GPUを用いた
惑星運動シミュレーション

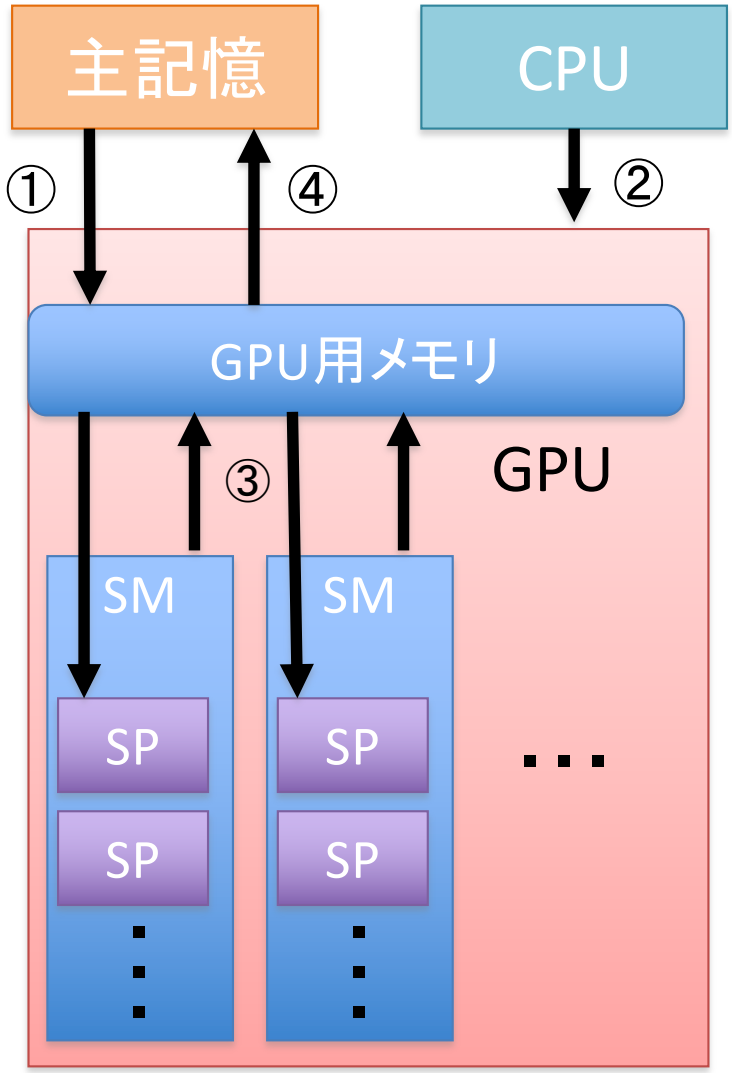
並列計算機

- 超並列コンピュータ
  - 数万個以上の汎用プロセッサと高速ネットワークで構成
  - 大規模計算向け
- 高速計算プロセッサ
  - 元々は動画などのマルチメディアを用いたアプリケーション向け

  →並列計算にも使用されている
  - Tesla/CUDA (nVIDIA)
  - AMD FirePro (AMD)

} GPU

CUDA (Compute Unified Device Architecture)



SM：Streaming Multiprocessor
SP：Streaming Processor

- nVIDIAが提供するGPU向けのC言語統合開発環境
- 対応環境
  - Geforce8 シリーズ以上(ネットブック/トップ用)
  - nVIDIA Tesla(ハイパフォーマンスコンピューティング用)
  - nVIDIA Quadro(CAD用)
- 対応言語
  - CUDA C：C言語とC++の一部の構文のみ対応。C言語を拡張
  - CUDA Fortran：The Prortland Groupから提供。Fortran 2003を拡張
- 並列モデル
  - SPMDモデル
- CUDAの処理の流れ
  1. 主記憶からデータをGPU用メモリにコピー
  2. CPUがGPUに対して処理を指示する
  3. GPUが必要なデータを取り込み、各コアで並列処理
  4. 結果をGPU用メモリから主記憶にコピー

