第157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

東京大学 情報基盤センター

担当：星野哲也

hoshino @ cc.u-tokyo.ac.jp

（内容に関するご質問はこちらまで）

講習会スケジュール

- 開催日時
  - ✓ 6月9日（水）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

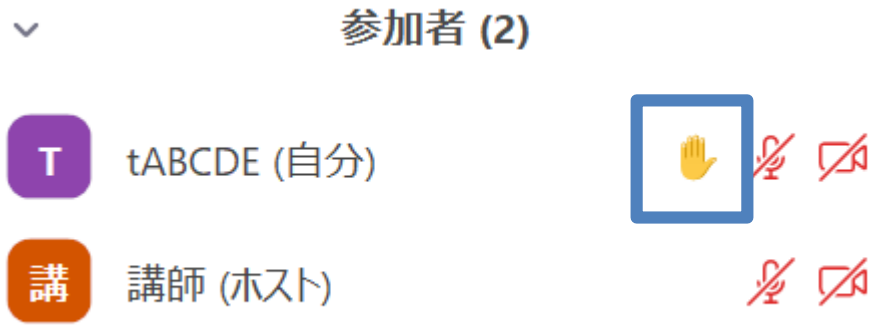
1. Zoomメニュー中の「リアクション」をクリック

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157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157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

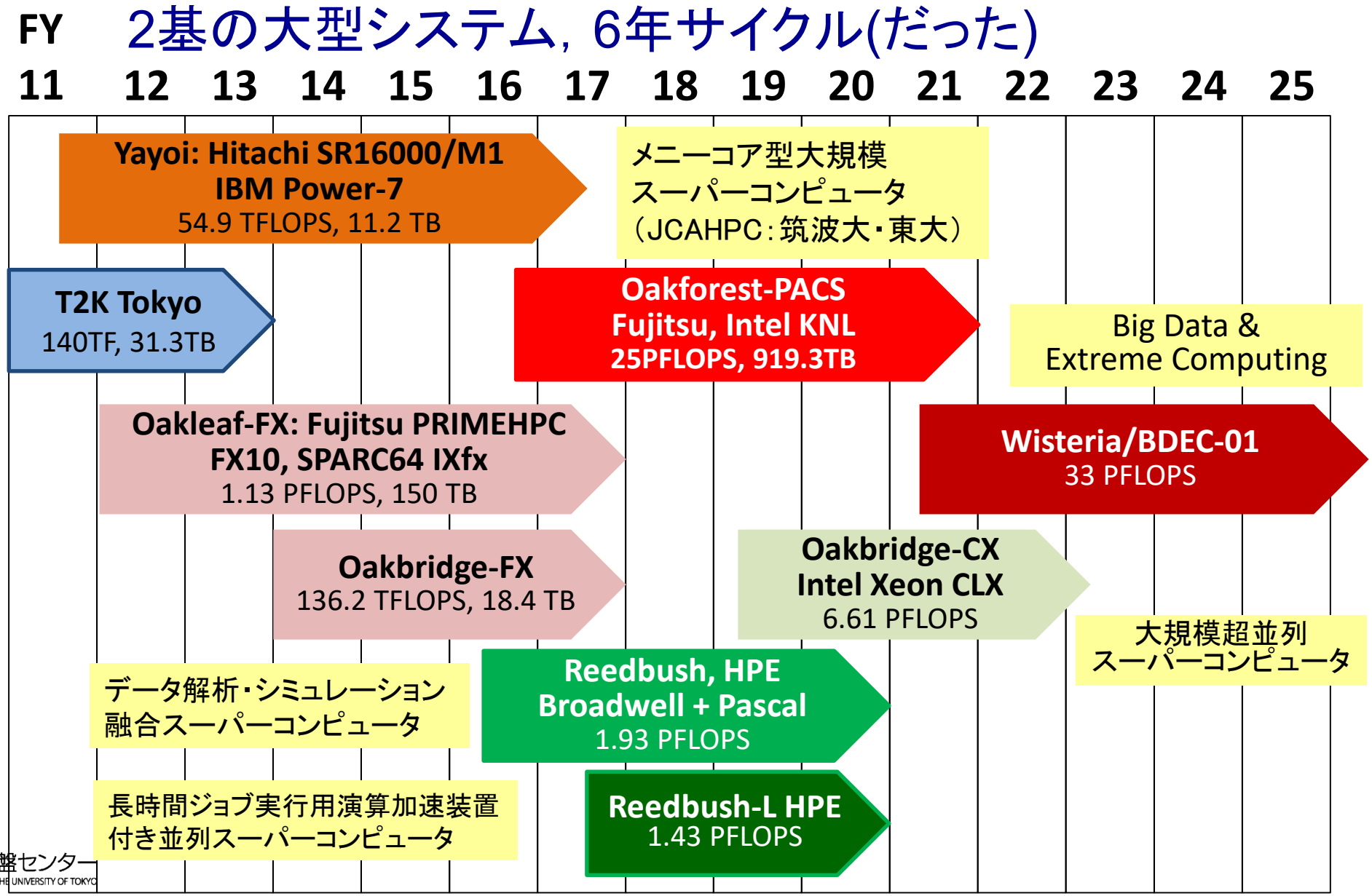
Slack: メッセージの入力方法

- 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

東大情報基盤センターの
スパコン概要

東大センターのスパコン



既存3システム：利用者2,000+，学外50+%

- 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
  - 3.36 PF, 2016年7月～ **2021年11月末**
  - 東大ITC初のGPUシステム (2017年3月より), DDN IME (Burst Buffer)

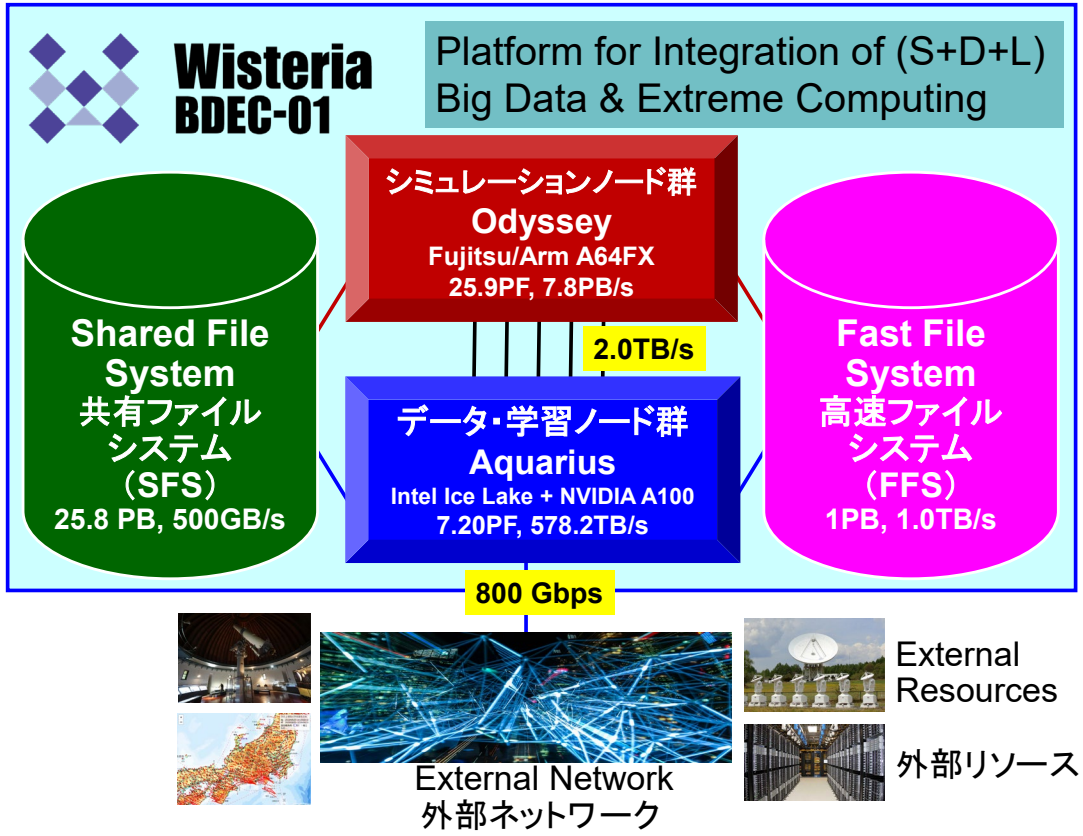
- Oakforest-PACS (OFP) (富士通, Intel Xeon Phi (KNL))
  - 2016年10月～**2022年3月末**
  - JCAHPC (筑波大CCS＆東大ITC)
  - 25 PF, TOP 500で6位 (2016年11月)（日本1位）（初登場時）
  - Omni-Path アーキテクチャ, DDN IME (Burst Buffer)

- Oakbridge-CX (富士通, Intel Xeon Platinum 8280)
  - 大規模超並列スーパーコンピュータシステム
  - 6.61 PF, 2019年7月 ～ 2023年6月
  - 全1,368ノードの内128ノードにSSDを搭載

Wisteria/BDEC-01

BDEC:「計算・データ・学習（S+D+L）」融合のためのプラットフォーム（Big Data & Extreme Computing）

- 2021年5月14日運用開始
  - 東京大学柏Ⅱキャンパス
- 33.1 PF, 8.38 PB/sec.，**富士通製**
  - ~4.5 MVA（空調込み），~360m^2
- Hierarchical, Hybrid, Heterogeneous (h3)
- **2種類のノード群**
  - **シミュレーションノード群 (S, SIM)：Odyssey**
    - **従来のスパコン**
    - **Fujitsu PRIMEHPC FX1000 (A64FX), 25.9 PF**
      - **7,680ノード（368,640 コア），20ラック，Tofu-D**
  - **データ・学習ノード群（D/L, DL）：Aquarius**
    - **データ解析，機械学習**
    - **Intel Xeon Ice Lake + NVIDIA A100, 7.2 PF**
      - **45ノード（Ice Lake：90基，A100：360基），IB-HDR**
    - **一部は外部リソース（ストレージ，サーバー，センサーネットワーク他）に直接接続**
  - ファイルシステム：共有（大容量）＋高速



Wisteria
BDEC-01

スパコン料金表（2021年4月時点 ）

- Wisteria/BDEC-01 は **2021/7/29 までは無料**で使えます（要申込）
  - https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/service/wisteria_test.php
- 最小セット料金表

	トークン※1	料金（大学・公共機関）	ストレージ容量	利用期間
Wisteria/BDEC-01	720 ※2	5,000円	2TB	2021/8/2以降～年度末まで
Reedbush	720 ※3	6,300円	1TB	2021/11/30まで
Oakforest-PACS	720	4,200円	1TB	年度末まで **(2021年度で運用終了予定)**
Oakbride-CX	720	8,400円	4TB	年度末まで

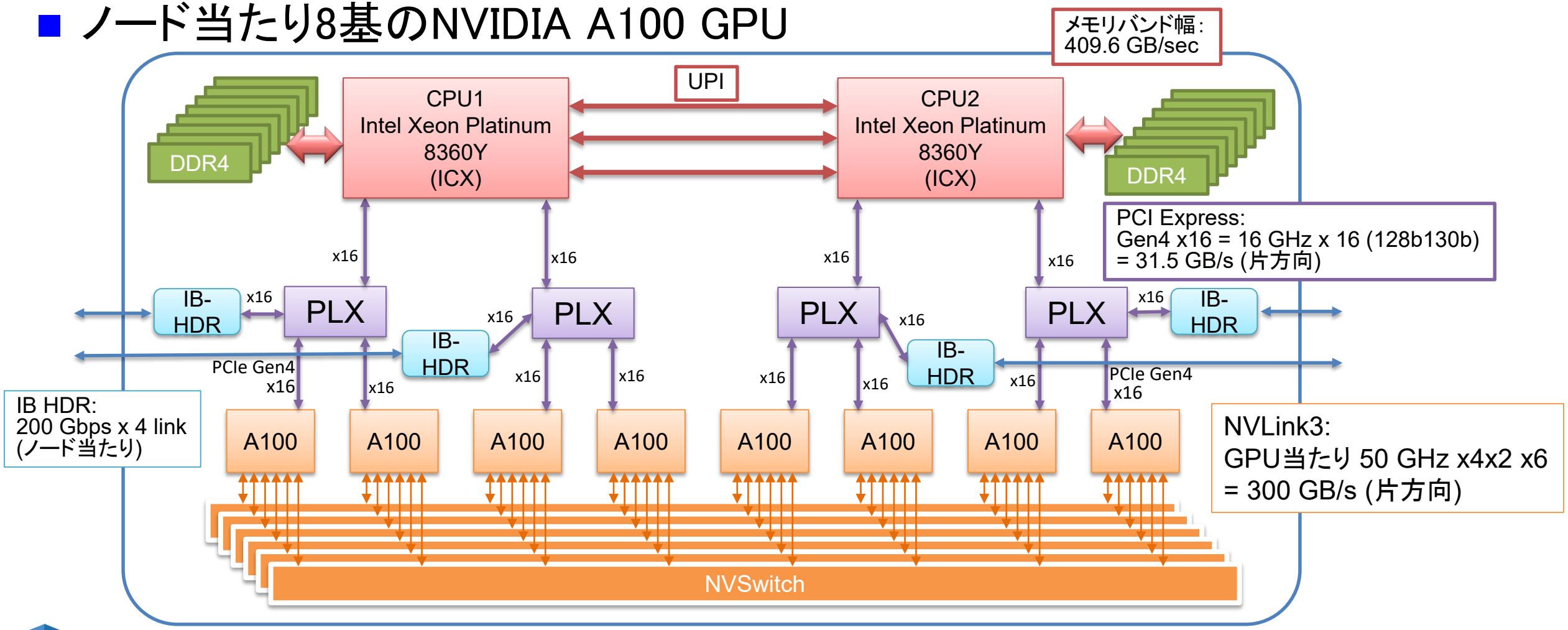
※1 トークン≒ノード時間 ※2, 3。720トークンなら、1ノードを720時間利用できる。

※2 Odyssey(CPUノード)基準。Aquarius(GPUノード)を使う場合、利用するGPU数x3.0倍のトークンを消費する。つまり1ノード(8GPU)を利用する場合、720/24=30時間しか利用できない。

※3 Reedbush-H,L はそれぞれ2.5, 4.0倍のトークンを消費する。つまり、Hの場合は720/2.5=288時間, Lの場合は720/4=180時間しか利用できない

Aquariusの構成

- Intel Xeon Platinum 8360Y (36c 2.4GHz) x 2ソケット, 512GBメモリ
- ノード当たり8基のNVIDIA A100 GPU



Wisteria 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 /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

Wisteria 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。

Wisteria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 ./〇〇.sh

- 投入されたジョブを確認する。（qstatではないので注意）

  $ pjstat

- 実行が終了すると、以下のファイルが生成される。

  〇〇.sh.??????.out

  〇〇.sh.??????.err　（?????? 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.??????.out

- 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 〇〇.sh.??????.out

コンパイラの種類と実行(Aquarius)

- ログインノードとAquarius計算ノードとでは、CPUの命令セットが(ほぼ)同じ
  - ログインノード: 命令セットアーキテクチャ Intel CascadeLake＋AVX512, **x86_64**
  - Aquarius計算ノード: 命令セットアーキテクチャ Intel IceLake＋AVX512, **x86_64**
- 様々なコンパイラが利用可能: GPU向けには gcc+CUDAか NVIDIAを推奨

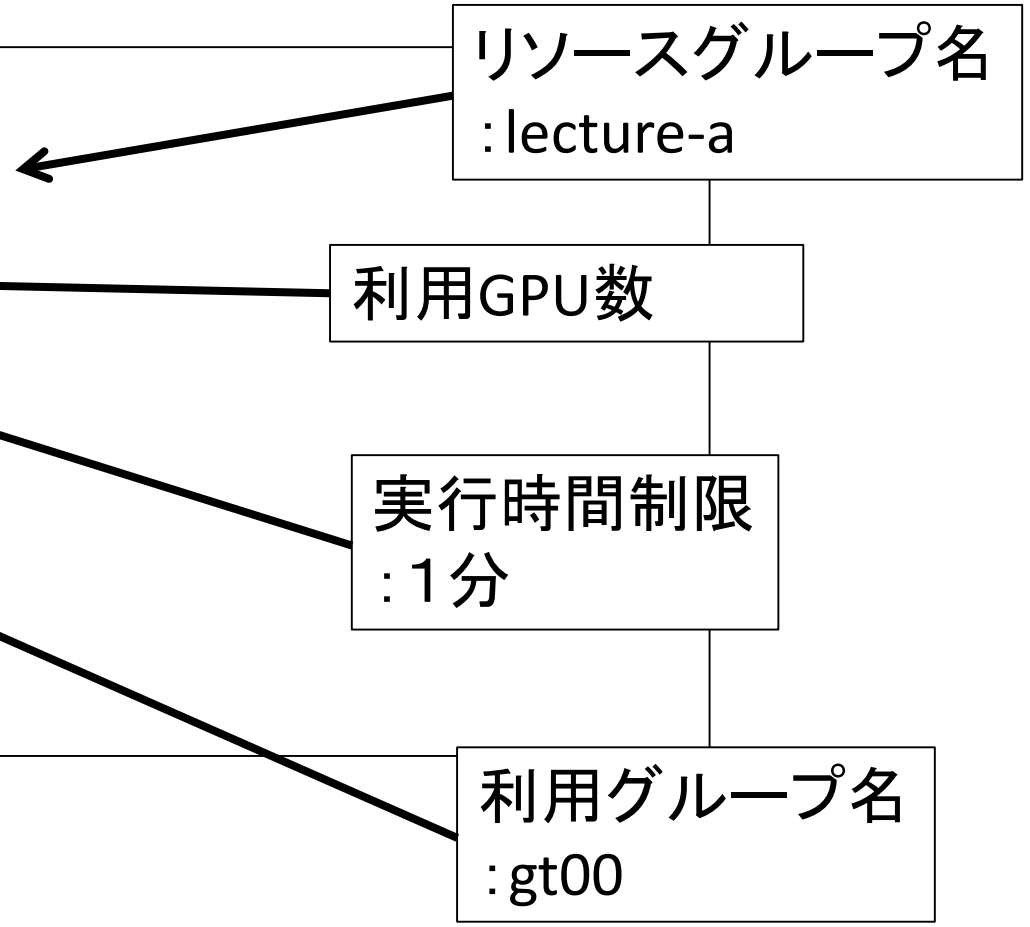
```
$ module load gcc cuda/11.2 ompi-cuda/4.1.1-11.2   または
$ module load nvidia/21.3 ompi-cuda/4.1.1-11.2
```

言語	GNUコンパイラ	Intelコンパイラ	NVIDIA コンパイラ (旧PGI)	CUDAコンパイラ
C	gcc	icc	nvc (pgcc)	nvcc
C++	g++	icpc	nvc++(pgc++)	
Fortran	gfortran	ifort	nvfortran (pgfortran)	
OpenACC			○	

JOBスクリプトサンプルの説明（Aquarius, MPIなし）

```
#!/bin/bash
#PJM -L rscgrp=lecture-a
#PJM -L gpu=4
#PJM -L elapse=00:01:00
#PJM -g gt00

module load nvidia
./a.out
```



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！

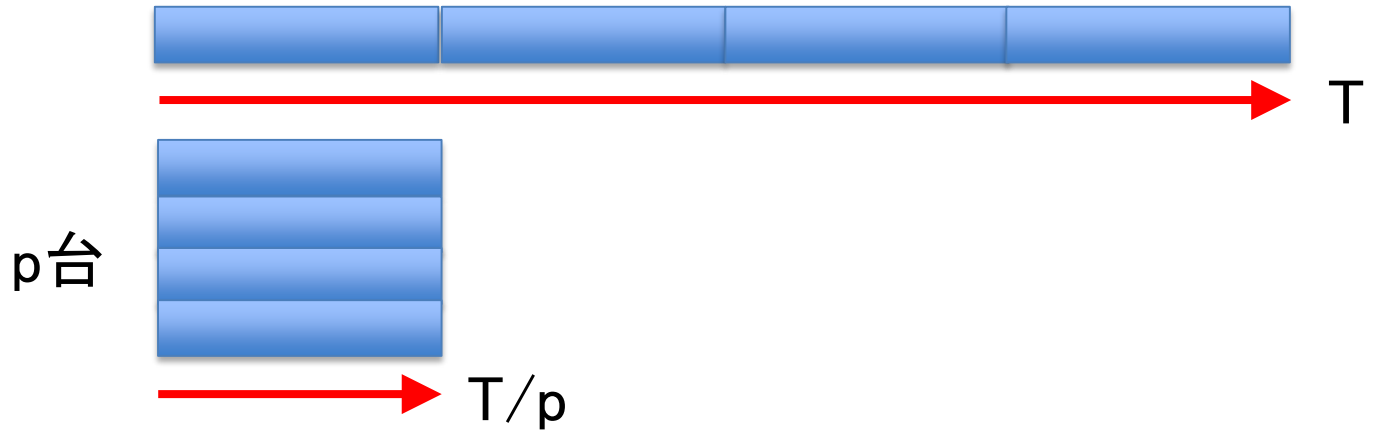
- GPUは**並列計算機**です！よって本講習会で学ぶのは**並列プログラミング**になります！
  - 並列プログラミングの例：MPI, OpenMP など
- 並列プログラミングは、**プログラムを高速化**するために行います！

