1. CUDAの導入，CUDAの基礎

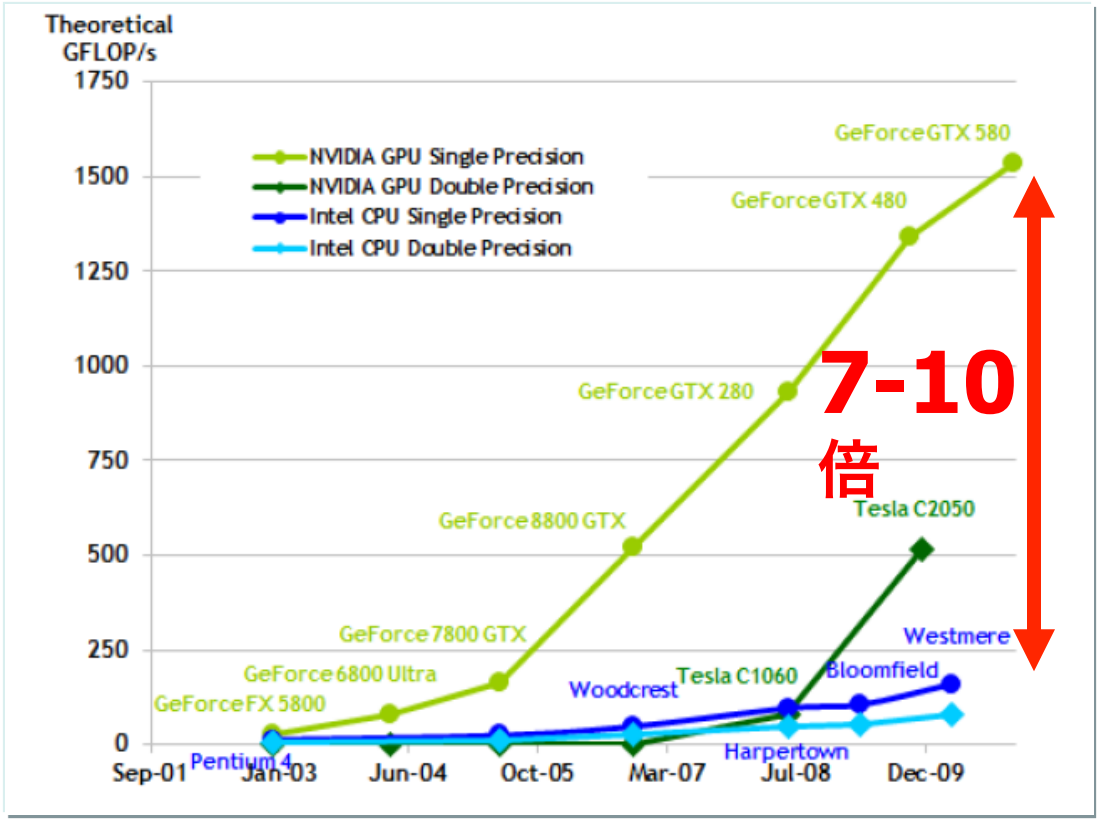
茨城大学工学部
教授　乾　正知

茨城大学
Ibaraki University

smartec

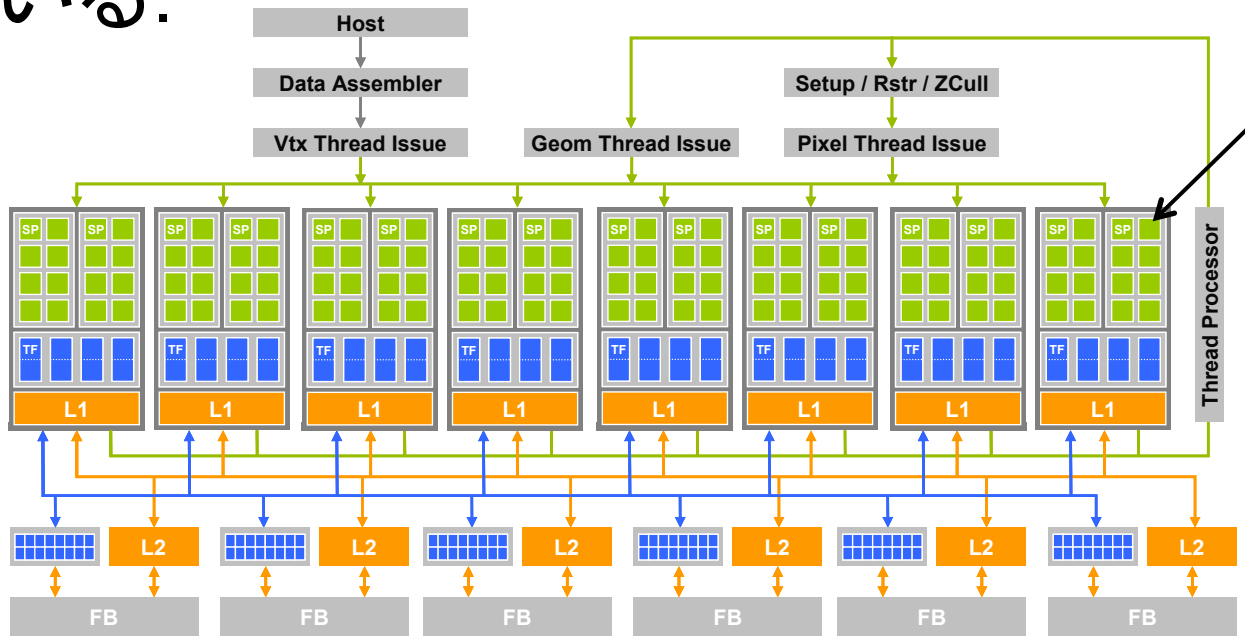
GPU（Graphics Processing Unit）

- GPUはもともとは，コンピュータグラフィックス画像を高速に画面表示するためのLSIとして開発された.
- GPUは同時期のCPUと比較して7～10倍高速.



GPUの高速性の理由

- CGにおけるほとんどの処理は;
  - モデルの全ポリゴンに同じ処理を実行.
  - 画像を構成する全画素に同じ処理を実行.
- このような計算を効率化するために，GPUはSIMD型の並列処理プロセッサとして設計されている.



SP: Streaming Processor（中心的な処理ユニット）

GeForce 8800 GPUのアーキテクチャ

SIMD（Single Instruction / Multiple Data）

- 同一手続きを用いて，複数のデータ処理（スレッド）を同時に実行するタイプの並列処理．
- 行列計算はその典型例．

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{bmatrix} + \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ b_{30} & b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} a_{00}+b_{00} & a_{01}+b_{01} & a_{02}+b_{02} \\ a_{10}+b_{10} & a_{11}+b_{11} & a_{12}+b_{12} \\ a_{20}+b_{20} & a_{21}+b_{21} & a_{22}+b_{22} \\ a_{30}+b_{30} & a_{31}+b_{31} & a_{32}+b_{32} \end{bmatrix}$$

行列の足し算

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}$$

行列の掛け算

GPUを用いたスーパーコンピュータ

- スーパーコンピュータは，従来型のCPUとGPUを多数用いたヘテロジーニアスな構成が一般的.
- GPUは高性能な割に消費電力が少なく，多くのスパコンで使われている.

TSUBAME 2.0：東京工業大学．Top500中第5位．世界で最もエコな（グリーンな）スパコンとしても知られる.

Tianhe-1A（中国）：Top500中第2位のスパコン.

概要

- CUDAとは，C言語に
  1. CPUからGPUデータを転送する機能，
  2. GPUに多数のスレッドを並列に起動させる機能，

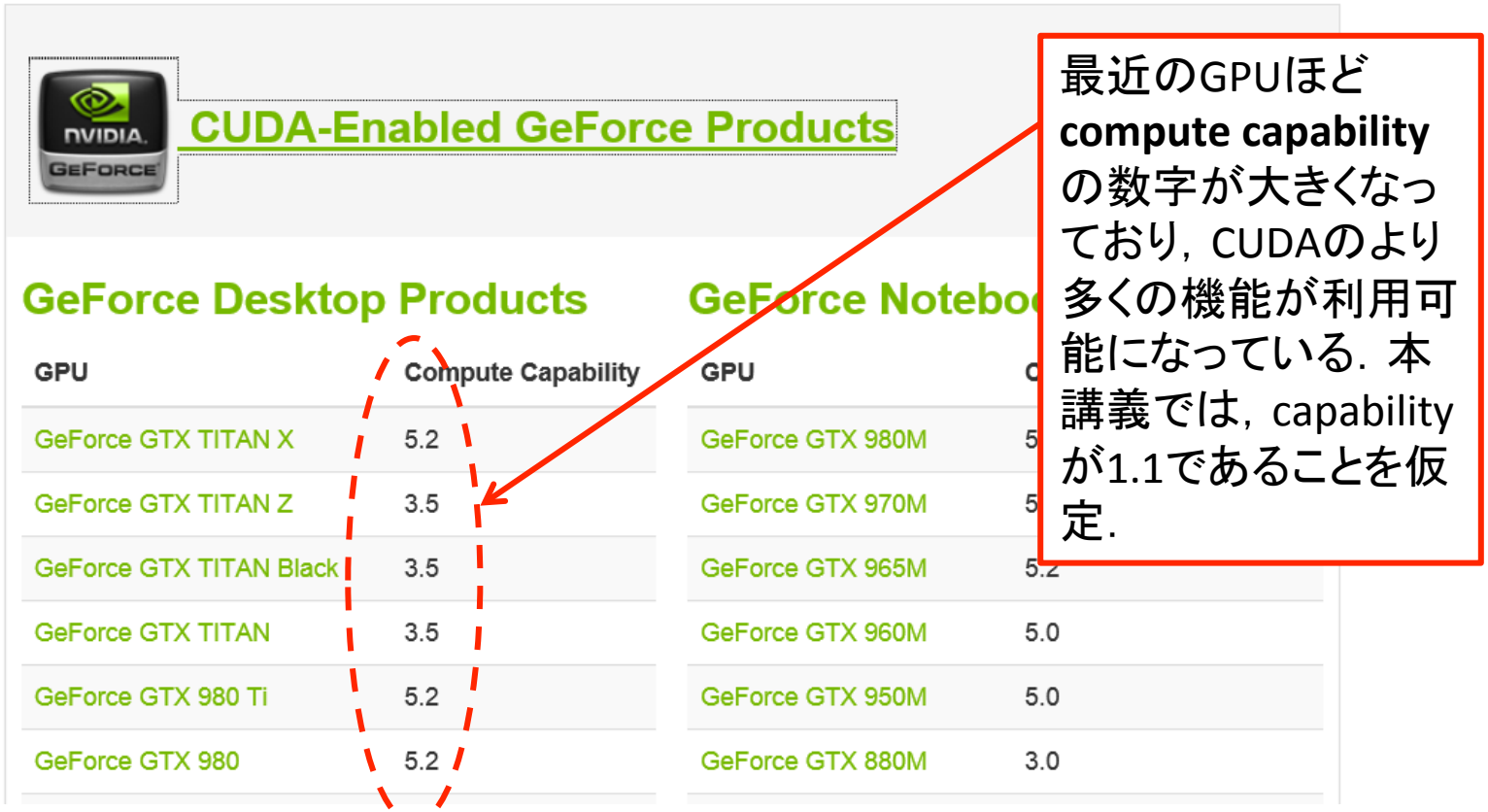
  を追加したもの．
- CUDAの開発はnVIDIA社が行っており，nVIDIA社のサイトから自由に入手できる．
  - CUDAが実行可能なのはnVIDIA社のGPUだけ．
- 今回は以下の3点の説明を行う；
  - Windows PCにCUDAをインストールする方法．
  - VisualStudioを用いてCUDAプログラムを開発する手法．
  - CUDAを用いた簡単な行列計算のプログラムの紹介．