本実験で利用するGPU

nVIDIA Quadro K620
プロセッサーコア：384
（3? SM x 128? SP ）
メモリ：2048 MB
単精度演算: 863.2 GFLOPS
（すべての学生用端末）

nVIDIA GeForce GTX 780
プロセッサーコア：2304
（12 SM x 192 SP ）
メモリ：3072 MB
単精度演算: 3977 GFLOPS
（gpu01.ced.cei.uec.ac.jp等、
リモートログインして使用）

CUDA プログラミング

- GPUの制御はCPUから行う

→CPU上で実行するプログラムでGPUを制御

通常のC言語プログラムの拡張

- CUDAのプログラム
  - 通常ルーチン
    - CPUの処理
    - GPUの制御
  - GPUの制御ルーチン
    - GPU上での計算
- CPU側(ホスト)とGPU側(デバイス)で異なるメモリを使用
  - 互いのメモリのデータは直接参照できない

⇒データを利用する前に転送が必要

課題

- 課題1：CUDAプログラミングの基礎
  - GPUでの計算
    - メモリの扱い
    - 関数呼び出し
    - ループ処理
    - ブロックとスレッド
  - コンパイルの方法と実行
  - 計算時間の測定
- 課題2：銀河シミュレーションの準備
  - 高速化
- 課題3：銀河シミュレーション

課題1

- 加法定理をGPUで計算

$$\sin(\alpha + \beta) = \sin\alpha \cos\beta + \cos\alpha \sin\beta$$

C言語のプログラム例

```
void add(int n, float *A, float *B, float *C){
  for(int i = 0; i < n; i++)
    C[i] = sin(A[i]) * cos (B[i]) + cos(A[i]) * sin(B[i]);
}
```

- GPUでの計算
  - メモリの扱い
  - ループ処理
  - 関数呼び出し
  - ブロックとスレッド

課題1: CUDAプログラミングの基礎

- ホスト、デバイス間のデータ転送とデバイスで計算するプログラムの例

```
int nBlocks       = 512;          /* ブロック数*/
int nThreads      = 512;          /* 1ブロック当たりのスレッド数*/
int nDimension = nBlocks * nThreads;      /* 全体の計算数*/
cudaSetDevice(0);                         /*使用するGPUの設定*/

float * ex_h = new float[nDimension];      /* ホストメモリの用意 */
randomInit(nDimension, ex_h);              /* 初期化(乱数生成関数は自己設定) */
float *ex_d;                               /* デバイスメモリの用意*/
cudaMalloc(&ex_d, nDimension * sizeof(float));

 /*デバイスメモリへ転送*/
cudaMemcpy(ex_d, ex_h, nDimension * sizeof(float), cudaMemcpyHostToDevice);
/*関数呼び出しによって、実際にGPUで計算する部分*/
kernel_A<<<nBlocks, nBlockSize>>>(ex_d);
/*ホストメモリへ転送*/
cudaMemcpy((void *)ex_h,  ex_d, nDimension * sizeof(float), cudaMemcpyDeviceToHost);
/*メモリの解放*/
delete[] ex_h;
cudaFree(ex_d);
```

課題1:メモリの扱い

```
float *ex_h, *ex_d;
cudaMallocHost(&ex_h, sizeof(float)*n)));      /*ホスト側のメモリの用意*/
memset(ex_h, 0, sizeof(float)*n);              /*0で初期化*/
cudaFreeHost(ex_h);                            /*メモリの解放*/

cudaMalloc(&ex_d, sizeof(float)*n));           /*デバイス側のメモリの用意*/
cudaMemset(ex_d, 0, sizeof(float)*n));         /*0で初期化*/
cudaFree(ex_d);                                /*メモリの解放*/
```

