GPUコンピューティング（CUDA）講習会

マルチGPUプログラミング

東工大学術情報センター　丸山直也

はじめに

- ノード内に複数GPUがある場合のマルチGPUプログラミングをとりあげます
- CUDAとOpenMPを使います
- 複数ノードにまたがる場合にはMPIなどを使う必要がありますが、今回は対象としません
- 本講習で取り上げる概念等はCUDAに限らずOpenCLプログラミングにも有効です
- CUDAの基礎的な内容を仮定しています

講習会サンプルコード

- /work/nmaruyam/gpu-tutorial/diffusion 以下にサンプルコードをおいてあります。各自のホームディレクトリにコピーしてください。
- 講習会ホームページにも掲載します

目次

1. マルチGPUのための準備
2. プログラミング概要
3. ステップ1：GPU間並列化
4. ステップ2：GPU内並列化
5. 例題

マルチGPUの利点

- パフォーマンス
- メモリ
  - 単一GPUではたかだか4GB
  - N台のGPUを使えばN倍のメモリを利用可能
- TSUBAME計算ノードでは1ノードあたり2枚存在するが、1枚でも2枚でも利用額は同じ→複数使えた方がお得です
- スペース効率に優れた計算機を構築可
  - TSUBAME2ではノードあたり3枚

準備：ハードウェア編

- 一台のマシンに複数のCUDAを実行可能なGPUをインストール（SLIは利用不可）
- GTX295のような単一ボードに複数GPUを搭載したものでも可
- CUDAを実行可能であれば異種GPUでも可
- TSUBAME（１＆２）計算ノード・Tesladebugノードでももちろん可

準備:ソフトウエア編

- CUDAにはマルチGPUのための支援もなければ(本質的な)制約もなし
- 標準的に利用可能なコンパイラ・ライブラリで実現可
  - CUDA
  - CPU側並列化のためのコンパイラ・ライブラリ
    - OpenMP、MPIなど

複数のGPUを使う際の注意点

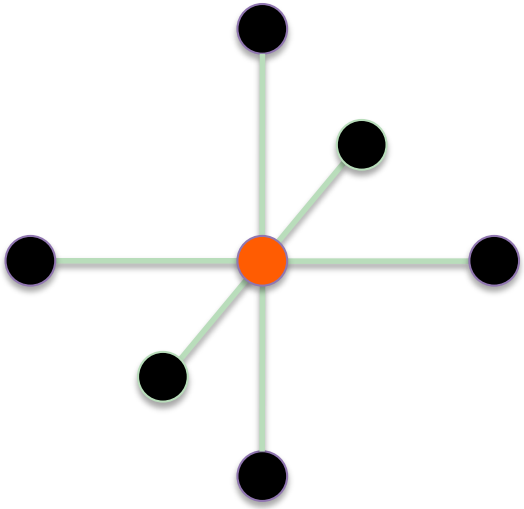
- GPUメモリ
  - GPUメモリは各GPUボード内で独立しており、共有されない
  - 異なるGPU間の直接通信不可。ホストメモリを介してデータ交換
- GPUコンテキスト
  - CUDAにおけるGPUデバイスの状態
  - CPUの1スレッドは単一の状態（コンテキスト）のみ利用可 → **複数GPUを使うには同数のCPUスレッドが必要**

プログラミング概要

- ２段階の問題分割（並列化）
- その１：GPU間並列化　←今回の内容
  - 計算対象問題をなるべく均等に利用するGPU数に分割
  - 分割した部分問題を各GPUにわりあて
  - GPU毎に1スレッド必要（プロセスでも可）
- その２：GPU内並列化
  - 割り当てられた部分問題をCUDAで並列化
  - 単一GPUを使う場合と同様

例題

- 単純な3次元拡散方程式
  - 3次元格子の各点について、X, Y, Zの3軸のそれぞれ前後の点の値をつかって更新



```
for (jz = 0; jz < nz; jz++) {
  for (jy = 0; jy < ny; jy++) {
    for (jx = 0; jx < nx; jx++) {
      FLOAT e, w, n, s, t, b, c;
      j = jz*nx*ny + jy*nx + jx;
      c = f[j];
      w = (jx == 0)    ? c : f[j-1];
      e = (jx == nx-1) ? c : f[j+1];
      n = (jy == 0)    ? c : f[j-nx];
      s = (jy == ny-1) ? c : f[j+nx];
      b = (jz == 0)    ? c : f[j-nx*ny];
      t = (jz == nz-1) ? c : f[j+nx*ny];
      fn[j] = cc*c
        + cw*w + ce*e
        + cs*s + cn*n
        + cb*b + ct*t;
    }
  }
}
```

サンプルコード

- /work/nmaruyam/gpu-tutorial/diffusion　以下にあります
  - ホームディレクトリへコピーしてお使いください
- cpu_benchmark.cpp
  - CPU逐次コード
- omp_benchmark.cpp
  - OpenMP並列CPUコード
- gpu_benchmark.cu
  - 単一GPUコード
- omp_gpu_benchmark.cpp
  - マルチGPUコード