CUDA; Compute Unified Device Architecture

- CUDA：nVIDIA社のGPUを，メニーコアのSIMD（Single Instruction / Multiple Data）型の並列処理プロセッサとして利用するためのプログラミング環境.
- C言語の拡張.
- Windows PC（32/64bit）だけでなく，Linux，Mac.でも利用可能.
  - Windows PCではVisualStudioの利用を仮定.
- nVIDIA社のサイトに膨大な量のサンプルプログラムが掲載されている. これらを読むことがCUDA理解の早道.
- 最新のCUDAは7.0. バージョンアップが非常に早い.

CUDAの導入(1/2)

- CUDAはnVIDIA社のGPUで動作する．以下のホームページで，自分のPCに搭載されているGPUを探し機能をチェックしておく．
  - https://developer.nvidia.com/cuda-gpus



CUDAの導入(2/2)

- グラフィクスボードのドライバーソフトを最新のものに更新しておく．最新のドライバーは以下のページから入手可．

  http://www.nvidia.co.jp/Download/index.aspx?lang=jp

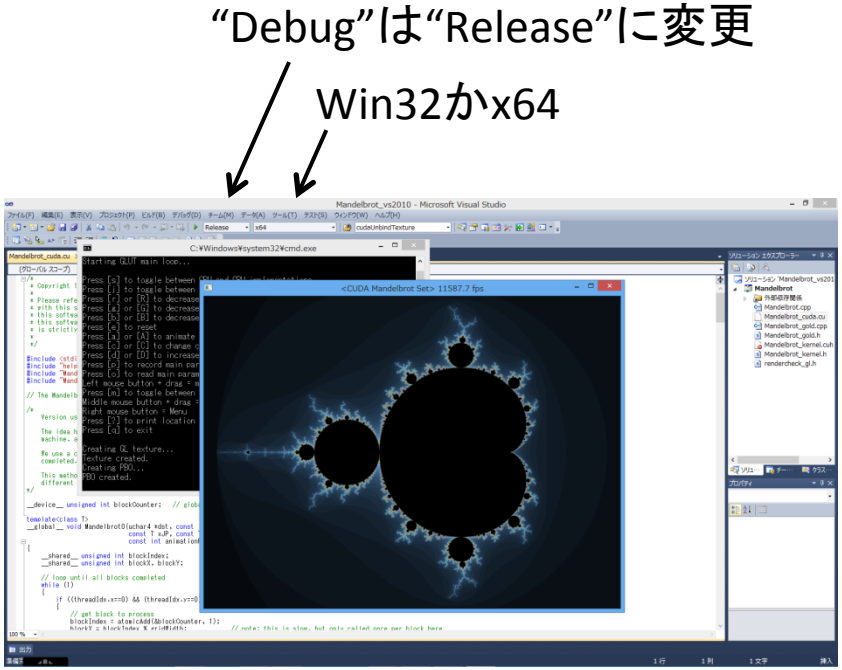
  – ノート型のPCの場合には，ドライバーの更新ができないものもある．最悪の場合，最新バージョンのCUDAが動作せず，古い(legacy)CUDAを使うことになる．
- CUDAのツールキットを以下のページから導入．

  https://developer.nvidia.com/cuda-downloads

  – ツールキットには；
    - コンパイラ，サンプルプログラム，ドキュメント(英語)が含まれる．
  – 最新版はVer.7.0だが7.5のリリースが予告されている．
  – 説明は7.0で行うが，6.0などの旧版でもそのまま稼働する．
- 導入はダウンロードされたインストーラを起動するだけ．設定はデフォルトのままでよい．10～20分かかる．

VisualStudioによるCUDAプログラミング（1/2）

- 導入を終えるとVisualStudioからCUDAが使える.
  - *.cppファイル：VS側のコンパイラが処理.
  - *.cuファイル：CUDAコンパイラのnvccが処理.
- 以下のフォルダーにたくさんのサンプルプログラムがある.
  - C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.0
- サンプルをビルドして欲しい.

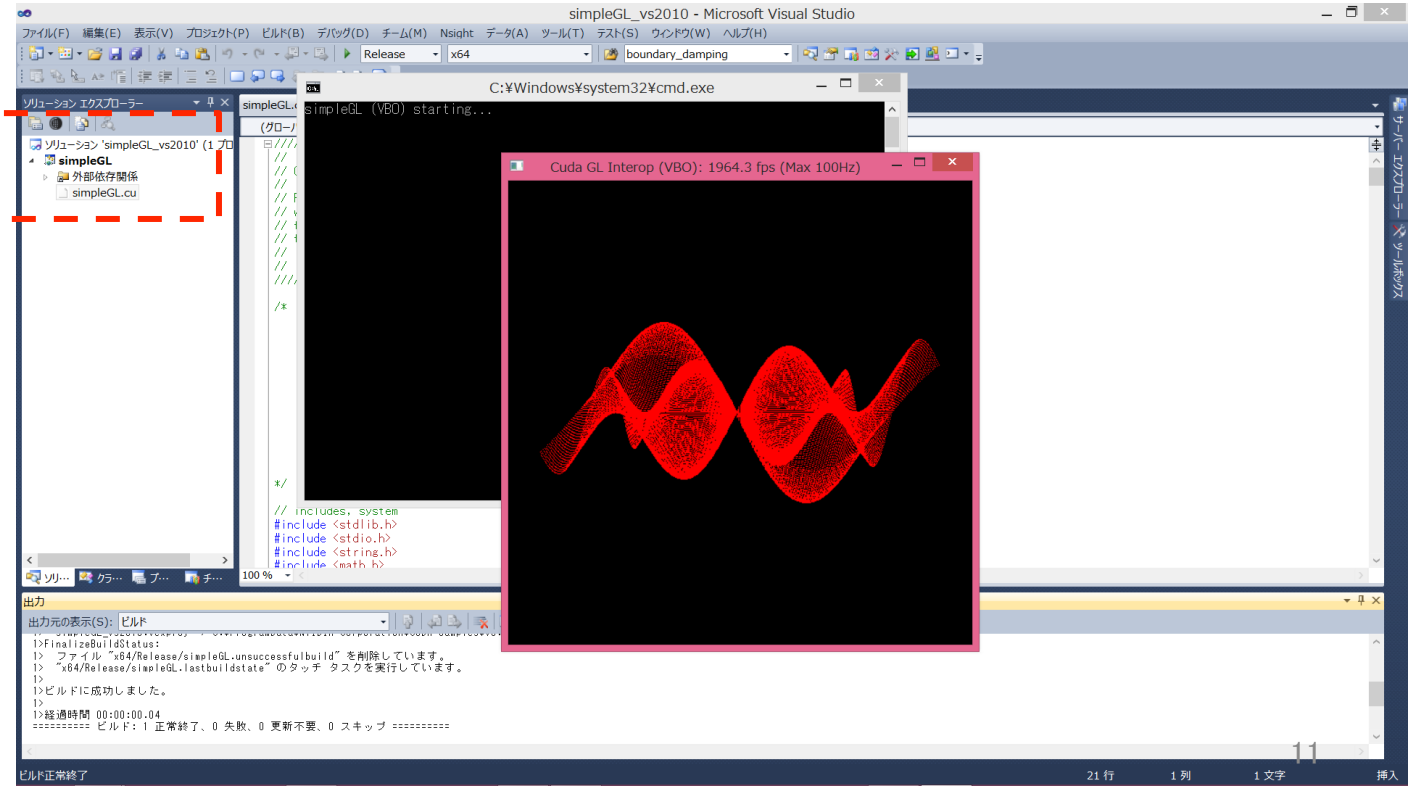


VisualStudioによるCUDAプログラミング（2/2）

- CUDAプログラムを開発する最も簡単な方法は，サンプルプログラムをコピーして，それを修正すること．
- この講義では，“2_Graphics”中の“simpleGL”プロジェクトをテンプレートとして利用する．

