第115回お試しアカウント付き 並列プログラミング講習会 「GPUプログラミング入門」

星野哲也 hoshino@cc.u-tokyo.ac.jp

2019年4月24日 (水)

東京大学情報基盤センター

スケジュール（改）

- 09:30 - 10:00　受付
- 10:00 - 12:00　Reedbush-Hログイン、スパコンの使い方、
  並列プログラミングとは、GPUとは
- 13:30 - 14:30　GPUプログラミング入門
- 14:45 - 18:00　GPUプログラミング演習　（適宜休憩）
  - 簡単なプログラムのGPU実行
  - 3次元拡散方程式
  - FDTD法による電磁波伝搬計算
  - 質問など

事前準備

1. Reedbushにログインする（別資料を参照）
   ※ログインノードは-U/-H共通
2. 利用上の注意

Reedbush 利用上の注意(1)

- ディレクトリについて(home と lustre)
  - ✓ ログイン時のディレクトリ(/home/gt00/txxxxx)にはログイン時に必要なファイルのみを置く
  - ✓ プログラム作成や実行などに必要なファイルは /lustre 以下のディレクトリ(/lustre/gt00/txxxxx)に置く
  - ✓ /home は計算ノードからは参照できない
  - ✓ cdw コマンドで Lustreファイルシステムへ移動できる。

    $ cdw

- 実行中のジョブの確認
  - ✓ ジョブ投入はqsubで行うが、ジョブ確認は qstat ではなく rbstat

Reedbush 利用上の注意(2)

- コンパイルおよび実行のための環境準備
  - ✓ コンパイルおよび実行のための環境を準備するために module コマンドを使用する。これによって様々な環境を簡単に切り替えて使用できる。

$ module load <*module_name*>

モジュール名 **<*module_name*>** のモジュールをロードして環境を準備。環境変数PATHなどが設定される。

$ module avail

使用可能なモジュール一覧を表示する。

$ module list

使用中のモジュールを表示する。

モジュールの切り替え

- PGIコンパイラ（OpenACCやCUDA Fortran）を使う場合
  $ module load pgi
- CUDA開発環境を使う場合
  $ module load cuda
- Intelコンパイラを使う場合
  $ module load intel
- MPIを使う場合
  $ module load openmpi/gdr/2.1.1/{gnu,intel,pgi}
  $ module load mvapich2/gdr/2.2/{gnu,intel,pgi}
  ✓ PGIなどのコンパイラに追加して load

- モジュールはジョブ実行時にもコンパイル時と同じものを load する。
- 組み合わせて利用できる

サンプルコードのコンパイル

- Reedbush へのログイン
  $ ssh -Y txxxxx@reedbush.cc.u-tokyo.ac.jp
  ✓ txxxxx 各自の利用者番号（アカウント）に置き換えてください。
  ✓ -Y をつけてください。
- cdw コマンドで Lustreファイルシステムへ移動する。
  $ cdw
- 自分のディレクトリにサンプルコードをコピーする。
  $ cp /lustre/gt00/share/openacc_samples.tar.gz .
- サンプルコードを展開する。
  $ tar zxvf openacc_samples.tar.gz
- サンプルコードへ移動する。
  $ cd openacc_samples
- モジュールをロードする。
  $ module load pgi
- コンパイルする。
  $ cd openacc_hello/01_hello_acc
  $ make
- 実行ファイルがてきていることを確認する。
  $ ls

プログラムの実行

- ジョブとして投入し、実行する。

  $ qsub ./run.sh

- 投入されたジョブを確認する。

  $ rbstat

- 実行が終了すると、以下のファイルが生成される。

  run.sh.o??????

  run.sh.e??????　（?????? は数字）

- 上記の標準出力ファイルの中身を確認する。

  $ cat run.sh.o??????

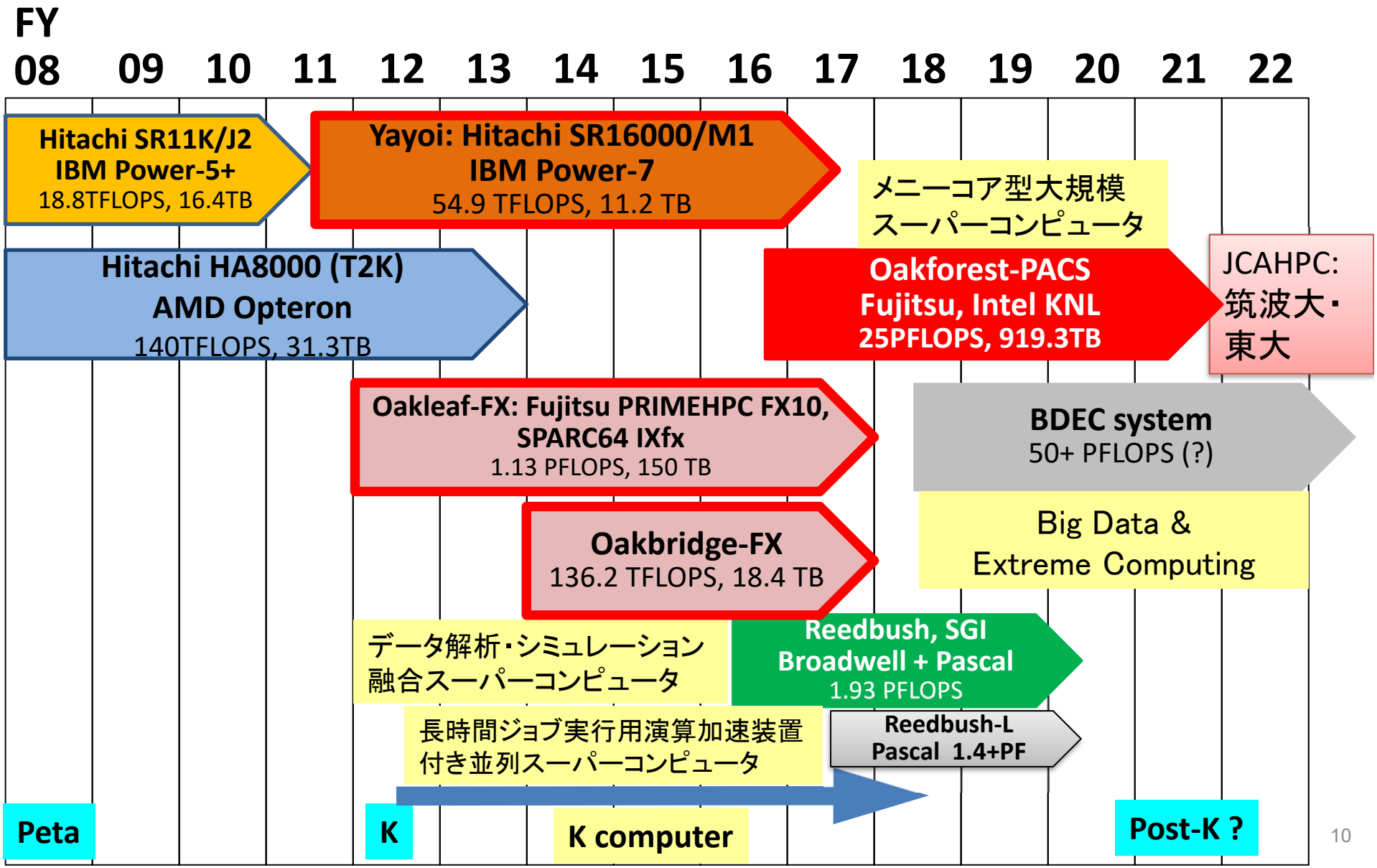
- 必要に応じて、上記のエラー出力ファイルの中身を確認する。

  $ cat run.sh.e??????

当センターの運用システム

東大センターのスパコン

2基の大型システム，6年サイクル



2システム運用中

- Reedbush (SGI, Intel BDW + NVIDIA P100 (Pascal))
  - データ解析・シミュレーション融合スーパーコンピュータ
  - 3.361 PF, 2016年7月～ 2020年6月
  - 東大ITC初のGPUシステム (2017年3月より), DDN IME (Burst Buffer)

- Oakforest-PACS (OFP) (富士通、Intel Xeon Phi (KNL))
  - JCAHPC (筑波大CCS＆東大ITC)
  - 25 PF, TOP 500で9位 (2017年11月) (日本で2位)
  - Omni-Path アーキテクチャ, DDN IME (Burst Buffer)

Reedbush

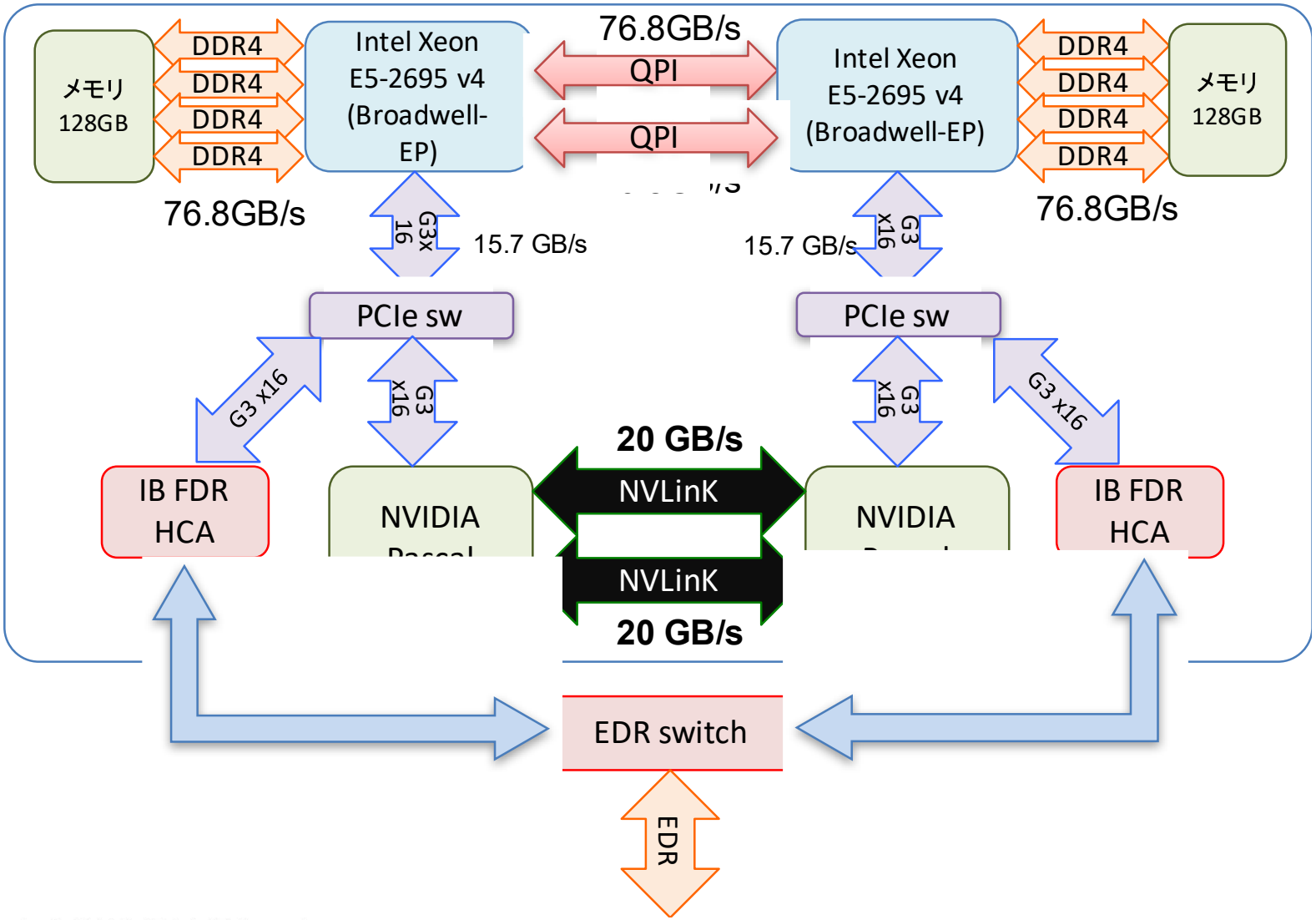
Reedbush（SGI Rackable クラスタシステム）

Reedbush-U (2016/7/1 ~)
- 理論性能：508TFlops
- ノード数：420
- ノード構成：Intel Xeon Broadwell x2

Reedbush-H (2017/3/1 ~)
- 理論性能：1418TFlops
- ノード数：120
- ノード構成：Intel Xeon Broadwell x2 + **NVIDIA P100 GPU x2**

Reedbush-L (**2017/10/1 ~**)
- 理論性能：1435TFlops
- ノード数：64
- ノード構成：Intel Xeon Broadwell x2 + **NVIDIA P100 GPU x4**

Reedbush-Hノードのブロック図



スーパーコンピュータシステムの詳細

- 以下のページをご参照ください
  - 利用申請方法
  - 運営体系
  - 料金体系
  - 利用の手引

  などがご覧になれます。

  http://www.cc.u-tokyo.ac.jp/guide

GPUプログラミングを始める前に！

1. 並列プログラミングって？

GPUプログラミングを始める前に！

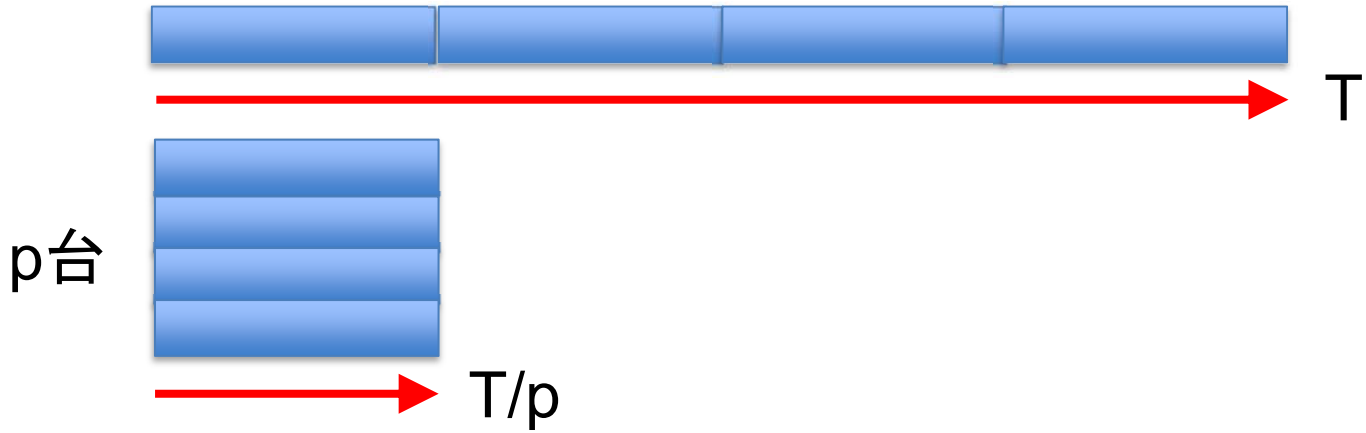
- GPUは**並列計算機**です！よって本講習会で学ぶのは**並列プログラミング**になります！
  - 並列プログラミングの例：MPI, OpenMP, pthread など

- 並列プログラミングは、**プログラムを高速化**するために行います！

並列プログラミング・高性能プログラミングについての事前知識があると有利！

並列計算による高速化

- 実行時間Tの逐次処理のプログラムをp台の計算機で並列計算することで、実行時間をT/pにする。

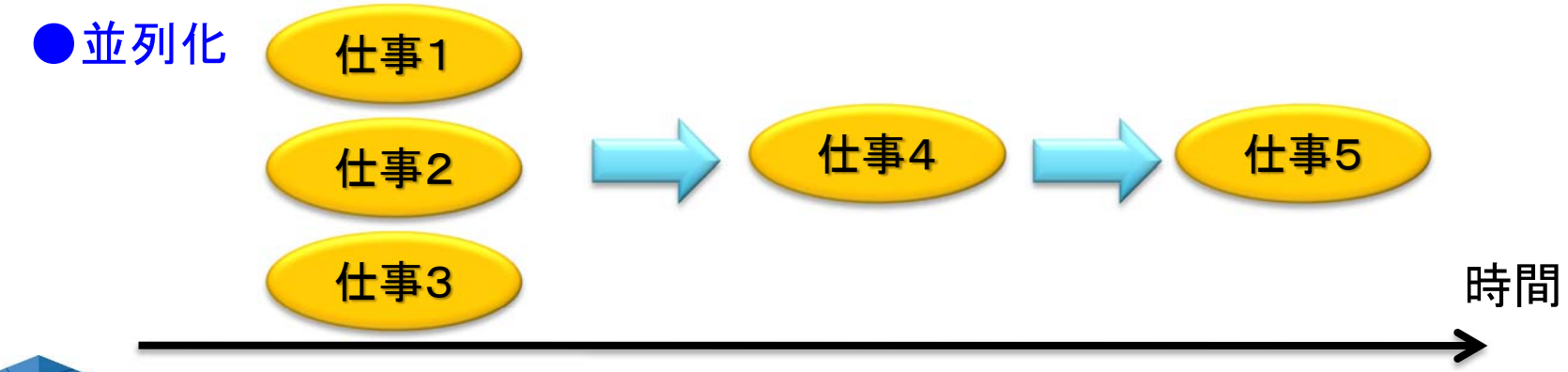


- 実際にできるかどうかは、処理内容(アルゴリズム)による。アルゴリズムによって難易度は異なる。
  - ✓ 部分的にでも並列化できない場所があると、どれだけ並列数を上げても、その時間は短縮されない。**→アムダールの法則**

タスク並列

- タスク(ジョブ)を分割することで並列化する。
- データの操作(＝演算)は異なるかもしれない。
- タスク並列の例:カレーを作る
  - 仕事1:野菜を切る
  - 仕事2:肉を切る
  - 仕事3:水を沸騰させる
  - 仕事4:野菜・肉を入れて煮込む
  - 仕事5:カレールゥを入れる

GPUは苦手



データ並列

- データを分割することで並列化する。
  - ✓ データは異なるが計算の手続きは同じ。
- データ並列の例：手分けをして算数ドリルを解く
  - ✓ 数字だけ異なるが計算の手続きは同じ。

2 + 1 =	12 − 88 =	34 + 3 =	2 + 4 =
3 + 19 =	− 20 + 29 =	1 + 2 =	99 − 72 =
4 − 6 =	4 + 10 =	−3 + 2 =	2 + 10 =
5 + 3 =	1 + 10 =	−10 − 10 =	3 − 11 =
−2 + 10 =	1 + 13 =	2 + 1 =	12 − 34 =
1 + 2 =	0 + 0 =	1 −10	45 + 19 =

GPUではこれが原則！

プログラムにおける並列化

- 基本的にはループを並列化する
  - ✓ 従ってループ以外の部分がアムダールの法則の並列化できない部分になる
  - ✓ ループの要素単位で並列化。ループ長が100なら最大100並列
  - ✓ ただし簡単には並列化できないループもある
    - 以下は並列化可能？

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

```
for ( i = 0; i < 3; i++)
    A[0] = A[0] + 1;
```

```
for ( i = 0; i < 3; i++)
    A[i+1] = A[i] + 1;
```

```
for ( i = 0; i < 3; i++)
    A[i] = B[i] + C[i];
```

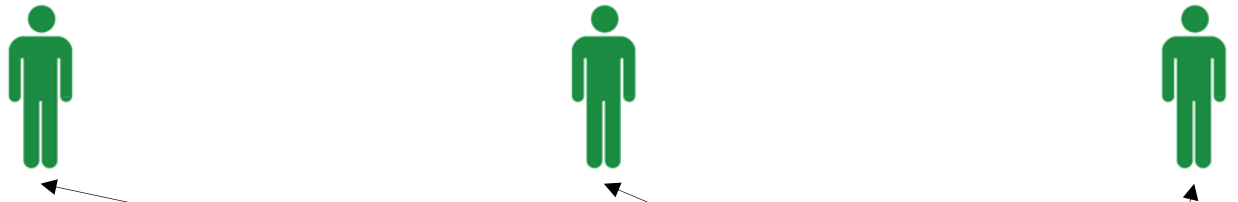
```
for ( i = 0; i < 3; i++)
    b = b + A[i];
```

```
for ( i = 0; i < 3; i++)
    b = A[i] / b;
```

簡単に並列化できるループ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

A[0] = A[0] + 1;　　**A[1] = A[1] + 1;**　　**A[2] = A[2] + 1;**



分担で計算を行う
スレッドさん

メモリ

3	6	1
A[0]	A[1]	A[2]

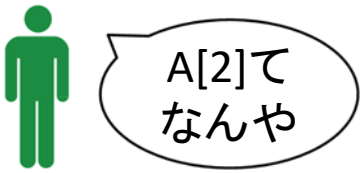
簡単に並列化できるループ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

A[0] = A[0] + 1;

A[1] = A[1] + 1;

A[2] = A[2] + 1;



メモリ

3	6	1
A[0]	**A[1]**	**A[2]**

簡単に並列化できるループ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

A[0] = A[0] + 1;

A[1] = A[1] + 1;

A[2] = A[2] + 1;

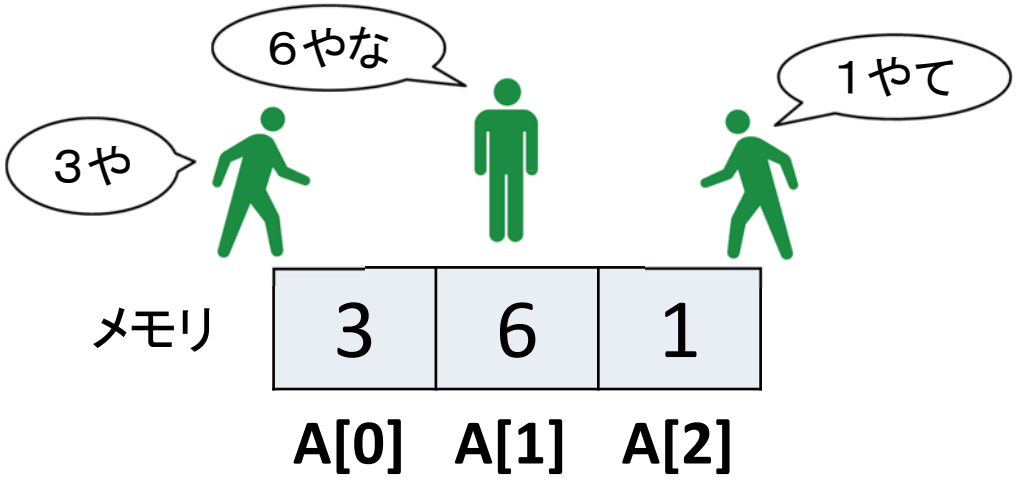
メモリ

3	6	1
A[0]	A[1]	A[2]

簡単に並列化できるループ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

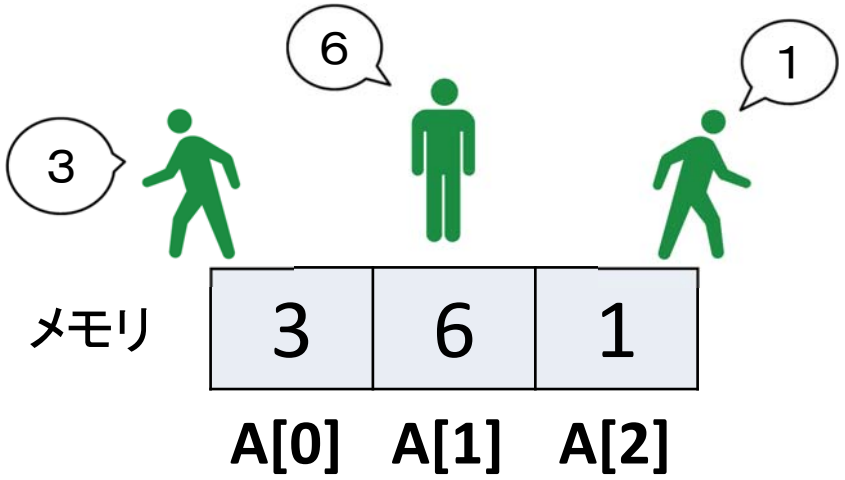
A[0] = A[0] + 1;　　**A[1] = A[1] + 1;**　　**A[2] = A[2] + 1;**



簡単に並列化できるループ

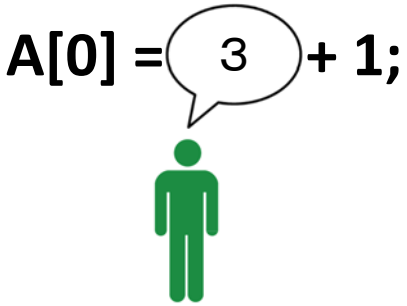
```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

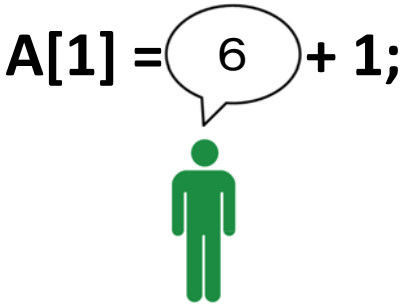
A[0] = A[0] + 1;　　**A[1] = A[1] + 1;**　　**A[2] = A[2] + 1;**

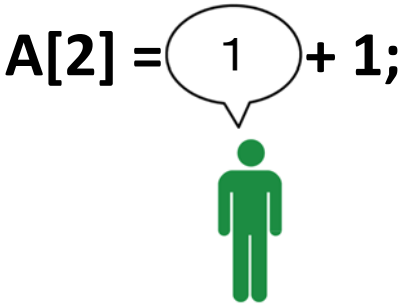


簡単に並列化できるループ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

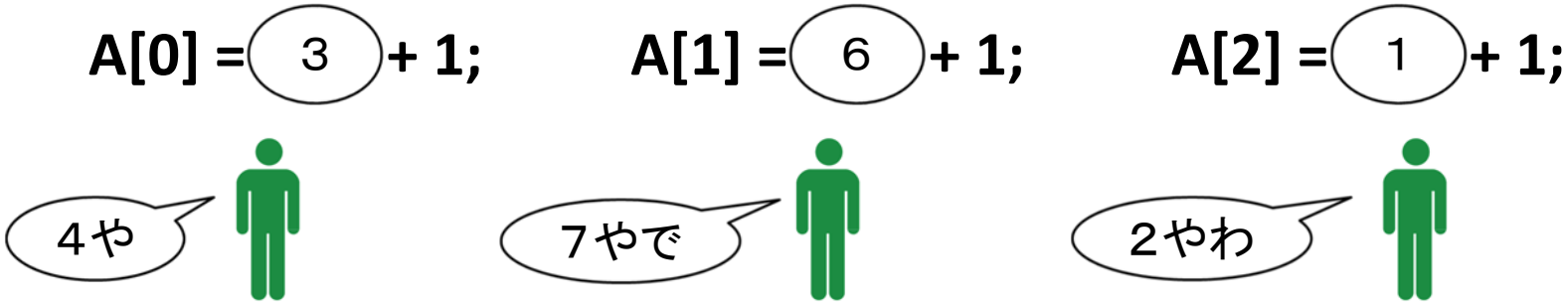






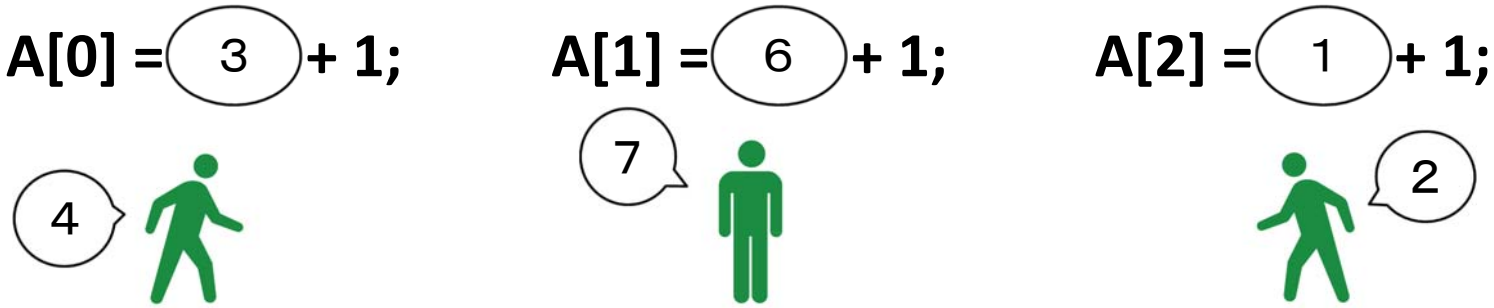
簡単に並列化できるループ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```



簡単に並列化できるループ

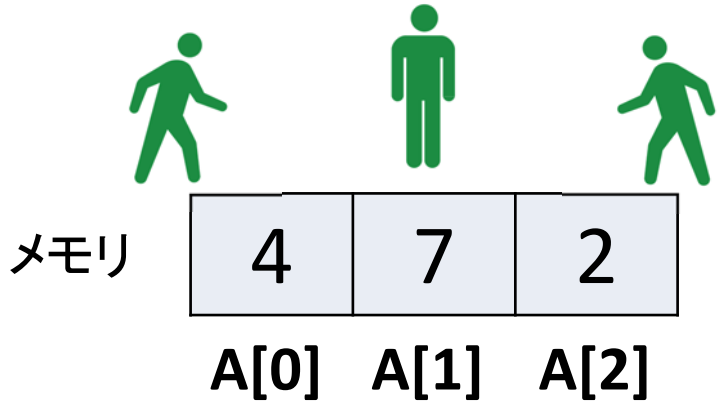
```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```