コンパイル＆実行方法

1. ホームディレクトリへコピー

```
$ cd -r /work/nmaruyam/gpu-tutorial/
   diffusion ~（一行で）
```

2. コンパイル

```
$ cd ~/diffusion
$ make
```

3. 実行

```
$ ./bench -cpu → 単一CPU実行
$ ./bench -openmp → OpenMP並列実行
$ ./bench -gpu → 単一GPU実行
$ ./bench -multi-gpu → 複数GPU実行
```

ステップ1：GPU間並列化

- 複数GPUを使うためには同数のCPUスレッド（プロセス）が必要
  - 本講習ではOpenMPを利用してGPUと同数のCPUスレッドを実行
  - 他のマルチスレッドプログラミングでも可（pthread, Windows threads, etc)
  - MPI等を用いたマルチプロセスでも可
    - 複数ノードも利用可なため、より汎用性に優れる
    - ただしプログラミングがより煩雑

OpenMP

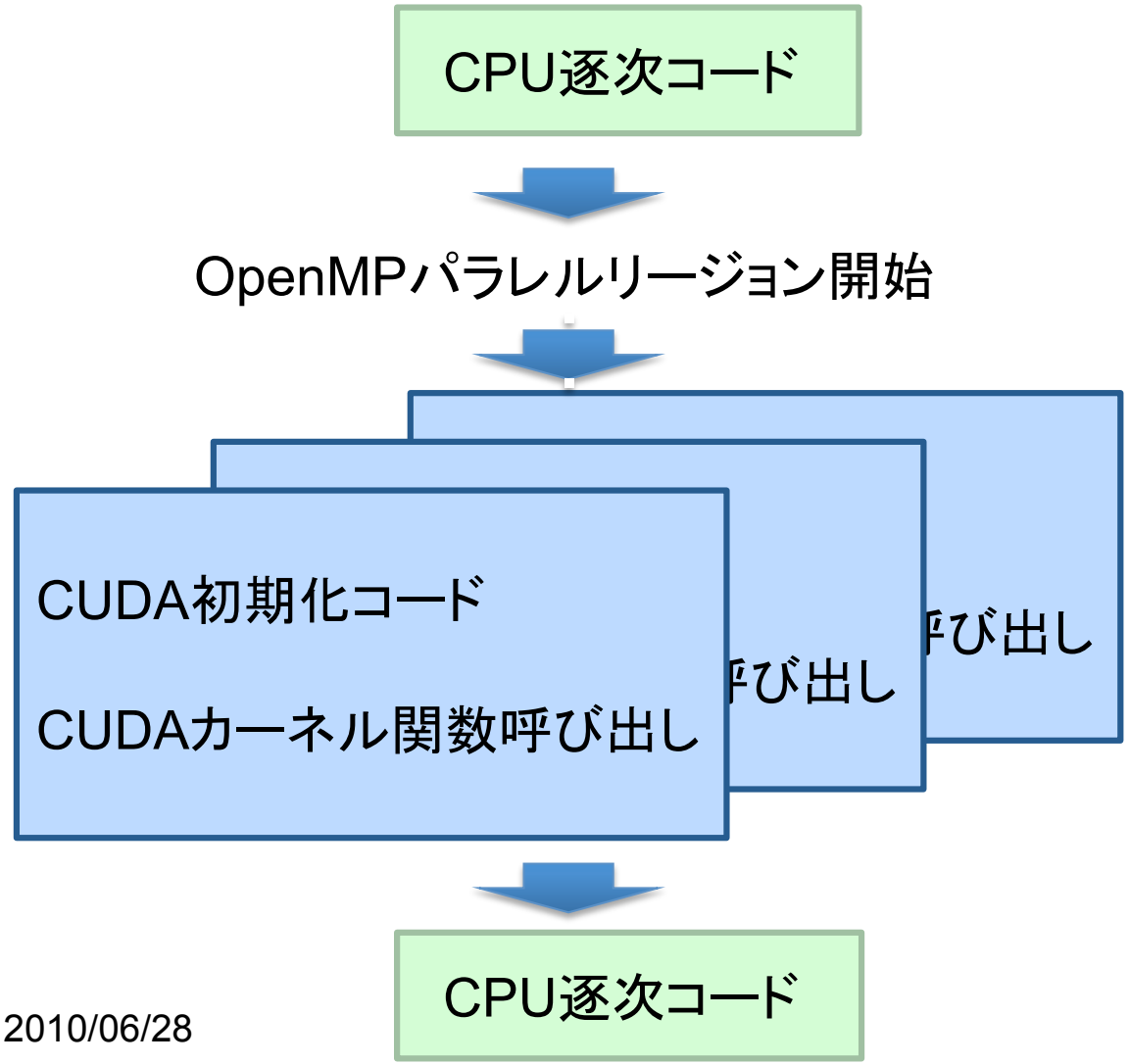
- 指示文(プラグマ)ベースのマルチスレッドプログラミング
- CとFortranを標準的にサポート
- パラレルリージョン
  - #pragma omp parallel
  - 続く文またはブロックを複数スレッドで並列実行
  - 例:Hello, world.を複数スレッドで表示

```
int main(int argc,char* argv
[]) {
   #pragma omp parallel
   printf("Hello, world.\n");
   return 0;
}
```

