GPUプログラミング・基礎編

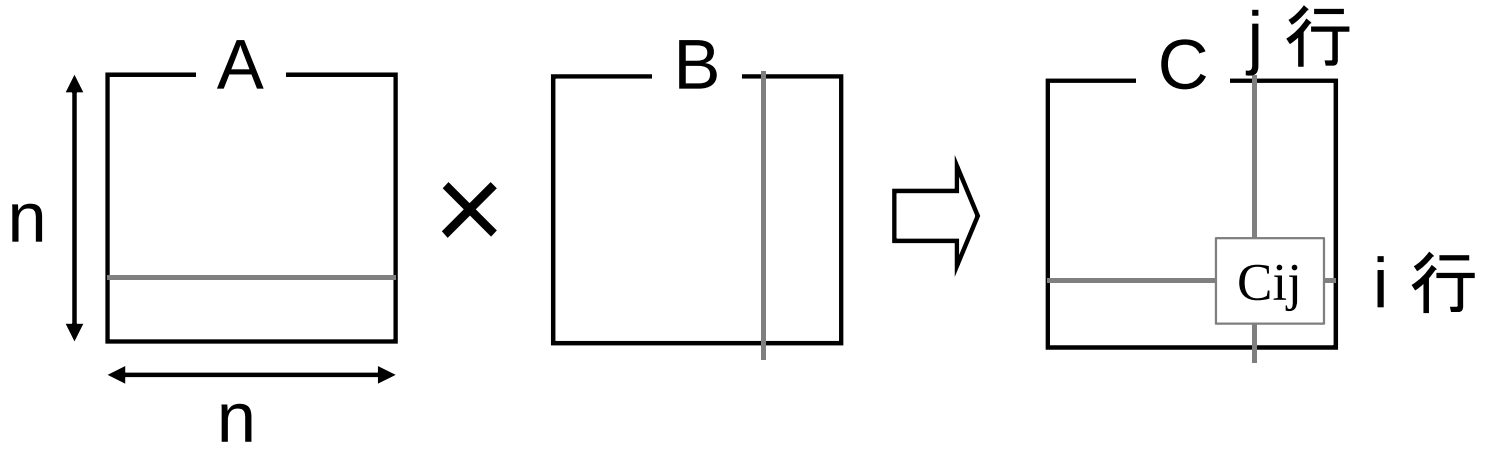
東京工業大学学術国際情報センター

この資料は，本選参加者に対する事前資料として配布するものです．できれば目を通しておいて下さい．ただし，わからなくても心配無用です．本選初日に，この資料を使って説明会をしますので．もちろん，CUDA プログラミングを事前に練習しておく必要もありません．

GPUプログラミング入門：行列積計算を例題に

行列積とは：行列×行列の計算

ここではｎｘｎの正方行列を対象とする



$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

計算の大まかな流れ：

```
for (i = 0; i < n; i++) {  // Cの第i行に注目
  for (j = 0; j < n; j++) {  // Cの第j列に注目
    for (k = 0; k < n; k++) {  // 総和計算のループ
      c[i][j] += a[i][k]*b[k][j];
    } } }
```

GPUの計算速度の威力

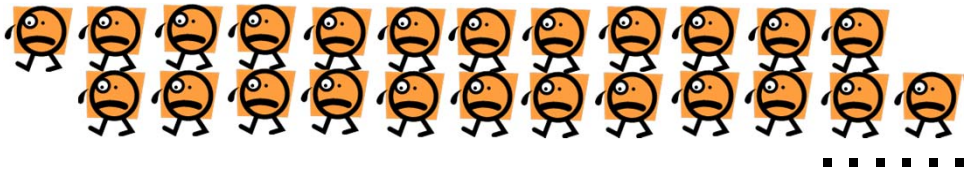
n=2048の場合の計算時間をTSUBAME2.5上で測ってみた

- CPU版 (C言語で書かれたmm.c)
  → 10.1秒かかった
- GPU版 (「CUDA」で書かれたmm_gpu.cu)
  → 0.32秒。CPU版より約31倍も速い！！

