

お試しアカウント付き
並列プログラミング講習会

MPI+OpenMPで並列化されたFortran プログラムのGPUへの移行手法

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

✓ 7月24日（水） 13:00 – 17:00

■ プログラム

13:00 – 13:50	OpenACC, OpenMP 5.x, do concurrent, CUDA Fortran の特徴
14:00 – 14:50	OpenMP for CPUからOpenACC, OpenMP 5.x, do concurrentへの移植手法
15:00 – 15:50	演習 1 : 簡単なサンプル
16:00 – 16:50	演習 2 : 有限要素法

講習会について

■ 本講習会は

- ✓ OFP-IIに向け、MPI+OpenMPで記述されたFortranプログラムのGPU移植の手法を中心に扱います。

■ 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-vqRIV7wmU0-w/videos?view=0&sort=p&flow=grid>

講習会の進め方

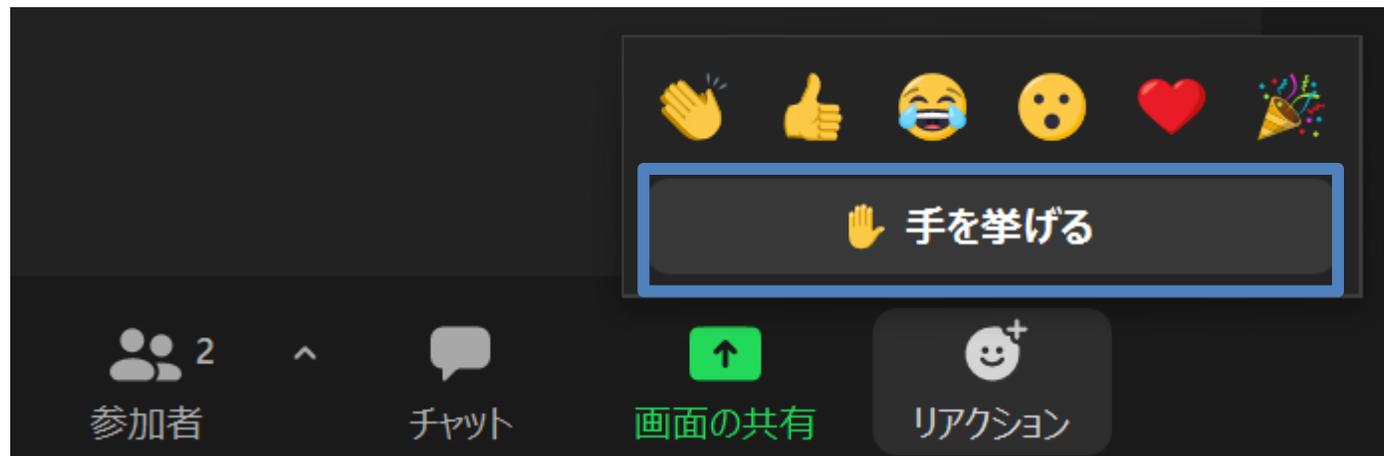
- Zoomを利用したオンライン講習会です
 - ✓ この講義は録画されています
 - ✓ 質問があるとき以外はミュートでお願いします
 - ✓ ビデオもオフを推奨します
- slackを使って質問に対応します
 - ✓ slackはリンクを知っている人は誰でも使える設定になっています
 - ✓ slackのリンクをzoomのチャットに貼るので、未登録の場合は今のうちに登録をお願いします
 - ✓ スクリーンショットなどで画像を共有することで、質問対応します
 - ✓ Windows : Alt + PrtScn で作業中ウィンドウのスクショがクリップボードにコピーされます。slackのチャット部分で貼り付け(Ctrl + V)することで画像をアップロードできます
 - ✓ Mac : command + shift + control + 4 の同時押し、その後撮りたいウィンドウ上でspaceを押すことで、スクリーンショットがクリップボードにコピーされます。slackのチャット部分で貼り付け(command + V)することで画像をアップロードできます

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

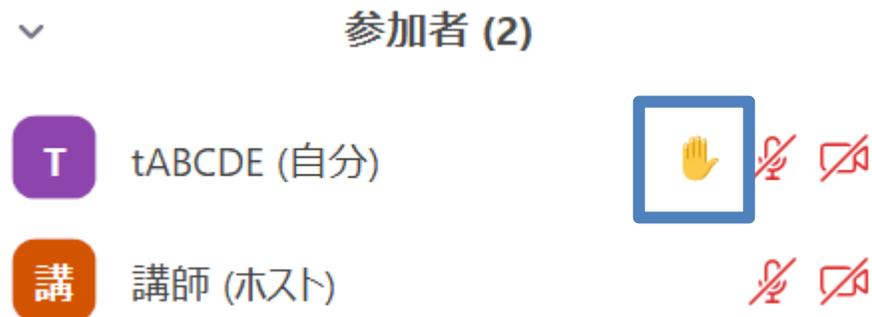


Zoom: 手が挙がっていることの確認方法

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

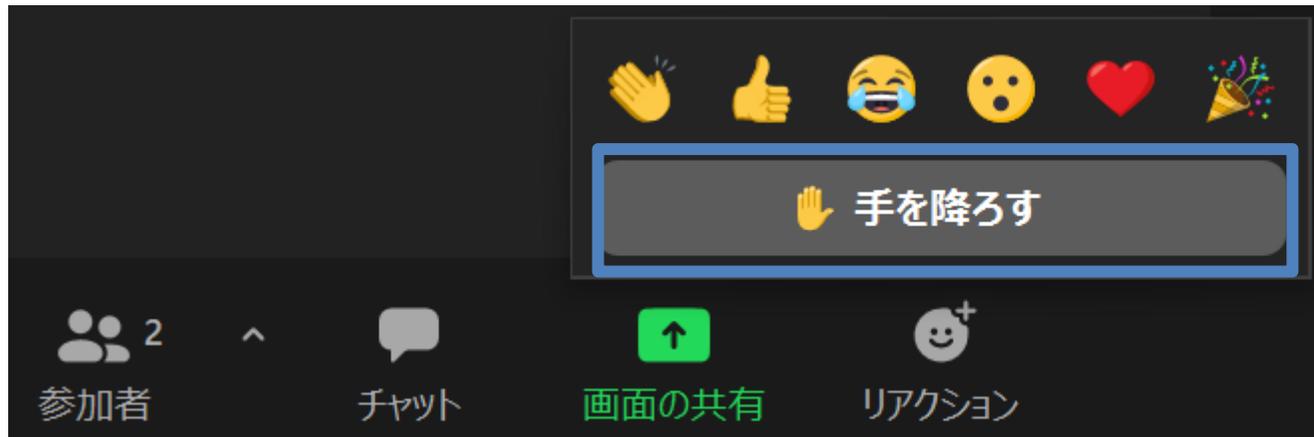


Zoom: 「手を降ろす」方法

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211回-fortran2gpu」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211回-fortran2gpu」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

2001-2005

2006-2010

2011-2015

2016-2020

2021-2025

2026-2030

Hitachi SR8000
1,024 GF

Hitachi SR11000
J1, J2
5.35 TF, 18.8 TF

Hitachi SR16K/M1
Yayoi
54.9 TF

Hitachi
SR2201
307.2GF

Hitachi
SR8000/MPP
2,073.6 GF

OBCX
(Fujitsu)
6.61 PF

Hitachi HA8000
T2K Todai
140 TF

Oakforest-
PACS (Fujitsu)
25.0 PF

OFP-II
150+ PF

Fujitsu FX10
Oakleaf-FX
1.13 PF

 Wisteria
BDEC-01 Fujitsu
33.1 PF

BDEC-
02
250+ PF

Reedbush-
U/H/L (SGI-HPE)
3.36 PF

Ipomoea-01 25PB

Ipomoea-
03

Ipomoea-02

東京大学情報基盤 センターのスパコン

利用者2,600+名

55%は学外

2001-2005

2006-2010

2011-2015

2016-2020

2021-2025

2026-2030

Hitachi SR8000

1.024 PF

SR8000

Hitachi SR11000

J1, J2

IBM Power5+

Hitachi SR16K/M1

Yayoi

IBM Power7

Hitachi SR2201

HARP-1E

Hitachi SR8000/MPP

SR8000

Hitachi HA8000

T2K Todai

140 TF

Oakforest-PACS (Fujitsu)

0.7 PF

OFP-II

150+ PF

Accelerators

AMD Opteron

Intel Xeon Phi

Fujitsu FX10

Oakleaf-FX

1.13 PF

Wisteria BDEC-01 Fujitsu

33.1 PF

BDEC-02

250+ PF

疑似ベクトル

汎用CPU

加速装置付

SPACR64 IXfx

Reedbush-U/H/L (SGI-HPE)

3.36 PF

A64FX, Intel Icelake+ NVIDIA A100

Accelerators

Intel BDW + NVIDIA P100

Ipomoea-01 25PB

Ipomoea-03

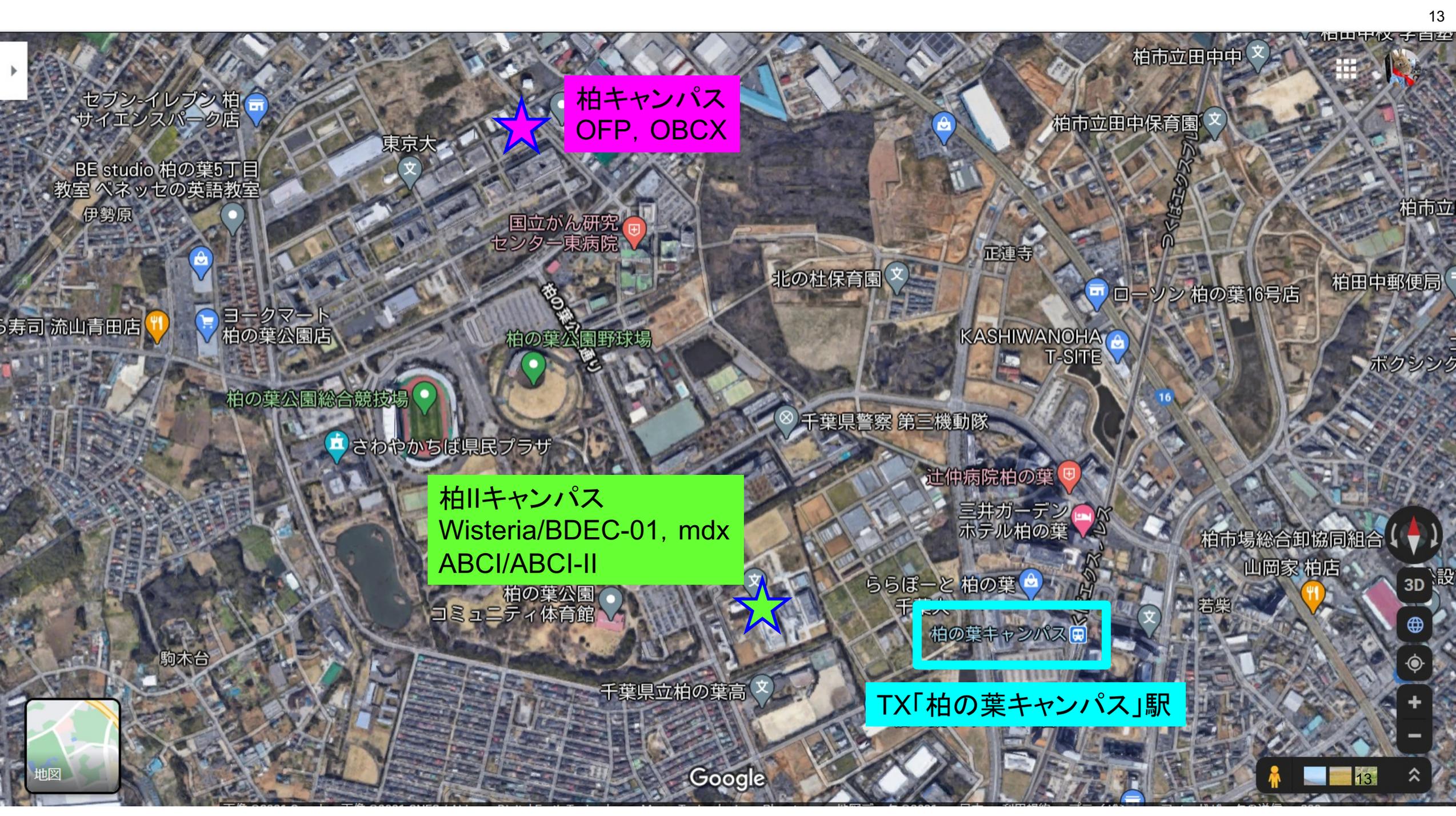
Ipomoea-02

★ 柏キャンパス
OFFP, OBCX

★ 柏IIキャンパス
Wisteria/BDEC-01, mdx
ABCI/ABCI-II

★ 柏の葉キャンパス

TX「柏の葉キャンパス」駅





Platform for Integration of (S+D+L)
Big Data & Extreme Computing

Shared File System (SFS)
25.8 PB, 500 GB/s

Simulation Nodes:
Odyssey
Fujitsu/Arm A64FX
25.9PF, 7.8 PB/s

Data/Learning Nodes:
Aquarius
Intel Ice Lake + NVIDIA A100
7.20 PF, 578.2 TB/s

Fast File System (FFS)
1 PB, 1.0 TB/s

2.0 TB/s

800 Gbps

External Resources



External Network



External Resources



Simulation Nodes
(Odyssey)



