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：山崎一哉
kyamazaki(at)cc.u-tokyo.ac.jp
（内容に関するご質問はSlackかこちらまで）

講習会スケジュール

- 開催日時
  - ✓ 5月19日(月) 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
  - ✓ Miyabi-Gにおける、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-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
  - ✓ 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

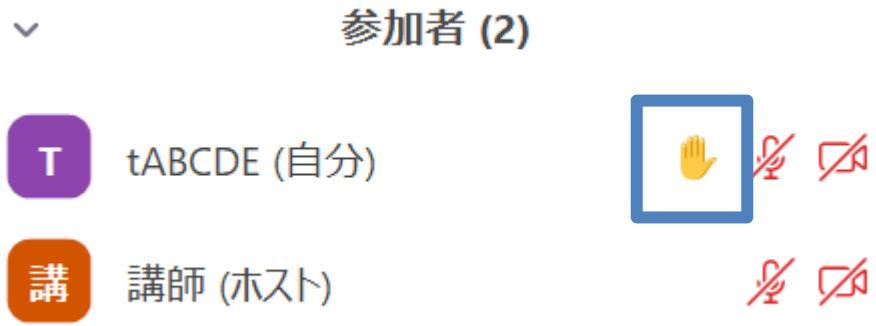
1. Zoomメニュー中の「リアクション」をクリック

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: 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

東大情報基盤センターで現在提供しているスパコン

Wisteria
BDEC-01

運用者：東京大学
運用期間（予定）：2021～2027年

Odyssey (**CPU**)

A64FX 7680基
25.9 PF, 7.8 PB/s

Aquarius (GPU搭載)

A100 360基
7.2 PF, 0.6 PB/s

東大情報基盤センターで現在提供しているスパコン

Wisteria
BDEC-01

運用者：東京大学
運用期間（予定）：2021～2027年

Odyssey (**CPU**)

A64FX 7680基
25.9 PF, 7.8 PB/s

Aquarius (GPU搭載)

A100 360基
7.2 PF, 0.6 PB/s

---

miyabi

運用者：JCAHPC (筑波大・東大)
2025年1月　運用開始

Miyabi-C (CPU)

Intel SPR HBM　380基
1.3 PF, 0.6 PB/s

Miyabi-G (**GPU**搭載)

GH200 1120基
78.8 PF, 5.1 PB/s

東大情報基盤センターで現在提供しているスパコン

Wisteria
BDEC-01

Odyssey (**CPU**)

Aquarius (GPU搭載)

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

miyabi

運用者：JCAHPC (筑波大・東大)
2025年1月　運用開始

Miyabi-C (CPU)

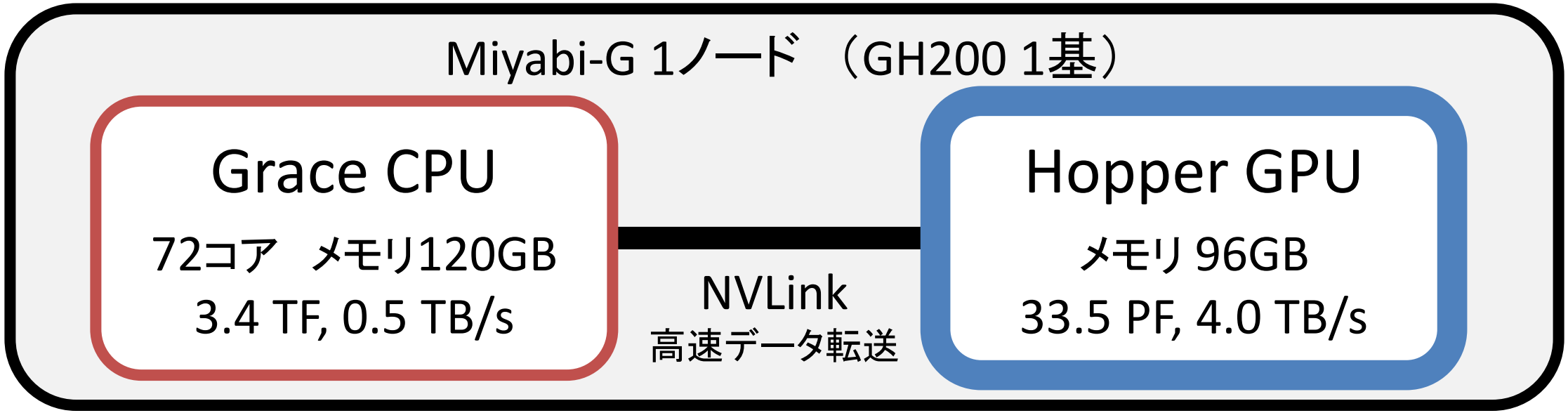
Intel SPR HBM　380基
1.3 PF, 0.6 PB/s

Miyabi-G (**GPU**搭載)

GH200 1120基
78.8 PF, 5.1 PB/s

Miyabi-Gの構成

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



Miyabi 利用上の注意(1)

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

Miyabi 利用上の注意(2)

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

$ module load <*module_name*>

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

$ module avail

使用可能なモジュール一覧を表示する。

$ module list

使用中のモジュールを表示する。

$ module purge

使用中のモジュールを全てunloadする。

コンパイラの種類と実行(Miyabi-G)

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

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

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

Miyabiでのプログラムの実行

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

  $ qsub ./〇〇.sh

- 投入されたジョブを確認する。（pjstatではないので注意）

  $ qstat

- 実行が終了すると、以下のファイルが生成される。

  〇〇.sh.o?????? （?????? はジョブID）

- 上記の標準出力ファイルの中身を確認する。

  $ cat 〇〇.sh.o??????

