp.int32)
```

- デバイスを切り替える

```
cp.cuda.Device(1)  
X1 = cp.array([1,2,3,4,5,6,7,8,9,10], dtype=cp.int32)
```

- 一時的にデバイス 2 に切り替える

```
with cp.cuda.Device(2):  
    X2 = cp.arange(100, dtype=cp.float32)  
Sizebin = 10000  
X4 = cp.zeros(sizebin, dtype=cp.int64) # back to default GPU with index 0
```



# Example 1

配列 A, B の和の計算

```
import cupy as cp

N = 10000

#select Device with index 1.
with cp.cuda.Device(1):
    #input data initialized
    d_A = cp.arange(N, dtype=cp.int32)
    d_B = cp.arange(N, dtype=cp.int32)
    d_C = cp.zeros(N, dtype=cp.int32) # initialize zero filled array
    d_C = d_A + d_B

#optional: copy result from Device to Host
h_C = cp.asnumpy(d_C)
print(h_C)
```



# Example 1 を使ってジョブ投入の練習 (1)

```
$ cd /work/EDU5/ユーザ名/hands-on/Example1
$ ls
example1.py  submit.sh
$ cp /work/EDU5/naruhikot/share/Example1_modified/* .
$ ls
example1.py  example1_modified.py  submit.sh  submit_modified.sh
```

このコマンドを実行してください  
example1\_modified.py と submit\_modified.sh  
をコピーします

通常とは異なるキュー構成のため、  
example1.py そのままだと、  
(おそらく)動作しません

submit\_modified.sh

```
#!/bin/bash

#PBS -A EDU5                #プロジェクト名 (今回の実習用、変更不要)
#PBS -q edu-b              #バッチキュー名 (今回の実習用、変更不要)
#PBS -l elapstim_req=00:05:00 #経過時間制限値 (実行に必要な十分な時間を設定)

module load python/3.8 cuda/11.6.2 #必要なモジュールの読み込み

cd $PBS_0_WORKDIR          #リクエストを投入 (qsub) したディレクトリに移動

source /work/EDU5/$USER/hands-on/venv/bin/activate #python 仮想環境の有効化

python example1_modified.py #コードの実行
```



## Example 1 を使ってジョブ投入の練習 (2)

```
$ qsub submit_modified.sh
Request xxxxxx.nqsv submitted to queue: edu-b.
$ qstat
```

RequestID	ReqName	UserName	Queue	Pri	STT	S	Memory	CPU	Elapse	R	H	M	Jobs
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
xxxxxx.nqsv	submit_m	username	edu-b	0	PRR	-	0.00B	0.00	0	N	Y	Y	1

- ジョブが終了すると、<ジョブスクリプト名>.exxxx, <ジョブスクリプト名>.oxxxx ようなファイルが出力される

```
$ ls
example1.py          submit.sh          submit_modified.sh.exxxxxx
example1_modified.py submit_modified.sh submit_modified.sh.oxxxxxx
```

- submit\_modified.sh.oxxxxxx の中身を確認して、以下のような出力が得られていれば成功

```
$ cat submit.oxxxxxx
[ 0 2 4 ... 19994 19996 19998]
```



# これ以降の Example について

- これ以降の Example は実行していただく必要はありません
- CuPy の機能を説明するための例題として、Example を使います
- 講義が終わった後に、Exercise に取り組んでいただきます



# 2-Demensional Arrays

## 行列-行列積

- 2つの Example を通して CuPy での行列-行列積の計算を紹介
- Example 2
  - 挙動を確認しやすいように行列サイズは  $4 \times 4$
- Example 3
  - 行列サイズは  $10,000 \times 10,000$
  - 行列要素の値は cuRAND で生成



# Example 2

行列 A, B の積の計算

```
A = [[0 0 0 0]
      [1 1 1 1]
      [2 2 2 2]
      [3 3 3 3]]
B = [[0 1 2 3]
      [0 1 2 3]
      [0 1 2 3]
      [0 1 2 3]]
A x B = [[ 0  0  0  0]
          [ 0  4  8 12]
          [ 0  8 16 24]
          [ 0 12 24 36]]
N = 4; Shape = (N,N)
```

