コースの概要

この動画で学ぶこと

- ✓ このコースはどんなコースかが分かる
- ✓ コースの進め方を理解できる

本コースの対象者/前提条件

対象者

Pythonを使ったCUDA*での並列計算を身に付けたい方

***NVIDIAが提供するGPU向けのプログラミング言語**

前提条件

Pythonのプログラミングの知識(基本的な知識+Numpy)
数値計算の経験がある方が望ましい(*必須の条件ではない)

本講座の内容

Pythonでの数値計算を逐次計算ではなく並列計算(GPU)で行う方法を学ぶ講座

Pythonを使った逐次計算
(普段のプログラミング)

計算負荷が大きな計算では
実行時間がかかる

GPUを使って
計算を高速化

Python + CUDAを使った並列計算
(本講座で取り扱う内容)

主な応用先*

・科学技術計算

・画像処理

・機械学習

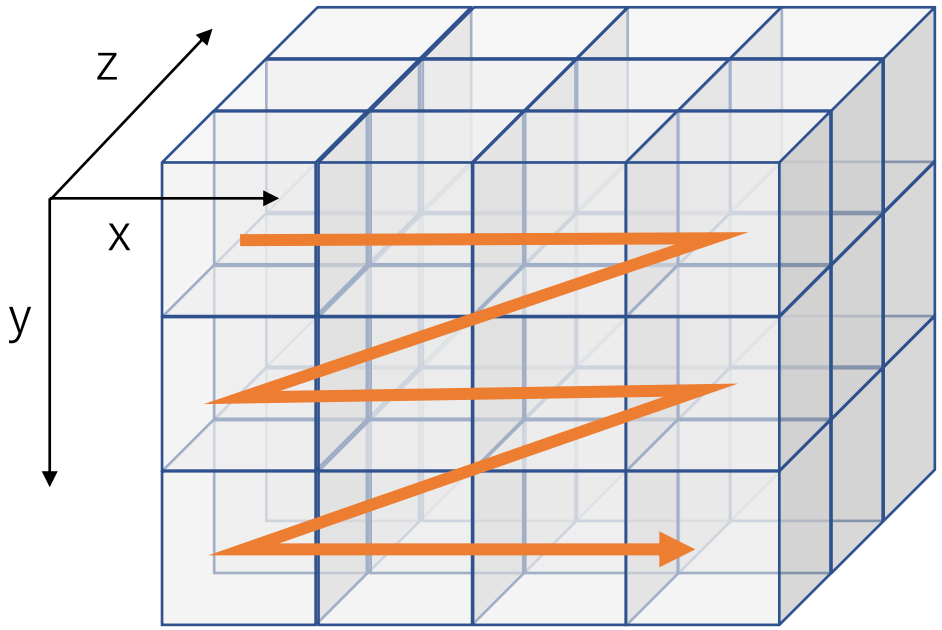
***(注意点)**
アルゴリズムを
自分で書く人向け

逐次計算とは

1の演算装置(コア*)で順番に計算する事

*四則演算など計算を行う部分の事

逐次計算のイメージ



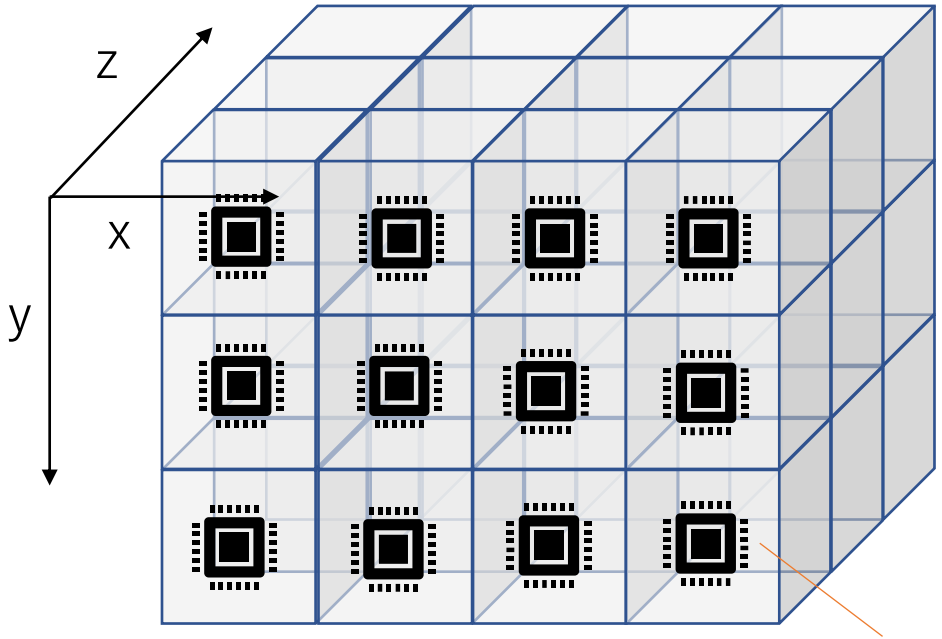
逐次計算のプログラムの例

```
for i in range(num_iterations):
    for z in range(nz):
        for y in range(ny):
            for x in range(nx):
                #
                # 配列要素毎の計算内容
                #
                res[z][y][x] = arr1[z][y][x] + arr2[z][y][x]
```

並列計算とは

複数のコアで仕事を分割すること。forループの部分が並列に実行され、高速化

並列計算のイメージ



コア

1つのコアが１つの要素を担当

並列計算のプログラムの例

```
for i in range(num_iterations):
    #
    #   演算装置毎の計算内容
    #
    add_two_arr(res, arr1, arr2)
```

forループが
3つ無くなった
⇒**計算の高速化**

GPU(Graphics Processing Unit)とは

CUDAコアを大量に積んだ高速な演算装置。コストパフォーマンスに優れる

演算装置としてのＣＰＵとGPUの比較

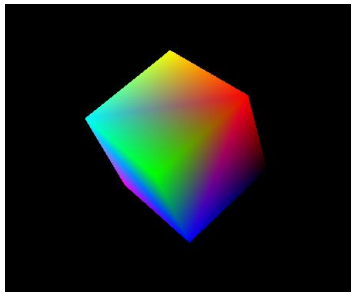
	CPU	GPU
コア数	数コア~28コア	2000 ~ 5000CUDAコア
演算性能[TFLOPS]	0.1~3	8 ~ 15
メモリ[GB]	4~128	5 ~ 48
コストパフォーマンス	△	◎
その他	if文などの分岐にも強い 複雑な処理が得意	単純な演算しかできない 複雑な処理は苦手

GeForce RTX 2080Ti

GPGPU(General Purpose GPU)と並列計算

もともとはグラフィック用途だったのを汎用的な計算に拡大(GPGPU)
GPGPUを扱うには3種類の言語(OpenCL / CUDA / OpenACC)がある

グラフィック用途



Shader言語

GPGPU
(General Purpose GPU)

- 量子化学
- AI
- 物理
- 数値流体力学
- 分子動力学法
- 金融

プログラミング言語

OpenCL マルチプラットフォーム

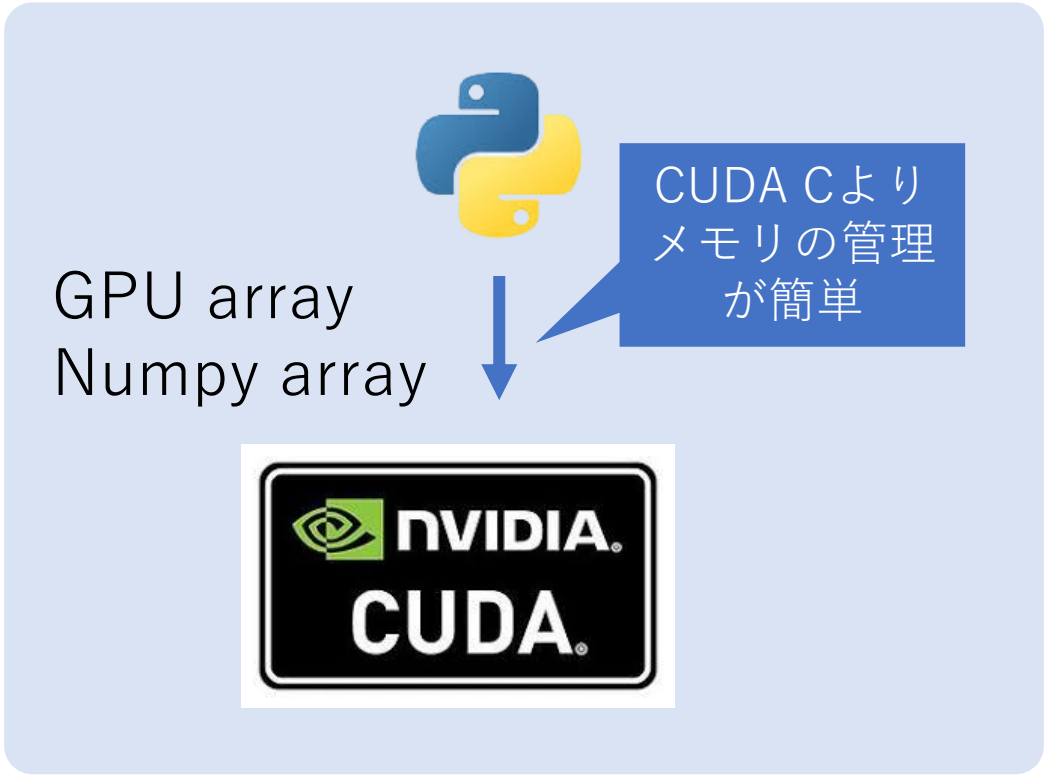
NVIDIA CUDA ポピュラー
計算が最も高速

OpenACC More Science, Less Programming 簡易にGPU計算可能

CUDA(Compute Unified Device Architecture)/PyCUDAとは

CUDAとはGPU並列を扱う為のプログラミング言語。C言語を拡張したもの
PyCUDAではCUDAをPythonから扱え、メモリ管理が楽

PyCUDA



CUDA

・NVIDIA社が開発環境を提供している
GPU用プログラミング言語

・GPU用言語として人気。最も高速化が可能

・C言語をGPU計算向けに拡張したもの(CUDA C)

PyCUDA

・CUDA CをPythonから扱う為のラッパーライブラリ

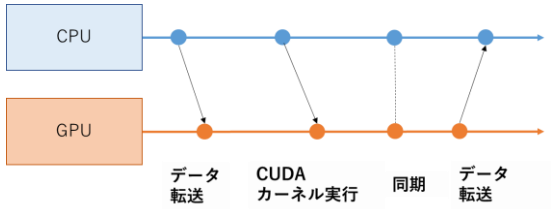
・低レベル ~ 高レベルまで広いインターフェース
柔軟にプログラミング可能

このコースはどんなコースか

GPU/CUDAの基礎知識から始めて、PyCUDAを使ったCUDAの入門講座

GPU/CUDAの基礎

製品仕様	NVIDIA® GeForce® RTX™2080 Ti
GPUアーキテクチャ	Turing
コードネーム	TU102
CUDAコア数	4352
Tensorコア数	544
単精度浮動小数点数演算性能	13.4[TFLOPS]
メモリ構成	11[GB]
放熱機構	Active
補助電源	8pin + 8pin
消費電力	250[W]
オプション対応	NVLink



得られる知識

GPUの基礎的な知識
CUDAの前提知識

PyCUDAプログラミング

colab

Google Colab を使うのでGPUが無くてもOK

```
[ ]  import os
     import sys
     import numpy as np
     import pycuda.gpuarray as gpuarray
     from pycuda.compiler import SourceModule

     # GPUの初期化
     import pycuda.autoinit

     # コンパイル時に余計なメッセージを表示させないようにする
     os.environ["CL"] = r'-Xcompiler "/wd 4819"'


     # CUDA Cカーネルファイルの参照先の絶対パスを得る
     cuda_file_path = os.path.abspath("./cuda")
     #
     #  CUDAカーネルの定義
     #
     module = SourceModule("""
     #include "kernel_functions_for_math_1d.cu"
     """, include_dirs=[cuda_file_path])

     # コンパイルしたコードからカーネルを得る
     plus_one_kernel = module.get_function("plus_one_kernel")

     # 計算対象のnumpyアレーの作成
     num_components = np.int32(10)
     x = np.arange(num_components, dtype=np.int32)

     # cpu to gpuへデータを送付
     x_gpu = gpuarray.to_gpu(x)
     y_gpu = gpuarray.zeros(num_components, dtype=np.int32)

     # ブロック、グリッドの決定
     threads_per_block = (256, 1, 1)
     blocks_per_grid = (math.ceil(num_components / threads_per_block[0]), 1, 1)

     # CUDAカーネルの実行
     plus_one_kernel(num_components, y_gpu, x_gpu, block=threads_per_block, grid=blocks_per_grid)

     # gpu to cpuへデータを送付
     y = y_gpu.get()

     print("x :", x)
     print("y :", y)

x : [0 1 2 3 4 5 6 7 8 9]
y : [ 1  2  3  4  5  6  7  8  9 10]
```

得られる知識

PyCUDAの基礎的な計算
PyCUDAの便利なライブラリの使い方
オマケ：Google Colab以外の環境

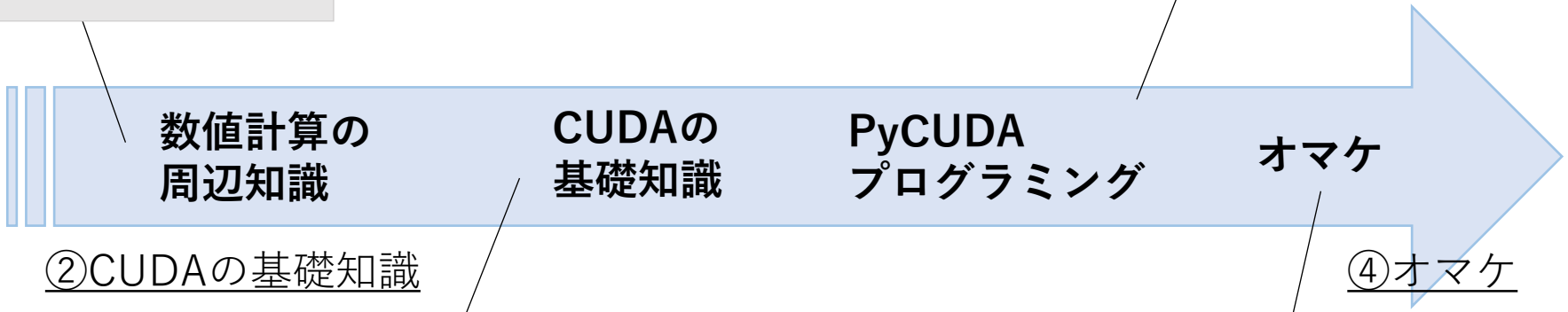
コース全体の流れ

①GPUの基礎知識

ハードウェアの知識
(マザーボード/メモリ/GPUの性能指標)

③PyCUDAプログラミング

GPU上での多次元配列の四則演算
GPUの各種メモリの使い方
GPUでのライブラリの使い方
(cuBLAS/thrust/atomic演算など)



②CUDAの基礎知識

CUDAでの計算の流れ
CUDAの専門用語
GPUのメモリ構造
C言語のミニマム知識

④オマケ

デスクトップＰＣへのGPU環境構築

それではコースでお待ちしております

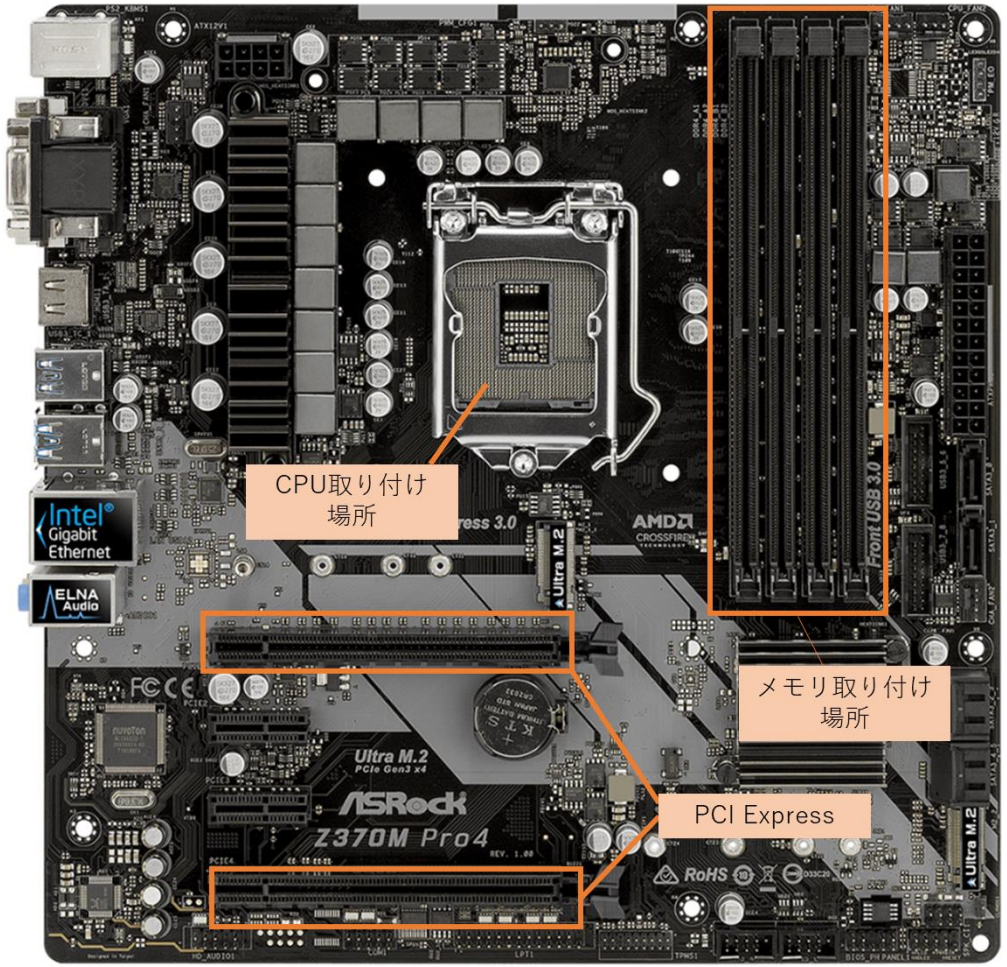
ハードウェアの基礎(GPU以外)

この動画で学ぶこと

- ✓ 数値計算で必要な知識としてマザーボードやメモリ、PCI Expressとは何かを理解する

マザーボード

CPU/GPUやメモリ、ハードディスクなどが取り付けられ、制御する基盤



ASRock Z370M PRO4

マザーボードについて知っておくべきこと

・CPU/GPUなどあらゆる機器をコントロール

・メモリを取り付ける場所がある

・PCI Expressにはグラフィックカードやサウンドカードなどを取り付け、機能を拡張できる

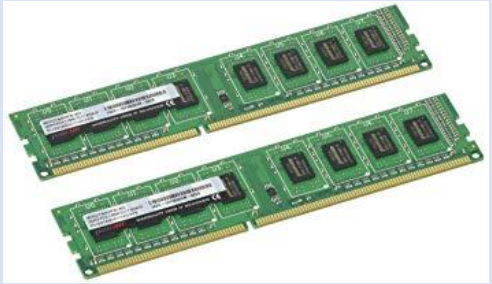
ポイント

CPUとGPUはマザーボードのPCI Expressを介して、メモリをやり取りしている

メモリ

プログラムの変数を格納しておく場所。CPUとGPUでメモリが独立している
メモリ以上に配列を確保できないので、メモリ容量は注意が必要

CPUメモリ(数 ~ 200[GB])



例: 16[GB] x 4基
＝ 64[GB]

GPUメモリ(数 ~ 48[GB])

例: 11 [GB] x 1基
＝ 11[GB]

100 x 100 x 100の要素を持つ
単精度(4byte)の３次元配列の場合

4 x 100 x 100 x 100[Byte]
⇒必要メモリ数： 約4[MB]

ポイント

CPUとGPUはメモリが
別々に分かれている
(GPU並列計算で重要なのはGPUメモリ)

ハードウェアの基礎(GPU)

この動画で学ぶこと

✓ GPUの性能について、代表的な指標を理解する。

GPUの性能指標

GPUには様々な指標がある。重要な指標としては、
CUDAコア数、FP32、メモリ構成、消費電力

製品仕様	NVIDIA® GeForce® RTX™2080 Ti
GPUアーキテクチャ	Turing
コードネーム	TU102
CUDAコア数	4352
Tensorコア数	544
単精度浮動小数点数演算性能 [FP32]	13.4[TFLOPS]
メモリ構成	11[GB]
放熱機構	Active
補助電源	8pin + 8pin
消費電力	250[W]
オプション対応	NVLink

CUDAコア

CUDAコアとはGPUで演算を行うコア。
CPUより演算能力が低いが、CPUと比べ、コア数が多い

製品仕様	NVIDIA® GeForce® RTX™2080 Ti
GPUアーキテクチャ	Turing
コードネーム	TU102
CUDAコア数	4352
Tensorコア数	544
単精度浮動小数点数演算性能[FP32]	13.4[TFLOPS]
メモリ構成	11[GB]
放熱機構	Active
補助電源	8pin + 8pin
消費電力	250[W]
オプション対応	NVLink

CPUとGPUのコア数

CPU: 多くは10コア前後
(*高額なCPUを除く)

GPU: **2000 ~ 4000コア**

ポイント

CUDAコア数が並列度の上限を決める
⇒計算能力に直結(次のFP32と関係)