なぜ？

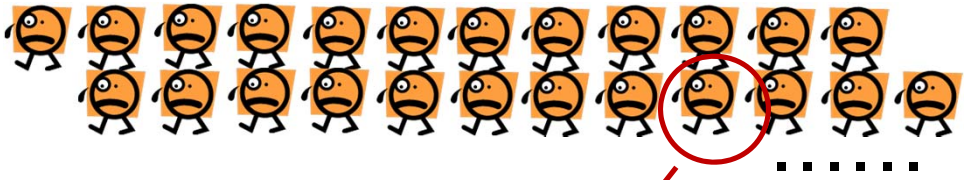
GPU は1つの中に 2688 個もの
演算装置（計算コア）がいる

ただ，2688 倍
はちょっと無理
1 つ 1 つは
CPU よりかなり劣る

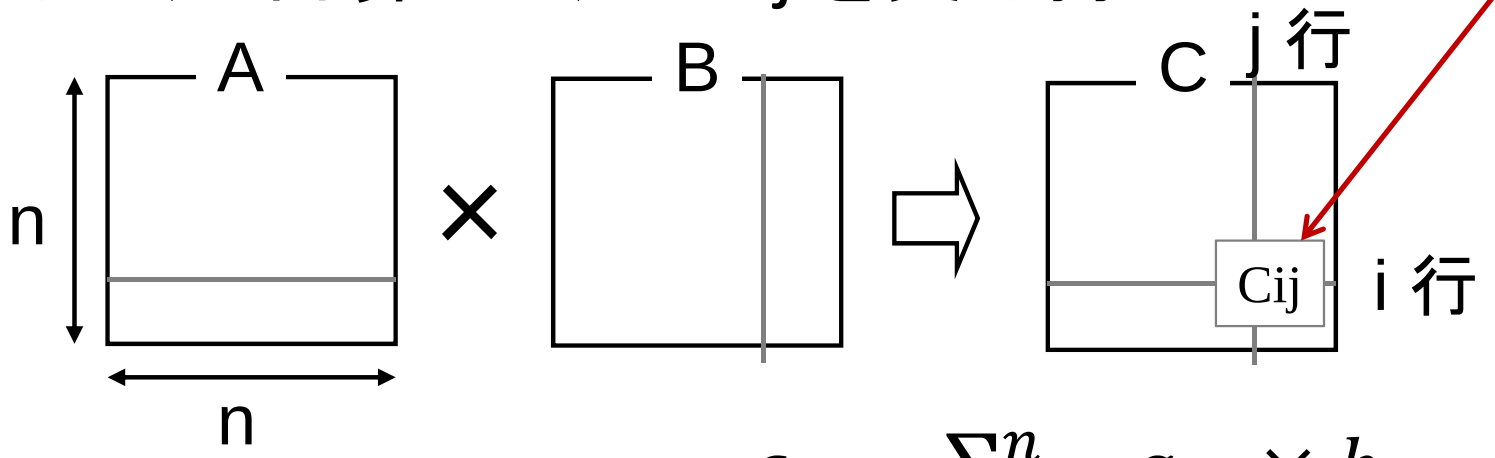


これらが上手く分業して
計算できれば，すごく速くなる

上手な分業化が必要！



各スレッド（≒計算コア）が Cij を受け持つと．．．



$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

計算の大まかな流れ：

```
for (i = 0; i < n; i++) {  // Cの第i行に注目
  for (j = 0; j < n; j++) {  // Cの第j列に注目
    for (k = 0; k < n; k++) {  // 総和計算のループ
      c[i][j] += a[i][k]*b[k][j];
    } } }
```

各スレッドでの計算は
これだけになる

これが実際のプログラム

正確には mm_gpu.cu という
プログラム内の GPU 用関数

```
__global__ void mm_gpu(double *A, double *B, double *C, int n)
{
    int i, j, k;
    i = blockIdx.y * blockDim.y + threadIdx.y;
    j = blockIdx.x * blockDim.x + threadIdx.x;
    // 自分の背番号から担当する (i,j) を決める
    if (i >= n || j >= n) return; // 行列からはみ出す部分は計算しない

    for (k = 0; k < n; k++) {  // 総和を計算する部分
        C[i*n+j] += A[i*n+k] * B[k*n+j]; // C[i][k] += A[i][k] * B[k][j]に相当
    }
}
```

まずは，これを理解しよう！

1. CUDA プログラムの概要

CUDA プログラムとは，CPU+GPU を動かすためのプログラム．C 言語の拡張版と思えばよい．

以下が，行列積のための CUDA プログラムmm_gpu.cu である．

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

__global__ void mm_gpu(double *A, double *B, double *C, int n)
{
   int i, j, k;
   i = blockIdx.y * blockDim.y + threadIdx.y;
   j = blockIdx.x * blockDim.x + threadIdx.x;
   // 自分の背番号から担当する (i,j) を決める
   if (i >= n || j >= n) return; // 行列からはみ出す部分は計算しない

   for (k = 0; k < n; k++) {  // 総和を計算する部分
        C[i*n+j] += A[i*n+k] * B[k*n+j]; // C[i][k] += A[i][k] * B[k][j]に相当
   }
}

int main(int argc, char *argv[])
{
   int i, j, n;
   double *A, *B, *C;
   double *DA, *DA, *DC;

   n = atoi(argv[1]);  // 行列の大きさ
   // A, B, Cのためにホストメモリを確保
   A = (double *)malloc(sizeof(double)*n*n);
   B = (double *)malloc(sizeof(double)*n*n);
   C = (double *)malloc(sizeof(double)*n*n);

   // A, Bの内容を設定し、Cをゼロクリア(略)

   // A, B, Cのためにデバイスメモリを確保
   cudaMalloc((void**)&DA, sizeof(double)*n*n);
   cudaMalloc((void**)&DB, sizeof(double)*n*n);
   cudaMalloc((void**)&DC, sizeof(double)*n*n);