```
N = 4
A = cp.array([[0,0,0,0],[1,1,1,1],[2,2,2,2],[3,3,3,3]], dtype=cp.int32)
B = cp.array([[0,1,2,3],[0,1,2,3],[0,1,2,3],[0,1,2,3]], dtype=cp.int32)

C = cp.dot(A,B)
C2 = A@B
print("dot ops:", C)
print("@ ops:", C2)

#expected output
#dot ops:
#[[ 0  0  0  0]
# [ 0  4  8 12]
# [ 0  8 16 24]
# [ 0 12 24 36]]

#@ ops:
#[[ 0  0  0  0]
# [ 0  4  8 12]
# [ 0  8 16 24]
# [ 0 12 24 36]]
```

# Example 3

## 行列 d\_A, d\_B の積の計算

- Step 1 : CuPy のインポートと配列サイズの指定

```
import cupy as cp
N = 10000
```

- Step 2 : 配列の値を生成 (cuRAND)

```
d_A = cp.random.random((N,N), dtype=cp.float32)
d_B = cp.random.random(N*N, dtype=cp.float32).reshape(N, N)
```

- Step 3 : @ 演算子で行列積を計算 (cuBLAS)

```
d_C = d_A@d_B
print(d_C)

#expected output
...
[2496.929 2493.3096 2512.024 ... 2523.2388 2486.2688 2502.8193]
[2512.366 2522.0713 2518.3489 ... 2529.164 2493.486 2488.1067]
[2493.215 2483.601 2493.606 ... 2523.578 2474.8271 2469.6057]
```



# Kernel Fusion

- いくつかの演算を融合した CUDA コードを実行時に動的に生成することにより、GPU カーネルの起動を減らしたり、グローバルメモリへのアクセスを減らすことによる高速化を行う
  - カーネル融合について詳細はこちらを参照: <https://tech.preferred.jp/ja/blog/cupy-kernel-fusion-extension/>
- Example 4
  - Kernel fusion を活用したベクトルの内積
- @cp.fuse で関数をデコレート

```
@cp.fuse(kernel_name='<function_name>')  
def function_name(<arguments>):  
    #<body code>
```

または

```
@cp.fuse()  
def function_name(<arguments>):  
    #<body code>
```

# Example 4

の計算

$$z = \sum_{i=0}^{N-1} x_i w_i$$

```
import cupy as cp

@cp.fuse()
def compute(x,w):
    return cp.sum(x * w)

N = 225
#input data
x = cp.random.random((N), dtype=cp.float32)
w = cp.random.random((N), dtype=cp.float32)

#calling fuse function
z = compute(x,w)
print(z)
```



# CuPy CUDA Kernels

以下のユーザ定義 CUDA kernel を紹介

- Elementwise kernels
  - Example 5
- Reduction kernels
  - Example 6
- Raw kernels
  - Example 7
- Raw modules
  - Example 8
- JIT kernel

# Elementwise Kernels

- ElementwiseKernel クラス
- 引数は以下の4つ
  - Input argument list
  - Output argument list
  - Loop body code
  - Kernel name
- データタイプは明示的に指示 or generic form
- 変数名として `n`, `i` は使用不可
- `_` で始まる変数名は使用不可



## Example 5

$$r = \sqrt{x^2 + y^2 + z^2} \text{ の計算}$$

- Step 1 : 入出力の引数とデータタイプを定義

```
input_list = 'float32 d_x, float32 d_y, float32 d_z'  
output_list = 'float32 r'
```

or

```
input_list = 'T d_x, T d_y, T d_z'  
output_list = 'T r'
```

- Step 2 : kernel body code を定義

```
code_body = 'r = sqrt(d_x*d_x + d_y*d_y + d_z*d_z)'
```

- Step 3 : Elementwise kernel を定義

```
compute_call = cp.ElementwiseKernel(input_list, output_list, code_body, 'compute')
```

- Step 4 : 配列の初期化

