GPUプログラミング・基礎編

東京工業大学学術国際情報センター

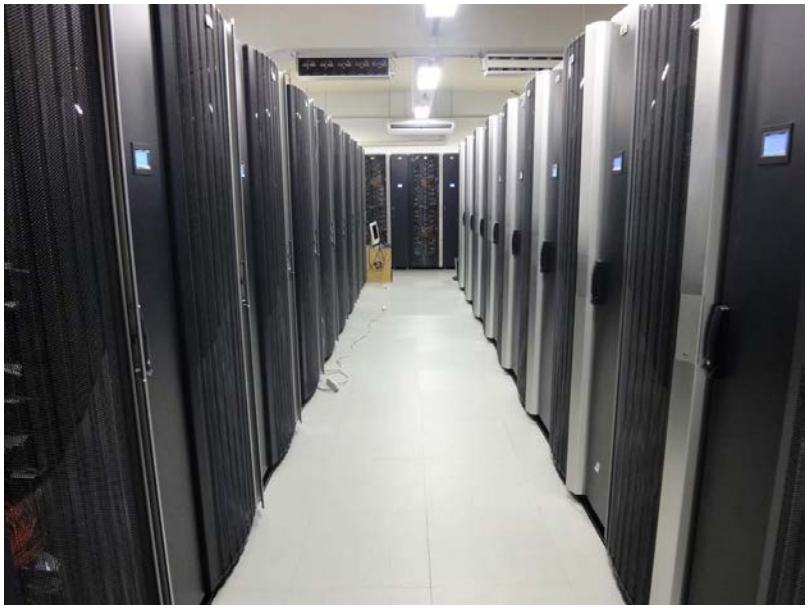
(8/16修正版)

1. GPUコンピューティングとTSUBAME2.0スーパーコンピュータ

GPUコンピューティングとは

- グラフィックプロセッサ (GPU)は、グラフィック・ゲームの画像計算のために、進化を続けてきた
  - 現在、CPUのコア数は2～12個に対し、GPU中には数百コア
- そのGPUを一般アプリケーションの高速化に利用！
  - GPGPU (General-Purpose computing on GPU) とも言われる
- 2000年代前半から研究としては存在。2007年にNVIDIA社のCUDA言語がリリースされてから大きな注目

TSUBAME2.0スーパーコンピュータ



Tokyo-Tech
Supercomputer and
UBiquitously
Accessible
Mass-storage
Environment

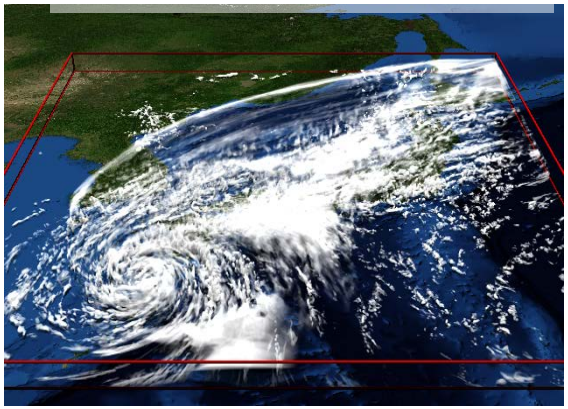
「ツバメ」は東京工業大学のシンボルマークでもある

- TSUBAME1: 2006年～2010年に稼働したスパコン
- TSUBAME2.0: 2010年に作られたスパコン
  - 2010年には、世界4位、日本1位の計算速度性能
  - 現在、世界20位、日本3位

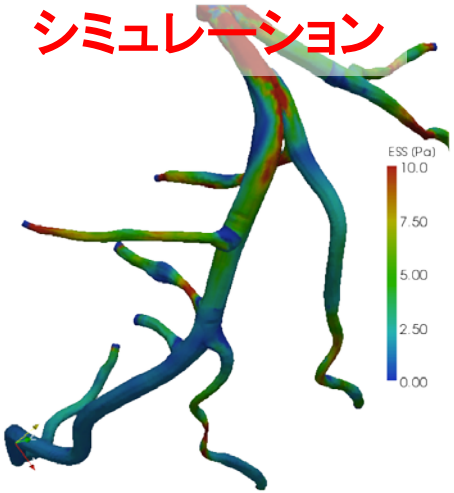
高性能の秘訣が GPUコンピューティング

TSUBAME2.0スパコン・GPUは様々な研究分野で利用されている

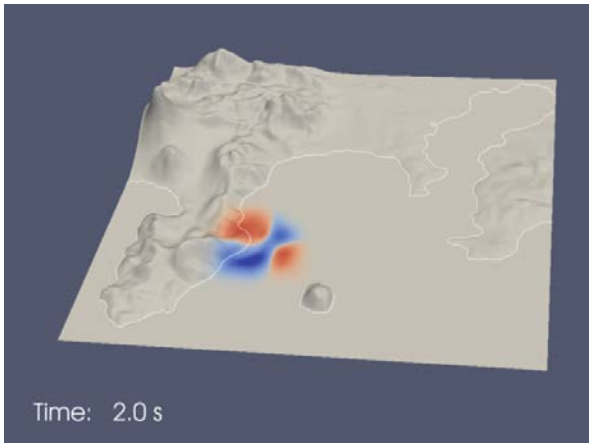
気象シミュレーション



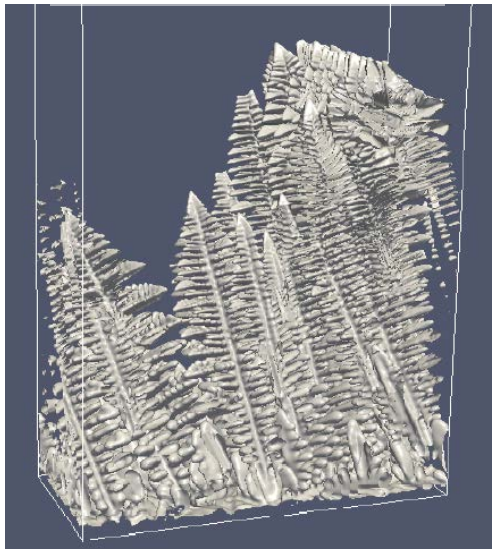
動脈血流
シミュレーション



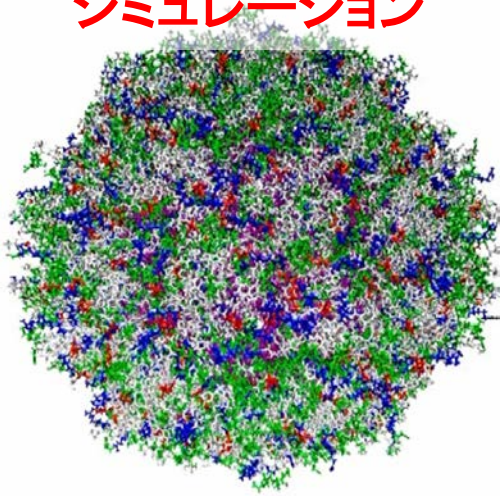
津波・防災
シミュレーション



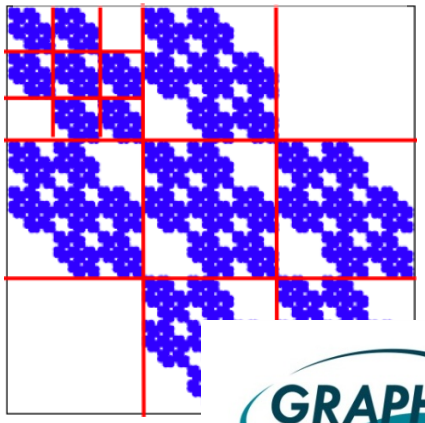
金属結晶凝固
シミュレーション



ウィルス分子
シミュレーション



グラフ構造解析



TSUBAME2.0の計算ノード

- TSUBAME2.0は、約1400台の計算ノード(コンピュータ)を持つ
- 各計算ノードは、CPUとGPUの両方を持つ
  - CPU: Intel Xeon 2.93GHz 6コア x 2CPU=12 コア
  - GPU: NVIDIA Tesla M2050 3GPU

    CPU 140GFlops + GPU 1545GFlops = 1685GFlops

GFlopsは計算速度の単位。
9割の性能がGPUのおかげ!

