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

東京大学 情報基盤センター

担当：山崎一哉

kyamazaki @ cc.u-tokyo.ac.jp

（内容に関するご質問はSlackかこちらまで）

講習会スケジュール

- 開催日時
  - ✓4月21日（月） 10:00 – 17:00

- プログラム
  - ✓10:00 - 10:50 スパコンの使い方など
  - ✓11:00 - 11:50 GPUとOpenACC基礎（座学）
  - ✓（昼休み）
  - ✓13:30 – 14:20 OpenACC演習 Ⅰ
  - ✓14:30 – 15:20 OpenACC演習 Ⅱ
  - ✓15:30 – 16:20 OpenACC演習 Ⅲ
  - ✓16:30 – 17:00 質問など

講習会について

- 本講習会は
  - ✓ GPUに関する基礎知識
  - ✓ OpenACCを用いたGPUプログラミングの基礎

  を中心に扱います。

- その他の講習会

  https://www.cc.u-tokyo.ac.jp/events/lectures/

- スパコンイベント情報メール配信サービス

  https://regist.cc.u-tokyo.ac.jp/announce/

  - ✓ 講習会や研究会の案内、トライアルユースの実施のお知らせなどを配信しています。

Youtubeにて過去の講習会を配信中！

https://www.youtube.com/channel/UC2CHaGp1AO-vqRlV7wmU0-w/videos?view=0&sort=p&flow=grid

講習会の進め方

- Zoomを利用したオンライン講習会です
  - ✓ この講義は録画されています
  - ✓ 質問があるとき以外はミュートでお願いします
  - ✓ ビデオもオフを推奨します
- slackを使って質問に対応します
  - ✓ slackはリンクを知っている人は誰でも使える設定になっています
  - ✓ slackのリンクをzoomのチャットに貼るので、未登録の場合は今のうちに登録お願いします
    - ✓ slackの登録メールの配送に小一時間かかることがあります

Zoom: 「手を挙げる」方法

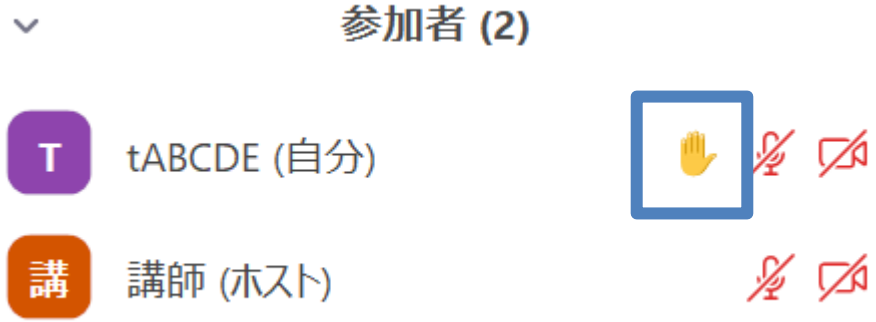
1. Zoomメニュー中の「リアクション」をクリック

2. ポップアップで表示された「手を挙げる」をクリック

Zoom: 手が挙がっていることの確認方法

1. Zoomメニュー中の「参加者」をクリックして，参加者一覧を表示

2. 表示された参加者一覧の，自分のところを見ると手が挙がっている



Zoom: 「手を降ろす」方法

1. Zoomメニュー中の「リアクション」をクリック

2. ポップアップで表示された「手を降ろす」をクリック

Slack: 質疑応答チャンネルへの移動

- 左側のメニューバーのチャンネル一覧内に「第242回-gpuプログラミング入門」があるので，クリック

- 表示されていない場合
  1. 「チャンネルを追加する」をクリック
  2. 「チャンネル一覧を確認する」をクリック
  3. 「第242回-gpuプログラミング入門」があるので，「参加する」をクリック

Slack: メッセージの入力方法

- 最下部に入力欄があるので，質問内容を記載して Ctrl+Enter
  - 入力後に右下の「メッセージを送信する」をクリックしても同じ
    （メッセージ入力前には，「メッセージを送信する」は押せない）

- コードを入力する際には，「コードブロック」がおすすめ
  - 枠が生成されるので，この中にコピペするのが簡単かつ見やすい
  - ```（JIS配列ならばShift+@を3連打）しても枠が生成される

サンプルコードのダウンロード

- 講習会で使うサンプルコードはGitHubにあります
  - https://github.com/kazuya-yamazaki/lecture_openacc

- ダウンロード手順
  1. $ ssh tXXXXX@miyabi-g.jcahpc.jp
     - Miyabi-Gへのssh。tXXXXXはアカウント名
  2. $ cd /work/gt00/tXXXXX
  3. $ git clone https://github.com/kazuya-yamazaki/lecture_openacc.git
  4. $ cd lecture_openacc/
  5. $ cd C or F
     - CまたはFortran好きな方を利用してください。

東大情報基盤センターの
スパコン概要

東大情報基盤センターで現在提供しているスパコン

Wisteria
BDEC-01

運用者：東京大学
運用期間（予定）：2021～2027年

Odyssey (**CPU**)

A64FX 7680基
25.9 PF, 7.8 PB/s

Aquarius (GPU搭載)

A100 360基
7.2 PF, 0.6 PB/s

東大情報基盤センターで現在提供しているスパコン

Wisteria
BDEC-01

運用者：東京大学
運用期間（予定）：2021～2027年

Odyssey (**CPU**)

A64FX 7680基
25.9 PF, 7.8 PB/s

Aquarius (GPU搭載)

A100 360基
7.2 PF, 0.6 PB/s

---

miyabi

運用者：JCAHPC (筑波大・東大)
2025年1月　運用開始

Miyabi-C (CPU)

Intel SPR HBM　380基
1.3 PF, 0.6 PB/s

Miyabi-G (**GPU**搭載)

GH200 1120基
78.8 PF, 5.1 PB/s

東大情報基盤センターで現在提供しているスパコン

Wisteria
BDEC-01

Odyssey (**CPU**)

Aquarius (GPU搭載)

GPUは価格および消費電力の観点で優れているため、
今後の大規模スパコンは、性能の大半をGPUが担う

Miyabi

運用者：JCAHPC (筑波大・東大)
2025年1月　運用開始

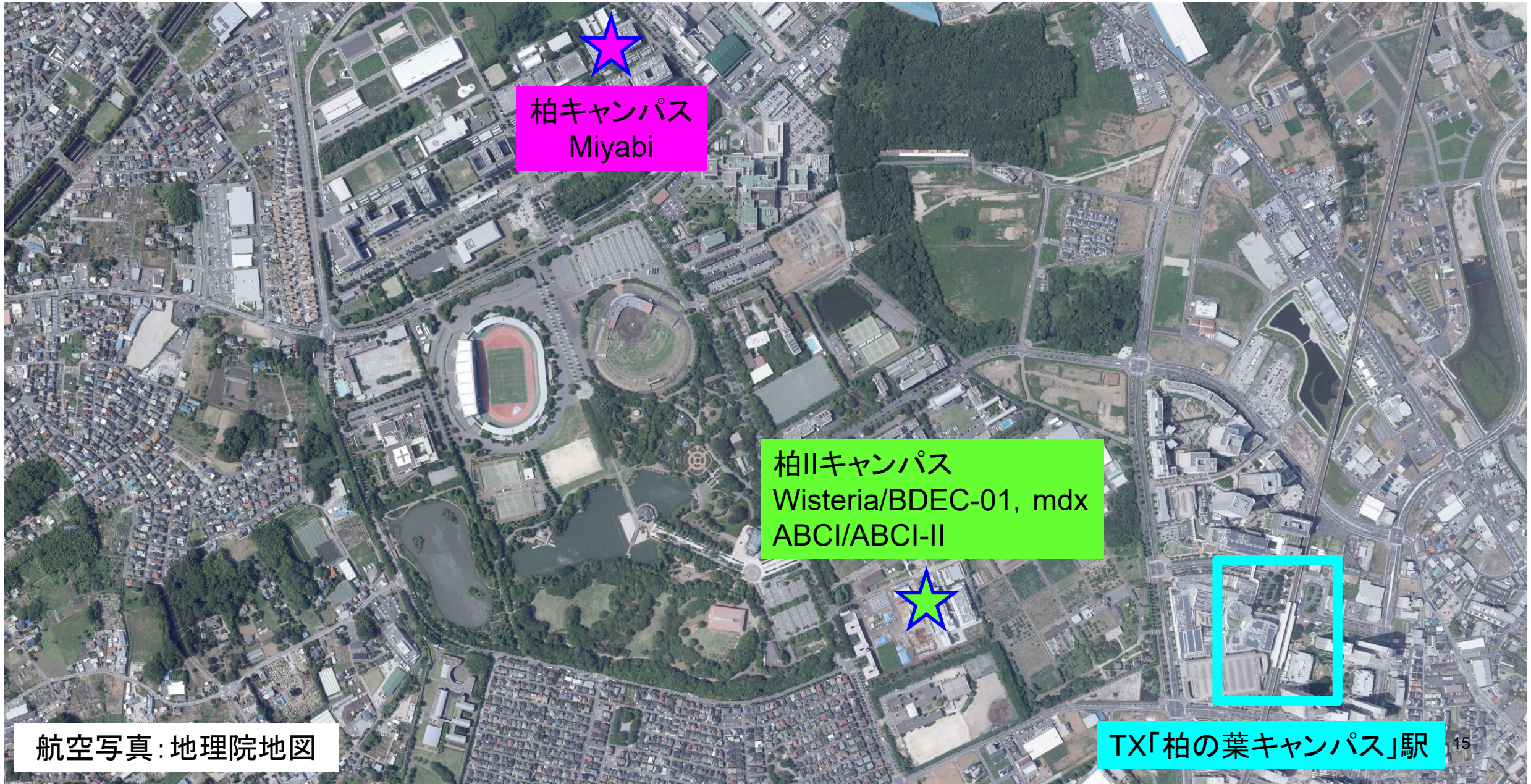
Miyabi-C (CPU)

Intel SPR HBM　380基
1.3 PF, 0.6 PB/s

Miyabi-G (**GPU**搭載)

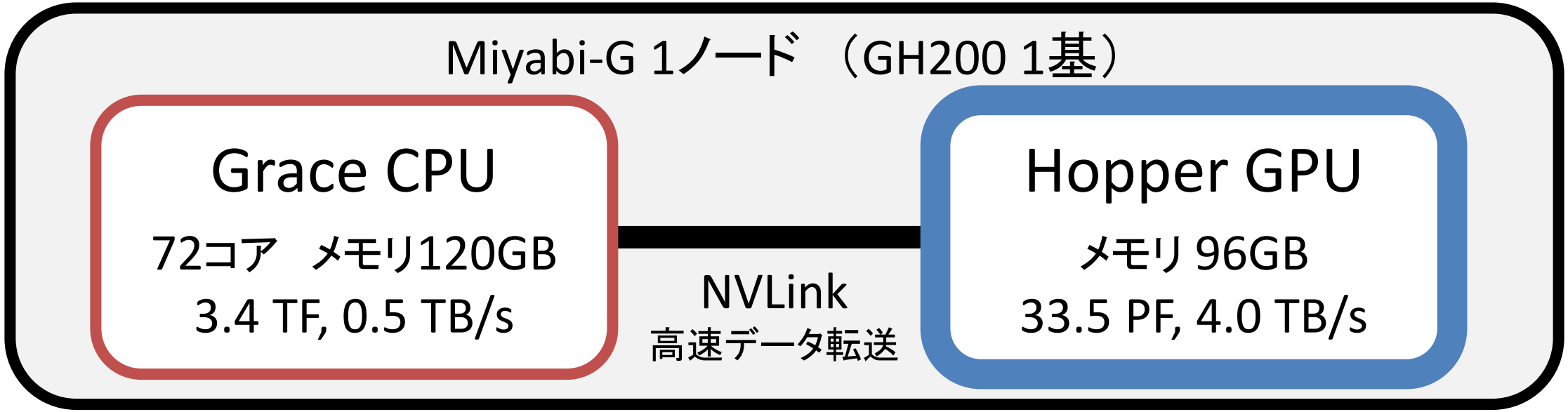
GH200 1120基
78.8 PF, 5.1 PB/s

千葉県柏市にはスパコン等が多数所在



Miyabi-Gの構成

- Miyabiの性能の大半を担う、GPU搭載ノード群。
- 1120個のノードを有する
- 1ノードあたり、Grace-Hopper Superchip (GH200) を1基搭載
  - NVIDIA社製 Grace CPUとHopper GPUが1基ずつ、密に結合
  - 小規模ジョブ用に、1ノードを4つに分割したMIGキューも提供



Miyabi 利用上の注意（１）

- ディレクトリについて（home と work)
  - ✓ ログイン時のディレクトリ（/home/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 /work 以下のディレクトリ（/work/gt00/txxxxx）に置く
    - ✓ /home は計算ノードからは参照できないスパコンが多い。容量も限られている
    - ✓ Miyabiでは、/homeにファイルを置きすぎて容量/ファイル数超過になると、ログインすら不可能になる場合がある

Miyabi 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。

$ module purge

使用中のモジュールを全てunloadする。

Miyabiでのプログラムの実行

- ジョブスクリプト(〇〇.sh)を作成し、ジョブとして投入、実行する。（pjsubではないので注意）

  $ qsub ./〇〇.sh

- 投入されたジョブを確認する。（pjstatではないので注意）

  $ qstat

- 実行が終了すると、以下のファイルが生成される。

  〇〇.sh.o??????（?????? はジョブID）

- 上記の標準出力ファイルの中身を確認する。

  $ cat 〇〇.sh.o??????

コンパイラの種類と実行(Miyabi-G)

- Miyabi-G用ログインノードとMiyabi-G計算ノードは、どちらもGrace CPUを使用しており、ログインノードで計算ノード用プログラムをコンパイル可能
  - **ログインノードで重い計算を走らせるのは禁止！**
- コンパイラ: GPU向けには主にNVIDIAを推奨 (gcc+CUDAも利用可能)

```
$ module load nvidia nv-hpcx または
$ module load gcc cuda ompi-cuda
```

言語	GNUコンパイラ	Intelコンパイラ	NVIDIA コンパイラ	CUDAコンパイラ
C	gcc	icc	nvc	nvcc
C++	g++	icpc	nvc++	
Fortran	gfortran	ifort	nvfortran	
OpenACC			○	

ジョブスクリプトサンプルの説明（Miyabi-G, MPIなし）

```
#!/bin/bash
#PBS -q lecture-g
#PBS -l select=1
#PBS -l walltime=00:01:00
#PBS -W group_list=gt00
#PBS -j oe

