GPUプログラミング・基礎編

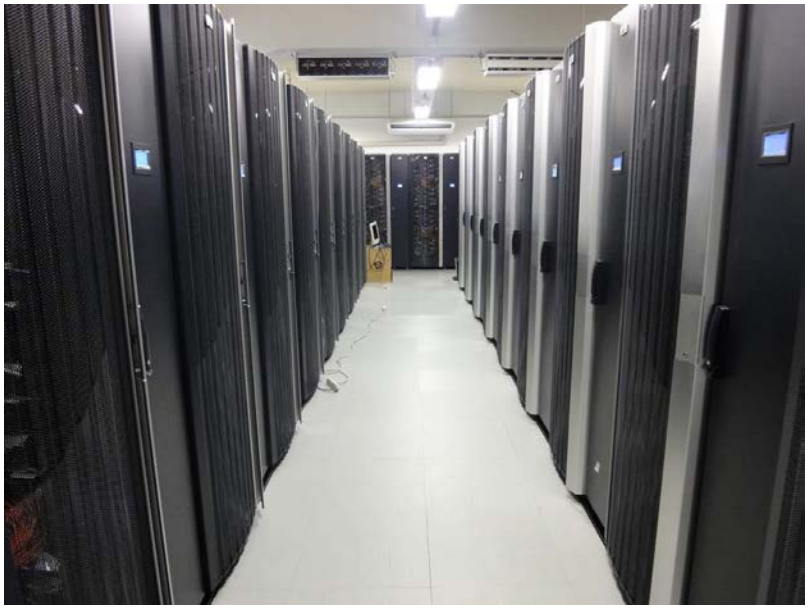
東京工業大学学術国際情報センター

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

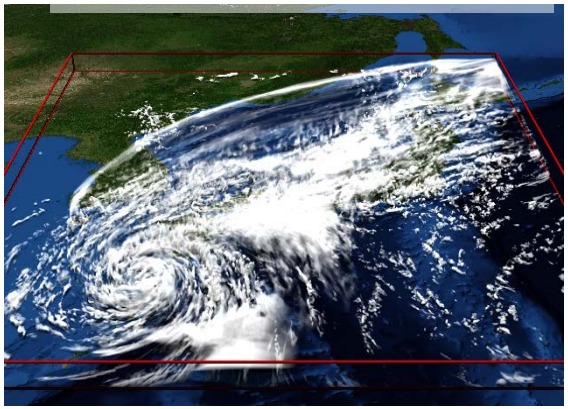
「ツバメ」は東京工業大学のシンボルマークでもある

- 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2位の計算速度性能
  - 現在、世界14位、日本3位