- ホスト側の扱いは例題0の方法でもOK

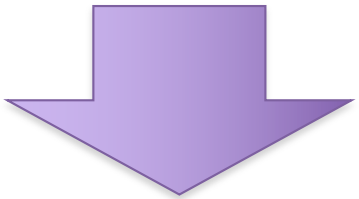
```
cudaMemcpy(ex_d, ex_h, sizeof(float)*n, cudaMemcpyHostToDevice);  /*ホスト側からデバイス側へコピー*/
cudaMemcpy(ex_h, ex_d, sizeof(float)*n, cudaMemcpyDeviceToHost);  /*デバイス側からホスト側へコピー*/
```

- ホスト側の初期化後にデバイス側へ転送したいときや、デバイス側の計算後にホスト側へ転送するときなどに用いる

課題1:ループ処理

CPUにおけるループ処理

```
for(i = 0; i < n; i++)
A[i] = B[i] + C[i];
```



GPUではn個の処理(スレッド)として複数コアで並列的に処理

```
スレッド 0 : A[0] = B[0] + C[0]
スレッド 1 : A[1] = B[1] + C[1]
スレッド 2 : A[2] = B[2] + C[2]
スレッド 3 : A[3] = B[3] + C[3]
・・・・・
```

この部分を1つの関数として設定

課題1:関数呼び出し

- カーネル関数：GPUでの処理を記述した関数(__global__ 必要)
- ブロックIDとスレッドIDと呼ばれる組み込み変数を用いてデータ参照

```
__global__ void kernel_A(float *A, float *B, float *C){
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  A[i]  = B[i] + C[i];
}
```

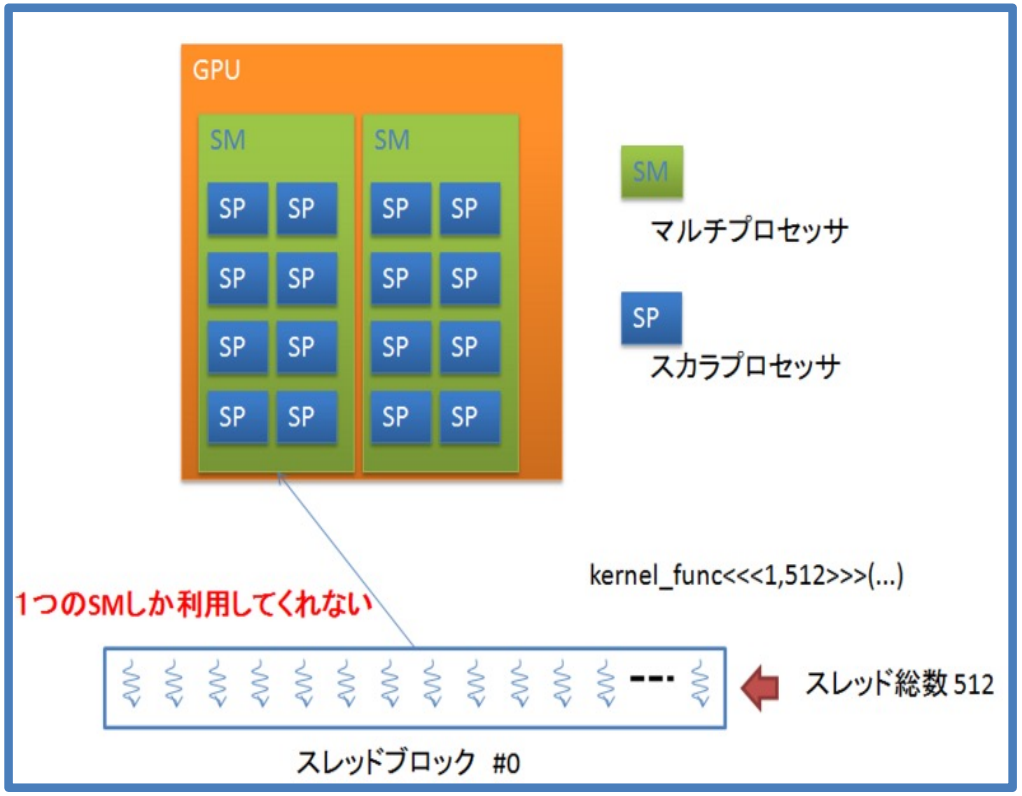
カーネル関数の呼び出し方
(ブロックとスレッドの話はこのあとで説明)

```
dim3 blocks  = dim3(nblocks,  1);        /*ブロック数の設定*/
dim3 threads = dim3(nthreads, 1);        /*スレッド数の設定 */
kernel_A<<<blocks, threads>>>(A, B, C);
```