cd $PBS_O_WORKDIR
module load nvidia

./run
```

リソースグループ名： lecture-g
1ノード使用
制限時間1分（講習会キューでは最長10分）
利用グループ名： gt00
標準出力と標準エラー出力を統合

ジョブ投入時のディレクトリに移動

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！

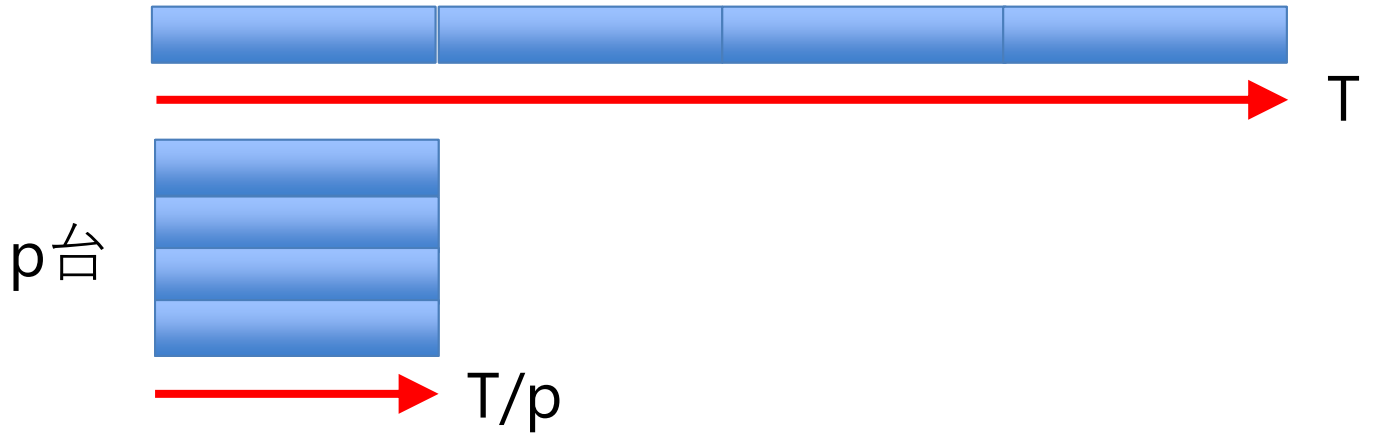
- GPUは**並列計算機**です！よって本講習会で学ぶのは**並列プログラミング**になります！
  - 並列プログラミングの例： MPI, OpenMP など
- 並列プログラミングは、**プログラムを高速化**するために行います！

並列プログラミングについての解説動画はこちら
https://www.youtube.com/channel/UC2CHaGp1AO-vqRlV7wmU0-w/videos?view=0&sort=p&flow=grid

並列プログラミング・高性能計算についての事前知識があると有利！

並列計算

- 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。

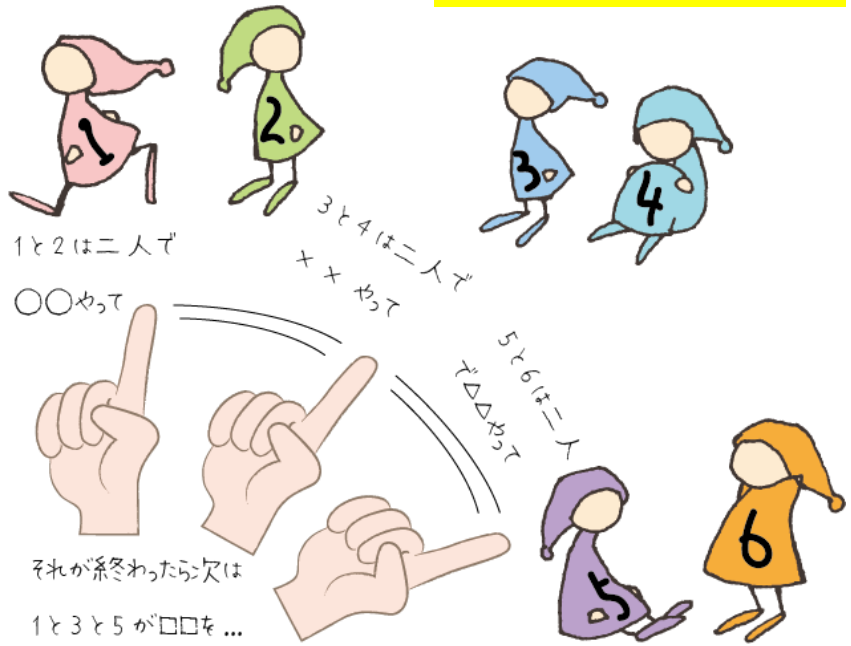


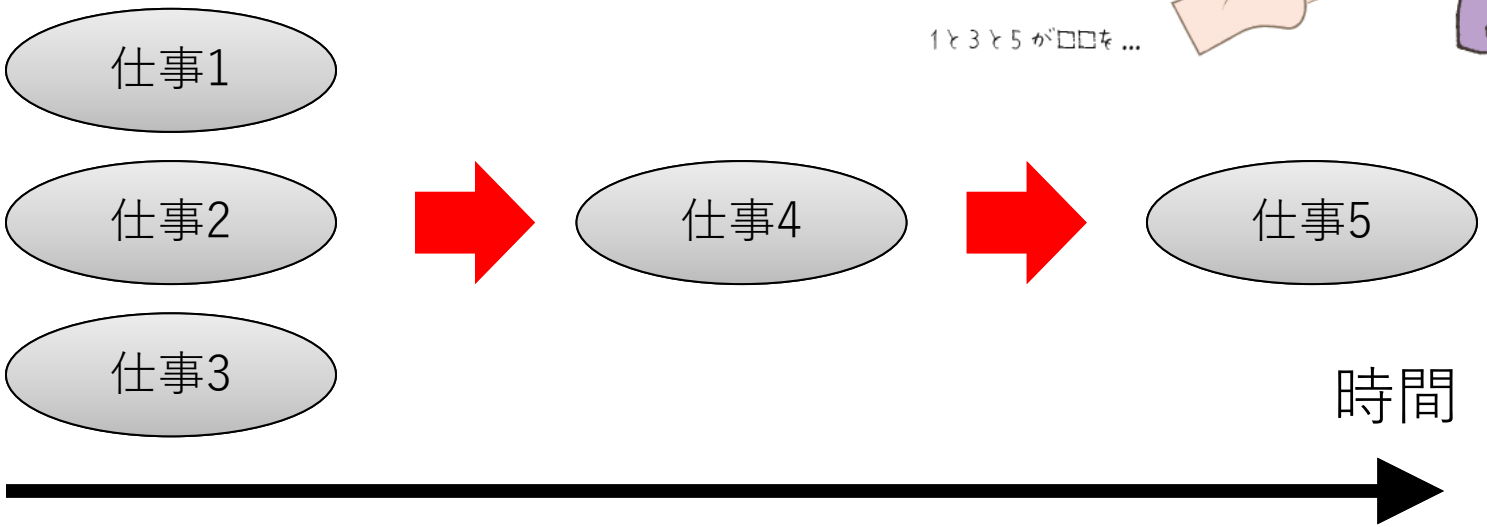
- 実際にできるかどうかは、処理内容（アルゴリズム）による。アルゴリズムによって難易度は異なる。
  - ✓ 並列化できないアルゴリズム、通信のオーバーヘッド
  - ✓ 部分的にでも並列化できないアルゴリズムがあると、どれだけ並列数を上げても、その時間は短縮されない。
- 並列処理（計算）の種類
  - ✓ 「タスク並列」と「データ並列」

タスク並列

- タスク（仕事）を分割することで並列化する。
- 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：野菜と肉を入れて煮込む
  - ✓ 仕事５：カレーのルウを入れる
- 並列化

GPUは苦手

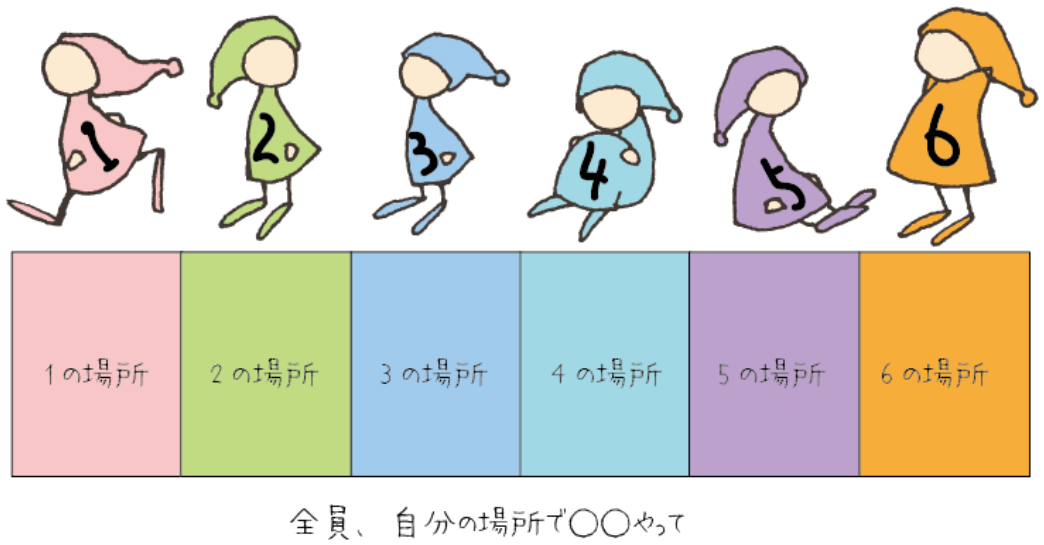




データ並列

- 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。

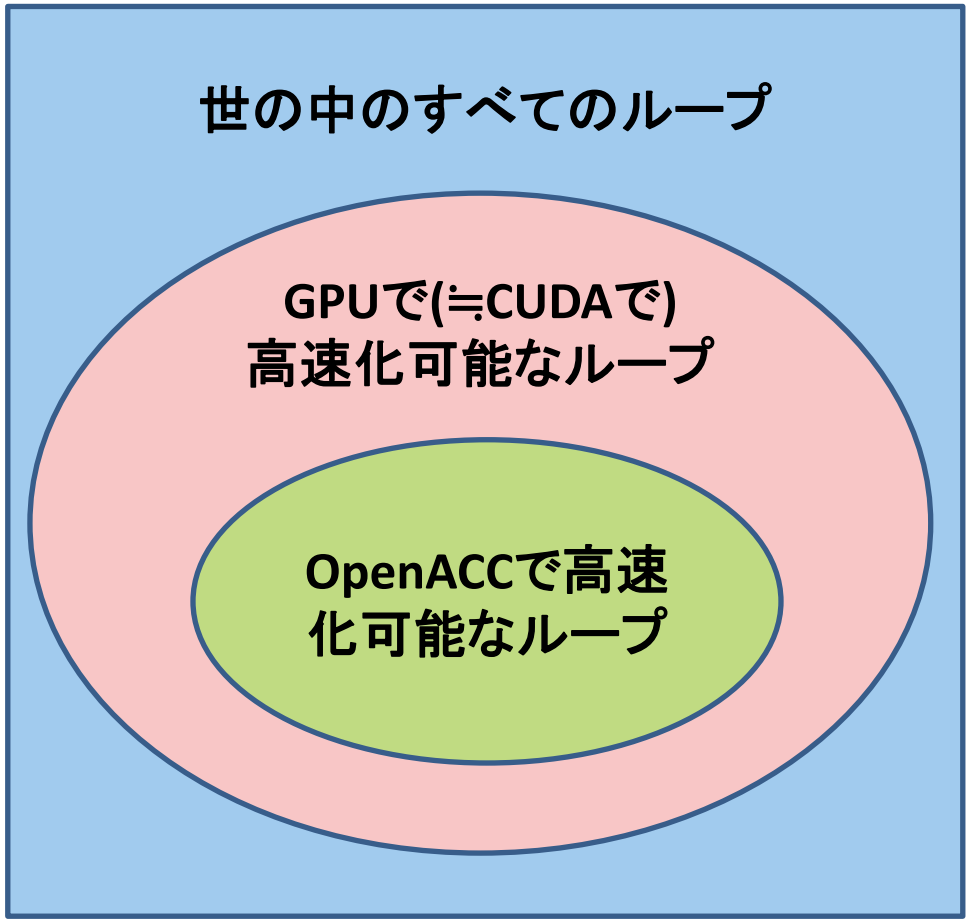
2 + 1	12 + (–8)
3 + 19	-20 + 29
4 + (-6)	4 + 10
-8 + 10	-32 + 12
10 + 3	-5 + 5



GPUの並列計算はこれが原則。
プログラムでは普通、配列とループで記述する
for (i = 0; i < N; i++) C[i] = A[i] + B[i];

GPUにおけるループ並列化

- GPU における高速化は通常、プログラム中の重たいループ構造を並列化することで達成する
- 今回学ぶOpenACC は**特定のループ構造を簡単に並列化**できる
  - 全てのループ構造を並列化できるわけではない
  - どのようなループなら並列化可能か知る必要がある



OpenACCで並列化できるループ

データ独立なループの例

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

リダクションの必要なループの例

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

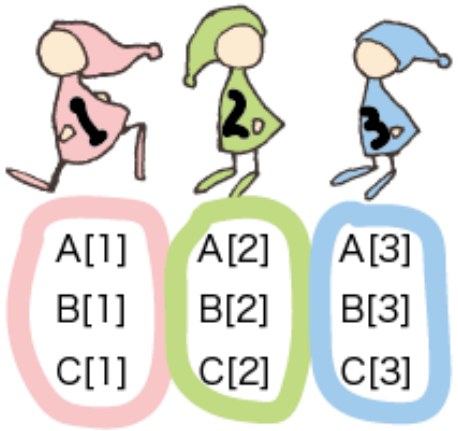
データ依存のあるループの例

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

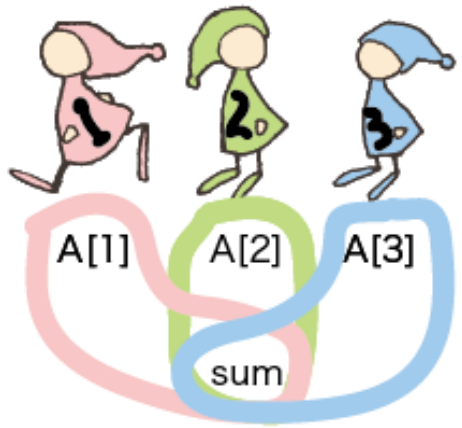
※OpenACCでも、GPUで正しく動くコードを書くことはできる。しかし遅いので意味がない

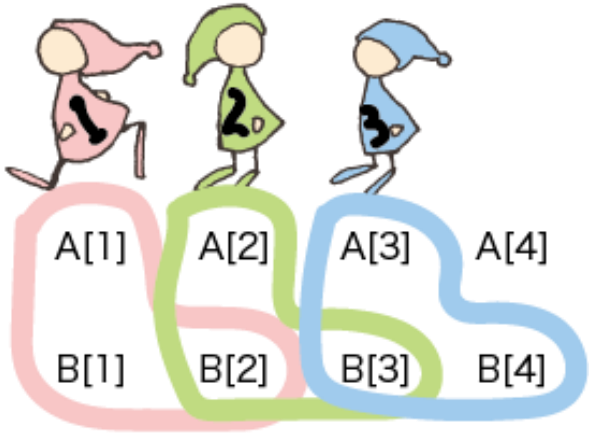
OpenACCで簡単に並列化できる

CUDAでも比較的簡単に並列化できる



CUDAで高速実装可能だが難しい (shared memory や warp shuffle を駆使する必要がある)





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]

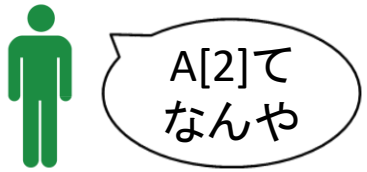
簡単に並列化できるループ

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

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

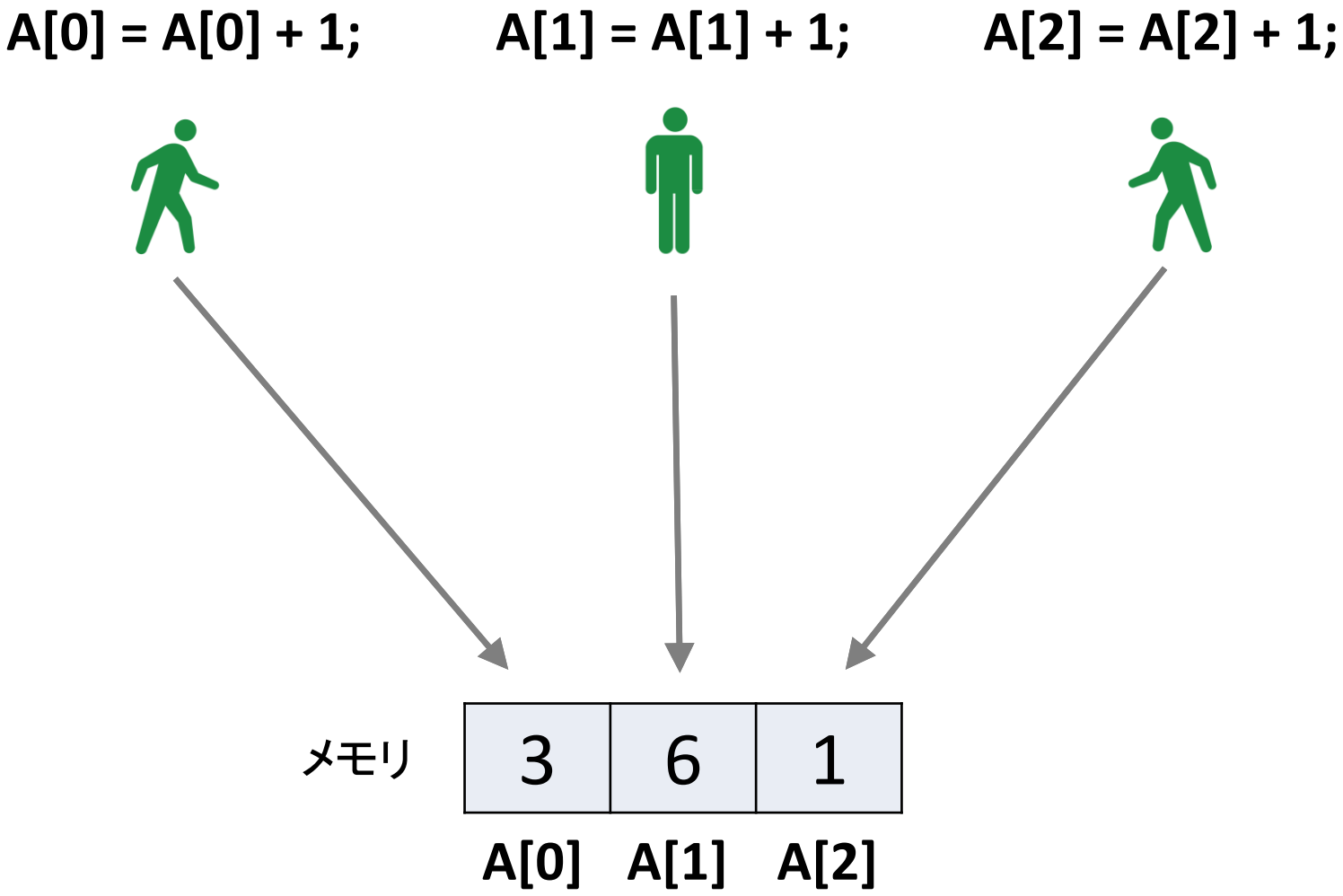
A[1] = A[1] + 1;

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



簡単に並列化できるループ

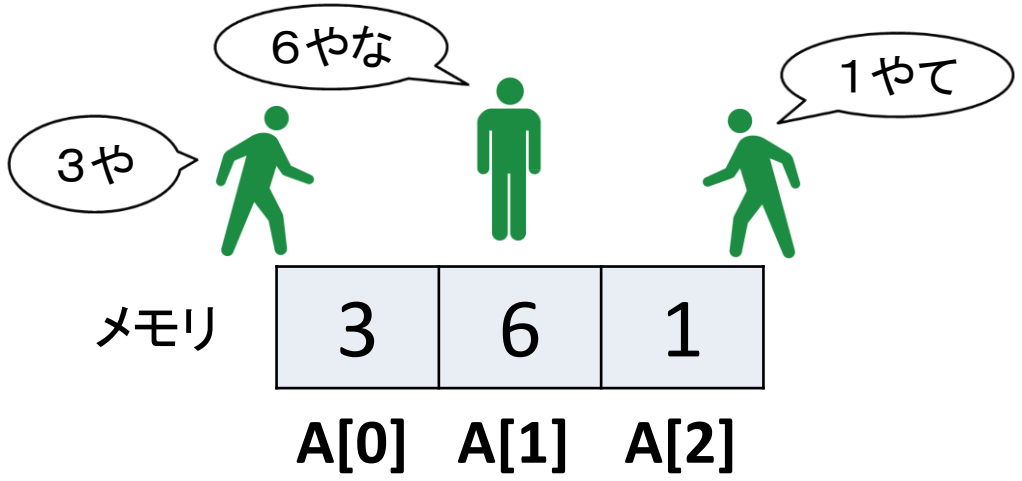
```
for ( i = 0; i < 3; i++)
  A[i] = A[i] + 1;
```



簡単に並列化できるループ

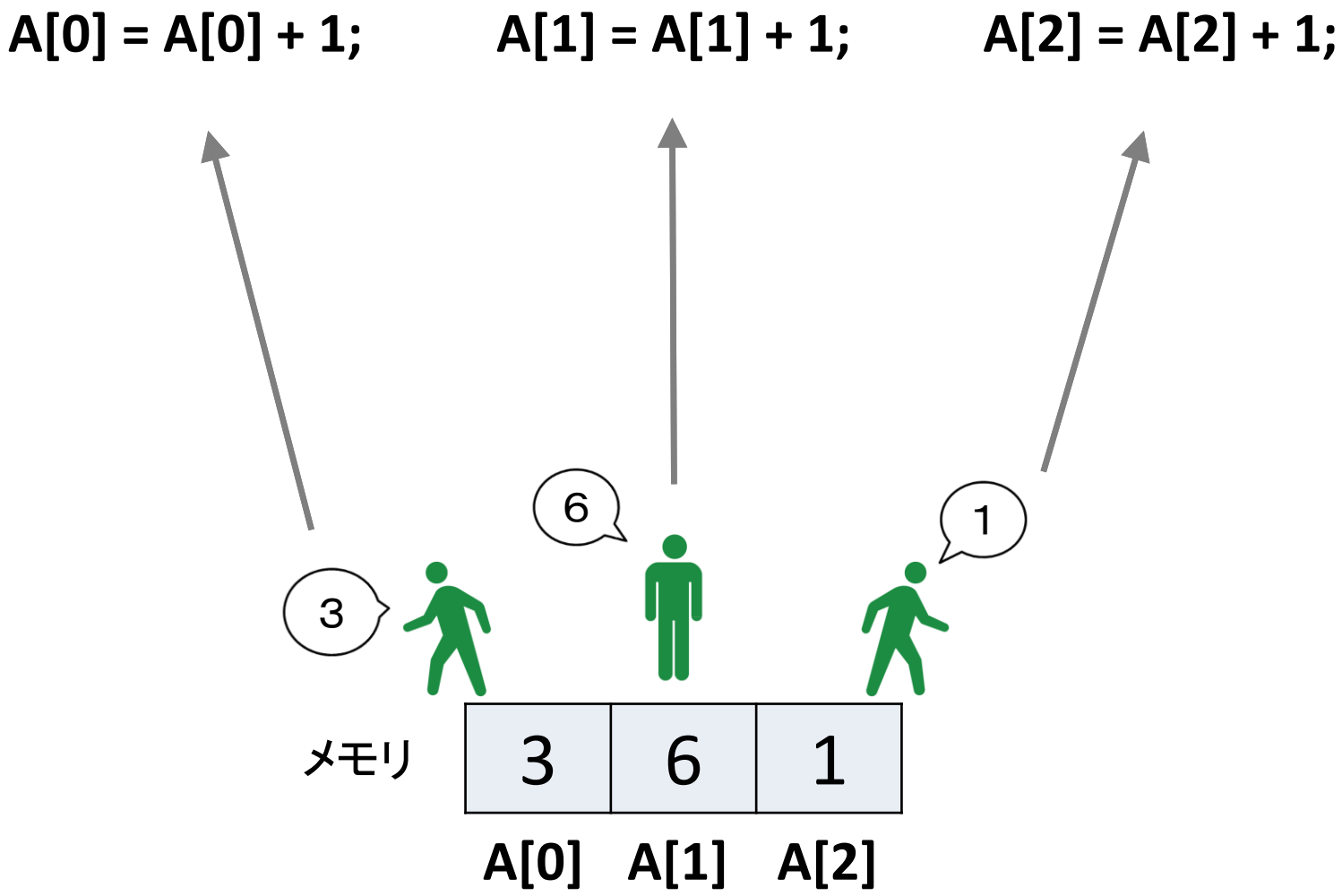
```
for ( i = 0; i < 3; i++)
  A[i] = A[i] + 1;
```

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



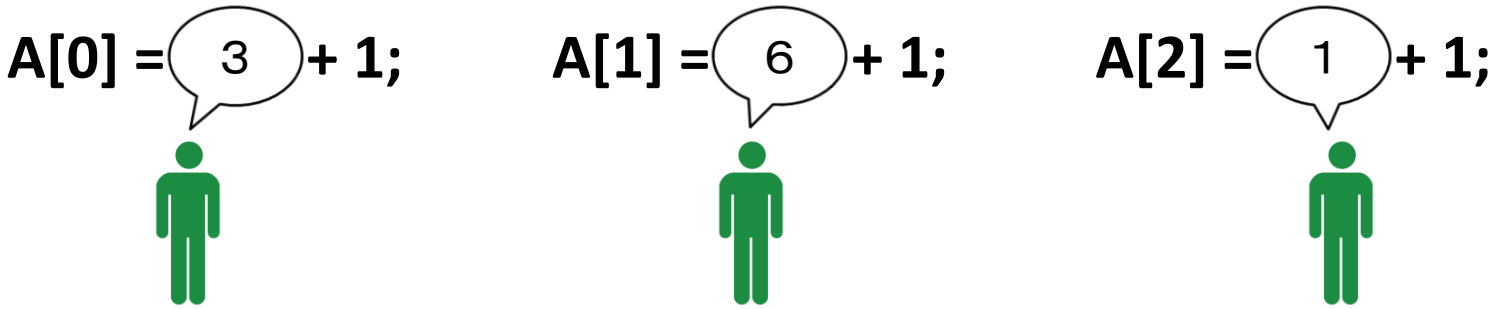
簡単に並列化できるループ

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



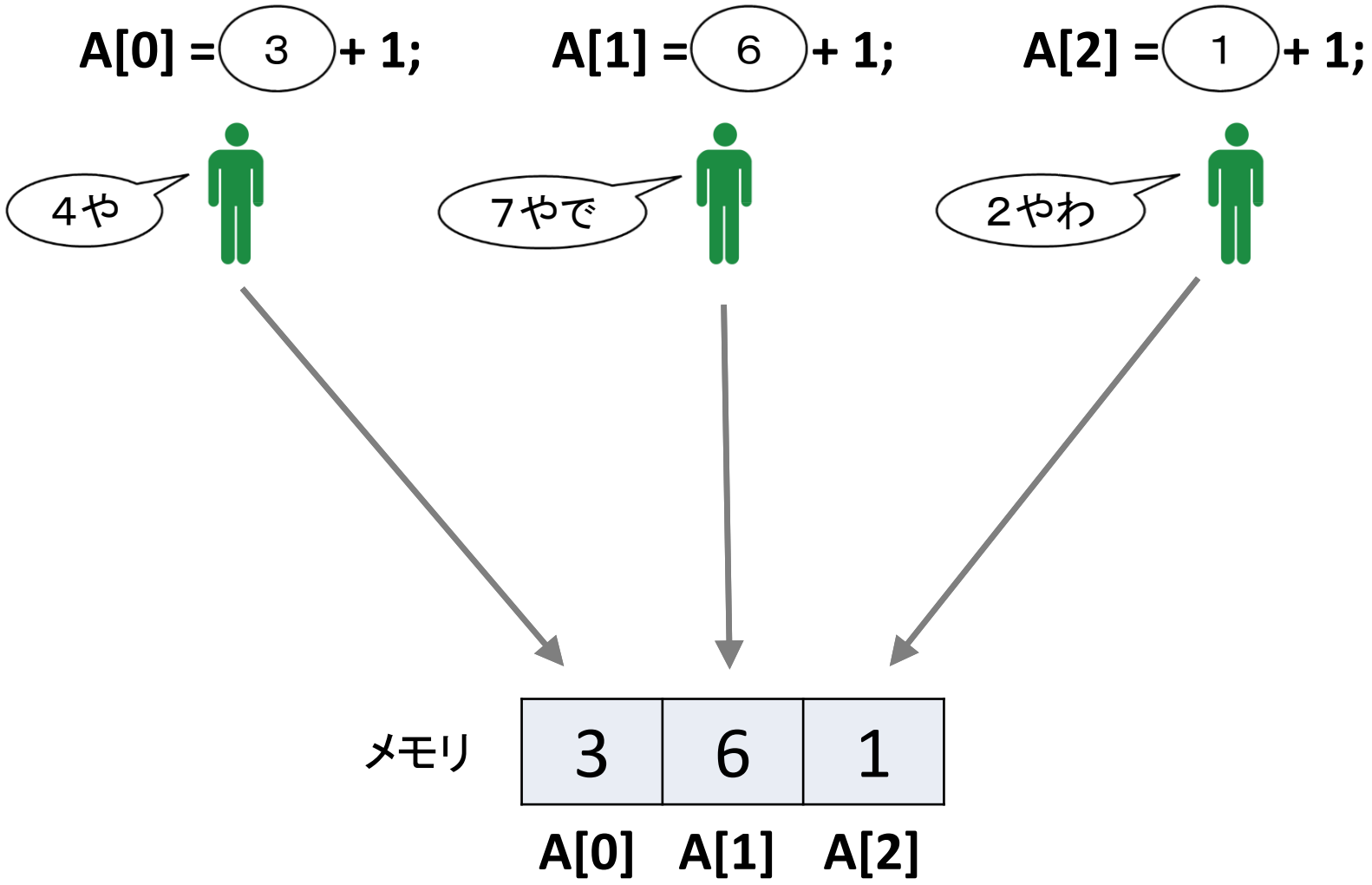
簡単に並列化できるループ

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



簡単に並列化できるループ

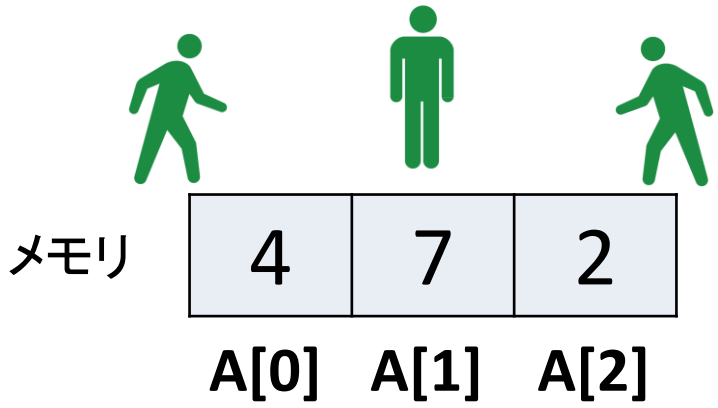
```
for ( i = 0; i < 3; i++)
  A[i] = A[i] + 1;
```



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

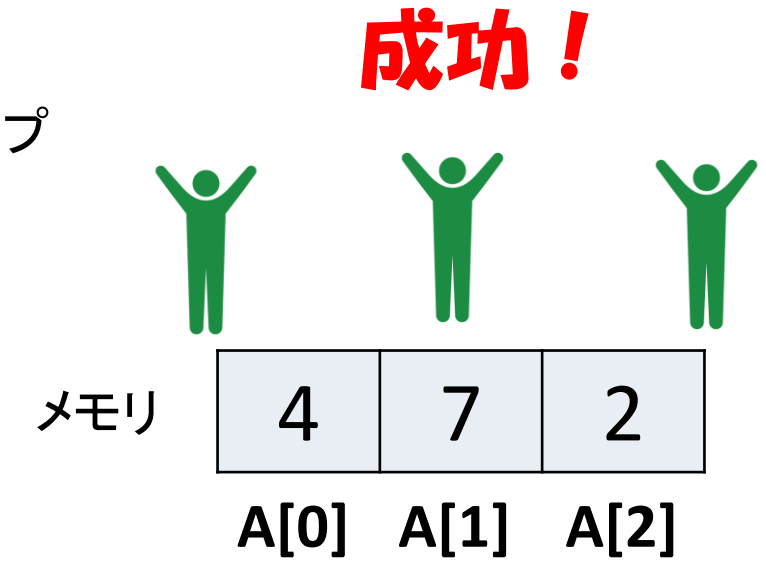
```
for ( i = 0; i < 3; i++)
    A[i] = A[i] + 1;
```

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(**independent**)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



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…

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;

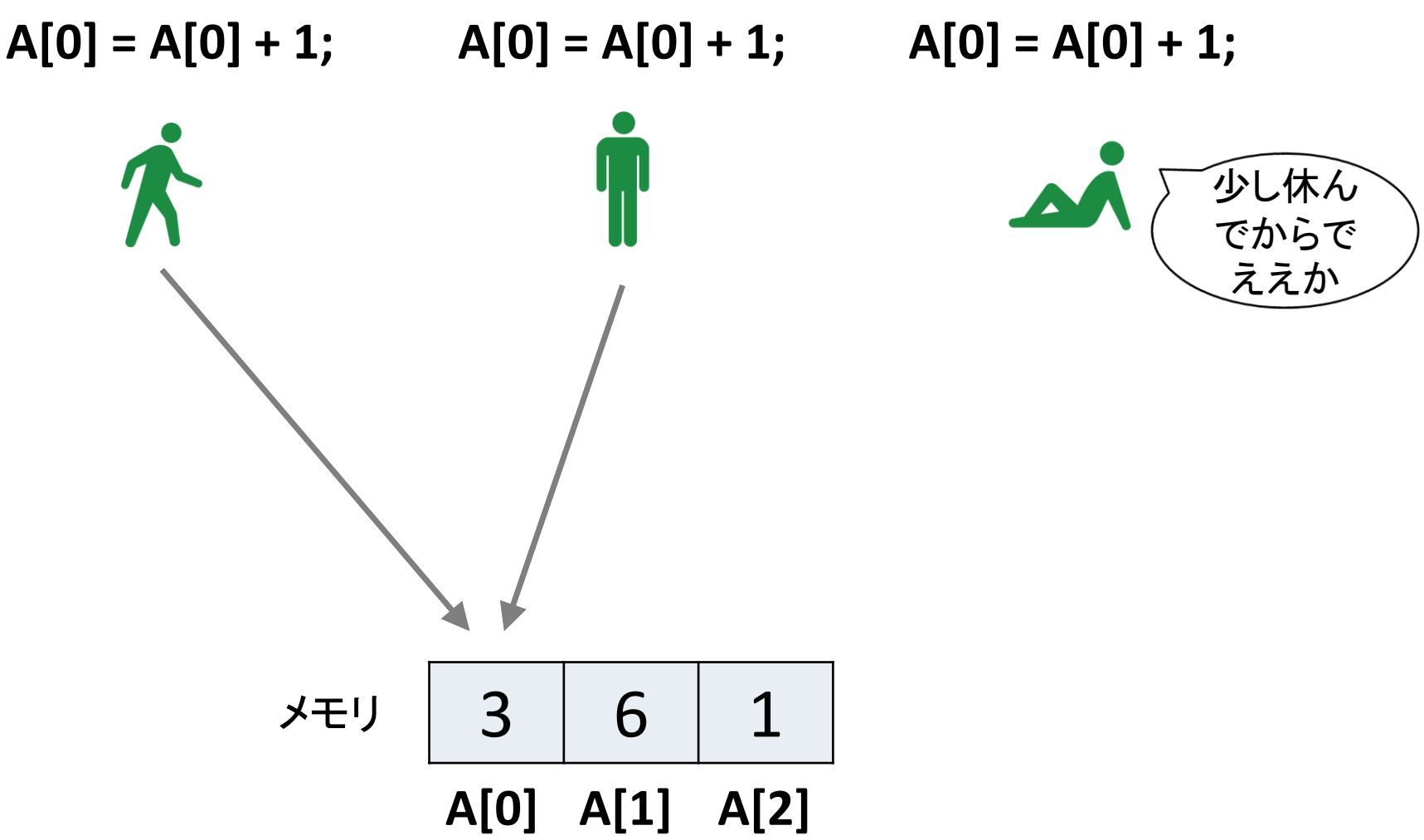
A[0] = A[0] + 1;

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

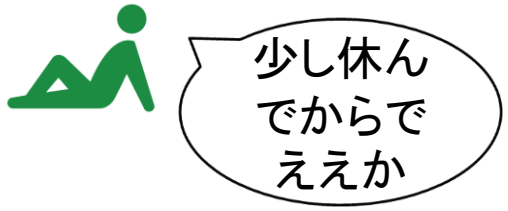
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

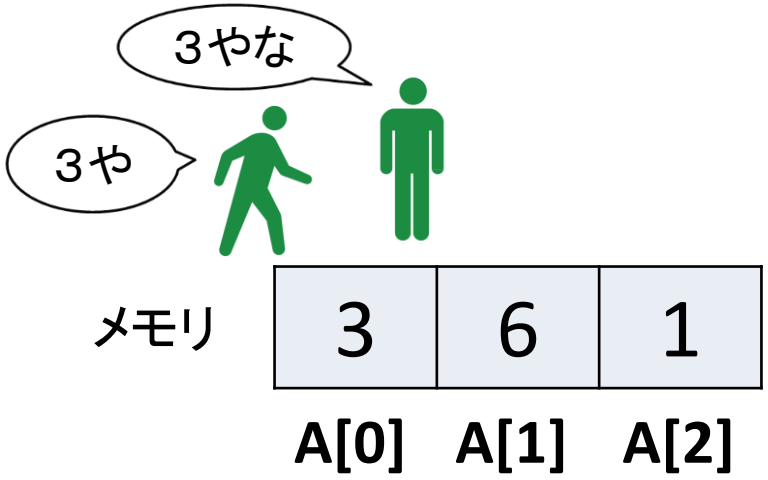


簡単に並列化できないループ

```
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;**

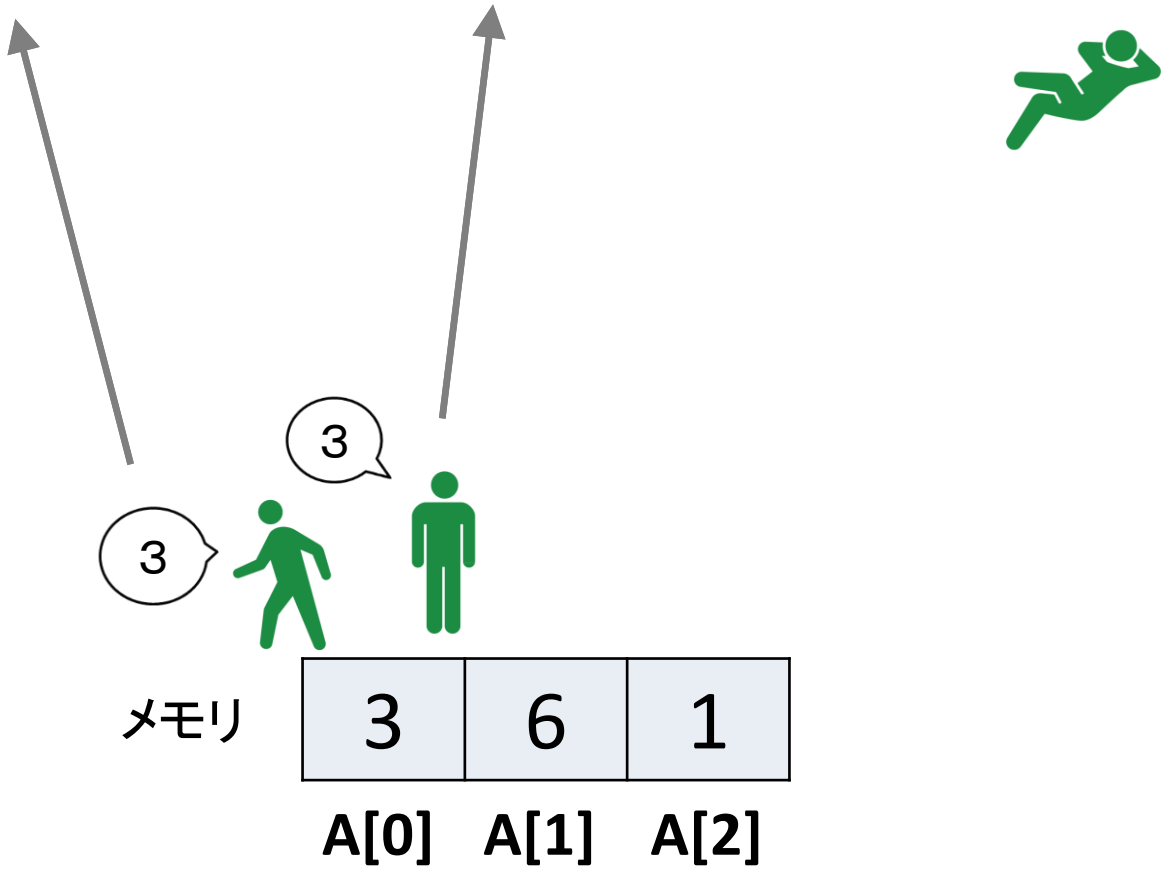




簡単に並列化できないループ

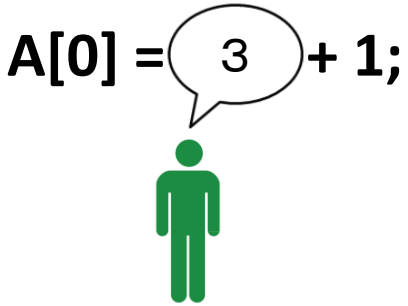
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

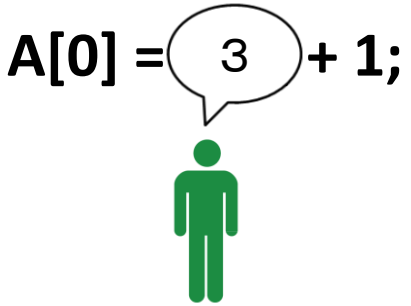
A[0] = A[0] + 1;　　A[0] = A[0] + 1;　　A[0] = A[0] + 1;