  - メインメモリ(CPU側メモリ): 54GB
  - SSD: 120GB
  - ネットワーク: QDR InfiniBand x 2 = 80Gbps
  - OS: SUSE Linux 11 (Linuxの一種)

GPUの特徴

- コンピュータにとりつける増設ボード

  ⇒単体では動作できず、CPUから指示を出してもらう

- CUDAプログラミング言語でプログラム作成
- 448コアを用いて計算

  ⇒多数のコアを活用するために、多数のスレッドが協力して計算。うまく使えばCPUよりはるかに速い計算が可能！

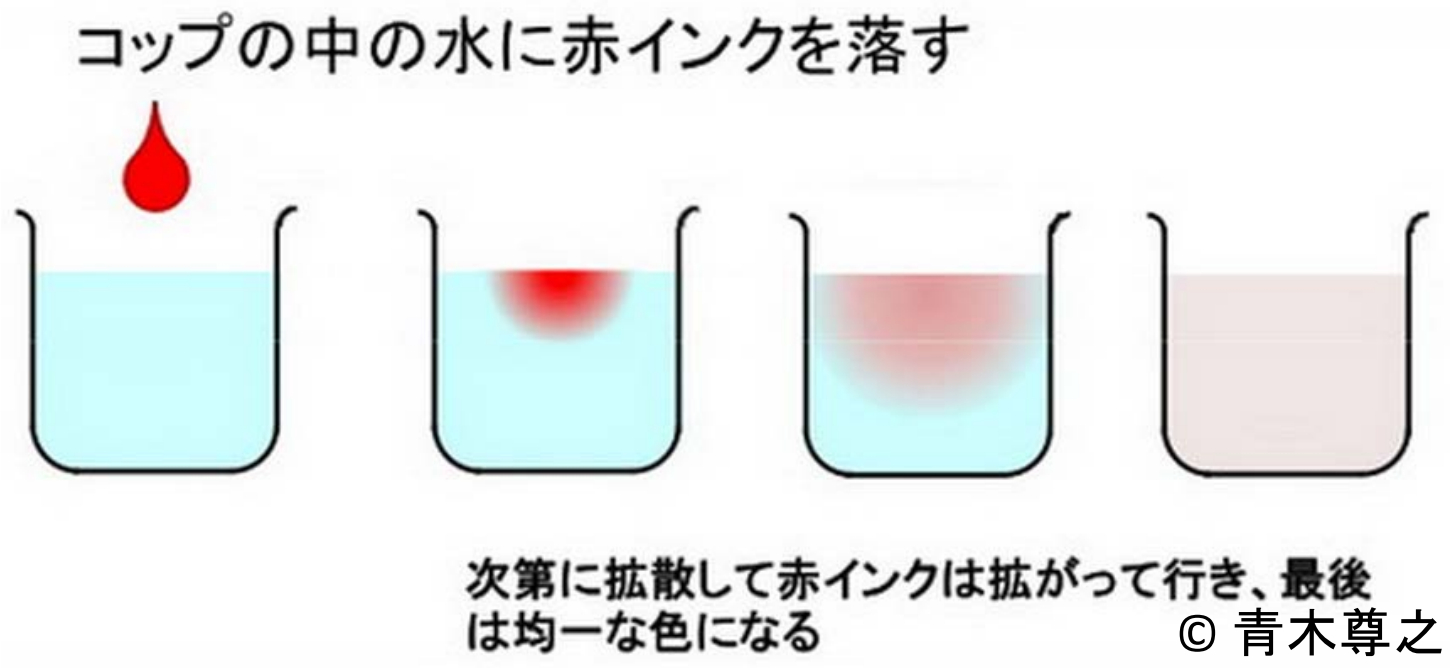
- メモリサイズ3GB (実際使えるのは約2.5GB)

  ⇒CPU側のメモリと別なので、「データの移動」もプログラミングする必要

上記のコア数・メモリサイズは、
M2050 GPU 1つあたり。
製品によっても違う

GPUの計算速度の威力
拡散現象シミュレーションを例題に

拡散現象



- 各点のインク濃度は、時間がたつと変わっていく
  → その様子を計算機で計算
  - 天気予報などにも含まれる計算

GPUの計算速度の威力

- コップを8192x8192の細かいマス目に区切り、100回の時間ステップをシミュレーションしてみた
  - TSUBAME2.0の計算ノード一つで実行
- CPU版 (C言語で書かれたdiffusion.c)

  → 20.8秒かかった

  ※1CPUコアだけ使うプログラム
- GPU版 (「CUDA」で書かれたdiffision_gpu.cu)

  → 1.26秒。CPU版より約16倍も速い！！

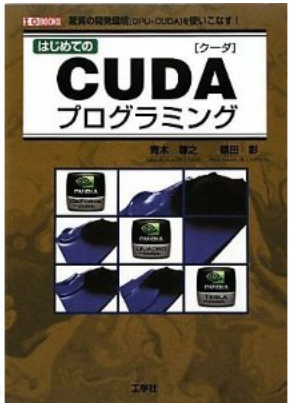
2. CUDAプログラムの流れ

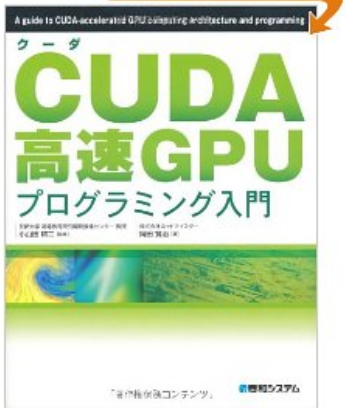
これからいよいよ、CUDAを用いて、どうやって
GPUプログラミングを行うか解説します

プログラミング言語CUDA

- NVIDIA GPU向けのプログラミング言語
  - 2007年2月に最初のリリース
  - TSUBAME2.0で使えるのはV5.0
  - Linux, Windows, MacOS対応。本講義ではLinux版
- 標準C言語サブセット＋GPGPU用拡張機能
  - C言語の基本的な知識(特にポインタ)は必要となります
- nvccコマンドを用いてコンパイル
  - ソースコードの拡張子は.cu

CUDA関連書籍もあり





著者は東工大の先生

CUDAプログラムのコンパイルと実行例

- サンプルプログラム inc_seq.cu を利用
- 以下のコマンドをターミナルから入力し、CUDAプログラムのコンパイル、実行を確認してください

```
$ nvcc inc_seq.cu -arch sm_21 -o inc_seq
$ ./inc_seq
```

- “$” はコマンドプロンプトを示しますので、”$”は入力しないでください
- -arch sm_21 は、最新のCUDA機能を使うためのオプション(普段つけておいてください)

CUDAプログラムの考え方

- 「C言語のプログラム」＋「GPU上で動く関数」

GPUカーネル関数と呼ばれる

- プログラムの実行はmain()関数からはじまり、最初はCPUだけが動作する
- CPU上の関数から、GPUカーネル関数が呼び出されてはじめて、GPUが動き出す
  - 時間のかかる処理をGPUにまかせることにより、高速化したい！
- ただしGPUカーネル関数から触れる変数・配列は限られているので、注意が必要
  - GPUカーネル関数から触れるのはGPUメモリのみ