Data/Learning Nodes
(Aquarius)



東京大学
THE UNIVERSITY OF TOKYO



東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

Reedbush (HPE, 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
- 2016年7月～2021年11月末(引退)
- 東大ITC初のGPUクラスタ, ピーク性能3.36 PF

Oakforest-PACS (OFP) (Fujitsu, Intel Xeon Phi (KNL))

- JCAHPC (筑波大CCS・東大ITC), 2016年10月～2022年3月末(予定)
- 25 PF, #39 in 58th TOP 500 (November 2021)

Oakbridge-CX (OBCX) (Fujitsu, Intel Xeon CLX)

- 2019年7月～2023年6月末(予定)
- 6.61 PF, #110 in 58th TOP500-June 2023 (Plan)



Wisteria/BDEC-01 (Fujitsu)

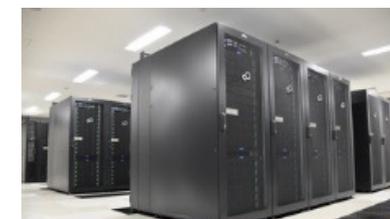
- シミュレーションノード群 (Odyssey) : A64FX (#17)
- データ・学習ノード群 (Aquarius) : Intel Icelake+NVIDIA A100 (#106)
- 33.1 PF, #13 in 57th TOP 500, 2021年5月14日運用開始
- 「計算・データ・学習 (S+D+L)」融合のためのプラットフォーム
- 革新的ソフトウェア基盤「h3-Open-BDEC」
(科研費基盤(S) 2019年度～2023年度)



Reedbush



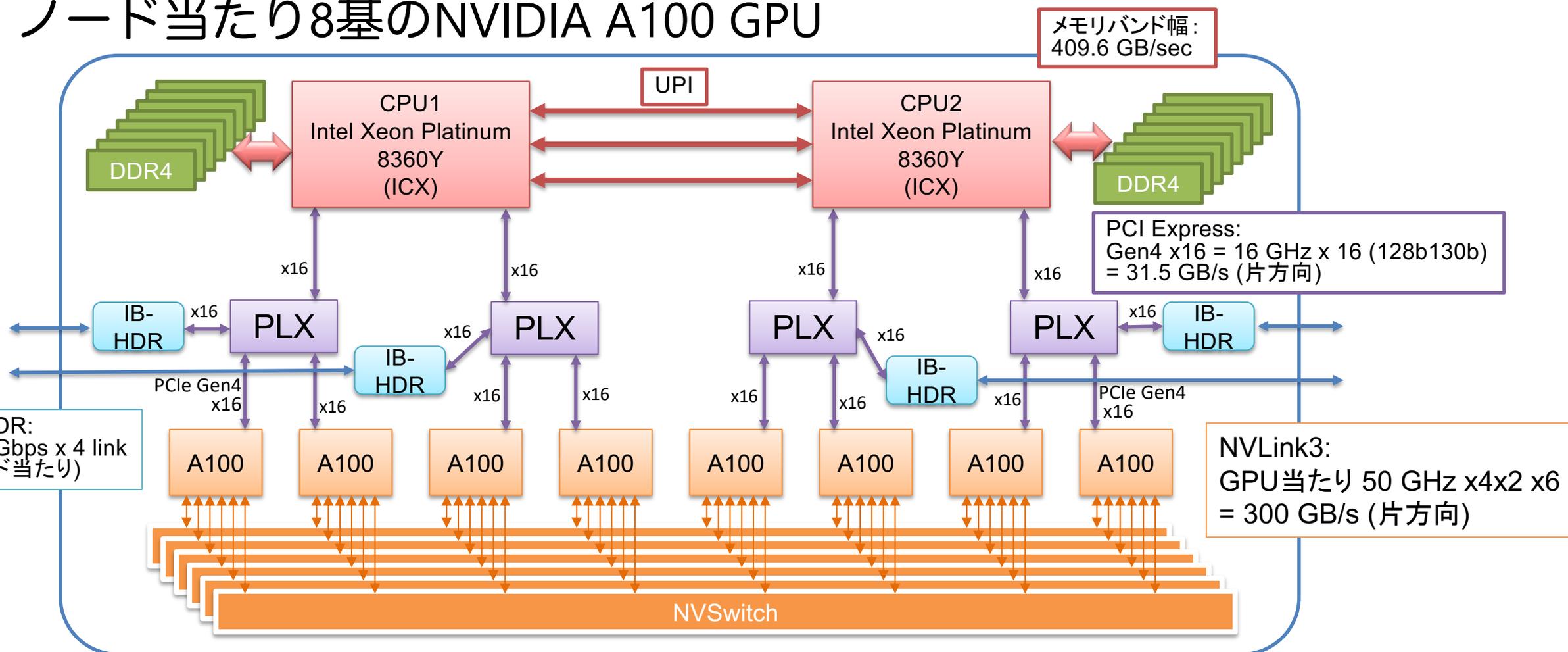
Oakforest-PACS



Oakbridge-CX 14

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 work)
 - ✓ 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 ompi-cuda または
 - \$ module load nvidia nvmpi <- 2023年4月にmodule環境変更

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=2
#PJM -L elapse=00:01:00
#PJM -g gt00