FP32

単精度浮動小数点数演算性能(FLOPS)とは”1秒間に行える単精度の計算回数”

製品仕様	NVIDIA® GeForce® RTX™2080 Ti
GPUアーキテクチャ	Turing
コードネーム	TU102
CUDAコア数	4352
Tensorコア数	544
単精度浮動小数点数演算性能[FP32]	13.4[TFLOPS]
メモリ構成	11[GB]
放熱機構	Active
補助電源	8pin + 8pin
消費電力	250[W]
オプション対応	NVLink

FLOPS*の計算方法

*FLoating point number Operations Per Second

FLOPS=
FLOPS/Clock数 x Clock数 x コア数

1[TFLOPS]:1秒間に1兆回計算可

T[テラ]: 10^{12}

ポイント

一般的なCPU(8コア):0.3[TFLOPS]
GPU: 8 ~ 15[TFLOPS]なので
GPUがCPUより数十倍高速

メモリ

GPUのメモリは5 ~ 11[GB]程度のものが多くCPUと比べ、あまり大きくない
メモリが計算限界を決めるので搭載メモリには注意

製品仕様	NVIDIA® GeForce® RTX™2080 Ti
GPUアーキテクチャ	Turing
コードネーム	TU102
CUDAコア数	4352
Tensorコア数	544
単精度浮動小数点数演算性能 [FP32]	13.4[TFLOPS]
メモリ構成	11[GB]
放熱機構	Active
補助電源	8pin + 8pin
消費電力	250[W]
オプション対応	NVLink

実際にありうるケース

3 次元配列で要素数が
3 x 3のテンソル量qの配列
(x, y, z) = 300 x 300 x 300の場合、
q[9][300][300][300]
⇒**必要なメモリは約0.9[GB]**

ポイント

3次元以上の多次元配列では、
メモリ消費量大
どれだけ使っているか概算する

消費電力・補助電源

GPUは200[W]前後のものが多く、比較的電力が大きい
GPUは必要な電力を供給するために補助電源が必要なものが多い

製品仕様	NVIDIA® GeForce® RTX™2080 Ti
GPUアーキテクチャ	Turing
コードネーム	TU102
CUDAコア数	4352
Tensorコア数	544
単精度浮動小数点数演算性能[FP32]	13.4[TFLOPS]
メモリ構成	11[GB]
放熱機構	Active
補助電源	8pin + 8pin
消費電力	250[W]
オプション対応	NVLink

補助電源の挿込み箇所

6-pin

ポイント

GPUを複数取り付ける場合は
消費電力の合計と
ピンの数が足りるかチェックが必要

Tesla / GeForce / Quadro

Teslaシリーズが最も科学技術計算向けだが、かなり高価
GeForceシリーズは計算速度と価格のバランスがよい
Quadroシリーズは過去のGPUと現在のGPUで性能が大きく異なる

*( )は最近のモデル

	Tesla	GeForce	Qudro*
計算速度	○	○	×(○)
メモリ	○	△	△(◎)
コスト パフォーマンス	×	○	×(×)
特徴	科学技術計算用 倍精度などの計算向き	ゲーミング用 汎用用途 基本的に単精度で計算	グラフィック向け

Tesla K80について

AWSやGoogle Colabなど多くのクラウドサーバー上で利用されている
世代としては古いが、メモリが多い(24GB)。放熱機構がpassiveで放熱に難

製品仕様	NVIDIA® Tesla K80
GPUアーキテクチャ	Kepler
コードネーム	-
CUDAコア数	4992
Tensorコア数	-
単精度浮動小数点数演算性能 [FP32]	5.6[TFLOPS]
メモリ構成	24[GB]
放熱機構	Passive
補助電源	8pin
消費電力	300[W]
オプション対応	-

amazon web services

Tesla K80

colab

CUDAの基礎知識(1)

この動画で学ぶこと

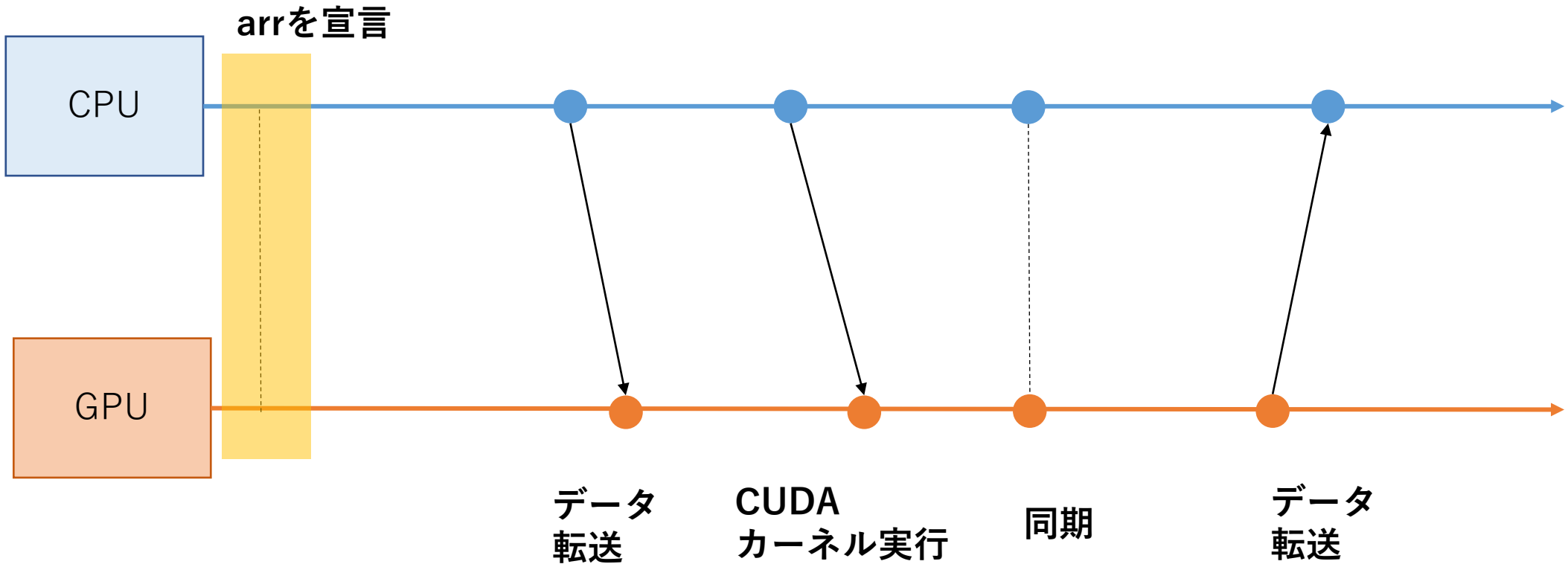
- ✓ CUDAでの計算の流れを理解する
- ✓ grid / block / threadなど基礎用語について学ぶ

*たくさんの用語が出てきますが、1度に全てを理解する必要はありません

CUDAでの計算の流れ

CPUからGPUへメモリを送信し、CUDAカーネル*を実行するのが基本の流れ

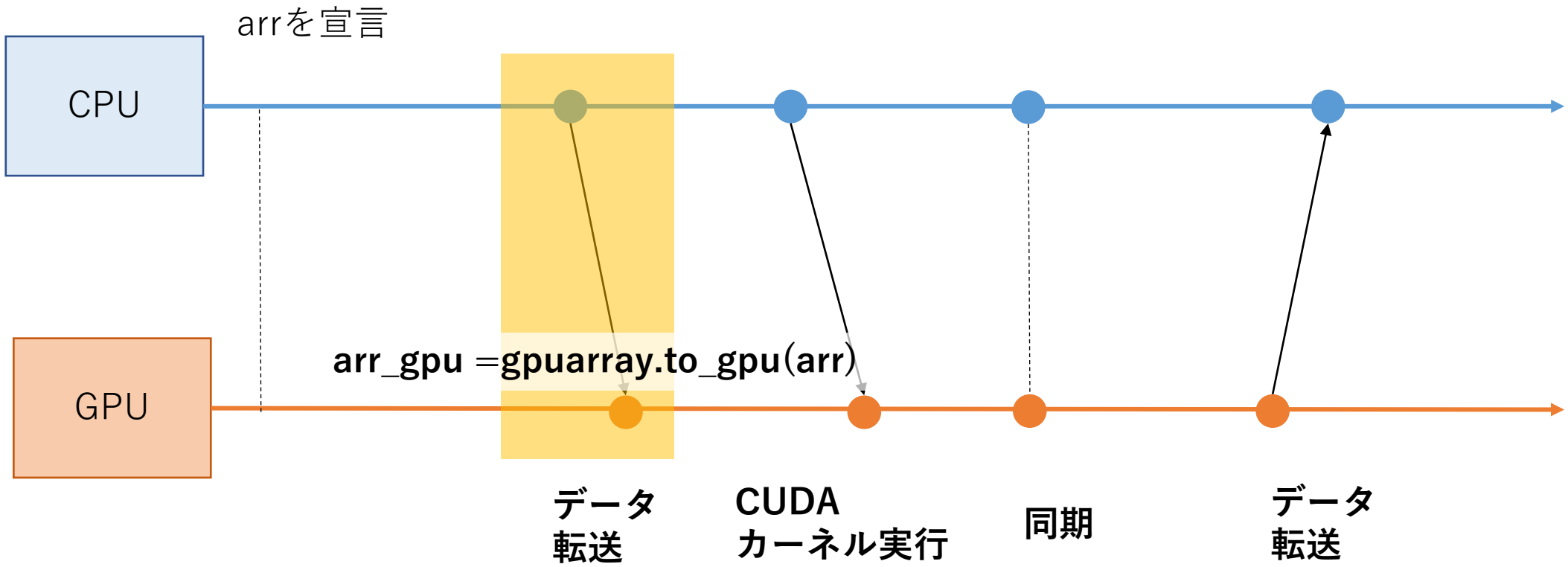
*実際の計算する関数部分



CUDAでの計算の流れ

CPUからGPUへメモリを送信し、CUDAカーネル*を実行するのが基本の流れ

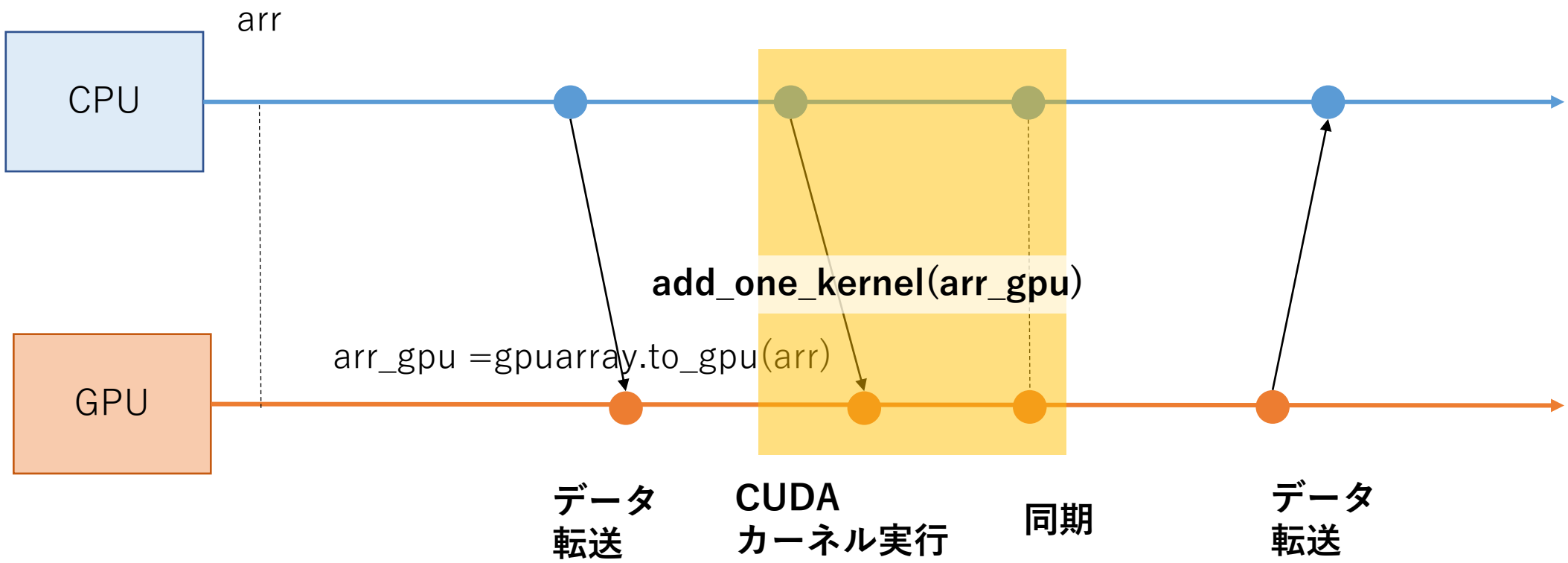
*実際の計算する関数部分



CUDAでの計算の流れ

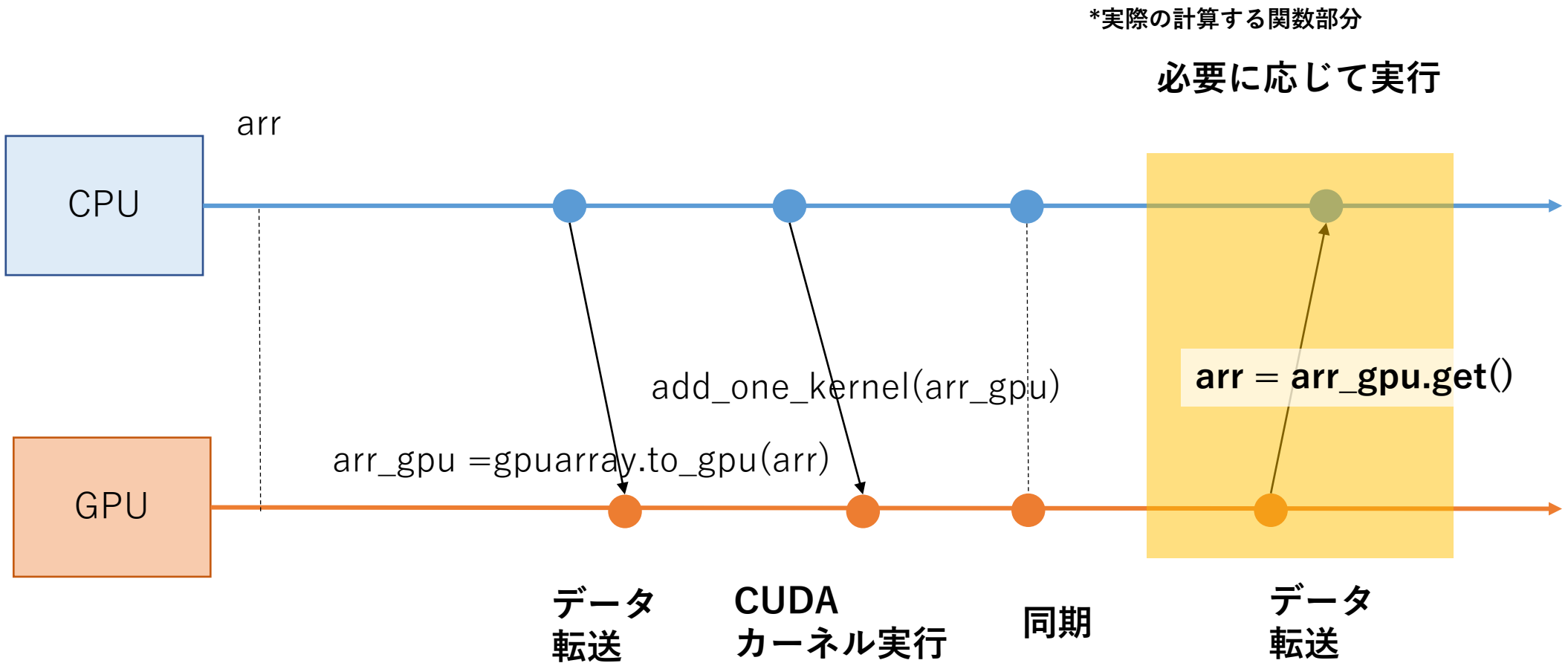
CPUからGPUへメモリを送信し、CUDAカーネル*を実行するのが基本の流れ

*実際の計算する関数部分



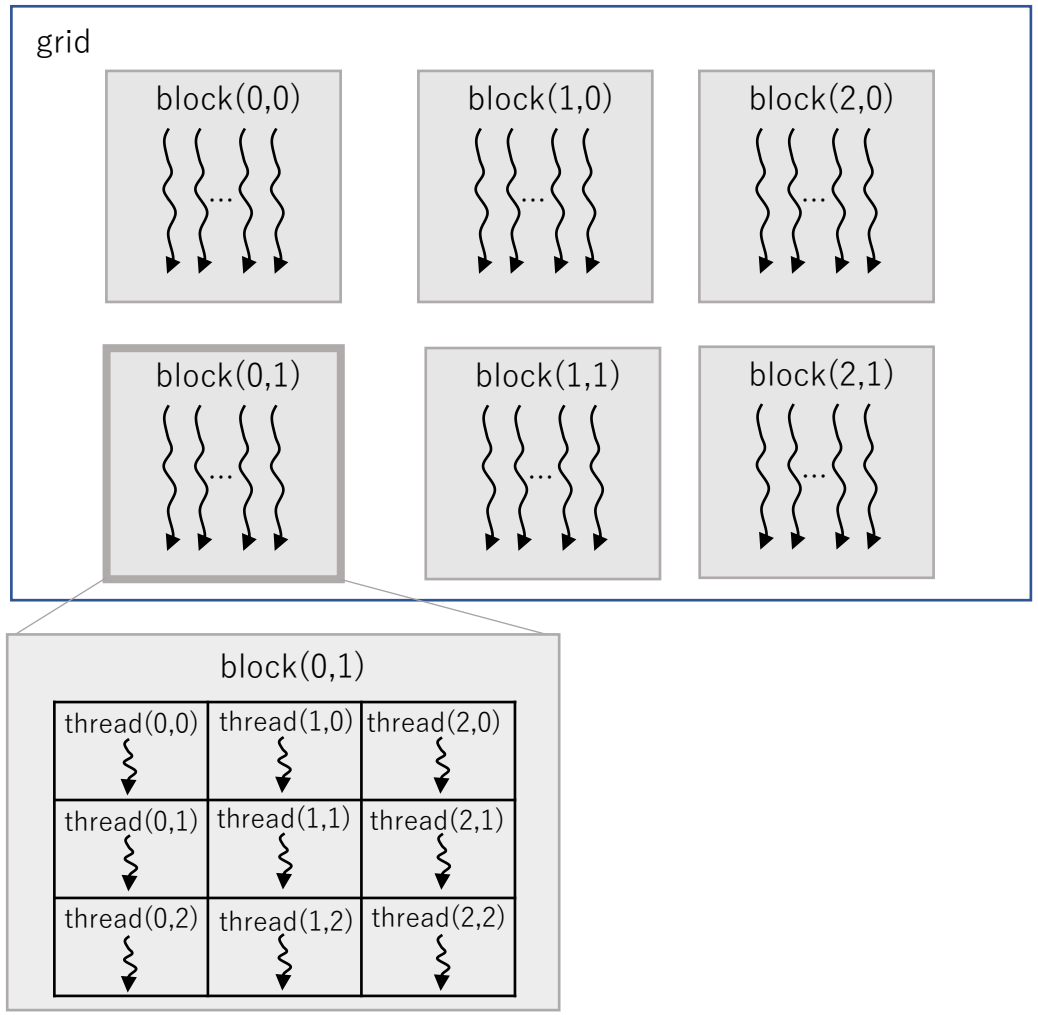
CUDAでの計算の流れ

CPUからGPUへメモリを送信し、CUDAカーネル*を実行するのが基本の流れ



grid / block / thread

CUDAではthreadと呼ばれる単位で並列化される。この他にも階層構造がある



grid

全てのブロックの集まり。GPUそのもの
イメージ的には”都道府県”