サンプルプログラム： inc_seq.cu

int型配列の全要素を1加算

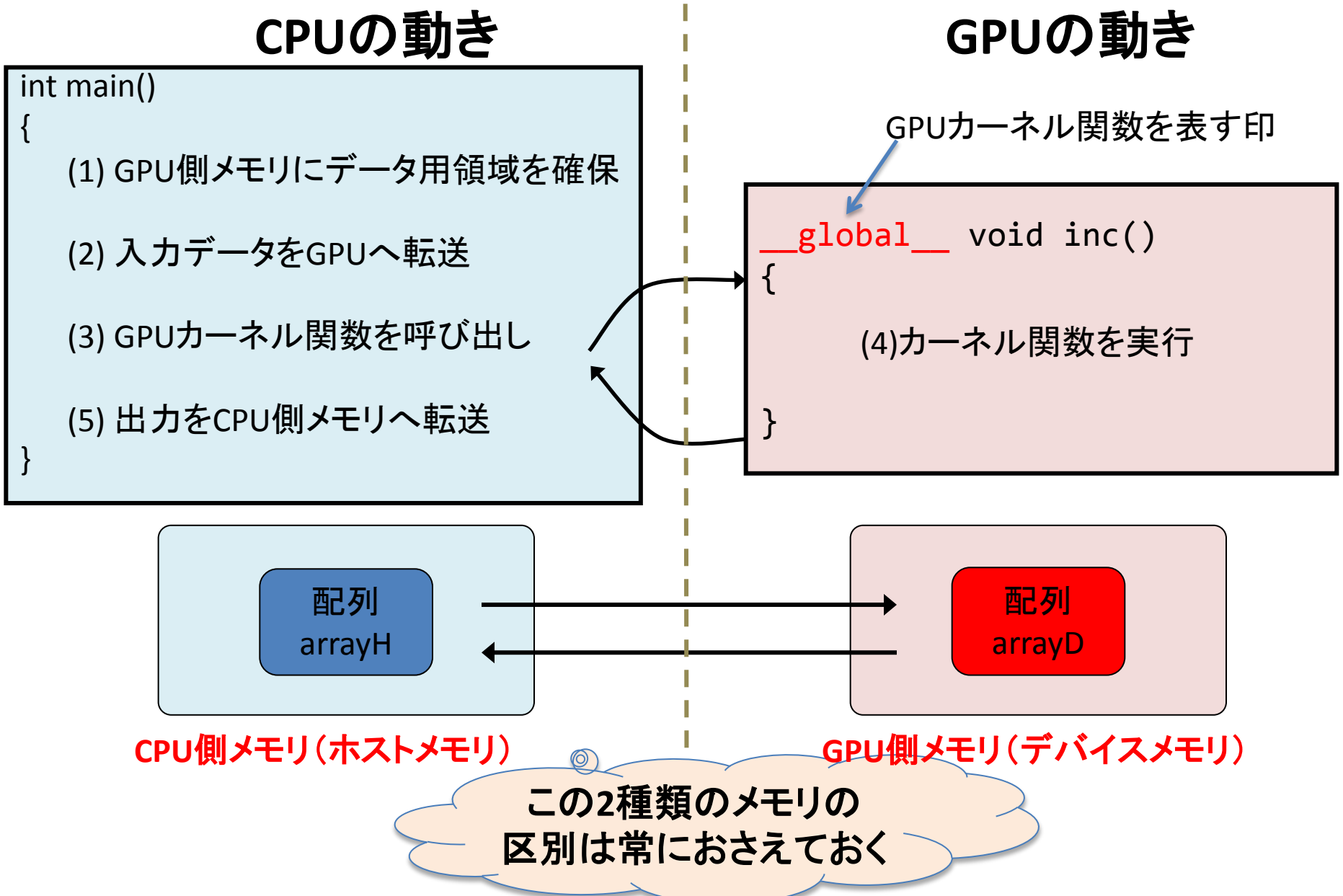
GPUであまり意味がない
(速くない)例ですが

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define N  (32)
__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++)
        array[i]++;
    return;
}
int main(int argc, char *argv[])
{
    int i;
    int arrayH[N];
    int *arrayD;
    size_t array_size;
```

無くてもok

```
    for (i=0; i<N; i++) arrayH[i] = i;
    printf("input: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("¥n");

    array_size = sizeof(int) * N;
    cudaMalloc((void **)&arrayD, array_size);
    cudaMemcpy(arrayD, arrayH, array_size,
            cudaMemcpyHostToDevice);
    inc<<<1, 1>>>(arrayD, N);
    cudaMemcpy(arrayH, arrayD, array_size,
                cudaMemcpyDeviceToHost);
    printf("output: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("¥n");
    return 0;
}
```

サンプルプログラムの流れ



(1) GPU側メモリに領域確保

- `cudaMalloc(void **devpp, size_t count)`
  - GPU側メモリ(*デバイスメモリ*と呼ばれる)に領域を確保
  - devpp: デバイスメモリアドレスへのポインタ。確保したメモリのアドレスがここへ書き込まれる
  - count: 確保したいサイズ(バイト単位)
- `cudaFree(void *devp)`
  - いらなくなった領域を解放する。

例: GPU側メモリ上に、長さ1024のintの配列を確保

```
int *arrayD;
cudaMalloc((void **)&arrayD, sizeof(int) * 1024);
// これ以降、arrayDをGPU側で配列として使うことができる
```

※ cudaMalloc/cudaFreeを呼べるのはCPU側だけ。GPUカーネル関数内では呼べない。

※ 確保した領域(上のarrayD)を読み書きできるのは(arrayD[i]++など)、GPUカーネル関数内だけ。CPU側ではアクセスできない

(2) 入力データの転送

- cudaMallocで確保しておいた領域に、CPU側メモリのデータをコピーしたい
- cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - dst: 転送先のメモリアドレス
  - src: 転送元のメモリアドレス
  - count: 転送したいサイズ(バイト単位)
  - kind: 転送の方向を指定する定数。CPUからGPUへ転送したいときは、cudaMemcpyHostToDeviceと書く

例： すでに確保した領域arrayDへCPU上のデータarrayHを転送

```
int arrayH[1024];
cudaMemcpy(arrayD, arrayH, sizeof(int)*1024,
           cudaMemcpyHostToDevice);
```

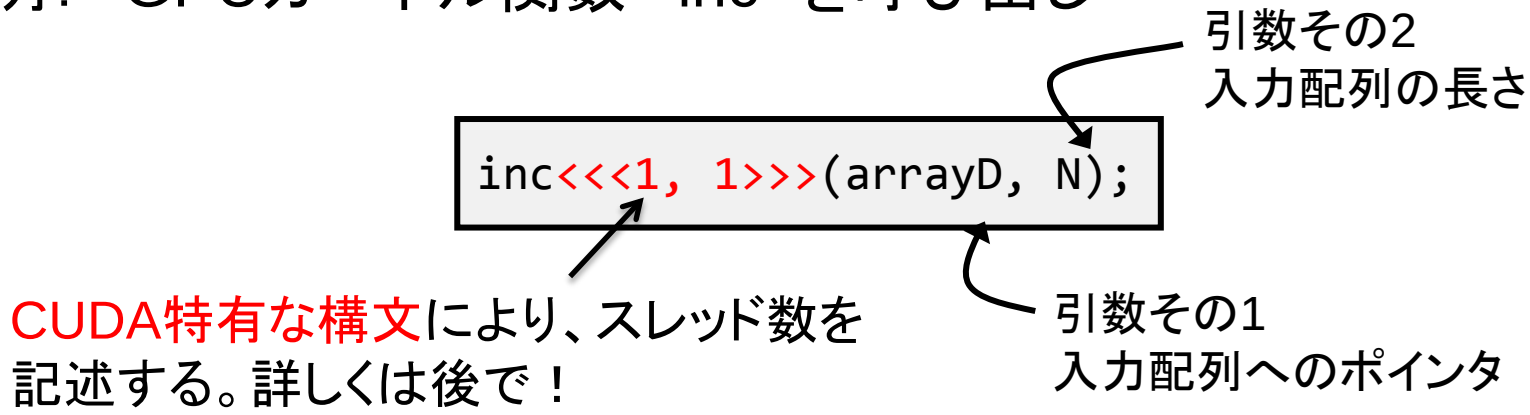
※ cudaMemcpyを呼べるのはCPU側だけ。GPUカーネル関数内では呼べない。

※ CPU(ホスト)側メモリを指すポインタと、GPU(デバイス)側メモリを指すポインタを混同しないよう注意！！間違えると謎のバグに！

(3) GPUカーネル関数の呼び出し

- `kernel_func<<<grid_dim, block_dim>>>(kernel_param1, …);`
  - `kernel_func`: GPUカーネル関数名
  - `kernel_param1,…`: GPUカーネル関数に与える引数
  - `grid_dim, block_dim`は「後で説明します」

例： GPUカーネル関数 “inc” を呼び出し



(4) GPUカーネル関数の実行

- GPUカーネル関数とは？
  - GPU上で実行される関数
  - __global__または__device__というキーワードをつける

  ※「global」「device」の前後にはアンダーバー2つずつ付きます
- GPU側メモリのみアクセス可、CPU側メモリはアクセス不可
- 引数利用可能
- __global__つき関数では値の返却は不可 (voidのみ)

例： int型配列をインクリメントするカーネル関数