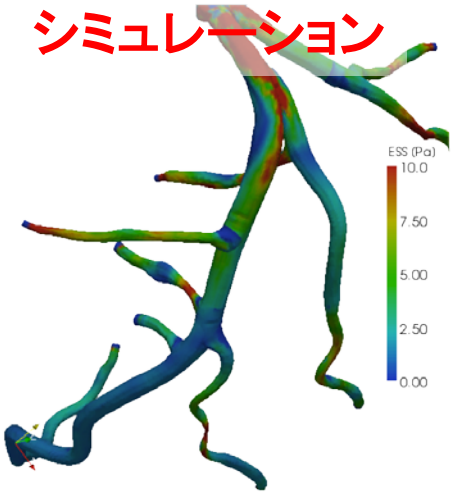
高性能の秘訣が GPUコンピューティング

TSUBAME2.0スパコン・GPUは様々な研究分野で利用されている

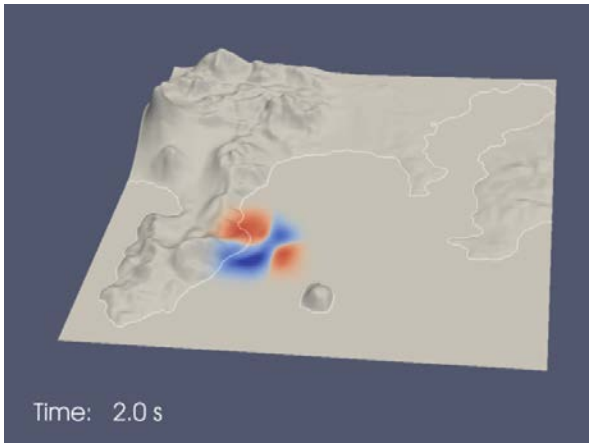
気象シミュレーション



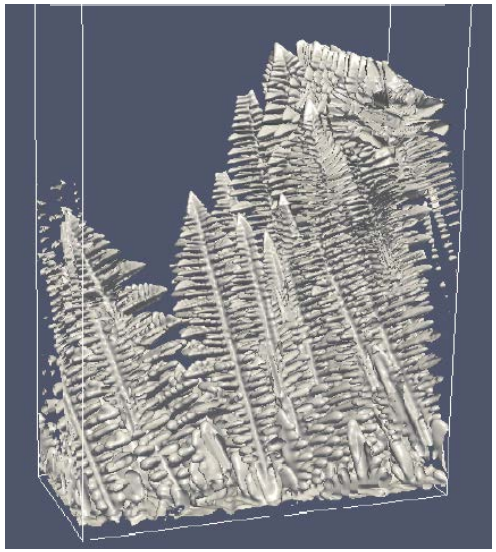
動脈血流
シミュレーション



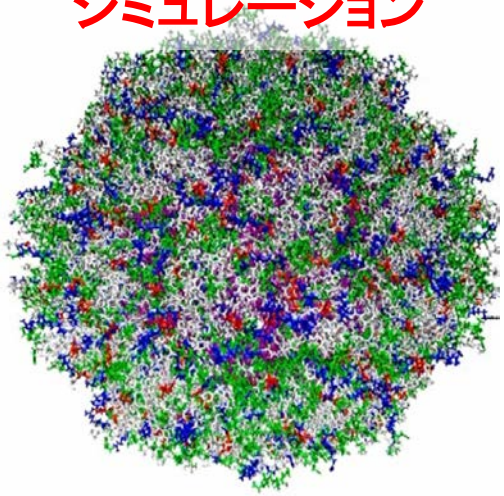
津波・防災
シミュレーション



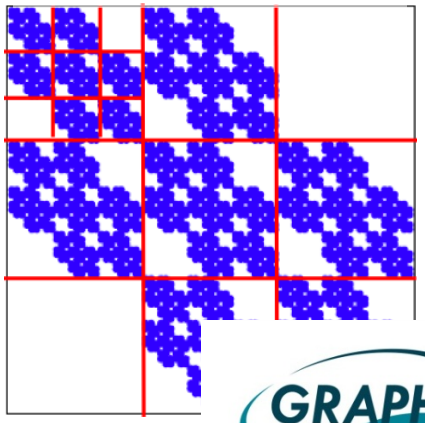
金属結晶凝固
シミュレーション



ウィルス分子
シミュレーション



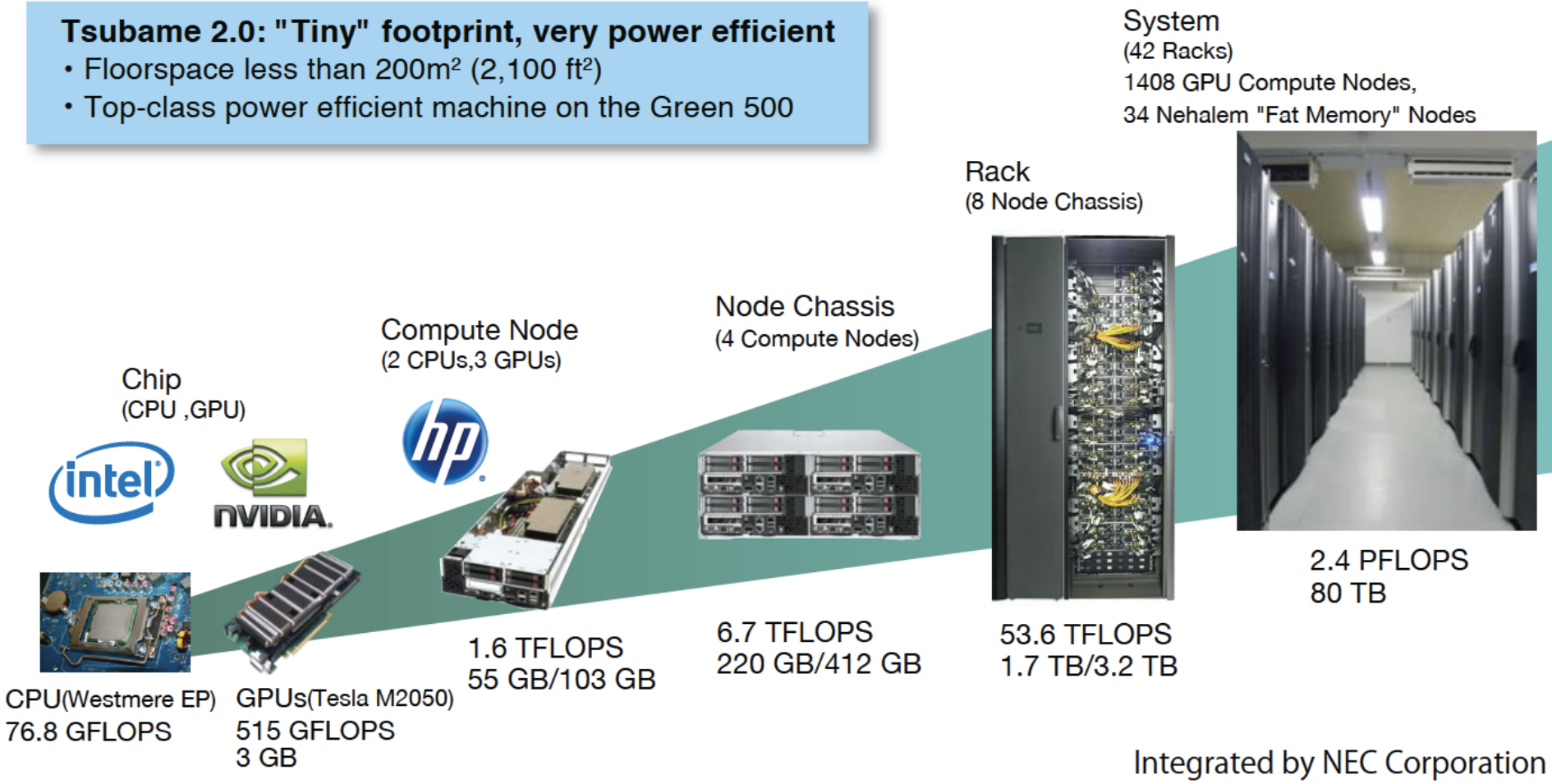
グラフ構造解析



TSUBAME 2.0 全体概要

TSUBAME2.0: A GPU-centric Green 2.4 Petaflops Supercomputer

Tsubame 2.0: "Tiny" footprint, very power efficient
- Floorspace less than 200m² (2,100 ft²)
- Top-class power efficient machine on the Green 500



