CUDAを用いた数値解析の高速化

千葉工業大学
情報科学研究科　情報科学専攻
前川研究室
富永浩文

本日の公演内容

GPUの使い方

CUDAの初歩的なプログラミング方法について

本日の公演では、詳細な最適化方法などの解説は行いません

少ない手間で高い高速化率が得られるCUDAの魅力を伝えて少しでも伝えられればと思います

公演プログラム

1. GPUについて

2. 簡単にGPUが使えます！
   （cuBLASやOpenACCで簡単プログラミング）

3. CUDAプログラミング

4. CUDAプログラミング実践（熱伝導解析を例として）

GPUとは？

Graphics Processor Unit（GPU）
　画像処理などの行列演算を
　高速に処理できる専用のプロセッサ

©NVIDIA Corporation

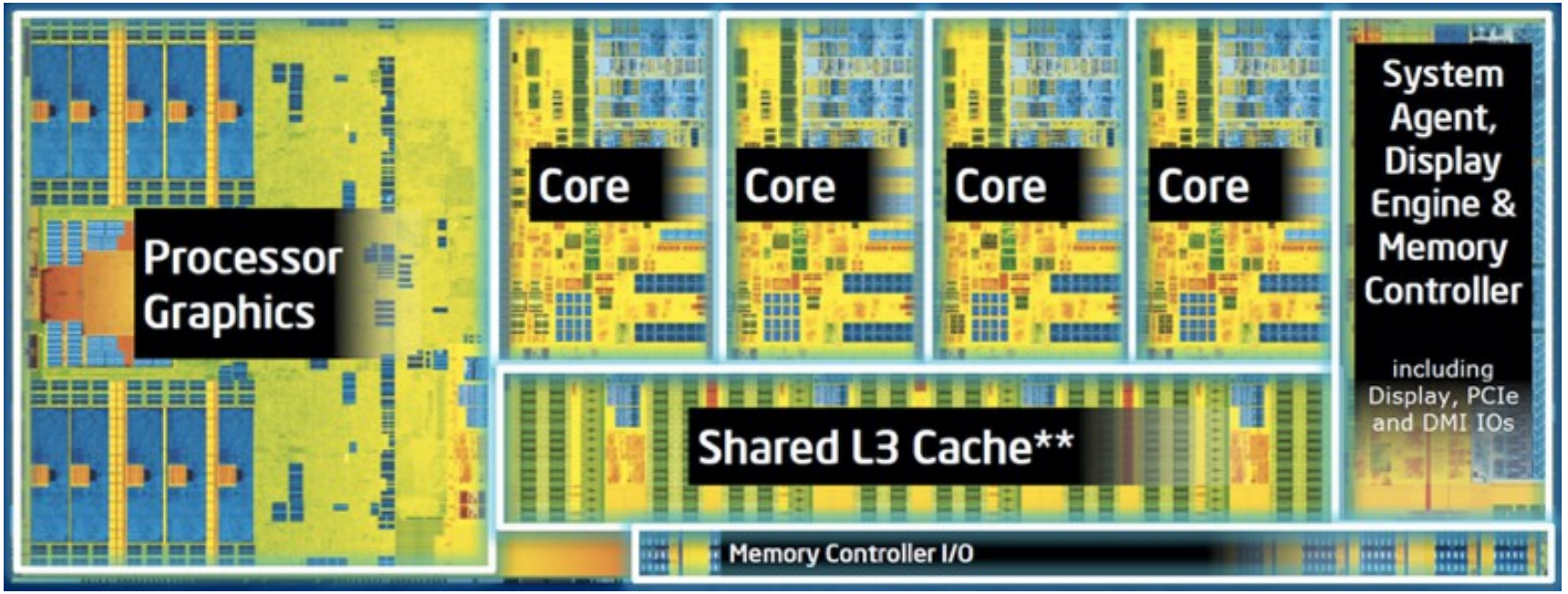
Single Instruction Multiple Thread(SIMT)アーキテクチャ
　データ並列性を利用するアーキテクチャ

大量のデータに同一の演算を行うことに特化

GPUとは？

Intel Corei7 Processor
　→8Core, 4Core, 2Core

最新のCPUにおいて単精度演算性能で数百GFLOPS



©Intel Corporation

GPUとは？

●Volta Architecthre

Cuda Core:5120

©NVIDIA Corporation

GPUとは？

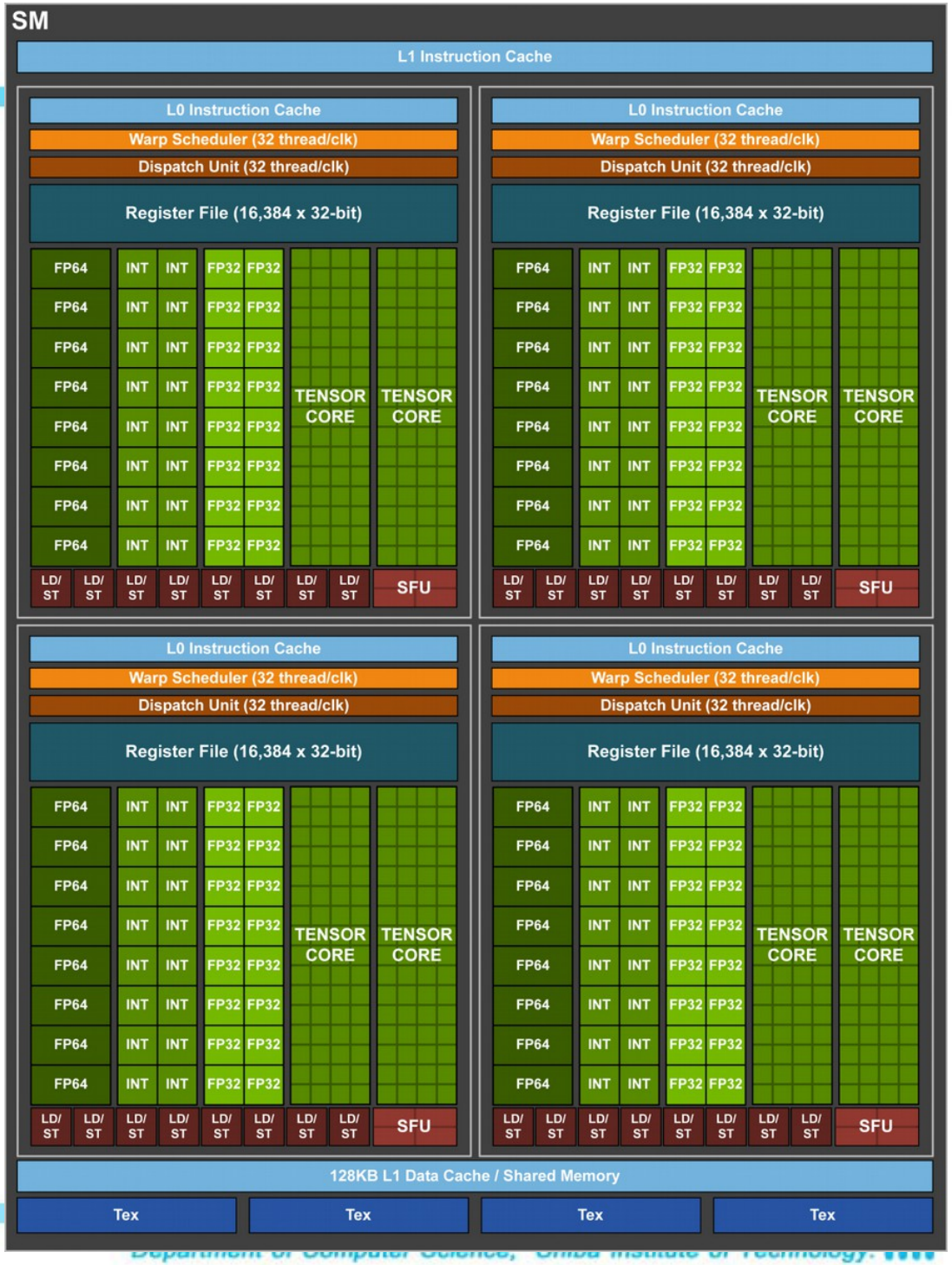
●**Volta Architecthre**

Cuda Core:5120

単精度浮動小数点演算性能
数十TFLOPS

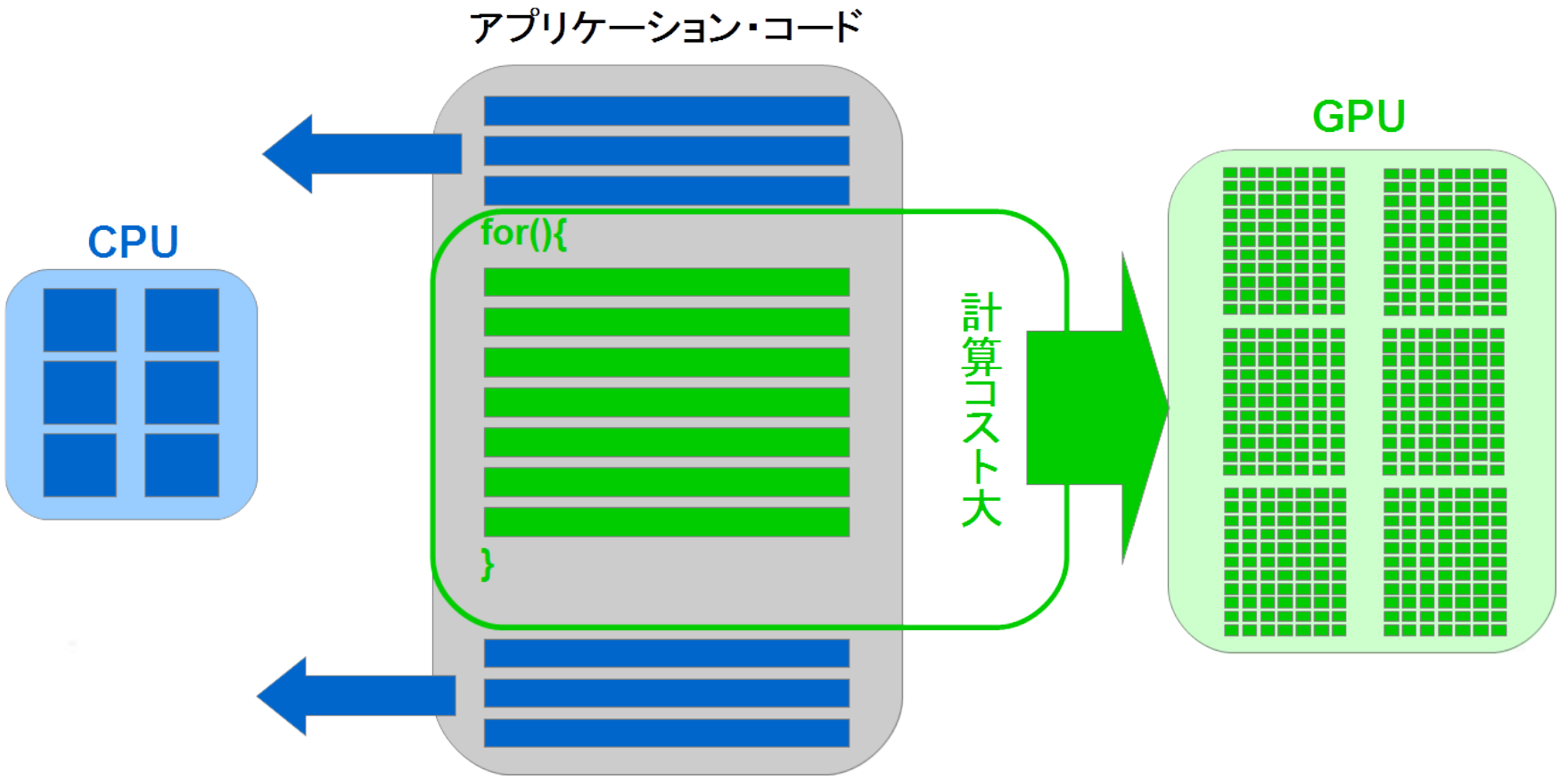
多くの軽量なコアを持つ
ハイスループットなプロセッサ

ストリーミングマルチプロセッサ
全てのプロセッサが
非同期で動作



GPUコンピューティング

逐次実行をCPUで実行し、並列化部分をGPUで実行



既存の数値計算アプリーケーションをGPU化

数値計算アプリケーション

LIBRAY	OpenACC	CUDA
簡単に始められる！ BLASなどのライブラリを CUDA対応ライブラリに 置き換え	簡単に高速化！ CPUに対応する 既存のコードに ディレクティブ構文を挿入	高い自由度！ 主要な計算を CUDAで記述

既存の数値計算アプリーケーションをGPU化

数値計算アプリケーション

LIBRAY	OpenACC	CUDA
簡単に始められる！ BLASなどのライブラリを CUDA対応ライブラリに 置き換え	簡単に高速化！ 既存のCPUコードに ディレクティブ構文を挿入	高い自由度！ 主要な計算を CUDAで記述

LIBRARY BLASとcuBLAS

Basic Linear Algebra Subprogranms（BLAS）
　ベクトルや行列の基本演算を行う線形ライブラリ

数値計算の基本である行列積や行列ベクトル積などを関数呼び出しで実行可能

GPUでも利用可能なライブラリ群
　NVIDIAであればcuBLAS、AMDであればclMath

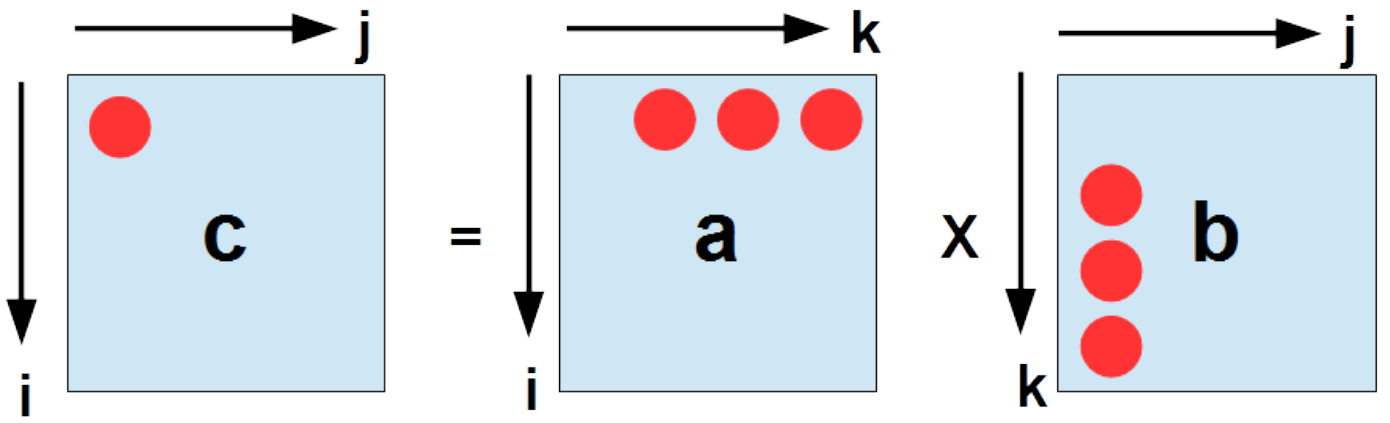
cuBLASは、NVIDIA社が提供するCUDA Toolkitに同封
　CUDA Toolkitをインストールすれば利用可能

NVIDIA社が提供するライブラリ以外にも多くのライブラリが提供

LIBRARYなしの行列積

行列-行列積（gemm）では？（C言語によるベーシックな例）

```
for(i=0;i<n;i<i++){
    for(j=0;j<n;j++){
        for(k=0;k<n;k++){
            c[i][j]=a[i][k]+b[k][j];
        }
    }
}
```

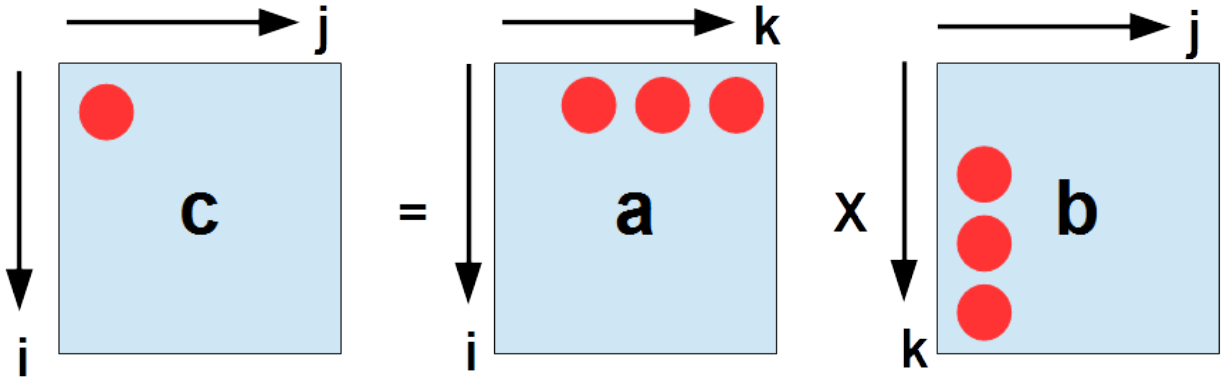


LIBRARY BLASの行列積

行列-行列積（gemm）では？（BLASを用いた例）

```
dgemm_("N", "N", &m, &n, &k, &alpha, a, &lda, b, &ldb, &beta, c, &ldc);

/*
1:行列の転置指定，2:行列の転置指定，3:行列aの行数，4:行列bの行数，
5:行列aの列数、行列bの行数，6:スカラーalpha，7:行列aのポインタ
8:行列aのleading dimension，9:行列bのポインタ，
10:行列bのleading dimension，　11:スカラーbeta，　12:行列cのポインタ
13:行列cのleading dimension
*/
```