block

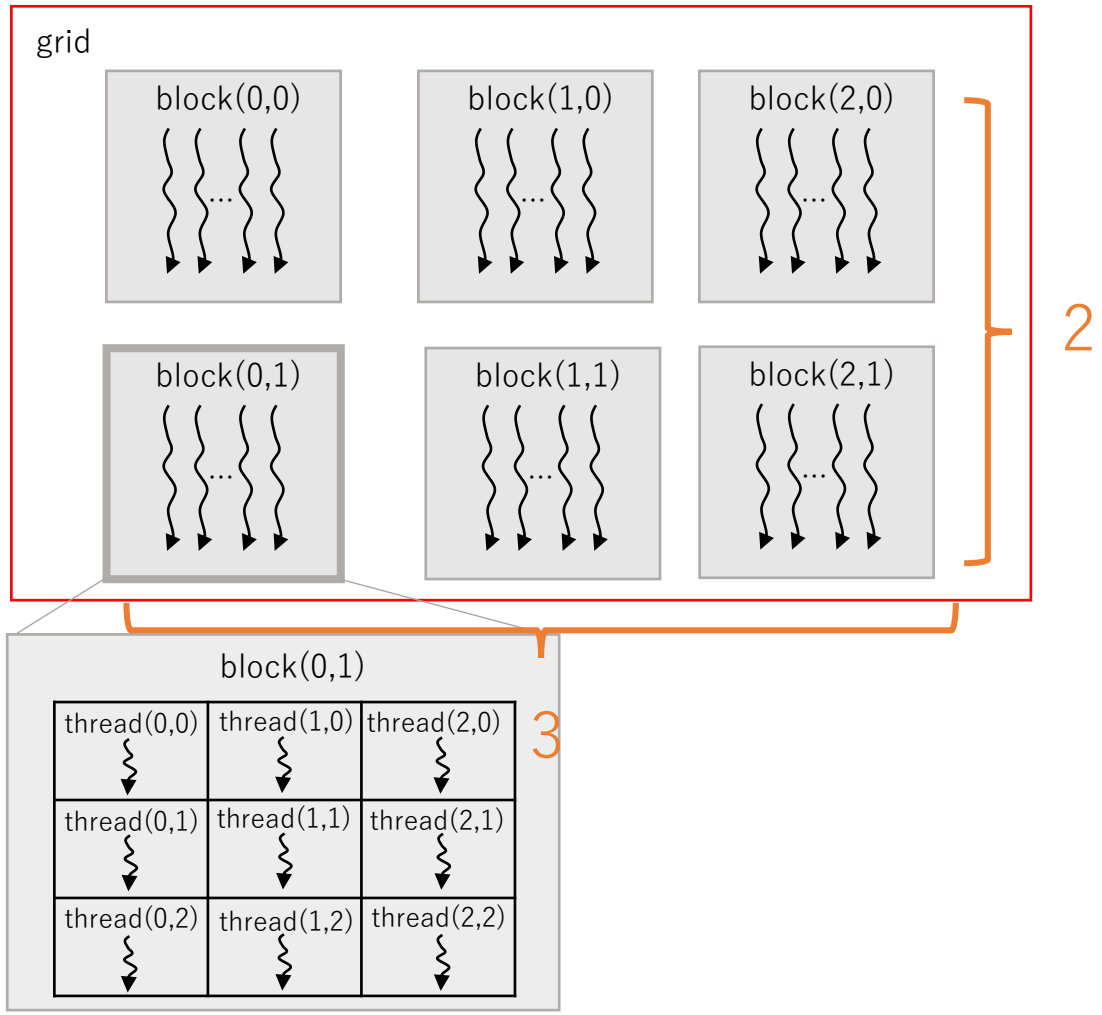
threadがいくつか集まった単位
gridの中にx, y, z座標を持っている
イメージ的には”市町村”

thread

GPU並列する際に計算を担う単位
blockの中にx, y, z座標を持っている
イメージ的には”番地”

grid

gridはブロックから構成され、gridのサイズgridDimがある



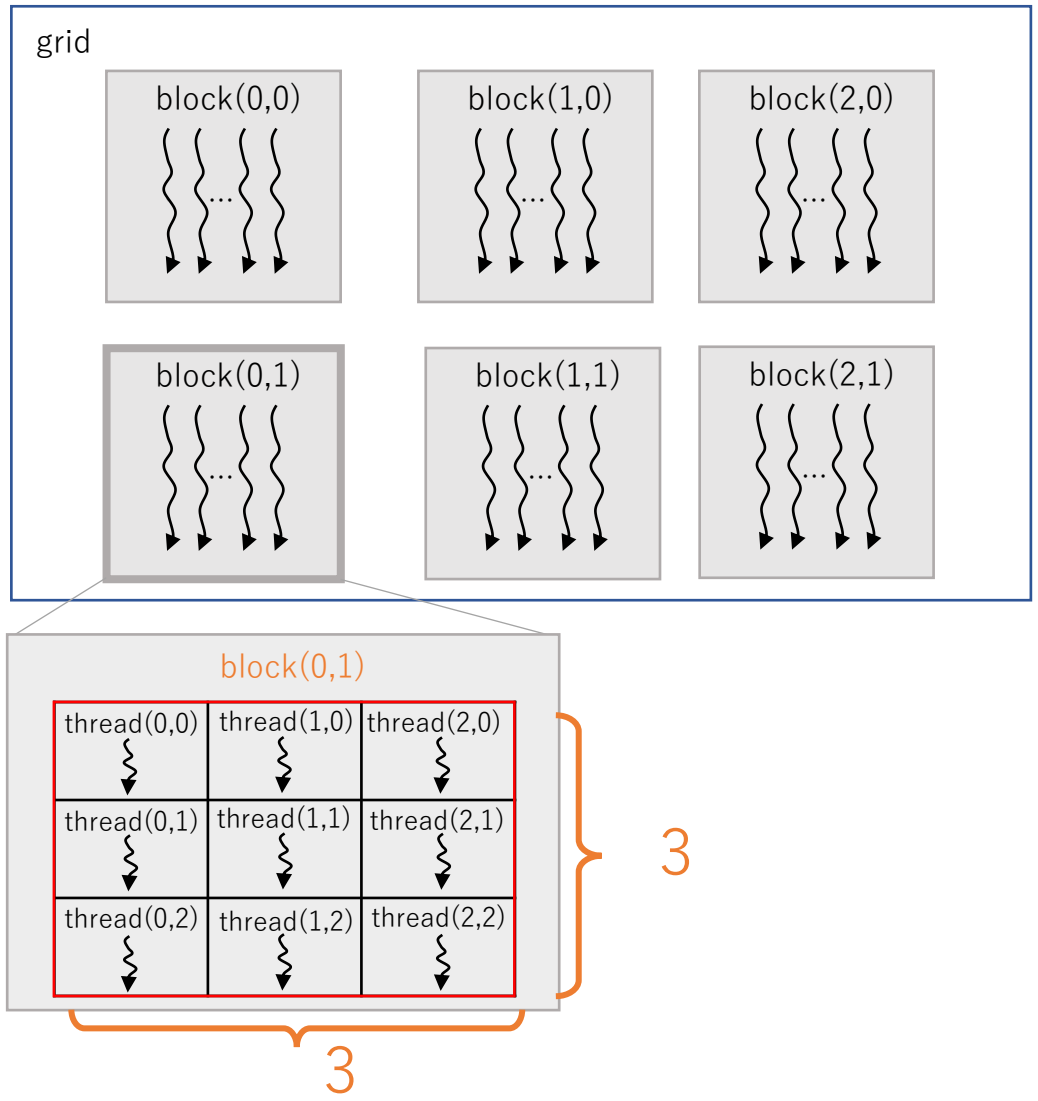
gridのサイズ

gridDim.x = 3
gridDim.y = 2
gridDim.z = 1

2次元の場合は1

block

blockにはブロックの座標idxとblockのサイズblockDimがある



blockの座標

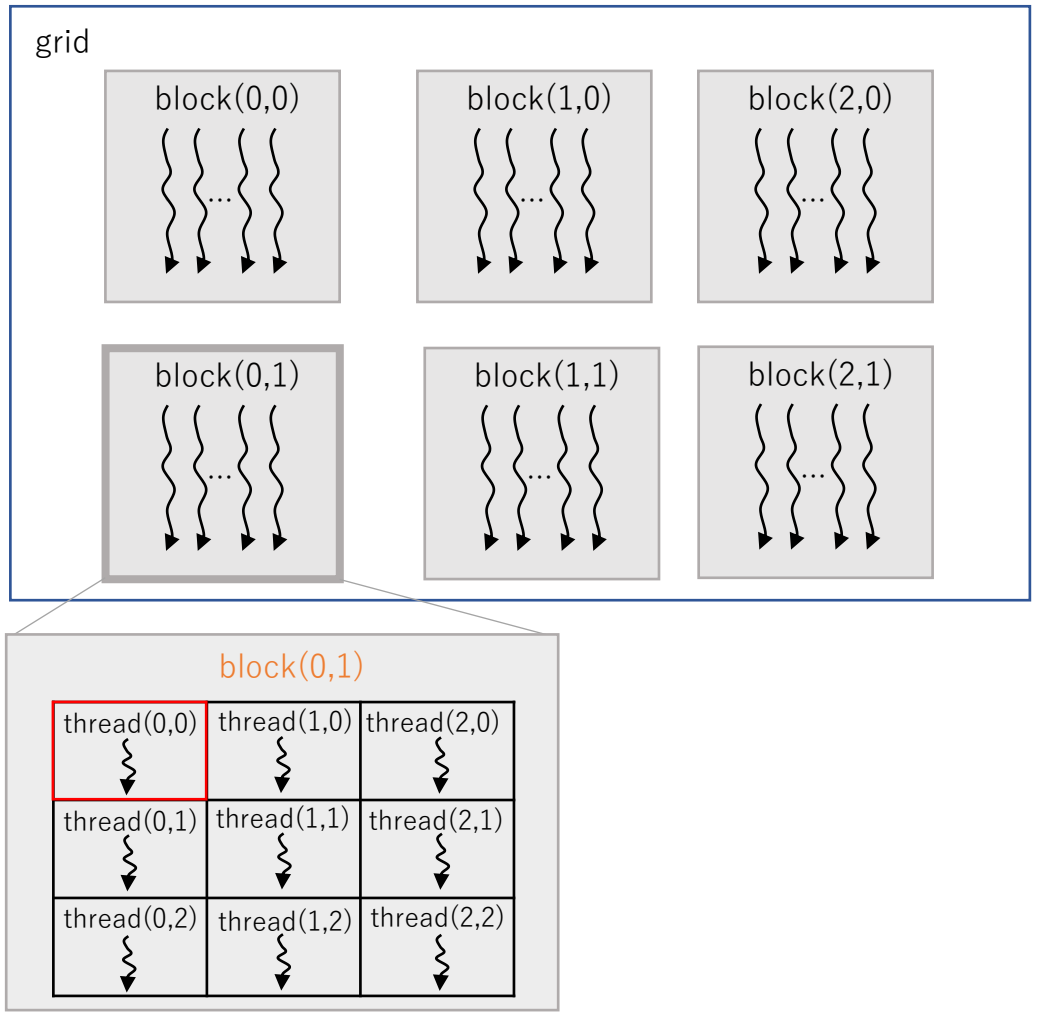
blockIdx.x = 0
blockIdx.y = 1
blockIdx.z = 0

blockのサイズ

blockDim.x = 3
blockDim.y = 3
blockDim.z = 1

thread

threadにはブロックごとにthreadの座標を表すidxがある



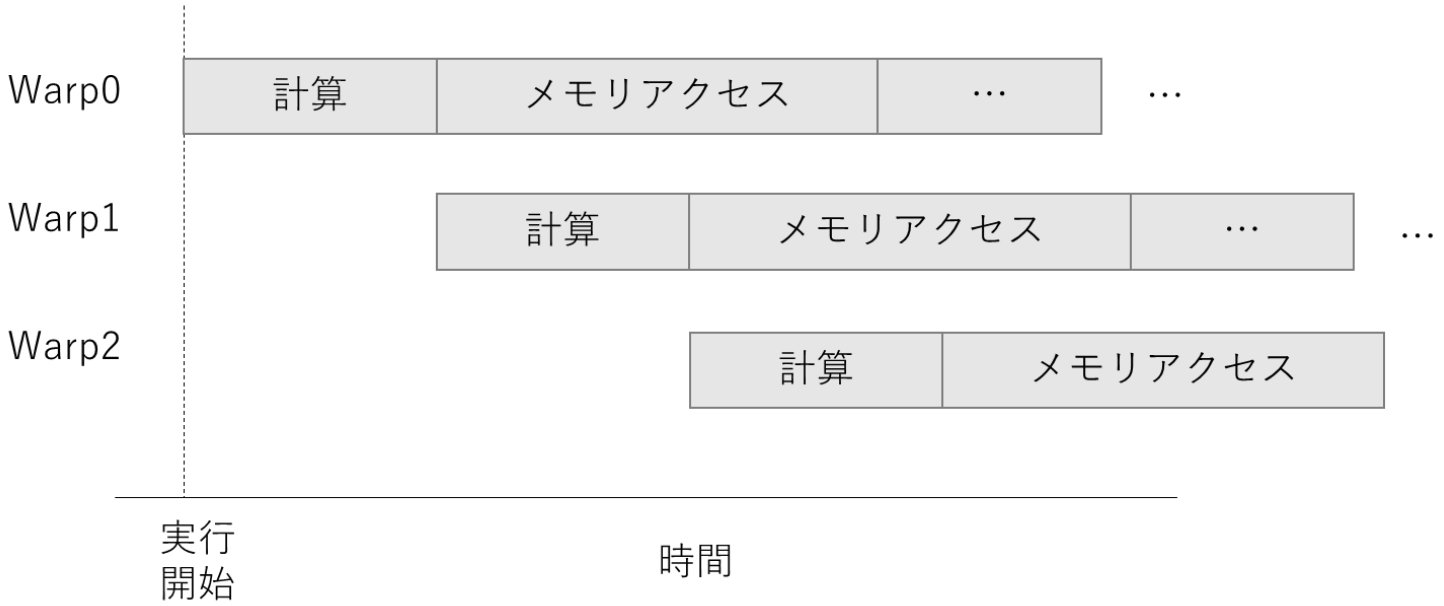
threadの座標

threadIdx.x = 0
threadIdx.y = 0
threadIdx.z = 0

Warp

threadは全てが同時に計算を始めるのではなく、Warp単位で同時に動く

CUDAカーネル実行時



Warp

32個のthreadが集まったもの

SIMT
(Single Instruction Multiple Thread)

Warp内の32個のthreadは
全て同じ命令を行う

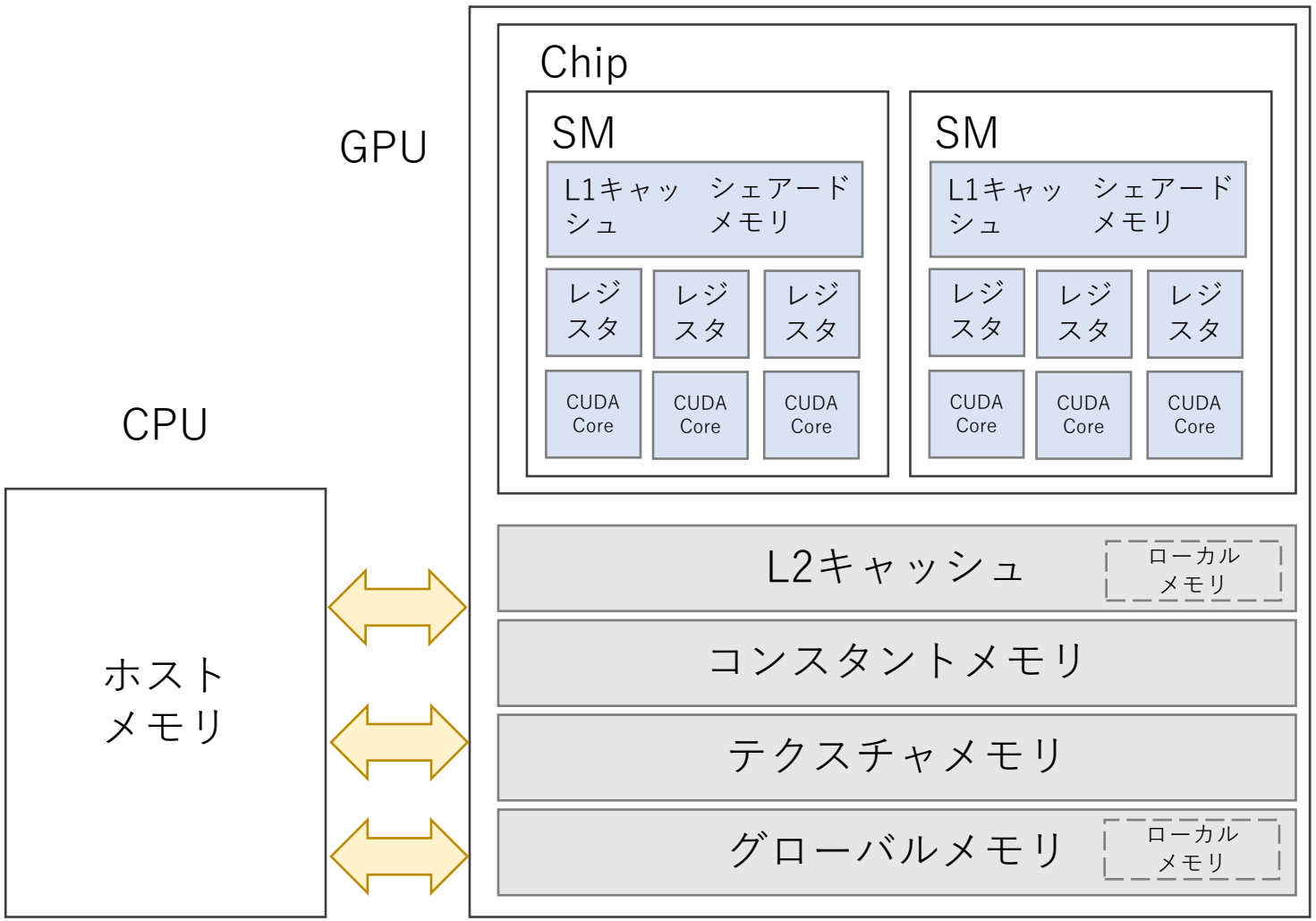
CUDAの基礎知識(2)