module load nvidia
./a.out
```

リソースグループ名
:lecture-a

利用GPU数

実行時間制限: 1分
(講習会では最大10分)

利用グループ名
:gt00

MPI+OPENMPからのGPUへの移行手法

MPI + "X" (Fortran向け)

■ 従来型のスパコン

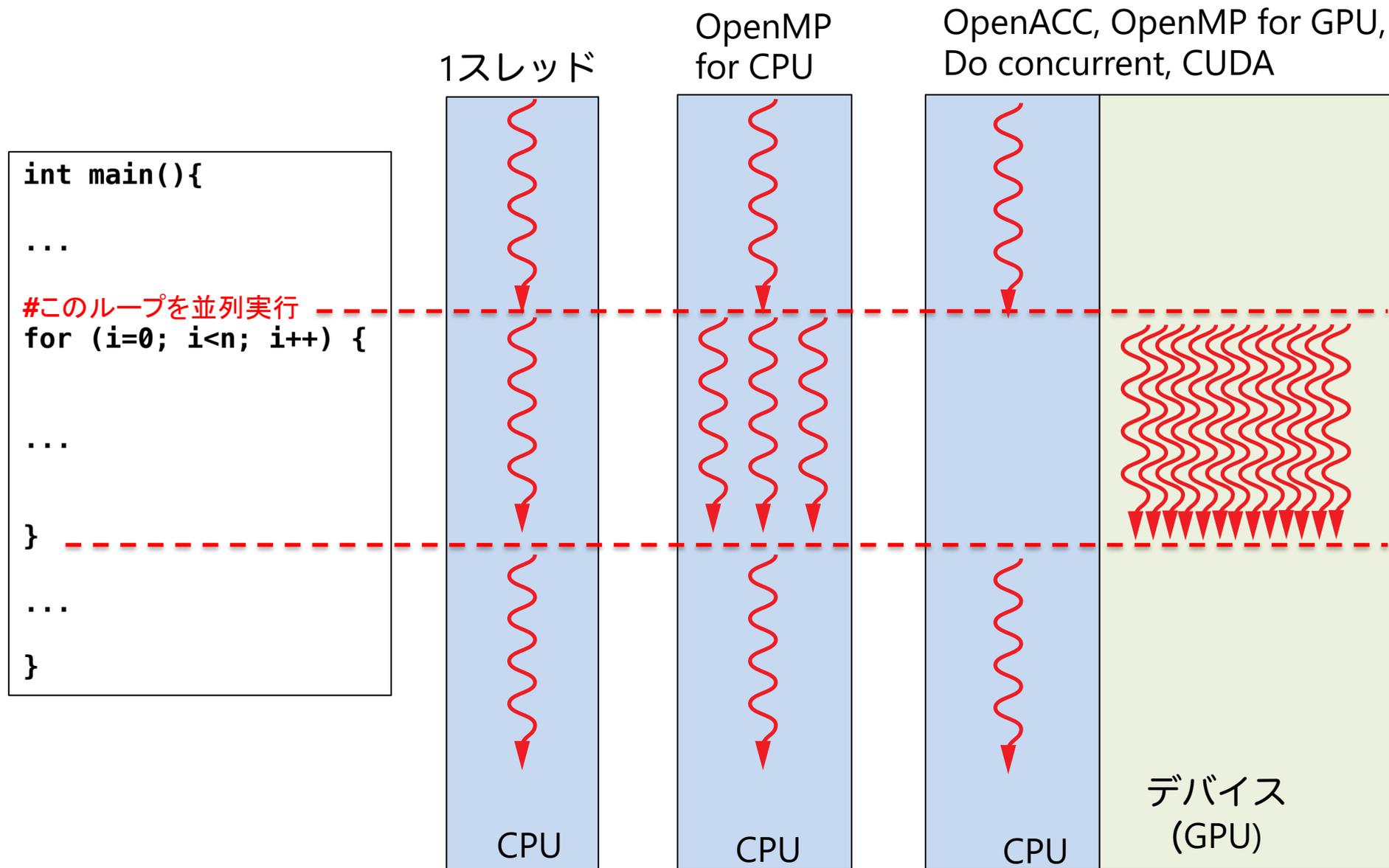
- MPI + OpenMP for CPU (どこのスパコンでもほぼ使える)

■ GPUスパコン

- MPI + OpenACC (NVIDIA)
- MPI + OpenMP for GPU (NVIDIA, AMD, Intel)
- MPI + do concurrent (NVIDIA)
- MPI + CUDA Fortran (NVIDIA)

従来型のスパコンではほとんどの場合"X=OpenMP"だったが、GPUスパコンでは"X"の選択肢が複数ある状態

GPUプログラムの実行イメージ



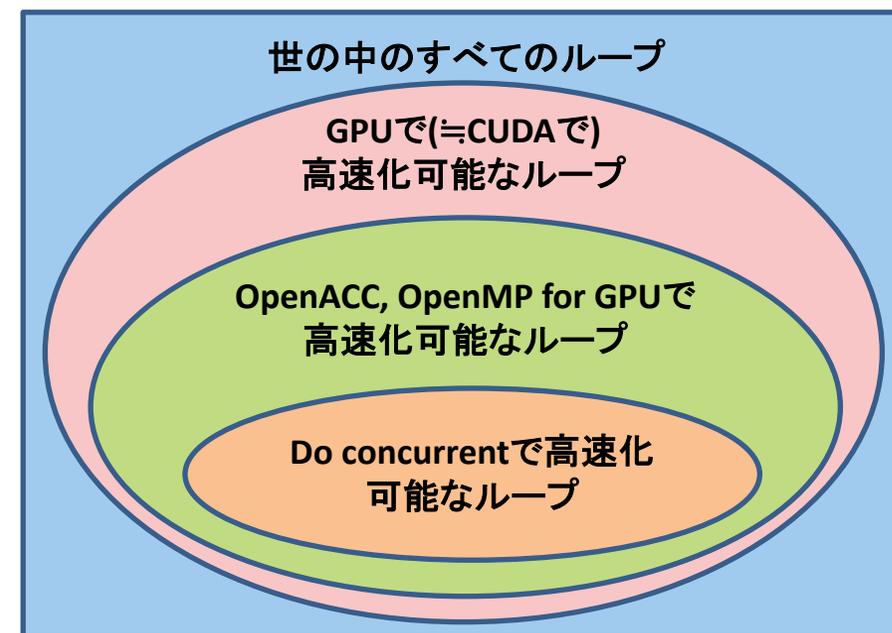
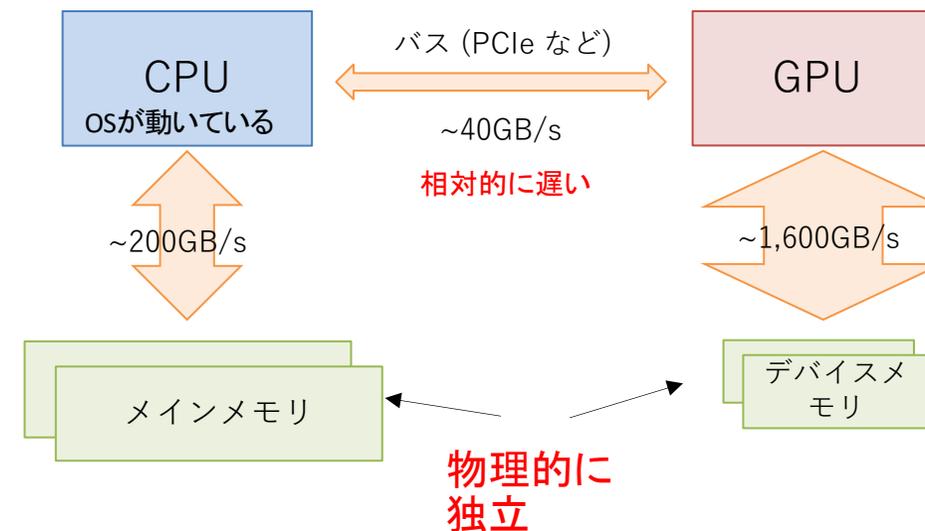
GPUプログラミングの必須要素

1. CPU-GPU間データ移動

- CPUとGPUで独立のメモリを持つため
- メモリ管理方針は“X”により異なる
 - CPU-GPU 変数独立型
 - 変数同一視型
 - Unified memory型（メモリアドレス空間ごと同一視）

2. ループ並列化

- いずれの“X”でも、プログラム中のループ構造を並列化することでGPUで高速実行する
- “X”によって並列化できるループ、できないループがある



仕様の比較 (nvidia compiler 22.5 の場合)

	OpenACC kernels	OpenMP 4.x target teams distribute parallel do	OpenMP 5.x target loop	Fortran do concurrent	CUDA Fortran
CPU-GPUメモリ管理	変数同一視 Unified Memory ※1	変数同一視 Unified Memory ※1	変数同一視 Unified Memory ※1	Unified Memory	変数独立
関数呼び出し	✓	✓	✓	N/A ※2	✓
リダクション	✓	✓	✓	✓ ※3	✓
atomic演算	✓	✓	✓	N/A	✓
スレッド数の調整	✓ ※4	✓ ※4	N/A	N/A	✓
Streamの非同期 実行	✓	N/A ※5	N/A ※5	N/A	✓
GPU組み込み関数	N/A	N/A	N/A	N/A	✓
単一ソースでの CPU・GPU両対応	✓	△	✓	✓	N/A

※1 Unified memoryはNVIDIA GPU, compilerの機能であり、OpenACC, OpenMPの仕様準拠ではない。

※2 Fortranの仕様上は、pure functionであれば呼び出せるはずだが、pure function呼び出しもサポートされていない。

※3 NVIDIA compilerの独自拡張であり、Fortran do concurrentの仕様準拠ではない。

※4 CUDA程自由ではない。32以上の2の冪乗数以外を指定しても無視されることが多い。OpenACCではkernelsではなくparallelを使うと自由度が少し上がる。

※5 omp task指示文と組み合わせると書けそうな気がするが、うまくいかない。

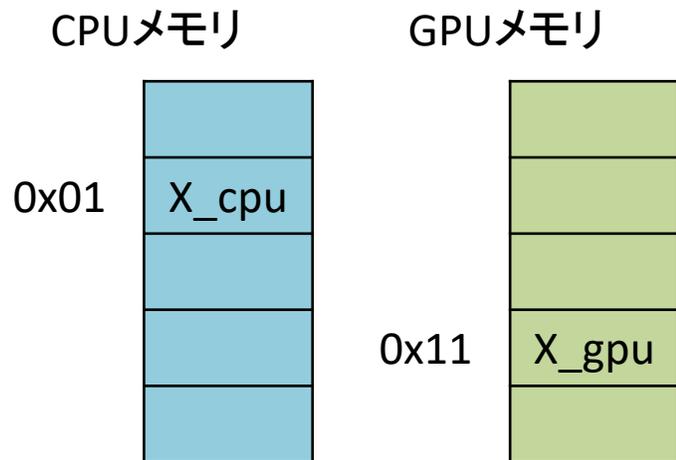
CPU-GPUのメモリ管理

- CPU-GPU変数独立型 (CUDA)
 - 配列ごとのPinned memoryの設定など、**細やかな指定ができる**
 - Pinned memory: ページロックメモリ。CPU-GPU間のデータ転送速度が速い
 - CPU-GPUの変数の一貫性をプログラマが明示的に管理する必要があり、**管理コストが大きい**
- 変数同一視型 (OpenACC, OpenMP)
 - CPU-GPUの変数の区別がプログラムの見かけ上はなくなるので、**コードの見通しがいいが、CPU-GPU間の一貫性の管理は依然プログラマの仕事**
 - 配列ごとのPinned memoryの設定など、**細かな指定はできない**
 - NVIDIAコンパイラではオプションによる一括指定はできる
- Unified Memory型 (do concurrent, OpenACC, OpenMP)
 - CPU-GPU間の一貫性の管理から解放されるため、**圧倒的に楽**
 - CPU-GPU間のデータ転送を**書く必要がない**
 - **ポインタを含む構造体などもそのまま使える**
 - **速度的に不利**
 - ページ単位 (ページサイズの設定不可、サイズ非公開) で転送するため
 - GPU Direct RDMA通信が使えない
 - CPU-GPU間通信がバックグラウンドで実行されるため、**高速化を目指す上でわかりづらい**

CPU-GPU間データ転送の書き方

■ 変数独立型

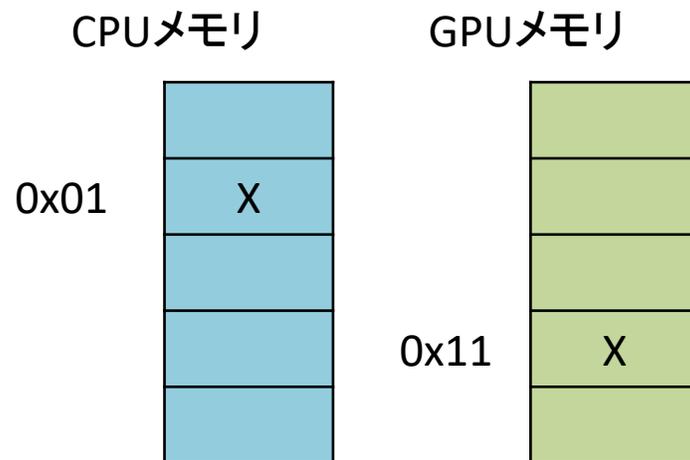
```
Real(8), allocatable :: X_cpu(:)
Real(8), allocatable, device :: X_gpu(:)
allocate(X_cpu(N))
allocate(X_gpu(N))
X_cpu(:) = 0
X_gpu(:) = X_cpu(:)
Call cuda_func<<<tb,th>>(X_gpu)
X_cpu(:) = X_gpu(:)
```



■ 変数同一視型

```
Real(8), allocatable :: X(:)
allocate(X(N))
X(:) = 0
!$acc data copy(X)
Call openacc_func(X)
!$acc end data
```

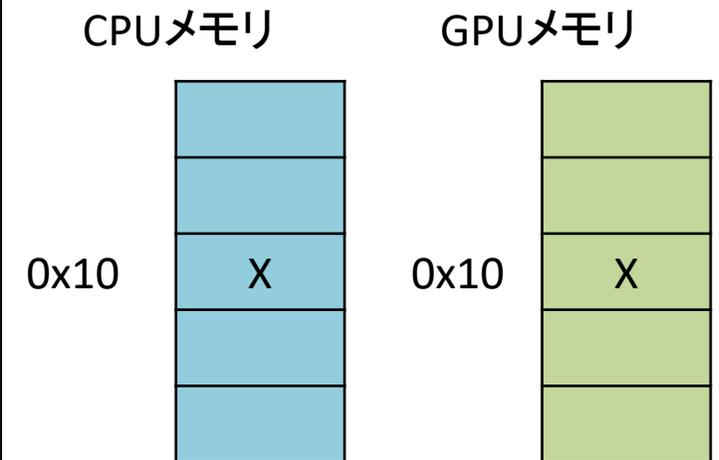
(X_cpu, X_gpu)のペアをソフトウェア的に変数Xとして同一視する



■ Unified Memory

```
Real(8), allocatable :: X(:)
allocate(X(N))
X(:) = 0
Call do_concurrent_func(X)
```

メモリアドレス空間を一致させることで、X_cpu, X_gpuの区別を無くす
データ転送はページフォルトが起きた際にバックグラウンドで行われる

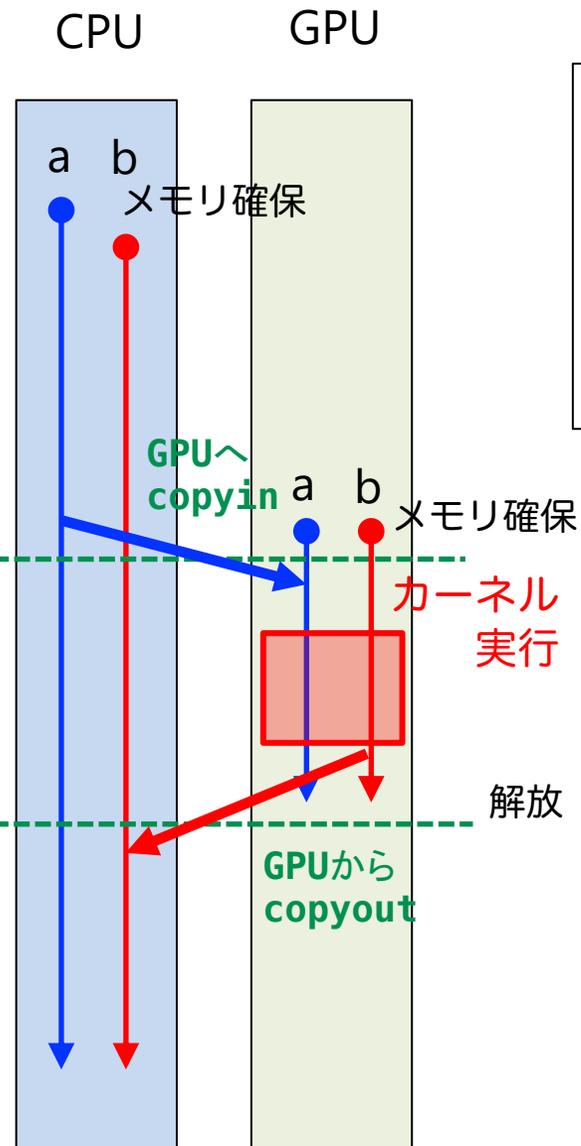


CPU-GPU間のデータ移動

OpenACC data指示文を使った場合