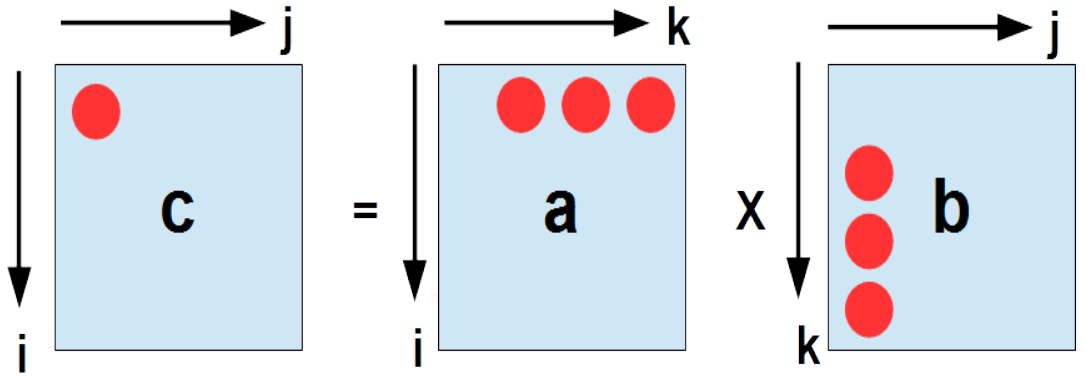


LIBRARY cuBLASの行列積

行列-行列積（gemm）では？（cuBLASを用いた例）

```
cublasDgemm(handle, transa, transb, m, n, k, &alpha, &A, lda, &B, ldb, &beta, &C, ldc)

/*
1:cublasのhandle, 2:行列aの転置指定，3:行列bの転置指定，4:行列aの行数,
5:行列bの行数, 6:行列aの列数、行列bの行数，7スカラーalpha，8:行列aのポインタ
9:行列aのleading dimension，10:行列bのポインタ,
11:行列bのleading dimension，　12:スカラーbeta，　13:行列cのポインタ
14:行列cのleading dimension
*/
```



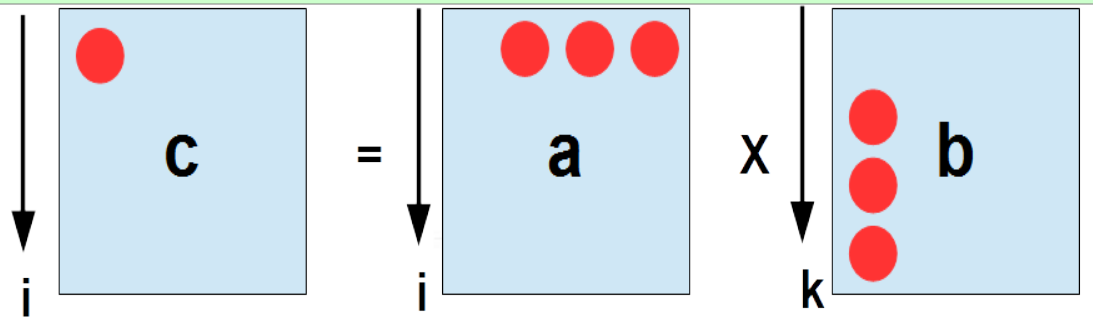
LIBRARY cuBLASの行列積

行列-行列積（gemm）では？（cuBLASを用いた例）

```
cublasDgemm(handle, transa, transb, m, n, k, &alpha, &A, lda, &B, ldb, &beta, &C, ldc)

/*
1:cublasのhandle, 2:行列aの転置指定，3:行列bの転置指定，4:行列aの行数,
5:行列bの行数, 6:行列aの列数、行列bの行数，7スカラーalpha，8:行列aのポインタ
9:行列aのleading dimension，10:行列bのポインタ,
```

cuBLASは、この他にGPU上のメモリ操作の関数を組み合わせる
配列a,b,cの領域を、GPU上のメモリ領域に確保する専用のmalloc関数と
GPU上へデータをコピーする専用のmemcpy関数



LIBRARY cuBLASの行列積

行列-行列積（gemm）では？（cuBLASを用いた例）

```
cublasDgemm(handle, transa, transb, m, n, k, &alpha, &A, lda, &B, ldb, &beta, &C, ldc)

/*
1:cublasのhandle, 2:行列aの転置指定，3:行列bの転置指定，4:行列aの行数,
5:行列bの行数, 6:行列aの列数、行列bの行数，7スカラーalpha，8:行列aのポインタ
9:行列aのleading dimension，10:行列bのポインタ,
```

cuBLASは、この他にGPU上のメモリ操作の関数を組み合わせる
配列a,b,cの領域を、GPU上のメモリ領域に確保する専用のmalloc関数と
GPU上へデータをコピーする専用のmemcpy関数

既存のBLASライブラリのプログラムから少ないコストでCUDA化可能！
CUDAのライブラリは最適化されており高速に実行可能

既存の数値計算アプリーケーションをGPU化

数値計算アプリケーション

LIBRAY

OpenACC

CUDA

簡単に始められる！

BLASなどのライブラリを
CUDA対応ライブラリに
置き換え

簡単に高速化！

既存のCPUコードに
ディレクティブ構文を挿入

高い自由度！

主要な計算を
CUDAで記述

OpenACC

Open Accelerator（OpenACC）
　ディレクティブベースの並列化プログラミング

異種プロセッサのプログラミング環境において並列化を容易に実現
　CPUやGPU、FPGAなどの環境

そもそもディレクティブベースの並列化プログラミングとは？
　プログラムに並列化のヒントを記述することでコンパイラが記述を解釈し並列化

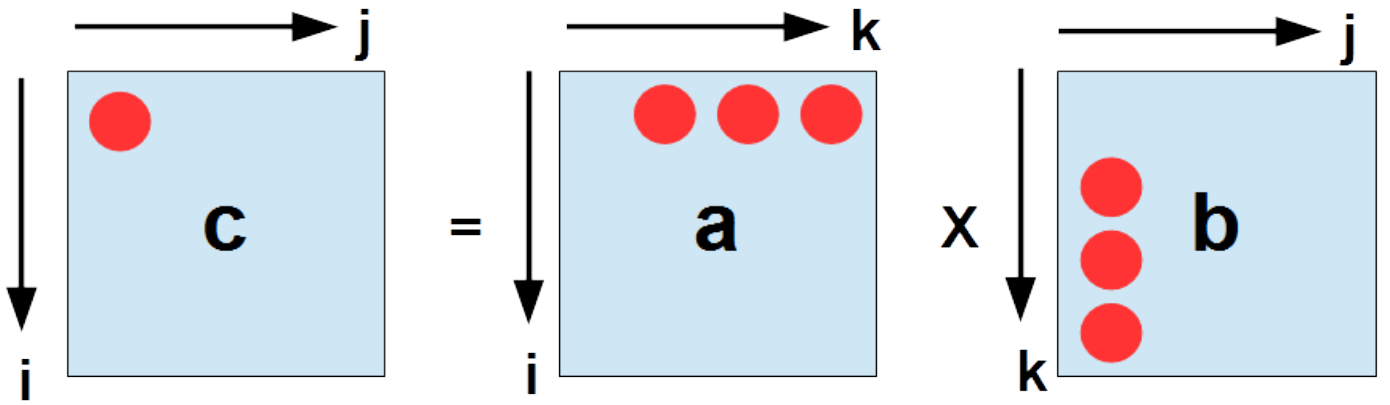
代表的なディレクティブベースの並列化プログラミング
　OpenMP

CUDA対応のOpenACCはNVIDIA社がPGI社の開発キットを配布
　フリーで利用可能（実行環境に一部制限がある）

ディレクティブベースのプログラミング（行列積：OpenMP）

行列-行列積（gemm）では？（OpenMPの例）

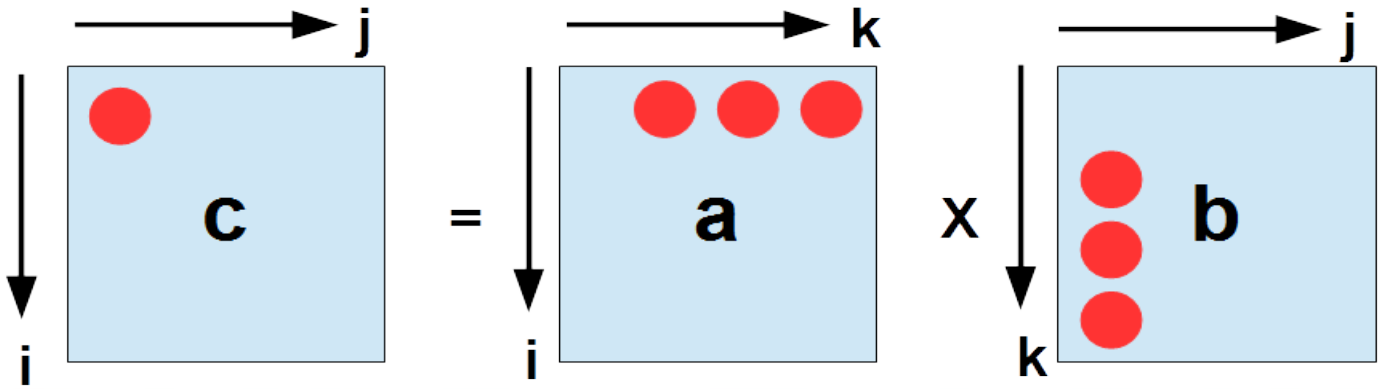
```
#pragma omp parallel for private(j,k)
for(i=0;i<n;i<i++){
    for(j=0;j<n;j++){
        for(k=0;k<n;k++){
            c[i][j]=a[i][k]+b[k][j];
        }
    }
}
```



ディレクティブベースのプログラミング（行列積：OpenACC）

行列-行列積（gemm）では？（OpenACCの例）

```
#pragma acc kernels present(a,b,c)
#pragma acc loop independent
for(i=0;i<n;i<i++){
    #pragma acc loop independent
     for(j=0;j<n;j++){
        #pragma acc loop independent
         for(k=0;k<n;k++){
             c[i][j]=a[i][k]+b[k][j];
         }
     }
}
```

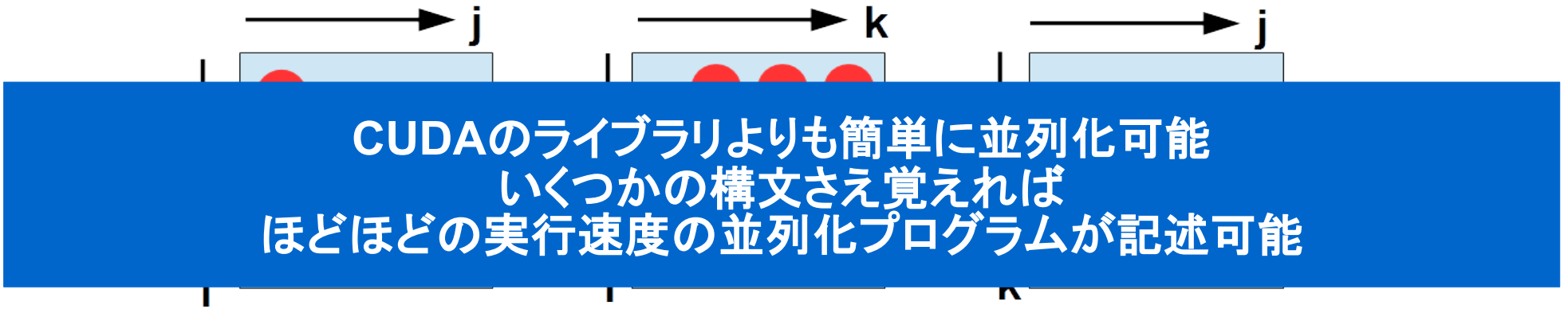


ディレクティブベースのプログラミング（行列積：OpenACC）

行列-行列積（gemm）では？（OpenACCの例）

```
#pragma acc kernels present(a,b,c)
#pragma acc loop independent
for(i=0;i<n;i<i++){
    #pragma acc loop independent
     for(j=0;j<n;j++){
        #pragma acc loop independent
         for(k=0;k<n;k++){
```

OpenACCは、並列化を行う箇所の直前におまじない文を追加
GPU上のメモリ確保などを全てコンパイラが行う



CUDAのライブラリよりも簡単に並列化可能
いくつかの構文さえ覚えれば
ほどほどの実行速度の並列化プログラムが記述可能

既存の[illegible]計算アプリーケーションをGPU化

[illegible]算アプリケーション

LIBRAY

[illegible]

CUDA

簡単に始められる！

BLASなどのライブラリを
CUDA対応ライブラリに
置き換え

簡単に高速化！

既存のCPUコードに
ディレクティブ構文を挿入

[illegible]！

[illegible]を
CU[illegible]記述

ライブラリやOpenACCなどのディレクティブベースのプログラミングを
使えば、GPUプログラミングは全て解決！

既存の[illegible]計算アプリーケーションをGPU化

[illegible]算アプリケーション

LIBRAY

CUDA

ライブラリやOpenACCなどのディレクティ[illegible]
使えば、GPUプログラミング[illegible]を

**複雑なアルゴリズムの実装やアルゴリズム最適化を行うは
ライブラリやOpenACCでは困難**

簡単に始められる！

BLASなどのライブラリを
CUDA対応ライブラリに
置き換え

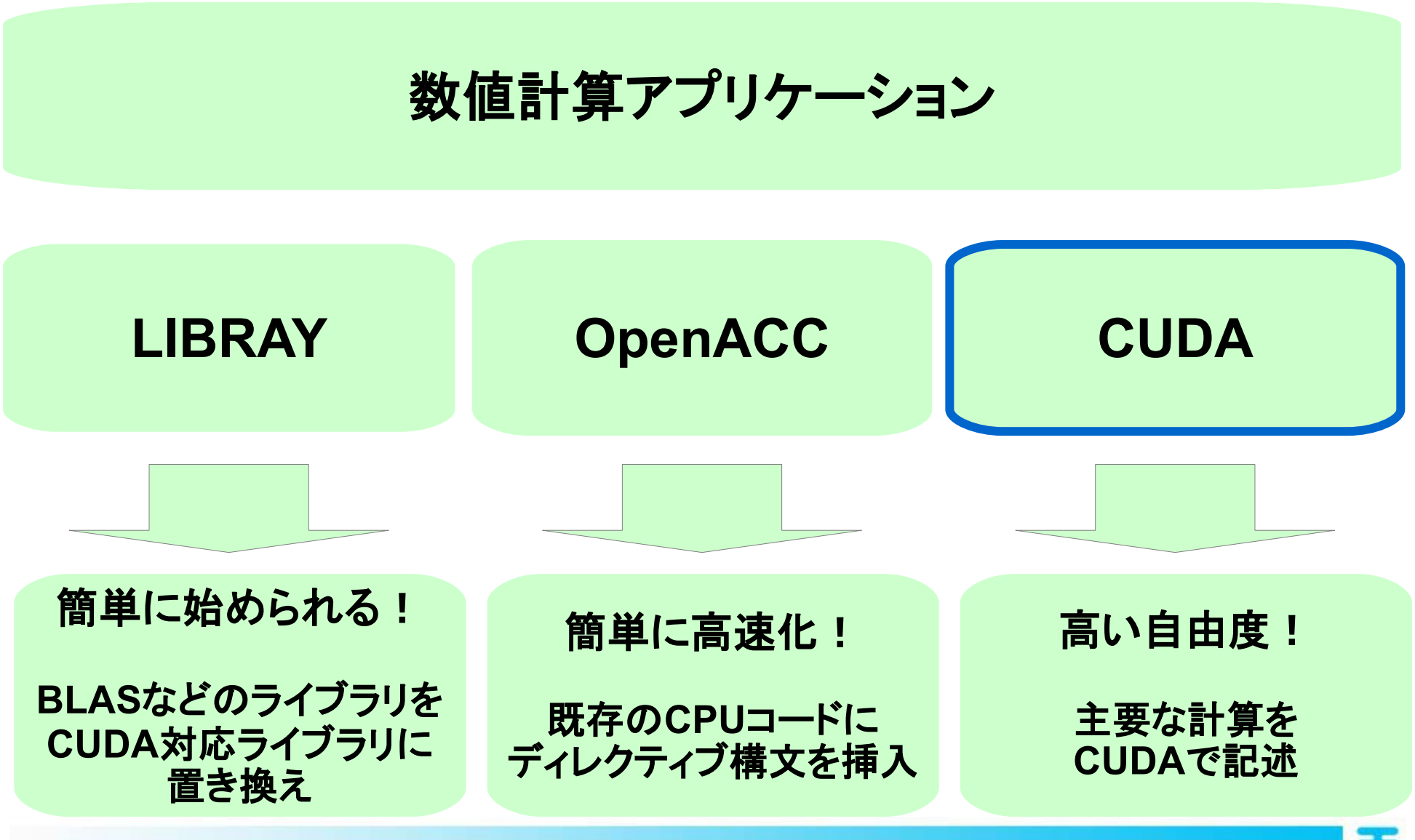
簡単に高速化！

既存のCPUコードに
ディレクティブ構文を挿入

[illegible]！

[illegible]を
CU[illegible]記述

既存の数値計算アプリーケーションをGPU化



Compute Unified Device Architecture（CUDA）

Compute Unified Device Architecture（CUDA）は
NVIDIA社が提供する並列コンピューティングプラットフォーム
　NVIDIA社のGPUプログラミング環境を提供