主要なOpenMPの関数

- スレッド番号取得
  - omp_get_thread_num
  - パラレルリージョン内のみ実行可能
- 総スレッド数取得
  - omp_get_num_threads
  - パラレルリージョン内のみ実行可能
- 実行スレッド数の設定
  - omp_set_num_threads
  - パラレルリージョン前に実行

OpenMPを用いた マルチGPUプログラミング



- #pragma omp parallel を利用

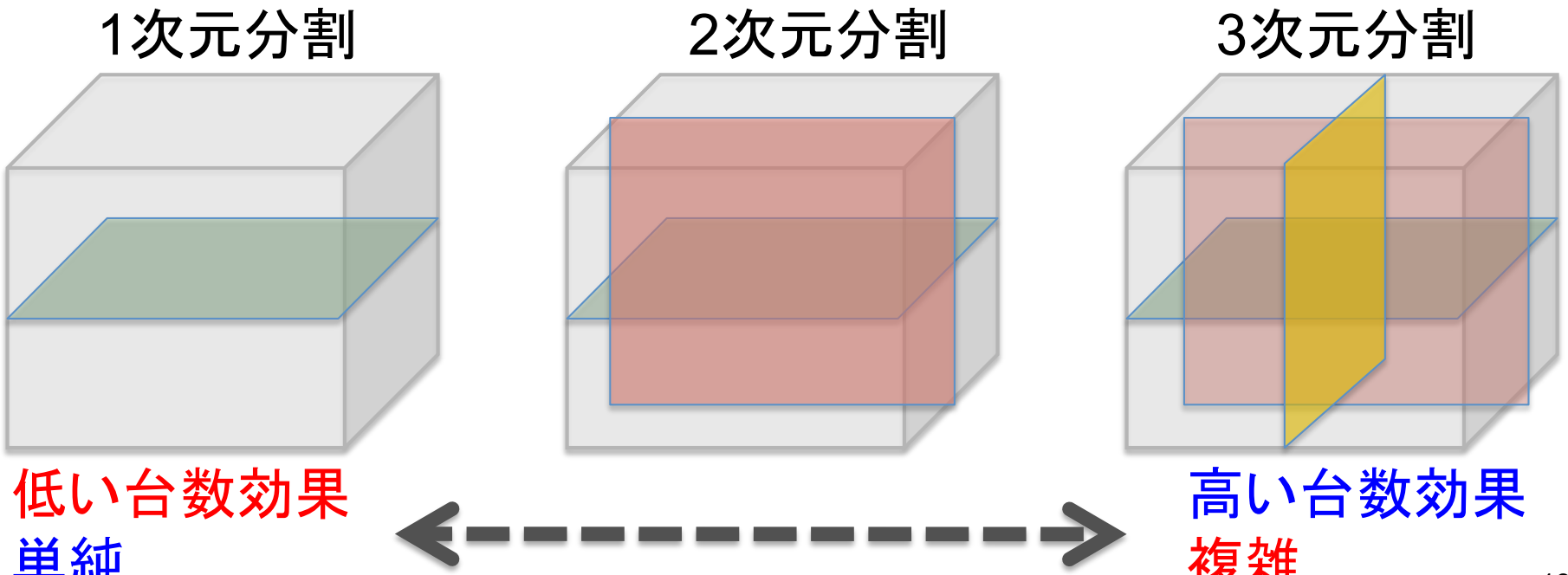
- デバイスへの接続
- GPUメモリ取得
- 部分問題のための入力データをGPUへ転送
- 問題の分割
- カーネル呼び出し

GPU間並列化のための問題分割

- 異なるGPU間ではデータの共有が不可
  - GPUカーネルは実行GPU内にあるデータのみ利用可能
  - 他のGPU上データを用いる場合は、ホストメモリを介してデータの送受信（cudaMemcpy）
  - 頻繁なデータ交換→大きな性能オーバーヘッド
- 局所性のある部分問題へ分割
  - 分散メモリ並列化（e.g., MPI並列化）と同様
  - 例：行列積→部分行列に分割
  - 例：格子系→部分格子に分割