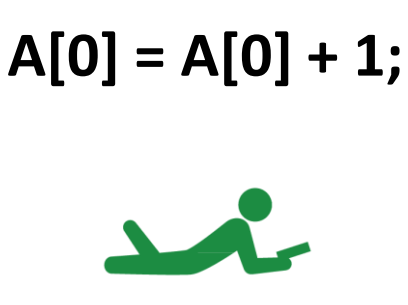


簡単に並列化できないループ

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

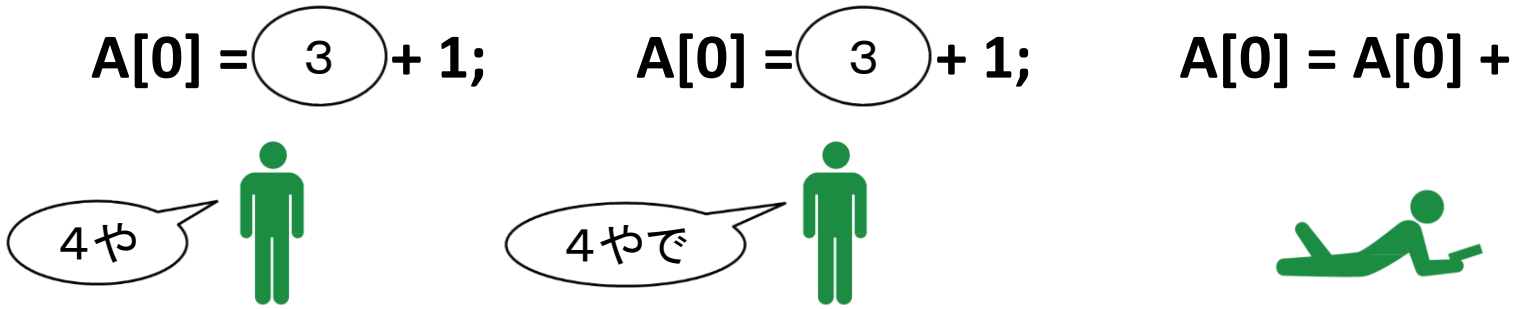






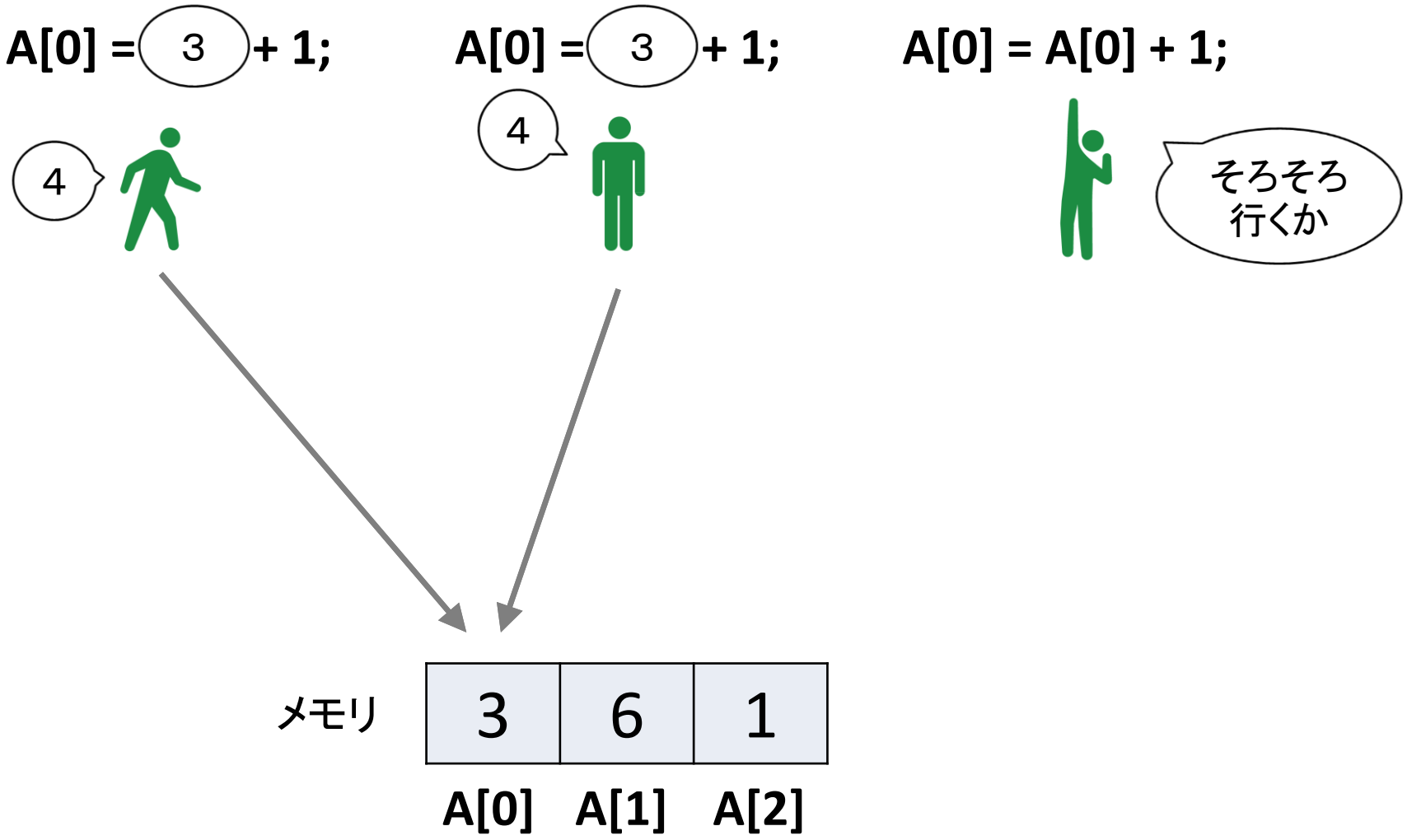
簡単に並列化できないループ

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



簡単に並列化できないループ

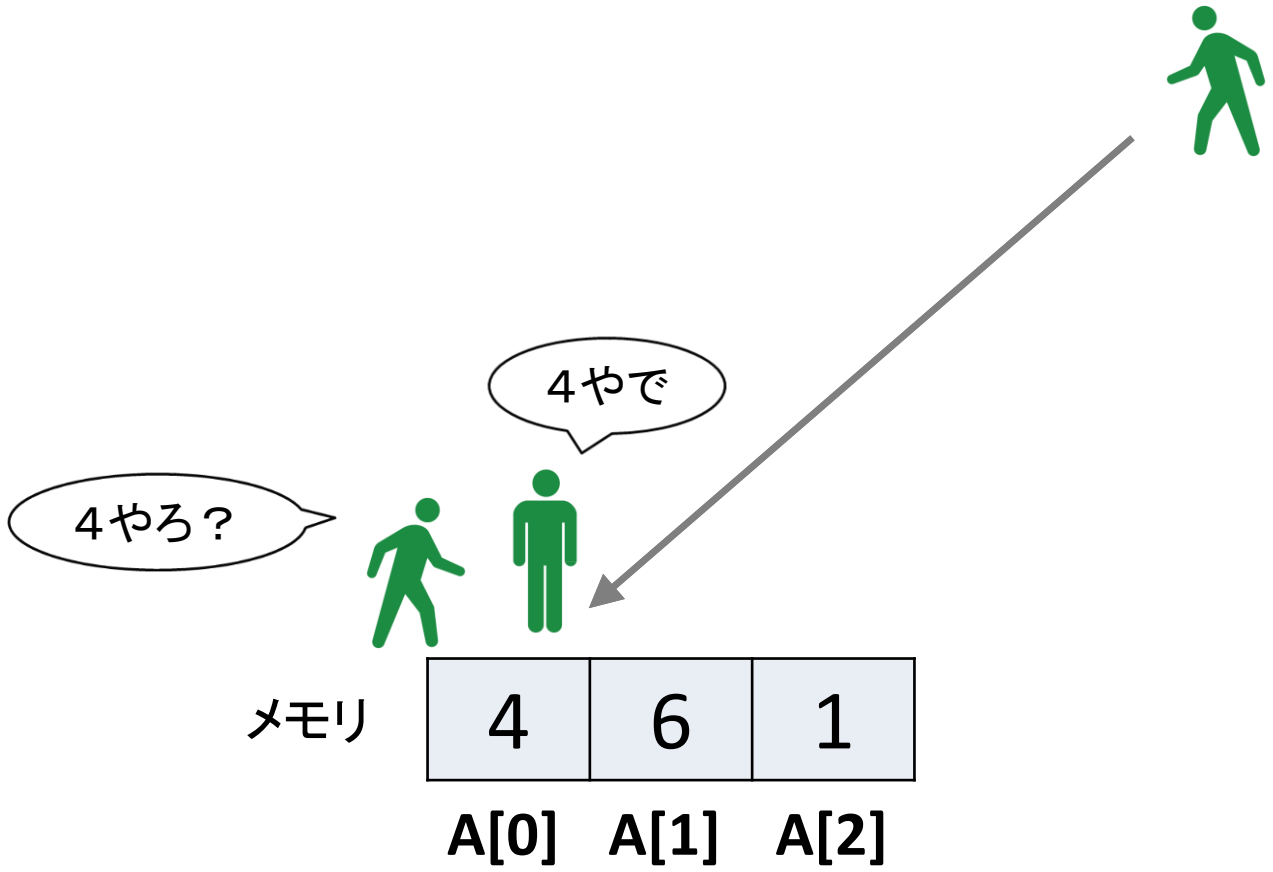
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```



簡単に並列化できないループ

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

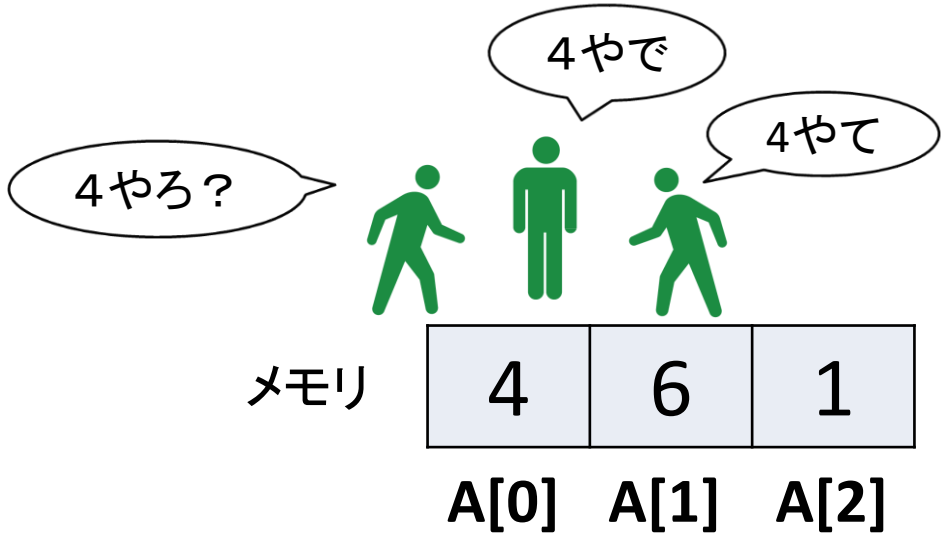
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

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

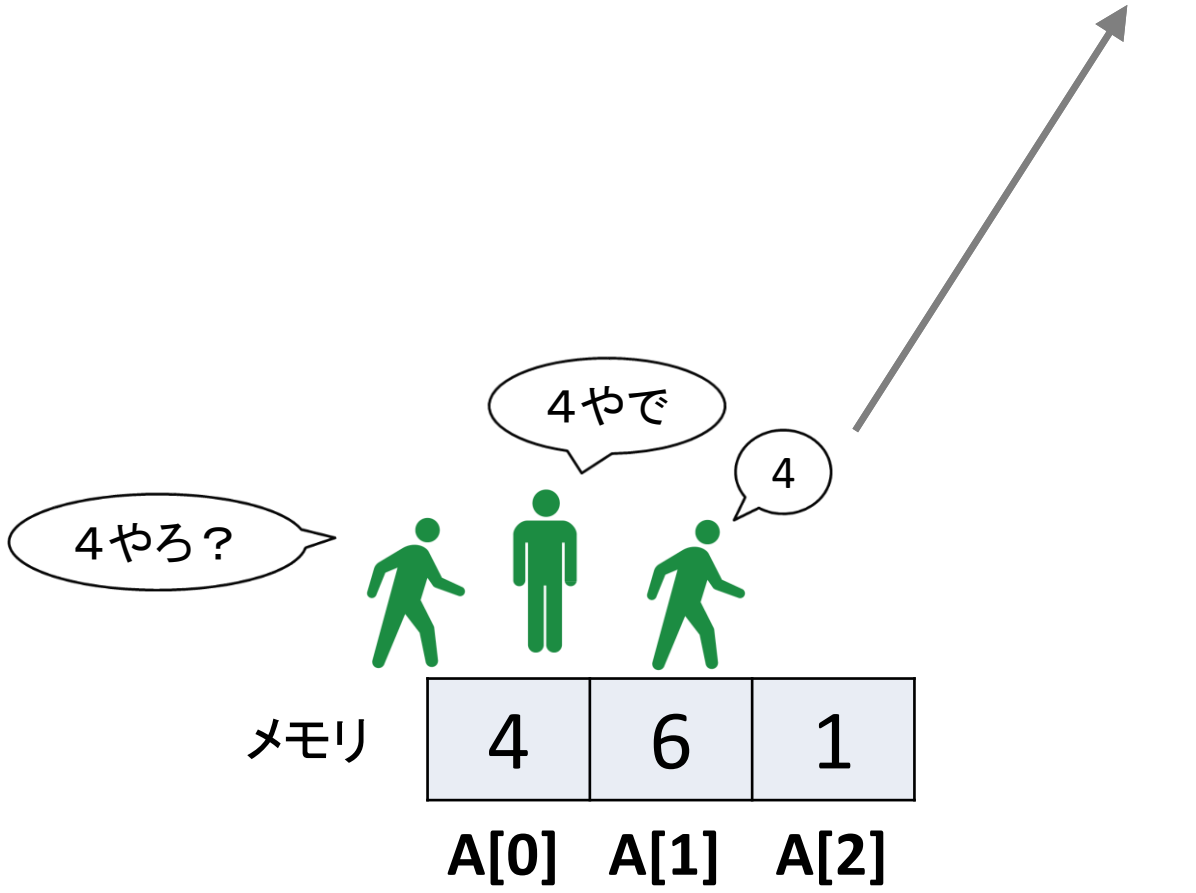
A[0] = (3) + 1;　　A[0] = (3) + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

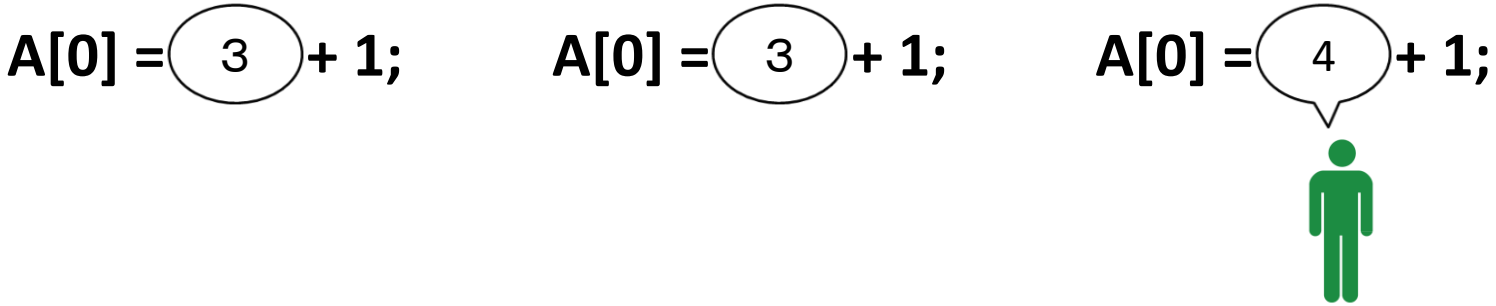
```
for ( i = 0; i < 3; i++)
    A[0] = A[0] + 1;
```

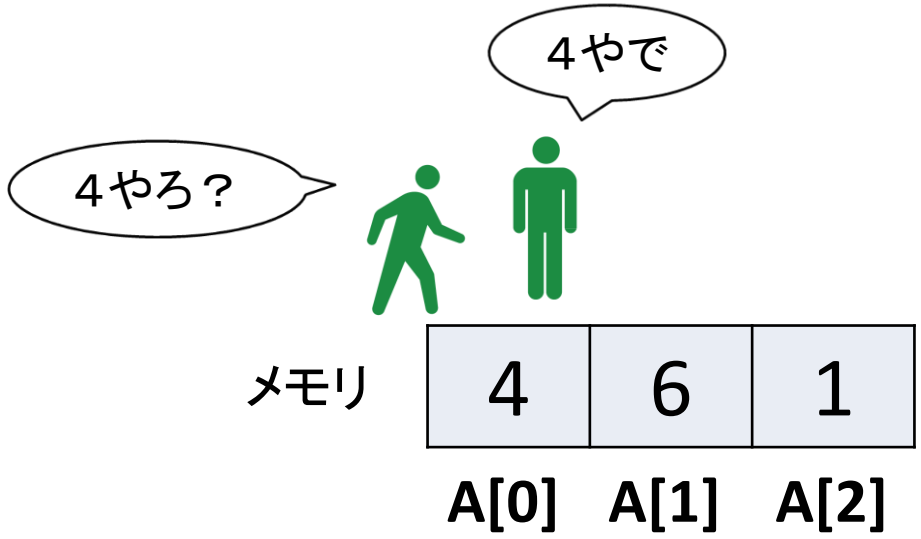
A[0] = ３ + 1;　　A[0] = ３ + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

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

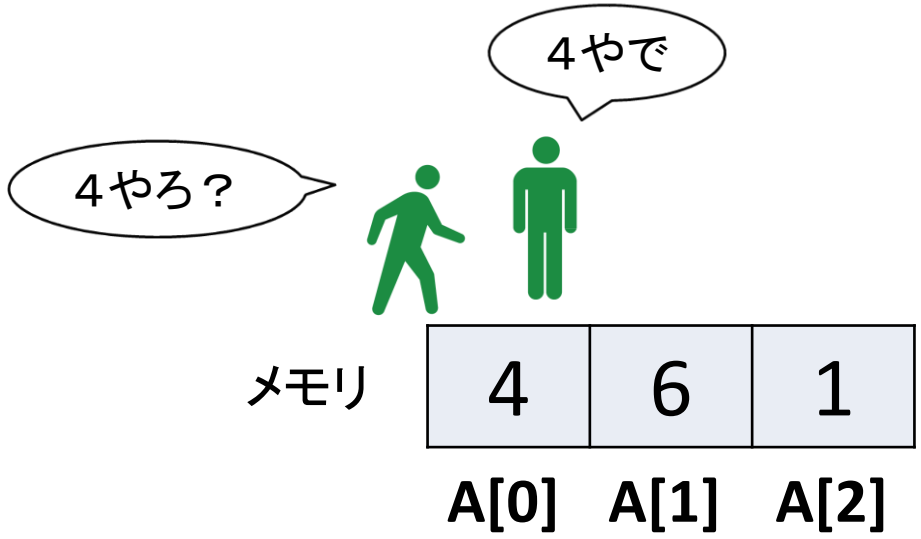




簡単に並列化できないループ

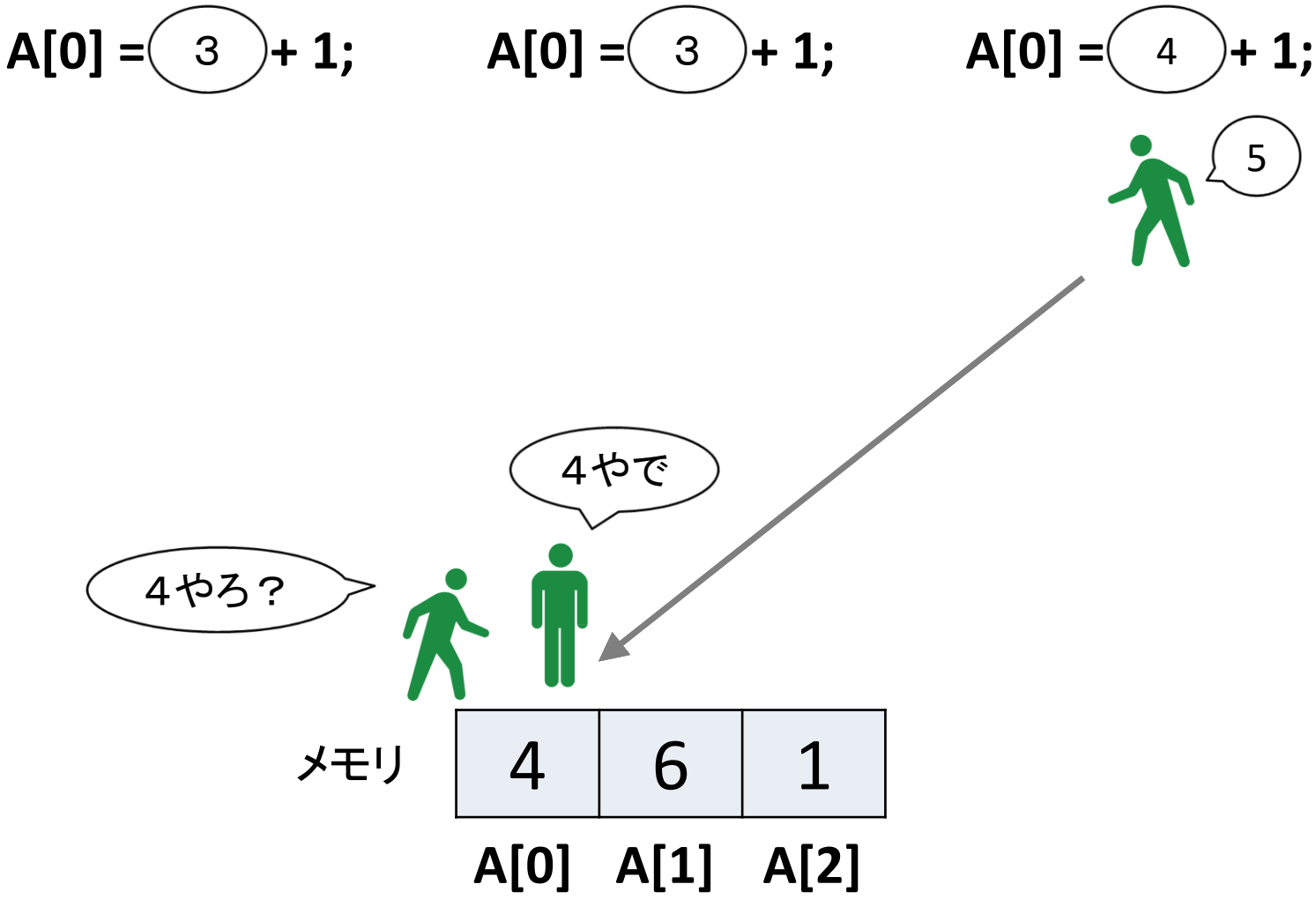
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

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

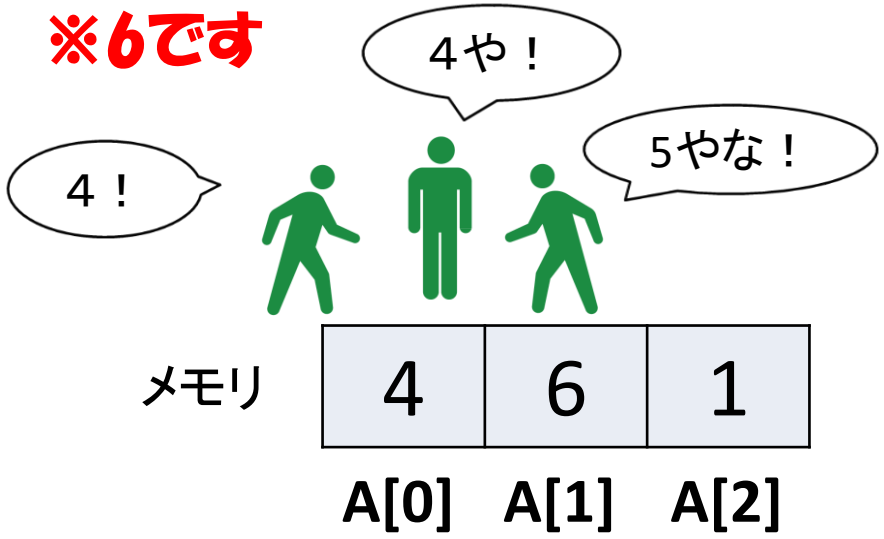
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```



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;



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。

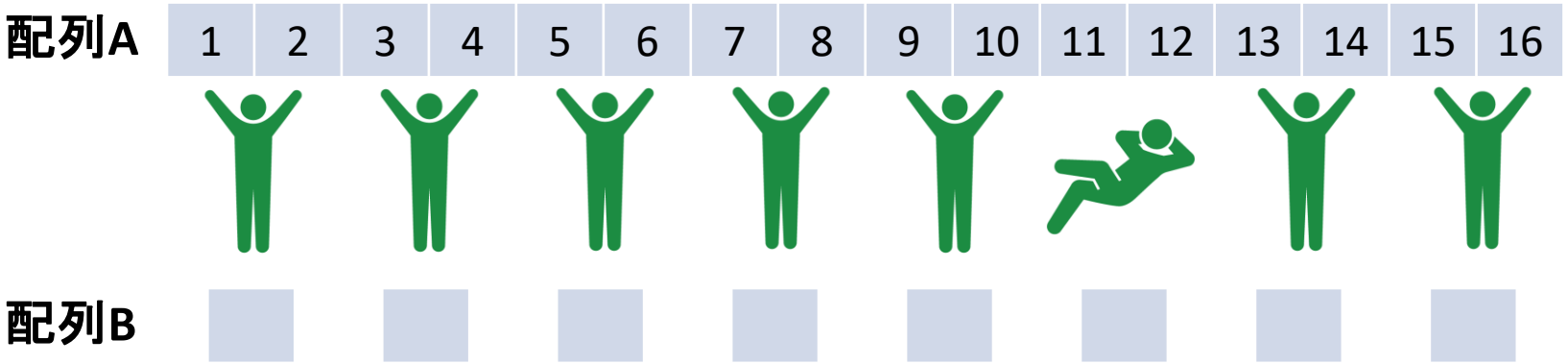
スレッドは各々独立に１～３を実行するため、タイミングによって結果が変わる！

（この例の場合は4,5,6のいずれかになる）

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

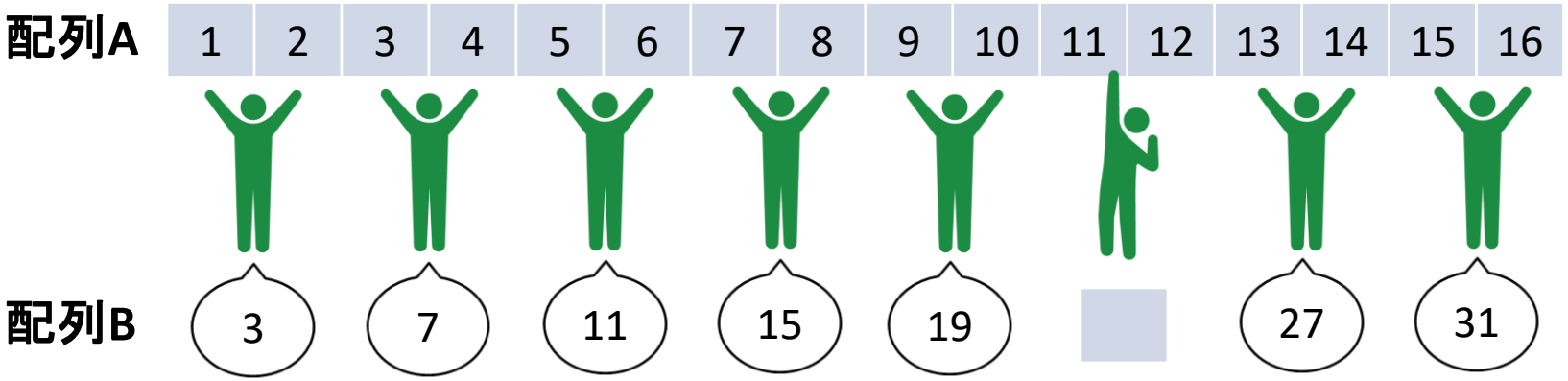
```
sum = 0;
for ( i = 0; i < 16; i++)
    sum = sum + A[i];
```