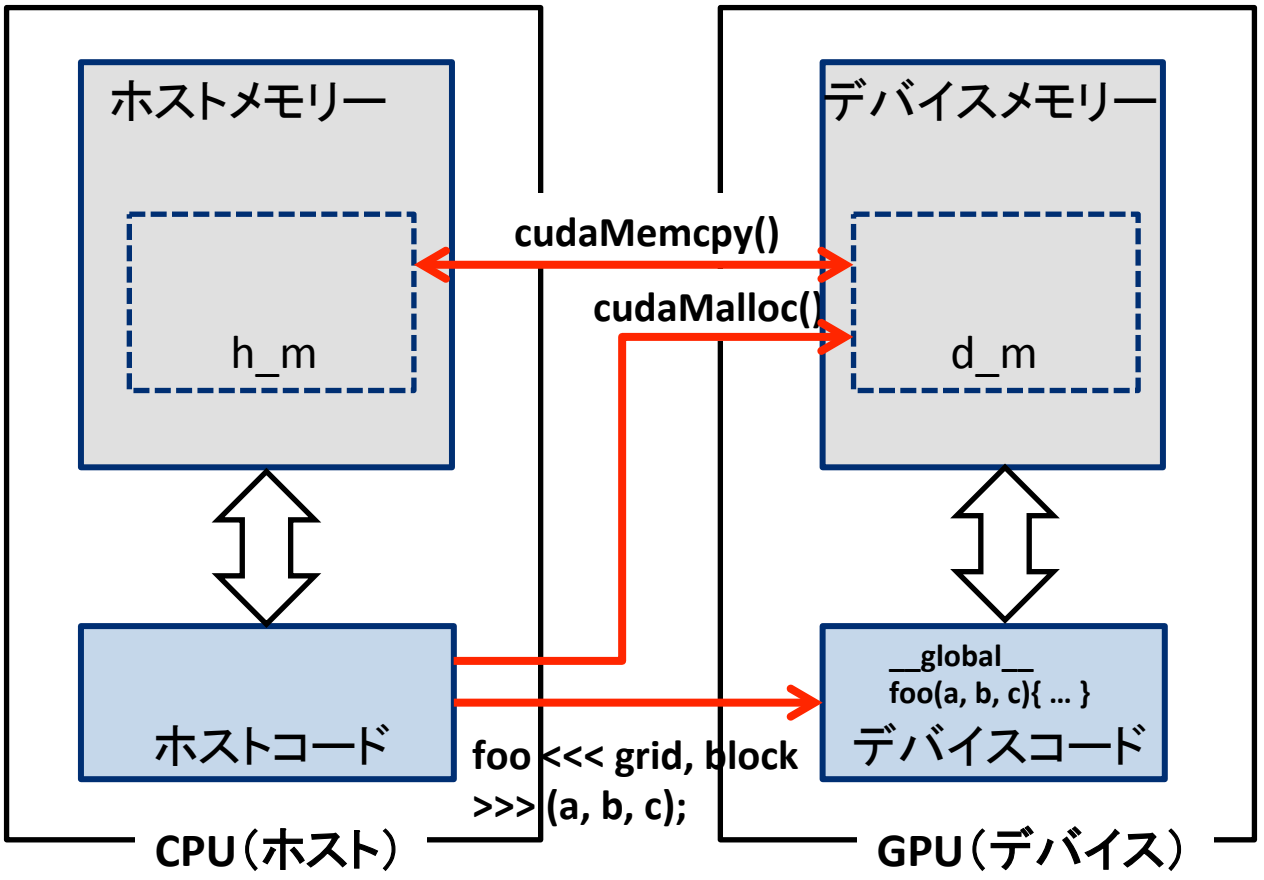
simpleGL.cu

既存のファイルを削除して，代わりに新しいファイルを追加．

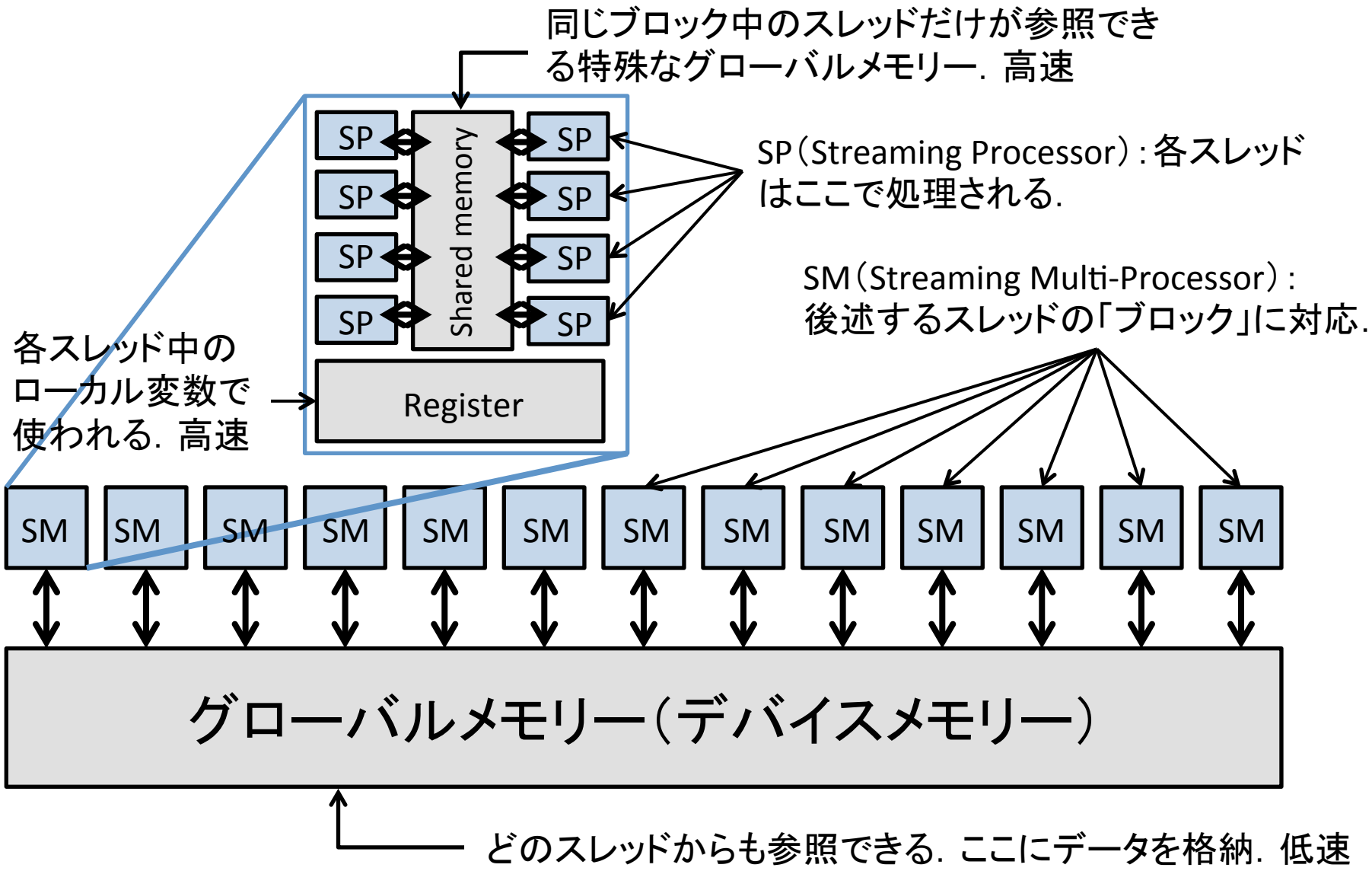


CUDAにおけるCPUとGPUの役割

- CPUとGPU：一つのPCに2つの異なるコンピュータと考える.
  - **ホスト**：CPUと主メモリー
  - **デバイス**：GPUとグラフィックスメモリー
- CUDAは，Cに以下の2つの拡張を加えてある；
  - ホストとデバイス間のデータ転送機能.
  - ホスト側から，デバイスに多数のスレッドを処理させる機能.



GPUのメニーコア・アーキテクチャ



CとCUDAの違い

- CUDAは基本的にC言語．Cプログラムはそのまま動く．ただし2つの拡張が加えられている．
  1. ホストメモリー（CPU側）とデバイスメモリー（GPU側）間でデータを転送する機能．
     - ホストとデバイスのメモリーは異なるシステム．
     - ホストメモリー上のデータをデバイスメモリーへコピーする．またはその逆が可能．
  2. ホスト側から，デバイスに多数のスレッドを並列に起動するよう命じる機能．
     - forループ中で繰り返し実行される手続きを，等価なスレッドの並列起動に置き換えることが一般的．
     - 置き換え作業にはそれなりのスキルが必要．

スレッドへの置き換え

- forループ中で繰り返し実行される互いに独立な手続きを，スレッドの並列起動に置き換えることが基本．
- 個々のスレッドはGPUのSPで処理される．SPの総数はスレッドの数より小さいので，GPU側でスケジューリングされる．

```
for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        for (k = 0; k < 10; k++) {
                ……

            func(i, j, k);

                ……

        }
    }
}
```

forループ中の*func()*の繰り返し

置き換え

CUDA処理に適した問題とは

- CUDAは万能ではない.
  - メニーコア型のアーキテクチャに適合した処理のみが高速化される.
- 特に以下のタイプの問題処理に適している.
  - forループ中で，比較的単純な処理を膨大な回数繰り返すケース.
    - 条件判定が多い処理はダメ．処理が途中で止まることもある.
  - 個々の処理は互いに独立でなくてはならない.
    スレッドの実行順序は仮定できない.