Integrated by NEC Corporation

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 (1)

- 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

- 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

- 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

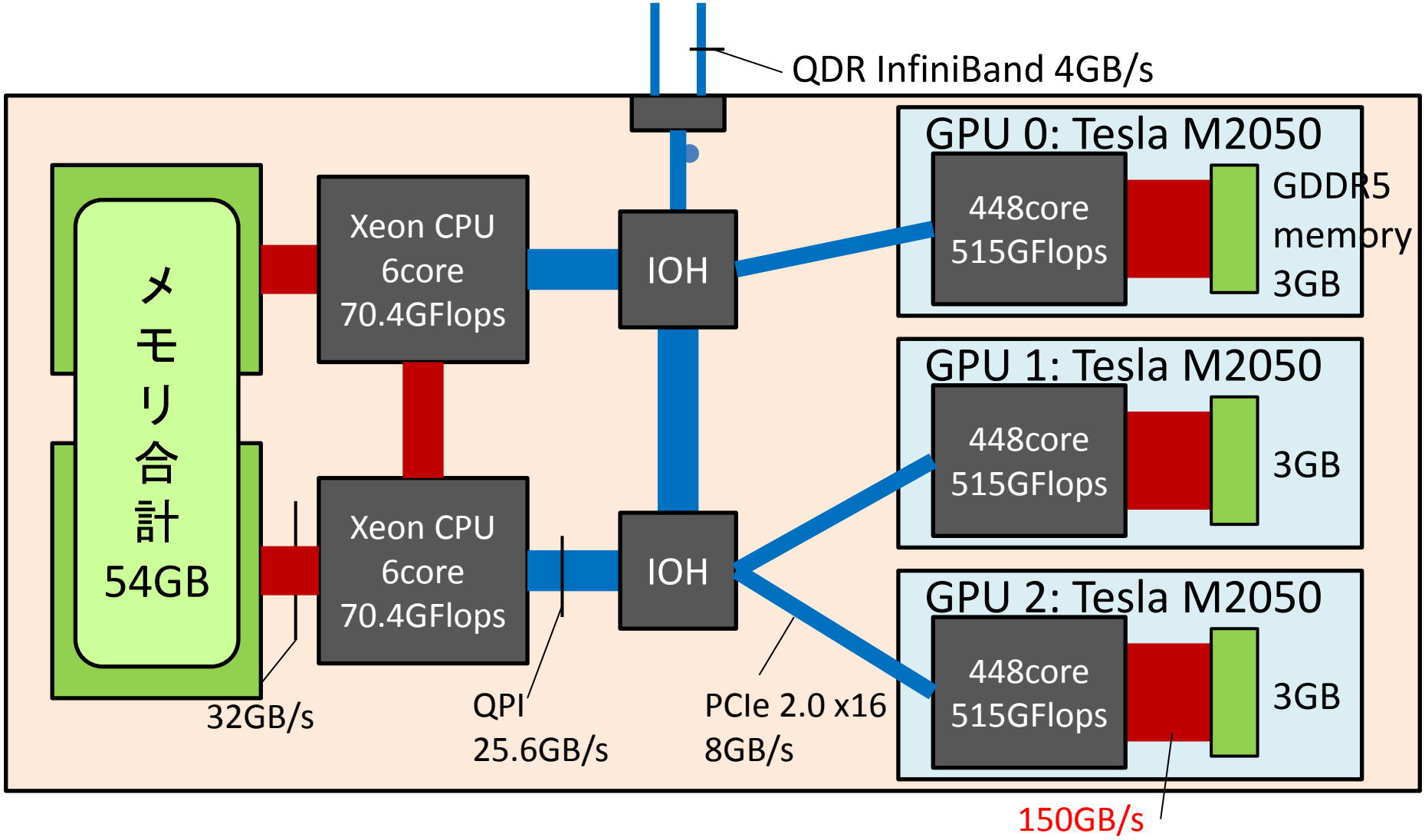
GPUの特徴 (2)

M2050 GPU 1つあたりの性能

- 計算速度: 515 GFLOPS
  - CPUは20～100GFlops程度
- メモリバンド幅: 約150 GB/s
  - CPUは10～32GB/s程度
- その他の特徴
  - ハードウェアキャッシュ
  - C++サポート
  - ECC

以前のGPUにはキャッシュメモリが無かったので、高速なプログラム作成がより大変だった

参考: 2CPUと3GPUを持つTSUBAME2.0計算ノードの構成



様々なGPUやアクセラレータ

- NVIDIA GPU
  - GeForceシリーズ：　一般のPCに搭載されているタイプで、比較的安価。パソコンショップで売っている
  - Teslaシリーズ：　GPUコンピューティング専用ハードウェア。TSUBAME2.0に搭載されているのは Tesla M2050
- AMD/ATI GPU
- 東芝・Sony・IBM Cellプロセッサ
  - プレイステーション3に搭載
- Intel MICアーキテクチャ