ジョブスクリプトサンプルの説明（Miyabi-G, MPIなし）

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

cd $PBS_O_WORKDIR

./run
```

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

ジョブ投入時のディレクトリに移動

モジュールnvidiaは自動load済。
要らない場合は、module purgeで一括unloadできる

2ノード、16プロセス、9スレッド/プロセスの場合

```
#!/bin/bash
#PBS -q lecture-g
#PBS -l select=2:mpiprocs=8:ompthreads=9
#PBS -l walltime=00:01:00
#PBS -W group_list=gt00
#PBS -j oe

cd $PBS_O_WORKDIR
module load nvidia

mpirun -n ${MPI_PROCS} ./run
```

select: ノード数
mpiprocs: 1ノードあたりのMPIプロセス数
ompthreads: 1プロセスあたりのスレッド数

- OMP_NUM_THREADSは自動設定
- 1ノード(Grace 1基)あたり72コア
- Grace CPUはNUMAドメインの区分がない

モジュールnvidia nv-hpcxは自動load済。
要らない場合は、一旦module purgeで一括unloadできる

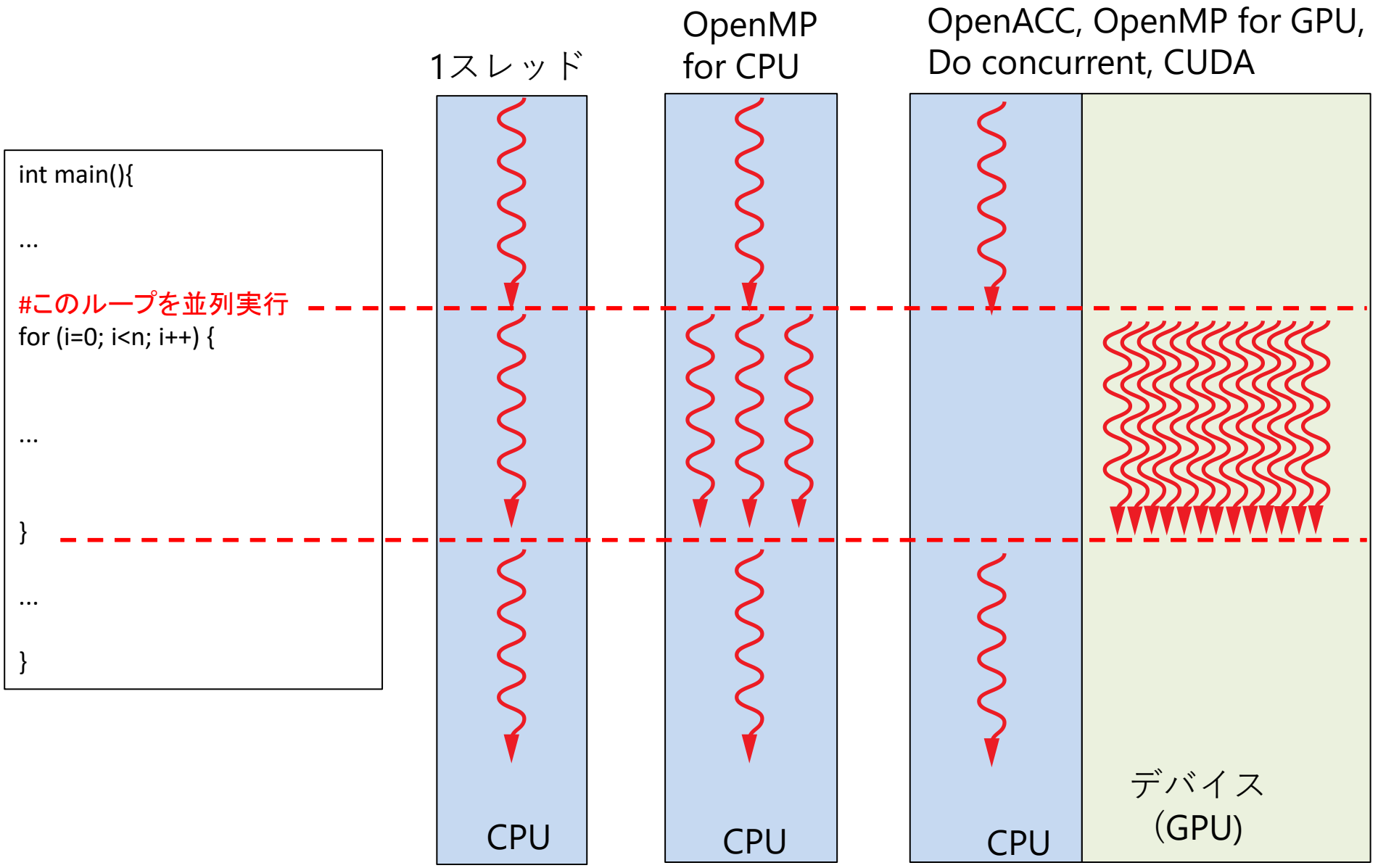
MPI+OPENMPからのGPUへの移行手法

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 , Intel)
  - MPI + CUDA Fortran (NVIDIA)

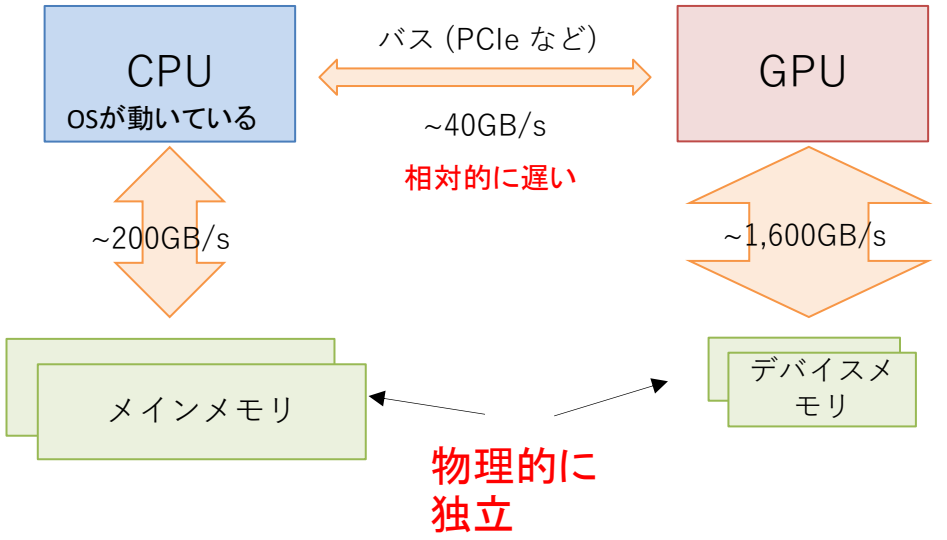