CUDAプログラムの処理の流れ

1. **cudaMalloc**関数を用いて，デバイスメモリーにデータを格納する領域を確保**.**
2. **cudaMemcpy**関数を用いて，ホストメモリーからデバイスメモリー上の確保した領域へデータを転送.
3. グリッド（grid）とブロック（block）に基づいて，カーネル関数（＝スレッド）を一斉に起動.
   - 各スレッドはデバイスメモリー上のデータを参照.
   - 計算結果もデバイスメモリーへ格納される.
4. **cudaMemcpy**関数を用いて，計算結果をデバイスメモリーからホストメモリーへ転送.
5. **cudaFree**関数を用いて，使い終わったデバイスメ

デバイスメモリー上の領域の確保

- デバイスコードは，デバイスメモリー上のデータを参照して計算を行う.
- 予め必要なメモリー領域を，デバイスメモリー上に確保しておく.

  cudaMalloc(void **devptr, size_t count);

  devptr：確保するデバイスメモリー上のアドレスへのポインタ

  count：確保する領域のサイズ（単位はバイト）

  （例）GPU側で配列 **float d_m[N]**を参照する場合.

  cudaMalloc((void)&d_m, N * sizeof(float));**

- メモリーを使い終えたら，**cudaFree(d_m)**関数を用いて，確保してあった領域を解放する.

データ転送用の関数

- ホストメモリーとデバイスメモリー間でデータを転送する際には，**cudaMemcpy**関数を用いる．

cudaMemcpy(void *dst, void *src, size_t count, enum cudaMemcpyKind, kind)

*dst：転送先（destination）のメモリー上のアドレス．

*src：転送元（source）のアドレス．

*count：転送するデータの大きさ．単位はバイト．

kind; cudaMemcpyHostToDevice (Host→Device)

cudaMemcpyDeviceToHost (Device→Host)

cudaMemcpyDeviceToDevice (Device→Device)

（例）ホストメモリー上の配列**float h_m[N]**のデータを，デバイスメモリ上一の配列**float d_m[N]**へ転送する場合．

cudaMemcpy(d_m, h_m, N * sizeof(float),

cudaMemcpyHostToDevice);

スレッドの定義（1/3）

- forループ中の単純な処理をスレッドに置き換える.
- 置き換えには，ブロック（block）とグリッド（grid）というスレッドの構造を利用.

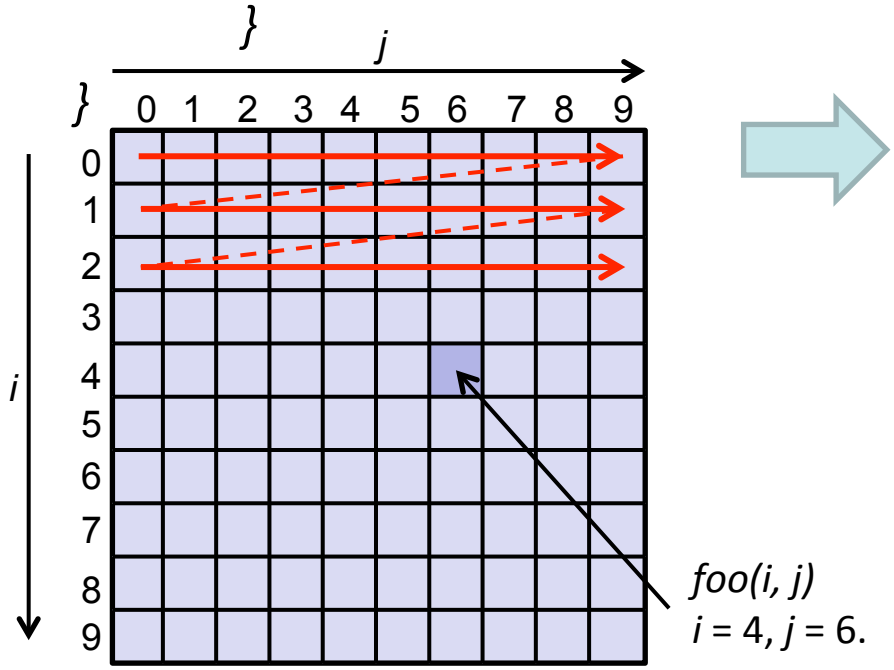
```
for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        foo(i, j);  ← スレッドに置き換える.
    }
}
```

変数*i*と*j*の役割
（1）*foo*の実行順序の制御：スレッドは並列に実行．順序なし.
（2）関数*foo*の引数：並列実行のスレッドではどうする?

スレッドの定義（2/3）

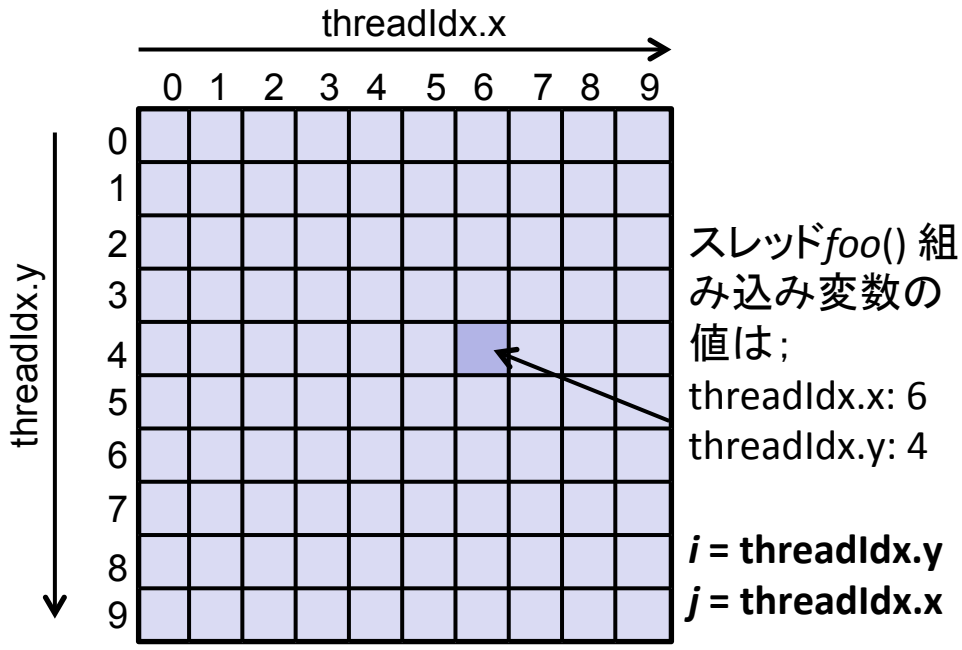
- 変数*i*と*j*を用いた簡単な処理*foo(i, j)*の繰り返し.

```
for (i = 0; i < 10; i++) {
        for (j = 0; j < 10; j++) {
                foo(i, j);
        }
}
```



- カーネル関数foo()とスレッドの2次元ブロックblock(10,10)の対応.

各スレッド（block中の1要素）の場所は，組み込み変数threadIdx.xとthreadIdx.yで参照できる.

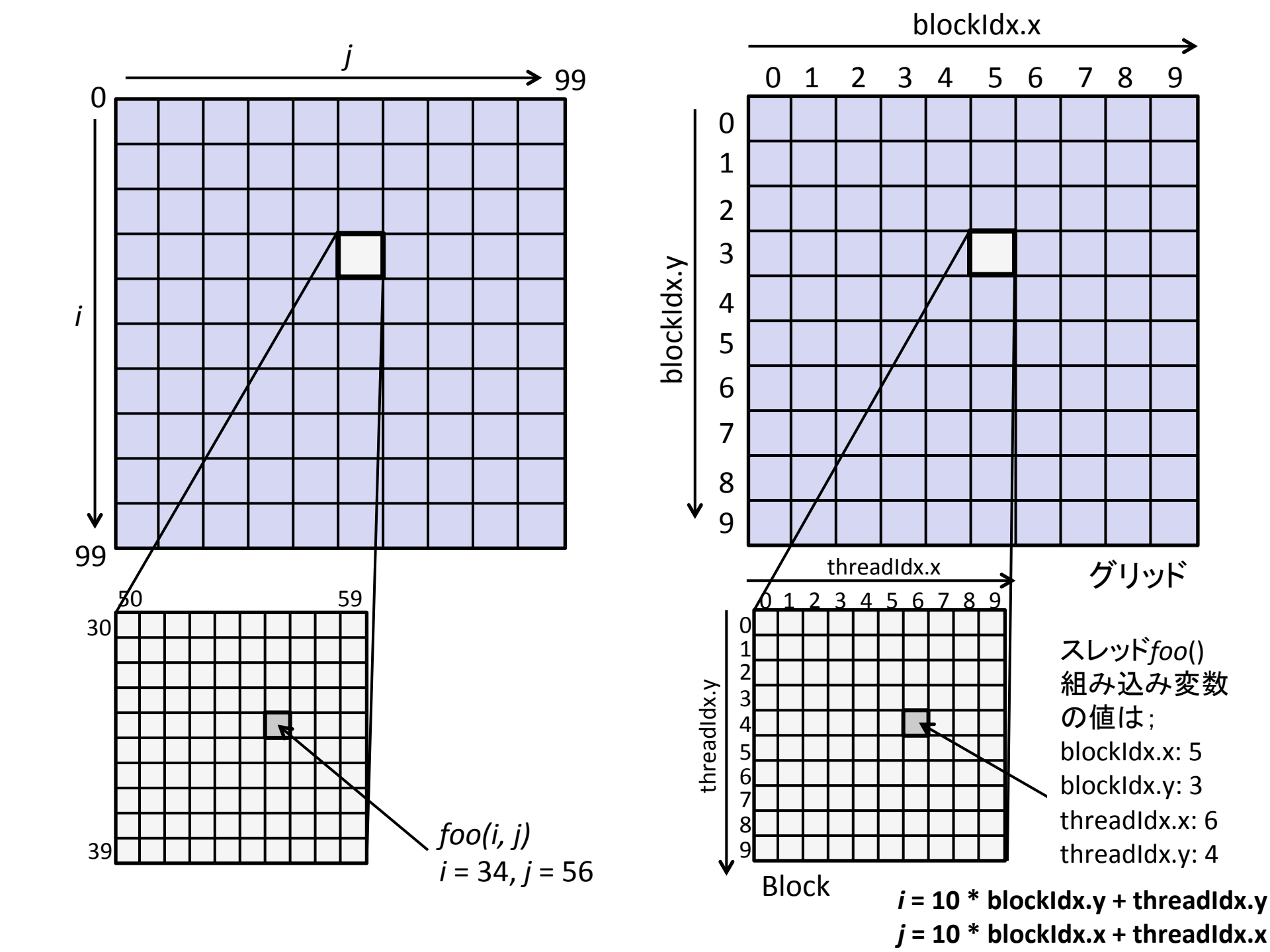


スレッドの定義(3/3)

- 1ブロック中のスレッドの総数は512に制限されている.
- 以下の繰り返し処理は，1ブロックでは扱えない.