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。

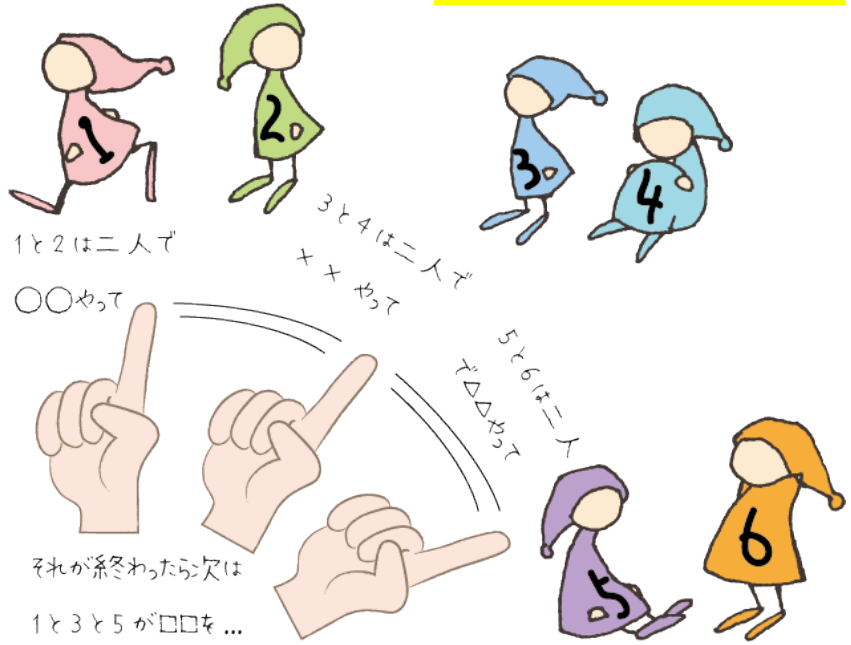


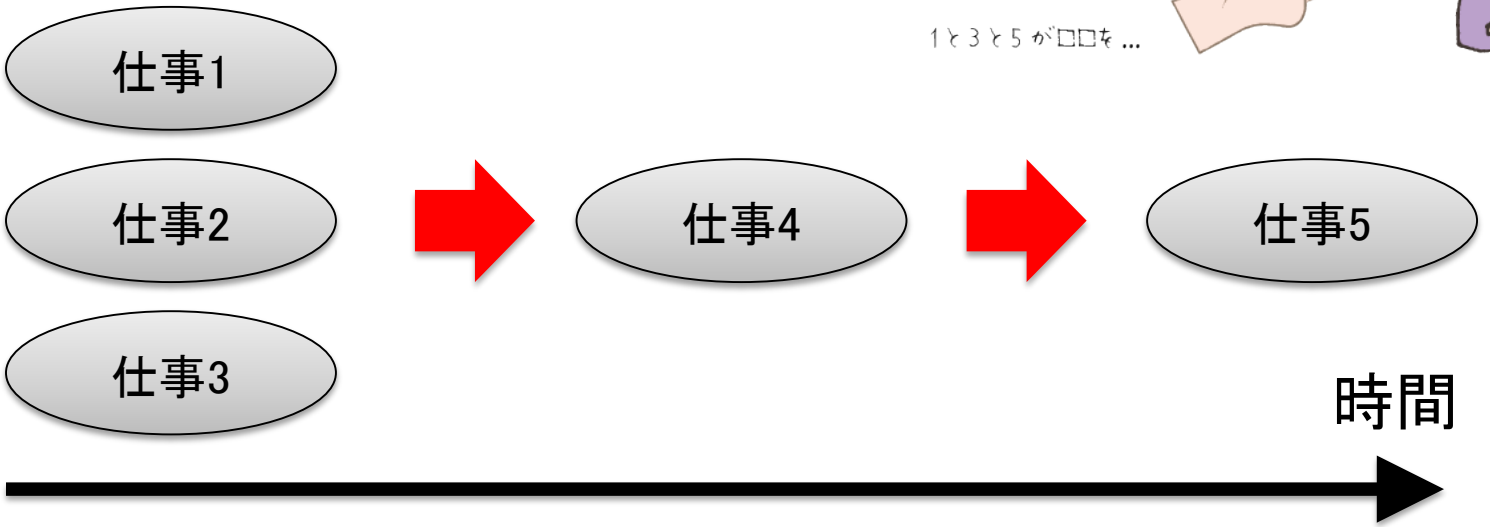
- 実際にできるかどうかは、処理内容（アルゴリズム）による。アルゴリズムによって難易度は異なる。
  - ✓ 並列化できないアルゴリズム、通信のオーバーヘッド
  - ✓ 部分的にでも並列化できないアルゴリズムがあると、どれだけ並列数を上げても、その時間は短縮されない。
- 並列処理（計算）の種類
  - ✓ 「タスク並列」と「データ並列」

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

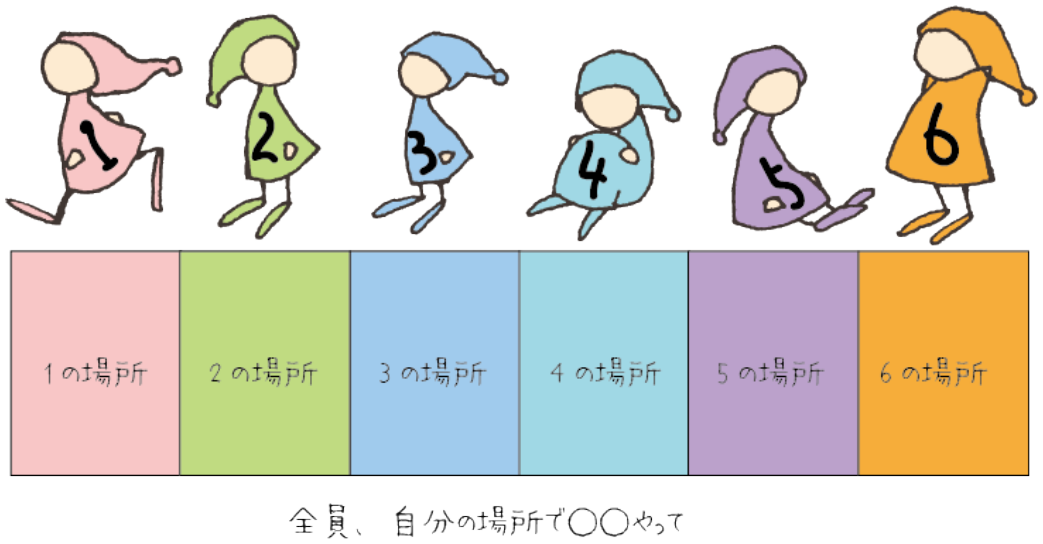




データ並列

- 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。

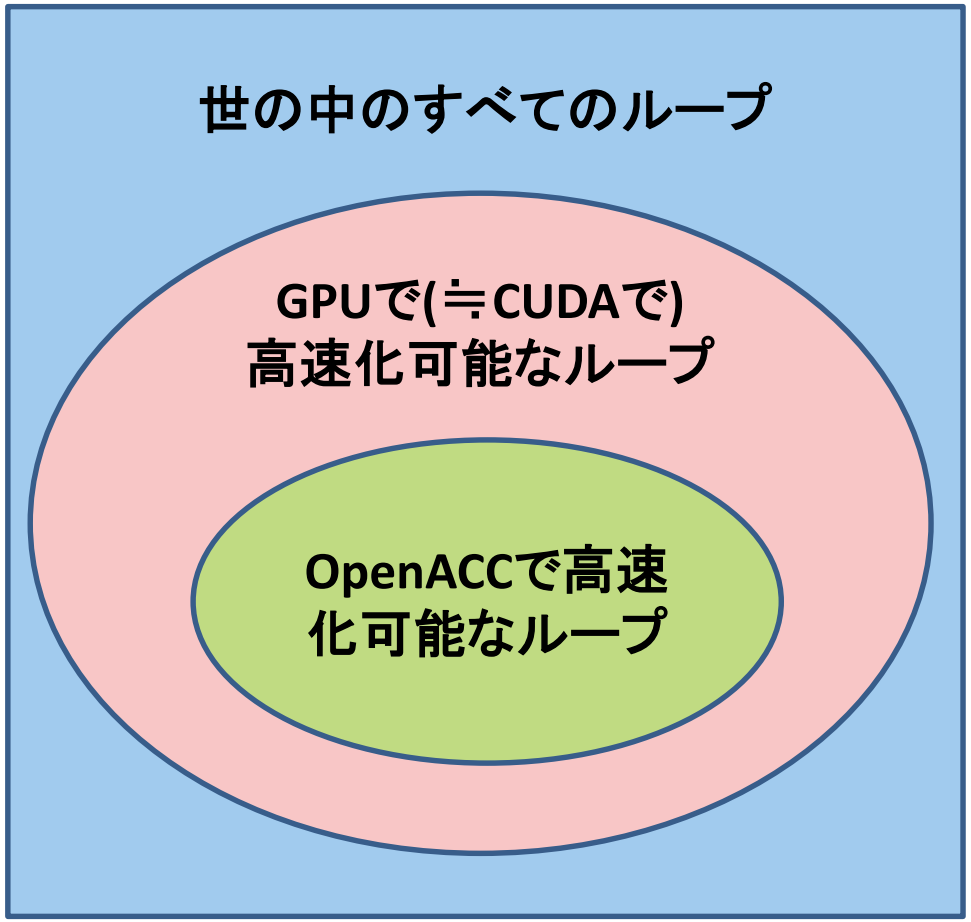
2 + 1 =	12 + (−88)=
3 + 19 =	−20 + 29 =
4 + (−6) =	4 + 10 =
−8 + 10 =	−32 + 12 =
10 + 3 =	−5 + 5 =



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

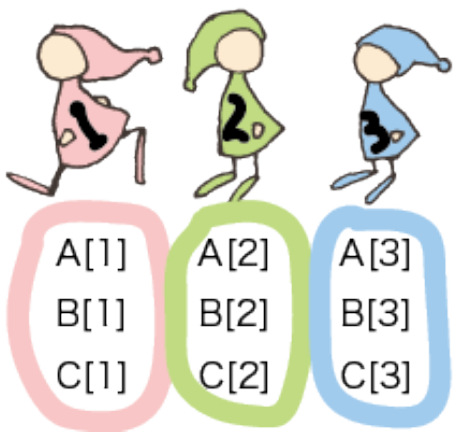
```
B[0] = 0;
for ( i = 0; i < N; i++)
  B[i+1] = B[i] + A[i];
```

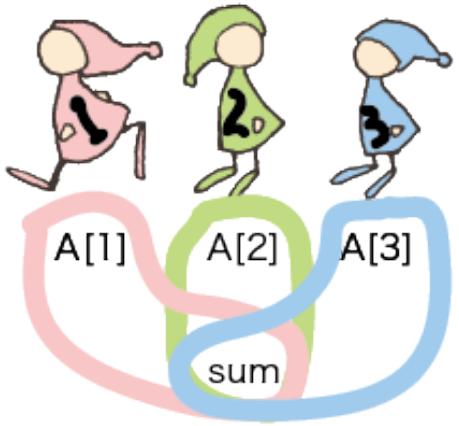
※OpenACCでも、GPUで正しく動くコードを書くことはできる。しかし遅いので意味がない

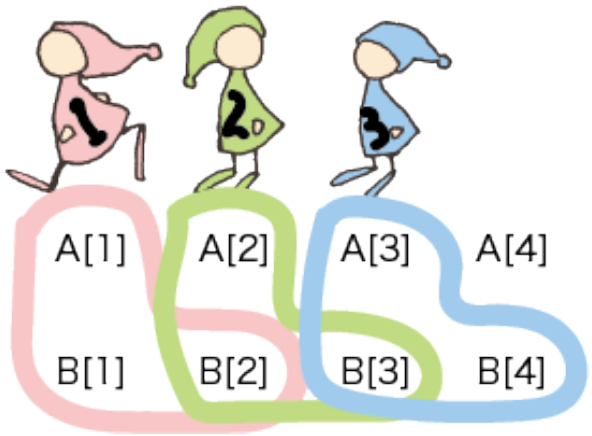
OpenACCで**簡単に並列化**できる

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）



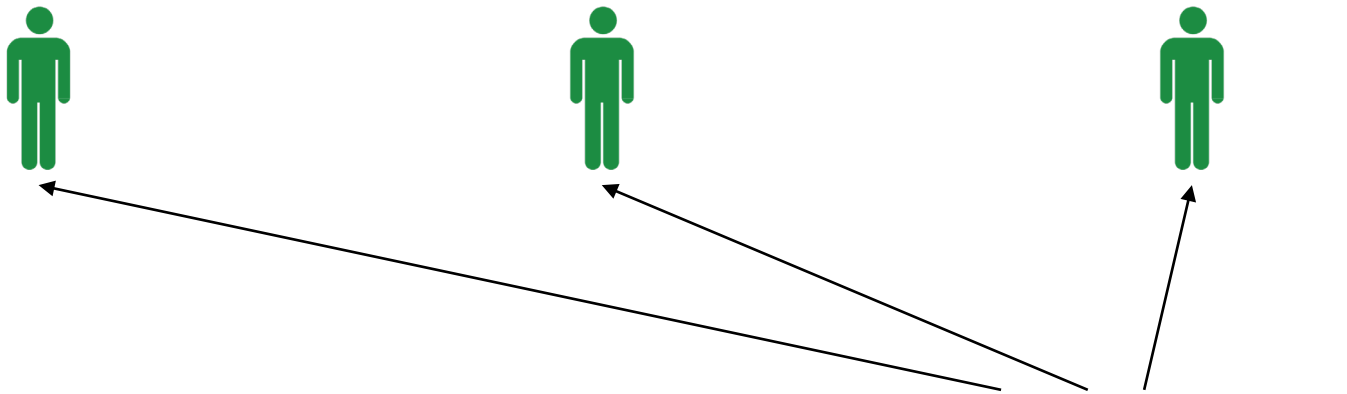




簡単に並列化できるループ

```
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

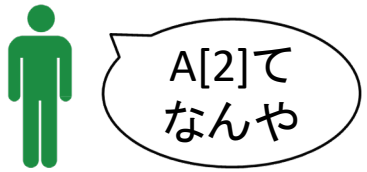
簡単に並列化できるループ

```
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;

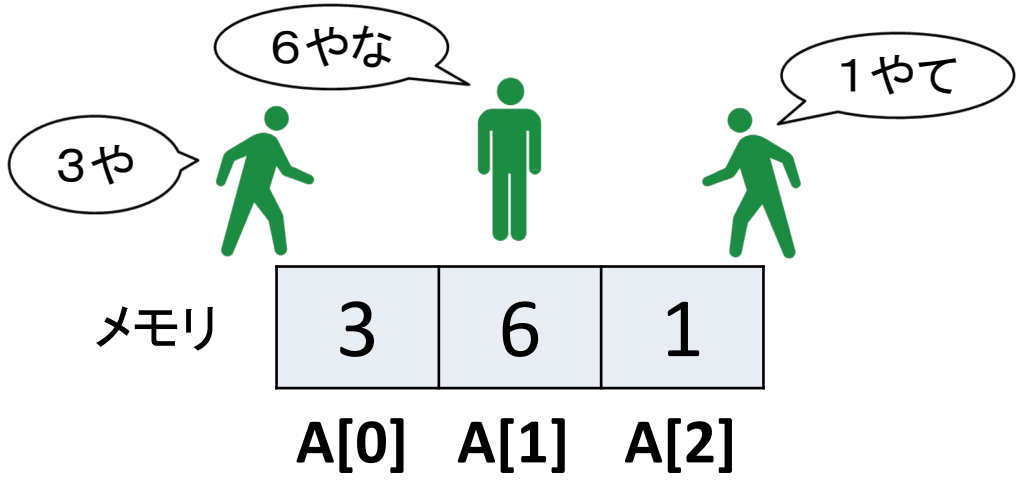
メモリ

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;
```

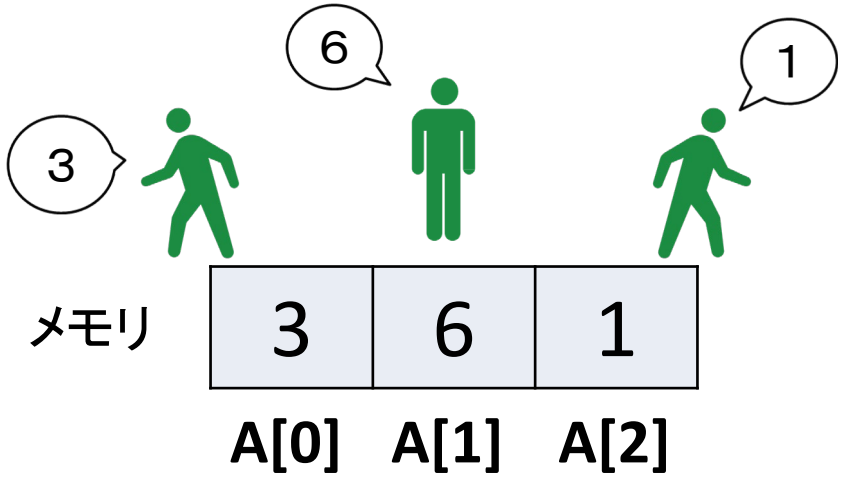
A[0] = A[0] + 1;　　**A[1] = A[1] + 1;**　　**A[2] = A[2] + 1;**



簡単に並列化できるループ

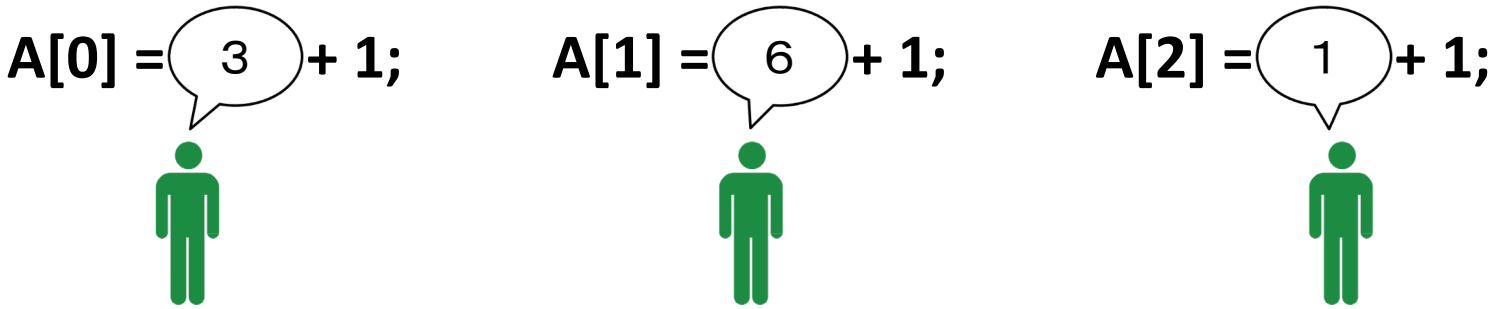
```
for ( i = 0; i < 3; i++)
  A[i] = A[i] + 1;
```

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



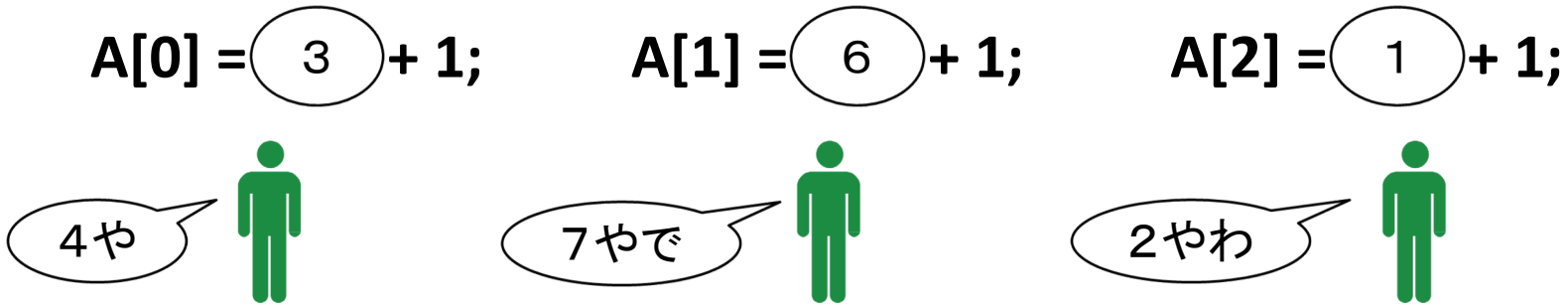
簡単に並列化できるループ

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



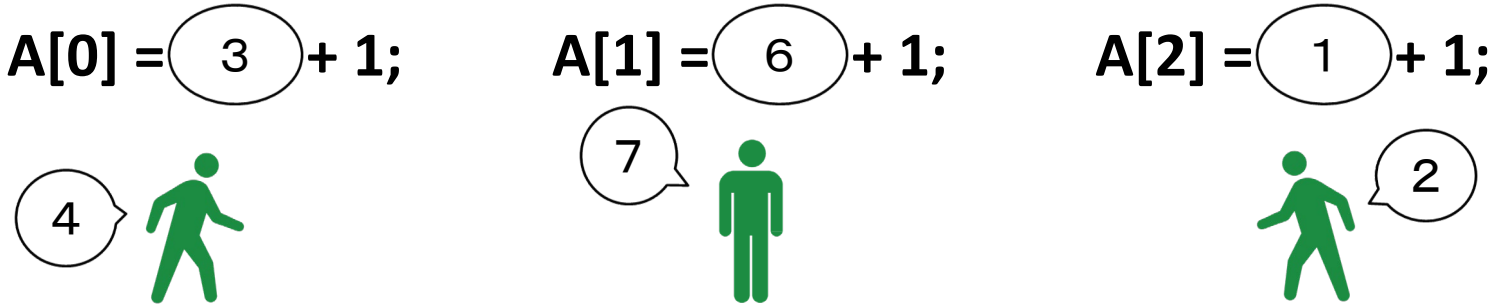
簡単に並列化できるループ

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



簡単に並列化できるループ

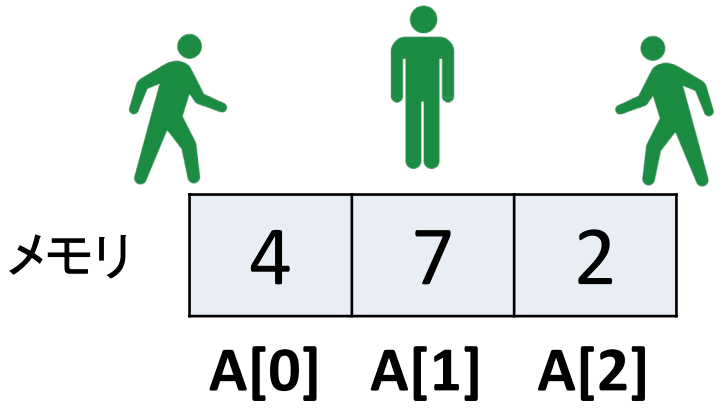
```
for ( i = 0; i < 3; i++)
  A[i] = A[i] + 1;
```



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(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。

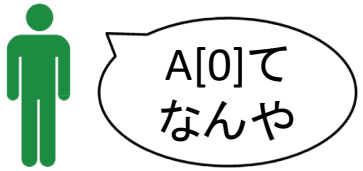
足し算なのでどんな順番で足しても
結果は変わらないはずだが…

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;

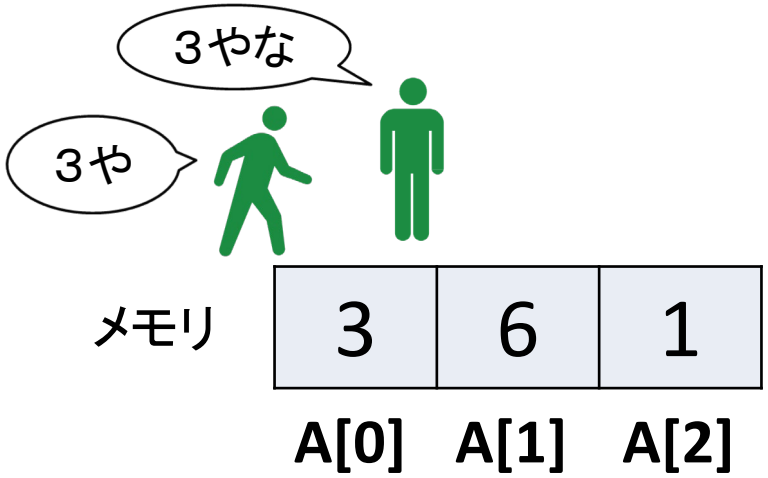
メモリ

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;
```

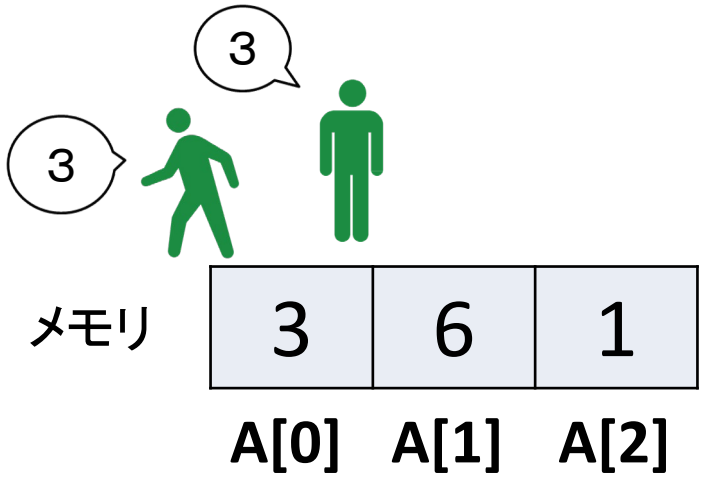
A[0] = A[0] + 1;　　A[0] = A[0] + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