```
__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++) array[i]++;
    return;
}
```

(5) 出力データの転送

- GPUカーネル関数が計算した結果(GPU側メモリに置かれている)を、CPU側メモリにコピーしたい
- (2)と同様にcudaMemcpyを用いる
- ただし、転送タイプは cudaMemcpyDeviceToHost を指定

例： 結果の配列をCPU側メモリへ転送

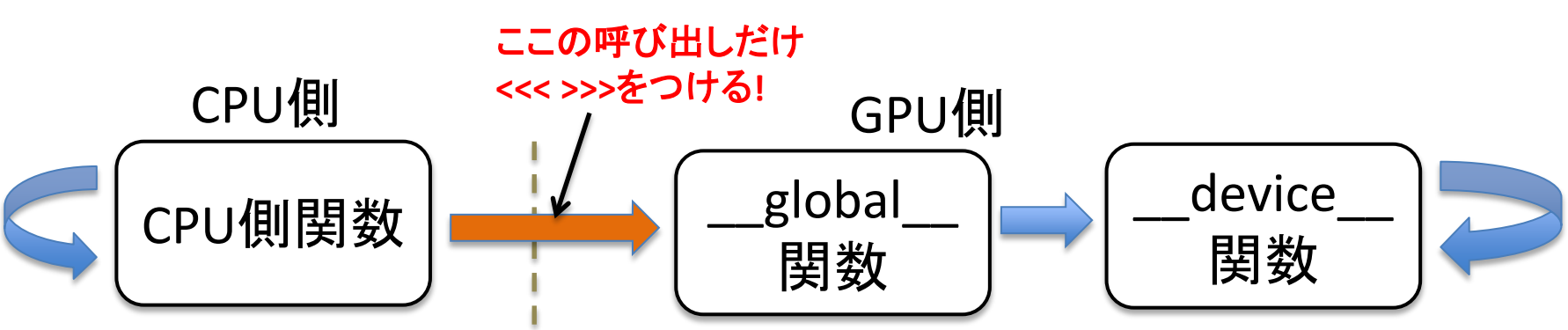
```
cudaMemcpy(arrayH, arrayD, sizeof(int)*N,
                cudaMemcpyDeviceToHost);
// (2)と比べ、第一・第二引数が入れ替わっている。
// この後であれば、arrayHに計算結果が入っているので
// CPU側でprintfなどできる
```

ここまで、inc_seq.cuサンプルプログラムを説明しました。重要なことばかりですので各自おさらいを。

二種類のGPUカーネル関数

- __global__つき関数
  - CPUから呼び出される「入口」
  - returnで値を返せない。__global__ void ～ のみ
- __device__つき関数
  - CPUからは呼べないが、returnで値を返せる

関数呼び出しのルール



- 図の矢印の方向にのみ呼び出しできる
  - たとえば、__global__関数内からCPU関数は呼べない

できること・できないこと

	CPU関数内	GPUカーネル関数内 (__global__, __device__)
If, for, whileなどの制御構文	○	○
局所変数の読み書き	○	○
大域変数の読み書き	普通の大域変数なら○	__device__つきなら○ (※1)
CPU側メモリへのアクセス	○	×
GPU側メモリへのアクセス	×	○
関数呼び出し	○ (前ページのルール参照)	○(前ページのルール参照)
Libcなどのライブラリ関数呼び出し	○	ほとんど× ただしprintf等は○
cudaMalloc, cudaMemcpy等呼び出し	○	×

※1: 大域変数(関数の外で宣言される変数)にも、「__device__」つきがある