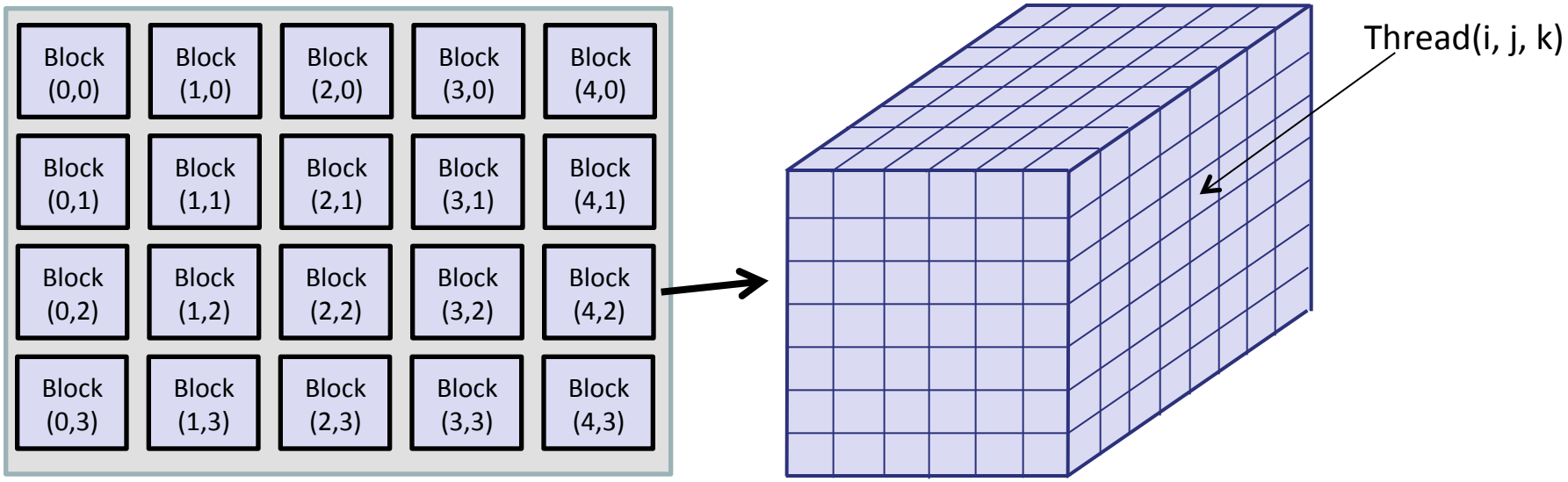
```
for (i = 0; i < 100; i++) {
        for (j = 0; j < 100; j++) {
                foo(i, j);
        }
}
```

- ブロックの配列構造であるグリッドを導入.


j
0
99
i
99
50
59
30
39
foo(i, j)
i = 34, j = 56
blockIdx.x
0 1 2 3 4 5 6 7 8 9
blockIdx.y
0
1
2
3
4
5
6
7
8
9
グリッド
threadIdx.x
0 1 2 3 4 5 6 7 8 9
threadIdx.y
Block
スレッドfoo()
組み込み変数
の値は;
blockIdx.x: 5
blockIdx.y: 3
threadIdx.x: 6
threadIdx.y: 4
i = 10 * blockIdx.y + threadIdx.y
j = 10 * blockIdx.x + threadIdx.x

スレッドのグリッドとブロック構造

- CUDAでは，最大65535 x 65535 x 512個のスレッドを並列に実行できる．SP数はずっと少ないので，内部で自動的にスケジューリングが行われる．
- このような膨大な数のスレッドを管理するために，グリッド（grid）とブロック（block）を用いる．



グリッド（Grid）：ブロックの2次元構造．グリッド中のブロックの各軸方向の最大数は**65535**.

ブロック（Block）：スレッドの3次元構造．1ブロック中のスレッドの最大数は**512**.

グリッドとブロック，カーネル関数の定義

- 特殊な**dim3**型の変数を用いてgridとblockを定義.

  （例）***dim3 grid(10, 20);*** 10 x 20個のblockからなるgrid.

  （例）***dim3 block(16, 8, 4);*** 16 x 8 x 4個のスレッドからなるblock.

- スレッドに対応するカーネル関数の定義は，C言語の文法に従う．ただしカーネル関数は値は返せない.

  （例）***__global__ void kernel(a, b, c) {…};*** 引数a，b，cは値かデバイスメモリーへのポインタ．先頭に*__global__*を付す.

- グリッドとブロックの定義は，カーネル関数起動の際に，特殊な記号<<<と>>>を用いて引き渡す.

  (Ex.) ***kernel <<< grid, block >>>(a, b, c)***

  スレッド実行の枠組み

典型的なパターン

```
for (i = 0; i < 100; i++) {
        for (j = 0; j < 100; j++) {
                foo(i, j);
        }
}
```

forループ

```
dim3 grid(10, 10);
dim3 block(10, 10, 1);
foo <<< grid, block >>> ();
```

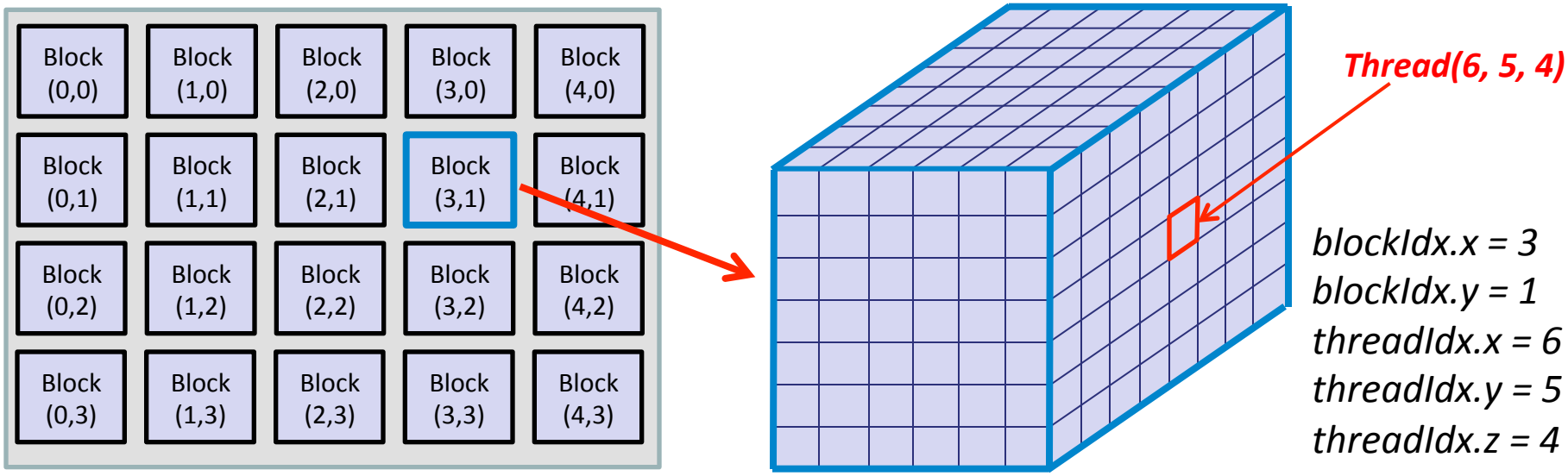
```
__global__ void foo(void)
{
        unsigned int i = 10 * blockIdx.y + threadIdx.y;
        unsigned int j = 10 * blockIdx.x + threadIdx.x;

        ＜iとjを用いた処理＞
}
```