   // A, B, Cの内容を、ホストメモリからデバイスメモリへコピー
   cudaMemcpy(DA, A, sizeof(double)*n*n, cudaMemcpyHostToDevice);
   cudaMemcpy(DB, B, sizeof(double)*n*n, cudaMemcpyHostToDevice);
   cudaMemcpy(DC, C, sizeof(double)*n*n, cudaMemcpyHostToDevice);

   // GPUカーネル関数を呼び出す!! 約n*n個のスレッドを使う
   mm_gpu<<<dim3((n+BS-1)/BS, ((n+BS-1)/BS)), dim3(BS, BS)>>>
        (DA, DB, DC, n);

   // 結果のCを、デバイスメモリからホストメモリへコピー
   cudaMemcpy(C, DC, sizeof(double)*n*n, cudaMemcpyDeviceToHost);

   // Cを出力などに利用(略)
   return 0;
}
```

※TSUBAMEにある本物mm_gpu.cuは時間測定なども含む

1. CUDA プログラムの概要

CUDA プログラムの構成

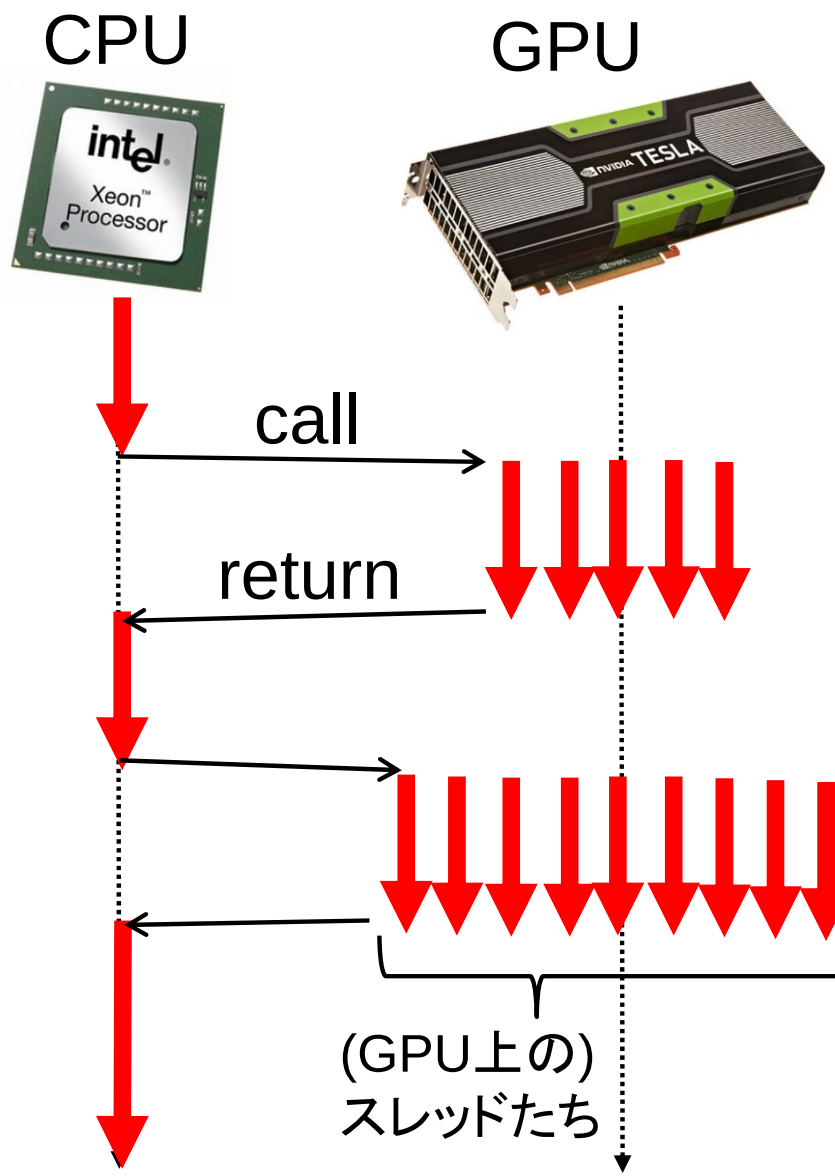
ホスト関数 ＋ GPUカーネル関数

二種類の関数の定義が xxx.cu ファイル内に混ざっている

- ホスト関数
  - CPU上で実行される関数
  - 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や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