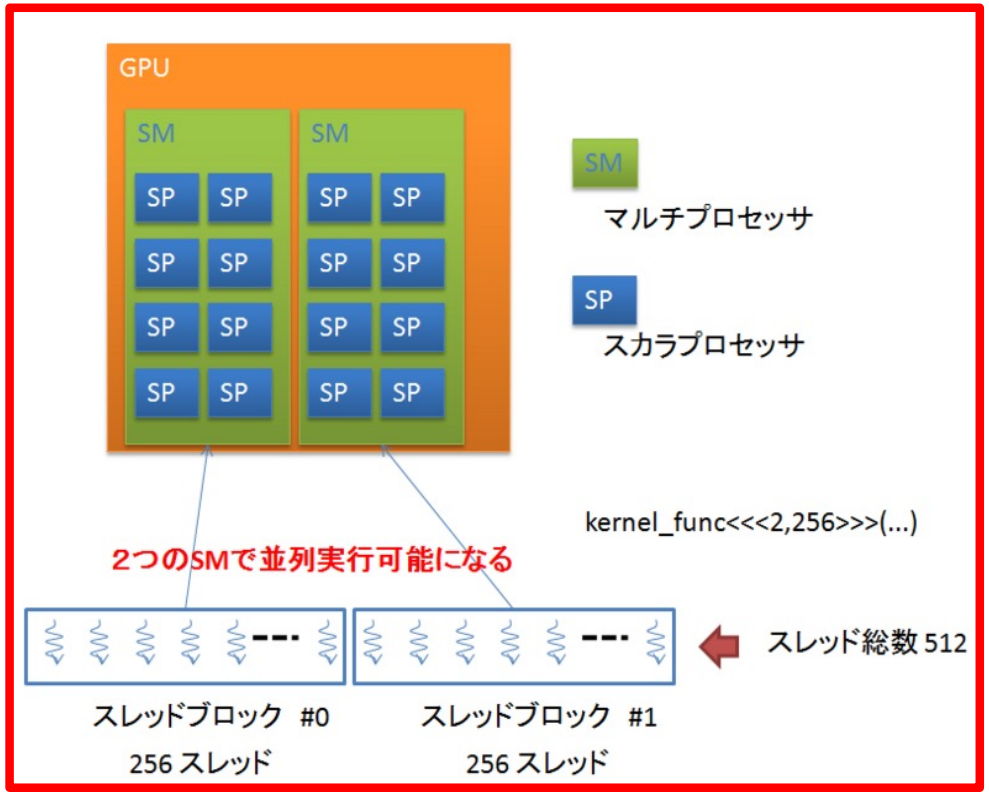
課題1:ブロックとスレッド

- GPUは複数のマルチプロセッサ(SM)があり、各SMに複数のスカラプロセッサ(SP)がある
- 各SM毎にブロックを割り当て、各SP毎にスレッドが割り当てられる