スレッドの同定

- カーネル関数では，実行中のスレッドのgridやblock中の位置を，以下の組み込み変数を使うことで参照できる．

  blockIdx.x, blockIdx.y：2次元gridにおけるブロックの位置．

  threadIdx.x, threadIdx.y, threadIdx.z：3次元blockにおけるスレッドの位置．

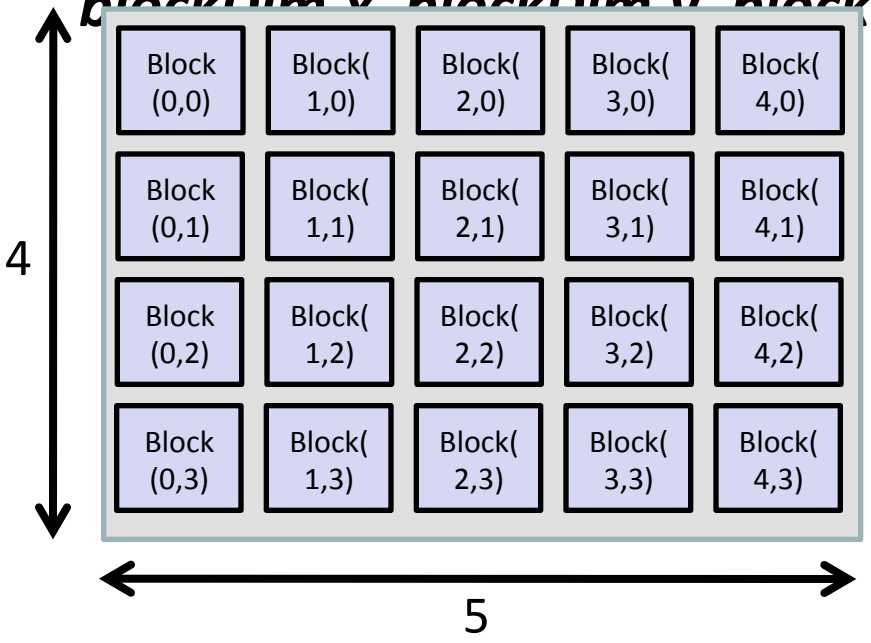


グリッドとブロックのサイズ

- カーネル関数では，実行中のスレッドのgridやblockのサイズを，以下の組み込み変数を使うことで参照できる．

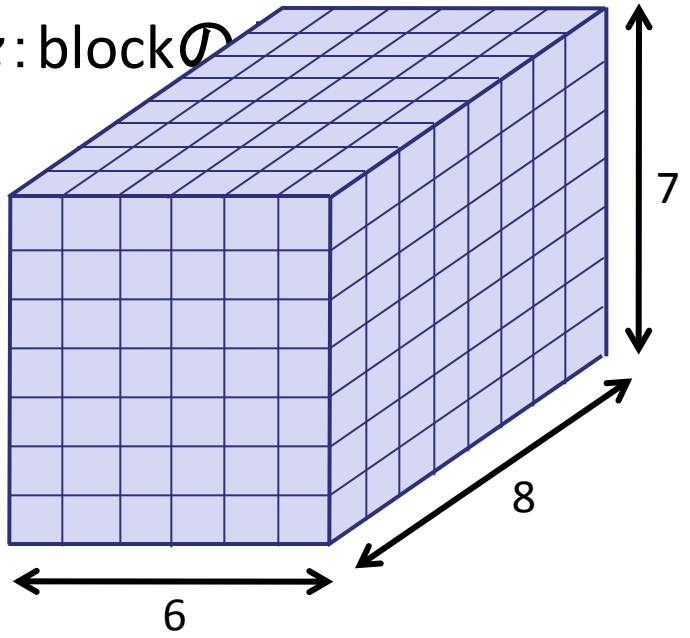
  gridDim.x, gridDim.y：gridのサイズ．

  blockDim.x, blockDim.y, blockDim.z：blockの



dim3 grid(5, 4)

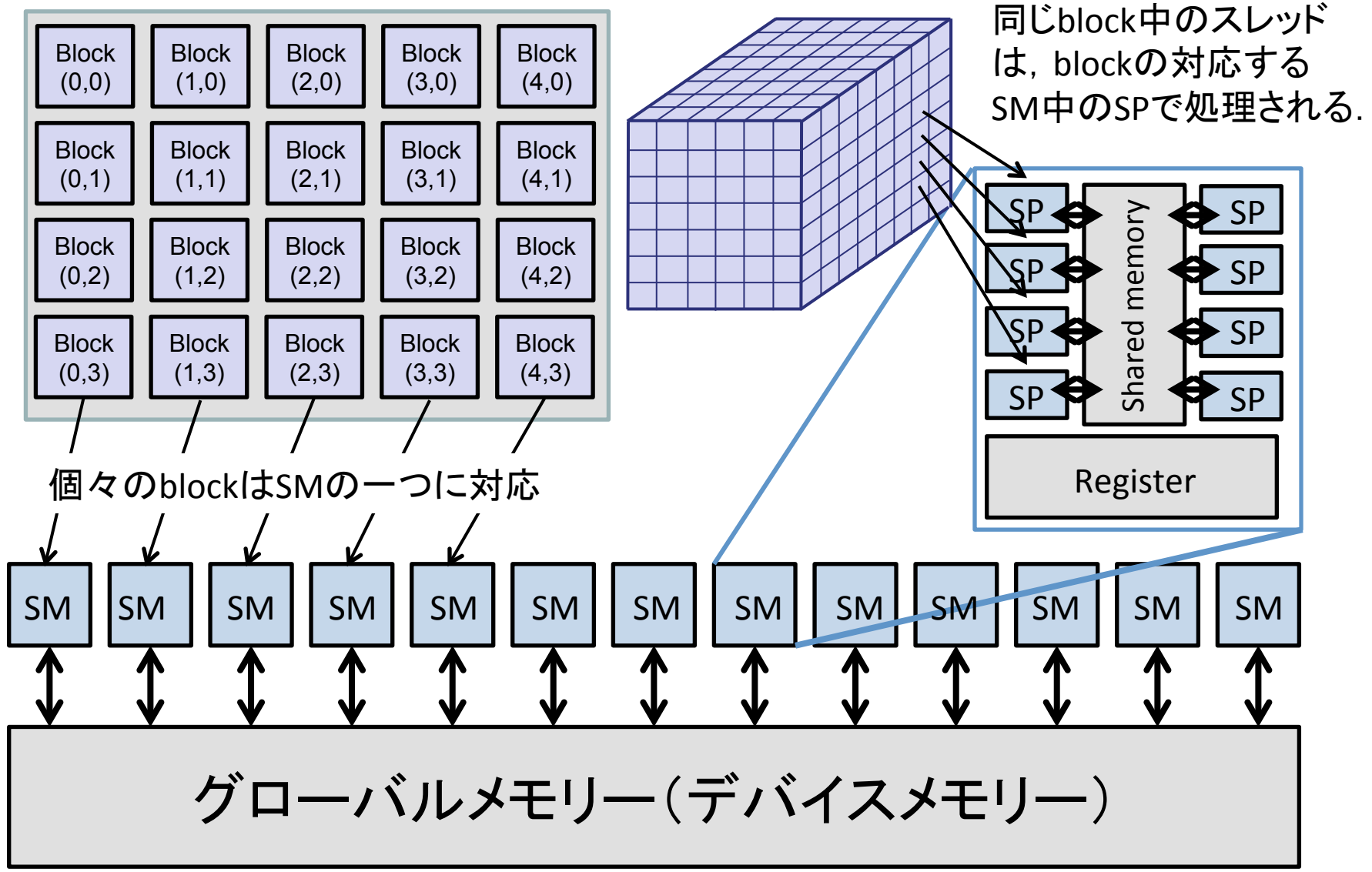
gridDim.x = 5, gridDim.y = 4



dim3 block(6, 8, 7)

blockDim.x = 6, blockDim.y = 8, blockDim.z = 7

GPUのアーキテクチャとスレッドの関係



デバイスコードの定義

- GPUが処理する関数（デバイスコード）の定義には，C言語の文法をそのまま使う.

  __device__ float foo(a, b, c) {…}; カーネル関数から呼び出せる内部関数．先頭に__device__を付す．値を返すことができる．threadIdx.xなどの組み込み変数は使えない.

  __global__ void kernel(a, b, c) {…}; カーネル関数.

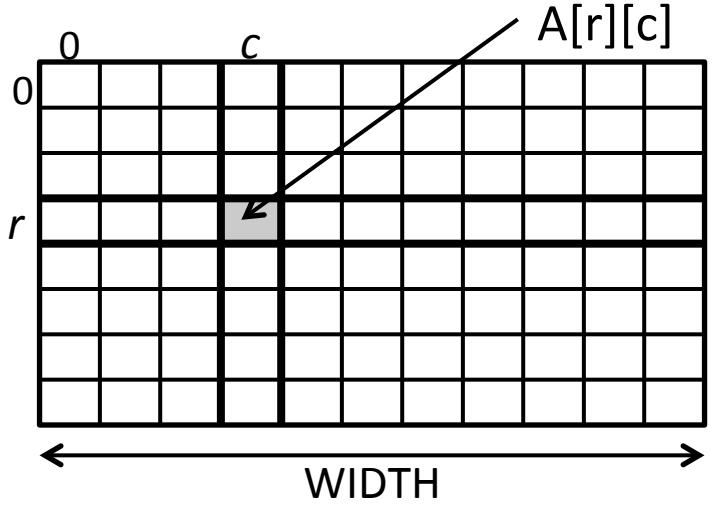
- 拡張機能.
  - ベクトルを定義する***int3***型や***float4***型.
  - ***__sinf(x), __cosf(x), __mul24()***; 高速だが精度が限定されている関数.

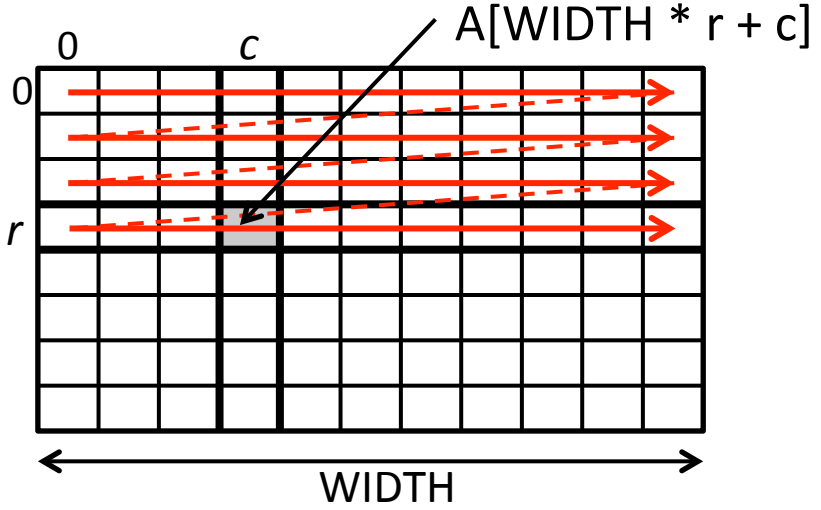
CUDAプログラムの例

- 行列の積;

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}$$

- 2次元行列を1次元配列で扱う.