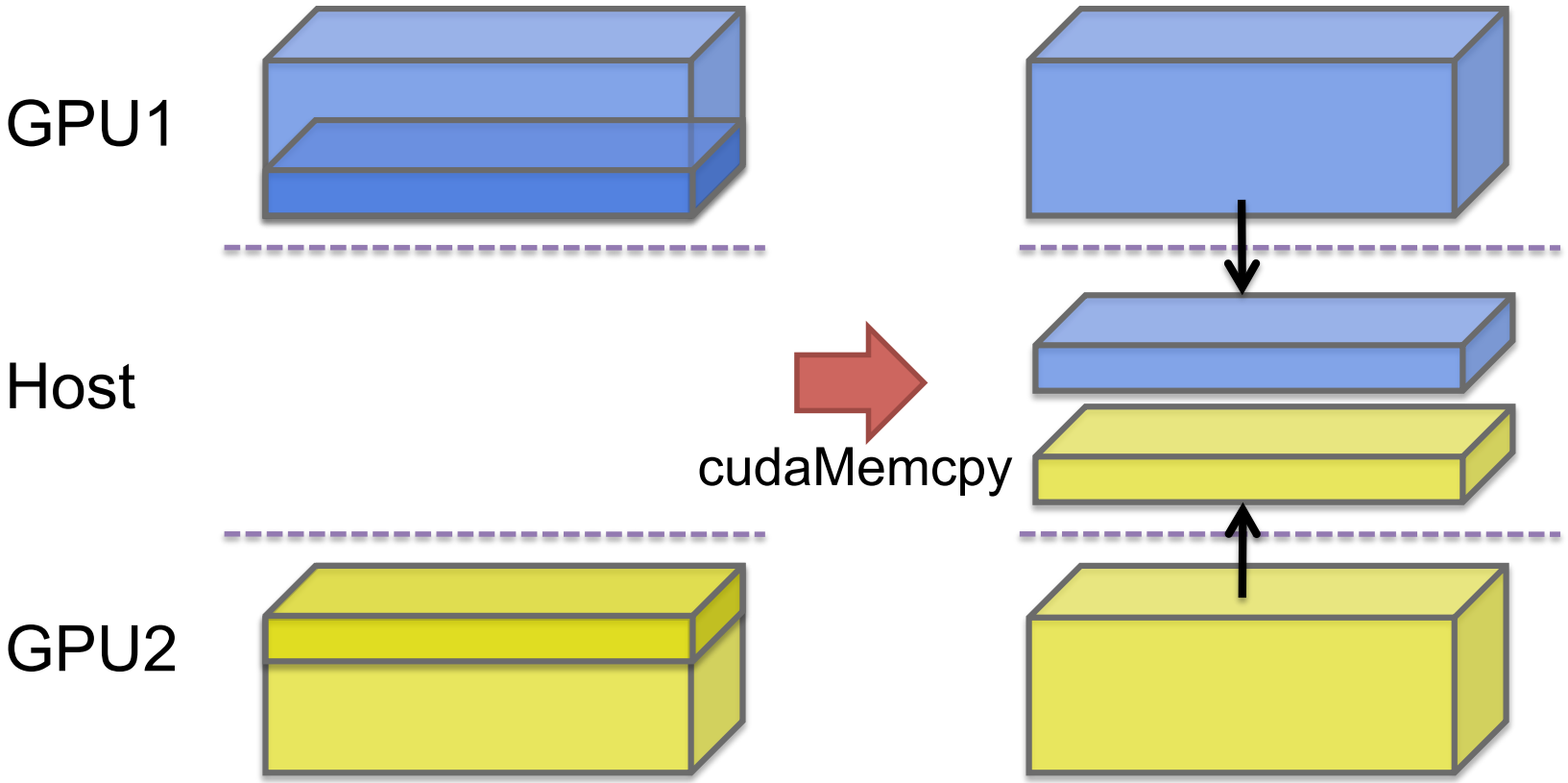
例題におけるGPU間問題分割

- 3次元グリッドをGPU間で部分グリッドへ分割
- 各部分グリッドを1GPUが計算
- 境界領域のデータ交換が必要



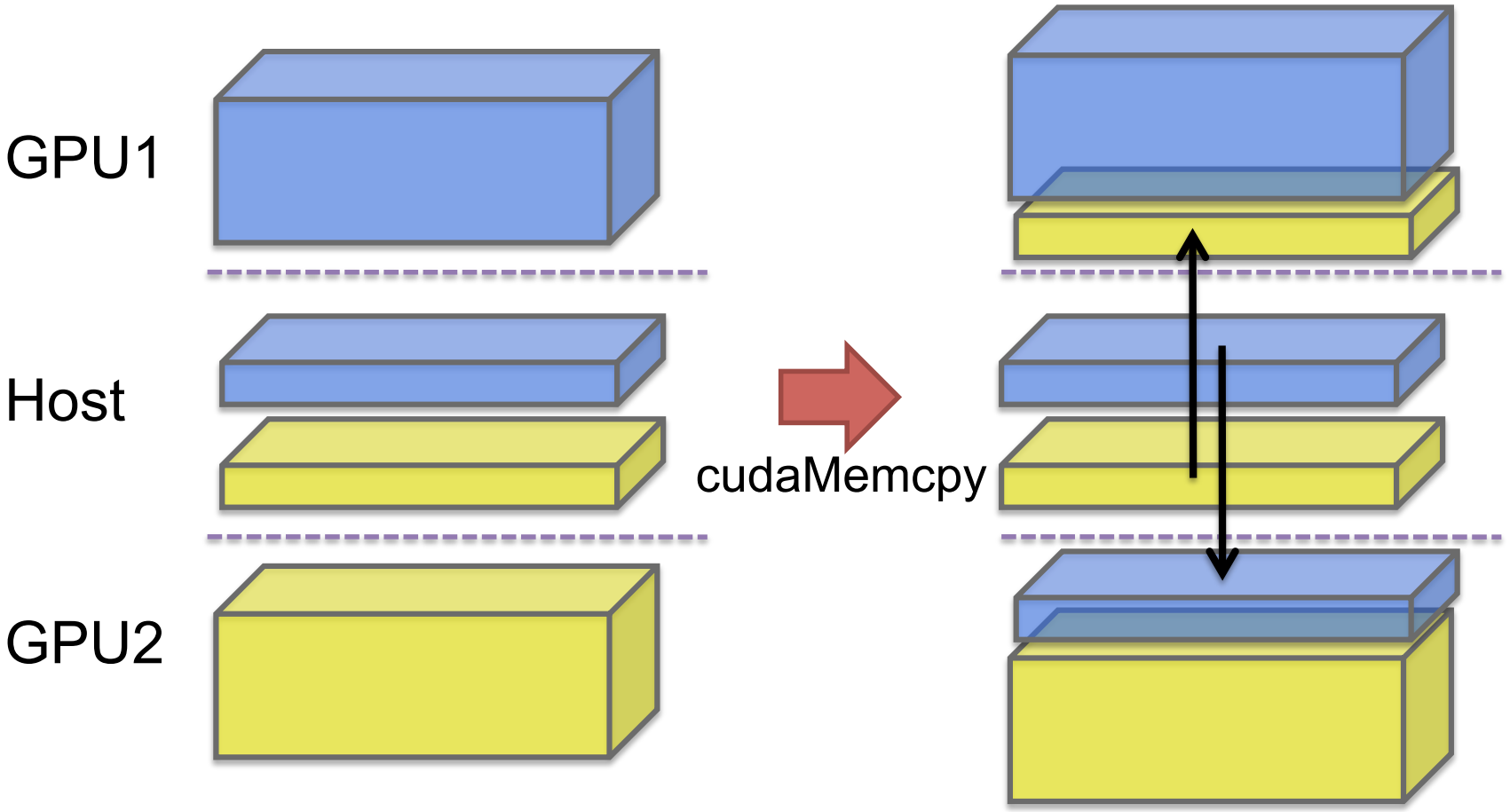
１次元分割によるマルチGPU化

ステップ１：境界領域をホストメモリへ転送



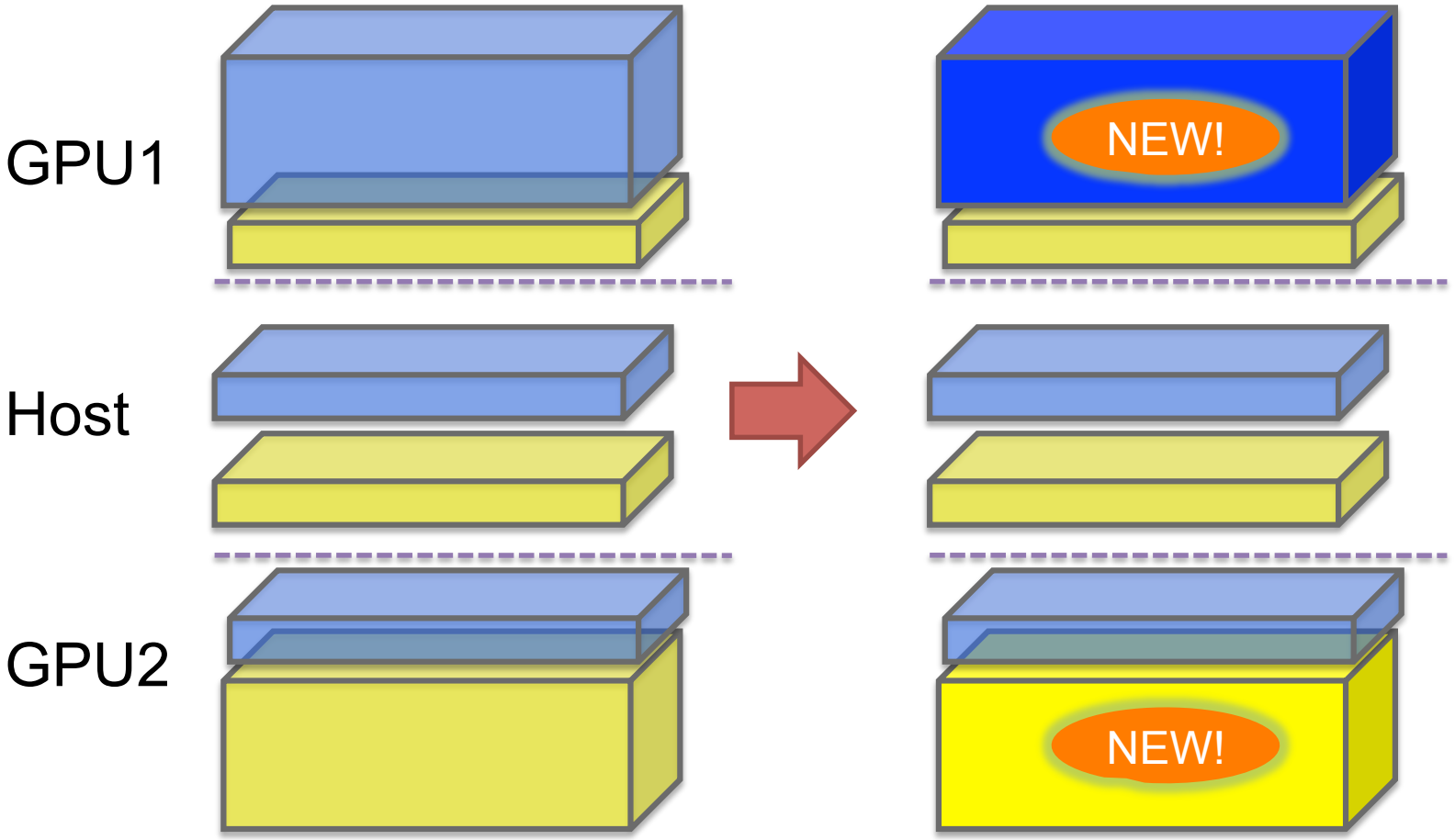
１次元分割によるマルチGPU化

ステップ２：境界領域を隣接GPUへ転送



１次元分割によるマルチGPU化

ステップ３：部分グリッドの計算（カーネル呼び出し）



ステップ2：GPU内並列化

- 単一GPUを使う場合と同様であり、複数GPUを使うための拡張は主にホスト部分
  - 扱う問題領域が部分問題に限定 → グリッドサイズを変更、スレッドブロックのサイズは一定で良い
- 例題ではカーネル関数は変更なし

例題の実装：CPU並列（OpenMP）

omp_benchmark.cpp

```
#pragma omp parallel
{
    FLOAT  time = 0.0;
    int    iter_count = 0;
    int    tid        = omp_get_thread_num();
    int    nthreads   = omp_get_num_threads();