様々なGPU向けプログラミング言語

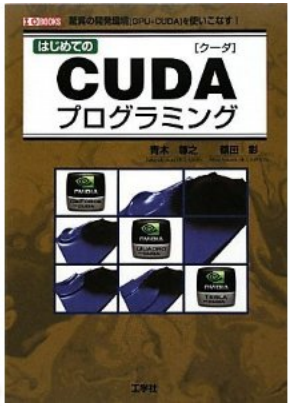
- CUDA (本講義でとりあげる)
  - NVIDIA GPU向けのプログラミング言語
- OpenCL
  - NVIDIA GPU, AMD GPU, 普通のIntelマルチコアCPUでも動く
  - ただし、CUDAよりさらに複雑な傾向
- OpenACC
  - お手軽なGPUプログラミングのために最近提案された
  - CPU用プログラムに、「ヒント」を追加

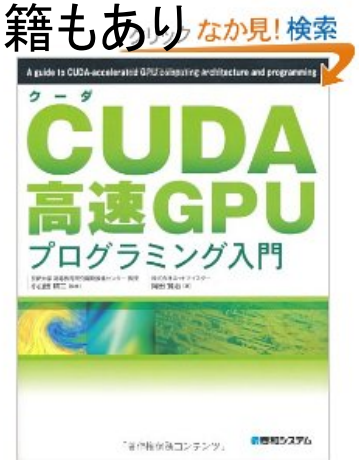
2. CUDAプログラムの流れ

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V4.1
  - 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
  - “$” はコマンドプロンプトです

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

- -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)

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;
```

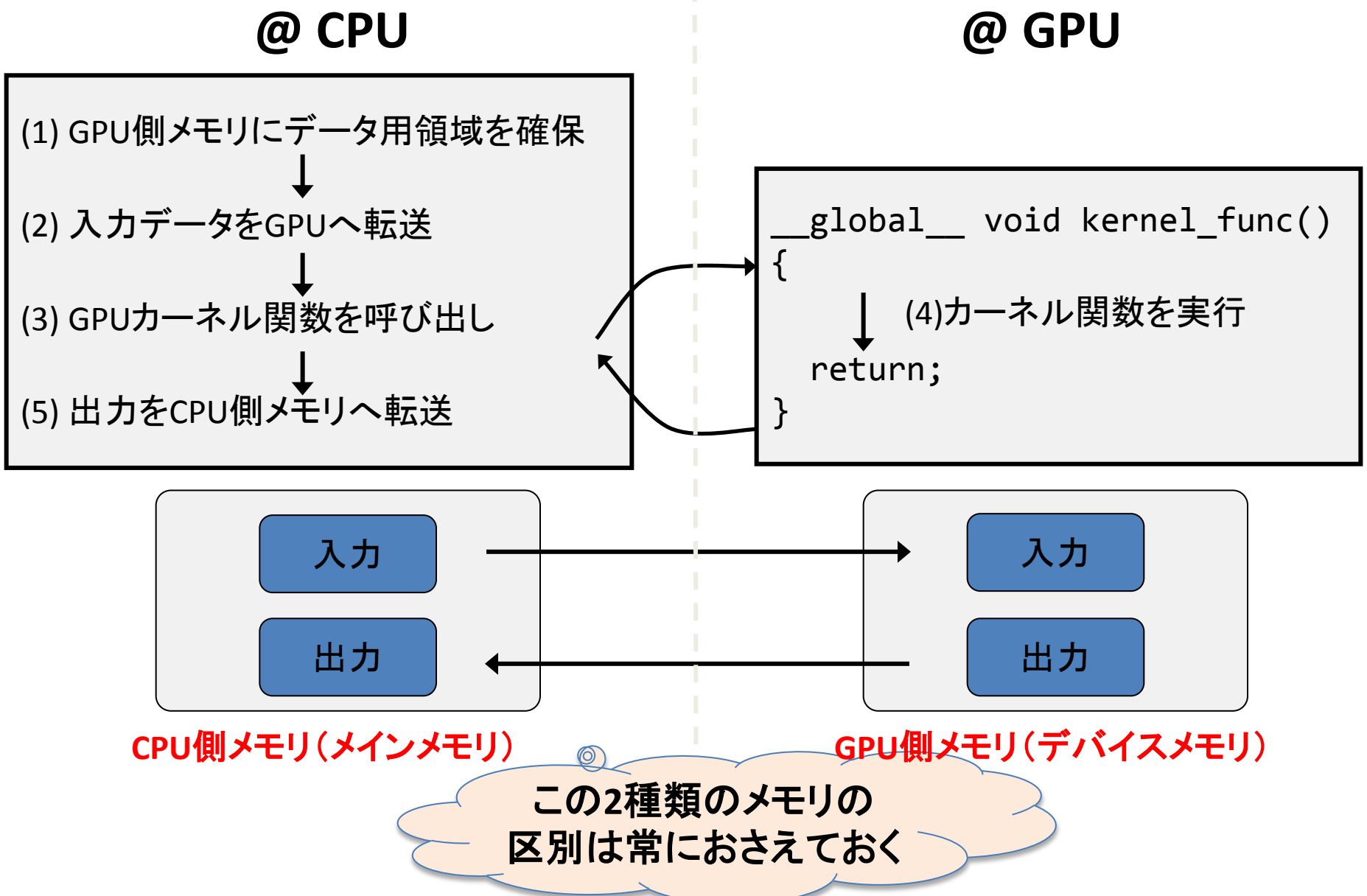
```
    for (i=0; i<N; i++) arrayH[i] = i;
    printf("input: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("¥n");