行列の積を計算するCプログラム（1/2）

Matrix.cpp

```
#include <stdio.h>

#define WIDTH 1024 // 処理対象の行列のサイズはWIDTH×WIDTH.

// ホスト(CPU)側の行列定義.
float h_A[WIDTH * WIDTH];
float h_B[WIDTH * WIDTH];
float h_C[WIDTH * WIDTH];

void h_multiply(float *A, float *B, float *C);

// メイン関数.
int main()
{
    unsigned int i;

    // ホスト側の行列に値をセット.
  for (i = 0; i < (WIDTH * WIDTH); i++) {
        h_A[i] = (float)i;
        h_B[i] = (float)i;
    }

    // ホスト側での計算結果.
  h_multiply(h_A, h_B, h_C);
    printf(" ホスト計算結果: %f\n", h_C[WIDTH * WIDTH - 1]);
}
```

ホストメモリー上の配列をであることを明示するため，先頭に”h_”を付す

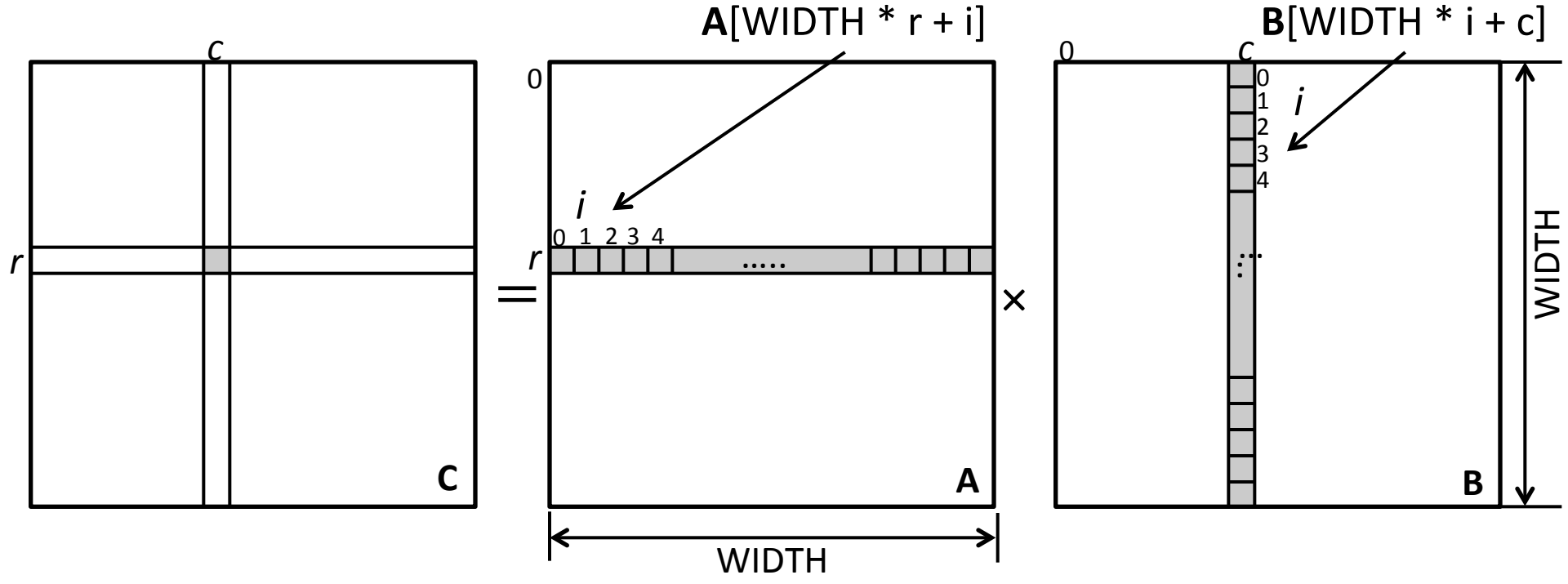
← ダミーデータを入力

行列の積

- 行列**A**と行列**B**の積として行列**C**を得る．**C**の[*r*, *c*]成分は，以下の式で計算できる．

$$\mathbf{C}_{rc} = \sum_{i=0}^{\mathrm{WIDTH}-1} \mathbf{A}_{ri} \times \mathbf{B}_{ic}$$

*r*と*c*はそれぞれ行と列を表す．



行列の積を計算するCプログラム（2/2）

Matrix.cpp

```
void h_multiply(float *A, float *B, float *C)
{
    unsigned int r, c, i;
    float tmp;
    for (r = 0; r < WIDTH; r++) {
       for (c = 0; c < WIDTH; c++) {
            tmp = 0.0;
            for(i = 0; i < WIDTH; i++)
                tmp += A[WIDTH * r + i] * B[WIDTH * i + c];
            C[WIDTH * r + c] = tmp;
        }
    }
}
```

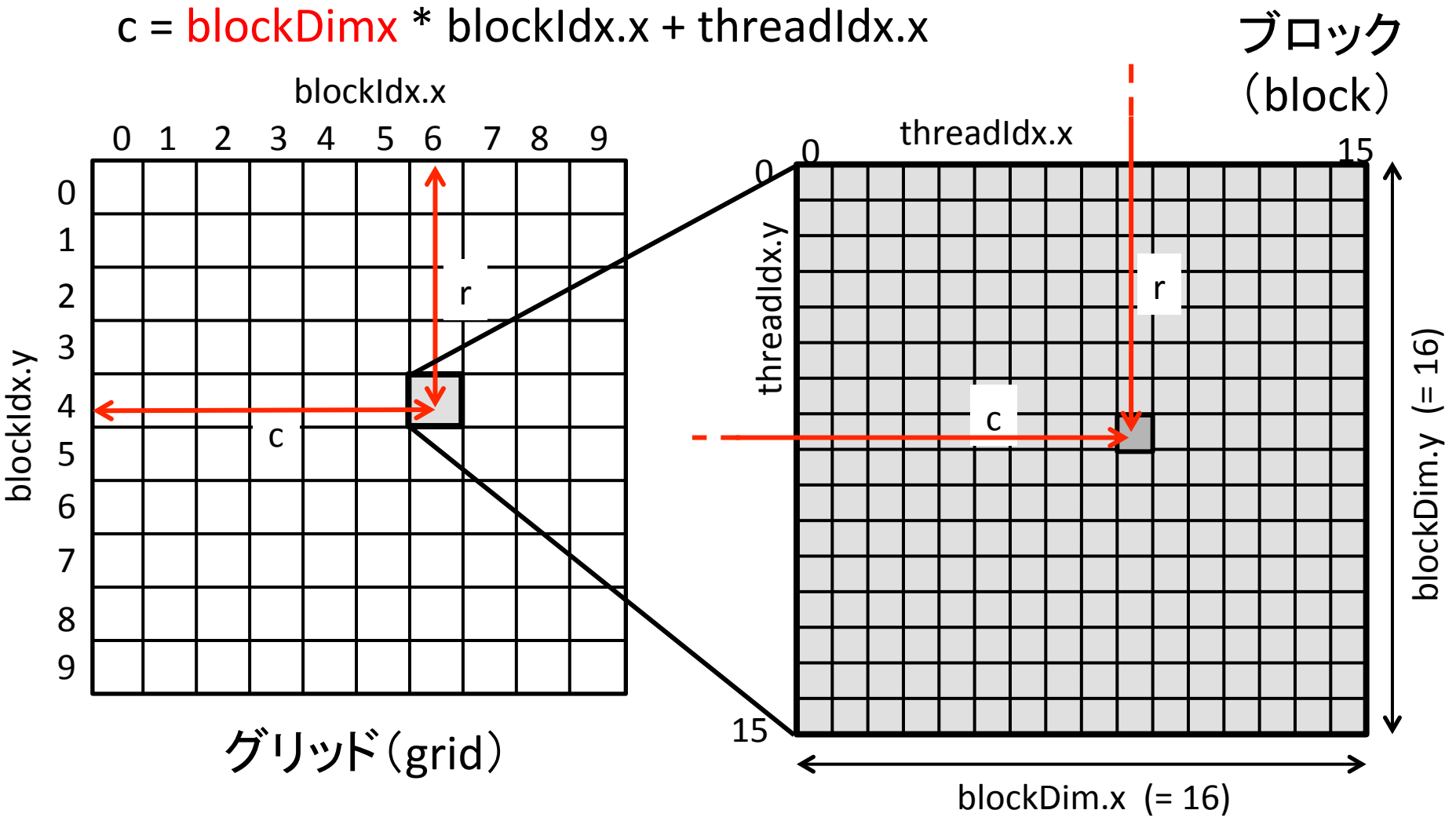
$$C_{rc} = \sum_{i=0}^{WIDTH-1} A_{ri} \times B_{ic}$$ 行列Cのr行c列要素の計算

大きな行列の取り扱い

- 行列を16 x 16のサイズのブロックのグリッドとして扱う.

  r = blockDim.y * blockIdx.y + threadIdx.y,

  c = blockDimx * blockIdx.x + threadIdx.x



行列の積を計算するCUDAプログラム（1/4）

- CUDAを使う場合には，以下のヘッダーファイルが必要．

  #include <cuda_runtime.h>

Matrix.cu

```
#include <stdio.h>

#include <cuda_runtime.h>

#define BLOCK 16 // 各ブロックは16×16個のスレッド.
#define WIDTH 1024 // 処理対象の行列のサイズはWIDTH×WIDTH.

// ホスト(CPU)側の行列定義.
float h_A[WIDTH * WIDTH];
float h_B[WIDTH * WIDTH];
float h_C[WIDTH * WIDTH];

// デバイス(GPU)側の行列へのポインタ.
float *d_A, *d_B, *d_C;
```

← デバイスメモリー上の配列をであることを明示するため，先頭に”d_”を付す

行列の積を計算するCUDAプログラム(2/4)

```
// メイン関数.
int main()
{
    unsigned int i;

    // デバイス側に行列用のメモリを確保.
  cudaMalloc((void**)&d_A, sizeof(float) * WIDTH * WIDTH);
    cudaMalloc((void**)&d_B, sizeof(float) * WIDTH * WIDTH);
    cudaMalloc((void**)&d_C, sizeof(float) * WIDTH * WIDTH);

    // ホスト側の行列に値をセット.
  for (i = 0; i < (WIDTH * WIDTH); i++) {
        h_A[i] = (float)i;
        h_B[i] = (float)i;
    }

    // ホスト側の行列のデータをデバイス側の行列へ転送.
  cudaMemcpy(d_A, h_A, sizeof(float) * WIDTH * WIDTH,
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, sizeof(float) * WIDTH * WIDTH,
        cudaMemcpyHostToDevice);
```