C++、Fortranを拡張した拡張言語によりプログラミングしGPUを利用
　その他にもJavaやPythonなどもある

多くのスレッドと特殊なメモリ環境を持つGPUコンピューティング
　効率良く利用するために最適化

CUDAを用いたGPUコンピューティング

©Intel Corporation

PCI-Express
Or
NVLink

©NVIDIA Corporation

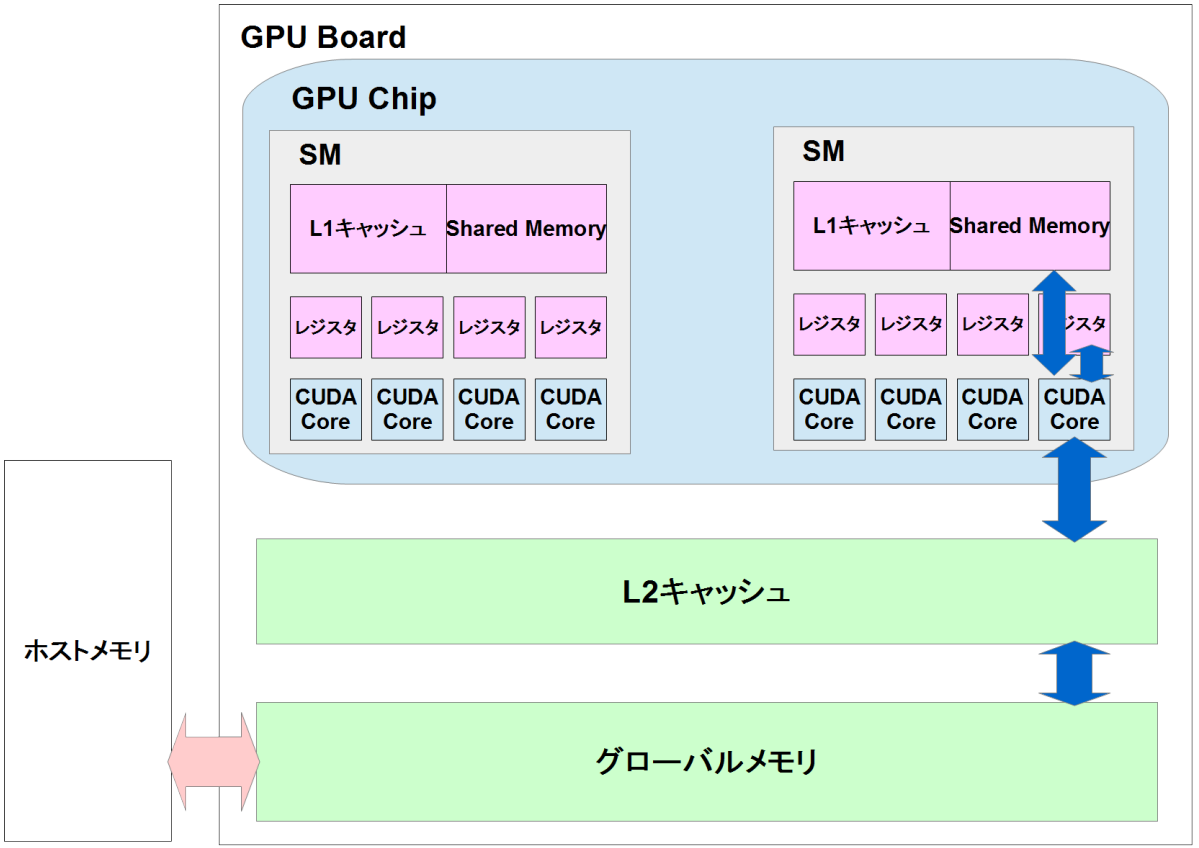
CPUとGPUはPCI-ExpressかNVLinkeなどのバスで接続
　異なるメモリ空間

NVIDIA社のGPUアーキテクチャ

GPUのチップ内は多数の計算コア（CUDA Core）があり、
Coreを複数まとめたStereming MultiProcessor(SM)で構成される

●**Volta Architecthre**
Cuda Core:5120
SM:80

●**Pascal Architecthre**
Cuda Core:3840
SM:56

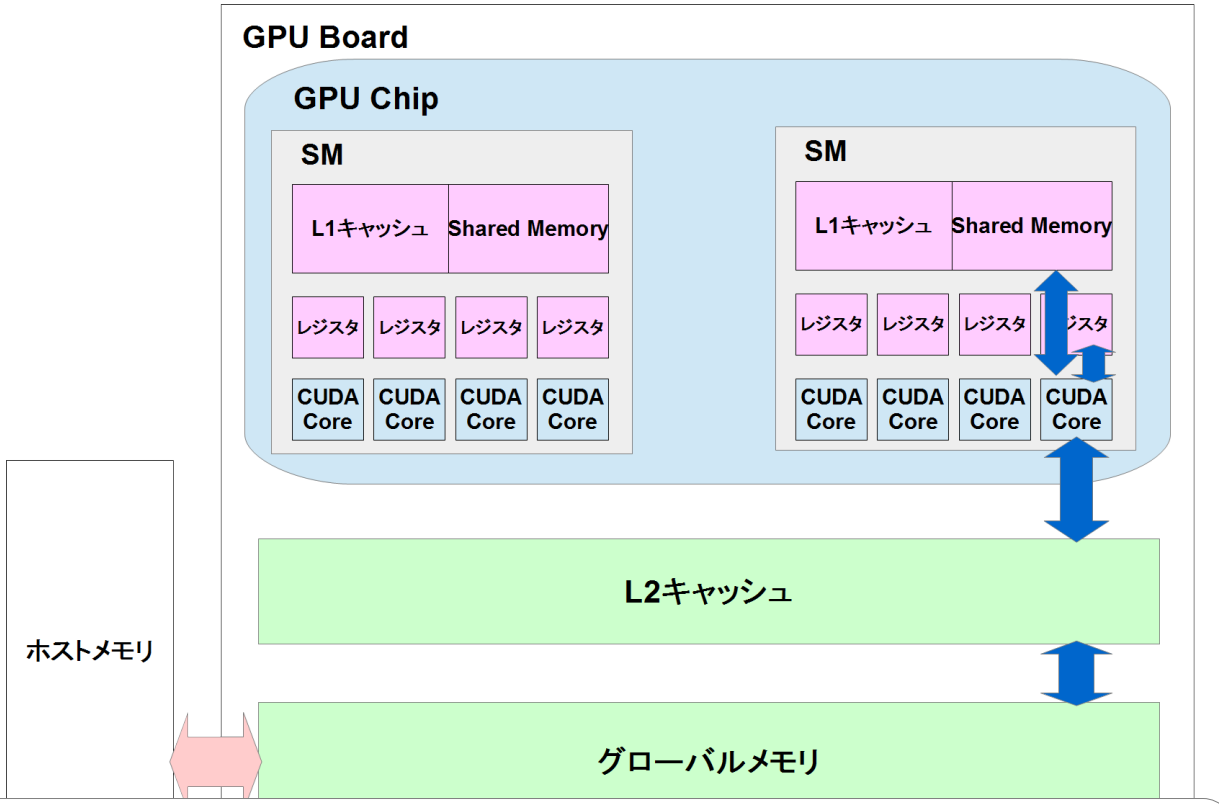


NVIDIA社のGPUアーキテクチャ

GPUのチップ内は多数の計算コア（CUDA Core）があり、
Coreを複数まとめたStereming MultiProcessor(SM)で構成される

●**Volta Architecthre**
Cuda Core:5120
SM:80

●**Pascal Architecthre**
Cuda Core:3840
SM:56



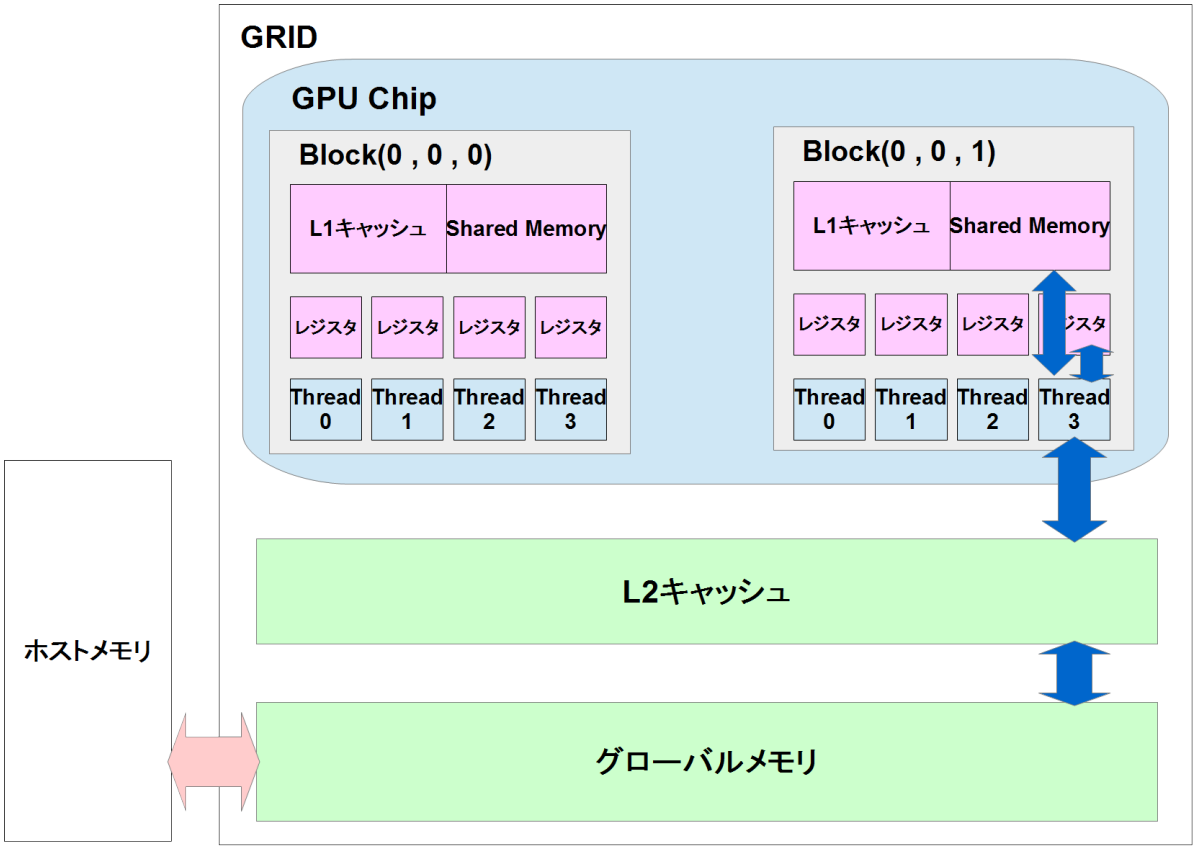
NVIDIA社の提供する様々なGPUアーキテクチャの違いを隠蔽し
最適な実行が可能

CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

メモリ階層
- グローバルメモリ
- シェアードメモリ
- レジスタ
- etc

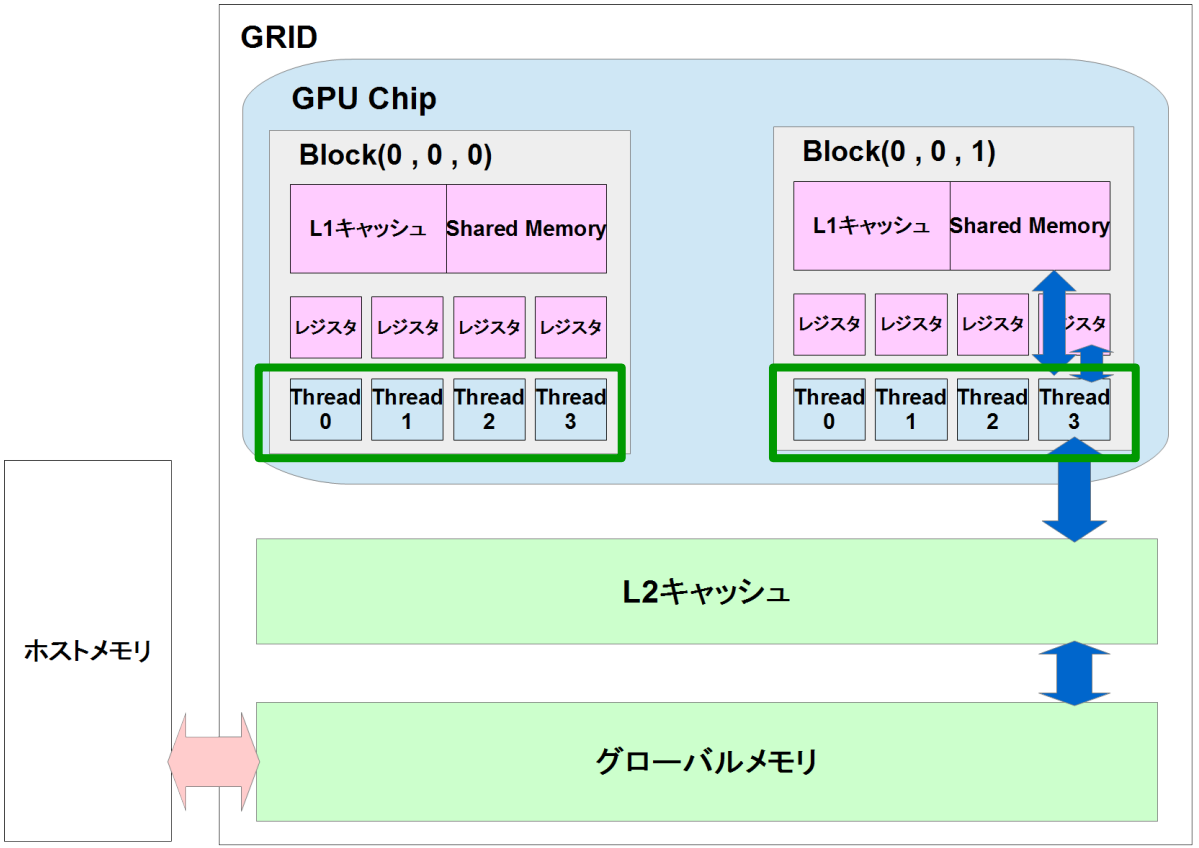


CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

メモリ階層
- グローバルメモリ
- シェアードメモリ
- レジスタ
- etc

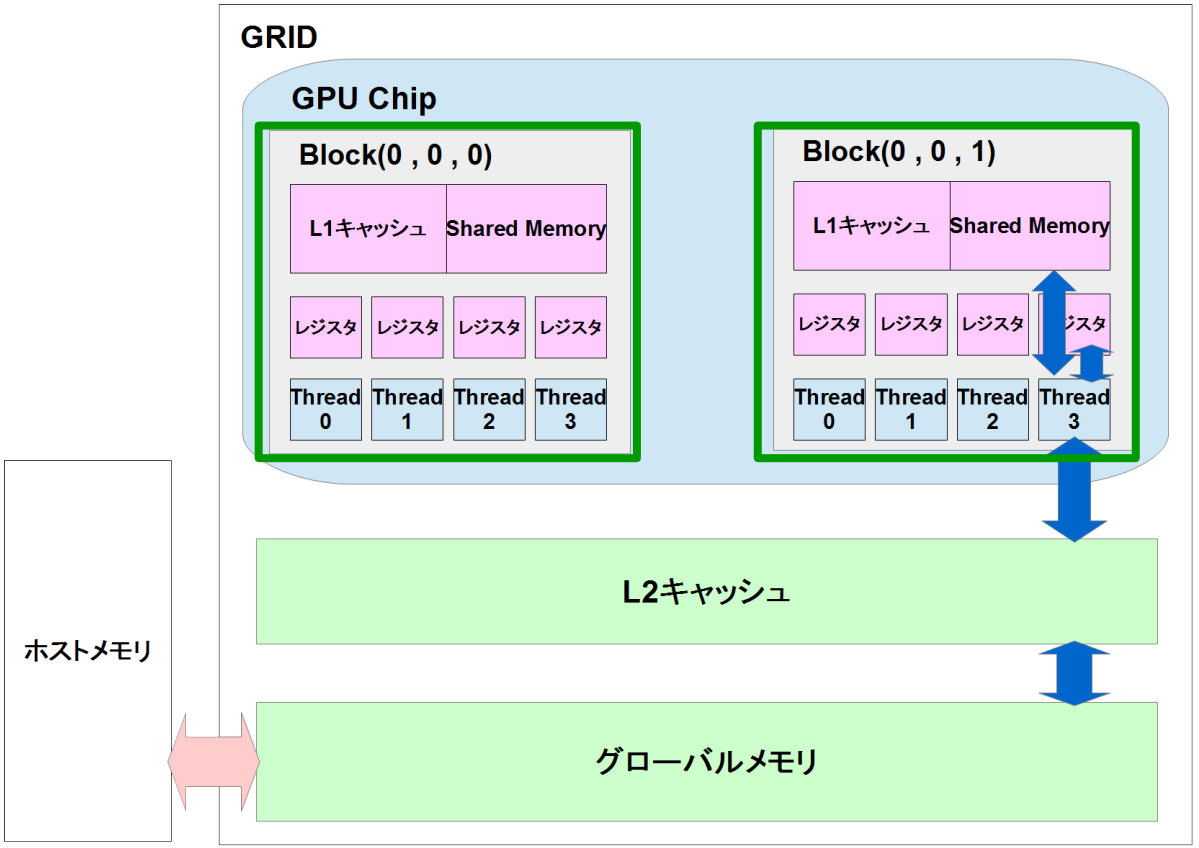


CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

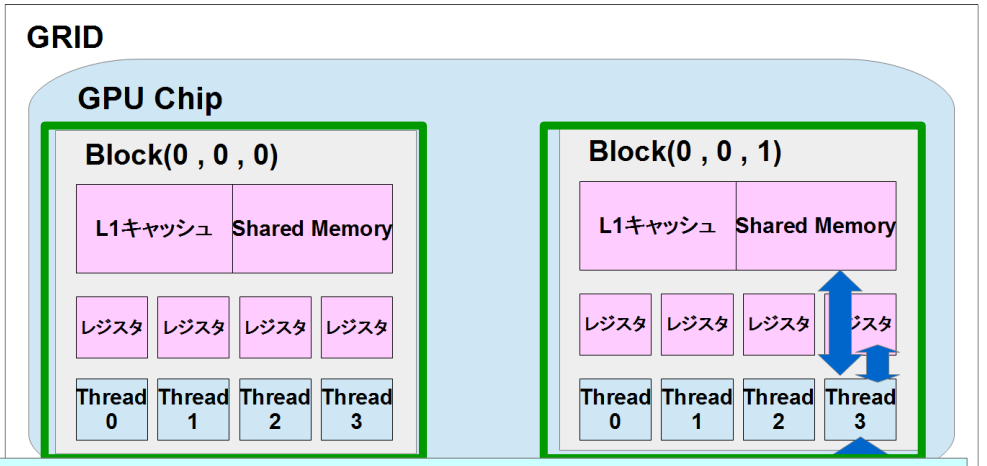
メモリ階層
- グローバルメモリ
- シェアードメモリ
- レジスタ
- etc



CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
・演算処理を行うスレッド
・スレッドを複数まとめた
スレッドブロック



スレッドブロック:
・指定したスレッド数の集合　・3次元ベクトルでサイズを指定(X,Y,Z)

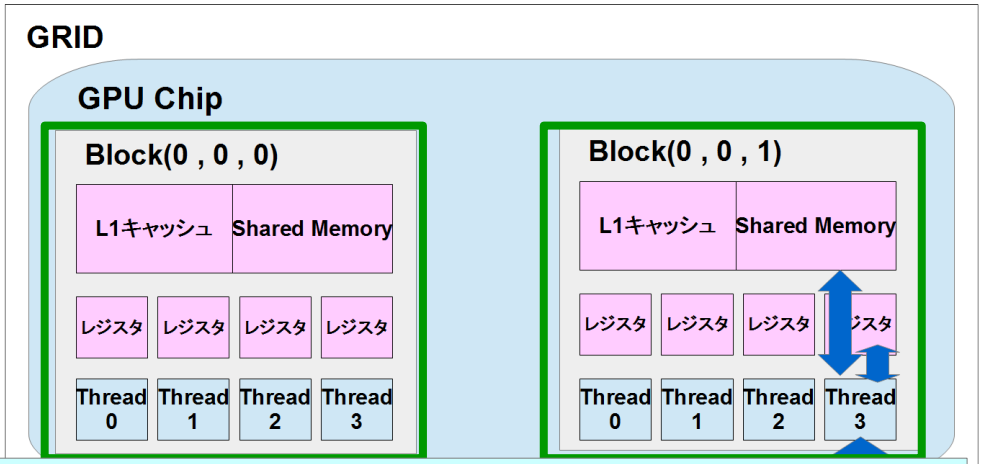
グリッド
・全スレッドブロックの集合　・2次元ベクトルでサイズを指定(X,Y)

スレッドID
・スレッドが担当する計算領域をスレッドブロックと位置、
スレッドブロックのグリッド内の位置により決定

CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
・演算処理を行うスレッド
・スレッドを複数まとめた
スレッドブロック



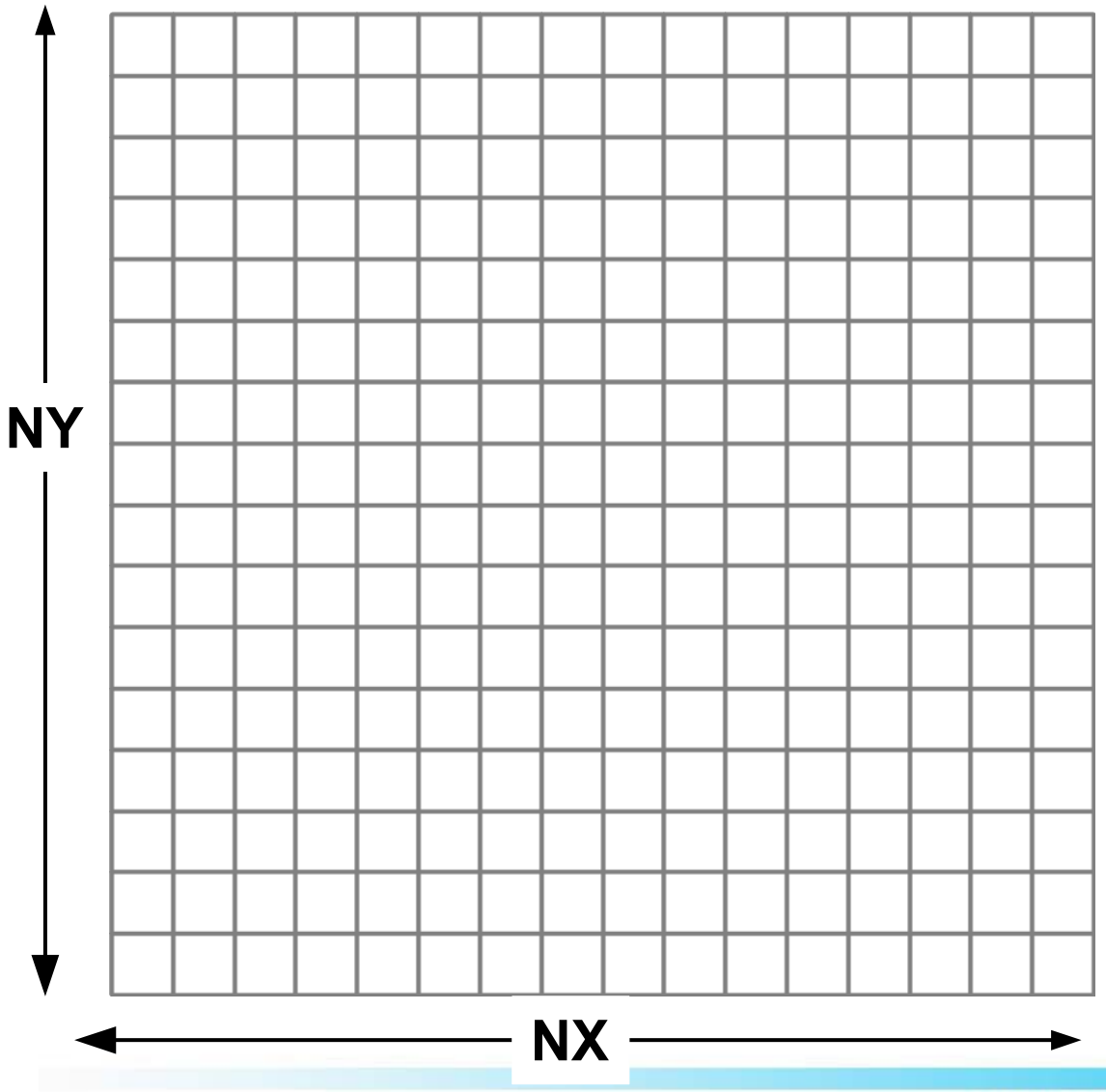
スレッドブロックID:
・blockIdx.x ・blockIdx.y ・blockIdx.z