  - 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)

xxx.cuファイルは、nvccコマンドでコンパイルする

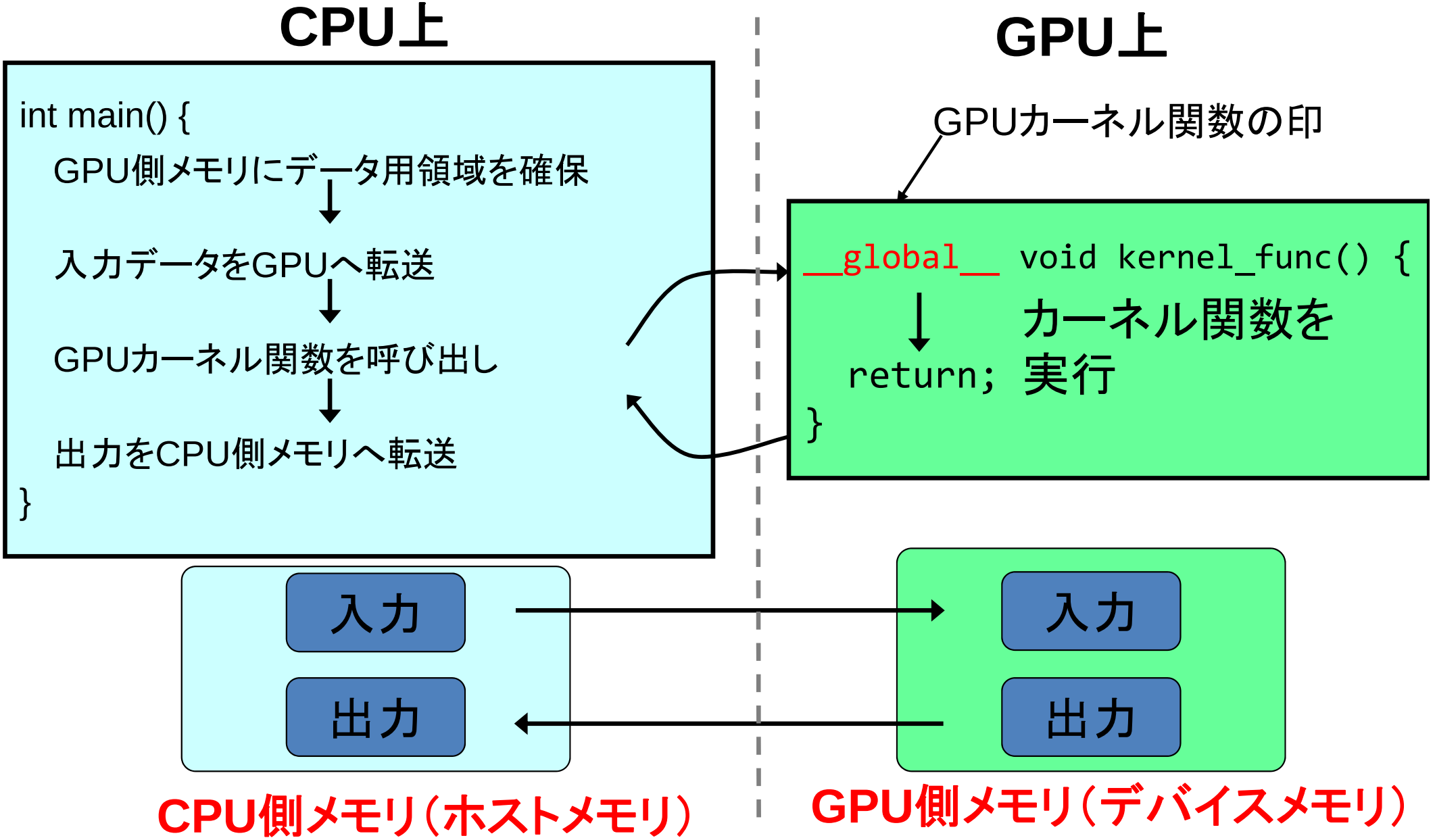
1. CUDA プログラムの概要



- main()から開始し、当初はCPUだけ動く
  - GPUは、CPUから処理を依頼された時だけ動く
  - GPU上で動く関数 = GPUカーネル関数
- CPUとGPU間ではメモリは別々
  - CPU から GPU にデータを転送して実行させるのが基本
- GPU上では多数のスレッド達が動く.
  スレッド達の間では同じメモリが見える

1. CUDA プログラムの概要

CUDAプログラムの流れの例



この2種類のメモリの区別は忘れずに！

2. mm_gpu.cu の詳細 (CPU側)

```
int main(int argc, char *argv[])
{
    int i, j, n;
    double *A, *B, *C;  // ホストメモリ用のポインタ
    double *DA, *DA, *DC;   // デバイスメモリ用のポインタ

    n = atoi(argv[1]);  // 行列の大きさ

    // A, B, Cのためにホストメモリを確保
    A = (double *)malloc(sizeof(double)*n*n);
    B = (double *)malloc(sizeof(double)*n*n);
    C = (double *)malloc(sizeof(double)*n*n);