従来型のスパコンではほとんどの場合”X=OpenMP for CPU”だったが、
GPUスパコンでは”X”の選択肢が複数ある状態

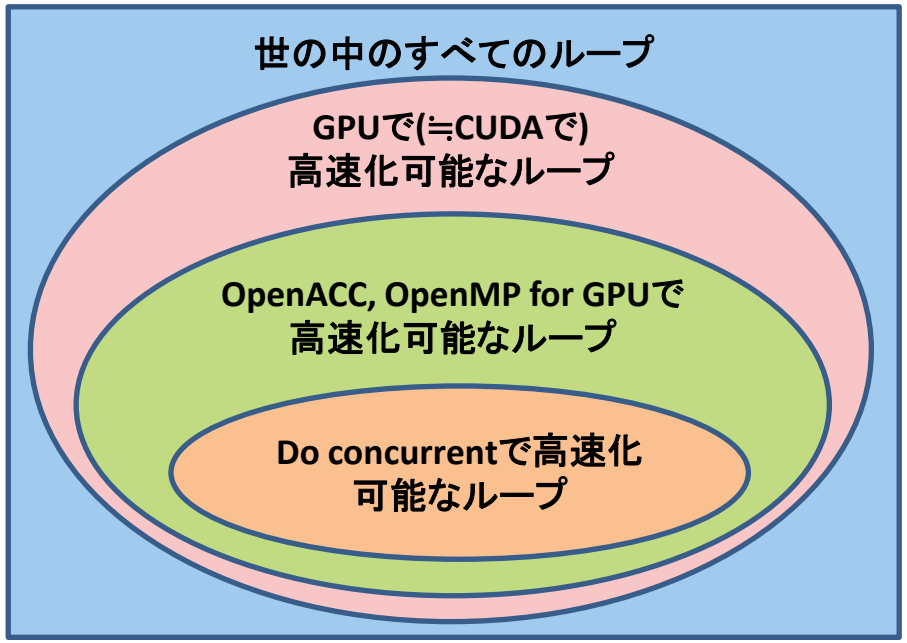
GPUプログラムの実行イメージ



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
     - Managed memory型
       （メモリアドレス空間ごと同一視、実体別々）
     - Unified Memory型
       （実体をどちらか片方のみに置く）
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 HPC SDK 25.3 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 Managed Memory Unified Memory ※1	変数同一視 Managed Memory Unified Memory ※1	変数同一視 Managed Memory Unified Memory ※1	Managed Memory Unified Memory	変数独立 Managed Memory Unified Memory
関数呼び出し	✓	※2	※2	N/A ※3	✓
リダクション	✓	✓	✓	✓	✓
atomic演算	✓	✓ ※4	✓ ※4	N/A	✓
スレッド数の調整	✓ ※5	✓ ※5	N/A	N/A	✓
Streamの非同期実行	✓ (キュー指定可)	✓	✓	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　Managed/Unified memoryはNVIDIA GPU, compilerの機能であり、OpenACC, OpenMPの仕様ではない
※2　動作するが、2025年5月現在のMiyabi-GではOpenACC版より100倍程度遅い
※3　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