```
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
```



GPUのメモリ上にコピーを作る。
これは変数独立型、変数同一視型、Unified Memory型のいずれの場合でも同じ。

“X”に何を選ぶべきか（私見）

- OpenACC（推奨）
 - NVIDIAのGPUで最も使い勝手が良く、コンパイラが成熟している
 - AMD, Intelがサポートすることはおそらくない
 - CUDA Fortranとも組み合わせやすい
- OpenMP for GPU
 - NVIDIA, AMD, Intel全てがサポートすることが最大のメリット
 - OpenACCと仕様上の差はそれほど大きく無いが、細かいところで微妙に使いにくい
 - ループに対するスレッドマッピングの制限など
 - AMDのFortranサポートは特に遅れており、Fortranを使うなら事実上NVIDIAしか選択肢がない
 - IntelはGPUの開発自体が遅れている（GPU開発のトップが今月辞めてしまった）
- Do concurrent（Fortranの言語標準）
 - 言語標準であるということが最大のメリット
 - NVIDIAが今最も力を入れている
 - 仕様上の制約が非常に大きく、実アプリで使うならそれなりに書き換える必要がある
 - 書き換えを進めていき、do concurrentではどうしようもなくなって、OpenACCやCUDAを混ぜた瞬間、言語標準というメリットを失う

データ転送方式は？（私見）

■ OpenACC

- ひとまずはUnified Memoryから始めることを推奨
- ループの並列化に集中できる
- 速度的な問題が出てきたら、データ指示文を追加すれば良い

■ OpenMP for GPU

- ひとまずはUnified Memoryから始めることを推奨
- メリットはOpenACCの場合と同じ
- Unified MemoryはOpenMPやOpenACCの仕様に組み込まれているわけではないので、**NVIDIA以外で動くかどうかはわからない**
 - OpenACCはNVIDIAしかサポートしていない（GCCとかもサポートしていることになっているが使い物にならない）ので、その点はデメリットにならない

■ Do concurrent

- Unified memoryしか選択肢がないため、**速度的な問題が起きても解消できない**
 - H100以降ではUnified Memory機能が強化され、速くなるらしい

CUDAの立ち位置は？（私見）

- ライブラリ的な利用方法を推奨
 - 単純なループにおいて、CUDA以外との性能差はほぼない
 - 指示文などではどうしてもない複雑なループや、少しでも速くしたいアプリケーションのメインのボトルネック部分などは、CUDAの利用を考えるべき
 - いずれの“X”でもCUDA Fortranと組み合わせられる
 - 少なくとも、アプリケーション全体をCUDAに置き換える積極的なモチベーションはもはや無い
- AMD, Intelは？
 - AMD: hipと呼ばれるほぼCUDA C互換のものがある。よく出来ているが、Fortran対応はない。Cメインのユーザであるなら、AMDは現時点でも強力な選択肢
 - Intel: DPC++なるものを推している。Fortran対応はない

OpenMP for CPUとの大きな違い

- OpenACC, OpenMP for GPU, do concurrentでは、**明示的にスレッド番号を取得できない**
 - `omp_get_thread_num()`, `omp_get_num_threads()` 相当のAPIはない
 - **OpenMP for GPU でも使えない**。OpenMP for GPUとOpenACCは似ているが、OpenMP for GPUとOpenMP for CPUは別物。
 - よって、`allocate(array(m, n, omp_num_threads))` みたいな使い方はできない
 - そのアルゴリズムはそもそもGPUに向いてない可能性が高い
 - CUDAではスレッド番号を取得できる
- OpenACC, OpenMP for GPU, do concurrentでは、**明示的なスレッド間同期が取れない**
 - `#pragma omp barrier` 相当の指示文がない
 - GPU化対象ループ終了時に暗黙に同期が取られる
 - CPUとGPUの同期を取る（GPUの終了を待つ）ための指示文は存在する
 - CUDAでは同ースレッドブロック内のスレッド間に限り同期を取れる
 - またCUDAではごく限定的なケースにおいて、GPU全体のスレッドで同期を取ることにも一応できるが、ほとんど使うケースはない

ループ並列化の例：一重ループ

OpenMP for CPU

```
!$omp parallel do
do i = 1, n
  y(i) = a * x(i) + y(i)
end do
!$omp end parallel do
```

OpenMP 4.x teams distribute parallel do

```
!$omp target
!$omp teams distribute parallel do
do i = 1, n
  y(i) = a * x(i) + y(i)
end do
!$omp end target
```

OpenMP 5.x target loop

```
!$omp target
!$omp loop
do i = 1, n
  y(i) = a * x(i) + y(i)
end do
!$omp end target
```

OpenACC

```
!$acc kernels
!$acc loop independent
do i = 1, n
  y(i) = a * x(i) + y(i)
end do
!$acc end kernels
```

Fortran do concurrent

```
do concurrent (i = 1: n)
  y(i) = a * x(i) + y(i)
end do
```

基本的に、このループは並列化可能ですよと宣言しているだけ。CUDA含め比較しても、性能はほぼ変わらない。

ループ並列化の例：多重ループ

OpenMP for CPU

```
!$omp parallel do private(i,j)
do k = 1, n
  do j = 1, n
    do i = 1, n
      y(i,j,k) = a * x(i,j,k) + y(i,j,k)
    end do
  end do
end do
!$omp end parallel do
```

OpenMP 4.x

teams distribute parallel do

```
!$omp target
!$omp teams distribute
do k = 1, n
  do j = 1, n
    !$omp parallel do
      do i = 1, n
        y(i,j,k) = a * x(i,j,k) + y(i,j,k)
      end do
    end do
  end do
end do
!$omp end target
```

OpenMP 5.x

target loop

```
!$omp target
!$omp loop
do k = 1, n
  !$omp loop
  do j = 1, n
    !$omp loop
    do i = 1, n
      y(i,j,k) = a * x(i,j,k) + y(i,j,k)
    end do
  end do
end do
!$omp end target
```

OpenACC

```
!$acc kernels
!$acc loop independent
do k = 1, n
  !$acc loop independent
  do j = 1, n
    !$acc loop independent
    do i = 1, n
      y(i,j,k) = a * x(i,j,k) + y(i,j,k)
    end do
  end do
end do
!$acc end kernels
```

Fortran do concurrent

```
do concurrent (k = 1: n) local(i,j)
  do concurrent (j = 1: n) local(i)
    do concurrent (i = 1: n)
      y(i,j,k) = a * x(i,j,k) + y(i,j,k)
    end do
  end do
end do
```

単純なループ構造であれば、基本的には各ループの並列可能性を示せばいい。

多重ループでは階層的な並列性を意識する必要性が出てくる。(特に OpenMP 4.x)

ループ並列化の例：多重ループの一重化

OpenMP for CPU

```
!$omp parallel do collapse(3)
do k = 1, n
  do j = 1, n
    do i = 1, n
      y(i,j,k) = a * x(i,j,k) + y(i,j,k)
    end do
  end do
end do
!$omp end parallel do
```

OpenMP 4.x teams distribute parallel do

```
!$omp target
!$omp teams distribute parallel do collapse(3)
do k = 1, n
  do j = 1, n
    do i = 1, n
      y(i,j,k) = a * x(i,j,k) + y(i,j,k)
    end do
  end do
end do
!$omp end target
```

OpenMP 5.x target loop

```
!$omp target
!$omp loop collapse(3)
do k = 1, n
  do j = 1, n
    do i = 1, n
      y(i,j,k) = a * x(i,j,k) + y(i,j,k)
    end do
  end do
end do
!$omp end target
```

OpenACC