```
int aaa[100];                  ← CPUメモリに置かれ、CPU関数だけがアクセスできる
__device__ int bbb[100];  ← GPUメモリに置かれ、GPUカーネル関数だけがアクセスできる
```

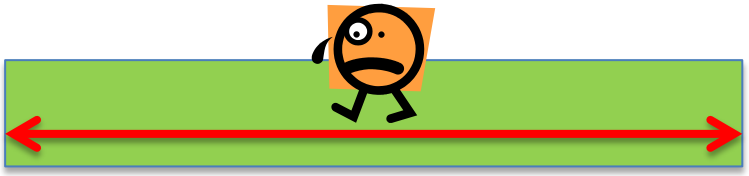
3. CUDAにおける並列化

2.では、「まずGPUを動かす」ことを説明しました。
「GPUで計算を早くする」には、これから説明する
「並列化」が必要です

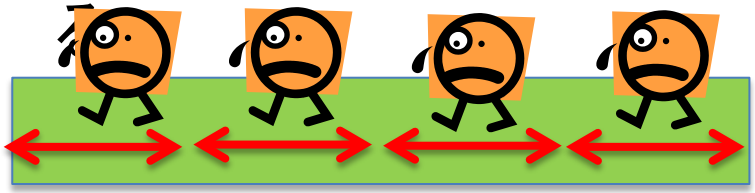
CUDAにおける並列化

- たくさんのスレッドがGPU上で並列に動作することにより、初めてGPUを有効活用できる
  - inc_seqプログラムは1スレッドしか使っていない
- データ並列性を基にした並列化が一般的
  - 例：巨大な配列があるとき、各スレッドが一部づつを分担して処理 → 高速化が期待できる

一人の小人(スレッド)が
大きな畑を耕す場合



複数の小人(スレッド)が
分担して耕すと速く終わ



CUDAにおけるスレッド(1)

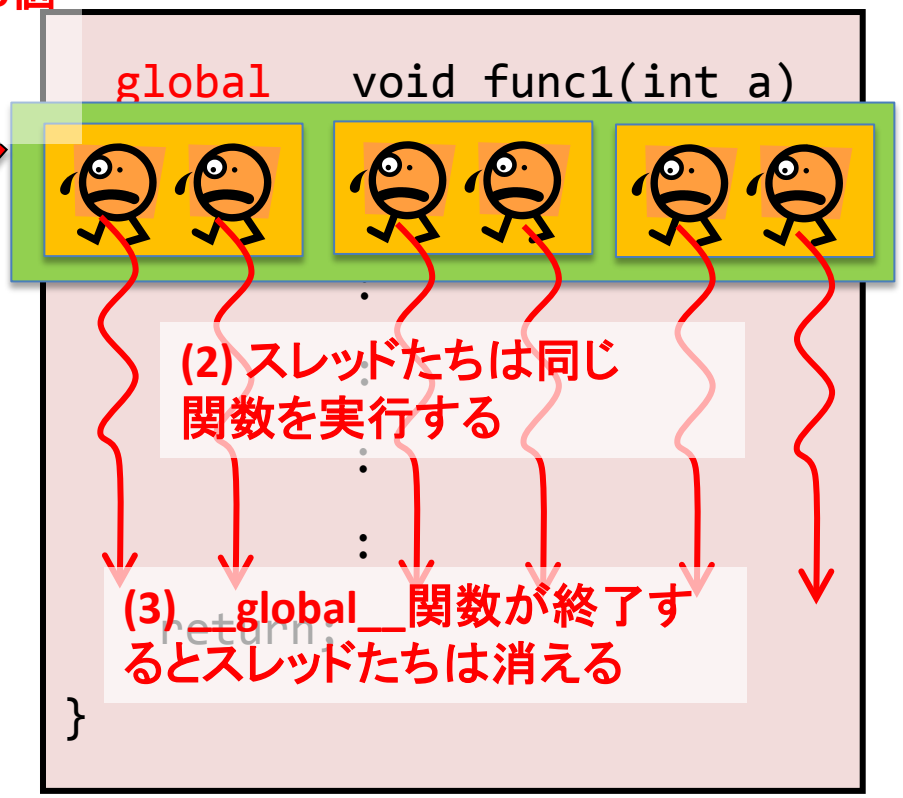
__global__関数の呼び出し時に、動作するスレッド数を指定する

CPU関数

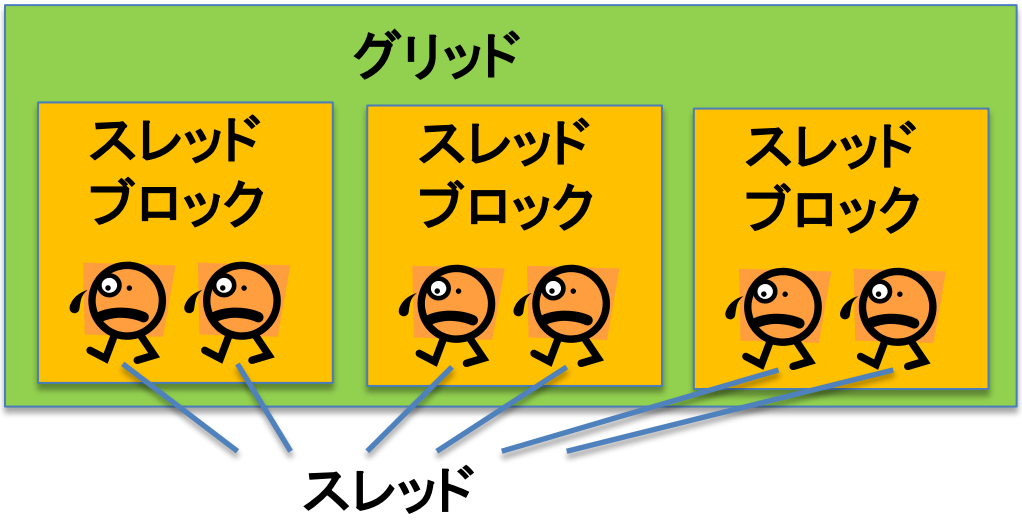
```
int main()
{
        :
    func1<<<3, 2>>>(a);
        :
        :
    func2<<<200, 100>>>(b, c);
        :
        :
    func1<<<64, 16>>>(a);
        :
        :
}
```

(1) GPU上で3x2=6個のスレッドが動き出す

GPUカーネル関数



CUDAにおけるスレッドは階層構造



- グリッド: GPUカーネル関数を実行するスレッドの集合。複数のスレッドブロックから成る
- スレッドブロックは、複数のスレッドから成る

```
func1<<<3, 2>>>(a);
```

スレッドブロックの数

スレッドブロックあたりのスレッドの数

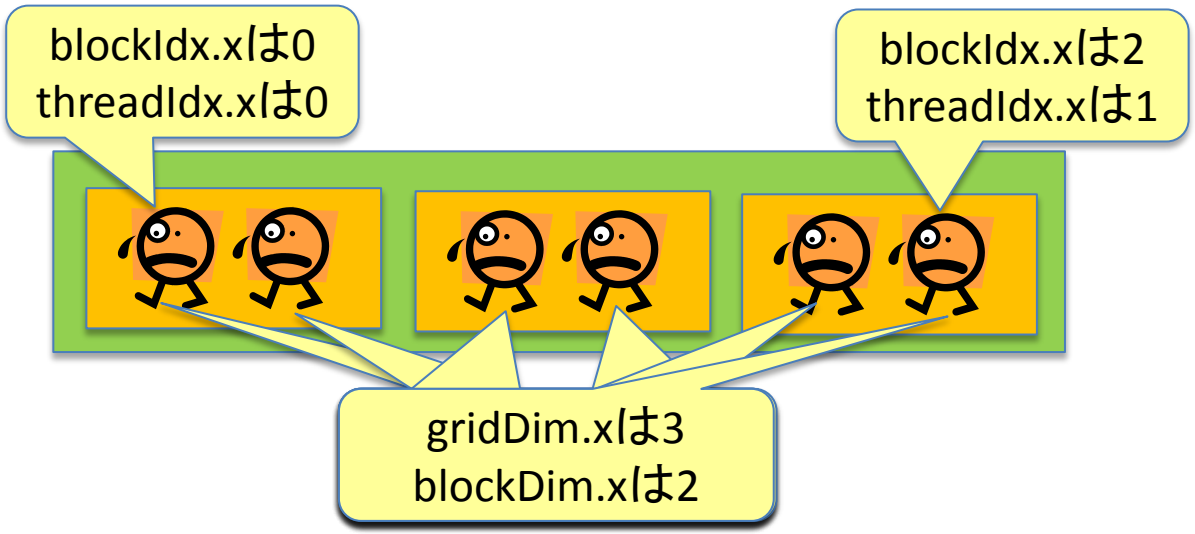
この例では3x2=6スレッドがGPU上で動作する

スレッドの番号付け

- 各スレッドに別々の仕事をさせるには、各スレッドが「自分の番号」を知る必要がある
  - 例：3組の出席番号25番

→ 特別な変数を読むと分かる

- gridDim.x: グリッドにいくつスレッドブロックがあるか
- blockIdx.x: 自分が何番のスレッドブロックにいるか
- blockDim.x: スレッドブロックにいくつスレッドがあるか
- threadIdx.x: 自分が何番のスレッドか



※ ～Idx.xの値は0からはじまる

※ ～Dimはだれが読んでも同じ値

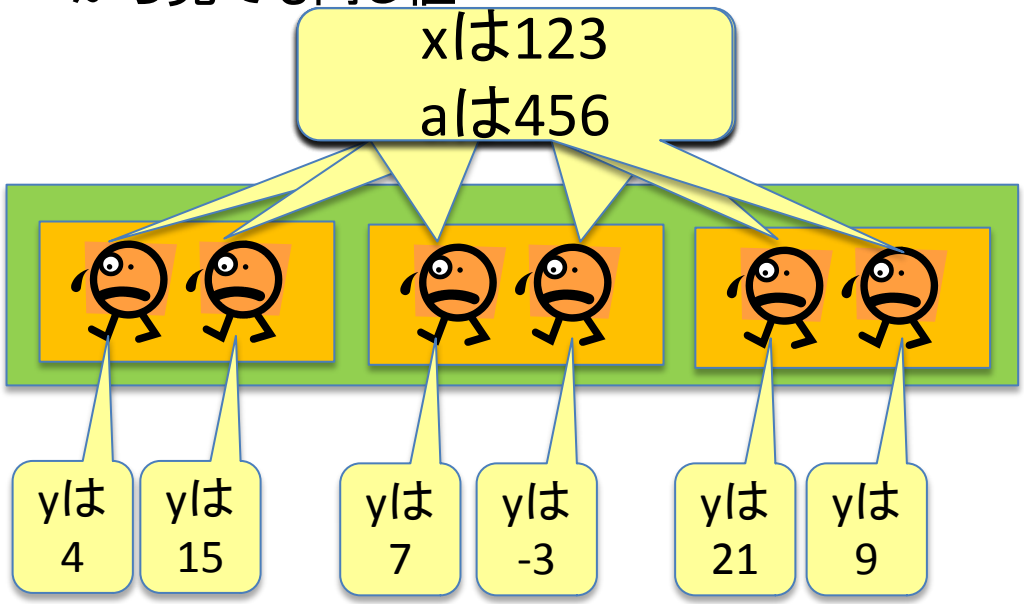
変数に関するルール