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];
```

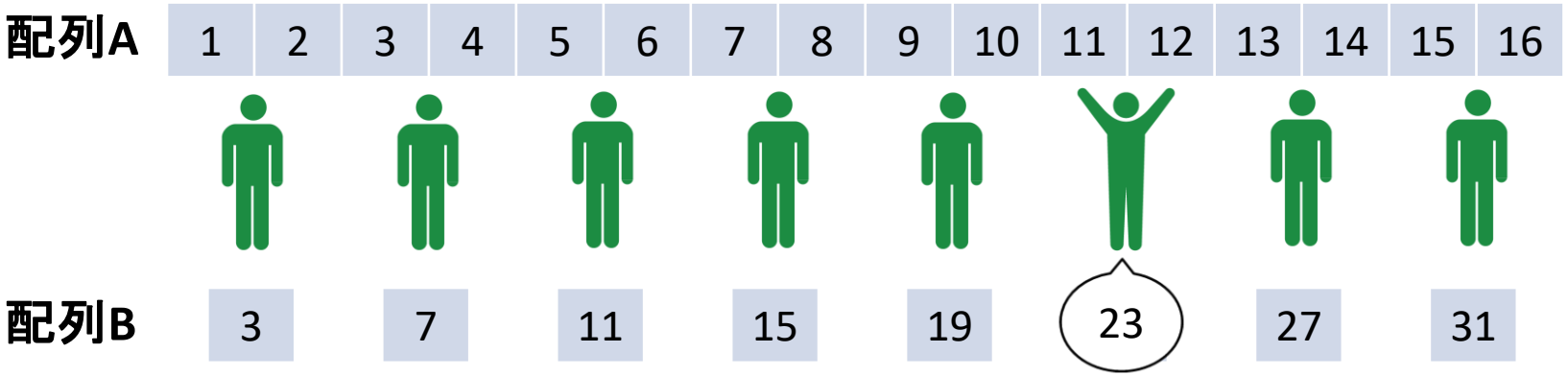


1. 各々自分の担当領域で足し算（結果を別の場所に保存）

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];
```

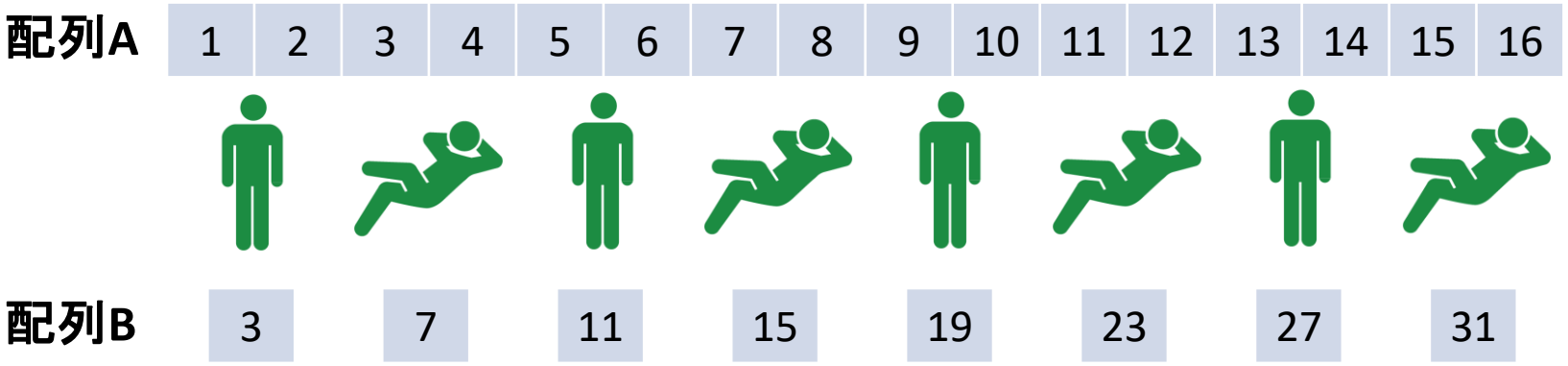


1. 各々自分の担当領域で足し算（結果を別の場所に保存）
2. 遅れているスレッドを待つ！（これを**同期(thread synchronization)**という）

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];
```



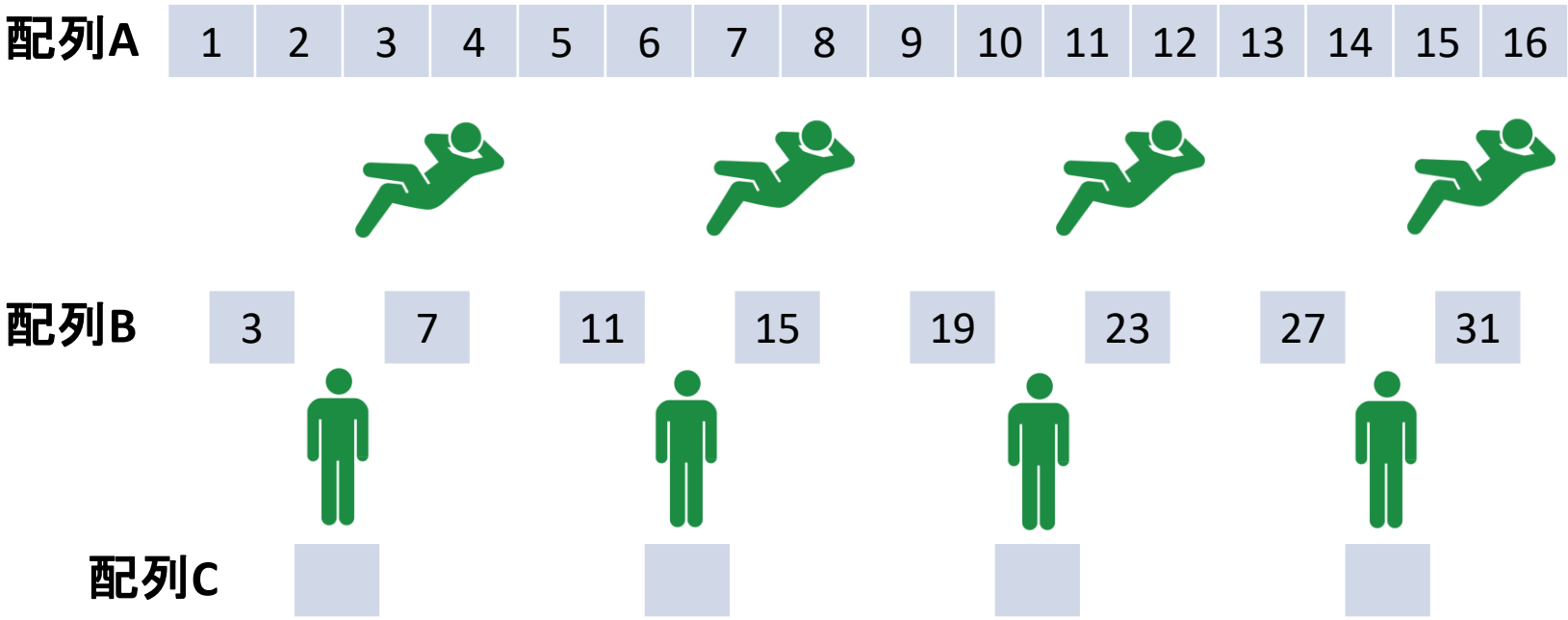
配列C

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];
```

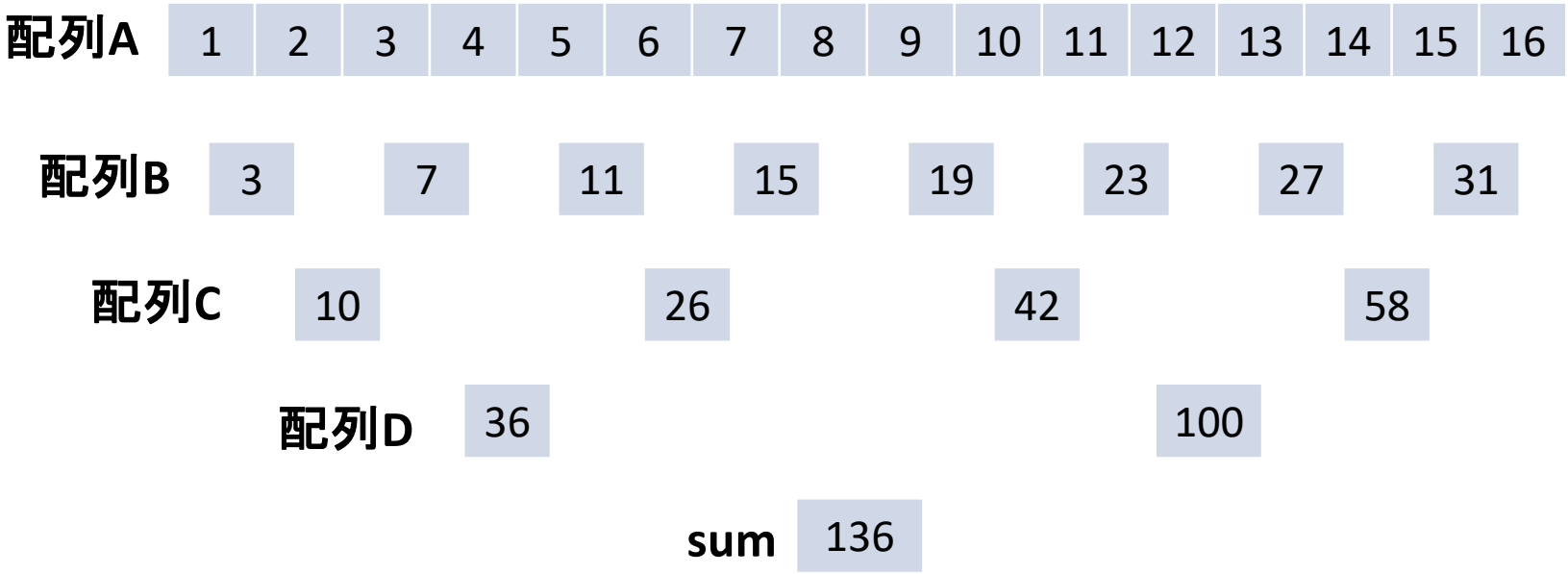


1. 各々自分の担当領域で足し算（結果を別の場所に保存）
2. 遅れているスレッドを待つ！（これを**同期(thread synchronization)**という）
3. 一部のスレッドを寝かせて、起きてるスレッドで(1)から繰り返し

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];
```



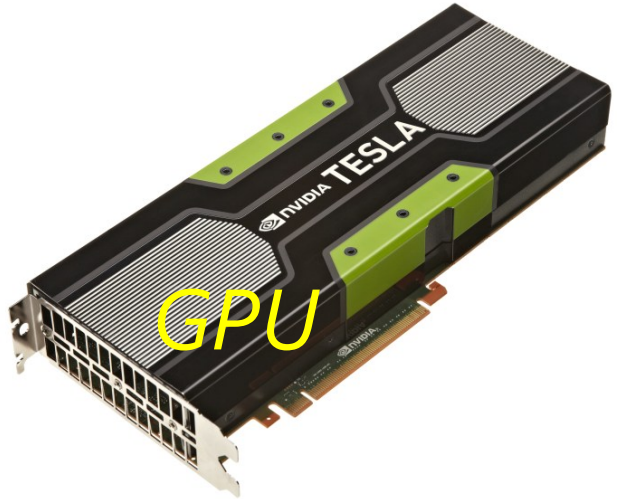
- これは一般的にリダクションと呼ばれる演算パターン
- 一番働くスレッドが4回の足し算。逐次の場合と比較して4倍の高速化
- メモリなどを介してスレッドの間でデータのやり取りをすることをスレッド間通信という

スレッドの同期・通信が入ると途端に難しくなる！

GPU入門

What's GPU ?

- Graphics Processing Unit
- もともと PC の3D描画専用の装置
- 昔は安価だったが、今のスパコン用製品は高性能化やAI需要増で非常に高価



http://www.nvidia.co.jp

GPUコンピューティング

- GPUはもともと、グラフィックスやゲームの画像計算のために発展した。
- CPUがコア数が数個～数十個程度に対し、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
- 最近は、ディープラーニング（深層学習）、機械学習、AI（人工知能）などでの利用が急増

抑えておくべきGPUの特徴

- 最低限知っておくべきこと
  - ✓ 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
    - プログラムの並列性（スレッド分割可能数）が小さいと速くならない
  - ✓ 多くの場合、CPU と GPUの間でのデータ転送が必要！
    - GPU は CPU の指示なしでは動けない
    - CPU と GPU は独立に動く
    - CPUとGPUは別々のメモリを読み書きする（一部例外あり）
      - ✓ CPUとGPUの同期を行い、データの一貫性を保つ必要がある

- さらなる高速化のためには
  - 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

これらはプログラミング言語が CUDA か OpenACCに関わらず、GPUプログラミングでは考慮する必要がある。

NVIDIA H100 Tensor Core GPU (1/2)

- 132 SM (Streaming Multiprocessor)

出典: NVIDIA H100 Tensor core GPUアーキテクチャ

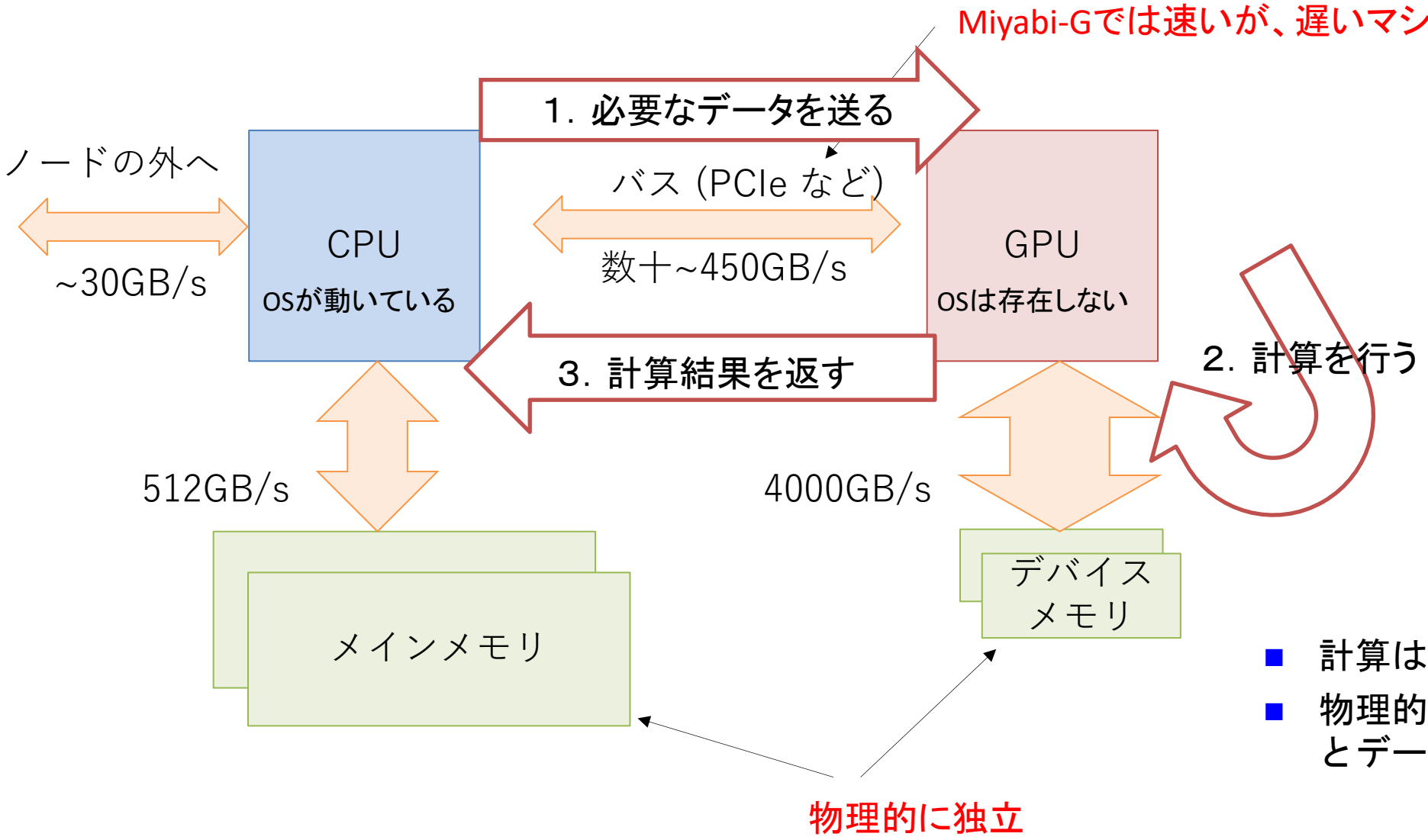
NVIDIA H100 Tensor Core GPU (2/2)

- 各SMに多数の演算コアが搭載されている
  - Tensor Coreは行列計算用で、CUDA専用
  - 低精度ほど、多量の演算を同時実行可能

- メモリの容量・速度は製品によって様々
  - H100：容量 80GB、バンド幅 2～3.35 TB/s
  - MiyabiのGH200：96GB、4 TB/s
  - GH200 144GB：4.9 TB/s
  - H200：141GB、4.8 TB/s

出典: NVIDIA H100 Tensor core GPUアーキテクチャ

CPUと独立のGPUメモリ



- 計算はOSのあるCPUから始まる
- 物理的に独立のデバイスメモリとデータのやり取り必須

どんなアプリなら、幅広いGPUで高速化できる？

- 原則：GPUに一度送ったデータを使い回せるアプリケーション
  - 最低でも100回は使いまわしたい
  - 例：データ量 N に対して計算量 $O(N^2)$ 以上の計算（行列積、多体問題など）や、反復法など
- 思考実験
  - 次のプログラムを、右の表のコンピュータの

(1) CPUを使った時の実行時間は？

(2) GPUを使った時の実行時間は？

あるコンピュータの性能

	データ転送性能
CPUのメモリ	100 GB/sec
GPUのメモリ	1000 GB/sec
CPU-GPU間のバス	20 GB/sec

```
double precision :: A(1:N), B(1:N)
if(GPU) BをCPUからGPUにコピー
do i = 1, N
      A(i) = B(i)
end do
if(GPU) AをGPUからCPUにコピー
```

どんなアプリなら、幅広いGPUで高速化できる？

- 原則：GPUに一度送ったデータを使い回せるアプリケーション
  - 最低でも100回は使いまわしたい
  - 例：データ量 N に対して計算量 $O(N^2)$ 以上の計算（行列積、多体問題など）や、反復法など
- 思考実験
  - 次のプログラムを、右の表のコンピュータの

(1) CPUを使った時の実行時間は？

(2) GPUを使った時の実行時間は？

あるコンピュータの性能

	データ転送性能
CPUのメモリ	100 GB/sec
GPUのメモリ	1000 GB/sec
CPU-GPU間のバス	20 GB/sec

```
double precision :: A(1:N), B(1:N)
if(GPU) BをCPUからGPUにコピー
do i = 1, N
      A(i) = B(i)
end do
if(GPU) AをGPUからCPUにコピー
```

(1)　配列A・Bのbyte数 / CPUのメモリ性能
= N * 2 * 8 / 100

(2)　配列A・Bのbyte数 / GPUのメモリ性能
+ 配列A・Bのbyte数 / CPU-GPU間バスのメモリ性能
= N * 2 * 8 / 1000 + N * 2 * 8 / 20

N = 10^9 (1G)　なら？

(1) 0.16 sec　(2) 0.816 sec

どんなアプリなら、幅広いGPUで高速化できる？

- 原則：GPUに一度送ったデータを使い回せるアプリケーション
  - 最低でも100回は使いまわしたい
  - 例：データ量 N に対して計算量 $O(N^2)$ 以上の計算（行列積、多体問題など）や、反復法など
- 思考実験
  - 次のプログラムを、右の表のコンピュータの

(1) CPUを使った時の実行時間は？

(2) GPUを使った時の実行時間は？

あるコンピュータの性能

	データ転送性能
CPUのメモリ	100 GB/sec
GPUのメモリ	1000 GB/sec
CPU-GPU間のバス	20 GB/sec

```
double precision :: A(1:N), B(1:N)
if(GPU) BをCPUからGPUにコピー
do t = 1, 100
      do i = 1, N
            A(i) = B(i)
      end do
end do
if(GPU) AをGPUからCPUにコピー
```

100回使い回してみる

どんなアプリなら、幅広いGPUで高速化できる？

- 原則：GPUに一度送ったデータを使い回せるアプリケーション
  - 最低でも100回は使いまわしたい
  - 例：データ量 N に対して計算量 $O(N^2)$ 以上の計算（行列積、多体問題など）や、反復法など
- 思考実験
  - 次のプログラムを、右の表のコンピュータの

(1) CPUを使った時の実行時間は？

(2) GPUを使った時の実行時間は？

あるコンピュータの性能

	データ転送性能
CPUのメモリ	100 GB/sec
GPUのメモリ	1000 GB/sec
CPU-GPU間のバス	20 GB/sec

```
double precision :: A(1:N), B(1:N)
if(GPU) BをCPUからGPUにコピー
do t = 1, 100
      do i = 1, N
            A(i) = B(i)
      end do
end do
if(GPU) AをGPUからCPUにコピー
```

100回使い回してみる

(1) 100 * 配列A・Bのbyte数 / CPUのメモリ性能
= 100 * N * 2 * 8 / 100

(2) 100 * 配列A・Bのbyte数 / GPUのメモリ性能
+ 配列A・Bのbyte数 / CPU-GPU間バスのメモリ性能
= 100 * N * 2 * 8 / 1000 + N * 2 * 8 / 20

$N = 10^9$ (1G) なら？

(1) 16 sec (2) 2.4 sec

どんなアプリなら、幅広いGPUで高速化できる？

- 原則：GPUに一度送ったデータを使い回せるアプリケーション
  - CPU-GPU通信が遅いマシンでは、まとまった範囲の処理をまずGPU化しないと、転送時間のせいで性能がガタ落ちしてしまう
  - Miyabi-GではCPU-GPU通信がかなり速いため、既存プログラムを少しずつGPU化していっても、段階的に効果を実感しやすい

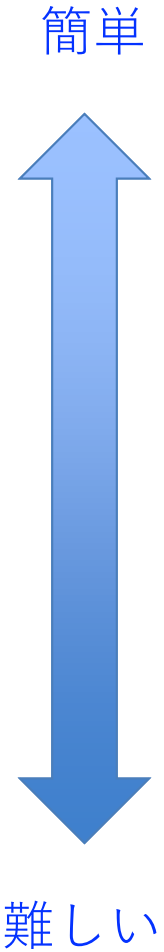
OPENACC入門

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。



OpenACC

- OpenACCとは
  - ✓ アクセラレータ（≒GPU）用プログラミングインターフェース
  - ✓ OpenMP のようなディレクティブ（指示文）・ベース
  - ✓ C 言語/C++, Fortran に対応
  - ✓ 2011年秋に OpenACC1.0、現在は 2.7が広く利用
  - ✓ コンパイラ：PGI → **NVIDIA HPC SDK (無料)**, Cray, GCC
  - ✓ WEBサイト：http://www.openacc.org/
- 指示文ベースの利点
  - ✓ 指示文：コンパイラへのヒント
  - ✓ アプリケーションの開発や移植が比較的簡単
  - ✓ ホスト（CPU）用コード、複数のアクセラレータ用コードを単一コードとして記述。メンテナンスが容易。高生産性。

C言語

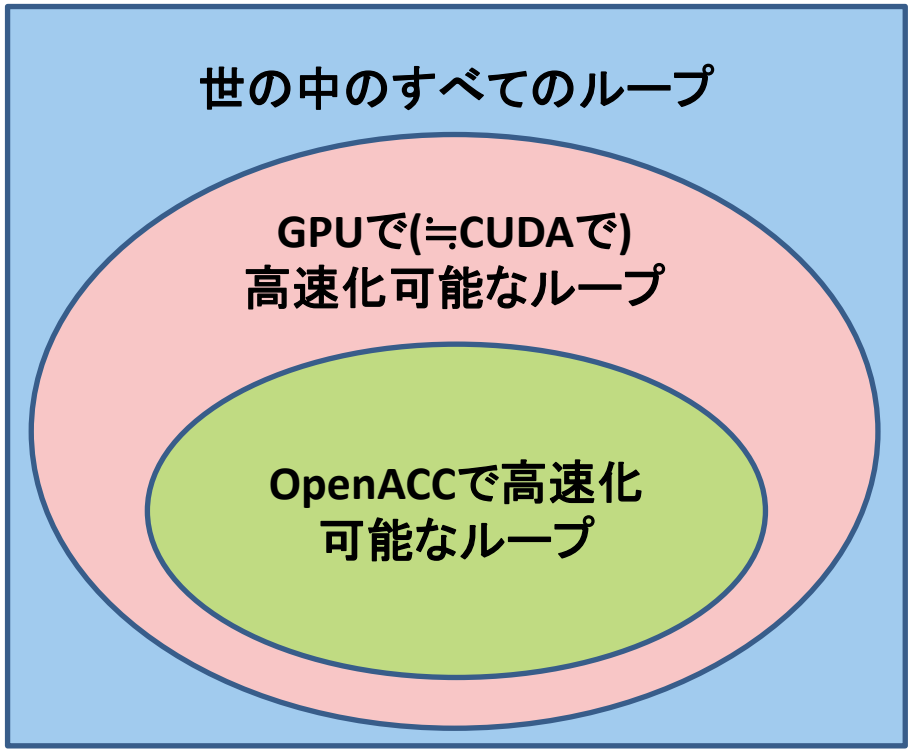
```
#pragma acc directive-name [clause, …]
{
   // C code
}
```

Fortran