```
!$acc kernels
!$acc loop independent collapse(3)
do k = 1, n
  do j = 1, n
    do i = 1, n
      y(i,j,k) = a * x(i,j,k) + y(i,j,k)
    end do
  end do
end do
!$acc end kernels
```

Fortran do concurrent

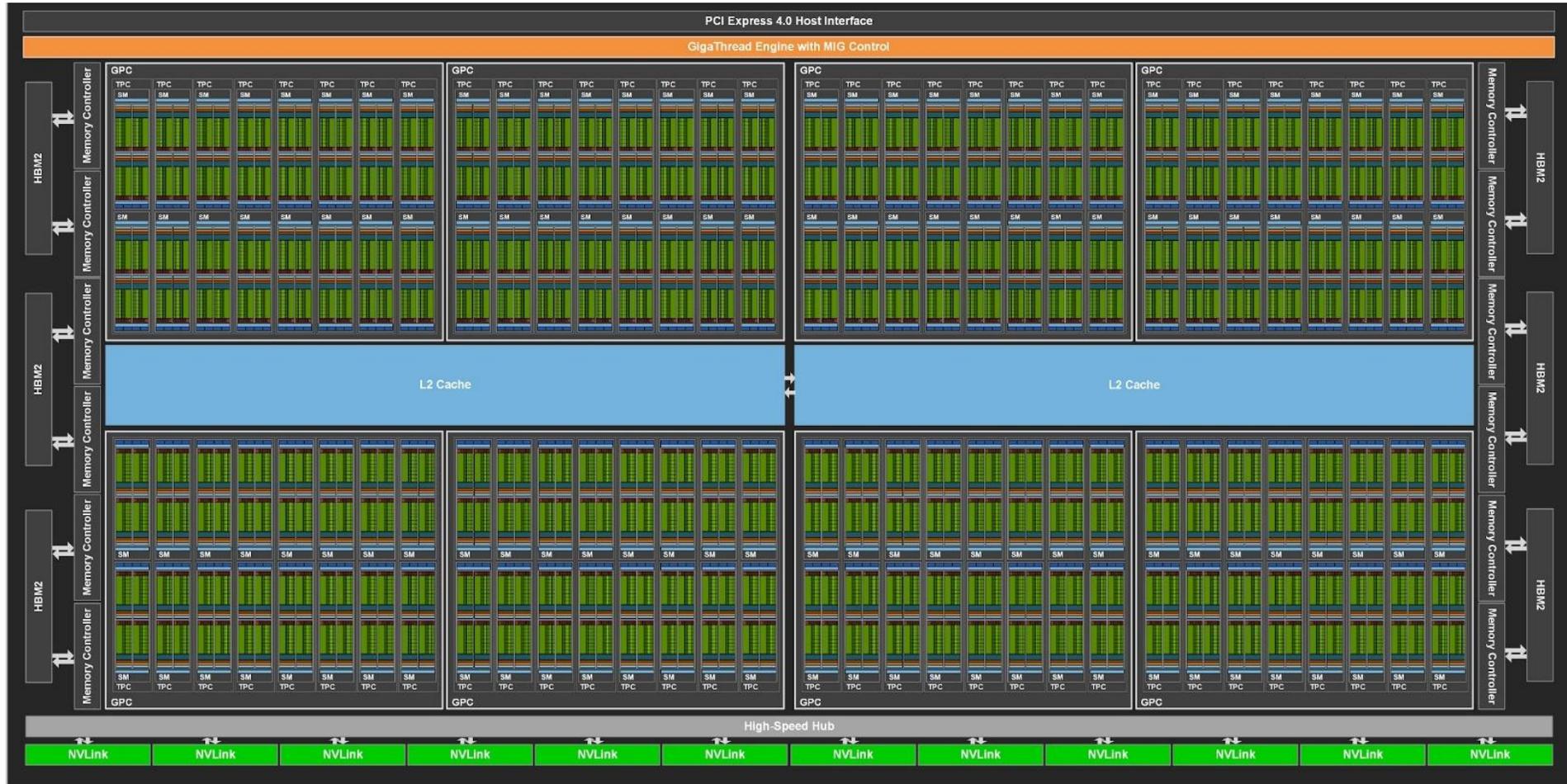
```
do concurrent (i=1:n, j=1:n, k=1:n)
  y(i,j,k) = a * x(i,j,k) + y(i,j,k)
end do
```

Do concurrentでこのように書いた時、仕様上ループの順序についての決めが無い場合、どのループを最内として扱うかはコンパイラが決めることになる

Collapse節によって一重化できるなら、した方が良くかもしれない。特にOpenMP 4.xでは並列性を確保するためにcollapseが重要。

GPUの階層構造 (1/2)

- A100は108個のSM (Streaming Multiprocessor)を持つ



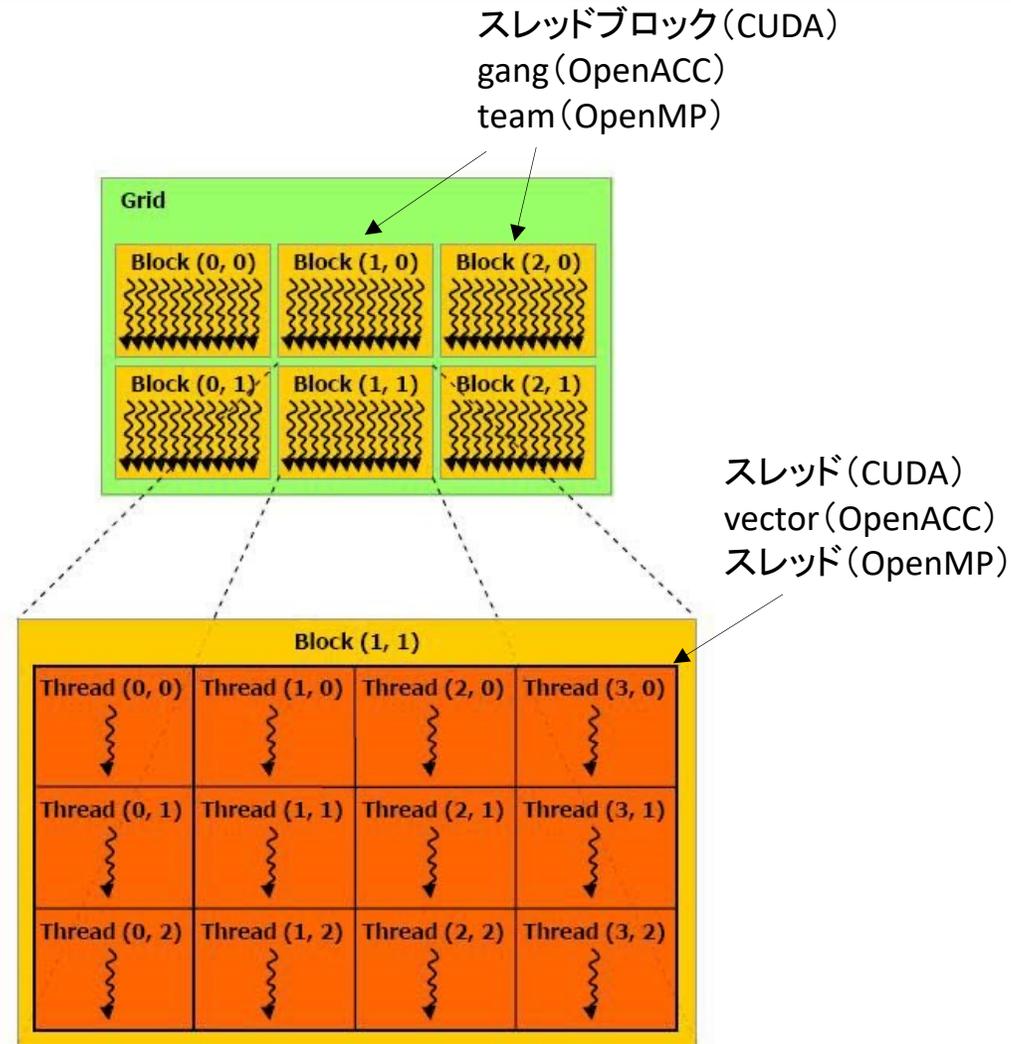
GPUの階層構造 (2/2)

- L2キャッシュはSM全体で共有
 - L3キャッシュはないので、L2がラストレベル
- L1キャッシュはSM毎に独立
 - Shared memoryと共有で192 KB
 - キャッシュブロッキングを考えるならL1キャッシュ単位
- 同期ができるのもSMの内部のみ
 - 限定的な条件下で全体同期もできるが、あまり使わない
- SM内のスレッドは更にWarpと呼ばれる32スレッド単位で実行される



GPUプログラミングの階層構造

- CUDA, OpenACC, OpenMP for GPU も、GPUの階層構造に合わせて、階層的な構造を持つ
 - CUDA: スレッドブロック、スレッド
 - OpenACC: gang, (worker,) vector
 - OpenMP: team, thread
- 一つのスレッドブロック/gang/teamに所属するスレッドは同じSMに割り当てられる
 - つまりL1キャッシュが共有される
 - 1ブロックあたりのスレッド数は1~1024
 - Warpを踏まえると、32の倍数、経験的に64, 128, 256あたりがいい
 - CPU-GPU間のデータ移動や、リソースを使い切ることの方が重要なので、最初からそこまで意識する必要はない



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

スレッド割り当ての自由度の制限に関する評価

- GPUへの移行対象のアプリの多くはCPU向けに最適化されている
- 代表的な最適化手法として、キャッシュブロッキングがある
 - キャッシュブロッキングはループを複雑化させるが、**do concurrent**で複雑なループは対応可能か？

• 行列・行列積

- ✓ 計算律速なカーネルの例
- ✓ キャッシュブロッキングが効果的
- ✓ 単に行列・行列積を高速化するならBLASを使うべきだが、例題として利用

```
do j = 1, n
  do i = 1, n
    do k = 1, n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end do
  end do
end do
```

C = A * B の行列積
(naïve実装)

```
do j = 1, n, BLK
  do i = 1, n, BLK
    do k = 1, n, BLK
      do jj = 0, BLK-1
        do ii = 0, BLK-1
          do kk = 0, BLK-1
            C(i+ii,j+jj) = C(i+ii,j+jj) &
              + A(i+ii,k+kk) * B(k+kk,j+jj)
          end do
        end do
      end do
    end do
  end do
end do
```

C = A * B の行列積
(ブロッキング実装)

ブロッキングされた行列積の実装

Do concurrent 実装

```
do concurrent (i = 1:n:BLOCKX, j = 1:n:BLOCKY)
  do k = 1, n, BLOCKX
    do concurrent (ii = 0:BLOCKX-1, jj = 0: BLOCKY-1)
      do kk = 0, BLOCKX-1
        C(i+ii, j+jj) = C(i+ii, j+jj) &
          + A(i+ii, k+kk) * B(k+kk, j+jj)
      end do
    end do
  end do
end do
```

- 最外のi, jループを一重化し、スレッドブロックレベルの並列化が行われることを**期待**
- 内側のii, jjループも一重化し、スレッドレベルの並列化が行われることを**期待**

OpenACC 実装

```
!$acc kernels present(A,B,C)
!$acc loop collapse(2) independent gang
do j = 1, n, BLOCKY
  do i = 1, n, BLOCKX
    !$acc loop seq
    do k = 1, n, BLOCKX
      !$acc loop independent tile(BLOCKX,BLOCKY)
      do jj = 0, BLOCKY-1
        do ii = 0, BLOCKX-1
          do kk = 0, BLOCKX-1
            C(i+ii, j+jj) = C(i+ii, j+jj) &
              + A(i+ii, k+kk) * B(k+kk, j+jj)
          end do
        end do
      end do
    end do
  end do
end do
!$acc end kernels
```

- 最外のi, jループを一重化し、スレッドブロックレベルの並列化を行うよう**指示**
- 内側のii, jjループを二次元のタイル状に並べたスレッドで並列化するよう**指示**

行列・行列積

- GPUではナイーブ実装の差は小さい
- Do concurrent (std) ではループが複雑化すると性能低下
 - CUDA等を組み合わせると標準性を失う
- Do concurrent (std) ではGPU向けに複数ループを並列化するとCPUで性能低下

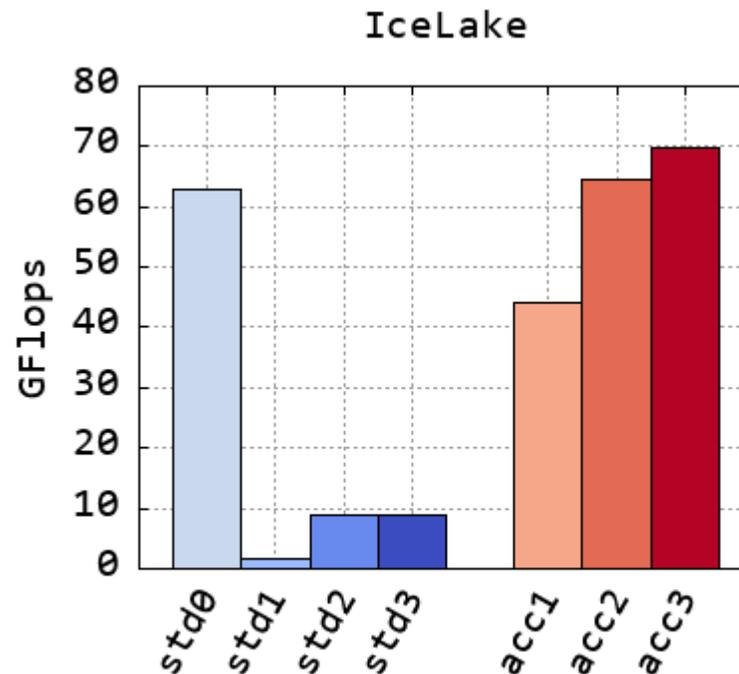
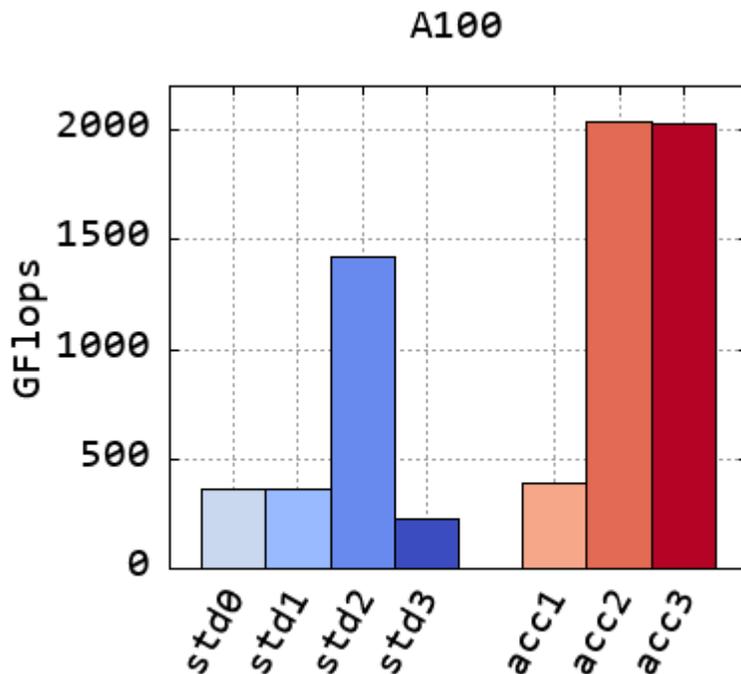
実装0	ナイーブな三重ループの最外ループだけ並列化
実装1	ナイーブな三重ループの外側2つのループの並列化
実装2	六重ループのブロッキング
実装3	ローカルな配列を使った六重ループのブロッキング

実装3