```
        :
    func1<<<3, 2>>>(456);
        :
```

```
// __device__つき大域変数
__device int x = 123;


__global__ void func1
    (int a)   // 引数
{
    int y; // 局所変数
      :
    return;
}
```

__device__大域変数：どのスレッドから見ても同じ値。誰かが書き換えると全員に伝わる

__global__関数への引数：どのスレッドから見ても同じ値



局所変数：各スレッドが別々の値を持ちうる。スレッド0番のyとスレッド1番のyは別物。

サンプルプログラムの改良

inc_parは、inc_seqと同じ計算を行うが、
N要素の計算のためにNスレッドを利用する点が違う

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define N  (32)
#define BS (8)
__global__ void inc(int *array, int len)
{
    int i = blockIdx.x * blockDim.x +
        threadIdx.x;   // iは局所変数
    array[i]++;
    return;
}

int main(int argc, char *argv[])
{
   int i;
   int arrayH[N];
   int *arrayD;
   size_t array_size;
```

```
   for (i=0; i<N; i++) arrayH[i] = i;
   printf("input: ");
   for (i=0; i<N; i++)
       printf("%d ", arrayH[i]);
   printf("¥n");

   array_size = sizeof(int) * N;
   cudaMalloc((void **)&arrayD, array_size);
   cudaMemcpy(arrayD, arrayH, array_size,
            cudaMemcpyHostToDevice);
   inc<<<N/BS, BS>>>(arrayD, N);
   cudaMemcpy(arrayH, arrayD, array_size,
               cudaMemcpyDeviceToHost);
   printf("output: ");
   for (i=0; i<N; i++)
       printf("%d ", arrayH[i]);
   printf("¥n");
   return 0;
}
```

inc_parプログラムのポイント (1)

- N要素の計算のためにNスレッドを利用

```
inc<<<N/BS, BS>>>(.....);
```

グリッドサイズ

スレッドブロックサイズ
この例では、前もってBS=8とした

ちなみに、<<<N, 1>>>や
<<<1, N>>>でも動くのだ
が非効率的である。

ちなみに、このままでは、NがBSで
割り切れないときに正しく動かない。
どう改造すればよいか？

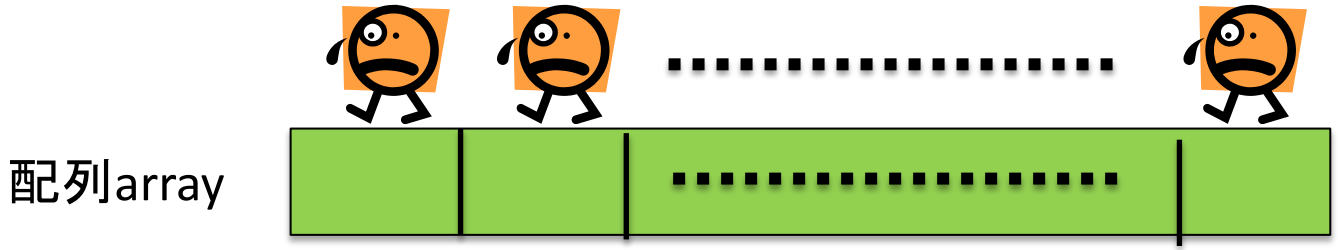
inc_parプログラムのポイント (2)

inc_parの並列化の方針

- (通算で)0番目のスレッドにarray[0]の計算をさせる
- 1番目のスレッドにarray[1]の計算

  :

- N-1番目のスレッドにarray[N-1]の計算



- 各スレッドは「自分は通算で何番目のスレッドか?」を知るために、下記を計算

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

使いまわせる
便利な式

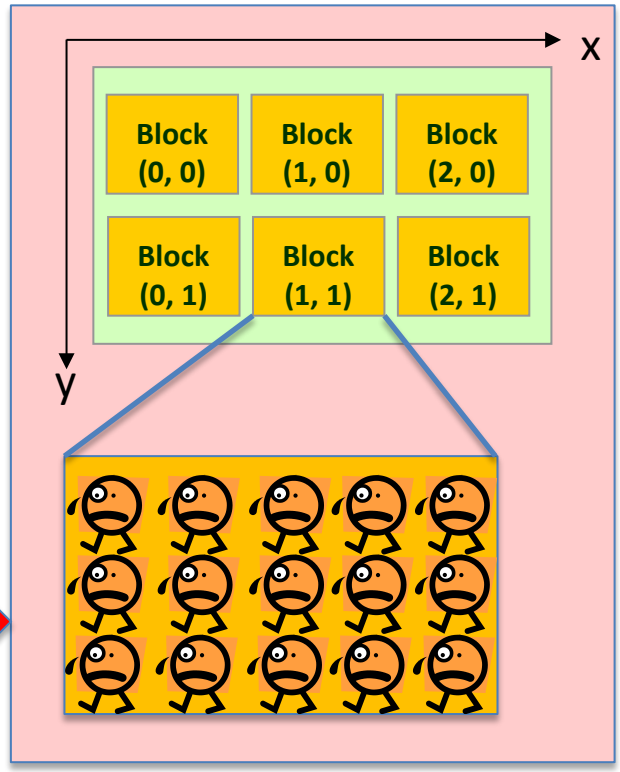
- 1スレッドは”array[i]”の1要素だけ計算 → forループは無し

スレッド数指定について補足(1)

```
func1<<< ,  >>>(a);
```

ここに書けるのは、整数だけでなく、三次元ベクトルのdim3型も

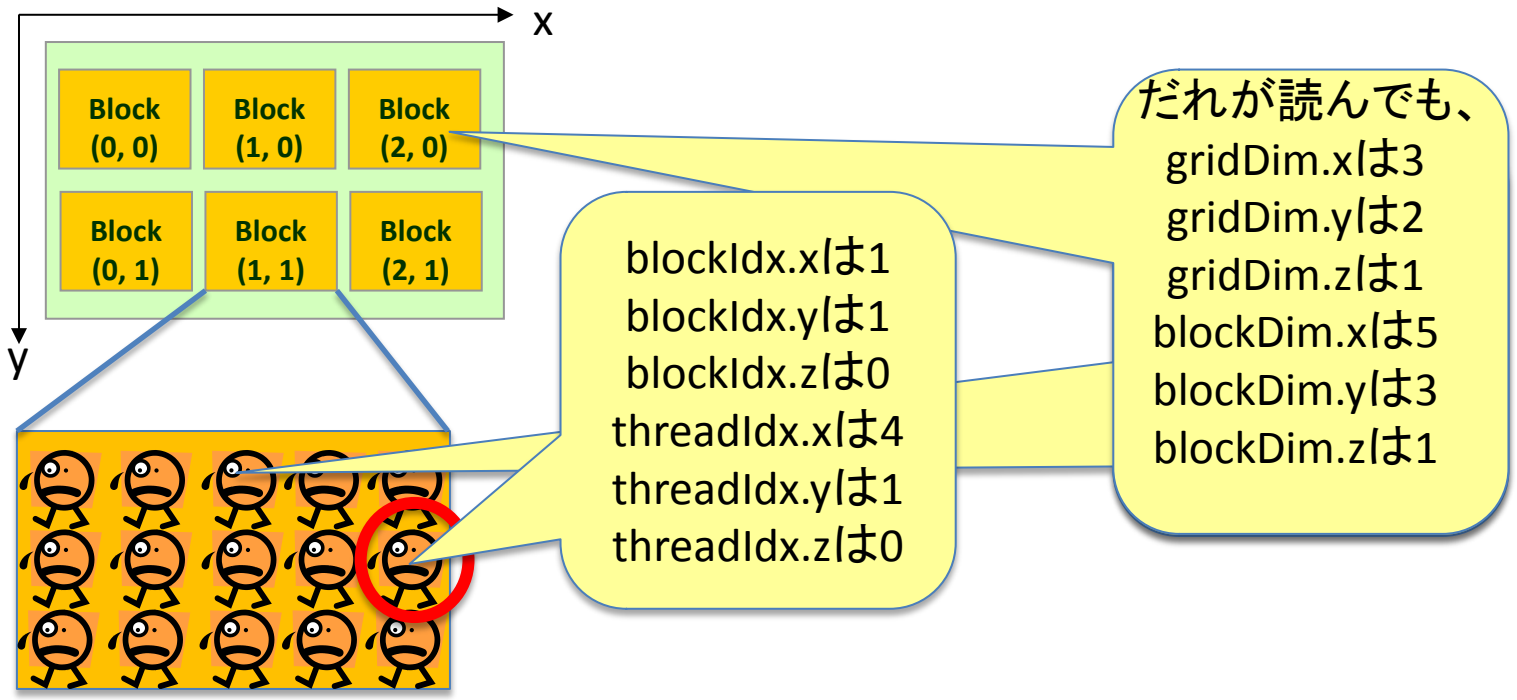
- 指定例
  - <<<100, 30>>>
    - 実は整数で100と書くとdim3(100,1,1)とみなされていた
  - <<<4, dim3(7, 9)>>>
    - dim3(7,9)はdim3(7,9,1)の意味
  - <<<dim3(100,20,5), dim3(4, 8, 4)>>>
    - この場合は(100x20x5)x(4x8x4)=128万スレッド!
  - <<<dim3(3,2), dim3(5,3)>>>