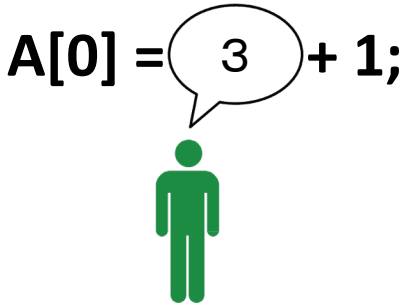
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```

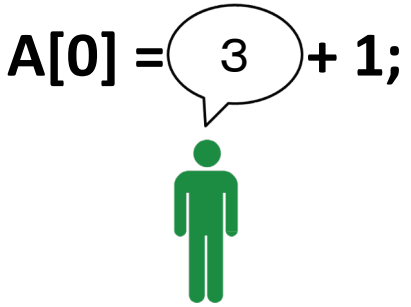
A[0] = A[0] + 1;　　A[0] = A[0] + 1;　　A[0] = A[0] + 1;

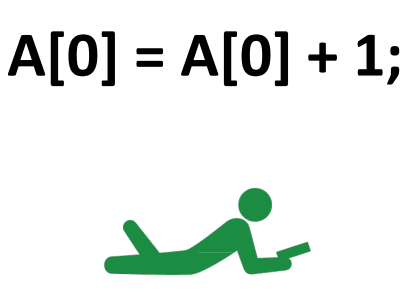


簡単に並列化できないループ

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

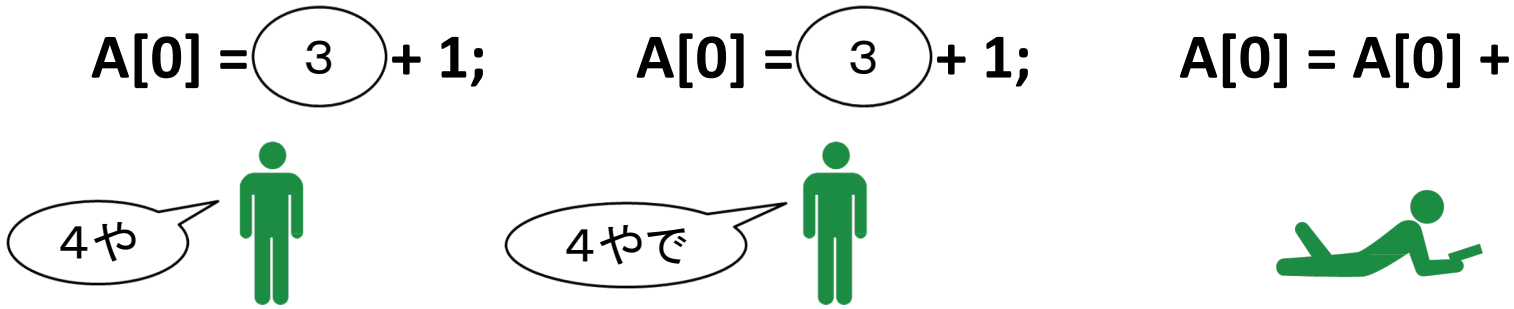






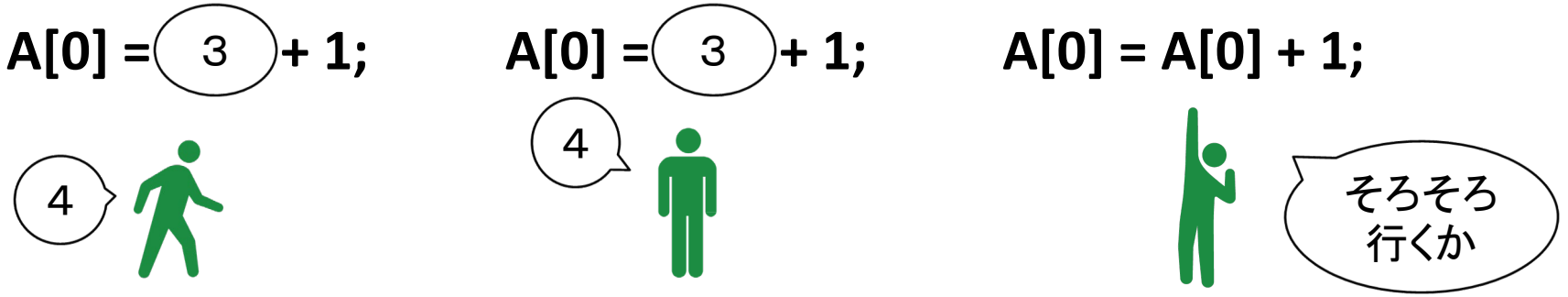
簡単に並列化できないループ

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



簡単に並列化できないループ

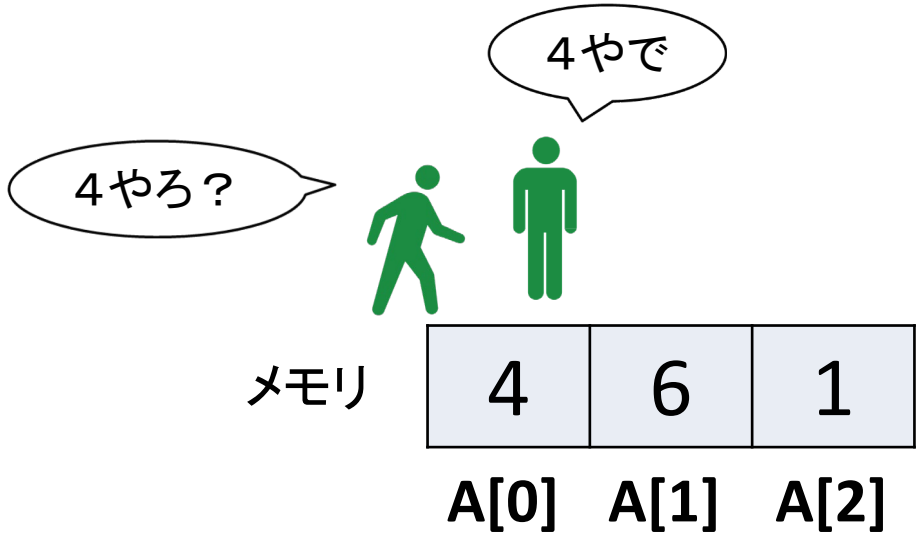
```
for ( i = 0; i < 3; i++)
  A[0] = A[0] + 1;
```



簡単に並列化できないループ

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

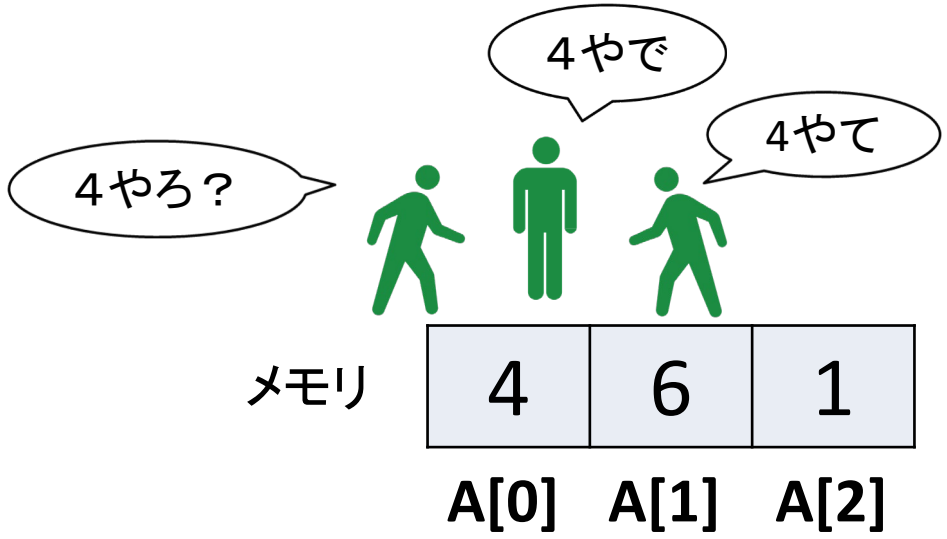
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

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

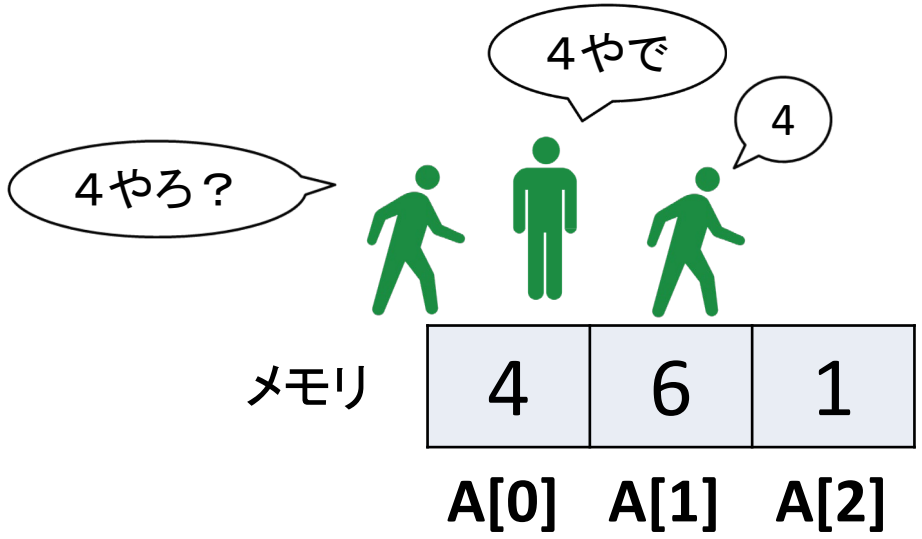
A[0] = (3) + 1;　　A[0] = (3) + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

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

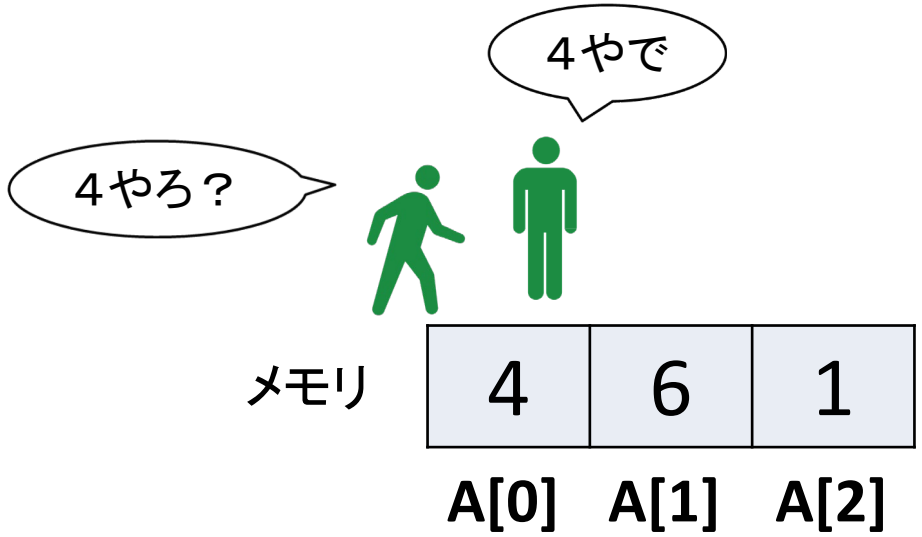
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = A[0] + 1;



簡単に並列化できないループ

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

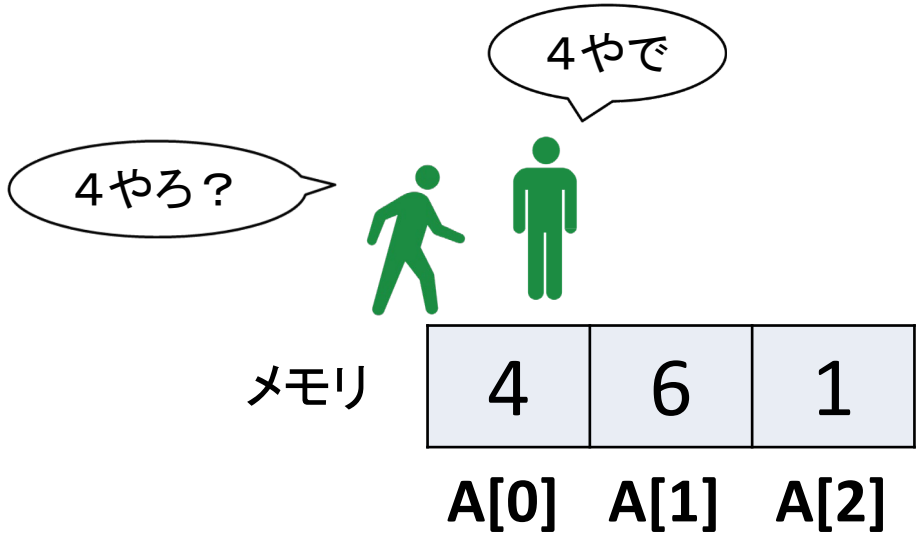
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = 4 + 1;



簡単に並列化できないループ

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

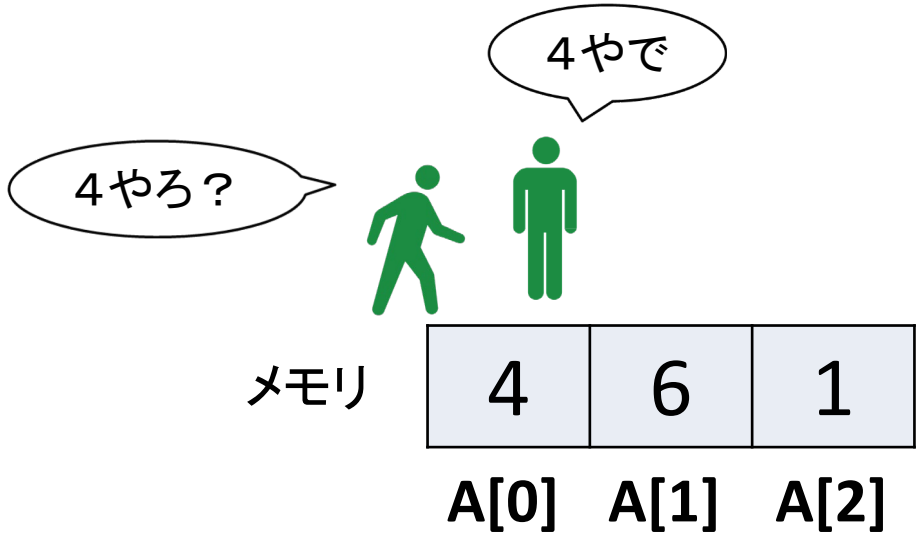
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = 4 + 1;



簡単に並列化できないループ

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

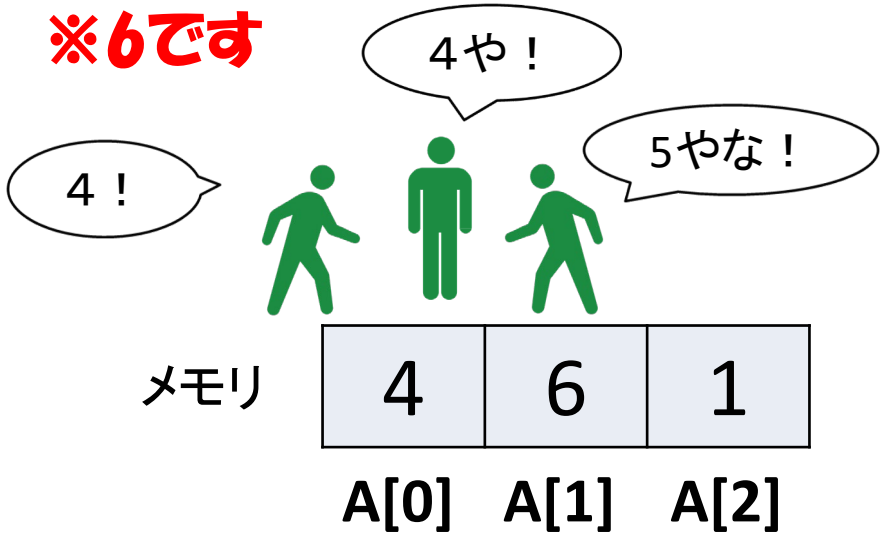
A[0] = 3 + 1;　　A[0] = 3 + 1;　　A[0] = 4 + 1;



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;



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。

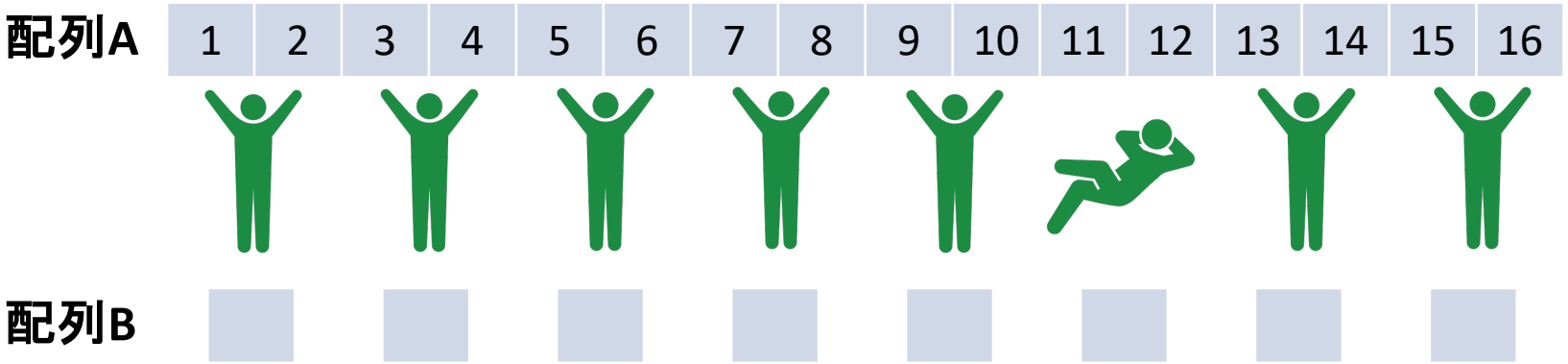
スレッドは各々独立に1～3を実行するため、タイミングによって結果が変わる！

（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

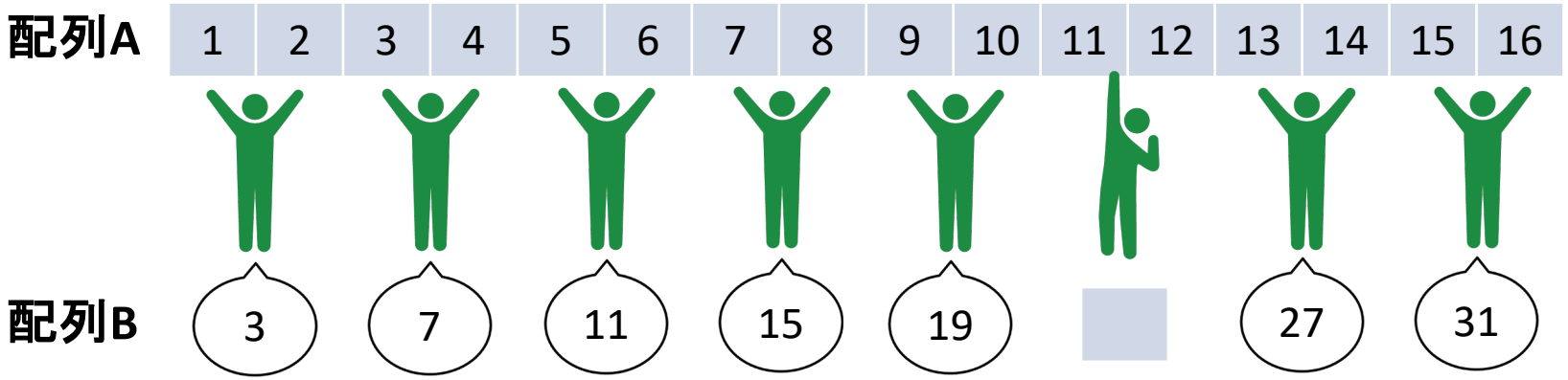
```
sum = 0;
for ( i = 0; i < 16; i++)
    sum = sum + A[i];
```



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];
```

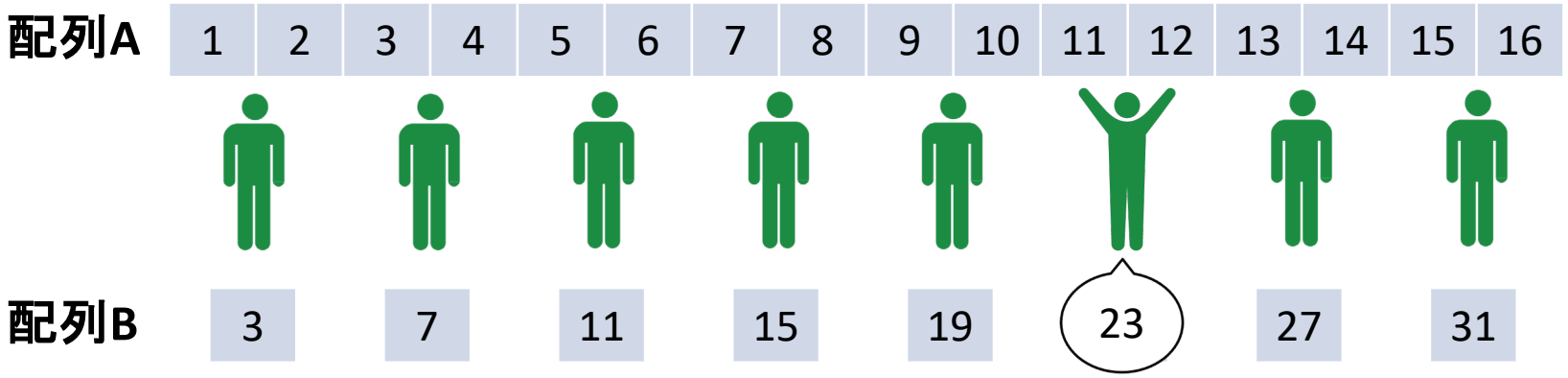


1. 各々自分の担当領域で足し算（結果を別の場所に保存）

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];
```

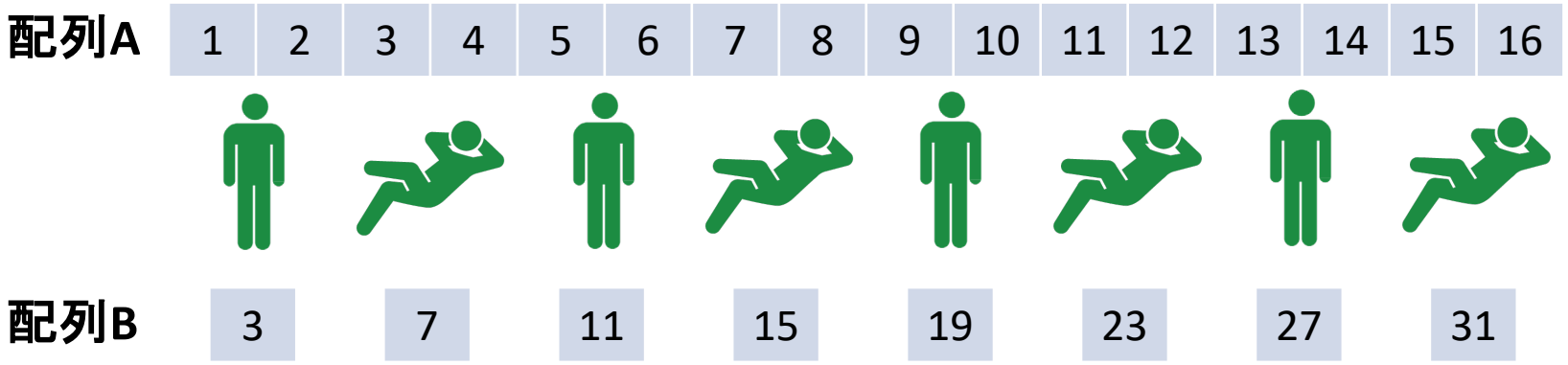


1. 各々自分の担当領域で足し算（結果を別の場所に保存）
2. 遅れているスレッドを待つ！（これを**同期(thread synchronization)**という）