```

do j = 1, n, BLK
  do i = 1, n, BLK
    do jj = 0, BLK-1
      do ii = 0, BLK-1
        cc1(ii,jj) = 0.0
      end do
    end do
  end do
  do k = 1, n, BLK
    do jj = 0, BLK-1
      do ii = 0, BLK-1
        ccc = 0.0d0
        do kk = 0, BLK-1
          ccc = ccc + A(i+ii,k+kk) * B(k+kk,j+jj)
        end do
        cc1(ii,jj) = cc1(ii,jj) + ccc
      end do
    end do
  end do
  do jj = 0, BLK-1
    do ii = 0, BLK-1
      C(i+ii,j+jj) = cc1(ii,jj)
    enddo
  enddo
end do
end do

```



※A100 コンパイルコマンド: nvfortran (version 22.5) (-stdpar=gpu -acc=gpu) -Minfo=accel,mp -gpu=cc80,managed
 ※IceLake コンパイルコマンド: nvfortran (version 22.5) (-stdpar=multicore -acc=multicore) -O3 -mavx512f

ループ並列化の例：リダクションループ

OpenMP for CPU

```
!$omp parallel do reduction(+:sum)
do i = 1, n
  sum = sum + x(i)
end do
!$omp end parallel do
```

OpenMP for GPU

```
!$omp target
!$omp loop reduction(+:sum)
do i = 1, n
  sum = sum + x(i)
end do
!$omp end target
```

OpenACC

```
!$acc kernels
!$acc loop independent reduction(+:sum)
do i = 1, n
  sum = sum + x(i)
end do
!$acc end kernels
```

Fortran do concurrent

```
do concurrent (i = 1: n)
  sum = sum + x(i)
end do
```

NVIDIAコンパイラでは、リダクションを勝手に判定してくれるが、Fortranの様上はまだリダクションはサポートされていない。

そのため上記コードが将来的に動くかどうかはわからない。

OpenACC, OpenMP for GPU のリダクションの実装はOpenMP for CPUと大差ない

ループ並列化の例：atomic演算

OpenMP for CPU

```
!$omp parallel do private(j)
do i = 1, size(x)
  j = x(i)
  !$omp atomic
  y(j) = y(j) + 1
  !$omp end atomic
end do
!$omp end parallel do
```

OpenMP for GPU

```
!$omp target
!$omp loop
do i = 1, size(x)
  j = x(i)
  !$omp atomic
  y(j) = y(j) + 1
  !$omp end atomic
end do
!$omp end target
```

OpenACC

```
!$acc kernels
!$acc loop independent
do i = 1, size(x)
  j = x(i)
  !$acc atomic
  y(j) = y(j) + 1
  !$acc end atomic
end do
!$acc end kernels
```

Fortran do concurrent

N/A

Atomic計算は、OpenMPでもOpenACCでも書き方はほぼ同じ。Fortran do concurrentではサポートされていない。

ループ並列化の例：配列代入

OpenMP for CPU

```
!$omp workshare  
y(:) = a * x(:) + y(:)  
!$omp end workshare
```

OpenMP for GPU

N/A?

OpenACC

```
!$acc kernels  
y(:) = a * x(:) + y(:)  
!$acc end kernels
```

Fortran do concurrent

N/A

使えると便利な配列代入だが、今のところOpenACCでしか使えない。OpenMP for GPUでは、コンパイルは通るものの並列化されない。

ループ並列化の例：関数呼び出しを含むループ

OpenMP for CPU

```
real(KIND=8) function madd(a,x,y)
  real(KIND=8),intent(in) :: a,x,y
  !$omp declare simd
  madd = a * x + y
end function madd
```

```
subroutine daxpy(x, y, a, n)
  real(KIND=8),dimension(:),intent(out) :: y
  real(KIND=8),dimension(:),intent(in) :: x
  real(KIND=8),intent(in) :: a
  integer,intent(in) :: n
  integer :: i

  !$omp parallel do simd
  do i = 1, n
    y(i) = madd(a,x(i),y(i))
  end do
  !$omp end parallel do
end subroutine daxpy
```

あまり認知されていないが、OpenMP for CPUでもSIMDループ内で関数呼び出しをする際にはdeclare simdが必要。

OpenMP for GPU

```
! !$omp declare target
real(KIND=8) function madd(a,x,y)
  real(KIND=8),intent(in) :: a,x,y

  madd = a * x + y
end function madd
```

```
subroutine daxpy(x, y, a, n)
  real(KIND=8),dimension(:),intent(out) :: y
  real(KIND=8),dimension(:),intent(in) :: x
  real(KIND=8),intent(in) :: a
  integer,intent(in) :: n
  integer :: i

  !$omp target
  !$omp loop
  do i = 1, n
    y(i) = madd(a,x(i),y(i))
  end do
  !$omp end target
end subroutine daxpy
```

Declare target指示文が必要なはずだが、NVIDIAコンパイラ(22.5)では付けなくても動くし、付けるとコンパイルが途中で止まる。原因不明。

OpenACC

```
real(KIND=8) function madd(a,x,y)
  real(KIND=8),intent(in) :: a,x,y
  !$acc routine seq
  madd = a * x + y
end function madd
```

```
subroutine daxpy(x, y, a, n)
  real(KIND=8),dimension(:),intent(out) :: y
  real(KIND=8),dimension(:),intent(in) :: x
  real(KIND=8),intent(in) :: a
  integer,intent(in) :: n
  integer :: i
  !$acc routine(madd)

  !$acc kernels
  !$acc loop independent
  do i = 1, n
    y(i) = madd(a,x(i),y(i))
  end do
  !$acc end kernels
end subroutine daxpy
```

呼び出される関数側と、呼び出し元の関数側にroutine指示文を書く必要がある。

ライブラリの呼び方

- 最も手っ取り早いのは、Unified memoryの使用の有無に関わらず、data指示文を使う方法

- OpenACC の指示文を使う場合

```
!$acc data copyin(A,B) copy(C)
call cublasDgemm('n','n', n, n, n, alpha, A, n, B, n, beta, C, n)
!$end data
```

- OpenMP のtarget data指示文でも可
- Do concurrentでも、コンパイルオプションに-accをつけて、上記のように書くのが手っ取り早い

MPIとの繋ぎ

- Unified Memoryを使う場合
 - コード上で変更することは特になし
 - GPUのメモリからCPUのメモリ（MPIで送受信するバッファ）への暗黙のコピーが発生するため、性能上不利になり得る
- Unified Memoryを使わない場合
 - CUDA-aware MPIを使うのが基本
 - MPI関数の引数に、GPU上のポインタを受け付けるMPI
 - OpenMPIからも提供されている
 - 指示文を使って、GPU上のポインタを直接MPI関数の引数として渡す
 - OpenACC: host_data指示文
 - OpenMP: use_device_ptr指示節

```
!$acc data copy(fn)
...
!$acc host_data use_device(fn)
call MPI_Isend(fn(1,1,nz) , nx*ny, MPI_DOUBLE, rank_up ,
call MPI_Irecv(fn(1,1,0) , nx*ny, MPI_DOUBLE, rank_down,

call MPI_Isend(fn(1,1,1) , nx*ny, MPI_DOUBLE, rank_down,
call MPI_Irecv(fn(1,1,nz+1), nx*ny, MPI_DOUBLE, rank_up ,
!$acc end host_data
...
!$acc end data
```

```
!$omp target &
!$omp& data &
!$omp& use_device_ptr(f)
call MPI_Irecv(f(1) , nx*ny, MPI_DOUBLE,
call MPI_Irecv(f(nx*ny*(nz+1)+1), nx*ny, MPI_DOUBLE,
call MPI_Isend(f(nx*ny*nz+1) , nx*ny, MPI_DOUBLE,
call MPI_Isend(f(nx*ny+1) , nx*ny, MPI_DOUBLE,
!$omp end target data
```

MPIランクとGPU割り当て

- OpenACC関数を呼ぶためヘッダーを追加

```
C: #include <openacc.h>
```

```
Fortran: use openacc
```

- ノード上のGPU数の取得(`acc_get_num_devices`)

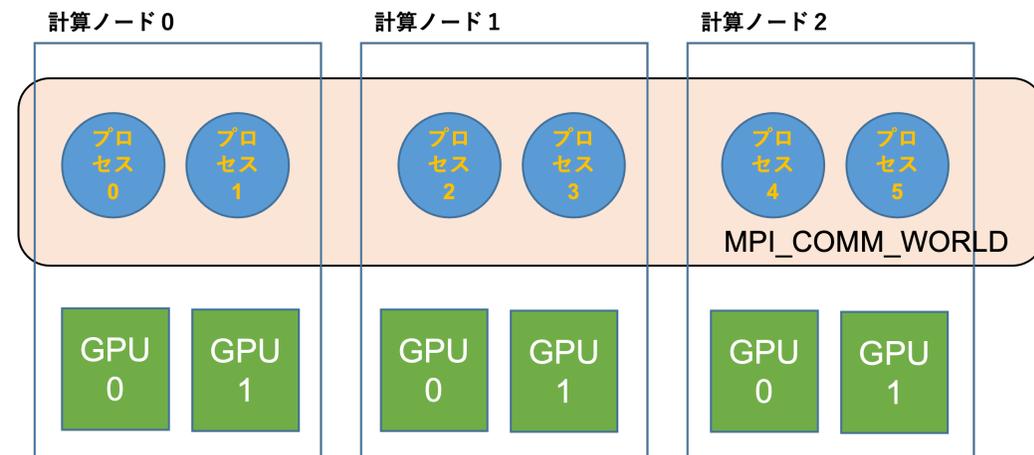
```
const int ngpus = acc_get_num_devices(acc_device_nvidia);
```

`acc_device_nvidia`を指定することで、NVIDIA GPUを数える。

- あるプロセスに特定のGPUを割り当て(`acc_set_device_num`)