※4　-gpu=mem:unified:nomanagedallocによってUnified Memoryを強制すると、2025年5月現在のMiyabi-GではSegmentation faultになる
※5　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

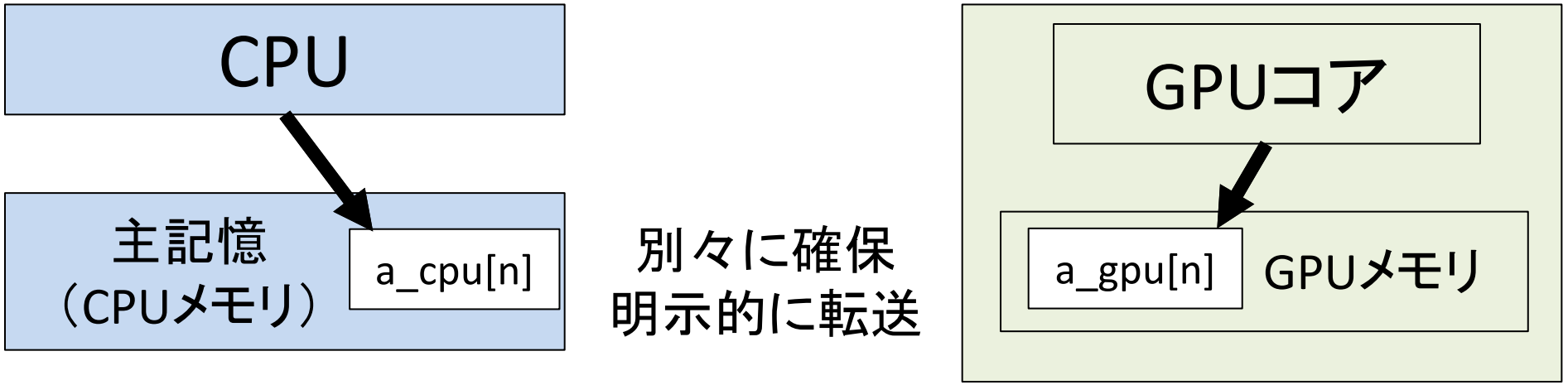
CPU-GPUのメモリ管理

- CPU-GPU変数独立型 (CUDA)
- 変数同一視・明示管理型 (OpenACC, OpenMP)
  - 手間がかかるが性能を高めやすい

- Managed Memory (do concurrent, OpenACC, OpenMP, CUDA)
- Unified Memory　(do concurrent, OpenACC, OpenMP, CUDA)
  - 圧倒的に楽だが、性能に制約がかかる場合がある

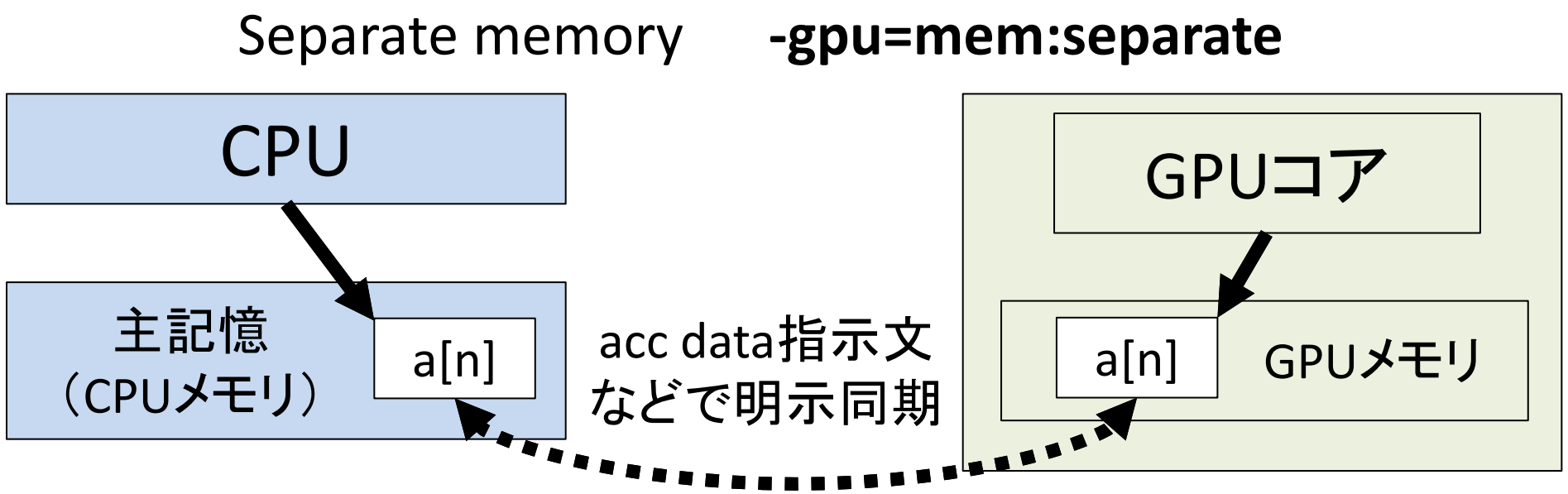
GPUメモリの管理 (1/4)

- 別々の変数として、明示的にメモリ管理(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
  - ✓ MPIでは、GPU上のデータを直接送受信可能
  - × 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



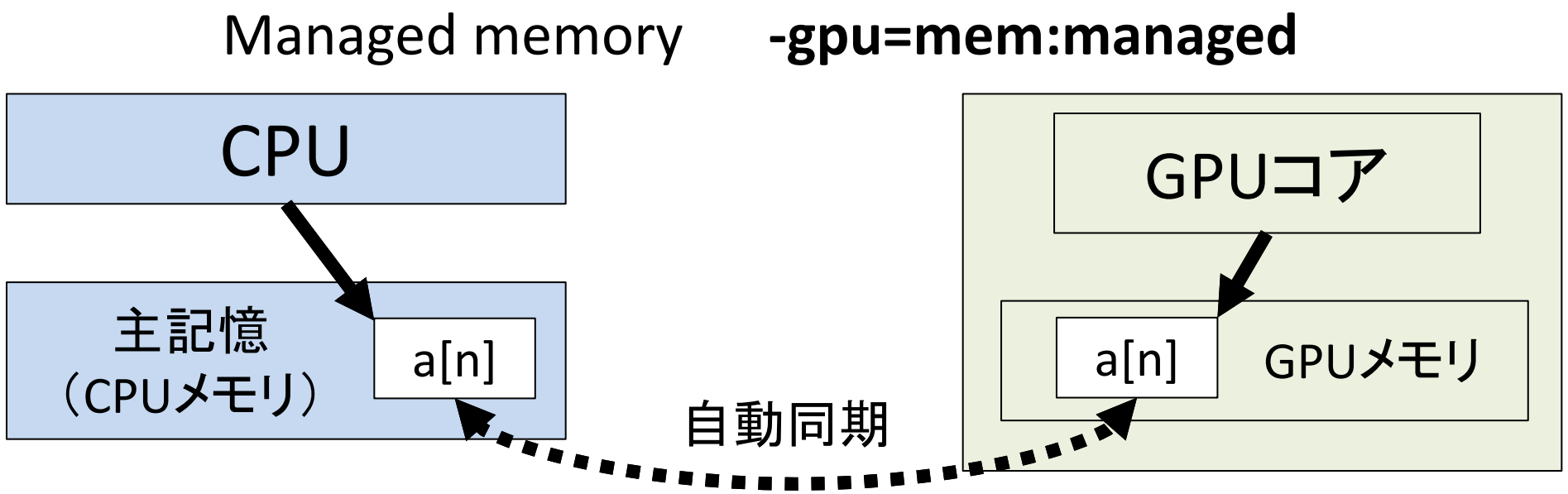
GPUメモリの管理 (2/4)

- 明示的にメモリ管理(Separate)
  - ➢ 全ての変数をCPUメモリに置くが、acc data指示文などで明示的にプログラマが指示した変数に関しては、GPUメモリにも領域を確保
  - ✓ CPU-GPU転送もdata指示文などで明示的に指示。関係ない演算の最中にバックグラウンドで転送することが可能