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];
```

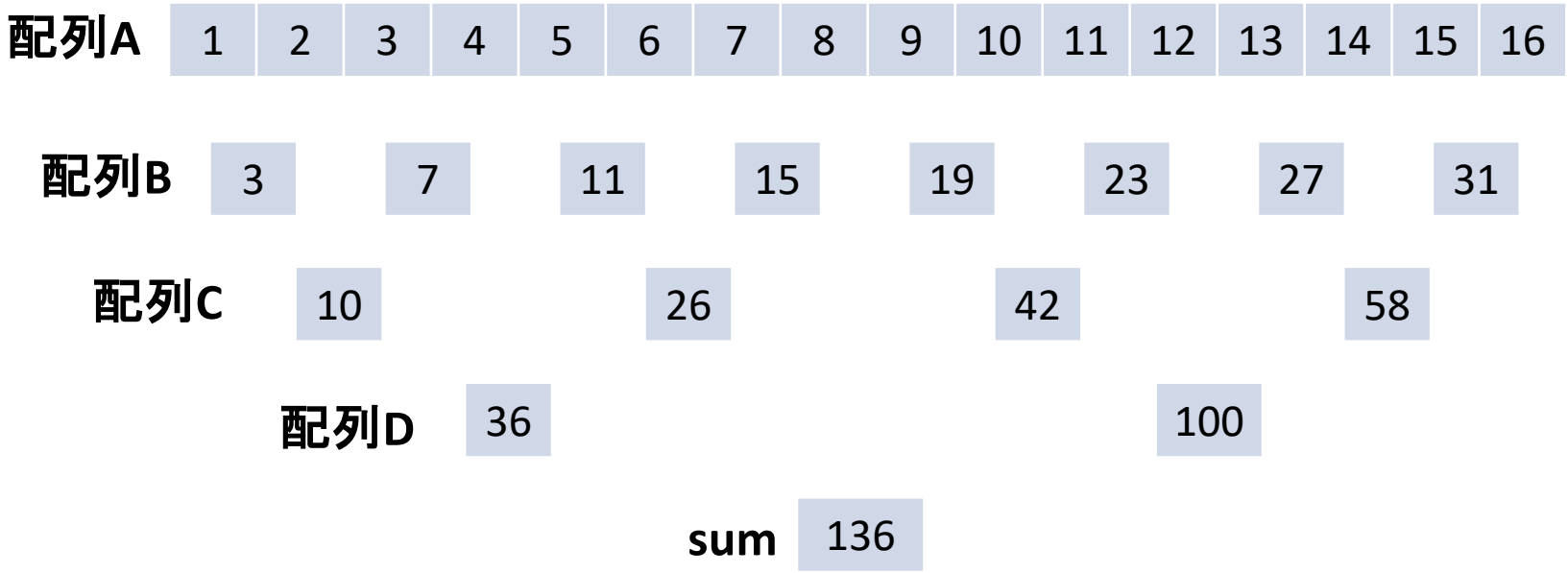


1. 各々自分の担当領域で足し算（結果を別の場所に保存）
2. 遅れているスレッドを待つ！（これを同期(thread synchronization)という）
3. 一部のスレッドを寝かせて、起きてるスレッドで(1)から繰り返し

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];
```



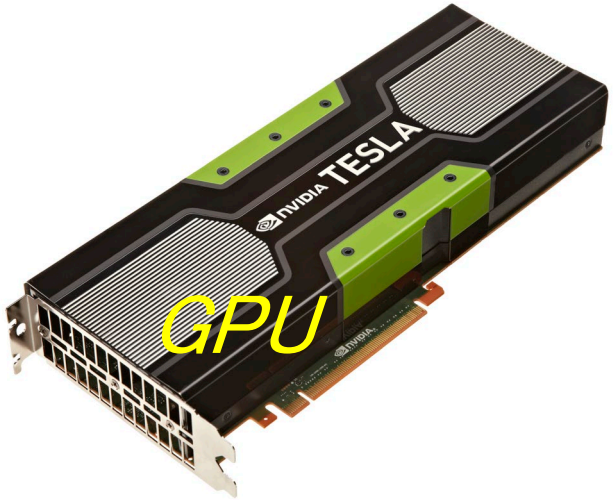
- これは一般的にリダクションと呼ばれる演算パターン
- 一番働くスレッドが4回の足し算。逐次の場合と比較して4倍の高速化
- メモリなどを介してスレッドの間でデータのやり取りをすることをスレッド間通信という

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
- パソコンの部品として量産されてる。
  = 非常に安価(だった)



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。

抑えておくべきGPUの特徴

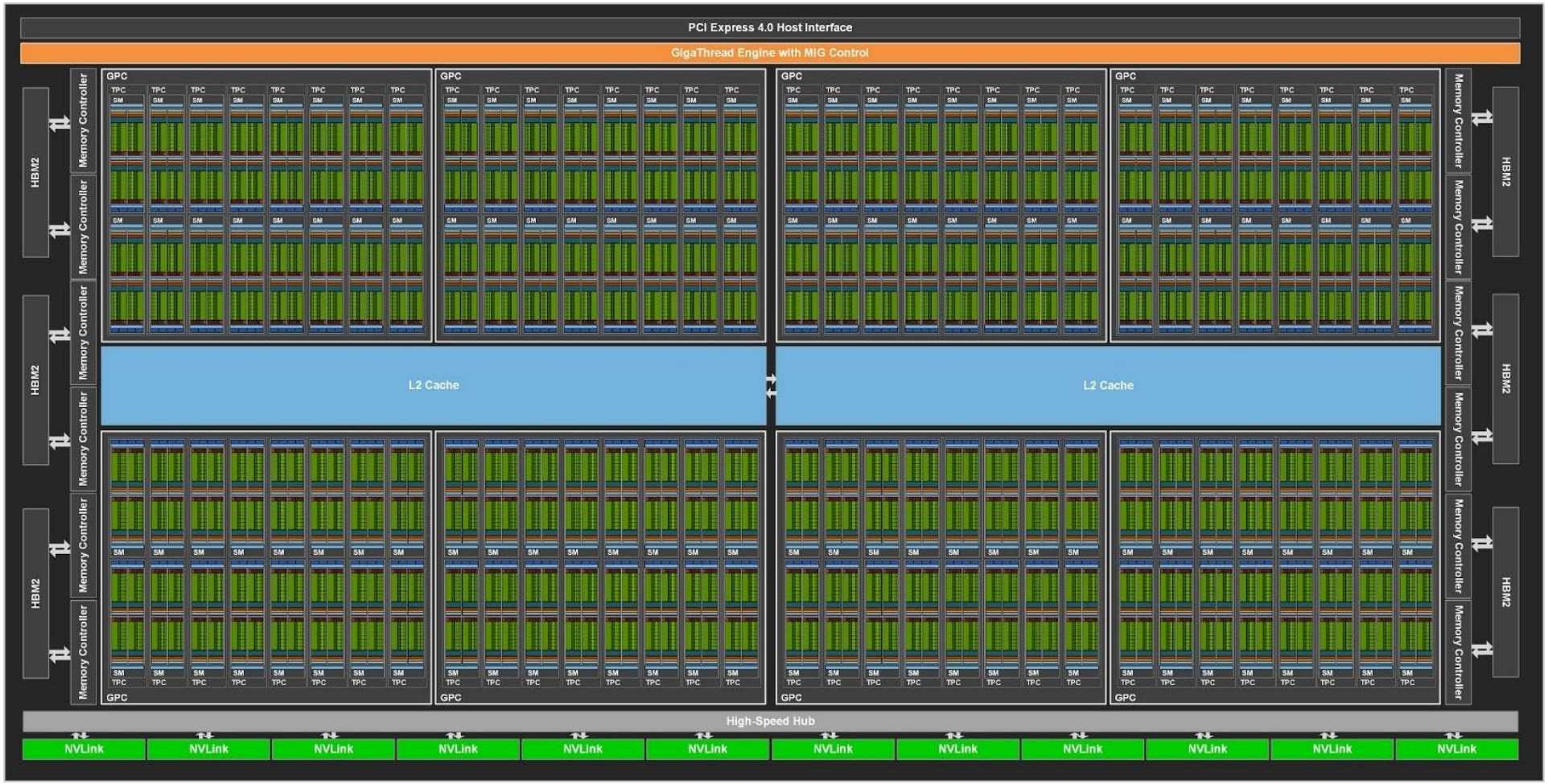
- 最低限知っておくべきこと
  - ✓ 超並列計算が必須！
    - 物理コア数が1000以上、論理コア数（スレッド）は数十万以上
    - プログラムの並列性（スレッド分割可能数）が小さいと速くならない
  - ✓ CPU と GPUの間でのデータ転送が必須！
    - GPU は CPU の指示なしでは動けない
    - CPU と GPU は独立に動く
      - ✓ CPUとGPUの同期を行い、データの一貫性を保つ必要がある

- 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 A100 Tensor Core GPU (1/2)

- 108 SM (Streaming Multiprocessor)



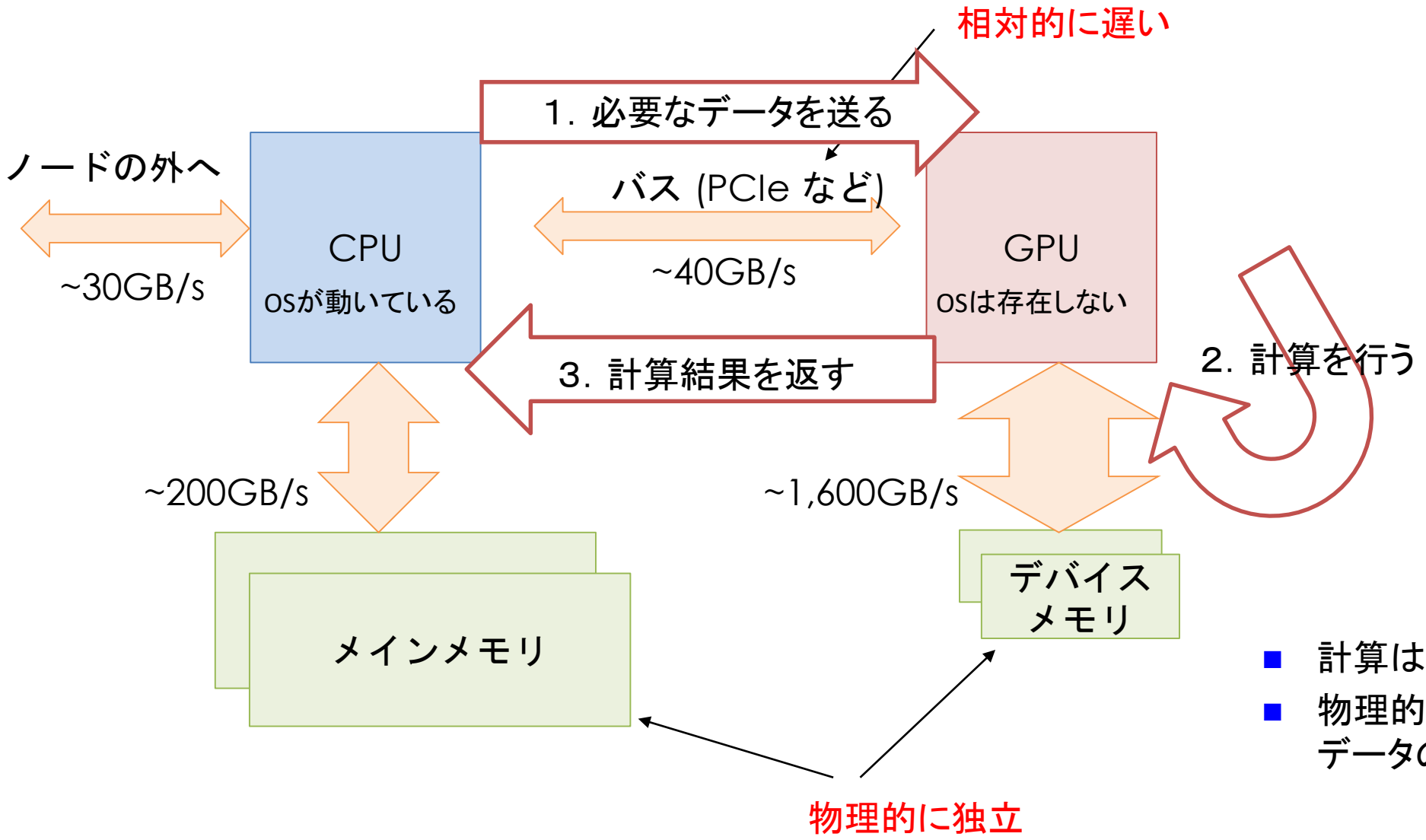
出典: NVIDIA A100 Tensor core GPUアーキテクチャ

NVIDIA A100 Tensor Core GPU (2/2)

- 倍精度にも対応したTensor Coreを搭載
  - 19.5 TF @ FP64, FP32
  - 156 TF @ TF32 (実質19bit)
  - 312 TF @ FP16, BF16
  - 624 TF @ INT8
  - 1248 TF @ INT4
- メモリ HBM2 40GB 搭載
  - 1.555 TB/s

出典: NVIDIA A100 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？

```
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
```