    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;
}
```

CUDAプログラム構成

ホストプログラム + GPUカーネル関数

- ホストプログラム
  - CPU上で実行されるプログラム
  - ほぼ通常のC言語。main関数から処理がはじまる
  - GPUに対してデータ転送、GPUカーネル関数呼び出しを実行
- GPUカーネル関数
  - GPU上で実行される関数 (サンプルではinc関数)
  - ホストプログラムから呼び出されて実行
  - (単にカーネル関数と呼ぶ場合も)

典型的な制御とデータの流れ



(1) @CPU: GPU側メモリ領域確保

- `cudaMalloc(void **devpp, size_t count)`
  - GPU側メモリ(*デバイスメモリ*、*グローバルメモリ*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書き込まれる
  - count: 領域のサイズ
- `cudaFree(void *devp)`
  - 指定領域を開放

例: 長さ1024のintの配列を確保

```
#define N (1024)
int *arrayD;
cudaMalloc((void **)&arrayD, sizeof(int) * N);
// arrayD has the address of allocated device memory
```

(2) @CPU: 入力データ転送

- cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - 先にcudaMallocで確保した領域に指定したCPU側メモリのデータをコピー
  - dst: 転送先デバイスメモリ
  - src: 転送元CPUメモリ
  - kind: 転送タイプを指定する定数。ここではcudaMemcpyHostToDeviceを与える

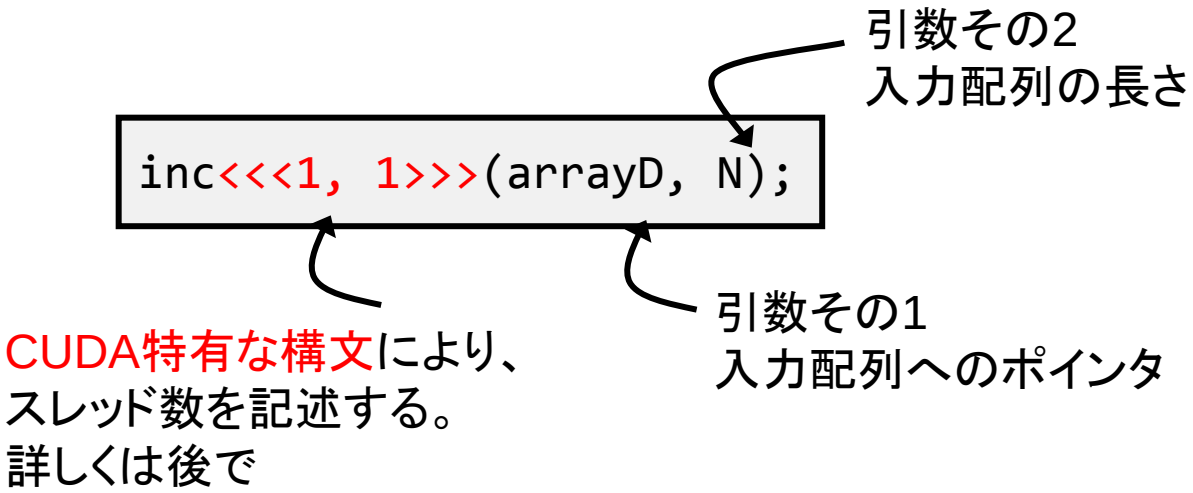
例: 先に確保した領域へCPU上のデータarrayHを転送

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

(3) @CPU: GPUカーネルの呼び出し

- `kernel_func<<<grid_dim, block_dim>>>(kernel_param1, …);`
  - `kernel_func`: カーネル関数名
  - `kernel_param`: カーネル関数の引数

例： カーネル関数 “inc” を呼び出し



(4) @GPU: カーネル関数

- GPU上で実行される関数
  - __global__というキーワードをつける

  注:「global」の前後にはアンダーバー2つずつ
- 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
- 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) @CPU: 結果の返却

- 入力転送と同様に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);
```

カーネル関数内でできること・できないこと

- if, for, whileなどの制御構文はok
- GPU側メモリのアクセスはok、CPU側メモリのアクセスは不可
  - inc_seqサンプルで、arrayDと間違ってarrayHをカーネル関数に渡してしまうとバグ!! (何が起こるか分からない)
- ファイルアクセスなどは不可
  - printfは例外的にokなので、デバグに役立つ
- 関数呼び出しは、「__device__つき関数」に対してならok