```
int rank = 0;  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
acc_set_device_num(rank % ngpus, acc_device_nvidia);
```

`device_num` を `rank % ngpus` とすることで、ノード内のプロセスは異なるGPUを利用することとなる。これを呼ばないと、0番を使いに行く。

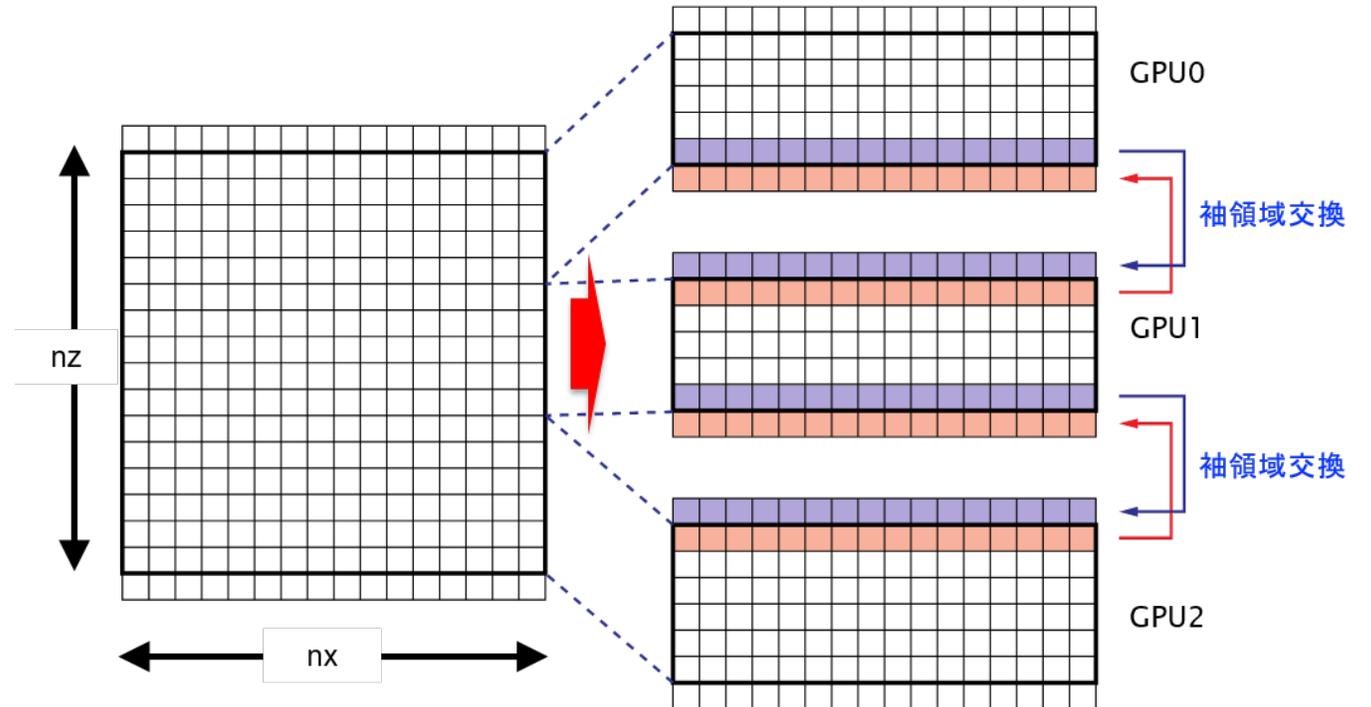


MPI通信に関する評価

- GPUへの移行対象のアプリケーションの多くはMPI+OpenMPで並列化されている
 - Unified memoryは通信性能にどの程度影響を与えるのか？
 - (細かい話として) GPUでの非同期のストリーム実行ができないが、性能への影響は？

- 拡散方程式カーネル

- ✓ メモリ律速なカーネル
- ✓ 7点テンシルと呼ばれ、よくベンチマークに用いられる
- ✓ Z軸一次元分割
- ✓ 袖領域通信と計算のオーバーラップ

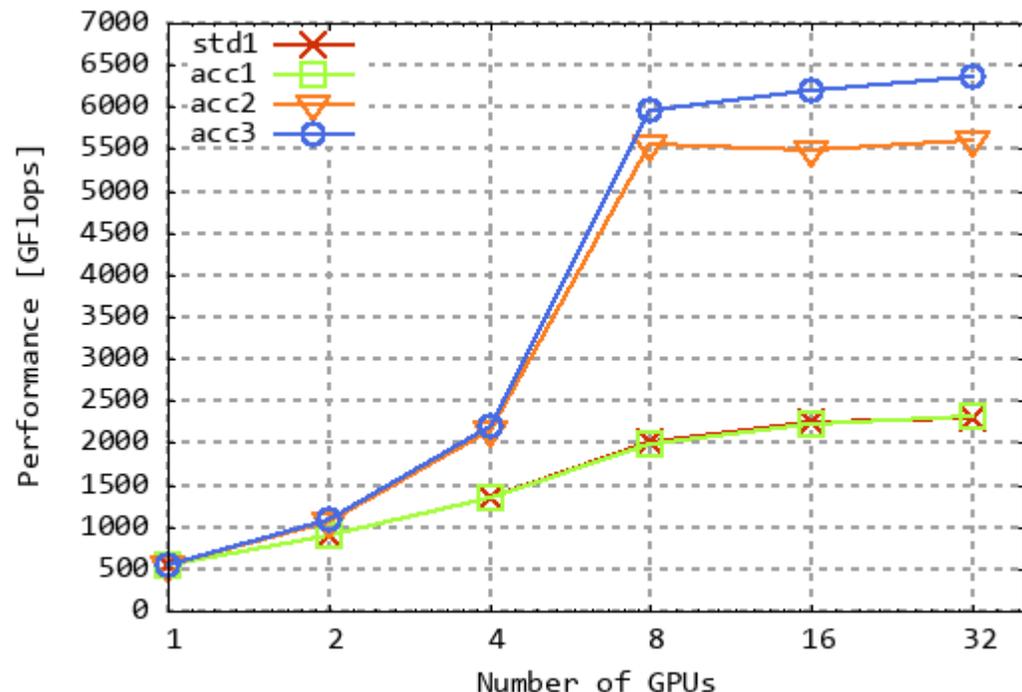


拡散方程式カーネル

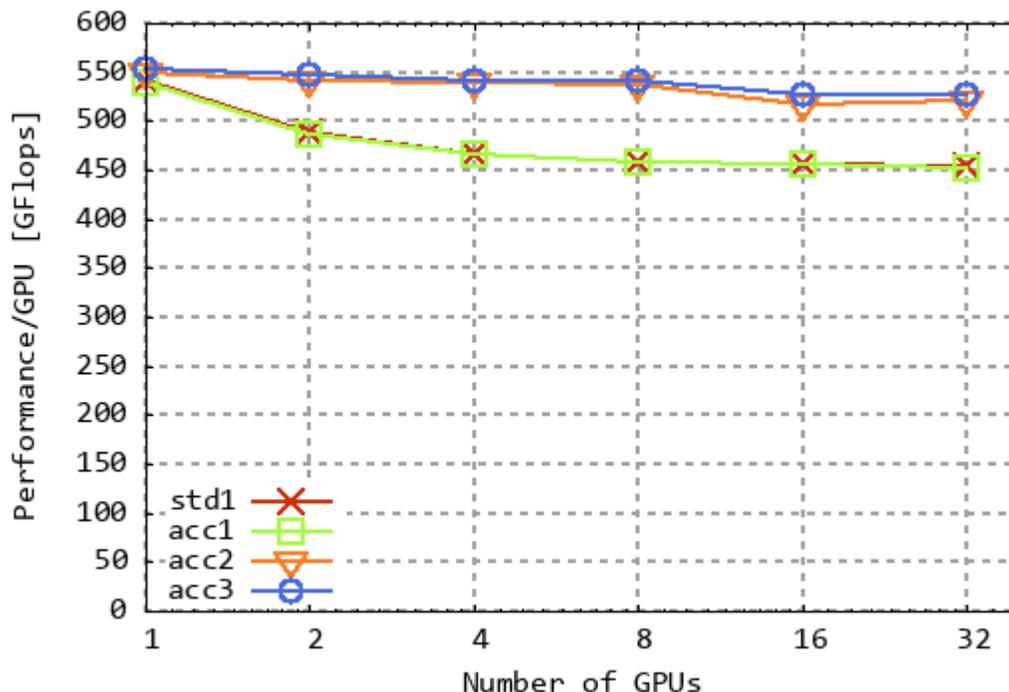
- Strong scalingでは特に、Unified memory使用時の性能低下が大きい
 - **do concurrent**を使う以上はどうしようもない

実装1	Unified Memory使用
実装2	Data指示文+GPU Direct RDMA使用
実装3	実装2に加えてasync節で袖領域計算の複数同時実行

Strong scaling



Weak scaling (GPUあたりの性能)



※問題サイズ 512³、倍精度

※WisteriaのAquariusノード群を使用。1ノードあたりA100 GPUを8枚搭載

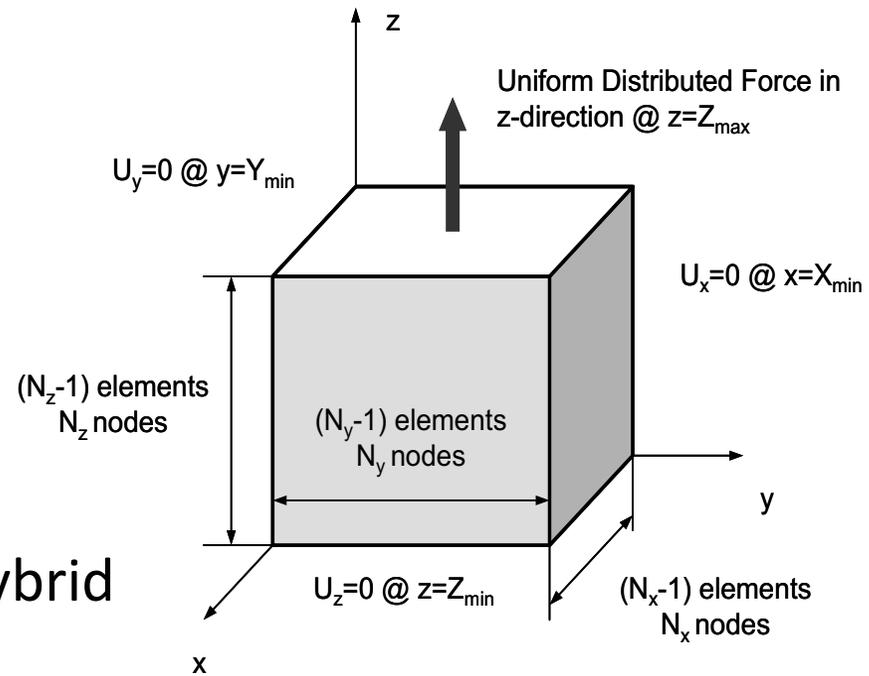
※A100 コンパイルコマンド: `nvfortran (version 22.5) (-stdpar=gpu -acc=gpu) -Minfo=acc,stdpar -gpu=cc80(,managed)`

※OpenMPI 4.1.4 を使用。

有限要素法コードのGPU移植実習

Target: GeoFEM/Cube

- Parallel FEM Code (& Benchmarks)
- 3D-Static-Elastic-Linear (Solid Mechanics)
- Performance of Parallel Preconditioned Iterative Solvers
 - 3D Tri-linear Hexa. Elements
 - SPD matrices: CG solver
 - Fortran90+MPI+OpenMP
 - Distributed Data Structure
 - Block Diagonal LU + CG
 - Block CRS Format
 - MPI, OpenMP, OpenMP/MPI Hybrid
 - only OpenMP case



GeoFEM ICCGソルバー

- 陰解法だが、隣接ノードとの通信は陽的に行う
 - 隣接ノード通信はisend, irecvにより実装されているが、通信隠蔽はアルゴリズム上困難
 - スカラ要素のMPI_Allreduceが必要
- カラーリング (CM-RCM法) により依存性は解消されており、並列化可能なループで構成される
- リダクションが必要となるため、**現在の do concurrent の正規仕様では並列化できない**
 - NVIDIA compilerは独自拡張でリダクションに対応しているので、並列化可能
 - 今後他の環境で動くかどうかはわからない

do iterPRE= 1, iterPREmax (=2)

Local Operation (Forward/Backward Substitution)

$$\text{calc. } M_{\Omega_1}^{-1}(r_{\Omega_1} - M_{\Omega_1} z_{\Omega_1}^{n-1} - M_{\Gamma_1} z_{\Gamma_1}^{n-1})$$

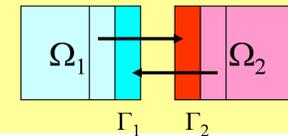
$$\text{calc. } M_{\Omega_2}^{-1}(r_{\Omega_2} - M_{\Omega_2} z_{\Omega_2}^{n-1} - M_{\Gamma_2} z_{\Gamma_2}^{n-1})$$



Global Nesting Correction: Iter's for Stable Convergence

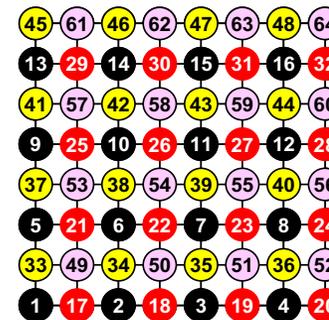
$$z_{\Omega_1}^n = z_{\Omega_1}^{n-1} + M_{\Omega_1}^{-1}(r_{\Omega_1} - M_{\Omega_1} z_{\Omega_1}^{n-1} - M_{\Gamma_1} z_{\Gamma_1}^{n-1})$$

$$z_{\Omega_2}^n = z_{\Omega_2}^{n-1} + M_{\Omega_2}^{-1}(r_{\Omega_2} - M_{\Omega_2} z_{\Omega_2}^{n-1} - M_{\Gamma_2} z_{\Gamma_2}^{n-1})$$



enddo

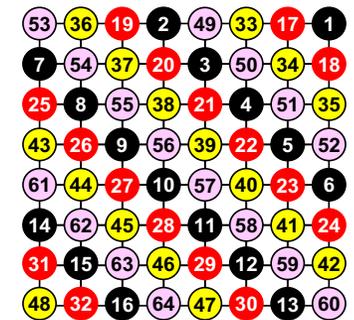
Ω_i : Internal Nodes ($i \leq N$)
 Γ_i : External Nodes ($i > N$)



MC (Color#=4)
Multicoloring



RCM
Reverse Cuthill-McKee



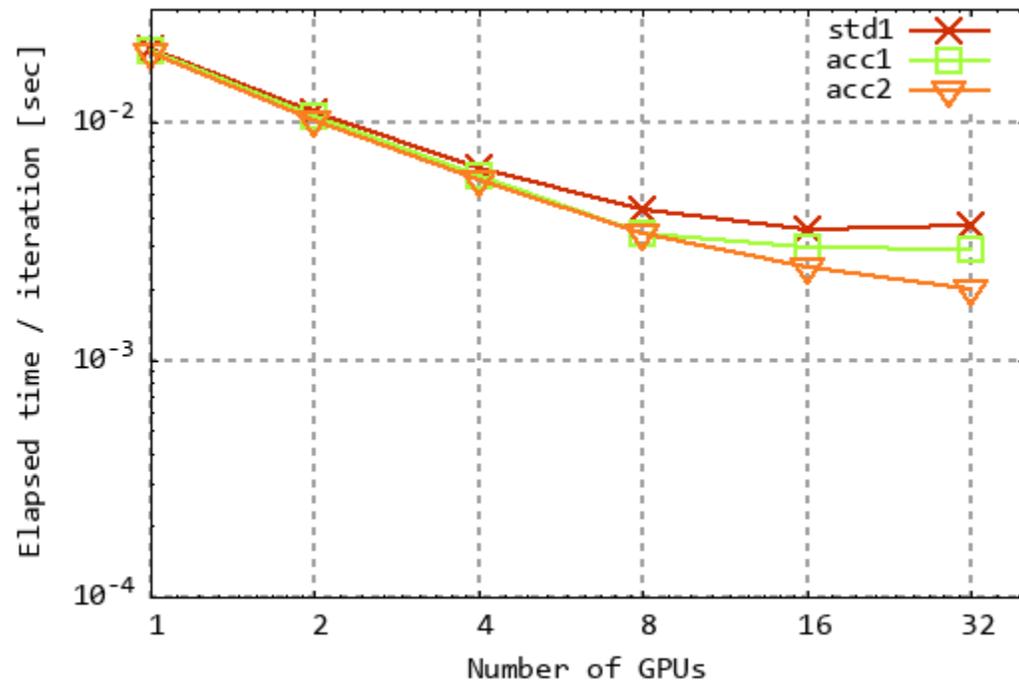
CM-RCM (Color#=4)
Cyclic MC + RCM

GeoFEM ICCGソルバー

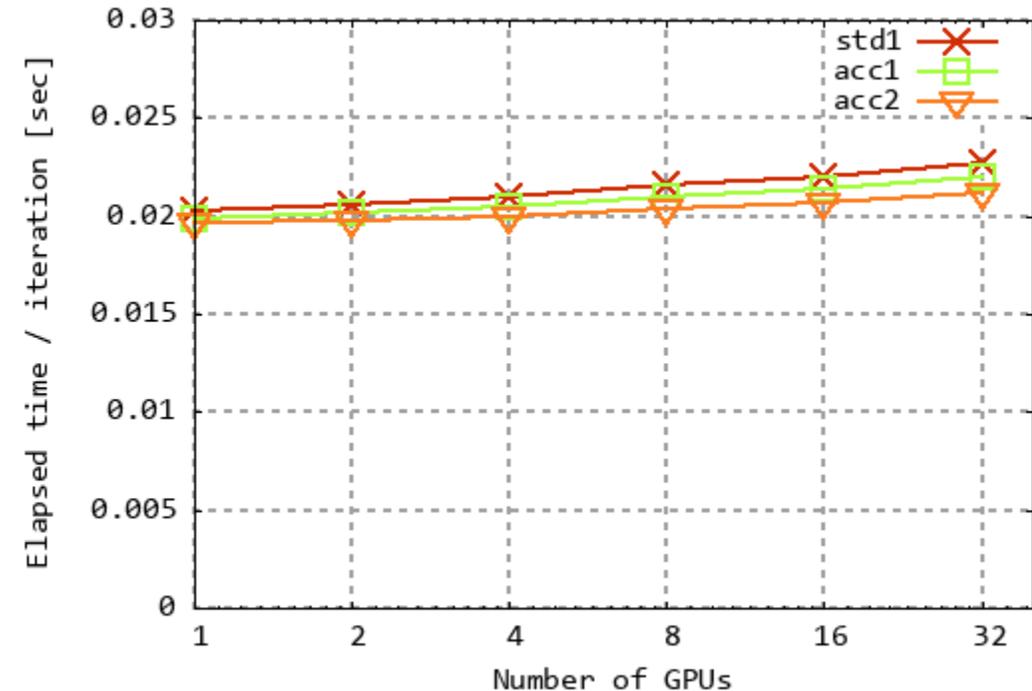
- ICCGソルバーの1ステップあたりの実行時間
- 収束回数には変化なし
- MPI通信の占める割合が小さいため、差は大きくない

実装1	Unified Memory使用
実装2	Data指示文+GPU Direct RDMA使用

Strong scaling



Weak scaling



※問題サイズ 128³、倍精度

※WisteriaのAquariusノード群を使用。1ノードあたりA100 GPUを8枚搭載

※A100 コンパイルコマンド: `nvfortran (version 22.5) (-stdpar=gpu -acc=gpu -mp=gpu) -Minfo=accel,mp -gpu=cc80(,managed)`

※OpenMPI 4.1.4 を使用。

サンプルコードのダウンロード

- 講習会で使うサンプルコードは以下にあります
 - /work/gt00/share/lecture_fortran2gpu.tar.gz

■ ダウンロード手順

1. `$ ssh tXXXXX@wisteria.cc.u-tokyo.ac.jp`
 - wisteriaへのssh。tXXXXXはアカウント名
2. `$ cd /work/gt00/tXXXXX`
3. `$ cp /work/gt00/share/lecture_fortran2gpu.tar.gz .`
4. `$ tar zxvf lecture_fortran2gpu.tar.gz`
5. `$ cd lecture_fortran2gpu/`

実習 1

- 有限要素法プログラムのCGソルバのGPU移行
- サンプルコード： [OpenMP_for_CPU/GeoFEM/01_naive](#)
- ステップ
 1. CPUで実行して、参照となる結果を得ましょう
 2. Makefileを書き換えましょう
 3. まずはUnified Memoryを利用し、ループの並列化をしましょう
 4. 指示文を使って、データ転送を実装しましょう

ステップ1 : CPUの結果

- 実行バイナリはOpenMP_for_CPU/GeoFEM/runに生成されます
- 実行パラメータはmesh.inpで調整できます