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

(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？

```
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

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

(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

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、最新は 3.1
  - ✓ コンパイラ:[商用]PGI → **NVIDIA HPC SDK**, Cray, [フリー]GCC(NVIDIA HPC SDKは無料版あり)
  - ✓ 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

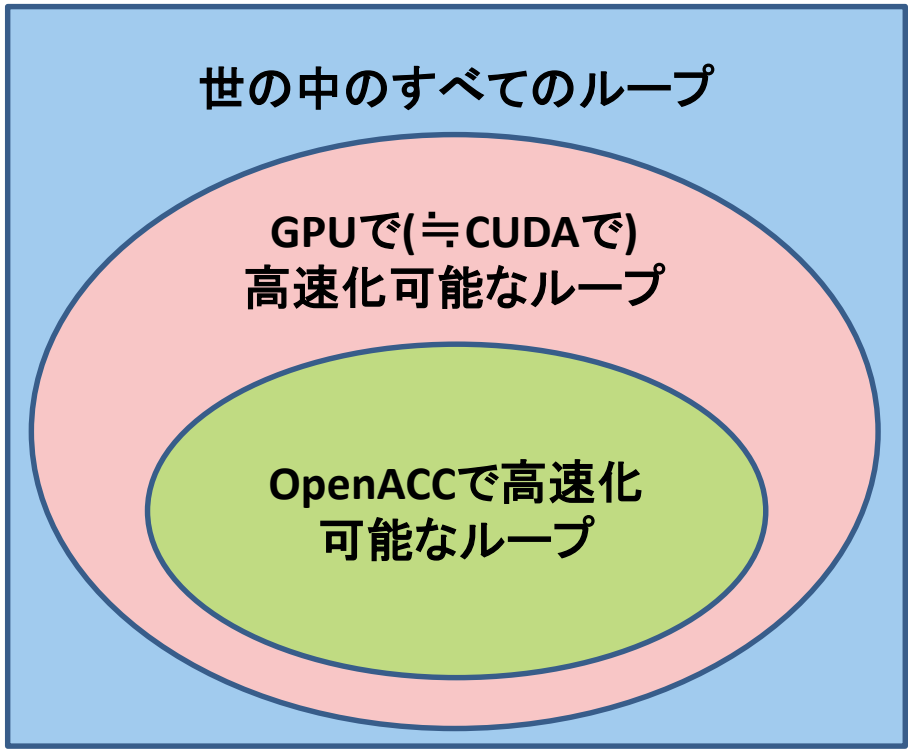
```
#pragma acc directive-name [clause, …]
{
    // C code
}
```

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**

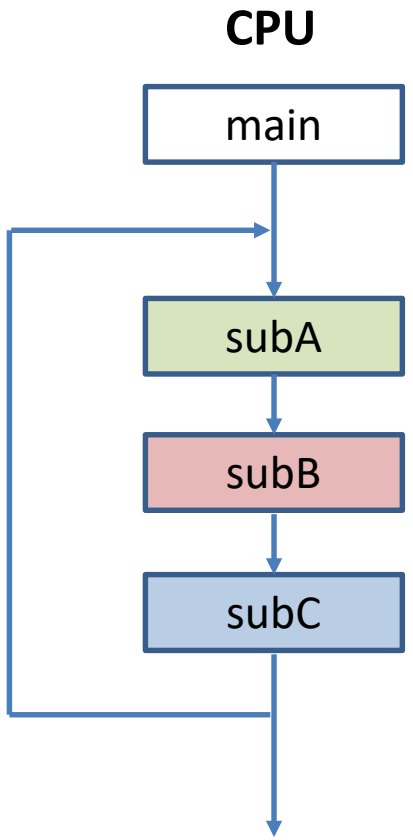
これが必要なのはアプリの一部分であることが多いので、
ここだけCUDAやライブラリを使えば良い。

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

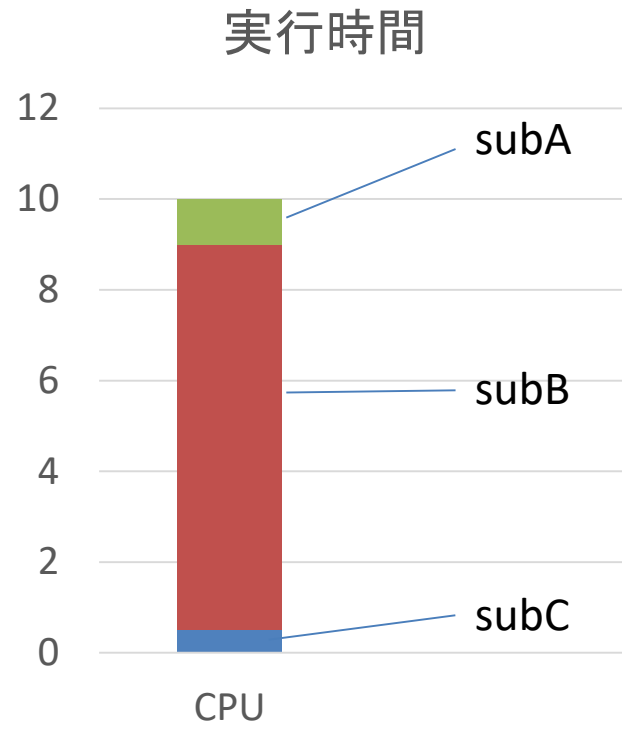
OpenACCを推奨する理由

- CPUプログラムの一般的なGPU化手順
  1. プログラムのプロファイリング（重い部分を特定する）
  2. 重い部分を並列化し、GPU上で実行する

OpenACCを推奨する理由

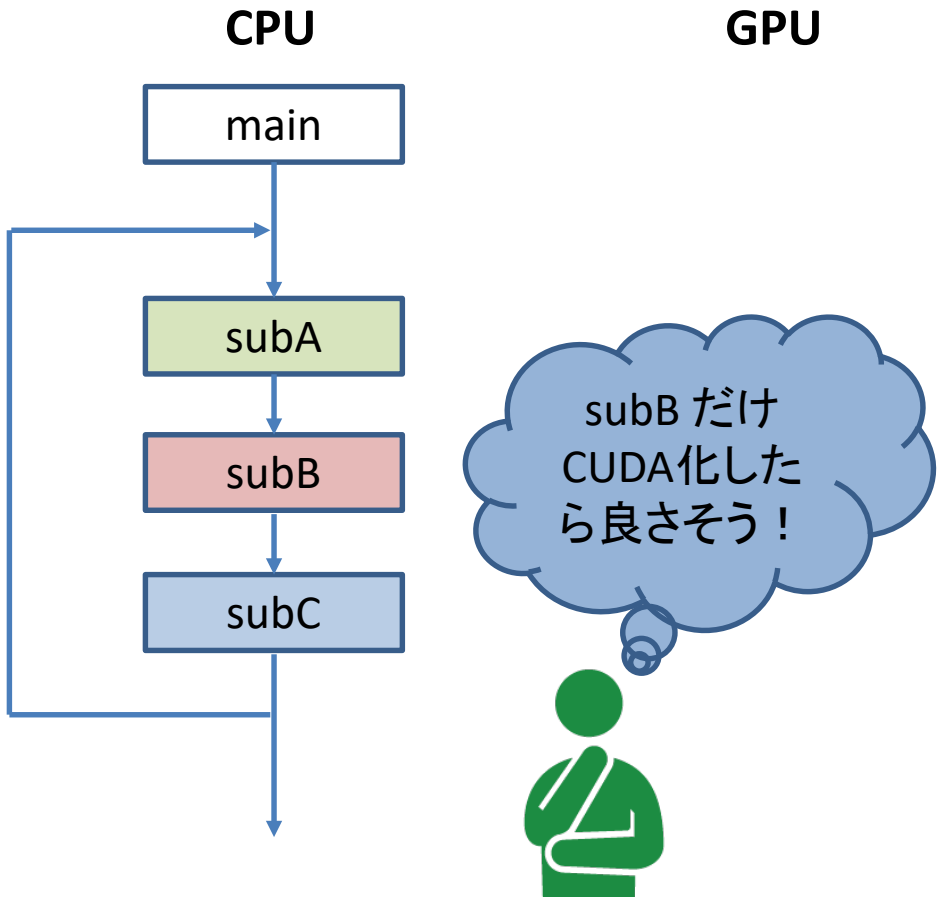


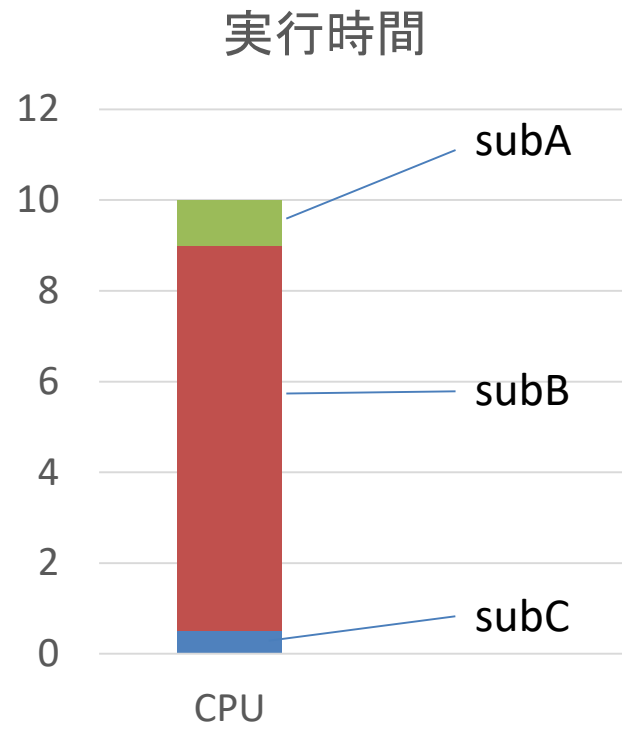
GPU



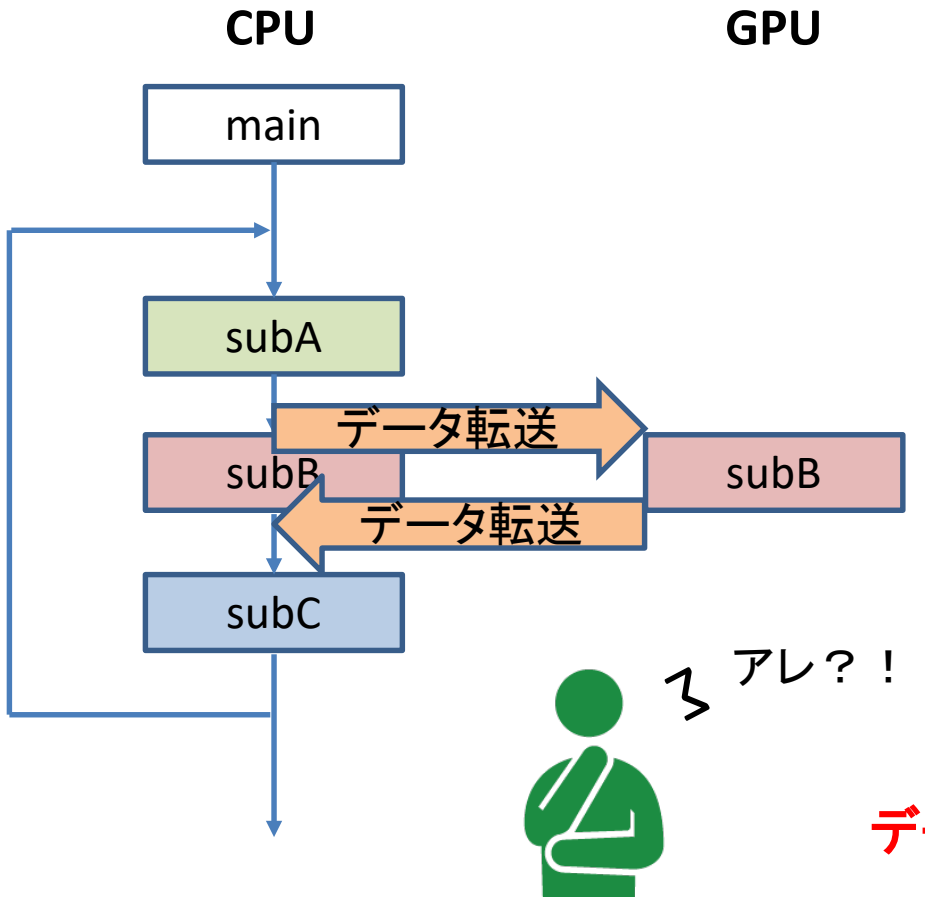
とあるプログラムの構造と実行時間を調べた結果

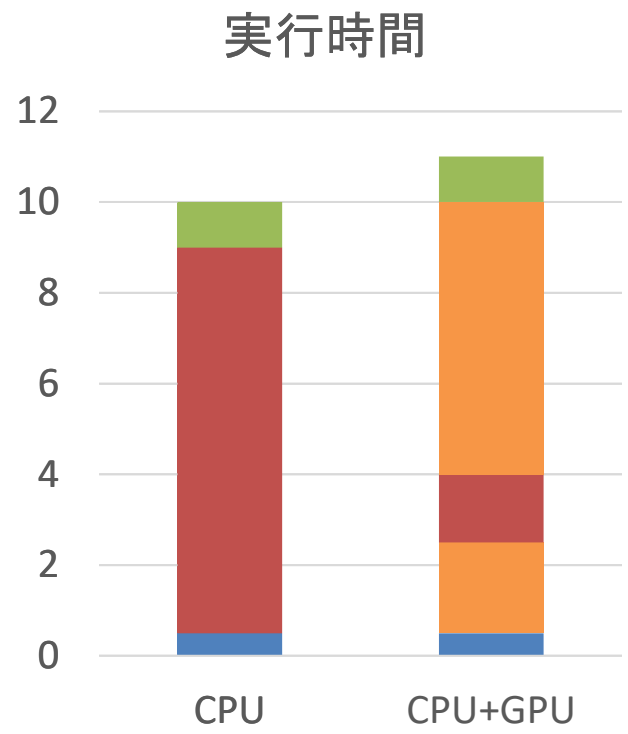
OpenACCを推奨する理由





OpenACCを推奨する理由





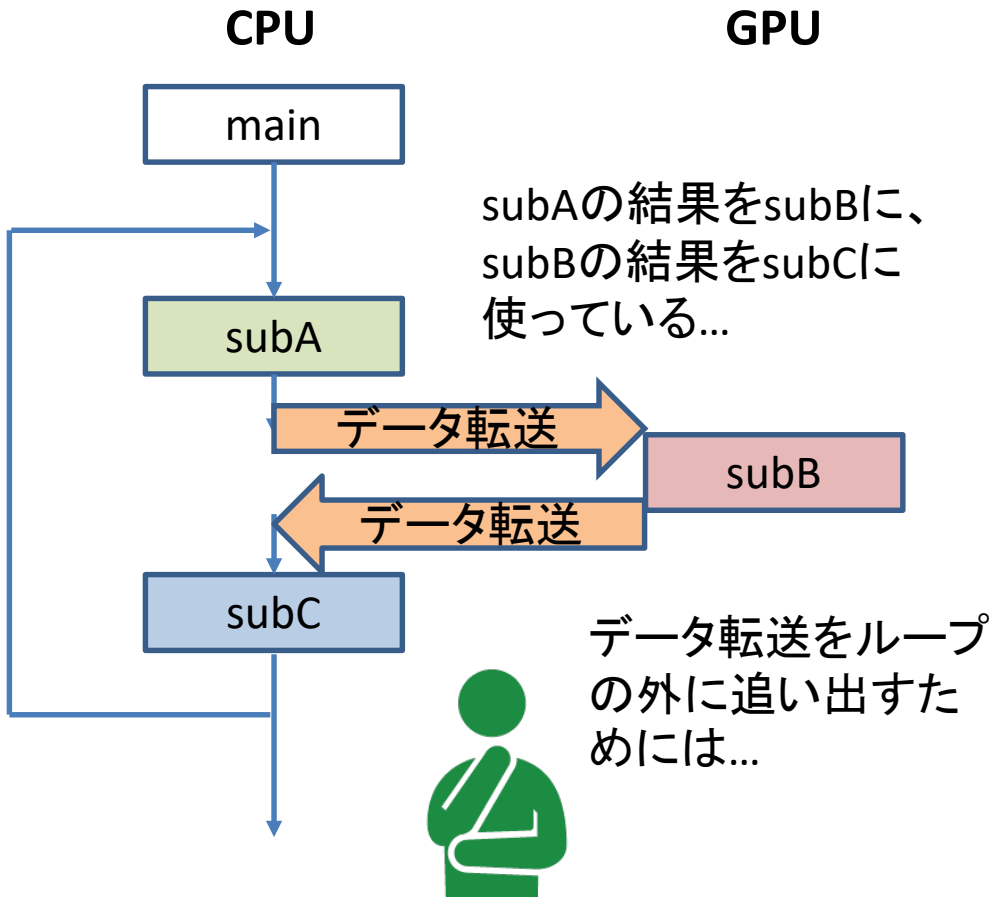
データ転送分遅くなった！

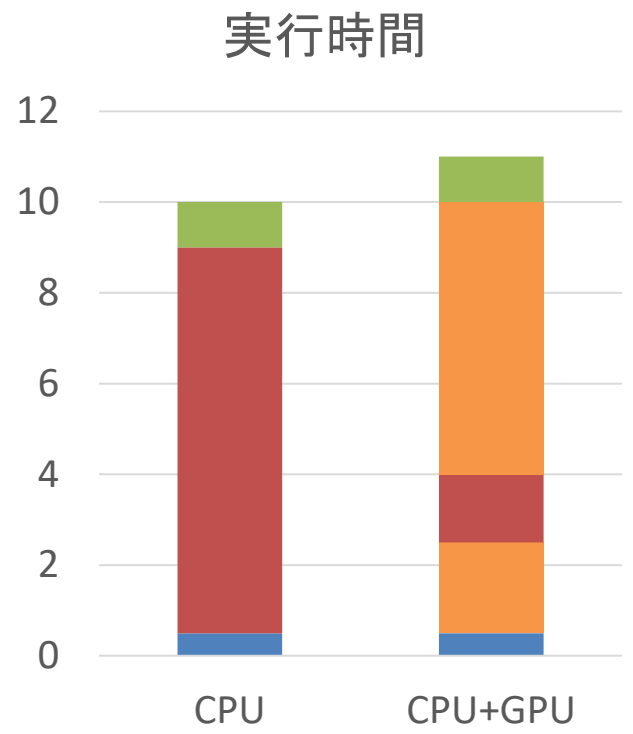
OpenACCを推奨する理由

- CPUプログラムの一般的なGPU化手順
  1. プログラムのプロファイリング（重い部分を特定する）
  2. 重い部分を並列化し、GPU上で実行する
  3. CPU-GPU間のデータ転送を最小化する

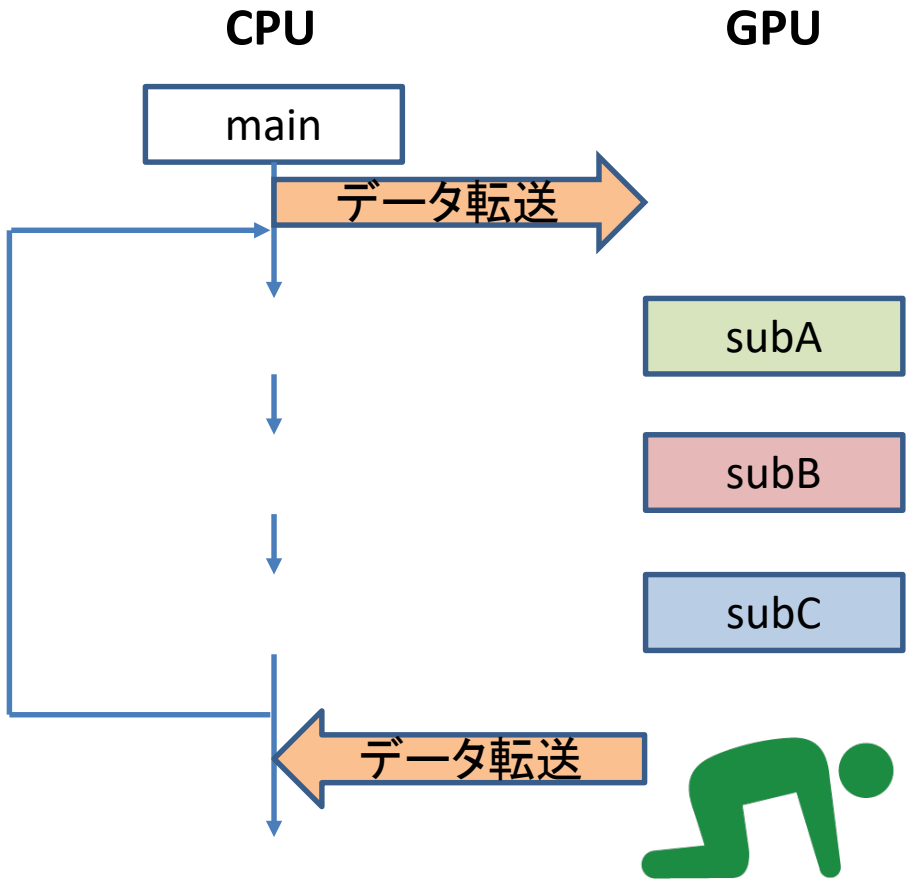
OpenACCであってもCUDAであっても、結局ここまでが必須！

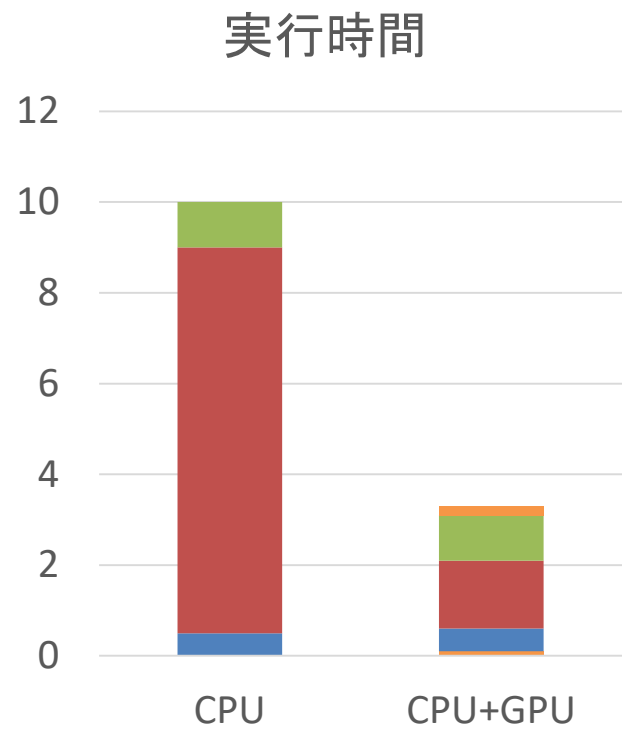
OpenACCを推奨する理由





OpenACCを推奨する理由





結局全部CUDA化した…

OpenACCを推奨する理由

- CPUプログラムの一般的なGPU化手順
  1. プログラムのプロファイリング（重い部分を特定する）
  2. 重い部分を並列化し、GPU上で実行する
  3. CPU-GPU間のデータ転送を最小化する
  4. GPU実行部でなお重い場所を最適化する

1,2,3をOpenACCで実装することで、最低限の実装までの工数を減らす。

4の最適化を場合によってはCUDAで行う。OpenACCにはCUDAと組み合わせるためのインターフェースが用意されている。

OpenACCを推奨する理由

- 実アプリをGPU化する場合、データ転送を最小化するためには、結局大部分を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）

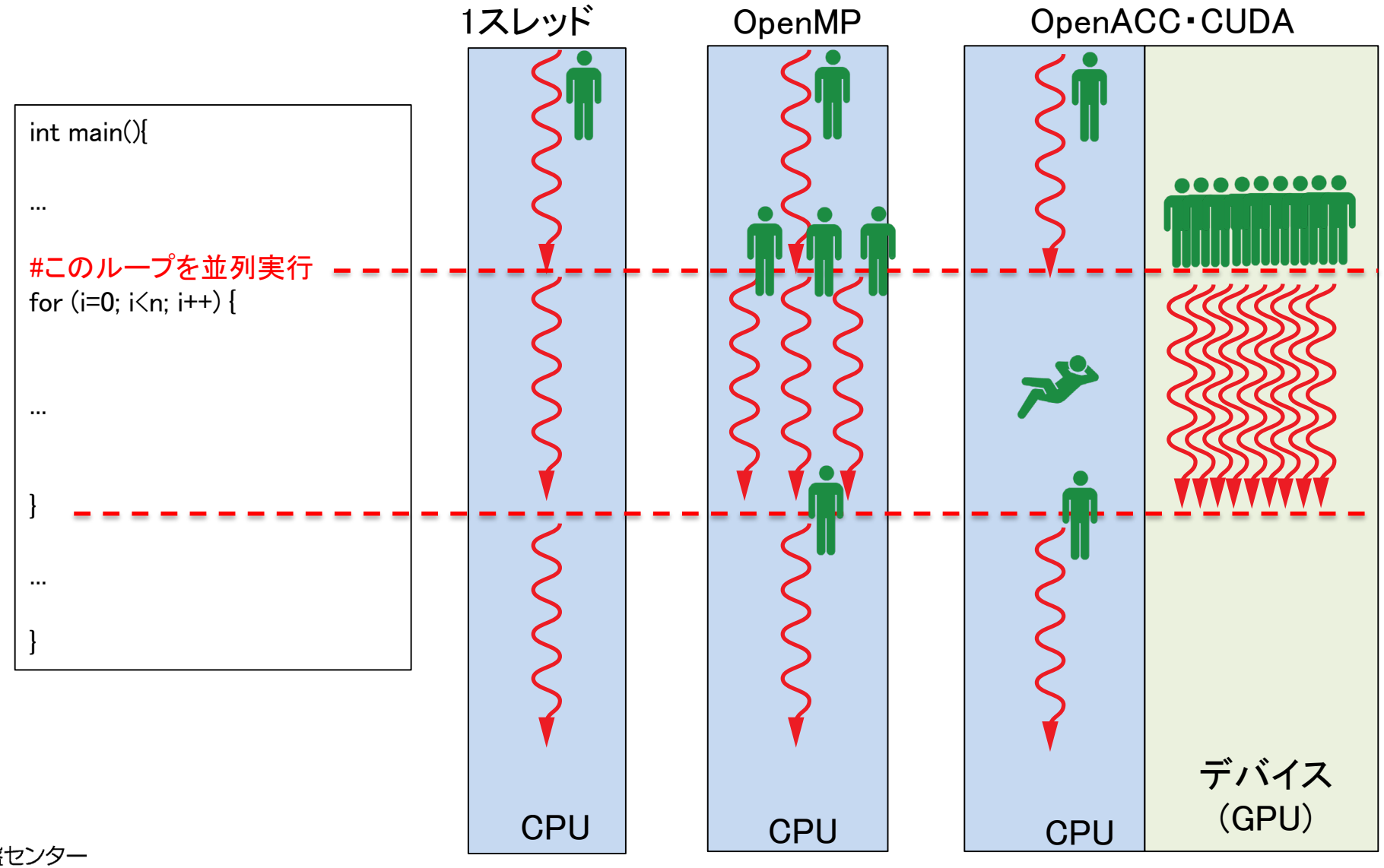
```
...
#pragma acc data copy(a[0:n])
{
...
#pragma acc host_data use_device(a)
{
    cuda_func(a, n)
}
...
}
...
```

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



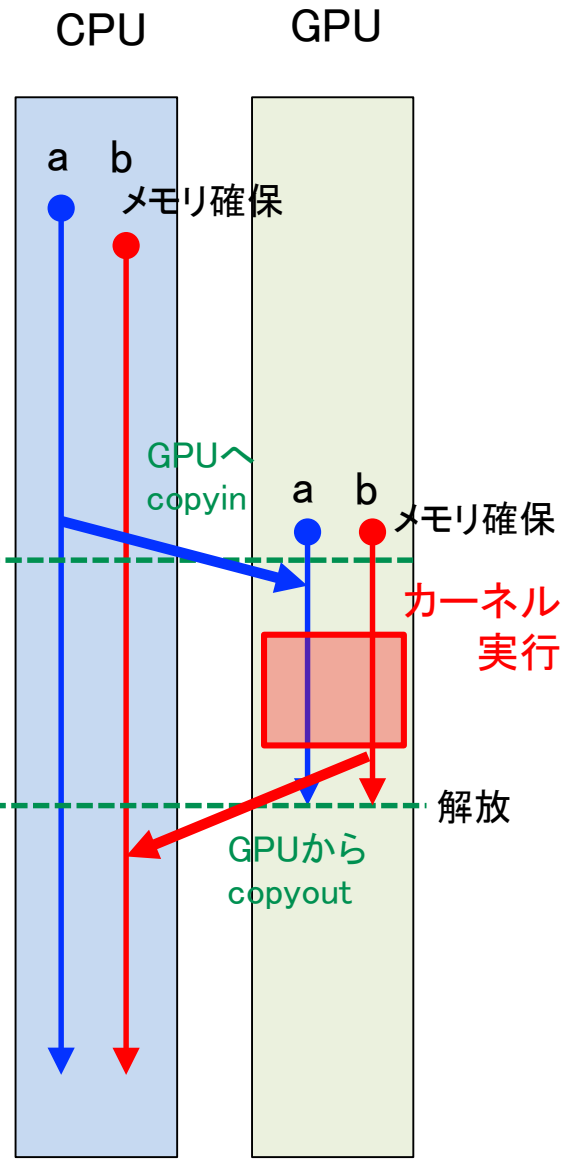
はじめてのOpenACCコード

openacc_hello/01_hello_acc

```
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;
}
```



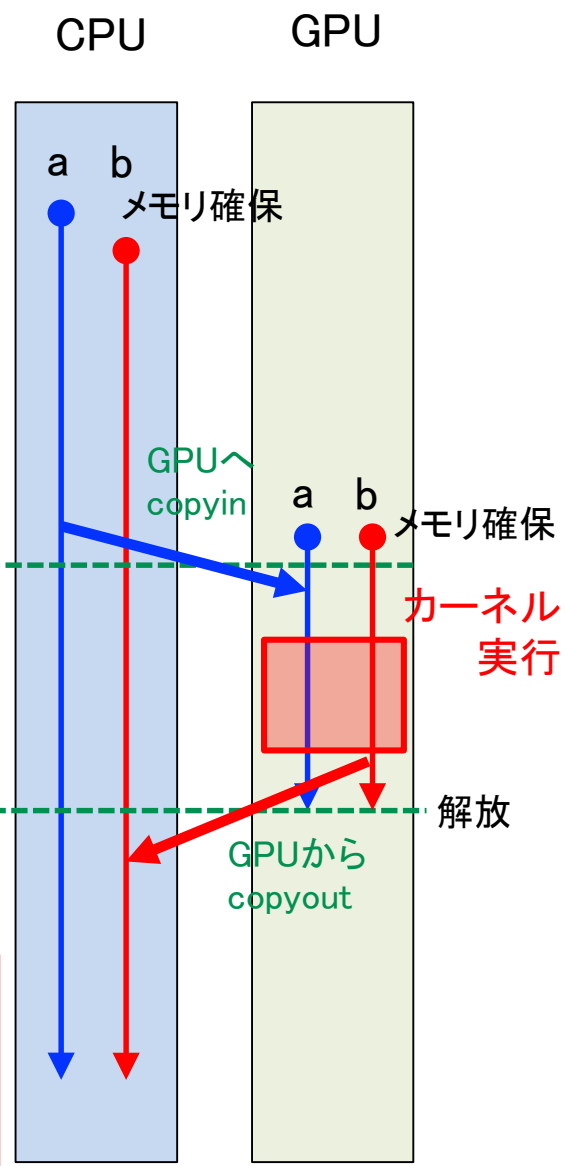
はじめてのOpenACCコード

openacc_hello/01_hello_acc

```
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

- 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(オプションだがほぼ必須)
  - ✓ loop
- データ移動指示文(オプションだがほぼ必須)
  - ✓ 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

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。
  - ✓ 似た指示文として、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

```
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

ここで同期。つまり kernel1 の終了が保証される。

kernel2

kernel2 が kernel1 に依存している

CPUコードのOpenACC化

C

openacc_hello/01_hello_acc

```
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

    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

1. GPUで実行したいループをkernelsで囲む

kernels直後の{}で囲まれる領域がGPU上で実行される

ループはベストエフォートで並列化される（C言語ではほとんどされない）

必要なデータ転送はベストエフォートで行われる（C言語ではよく失敗する）

CPUコードのOpenACC化

F

openacc_hello/01_hello_acc

```
program main
  implicit none
  ! 変数宣言
  allocate(a(n),b(n))
  c = 2.0

  do i = 1, n
     a(i) = 10.0
  end do

!$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
  print *, sum/n
  deallocate(a,b)
end program main
```