```
!$acc directive-name [clause, …]
   ! Fortran code
!$acc end directive-name
```

OpenACCでできること

- OpenACC は**特定のループ構造を簡単に並列化**できる
  - ✓ 全てのループ構造を並列化できるわけではない
- 主に以下の３つを記述できる
  - ✓ どこを GPU で実行するか
  - ✓ どこでデータを移動するか
  - ✓ （GPUで実行する領域ないに出てくる）ループが、データ独立か、リダクションか、それ以外か



OpenACCでできないこと

- CUDAならshared memoryなど使って頑張れば並列化できる、
  データ依存性のあるループの並列化
  - 例外：atomic演算で解決可能な書き込み競合を含むループ
- shared memoryなど使った性能限界を目指す**最適化**

これが必要なのはアプリの一部分であることが多いので、
ここだけCUDAやライブラリを使えば良い。

OpenACC と CUDAやライブラリの併用など、
上級者は楽するためにOpenACCを使う

OpenACCを推奨する理由

幅広いGPU搭載機で十分な性能を出すには、結局プログラムの大部分をGPU化する必要がある

- GPU搭載機は強力な代わりにノード数が少ないため、合計CPU数も少ない。本来は軽い処理でも、GPU化しておかないとボトルネックになる
- Miyabi以外の多くのGPU搭載機では、CPU-GPUデータ転送が遅い。転送時間を最小化するためには、配列を操作するCPU処理を、しらみつぶしにGPU化する必要がある

OpenACCを推奨する理由

- 幅広いGPU搭載機で十分な性能を出すには、結局プログラムの大部分をGPU化する必要がある
- しかし実アプリ全体をCUDA化するのは非常に工数が掛かるため、まずはOpenACCで全体をGPU化する
  - この時点で性能が十分であれば、GPU化を終了する
- OpenACCで並列化できないループや、OpenACCでは性能が十分ではないループに関して、CUDA化を行う
  - 多くの場合このようなループは、アプリケーションの一部に限られる

以上により、CUDA化と遜色ない性能を少ない工数で達成できる

OpenACC と CUDA の組み合わせ

- host_data指示文を使う：data指示文でCPU・GPUでペアで確保されたデータの、GPU側のアドレスをゲットできる
  → 後はやりたい放題
- GPU側のアドレスを使いたい例
  - ✓ GPU用のライブラリの呼び出し
  - ✓ CUDA で書かれた関数を呼ぶ
  - ✓ CUDA-aware MPIによる通信（GPUDirectの利用）

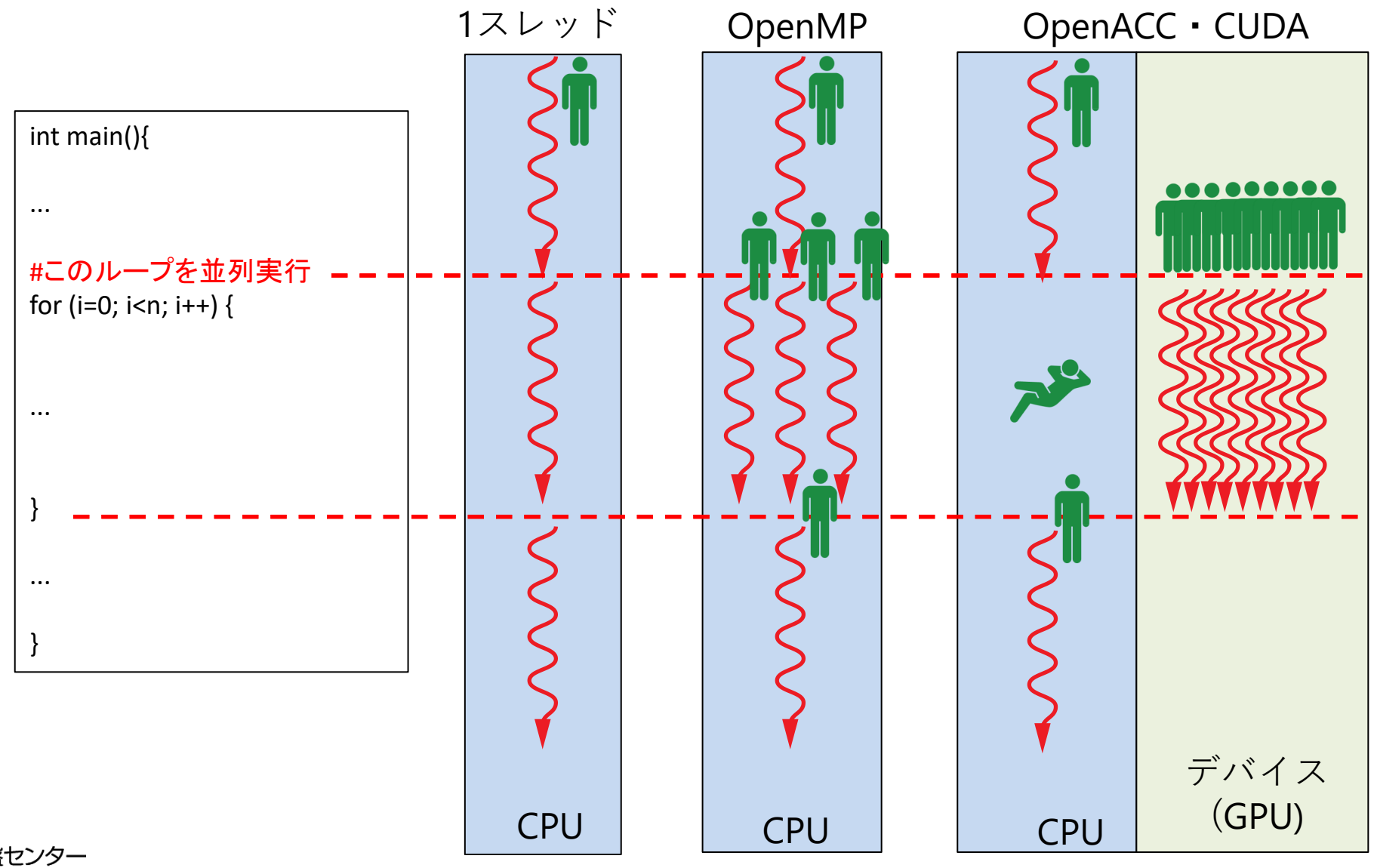
```
…
#pragma acc data copy(a[0:n])
{
…
#pragma acc host_data use_device(a)
{
   cuda_func(a, n)
}
…
}
…
```

allocate, H->D

host_data 内ではホストコードにも関わらず a はデバイス側のアドレスが使われる。

deallocate

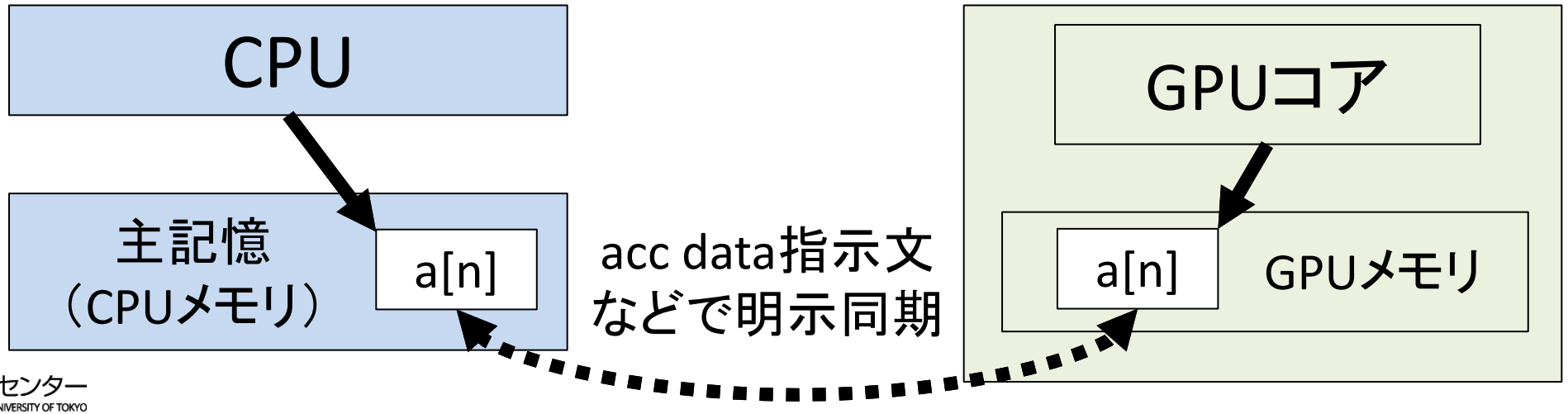
OpenACCの実行イメージ



GPUメモリの使い方は3種類ある

- 明示的にメモリ管理（Separate）
  - ➢ データをどちらのメモリに置くかを、acc data指示文などで明示的にプログラマが指示する
  - ✓ CPU-GPU転送も明示的に指示。関係ない演算の最中にバックグラウンドで転送することが可能
  - ✓ MPIでは、GPU上のデータを直接送受信可能
  - × 多数の配列を使う場合は、data指示文を整備するのが面倒。間違えればバグになる

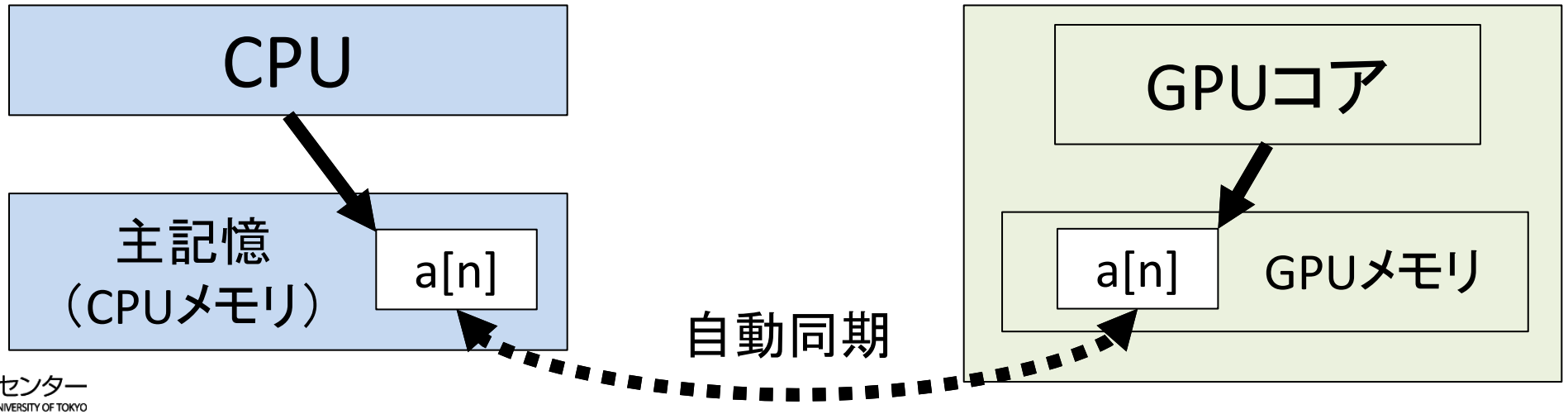
Separate memory　**-gpu=mem:separate**



GPUメモリの使い方は3種類ある

- Managed Memory (従来はUnified Memoryとも呼ばれた)
  - ➢ CPUとGPUの両方にデータを置いておき、システムが必要に応じて自動的にデータを同期してくれる
  - ✓ データの置き場や転送に関して一切指示しなくてよい（acc data指示文がある場合、コンパイラは参考にするが、必ずしも拘束されない模様）
  - × あらぬ時に自動転送が走ると遅くなる。明示的にprefetchできるが、面倒
  - × GPU上のデータをMPI送受信するときはCPUメモリを経由してしまう（はず）

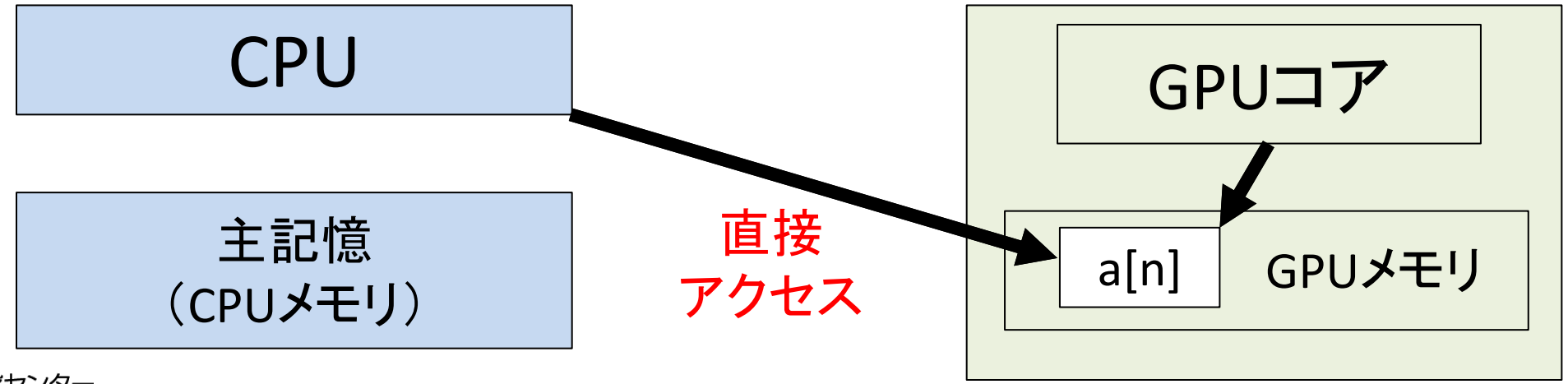
Managed memory　**-gpu=mem:managed**



GPUメモリの使い方は3種類ある

- Unified Memory
  - ➢ データは片方のメモリのみに置き、CPU・GPU双方から直接アクセス
  - ➢ 変数に最初に書き込むのがCPUだった場合、変数はCPU上に置かれるが、しばらくGPUからアクセスしているとGPUに自動で引っ越す
  - ✓ データ置き場の管理が不要。GPU上にあると、CPU・GPU双方から高速アクセス可能
  - × GPU上のデータをMPI送受信するときはCPUメモリを経由してしまう（はず）
  - × サポートされていない機種が多い

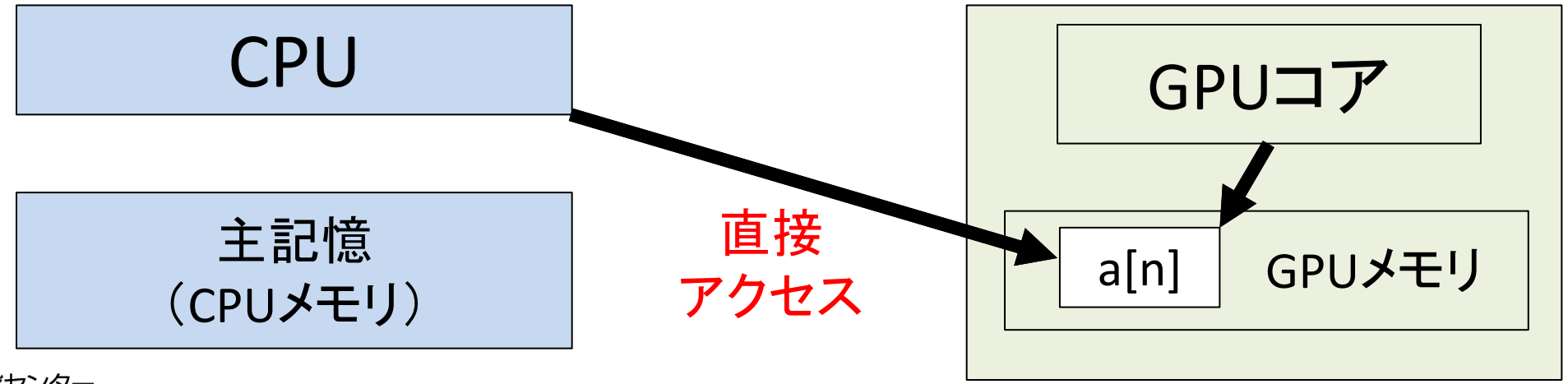
Unified memory　**-gpu=mem:unified:nomanagedalloc**



GPUメモリの使い方は3種類ある

- ✓ Miyabiでは、**まずUnified MemoryでGPU化を始めるのがおすすめ**
  - ✓ GPU化の過渡期にデータ指示文を適切に維持するのは面倒なため
- ✓ 一通りGPU化が完了し、他システムへの移植性向上やMPI通信の高速化を狙う段階に達したら、データ指示文を追加してSeparate Memory化すればよい

Unified memory　**-gpu=mem:unified:nomanagedalloc**



はじめてのOpenACCコード

openacc_hello/01_hello_acc

C

```
  const int n = 1000;

  float *a = malloc(n*sizeof(float));
  float *b = malloc(n*sizeof(float));
  float  c = 2.0;

#pragma acc kernels
#pragma acc loop independent
  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];
  }
```

F

```
integer, parameter :: n = 1000
real(4), allocatable :: a(:), b(:)
real(4) :: c
integer :: i
real(8) :: sum

allocate(a(n),b(n))
c = 2.0

!$acc kernels
do i = 1, n
  a(i) = 10.0
end do
!$acc end kernels

!$acc kernels
do i = 1, n
  b(i) = a(i) + c
end do
!$acc end kernels

sum = 0.d0
do i = 1, n
  sum = sum + b(i)
end do
```

アクセラレータ実行領域の指定

- kernels 指示文（**必須**）
  - ✓ 囲まれた領域がアクセラレータで実行されるカーネルに
  - ✓ 複数のループネストを囲んだ時、一般にはそれぞれのループネストが別々のカーネルに
    - ✓ 右の例ではカーネルが２つ生成されると思われるが、コンパイラの実装次第であるため、２つに分ける必要があるならkernels指示文を２つ使うべき
  - ✓ **推奨**：基本的には、ループネスト一つにつき一つのkernels指示文
  - ✓ **注意点**： kernels 指示文終了時に暗黙の**同期(GPU内のスレッド）**が取られる。

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

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

kernel

ループネストが独立なら、まとめて囲んでも大丈夫。どのように実行されるかはコンパイラ次第。

推奨

```
int main() {
#pragma acc kernels
  for (int i=0; i<n; i++) {
    A[i] = 0;
  }
#pragma acc kernels
  for (int i=0; i<n; i++) {
    B[i] = A[i];
  }
}
```

kernel1

kernel2

ここで同期。つまり kernel1 の終了が保証される。

kernel2 が kernel1 に依存している

アクセラレータ実行領域の指定

- kernels 指示文（**必須**）
  - ✓ kernels指示文は、Fortranの配列一括代入にも対応している
    - ✓ 独立な処理であれば、1つのkernels指示文に複数入れてもよい
  - ✓ 並列化可能かはコンパイラが判断する
    - ✓ Fortran：通常はコンパイラ判断で正しく並列化される
    - ✓ C/C++：コンパイラは、並列化可能とはほとんど判断できない。後述するloop指示文がほぼ必須

```
!$acc kernels
a(:,:) = 0
b(:,:,:) = c(:,:,:) * d(:,:,:)
!$acc end kernels
```

- parallel 指示文
  - ✓複数のカーネルに分割はしてくれない
  - ✓特に指定しない限り、全てのループが並列化可能だとみなされる
  - ✓acc parallel loop collapse(3)とすると、独立したloop指示文を一切書かずに、3重ループを完全並列実行できる

はじめてのSeparate Memory

openacc_hello/02_hello_acc_mem_separate

C

```
   float *a = malloc(n*sizeof(float));
   float *b = malloc(n*sizeof(float));
   float  c = 2.0;

#pragma acc data create(a[0:n]) copyout(b[0:n])
   {

#pragma acc kernels
#pragma acc loop independent
   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];
   }
```

F

```
  allocate(a(n),b(n))
  c = 2.0

  !$acc data create(a) copyout(b)

  !$acc kernels
  do i = 1, n
    a(i) = 10.0
  end do
  !$acc end kernels

  !$acc kernels
  do i = 1, n
    b(i) = a(i) + c
  end do
  !$acc end kernels

  !$acc end data

sum = 0.d0
  do i = 1, n
    sum = sum + b(i)
  end do
```

はじめてのSeparate Memory

openacc_hello/02_hello_acc_mem_separate

```
float *a = malloc(n*sizeof(float));
float *b = malloc(n*sizeof(float));
float  c = 2.0;

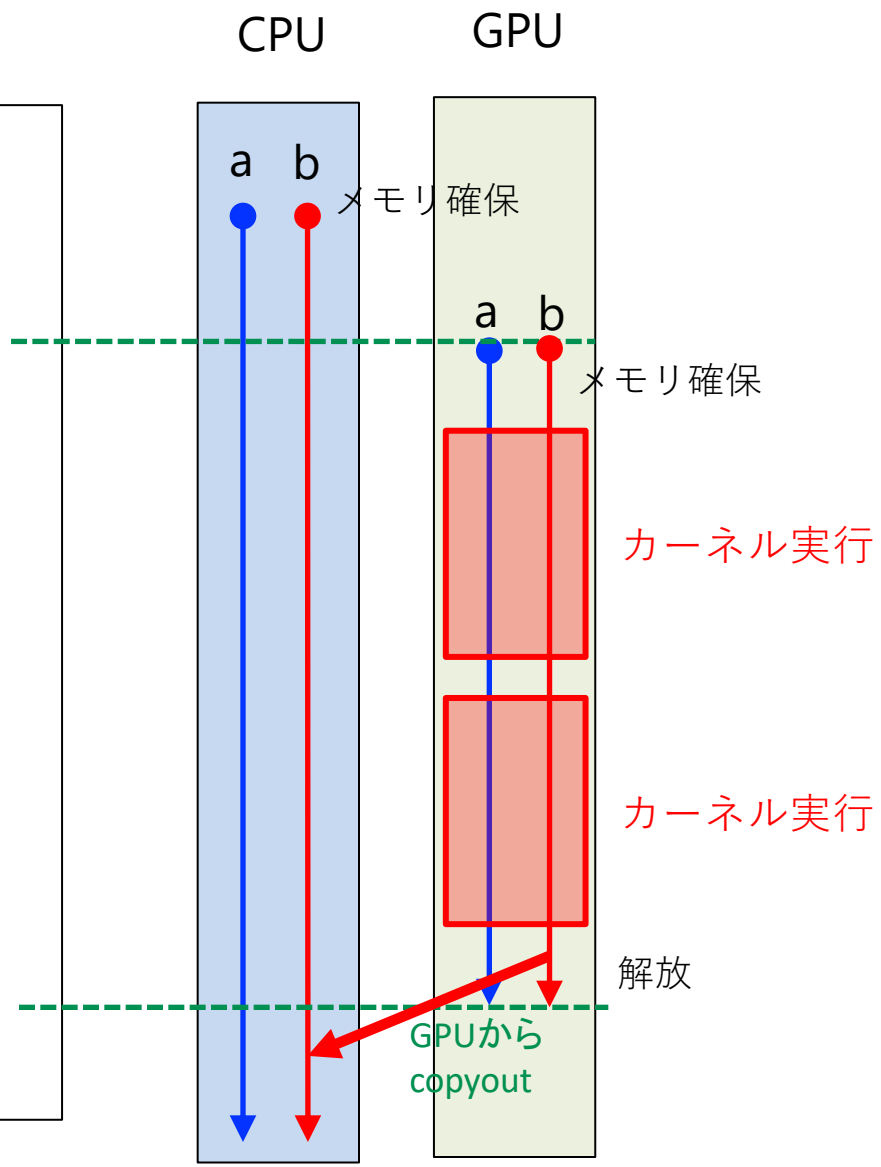
#pragma acc data create(a[0:n]) copyout(b[0:n])
  {

#pragma acc kernels
#pragma acc loop independent
  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;
  }

  }

  /* CPU上での処理… */
```



OpenACCの主な指示文

- アクセラレータ（GPU）実行領域指定指示文（**必須**）
  - ✓ kernels, parallel
- ループ最適化指示文（**オプションだが、C言語ではほぼ必須**）
  - ✓ loop
- データ移動指示文（Separate Memory**のみほぼ必須**）
  - ✓ data, enter data, exit data, update
- その他
  - ✓ host_data, atomic, routine, declare

赤字：この講習会で扱うもの

ループ最適化指示文

- loop 指示文（Cではほぼ必須）
  - ✓ ループの並列化の可否を教える
    - ➢ データ独立なループ(independent)
    - ➢ リダクションループ (reduction)
    - ➢ 並列化すべきでないループ (seq)
  - ✓ ループマッピングのパラメータの調整
    - ➢ 難しいので、最初は考える必要はない
      - コンパイラがある程度最適な値を決定してくれるので任せていい
    - ➢ gang, worker, vector を用いて指定する
      - gang: CUDA で言う thread block 数の指定。グループ単位での処理の分散を行う際に用いる。よほどの玄人以外はgangの数まで指定すべきではない。
      - worker: GPU では使わない
      - vector: CUDA で言う thread block 内の thread 数の指定。グループ内での処理の分散を行う際に用いる。数を指定するなら、1024以下の32の倍数が良い。

ループ指示文指定例

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

データ独立ループ

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

リダクションループ

```
double sum = 0;
#pragma acc kernels
#pragma acc loop independent gang
  for (int j=0; j<n; j++) {
#pragma acc loop independent vector(64)
    for (int i=0; i<n; i++) {
      sum += A[i];
    }
  }
```

多重ループへの
gang, vector適用

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

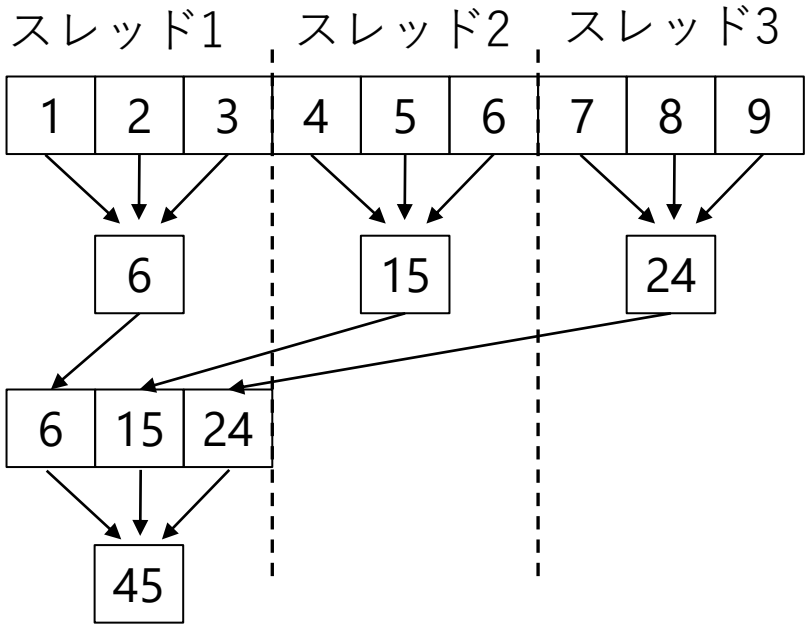
```
// これは正しくない

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

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

Separate Memory: Kernels指示文の自動データ転送

- Separate Memoryでは、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。
  - 往々にして失敗するため、後述のdata指示文を利用すべき
  - ✓ 配列はGPUのメモリに確保され、shared 変数として扱われる。
    - 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。
    - C言語の場合特に、配列のサイズがわからないなどで失敗する。
    - 各スレッドでprivateに扱うべき小さな配列は、acc kernels **private(配列名）**とする。
  - ✓ スカラ変数は firstprivate または private 変数として扱われる。
    - 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。
    - スカラ変数に関しては、自動転送に任せていい
  - ✓ 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

C

- 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

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

#pragma acc kernels
#pragma acc loop independent
  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;
  }

  }
```

openacc_hello/02_hello_acc_mem_separate

Data指示文の直後の { のタイミングで、mallocとCPU -> GPUのデータコピー（copy/copyin指定時）が行われる

対応する } の終了タイミングで、GPU -> CPUのデータコピー（copy/copyout指定時）とfreeが行われる

データ管理・移動

F

- 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

```
!$acc data create(a) copyout(b)

!$acc kernels
do i = 1, n
  a(i) = 10.0
end do
!$acc end kernels

!$acc kernels
do i = 1, n
  b(i) = a(i) + c
end do
!$acc end kernels

!$acc end data
```

openacc_hello/02_hello_acc_mem_separate

Fortranでは配列サイズ情報が変数に付随するため(lbound,ubound,sizeなどの組み込み関数をサポートしている)、基本的にサイズを書く必要がない。

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
  - ✓ コピーしない。既にデバイス上で確保済みか確認。確保していないと実行時エラー
- 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 として振る舞う。

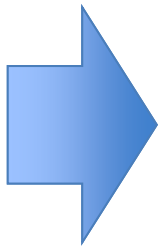
data 指示文の指示節

xxxの選択肢は
copy
copyin
copyout
create
present

```
#pragma acc data XXX(a[0:N])
{

  /* C コード */

}
```



```
if(配列aのペア、a_GPUがGPU上にまだない) {
  if(XXX == copy, copyin, copyout, create){
    a_GPU を GPU上に確保
  }
  if(XXX == copy, copyin){
    a_GPU[0:N] = a[0:N];
  }
  if(XXX == present){
    print(エラー! a はGPU上にありません！);
  }
}
{

  /* C コード */

}
if(上のif文がtrueだった時) {
  if(XXX == copy, copyout){
    a[0:N] = a_GPU[0:N];
  }
  if(XXX == copy, copyin, copyout, create){
    free(a_GPU);
  }
}
```

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])
...
```

変則的なデータ転送：enter data / exit data

- data領域を、1つのコードブロックとして囲えない場合

```
#pragma acc enter data copyin(A[0:n]) create(B[0:n])
#pragma acc exit data copyout(A[0:n]) delete(B[0:n])
```

```
!$acc enter data copyin(A) create(B)
!$acc exit data copyout(A) delete(B)
```

  - enterとexitは全く別の場所に書いてよい
    - C言語では、{ でdata領域を明示しなくてよい
  - copyin/createと、copyout/deleteの組み合わせは自由

  - 同じ変数をenterとexitで2回書くことになるので、何でもかんでもenter/exitで書くのは非推奨

変則的なデータ転送：update host / device

- data領域の中で、任意のタイミングで転送したい場合

```
#pragma acc update host(A[0:n])
#pragma acc update device(A[0:n])
```

```
!$acc update host(A)
!$acc update device(A)
```

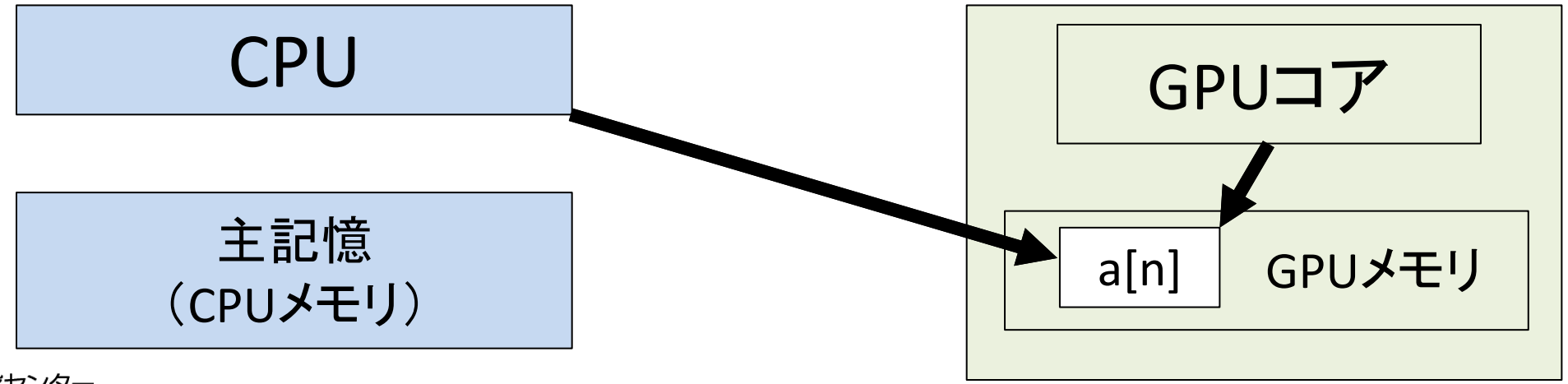
  - update hostは、デバイス（GPU）からホスト（CPU）に転送
  - update deviceは、ホスト（CPU）からデバイス（GPU）に転送
  - 使用例
    - 変数の計算途中での出力（update host）
      - デバッグ用に、GPU上で計算している変数の内容を出力する時も必要！
    - GPU化できない処理の結果をGPUに取り込むとき（update device）
      - GPUは乱数生成が苦手。標準（ライブラリ）の乱数生成器を使うなら、CPU上で生成し、update deviceでGPUに取り込むことになる