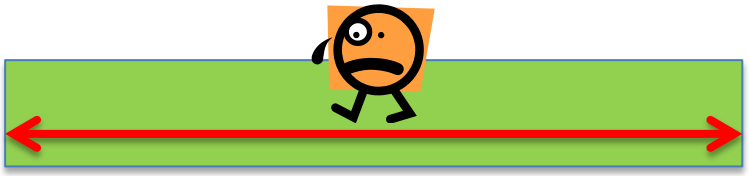
- 上図の矢印の方向にのみ呼び出しできる
  - GPU内からCPU関数は呼べない
- __device__つき関数は、返り値を返せるので便利

3. CUDAにおける並列化

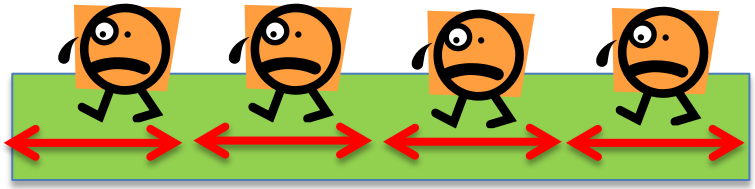
CUDAにおける並列化

- 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

一人の小人が大きな畑を耕す場合



複数の小人が分担して耕すと速く終わる



CUDAにおけるスレッド(1)

- CUDAでのスレッドは階層構造になっている
  - グリッドは、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
- カーネル関数呼び出し時にスレッド数を二段階で指定

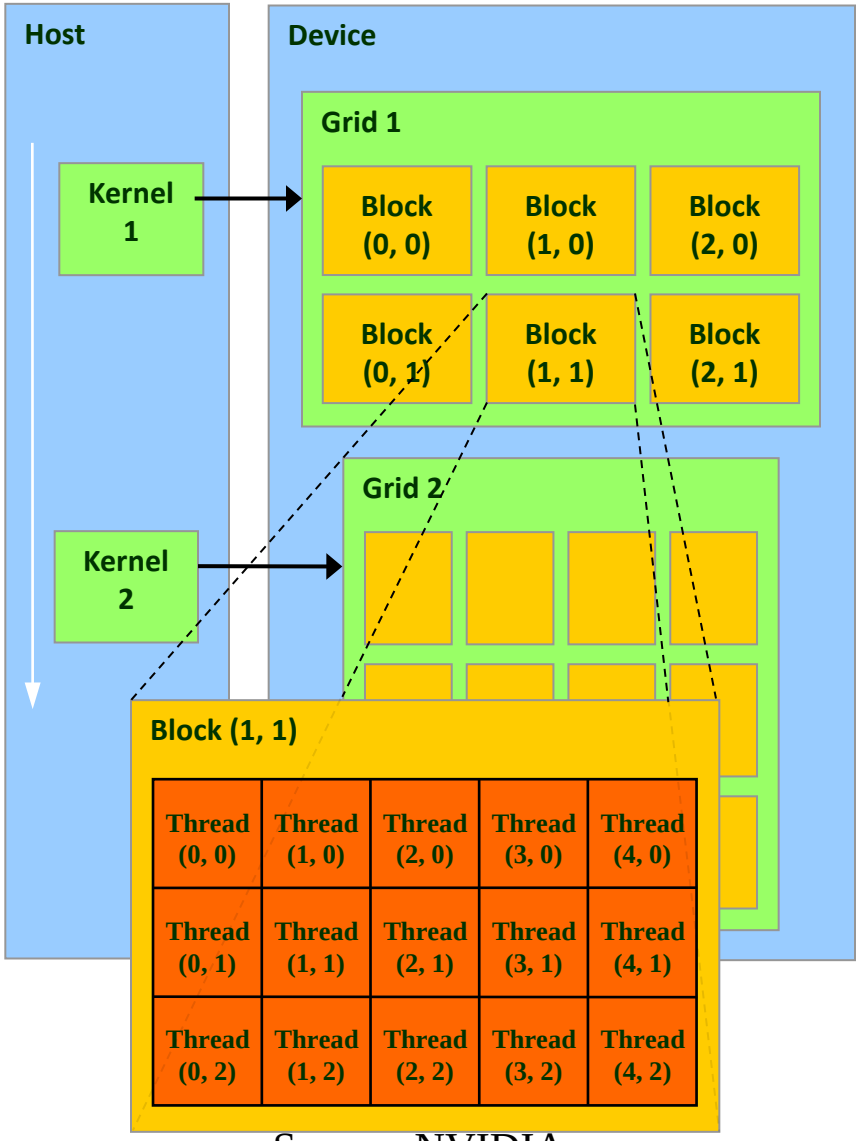
```
kernel_func<<<100, 30>>>(a, b, c);
```

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、100x30=3000個のスレッドがkernel_funcを　並列に実行する

CUDAでのスレッド(2)