スレッド数指定について補足(2)

- ～Dim, ～Idxもdim3型
  - dim3型: x, y, zという中身を持つ構造体

<<<dim3(3,2), dim3(5,3)>>>



スレッド数の最大値

	最大値
gridDim.x	65535
gridDim.y	65535
gridDim.z	65535
blockDim.x	1024
blockDim.y	1024
blockDim.z	64
スレッドブロック中のスレッド数	1024 (※1)

※1: スレッドブロックサイズに、dim3(1024, 1, 1)やdim3(4, 4, 64)はokだが、dim3(1024, 1024, 64)はダメ。掛け算して1024超えるので。

なお、最大値はGPUの種類によって異なる。この表の数値は今回使うM2050 GPUの場合

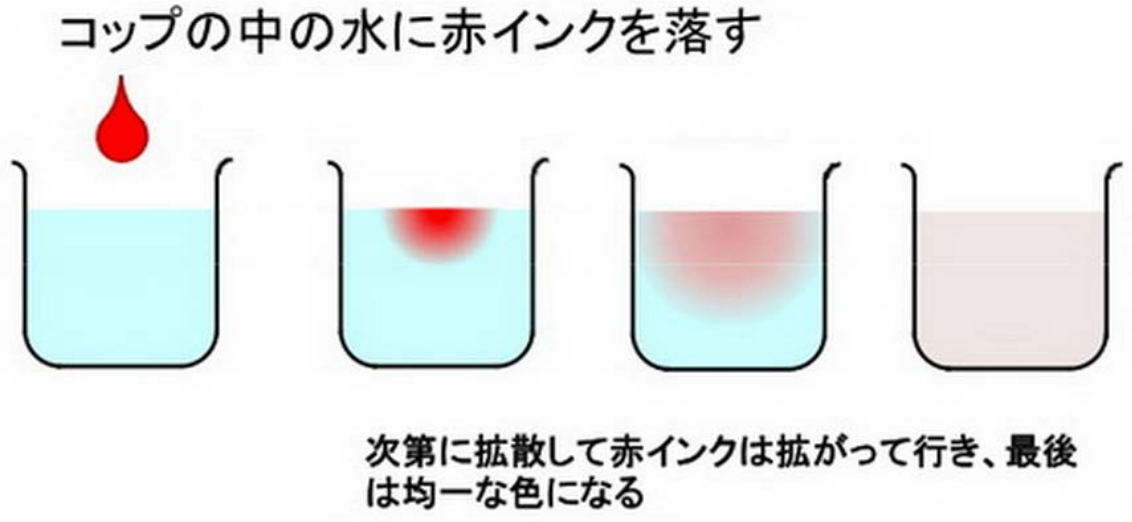
効率のよいプログラムのために

- グリッドサイズが14以上、かつスレッドブロックサイズが32以上の場合に効率的
  - M2050 GPUのハードウェアの性質が下記の通りなので
    - GPU中のSM数=14
    - SM中のCUDA core数=32
  - ぎりぎりよりも、数倍以上にしたほうが効率的な場合が多い(ベストな点はプログラム依存)

ほかにも色々効率化のポイントあり → 応用編で

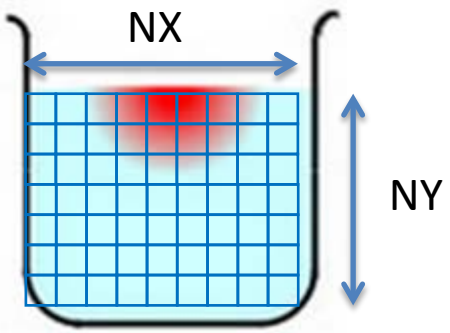
4. 少し難しいCUDAプログラムの例

1.の最後に出てきた「拡散現象シミュレーション」
について、CPU版とGPU版の違いを解説します

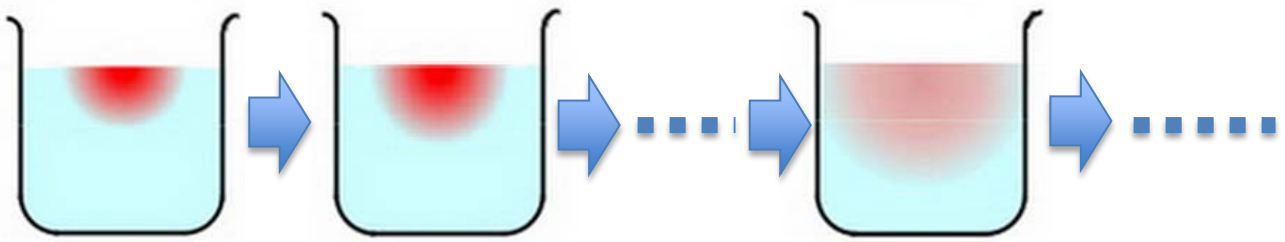


diffusion.c(CPU版)の基本

- シミュレーションしたい空間をマス目で区切り、配列で表す
  (本プログラムでは二次元配列)