kernel<<<1, 512>>>(….) の場合



kernel<<<2, 256>>>(….) の場合



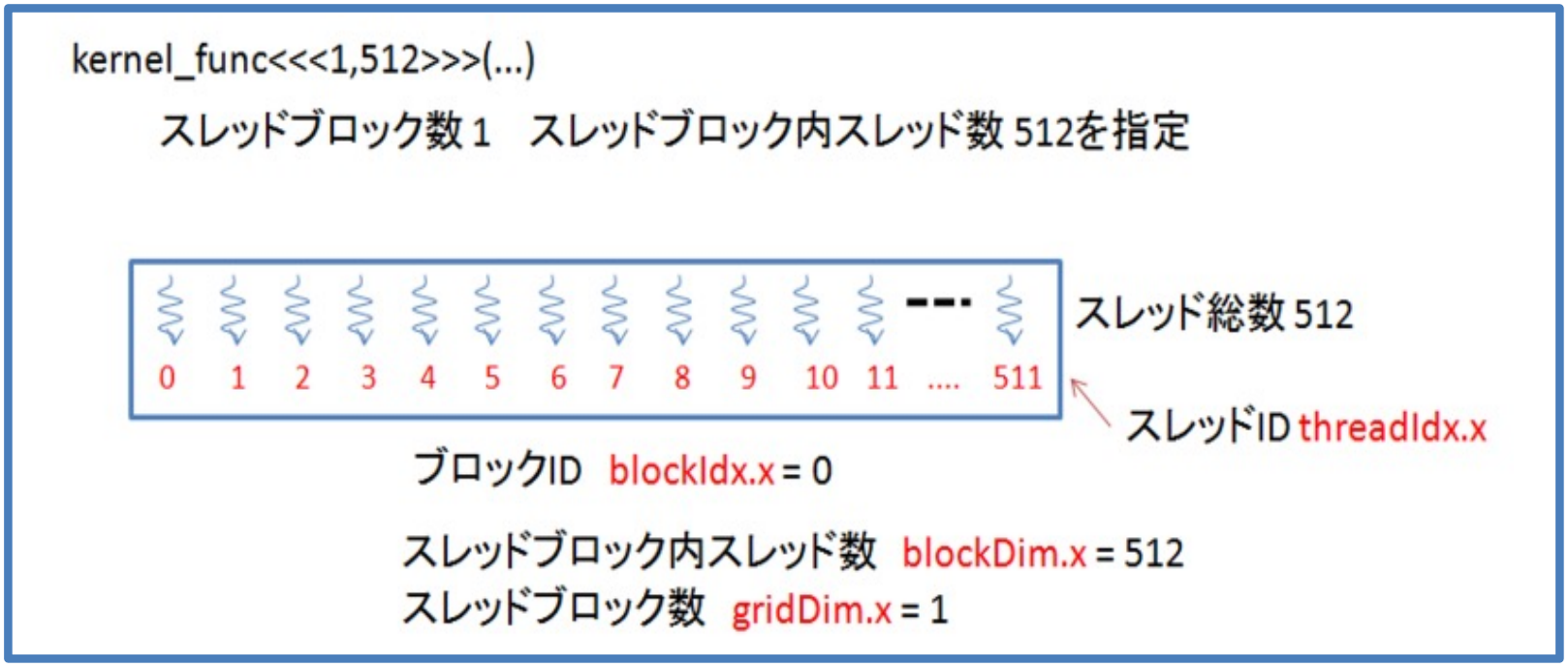
- ブロックに対してSM数が足りない場合、1つのSMが複数のブロックを順次処理
- 適切なブロック数とスレッド数を設定することにより、処理が早くなる