グリッドID:
・gridDim.x ・gridDim.y

スレッドID
・threadIdx.x ・threadIdx.y

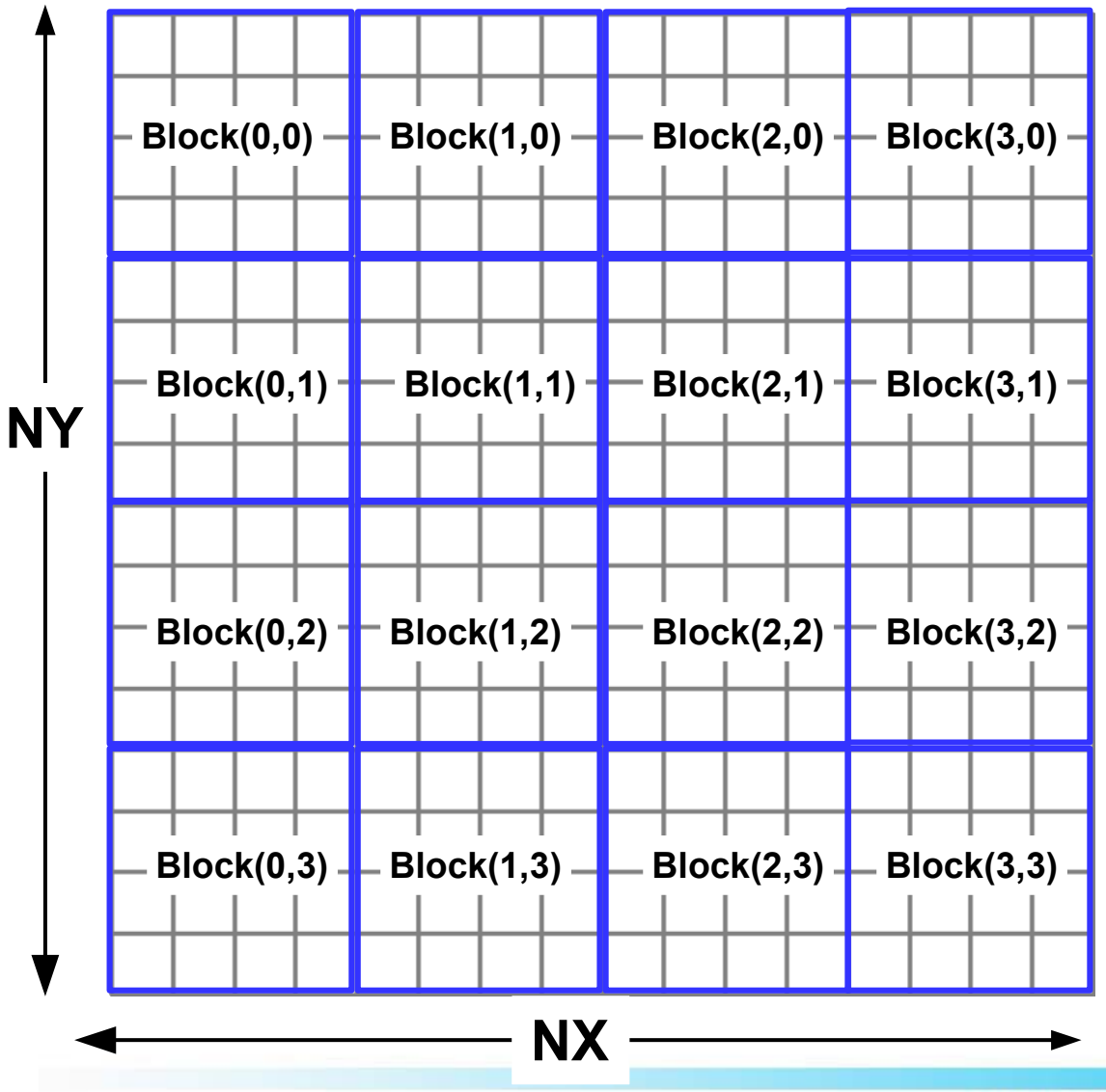
スレッドの担当位置の計算方法

スレッドブロックのサイズを(4,4,0)で指定　　1スレッドが1格子点を計算



スレッドの担当位置の計算方法

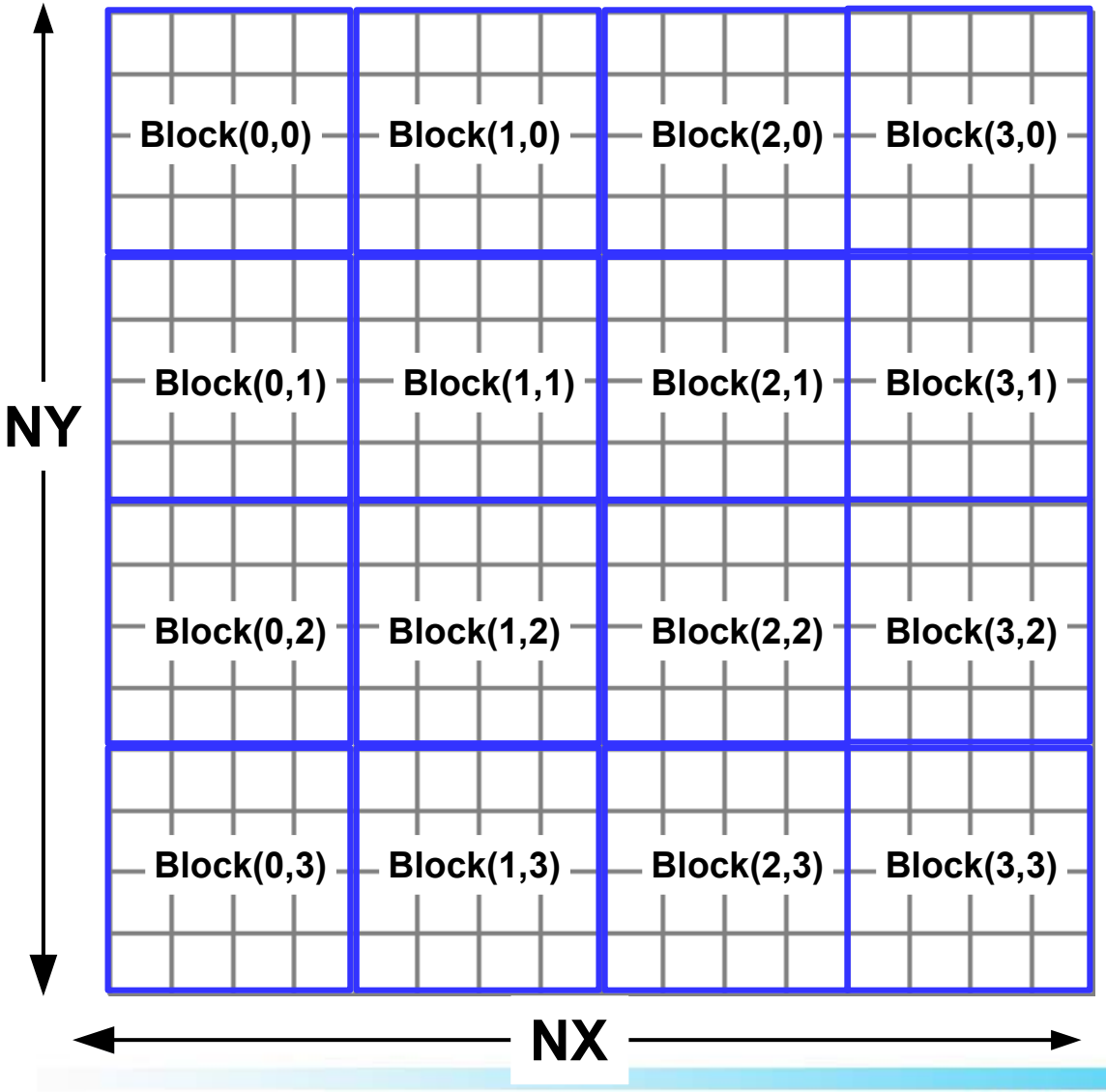
スレッドブロックのサイズを(4,4,0)で指定　　1スレッドが1格子点を計算



スレッドの担当位置の計算方法

スレッドブロックのサイズを(4,4,0)で指定

1スレッドが1格子点を計算



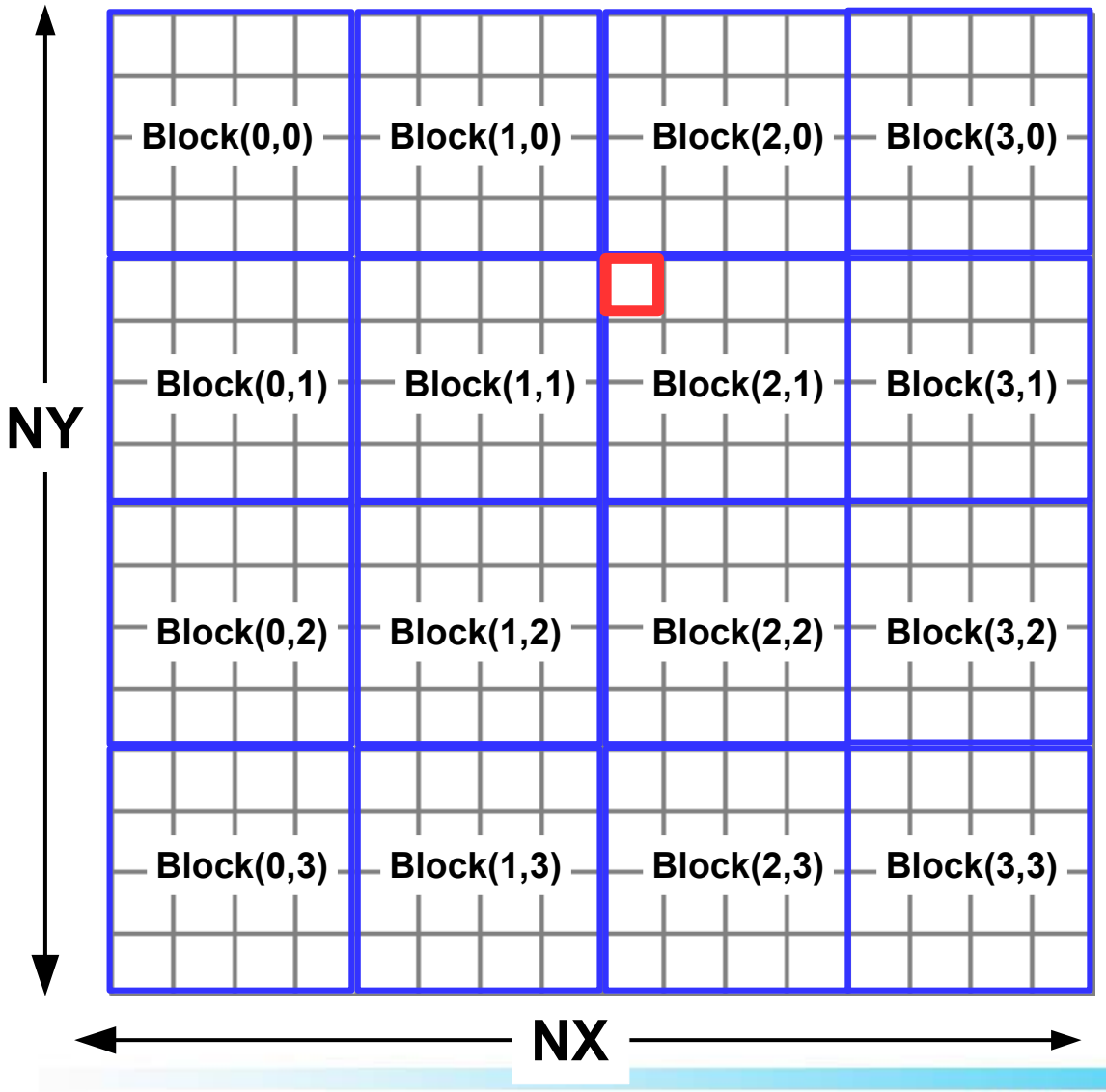
グリッドサイズが決定
（NX/4, NY/4）→gridDim（4,4）

スレッドブロックのIDが
一意に決定
（blockIdx.x, blockIdx.y）

スレッドの担当位置の計算方法

スレッドブロックのサイズを(4,4,0)で指定

1スレッドが1格子点を計算



グリッドサイズが決定
（NX/4, NY/4）→gridDim（4,4）

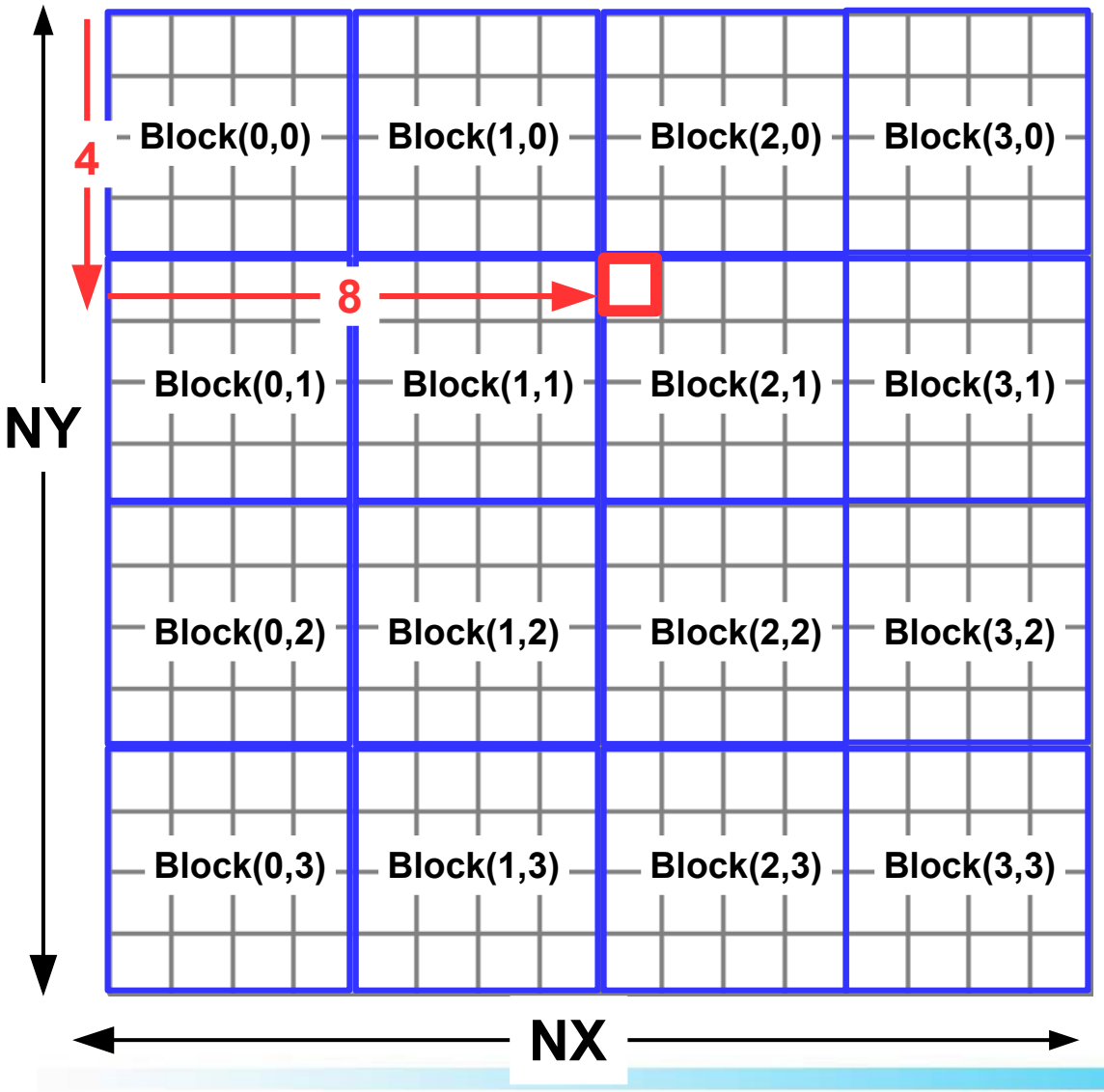
スレッドブロックのIDが
一意に決定
（blockIdx.x, blockIdx.y）

赤枠を担当するスレッドを指定
・NY方向の計算
gridDim.y*blockIdx.y+threadIdx.y
(4 * 1 + 0) = 4

・NX方向の計算
gridDim.x*blockIdx.x+threadIdx.x
(4 * 2 + 0) = 8

スレッドの担当位置の計算方法

スレッドブロックのサイズを(4,4,0)で指定



1スレッドが1格子点を計算

グリッドサイズが決定
（NX/4, NY/4）→gridDim（4,4）

スレッドブロックのIDが
一意に決定
（blockIdx.x, blockIdx.y）

赤枠を担当するスレッドを指定
・NY方向の計算
gridDim.y*blockIdx.y+threadIdx.y
(4 * 1 + 0) = 4

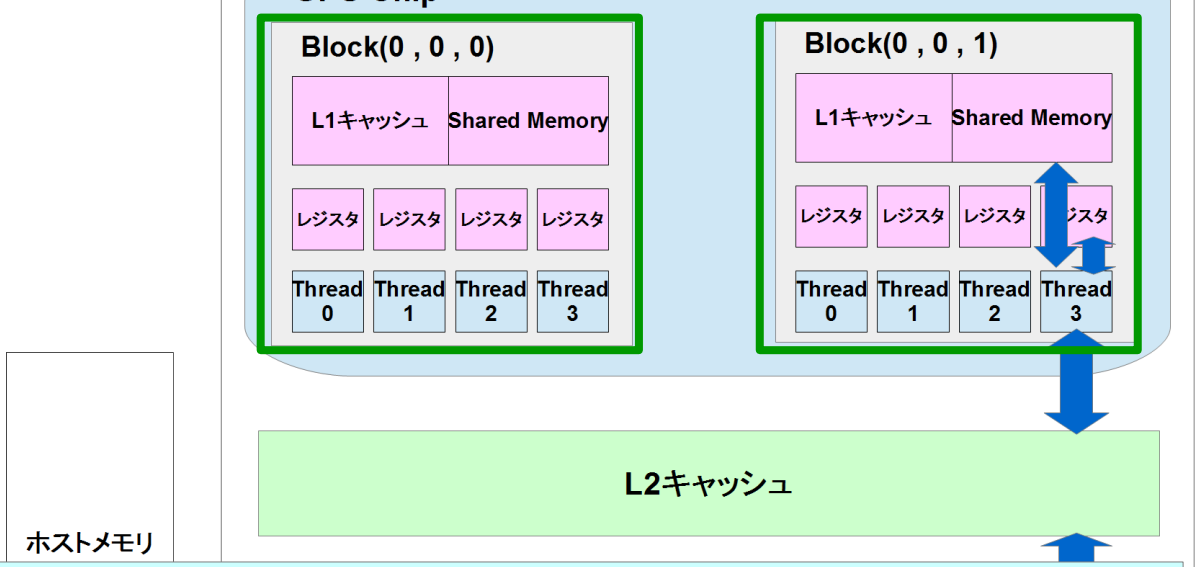
・NX方向の計算
gridDim.x*blockIdx.x+threadIdx.x
(4 * 2 + 0) = 8

CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

メモリ階層
- グローバルメモリ
- シェアードメモリ



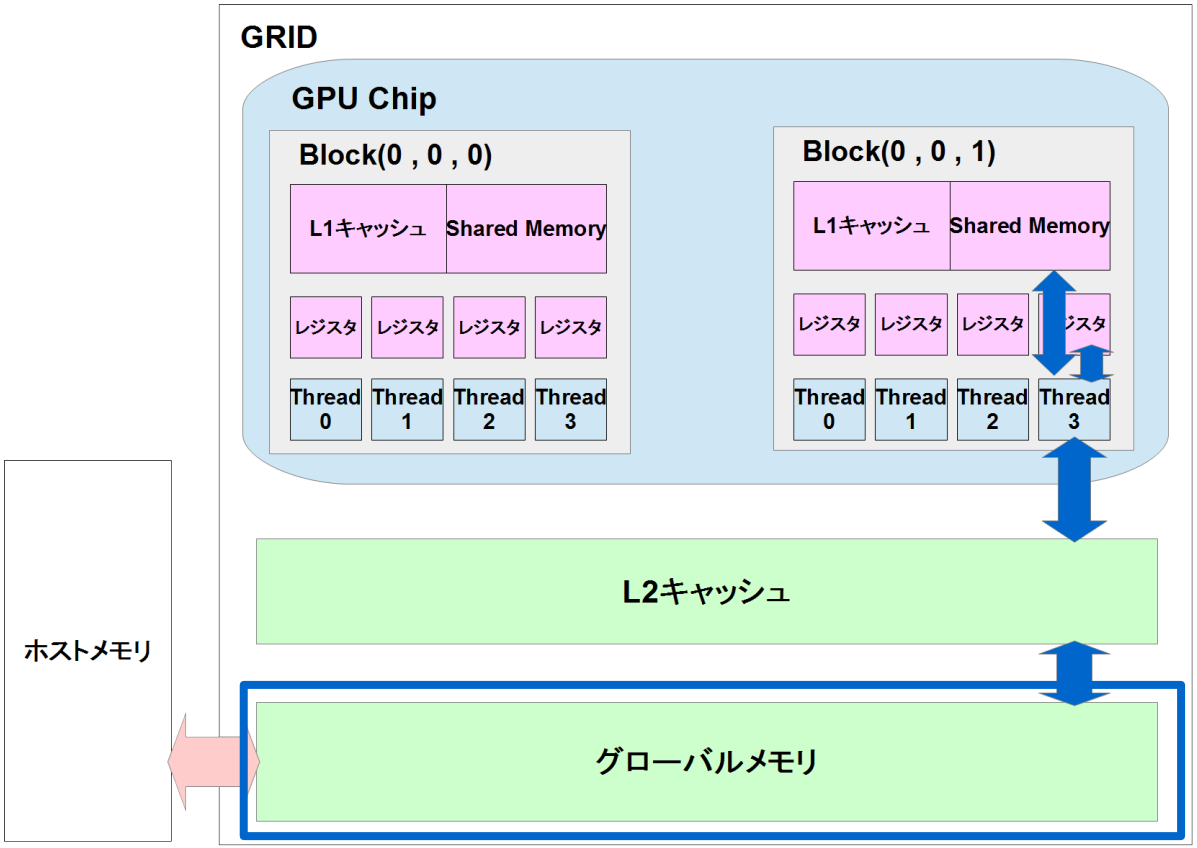
スレッドを複数まとめたBlockのサイズは任意
・・・ですが、内部のアーキテクチャでは32スレッドを一単位として
協調動作されるように設計されている
このまとまりを、CUDAではワープと呼ぶ

CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

メモリ階層
- グローバルメモリ
- シェアードメモリ
- レジスタ
- etc



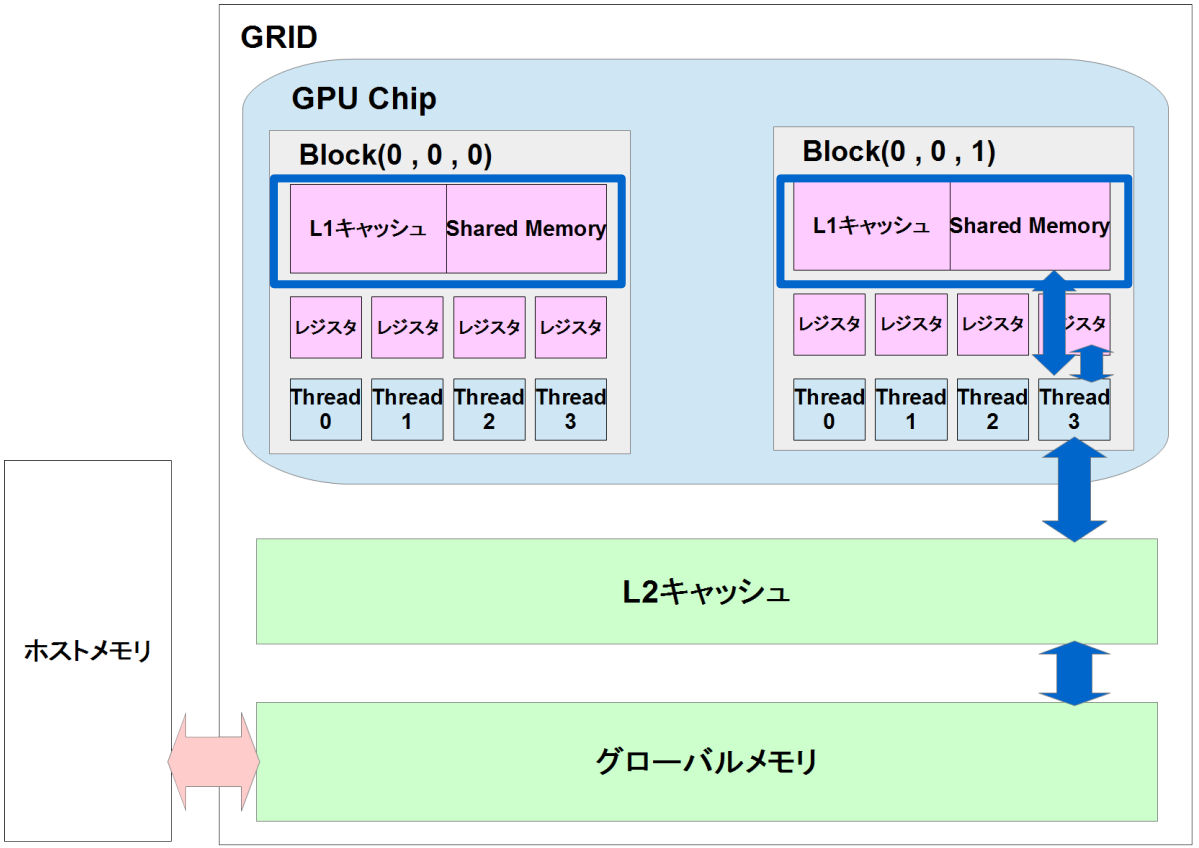
GRID内の全てのBlockで参照可能

CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

メモリ階層
- グローバルメモリ
- シェアードメモリ
- レジスタ
- etc



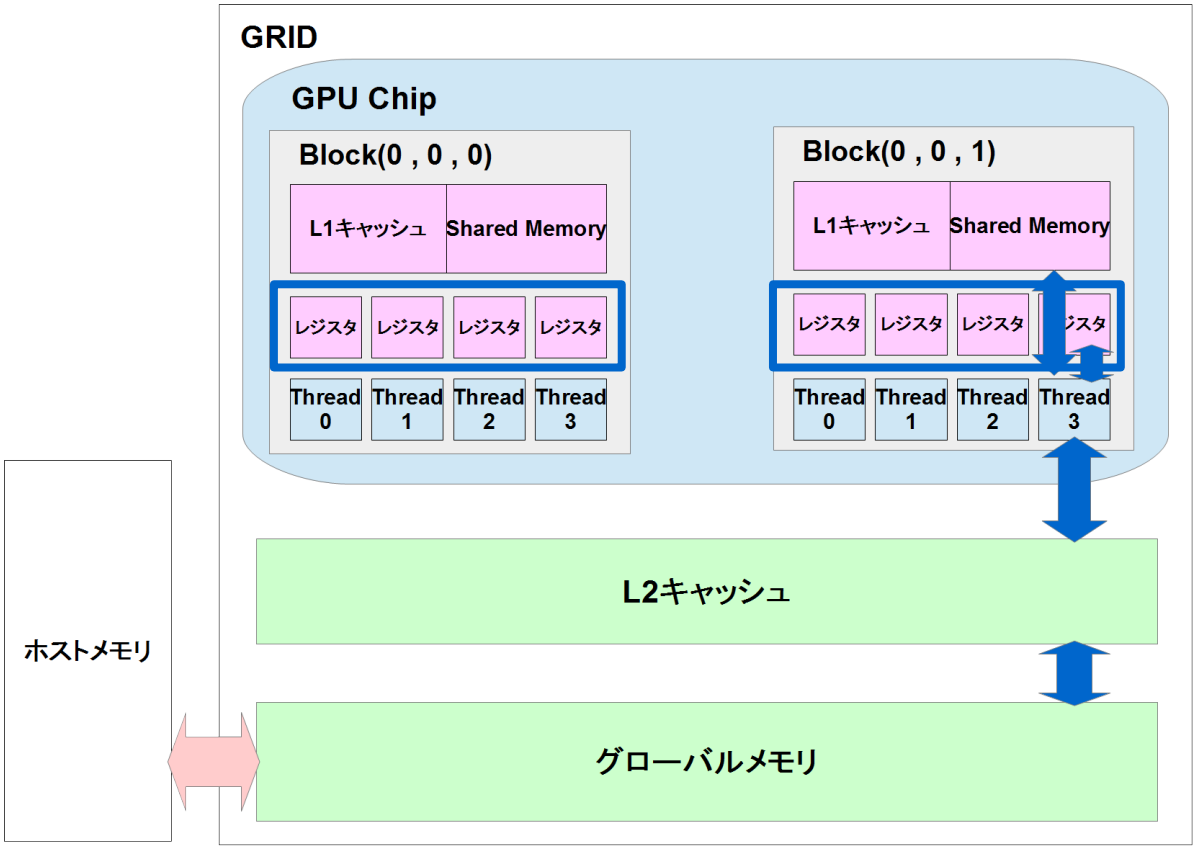
Block内の全てのスレッドで参照可能（※Block間の参照は不可）

CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

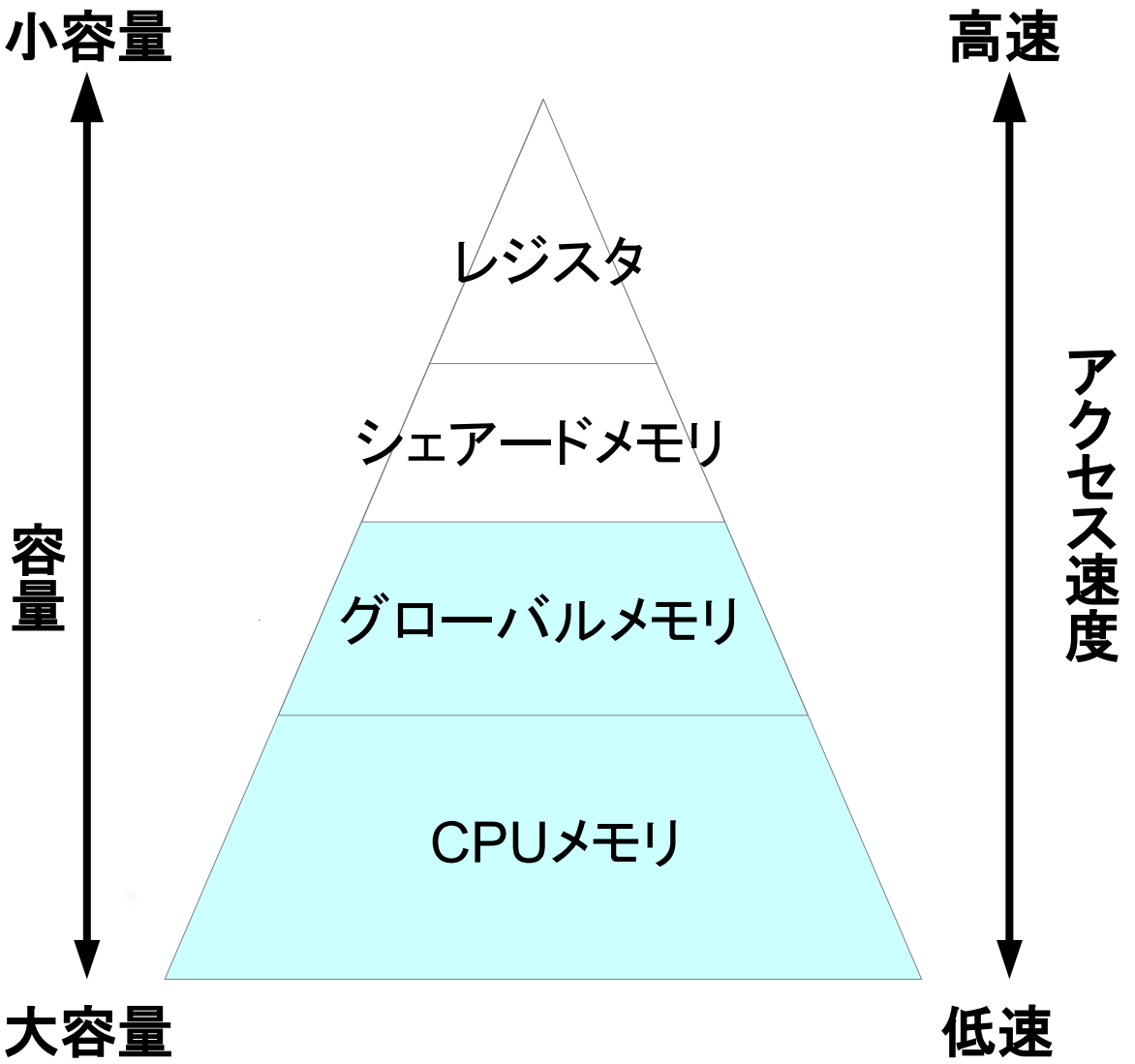
スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

メモリ階層
- グローバルメモリ
- シェアードメモリ
- レジスタ
- etc

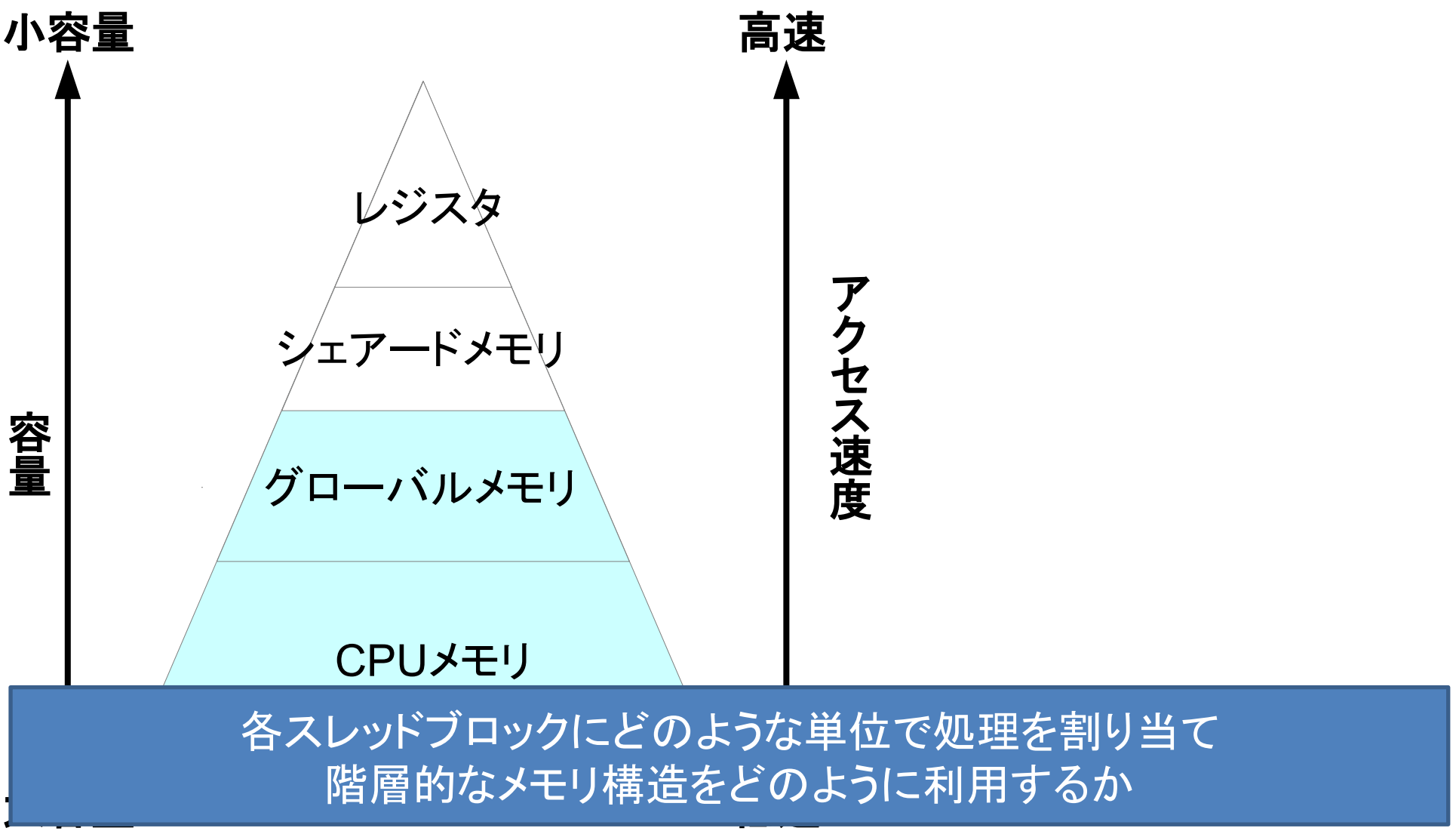


Threadでのみ参照可能（※Thread間の参照は不可）

CUDAのメモリ階層



CUDAのメモリ階層

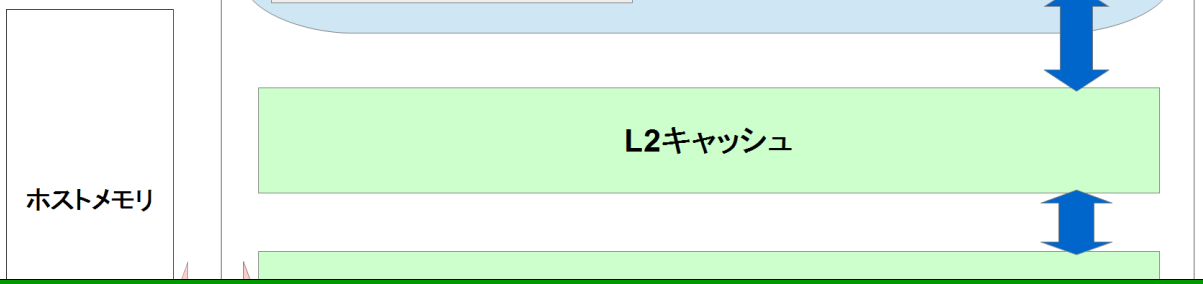

小容量
高速
容量
アクセス速度
レジスタ
シェアードメモリ
グローバルメモリ
CPUメモリ
各スレッドブロックにどのような単位で処理を割り当て
階層的なメモリ構造をどのように利用するか

CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

メモリ階層
- グローバルメモリ
- シェアードメモリ
- レジスタ

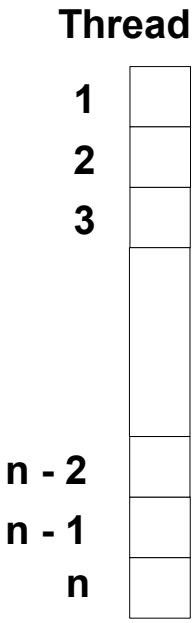


ワープ単位で実行する際に注意するべき最適化

ワープダイバージェンス

CUDAのスレッドは、ワープ単位である32スレッドが同一命令を実行

同一warpに異なる命令を実行

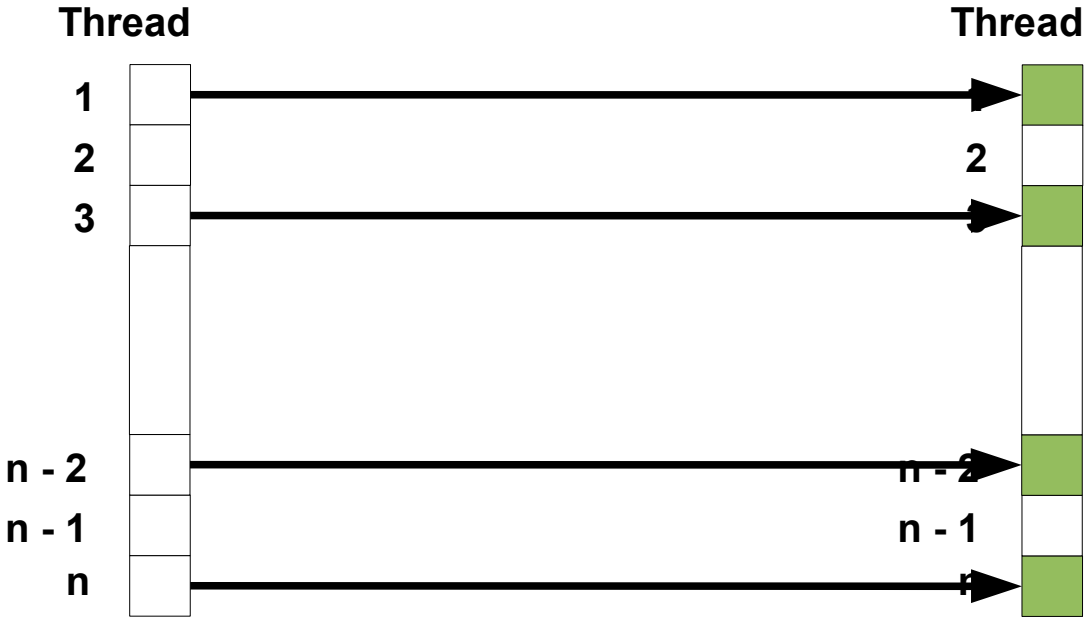



```
if ( Thread.no % 2){
    (処理A);
}else{
    (処理B);
}
```

ワープダイバージェンス

CUDAのスレッドは、ワープ単位である32スレッドが同一命令を実行

同一warpに異なる命令を実行

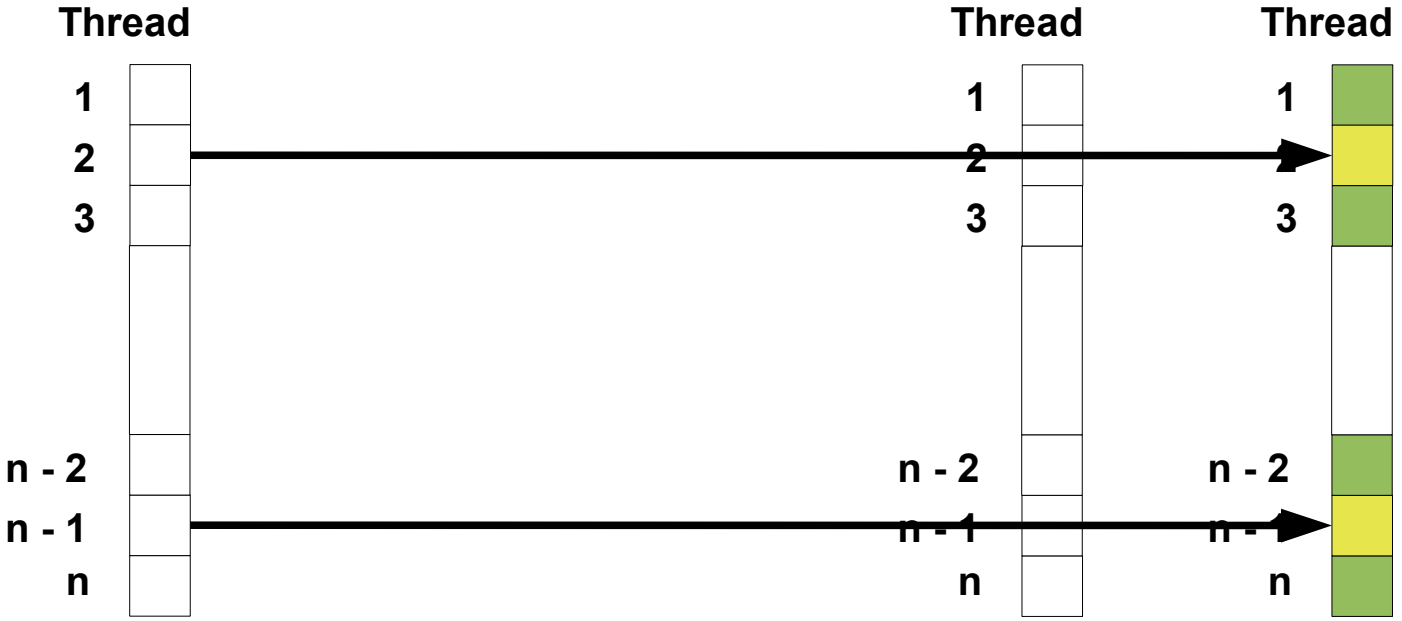



```
if ( Thread.no % 2){
        (処理A); //1-Pass
}else{
        (処理B);
}
```

ワープダイバージェンス

CUDAのスレッドは、ワープ単位である32スレッドが同一命令を実行

同一warpに異なる命令を実行

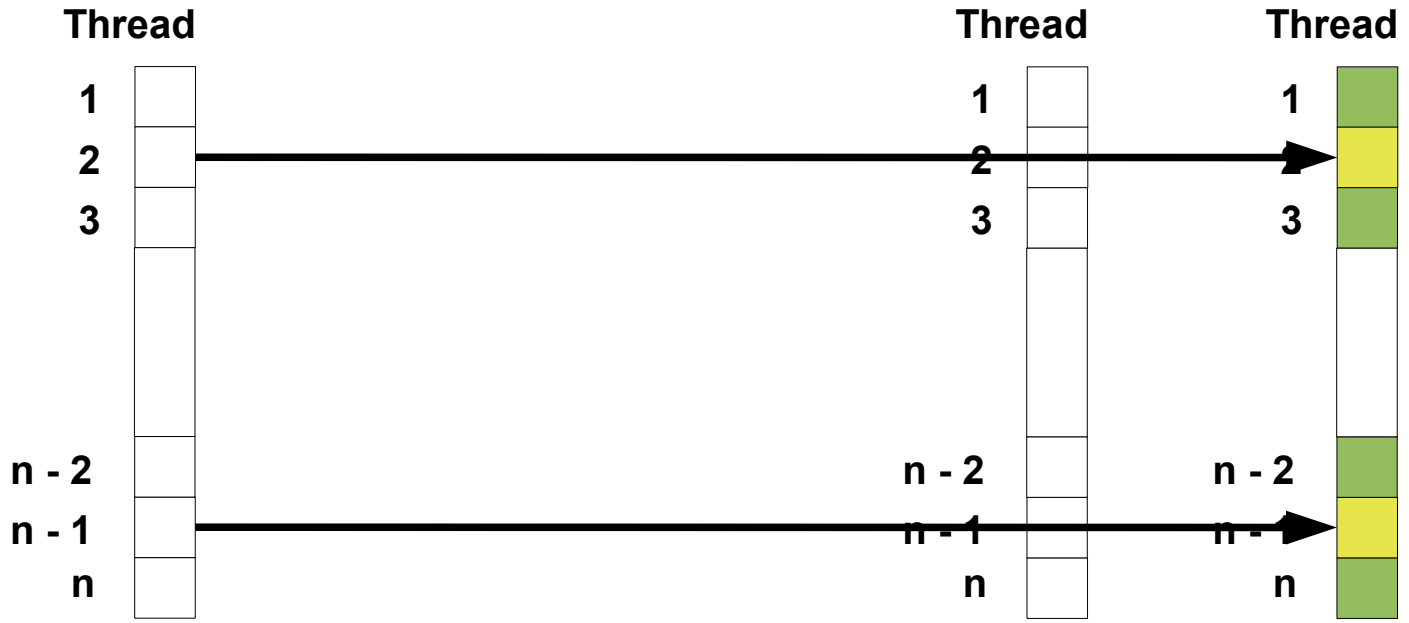



```
if ( Thread.no % 2){
        (処理A); //1-Pass
}else{
        (処理B); //2-Pass
}
```