- 時間を少しずつ、パラパラ漫画のように進めながら計算する



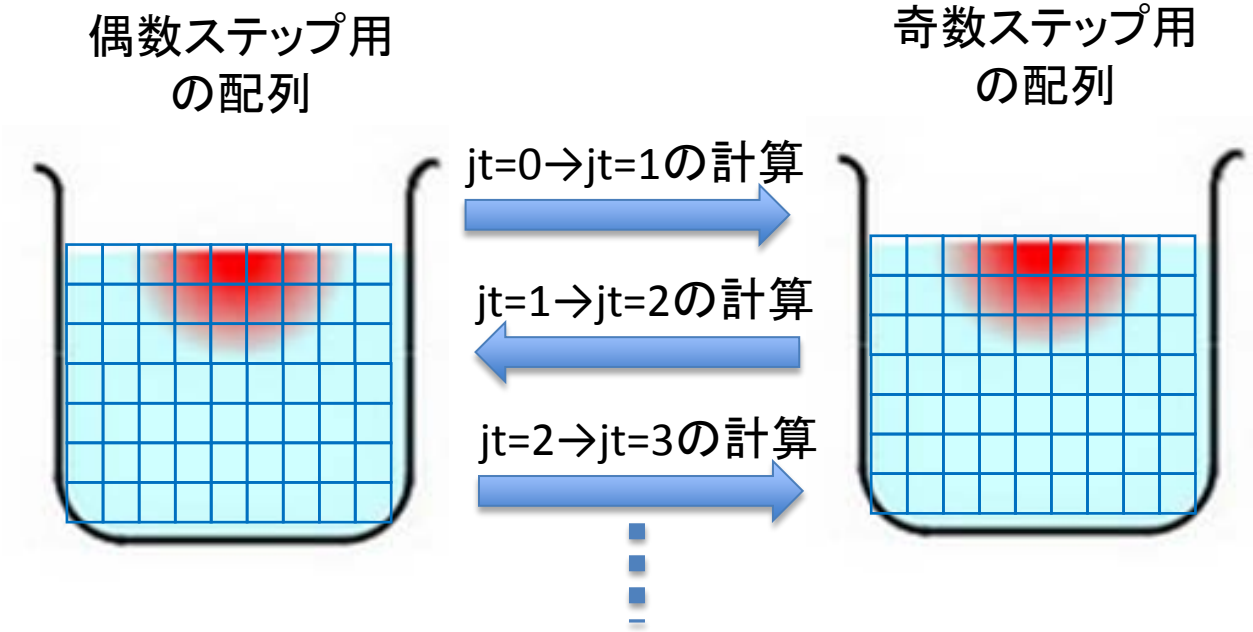
※ある時間ステップの計算には、一つ前の時間ステップの、一つ隣の(周囲の)配列の値が必要

※本プログラムでは、領域の上下左右の端の値は不変とする

ダブルバッファリング技術

- 全時間ステップの配列を覚えておくとメモリ容量を食い過ぎる

→ 二ステップ分だけ覚えておき、二つの配列(ダブルバッファ)を使いまわす



※ CPU版プログラムでは、大域変数

float data[2][NY][NX];

で表現

diffusion_gpu.cuのCUDA化の方針(1)

- 配列の全点の計算のために最も時間がかかる

⇒この部分をGPUカーネル関数にして高速化をねらおう！

```
        :
    for (jt = … ) { // 時間方向のループ
      for (jy = …) {  // y方向のループ
        for (jx = …) { // x方向のループ
          (一点の計算)
        }
      }
      ダブルバッファの切り替え
    }
        :
※このコードは非常に簡略化してあります
```

ここを、GPUカーネル関数
__global__ void calc(int from)
とした

※ calc関数がアクセスする二つの配列を、GPU側メモリに置いておく必要がある。大域変数

__device__ gdata[2][NY][NX];

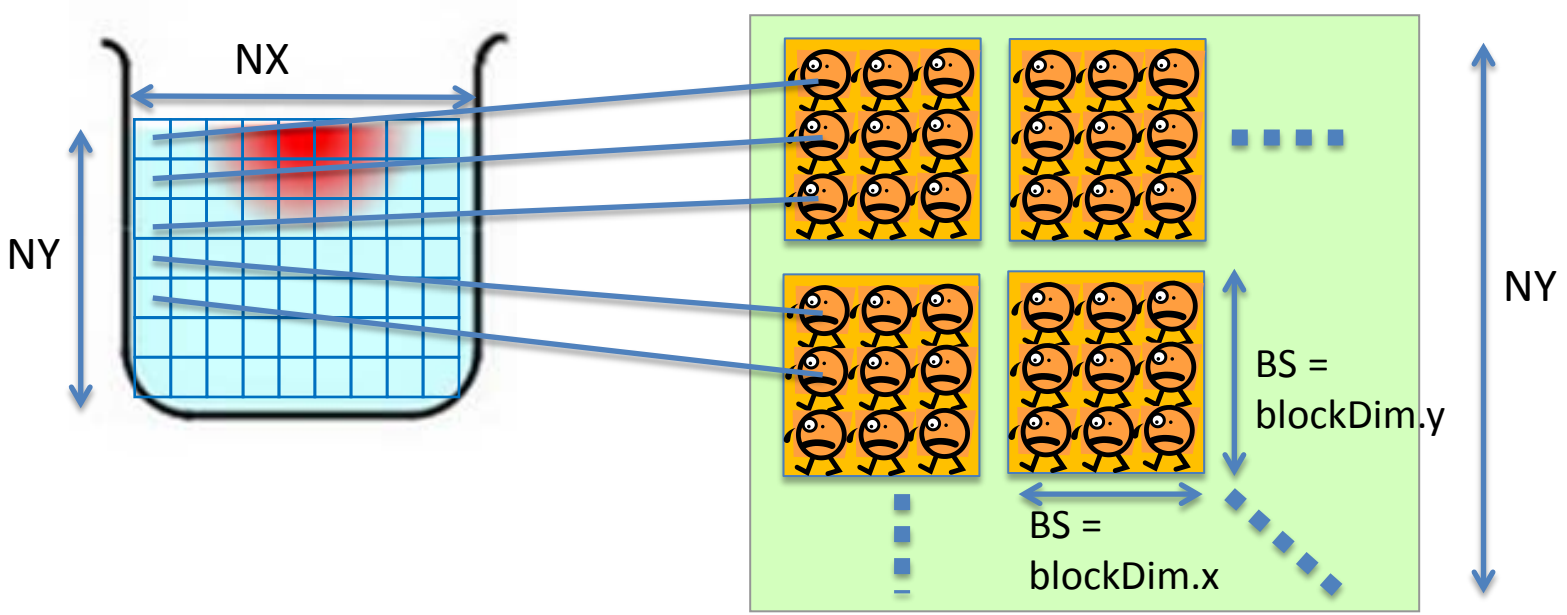
で表現した。

※ CPU側でdata配列を初期化した後、gdataへcudaMemcpyすることにした

CUDA化の方針(2)

一点を計算するために、1スレッドを起動するようにした

→　全体でNX×NY個のスレッドを使う！



次に、いくつのスレッドをスレッドブロックにまとめるかを決める必要がある。

今回はスレッドブロックサイズを16x16に固定した(上図では3x3)

```
#define BS 16
calc<<<dim3((NX+BS-1)/BS, (NY+BS-1)/BS), dim3(BS, BS)>>>(from);
```

※ NX, NYがBSで割り切れない時のために、切り上げるために(NX+BS-1)/BSとした

__device__大域変数の注意

- __device__ gdata[2][NY][NX]; はGPU側メモリに置かれ、GPUカーネル関数からのみアクセスできる

→ gdataをcudaMemcpyの宛先アドレスに指定することすらできない・・・ 

- このようなときは「cudaGetSymbolAddress」関数を使う

```
// __device__つき大域変数
__device float gdata[2][NY][NX];
```

```
{
   void *gdata_ptr;
      :
   cudaGetSymbolAddress(&gdata_ptr,
      gdata); // ←変数名を指定 (※1)
   cudaMemcpy(gdata_ptr, data,
      sizeof(float)*2*NX*NY);
   // 1ここでgdataそのものを宛先に
   //      するとコンパイルエラー
```

※ inc_par.cuプログラムのように、cudaMallocを使う場合はこの心配は不要

※1 紙配布版で、”gdata”とあったのは間違い(古い)です

基礎編のまとめ

- GPUプログラミングとTSUBAME2.0スパコンについて説明した
- CUDAプログラミング言語の基礎について説明した
  - CPU側メモリ(メインメモリ)とGPU側メモリ(デバイスメモリ)は異なることに注意。「どの変数はどちらのメモリにあるか？」を間違えると謎なエラーを引き起こす。
  - 両者のメモリの間でデータをコピーするにはcudaMemcpy
  - GPUカーネル関数を呼ぶ際には、グリッドサイズとスレッドブロックサイズ(その積がスレッド数)を指定