この動画で学ぶこと

- ✓ GPUがもつ様々なメモリ(グローバルメモリ、シェアードメモリなど)を理解する

GPUのメモリ構造

GPUは独自のメモリ構造を持つ。大まかにはオンチップとオフチップに分かれる



オンチップメモリ

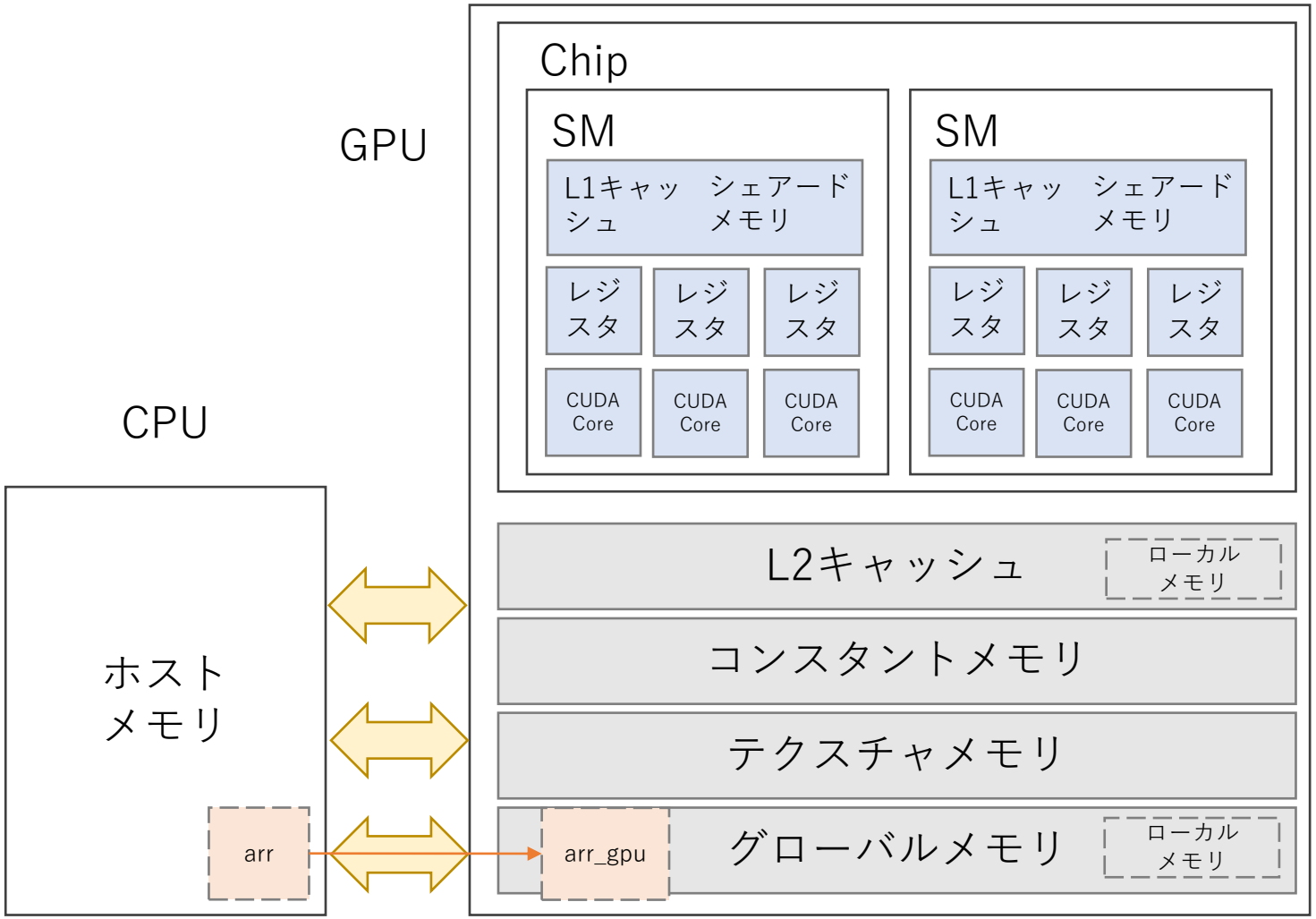
CUDA Coreに近い場所に
メモリがある
メモリ容量：小
アクセス速度：大

オフチップメモリ

CUDA Coreから遠い場所に
メモリがある
メモリ容量：大
アクセス速度：小

メモリ受け渡しの流れ-1

CPUからGPUへメモリを受け渡すと通常グローバルメモリに保存される

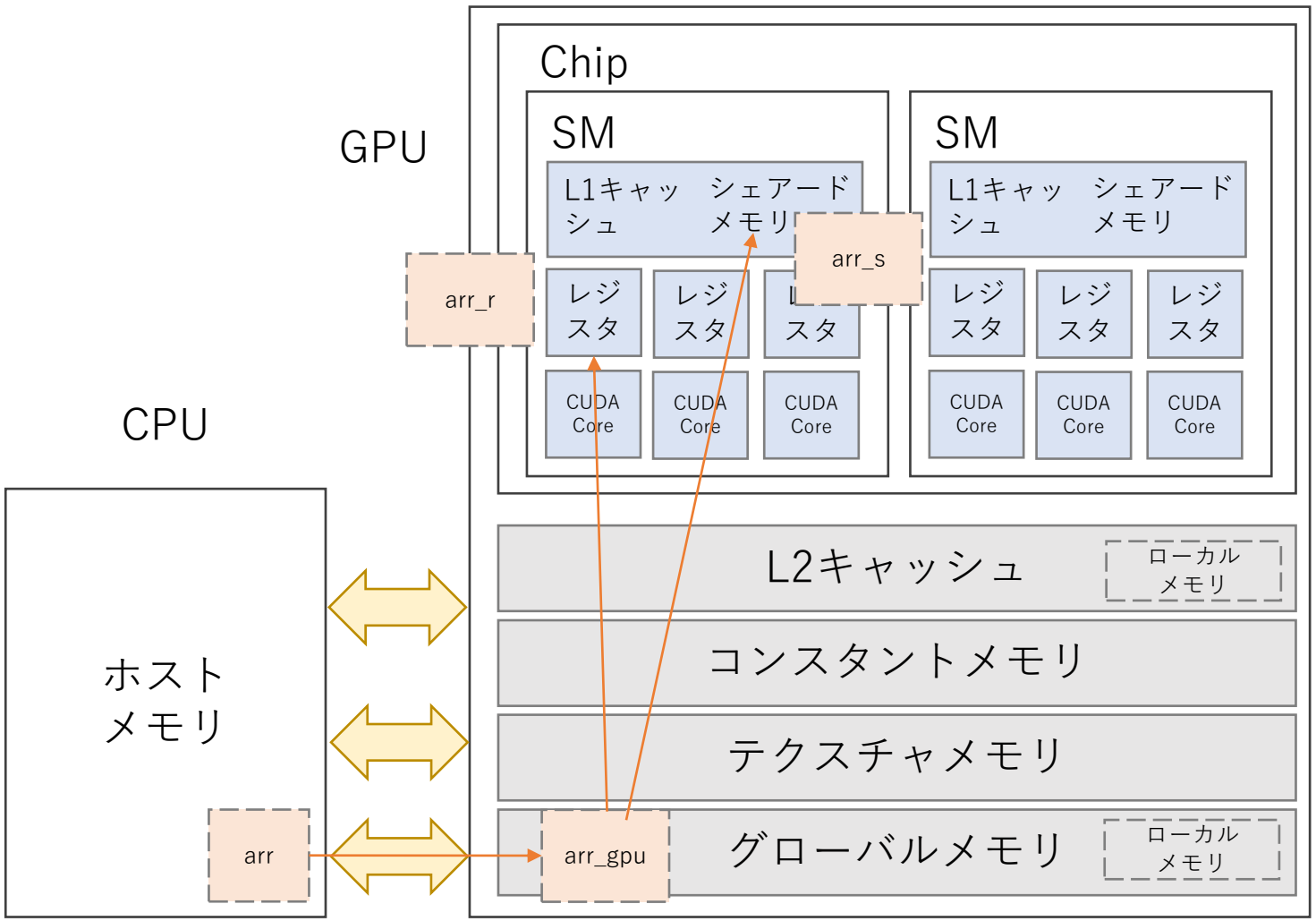


グローバルメモリ

最も一般的なメモリ
メモリ容量が大きい
アクセス速度が遅い

メモリ受け渡しの流れ-1

グローバルメモリはアクセスが遅い。レジスタやシェアードメモリに移して使う



グローバルメモリ

最も一般的なメモリ
メモリ容量が大きい
アクセス速度が遅い

レジスタ

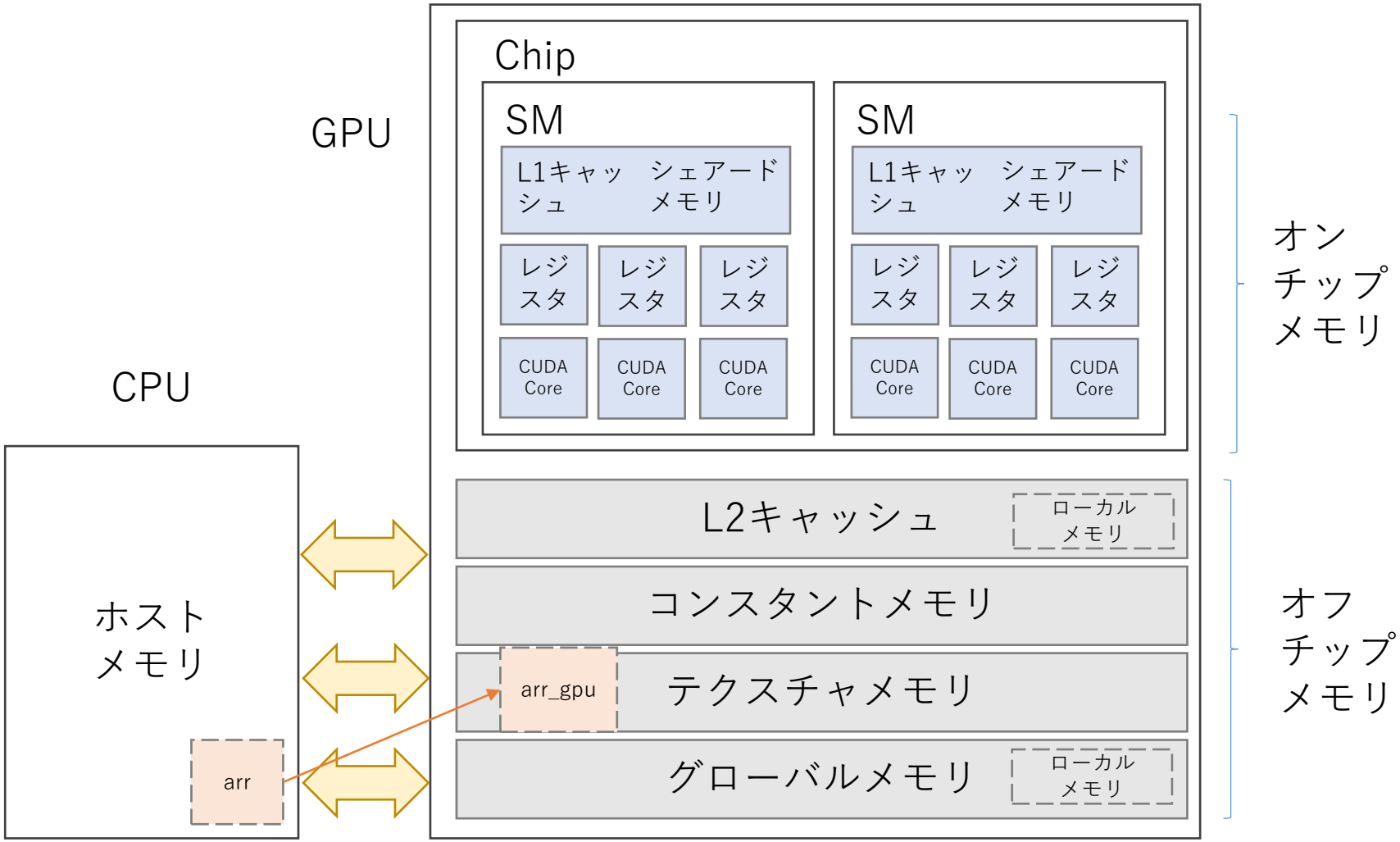
スレッド毎に異なる値を保存
メモリ容量が小さい
アクセス速度が速い

シェアードメモリ

ブロックで共有されるメモリ
メモリ容量が小さい
アクセス速度が速い

メモリ受け渡しの流れ-2

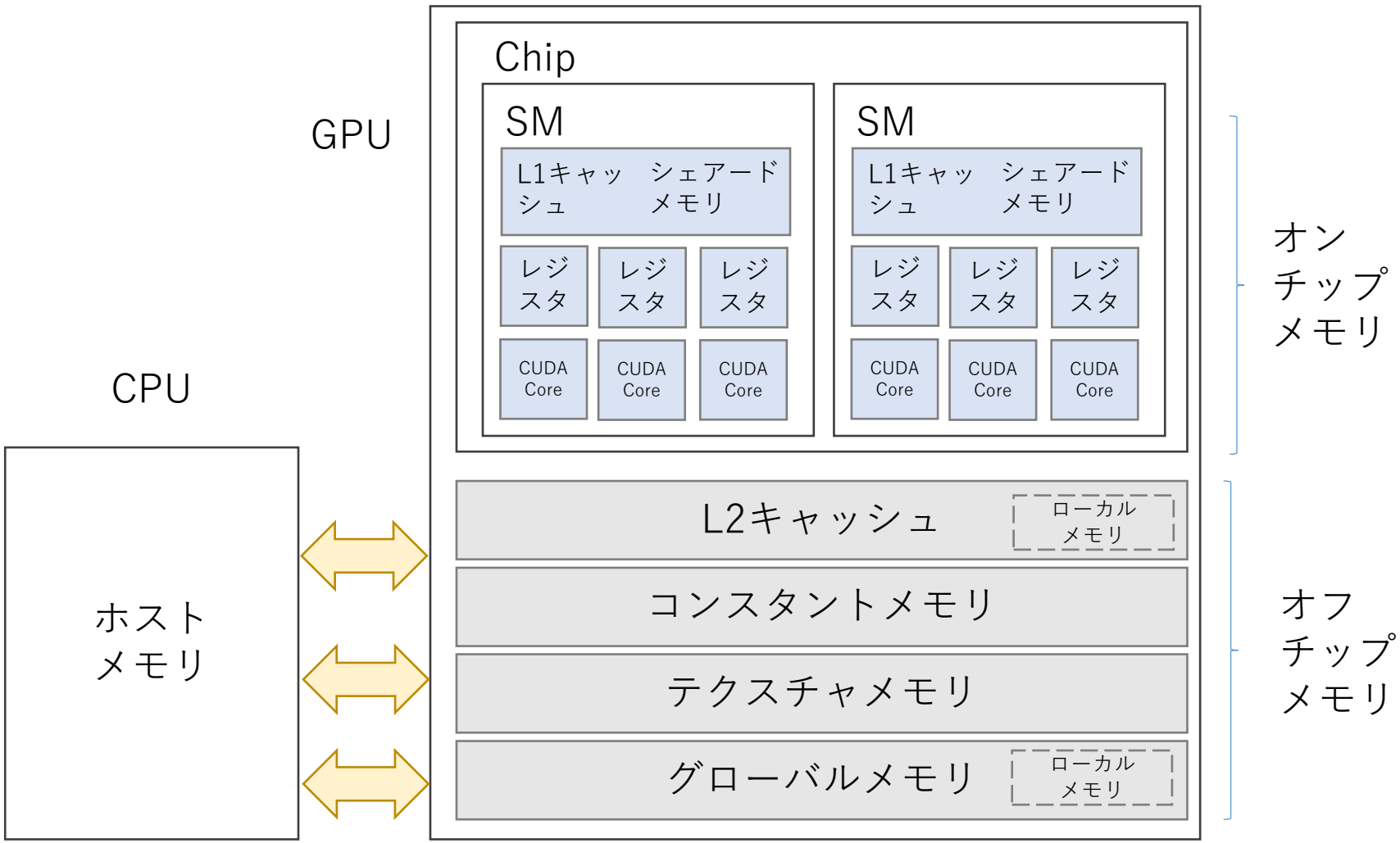
CPUからメモリをバインドすると高速・大容量なテクスチャメモリを利用できる



テクスチャメモリ

GPU上では読み取り専用
(書き換えできない)
メモリ容量が大きい
アクセス速度が大きい

GPUのメモリ構造



ポイント

メモリの種類によって
- ✓ 容量
- ✓ アクセス速度
- ✓ 制限(読取専用など)

特徴が異なるので
用途に応じて使い分ける

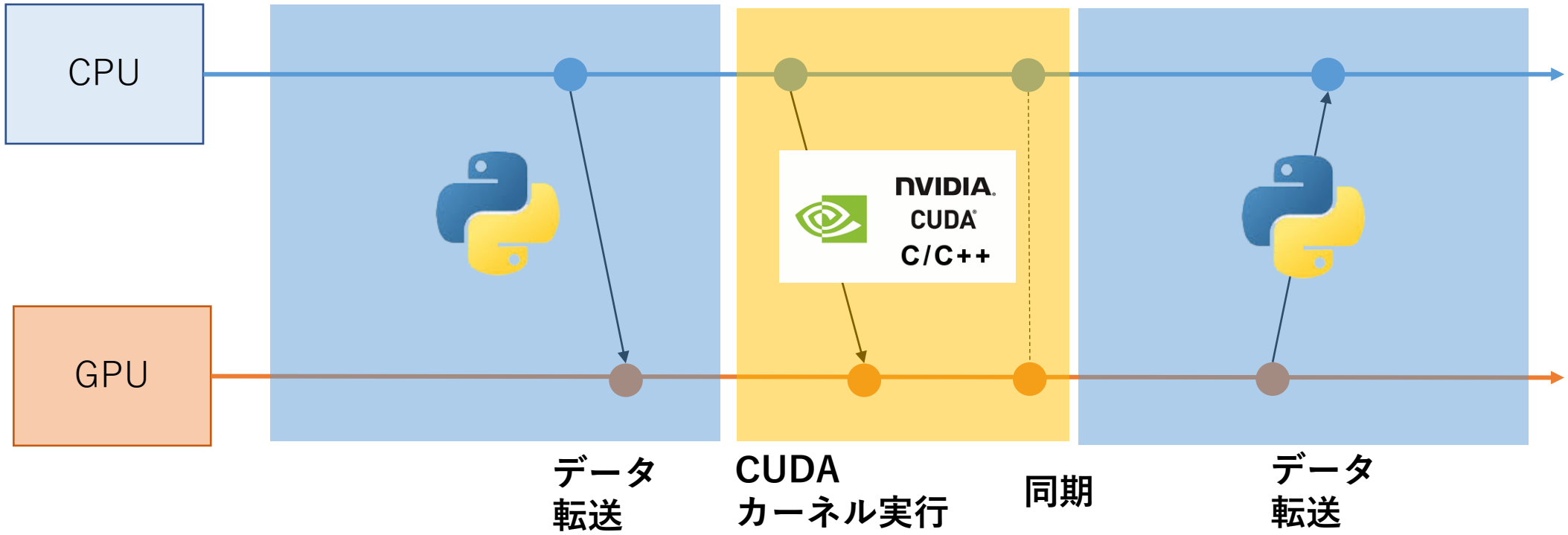
C言語ミニマム

この動画で学ぶこと

- ✓ PythonユーザーがCUDAを使う上で最低限知っておくべきC言語の文法を理解する

なぜC言語を学んでおく必要があるか?

PyCUDAではデータのインターフェースはPythonで扱い、
CUDAカーネルはCUDA Cで書く為、C言語の知識が必要



C言語とPythonの変数の型対応

CUDA CとPythonで変数の型を対応させる必要
複素数はPyCUDAのユーティリティを利用

主なCの変数の型と対応するnumpyのdtype

	CUDA C	Python
単精度整数	int	numpy.int32
倍精度整数	long long	numpy.int64
単精度小数	float	numpy.float32
倍精度小数	double	numpy.float64
単精度複素数	pycuda::complex64	numpy.complex64

実数と虚数の２つの単精度を保存するため, 64Bit

Pythonのコード

```
arr = np.array(xxx, dtype=np.float32)
arr_gpu = gpuarray.to_gpu(arr)
add_one_kernel(arr_gpu, xxx)
```

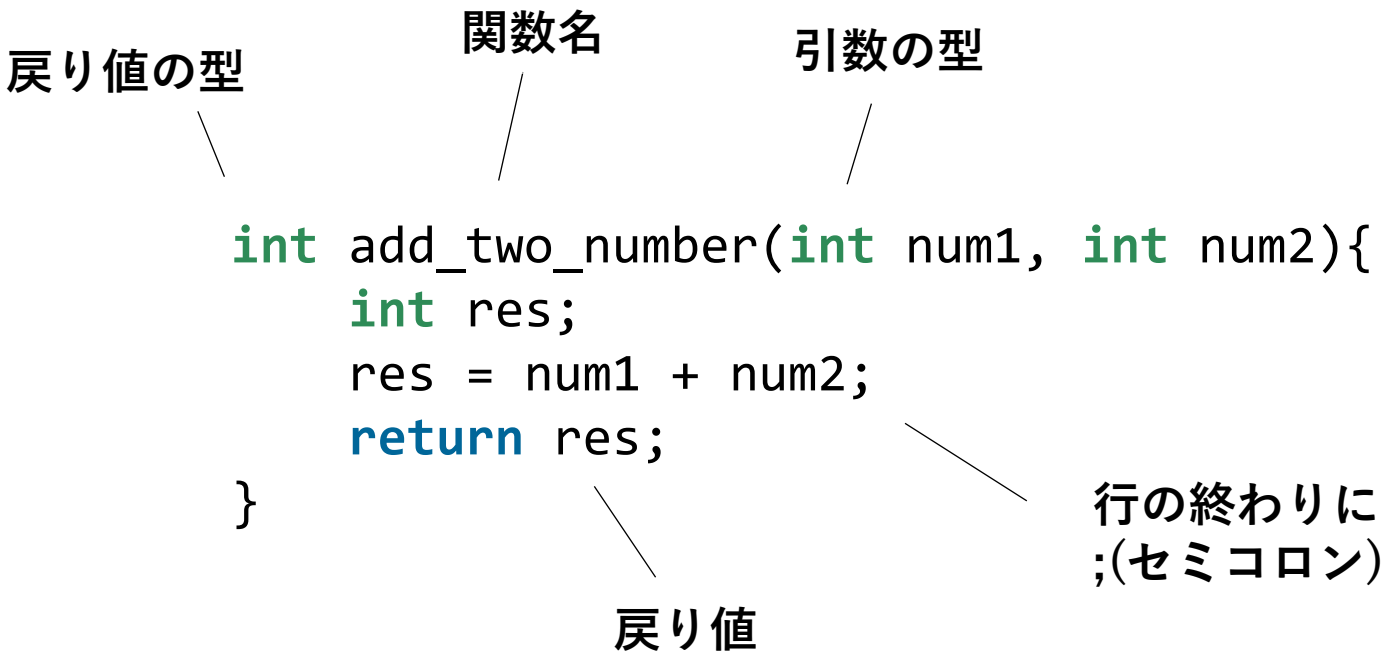
ポイント　型を対応させる

```
__global__ add_one_kernel(float *arr, xxx)
{
// 処理内容
}
```

CUDA Cのコード

関数の書き方

関数には戻り値、引数に型の指定が必要



四則演算

四則演算はPythonと同様。ただし、べき乗演算子が無い為、関数の利用が必要
ライブラリのimportには# includeを用いる

戻り値を返さない場合はvoid

ライブラリのimport

```
#include <stdio.h>
#include <math.h>

void basic_calculation(float num1, float num2) {
    // 四則演算 + - x /
    printf("a + b = %f¥n", num1 + num2);
    printf("a - b = %f¥n", num1 - num2);
    printf("a x b = %f¥n", num1 * num2);
    printf("a / b = %f¥n", num1 / num2);
    // べき乗・平方根の計算
    printf("a ** 2 = %f¥n", powf(num1, 2.0));
    printf("a ** 0.5 = %f¥n", sqrtf(num1));
}
```

べき乗はpowfやsqrtfを使う

forループ / if文

forループ, if文は条件/実行内容を( )や{ }で囲う。それ以外は概ねPythonと同じ

インクリメント演算子

```c
#include <stdio.h>

void print_odd_even(int num){
    for (int i = 0; i < num; i++){
        if (i % 2 == 0){
            printf("%d is an even number.¥n", i);
        } else if (!(i % 2 == 0)){
            printf("%d is an odd number.¥n", i);
        } else {
            printf("Something wrong...¥n");
        }
    }
}
```