```
$ cat mesh.inp
128 128 128
 1  1  1
 1  1
2000
-8
```

プロセスあたりの
問題サイズ (NX*NY*NZ)

MPI プロセス数
例えば 1 2 2 にすると、
128 * 256 * 256 の問題
を4プロセスで解くための
問題になる

講習会コードでは未使用。
今回は気にしなくて良い

並列化の際のカラーリ
ング手法と色数。
今回は気にしなくて良
い。

```
128      128      128
 1        1        1
 1        1
-8
### NORMAL
### CM-RCM
color number:      8
color number:      8
5.239190E-02  coloring of elements
1.402784E+01  matrix assembling
100  7.200837E-02
200  7.698423E-04
300  2.321115E-05
400  3.515262E-07
457  9.940129E-09
elapsed  457  1.783206E+02  1.783206E+02  1.783206E+02
iter     457  3.901982E-01  3.901982E-01  3.901982E-01
2097152  -3.810000E+01  -3.810000E+01  1.270000E+02
```

収束履歴(多少ブレる)

収束後のある要素の値。
128³の時はこの値になる。

CG法全体の時間(elapsed),
反復あたりの時間(iter)

ステップ2 : Makefileの書き換え

- 書き換えの前に、workディレクトリを作ります
 - `$ cd OpenMP_for_CPU/GeoFEM/`
 - `$ cp -r 01_naive 00_work`
 - `$ cd 00_work`

```
F90      = mpifort
F90OPT   = -Mpreprocess -fast -mp
LOPT     = -Mpreprocess -fast -mp
TARGET   = ../run/01_naive
```



OpenACCの場合

```
F90      = mpifort
F90OPT   = -Mpreprocess -fast -acc -gpu=managed -Minfo=acc
LOPT     = -Mpreprocess -fast -acc -gpu=managed -Minfo=acc
TARGET   = ../run/00_work
```

- acc : OpenACCのフラグ。-acc=multicoreとするとCPUで実行できる。
- mp=gpu : -accの代わりに使うと、OpenMP for GPUが有効化される。
- stdpar=gpu : -accの代わりに使うと、Do concurrentがGPUで実行される。
- gpu=managed : Unified Memoryモードで実行するときのオプション。データ指示文を使うときは外す。
- Minfo=acc/mp/stdpar : コンパイル時に並列化に関する情報を出力するオプション。

ステップ3：ICCGソルバの並列化

- まずは solver_CG_3_SMP_novec.f を書き換えます
 - OpenMP 指示文のついたループを、OpenACCやOpenMP for GPUなどに書き換えてください
 - この時、結果に変動がないかどうかを実行して確認しながら進めると良いです
 - コンパイルのためには以下の環境設定が必要です
 - \$ module load nvidia nvmpi

```
!$omp parallel do private(in)
  do i= 1, NP
    in= 0toN(i)
    WS(3*in-2)= B(3*i-2)
    WS(3*in-1)= B(3*i-1)
    WS(3*in )= B(3*i )
  enddo
!$omp end parallel do
```



```
!$acc kernels
!$acc loop independent
  do i= 1, NP
    in= 0toN(i)
    WS(3*in-2)= B(3*i-2)
    WS(3*in-1)= B(3*i-1)
    WS(3*in )= B(3*i )
  enddo
!$acc end kernels
```

ステップ4：指示文によるデータ移動

- Makefileを編集し、managedを削除します

```
F90      = mpifort
F90OPT   = -Mpreprocess -fast -acc -gpu=managed -Minfo=acc
LOPT     = -Mpreprocess -fast -acc -gpu=managed -Minfo=acc
TARGET   = ../run/00_work
```

- このコードはgo to文が含まれるため、通常のでata指示文が使えません。代わりにenter data, exit data指示文を使います
 - !\$acc enter data copyin(a) create(b)
 - !\$acc exit data delete(a) copyout(b)

チャレンジ：行列生成部の並列化

- `mat_ass_main.f` を書き換えます
- 単純に並列化するだけならあまり難しくありませんが、かなり複雑な構造をしているため、性能を出すためには色々工夫する余地があります
 - 実装をいくつか用意してあるので、興味があればみてください

実装まとめ

実装言語	データ転送方式	ループ分割	書き込み競合	実装名	実装言語	データ転送方式	ループ分割	書き込み競合	実装名
Do concurrent (stdpar)	Unified Memory	fusion	coloring	Std_U_f_c	OpenACC	Unified Memory	fusion	coloring	ACC_U_f_c
			atomic	N/A				atomic	ADD_U_f_a
		split	coloring	Std_U_s_c			split	coloring	ACC_U_s_c
			atomic	N/A				atomic	E
	Directive	fusion	coloring	N/A		Directive	fusion	coloring	ACC_D_f_c
			atomic	N/A				atomic	ACC_D_f_a
		split	coloring	N/A			split	coloring	ACC_D_s_c
			atomic	N/A				atomic	E
OpenMP for GPU	Unified Memory	fusion	coloring	Omp_U_f_c	OpenACC + CUDA	Unified Memory	fusion	coloring	-
			atomic	Omp_U_f_a				atomic	CUDA_U_f_a
		split	coloring	Omp_U_s_c			split	coloring	-
			atomic	Omp_U_s_a				atomic	-
	Directive	fusion	coloring	-		Directive	fusion	coloring	-
			atomic	-				atomic	-
		split	coloring	-			split	coloring	-
			atomic	-				atomic	-

N/A: 言語の制約で実装できない

E: 実装できるはずだが結果がおかしい

-: 未実装