  - ✓ MPIでは、GPU上のデータを直接送受信可能
  - × 多数の配列を使う場合は、data指示文を整備するのが面倒。間違えるとバグになる



GPUメモリの管理 (3/4)

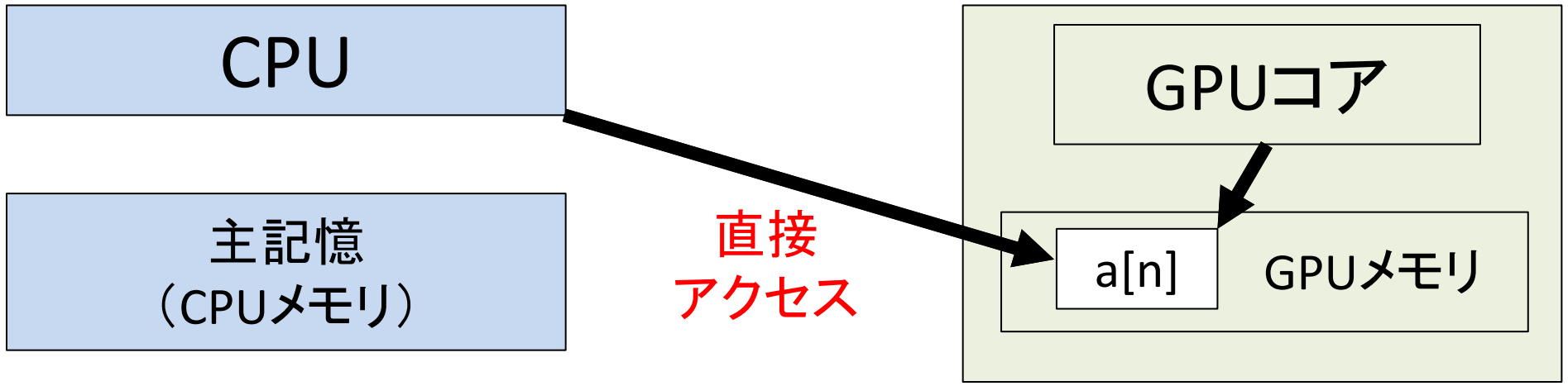
- Managed Memory (従来はUnified Memoryとも呼ばれた)
  - ➢ 全ての変数がCPUに置かれるが、GPUから触る変数に関しては、自動的にGPUメモリ確保・CPU-GPU同期が行われる
  - ✓ データの置き場や転送に関して一切指示しなくてよい（acc data指示文がある場合、コンパイラは参考にするが、必ずしも拘束されない模様）
  - ✓ ポインタを含む派生型などもそのまま使える
  - × 計算中に自動転送が走ると遅くなる。明示的にprefetchできるが、面倒
  - × GPU上のデータをMPI送受信するときはCPUメモリを経由してしまう（はず）



GPUメモリの管理 (4/4)

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

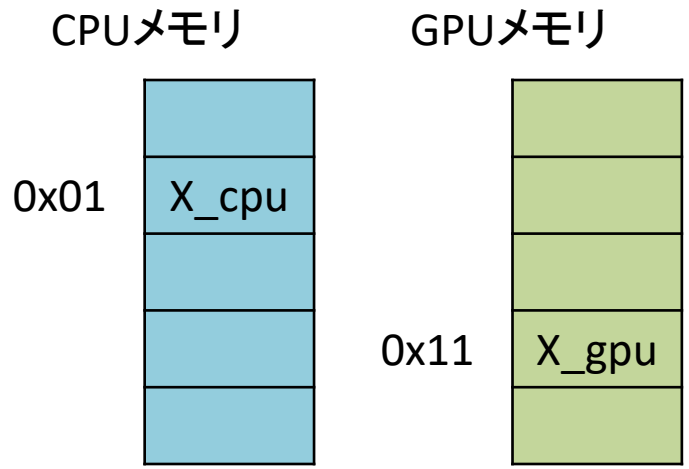
Unified memory　**-gpu=mem:unified:nomanagedalloc**



CPU-GPU間データ転送の書き方（明示系）

■ 変数独立型 (CUDA用)

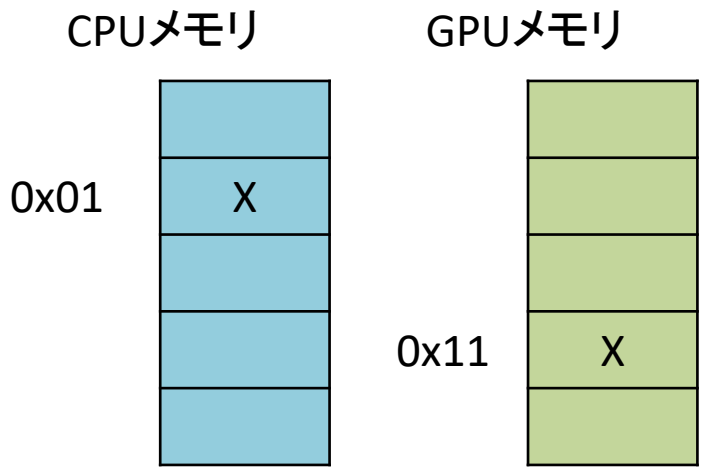
```
real(8), allocatable :: X_cpu(:)
real(8), allocatable, device :: X_gpu(:)
allocate(X_cpu(N))
allocate(X_gpu(N))
X_cpu(:) = 0
X_gpu(:) = X_cpu(:)
Call cuda_func<<<tb,th>>>(X_gpu)
X_cpu(:) = X_gpu(:)
```



■ 変数同一視型　(separate)