簡単に並列化できるループ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

A[0] = (3) + 1;　　A[1] = (6) + 1;　　A[2] = (1) + 1;



簡単に並列化できるループ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

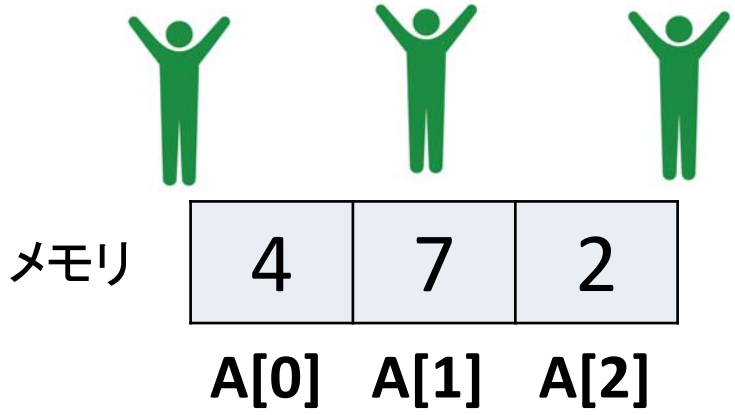
A[0] = 3 + 1;　　A[1] = 6 + 1;　　A[2] = 1 + 1;

このようなデータ並列を簡単に
適用できるループを、

- データ独立なループ
- 依存性のないループ
- 自明な並列性を持つループ

などと呼ぶ

成功！



簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

A[0] = A[0] + 1;　　**A[0] = A[0] + 1;**　　**A[0] = A[0] + 1;**

A[0]に3回1を足してるだけなので、
最終結果は 3 + 1 + 1 + 1 = 6。

足し算なのでどんな順番で足しても
結果は変わらないはずだが…

メモリ

3	6	1
A[0]	**A[1]**	**A[2]**

簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

A[0] = A[0] + 1;

A[0] = A[0] + 1;

A[0] = A[0] + 1;

メモリ

3	6	1
A[0]	**A[1]**	**A[2]**

簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

A[0] = A[0] + 1;　　**A[0] = A[0] + 1;**　　**A[0] = A[0] + 1;**

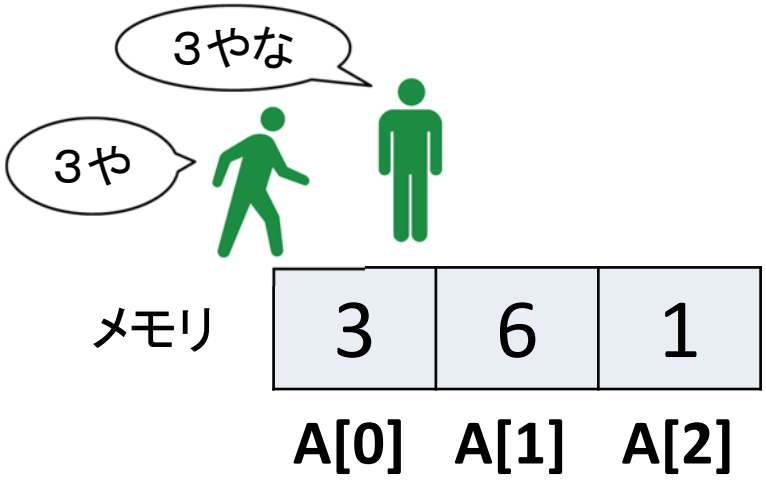
メモリ

3	6	1
A[0]	A[1]	A[2]

簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

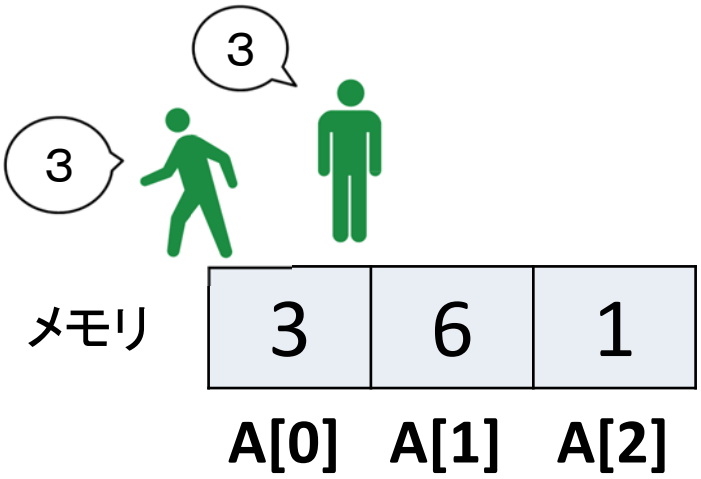
A[0] = A[0] + 1;　　**A[0] = A[0] + 1;**　　**A[0] = A[0] + 1;**



簡単に並列化できないループ

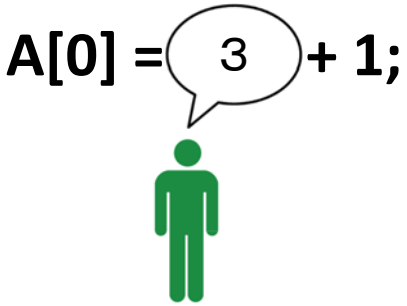
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

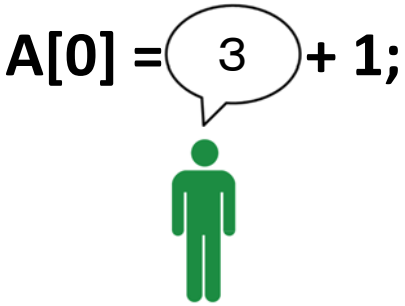
A[0] = A[0] + 1;　　**A[0] = A[0] + 1;**　　**A[0] = A[0] + 1;**

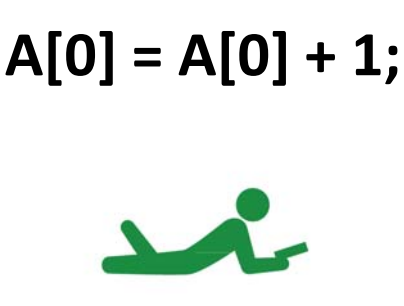


簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```





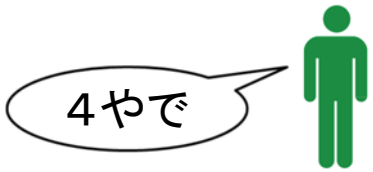


簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

A[0] = 3 + 1;

A[0] = 3 + 1;



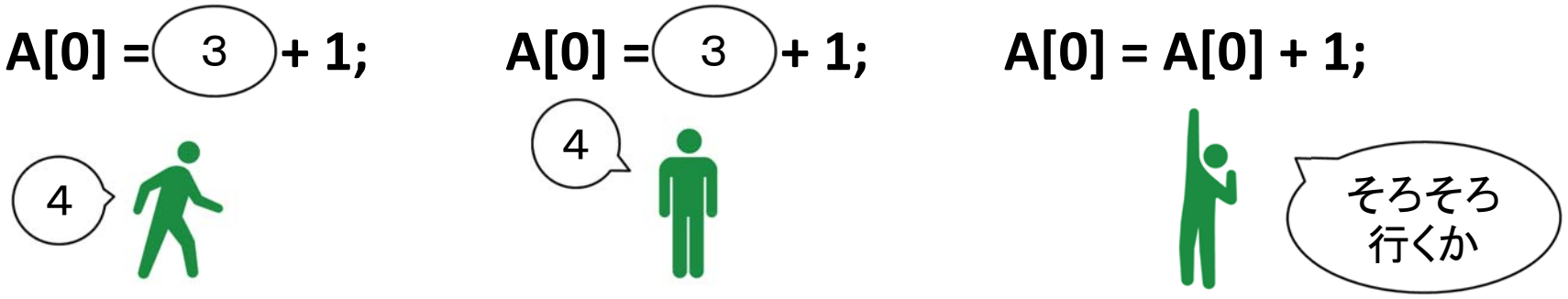
A[0] = A[0] + 1;

メモリ

3	6	1
A[0]	A[1]	A[2]

簡単に並列化できないループ

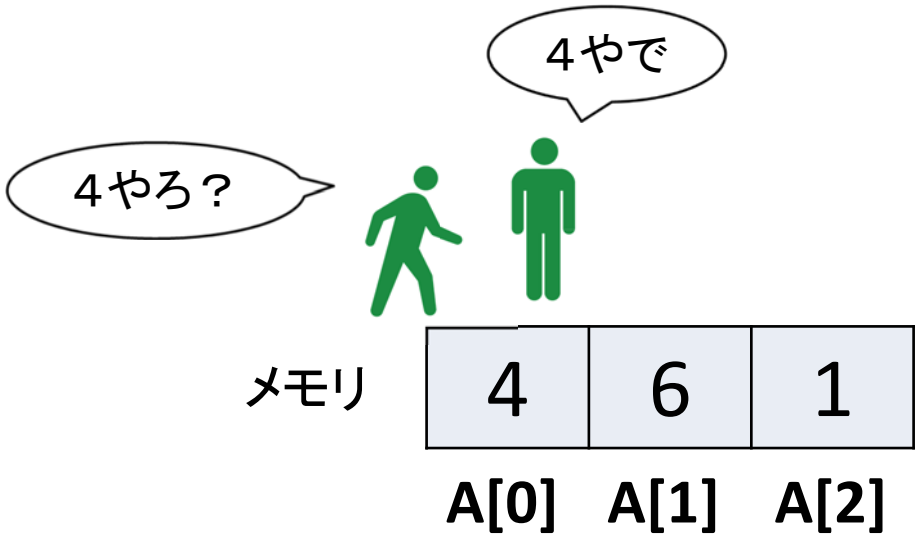
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```



簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

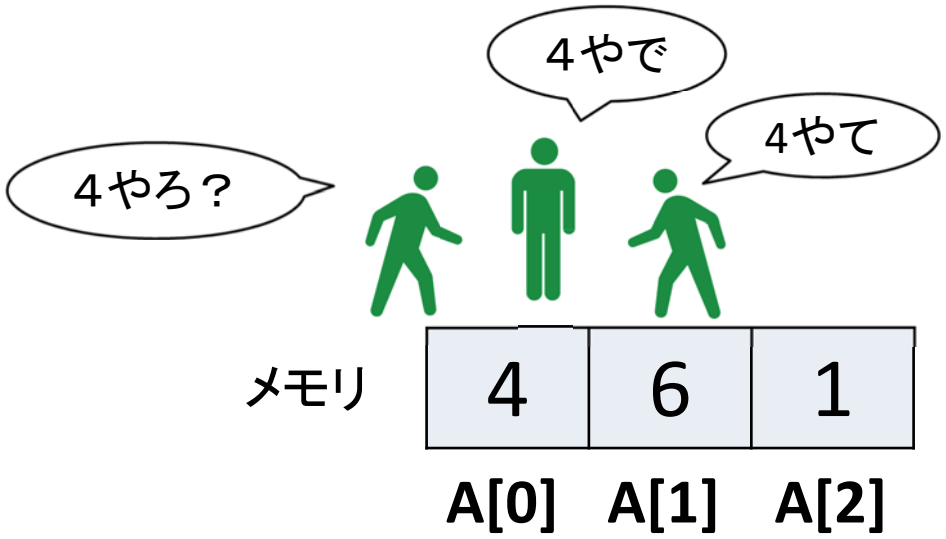
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

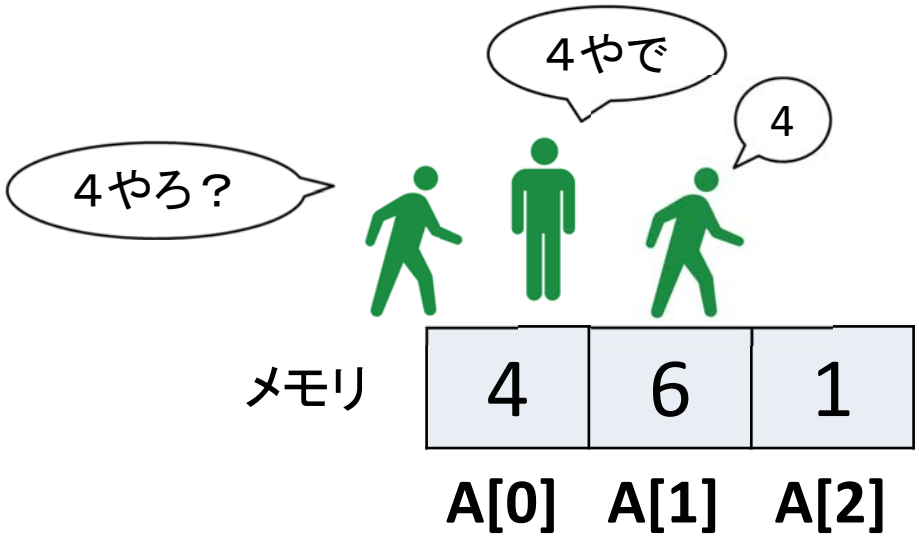
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

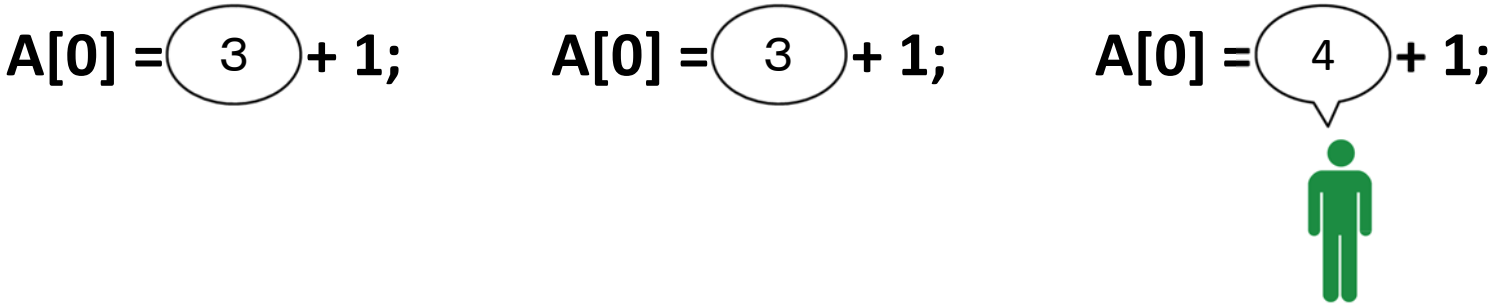
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

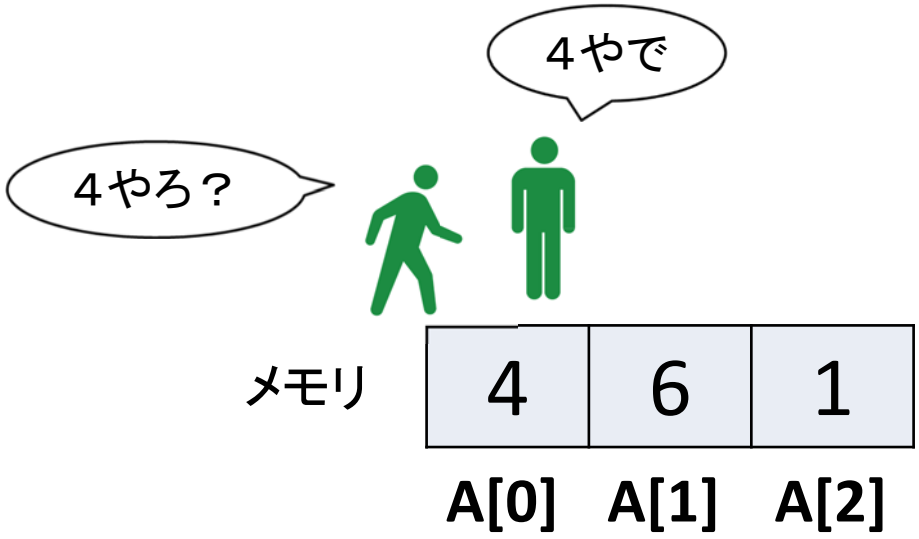
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

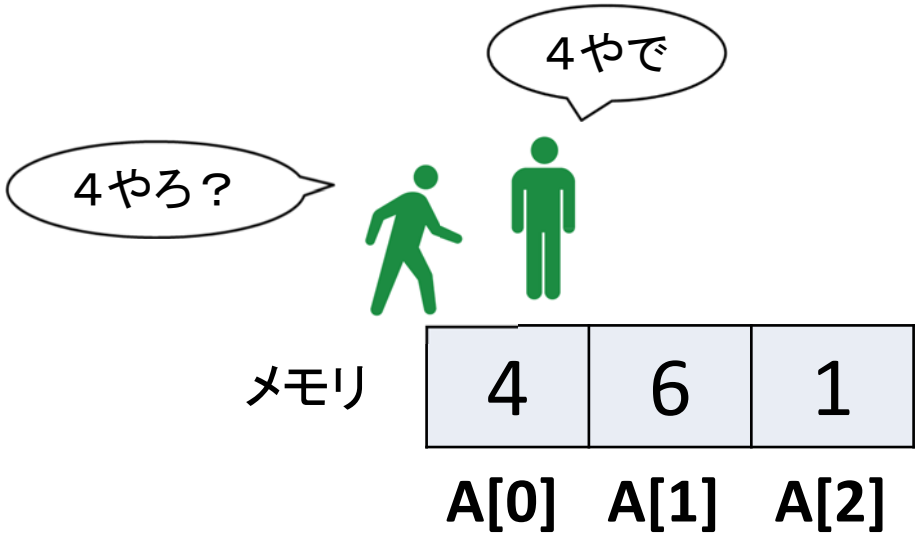




簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

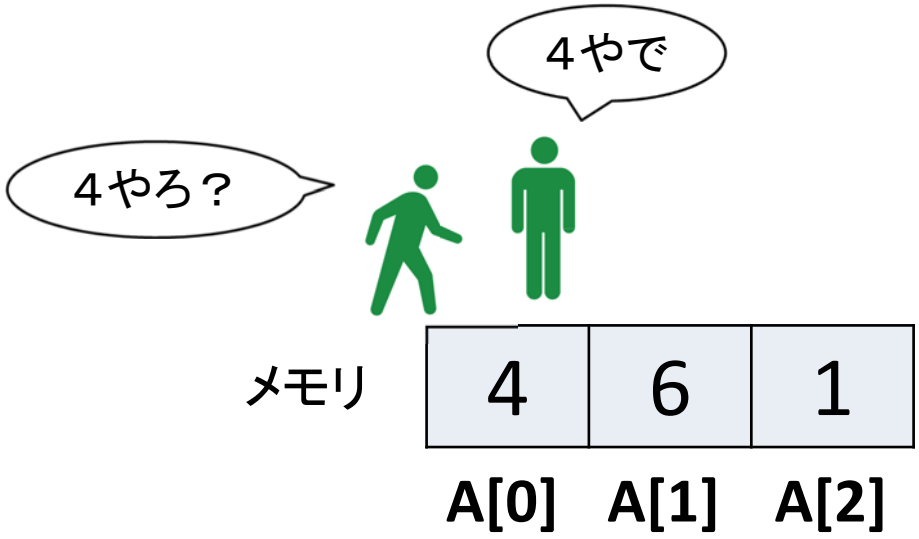
A[0] = (3) + 1;　　A[0] = (3) + 1;　　A[0] = (4) + 1;



簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = 4 + 1;

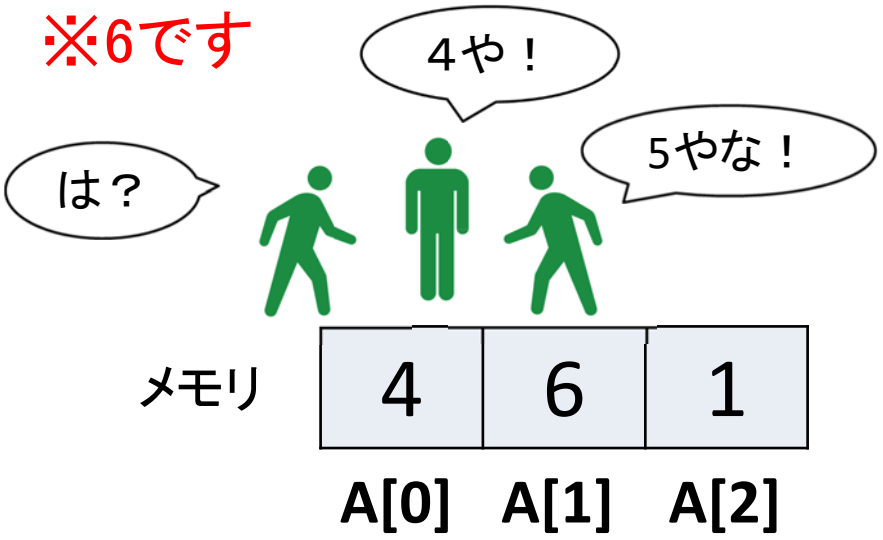


簡単に並列化できないループ

```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

A[0] = (3) + 1;　　A[0] = (3) + 1;　　A[0] = (3) + 1;

※6です



CPUで実行される足し算は、

1. データの読み込み
2. 足し算
3. データの書き込み

の3パートからなる。

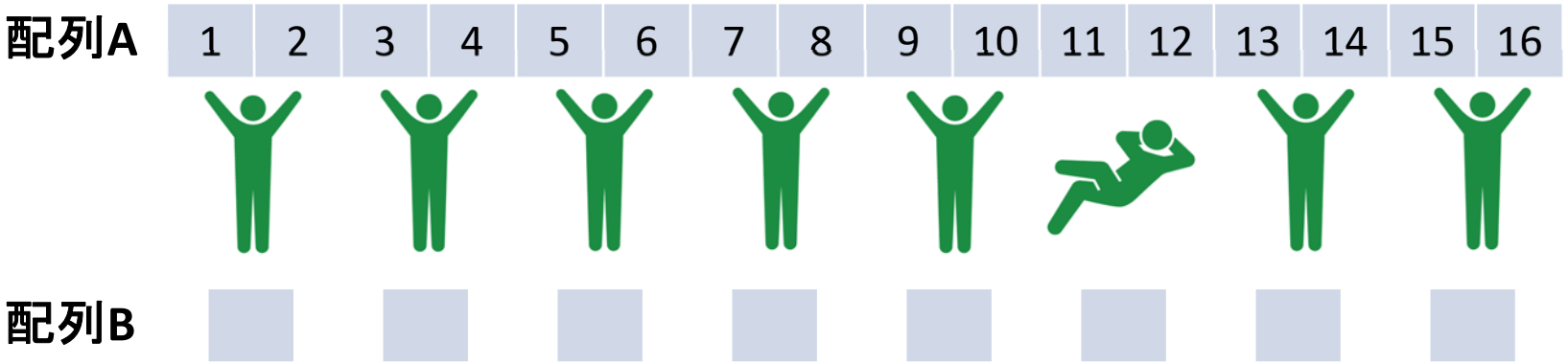
スレッドは各々独立に1～3を実行するため、タイミングによって結果が変わる！

（この例の場合は4,5,6のいずれかになる）

例えば以下を8スレッドで並列化

どうやって並列化するか？

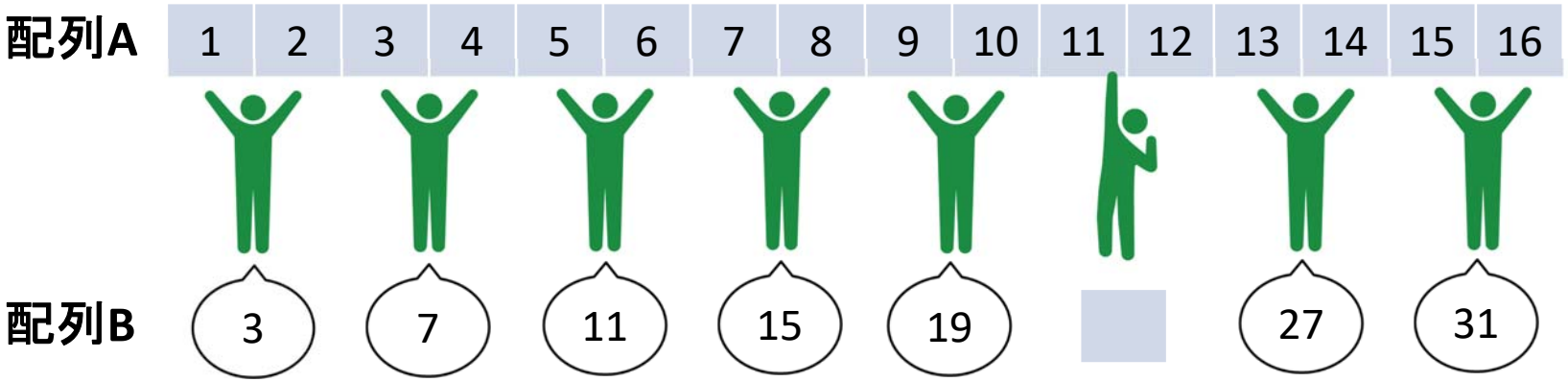
```
sum = 0;
for ( i = 0; i < 16; i++)
    sum = sum + A[i];
```



どうやって並列化するか？

例えば以下を8スレッドで並列化

```
sum = 0;
for ( i = 0; i < 16; i++)
    sum = sum + A[i];
```

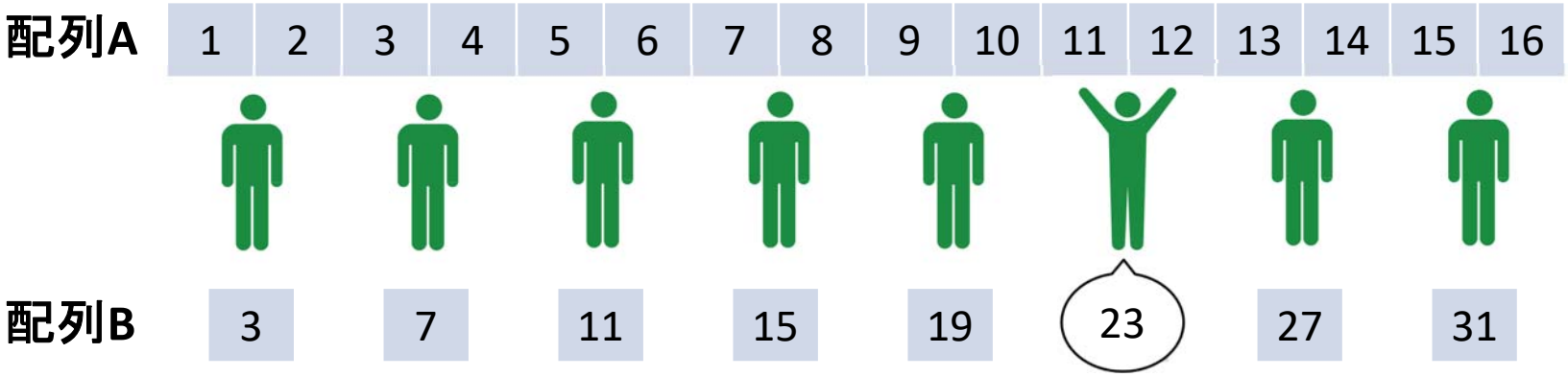


1. 各々自分の担当領域で足し算（結果を別の場所に保存）

例えば以下を8スレッドで並列化

どうやって並列化するか？