✓ MPI通信は、CUDA-aware MPIによりGPU上のデータを送受信できるので、update host/deviceは不要（代わりにacc host_data use_deviceを指定）

Unified Memory: 配列置き場はfirst touchで決まる

- 変数に最初に書き込むのがCPUだった場合、変数はCPU上に置かれる
  - しばらくGPUからアクセスしているとGPUに自動で引っ越すが、いつ引っ越すのかは処理系次第なので、遅くなる可能性がある
  - CPUで初期化する変数であっても、その前にGPUでダミーデータを書き込んでおくことで、最初からGPU上に置かれる

Unified memory　**-gpu=mem:unified:nomanagedalloc**



OpenACCコードのコンパイル

NVIDIAコンパイラによるコンパイル

✓ Miyabi-GではOpenACCはNVIDIAコンパイラで利用できます。

```
$ module load nvidia
$ nvc -fast -acc -Minfo=accel -gpu=cc90,mem:unified:nomanagedalloc  main.c
$ nvc -fast -acc -Minfo=accel -gpu=cc90,mem:separate                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。

-gpu=cc90

GPUの種類を指定する。compute capability 9.0 (cc90) 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 nvidia
$ make
```

簡単なOpenACCコード

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

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];
  }
}
```

F

```
do j = 1,ny
  do i = 1,nx
    c(i,j) = c(i,j) + a(i,j) + b(i,j)
  end do
end do
```

  - ✓ ソースコード

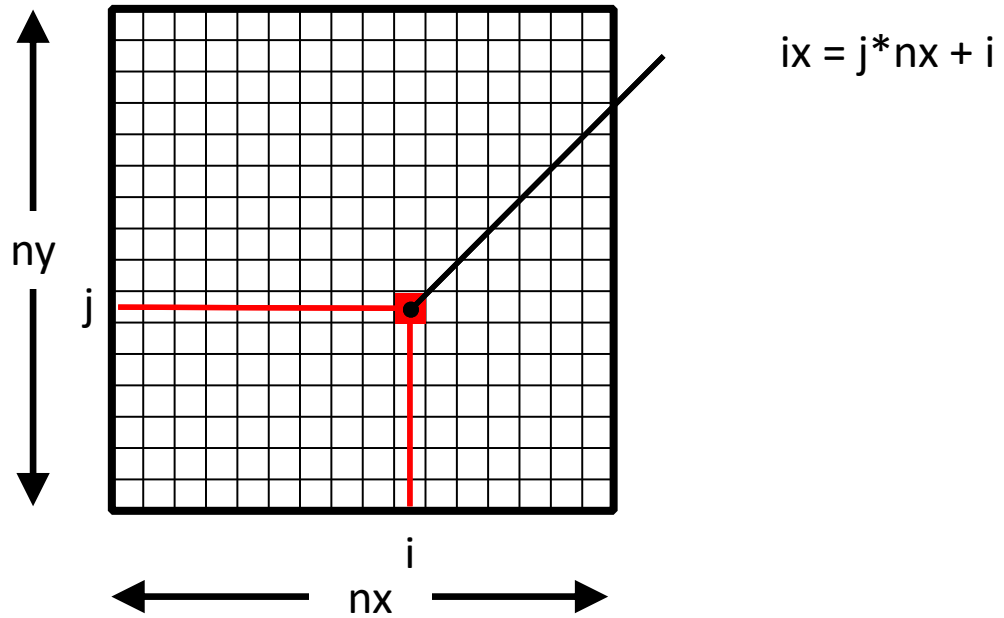
openacc_basic/01_original	CPUコード。
openacc_basic/02_kernels	OpenACCコード。上にkernels指示文のみ追加。
openacc_basic/03_loop	OpenACCコード。上にloop指示文を追加。
openacc_basic/04_firsttouch	OpenACCコード。GPU上でのfirst touchを追加。

配列のインデックス計算

C

- 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];
    }
  }
}
```



ix = j*nx + i

配列のインデックス計算

F

- 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

```
subroutine calc(nx, ny, a, b, c)
  implicit none
  integer,intent(in) :: nx,ny
  real(KIND=4),dimension(:,:),intent(in)  :: a,b
  real(KIND=4),dimension(:,:),intent(out) :: c
  integer :: i,j

  do j = 1,ny
    do i = 1,nx
      c(i,j) = c(i,j) + a(i,j) + b(i,j)
    end do
  end do

end subroutine calc
```

Fortran版では多次元配列を利用

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 =   12.105 [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）

C

F

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

openacc_basic/02_kernels

C

```
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
  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];
    }
  }
}
```

F

```
subroutine calc(nx, ny, a, b, c)
  implicit none
  integer, intent(in) :: nx,ny
  real(4), intent(in)  :: a(:,:), b(:,:)
  real(4), intent(out) :: c(:,:)
  integer :: i,j
  !$acc kernels
  do j = 1,ny
    do i = 1,nx
      c(i,j) = c(i,j) + a(i,j) + b(i,j)
    end do
  end do
  !$acc end kernels
end subroutine calc
```

OpenACC コンパイラは配列 (a, b, c) を shared 変数として自動で転送してくれるはずだが…

簡単なOpenACC: kernels 指示文（２）

C

- 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。

```
13, Complex loop carried dependence of b->,a-> prevents parallelization
      Loop carried dependence due to exposed use of c[:*] prevents parallelization
      Complex loop carried dependence of c-> prevents parallelization
      Accelerator serial kernel generated
      Generating NVIDIA GPU code
      13, #pragma acc loop seq
      14, #pragma acc loop seq
    13, Complex loop carried dependence of a->,c-> prevents parallelization
    14, Complex loop carried dependence of b->,a->,c-> prevents parallelization
      Loop carried dependence due to exposed use of c[:*] prevents parallelization
```

簡単なOpenACC: kernels 指示文（２）

C

- 実行
  - ✓ 並列化されたかったため極端に遅く、実行時間オーバーで異常終了

run.sh.o123456

```
=>> PBS: job killed: walltime 77 exceeded limit 60
```

登録アドレスに届くメール

```
PBS JOB 123456 Aborted by PBS Server

（ジョブに関するいろいろな情報：省略）
Job 123456 Aborted by PBS Server
Job exceeded resource walltime
See job standard error file
```

簡単なOpenACC: kernels 指示文（５）

F

- コンパイル
  - ✓ データの独立性を見切り、並列化。

```
14, Loop is parallelizable
    15, Loop is parallelizable
        Generating NVIDIA GPU code
        14, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
        15,   ! blockidx%x threadidx%x auto-collapsed
```

Tips: なぜデータの独立性を見切れないか

- エイリアス （変数の別名）
  - ✓ 主にポインタの利用
    - 右は一見データ独立でも…
    - foo(&a[0],&a[1]) のような呼び出しをすればデータ独立でない！
- 不明瞭な書き込み参照先
  - ✓ インデックス計算
    - 計算結果がループ変数に対して独立かどうかわからない
    - Fortranでも、多次元配列を一次元化すると起こる
    - 逆にCでも、多次元配列を使えば独立性を見切れる
  - ✓ 間接参照

これってデータ独立？

```
void foo(float *a, float *b){
 for (int i=0; i<N; i++)
   b[i] = a[i];
}
```

インデックス計算

```
for (int i=0; i<N; i++){
  j = i % 10;
  b[j] = a[i];
}
```

間接参照

```
for (int i=0; i<N; i++){
  b[idx[i]] = a[i];
}
```

簡単なOpenACC: loop 指示文（１）

C

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

openacc_basic/03_loop

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

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

簡単なOpenACC: loop 指示文（２）

F

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

openacc_basic/03_loop

```
! main 関数内
!$acc kernels
!$acc loop independent
do j = 1,ny
 !$acc loop independent
 do i = 1,nx
  b(i,j) = b0
 end do
end do
!$acc end kernels

!$acc kernels
c(:,:) = 0.0
!$acc end kernels
```

```
 subroutine calc(nx, ny, a, b, c)
  ...
!$acc kernels copyin(a,b) copyout(c)
!$acc loop independent
  do j = 1,ny
!$acc loop independent
    do i = 1,nx
      c(i,j) = a(i,j) + b(i,j)
    end do
  end do
!$acc end kernels
 end subroutine
```

```
! main 関数内
!$acc kernels
!$acc loop reduction(+:sum)
do j = 1,ny
  !$acc loop reduction(+:sum)
  do i = 1,nx
    sum = sum + c(i,j)
  end do
end do
!$acc end kernels
```

簡単なOpenACC: loop 指示文（３）

C

- 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。実行時間は0.1秒程度。

```
calc:
    14, Loop is parallelizable
    16, Loop is parallelizable
        Generating NVIDIA GPU code
        14, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
        16,   /* blockIdx.x threadIdx.x auto-collapsed */
main:
    45, Loop is parallelizable
        Generating NVIDIA GPU code
        45, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    50, Loop is parallelizable
        Generating NVIDIA GPU code
        50, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    63, Loop is parallelizable
        Generating NVIDIA GPU code
        63, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
            Generating reduction(+:sum)
```

※Fortran版は既に並列化されていたため省略。Loop指示文をつける事による挙動の変化はない。

簡単なOpenACC: GPUでのfirst touch

- 04_firsttouch コード

openacc_basic/04_firsttouch

  - 配列aはinit_cpu関数でCPU上で初期化する。
  - その前にGPUでダミーデータを入れておくことで、aが最初からGPUに置かれ、後の処理が高速化する。

C

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

   init_cpu(n, a);
```

F

```
!$acc kernels
a(:,:) = 0
!$acc end kernels

call init_cpu(nx, ny, a)
```

calc関数の実行時間は1割弱短縮する。
Separate Memoryを使う場合は、このfirst touchの処理は不要。

簡単なOpenACC: Separate Memory（１）

C F

- openacc_basic_mem_separate/03_no_datadirコード
  - ✓ Unified Memory版の03_loopと同じコードで、コンパイルオプションのみmem:separateに変更

➢ 個別のカーネルに入るたびにCPU-GPU通信が走るので遅い。1秒近くかかる。Cでは計算結果も間違っている。

```
calc:
    16, Generating implicit copy(c[:]) [if not already present]
        Generating implicit copyin(b[:],a[:]) [if not already present]
main:
    42, Generating implicit copyout(b[:16777216]) [if not already present]
    48, Generating implicit copyout(c[:16777216]) [if not already present]
    59, Generating implicit copy(sum) [if not already present]
      Generating implicit copyin(c[:16777216]) [if not already present]
```

（Unified Memoryであれば上記のコンパイル出力は無視してよい）

簡単なOpenACC: data指示文（１）

C

- 04_dataコード
  - ✓ 03_no_datadirにdata指示文とpresent指示節を追加

openacc_basic_mem_separate/04_datadir

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

  （省略）

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

  （省略）

  } /* acc end data */
```

```
#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];
        }
  }
```

✓ 0.1秒程度で終わるようになる

- 転送を伴うcopy、copyin、copyout指示節は、広域をカバーする#pragma acc data指示文に書き、個別カーネルでは全てpresentとするのがおすすめ
- GPUメモリを確保してあると思っていたのに、実際には確保し忘れていた場合、
  - present指示節に書いている場合：実行時エラー
  - present指示節に書いていない場合：カーネルごとに転送が走り、遅くなる

簡単なOpenACC: data指示文（１）

F

- 04_dataコード
  - ✓ 03_no_datadirにdata指示文とpresent指示節を追加

openacc_basic_mem_separate/04_datadir

```
!$acc data copyin(a) create(b, c)

 !$acc kernels present(b)
（省略）
 !$acc end kernels

 !$acc kernels present(c)
 c(:,:) = 0.0
 !$acc end kernels

 （省略）

 !$acc kernels present(c)
 !$acc loop reduction(+:sum)
 do j = 1,ny
   !$acc loop reduction(+:sum)
   do i = 1,nx
     sum = sum + c(i,j)
   end do
 end do
 !$acc end kernels

 !$acc end data
```

```
!$acc kernels present(a, b, c)
!$acc loop independent
do j = 1,ny
  !$acc loop independent
  do i = 1,nx
    c(i,j) = c(i,j) + a(i,j) + b(i,j)
  end do
end do
!$acc end kernels
```

✓ 0.1秒程度で終わるようになる

- 転送を伴うcopy、copyin、copyout指示節は、広域をカバーする!$acc data指示文に書き、個別カーネルでは全てpresentとするのがおすすめ
- GPUメモリを確保してあると思っていたのに、実際には確保し忘れていた場合、
  - present指示節に書いている場合：実行時エラー
  - カーネルごとに転送が走り、遅くなる

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
  - ✓ loop 指示文を用い、並列処理の指定

Unified Memoryを使う場合
（CPUで初期化する配列に関して、GPUからのfirst touchを行う）

Separate Memoryを使う場合
- ✓ data 指示文を用い、ホスト-デバイス 間の通信を最適化

OPENACC入門実習

実習

C

- 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, loop (, data) 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

実習

F

- 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 , loop (, data)指示文を追加し、GPUで高性能で実行しましょう。

diffusion.f90, diffusion3d 関数内

```
do k = 1, nz
  do j = 1, ny
    do i = 1, nx

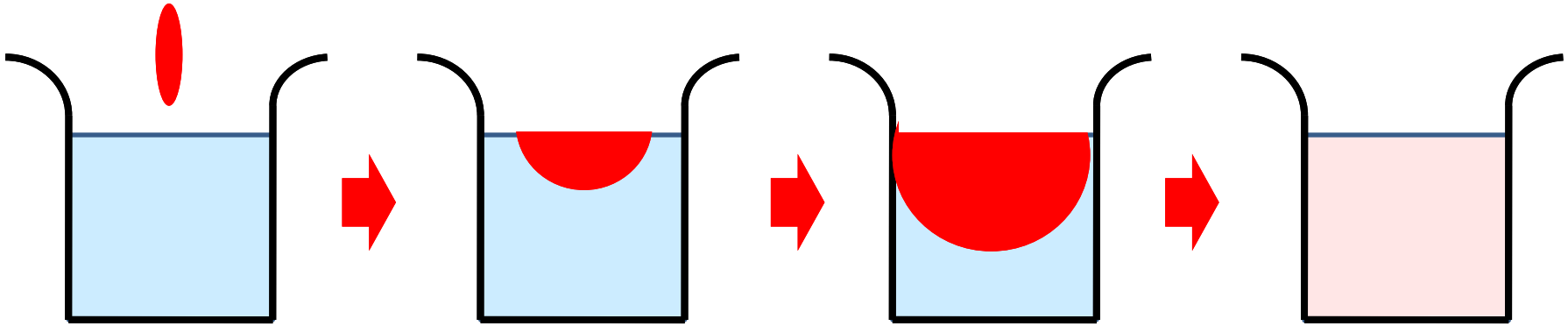
      w = -1; e = 1; n = -1; s = 1; b = -1; t = 1;
      if(i == 1)  w = 0
      if(i == nx) e = 0
      if(j == 1)  n = 0
      if(j == ny) s = 0
      if(k == 1)  b = 0
      if(k == nz) t = 0
      fn(i,j,k) = cc * f(i,j,k) + cw * f(i+w,j,k) &
          + ce * f(i+e,j,k) + cs * f(i,j+s,k) + cn * f(i,j+n,k) &
          + cb * f(i,j,k+b) + ct * f(i,j,k+t)

    end do
  end do
end do
```

openacc_diffusion/01_original

拡散現象シミュレーション（１）

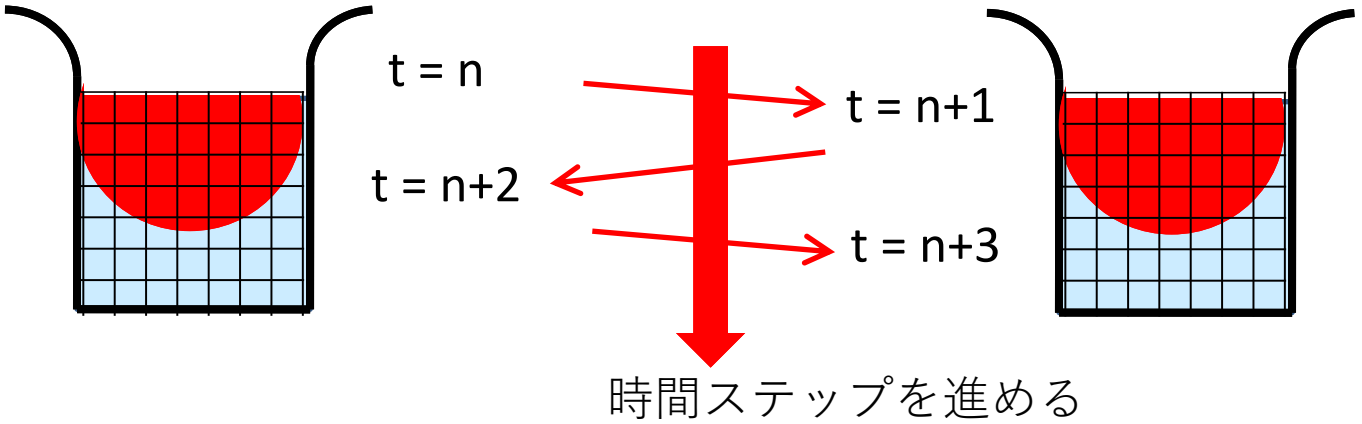
- 拡散現象
  - ✓ コップの中に赤インクを落とすと水中で拡がる
  - ✓ 次第に拡散し赤インクは拡がり、最後は均一な色になる。



- 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。

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 =    4.437 [sec]
Performance=   10.07 [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

C

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

CFLAGS    = -fast -acc -Minfo=accel -gpu=cc90,mem:unified:nomanagedalloc
GFLAGS    = -Wall -std=c99
CXXFLAGS  = $(CFLAGS)
LDFLAGS   =
...
```

F

```
F90  = nvfortran
RM   = rm -f


FFLAGS    = -fast -mp -acc -Minfo=accel -gpu=cc90,mem:unified:nomanagedalloc
...
```

openacc_diffusion/02_openacc

OpenACC化(1): kernels

C

- diffusion3d関数に kernelsを追加しましょう

openacc_diffusion/02_openacc

```
#pragma acc kernels
  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 してみましょう。

OpenACC化(2): loop

C

- diffusion3d関数に loopを追加しましょう

openacc_diffusion/02_openacc

```
#pragma acc kernels
#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。

OpenACC化(1): kernels

F

- diffusion3d関数に kernelsを追加しましょう

openacc_diffusion/02_openacc

```
!$acc kernels
 do k = 1, nz
   do j = 1, ny
     do i = 1, nx

       w = -1; e = 1; n = -1; s = 1; b = -1; t = 1;
       if(i == 1)  w = 0
       if(i == nx) e = 0
       if(j == 1)  n = 0
       if(j == ny) s = 0
       if(k == 1)  b = 0
       if(k == nz) t = 0
       fn(i,j,k) = cc * f(i,j,k) + cw * f(i+w,j,k) &
           + ce * f(i+e,j,k) + cs * f(i,j+s,k) + cn * f(i,j+n,k) &
           + cb * f(i,j,k+b) + ct * f(i,j,k+t)

     end do
   end do
 end do
!$acc end kernels
```

diffusion.f90, diffusion3d 関数内

make してジョブ投入 qsub ./run.shしてみましょう。

!$acc loop independentを入れてもよいですが、何も変わらないはずです

OpenACC化(3): init()とaccuracy()のGPU化

- init関数とaccuracy関数にもkernels指示文とloop指示文を追加しましょう
  - 両方ともdiffusion.cまたはdiffusion.f90にあります
  - 正解はopenacc_diffusion/02_openacc/にあります
  - accuracy関数はリダクションが必要なのでご注意ください
- Unified Memoryを使う場合は、これでGPU化は完了です

OpenACC化(4): data指示文の追加

C

- Makefileで、mem:unified:nomanagedallocをmem:separateに変えましょう
- main関数に data指示文 を追加しましょう

```
  float *f  = (float *)malloc(sizeof(float)*n);
  float *fn = (float *)malloc(sizeof(float)*n);

#pragma acc data create(f[0:n], fn[0:n])
  {
    init(nx, ny, nz, dx, dy, dz, f);

    start_timer();

    （省略）

    const double ferr = accuracy(time, nx, ny, nz, dx, dy, dz, kappa, f);
    fprintf(stdout, "Error[%d][%d][%d] = %10.6e¥n", nx, ny, nz, ferr);

  }

  free(f);  f  = NULL;
  free(fn); fn = NULL;
```

main.c, main 関数内

- 今回はinit()、diffusion3d()、accuracy()をGPU化したので、それらを全て包含するようにdata領域を設定します
- Data領域外ではfやfnの値には一切読み書きしないので、create指示節で十分です

OpenACC化(4): data指示文の追加

F

- Makefileで、mem:unified:nomanagedallocをmem:separateに変えましょう
- メインプログラム内に data指示文 を追加しましょう

```
allocate(f(nx,ny,nz))
allocate(fn(nx,ny,nz))

!$acc data create(f, fn)

call init(nx, ny, nz, dx, dy, dz, f)

do icnt = 0, nt-1
  flop = flop + diffusion3d(nx, ny, nz, dx, dy, dz, dt, kappa, f, fn)
  call swap(f, fn)

  time = time + dt
  if(time + 0.5*dt >= 0.1) exit
end do

（省略）

ferr = accuracy(time, nx, ny, nz, dx, dy, dz, kappa, f)
write(*, "(A6,I0,A2,I0,A2,I0,A4,E12.6)"), "Error[",nx,"][",ny,"][",nz,"] = ",ferr

!$acc end data
```

- 今回はinit()、diffusion3d()、accuracy()をGPU化したので、それらを全て包含するようにdata領域を設定します
- Data領域外ではfやfnの値には一切読み書きしないので、create指示節で十分です

main.f90, メインプログラム内

OpenACC化(4): data指示文の追加

C

- diffusion3d、init、accuracyの各関数に書いたカーネルに、データ指示節を追加しましょう

- diffusion3d()
  - #pragma acc kernels **present(f, fn)**

- init()
  - #pragma acc kernels **present(f)**

- accuracy()
  - #pragma acc kernels **present(f)**

OpenACC化(4): data指示文の追加

F

- diffusion3d、init、accuracyの各サブルーチンに書いたカーネルに、データ指示節を追加しましょう

- diffusion3d()
  - !$acc kernels **present(f, fn)**

- init()
  - !$acc kernels **present(f)**

- accuracy()
  - !$acc kernels **present(f)**

詳しい性能測定の方法（１）Nsight Systems

全体の実行時間をプログラム内で測定するだけだと、期待する性能が出ない場合に、プログラム内のどこが遅いのか分からない

→ ここでは、NVIDIAコンパイラを使用する場合の、詳しい性能情報の取得法を2つ紹介

※ 2つとも、Unified Memory使用時はカーネルの速度を損なうことがある

プロファイラNsight Systemsを利用する

① GPU化したプログラムを起動するときに、実行ファイルの前にnsys（とプロファイリング設定）を挿入する

```
$ nsys profile -f true -t cuda,nvtx,openacc,mpi -o report ./run
```

② nsys-uiを起動し、File - Open から、report.nsys-rep を選択

- nsys-uiはローカルPCにインストールしてもよいが、Miyabiにssh -Yで接続している場合は、ログインノードのターミナル上で起動できる

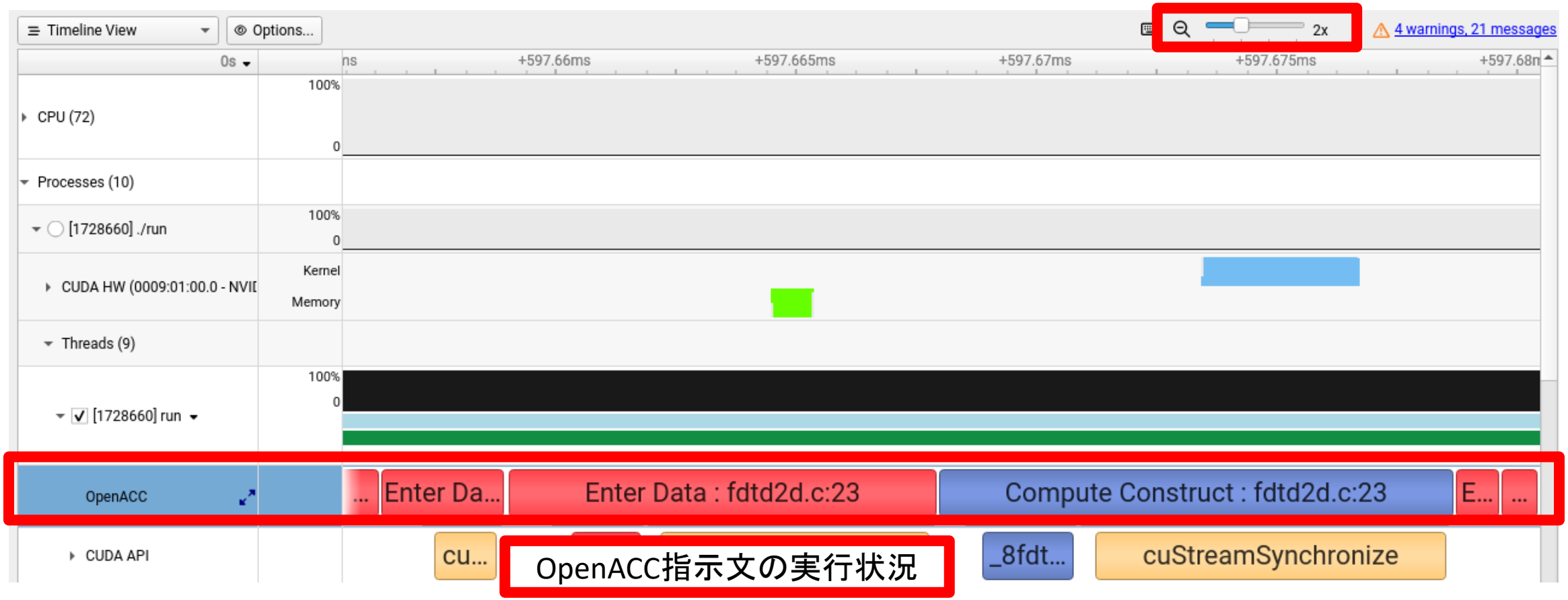
詳しい使い方は、第239回講習会の資料をご参照ください

サンプルコードは、 openacc_diffusion/03_openacc_datadir

詳しい性能測定の方法（１）Nsight Systems 画面例

横方向の拡大：Ctrl＋マウスホイール

縦方向の拡大



プロファイル採取時に -t cuda,openacc と書いておかないと、OpenACCの行は出ない

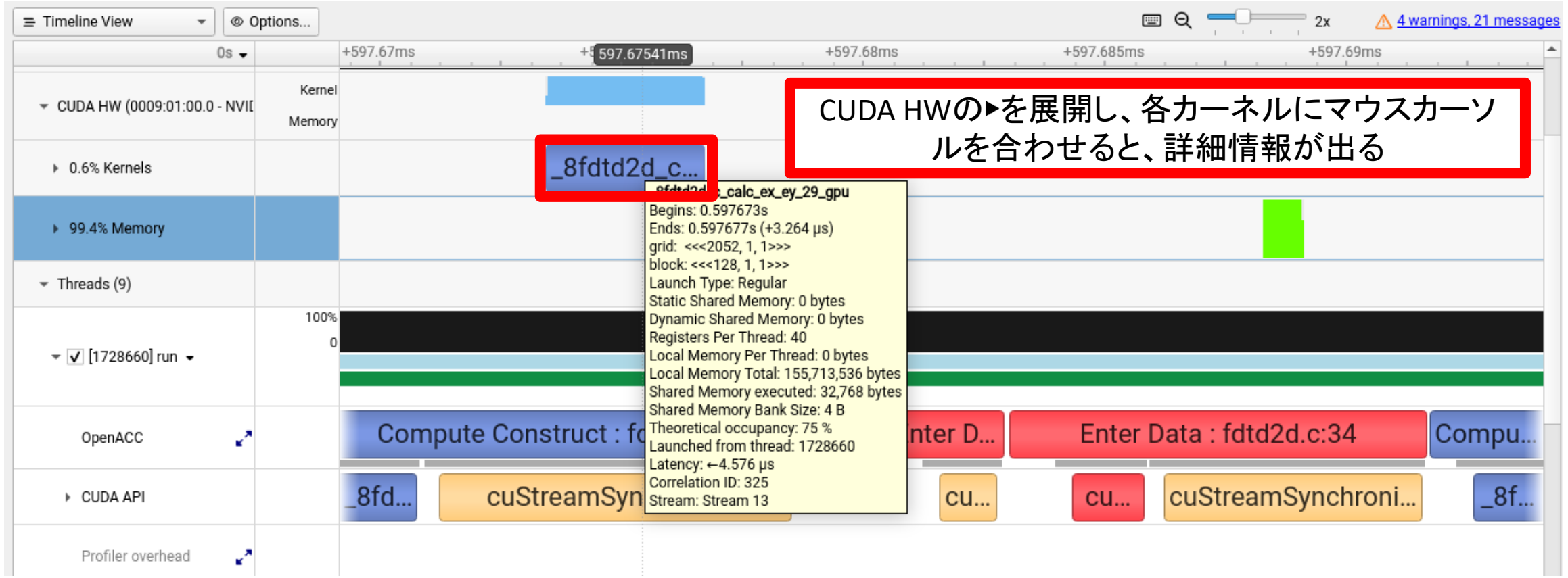
詳しい性能測定の方法（１） Nsight Systems 画面例

横方向の拡大：Ctrl＋マウスホイール

縦方向の拡大

詳しい性能測定の方法（１）Nsight Systems 画面例

横方向の拡大：Ctrl＋マウスホイール



詳しい性能測定の方法（１）Nsight Systems

プログラムの性能が出ない原因と検出法

- 重い処理がGPU化されていない
  - Nsight Systemsでは、カーネルがない隙間時間として見つけられる
- カーネルが十分に並列化されていない
  - 長時間かかっているカーネルにカーソルを合わせ、grid<?,?,?>とblock<?,?,?>の中の6つの数値を全て掛け合わせてみる。その数値が数十万程度以上、あるいは当該ループの全長になっていれば十分。それより少なければ、並列化が不十分なことが多い
  - コンパイル時に、当該カーネルに関して、意図しないacc loop seqが出力されていないか確認
- CPU-GPU通信が多すぎる
  - CUDA HWの行であれば緑・赤、OpenACCの行であれば赤（Enter Data/Exit Data）がCPU-GPU通信

詳しい性能測定の方法（２） NVCOMPILER_ACC_TIME

全体の実行時間をプログラム内で測定するだけだと、期待する性能が出ない場合に、プログラム内のどこが遅いのか分からない

→ ここでは、NVIDIAコンパイラを使用する場合の、詳しい性能情報の取得法を2つ紹介

※ 2つとも、Unified Memory使用時はカーネルの速度を損なうことがある

環境変数NVCOMPILER_ACC_TIMEを1に設定する

- Linuxなどでは、環境変数NVCOMPILER_ACC_TIME を1に設定し、プログラムを実行する。

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

- Miyabi でジョブに環境変数NVCOMPILER_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
…

export NVCOMPILER_ACC_TIME=1
./run
```