マクロ

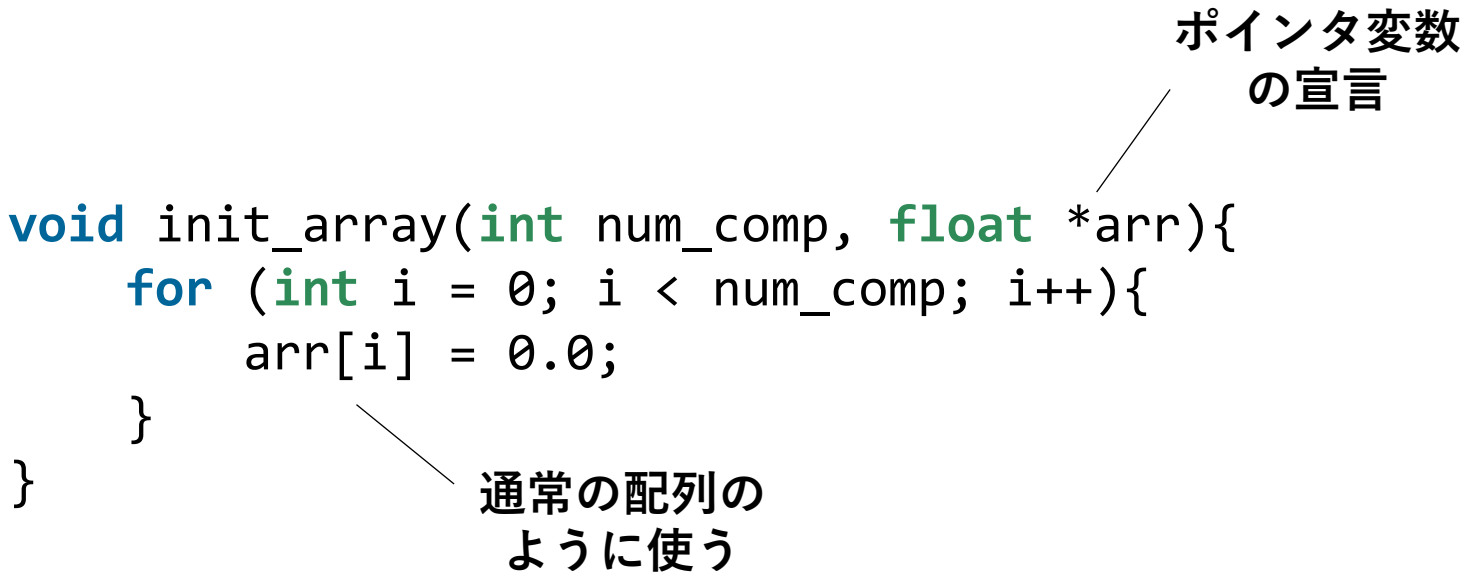
マクロを使用する事でコンパイル時に文字列に置換。簡易関数の作成も可能

マクロ
大文字を
使う(慣例)

```
# define PI 3.14159265359
# define MAX(a, b) (((a) > (b) ? (a) : (b)))
#include <stdio.h>
#include <math.h>


void circle_area(float radius){
    printf("area : %f¥n", PI * radius * radius );
}

void trim_negative(float val){
    float res = MAX(val, 0.0);
return res;
}
```

ポインタ

C言語で関数に配列を受け渡す時はポインタを使う。配列名に*を付けるだけ



Google Colabについて

この動画で学ぶこと

- ✓ 本講座で使用するプログラミング環境のGoogle Colabとは何かを理解する。

Google Colabとは

Googleの提供するPythonのインタラクティブ実行環境。
GPUとしてTesla K80が無料で使用できる。利用するにはGoogleアカウントが必要

Google Colabの画面

```
[ ]  import pycuda .driver as drv
     import pycuda.autoinit
     from pycuda.compiler import SourceModule
```

```
[ ]  import math
     import os
     import numpy as np
     import pycuda.gpuarray as gpuarray
     from pycuda.compiler import SourceModule

     # GPUの初期化
     import pycuda.autoinit

     # コンパイル時に余計なメッセージを表示させないようにする
     os.environ["CL"] = r'-Xcompiler "/wd 4819"'

     #
     #  CUDAカーネルの定義
     #

     module = SourceModule("""
         __global__ void plus_one_kernel(int num_comp, int *y, int *x){
            int i = threadIdx.x + blockDim.x * blockIdx.x;
            if (i < num_comp){
                y[i] = x[i] + 1;
             }
         }
     """)

     # コンパイルしたコードからカーネルを得る
     plus_one_kernel = module.get_function("plus_one_kernel")

     # 計算対象のnumpyアレーの作成
     num_components = np.int32(10)
     x = np.arange(num_components, dtype=np.int32)

     # cpu to gpuへデータを送付
     x_gpu = gpuarray.to_gpu(x)
     y_gpu = gpuarray.zeros(num_components, dtype=np.int32)

     # ブロック、グリッドの決定
     threads_per_block = (256, 1, 1)
     blocks_per_grid = (math.ceil(num_components / threads_per_block[0]), 1, 1)

     # CUDAカーネルの実行
     plus_one_kernel(num_components, y_gpu, x_gpu, block=threads_per_block, grid=blocks_per_grid)

     # gpu to cpuへデータを送付
     y = y_gpu.get()

     print("x :", x)
     print("y :", y)
```

colab 特徴

Jupyter notebookの拡張版のようなもの
(使い勝手はほぼ同じ)

GPU(Tesla K80)が無料で使用できる

Googleドライブ やGithubとの連携など
使い勝手が工夫されている

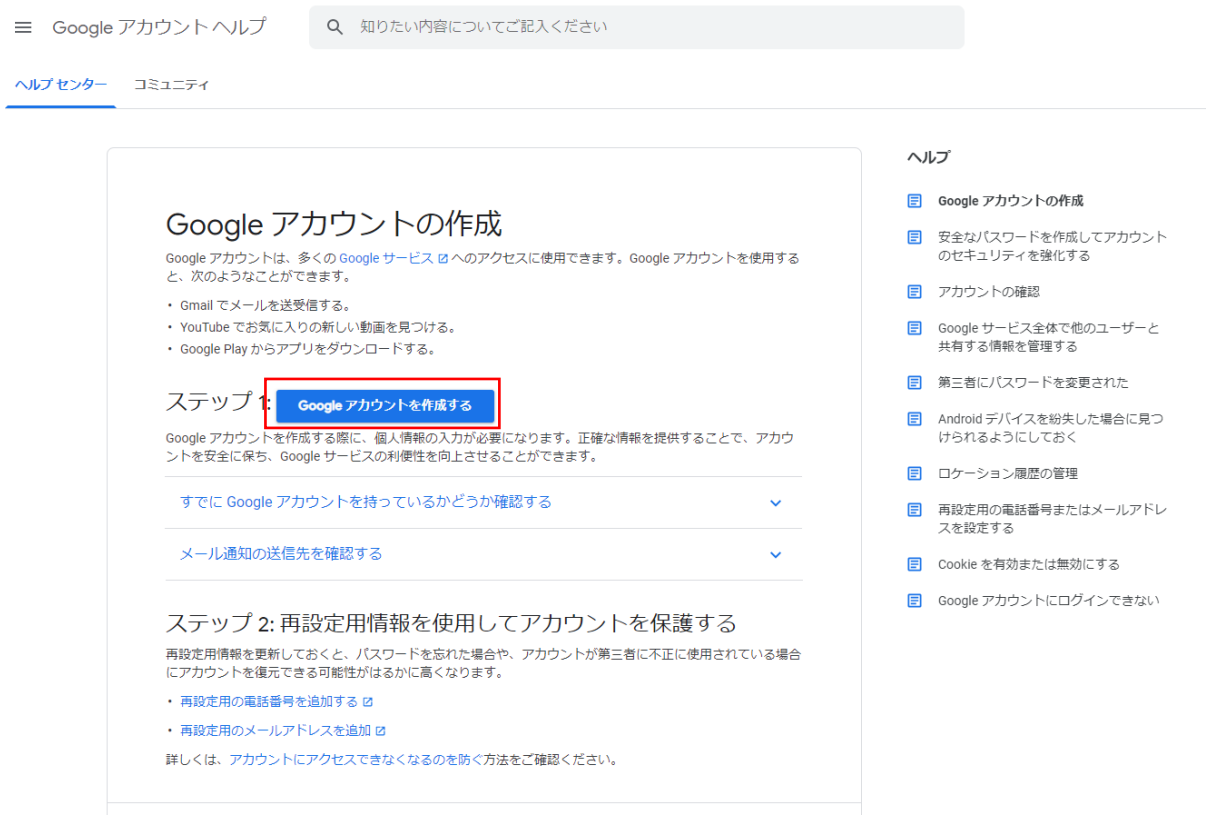
制限

90分以上放置するとセッションが切れる

12時間まで連続で計算可能

Googleアカウント作成

Google Colab利用にはGoogleアカウント作成の必要あり。手順は特に難しくない



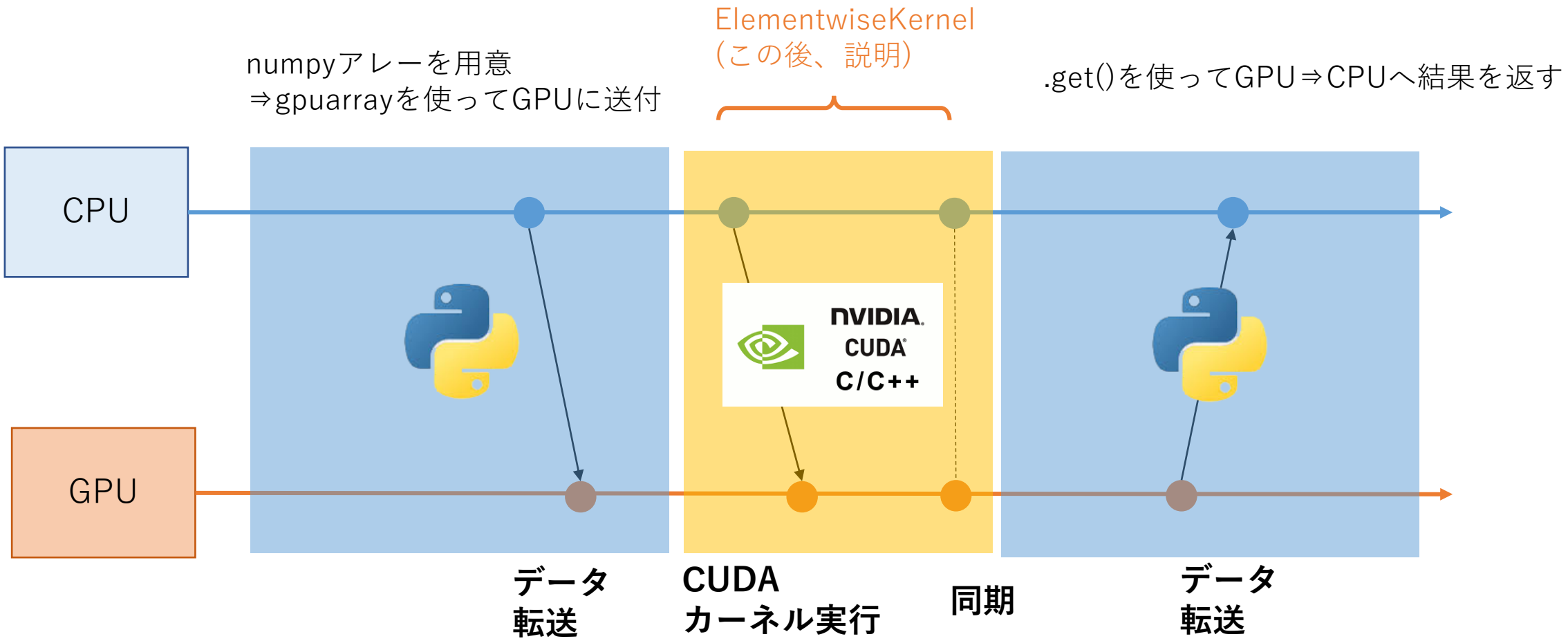
CUDAでのHello World

この動画で学ぶこと

- ✓ 最も扱いの簡単なElementwiseKernelを通して、CUDAプログラミングの流れに慣れる

CUDAでの計算の流れの復習

今回は、CUDAカーネルとしてElementwiseKernelを利用



ElementwiseKernel

CUDAカーネルを簡易に書くことが出来る。実行時にコンパイルされる

```
plus_one_kernel = ElementwiseKernel(
    "int *y, int *x",
    "y[i] = x[i] + 1",
    "plus_one")
```

引数

カーネルの実行内容

カーネル名

特徴

実行時にblockやgridの設定をする必要が無い

C言語で必要な;(セミコロン)を付ける必要が無い

“i”というインデックス用の変数が予め宣言されている

ポイント

各threadが実行する内容をカーネルに記述
(全てのカーネルに共通)

SourceModule

この動画で学ぶこと

✓ 一般的なPyCUDAの実行方法として、
SourceModuleの使い方を理解する

ElementwiseKernel

ElementwiseKernelにより、簡単にCUDAカーネルが利用できた

```
plus_one_kernel = ElementwiseKernel(
    "int *y, int *x",
    "y[i] = x[i] + 1",
    "plus_one")
```

引数

カーネルの実行内容

カーネル名

特徴

実行時にblockやgridの設定をする必要が無い

C言語で必要な;(セミコロン)を付ける必要が無い

“i”というインデックス用の変数が予め宣言されている

もっと複雑なカーネルを書く場合は、ElementwiseKernelよりSourceModuleが適している

SourceModule

SourceModuleはCUDA Cカーネルをコンパイルする関数
SourceModuleにより一般的なCUDAカーネルを実行することが出来る

CUDA Cのカーネル

```
__global__ void plus_one_kernel(int num_comp,
 int *y, int *x){
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i < num_comp){
       y[i] = x[i] + 1;
   }
}
```

```
SourceModule("CUDA Cのコード")
```

nvccによりコンパイル

**SourceModuleを利用する事で、
任意のCUDAカーネルをコンパイルして実行可能**

CUDA Cの書き方

__global__や__device__といった関数の属性を追加
各スレッドが配列の要素を担当するようにインデックスiを計算

関数の属性

```
__global__ void plus_one_kernel(int num_comp,
 int *y, int *x){
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i < num_comp){
      y[i] = x[i] + 1;
   }
}
```

インデックスの計算

if文による領域外参照のチェック

__global__ / __device__

CPUから呼び出されるのかGPUから呼び出されるのかによって属性がある

関数の属性

```
__global__ void plus_one_kernel(int num_comp,
 int *y, int *x){
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i < num_comp){
       y[i] = x[i] + 1;
   }
}
```

関数の属性

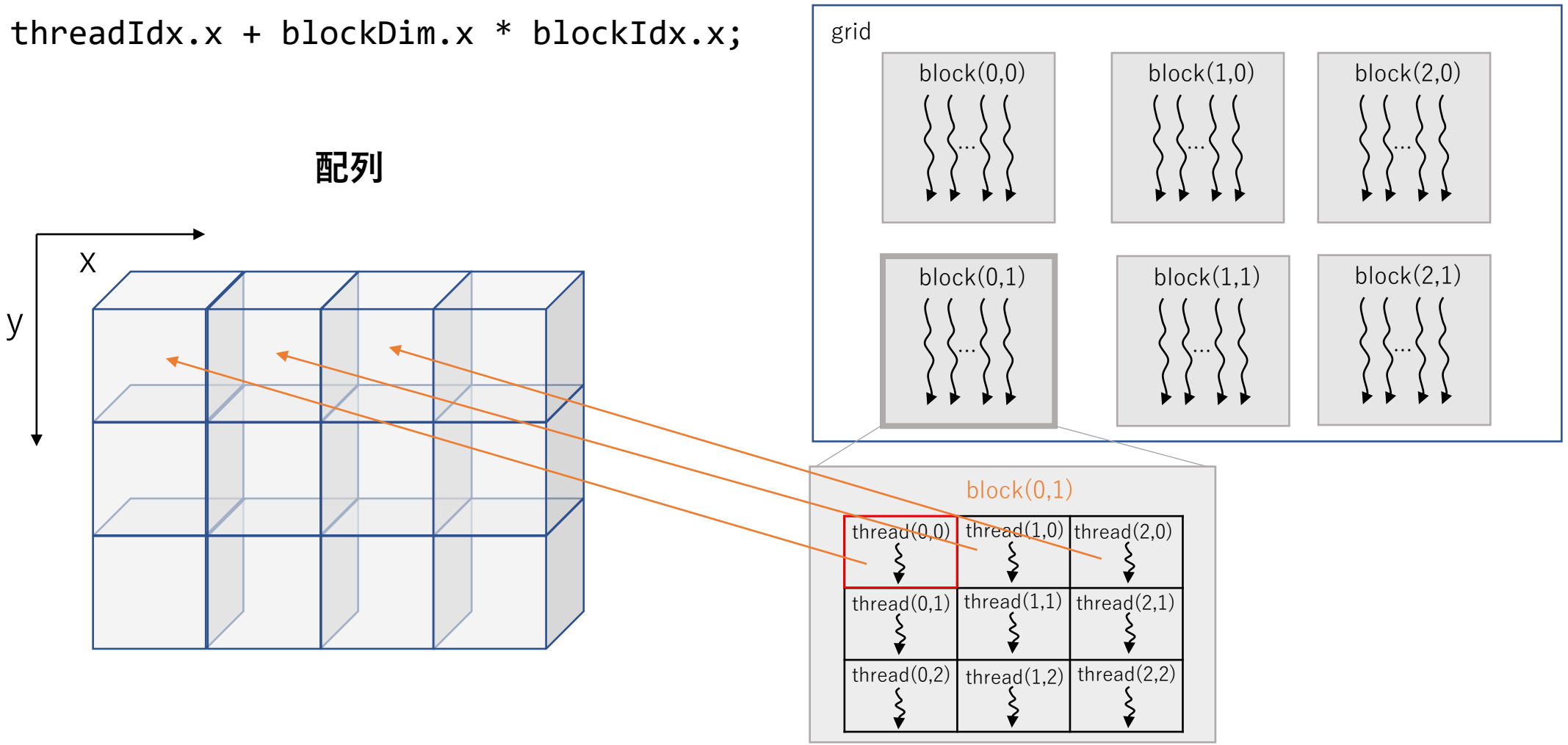
__global__ : CPU/GPUの両方から呼び出される関数。戻り値はvoidとなる

__device__ : __global__の関数など別のCUDAカーネル内から呼び出される関数。GPU上でのみ呼び出される関数

インデックスの計算

各スレッドが配列の各要素を担当するようにインデックスを計算する
慣れないうちは下記計算式をおまじないとして覚えましょう

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```

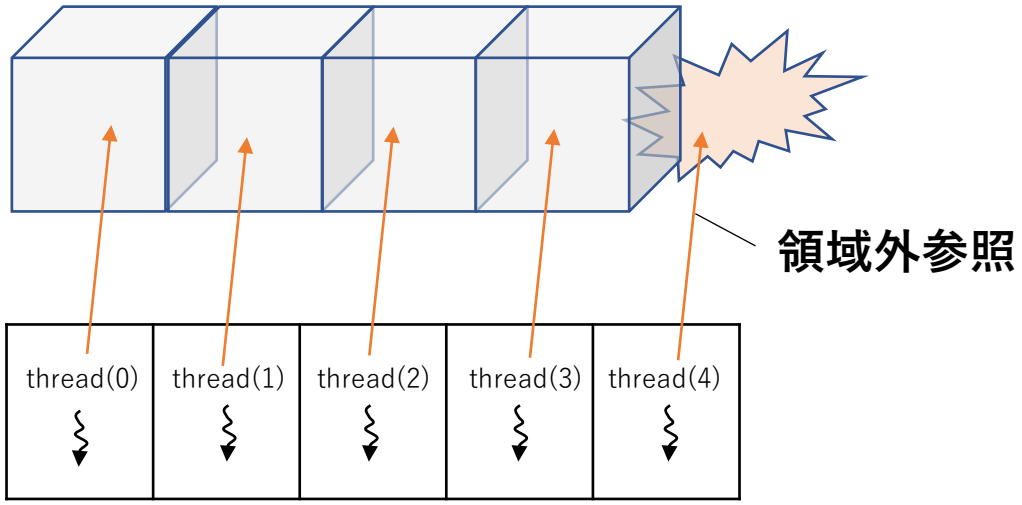


if文による領域外参照のチェック

配列要素 < ブロック当たりのスレッド数の場合、余ったスレッドが領域外参照する
これを防ぐためにif文で必ずチェックする(おまじない)

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
```

配列



領域外参照

スレッド

```
__global__ void plus_one_kernel(int num_comp,
 int *y, int *x){
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i < num_comp){
       y[i] = x[i] + 1;
   }
}
```

thread(4)の
領域外参照をはじく

threads_per_block / blocks_per_grid

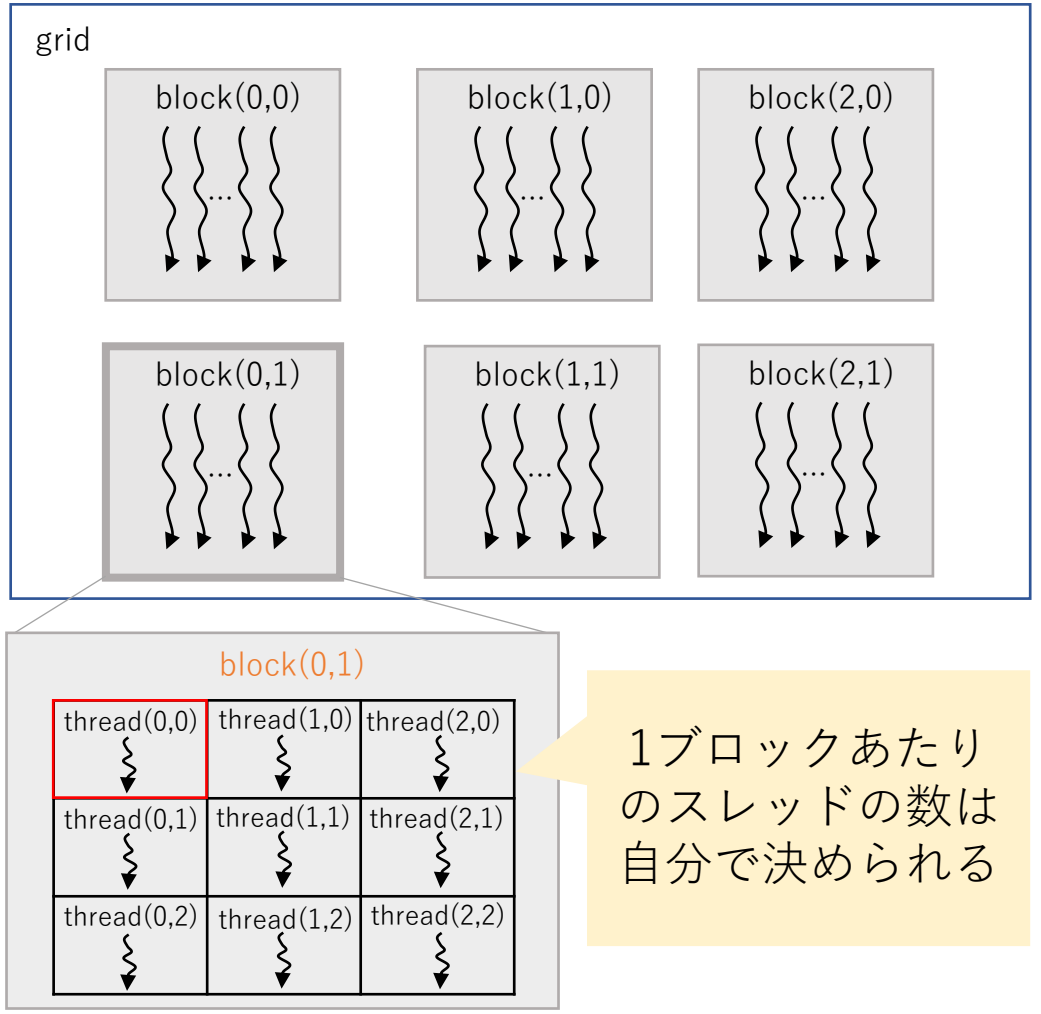
スレッド数やブロック数は自分で設定する必要がある
設定方法はスレッド数の設定⇒ブロック数の設定の順で行う

スレッド数/ブロック数の決め方

1000個の配列要素を持ったnumpy array

1ブロック当たりスレッド数: 100を指定
⇒必要なブロックの数 1000 / 100 = 10