```
N = 2000  
d_x = cp.arange(N, dtype=cp.float32)  
d_y = cp.arange(N, dtype=cp.float32)  
d_z = cp.arange(N, dtype=cp.float32)  
r = cp.empty(N, dtype=cp.float32)
```

- Step 5 : カーネルのコール

```
compute_call(d_x, d_y, d_z, r)
```

# Reduction Kernels

- ReductionKernel クラス
- 引数は以下
  - Input argument list
  - Output argument list
  - Mapping expression
  - Reduction expression
  - Post mapping expression
  - Identity value
  - Kernel name
- Mapping expression は、reduction 前の処理
- Reduction expression では、演算の指示として特別な変数 a, b が使われる
- Post mapping expression は、reduction 後の変数に対して行われる処理で、入力として変数 a が使われ、演算結果は output argument を指定



## Example 6

$$y = \sum_{i=0}^{N-1} x_i w_i + \text{bias} \quad \text{の計算}$$

- Step 1 : 入出力の引数とデータタイプを定義

```
input_list = 'float32 x, float32 w, float32 bias'  
output_list = 'float32 y'
```

or

```
input_list = 'T x, T w, T bias'  
output_list = 'T y'
```

- Step 2 : mapping expression を定義

```
mapping_expr = 'x * w'
```

- Step 3 : reduction expression を定義

```
reduction_expr = 'a + b'
```

- Step 4 : post expression を定義

```
post_expr = 'y = a + bias'
```

- Step 5 : Identity value を定義

```
identity_value = '0'
```

# Example 6

Cont'd

- Step 6 : Reduction kernel を定義

```
dnnLayer = cp.ReductionKernel(  
    input_list,  
    output_list,  
    mapping_expr,  
    reduction_expr,  
    post_expr,  
    identity_value,  
    'dnnLayer')
```

- Step 7 : 入力引数の初期化

```
N = 2000  
x = cp.random.random(N, dtype=cp.float32)  
w = cp.random.random(N, dtype=cp.float32)  
bias = -0.01
```

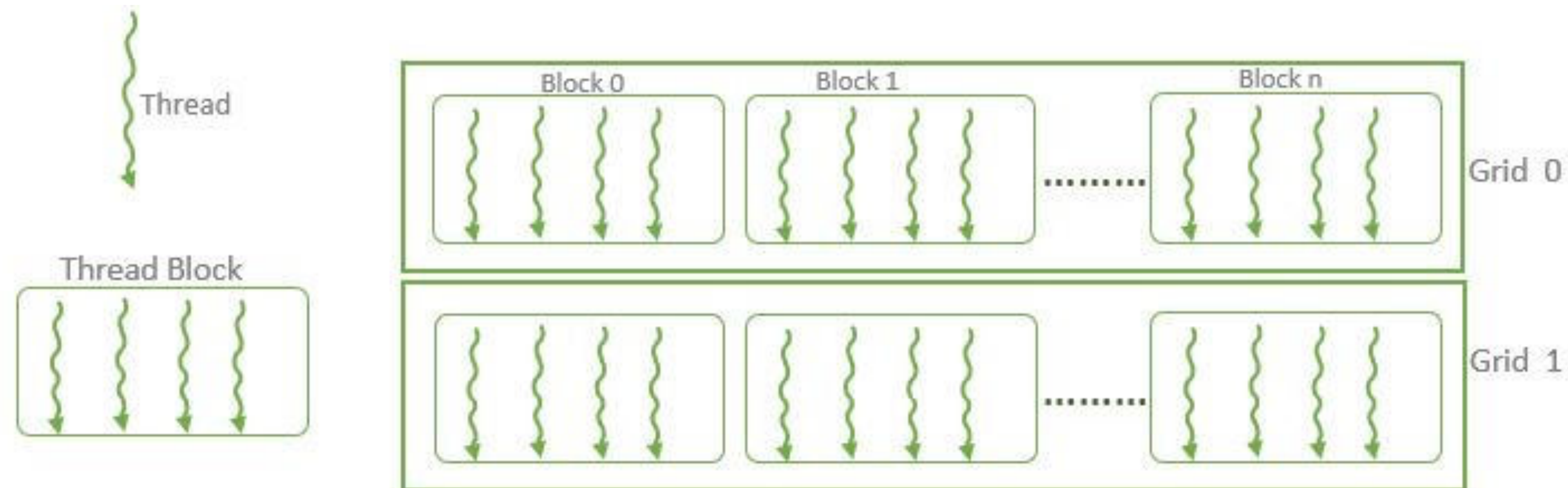
- Step 8 : カーネルのコール