- ループのOpenACC 化
  1. GPUで実行したいループをkernelsで囲む

Fortranの場合、kernels ~ end kernels の間がGPUで実行される

ループはベストエフォートで並列化される（Fortranでは概ね成功する）

必要なデータ転送はベストエフォートで行われる（Fortranでは概ね成功する）

ループ指示文による並列化

C

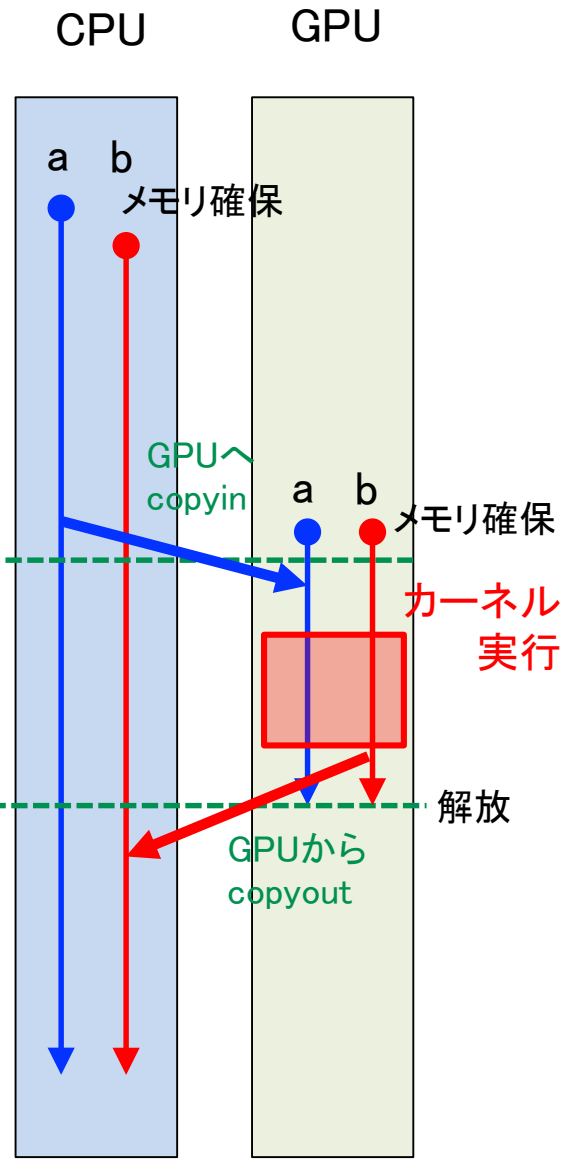
openacc_hello/01_hello_acc

```
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



ループ指示文による並列化

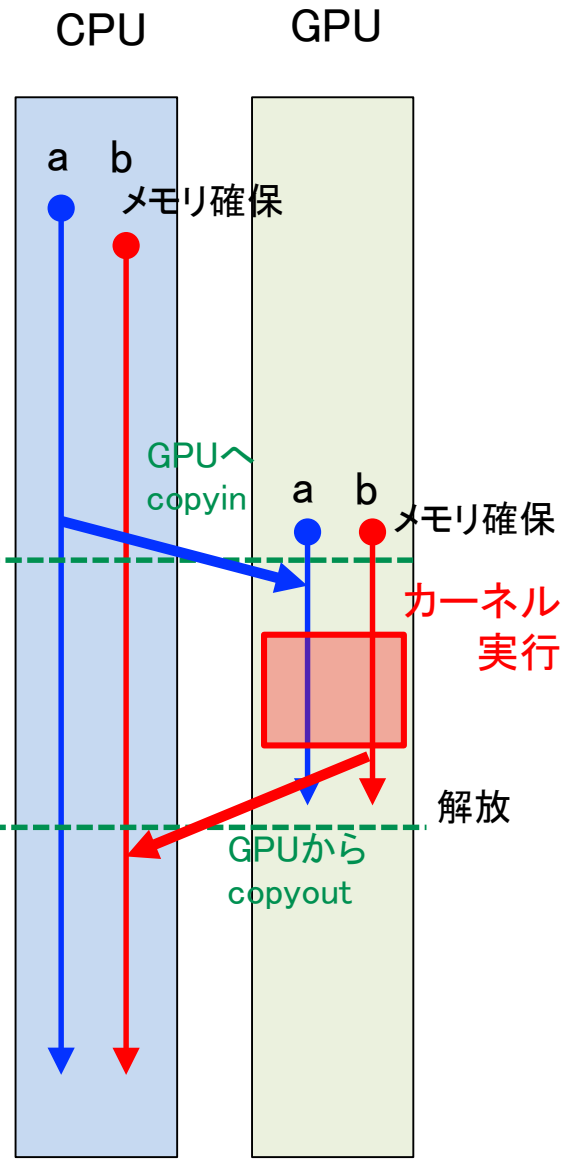
F

openacc_hello/01_hello_acc

```
program main
  implicit none
  ! 変数宣言
  allocate(a(n),b(n))
  c = 2.0

  do i = 1, n
     a(i) = 10.0
  end do
!$acc data copyin(a) copyout(b)
!$acc kernels
!$acc loop independent
  do i = 1, n
     b(i) = a(i) + c
  end do
!$acc end kernels
!$acc end data
  sum = 0.d0
  do i = 1, n
     sum = sum + b(i)
  end do
  print *, sum/n
  deallocate(a,b)
end program main
```

loop指示文



ループ最適化指示文

- loop 指示文（オプションだがほぼ必須）
  - ✓ 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

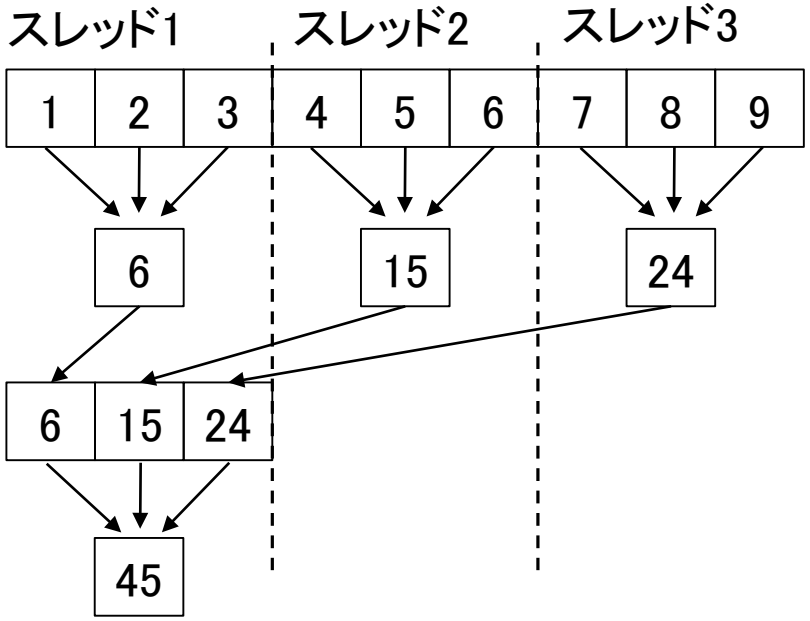
```
// これは正しくない

#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

CPUコードのOpenACC化

C

openacc_hello/01_hello_acc

```
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

1. GPUで実行したいループをkernelsで囲む
2. loop independent でループが並列化可能であることを教える

ループが並列化可能と見なされる（並列化可能でないループにindependent を付けると結果が間違う）

必要なデータ転送はベストエフォートで行われる（C言語ではよく失敗する）

CPUコードのOpenACC化

F

openacc_hello/01_hello_acc

```
program main
  implicit none
  ! 変数宣言
  allocate(a(n),b(n))
  c = 2.0

  do i = 1, n
     a(i) = 10.0
  end do

!$acc kernels
!$acc loop indepnedent
  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
  print *, sum/n
  deallocate(a,b)
end program main
```

■ ループのOpenACC 化

1. GPUで実行したいループをkernelsで囲む
2. loop independent でループが並列化可能であることを教える

ループが並列化可能と見なされる（並列化可能でないループにindependent を付けると結果が間違う）

必要なデータ転送はベストエフォートで行われる（Fortranでは概ね成功する）

Kernels指示文の自動データ転送

- 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、またはGPUのUnified memory機能を利用すべき
  - ✓ 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

```
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_acc

直後の { のタイミングで、mallocと
CPU -> GPUのデータコピーが行われる

直後の } の終了タイミングで、
GPU -> CPUのデータコピーと
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

```
program main
  implicit none
  integer,parameter :: n = 1000
  real(KIND=4),allocatable,dimension(:) :: a,b
  real(KIND=4) :: c
  integer :: i
  real(KIND=8) :: sum
  allocate(a(n),b(n))
  c = 2.0
  do i = 1, n
     a(i) = 10.0
  end do
!$acc data copyin(a) copyout(b)
!$acc kernels
!$acc loop independent
  do i = 1, n
     b(i) = a(i) + c
  end do
!$acc end kernels
!$acc end data
```

openacc_hello/01_hello_acc

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
  - ✓ 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。
- 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。（OpenACC2.5以降）

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])
...
```

Unified Memory

- Unified Memory とは…
  - ✓ 物理的に別物のCPUとGPUのメモリをあたかも一つのメモリのように扱う機能
  - ✓ NVIDIA A100 GPUではハードウェアサポート
    - ➢ ページフォルトが起こると勝手にマイグレーションしてくれる
- OpenACC と Unified Memory
  - ✓ OpenACCの仕様にUnified Memoryを直接使う機能はない
    - ➢ nvidia コンパイラではオプションを与えることで使える
    - ➢ nvfortran -acc -ta=tesla,managed
  - ✓ 使うとデータ指示文が無視され、代わりにUnified Memoryを使う
    - ➢ ハイエンドのNVIDIA GPU + NVIDIA compilerの環境が揃いさえすれば、データ転送を考える必要がなく非常に楽 (Wisteria環境では利用推奨)
    - ➢ data指示文が間違ったOpenACCコードでも正しく動いてしまう
    - ➢ Unified memory を使うと、GPU direct というMPIの直接通信機能が使えない
    - ➢ CPU側のメモリアロケーションを全部監視してるので、遅くなるケースがある

CPUコードのOpenACC化

C

openacc_hello/01_hello_acc

```
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;
}
```

- ループのOpenACC 化
  1. GPUで実行したいループをkernelsで囲む
  2. loop independent でループが並列化可能であることを教える
  3. data 指示文でデータ転送を行う
     - このケースではあまり data 指示文の意味はない。後の最適化で本領発揮。

CPUコードのOpenACC化

F

openacc_hello/01_hello_acc

```
program main
  implicit none
  ! 変数宣言
  allocate(a(n),b(n))
  c = 2.0

  do i = 1, n
     a(i) = 10.0
  end do
!$acc data copyin(a) copyout(b)
!$acc kernels
!$acc loop indepnedent
  do i = 1, n
     b(i) = a(i) + c
  end do
!$acc end kernels
!$acc end data
  sum = 0.d0
  do i = 1, n
     sum = sum + b(i)
  end do
  print *, sum/n
  deallocate(a,b)
end program main
```

- ループのOpenACC 化
  1. GPUで実行したいループをkernelsで囲む
  2. loop independent でループが並列化可能であることを教える
  3. data 指示文でデータ転送を行う
     - このケースではあまり data 指示文の意味はない。後の最適化で本領発揮。

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 指示文でループの並列化
- ✓ data 指示文でデータ転送を制御。

```
// 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

NVIDIAコンパイラによるコンパイル

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

```
$ module load nvidia
$ nvc -O3 -acc -Minfo=accel -ta=tesla,cc80 -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,cc80:

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 8.0 (cc80) 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, data, 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

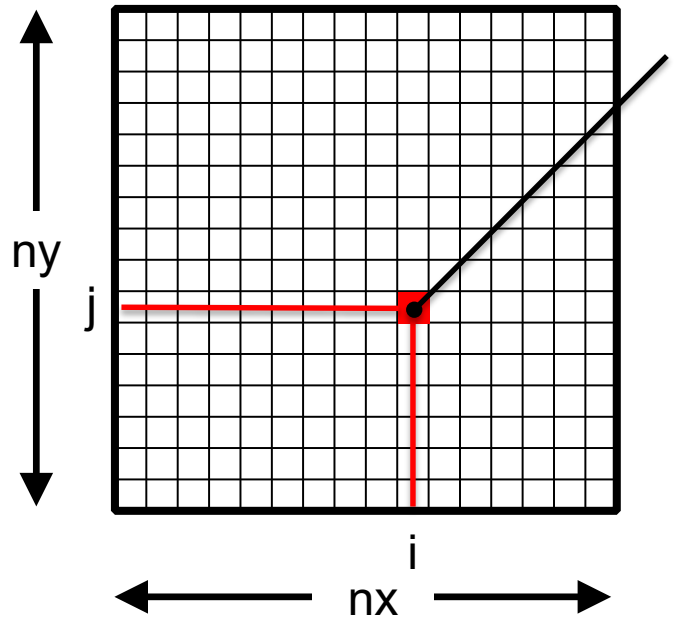
openacc_basic/01_original	CPUコード。
openacc_basic/02_kernels	OpenACCコード。上にkernels指示文のみ追加。
openacc_basic/03_kernels_copy	OpenACCコード。上にcopy指示節追加。
openacc_basic/04_loop	OpenACCコード。上にloop指示文を追加。
openacc_basic/05_data	OpenACCコード。上にdata指示文を明示的に追加。
openacc_basic/06_present	OpenACCコード。上でpresent指示節を使用。
openacc_basic/07_reduction	OpenACCコード。上にreduction指示節を使用。

配列のインデックス計算

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
$ pjsub ./run.sh
$ cat run.sh.??????.out
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(KIND=4),dimension(:,:),intent(in)  :: a,b
    real(KIND=4),dimension(:,:),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
F

- コンパイル

データサイズがわからずコンパイルエラー
C言語では配列サイズの指定がほぼ必須！

C

```
$ make
nvc -O3 -acc -Minfo=accel  -ta=tesla,cc80 -c main.c
NVC++-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index
for symbol - b (main.c: 11)
calc:
     14, Complex loop carried dependence of a->,c->,b-> prevents parallelization
         Accelerator serial kernel generated
         Generating Tesla code
         14, #pragma acc loop seq
         15, #pragma acc loop seq
     15, Accelerator restriction: size of the GPU copy of c,b,a is unknown
         Complex loop carried dependence of a->,c->,b-> prevents parallelization
NVC++-F-0704-Compilation aborted due to previous errors. (main.c)
NVC++/x86-64 Linux 21.3-0: compilation aborted
make: *** [Makefile:33: main.o] エラー 2
```

F

```
$ make
nvfortran -O3 -mp -acc -ta=tesla,cc80 -Minfo=accel  -c main.f90
calc:
     13, Generating implicit copyin(b(:nx,:ny)) [if not already present]
         Generating implicit copy(c(:nx,:ny)) [if not already present]
         Generating implicit copyin(a(:nx,:ny)) [if not already present]
     14, Loop is parallelizable
     15, Loop is parallelizable
         Generating Tesla code
         14, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         15,   ! blockidx%x threadidx%x auto-collapsed
```

データサイズを検知して自動転送
Fotranではサイズ情報が配列に付随するため

簡単なOpenACC: kernels 指示文(3)

C

- 03_kernels_copyコード: calc関数
  - ✓ 配列サイズを明示的に指定

openacc_basic/03_kernels_copy

```
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 指示文の指示節が使える
  - ✓ 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(4)

C F

- 03_kernels_copyコード:初期化
  - ✓ CPUコードにkernels 指示文の追加

openacc_basic/03_kernels_copy

C

```
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;
      }
    }

...
}
```

F

```
program main
...
!$acc kernels copyout(b,c)
  do j = 1,ny
    do i = 1,nx
      b(i,j) = b0
    end do
  end do

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

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。

Fortranの配列代入形式も使える

簡単なOpenACC: kernels 指示文(5)

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。

```
$ make
nvc -O3 -acc -Minfo=accel  -ta=tesla,cc80 -c main.c
calc:
     11, Generating copy(a[:n],c[:n],b[:n]) [if not already present]
     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 serial 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:
     44, Generating copyout(c[:16777216],b[:16777216]) [if not already present]
     45, Loop is parallelizable
         Generating Tesla code
         45, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     48, Loop is parallelizable
         Generating Tesla code
         48, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

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。

```
nvfortran -O3 -mp -acc -ta=tesla,cc80 -Minfo=accel  -c main.f90
calc:
     13, Generating copyin(a(:,:)) [if not already present]
         Generating copyout(c(:,:)) [if not already present]
         Generating copyin(b(:,:)) [if not already present]
     14, Loop is parallelizable
     15, Loop is parallelizable
         Generating Tesla code
         14, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         15,   ! blockidx%x threadidx%x auto-collapsed
main:
     61, Generating copyout(b(:,:),c(:,:)) [if not already present]
     62, Loop is parallelizable
     63, Loop is parallelizable
         Generating Tesla code
         62, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         63,   ! blockidx%x threadidx%x auto-collapsed
     68, Loop is parallelizable
         Generating Tesla code
         68,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
```

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

- 04_loopコード
  - ✓ 03_kernelsコードにloop independent の追加

openacc_basic/04_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(1)

F

- 04_loopコード
  - ✓ 03_kernelsコードにloop independent の追加

openacc_basic/04_loop

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

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

各次元についてloop指示文を指定する(並列サイズなどを指定したいなど)場合、do文で書き下す必要がある。

簡単なOpenACC: loop 指示文(2)

C

openacc_basic/04_loop

- コンパイル
  - ✓ ループが並列化され、カーネルが生成された。

```
$ make
nvc -O3 -acc -Minfo=accel  -ta=tesla,cc80 -c main.c
calc:
     11, Generating copy(a[:n],c[:n],b[:n]) [if not already present]
     15, Loop is parallelizable
     17, Loop is parallelizable
         Generating Tesla code
         15, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         17,   /* blockIdx.x threadIdx.x auto-collapsed */
main:
     46, Generating copyout(c[:16777216],b[:16777216]) [if not already present]
     48, Loop is parallelizable
         Generating Tesla code
         48, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     52, Loop is parallelizable
         Generating Tesla code
         52, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

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

簡単なOpenACC: loop 指示文(3)

- 04_loopコードの実行

openacc_basic/04_loop

  - ✓ 答えは正しいが、実行時間が大変長い。

```
$ pjsub ./run.sh
$ cat run.sh.??????.out
mean = 3000.00
Time =   42.990 [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)

C

- 05_dataコード
  - ✓ 04_loopにdata指示文追加

openacc_basic/05_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(1)

F

- 05_dataコード
  - ✓ 04_loopにdata指示文追加

openacc_basic/05_data

```
! main関数内
!$acc data copyin(a) create(b) copyout(c)
!$acc kernels copyout(b,c)
!$acc loop independent
  do j = 1,ny
!$acc loop independent
     do i = 1,nx
        b(i,j) = b0
     end do
  end do

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

  do icnt = 1,nt
     call calc(nx, ny, a, b, c)
  end do

!$acc end data
```

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)

- 05_dataコードの実行
  - ✓ 答えは正しく、速度が上がった。

openacc_basic/05_data

```
$ pjsub ./run.sh
$ cat run.sh.??????.out
mean = 3000.00
Time =    0.376 [sec]
```

簡単なOpenACC: present指示節

- 06_presentコード
  - ✓ 05_dataコードで present 指示節を使用

openacc_basic/06_present

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 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

F

```
  subroutine calc(nx, ny, a, b, c)
   ...
!$acc kernels present(a, b, c)
!$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
  end subroutine
```

- ◆ 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。

簡単なOpenACC: reduction指示節（1）

C

- 07_reductionコード
  - ✓ 06_presentコードで reductionを使用

openacc_basic/07_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。
- 07_reductionコード
  - ✓ リダクションコードが生成された。

```
$ make
nvc -O3 -acc -Minfo=accel  -ta=tesla,cc80 -c main.c
(省略)
main:
(省略)
 67, Loop is parallelizable
         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（1）

F

- 07_reductionコード
  - ✓ 06_presentコードで reductionを使用

openacc_basic/07_reduction

```
  sum = 0
!$acc kernels present(c)
!$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
```

並列ループ毎にreduction

  - ✓ data 指示文で c を create に変更。
- 07_reductionコード
  - ✓ リダクションコードが生成された。

```
$ make
nvfortran -O3 -mp -acc -ta=tesla,cc80 -Minfo=accel  -c main.f90
(省略)
main:
(省略)
 86, Loop is parallelizable
         Generating Tesla code
         84, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
             Generating reduction(+:sum)
         86,   ! blockidx%x threadidx%x auto-collapsed
```

簡単なOpenACC: reduction指示節（2）

- 07_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。

openacc_basic/07_reduction

```
$ pjsub ./run.sh
$ cat run.sh.??????.out
mean = 3000.00
Time =    0.353 [sec]
```

OpenACC化のステップのまとめ

C

- 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、ホストーデバイス 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/07_reduction

OpenACC化のステップのまとめ

F

- 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

```
!$acc data copyin(a) create(b,c)
!$acc kernels present(b,c)
!$acc loop independent
  do j = 1,ny
!$acc loop independent
     do i = 1,nx
        b(i,j) = b0
     end do
  end do

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

  do icnt = 1,nt
     call calc(nx, ny, a, b, c)
  end do
                        !続く
```

```
!続き

  sum = 0
!$acc kernels present(c)
!$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
```

openacc_basic/07_reduction

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, 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

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, data, loop 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

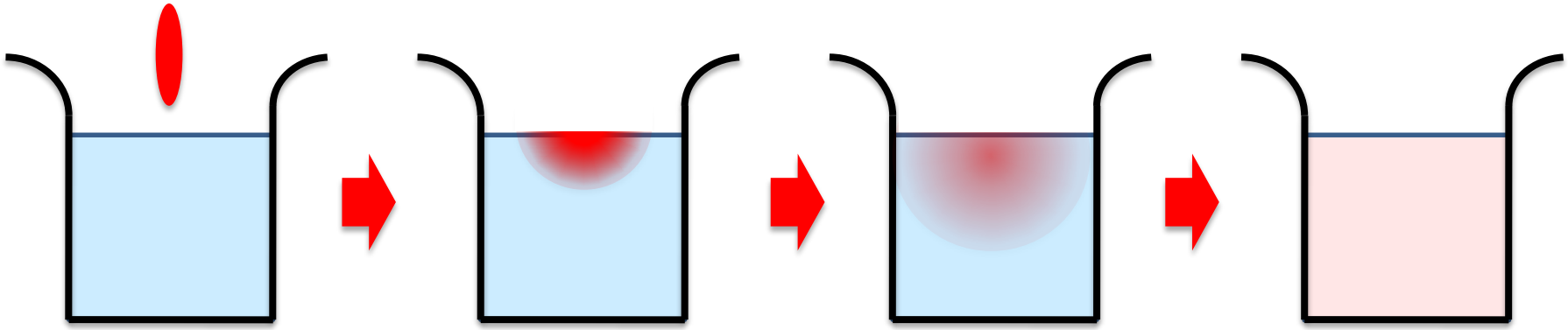
      w = -1; e = 1; n = -1; s = 1; b = -1; t = 1;
      if(i == 1)  w = 0
      if(i == nx) e = 0
      if(j == 1)  n = 0
      if(j == ny) s = 0
      if(k == 1)  b = 0
      if(k == nz) t = 0
      fn(i,j,k) = cc * f(i,j,k) + cw * f(i+w,j,k) &
          + ce * f(i+e,j,k) + cs * f(i,j+s,k) + cn * f(i,j+n,k)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

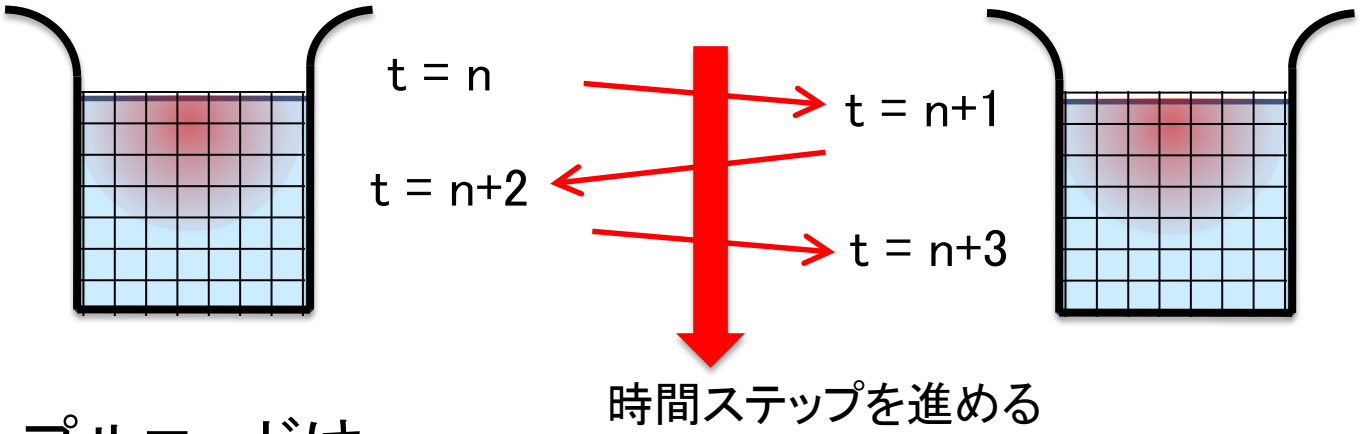
- 拡散現象
  - ✓ コップの中に赤インクを落とすと水中で拡がる
  - ✓ 次第に拡散し赤インクは拡がり、最後は均一な色になる。



- 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。

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
$ pjsub ./run.sh
# cat run.sh.??????.out
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 =    8.677 [sec]
Performance=    5.15 [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    = -O3 -acc -Minfo=accel -ta=tesla,cc80
GFLAGS    = -Wall -O3 -std=c99
CXXFLAGS  = $(CFLAGS)
LDFLAGS   =
...
```