```
sum = 0;
for ( i = 0; i < 16; i++)
    sum = sum + A[i];
```

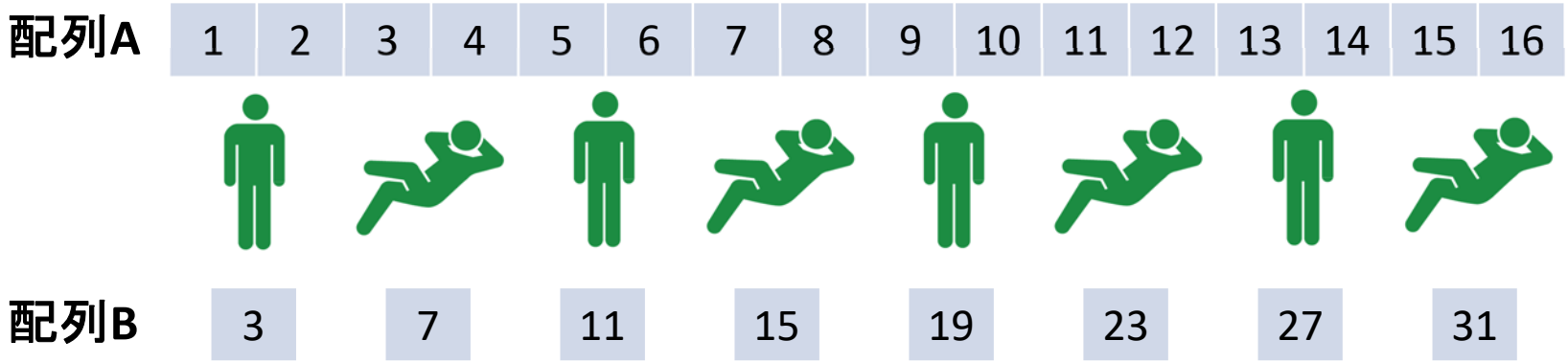


1. 各々自分の担当領域で足し算（結果を別の場所に保存）
2. 遅れているスレッドを待つ！（これを**同期(thread synchronization)**という）

例えば以下を8スレッドで並列化

```
sum = 0;
for ( i = 0; i < 16; i++)
    sum = sum + A[i];
```

どうやって並列化するか？

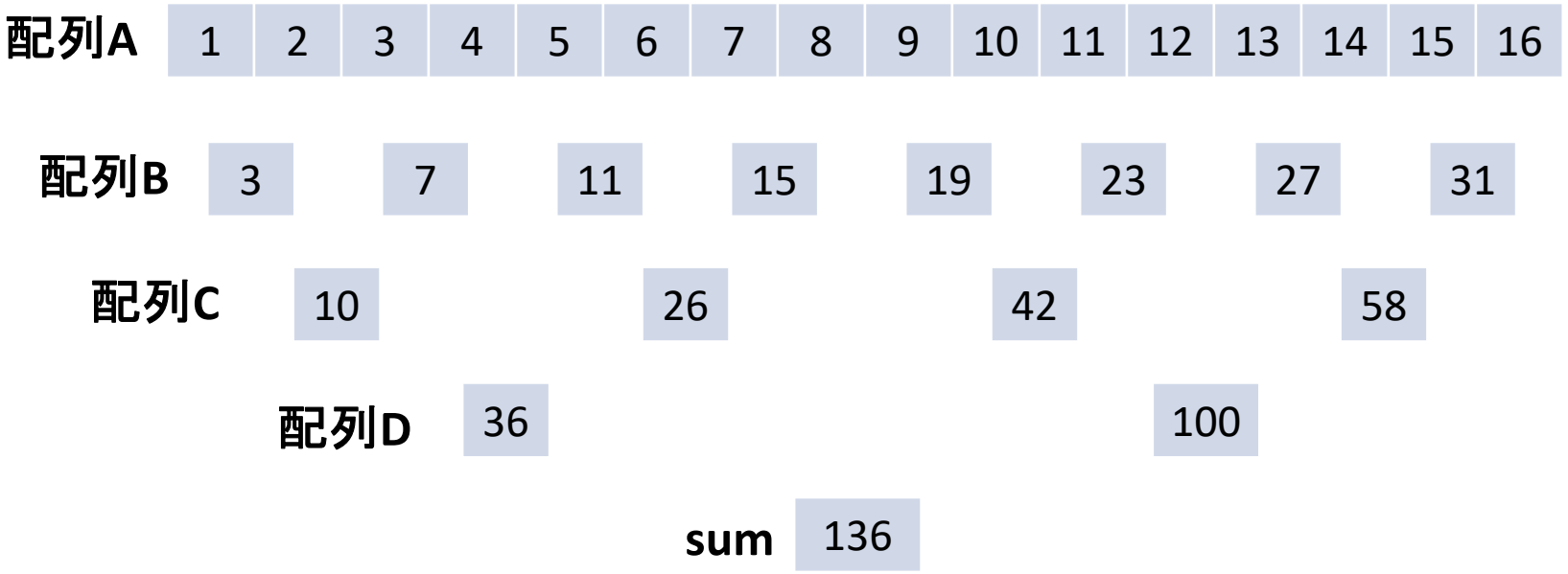


1. 各々自分の担当領域で足し算（結果を別の場所に保存）
2. 遅れているスレッドを待つ！（これを**同期(thread synchronization)**という）
3. 一部のスレッドを寝かせて、起きてるスレッドで(1)から繰り返し

どうやって並列化するか？

例えば以下を8スレッドで並列化

```
sum = 0;
for ( i = 0; i < 16; i++)
    sum = sum + A[i];
```



- これは一般的にリダクションと呼ばれる演算パターン
- 一番働くスレッドが4回の足し算。逐次の場合と比較して4倍の高速化
- メモリなどを介してスレッドの間でデータのやり取りをすることをスレッド間通信という！

スレッドの同期・通信が入ると途端に難しくなる！

GPUにおける並列化

簡単なやつ

```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

```
for ( i = 0; i < 3; i++)
    A[i] = B[i] + C[i];
```

リダクション

```
for ( i = 0; i < 3; i++)
    A[0] = A[0] + 1;
```

```
for ( i = 0; i < 3; i++)
    b = b + A[i];
```

難しいやつ

```
for ( i = 0; i < 3; i++)
    A[i+1] = A[i] + 1;
```

```
for ( i = 0; i < 3; i++)
    b = A[i] / b;
```

OpenACCの守備範囲

※OpenACCでも、GPUで正しく動くコードを書くことはできる。しかし遅いので意味がない

CUDAの守備範囲

- この講習会では、簡単なやつ・リダクションまでを扱います

GPU入門

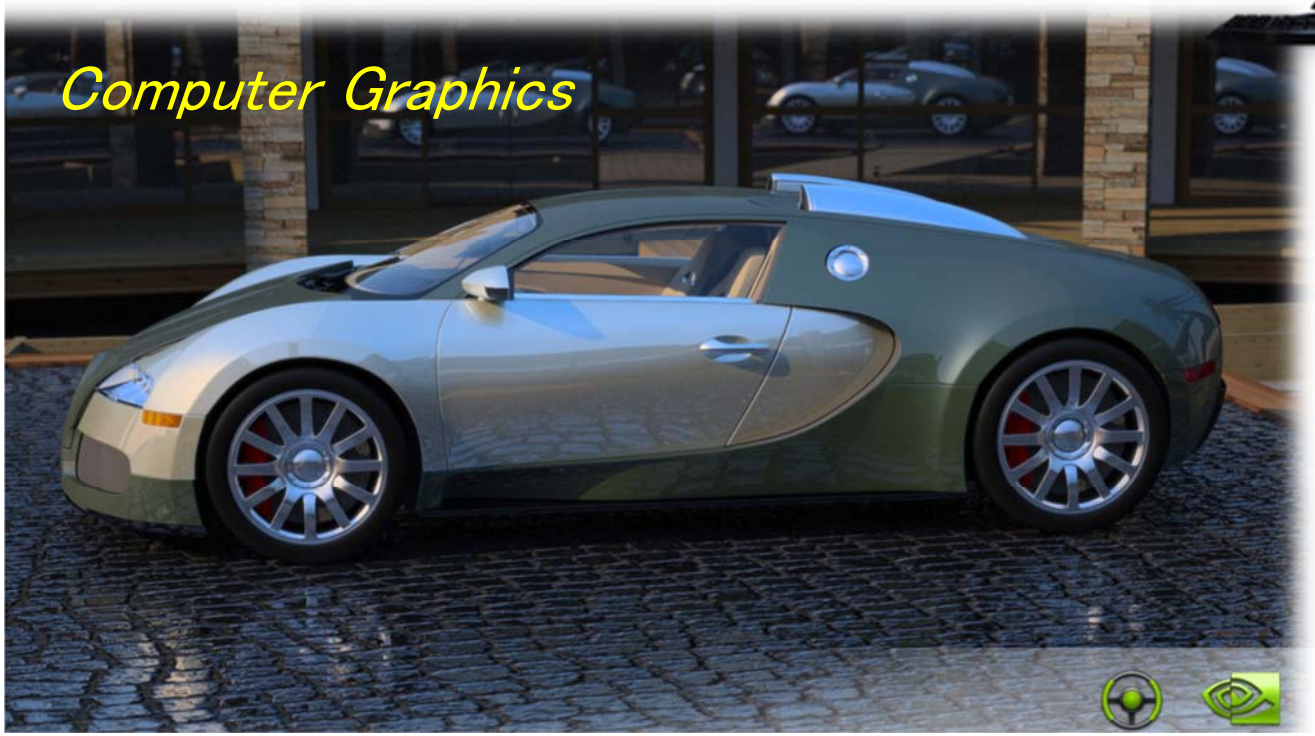
「GPUプログラミング入門」講習会の目的

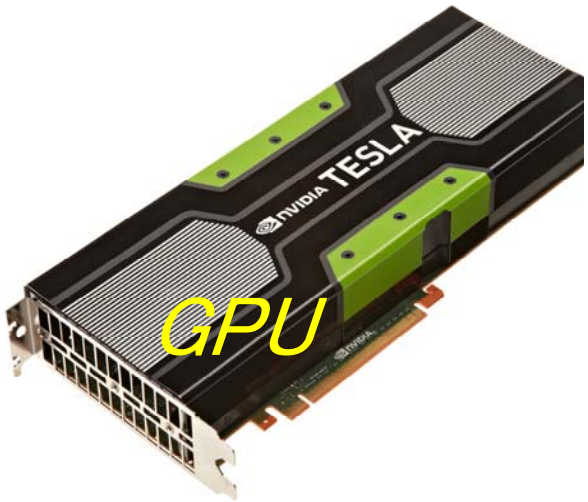
GPUのアーキテクチャ

GPUの使い方

What's GPU ?

- ✓ Graphics Processing Unit
- ✓ もともと PC の3D描画専用の装置
- ✓ パソコンの部品として量産されてる。
  = 非常に安価





http://www.nvidia.co.jp

GPUコンピューティング

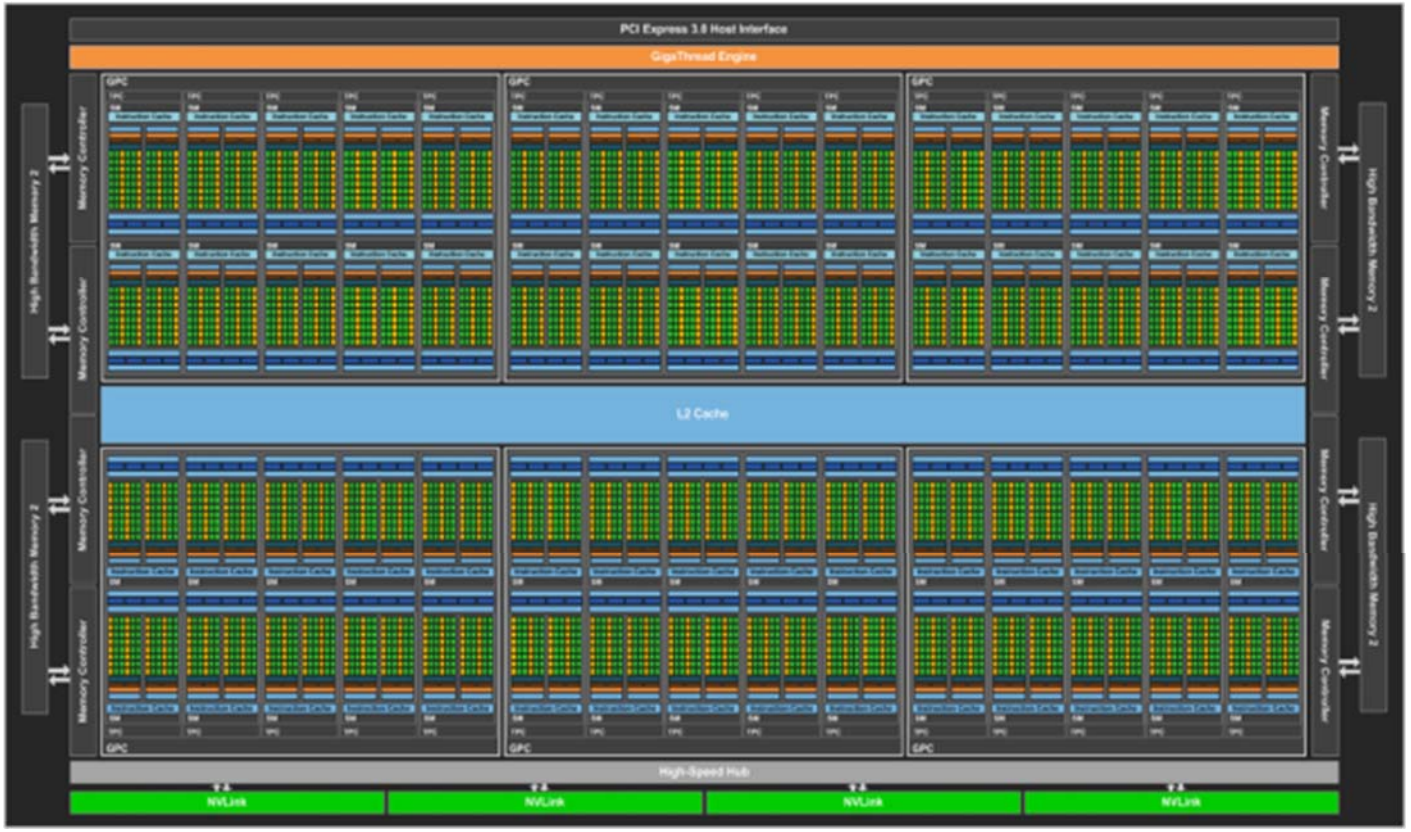
- GPUはグラフィックスやゲームの画像計算のために進化を続けている。
- CPUがコア数が2-12個程度に対し、GPUは1000以上のコアがある。
- GPUを一般のアプリケーションの高速化に利用することを「GPUコンピューティング」「GPGPU (General Purpose computation on GPU)」などという。
- 2007年にNVIDIA社のCUDA言語がリリースされて大きく発展
- ここ数年、ディープラーニング（深層学習）、機械学習、AI（人工知能）などでも注目を浴びている。

GPUコンピューティングにおいてとりあえず抑えておくべきGPUの特徴

- 今日覚えて帰ること
  - 超並列計算が必須！
    - 物理コア数が1000以上、論理コア数（スレッド）は数十万以上！
    - 並列性（プログラムのスレッド分割可能数）が小さいと速くなりません！
  - CPU と GPUの間での同期・通信が必須！
    - GPU は CPU の指示なしでは動けない！
    - さらに、CPU と GPU が並列に動く！
      - CPU と GPU の間では、データ並列ではなくタスク並列になる
- さらなる高速化のために(資料後ろの方参照)
  - 階層的スレッド管理と同期・通信
  - Warp 単位の実行
  - コアレスドアクセス

NVIDIA Tesla P100

- 56 SMs
- 3584 CUDA Cores
- 16 GB HBM2



P100 whitepaper より

NVIDIA Tesla P100 の SM

GPU TECHNOLOGY CONFERENCE

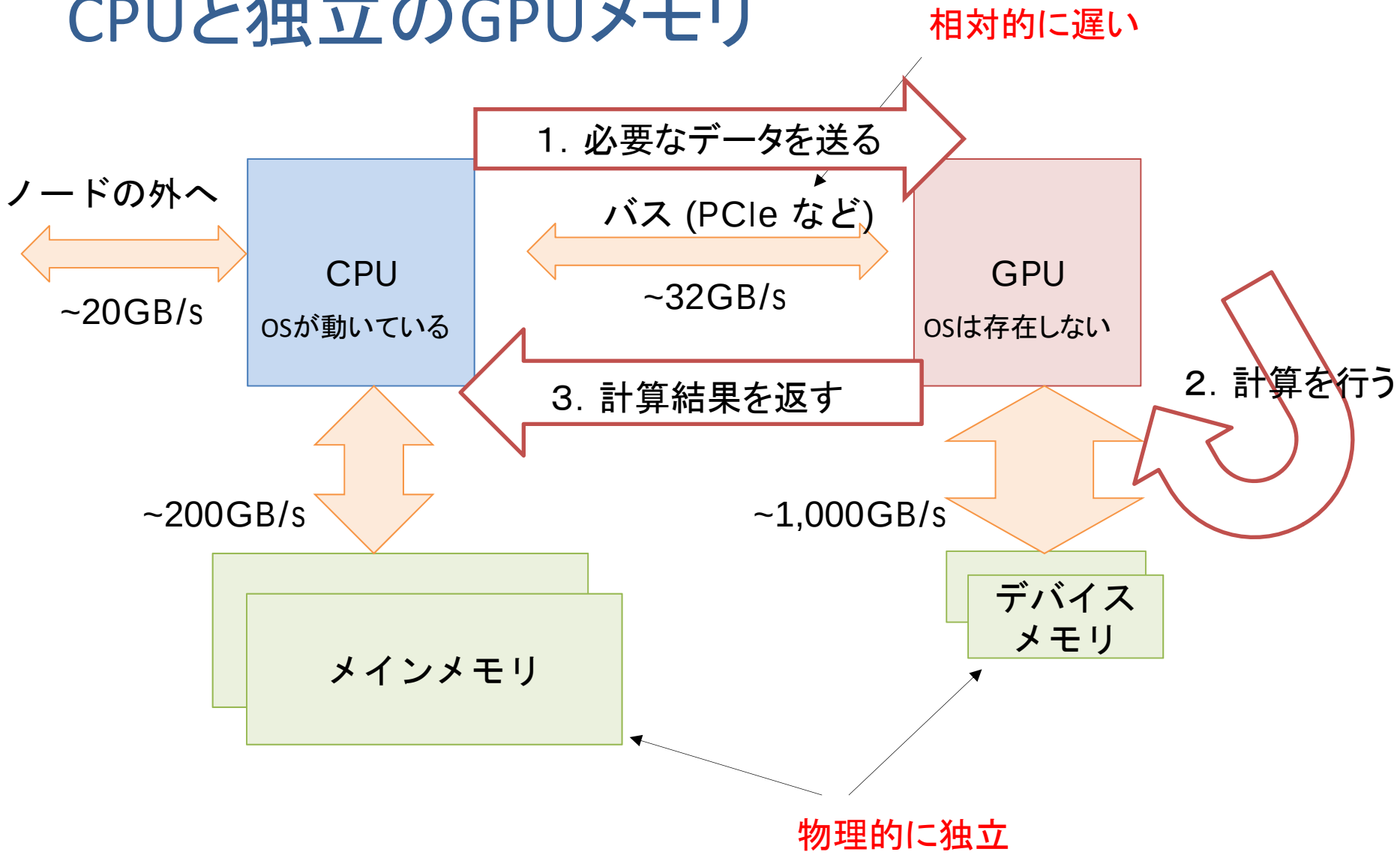
GP100 SM

	GP100
CUDA Cores	64
Register File	256 KB
Shared Memory	64 KB
Active Threads	2048
Active Blocks	32

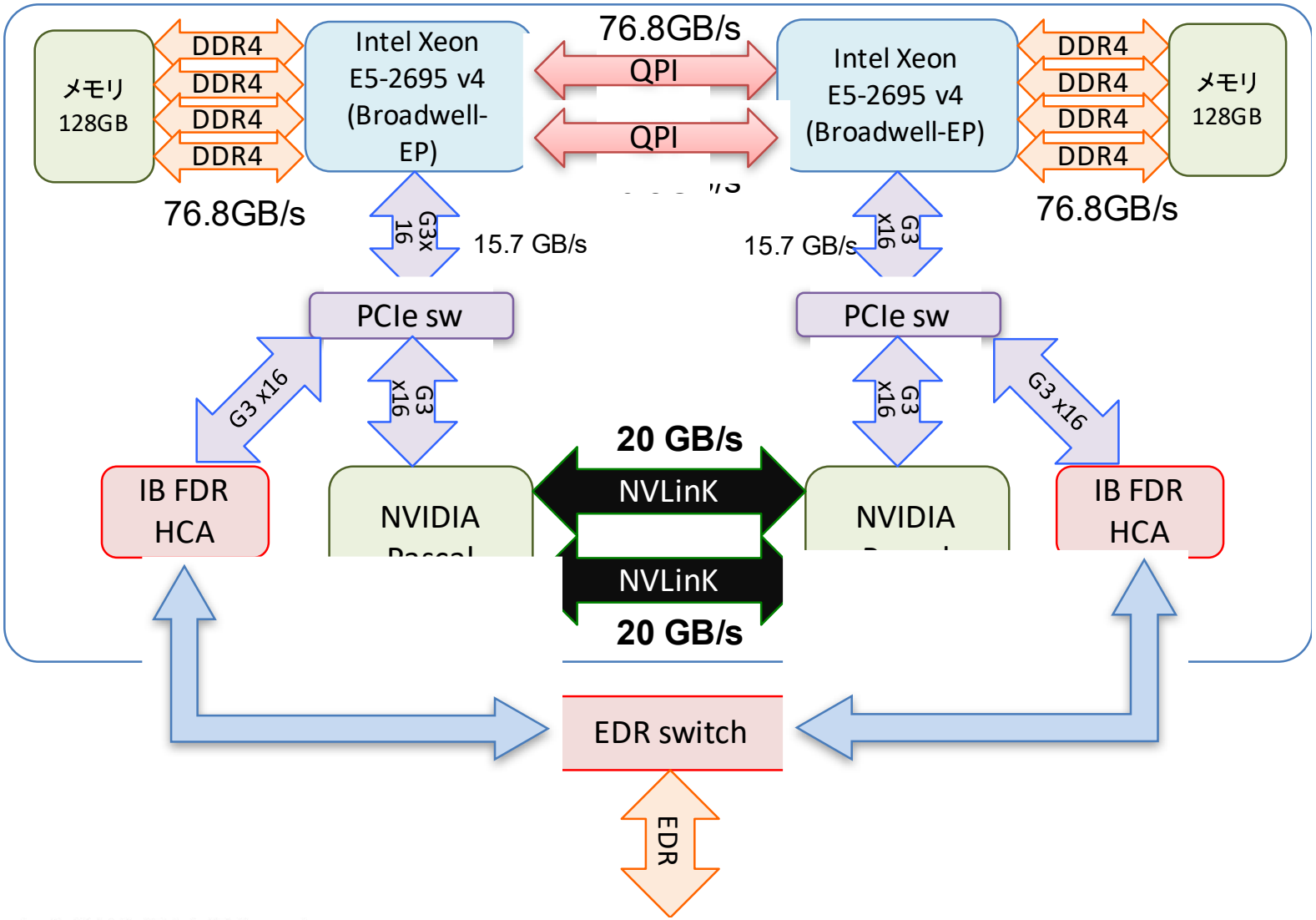
P100 whitepaper より

- 64 コアのグループが 56 個 = 3,584 コア
- 1 個のコアは、複数のスレッドを扱える
- 如何にすべてのコアを効率よく使うかが重要！

CPUと独立のGPUメモリ



Reedbush-Hノードのブロック図



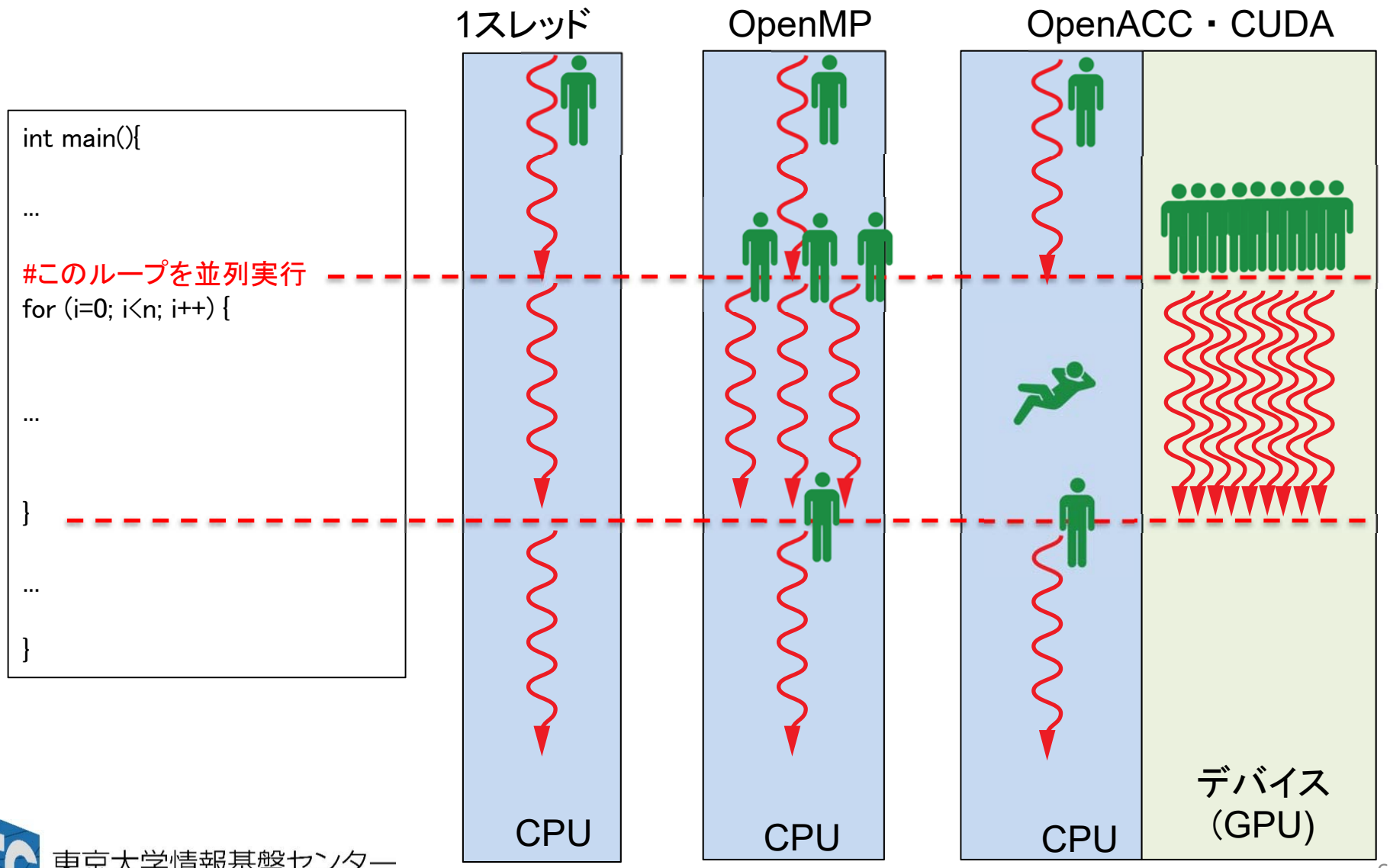
GPUプログラミング入門

1. まずは CUDA を学ぼう！
2. OpenACC も学ぼう！

GPUコンピューティングの方法

- ライブラリの利用（CUFFT, CUBLAS など）
  - ✓ GPU用ライブラリを呼ぶだけで、すぐに利用できる。
  - ✓ ライブラリ以外の部分は高速化されない。
- 指示文ベース（OpenACC）
  - ✓ 指示文（ディレクティブ）を挿入するだけである程度高速化。
  - ✓ 既存のソースコードを活用できる。
- プログラミング言語（CUDA、OpenCLなど）
  - ✓ GPUの性能を最大限に活用。
  - ✓ プログラミングにはGPGPU用言語を使用する必要あり。