    // A, Bの内容を設定し、Cをゼロクリア
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            A[i*n+j] = 3.0;  // ここでのA, Bの内容はいい加減
            B[i*n+j] = 0.1;   //に決めた.（あくまで例なので）
            C[i*n+j] = 0.0;
        } }
```

実行はmain関数から、CPU上で始まる

← これは後で使う

行列A, B, Cのためのメモリ領域をmallocで準備
これらはホストメモリ上に確保される
このプログラムでは、二次元配列でなく「サイズn*nの一次元配列」にした

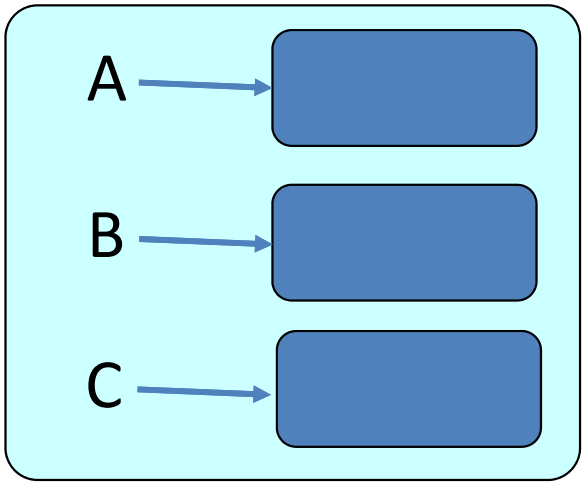
一次元になっているため、たとえばCijをアクセスしたいときには、C[i*n+j]とする

2. mm_gpu.cu の詳細(CPU側)

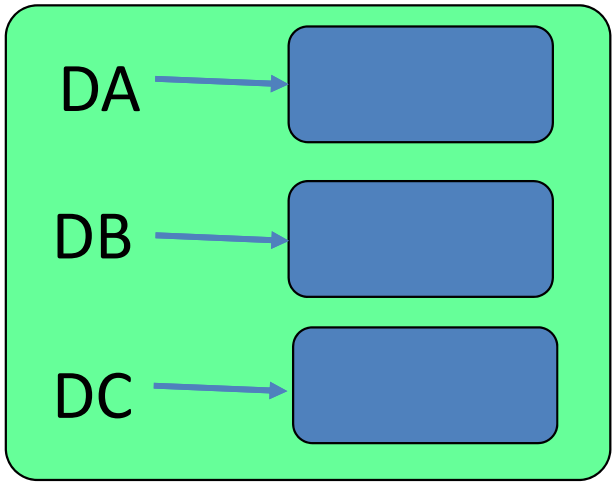
```
// A, B, Cのためにデバイスメモリを確保
cudaMalloc((void**)&DA, sizeof(double)*n*n);
cudaMalloc((void**)&DB, sizeof(double)*n*n);
cudaMalloc((void**)&DC, sizeof(double)*n*n);
```

結果のポインタはここに入る

これを実行すると、DA, DB, DCはそれぞれ、
「sizeof(double)*n*n」バイトの大きさの、デバイスメモリ上の領域を指す



ホストメモリ

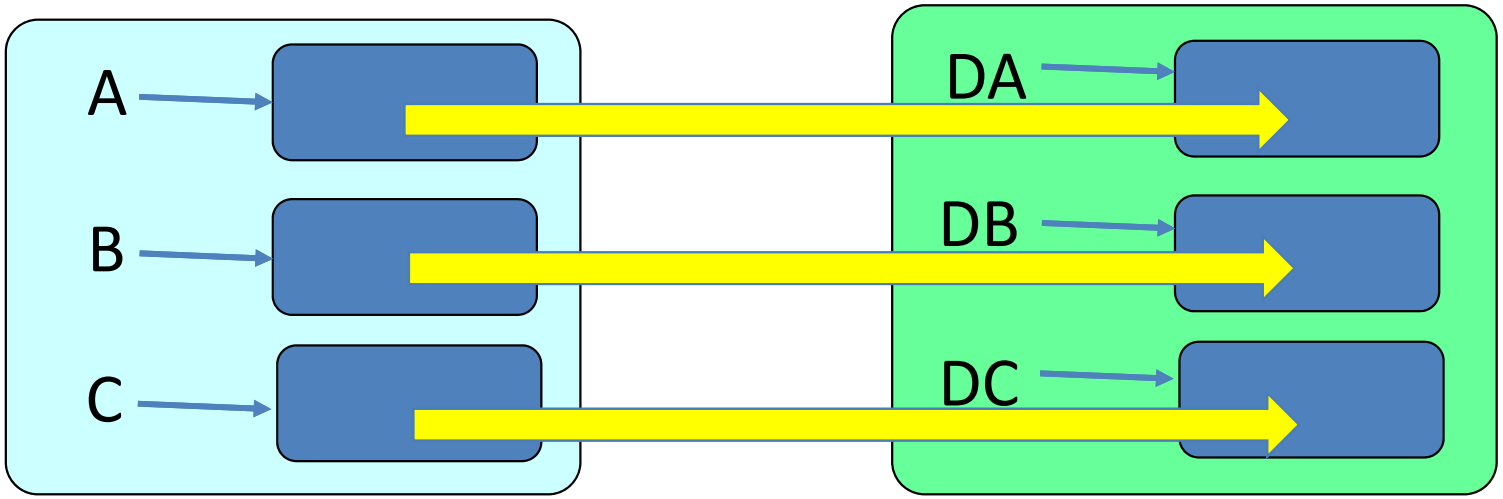


デバイスメモリ

注意! ホスト関数(CPU側)からは、DA, DB, DCの中身には触れない。
DA[i*n+j] = … などしてはダメ！

2. mm_gpu.cu の詳細(CPU側)