    int    dim_z      = nz / nthreads;
    int    rem_z      = nz % nthreads;
    int    nz_self    = tid == 0 ? dim_z + rem_z : dim_z;
    do {
        diffusion3d(f,fn,nx,ny,nz_self,dx,dy,dz,dt,k);
        #pragma omp barrier
        std::swap(f, fn);
        time += dt;
        iter_count++;
    } while (time + 0.5*dt < 0.1);
}
```

例題の実装:シングルGPU

gpu_benchmark.cpp

```
cudaMalloc((void**)&f, array_size);
cudaMalloc((void**)&fn, array_size);
cudaMemcpy(f,host_buffer_,array_size,
           cudaMemcpyHostToDevice);
dim3 grid(nx/DIM_X, ny/DIM_Y, 1);
dim3 threads(DIM_X, DIM_Y, 1);
FLOAT ce = kappa*dt/(dx*dx), cw = kappa*dt/(dx*dx),
      cn = kappa*dt/(dy*dy), cs = kappa*dt/(dy*dy),
      ct = kappa*dt/(dz*dz), cb = kappa*dt/(dz*dz),
      cc = 1.0 - (ce + cw + cn + cs + ct + cb);
do {
   gpu_diffusion3d<<<grid,threads>>>
              (f,fn,nx,ny,nz,ce,cw,cn,cs,ct,cb,cc);
   std::swap(f, fn);
   time += dt; count++;
} while (time + 0.5*dt < 0.1);
```

例題の実装 : マルチGPU (1/4)

omp_gpu_benchmark.cpp

```
omp_set_num_threads(2); // Using 2 GPUs

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int  nthreads  = omp_get_num_threads();
    omp_gpu_run_diffusion3d(tid, nthreads, problem_,
                            final_time, final_iter_count);
}
```