逐次、OpenMP(CPU向け並列言語)、GPUコンピューティングの実行イメージ



CUDAとは

- NVIDIA GPU 向けのプログラミング言語
  - AMD の GPU、CPU 内蔵型の GPU などでは使えない
  - OpenCL は上記でも使えるが、CUDA より難しい
- C/C++/Fortran をベースにした言語拡張
  - フリーのコンパイラがあります
    - C/C++ : NVCC compiler
    - Fortran : PGI compiler　※PGI は現在NVIDIAの子会社

詳しくは公式ページへ：
CUDA C/C++: https://docs.nvidia.com/cuda/
CUDA Fortran: https://www.pgroup.com/resources/cudafortran.htm

はじめてのCUDAコード

cuda_hello/00_hello_cuda/main.cu

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#ひとまずこのループを並列化したい
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```

※ まだ並列化してないCPUのコードです

- .cu が CUDA の拡張子
- 普通の C/C++ として記述すれば、CPU 上でそのまま動く
- 左のプログラムをGPU上で実行したい

はじめてのCUDAコード

cuda_hello/01_hello_cuda/main.cu

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

    float *a_gpu, *b_gpu;
    cudaMalloc((void**)&a_gpu, n*sizeof(float));
    cudaMalloc((void**)&b_gpu, n*sizeof(float));
    cudaMemcpy(a_gpu, a, n*sizeof(float), cudaMemcpyHostToDevice);
    dim3 blocks = dim3(1,1,1);
    dim3 threads = dim3(1,1,1);
    hello_cuda<<<blocks,threads>>>(a_gpu,b_gpu,c,n);
    cudaMemcpy(b,b_gpu, n*sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(a_gpu);
    cudaFree(b_gpu);

    ....
```

※CPU側で実行される関数

CPUとGPUの通信を制御するコード

はじめてのCUDAコード

cuda_hello/01_hello_cuda/main.cu

cudaMalloc(void **devpp, size_t count) :
GPUは自分のメモリを持っているので、まずはGPU上にメモリ確保をする必要がある。GPUのメモリ上にmallocするための関数。第一引数はポインタのポインタ。

```
float *a_gpu;
cudaMalloc((void**)&a_gpu, n*sizeof(float));
```

cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind) :
CPU と GPU の間でデータ通信をするための関数。この関数は、内部的にCPUとGPUの同期を行ってからデータコピーをする。同期せずにコピーする関数もあるが、ここでは扱わない。cudaMemcpyKind には、cudaMemcpyHostToDevice (CPUからGPU) と cudaMemcpyDeviceToHost (GPUからCPU) がある。

```
cudaMemcpy(a_gpu, a, n*sizeof(float), cudaMemcpyHostToDevice);
```

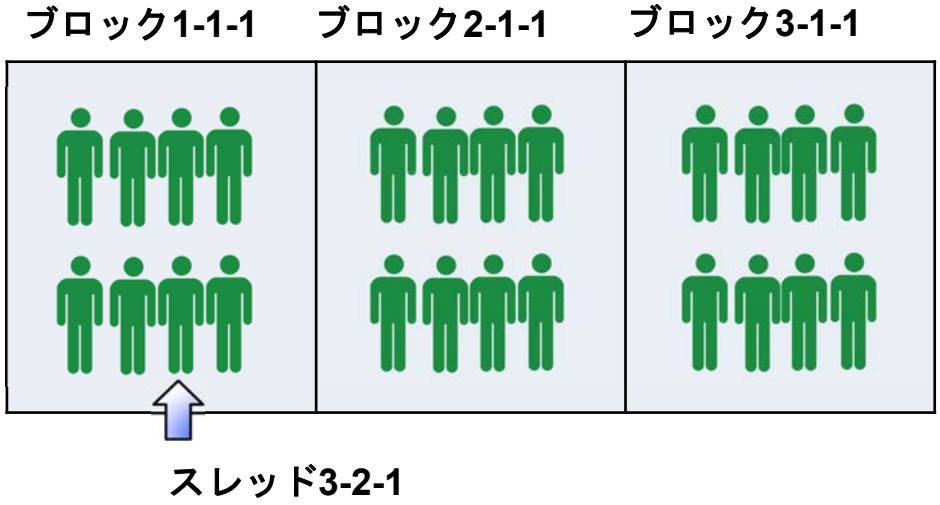
はじめてのCUDAコード

cuda_hello/01_hello_cuda/main.c
u

func<<< dim3(gx,gy,gz), dim3(bx,by,bz) >>> (…)

CUDA では、<<< >>> の中で関数毎のスレッド数を定義する。
CUDA では、スレッドに背番号をつけて、それぞれの制御を行う。
背番号は、2つの3次元変数(dim3)からなる。

```
dim3 grid = dim3(2,1,1);
dim3 block = dim3(4,3,1);
foo<<< dim3(3,1,1), dim3(4,2,1) >>>
```



ブロック X-Y-Z のスレッド A-B-C さんは〇〇をやってください。
とすれば、全スレッドにバラバラの指示を出すことも可能！
（ただしやるべきではない）

はじめてのCUDAコード

cuda_hello/01_hello_cuda/main.cu

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

    float *a_gpu, *b_gpu;
    cudaMalloc((void**)&a_gpu, n*sizeof(float));
    cudaMalloc((void**)&b_gpu, n*sizeof(float));
    cudaMemcpy(a_gpu, a, n*sizeof(float), cudaMemcpyHostToDevice);
    dim3 blocks = dim3(1,1,1);
    dim3 threads = dim3(1,1,1);
    hello_cuda<<<blocks,threads>>>(a_gpu,b_gpu,c,n);
    cudaMemcpy(b,b_gpu, n*sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(a_gpu);
    cudaFree(b_gpu);

    ....
```

※CPU側で実行される関数

メモリ確保

1 x 1 スレッドでの CUDA kernel 起動

メモリ解放

CPU -> GPU の通信

GPU -> CPU の通信

はじめてのCUDAコード

cuda_hello/01_hello_cuda/main.c

```
__global__ void hello_cuda(float* a, float* b, float c, int n)
{
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }
}
```

※GPU側で実行される関数

並列化はされていない

__global__ :
CPU側から呼ぶことのできる、GPU関数につける修飾子。戻り値を使うことはできない(void のみ)。通信は基本的にcudaMemcpyを介して行う（関数の引数は例外）。

__device__ :
GPUの関数 (__global__ または __device__ 付きの関数) から呼ぶことのできるGPU関数につける修飾子。戻り値も使える。再帰もできるがオススメしない。

はじめてのCUDAコード

cuda_hello/02_hello_cuda/main.c

```
u
__global__ void hello_cuda(float* a, float* b, float c, int n)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if(i > n) return;
   b[i] = a[i] + c;
}
```

※GPU側で実行される関数

※CPUの側の呼び出し部分

```
....
   dim3 blocks = dim3((n-1)/32+1,1,1);
   dim3 threads = dim3(32,1,1);
   hello_cuda<<<blocks,threads>>>(a_gpu,b_gpu,c,n);
....
```

ループの **i** 番目をひとりのスレッドが担当する。
if(i > n) return; の行は、ループの終了条件の代わり。

呼び出し部分と、GPU実行部分のスレッド割り当てが一致して初めて正しく動く！

背番号の決まり方

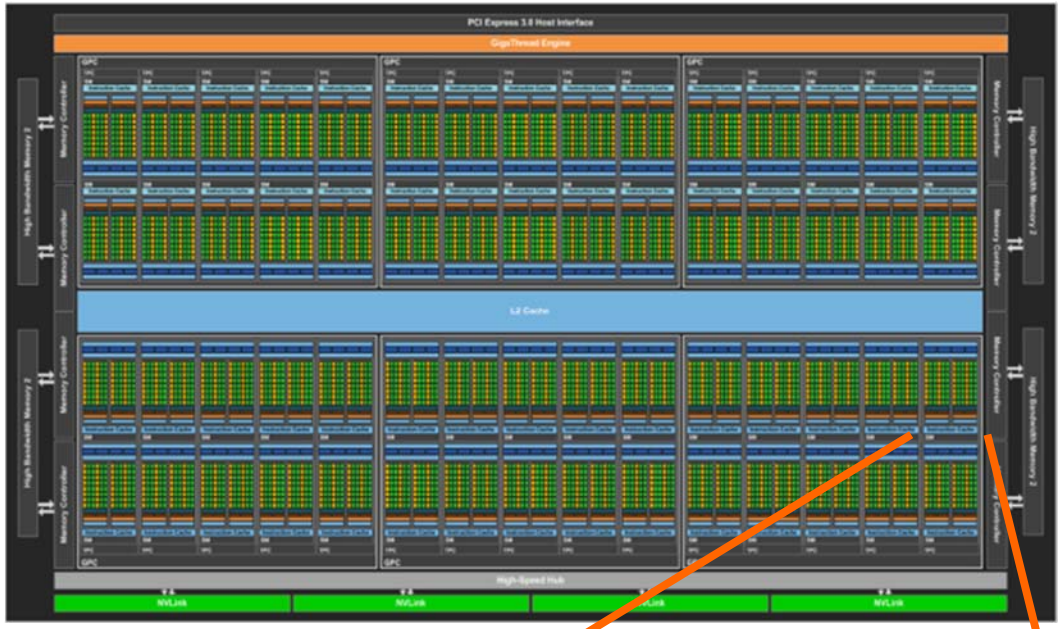
0 <= blockIdx.x < blocks.x

blockDim.x == threads.x

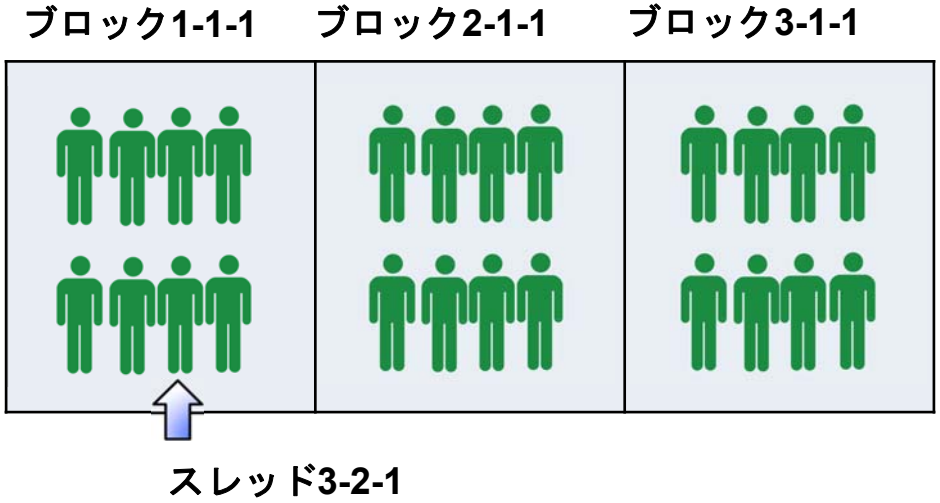
0 <= threadIdx.x < threads.x

なぜ foo <<< dim3(1,1,1), dim3(n,1,1) >>> ではダメか？

- スレッドブロックは、SMのどこかにいつか割り当てられ、実行される。
  - ブロック1つでは、56あるSMの一つしか使えない！
- 一つのSMsが一度に担当できるスレッド数は、1024が限界！
  - ブロックの中のスレッド数を1024以下にしなくてはならない
  - ブロック数は、2^{32}-1 まで大丈夫



SMs



P100 whitepaper より

はじめての CUDA まとめ

1. cudaMalloc を使って、GPU上に領域を確保する
2. cudaMemcpy を使って、CPUからGPUにデータを通信する
3. <<< >>> と dim3 で、スレッドの背番号を定義する
4. __global__ 修飾子を使って、GPUプログラムを記述する
5. threadIdx.x, blockIdx.x などを使って、各スレッドの処理を記述する
6. cudaMemcpy を使って、GPUからCPUにデータを通信する
7. cudaFree を使って、GPU上の領域を解放して終了

OpenACC

- OpenACCとは… **アクセラレータ(GPUなど)向けのOpenMPのようなもの**
  - 既存のプログラムのホットスポットに指示文を挿入し、計算の重たい部分をアクセラレータにオフロード
  - 対応言語：C/C++, Fortran
- 指示文ベース
  - 指示文：コンパイラへのヒント
  - 記述が簡便, メンテナンスなどをしやすい
  - コードの可搬性(portability)が高い
    - 対応していない環境では無視される

C/C++

```
#pragma acc kernels
for(i = 0;i < N;i++) {
  ….
}
```

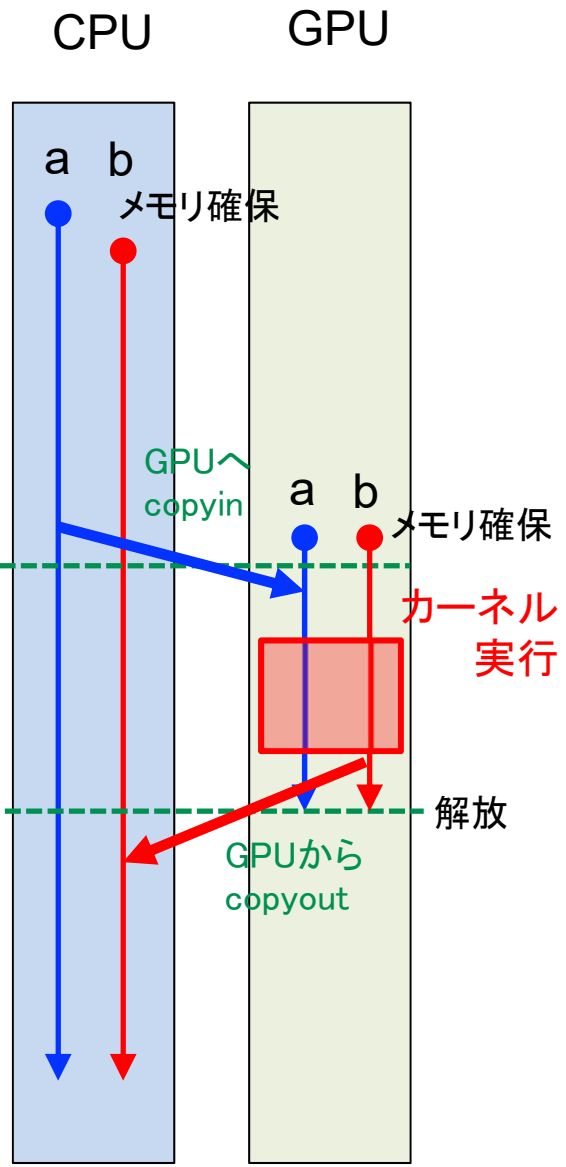
Fortran

```
!$acc kernels
do i = 1, N
  ….
end do
!$acc end kernels
```

はじめてのOpenACCコード

openacc_hello/01_hello_ac
c

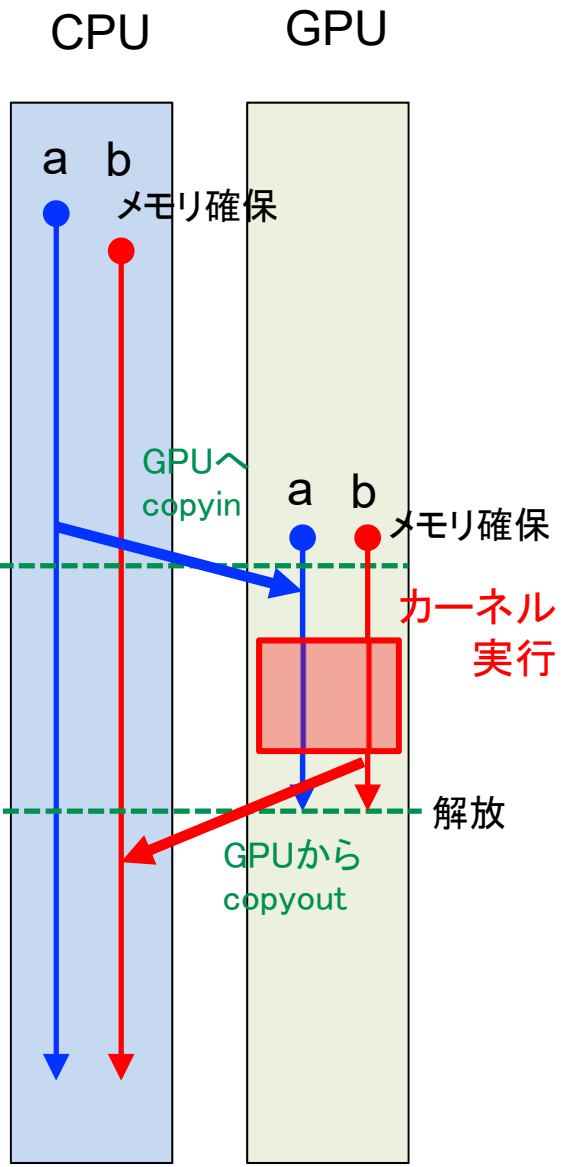
```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];    }
    fprintf(stdout, "%f¥n", sum/n);

    free(a); free(b);
    return 0;
}
```



はじめてのOpenACCコード

openacc_hello/01_hello_ac
c

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];    }
```



コード上同じ a, b であっても、原則として
ホストコードはホストメモリで確保された a, b 、GPUで実行される並列領域（カーネル）はデバイスメモリで確保された a, b
を参照しにいく。

OpenACCの主な指示文

- 並列領域指定指示文
  - ✓ kernels, parallel
- データ管理・移動指示文
  - ✓ data, enter data, exit data, update
- 並列処理の指定
  - ✓ loop
- その他
  - ✓ host_data, atomic, routine, declare

赤字：この講習会で扱うもの

OpenACC によるアクセラレータでの実行

- kernels 指示文
  - ✓ 指定された領域がアクセラレータで実行されるカーネルへ
  - ✓ 一般には、それぞれのループが別々のカーネルへコンパイル

```
int main() {

#pragma acc kernels
{
    for (int i=0; i<n; i++) {
        A;
    }

    for (int i=0; i<n; i++) {
        B;
    }
}

}
```

kernel 1

kernel 2

  - ✓ 同様な指示文として、領域内が一つのカーネルとして生成される parallel 指示文もある

CPUコードのOpenACC化

openacc_hello/01_hello_ac
c

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }




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

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```

- ループのOpenACC 化
  - ✓ 並列化したいループにkernels, loop 指示文を追加

CPUコードのOpenACC化

openacc_hello/01_hello_ac
c

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }


#pragma acc kernels
#pragma acc loop independent
for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```

- ループのOpenACC 化
  - ✓ 並列化したいループにkernels, loop 指示文を追加

CPUコードのOpenACC化

openacc_hello/01_hello_ac
c

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }



#pragma acc kernels
#pragma acc loop independent
for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```

- ループのOpenACC 化
  - ✓ 並列化したいループにkernels, loop 指示文を追加

カーネルとしてコンパイルされ、
GPU上で実行される
配列の1要素が1スレッドで処理されるイメージ

OpenACCのデフォルトの変数の扱い

- スカラ変数
  - ✓ firstprivate または private
  - ✓ ホストからデバイスへコピーが渡され初期化。ホストに戻せない。
- 配列
  - ✓ shared
  - ✓ デバイスメモリに動的に確保され、スレッド間で共有。
  - ✓ デバイスからホストへコピーすることが可能。

- kernels 構文に差し掛かると、
  - ✓ OpenACCコンパイラは実行に必要なデータを自動で転送する。
  - ✓ 配列はデバイスメモリに確保され、shared になる。
  - ✓ 構文に差し掛かるたびに転送を行う。data 指示文で制御できる。

データ管理・移動

- data 指示文
  - ✓ デバイス(GPU)メモリの確保と解放、ホスト(CPU)とデバイス(GPU)間のデータ転送を制御

    kernels指示文では、データ転送は自動的に行われる。data指示文でこれを制御することで、不要な転送を避け、性能向上できる
  - ✓ CUDA で言うところの cudaMalloc, cudaMemcpy に相当

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }
```

openacc_hello/01_hello_ac
c

変数 c はスカラ変数のため、自動的に デバイスへコピーされ、プライベート変数となる。

data 指示文の指示節

- copy
  - ✓ allocate, memcpy(H->D), memcpy(D->H), deallocate
- copyin
  - ✓ allocate, memcpy(H->D), deallocate
  - ✓ 解放前にホストへデータをコピーしない
- copyout
  - ✓ allocate, memcpy(D->H), deallocate
  - ✓ 確保後にホストからデータをコピーしない
- create
  - ✓ allocate, deallocate
  - ✓ コピーしない
- present
  - ✓ 何もしない。既にデバイス上で確保済みであることを伝える。

データの移動範囲の指定

- ホストとデバイス間でコピーする範囲を指定
  - 部分配列の転送が可能
  - Fortran とC言語で指定方法が異なるので注意

- 二次元配列A転送する例
  - Fortran: 下限と上限を指定

```
!$acc data copy(A(lower1:upper1, lower2:upper2) )
...
!$acc end data
```

  - C言語: 始点とサイズを指定

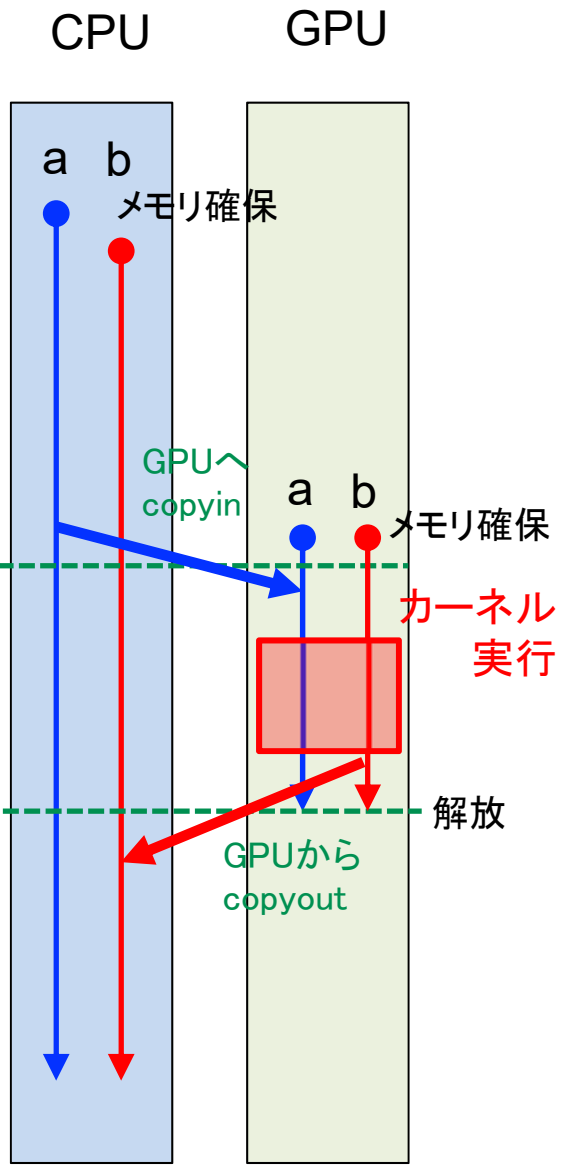
```
#pragma acc data copy(A[begin1:length1][begin2:length2])
...
```

並列処理の指定

openacc_hello/01_hello_ac
c

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
```

loop指示文



並列処理の指定：loop 指示文

- loop 指示文
  - ✓ ループマッピングのパラメータの調整（CUDAのスレッドブロック数を指定）

    gang, worker, vector を用いて指定する。大まかに以下のように考えると良い。
    - gang: CUDA の thread block 数の指定
    - vector: CUDA の block 内の threads 数の指定
  - ✓ ループがデータ独立であることを指定 (independent clause)

    データ独立でないと並列化できない。C言語ではコンパイラがしばしばデータ独立であることを判断できないので、その場合はこれを指定。
  - ✓ リダクション処理 (reduction clause)
  - ✓ 逐次処理 (seq clause)

データの独立性

- independent 指示節 により指定
  - ✓ ループがデータ独立であることを明示する
  - ✓ コンパイラが並列化できないと判断したときに使用する

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

並列化可能（データ独立）なので、independent を指定
（コンパイラは並列化可能とは判断してくれなかった）

- データ独立でない（並列化可能でない）例

```
// これは正しくない

#pragma acc kernels
#pragma acc loop independent
    for (int i=1; i<n; i++) {
        d[i] = d[i-1];
    }
```

参考：OpenACC 化とCUDA化の比較

```
// OpenACC

void calc(int n, const float *a,
const float *b, float c, float *d)
{
#pragma acc kernels present(a, b, d)
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        d[i] = a[i] + c*b[i];
    }
}

int main()
{
    ...
#pragma acc data copyin(a[0:n], b[0:n]) copyout(d[0:n])
    {
        calc(n, a, b, c, d);
    }
    ...
}
```

kernel

- ✓ kernels 指示文でGPUでの実行領域を指定。
- ✓ loop 指示文で並列処理の最適化。
- ✓ data 指示文でデータ転送を制御。kernels 指示文でデータ転送を自動的に行うこともできる。

```
// CUDA

__global__
void calc_kernel(int n, const float *a, const float *b, float c, float *d)
{
    const int i   = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        d[i] = a[i] + c*b[i];
    }
}

void calc(int n, const float *a, const float *b, float c, float *d)
{
    dim3 threads(128);
    dim3 blocks((n + threads.x - 1) / threads.x);

    calc_kernel<<<blocks, threads>>>(n, a, b, c, d);
    cudaThreadSynchronize();
}

int main()
{
    ...

    float *a_d, *b_d, *d_d;
    cudaMalloc(&a_d, n*sizeof(float));
    cudaMalloc(&b_d, n*sizeof(float));
    cudaMalloc(&d_d, n*sizeof(float));

    cudaMemcpy(a_d, a, n*sizeof(float), cudaMemcpyDefault);
    cudaMemcpy(b_d, b, n*sizeof(float), cudaMemcpyDefault);
    cudaMemcpy(d_d, d, n*sizeof(float), cudaMemcpyDefault);

    calc(n, a_d, b_d, c, d_d);

    cudaMemcpy(d, d_d, n*sizeof(float), cudaMemcpyDefault);
    ...
}
```

OpenACCコードのコンパイル

- PGIコンパイラによるコンパイル
  - ✓ ReedbushではOpenACCはPGIコンパイラで利用できます。

```
$ module load pgi
$ pgcc -O3 -acc -Minfo=accel -ta=tesla,cc60 -c main.c
```

-acc: OpenACCコードであることを指示

-Minfo=accel:

OpenACC指示文からGPUコードが生成できたかどうか等のメッセージを出力する。このメッセージがOpenACC化では大きなヒントになる。

-ta=tesla,cc60:

ターゲット・アーキテクチャの指定。NVIDIA GPU Teslaをターゲットとし、compute capability 6.0 (cc60) のコードを生成する。

- Makefileでコンパイル

講習会のサンプルコードには Makefile がついているので、コンパイルするためには、単純に下記を実行すれば良い。

```
$ module load pgi
$ make
```

簡単なOpenACCコード

- サンプルコード: openacc_basic/
  - ✓ OpenACC指示文 kernels, data, loop を利用したコード
  - ✓ 計算内容は簡単な四則演算

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    for (unsigned int j=0; j<ny; j++) {
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```

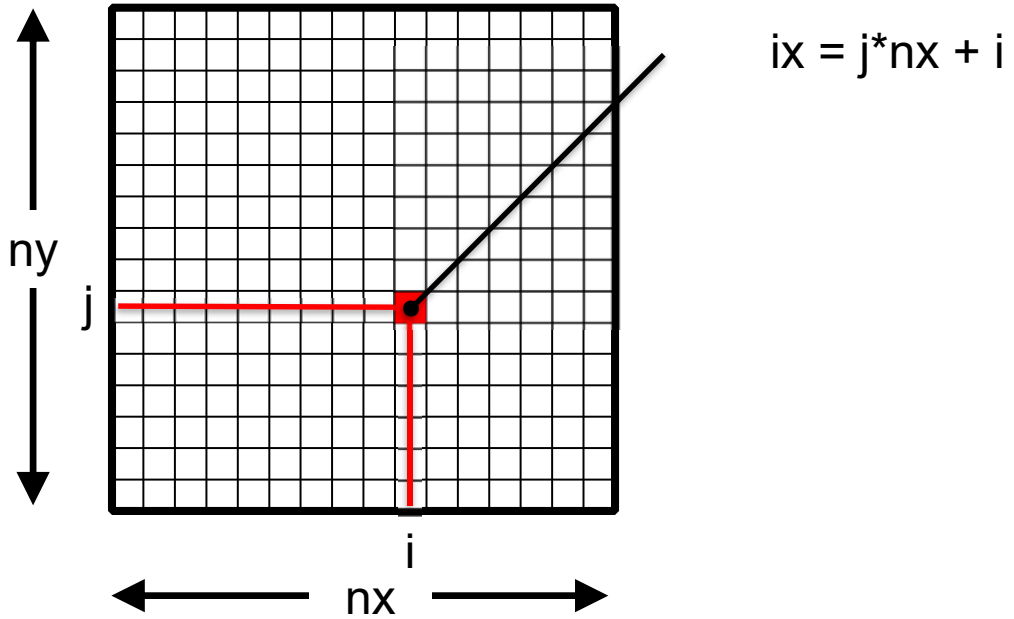
  - ✓ ソースコード

openacc_basic/01_original	CPUコード。
openacc_basic/02_kernels	OpenACCコード。上にkernels/loop指示文を追加。
openacc_basic/03_data	OpenACCコード。上にdata指示文を明示的に追加。
openacc_basic/04_present	OpenACCコード。上でpresent指示節を使用。
openacc_basic/05_reduction	OpenACCコード。上にreduction指示節を使用。

配列のインデックス計算

- サンプルコード: openacc_basic/
  - ✓ OpenACC指示文 kernels, data, loop を利用したコード
  - ✓ 計算内容は簡単な四則演算

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    for (unsigned int j=0; j<ny; j++) {
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```



簡単なOpenACC: CPUコード

- CPUコードのコンパイルと実行
  - ✓ 配列の平均値と実行時間が出力されています。