```
# ブロック、グリッドの決定
threads_per_block = (256, 1, 1)
blocks_per_grid = (math.ceil(num_components / threa
ds_per_block[0]), 1, 1)

# CUDAカーネルの実行
plus_one_kernel(num_components, y_gpu, x_gpu, block
=threads_per_block, grid=blocks_per_grid)
```



1ブロックあたりのスレッドの数は自分で決められる

外部ファイルの取込

この動画で学ぶこと

- ✓ CUDAカーネルを別ファイルからPyCUDAに読み込む方法を理解する

SourceModule

SourceModuleはCUDA Cカーネルをコンパイルする関数
SourceModuleにより一般的なCUDAカーネルを実行することが出来る

CUDA Cのカーネル

```
__global__ void plus_one_kernel(int num_comp,
 int *y, int *x){
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i < num_comp){
       y[i] = x[i] + 1;
   }
}
```

```
SourceModule(“CUDA Cのコード”)
```

nvccによりコンパイル

**SourceModuleを利用する事で、
任意のCUDAカーネルをコンパイルして実行可能**

外部ファイルの取込

大規模なプログラム作成時はCUDA Cカーネルを外部ファイルに分離
メインプログラム＋CUDAカーネルファイルとする事で、管理が容易に

sample.cu

```
__global__ void plus_one_kernel(int num_comp,
 int *y, int *x){
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i < num_comp){
       y[i] = x[i] + 1;
   }
}
```

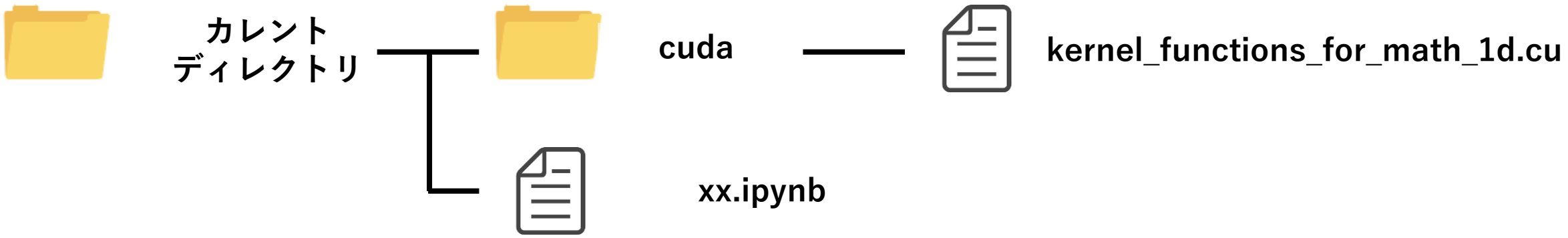
main.py

```
SourceModule(#include
“sample.cu”)
```

別々のファイルに分離する事で大規模なプログラム作成時でも機能を分けて管理できる

フォルダ構成の例

cudaディレクトリを作成し、その中にCUDA Cカーネルファイルを保存



CPU vs GPU

この動画で学ぶこと

- ✓ CPUとGPUで実行速度の比較を行う方法を理解する
- ✓ CUDAで高速化するためのポイントを理解する

CPUの実行速度測定

time関数で計算の前後を挟み、差を取ることで実行速度を測定できる

```
import time
import numpy as np

time_start_cpu = time.time()
#
# ここに計算内容を書き込む
#
time_end_cpu = time.time()

#
# 実行時間
#
print("CPU calculation {0} [msec]".format(1000 * (time_end_cpu -
 time_start_cpu)))
```

time.time():
呼ばれた瞬間の時刻を記録

GPUの実行速度測定

CPUとGPUで動作が独立している為、time.time()関数は正確ではない
PyCUDAが用意しているイベントを使って測定

```
import numpy as np
import pycuda.gpuarray as gpuarray
import pycuda.driver as drv
from pycuda.compiler import SourceModule

…(中略)
# 計測用イベント変数の用意
time_start_gpu = drv.Event()
time_end_gpu = drv.Event()

time_start_gpu.record()
#
# ここでCUDAカーネルを実行する
#
time_end_gpu.record()
# カーネルを同期する *必ず行う
time_end_gpu.synchronize()

print("kernel exec {0} [msec]".format(time_start_gpu.time_till(tim
e_end_gpu)))
```

drv.Event():
CUDAカーネルの開始や終了を記録できるイベント

.record():
イベントのメソッドで
time.time()のように働く

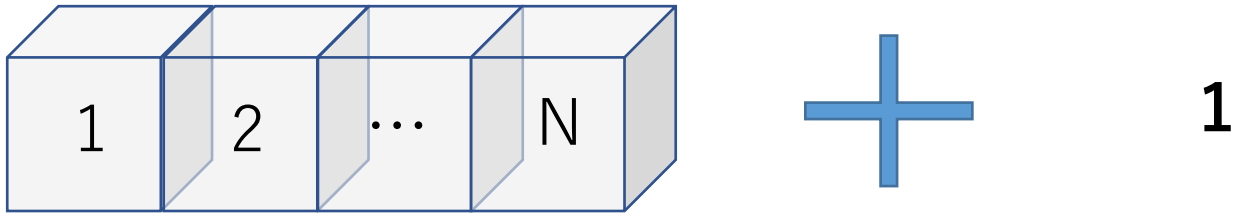
ポイント

CPUとGPUで実行速度の計測方法が異なる

サンプルプログラム

1次元Numpyアレーに1を足すというサンプルプログラムで計算速度の比較を行う

1d Numpy array

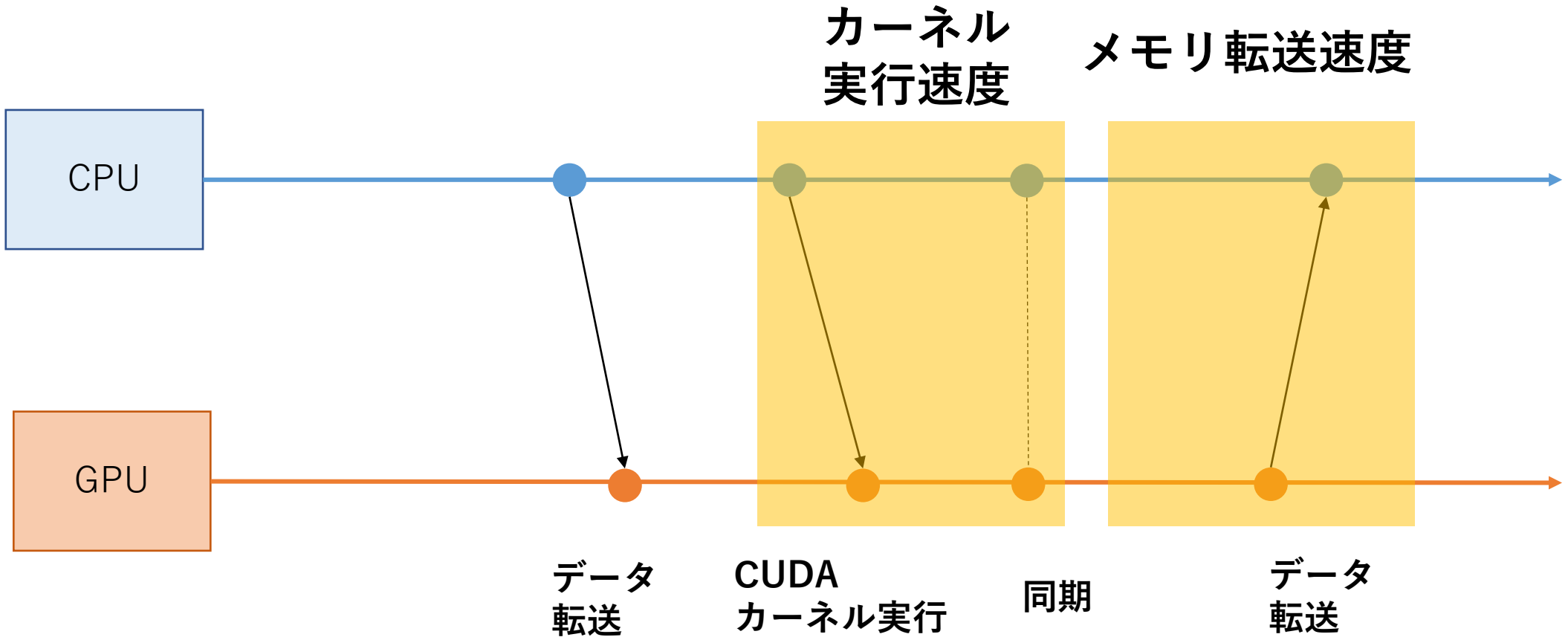


目的

配列の要素数が増加していった場合に
CPUとGPUで実行速度の変化を比較

GPUの実行速度について

今回はカーネルの実行とCPUへのメモリ転送を分けて計測



2次元配列の四則演算

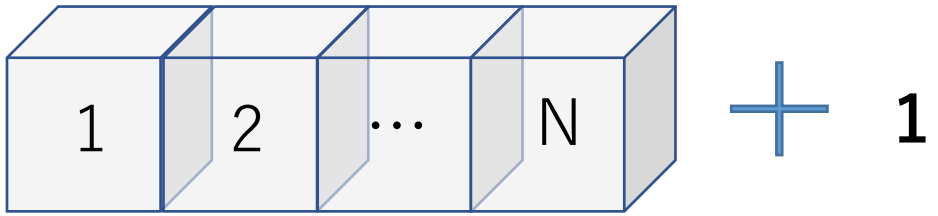
この動画で学ぶこと

- ✓ 2次元配列に対してPyCUDAで四則演算を行う際のインデックスの指定方法を理解する

1次元配列の四則演算

インデックスiを計算して実行

1次元の場合



＋ **1**

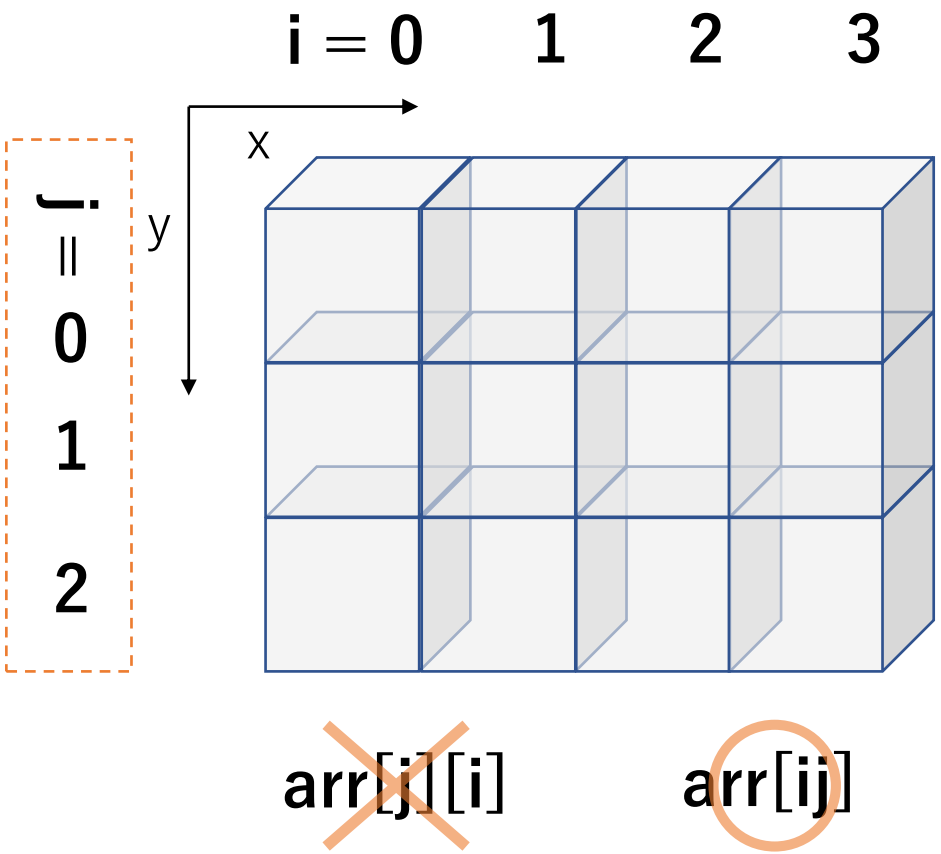
i = 0　1　2　3

```
__global__ void plus_one_kernel(int num_comp,
 int *y, int *x){
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i < num_comp){
       y[i] = x[i] + 1;
   }
}
```

2次元配列の四則演算

y方向のインデックスjも計算する
ただし、arr[j][i]のように2次元のインデックスは使えない

2次元の場合

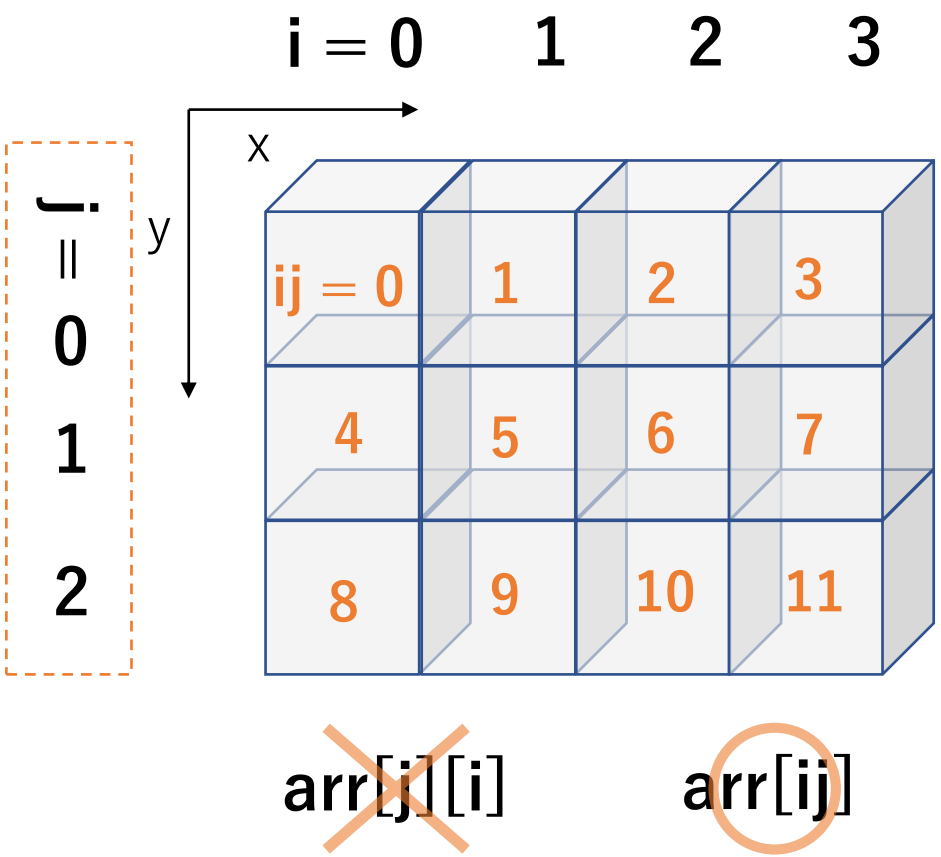



```
__global__ void add_plus_one_2d_kernel(int nx, int ny, flo
at *output, float *arr){
    const int i = threadIdx.x + blockDim.x * blockIdx.x;
    const int j = threadIdx.y + blockDim.y * blockIdx.y;
    int ij = nx * j + i;
    if (i < nx && j < ny){
        output[ij] = arr[ij] + 1;
    }
}
```

2次元配列の四則演算

CUDAでは2次元配列は1次元化してインデックスを計算する必要がある

2次元の場合




```
__global__ void add_plus_one_2d_kernel(int nx, int ny, flo
at *output, float *arr){
    const int i = threadIdx.x + blockDim.x * blockIdx.x;
    const int j = threadIdx.y + blockDim.y * blockIdx.y;
    int ij = nx * j + i;
    if (i < nx && j < ny){
        output[ij] = arr[ij] + 1;
    }
}
```

ポイント

CUDAでは多次元配列が
全て1次元化される

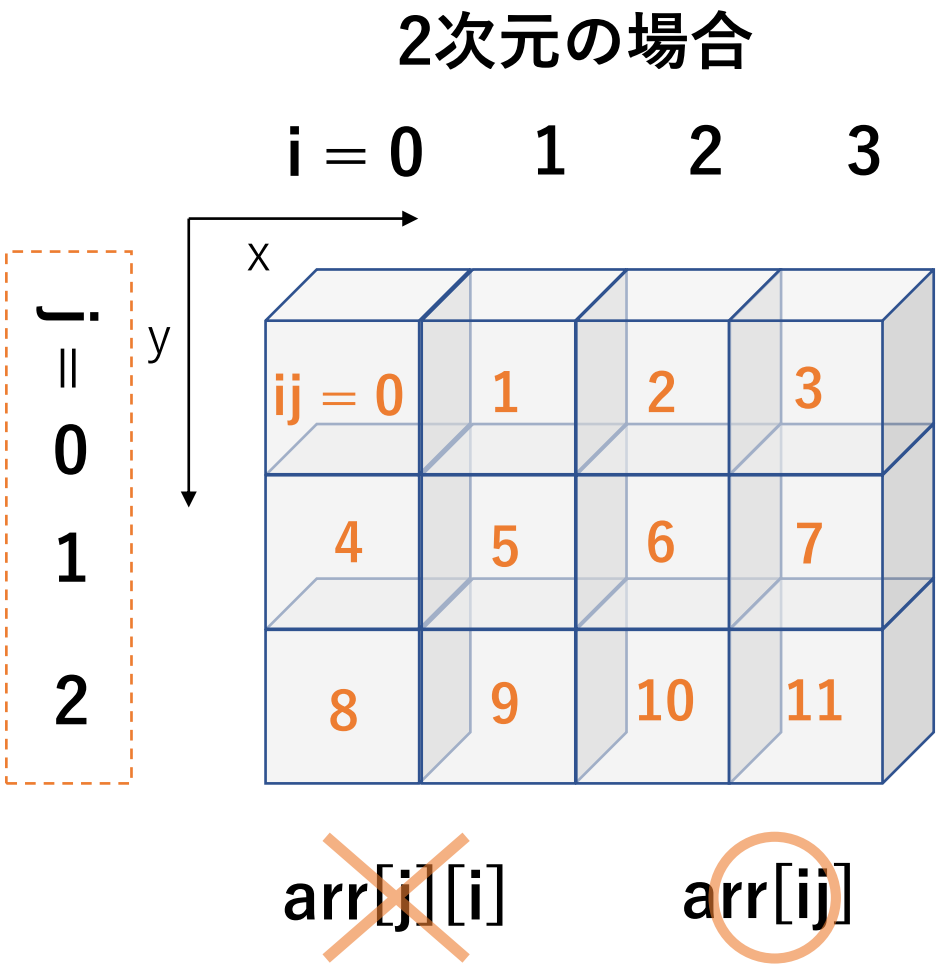
多次元配列の四則演算

この動画で学ぶこと

- ✓ 3次元以上など多次元の配列の四則演算方法について理解する

2次元配列の四則演算

CUDAでは2次元配列は1次元化してインデックスを計算する必要がある



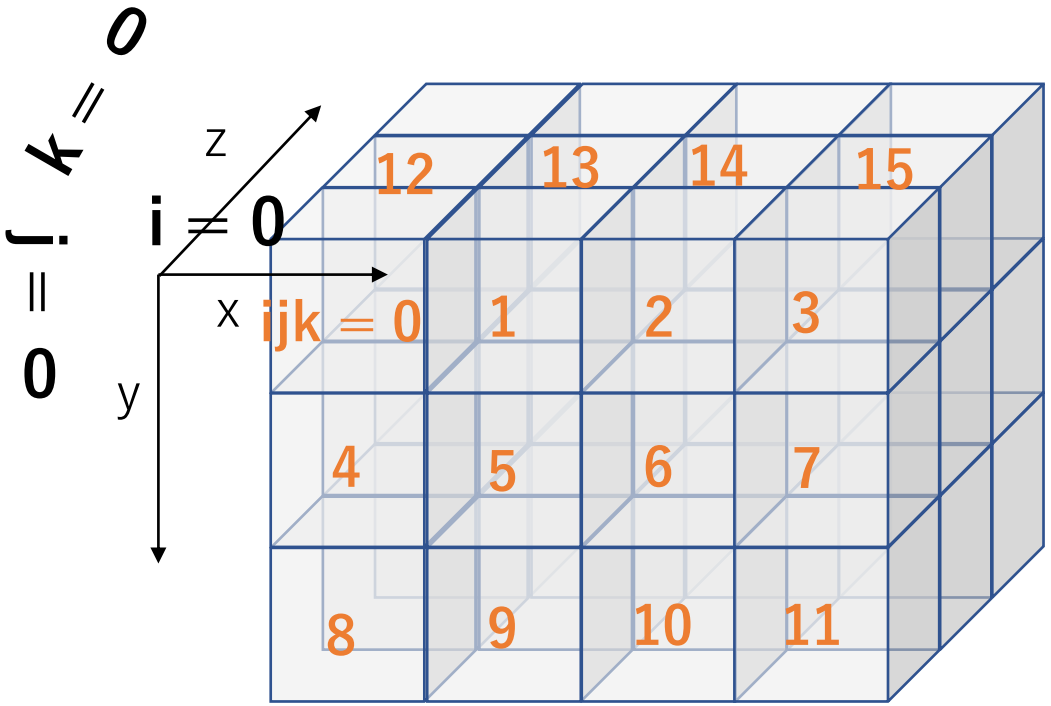