```
y = dnnLayer(x, w, bias)
```



# Raw Kernels

- RawKernel クラス
- 処理内容を CUDA C で記述
- スレッド、ブロック、グリッド…
- カーネル起動に際してブロックサイズ、ブロック数を指定
- 最大スレッド数/ブロックは 1,024 で、最大ブロック数/グリッドは 65,535



# Example 7

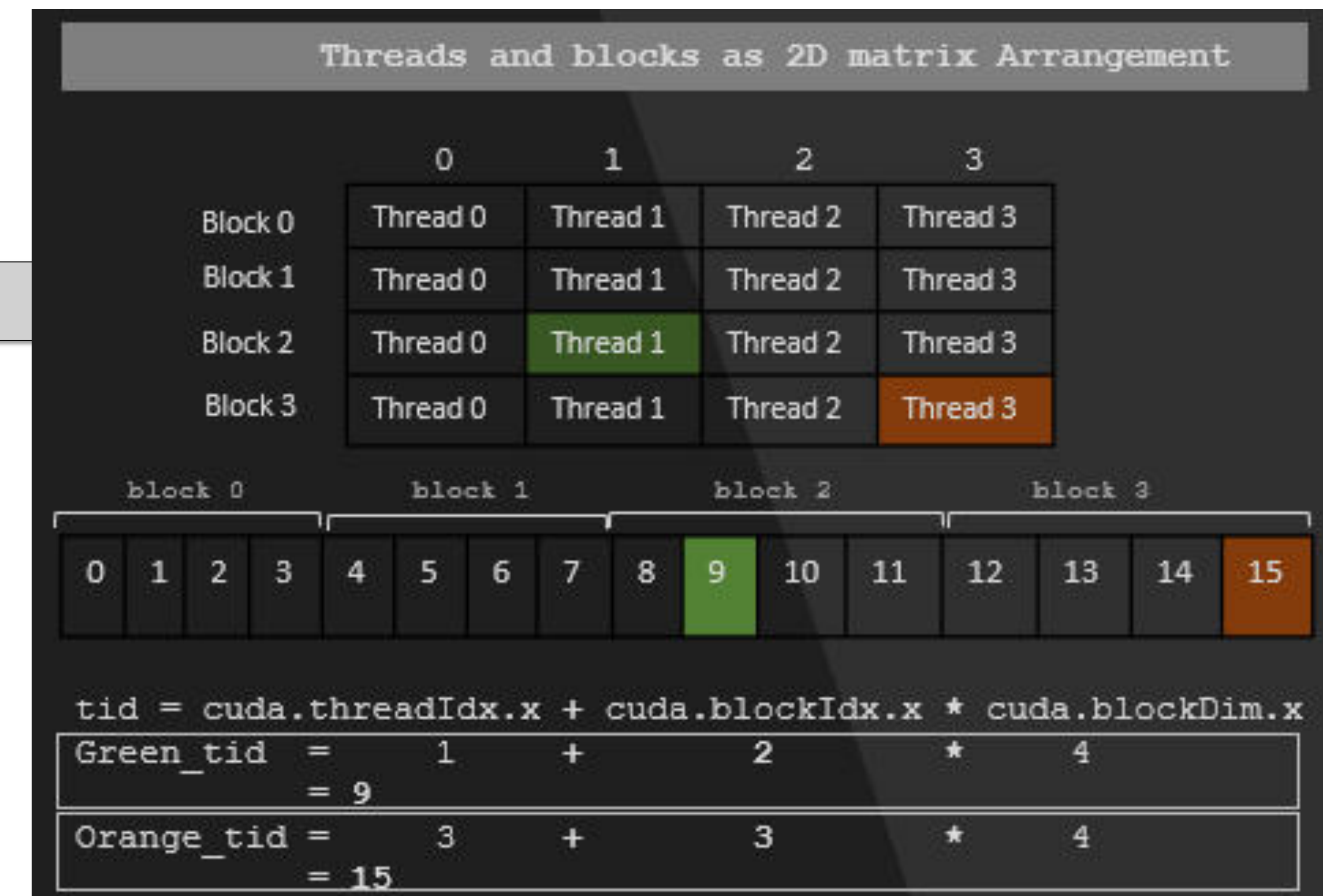
## 配列同士の和の計算

- Step 1: デバイスコードを書く
  - CuPy をインポートし、raw kernel function を定義