Source: NVIDIA

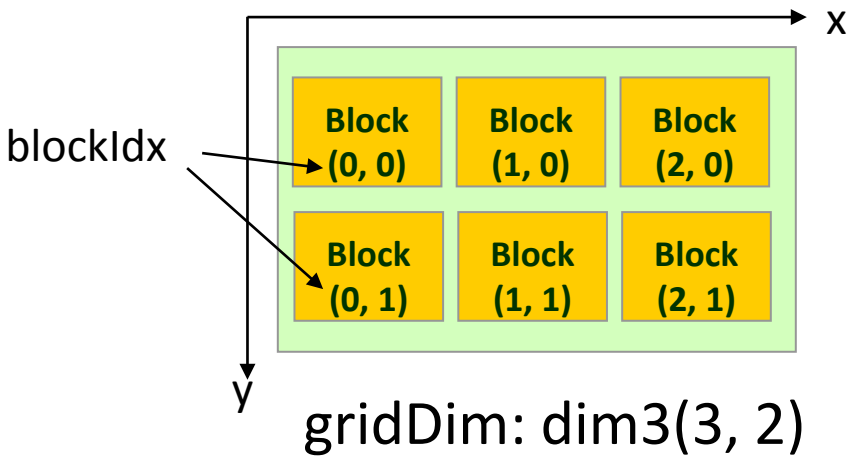
- スレッドブロック数およびスレッド数はそれぞれが
  - int型整数
  - 三次元のdim3型 (CUDA特有)

  のどちらか
- 指定例
  - <<<100, 30>>>
  - <<<dim3(100,20,5), dim3(4, 8, 4)>>>
  - <<<4, dim3(20, 9)>>>

  なお、dim3(100,1,1)と100は同じ意味となる

グリッドとスレッドブロック

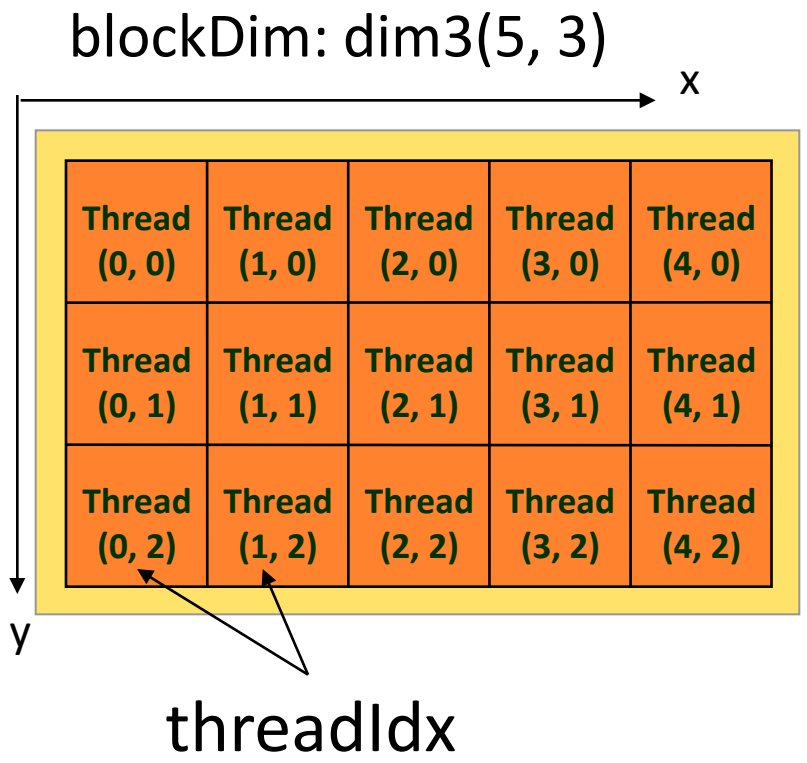
- 1次元、2次元、3次元でグリッドのサイズを指定可
- 各スレッドが「自分は誰か？」を知るために、以下を利用可能
  - dim3 gridDim
    - グリッドサイズ
  - dim3 blockIdx
    - グリッド内のブロックのインデックス、つまり自分が何番目のブロックに属するか。(0からはじまる)



- 1次元目は gridDim.x, blockIdx.xとして利用
- 同様に、2次元目は～.y, 3次元目は～.z
- 最大サイズ（M2050 GPUでは)
  - 65535 x 65535 x 65535

スレッドブロックとスレッド

- 1次元、2次元、3次元でスレッドブロックのサイズを指定可
- 各スレッドが「自分は誰か？」を知るために、以下を利用可能
  - dim3 blockDim
    - スレッドブロックサイズ
  - dim3 threadIdx
    - ブロック内のスレッドインデックス、つまりブロック内で自分が何番目のスレッドか。(0からはじまる)
- 最大サイズの制限有り
  - M2050 GPU では xは1024まで、yは1024まで、zは64まで
  - 全体で1024まで



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;
    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？

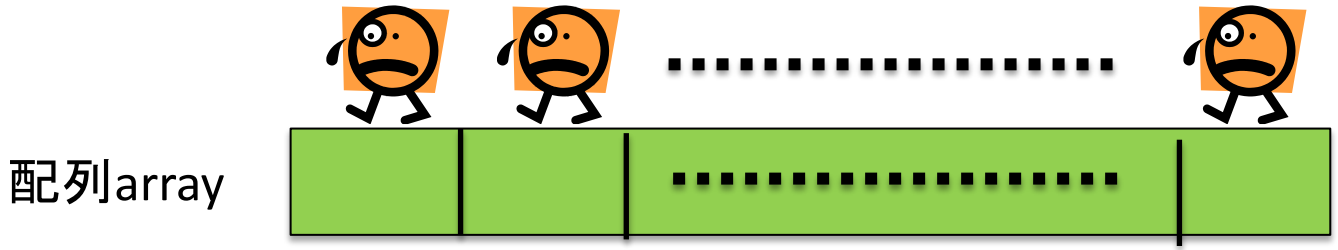
inc_parプログラムのポイント (2)