```
$ cd openacc_basic/01_original
$ make
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =    19.886 [sec]
```

? の数字はジョブごとに変わります。

答えは常に3000.0

openacc_basic/01_original

- 計算内容
  - ✓ 配列 a、b、cをそれぞれ 1.0, 2.0, 0.0 で初期化
  - ✓ calc関数内で c += a *b を nt(=1000)回実行。
  - ✓ この実行時間を測定

簡単なOpenACC: kernels 指示文(1)

- 02_kernelsコード: calc関数
  - ✓ CPUコードにkernels 指示文の追加

openacc_basic/02_kernels

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    const unsigned int n = nx * ny;
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])
    for (unsigned int j=0; j<ny; j++) {
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```

allocate, H -> D

D->H, deallocate

  - ✓ kernels 指示文では data 指示文が使える
  - ✓ 上の場合は、copy を指定
    - ✓ カーネル前後でGPUとCPU間のメモリ転送が行われる。

簡単なOpenACC: kernels 指示文(2)

- 02_kernelsコード:初期化
  - ✓ CPUコードにkernels 指示文の追加

openacc_basic/02_kernels

```
int main(int argc, char *argv[])
{
...
#pragma acc kernels copyout(b[0:n], c[0:n])
    {
      for (unsigned int i=0; i<n; i++) {
        b[i] = b0;
      }
      for (unsigned int i=0; i<n; i++) {
        c[i] = 0.0;
      }
    }

...
}
```

allocate

b0 はスカラ変数のため自動的に各スレッドへコピーが渡される。

D->H, deallocate

  - ✓ kernels 指示文では data 指示文が使える
  - ✓ 上の場合は、copyout を指定
    - ✓ GPU上で値を初期化するため、CPUからGPUへのコピーは不要。

簡単なOpenACC: kernels 指示文(3)

- コンパイル
  - ✓ データの独立性がコンパイラにはわからず、並列化されない。

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
calc:
     13, Generating copy(a[:n],c[:n],b[:n])
     14, Complex loop carried dependence of a-> prevents parallelization
         Loop carried dependence due to exposed use of c[:n] prevents parallelization
         Complex loop carried dependence of c->,b-> prevents parallelization
         Accelerator scalar kernel generated
         Accelerator kernel generated
         Generating Tesla code
         14, #pragma acc loop seq
         15, #pragma acc loop seq
     15, Complex loop carried dependence of a->,c->,b-> prevents parallelization
         Loop carried dependence due to exposed use of c[:i1+n] prevents parallelization
main:
     43, Generating copyout(c[:n],b[:n])
     45, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         45, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     48, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         48, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60   main.o   -o run
```

簡単なOpenACC: loop 指示文(1)

- 03_loopコード
  - ✓ 02_kernelsコードにloop independent の追加

openacc_basic/03_loop

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    const unsigned int n = nx * ny;
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])
#pragma acc loop independent
    for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```

```
// main 関数内
#pragma acc kernels copyout(b[0:n], c[0:n])
    {
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            b[i] = b0;
        }
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            c[i] = 0.0;
        }
    }
```

簡単なOpenACC: loop 指示文(2)

- コンパイル

openacc_basic/03_loop

✓ ループが並列化され、カーネルが生成された。

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
calc:
     13, Generating copy(a[:n],c[:n],b[:n])
     15, Loop is parallelizable
     17, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
main:
     45, Generating copyout(c[:n],b[:n])
     48, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         48, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     52, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         52, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

簡単なOpenACC: loop 指示文(3)

- 03_loopコードの実行

openacc_basic/03_loop

✓ 答えは正しいが、実行時間が大変長い。

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =   70.414 [sec]
```

✓ ソースコードをみると、calc関数でカーネル前後にGPUとCPU間のデータ転送が発生する。これが性能低下させている。

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    const unsigned int n = nx * ny;
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])
#pragma acc loop independent
    for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
    }
}
```

allocate, H -> D

D->H, deallocate

簡単なOpenACC: data指示文(1)

- 04_dataコード
  - ✓ 03_loopにdata指示文追加

openacc_basic/04_data

```
// main関数内
#pragma acc data copyin(a[0:n]) create(b[0:n]) copyout(c[0:n])
   {
#pragma acc kernels copyout(b[0:n], c[0:n])
      {
#pragma acc loop independent
         for (unsigned int i=0; i<n; i++) {
            b[i] = b0;
         }
#pragma acc loop independent
         for (unsigned int i=0; i<n; i++) {
            c[i] = 0.0;
         }
      }

      for (unsigned int icnt=0; icnt<nt; icnt++) {
         calc(nx, ny, a, b, c);
      }
   }
```

a: allocate, H -> D
b: allocate
c: allocate

present として振舞う。

a: deallocate
b: deallocate
c: D->H, deallocate

  - ✓ copy/copyin/copyout/create は既にデバイス上確保されているデータに対しては何もしない。present として振舞う。(OpenACC2.5以降)
  - ✓ 配列 a, b, c は利用用途に合わせた指示節を指定。

簡単なOpenACC: data指示文（2）

- 04_dataコードの実行
  - ✓ 答えは正しく、速度が上がった。

openacc_basic/04_data

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =    1.174 [sec]
```

簡単なOpenACC: present指示節

- 05_presentコード
  - ✓ 04_dataコードで present 指示節を使用

openacc_basic/05_present

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
    const unsigned int n = nx * ny;
#pragma acc kernels present(a, b, c)
#pragma acc loop independent
    for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
        for (unsigned int i=0; i<nx; i++) {
            const int ix = i + j*nx;
            c[ix] += a[ix] + b[ix];
        }
```

present へ変更

```
// main 関数内
#pragma acc kernels
   {
#pragma acc loop independent
        for (unsigned int i=0; i<n; i++) {
            b[i] = b0;
        }
```

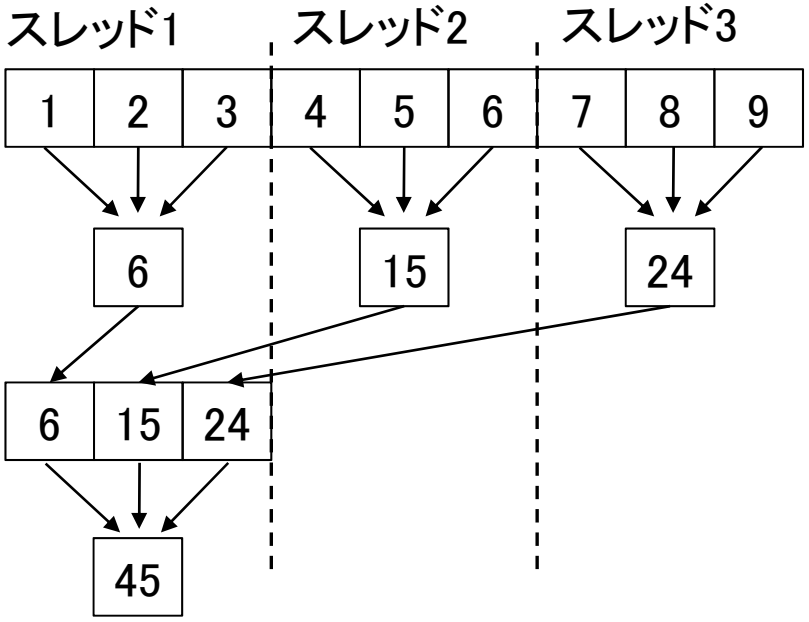
指示節を削除

- ✓ データ転送の振る舞いは変化しないため、性能変化はなし。
- ✓ present ではメモリ確保、データ転送をしないため、配列サイズの指定は不要。
- ✓ コードとしては見通しがよい。

リダクション計算（1）

- リダクション計算
  - ✓ 配列の全要素から一つの値を抽出
  - ✓ 総和、総積、最大値、最小値など
  - ✓ 出力が一つのため、並列化に工夫が必要（CUDAでの実装は煩雑）

```
double sum = 0.0;
for (unsigned int i=0; i<n; i++) {
    sum += array[i];
}
```



1. 各スレッドが担当する領域をリダクション
2. 一時配列に移動
3. 一時配列をリダクション
4. 出力を得る

リダクション計算(2)

- loop 指示文に reduction 指示節を指定
  - ✓ reduction 演算子と変数を組み合わせて指定

```
double sum = 0.0;
#pragma acc kernels
#pragma acc loop reduction(+:sum)
for (unsigned int i=0; i<n; i++) {
    sum += array[i];
}
```

- Reduction 指示節
  - ✓ acc loop reduction(+:sum)
    - ✓ 演算子と対象とする変数(スカラー変数)を指定する。
- 利用できる主な演算子と初期値
  - ✓ 演算子: +, 初期値: 0
  - ✓ 演算子: *, 初期値: 1
  - ✓ 演算子: max, 初期値: least
  - ✓ 演算子: min, 初期値: largest

簡単なOpenACC: reduction指示節（1）

- 06_reductionコード
  - ✓ 05_presentコードで reductionを使用

openacc_basic/06_reduction

```
// main 関数内
        for (unsigned int icnt=0; icnt<nt; icnt++) {
            calc(nx, ny, a, b, c);
        }

#pragma acc kernels
#pragma acc loop reduction(+:sum)
        for (unsigned int i=0; i<n; i++) {
            sum += c[i];
        }
```

  - ✓ data 指示文で c を create に変更。

- 06_reductionコード
  - ✓ リダクションコードが生成された。

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
(省略)
main:
(省略)
67, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
        67, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
            Generating reduction(+:sum)
```

簡単なOpenACC: reduction指示節（2）

openacc_basic/06_reduction

- 06_reductionコードの実行
  - ✓ 答えは正しく、速度が上がった。
  - ✓ 配列 c の転送が削減されたこと、リダクションがGPU上で行われることによる性能向上。

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =    1.089 [sec]
```

OpenACC化のステップのまとめ

- OpenACC化のための3つの指示文の適用
  - ✓ kernels 指示文を用いてGPUで実行する領域を指定
  - ✓ data 指示文を用い、ホスト-デバイス 間の通信を最適化
  - ✓ loop 指示文を用い、並列処理の指定

```
#pragma acc data copyin(a[0:n]) create(b[0:n], c[0:n])
  {

#pragma acc kernels
      {
#pragma acc loop independent
          for (unsigned int i=0; i<n; i++) {
              b[i] = b0;
          }
#pragma acc loop independent
          for (unsigned int i=0; i<n; i++) {
              c[i] = 0.0;
          }
      }

      for (unsigned int icnt=0; icnt<nt; icnt++) {
          calc(nx, ny, a, b, c);
      }

#pragma acc kernels
#pragma acc loop reduction(+:sum)
      for (unsigned int i=0; i<n; i++) {
          sum += c[i];
      }
  }
```

openacc_basic/06_reduction

OPENACC入門実習

実習

- 3次元拡散方程式のOpenACC化
  - ✓ サンプルコード: openacc_diffusion/01_original
- 3次元拡散方程式のCPUコードにOpenACC の kernels, data, loop 指示文を追加し、GPUで高性能で実行しましょう。

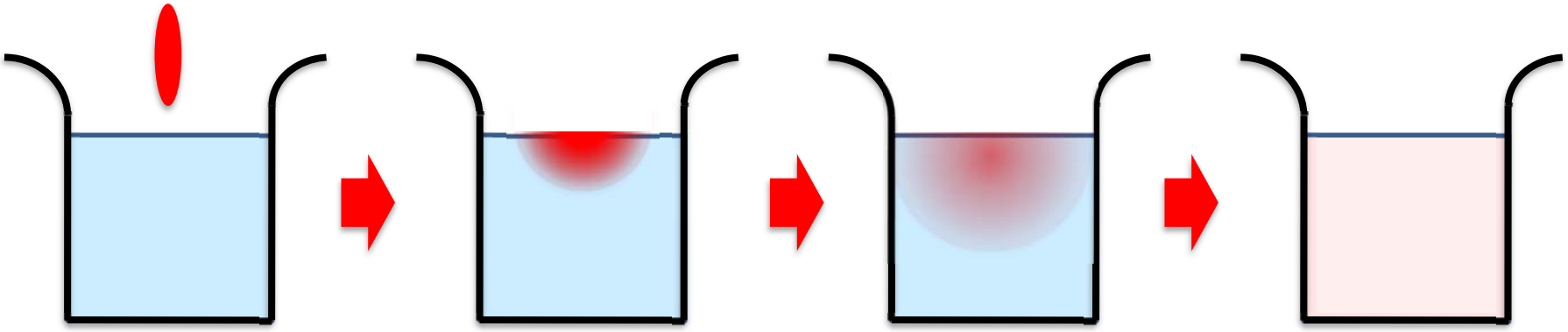
diffusion.c, diffusion3d 関数内

```
for(int k = 0; k < nz; k++) {
  for (int j = 0; j < ny; j++) {
    for (int i = 0; i < nx; i++) {
      const int ix = nx*ny*k + nx*j + i;
      const int ip = i == nx - 1 ? ix : ix + 1;
      const int im = i == 0      ? ix : ix - 1;
      const int jp = j == ny - 1 ? ix : ix + nx;
      const int jm = j == 0      ? ix : ix - nx;
      const int kp = k == nz - 1 ? ix : ix + nx*ny;
      const int km = k == 0      ? ix : ix - nx*ny;

      fn[ix] = cc*f[ix]
             + ce*f[ip] + cw*f[im]
             + cn*f[jp] + cs*f[jm]
             + ct*f[kp] + cb*f[km];
    }
  }
}
```

openacc_diffusion/01_original

拡散現象シミュレーション（1）

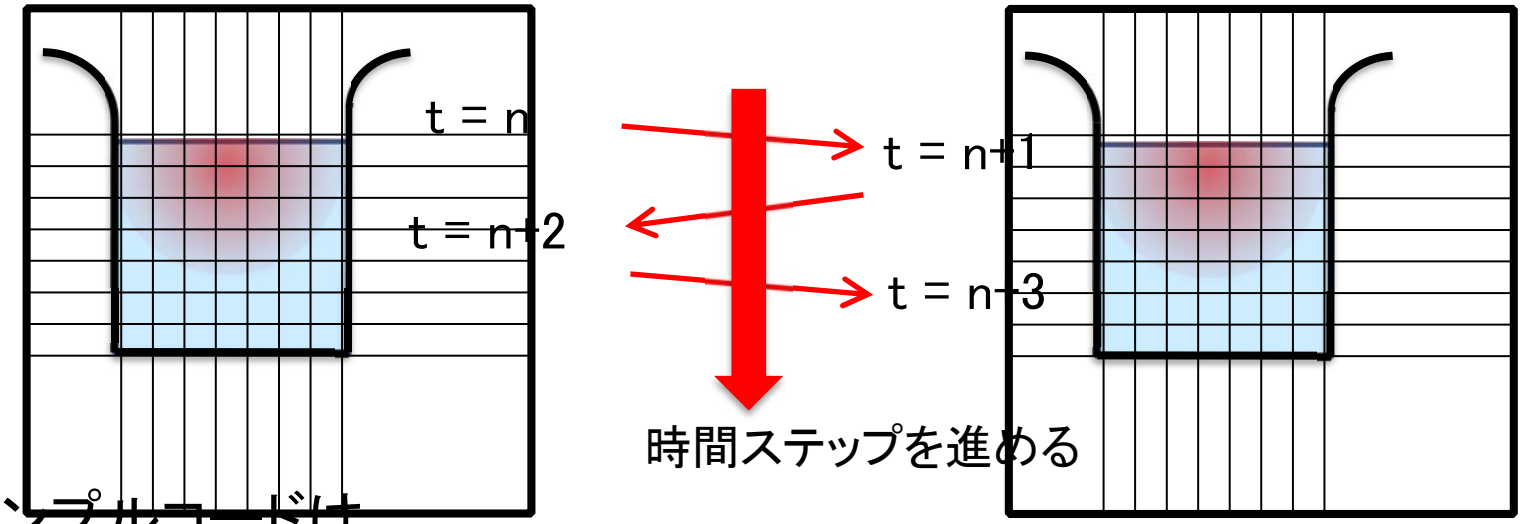
- 拡散現象
  - ✓ コップの中に赤インクを落とすと水中で拡がる
  - ✓ 次第に拡散し赤インクは拡がり、最後は均一な色になる。



- 拡散方程式のシミュレーション
  - ✓ 各点のインク濃度の時間変化を計算する

拡散現象シミュレーション（2）

- データ構造
  - ✓ 計算したい空間を格子に区切り、一般に配列で表す。
  - ✓ 計算は3次元であるが、C言語では1次元配列として確保することが一般的。
  - ✓ 2ステップ分の配列を使い、タイムステップを進める（ダブルバッファ）。



- サンプルコードは、
  - ✓ 計算領域: nx * ny * nz （3次元）
  - ✓ 最大タイムステップ: nt

  となっている。

拡散現象シミュレーション（3）

- 2次元拡散方程式の離散化の一例

$$f_{i,j}^{n+1} = (f_{i-1,j}^{n} + f_{i+1,j}^{n} + f_{i,j-1}^{n} + f_{i,j+1}^{n} + 4f_{i,j}^{n})/8$$

平均後の
自分自身の値

上下左右の値

自分自身の値の4倍

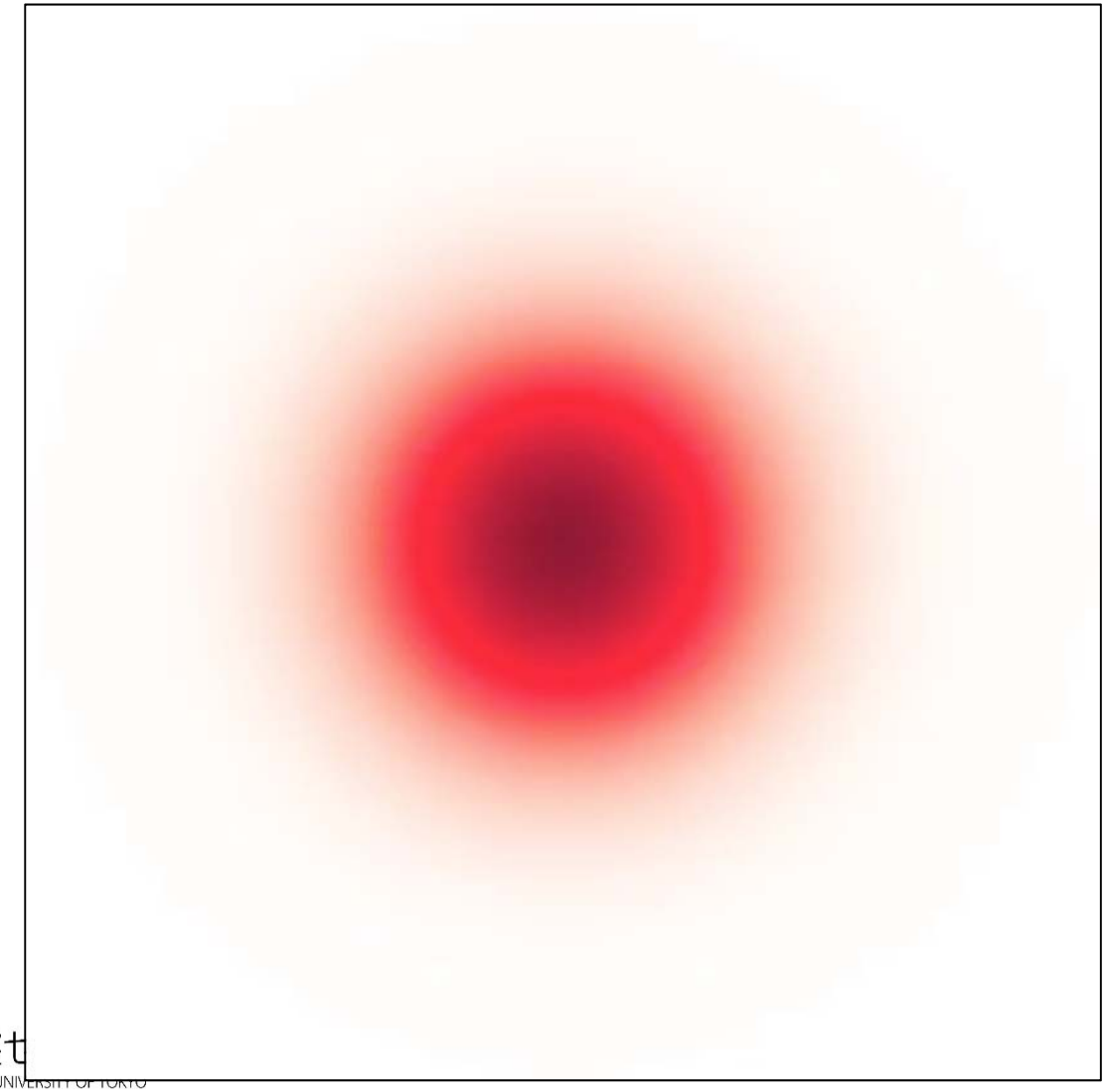
j \ i	1	2	3	4	5
1	0	0	1	0	0
2	0	2	8	2	0
3	1	8	20	8	1
4	0	2	8	2	0
5	0	0	1	0	0

2回目の平均後

繰り返し平均化を行うと、インクが拡散します。

拡散現象シミュレーション（4）

- 2次元拡散方程式の計算例



CPUコード

- CPUコードのコンパイルと実行

```
$ cd openacc_diffusion/01_original
$ make
$ qsub ./run.sh
# cat run.sh.o??????
time(   0) = 0.00000
time( 100) = 0.00610
time( 200) = 0.01221
...
time(1000) = 0.06104
time(1100) = 0.06714
time(1200) = 0.07324
time(1300) = 0.07935
time(1400) = 0.08545
time(1500) = 0.09155
time(1600) = 0.09766
Time =   20.564 [sec]
Performance=    2.17 [GFlops]
Error[128][128][128] = 4.556413e-06
```

実行性能

解析解との誤差

- OpenACCコードでは、どのくらいの実行性能が達成できるでしょうか？

OpenACC化(0): Makefile の修正

- Makefile に OpenACC をコンパイルするよう –acc などを追加しましょう

```
CC   = pgcc
CXX  = pgc++
GCC  = gcc
RM   = rm -f
MAKEDEPEND = makedepend

CFLAGS    = -O3 -acc -Minfo=accel -ta=tesla,cc60
GFLAGS    = -Wall -O3 -std=c99
CXXFLAGS  = $(CFLAGS)
LDFLAGS   =
…
```

OpenACC化(1): kernels

- diffusion3d関数に kernelsを追加しましょう

```
#pragma acc kernels copyin(f[0:nx*ny*nz]) copyout(fn[0:nx*ny*nz])
    for(int k = 0; k < nz; k++) {
        for (int j = 0; j < ny; j++) {
            for (int i = 0; i < nx; i++) {
                const int ix = nx*ny*k + nx*j + i;
                const int ip = i == nx - 1 ? ix : ix + 1;
                const int im = i == 0      ? ix : ix - 1;
                const int jp = j == ny - 1 ? ix : ix + nx;
                const int jm = j == 0      ? ix : ix - nx;
                const int kp = k == nz - 1 ? ix : ix + nx*ny;
                const int km = k == 0      ? ix : ix - nx*ny;

                fn[ix] = cc*f[ix]
                       + ce*f[ip] + cw*f[im]
                       + cn*f[jp] + cs*f[jm]
                       + ct*f[kp] + cb*f[km];
            }
        }
    }

    return (double)(nx*ny*nz)*13.0;
}
```

diffusion.c, diffusion3d 関数内

make して実行してみましょう。

OpenACC化(2): loop

- diffusion3d関数に loopを追加しましょう

```
#pragma acc kernels copyin(f[0:nx*ny*nz]) copyout(fn[0:nx*ny*nz])
#pragma acc loop independent
    for(int k = 0; k < nz; k++) {
#pragma acc loop independent
        for (int j = 0; j < ny; j++) {
#pragma acc loop independent
            for (int i = 0; i < nx; i++) {
                const int ix = nx*ny*k + nx*j + i;
                const int ip = i == nx - 1 ? ix : ix + 1;
                const int im = i == 0      ? ix : ix - 1;
                const int jp = j == ny - 1 ? ix : ix + nx;
                const int jm = j == 0      ? ix : ix - nx;
                const int kp = k == nz - 1 ? ix : ix + nx*ny;
                const int km = k == 0      ? ix : ix - nx*ny;

                fn[ix] = cc*f[ix]
                       + ce*f[ip] + cw*f[im]
                       + cn*f[jp] + cs*f[jm]
                       + ct*f[kp] + cb*f[km];
            }
        }
    }

    return (double)(nx*ny*nz)*13.0;
}
```

diffusion.c, diffusion3d 関数内

高速化よりも、まずは正しい計算を行うコードを保つことが大事です。
末端の関数から修正を進めます。

make してジョブ投入 qsub ./run.shしてみましょう。遅いですが実行できます。

OpenACC化(3): データ転送の最適化(1)

- diffusion3d関数で present とし、main関数で data を追加

diffusion.c, diffusion3d 関数内

```
#pragma acc kernels present(f, fn)
#pragma acc loop independent
    for(int k = 0; k < nz; k++) {
#pragma acc loop independent
        for (int j = 0; j < ny; j++) {
#pragma acc loop independent
            for (int i = 0; i < nx; i++) {
                const int ix = nx*ny*k + nx*j + i;
                const int ip = i == nx - 1 ? ix : ix + 1;
                const int im = i == 0      ? ix : ix - 1;
                const int jp = j == ny - 1 ? ix : ix + nx;
                const int jm = j == 0      ? ix : ix - nx;
                const int kp = k == nz - 1 ? ix : ix + nx*ny;
                const int km = k == 0      ? ix : ix - nx*ny;

                fn[ix] = cc*f[ix]
                       + ce*f[ip] + cw*f[im]
                       + cn*f[jp] + cs*f[jm]
                       + ct*f[kp] + cb*f[km];
            }
        }
    }

    return (double)(nx*ny*nz)*13.0;
}
```

なお、present にしなくても期待通りに動作します。

OpenACC化(4): データ転送の最適化(2)

- diffusion3d関数で present とし、main関数で data を追加

main.c, main 関数内

```
#pragma acc data copy(f[0:n]) create(fn[0:n])
  {
      start_timer();

      for (; icnt<nt && time + 0.5*dt < 0.1; icnt++) {
          if (icnt % 100 == 0)
              fprintf(stdout, "time(%4d) = %7.5f¥n", icnt, time);

          flop += diffusion3d(nx, ny, nz, dx, dy, dz, dt, kappa, f, fn);

          swap(&f, &fn);

          time += dt;
      }

      elapsed_time = get_elapsed_time();
  }
```

copy/create など適切なものを選びます。

make して実行してみましょう。どのくらいの実行性能が出ましたか？

OpenACC化の例は、openacc_diffusion/02_openacc

PGI_ACC_TIME によるOpenACC 実行の確認

- PGIコンパイラを利用する場合、OpenACCプログラムがどのように実行されているか、環境変数PGI_ACC_TIMEを設定すると簡単に確認することができる。
- Linuxなどでは、環境変数PGI_ACC_TIME を1に設定し、プログラムを実行する。

```
$ export PGI_ACC_TIME=1
$ ./run
```