```
import cupy as cp
add_array = cp.RawKernel(r'''
extern "C" __global__
void <function_name>(<arguments>) {
    <body code>
} ''', '<function_name>')
```

- 処理内容を CUDA C で記述
  - 基本 --- 1つのGPUスレッドが、1つの配列要素を担当
  - Global スレッド ID の取得

```
tid = threadIdx.x + blockIdx.x * blockDim.x
```





# Example 7

Cont'd

- 以上を踏まえると、、、

```
import cupy as cp

N = 10000 #initialize array size

add_array = cp.RawKernel(r'''
extern "C" __global__
void addFunc(const int* d_A, const int* d_B, int* d_C ) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    d_C[tid]= d_A[tid] + d_B[tid];
} ''', 'addFunc')
```

# Example 7

Cont'd

- Step 2 : ホスト コードを書く
  - 入力配列の初期化

```
import numpy as np
h_A = np.arange(N, dtype=np.int32)
h_B = np.arange(N, dtype=np.int32)
```

- `cp.asarray()` を使ってホストからデバイスへデータ転送

```
d_A = cp.asarray(h_A)
d_B = cp.asarray(h_B)
d_C = cp.zeros(N, dtype=cp.int32) # initialize zero filled array
```



# Example 7

Cont'd

- Step 3 : Raw kernel のコール

- ブロックサイズ、ブロック数の指定

```
<raw_kernel_name>((<num_of_blocks_per_grid>), (<num_of_threads_per_block>), (<arguments>))
```

- 配列サイズをカバーするスレッドを起動する必要があるので、ブロックサイズを 256 と仮定すると、

```
num_of_threads_per_block = 256
```

- (グリッドあたりの) ブロック数は、

```
num_of_blocks_per_grid = math.ceil (N / num_of_threads_per_block)
```

- 以上を踏まえて、raw kernel は以下のようにコールされる

```
add_array((num_of_blocks_per_grid, ), (num_of_threads_per_block, ), (d_A, d_B, d_C))
```

# Example 7

Cont'd

- Step 4 : `cp.asnumpy()` を使って、デバイスからホストへデータ転送

```
h_C = cp.asnumpy(d_C)
```



# Raw Modules

- RawModule クラス
- RawKernel と同様に、処理内容を CUDA C で記述
- Raw module には、複数の CUDA kernel を含めることができる
- `get_function()` メソッドで、Raw module 内の CUDA kernel にアクセス

## Example 8

$$z = \sum_{i=0}^{N-1} x_i w_i \quad r = \sqrt{x^2 + y^2 + z^2} \quad \text{の計算}$$

- RawModule クラスを使って、2つの CUDA kernel、sum\_mul と compute\_xyz を定義
- sum\_mul では、足し込みを行う前にブロック中のスレッドの同期をとるために、\_\_syncthread()
- atomicAdd() により、あるスレッドが足し込みの実行 (read-modify-write) 中に、他のスレッドが同じメモリ領域に書き込むのを防止
  - [https://www.olcf.ornl.gov/wp-content/uploads/2019/12/05\\_Atomics\\_Reductions\\_Warp\\_Shuffle.pdf](https://www.olcf.ornl.gov/wp-content/uploads/2019/12/05_Atomics_Reductions_Warp_Shuffle.pdf)

```
raw_module_code = r'''
extern "C" {
    __global__ void sum_mul(float* d_x, float* d_w, float* d_z) {
        float sum[2000];
        int tid = blockDim.x * blockIdx.x + threadIdx.x;
        sum[tid] = d_x[tid] * d_w[tid];
        __syncthreads();
        atomicAdd(d_z, sum[tid]);
    }