```
// A, B, Cの内容を、ホストメモリからデバイスメモリへコピー
  cudaMemcpy(DA, A, sizeof(double)*n*n, cudaMemcpyHostToDevice);
  cudaMemcpy(DB, B, sizeof(double)*n*n, cudaMemcpyHostToDevice);
  cudaMemcpy(DC, C, sizeof(double)*n*n, cudaMemcpyHostToDevice);
```



cudaMemcpyの引数は4つ

1. コピー先
2. コピー元
3. 何バイトコピーしたいか
4. コピーの種類
   - cudaMemcpyHostToDevice: ホスト→デバイスのとき
   - cudaMemcpyDeviceToHost: デバイス→ホストのとき

ここまでで、
やっとGPUが動ける
準備ができた！

2. mm_gpu.cu の詳細(CPU側)

いよいよGPU上で動く関数を呼び出そう !!

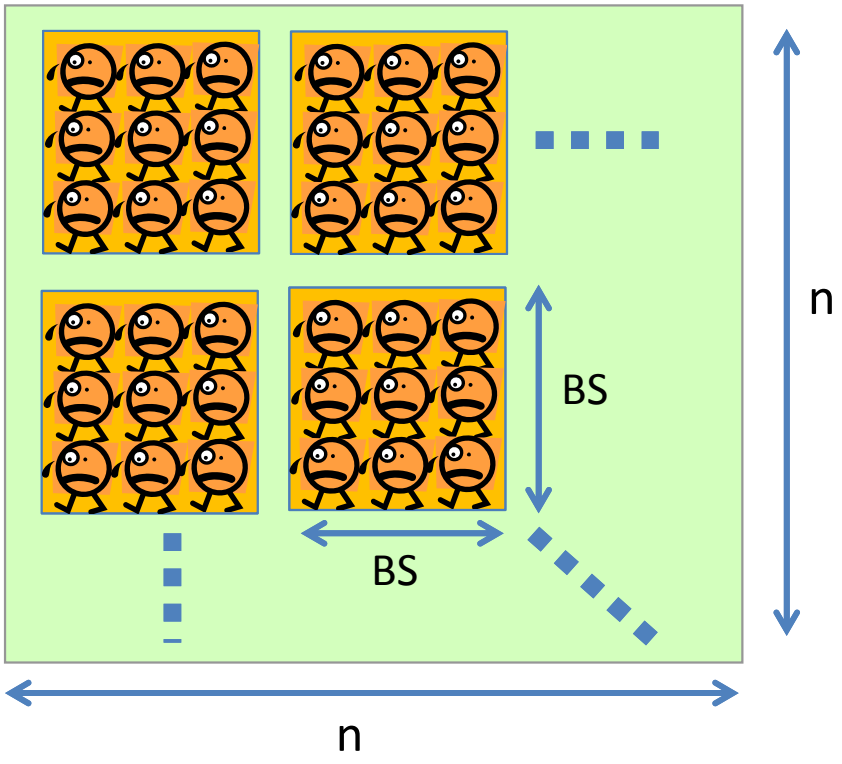
```
// GPUカーネル関数を呼び出す!! 約n*n個のスレッドを使う
// このプログラムではBSは16
mm_gpu <<<dim3((n+BS-1)/BS, ((n+BS-1)/BS)), dim3(BS, BS)>>> (DA, DB, DC, n);
```

呼びたい関数名

いくつのスレッドで関数を実行するか

渡したい引数

この例はちょっと難しいのですが・・・
結局は、約(n * n)個のスレッドを使います
それぞれのスレッドには、二次元の背番号をつけます



2. mm_gpu.cu の詳細 (GPU側)

GPU上で、超多数のスレッドがmm_gpu関数を実行開始する

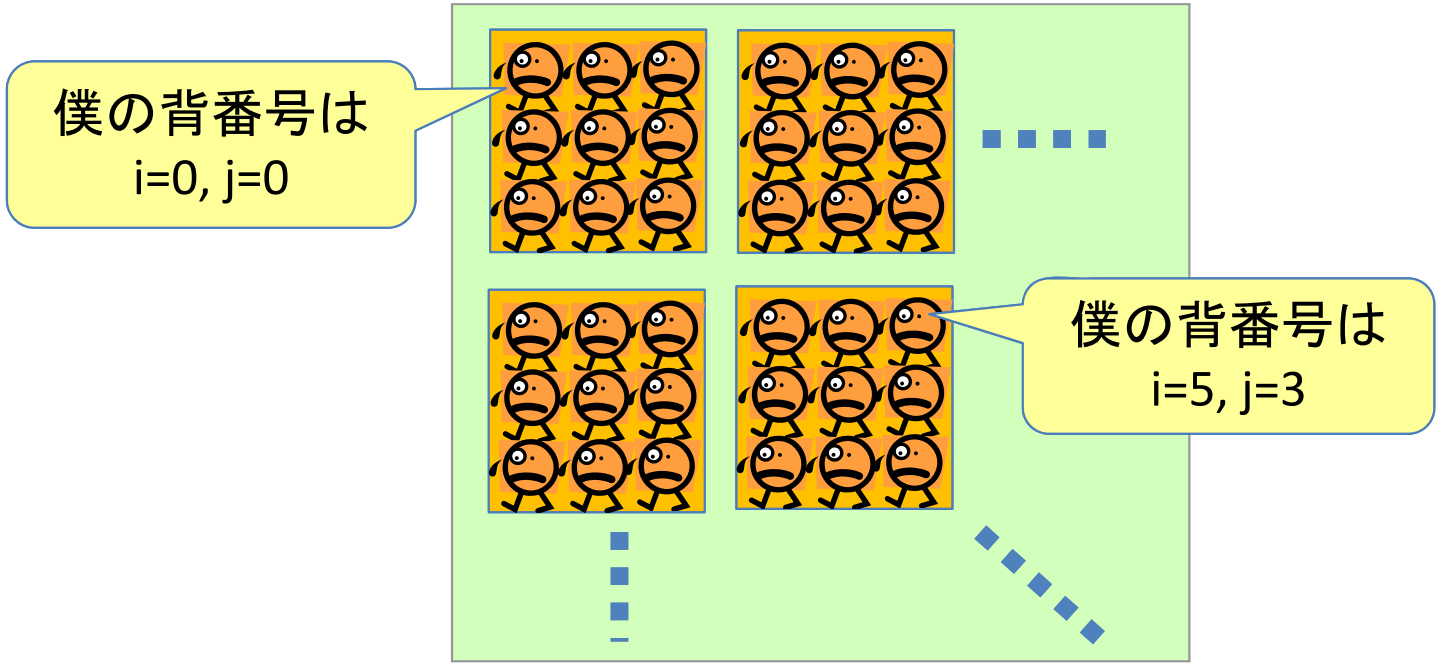
GPUカーネル関数の印