利用するGPU数と同数のスレッド数を指定

例題の実装 : マルチGPU (2/4)

omp_gpu_benchmark.cpp

```
do {
    // Copy bottom boundary to device
    if (tid > 0) {
        cudaMemcpy(f, host_buffer - z_bound,
                   z_bound*sizeof(FLOAT),
                   cudaMemcpyHostToDevice);
    }
    // Copy top boundary to device
    if (tid < nthreads - 1) {
        cudaMemcpy(f + z_bound + size, host_buffer+size,
                   z_bound * sizeof(FLOAT),
                   cudaMemcpyHostToDevice);
    }

    ... // つづく
```

例題の実装 : マルチGPU (3/4)

omp_gpu_benchmark.cpp

```
do {
    ... // つづき
    // カーネル実行
    omp_gpu_diffusion3d<<<grid,threads>>>
            (f,fn,nx,ny,nz_self,ce,cw,cn,cs,ct,cb,cc);

    ... // つづく
```

例題の実装 : マルチGPU (4/4)

omp_gpu_benchmark.cpp

```
do {
    ... // つづき
    // Copy bottom boundary to host
    if (tid > 0) {
        cudaMemcpy(host_buffer, fn + z_bound,
                   z_bound*sizeof(FLOAT),
                   cudaMemcpyDeviceToHost);
    }
    // Copy top boundary to host
    if (tid < nthreads - 1) {
        cudaMemcpy(host_buffer + size - z_bound,
                   fn + size, z_bound*sizeof(FLOAT),
                   cudaMemcpyDeviceToHost);
    }
    std::swap(f, fn);
    time += dt; iter_count++;
} while (time + 0.5*dt < 0.1);
```

コンパイル方法

- OpenMPプログラムとCUDAプログラムが混在するためやや煩雑
- 手順
  1. 各ソースファイルをオブジェクトファイルへコンパイル
  2. 生成されたオブジェクトファイルすべてをリンク
- OpenMP部分のコンパイル方法
  - gcc: gcc –c –fopenmp foo.c
  - PGIコンパイラではオプション不要でサポート
- CUDA部分のコンパイル方法
  - nvcc
- CUDAとOpenMPが同一ソースファイルに共存する場合
  - nvcc –c –Xcompiler –fopenmp foo.cu

コンパイル方法

- リンク方法
  - nvccもしくはg++を用いてリンク
  - OpenMPのライブラリを指定する必要あり (libgomp)
  - g++ host.o gpu.o –lgomp
- 注意点
  - nvccはC++オブジェクトコードを生成するため、リンク時にはC++用のコマンドでリンクします
  - コンパイルにはPGIコンパイラなども使えますが、リンクはnvccもしくはg++である必要があります

コンパイルの注意点

- nvccはC++オブジェクトコードを生成するため、リンク時にはC++用のコマンドでリンクします
- コンパイルにはPGIコンパイラなども使えますが、リンクはnvccもしくはg++である必要があります
  - ただしC++プログラムの場合はコンパイルをg++で行う必要あり

コンパイルの注意点（続き）

- （TSUBAME固有）TSUBAMEのデフォルトgccではなく、以下の場所にある新しいバージョンのgccを使う必要あり
  - /work/nmaruyam/gcc-4.2.4
  - 理由
    - デフォルトのgccのバージョンは4.1と古いため、OpenMPをサポートしてません。TSUBAMEではOpenMPのコンパイルには通常はPGIコンパイラを用いますが、C++の場合はPGIコンパイラによって生成されたオブジェクトファイルとCUDAプログラムをリンクできず、gccを使う必要があります（nvccの制約）。従って、OpenMPをサポートしたよりバージョン4.2以降のgccを使う必要があります
    - TSUBAME2ではシステムが更新されるためデフォルトのgccを利用可能になります

コンパイルの注意点(続き)