- Reedbush でジョブに環境変数PGI_ACC_TIME を設定する場合は、ジョブスクリプト中に記載する。

```
$ cat run.sh
...
 . /etc/profile.d/modules.sh
module load pgi
export PGI_ACC_TIME=1

./run
```

サンプルコードは、 openacc_diffusion/03_openacc _pgi_acc_time

PGI_ACC_TIME によるOpenACC 実行の確認

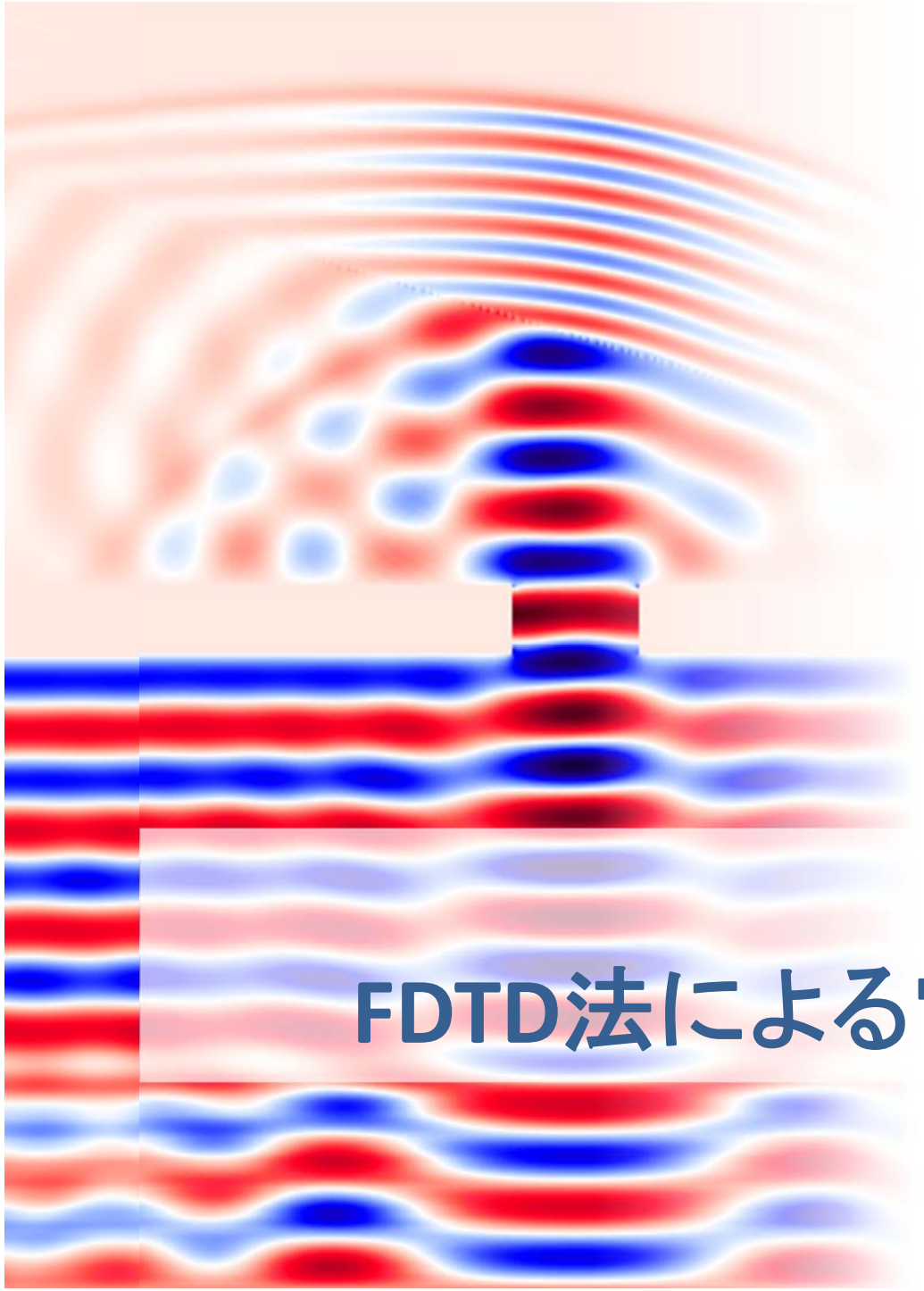
- ジョブ実行が終わると、標準エラー出力にメッセージが出力される。

```
$ cat run.sh.e??????
Accelerator Kernel Timing data
/lustre/pz0115/z30115/lecture/lecture_samples/openacc_diffusion/03_openacc_pgi_acc_time/main.c
  main  NVIDIA  devicenum=0
    time(us): 6,359
    38: data region reached 2 times
        38: data copyin transfers: 1
             device time(us): total=3,327 max=3,327 min=3,327 avg=3,327
        55: data copyout transfers: 1
             device time(us): total=3,032 max=3,032 min=3,032 avg=3,032
/lustre/pz0115/z30115/lecture/lecture_samples/openacc_diffusion/03_openacc_pgi_acc_time/diffusion.
c
  diffusion3d  NVIDIA  devicenum=0
    time(us): 101,731
    19: compute region reached 1638 times
        25: kernel launched 1638 times
            grid: [4x128x32]  block: [32x4]
             device time(us): total=101,731 max=64 min=62 avg=62
            elapsed time(us): total=136,255 max=540 min=81 avg=83
    19: data region reached 3276 times
```

データ移動の回数

起動したスレッド

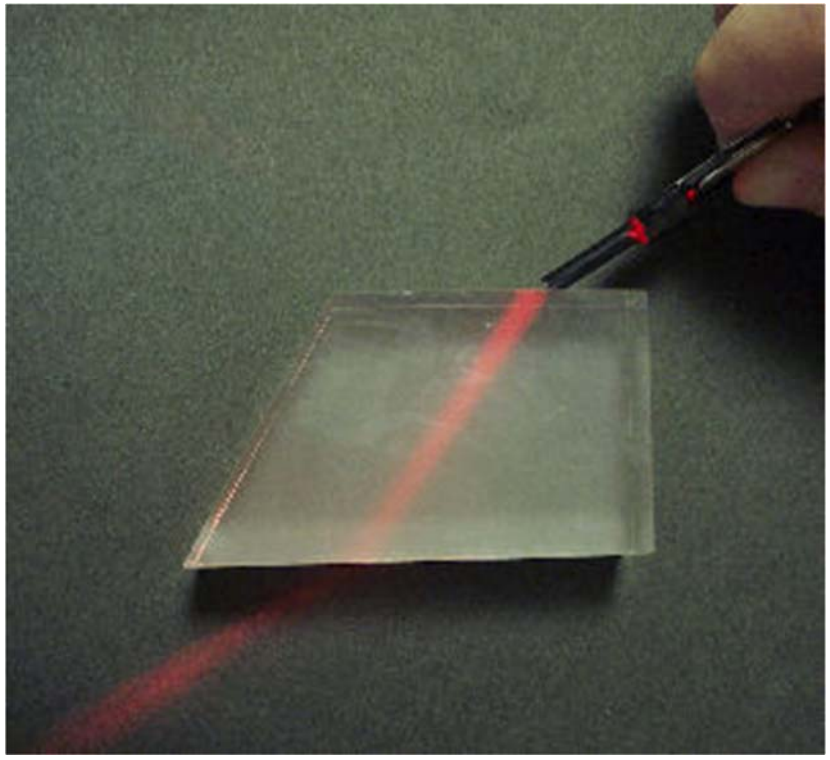
カーネル実行時間

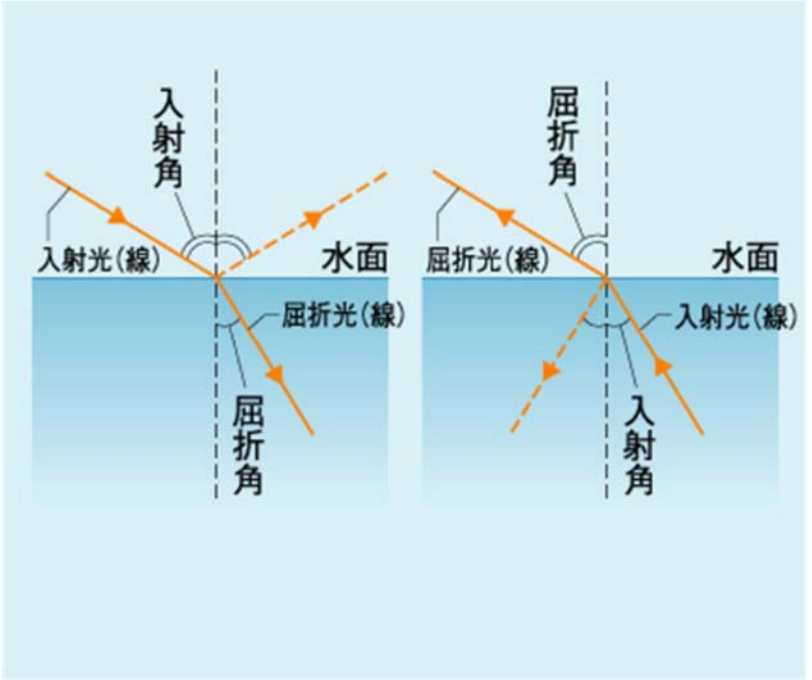


GPUを用いた
FDTD法による電磁波伝搬計算

光の屈折と回折（１）

- 屈折
  - ✓ 光が異なる媒質の境界で進行方向を変えること
  - ✓ 波の進む速度（位相速度）が媒質によってことなるため

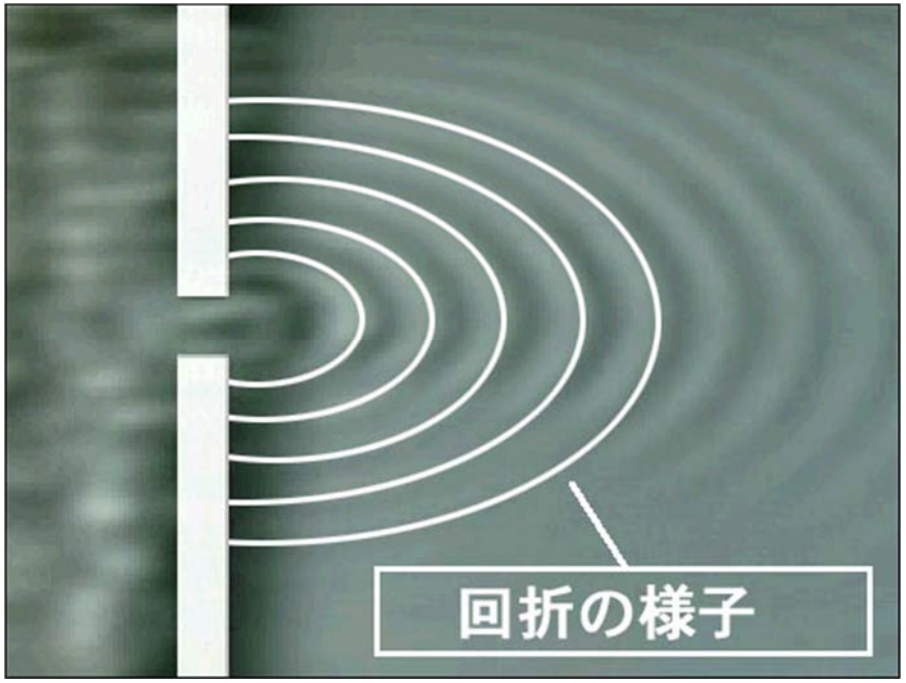


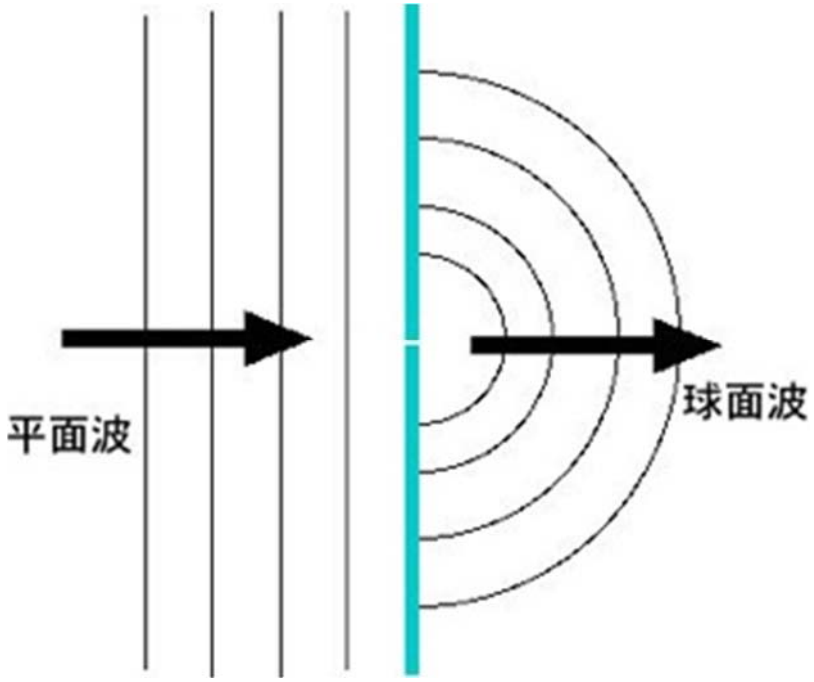


http://www.kagakunavi.jp
http://www2e.biglobe.ne.jp/~shinzo/jikken/pikatto/pikatto.html

光の屈折と回折(2)

- 回折
  - ✓ 光の進路に障害物があるとき、その障害物の陰など、一見すると幾何学的には到達できない領域に回り込んで伝わる現象

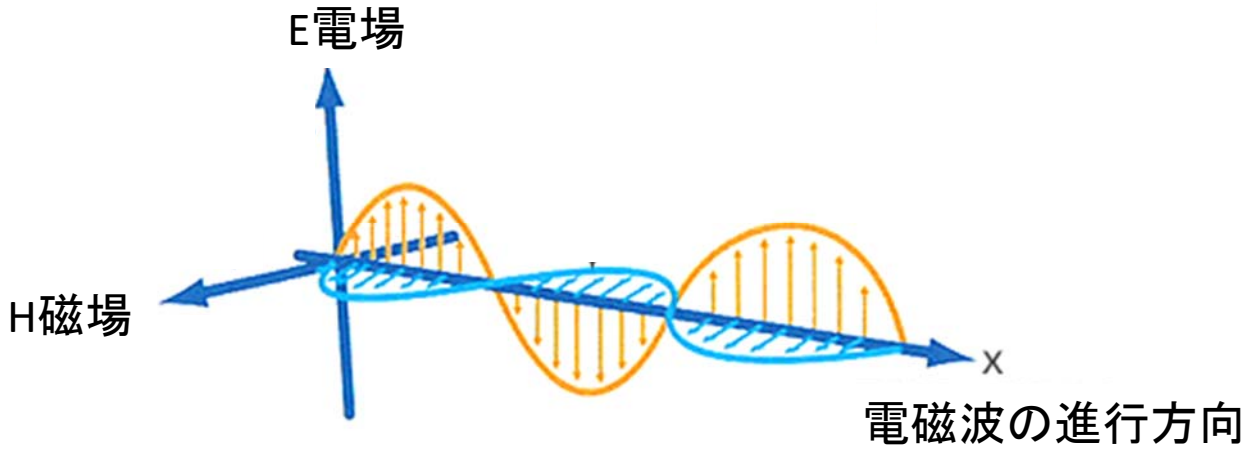




http://rikanet2.jst.go.jp
http://web.me.com/takeda_t/atelier_bonryu/PH_Salon_R.4.html

電磁波の伝播

- 光は電磁波の一種
- 電場と磁場と電磁波の進行方向



✓ 電磁波は、空間の電場と磁場がお互いの電磁誘導によって相互に発生して、空間を横波となって伝播する

http://www.tecnoao-asia.com/about_abc.html

電磁波の方程式

- 真空での電場Eと磁場Hの時間発展

  Maxwell 方程式の一部

$$\frac{\partial \boldsymbol{E}}{\partial t} = \frac{1}{\varepsilon}\nabla \times \boldsymbol{H} \qquad \frac{\partial \boldsymbol{H}}{\partial t} = -\frac{1}{\mu}\nabla \times \boldsymbol{E}$$

（ε ：誘電率）　　（μ ：透磁率）

この方程を、2次元FDTD法（Finite-difference time-domain 法）*を用いて解いて行きます。

* K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," IEEE Trans. on Antennas and Propagat., vol. 14, pp. 302-307, May 1966.

FDTD法（1）

- EとHの時間発展

$$\frac{\boldsymbol{E}^n - \boldsymbol{E}^{n-1}}{\Delta t} = \frac{1}{\varepsilon}\nabla \times \boldsymbol{H}^{n-\frac{1}{2}}$$

$$\frac{\boldsymbol{H}^{n+\frac{1}{2}} - \boldsymbol{H}^{n-\frac{1}{2}}}{\Delta t} = -\frac{1}{\mu}\nabla \times \boldsymbol{E}^{n}$$

変形して、

$$\boldsymbol{E}^n = \boldsymbol{E}^{n-1} + \frac{\Delta t}{\varepsilon}\nabla \times \boldsymbol{H}^{n-\frac{1}{2}}$$

$$\boldsymbol{H}^{n+\frac{1}{2}} = \boldsymbol{H}^{n-\frac{1}{2}} - \frac{\Delta t}{\mu}\nabla \times \boldsymbol{E}^{n}$$

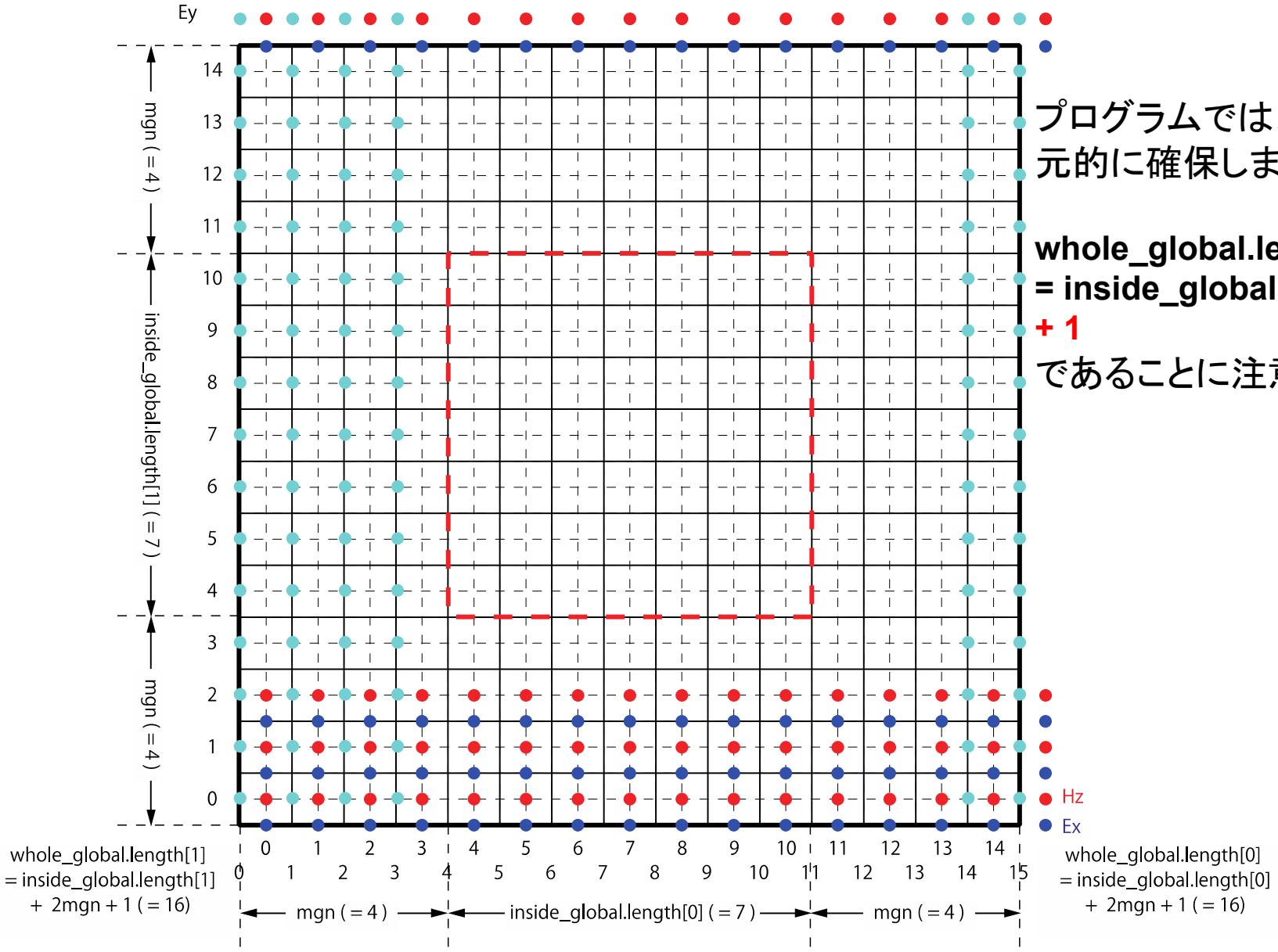
FDTD法（2）

- 例えば、

$$E_x^n(i+\frac{1}{2},j) = E_x^{n-1}(i+\frac{1}{2},j) + \frac{\Delta t}{\varepsilon(i+\frac{1}{2},j)}\left(\frac{H_z^{n-\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2}) - H_z^{n-\frac{1}{2}}(i+\frac{1}{2},j-\frac{1}{2})}{\Delta y}\right)$$

$$H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2}) = H_z^{n-\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2}) - \frac{\Delta t}{\mu(i+\frac{1}{2},j+\frac{1}{2})}\left(\frac{E_y^n(i+1,j+\frac{1}{2}) - E_y^n(i,j+\frac{1}{2})}{\Delta x} - \frac{E_x^n(i+\frac{1}{2},j+1) - E_x^n(i+\frac{1}{2},j)}{\Delta y}\right)$$

2次元FDTD法の変数配置



プログラムでは、配列は1次元的に確保します。

whole_global.length[0]
= inside_global[0] + 2*mgn
+ 1
であることに注意

ソースコード（1）

- サンプルコード: openacc_fdtd/
  - ✓ OpenACCを利用したFDTD法（電磁波解析）

openacc_fdtd/01_original	MPI並列化されたCPUコード。
openacc_fdtd/02_openacc1	calc_ex_ey, pml_boundary_ex, pml_boundary_ey, がOpenACC。
openacc_fdtd/03_openacc2	時間更新ループ全体が OpenACC。
openacc_fdtd/04_openacc3	初期化を含め OpenACC。
openacc_fdtd/05_openacc4	データ移動の最適化。

※本プログラムはMPI化されていますが、本講習会では扱いません

ソースコード（2）

- それぞれのファイルの内容

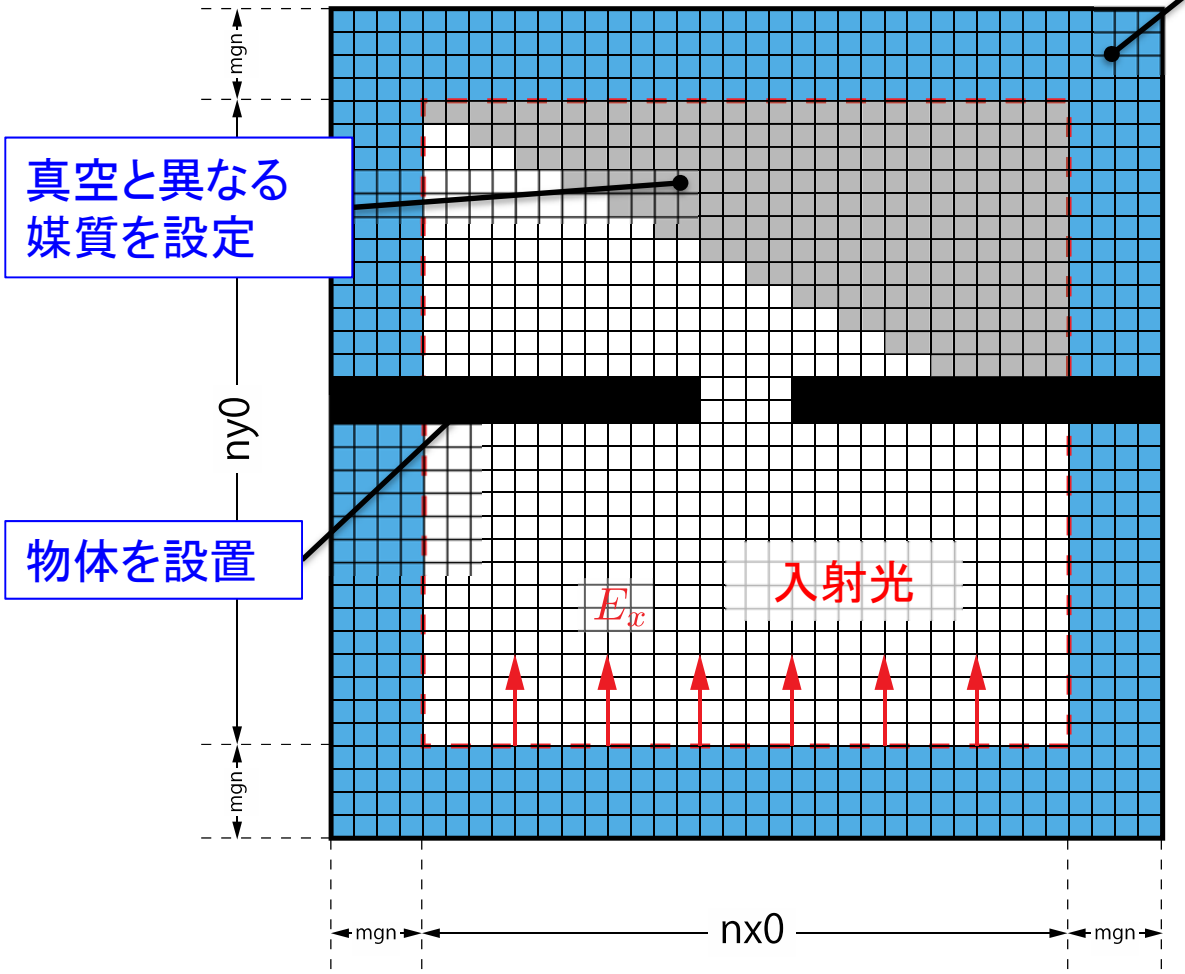
main.c	プログラムのメインコード
fdtd2d.{c, h}	2次元 FDTD の 計算コード
fdtd2d_sources.{c, h}	入射光設定のための関数
setup.c	計算条件の設定と変数の初期化
config.{c, h}	物理定数の定義
output.{cc, h}	計算結果出力のための関数
bitmap*	BMPファイル作成のための関数