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

変数・メモリに関するルール

- カーネル関数内で宣言される変数は、各スレッド独自の値を持つ
  - あるスレッドではi=0, 別のスレッドではi=1・・・
- カーネル関数に与えられた引数は、全スレッド同じ値
  - inc_parプログラムでは、arrayポインタとlen
- 全スレッドはGPU側メモリを共有しており、読み書きできる
  - ただし、複数スレッドが同じ場所に書き込むとぐちゃぐちゃ(race condition)になるので注意
  - 同じ場所を読み込むのはok

4. GPUの計算速度の威力

少し高度な例：行列積演算 (1)

- 行列積演算サンプルプログラム

  サイズ1024x1024の行列A, B, Cがあるとき、C=A × Bを計算する

  いくつかのバージョンを比較：

  – matmul_cpu.c
    - CPUで計算

    → **約8.3秒** (gcc –O2でコンパイルした場合)

  – matmul_seq.cu
    - GPUの1スレッドで計算

    → **約200秒。**CPUより遅くなってしまった

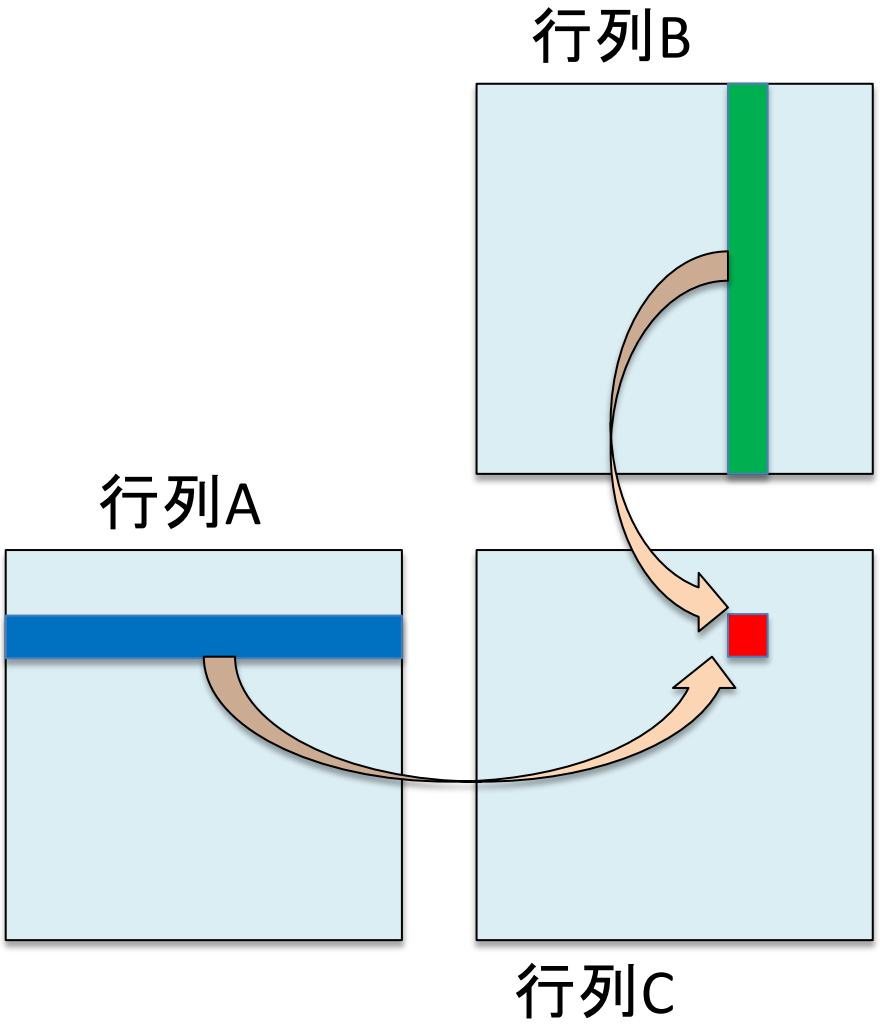
  

  – matmul_par.cu
    - GPUの複数スレッドで計算

    → **約0.027秒。**けた違いに速い!!

行列積演算(2): cpu版/seq版



行列Cの要素$C_{i,j}$を求めるには

- Aの第i行全体
- Bの第j列全体

の内積計算を行う

→ このためにforループ

C全体を計算するためには、
三重のforループ

行列積演算 (3): par版

- matmul_parでは、1024x1024個のスレッドを用い、1スレッドがCの1要素を計算

```
matmul<<<dim3(N / BS, L / BS), dim3(BS, BS)>>>
        (Ad, Bd, Cd, L, M, N);
```

ここで、L=M=N=1024。
BSは前もって適当に決めた数(16)

  - カーネル関数は内積のための一重forループ
  - グリッドサイズ・ブロックサイズとも二次元で指定

ちなみに、更なる並列化のために、Cの1要素の計算を複数スレッドで行うのは容易ではない (合計の計算時にスレッド間の同期が必要)

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では
    - GPU中のSM数=14
    - SM中のCUDA core数=32 なので
  - 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

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)は異なるため、cudaMemcpyでデータをコピーする
  - 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