```
__global__  void  mm_gpu(double *A, double *B, double *C, int n)
{
    int i, j, k;
    i = blockIdx.y * blockDim.y + threadIdx.y;
    j = blockIdx.x * blockDim.x + threadIdx.x;
    // 自分の背番号から担当する (i,j) を決める
    if (i >= n || j >= n) return; // 行列からはみ出す部分は計算しない
```

デバイスメモリ上のポインタが来ているはず

自分のY方向の背番号を計算

自分のX方向の背番号を計算

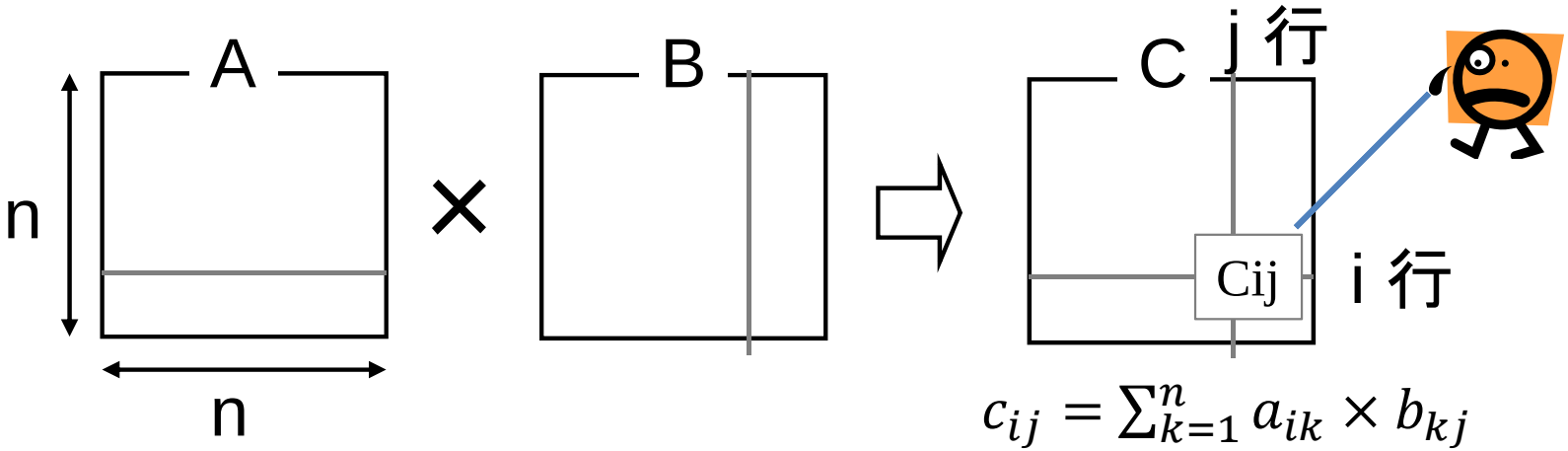


2. mm_gpu.cu の詳細(GPU側)

背番号にしたがって、Cijを一個だけ計算する