本講習では、“main.c”、“fdtd2d.c”、” fdtd2d_sources.c”、”setup.c” のソースコードを追記・修正していきます。

計算条件

- 2次元波動伝搬
  - ✓ 成分: Ex、Ey、Hz
  - ✓ y 方向下側から平面波を入射

電磁波の境界での非物理的な反射を防ぐための吸収境界条件（PML）



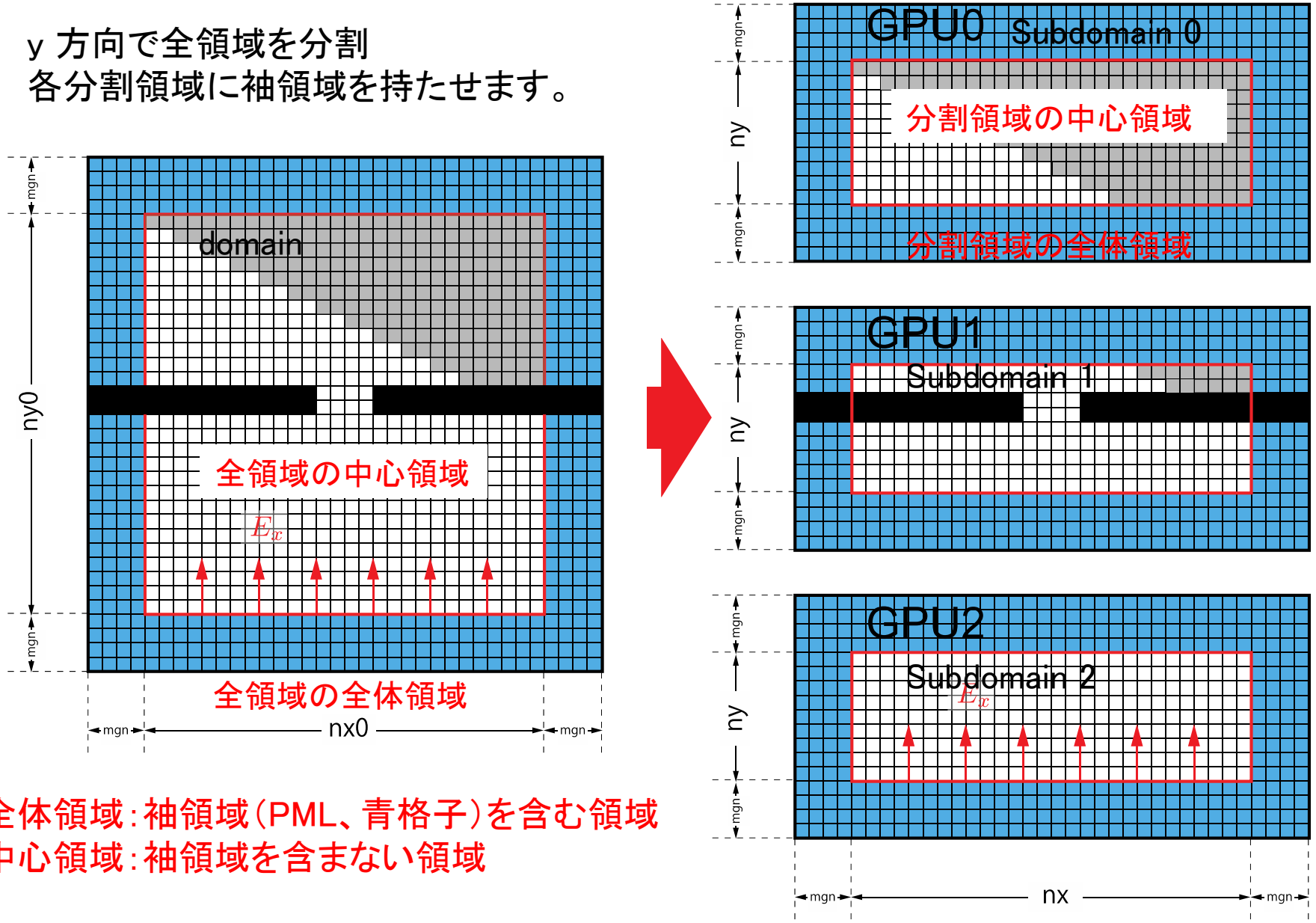
dx = lx/nx
dy = ly/ny

デフォルト設定:
nx = 512
ny = 512
mgn = 8
lnx = 529
lny = 529

プログラム中では下記の変数が使われているので注意
inside_global.length[0] = nx0
inside_global.length[1] = ny0
whole_global.length[0]
= nx0 + 2*mgn + 1
whole_global.length[1]
= ny0 + 2*mgn + 1

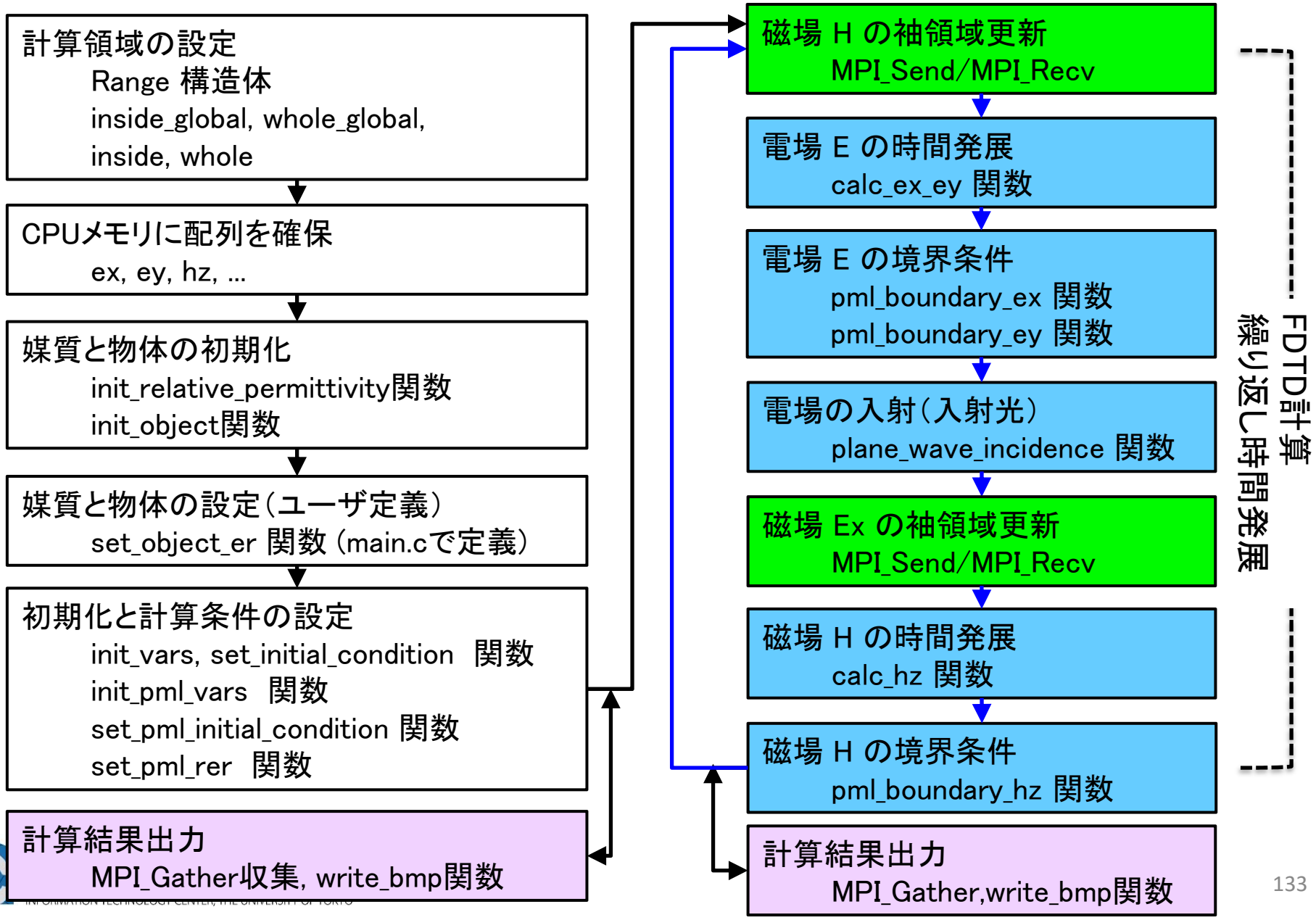
マルチGPU化のための領域分割

y 方向で全領域を分割
各分割領域に袖領域を持たせます。



全体領域：袖領域（PML、青格子）を含む領域
中心領域：袖領域を含まない領域

コード全体の流れ（main.c 内）



計算領域の設定(1)

- Range 構造体
  - ✓ 計算領域の始点と大きさを保持

```
// config.h
struct Range {
    int length[2];
    int begin [2];
};

// main.c
const struct Range inside_global = { { atoi(argv[1]), atoi(argv[2]) },
                                     { 0, 0 } };
const struct Range whole_global  = { { inside_global.length[0] + 2*mgn + 1,
                                       inside_global.length[1] + 2*mgn + 1},
                                     { inside_global.begin[0]  - mgn       ,
                                       inside_global.begin[1]  - mgn    } };

const struct Range inside        = { { inside_global.length[0],
                                       inside_global.length[1]/nsubdomains },
                                     { 0,
                                       inside_global.length[1]/nsubdomains * rank } };
const struct Range whole         = { { inside.length[0] + 2*mgn + 1,
                                       inside.length[1] + 2*mgn + 1},
                                     { inside.begin[0]  - mgn       ,
                                       inside.begin[1]  - mgn    } };
```

全領域の中心領域

全領域の
全体領域

分割領域の
中心領域

分割領域の
全体領域

計算領域の設定(2)

- Range 構造体
  - ✓ 計算領域の始点と大きさを保持

```
struct Range {
    int length[2];
    int begin [2];
};

const struct Range inside       = { { inside_global.length[0],
                                      inside_global.length[1]/nsubdomains },
                                    { 0,
                                      inside_global.length[1]/nsubdomains * rank } };
const struct Range whole        = { { inside.length[0] + 2*mgn + 1,
                                      inside.length[1] + 2*mgn + 1},
                                    { inside.begin[0]  - mgn      ,
                                      inside.begin[1]  - mgn    } };
```

分割領域の
中心領域

分割領域の
全体領域

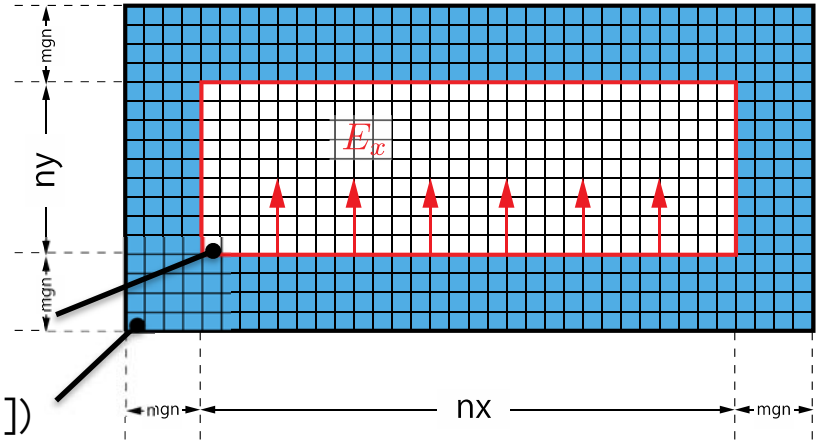
プログラムでは下記の通り

inside.length[0] = nx
inside.length[1] = ny

whole.length[0] = nx + 2*mgn + 1
whole.length[1] = ny + 2*mgn + 1

座標(inside.begin[0], inside.begin[1])

座標(whole.begin[0], whole.begin[1])



配列の確保

- 物理変数配列は main.c で確保

```
// main.c
const int    nelems      = whole.length[0] * whole.length[1];
const int    nelems_x    = whole.length[0];
const int    nelems_y    = whole.length[1];
const size_t size        = sizeof(FLOAT)*nelems;
const size_t size_x      = sizeof(FLOAT)*nelems_x;
const size_t size_y      = sizeof(FLOAT)*nelems_y;
const size_t size_global = sizeof(FLOAT)* whole_global.length[0] * whole_global.length[1];

FLOAT *ex    = (FLOAT *)malloc(size);  // 電場 Ex
FLOAT *ey    = (FLOAT *)malloc(size); // 電場 Ey
FLOAT *hz    = (FLOAT *)malloc(size); // 磁場 Hz
...
// For output
FLOAT *ex_global = (FLOAT *)malloc(size_global);
FLOAT *ey_global = (FLOAT *)malloc(size_global);
FLOAT *hz_global = (FLOAT *)malloc(size_global);
```

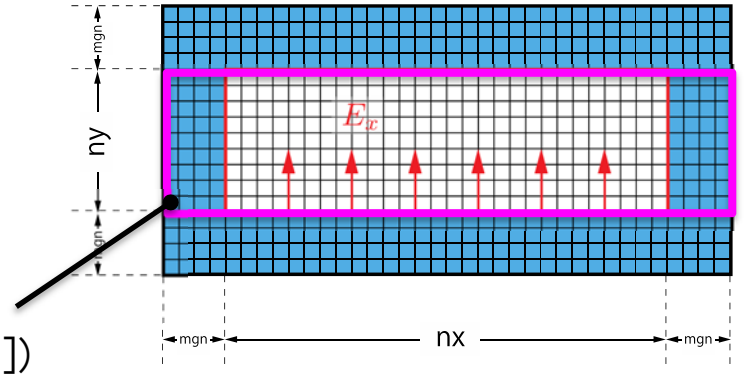
  - 多くの配列は whole.length[0] * whole.length[1]
  - ex_global, ey_global, hz_global はファイル出力に使うため、
    whole_global.length[0] * whole_global.length[1]

計算結果出力

- 各ランクの ex を ex_global へMPI_Gatherで収集
  - ✓ ey, hz も同様

```
// main.c
const int rank_root  = 0;
const int sendnelems = whole.length[0] * inside.length[1];
const int src        = whole.length[0] * (inside.begin[1] - whole.begin[1]);
const int dst        = whole.length[0] * (inside.begin[1] - whole.begin[1]);

MPI_Gather(&ex[src], sendnelems, MPI_FLOAT_T, &ex_global[dst],
           sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);
MPI_Gather(&ey[src], sendnelems, MPI_FLOAT_T, &ey_global[dst],
           sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);
MPI_Gather(&hz[src], sendnelems, MPI_FLOAT_T, &hz_global[dst],
           sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);

if (rank == rank_root) {
    write_bmp(icnt, time, whole_global.length, dx, dy, ex_global, ey_global, hz_global);
}
```

袖領域を落とすため、y方向には中心領域のみ切り出して、MPI_Gatherする



src = whole.length[0] * (inside.begin[1] - whole.begin[1])

時間発展(1)

- 前半
  - ✓ 電場Eの時間発展(calc_ex_ey)、境界条件(pml_boundary_...)
  - ✓ 入射光(plane_wave_incidence)

```
while (icnt < nt) {

    MPI_Status status;
    const int tag = 0;
    const int nhalo       = whole.length[0];
    const int inside_end1 = inside.begin[1] + inside.length[1];

    const int src_hz      = whole.length[0] * (inside_end1      - whole.begin[1] - 1);
    const int dst_hz      = whole.length[0] * (inside.begin[1] - whole.begin[1] - 1);

    MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up   , tag, MPI_COMM_WORLD);
    MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD, &status);

    calc_ex_ey(&whole, &inside, hz, cexly, ceylx, ex, ey);
    pml_boundary_ex(&whole, &inside, hz, cexy, cexyl, rer_ex, ex, exy);
    pml_boundary_ey(&whole, &inside, hz, ceyx, ceyxl, rer_ey, ey, eyx);

    const int j_in = 0;
    plane_wave_incidence(&whole, &inside, time, j_in, wavelength, ex, ey);
    time += 0.5*dt;

    (後半へ)
```

時間発展(2)

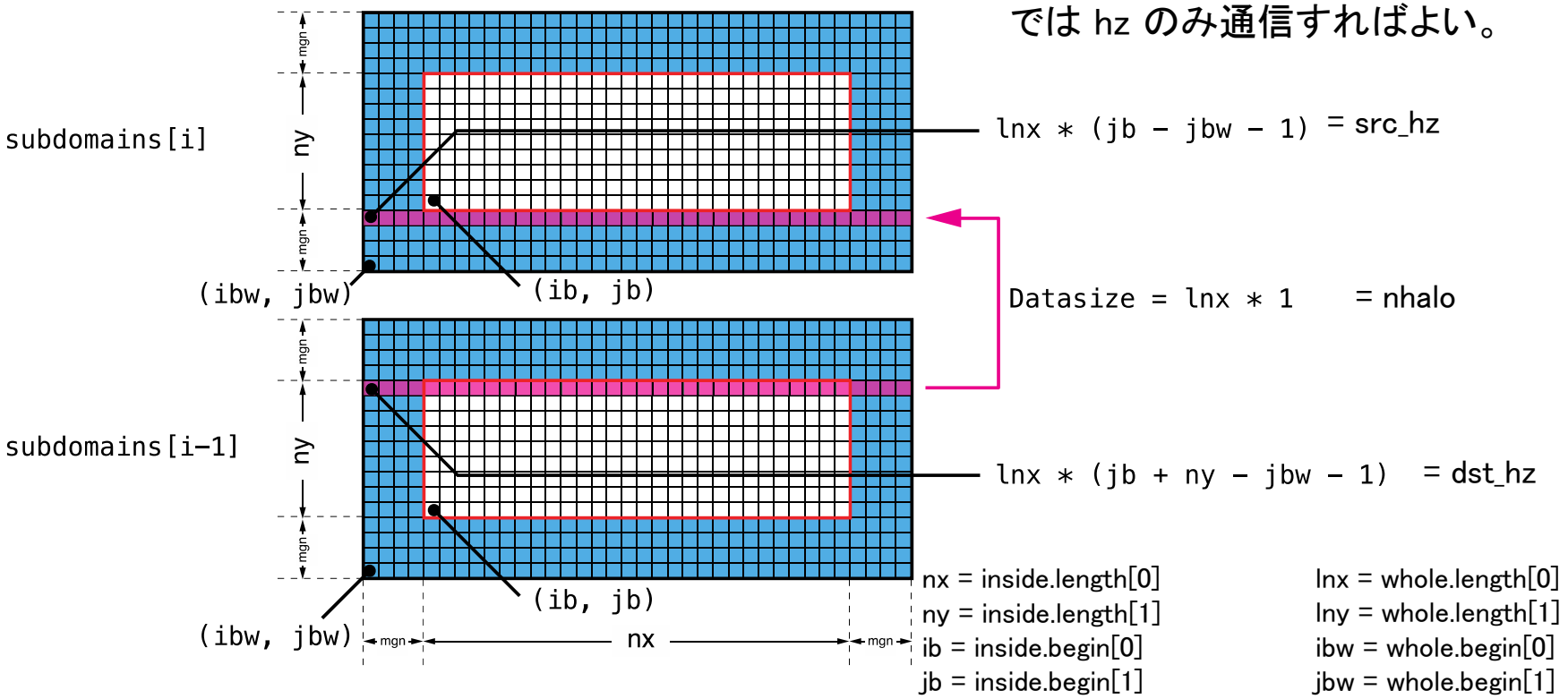
- 後半
  - ✓ 磁場Hの時間発展(calc_hz)、境界条件(pml_boundary_hz)

```
    (前半から)

    const int src_ex      = whole.length[0] * (inside.begin[1] - whole.begin[1]);
    const int dst_ex      = whole.length[0] * (inside_end1     - whole.begin[1]);

    MPI_Send(&ex[src_ex], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD);
    MPI_Recv(&ex[dst_ex], nhalo, MPI_FLOAT_T, rank_up  , tag, MPI_COMM_WORLD, &status);

    calc_hz(&whole, &inside, ey, ex, chzlx, chzly, hz);
    pml_boundary_hz(&whole, &inside, ey, ex, chzx, chzxl, chzy, chzyl, hz, hzx, hzy);
    time += 0.5*dt;

    icnt++;

    (出力など)
}
```

境界領域のデータ交換(1)

- hz の境界領域のデータ交換

データアクセスが非対称のため、ここでは hz のみ通信すればよい。

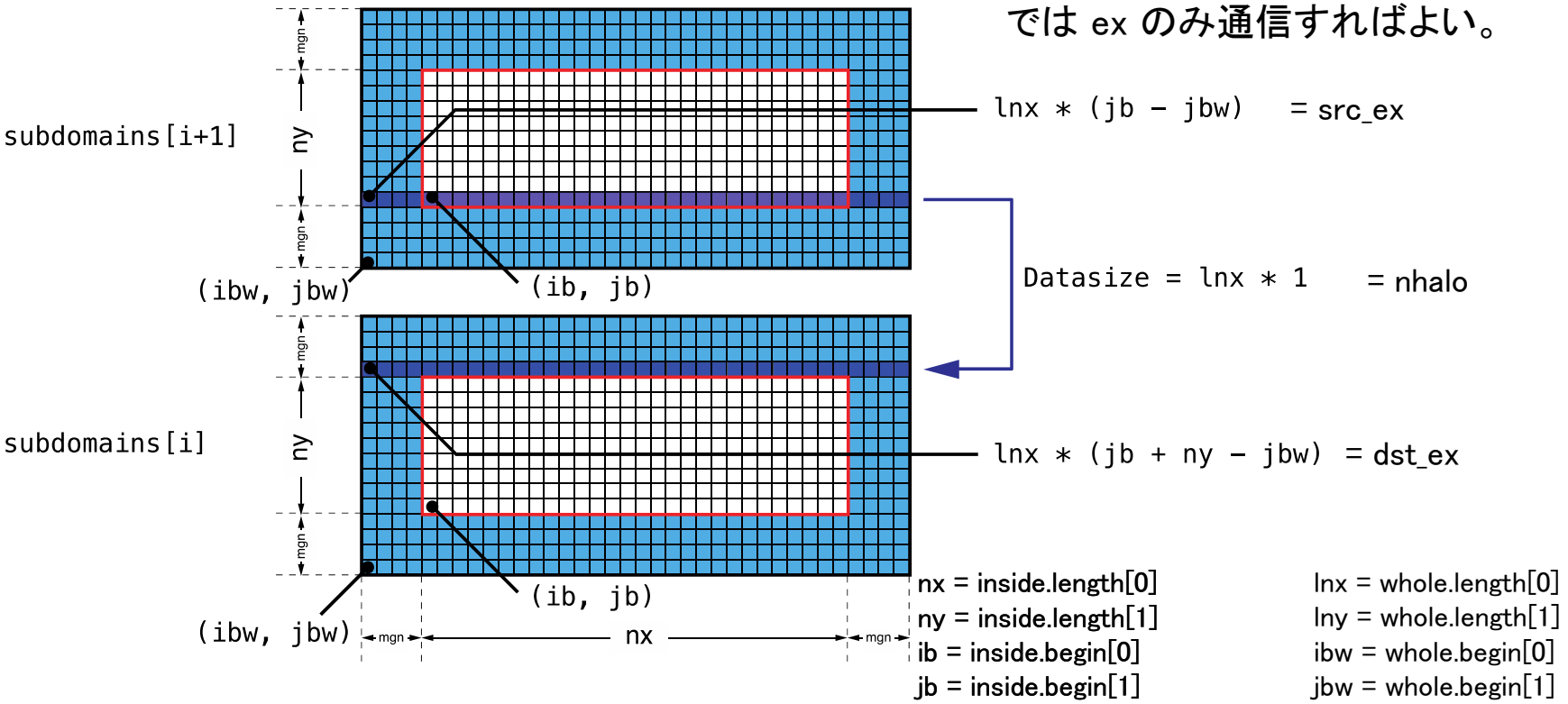



```
const int nhalo       = whole.length[0];
const int inside_end1 = inside.begin[1] + inside.length[1];

const int src_hz      = whole.length[0] * (inside_end1      - whole.begin[1] - 1);
const int dst_hz      = whole.length[0] * (inside.begin[1] - whole.begin[1] - 1);

MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up   , tag, MPI_COMM_WORLD);
MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD, &status);
```

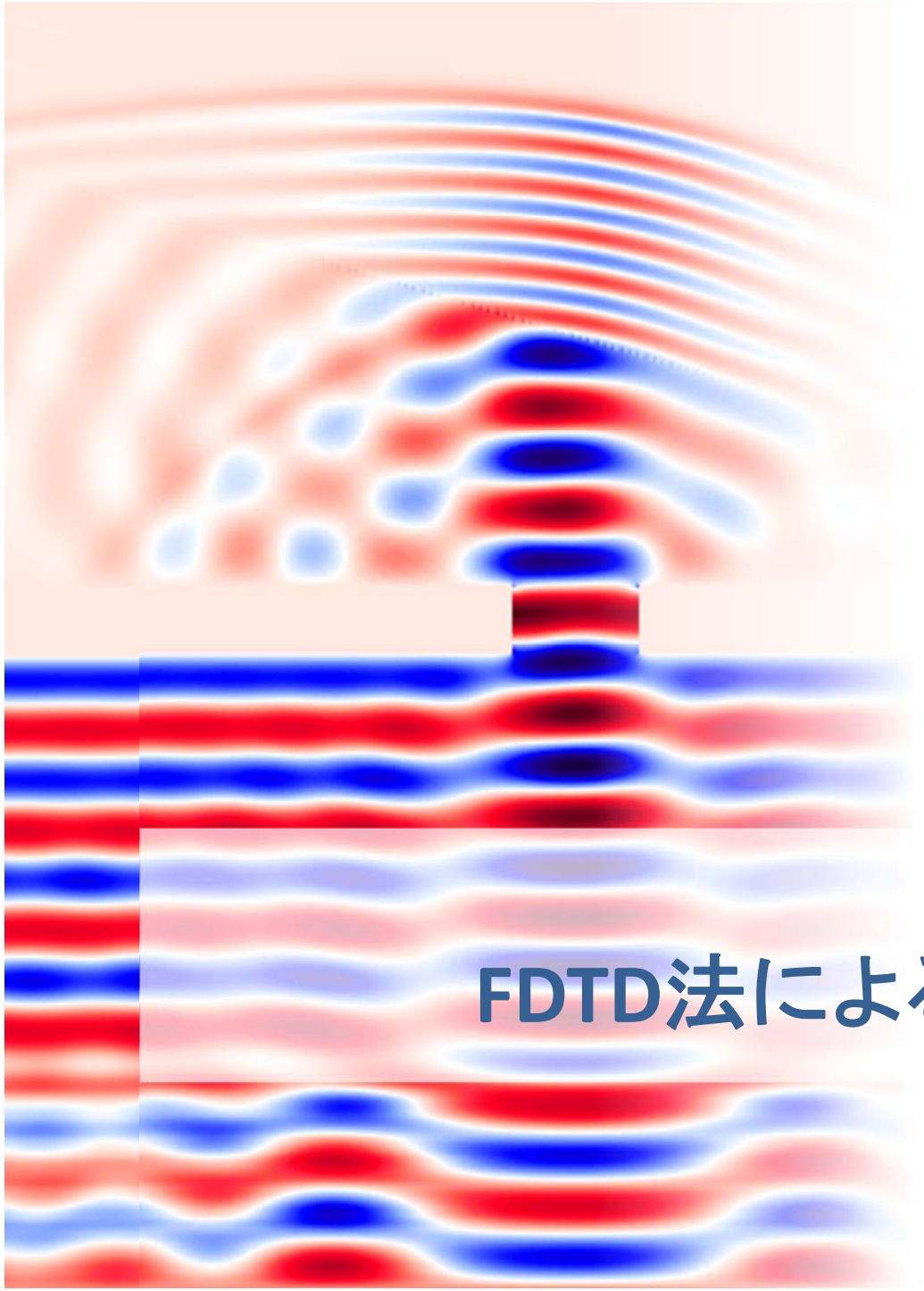
境界領域のデータ交換(2)

- ex の境界領域のデータ交換

データアクセスが非対称のため、ここでは ex のみ通信すればよい。




```
const int nhalo       = whole.length[0];
const int inside_end1 = inside.begin[1] + inside.length[1];

const int src_ex      = whole.length[0] * (inside.begin[1] - whole.begin[1]);
const int dst_ex      = whole.length[0] * (inside_end1     - whole.begin[1]);

MPI_Send(&ex[src_ex], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD);
MPI_Recv(&ex[dst_ex], nhalo, MPI_FLOAT_T, rank_up  , tag, MPI_COMM_WORLD, &status);
```



GPUを用いた FDTD法による電磁波伝搬計算の実習

プログラムのコンパイルと実行（1）

- CPUコードのコンパイルと実行

openacc_fdtd/01_original

```
$ cd openacc_mpi_fdtd/01_original
$ make
$ qsub ./run.sh
$ cat run.sh.o??????
Rank 0: hostname = a090
Rank 1: hostname = a090
Rank 2: hostname = a091
Rank 3: hostname = a091
Calculation condition
  nx_global      =   512

（省略）

icnt =  4900, time = 2.3115e-14 [sec]
icnt =  5000, time = 2.3587e-14 [sec]
------------------------------
Domain      = 512 x 512
nsubdomains = 4
output_file = 1
Time        =   4.103535 [sec]
------------------------------
```

? の数字はジョブごとに変わります。

利用したノード

計算領域サイズ、領域分割数、出力の有無、計算時間

なお、qsub ./run_no_out.sh すると出力なしで実行する。性能測定用。

プログラムのコンパイルと実行(2)