```
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

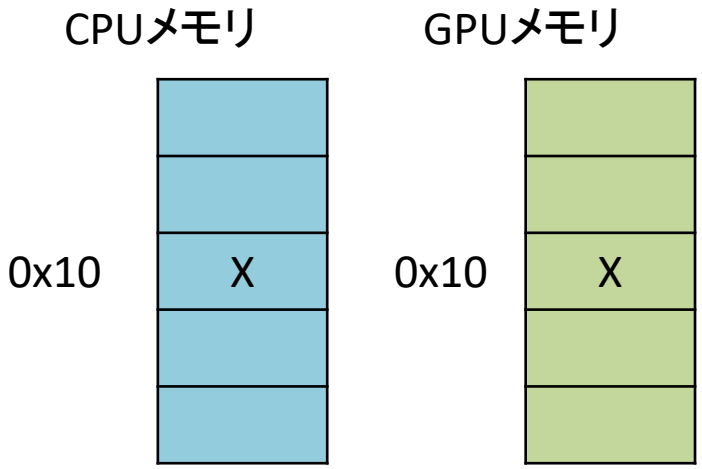


CPU-GPU間データ転送の書き方（自動管理系）

Managed Memory

```
real(8), allocatable :: X(:)
allocate(X(N))
X(:) = 0
call do_concurrent_func(X)
```

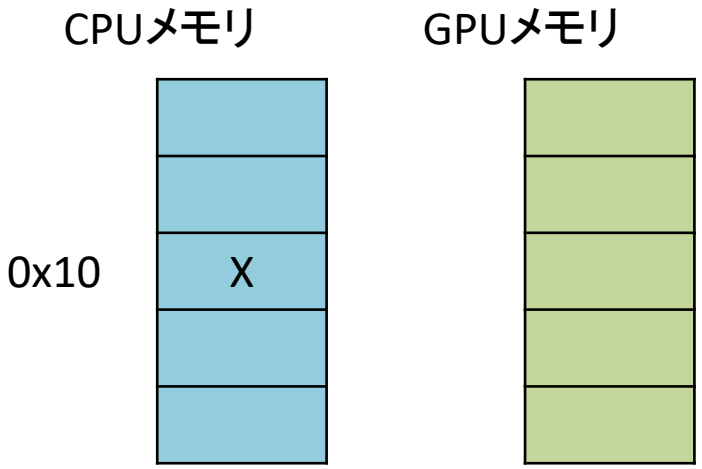
メモリアドレス空間を一致させることで、
X_cpu, X_gpuの区別を無くす
データ転送はページフォルトが起きた
際にバックグラウンドで行われる



Unified Memory

```
real(8), allocatable :: X(:)
allocate(X(N))
X(:) = 0
call do_concurrent_func(X)
```

CPU・GPU両方から直接アクセス
最初に書き込んだ側のメモリに置かれる。GPUからしばらくアクセスしているとGPUに引っ越す

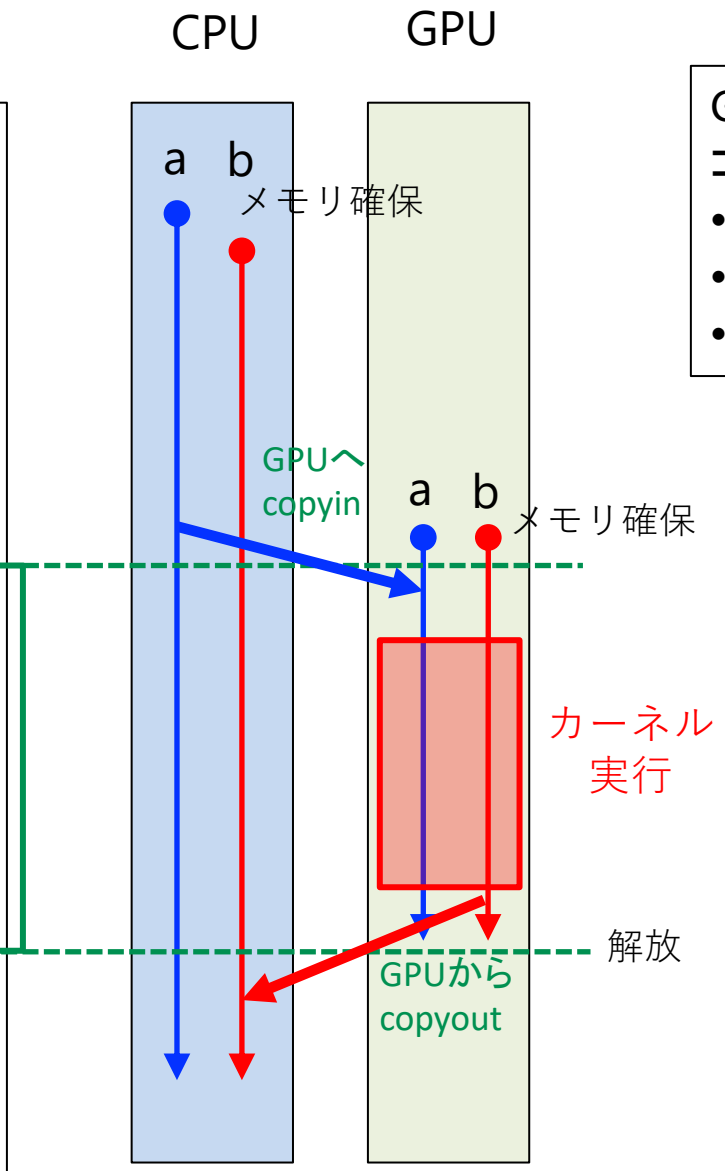


CPU-GPU間のデータ移動(Unified Memory以外)