```
__global__ void add_plus_one_2d_kernel(int nx, int ny, flo
at *output, float *arr){
    const int i = threadIdx.x + blockDim.x * blockIdx.x;
    const int j = threadIdx.y + blockDim.y * blockIdx.y;
    int ij = nx * j + i;
    if (i < nx && j < ny){
        output[ij] = arr[ij] + 1;
    }
}
```

ポイント

CUDAでは多次元配列が
全て1次元化される

多次元配列の四則演算

3次元以上でも2次元と同じように1次元化して考える




```
const int i = threadIdx.x + blockDim.x * blockIdx.x;
const int j = threadIdx.y + blockDim.y * blockIdx.y;
const int k = threadIdx.z + blockDim.z * blockIdx.z;
int ijk = nx * ny * z + nx * y + x;
```

ポイント

CUDAでは多次元配列が
全て1次元化される

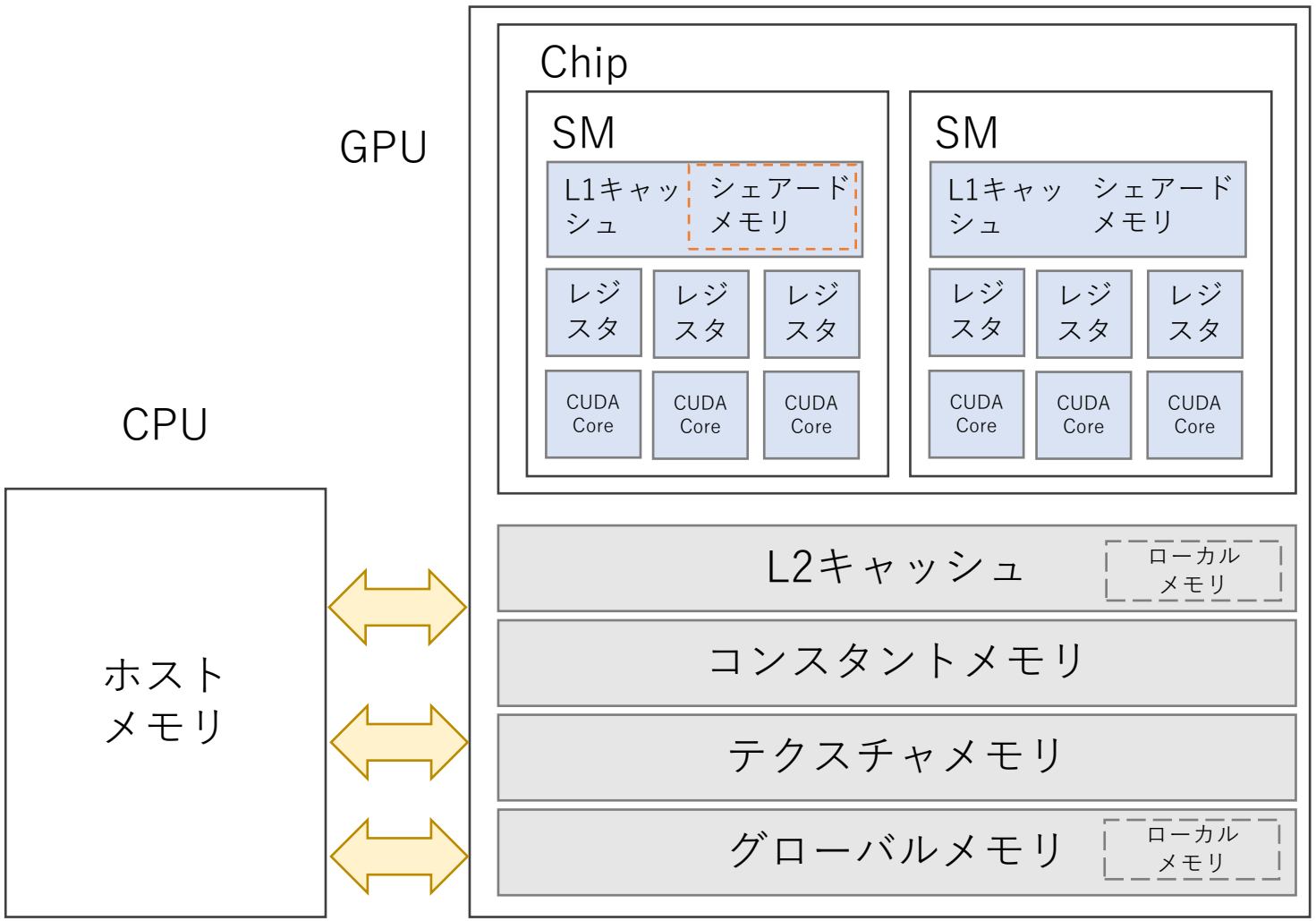
シェアードメモリ

この動画で学ぶこと

- ✓ シェアードメモリの特徴と実際の使い方について理解する

シェアードメモリの特徴

GPUは独自のメモリ構造を持つ。大まかにはオンチップとオフチップに分かれる
シェアードメモリは代表的なオンチップメモリ



オンチップメモリ

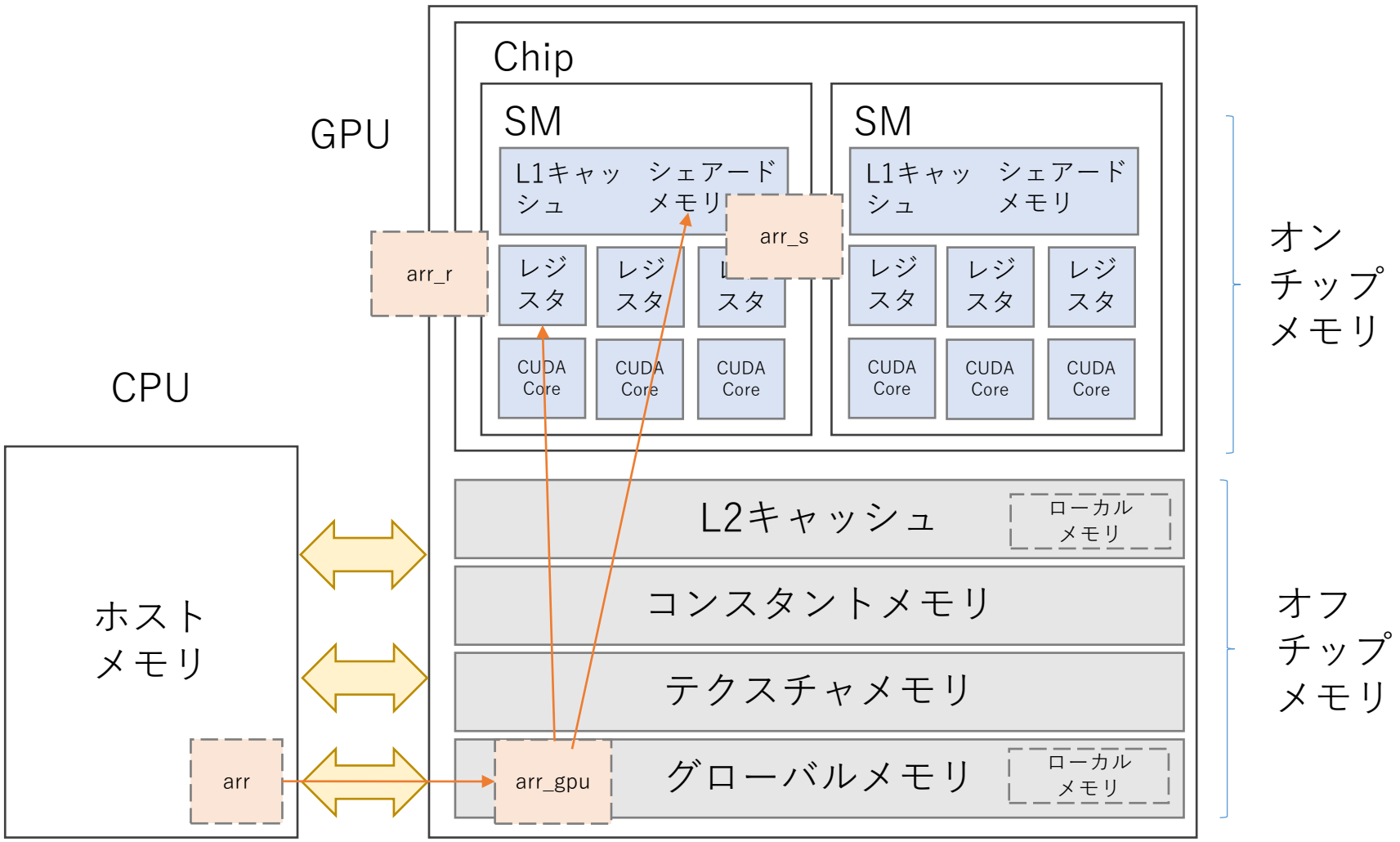
CUDA Coreに近い場所に
メモリがある
メモリ容量：小
アクセス速度：大

シェアードメモリ

同じブロック内で
メモリを相互に参照できる
(共有されている)

メモリ受け渡しの流れ-1

グローバルメモリはアクセスが遅い。レジスタやシェアードメモリに移して使う



グローバルメモリ

最も一般的なメモリ
メモリ容量が大きい
アクセス速度が遅い

レジスタ

スレッド毎に異なる値を保存
メモリ容量が小さい
アクセス速度が速い

シェアードメモリ

ブロックで共有されるメモリ
メモリ容量が小さい
アクセス速度が速い

シェアードメモリの使い方

```
# define NUM_THREADS 6
# define X_DIRECTION 0
# define Y_DIRECTION 1
# define Z_DIRECTION 2
# define NUM_HALO 2
__global__ void sample_shared_3d(int nx, int ny, int nz, float dx
, float *arr_grad, float *arr){
    __shared__ float arr_s[NUM_THREADS+NUM_HALO][NUM_THREADS+NUM_
HALO][NUM_THREADS+NUM_HALO];
    const int x = threadIdx.x + blockDim.x * blockIdx.x;
    const int y = threadIdx.y + blockDim.y * blockIdx.y;
    const int z = threadIdx.z + blockDim.z * blockIdx.z;
    const int tx = threadIdx.x + 1;
    const int ty = threadIdx.y + 1;
    const int tz = threadIdx.z + 1;
    const int nxyz = nx * ny * nz;
    if (x < nx && y < ny && z < nz){
        int ijk = nx * ny * z + nx * y + x;
        //  グローバルメモリの中身をシェアードメモリにコピー
        arr_s[tz][ty][tx] = arr[ijk];
        //  ここで、Halo領域にも値をコピー
        //  必要に応じてブロック内で同期をとる
        __syncthreads();
        //
        // ここにカーネルの実行内容を書く
        //
    }
}
```

シェアードメモリを
使った計算の流れ

CUDAカーネル内に
シェアードメモリを宣言

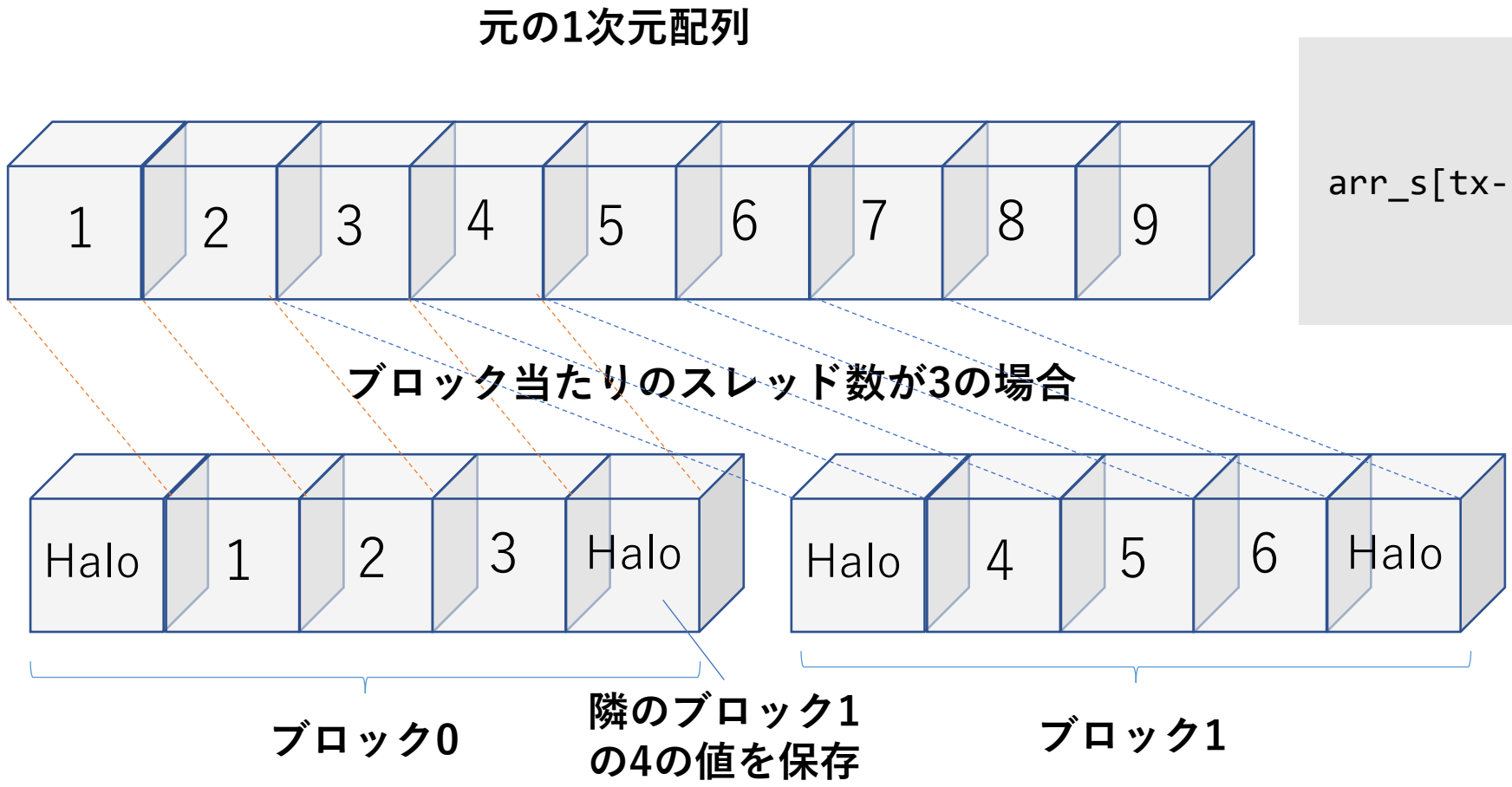
シェアードメモリに
グローバルメモリの中身をコピー

シェアードメモリを使った
演算内容を書く

Haloについてこの後、説明

Halo(袖領域)について

多くの演算では隣のメモリを参照する事が多い
各ブロックには袖領域(Halo)という１つ多くメモリを確保しておく必要がある