デバイスメモリー上に配列を確保

ホストメモリー上の配列からデバイスメモリーの対応する配列へコピー

行列の積を計算するCUDAプログラム(3/4)

Matrix.cu

```
  // グリッドとブロックの定義.
  dim3 grid(WIDTH / BLOCK, WIDTH / BLOCK);
  dim3 block(BLOCK, BLOCK);

  // GPU処理の起動.
  d_multiply0 <<< grid, block >>> (d_A, d_B, d_C);

  // 計算結果はd_Cに格納されているので, それをホスト側のh_Cへ転送.
  cudaMemcpy(h_C, d_C, sizeof(float) * WIDTH * WIDTH,
        cudaMemcpyDeviceToHost);

  // デバイス側のメモリを解放.
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

ブロックとグリッドの定義
WIDTHがBLOCKで割り切れることを仮定

カーネル関数(後述)の呼び出し

計算結果をホストメモリー上の配列へコピー

- BLOCKの大きさは16の倍数がよい(メモリーアクセスが高速).
- WIDTHがBLOCKで割り切れない場合には, 処理が少し複雑になる. あとで議論.

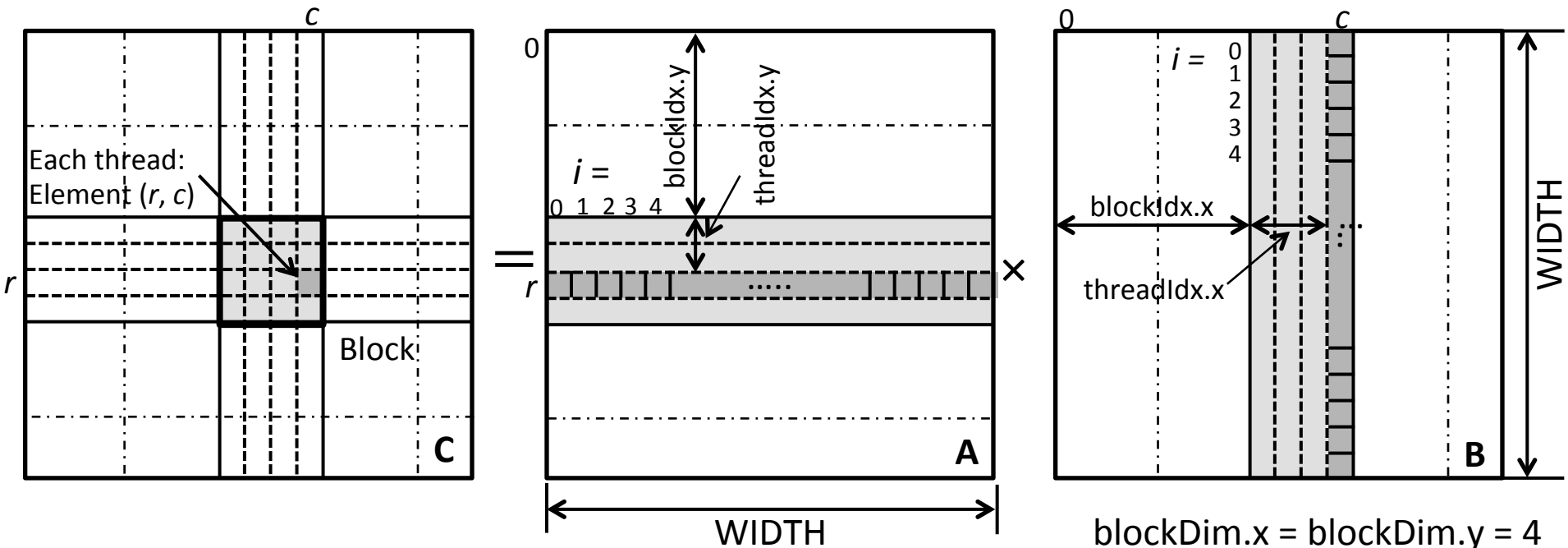
行列の積を計算するCUDAプログラム（4/4）

Matrix.cu

```
__global__ void d_multiply0(float *A, float *B, float *C)
{
        unsigned int r = blockDim.y * blockIdx.y + threadIdx.y;
        unsigned int c = blockDim.x * blockIdx.x + threadIdx.x;
        unsigned int i;
        float tmp;
        tmp = 0.0f;
        for (i = 0; i < WIDTH; i++)
            tmp += A[WIDTH * r + i] * B[WIDTH * i + c];
        C[WIDTH * r + c] = tmp;
    }
}
```

A, B, Cはデバイスメモリー上の配列へのポインタ.

行列Cのr行c列要素を計算するスレッド



blockDim.x = blockDim.y = 4

計算結果


C:¥Windows¥system32¥cmd.exe
デバイス計算時間:　34.094693(ms)　デバイス計算結果: 563314824314880.000000
ホスト計算時間: 2502.777966(ms)　ホスト計算結果: 563314824314880.000000
続行するには何かキーを押してください・・・
GPU計算は74倍高速.

WIDTHがBLOCKで割り切れない時（1/4）

- WIDTHがBLOCKで割り切れない場合はどうすべきか.
- WIDTH=1023，BLOCK=16の場合．r（行）とc（列）の範囲は0～1022.

```
dim3 grid(WIDTH/BLOCK, WIDTH/BLOCK);
dim3 block(BLOCK, BLOCK);

unsigned int r = blockDim.y * blockIdx.y + threadIdx.y;
unsigned int c = blockDim.x * blockIdx.x + threadIdx.x;
```

- BLOCK / WIDTH = 63．つまりgrid(63, 63)，block(16, 16)となる.
- 63 x 16 = 1008なので，r（行）やc（列）の範囲は0～1007となり，1008～1022番目の行や列をカバーできない.

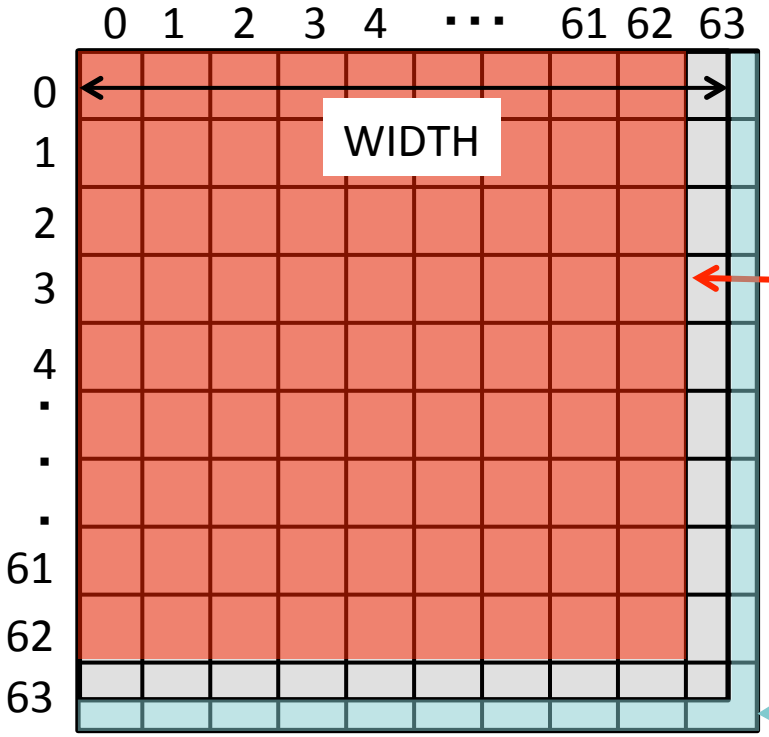
WIDTHがBLOCKで割り切れない時（2/4）

```
dim3 grid(WIDTH/BLOCK+1, WIDTH/BLOCK+1);
dim3 block(BLOCK, BLOCK);
```

- 今度はblockIdx.xとblockIdx.yが0～64となるので，rとcの値の範囲は0～1023の値となる．
- 配列のサイズ（＝WIDTH）が1023のとき，r（行）とc（列）の範囲は0～1022のはず．今度はrやcが行や列の範囲を超える場合が発生する．

WIDTHがBLOCKで割り切れない場合（3/4）

- WIDTH=1023，BLOCK=16の場合．r（行）とc（列）の範囲は0～1022.



グリッド（grid）

```
dim3 grid(WIDTH / BLOCK, WIDTH / BLOCK);
  = grid(63, 63)
dim3 block(BLOCK, BLOCK);
```

WIDTHがBLOCKで割り切れない場合には，生成されるスレッドは左の図中の赤い部分となる．つまりスレッドが不足する．

```
dim3 grid(WIDTH / BLOCK + 1, WIDTH / BLOCK + 1);
  = grid(64, 64)
dim3 block(BLOCK, BLOCK);
```

今度はブロックが1つずつ余分に生成されるので，スレッドが左の図中の青い部分まで生成され行列を範囲を超えてしまう．

WIDTHがBLOCKで割り切れない時（1/2）

- 生成されるブロック数を，以下のように1ずつ増やす．

```
dim3 grid(WIDTH/BLOCK+1, WIDTH/BLOCK+1);
dim3 block(BLOCK, BLOCK);
```

- 配列のサイズ（＝WIDTH）が1023のとき，r（行）とc（列）の範囲は0～1022のはず．rやcが行や列の範囲を超える場合が発生するので，事前チェック．

```
__global__ void d_multiply0(float *A, float *B, float *C)
{
    unsigned int r = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int c = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int i;
    float tmp;
    if ((r < WIDTH) && (c < WIDTH)) {
        tmp = 0.0f;
        for (i = 0; i < WIDTH; i++)
            tmp += A[WIDTH * r + i] * B[WIDTH * i + c];
        C[WIDTH * r + c] = tmp;
    }
}
```