課題1:ブロックIDとスレッドID #1

- カーネル関数コードの記述内容は、1スレッド分の処理内容
- 各スレッドで違うデータを扱いたい

→ブロックIDとスレッドIDを利用して参照

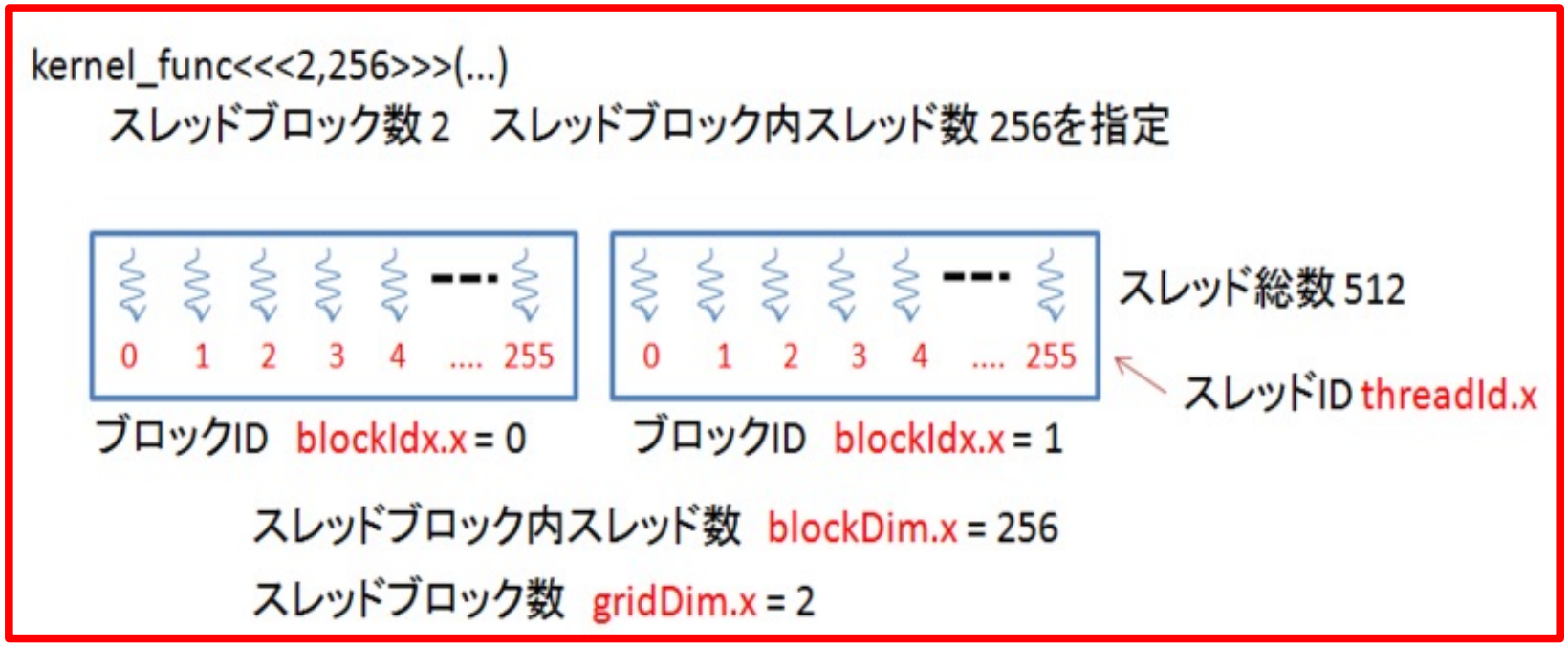
1ブロックの場合



参照例 : int i = threadIdx.x + blockIdx.x * blockDim.x

課題1:ブロックIDとスレッドID #2

複数ブロックの場合



参照例 : int i = threadIdx.x + blockIdx.x * blockDim.x

- 扱えるブロックとスレッドの数には上限がある

→上限はマシンスペックに因る（今回は1ブロック当たり最大1024スレッド）

← */usr/local/cuda/extras/demo_suite/deviceQuery*で確認

課題1:コンパイルと実行

- コンパイル
  - http://wwwhnl.cs.uec.ac.jp/lecture/MICS/ にサンプルプログラム等がある(ID, pass: mics)
  - nvcc −O3 ex.cu −o ex