- プログラムの実行時オプション

openacc_fdtd/01_original

```
$ cat run.sh
#! /bin/sh
#PBS -q h-tutorial
#PBS -l select=1:mpiprocs=1:ompthreads=0

(省略)

mkdir -p sim_run
cd sim_run

nprocs=1
mpirun -np $nprocs ../run 512 512 $nprocs 5000 50
```

mpirun -np <nprocs> ../run <nx> <ny> <nprocs> <nt> <nout>

nprocs: 全ランク数(=分割数) ※今回は1

nx, ny: 計算領域サイズ

nt: 全時間ステップ

nout: 出力を行うタイムステップ数。50 の場合、50ステップに1回出力する。0 を指定すると出力しない。

計算結果の表示

- 計算結果は sim_run に BMP として出力される

```
$ cd sim_run/
```

openacc_mpi_fdtd/01_original

- 計算結果の表示
  - ✓ 1枚のBMPを見る

```
$ display e05000.bmp
```

  - ✓ 複数のBMPファイルをアニメーションで表示

```
$ animate *.bmp
```

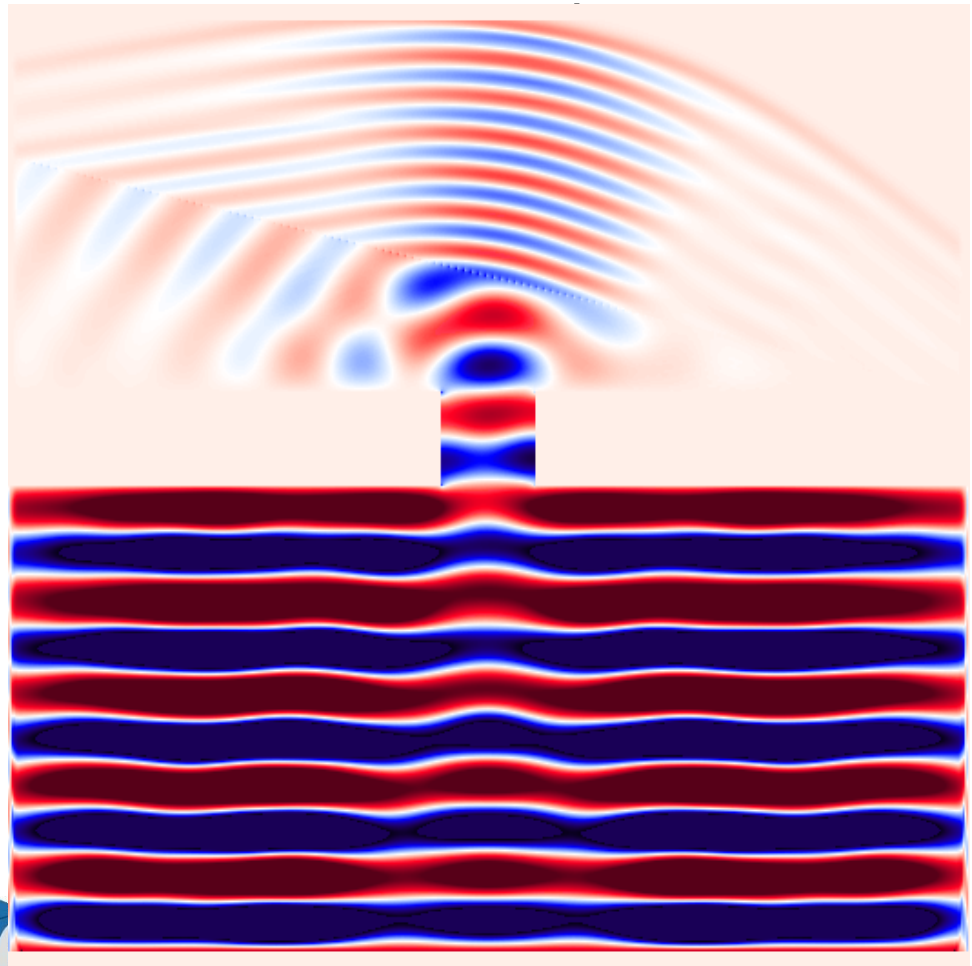
なお

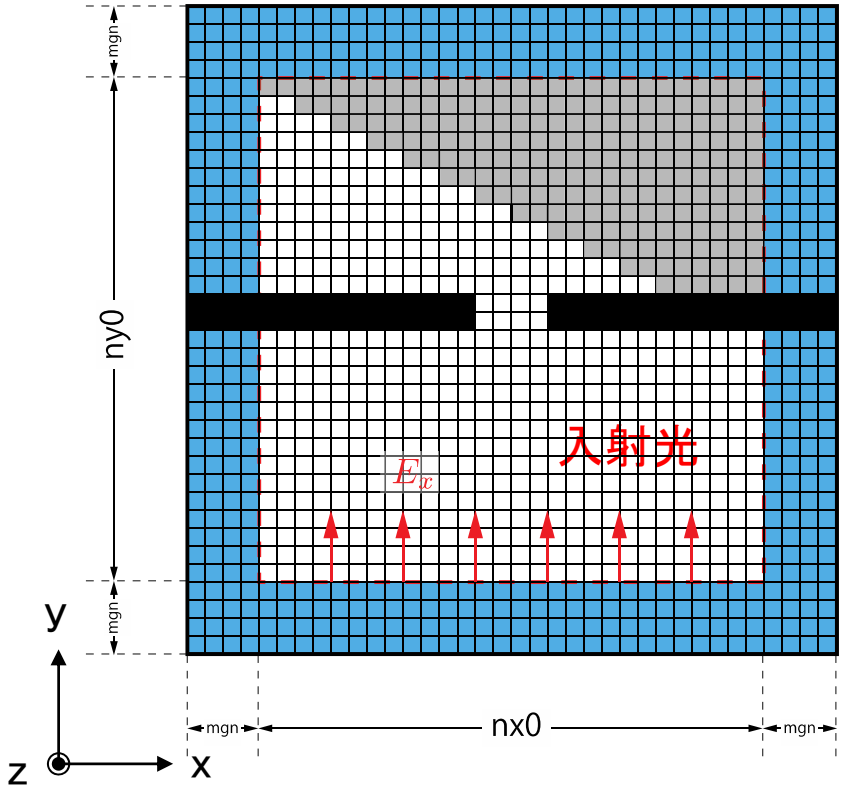
ssh -Y txxxxx@reedbush.cc.u-tokyo.ac.jp

と –Y をつけていないと表示されない。うまく表示できない場合は画像を手元にコピーして表示してください。

計算結果の例

- 出力されたBMPファイルの一例





実習1

- calc_ex_ey, pml_boundary_ex, pml_boundary_ey を OpenACC化しましょう。

- Makefile
  - ✓ コンパイルオプションの修正
- main.c
  - ✓ OpenACCヘッダーの追加
  - ✓ data 指示文の追加
- fdtd2d.c
  - ✓ kernels 指示文、loop 指示文の追加

実行速度が遅くても、動くプログラムである状態を保ちながらOpenACC化します。
末端の関数からOpenACC化するのがよいでしょう。

解答例は、openacc_fdtd/02_openacc1

data , host_data指示文

- main関数のwhile 内で data, host_dataを追加

```
#pragma acc data ¥
    copy(ex[0:nelems], ey[0:nelems], hz[0:nelems])                    ¥
    copyin(cexly[0:nelems], ceylx[0:nelems], chzlx[0:nelems], chzly[0:nelems]) ¥
    copy(exy[0:nelems], eyx[0:nelems], hzx[0:nelems], hzy[0:nelems])  ¥
    copyin(cexy[0:nelems_y], ceyx[0:nelems_x], chzx[0:nelems_x], chzy[0:nelems_y]) ¥
    copyin(cexyl[0:nelems_y], ceyxl[0:nelems_x], chzxl[0:nelems_x], chzyl[0:nelems_y]) ¥
    copyin(obj[0:nelems], er[0:nelems]) ¥
    copyin(rer_ex[0:nelems], rer_ey[0:nelems])
        {

#pragma acc host_data use_device(hz)
            {
            MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up   , tag, MPI_COMM_WORLD);
            MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag,
                    MPI_COMM_WORLD, &status);
            }

            calc_ex_ey(&whole, &inside, hz, cexly, ceylx, ex, ey);
            pml_boundary_ex(&whole, &inside, hz, cexy, cexyl, rer_ex, ex, exy);
            pml_boundary_ey(&whole, &inside, hz, ceyx, ceyxl, rer_ey, ey, eyx);

        } // acc data

    const int j_in = 0;
    plane_wave_incidence(&whole, &inside, time, j_in, wavelength, ex, ey);
    time += 0.5*dt;
```

kernels, loop指示文

- fdtd2d.c 内の関数

```
void calc_ex_ey(const struct Range *whole, const struct Range *inside,
        const FLOAT *hz, const FLOAT *cexly, const FLOAT *ceylx, FLOAT *ex, FLOAT *ey)
{
    const int nx    = inside->length[0];
    const int ny    = inside->length[1];
    const int mgn[] = { inside->begin[0] - whole->begin[0],
                        inside->begin[1] - whole->begin[1] };
    const int lnx   = whole->length[0];

#pragma acc kernels present(hz, cexly, ex)
#pragma acc loop independent
    for (int j=0; j<ny+1; j++) {
#pragma acc loop independent
        for (int i=0; i<nx; i++) {
            const int ix = (j+mgn[1])*lnx + i+mgn[0];
            const int jm = ix - lnx;
            //ex[ix] += cexly[ix]*(hz[ix]-hz[jm]) - cexlz[ix]*(hy[ix]-hy[km]);
            ex[ix] += cexly[ix]*(hz[ix]-hz[jm]);
        }
    }

(省略)

}
```

実習2

- main 関数内の while 内をすべて OpenACCにしましょう。

- main.c
  - ✓ data 指示文の移動と copyin などの最適化
- fdtd2d.c
  - ✓ 残りの関数にkernels 指示文、loop 指示文の追加
- fdtd2d_sources.c
  - ✓ kernels 指示文、loop 指示文の追加

解答例は、 openacc_fdtd/03_openacc2

data 指示文

- main関数のwhile 外に data を移動

```
#pragma acc data ¥
    copyin(ex[0:nelems], ey[0:nelems], hz[0:nelems])                    ¥
    copyin(cexly[0:nelems], ceylx[0:nelems], chzlx[0:nelems], chzly[0:nelems]) ¥
    copyin(exy[0:nelems], eyx[0:nelems], hzx[0:nelems], hzy[0:nelems])  ¥
    copyin(cexy[0:nelems_y], ceyx[0:nelems_x], chzx[0:nelems_x], chzy[0:nelems_y]) ¥
    copyin(cexyl[0:nelems_y], ceyxl[0:nelems_x], chzxl[0:nelems_x], chzyl[0:nelems_y]) ¥
    copyin(obj[0:nelems], er[0:nelems]) ¥
    copyin(rer_ex[0:nelems], rer_ey[0:nelems])
    {

    while (icnt < nt) {

        MPI_Status status;
        const int tag = 0;
        const int nhalo       = whole.length[0];
        const int inside_end1 = inside.begin[1] + inside.length[1];

        const int src_hz      = whole.length[0] * (inside_end1      - whole.begin[1] - 1);
        const int dst_hz      = whole.length[0] * (inside.begin[1] - whole.begin[1] - 1);

#pragma acc host_data use_device(hz)
        {
        MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up   , tag, MPI_COMM_WORLD);
        MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD, &status);
        }
```

host_data 指示文

- MPI_Gather に対する host_data の追加

```
    const int rank_root  = 0;
    const int sendnelems = whole.length[0] * inside.length[1];
    const int src        = whole.length[0] * (inside.begin[1] - whole.begin[1]);
    const int dst        = whole.length[0] * (inside.begin[1] - whole.begin[1]);

#pragma acc host_data use_device(ex)
    MPI_Gather(&ex[src], sendnelems, MPI_FLOAT_T, &ex_global[dst],
               sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);
#pragma acc host_data use_device(ey)
    MPI_Gather(&ey[src], sendnelems, MPI_FLOAT_T, &ey_global[dst],
               sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);
#pragma acc host_data use_device(hz)
    MPI_Gather(&hz[src], sendnelems, MPI_FLOAT_T, &hz_global[dst],
               sendnelems, MPI_FLOAT_T, rank_root, MPI_COMM_WORLD);

    if (rank == rank_root) {
        write_bmp(icnt, time, whole_global.length, dx, dy, ex_global, ey_global,hz_global);
    }
```

実習3

- 初期化を含めて全てOpenACCにします。ただし、set_object_er がCPU上のユーザ定義関数のため、これ以降の初期化関数をOpenACCにします。
- main.c
  - ✓ data 指示文の移動と最適化（多くが create になるはずです）
- setup.c
  - ✓ kernels 指示文、loop 指示文の追加

解答例は、openacc_fdtd/04_openacc3

実習4

- 計算領域のサイズなどを変更して性能測定してみましょう。
- OpenACCコードをさらに最適化しましょう。
  - ✓ PGI_ACC_TIMEも活用しましょう。
  - ✓ 実は単純に fdtd2d.c に kernels と loop を入れても、いくつかの関数で暗黙の copyin が発生します。これも修正していきましょう。

```
$ make
calc_ex_ey:
     25, Generating present(ex[:],cexly[:])
         Generating implicit copyin(mgn[:])
         Generating present(hz[:])
     27, Loop is parallelizable
     29, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         27, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         29, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     37, Generating present(ey[:],ceylx[:])
         Generating implicit copyin(mgn[:])
```

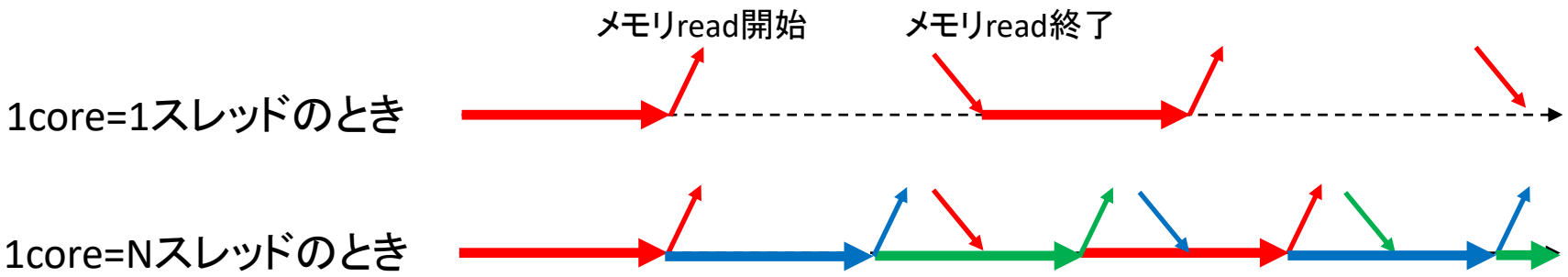
解答例は、openacc_fdtd/05_openacc4

Q & A

- アカウントは1ヶ月有効です。
- 資料のPDF版はWEBページにございます。
  - https://www.cc.u-tokyo.ac.jp/events/lectures/97/lec097.php
- アンケートへの協力をお願いします。

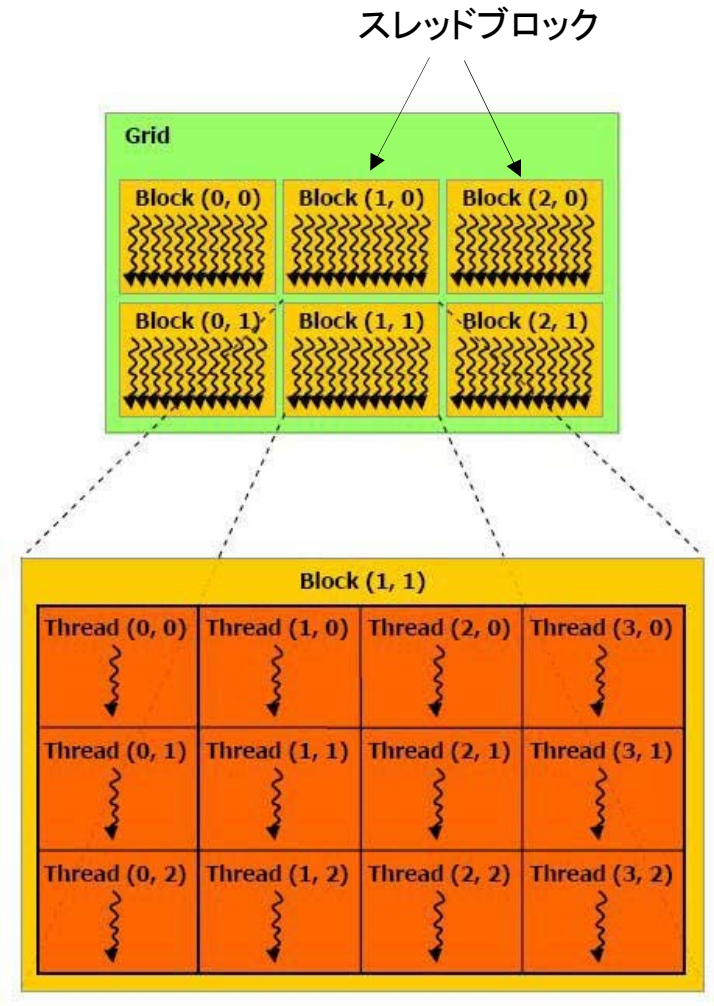
性能を出すためにはスレッド数>>コア数

- 推奨スレッド数
  - CPU: スレッド数=コア数 (高々数十スレッド)
  - GPU: スレッド数>=コア数*4~ (数万~数百万スレッド)
    - 最適値は他のリソースとの兼ね合いによる
- 理由:高速コンテキストスイッチによるメモリレイテンシ隠し
  - CPU : レジスタ・スタックの退避はOSがソフトウェアで行う(遅い)
  - GPU : ハードウェアサポートでコストほぼゼロ
    - メモリアクセスによる暇な時間(ストール)に他のスレッドを実行



階層的スレッド管理とコミュニケーション

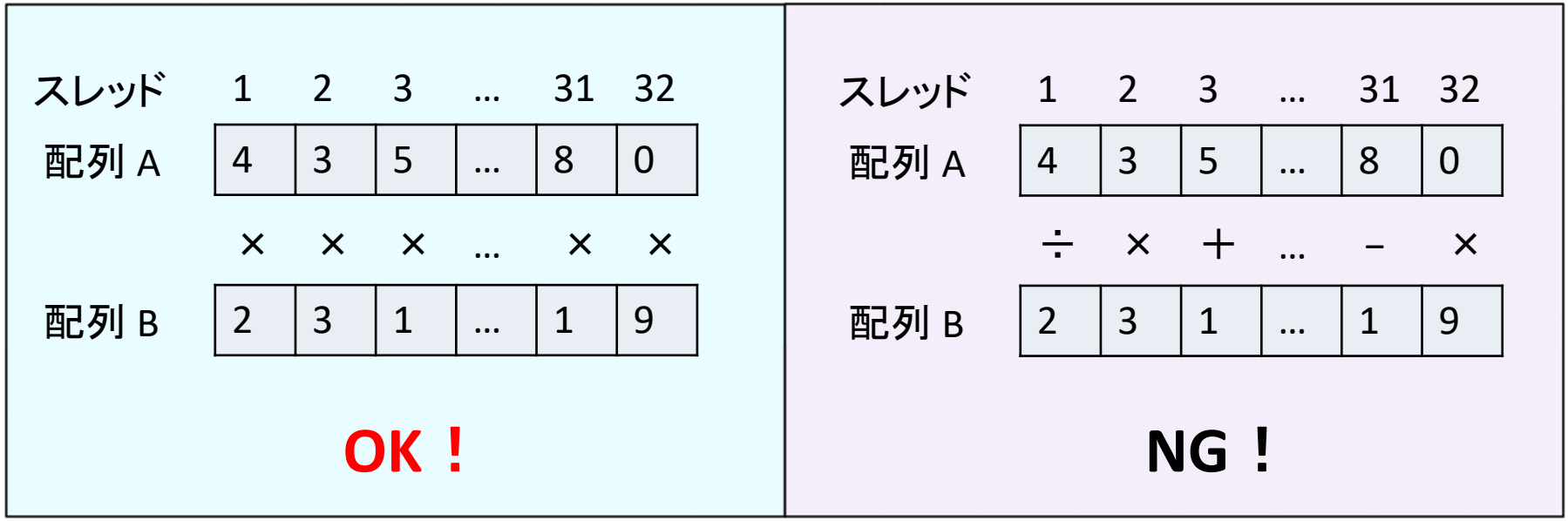
- 階層的なコア/スレッド管理
  - P100は56 SMを持ち、1 SMは64 CUDA coreを持つ。トータル3584 CUDA core
  - 1 SMが複数のスレッドブロックを担当し、1 CUDA core が複数スレッドを担当
- スレッド間のコミュニケーション
  - 同一スレッドブロック内のスレッドは高速コミュニケーション可能
  - 異なるスレッドブロックに属するスレッド間はコミュニケーションが低速
    - いったんメモリに書き出したり、CPUに処理を戻さなくてはならない



cited from : http://cuda-programming.blogspot.jp/2012/12/thread-hierarchy-in-cuda-programming.html

Warp 単位の実行

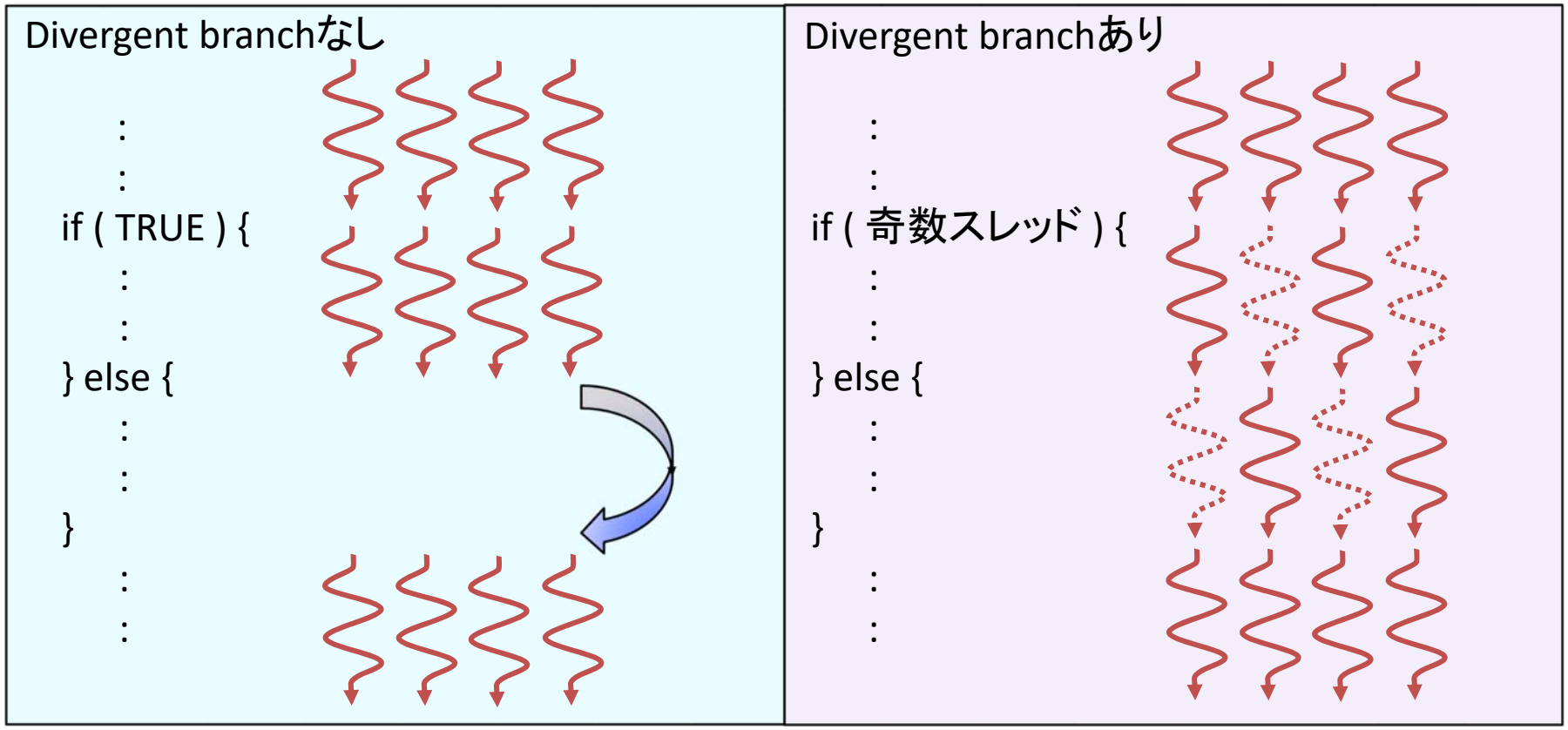
- 連続した32スレッドを1単位 = Warp と呼ぶ
- このWarpは足並み揃えて動く
  - 実行する命令は32スレッド全て同じ
  - データは違ってもいい



Warp内分岐

CUDA 8 以前のバージョン
（本講習会はCUDA 8 準拠）

- Divergent Branch
  - Warp 内で分岐すること。Warp単位の分岐ならOK。



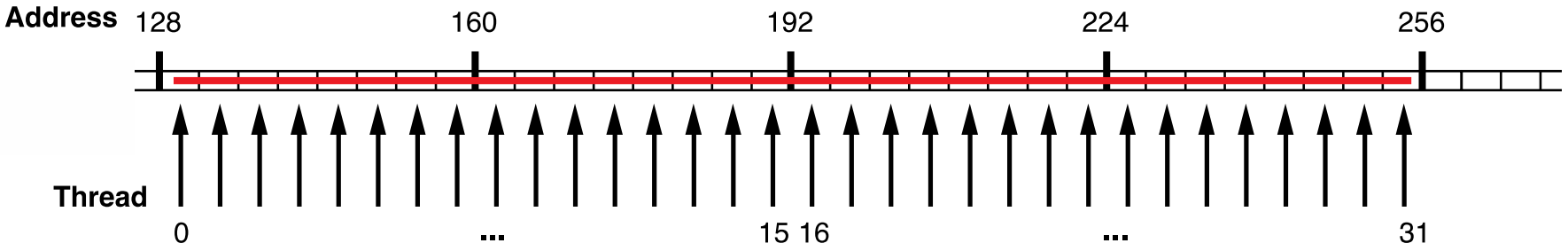
else 部分は実行せずジャンプ

一部スレッドを眠らせて全分岐を実行
最悪ケースでは32倍のコスト

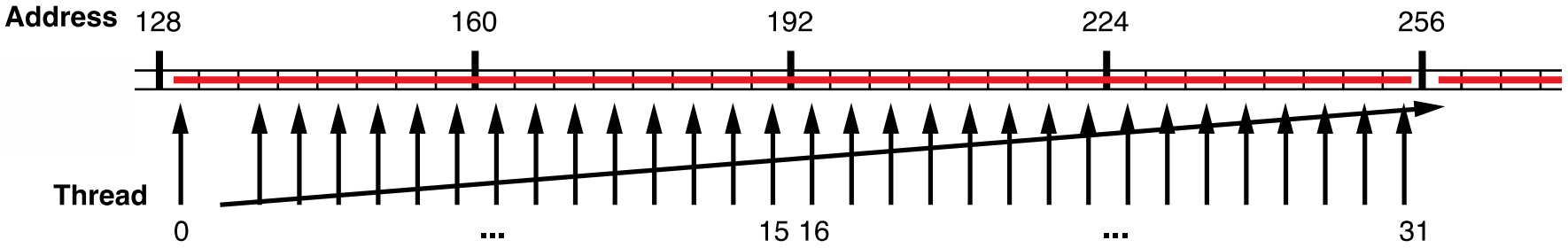
コアレスドアクセス

- 同じWarp内のスレッド（連続するスレッド）は近いメモリアドレスへアクセスすると効率的
  - ✓ コアレスドアクセス（coalesced access）と呼ぶ
  - ✓ メモリアクセスは128 Byte 単位で行われる。128 Byte に収まれば1回のアクセス、超えれば128 Byte アクセスをその分繰り返す。

128 byte x 1回のメモリアクセス



128 byte x 2回のメモリアクセス



ストライドアクセスがあるとどうなるか

- GPUはストライドアクセスに弱い！

```
void AoS_STREAM_Triad(STREAM_TYPE scalar)
{
  ssize_t i,j;
#pragma omp parallel for private(i,j)
#pragma acc kernels present(a_aos[0:STREAM_ARRAY_SIZE] ¥
      ,b_aos[0:STREAM_ARRAY_SIZE],c_aos[0:STREAM_ARRAY_SIZE])
#pragma acc loop gang vector independent
  for (j=0; j<STREAM_ARRAY_SIZE/STRIDE; j++)
    for (i=0; i<STRIDE; i++)
      a_aos[j*STRIDE+i] = b_aos[j*STRIDE+i]+scalar*c_aos[j*STRIDE+i];
}
```

ストライドアクセス付き stream triad