OpenACC data指示文を使った場合

```
  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
  write(*,*) b(1)
  deallocate(a,b)
end program main
```



GPUのメモリ上に
コピーを作る。
- 変数独立型(CUDA)
- 変数同一視型
- Managed Memory

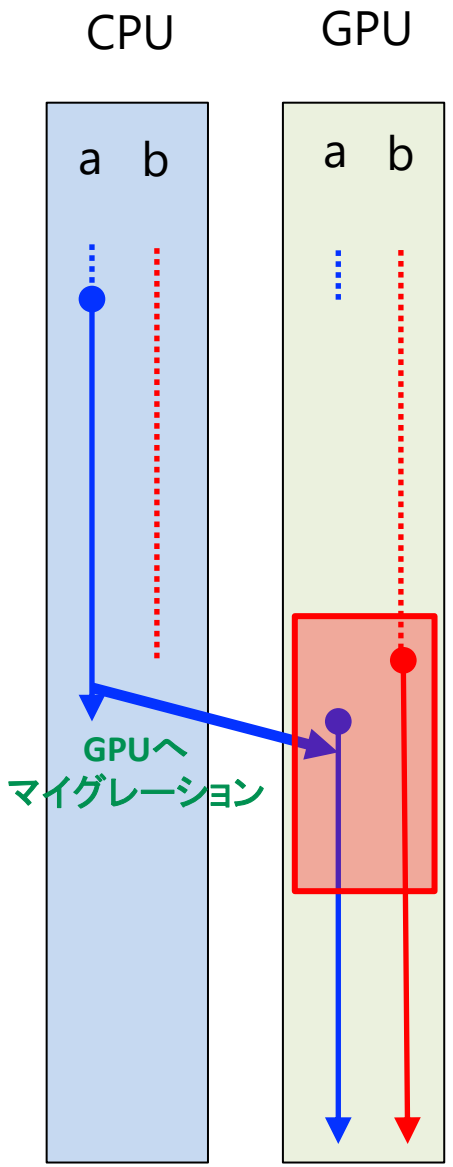
CPU-GPU間のデータ移動(Unified Memory)