  (nvccのパス　:　/usr/local/cuda/bin/)
- 実行
  - ./ex

課題1:計算時間の計測

- カーネル(GPUで計算を行っている部分)の計算時間を測定する

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );
kernel_A<<<grid, threads >>> (ex_d, size_x, size_y);
cudaEventRecord( stop, 0);
cudaEventSynchronize( stop);
 cudaEventElapsedTime ( &time, start, stop );  /*変数 time にstartとstopの時間差が入る(ミリ秒)*/
printf("%15.7f ", time);

cudaEventDestroy( start );
cudaEventDestroy( stop);
```

課題1

- 加法定理をGPUで計算
  - ホストメモリで乱数を生成、デバイスメモリに転送
  - デバイスで計算
  - デバイスメモリからホストメモリへ転送
- スレッド数、ブロック数の変更
  - スレッド数、ブロック数を変化させ、計算時間を測定
  - 最適なスレッド数とブロック数を調査する
- 計算精度
  - CPU(倍精度計算)とGPU(単精度計算)の結果を比較

課題2:銀河シミュレーションの準備

銀河シミュレーションと同様のデータアクセスをする簡単なプログラムを作成し、高速化技法の効果を調べる

- 乱数を生成し、総和を求める
  - ホストメモリで乱数を生成、デバイスメモリに転送
  - デバイスで総和計算 （高速化対象）
  - デバイスメモリからホストメモリへ転送
- 高速化
  - スレッド、ブロック数の設定やメモリアクセス回数削減による影響の調査

課題2:銀河シミュレーションの準備

全てのスレッドで全てのデータを参照し、加算

```
__global__ void kernel_B(int n, float *ex_d, float *add_d){
int i   = threadIdx.x + blockIdx.x * blockDim.x;
add_d[i] = ex_d[i];

  for(int  j = 0,; j < n; j++){
     if(i != j)
        add_d[i] += ex_d[j];  /*全データを参照し、加算*/
  }
}
```

C言語の例

```
void add_ALL(int n, float
*data, float *add){
  for(int i=0; i<n; i++){
    add[i] = data[i];
    for(int j=0; j<n; j++){
      if(i != j)
        add[i] += data[j];
    }
  }
}
```

- 高速化効果の確認
  - 最適なブロック数やスレッド数の設定
  - メモリアクセス回数を減らす
  - 分岐の削減　etc…

課題2:高速化例

- メモリアクセスが多いほど、計算に時間がかかる
- 例:各スレッド毎に変数を用意し、加算していく
  - add_dへのメモリアクセス回数は、最後の1回だけ

```
__global__ void kernel_C(int n, float *ex_d, float *add_d){
  int i   = threadIdx.x + blockIdx.x * blockDim.x;
  float add = ex_d[i];                    /*変数の用意*/

  for(int  j = 0;  j < n; j++){
    if(i != j)
      add += ex_d[j];                /*変数に加算していく*/
  }
  add_d[i]  =  add;                  /*最後に変数のデータをコピー*/
}
```

課題3:銀河シミュレーション ～概要～

- 前回作成したプログラムを参考に、GPUで動作するプログラムを作成
- 星の数を増やして計算時間を測定、比較
  - CPUでの計算時間と比較
  - 星の数は倍に増やしていく

資料、サンプルプログラム、レポート〆切

- 資料・サンプルプログラムは以下のページでダウンロード可能

  http://wwwhnl.cs.uec.ac.jp/lecture/MICS/

  ID, PASS : mics

- レポートの〆切、提出場所は以下の通り

  レポートの〆切： 1/11(水) PM1:00 (厳守)

  合格の必要条件： 全ての課題を解くこと

  提出方法： レポート（pdf）を添付ファイルとしてメールで仲谷（nakatani@cs.uec.ac.jp）に送付

  メールのタイトル： MICS実験レポート3G

  添付ファイル名： 学籍番号（半角数字）