    __global__ void compute_xyz(float* x, float* y, float* z, float* r ) {
        int tid = blockDim.x * blockIdx.x + threadIdx.x;
        r[tid] = sqrt(x[tid] * x[tid] + y[tid] * y[tid] + z[tid] * z[tid]);
    }
}
'''
```

# Example 8

Cont'd

- RawModule クラスからオブジェクトを生成

```
raw_module_object = cp.RawModule(code = raw_module_code)
```

- get\_function() メソッドで、raw module 内の CUDA kernel を取得

```
sum_mul = raw_module_object.get_function('sum_mul')  
compute_xyz = raw_module_object.get_function('compute_xyz')
```

- 配列サイズ、ブロックサイズ、ブロック数の初期化

```
N = 2000  
num_of_threads_per_block = 128  
num_of_blocks_per_grid = math.ceil(N / num_of_threads_per_block)
```

- 配列の初期化とデバイスへのデータ転送 (sum\_mul の例)

```
h_x = np.arange(N, dtype=np.float32)  
h_w = np.arange(N, dtype=np.float32)  
d_x = cp.asarray(h_x)  
d_w = cp.asarray(h_w)  
d_z = cp.zeros(1, dtype=cp.float32)
```



# Example 8

Cont'd

- `sum_mul` のコールと、ホストへのデータ転送

```
sum_mul((num_of_blocks_per_grid,),(num_of_threads_per_block,),(d_x, d_w, d_z))
h_z = cp.asnumpy(d_z)
print("h_z:", h_z)
```

- 計算結果の確認

```
print("non kernel:", cp.sum(h_x * h_w))
```

- `compute_xyz` のための配列の初期化と、デバイスへのデータ転送

```
x = cp.arange(N, dtype=cp.float32)
y = cp.arange(N, dtype=cp.float32)
z = cp.arange(N, dtype=cp.float32)
r = cp.empty(N, dtype=cp.float32)
```

- `compute_xyz` のコールと、ホストへのデータ転送

```
compute_xyz((num_of_blocks_per_grid,),(num_of_threads_per_block,),(x, y, z, r))
h_r = cp.asnumpy(r) print("h_r:", h_r)
```

- 計算結果の確認

```
print("non kernel:", cp.sqrt(x * x + y * y + z * z ))
```

# JIT Kernel

- デコレータ `cupyx.jit.rawkernel` で実現
- 基本的には raw kernel と同じだが、jit kernel では CUDA C の代わりに python で関数を記述する
- Example 7 (raw kernel) を jit kernel で書き直し

- 必要なモジュールをインポート

```
import cupy as cp
from cupyx import jit
```

- Jit kernel の記述

```
@jit.rawkernel()
def addFunc(d_A, d_B, d_C):
    tid = jit.blockDim.x * jit.blockIdx.x + jit.threadIdx.x
    d_C[tid] = d_A[tid] + d_B[tid]
```

# JIT Kernel

Cont'd

- データサイズ、ブロックサイズ、ブロック数の初期化

```
N = 10000 #initialize array size
num_of_threads_per_block = 128
num_of_blocks_per_grid = math.ceil(N / num_of_threads_per_block)
```

- `cp.arange` で配列をデバイス上に確保、初期化

```
d_A = cp.arange(N, dtype=cp.int32)
d_B = cp.arange(N, dtype=cp.int32)
d_C = cp.zeros(N, dtype=cp.int32) # initialize zero filled array
```

- `addFunc` をコール