```
    for (k = 0; k < n; k++) {  // 総和を計算する部分
        C[i*n+j] += A[i*n+k] * B[k*n+j];
    }
}
```

一次元配列なのでこの表記



$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

参考:
CPUの場合は一人で
全体を計算するので
3重ループだった

```
for (i = 0; i < n; i++) {  // Cの第i行に注目
  for (j = 0; j < n; j++) {  // Cの第j列に注目
    for (k = 0; k < n; k++) {  // 総和計算のループ
      c[i][j] += a[i][k]*b[k][j];
    } } }
```

2. mm_gpu.cu の詳細(CPU側)

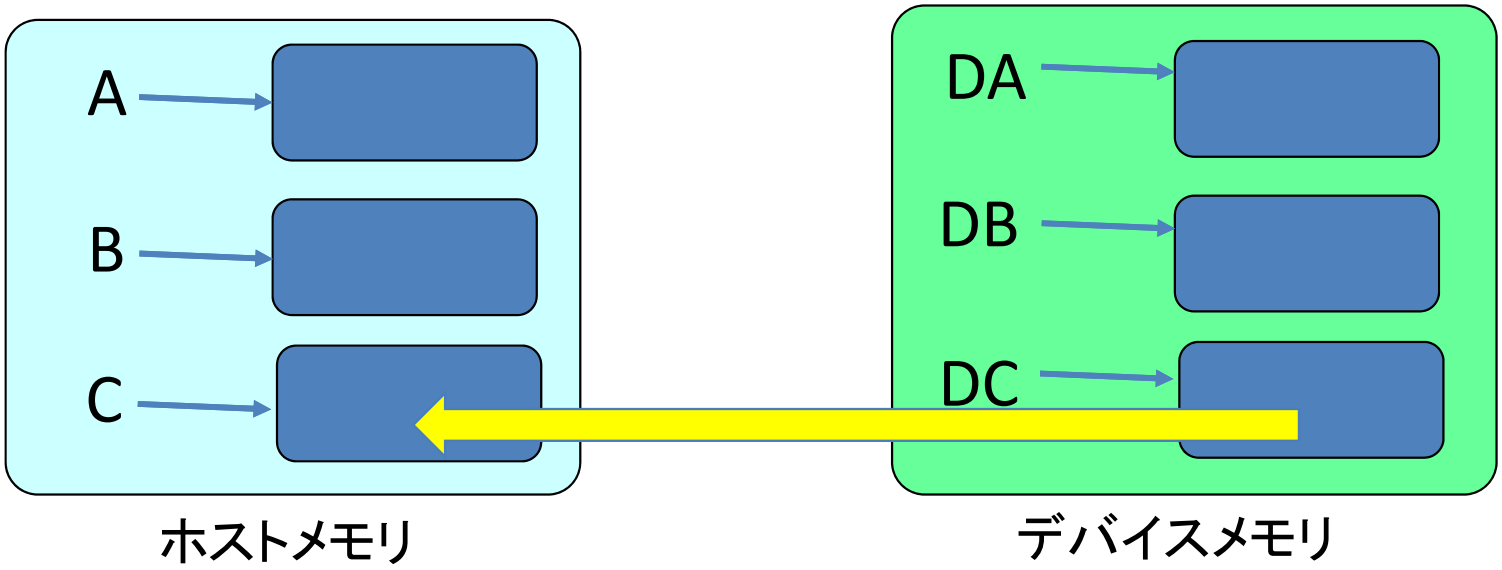
GPUカーネル関数からmain関数へ戻ってきた

```
// 結果のCを、デバイスメモリからホストメモリへコピー
cudaMemcpy(C, DC, sizeof(double)*n*n, cudaMemcpyDeviceToHost);

// Cを出力などに利用(略)
}
```

デバイスからホスト行きなので

ここでは、Cだけをコピーすることにした。A, Bはいらないので



3. 次のステップへ

「動く」プログラムを作った後は、「速い」プログラムにしたい
ポイントは色々ある

<<<〇, 〇 >>>のスレッド数の調整によって速度が変わる

GPU上で、隣のスレッドどうしは、同時に配列の近い場所にアクセスすると速い (コアレスドアクセス)

GPU上でif文で分岐すると非効率になることがある

不要なcudaMemcpyは削ったほうがよい

詳しくは、次のステップの資料で！