ワープダイバージェンス

CUDAのスレッドは、ワープ単位である32スレッドが同一命令を実行

同一warpに異なる命令を実行



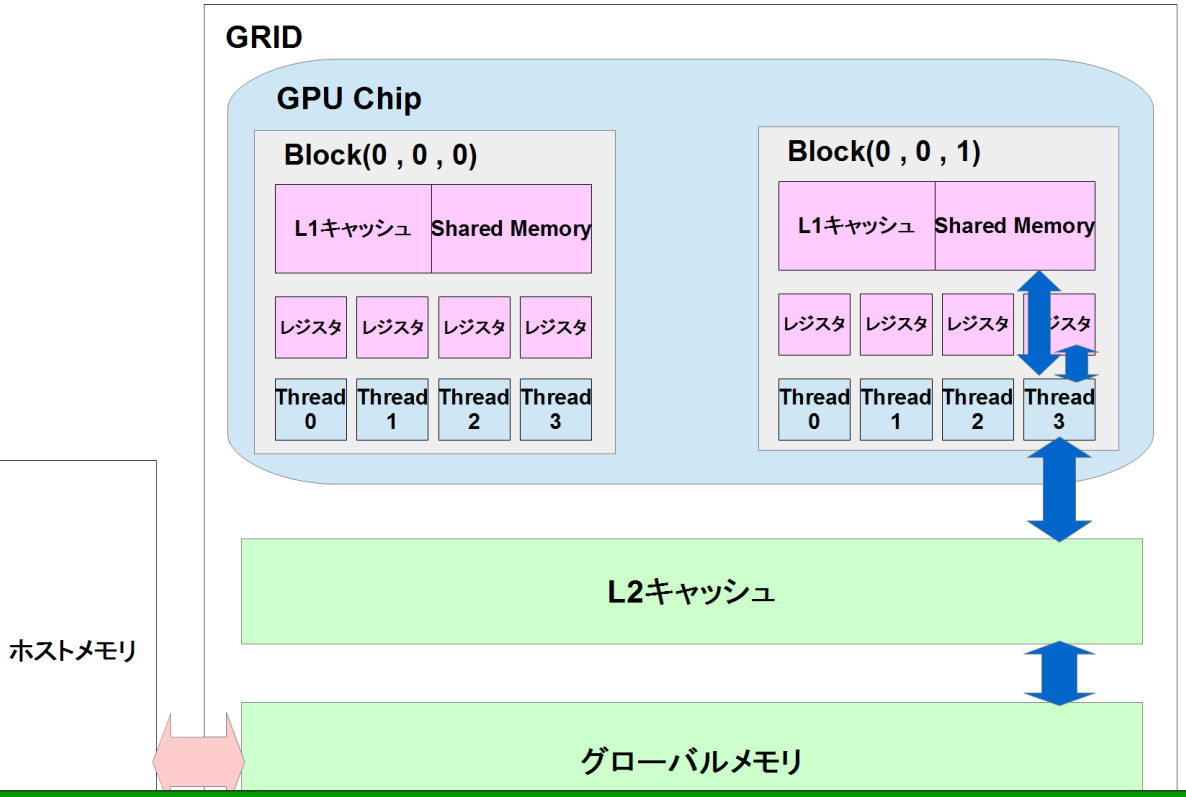
同一ベクトル命令内に異なる命令が含まれる場合、処理効率が低下する
ワープダイバージェントが発生

CUDAによるスレッド階層とメモリ階層

アーキテクチャの違いを隠蔽するためにCUDAでは複数のスレッド階層
GRID、Block、Thread

スレッド階層
- 演算処理を行うスレッド
- スレッドを複数まとめた
  スレッドブロック

メモリ階層
- グローバルメモリ
- シェアードメモリ
- レジスタ
- etc



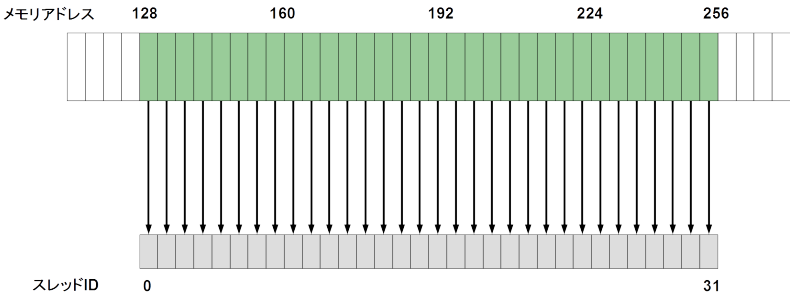
グローバルメモリとシェアードメモリのアクセスに関する最適化

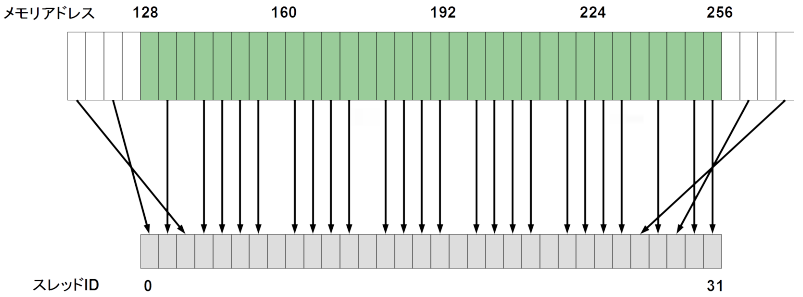
グローバルメモリへのアクセス

グローバルメモリの広帯域なメモリを最大限活用するためにアクセス方法
　コアレスアクセス

コアレスアクセス
　データのアクセス単位が連続になること

データのアクセス単位
　32スレッドの各スレッドが連続したアクセス



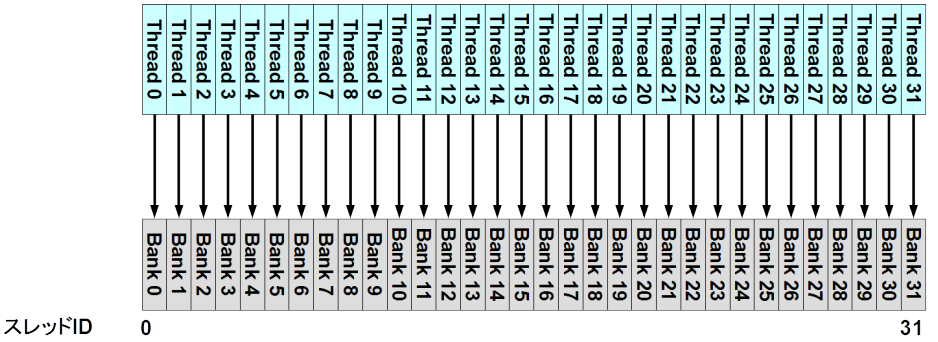


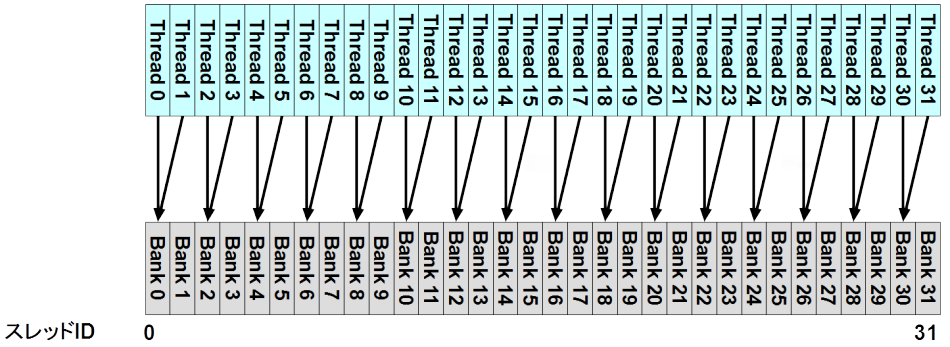
シェアードメモリへのアクセス

シェアードメモリを最大限活用するためにアクセス方法
　32スレッドの各スレッドが異なるバンクにアクセスすることが重要

32スレッドの複数のスレッドが同一のバンクにアクセス
　バンクコンクリフト

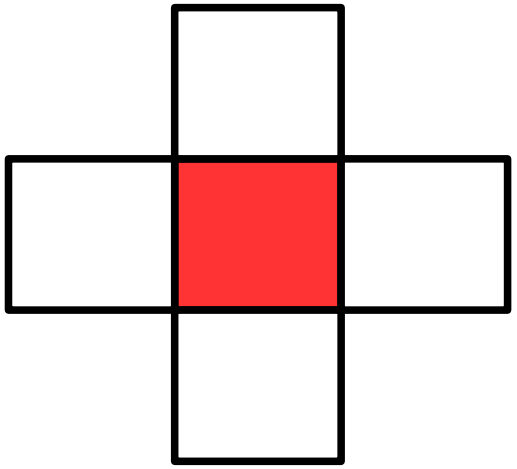
データのアクセス単位
　32スレッドの各スレッドが
　異なる32のバンクへアクセス





ポアソン方程式（5点差分法）を題材にCUDAプログラミング

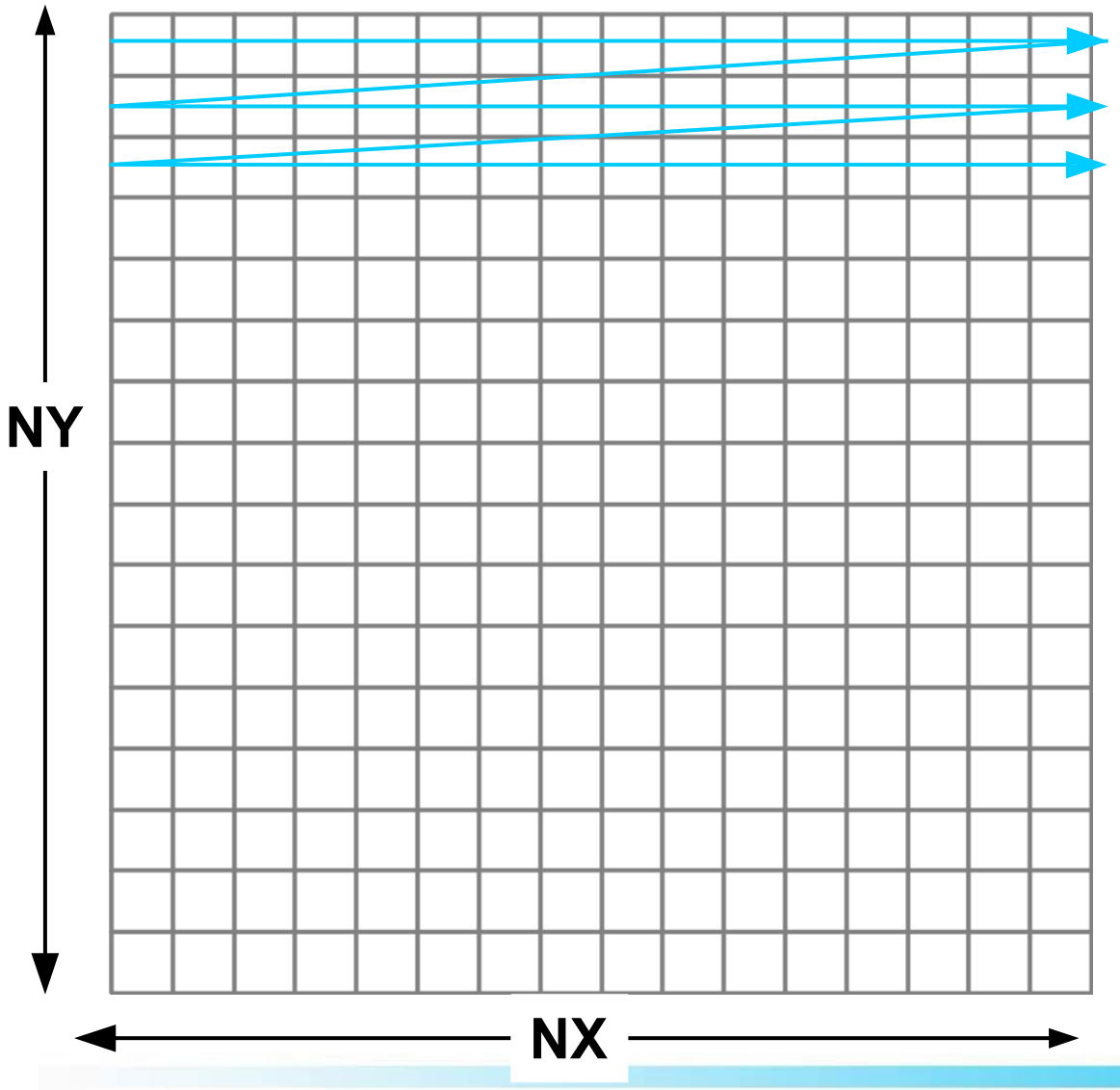
ポアソン方程式による熱伝導問題を題材にCUDAプログラミングを実践
差分方程式を解く



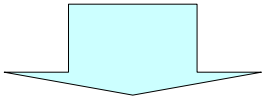
$$4u_i - u_{i-nx} - u_{i-1} - u_{i+1} - u_{i+nx} / d^2 = f_i$$

メモリ配置

CUDAは2次元配列を1次元配列で扱う



ARRAY[NX][NY];

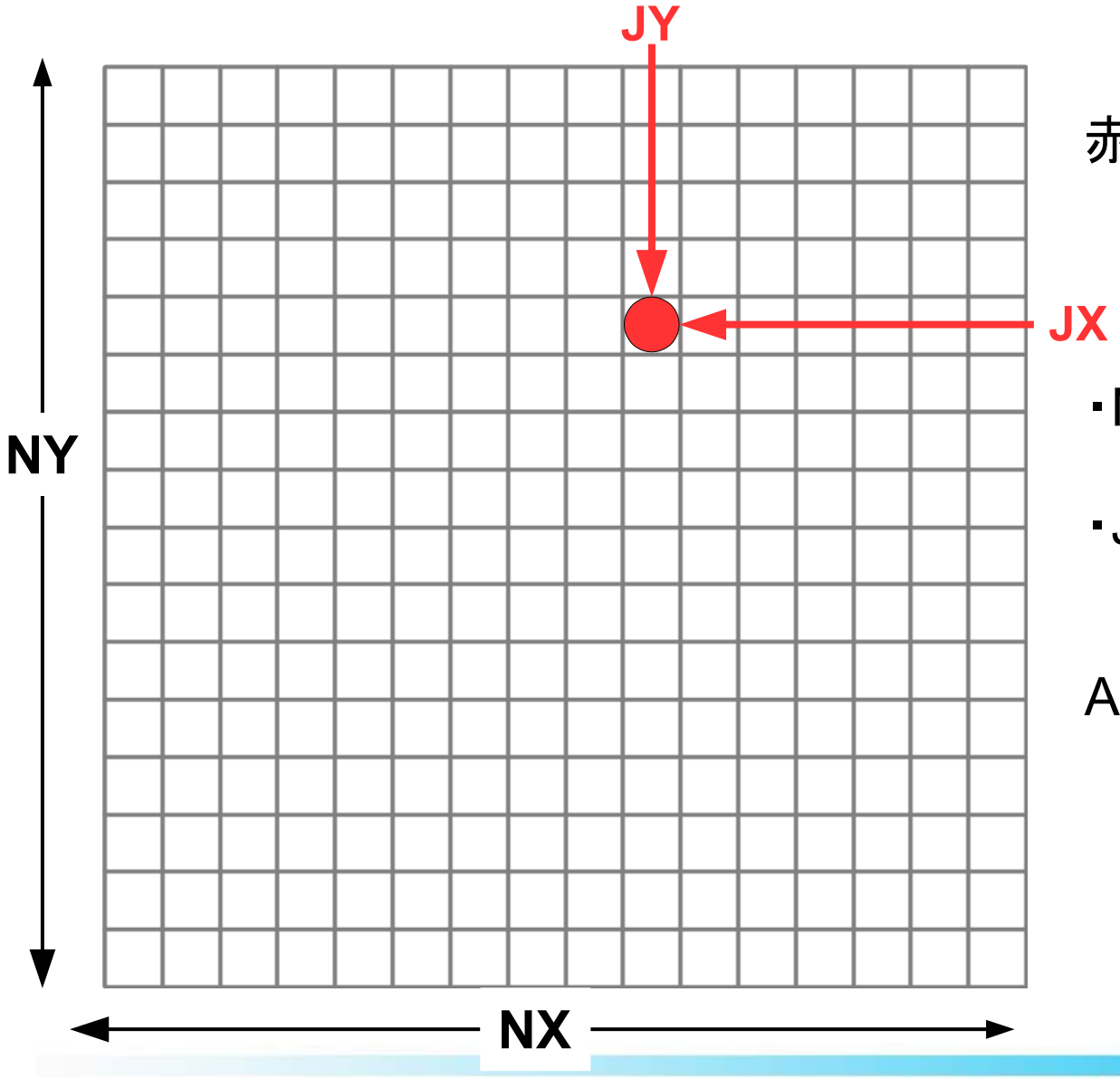


1次元配列で扱う場合
サイズは、NX * NY

ARRAY[NX*NY]と置き換え可能

メモリアクセス

CUDAは2次元配列を1次元配列で扱う



赤い●にアクセスする場合

・NX＊JYでアクセスする行を決定

・JXでアクセスする行の何列目かを決定

ARRAY[NX*JY+JX]
→●のアドレスにアクセス

ポアソン方程式（5点差分法）を題材にCUDAプログラミング

```
#define MAX_ITER 10000

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    initialize(array, nx, ny);

    jacobi(array,nx,ny,omega);

    free(array[0]);
    free(array[1]);
    return 0;
}
```

```
void jacobi(float **array, int nx, int ny, float omega)
{
    int i, j;
    int iter;
    /* buffer id */
    int current = 0;
    int next = 1;

    for(iter=0;iter< MAX_ITER;iter++) {
        for(j=1;j<ny-1;j++) {
            for(i=1;i<nx-1;i++) {
                float curv=array[current][j*nx+i];
                float nextv;
                /* 方程式求解 */
                nextv=(array[current][(j-1)*nx+i] +
                        array[current][(j+1)*nx+i] +
                        array[current][j*nx+(i-1)] +
                        array[current][j*nx+(i+1)]) * 0.25f;
                /* 計算結果の格納  */
                array[next][j*nx+i]=curv+(nextv-curv)*omega;
            }
        }
        /* 計算領域の切替 */
        current=1-current;
        next=1-next;
    }
    return;
}
```