サンプルコードは、 openacc_diffusion/03_openacc_datadir

詳しい性能測定の方法（２） NVCOMPILER_ACC_TIME

- 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。

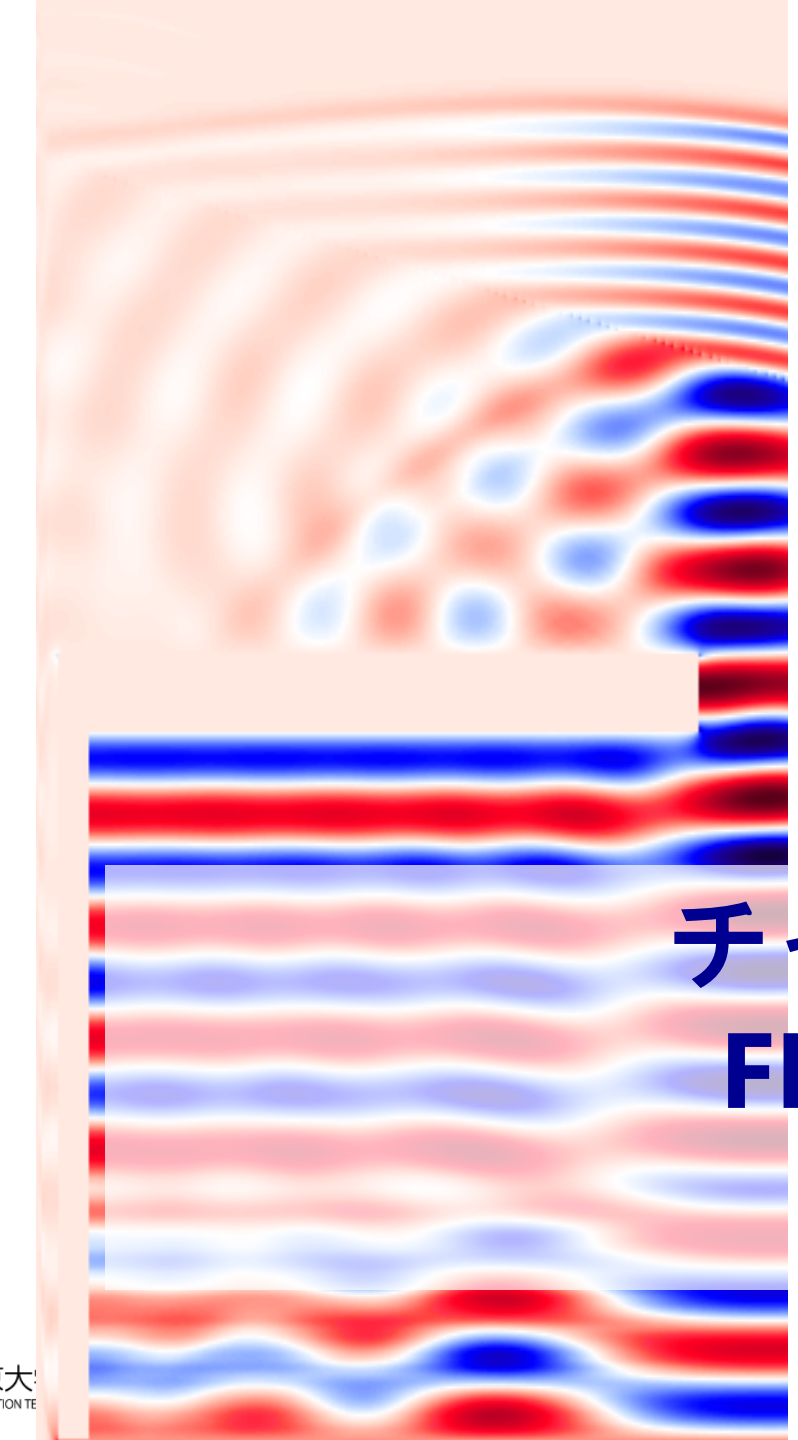
run.sh.o?????? (一部)

```
Accelerator Kernel Timing data
    78: compute region reached 1 time
    86: kernel launched 1 time
        grid: [16384]  block: [128]
         device time(us): total=36 max=36 min=36 avg=36
        elapsed time(us): total=48 max=48 min=48 avg=48
    86: reduction kernel launched 1 time
        grid: [1]  block: [256]
         device time(us): total=16 max=16 min=16 avg=16
        elapsed time(us): total=28 max=28 min=28 avg=28
  78: data region reached 4 times
    78: data copyin transfers: 1
         device time(us): total=3 max=3 min=3 avg=3
    97: data copyout transfers: 1
         device time(us): total=19 max=19 min=19 avg=19
/work/gt00/z30127/lectures/GPU_intro/lecture_openacc/C/openacc_diffusion/03_openacc_datadir/mai
n.c
 main  NVIDIA  devicenum=0
  time(us): 0
  38: data region reached 2 times
```

起動したスレッド

カーネル実行時間

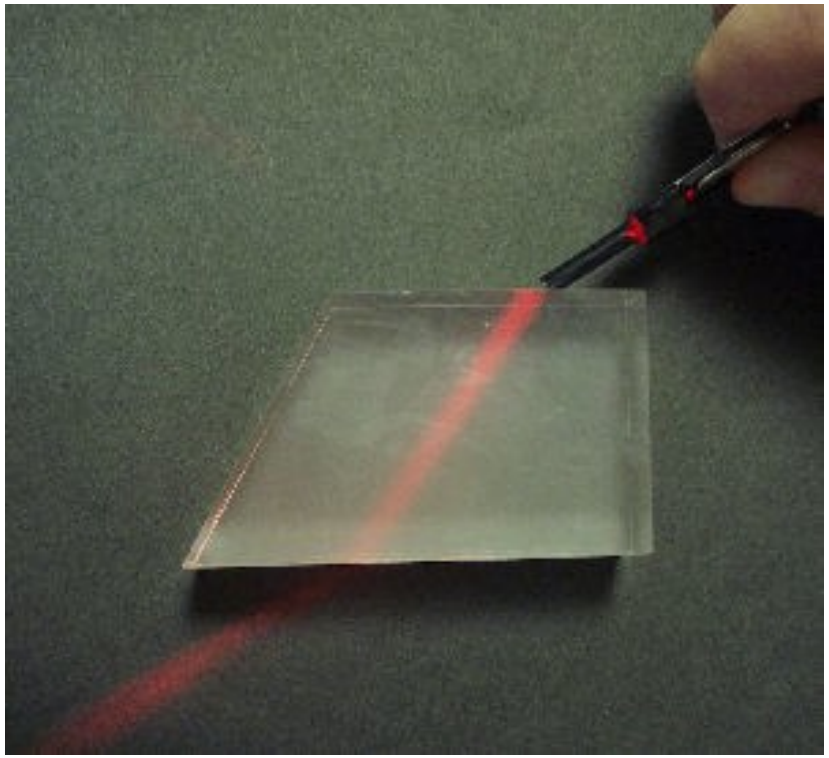
データ移動の回数

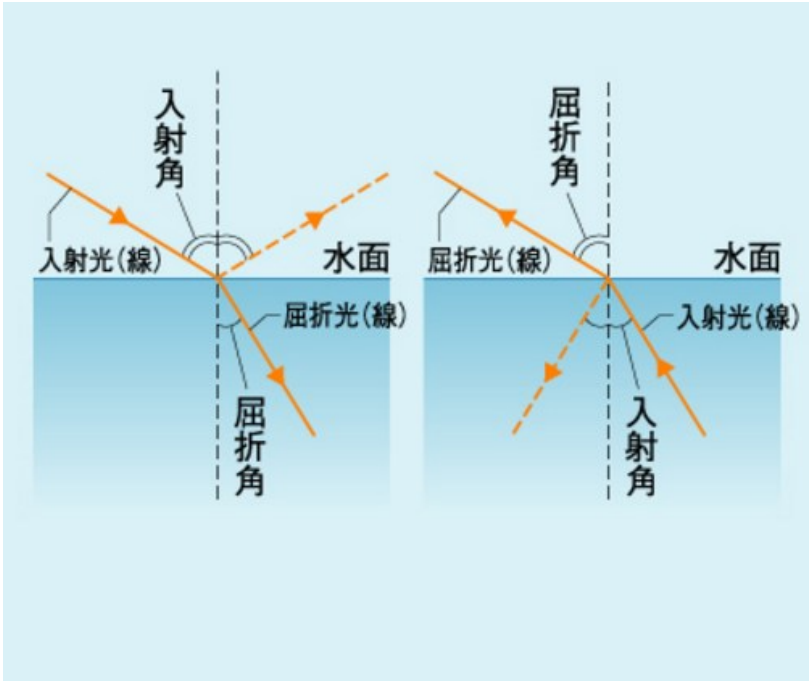


チャレンジ課題：GPUを用いたFDTD法による電磁波伝搬計算(C言語版のみ)

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

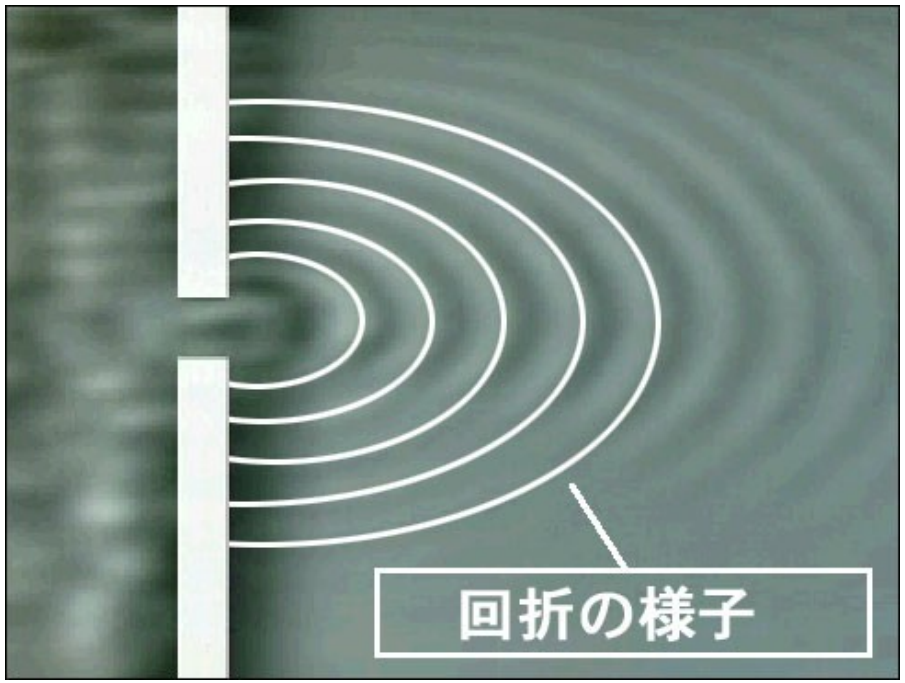


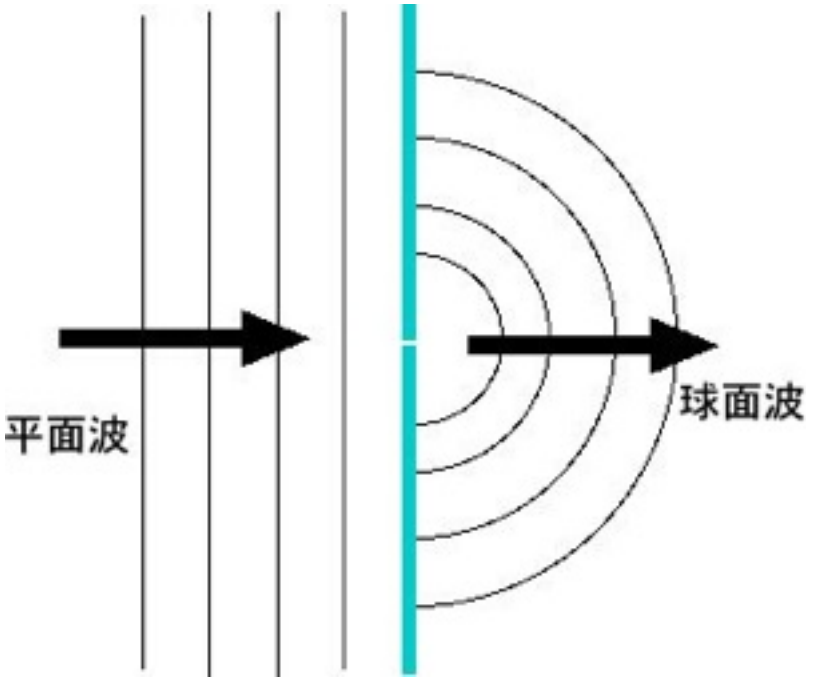


http://www.kagakunavi.jp
http://www2e.biglobe.ne.jp/~shinzo/jikken/pikatto/pikatto.html

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

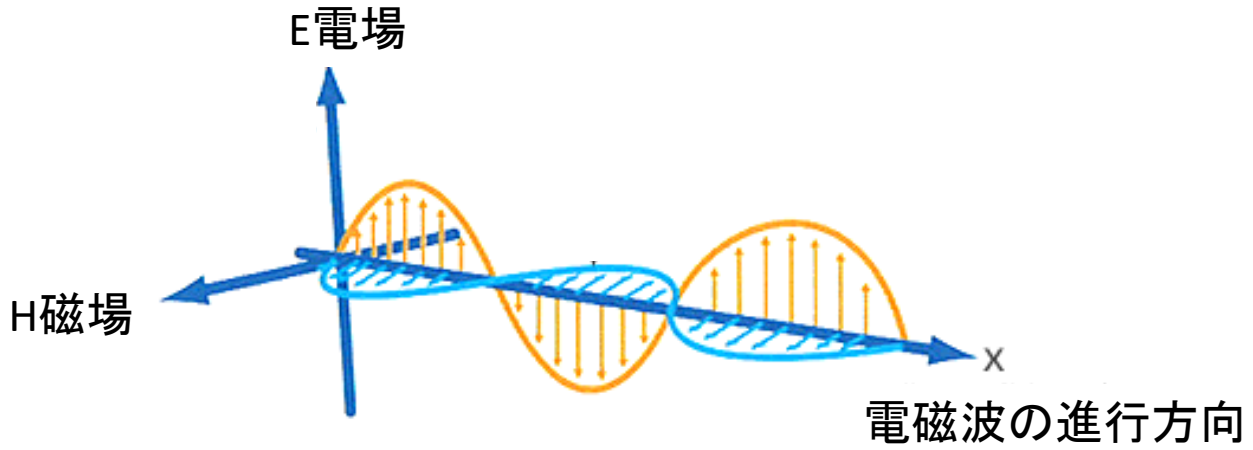




http://rikanet2.jst.go.jp
http://web.me.com/takeda_t/atelier_bonryu/PH_Salon_R.4.html

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)

この方程式を、2次元FDTD法（Finite-difference time-domain 法）*を用いて解いていきます。

* 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、

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

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

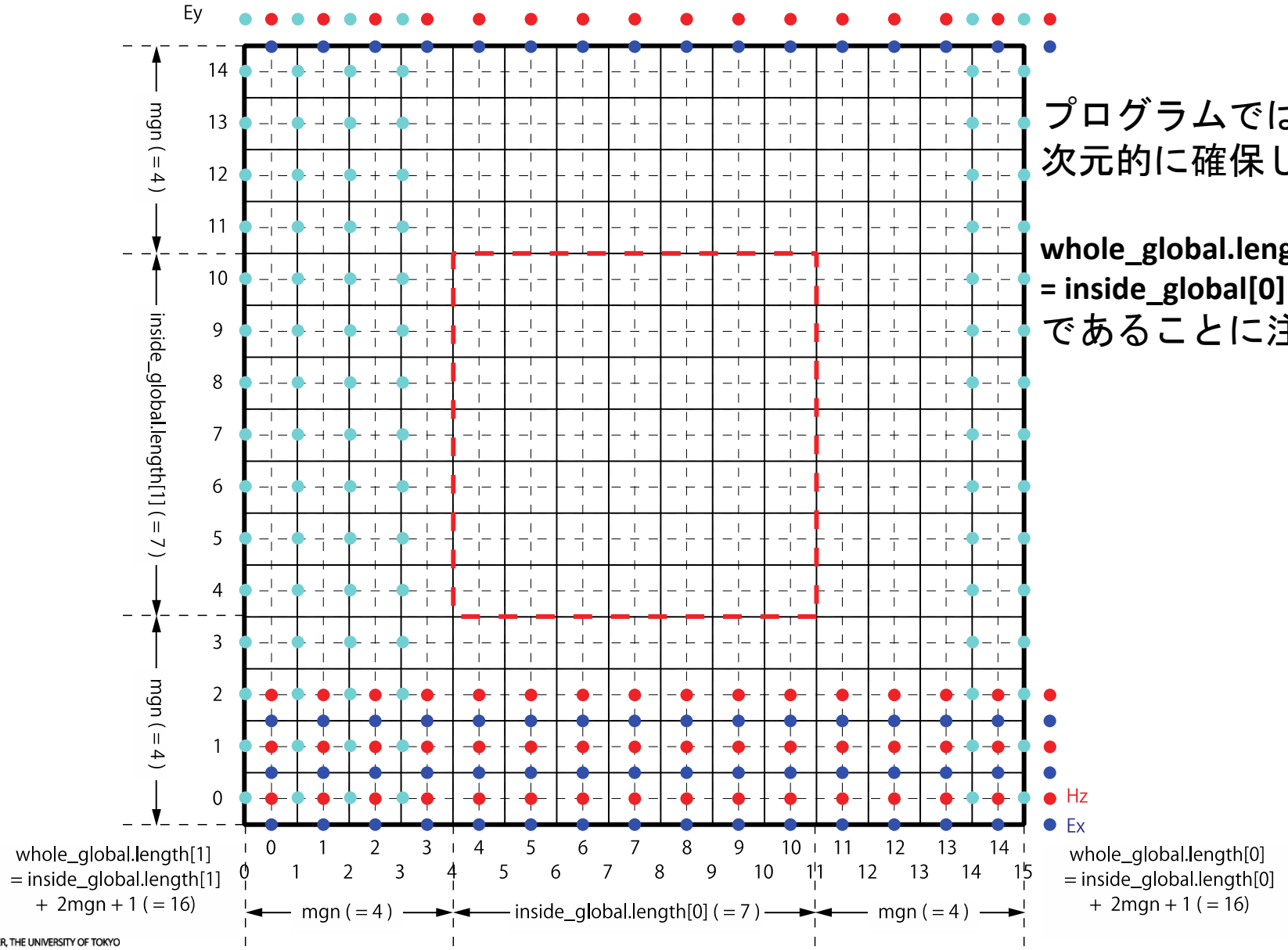
FDTD法（２）

- 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	CPUコード。
openacc_fdtd/02_openacc	Unifoed MemoryによるOpenACC化。
openacc_fdtd/03_openacc_mem_separate	Separate MemoryによるOpenACC化。
openacc_fdtd/04_openacc_mem_separate_optimized	データ移動の最適化。

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

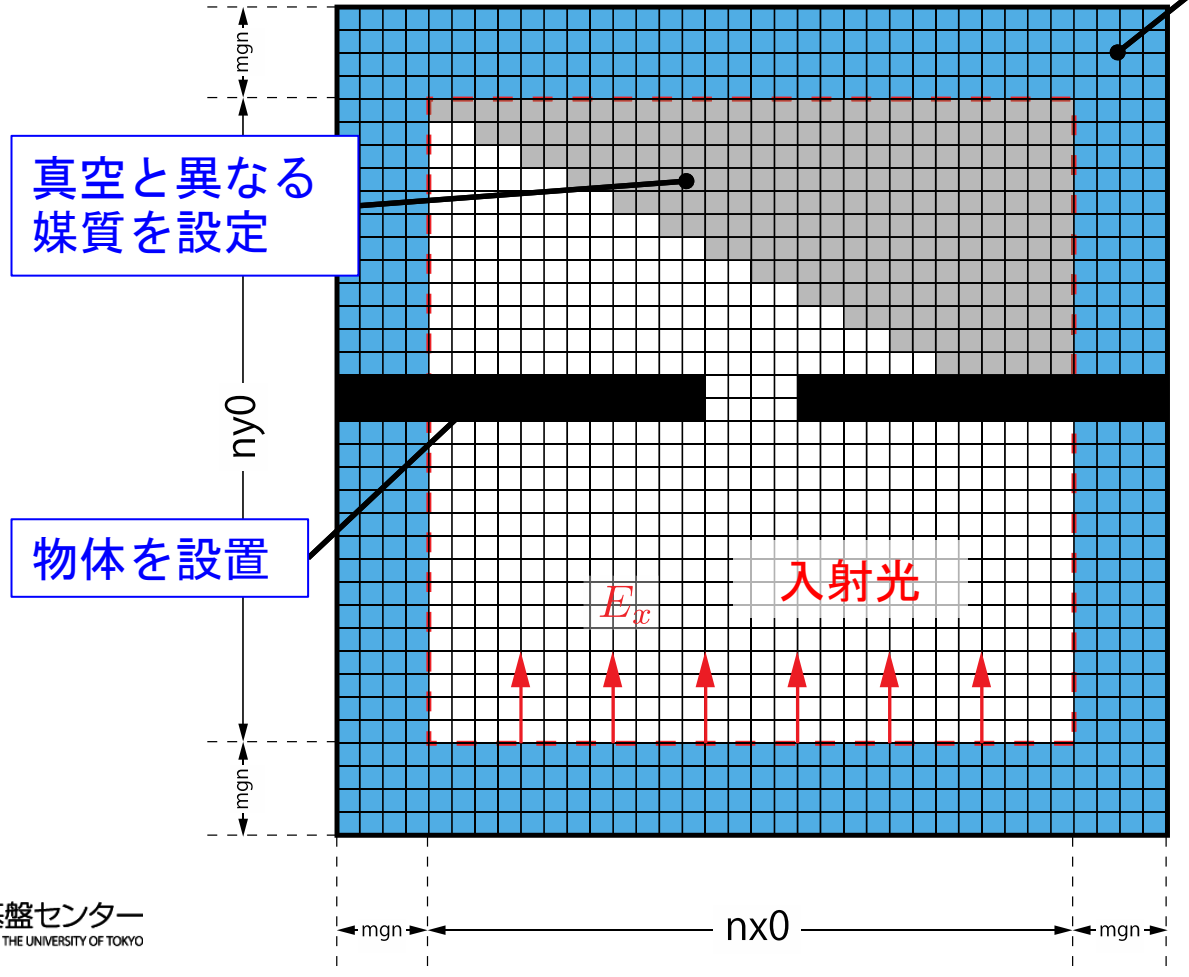
本講習では、“main.c”、“fdtd2d.c”、” fdtd2d_sources.c”、”setup.c” のソースコードを追記・修正していきます。

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)



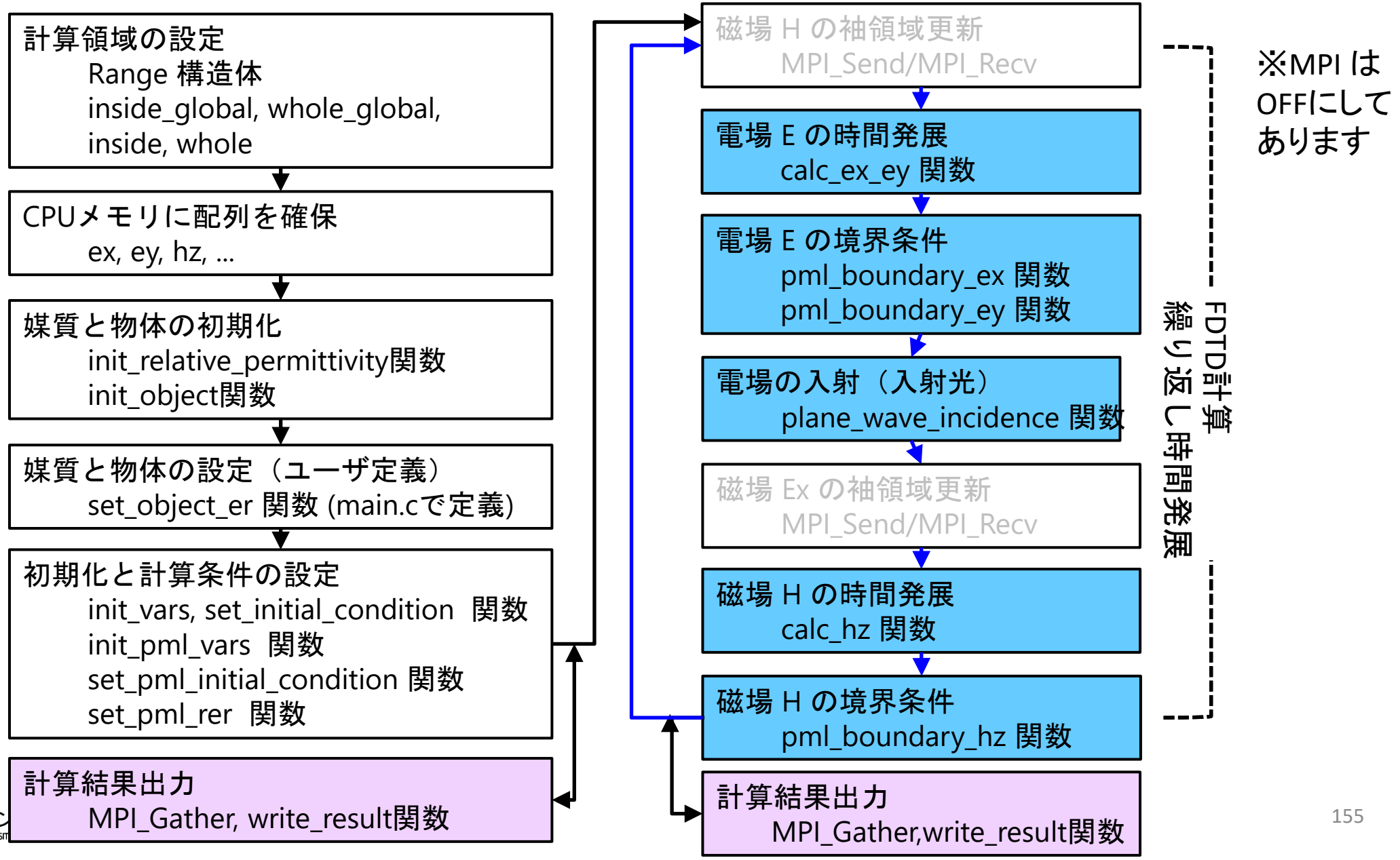
dx = lx/nx
dy = ly/ny

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

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

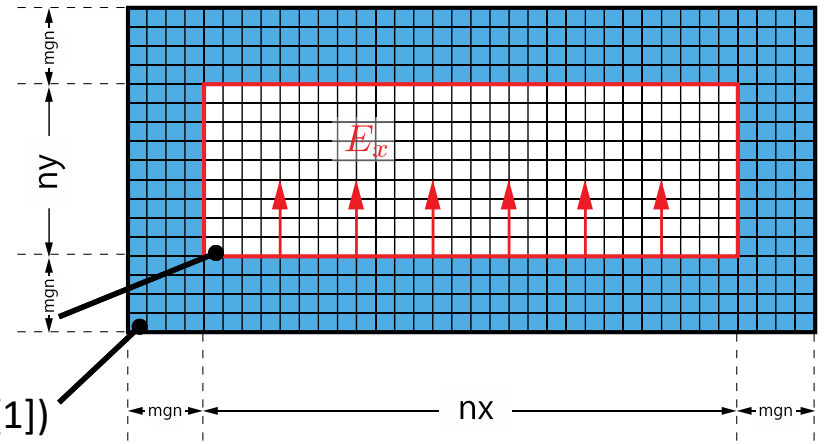
プログラムでは
下記の通り

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]