```
arr_ave =
arr_s[tx-1] + arr_s[tx] + arr_s[tx+1] ;
arr_ave /= 3.0;
```

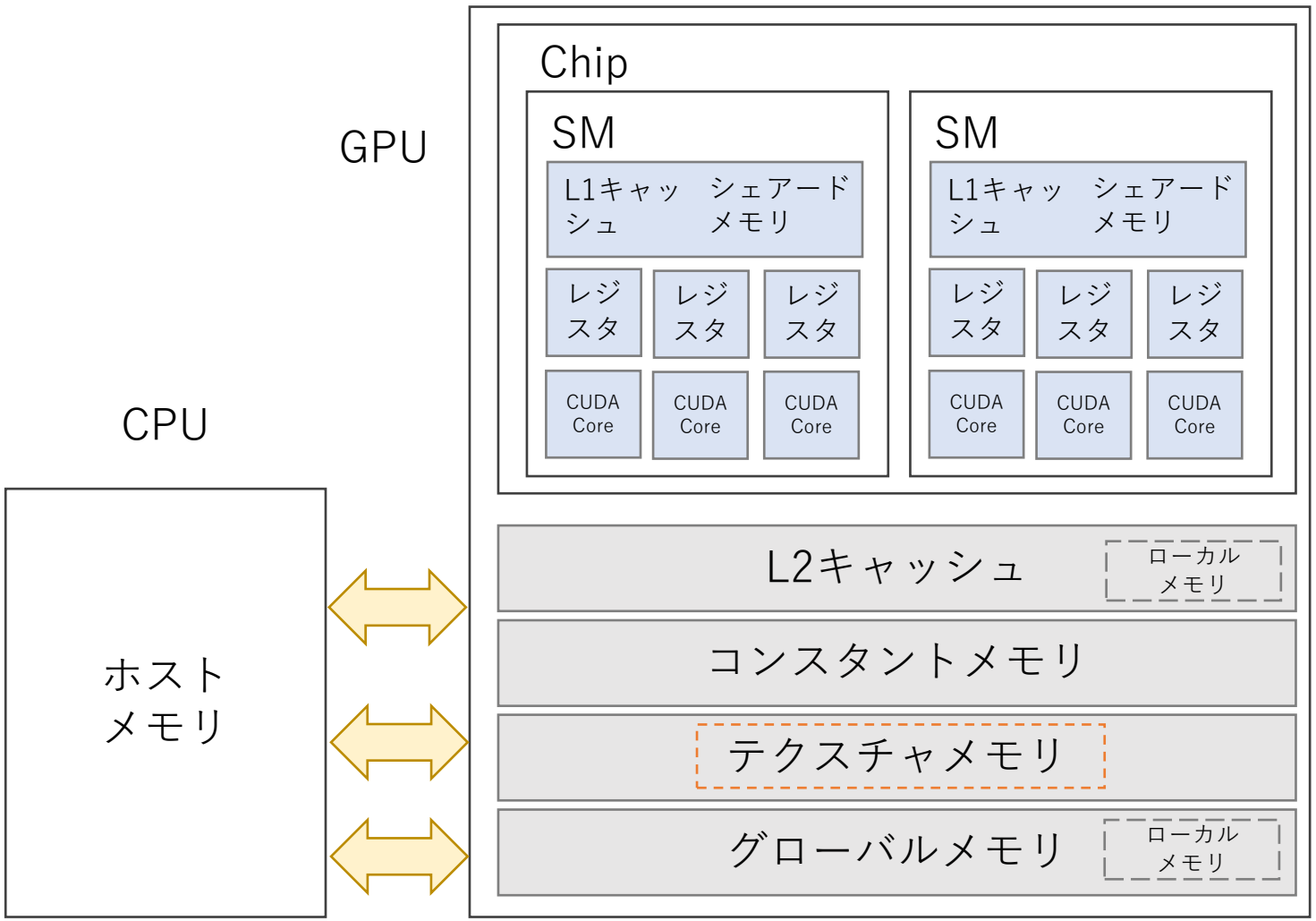
テクスチャメモリ

この動画で学ぶこと

- ✓ テクスチャメモリの特徴と使い方を理解する

テクスチャメモリの特徴

テクスチャメモリは大容量かつアクセスが高速なオフチップメモリ
ただし、GPU上からは読み取り専用のメモリ



オフチップメモリ

CUDA Coreから遠い場所に
メモリがある
メモリ容量：大
アクセス速度：遅*

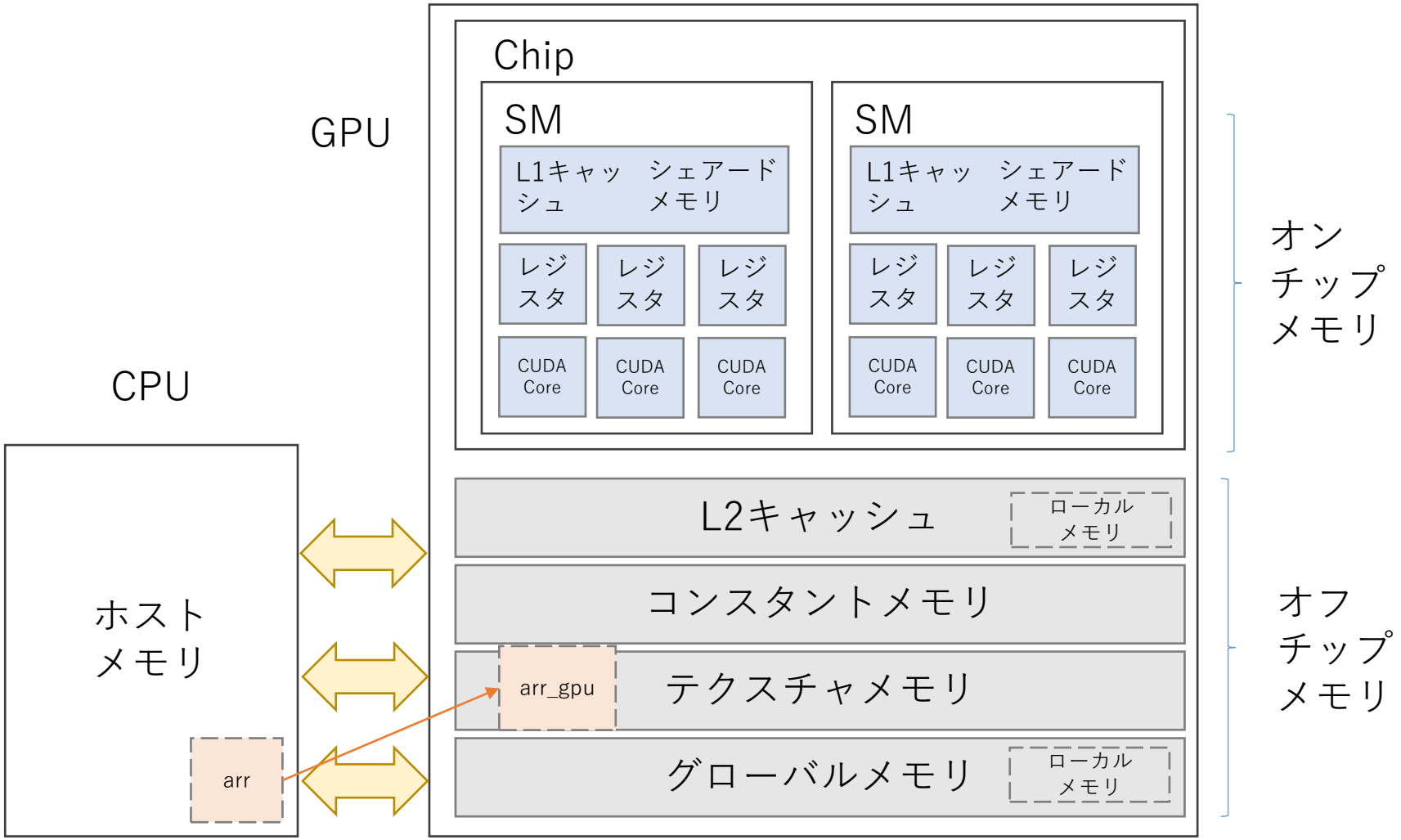
*テクスチャメモリを除く

テクスチャメモリ

オフチップメモリだが高速に
アクセスできる。
その代わり、GPU上で
書き換えができない

メモリ受け渡しの流れ-2

CPUからメモリをバインドすると高速・大容量なテクスチャメモリを利用できる



テクスチャメモリ

GPU上では読み取り専用
(書き換えできない)
メモリ容量が大きい
アクセス速度が大きい

テクスチャメモリの使い方

CPUからメモリをバインドし、CUDAカーネル内で関数から参照。
次元によってバインドの仕方や参照する関数が異なる

テクスチャメモリを使った計算の流れ

CUDAカーネル内に
テクスチャメモリ変数を宣言

↓

CPUからメモリをバインド

↓

カーネル内でテクスチャメモリを参照

次元ごとの違い

	バインド	参照関数
1次元	.bind_to_texref_ext	tex1Dfetch
2次元	drv.matrix_to_texref	tex2D
3次元	独自の関数を定義*	tex3D

*本講座ではユーティリティ関数を用意

ライブラリの利用

この動画で学ぶこと

- ✓ 並列化しにくい処理について理解する
- ✓ PyCUDAのライブラリについて理解する

並列化が難しい処理

ブロックをまたいだ変数を参照、メモリの交換処理は手続きが面倒
代表的な処理として 縮約（総和・内積）/ 最大・最小 / 並び替えなど

総和
内積

最大・最小

並び替え

並列化が難しい処理

これらの処理は自作でプログラミングするのではなくライブラリを利用する

	利用するライブラリ・関数
総和 内積	InclusiveScanKernel atomic演算 ReductionKernel
最大・最小	InclusiveScanKernel cuBLAS
並び替え	thrust
番外編	テンプレートの利用 ダイナミックパラレル

総和

この動画で学ぶこと

✓ 複数の方法で総和計算が行える

並列化が難しい処理

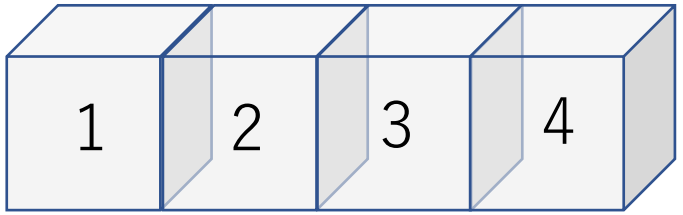
これらの処理は自作でプログラミングするのではなくライブラリを利用する

	利用するライブラリ・関数
総和 内積	InclusiveScanKernel atomic演算 ReductionKernel
最大・最小	InclusiveScanKernel cuBLAS
並び替え	thrust
番外編	テンプレートの利用 ダイナミックパラレル

スキャン/プレフィックスサムを用いた総和計算

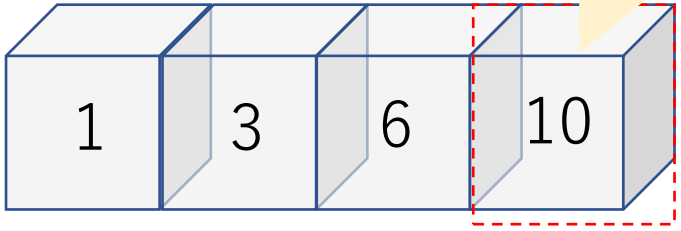
PyCUDAには累積和(スキャン)を計算するための関数が用意されている
スキャンを計算する事で、総和算出もできる

1d Numpy array



スキャン

最後の要素は
総和



累積和(np.cumsum)の事を
スキャンやプレフィックスサムと呼ぶ

スキャンのための関数

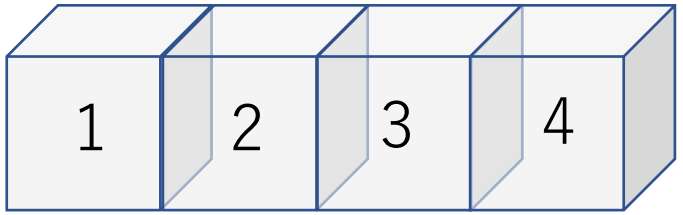
```
from pycuda.scan import InclusiveScanKernel

scan_kernel =InclusiveScanKernel(np.int32, "a+b")
```

2項演算子

アトミック演算を用いた総和計算

アトミックとは”不可分性”。メモリを書き込んでいる時に他からの書き込みを禁止
atomicAddでは配列の総和を安全に計算

1d Numpy array



アトミック演算

```
int sum = 0;
for (int i = 0; i < num_comp; i++){
    sum += arr[i];
}
```

atomicAdd

GPU上でスレッドの１つが
合計値を計算し、メモリ(sum)に書き込み

他のスレッドはsumが書き込まれるまで
待っている

GPU上で逐次計算をしているようなもの
大きな配列では遅くなるので多用は禁物

内積

この動画で学ぶこと

- ✓ ReductionKernelを使った配列の内積について理解する

並列化が難しい処理

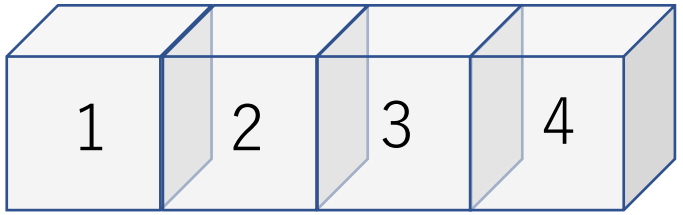
これらの処理は自作でプログラミングするのではなくライブラリを利用する

	利用するライブラリ・関数
総和 内積	InclusiveScanKernel atomic演算 ReductionKernel
最大・最小	InclusiveScanKernel cuBLAS
並び替え	thrust
番外編	テンプレートの利用 ダイナミックパラレル

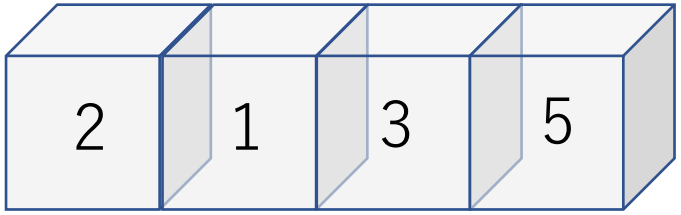
内積

２つのベクトルから１つの出力値を計算
縮約やリダクションと呼ばれる

array 1

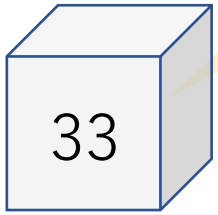


array 2



内積

1 x 2 + 2 x 1 +
3 x 3 + 4 x 5



1つの値に
縮約されている

ReductionKernel

２項演算子と縮約操作の２つを定義する事で、内積を計算できる

```
from pycuda.reduction import ReductionKernel

reduction_kernel = ReductionKernel(np.float32, neutral="0",
                                   reduce_expr="a+b", map_expr="x[i]*y[i]",
                                   arguments="float *x, float *y")
```

縮約操作

2項演算子

最大値・最小値

この動画で学ぶこと

- ✓ InclusiveScanKernelを使って、最大値・最小値を求める
- ✓ cuBLASを使って、最大値のインデックスを求める

並列化が難しい処理

これらの処理は自作でプログラミングするのではなくライブラリを利用する

	利用するライブラリ・関数
総和 内積	InclusiveScanKernel atomic演算 ReductionKernel
最大・最小	InclusiveScanKernel cuBLAS
並び替え	thrust
番外編	テンプレートの利用 ダイナミックパラレル

InclusivescanKernelでの最大・最小

2項演算子を変更する事で、最大・最小値も簡単に求まる

最大値を計算すると2項演算子

```
from pycuda.scan import InclusiveScanKernel

# 配列の最大値を求める為の2項演算子
scan_kernel =InclusiveScanKernel(np.int32, "a > b ? a: b")
```

2項演算子

cuBLASで最大・最小位置

BLASとはベクトル・行列用の演算が高速に行えるライブラリ
cuBLASを使うと、値が最大・最小の位置を得る事が出来る

BLAS(**B**asic **L**inear **A**lgebra **S**ubprograms)

Level 1:
ベクトル同士の演算
$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$

Level 2:
ベクトルと行列の演算
$\boldsymbol{y} \leftarrow \alpha \boldsymbol{A}\boldsymbol{x} + \beta \boldsymbol{y}$

Level 3:
行列同士の演算
$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$

それ以外:
ベクトルの最大/最小位置などを求める

cuBLASの一般的な流れ

skcudaからimport

ハンドル変数の設定

```
from skcuda import cublas

# cublasで最大値を求める
h = cublas.cublasCreate()
max_id = cublas.cublasIsamax(h, x_gpu.size, x_gpu.gpudata,
 1)
cublas.cublasDestroy(h)
```

cuBLASの関数を呼ぶ

使い終わったらセッションを消去

ベクトル演算

この動画で学ぶこと

- ✓ cuBLASのLevel 1の演算を使ってベクトル演算を行ってみる

並列化が難しい処理

これらの処理は自作でプログラミングするのではなくライブラリを利用する

	利用するライブラリ・関数
総和 内積	InclusiveScanKernel atomic演算 ReductionKernel
最大・最小	InclusiveScanKernel cuBLAS
並び替え	thrust
番外編	テンプレートの利用 ダイナミックパラレル

cuBLAS

ベクトル演算はCUDA Cでも書けるがBLASを利用する事でより高速に動作

BLAS(**B**asic **L**inear **A**lgebra **S**ubprograms)

Level 1:
ベクトル同士の演算
$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$

Level 2:
ベクトルと行列の演算
$\boldsymbol{y} \leftarrow \alpha \boldsymbol{A}\boldsymbol{x} + \beta \boldsymbol{y}$

Level 3:
行列同士の演算
$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$

それ以外:
ベクトルの最大/最小位置などを求める

Level 1の演算

```
h = cublas.cublasCreate()
# y <- a * x + yを行う
cublas.cublasSaxpy(h, x_gpu.size, a, x_gpu.gpudata, 1, y_g
pu.gpudata, 1)
cublas.cublasDestroy(h)
```

関数名で
計算内容が分かる

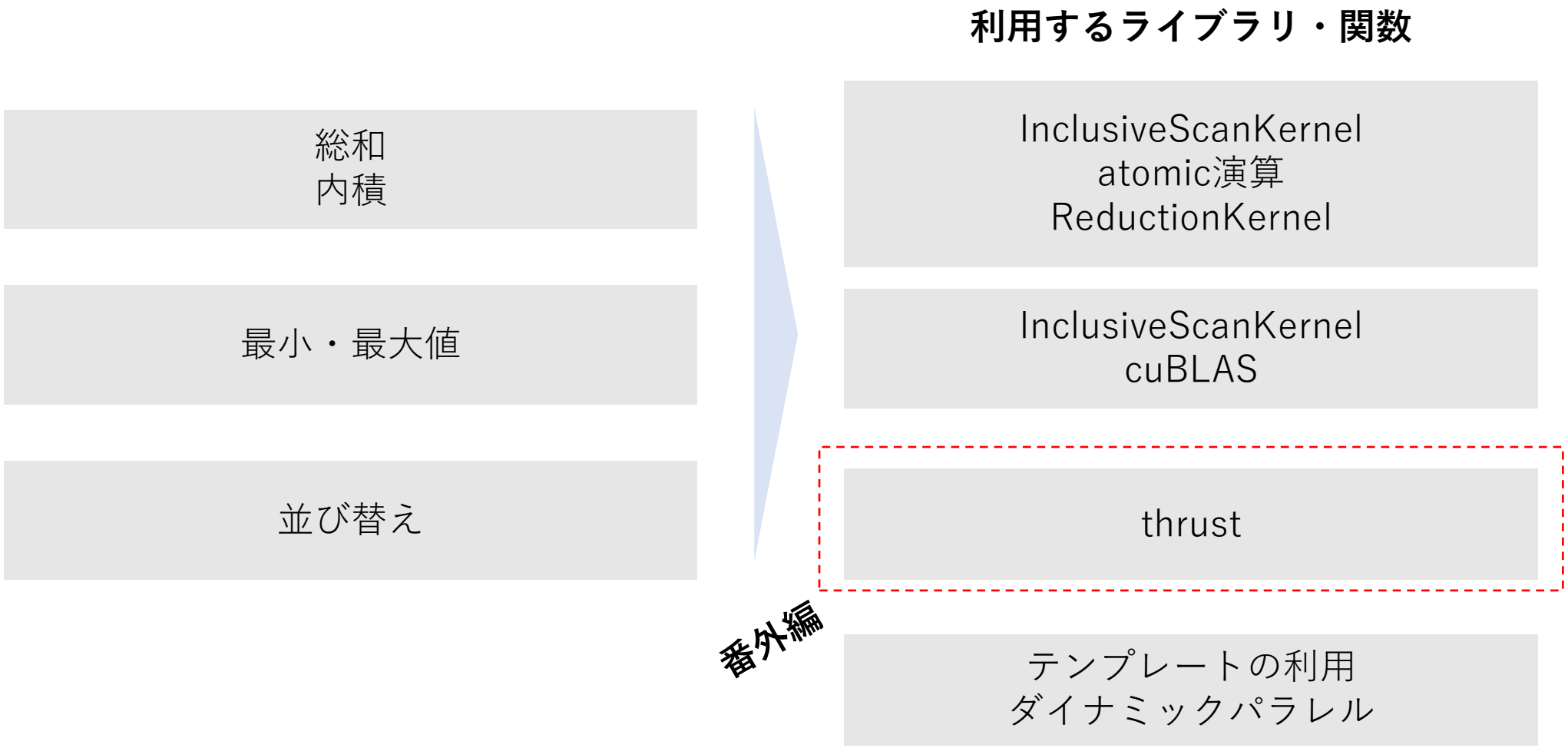
並び替え

この動画で学ぶこと

- ✓ Thrustを使った配列の並び替えについて理解する

並列化が難しい処理

これらの処理は自作でプログラミングするのではなくライブラリを利用する



thrustとは

C++で書かれたCUDA C用のライブラリ
CUDA Cで面倒なメモリ管理や並列化の面倒な処理を簡単に扱える

CUDA Cでのメモリ管理

```
// 変数サイズの計算
int num_comp = 3;
float x[num_comp] = {1, 2, 3};
size_t n_bytes = num_comp * sizeof((float));

// GPU用のポインタを宣言
float *x_gpu;

// GPUにヒープメモリを確保
cudaMalloc((float**) &x_gpu, n_bytes);

// CPUからGPUにメモリをコピー
cudaMemcpy(x_gpu, x, n_bytes, cudaMemcpyHostToDevice);

//
//  ここでCUDAカーネルを実行
//

// GPUからCPUへメモリをコピー
cudaMemcpy(x, x_gpu, n_bytes, cudaMemcpyDeviceToHost);

// GPU上のメモリを解放
cudaFree(x_gpu);
```

thrustでのメモリ管理

```
#include<thrust/host_vector.h>
#include<thrust/device_vector.h>

int main(){
    // CPU側でメモリを確保
    thrust::host_vector<int> arr(3);

    arr[0] = 1;
    arr[1] = 2;
    arr[2] = 3;
    // GPU側にメモリをコピー
    thrust::device_vector<int> arr_gpu = arr;

    return 0;
}
```

cudaMalloc
cdaMemcpy
cudaFreeなど不要

thrustとは

メモリ管理が簡単なこと以外にもスキャン、縮約、並び替えなどを実装
PyCUDAでは主に並び替えの為に利用

```
#include <thrust/sort.h>
#include <thrust/execution_policy.h>

extern "C" {

__global__ void sort_thrust(int num_component, int *arr){
thrust::sort(thrust::device, arr, (arr + num_component));
}

__global__ void sort_by_key_thrust( int num_component, int *key, int *
value){
thrust::sort_by_key(thrust::device, key, (key + num_component), value)
;
}
}
```

並び替えるには
関数を呼び出すだけ

thrustの主な機能

スキャン
縮約
並び替え

テンプレート

この動画で学ぶこと

- ✓ 同じ関数を別の変数の型でも再利用できるテンプレートの使い方を理解する

並列化が難しい処理

これらの処理は自作でプログラミングするのではなくライブラリを利用する

	利用するライブラリ・関数
総和 内積	InclusiveScanKernel atomic演算 ReductionKernel
最小・最大値	InclusiveScanKernel cuBLAS
並び替え	thrust
番外編	テンプレートの利用 ダイナミックパラレル

テンプレートとは

C++の機能でclass Tの部分を指定した型に読み替えることができる

templateの宣言

templateの利用例

おまじない

```
template <class T>
__device__ T add_two_vector(T x, T y){
    return (x + y);
}
```

戻り値にTを指定

```
extern "C" {
__global__ void add_two_vector_kernel(int nx, int *a, int *b,
int *res){
    const int x = threadIdx.x + blockDim.x * blockIdx.x;
    if (x < nx){
        res[x] = add_two_vector<int>(a[x], b[x]);
    }
}
}
```

呼び出し時に
Tの型を指定

使いどころ

同じ処理内容で
変数の型が違うだけという
CUDAカーネルを整理できる

ダイナミックパラレル

この動画で学ぶこと

- ✓ CUDAカーネルから別のCUDAカーネルを呼び出す方法について理解する

並列化が難しい処理

これらの処理は自作でプログラミングするのではなくライブラリを利用する

	利用するライブラリ・関数
総和 内積	InclusiveScanKernel atomic演算 ReductionKernel
最小・最大値	InclusiveScanKernel cuBLAS
並び替え	thrust
番外編	テンプレートの利用 ダイナミックパラレル

ダイナミックパラレルとは

ダイナミックパラレルではCUDAカーネル中で別のカーネルを呼び出し可能

呼ばれるCUDAカーネル

```
__global__ void add_two_vector(int nx, float *arr1, float *arr2, float *res){
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    if (x < nx){
        res[x] = arr1[x] + arr2[x];
    }
}
```

使いどころ

**複数のCUDAカーネルを
１つCUDAカーネルに
束ねることが出来る**

**ブロック,
グリッドの設定**

呼び出す側のCUDAカーネル

```
__global__ void add_two_vector_dynamic(int *grid, int *block
, int nx, float *arr1, float *arr2, float *res){
dim3 grid_ = dim3(grid[0], grid[1], grid[2]);
dim3 block_ = dim3(block[0], block[1], block[2]);
// カーネルを呼び出す
add_two_vector<<<grid_, block_>>>(nx, arr1, arr2, res);
}
```

**別のカーネルの呼び出し
（CUDA Cのマナー）**