F

```
F90  = nvfortran
RM   = rm -f


FFLAGS    = -O3 -mp -acc -ta=tesla,cc80 -Minfo=accel
...
```

OpenACC化(1): kernels

C

- 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(1): kernels

F

- diffusion3d関数に kernelsを追加しましょう

```
!$acc kernels copyin(f) copyout(fn)
 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 して実行してみましょう。

OpenACC化(2): loop

C

- diffusion3d関数に loopを追加しましょう

```
#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 pjsub ./run.shしてみましょう。遅いですが実行できます。

OpenACC化(2): loop

F

- diffusion3d関数に loopを追加しましょう

```
!$acc kernels copyin(f) copyout(fn)
!$acc loop independent
 do k = 1, nz
!$acc loop independent
    do j = 1, ny
!$acc loop independent
       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

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 pjsub ./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)

C

- 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(3): データ転送の最適化(1)

F

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

diffusion.f90, diffusion3d 関数内

```
!$acc kernels copyin(f) copyout(fn)
!$acc loop independent
 do k = 1, nz
!$acc loop independent
    do j = 1, ny
!$acc loop independent
       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
```

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)

C

- 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

OpenACC化(4): データ転送の最適化(2)

F

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

main.f90, main 関数内

```
!$acc data copy(f) create(fn)
  call start_timer()

  do icnt = 0, nt-1
    if(mod(icnt,100) == 0) write (*,"(A5,I4,A4,F7.5)"), "time(",icnt,") = ",time

    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

  elapsed_time = get_elapsed_time()
!$acc end data
```

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

NVCOMPILER_ACC_TIME によるOpenACC 実行の確認

- NVIDIA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NVCOMPILER_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
```

- Wisteria 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_nvcompiler_acc_time

NVCOMPILER_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.??????.err
Accelerator Kernel Timing data
/work/01/gt00/z30108/openacc_samples_test/openacc_samples/C/openacc_diffusion/03_openacc_pgi_
acc_time/main.c
  main  NVIDIA  devicenum=0
    time(us): 725
    39: data region reached 2 times
        39: data copyin transfers: 1
             device time(us): total=342 max=342 min=342 avg=342
        53: data copyout transfers: 1
             device time(us): total=383 max=383 min=383 avg=383
/work/01/gt00/z30108/openacc_samples_test/openacc_samples/C/openacc_diffusion/03_openacc_pgi_
acc_time/diffusion.c
  diffusion3d  NVIDIA  devicenum=0
    time(us): 0
    17: compute region reached 1638 times
        25: kernel launched 1638 times
            grid: [16384]  block: [128]
            elapsed time(us): total=67,084 max=54 min=40 avg=40
    17: 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

Unified MemoryによるOpenACC 実行の確認

- NVIDIA compiler で Unified memory を利用する場合、-ta オプションに managed を付けるだけ！
  - ✓ data指示文が無視されるようになる
  - ✓ サンプルコードでは、data指示文をコメントアウトしている
  - ✓ 性能はどのくらい違うだろうか？

C

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

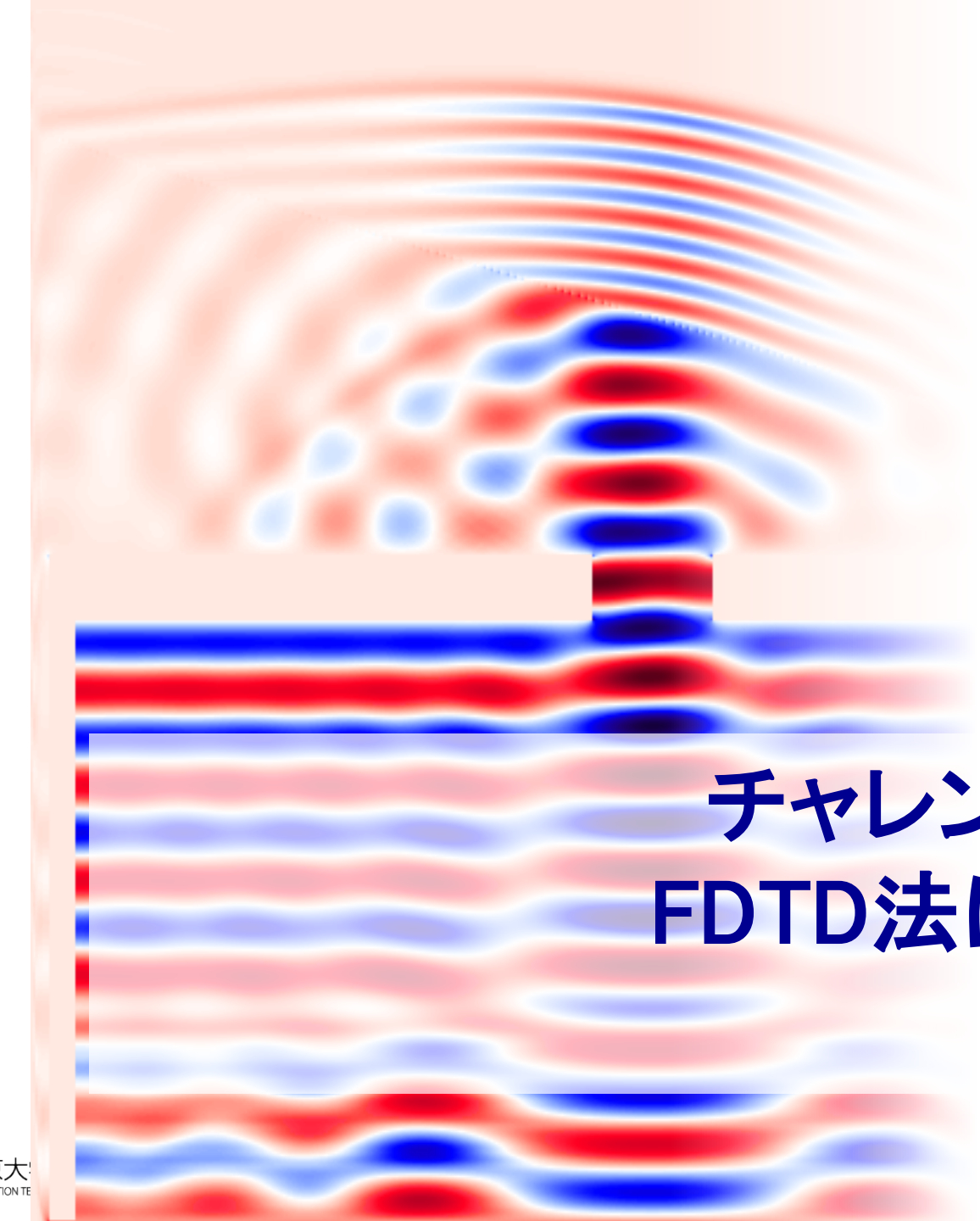
CFLAGS    = -O3 -acc -Minfo=accel -ta=tesla,cc80,managed
...
```

F

```
F90  = nvfortran
RM   = rm -f


FFLAGS    = -O3 -mp -acc -ta=tesla,cc80,managed -Minfo=accel
...
```

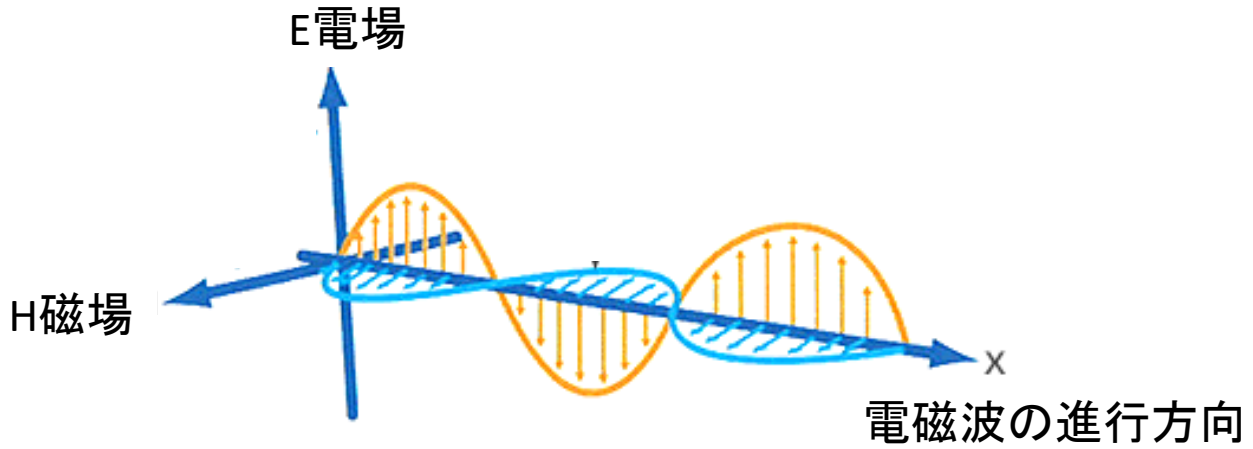
サンプルコードは、 openacc_diffusion/04_openacc_managed



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）

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、

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

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

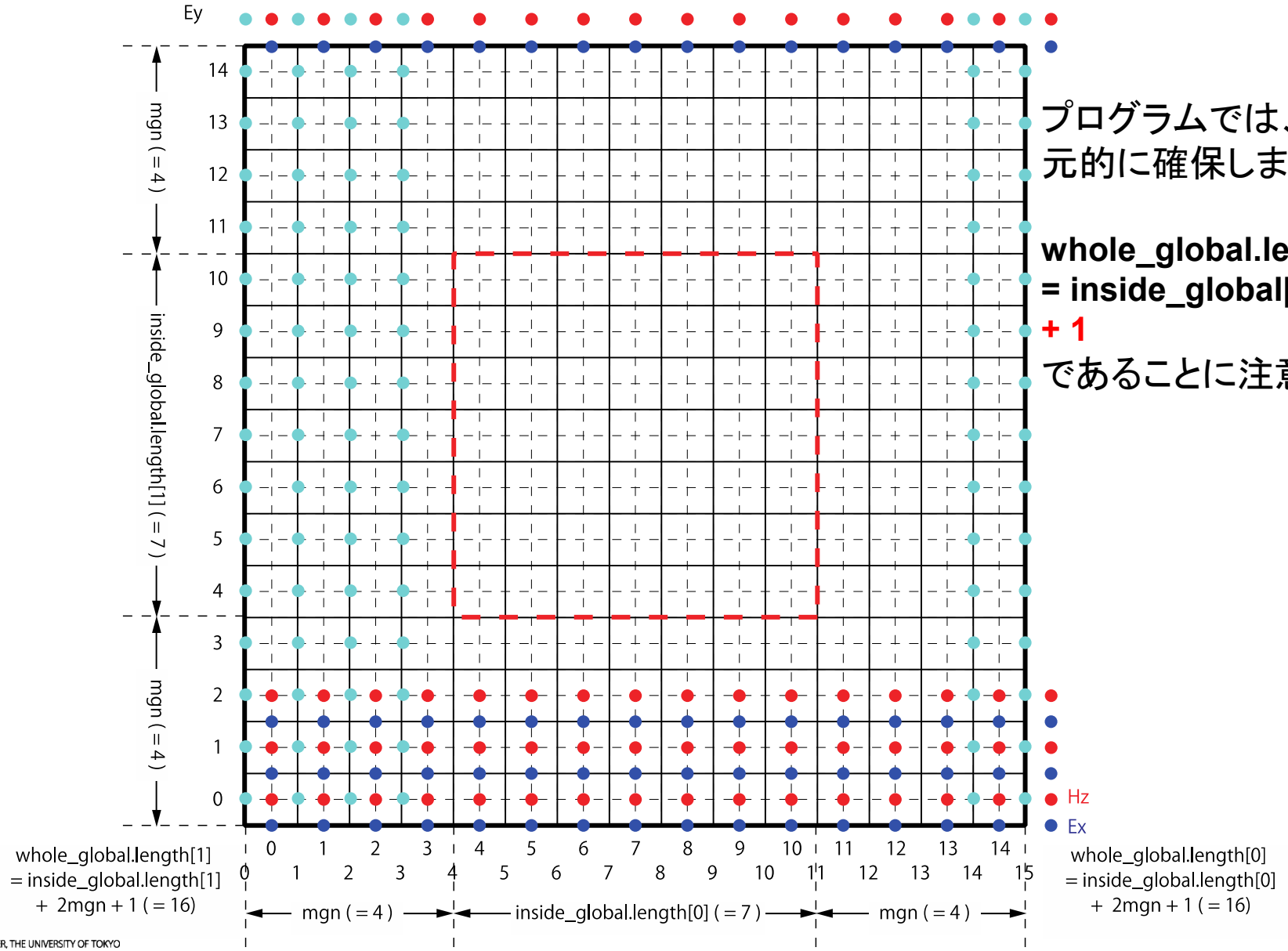
FDTD法（2）

- 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	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。

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

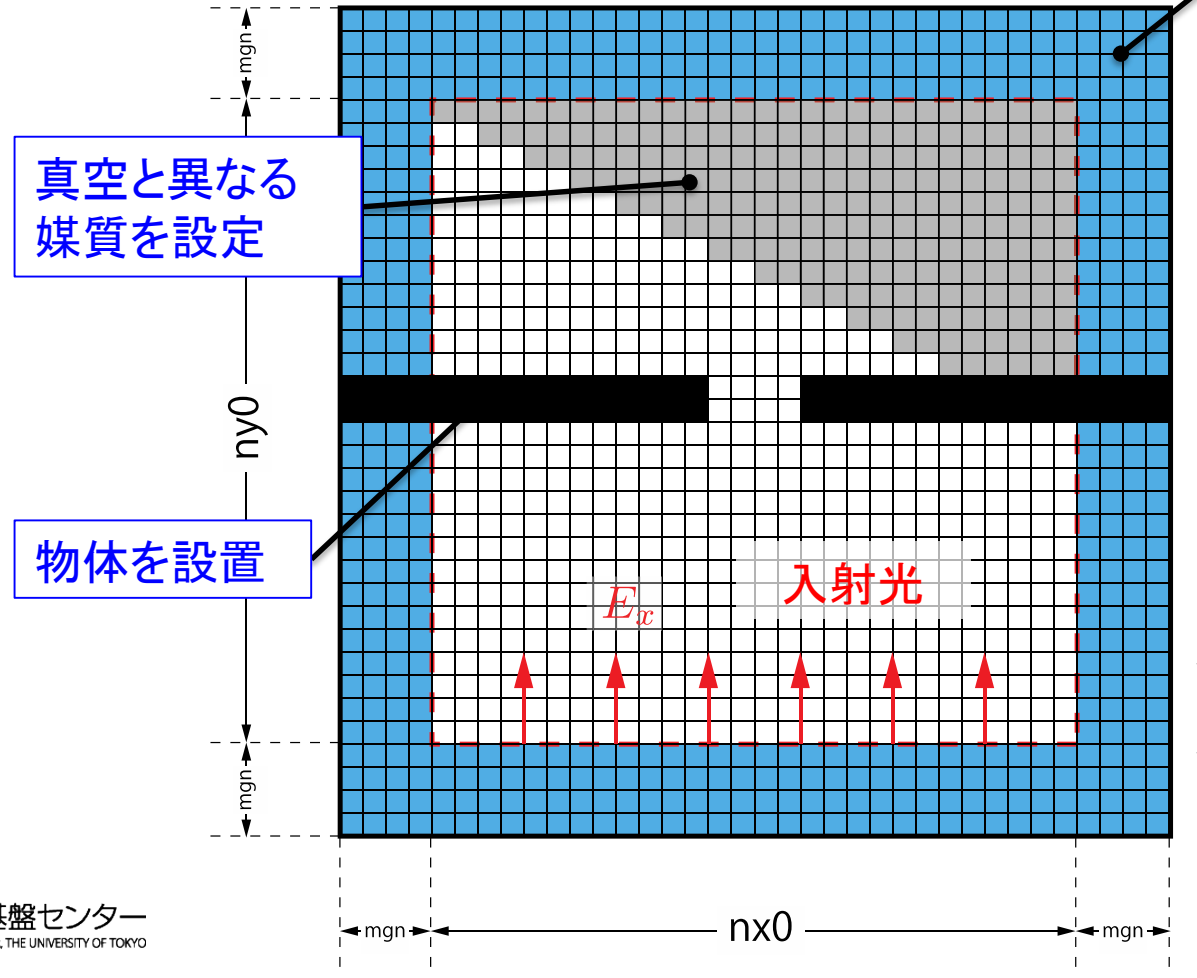
本講習では、“main.c”、“fdtd2d.c”、” fdtd2d_sources.c”、”setup.c” のソースコードを追記・修正していきます。

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）

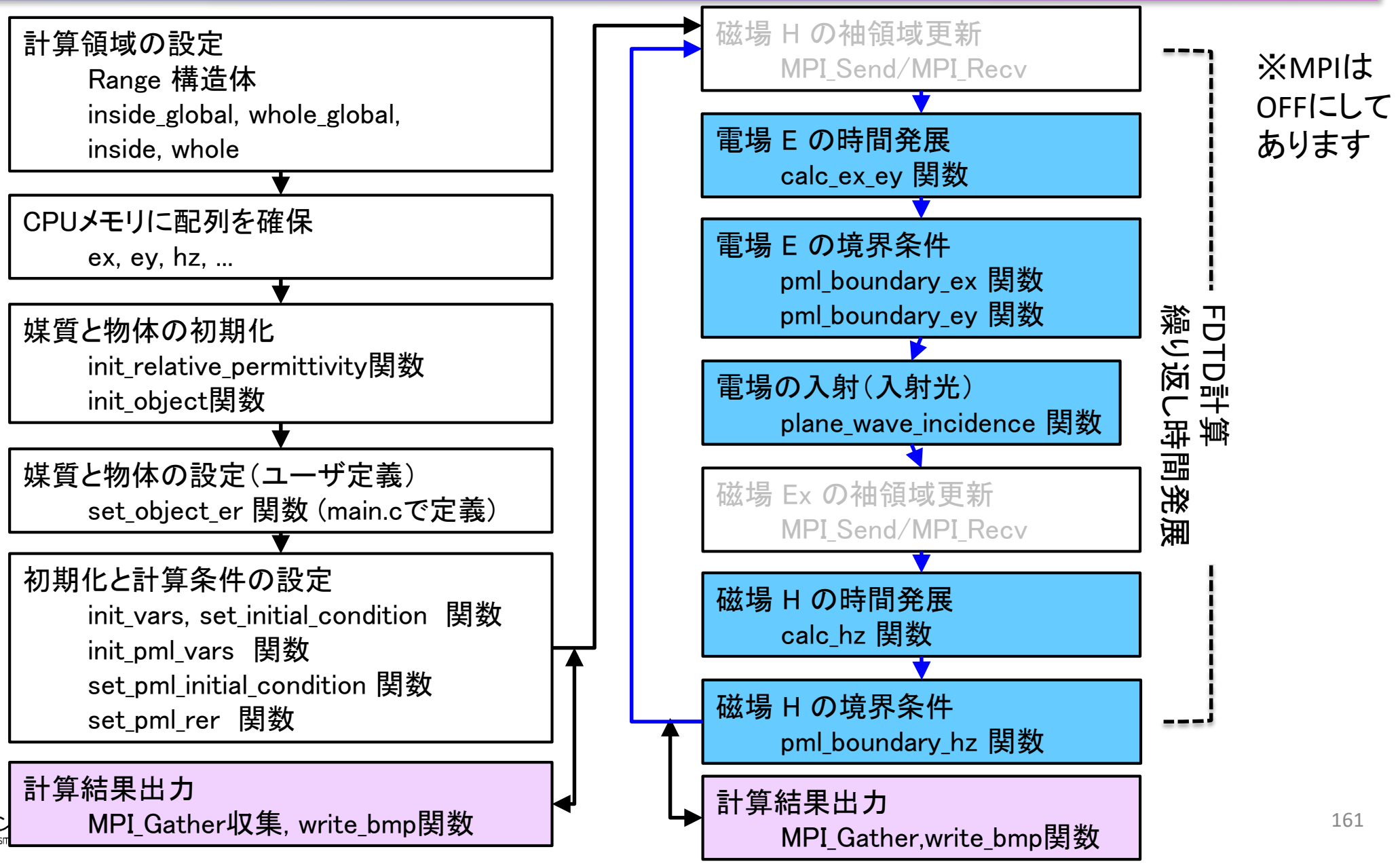


dx = lx/nx
dy = ly/ny

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

コード全体の流れ（main.c 内）



※MPIは
OFFにして
あります

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

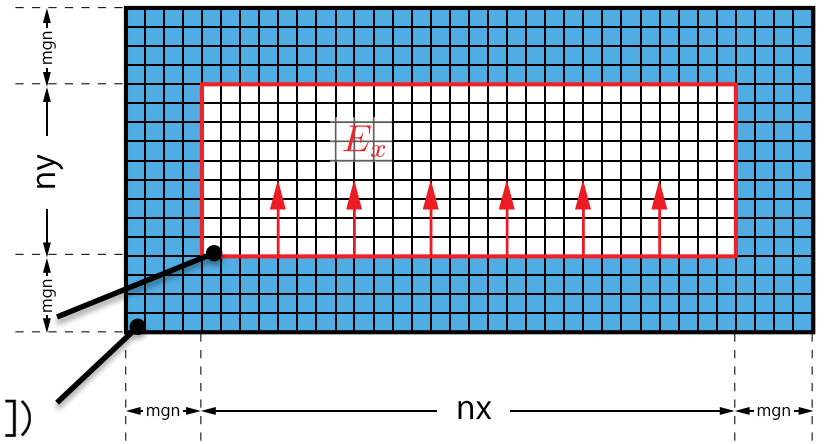
プログラムでは下記の通り

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）

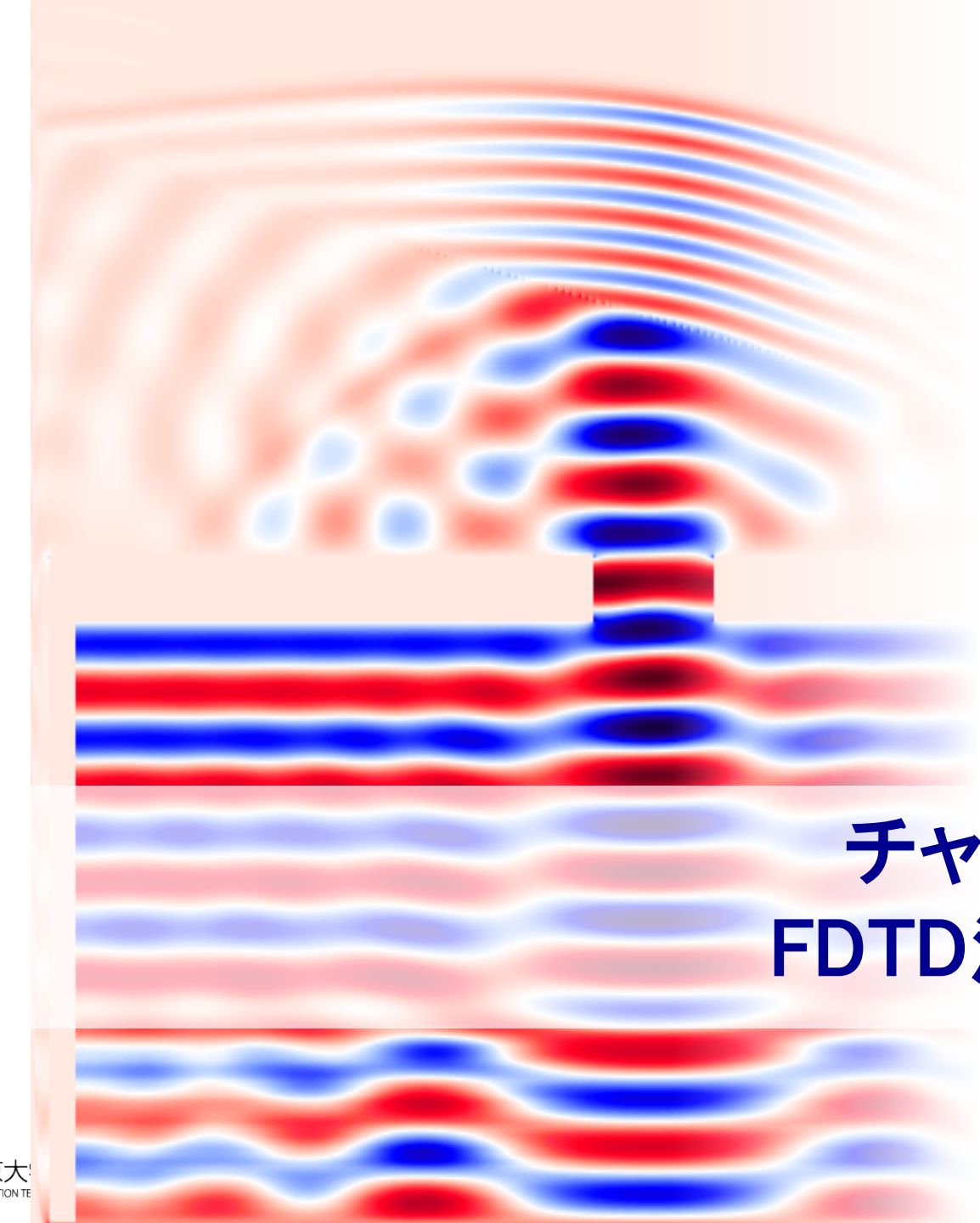
    const int src_ex      = whole.length[0] * (inside.begin[1] - whole.begin[1]);
    const int dst_ex      = whole.length[0] * (inside_end1     - whole.begin[1]);

    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 ompi-cuda
$ cd openacc_mpi_fdtd/01_original
$ make
$ pjsub ./run.sh
$ cat run.sh.??????.out
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]
------------------------------
```