CUDAのプログラム構成

ホストプログラムとGPUカーネル関数で構成される

ホストプログラム

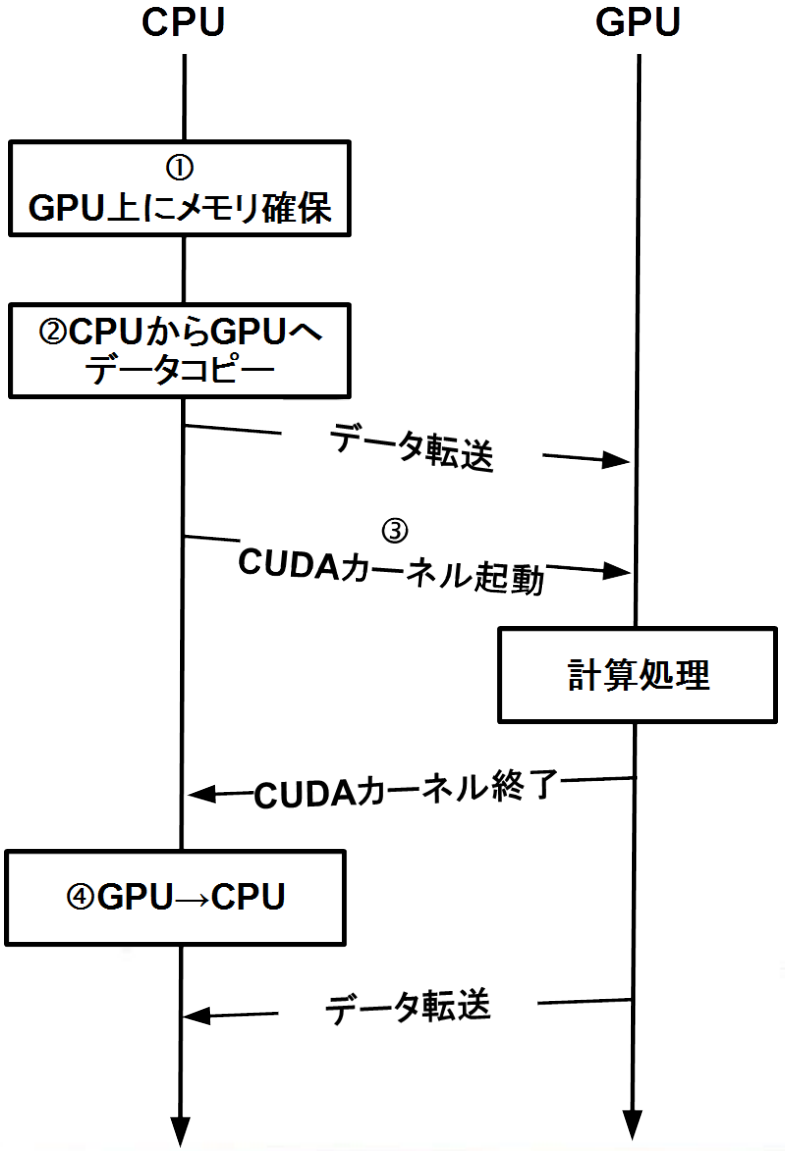
- CPUで実行されるプログラム
- GPUに対してメモリコピーやプログラムのコールを行う

GPUカーネル関数

- GPUで実行されるプログラム
- ホストプログラムから呼び出されたプログラムを実行

CUDAのプログラム構成

ホストプログラムとGPUカーネル関数で構成される



①ホストプログラムからGPU上のメモリ領域を確保

グローバルメモリの領域の確保
　関数【cudaMalloc】を利用

C言語版のMallocとほぼ同様

```
cudaMalloc( void **deviceptr, size_t size);
```

●deviceptr：グローバルメモリ上のアドレス
●size：配列の長さ

メモリ領域の利用が終了したら、freeを忘れずに

```
cudaFree( void **deviceptr );
```

●deviceptr：グローバルメモリ上のアドレス

①ホストプログラムからGPU上のメモリ領域を確保

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    initialize(array, nx, ny);

    jacobi(array,nx,ny,omega);

    free(array[0]);
    free(array[1]);
    return 0;
}
```

①ホストプログラムからGPU上のメモリ領域を確保

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    initialize(array, nx, ny);

    jacobi(array,nx,ny,omega);

    free(array[0]);
    free(array[1]);
    return 0;
}
```

②ホストプログラムからGPU上へデータをコピー

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    // GPUメモリ確保 //

    jacobi(array,nx,ny,omega);

    free(array[0]);
    free(array[1]);
    return 0;
}
```

②ホストプログラムからGPU上へデータをコピー

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;
    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));

    jacobi(array,nx,ny,omega);

    free(array[0]);
    free(array[1]);
    return 0;
}
```

②ホストプログラムからGPU上へデータをコピー

ホストプログラムからGPU上へデータをコピー
　関数【cudaMemcpy】を利用

C言語版のmemcpy関数とほぼ同様

cudaMemcpy(void *dst, const void *src, size_t size,
emue cudaMemcpyKind kind);

- dst : コピー先の配列のアドレス
- src : コピー元の配列のアドレス
- size : 配列のサイズ
- size : 転送タイプを指定
  （1 : cudaMemcpyHostToDevice :　CPU→GPU）
  （2 : cudaMemcpyDeviceToHost :　GPU→CPU）
  （3 : cudaMemcpyDeviceToDevice : GPU→GPU）

②ホストプログラムからGPU上へデータをコピー

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;
    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));

jacobi(array,nx,ny,omega);
```

②ホストプログラムからGPU上へデータをコピー

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;
    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));

    cudaMemcpy(darray, array, 2*nx*ny*sizeof(float), cudaMemcpyHostToDevice);

jacobi(array,nx,ny,omega);
```

③ホストプログラムからGPUカーネルをコール

ホストプログラムからGPUカーネルをコール

C言語版の関数呼び出しと区別するために
【<<< >>>】を付与する

```
kernel_function<<< grid_dim, block_dim >>>(kernel_param*, …);
```

- kernel_function : GPUカーネルの関数名
- kernel_param* : カーネル関数の引数

CUDAにおけるGPUのカーネル動作は非同期で動作するため
必ず同期処理が必要
（多くの場合、カーネル関数と同期関数はセットで運用）

```
cudaDeviceSynchronize();
```

③ホストプログラムからGPUカーネルをコール

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    /* GPUメモリ確保 */
    cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));
```

```
    /* GPUカーネルをコール */
    /* GPUカーネルが起動するブロック数とスレッド数を計算 dim3関数
       で指定可能 */
    /* 反復ループをCPU側で制御 */
```

```
    free(array[0]);
    free(array[1]);
    return 0;
}
```

③ホストプログラムからGPUカーネルをコール

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

    /* GPUメモリ確保 */
    cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));
```

```
    /* GPUカーネルをコール */
    /* GPUカーネルが起動するブロック数とスレッド数を計算 dim3関数
       で指定可能 */
    /* 反復ループをCPU側で制御 */

    dim3 grid(((nx-2)+BLOCK_X-1)/BLOCK_X,
              ((ny-2)+BLOCK_Y-1)/BLOCK_Y, 1);
    dim3 block(BLOCK_X, BLOCK_Y, 1);

    free(array[0]);
    free(array[1]);
    return 0;
}
```

③ホストプログラムからGPUカーネルをコール

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

   /* GPUメモリ確保 */
   cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));
```

```
   /* GPUカーネルをコール */
   /* GPUカーネルが起動するブロック数とスレッド数を計算 dim3関数
      で指定可能 */
   /* 反復ループをCPU側で制御 */
   dim3 grid(((nx-2)+BLOCK_X-1)/BLOCK_X,
                ((ny-2)+BLOCK_Y-1)/BLOCK_Y, 1);
   dim3 block(BLOCK_X, BLOCK_Y, 1);
   for(int iter=0;iter<MAX_ITER;iter++) {


   }




    free(array[0]);
    free(array[1]);
    return 0;
}
```

③ホストプログラムからGPUカーネルをコール

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

   /* GPUメモリ確保 */
   cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));
```

```
   /* GPUカーネルをコール */
   /* GPUカーネルが起動するブロック数とスレッド数を計算 dim3関数
      で指定可能 */
   /* 反復ループをCPU側で制御 */
   dim3 grid(((nx-2)+BLOCK_X-1)/BLOCK_X,
             ((ny-2)+BLOCK_Y-1)/BLOCK_Y, 1);
   dim3 block(BLOCK_X, BLOCK_Y, 1);
   for(int iter=0;iter<MAX_ITER;iter++) {
     jacobi<<<grid, block>>>(darray, nx, ny, omega, current, next);
     CudaDeviceSynchronize();

     /* 計算領域の切替 */
       current = 1-current;
       next = 1-next;
   }


    free(array[0]);
    free(array[1]);
    return 0;
}
```

GPUカーネル関数

GPU上で実行される関数

GPUカーネルは、GPU側のメモリのみアクセス可能

値の返却は不可

GPUカーネルの区別するために、GPUカーネル関数の先頭に
【__global__】を付与する

```
__global__ void kernel_function(kernel_param*, …){

}
```

- kernel_function : GPUカーネルの関数名
- kernel_param* : カーネル関数の引数

GPUカーネル関数

```
__global__
void jacobi(float *array, int nx, int ny, float omega, int current, int next)
{
  int nn = nx*ny;
  /* どの位置を計算担当しているか？を計算 */

  float curv = array[nn*current+(j*nx+i)];
  float nextv;
  nextv = (array[nn*current+((j-1)*nx+i)]+
           array[nn*current+((j+1)*nx+i)]+
           array[nn*current+(j*nx+(i-1))]+
           array[nn*current+(j*nx+(i+1))])*0.25;
  /* 計算結果の格納   */
  array[nn*next+(j*nx+i)] = curv+(nextv-curv)*omega;

  return;
}
```

GPUカーネル関数

```
__global__
void jacobi(float *array, int nx, int ny, float omega, int current, int next)
{
  int nn = nx*ny;
  /* どの位置を計算担当しているか？を計算 */
    int i = blockDim.x*blockIdx.x+threadIdx.x+1;
    int j = blockDim.y*blockIdx.y+threadIdx.y+1;

  float curv = array[nn*current+(j*nx+i)];
  float nextv;
  nextv = (array[nn*current+((j-1)*nx+i)]+
           array[nn*current+((j+1)*nx+i)]+
           array[nn*current+(j*nx+(i-1))]+
           array[nn*current+(j*nx+(i+1))])*0.25;
  /* 計算結果の格納  */
  array[nn*next+(j*nx+i)] = curv+(nextv-curv)*omega;

  return;
}
```

GPUカーネル関数

```
__global__
void jacobi(float *array, int nx, int ny, float omega, int current, int next)
{
  int nn = nx*ny;
  /* どの位置を計算担当しているか？を計算 */
    int i = blockDim.x*blockIdx.x+threadIdx.x+1;
    int j = blockDim.y*blockIdx.y+threadIdx.y+1;

  float curv = array[nn*current+(j*nx+i)];
  float nextv;
  nextv = (array[nn*current+((j-1)*nx+i)]+
           array[nn*current+((j+1)*nx+i)]+
           array[nn*current+(j*nx+(i-1))]+
           array[nn*current+(j*nx+(i+1))])*0.25;
  /* 計算結果の格納  */
  array[nn*next+(j*nx+i)] = curv+(nextv-curv)*omega;

  return;
}
```

方程式求解にFOR文がない
これは、自身が担当する領域を
blockIDとthreadIDで計算するため

④GPU上からホストへデータをコピー

ホストプログラムからGPU上へデータをコピー
　関数【cudaMemcpy】を利用

C言語版のmemcpy関数とほぼ同様

cudaMemcpy(void *dst, const void *src, size_t size,
emue cudaMemcpyKind kind);

- dst : コピー先の配列のアドレス
- src : コピー元の配列のアドレス
- size : 配列のサイズ
- size : 転送タイプを指定
  （1 : cudaMemcpyHostToDevice :　CPU→GPU）
  （2 : cudaMemcpyDeviceToHost :　GPU→CPU）
  （3 : cudaMemcpyDeviceToDevice : GPU→GPU）

③ホストプログラムからGPUカーネルをコール

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

   /* GPUメモリ確保 */
    cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));
```

```
    /* GPUカーネルをコール */
    /* GPUカーネルが起動するブロック数とスレッド数を計算 dim3関数
       で指定可能 */
    /* 反復ループをCPU側で制御 */
    dim3 grid(((nx-2)+BLOCK_X-1)/BLOCK_X,
                  ((ny-2)+BLOCK_Y-1)/BLOCK_Y, 1);
    dim3 block(BLOCK_X, BLOCK_Y, 1);
    for(int iter=0;iter<MAX_ITER;iter++) {
      jacobi<<<grid, block>>>(darray, nx, ny, omega, current, next);
      CudaDeviceSynchronize();

      /* 計算領域の切替 */
       current = 1-current;
       next = 1-next;
    }

    free(array[0]);
    free(array[1]);
    return 0;
}
```

③ホストプログラムからGPUカーネルをコール

```
#define MAX_ITER 10000
#define BLOCK_X 16
#define BLOCK_Y 16

int main(int argc, char *argv[])
{
    int nx, ny;
    float omega = 0.5;
    float **array; /* for double buffer */
    struct timeval st;
    struct timeval et;
    long us;

    /* GPUメモリ宣言 */
    float *darray;

    if (argc == 3){
        nx=atoi(argv[1])+2;
        ny=atoi(argv[2])+2;
    }else{
          return 0;
    }

     array=(float**)calloc(2,sizeof(float*));
    array[0]=(float*)calloc(nx*ny,sizeof(float));
    array[1]=(float*)calloc(nx*ny,sizeof(float));

   /* GPUメモリ確保 */
    cudaMalloc((void **)&darray, 2*nx*ny*sizeof(float));

   /* GPUカーネルをコール */
   /* GPUカーネルが起動するブロック数とスレッド数を計算 dim3関数
      で指定可能 */
   /* 反復ループをCPU側で制御 */
    dim3 grid(((nx-2)+BLOCK_X-1)/BLOCK_X,
                  ((ny-2)+BLOCK_Y-1)/BLOCK_Y, 1);
    dim3 block(BLOCK_X, BLOCK_Y, 1);
    for(int iter=0;iter<MAX_ITER;iter++) {
      jacobi<<<grid, block>>>(darray, nx, ny, omega, current, next);
      CudaDeviceSynchronize();

      /* 計算領域の切替 */
       current = 1-current;
       next = 1-next;
    }

    cudaMemcpy(array, darray, 2*nx*ny*sizeof(float),
                        cudaMemcpyDeviceToHost);

    free(array[0]);
    free(array[1]);
    return 0;
}
```

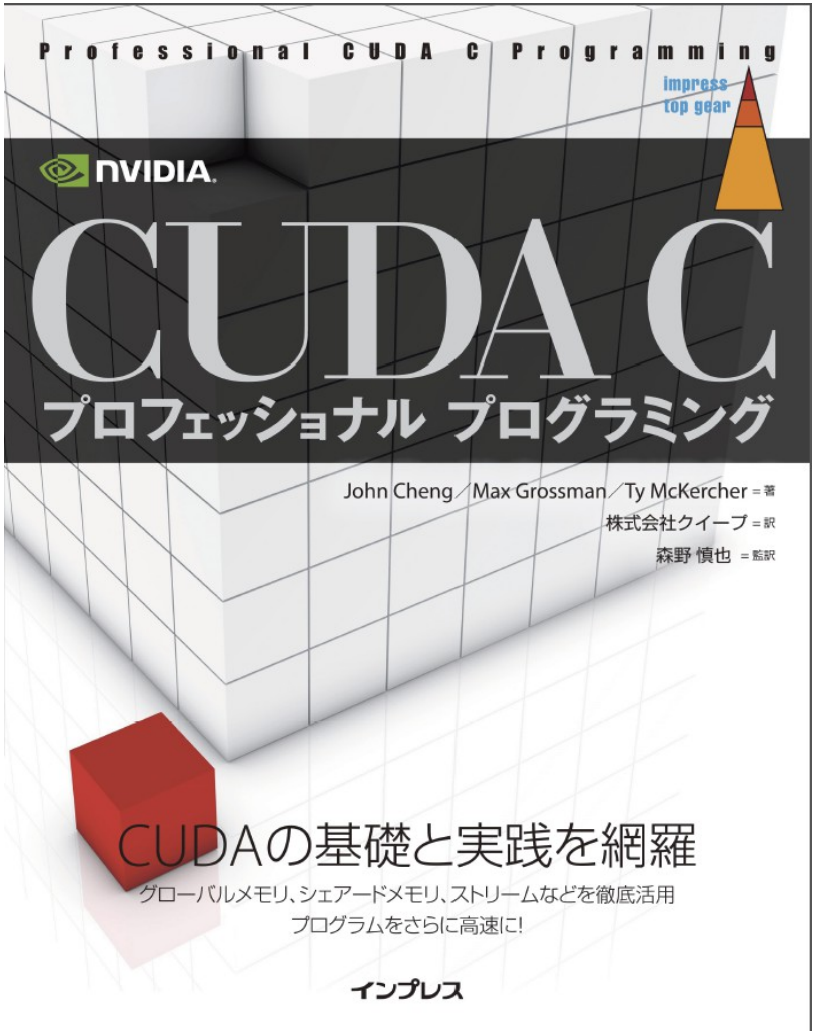
cudaのコンパイル

NVIDIAのcuda-zoneよりcuda toolkitをダウンロードしてインストール
https://developer.nvidia.com/cuda-zone

コンパイル方法は、gccとほぼ同様でnvccという専用コンパイラを用いる
$nvcc ファイル名 オプション
※主なオプション
・-O [ラージオーオプション]
・-arch_sm** [cudaのデバイス最適化 ex. sm70:volta sm60:pascal]

CUDAのオススメ参考書

インプレス社
　CUDA C プロフェッショナル　プログラミング



おわりに

GPUコンピューティング
- ライブラリを用いる方法
- ディレクティブ構文を用いる方法
- 専用の拡張言語であるCUDAを用いる方法

既存のプログラムをCUDA化する単純な方法について解説