```
addFunc((num_of_blocks_per_grid,), (num_of_threads_per_block,), (d_A, d_B, d_C))
print("d_C:", d_C)
```



# ハンズオンについて

- この講義は以下の教材をもとにしています

[https://github.com/openhackathons-org/gpubootcamp/blob/58e1329572bebc508ba7489a9f9415d7e0592ab8/hpc/nways/nways\\_labs/nways\\_MD/English/Python/jupyter\\_notebook/cupy/cupy\\_guide.ipynb](https://github.com/openhackathons-org/gpubootcamp/blob/58e1329572bebc508ba7489a9f9415d7e0592ab8/hpc/nways/nways_labs/nways_MD/English/Python/jupyter_notebook/cupy/cupy_guide.ipynb)

- **Lab 1**

- CuPy による GPU コンピューティングの講義
- **ハンズオン**
  - **Exercise 1-4 を実施**

- Lab 2

- ハンズオンで使うコードの概要説明
- Nsight Systems の概要説明
- ハンズオン
  - Lab Task を実施

# ハンズオン

- Exercise 1

- Follow the steps highlighted in Example 1 and write a CuPy program to add two arrays. The size of each array is 500,000.
- 回答例については、以下のコマンドを実行して、回答例の修正版をコピーしてください

```
$ cd /work/EDU5/<ユーザ名>/hands-on/Lab1/exercise1/example_solution
$ cp /work/EDU5/naruhikot/share/exercise1_modified/* .
$ ls
exercise1.py  exercise1_modified.py  submit.sh  submit_modified.sh
```

- Exercise 2

- Follow the steps highlighted in Example 3 and write a CuPy program that multiply two matrices of dimensions 225 x 225.

- Exercise 3

- Follow the steps highlighted in Example 7 and write a CuPy Raw Kernel program that multiply two arrays and store the result in a third array. The size of each array is 10,000.

- Exercise 4

- Follow the steps highlighted above and write a CuPy JIT Kernel program that multiply two arrays and store the result in a third array. The size of each array is 10,000.

- 必要に応じて、前記講義教材で復習

- 注意事項：ジョブを投入する際には、qstat で自分のジョブが積まれていないことを確認してから！

# 回答例

## Exercise 1

```
import cupy as cp

N = 500000

d_A = cp.arange(N, dtype=cp.int32)
d_B = cp.arange(N, dtype=cp.int32)
d_C = cp.zeros(N, dtype=cp.int32)

d_C = d_A + d_B

h_C = cp.asnumpy(d_C)
print(h_C)
```



# 回答例

## Exercise 2

```
N = 225

#generate matrix
d_A = cp.random.random((N, N), dtype=cp.float32)
d_B = cp.random.random(N*N, dtype=cp.float32).reshape(N, N)

#apply matrix operator @ or cp.dot()
d_C = d_A @ d_B
d_C2 = cp.dot(d_A, d_B)

print(d_C)
print(d_C2)
```

# 回答例

## Exercise 3

```
import cupy as cp
import numpy as np
import math

N = 10000 #initialize array size

add_array = cp.RawKernel(r'''
    extern "C" __global__
    void addFunc(const int *d_A, const int *d_B, int *d_C) {
        int tid = threadIdx.x + blockDim.x * blockIdx.x;
        d_C[tid] = d_A[tid] * d_B[tid];
    }
''', 'addFunc')

h_A = np.arange(N, dtype=np.int32)
h_B = np.arange(N, dtype=np.int32)

d_A = cp.asarray(h_A)
d_B = cp.asarray(h_B)
d_C = cp.zeros(N, dtype=cp.int32)

num_threads_per_block = 256
num_blocks_per_grid = math.ceil(N / num_threads_per_block)

add_array((num_blocks_per_grid,), (num_threads_per_block,), (d_A, d_B, d_C))
h_C = cp.asnumpy(d_C)
print(h_C)
```

# 回答例

## Exercise 4

```
import cupy as cp
from cupyx import jit
import math

N = 10000 #initialize array size

@jit.rawkernel()
def mulFunc(d_A, d_B, d_C):
    tid = jit.threadIdx.x + jit.blockDim.x * jit.blockIdx.x
    d_C[tid] = d_A[tid] * d_B[tid]

num_threads_per_block = 128
num_blocks_per_grid = math.ceil(N / num_threads_per_block)

d_A = cp.arange(N, dtype=cp.int32)
d_B = cp.arange(N, dtype=cp.int32)
d_C = cp.zeros(N, dtype=cp.int32)

mulFunc( (num_blocks_per_grid,), (num_threads_per_block,), (d_A, d_B, d_C) )
print("d_C:", d_C)
```