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

なお、pjsub ./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_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

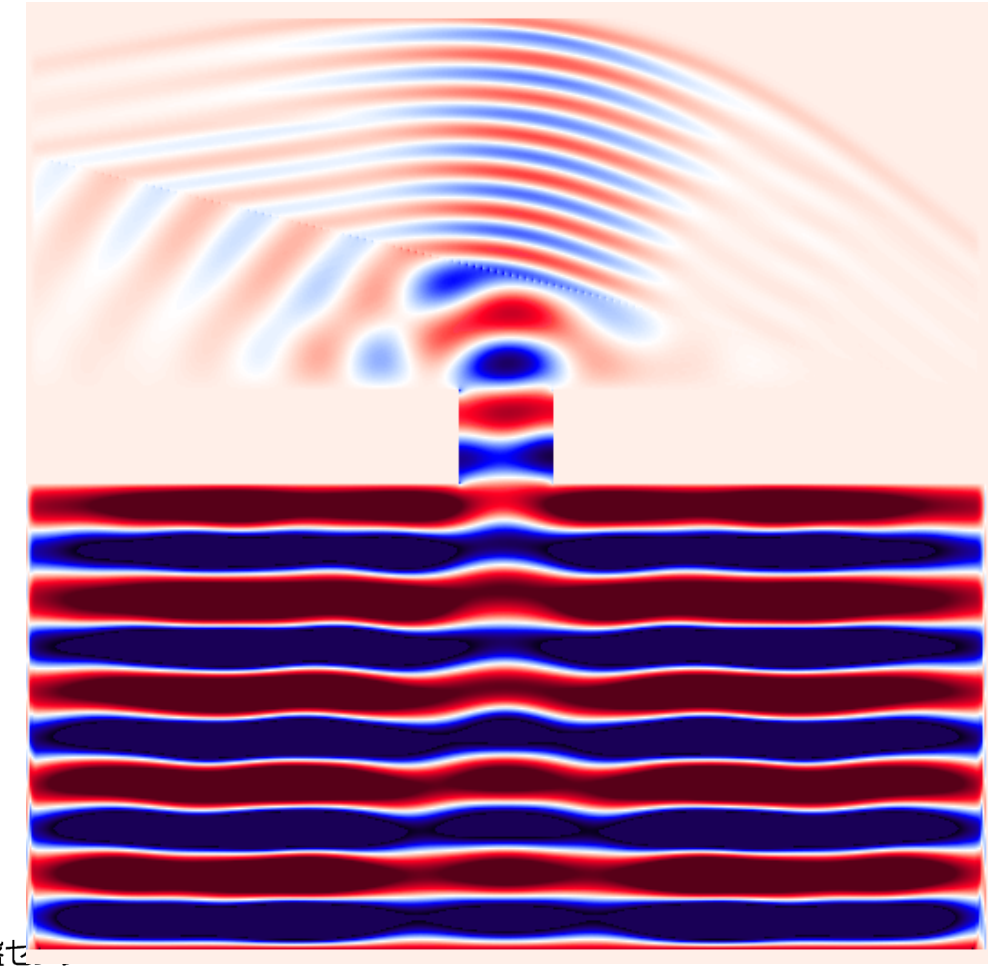
なお

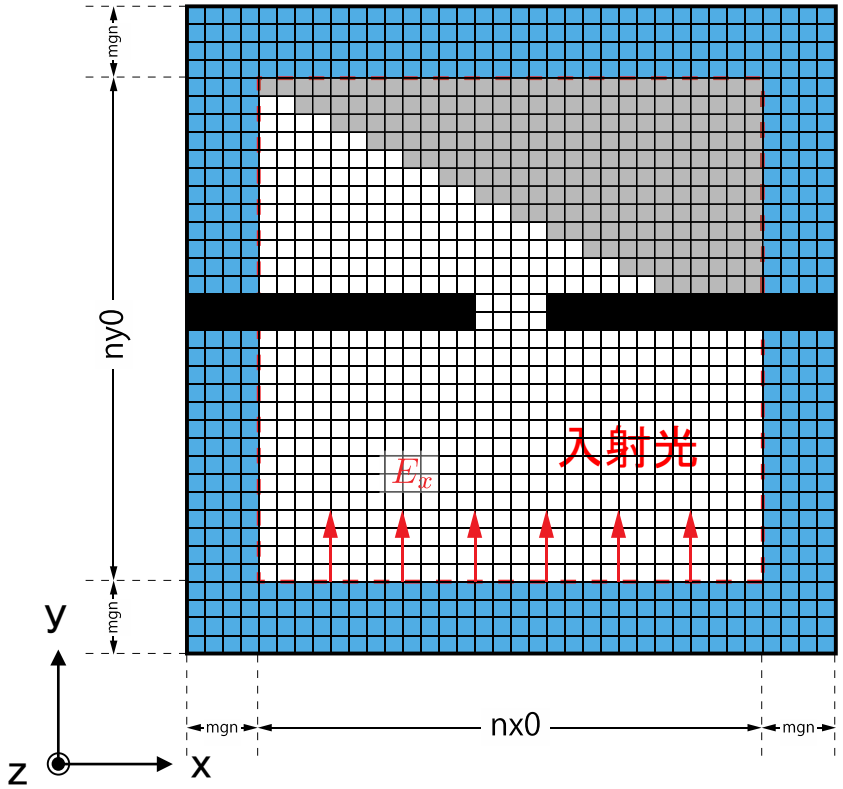
ssh -Y txxxxx@wisteria.cc.u-tokyo.ac.jp

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
  - ✓ Ex（電場の x 成分）の出力





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

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);
        }
```

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。
  - ✓ 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/05_openacc4

実習5

- 実はUnified Memoryを利用するとずっと簡単に実装できます。
  - 実習2・3で行ったdata 指示文に関する実装する必要がないため
- –ta=tesla,cc80,managed として性能を比較してみましょう
  - 性能がだいぶ違うと思います。改善するならどうすべきでしょうか。
    - managed memory はページ単位（結構大きい）でデータ転送を行います
    - 05_openacc4 で使っている update 指示文は、data 指示文で確保済みのデータを CPU-GPU間でコピーするための指示文です。

```
#pragma acc update host(ex[src:sendnelems],ey[src:sendnelems],hz[src:sendnelems])
        for(i = 0;i < sendnelems;i++){
          ex_global[dst+i] = ex[src+i];
          ey_global[dst+i] = ey[src+i];
          hz_global[dst+i] = hz[src+i];
        }
```

src から始まりsendnelems 個の要素をhost(CPU)にコピーする

性能改善済みの解答例は、openacc_fdtd/06_openacc5

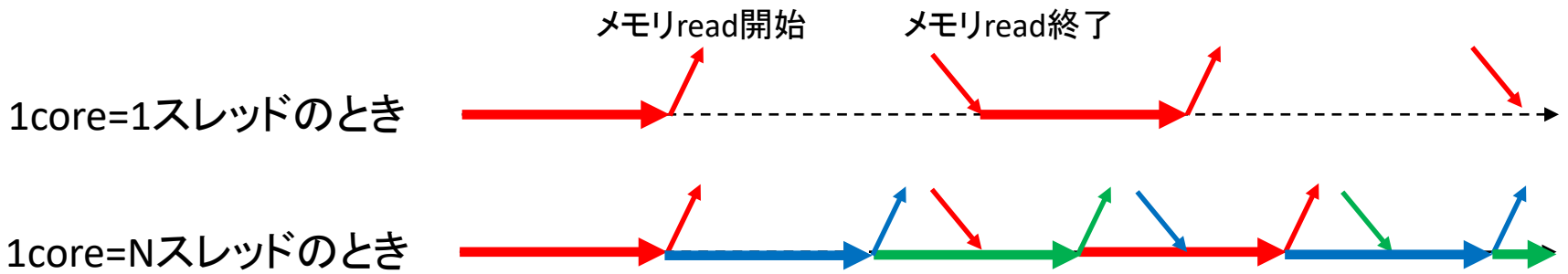
Q & A

- 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/157/
  - アンケートへの協力をお願いします。

補足スライド

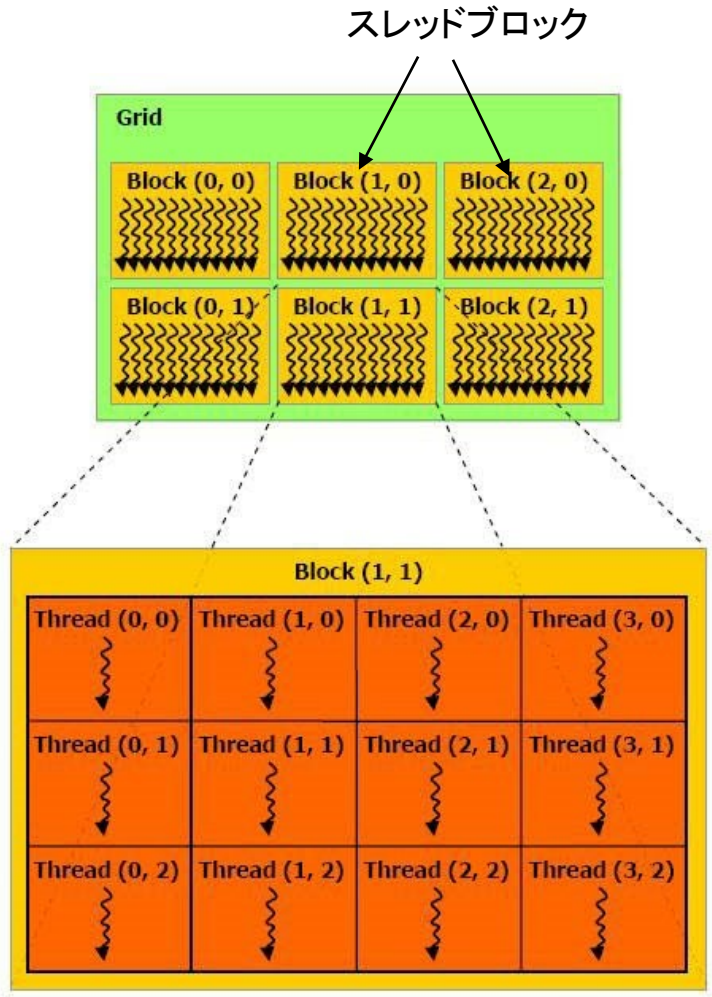
性能を出すためにはスレッド数>>コア数

- 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

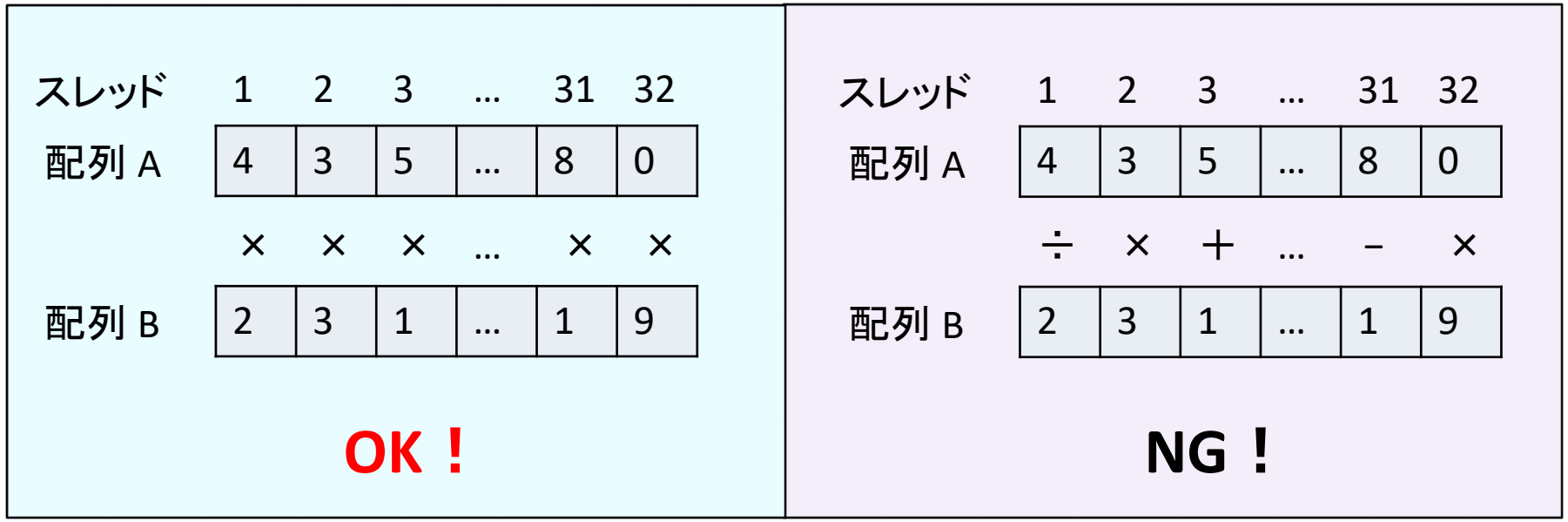
- 階層的なコア/スレッド管理
  - P100は56 SMを持ち、1 SMは64 CUDA coreを持つ。トータル3584 CUDA core
  - 1 SMが複数のスレッドブロックを担当し、1 CUDA core が複数スレッドを担当
- スレッド間のコミュニケーション
  - 同一スレッドブロック内のスレッドは高速コミュニケーション可能
  - 異なるスレッドブロックに属するスレッド間はコミュニケーションが低速
    - いったんメモリに書き出したり、CPUに処理を戻さなくてはならない



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

Warp 単位の実行

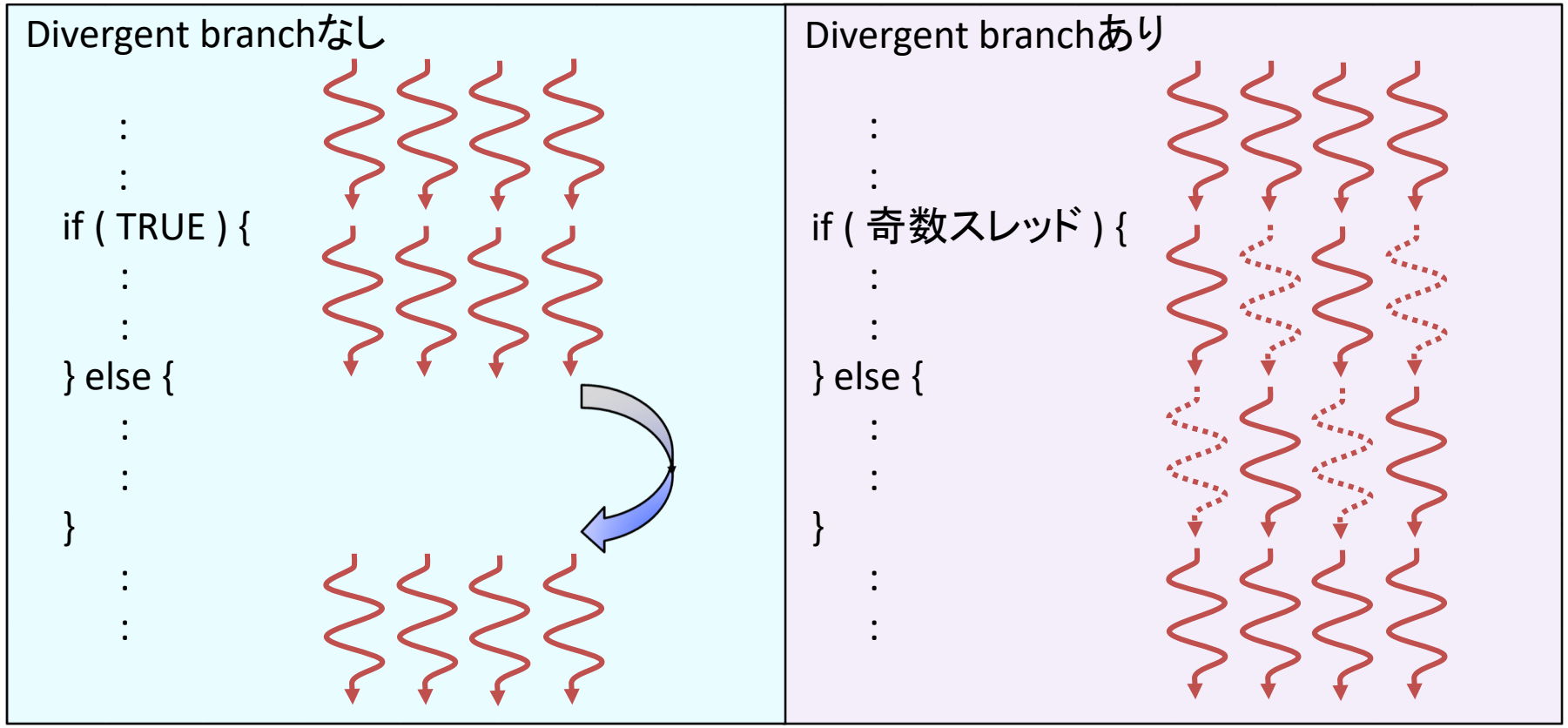
- 連続した32スレッドを1単位 = Warp と呼ぶ
- このWarpは足並み揃えて動く
  - 実行する命令は32スレッド全て同じ
  - データは違ってもいい



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。



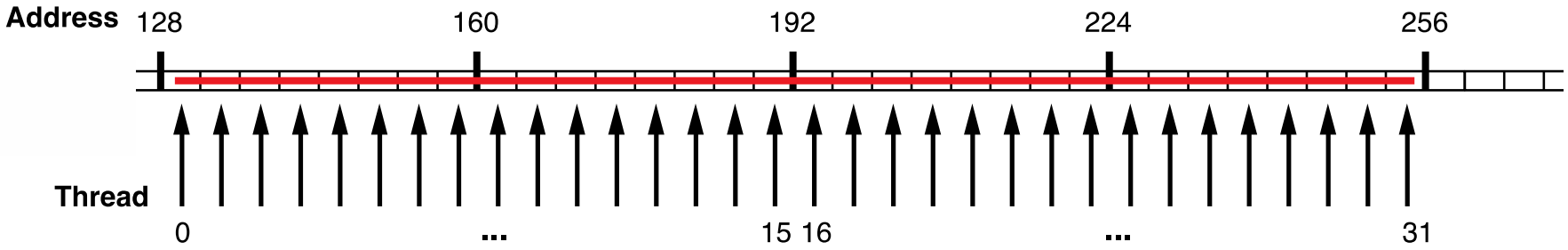
else 部分は実行せずジャンプ

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

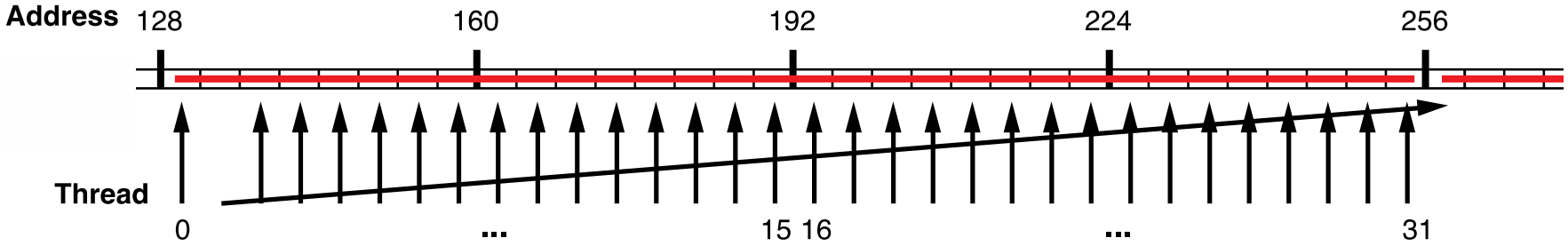
コアレスドアクセス

- 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