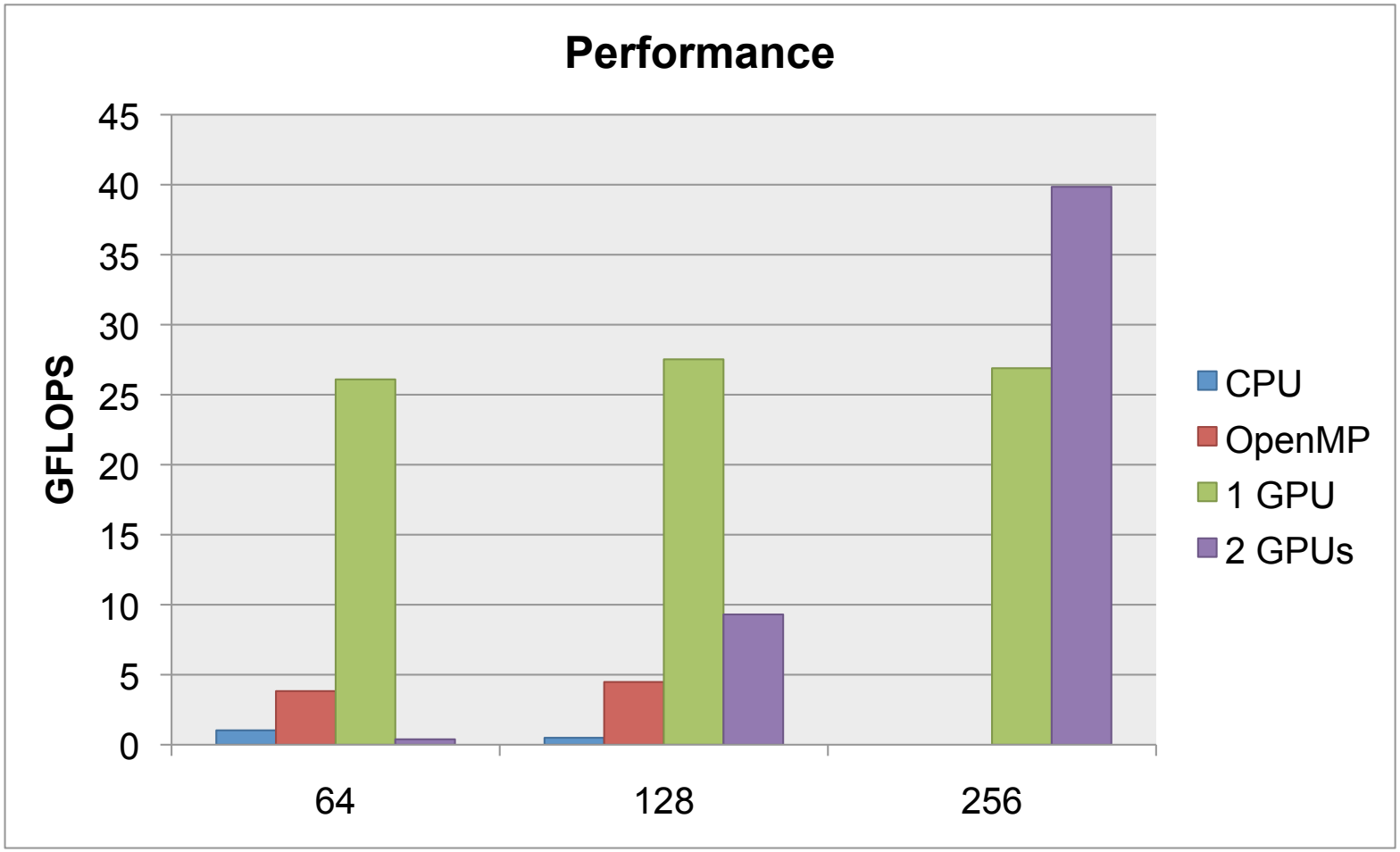
- TSUBAMEでの方法(おすすめ)

```
$ PATH=/work/nmaruyam/gcc-4.2.4/bin:$PATH
$ nvcc -c test.cu -Xcompiler -fopenmp
$ nvcc test.o -o test -lgomp -Xlinker \
    -rpath=/work/nmaruyam/gcc-4.2.4/lib64
```

- サンプルプログラムのMakefileも参照
- PGIコンパイラなどとの併用も可
  - ご相談ください

性能比較

TSUBAME 1ノードで計測 （CPU： CPU1スレッド、OpenMP: CPU16スレッド）



考察

- 問題サイズが小さい場合は1GPUより遅い！
  - 境界領域をGPU間で交換する時間がかかるため
- 転送コストの削減
  - DMA転送を有効にする
    - cudaMemcpyは通常のmallocやnewで確保した領域とGPUメモリとの間ではDMA転送不可（今回の実装）
    - ホスト側バッファをcudaMallocHostで確保した場合はDMA転送が有効
  - 非同期転送を用い、カーネルの実行と転送をオーバーラップさせる
    - 例題では境界領域の計算と内部の計算を別のカーネルにすることで可能
- １次元分割ではなく、２次元、３次元分割にする
  - 特に多数のGPUを使う場合に有効

補足資料

GPUコンテキスト

- CPUスレッドが保持する、利用中GPUの状態をあらわすデータ
  - GPU上のメモリ、GPUカーネルなど
- １つのコンテキストは単一のGPUに限定
  - 複数のGPUにまたがるようなコンテキストは不可
- 複数のスレッド（プロセス）間でのコンテキストの共有不可
- １つのCPUスレッド（プロセス）が持てるコンテキストは同時に1つまで

コンテキスト管理

- ランタイムAPIではコンテキストは暗黙的にCUDAランタイムが管理
  - 最初にCUDA APIを呼び出した時点で作成
  - コンテキストをもったCPUスレッドの終了時、もしくはcudaThreadExitの呼び出しによって破棄
- ドライバAPIではより詳細なコンテキストが管理が可能
  - 異なるスレッド間でのコンテキストの受け渡しなど
  - CUDA SDK内のthreadMigrationサンプルコードを参照