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）

※MPIは
OFFにして
あります

```
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）

※MPIは
OFFにして
あります

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

```
  （前半から）

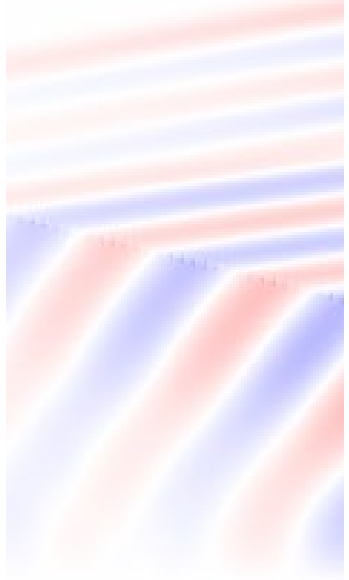
  const int src_ex     = whole.length[0] * (inside.begin[1] - whole.begin[1]);
  const int dst_ex     = whole.length[0] * (inside_end1     - whole.begin[1]);

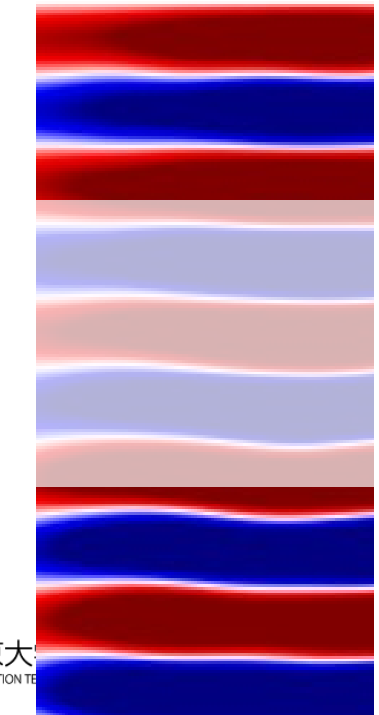
  MPI_Send(&ex[src_ex], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD);
  MPI_Recv(&ex[dst_ex], nhalo, MPI_FLOAT_T, rank_up  , tag, MPI_COMM_WORLD, &status);

  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)
}
```





チャレンジ課題:GPUを用いたFDTD法による電磁波伝搬計算の実習（C言語版のみ）

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

```
$ module load nvidia nv-hpcx
$ cd openacc_mpi_fdtd/01_original
$ make
$ qsub ./run.sh
$ cat run.sh.o??????
Rank 0: hostname = mg1097
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 = 1
output_file = 1
Time       =   3.130138 [sec]
-----------------------------
```

MPIのmoduleが必要

?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

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

```
#!/bin/bash
#PBS -q lecture-g
#PBS -l select=1
#PBS -l walltime=00:01:00
#PBS -W group_list=gt00
#PBS -j oe

cd $PBS_O_WORKDIR
module load nvidia nv-hpcx

mkdir -p images

size=512
total_step=5000
img_step=50
nprocs=1
mpirun -n $nprocs ./run $size $size $nprocs $total_step $img_step

module load python
python3 ../create_images.py $size $total_step $img_step

rm result.grd
```

size: 計算領域サイズ
total_step: 全時間ステップ
Img_step: 出力を行うタイムステップ数の頻度。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。
nprocs: 全ランク数（=分割数）※今回は1

最後にPythonスクリプトで出力を画像化しているが、これはGPU化不要

計算結果の表示

- 計算結果は images/ に PNG画像 として出力される

$ cd images/

openacc_fdtd/01_original

- 計算結果の表示
  - ✓ 1枚の画像を見る

$ display 5000.png

  - ✓ 複数の画像をアニメーションで表示（時間がかかる）

$ animate *.png

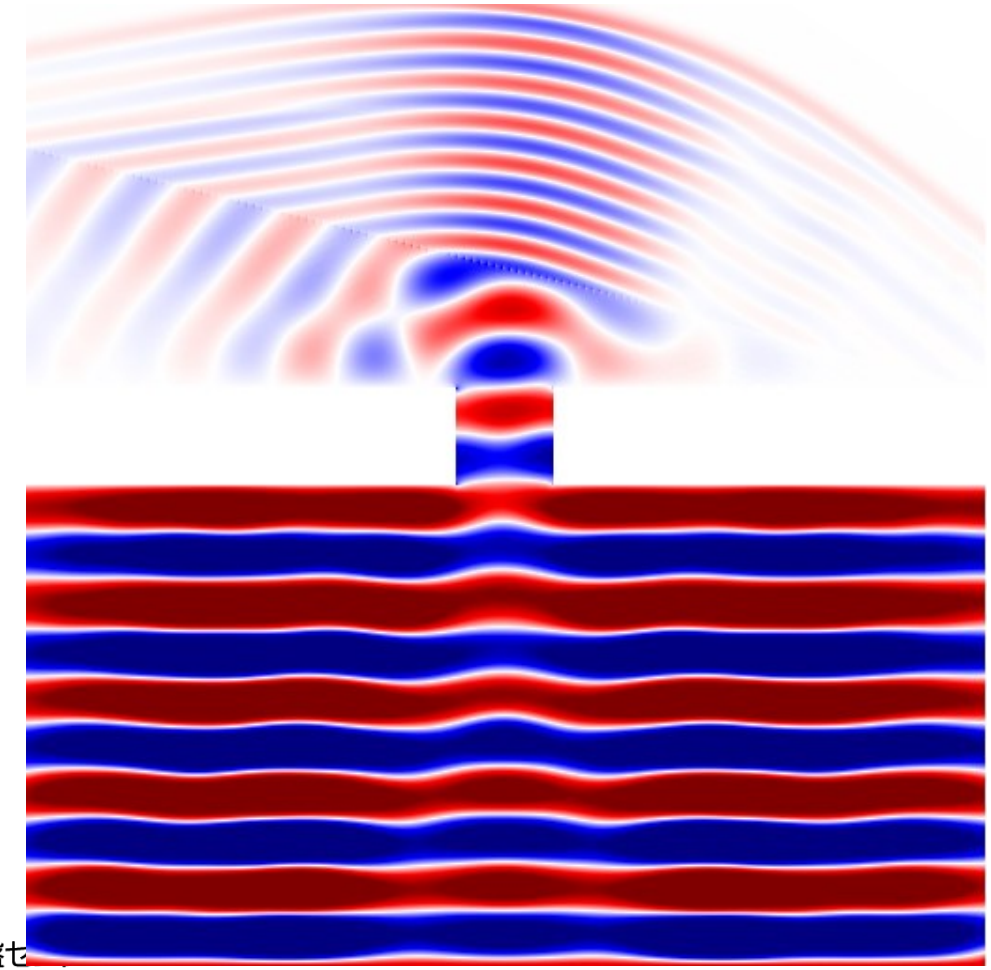
なお

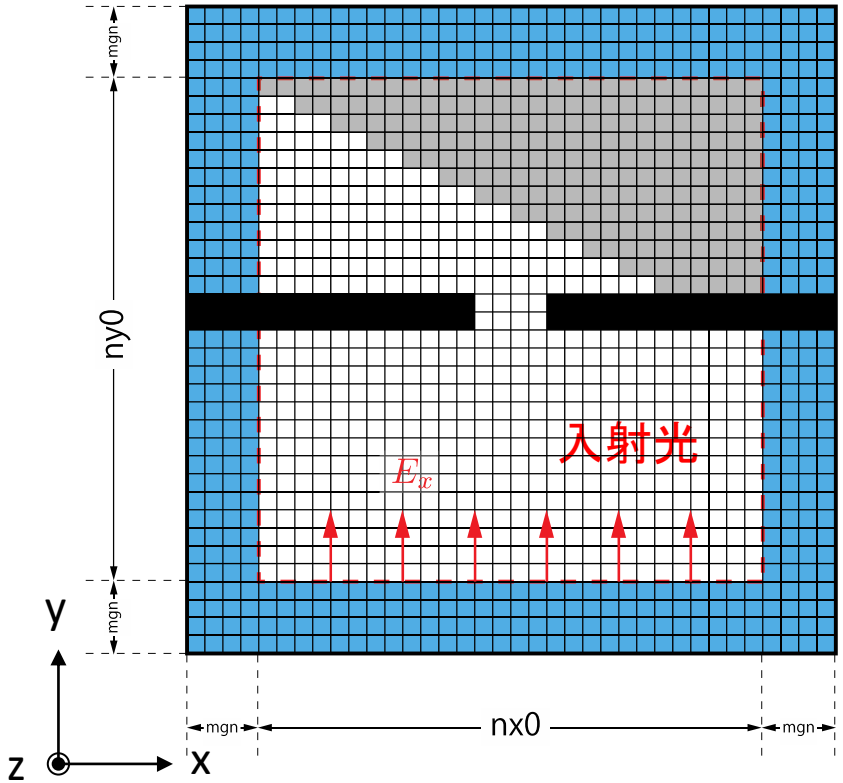
ssh -Y txxxxx@miyabi-g.jcahpc.jp

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

- 出力された画像の一例
  - ✓ Ex （電場の x 成分）の出力





実習1-1

- calc_ex_ey関数, calc_hz関数 を OpenACC化しましょう。

- Makefile
  - ✓ コンパイルオプションの修正
- 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_openacc

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
#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1-2

- pml_boundary_ex、pml_boundary_ey、pml_boundary_hz 関数をOpenACCにしましょう。

- fdtd2d.c
  - ✓ 残りの関数にkernels 指示文、loop 指示文の追加

解答例は、openacc_fdtd/02_openacc

実習1-2

- 外側のlのループと内側のiのループは並列化されるものの、真ん中のjのループは#pragma acc loop independentを付けても並列化されない

160, #pragma acc loop gang /* blockIdx.x */
162, #pragma acc loop seq
164, #pragma acc loop vector(128) /* threadIdx.x */

  - 今回はループ長に配列が絡んで複雑なので、複数ループを自動一重化するcollapseは使用不可
  - 繰り返し数はlのループよりもjのループのほうが多いので、本当はjのループを並列化してほしい…

  - ✓ acc kernelsをacc parallelに変更し、外側のlのループをあえてacc loop seqにする（並列化させない）ことで、jのループがちゃんと並列化される → 高速化！

解答例は、openacc_fdtd/02_openacc

実習1-3

- メインループ（main.c main関数のwhile (icnt < nt)のループ）内から呼び出している、全ての大規模配列操作をOpenACCにしましょう。

- fdtd2d_sources.c
  - plane_wave_incidence関数の最後のループ
    - kernels 指示文、loop 指示文の追加
- main.c
  - write_result()呼び出しに先立ち、 ex_global等に値を入れているループ
    - kernels 指示文、loop 指示文の追加

- ファイル出力（main.c write_result関数）もGPU化は可能だが、発展的な内容なのでここでは省略。GPUDirect Storageで検索！

解答例は、openacc_fdtd/02_openacc

実習1-4

- 初期化処理のうち、set_object_er関数より後に呼ばれているものをOpenACCにしましょう。
  - set_object_er関数はユーザー定義関数としており、ここではGPU化しない

- setup.c
  - init_vars、set_initial_condition、init_pml_vars、set_pml_conf、set_pml_rer関数
    - kernels 指示文、loop 指示文の追加

これで、Unified MemoryでのGPU化は完了です

解答例は、openacc_fdtd/02_openacc

実習2：Separate Memory

- Separate Memoryに変更してみましょう。

- Makefile
  - コンパイルオプションの変更

- main.c
  - set_object_er関数の直後にdata指示文と { を入れ、下記でpresentに指定した全ての変数に対して、適切なデータ指示節を付ける
  - GPU処理が全て終わってから、data領域を } で閉じる
  - output関数で、ex・ey・hzに対してacc update hostを指示する

- fdtd2d.c、fdtd2d_sources.c、setup.c、main.c
  - 全てのOpenACCカーネル（acc kernels、acc parallel）
    - 一部の小さな配列（r、bw、whole->length、mgn）を除く、全ての配列をpresentに指定

解答例は、openacc_fdtd/03_openacc_mem_separate

data 指示文

```
// User-defined function
set_object_er(&whole, lx, ly, dx, dy, obj, er);

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

  init_vars(whole.length, ex, ey, hz);
  …
```

- objとerは、このdata指示文より上のCPU処理で初期化しているので、ここでGPUにコピーが必要
- それ以外は、初期化もGPU化したのでH->Dコピー不要
- どの変数も、最後にCPUへの転送は不要

↓

- objとerはcopyin
- それ以外はcreate

update host 指示文

```
void write_result(FILE *output_file, const int length,
            const FLOAT *ex, const FLOAT *ey, const FLOAT *hz)
{
#pragma acc update host(ex[0:length], ey[0:length], hz[0:length])
   fwrite(ex, sizeof(FLOAT), length, output_file);
   fwrite(ey, sizeof(FLOAT), length, output_file);
   fwrite(hz, sizeof(FLOAT), length, output_file);
}
```

main.c

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。
  - ✓ Nsight SystemsまたはNVCOMPILER_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/04_openacc_mem_separate_optimized

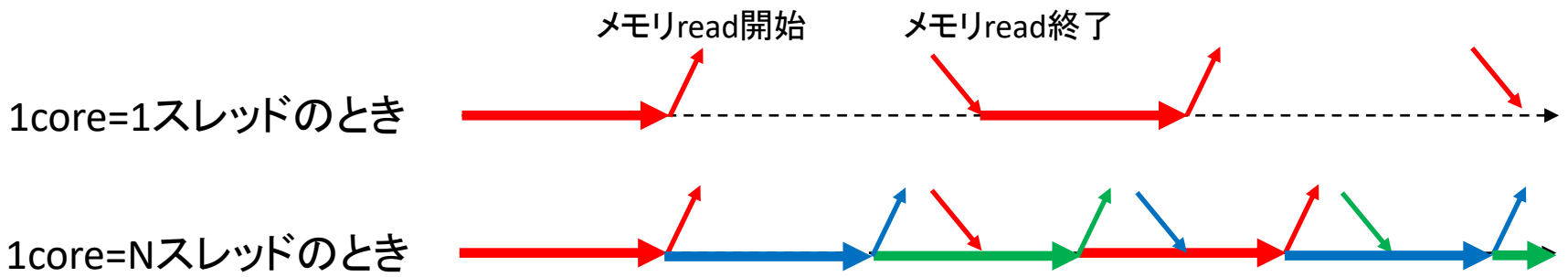
Q & A

- アカウントは1ヶ月有効です。
- 資料のPDF版はWEBページに掲載します。
  - https://www.cc.u-tokyo.ac.jp/events/lectures/242/
  - アンケートへの協力をお願いします。

補足スライド

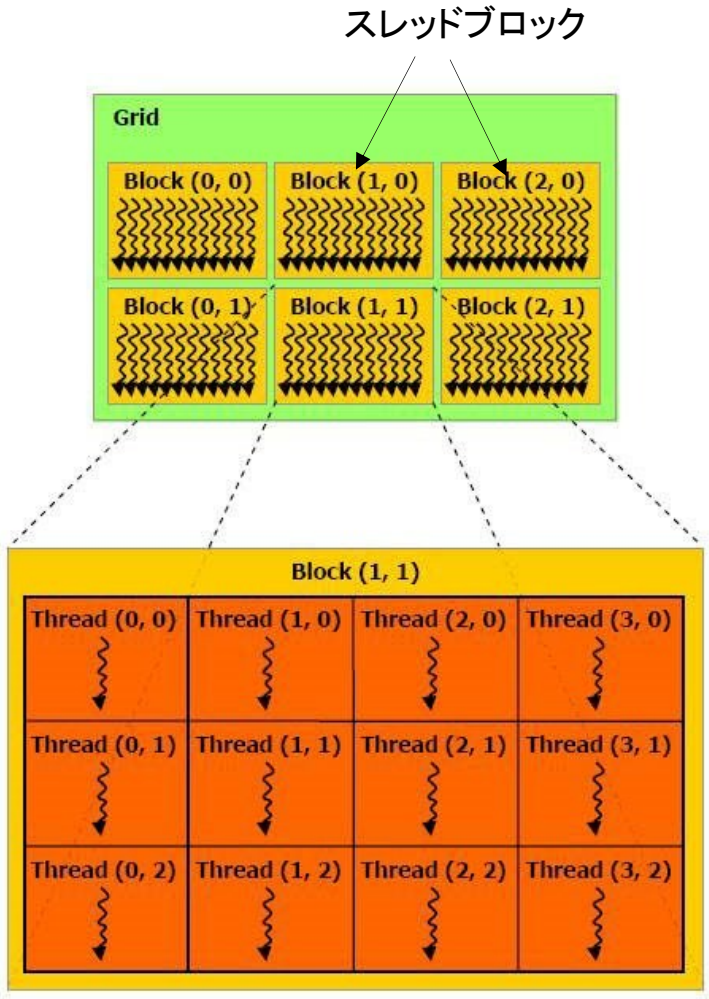
性能を出すためにはスレッド数>>コア数

- 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

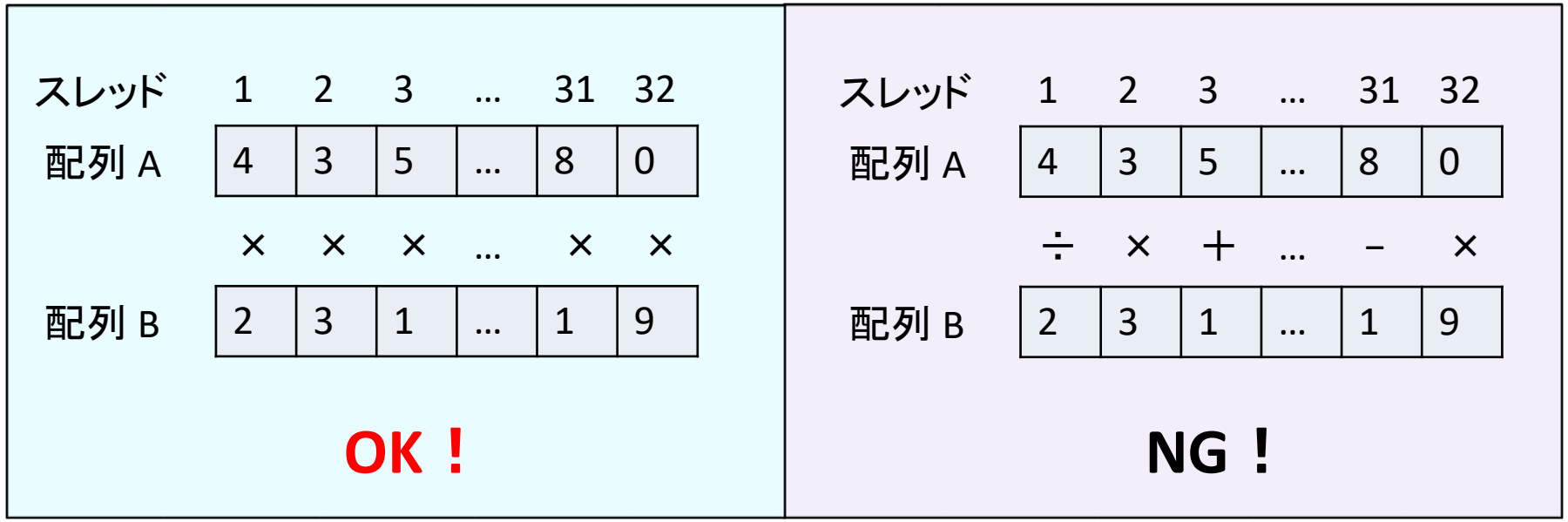
- 階層的なコア/スレッド管理
  - P100は56 SMを持ち、1 SMは64 CUDA coreを持つ。トータル3584 CUDA core
  - 1 SMが複数のスレッドブロックを担当し、1 CUDA core が複数スレッドを担当
- スレッド間のコミュニケーション
  - 同一スレッドブロック内のスレッドは高速コミュニケーション可能
  - 異なるスレッドブロックに属するスレッド間はコミュニケーションが低速
    - いったんメモリに書き出したり、CPUに処理を戻さなくてはならない



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

Warp 単位の実行

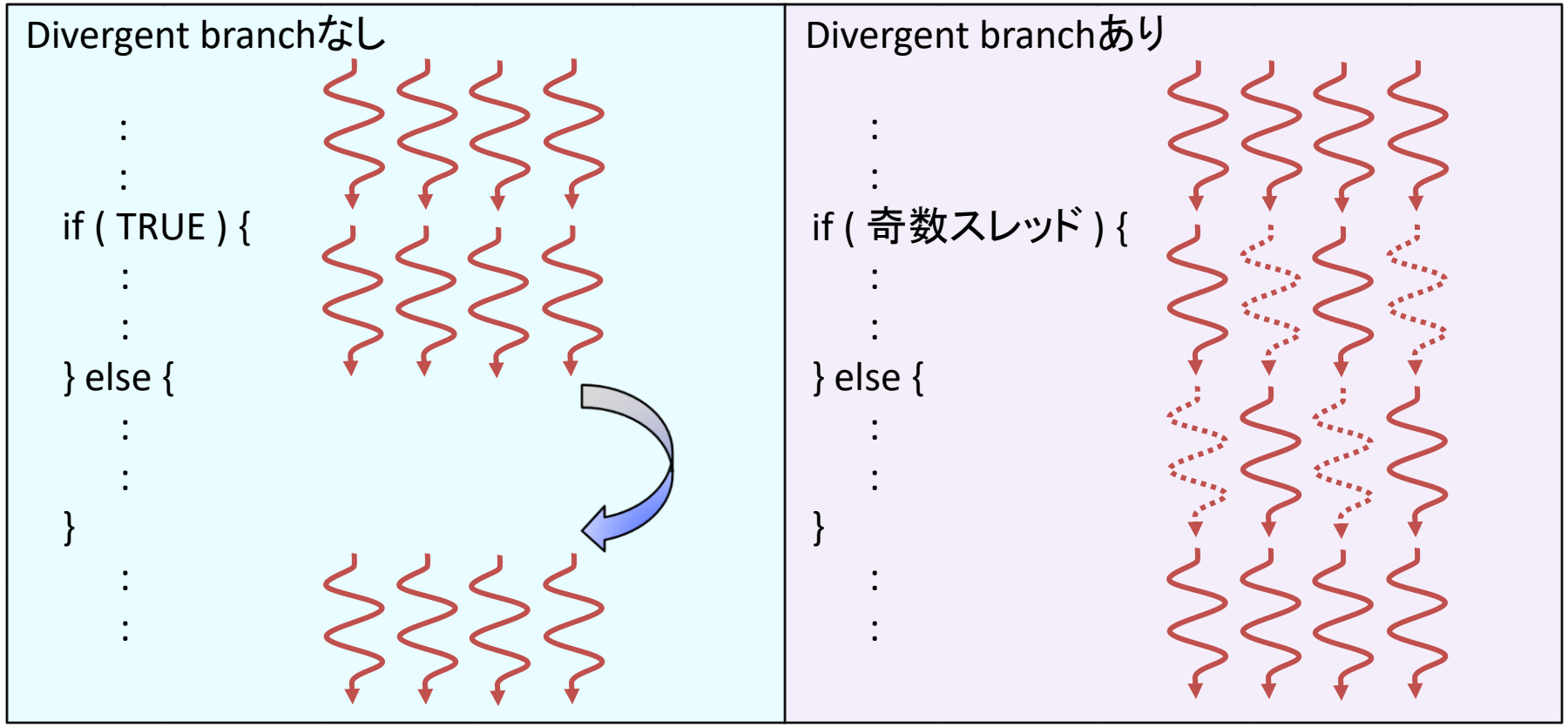
- 連続した32スレッドを1単位 = Warp と呼ぶ
- このWarpは足並み揃えて動く
  - 実行する命令は32スレッド全て同じ
  - データは違ってもいい



Warp内分岐

CUDA 8 以前のバージョン
CUDA 9 以上では多少マシになるが、
ペナルティがあることに変わりはない

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



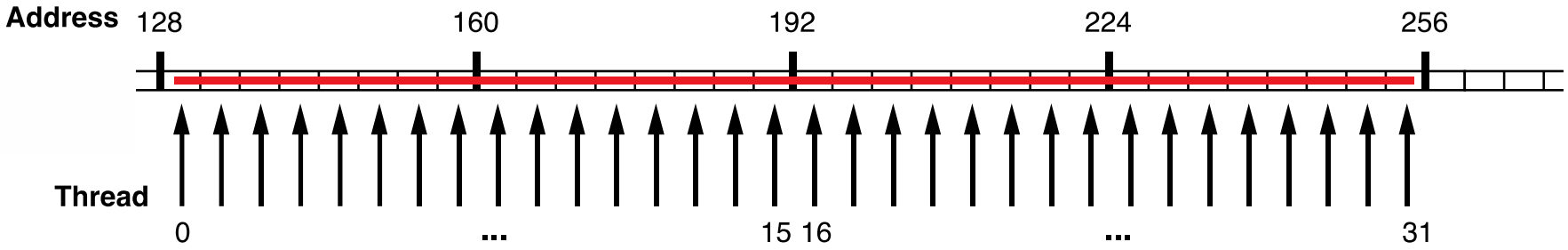
else 部分は実行せずジャンプ

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

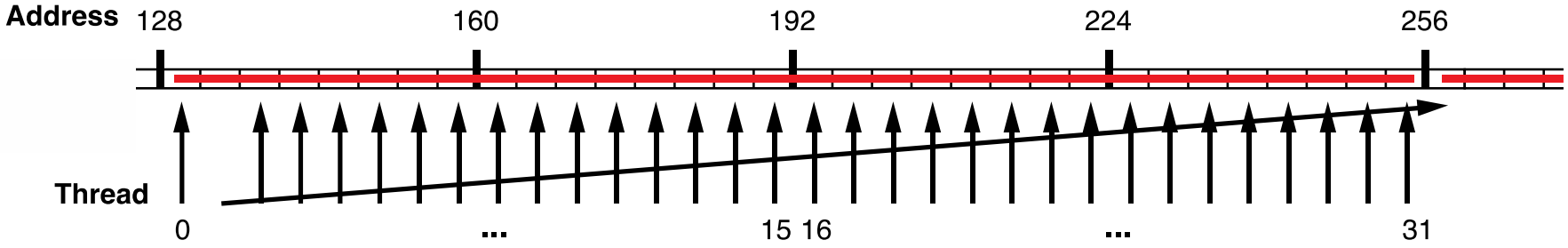
コアレスドアクセス

- 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