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

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

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

write(*,*) b(1)
deallocate(a,b)
end program main
```



カーネル
実行

allocateした段階では、
どちらに置かれるか未定

First touchしたほうの
メモリに置かれる

CPU上のデータは、
GPUからしばらく触ると
GPUにマイグレーション

Managed Memory vs. Unified Memory

- 環境によってはUnified Memoryは使えない
- Miyabi-Gでは、Unified MemoryでのGPUメモリ上の変数はほぼ”最強”
  - ✓ GPUからは当然高速にアクセスできる
  - ✓ CPUからGPUメモリへのアクセスは、CPUからCPUメモリにアクセスするのとそれほど変わらない性能
  - ✓ GPU処理の合間に、何度もCPUから書き込む場合、Managed Memoryより明確に速いと思われる
- CPU上で初期化したい変数はどうする？
  - Managed Memory：初期化後に明示的にprefetchするか、GPU処理が始まってからの自動転送に任せる
  - Unified Memory：CPUで初期化する前に、ダミーデータをGPUから書き込むことで、置き場が最初からGPUになる

Unified Memoryでのfirst touch

- CPUで初期化したい変数も、その前にダミーデータをGPUから書き込むことで、置き場が最初からGPUになる
- #if defined(__NVCOMPILER_GPU_UNIFIED_MEM) && !defined(__NVCOMPILER_GPU_MANAGED_MEM)
  （ダミーデータ書き込みコード、複数行でもよい）
  #endif
  とすると、Unified Memory使用時のみダミーデータを書き込むことができる
  - __NVCOMPILER_GPU_UNIFIED_MEM：Unified Memory使用時にコンパイラが自動定義
  - __NVCOMPILER_GPU_MANAGED_MEM：Managed Memory使用時にコンパイラが自動定義
  - mem:unifiedとする(:nomanagedallocを付けない)と、両方とも定義されている可能性がある

“X”に何を選ぶべきか（現段階での私見）

- 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当面ない
  - 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ということが最大のメリット
  - 現状、NVIDIA機では、性能がOpenACCよりやや低い事例やバグが散見される
  - 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
    - 単一ストリームでの非同期実行、ループに対するスレッドマッピングの制限など
  - AMDのコンパイラはまだ発展途上。Fortran版は 動作しない or 非常に遅い ことが多々ある
- 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
  - 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
  - 処理を全部GPU化できれば、Managed Memoryとして別システムに移植しても高速
  - 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とかも名目上サポートしているが遅い模様）ので、その点はデメリットにならない
- Do concurrent
  - Managed/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

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よう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（特にコードが長い場合）
- 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
  - $!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では同一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も一応できるが、ほとんど使うケースはない

GPUの階層構造 (1/2)

- H100は132個のSM (Streaming Multiprocessor)を持つ

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

GPUの階層構造 (2/2)

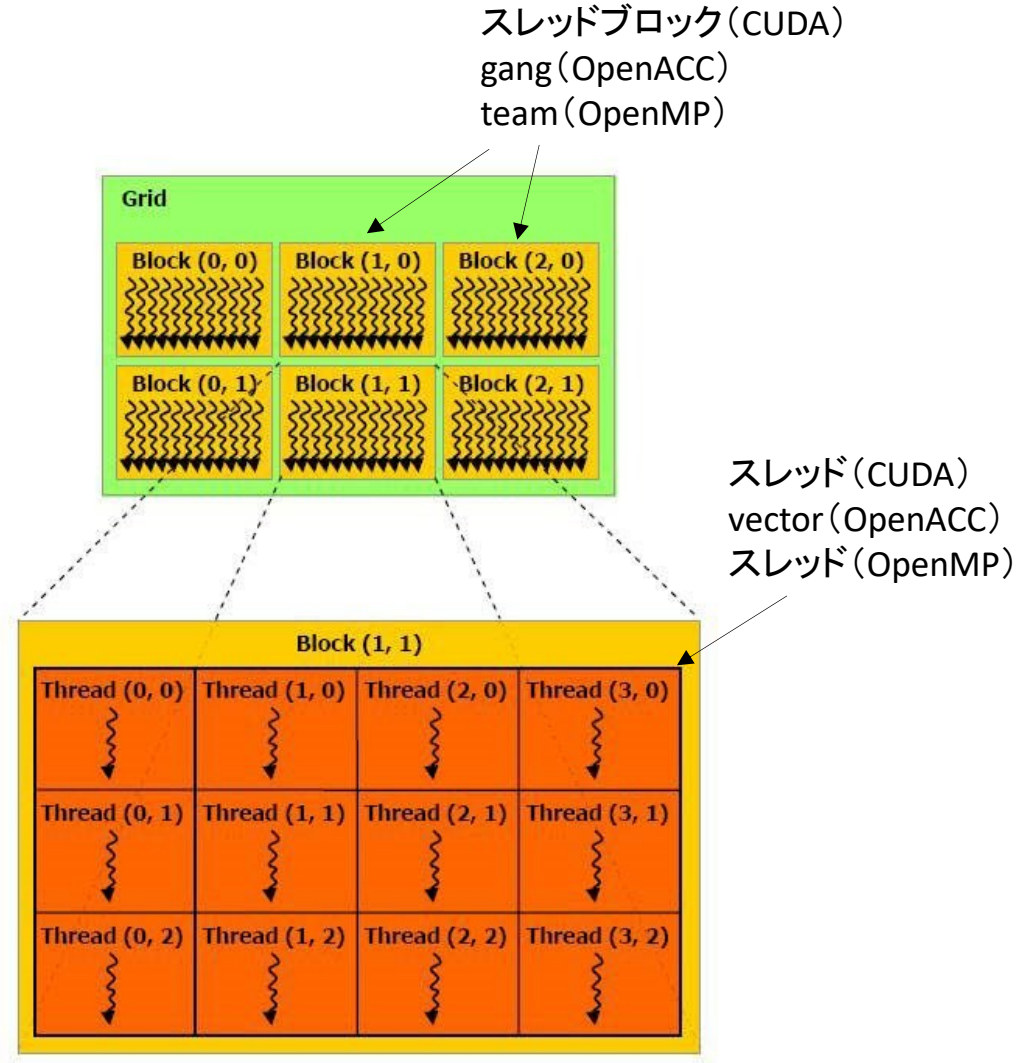
- 各SMに多数の演算コアが搭載されている
  - Tensor Coreは行列計算用で、CUDA専用
  - 低精度ほど、多量の演算を同時実行可能
  - L1キャッシュはSMごと。L2キャッシュは全体共有

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

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

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

サンプルコードのダウンロード

- 講習会で使うサンプルコードはGitHubにあります
  - https://github.com/kazuya-yamazaki/lecture_fortran2gpu

- ダウンロード手順
  1. $ ssh tXXXXX@miyabi-g.jcahpc.jp
     - Miyabi-Gへのssh。tXXXXXはアカウント名
  2. $ cd /work/gt00/tXXXXX
  3. $ git clone https://github.com/kazuya-yamazaki/lecture_fortran2gpu.git
  4. $ cd lecture_fortran2gpu/

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
      !$omp parallel do
      do i = 1, n
         y(i,j,k) = a * x(i,j,k) + y(i,j,k)
      end do
   end do
end do
!$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まり
が無いため、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、
しておくとよい。
特にOpenMP 4.xでは並列性を確保す
るためにcollapseが重要。

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
```

どれもOpenMP for CPUと大差ない

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) reduce(+:sum)
        sum = sum + x(i)
end do
```

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
```

OpenMP for GPUでは、
-gpu=mem:unified:nomanagedallocで
Unified Memory使用を強制すると、
2025年5月現在のMiyabi-Gでは
Segmentation faultになる

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 concurret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?

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。

OpenACC

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

Fortran do concurrent

N/A

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

```
real(KIND=8) function madd(a,x,y)
  real(KIND=8),intent(in) :: a,x,y
  !$omp declare target
  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 simd と同様の書き方
※2025年5月現在、Miyabi-Gでは
OpenACC版より100倍程度遅い

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

- Managed/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
```

複数プロセス使用時にGPUを割り振る

- CUDA及びMPI処理系の環境変数を使うことで、複数のプロセスが同じGPUを奪い合わないように、ランク別にGPUを割り付けられる。

run.sh

```
#!/bin/bash
#PBS -q lecture-g
#PBS -l select=4:mpiprocs=1
#PBS -l walltime=00:03:00
#PBS -W group_list=gt00
#PBS -j oe

cd $PBS_O_WORKDIR

module purge
module load nvidia nv-hpcx

mpirun -n ${MPI_PROC} ./wrapper.sh ./run
```

wrapper.sh（chmodで実行権を与える必要あり）

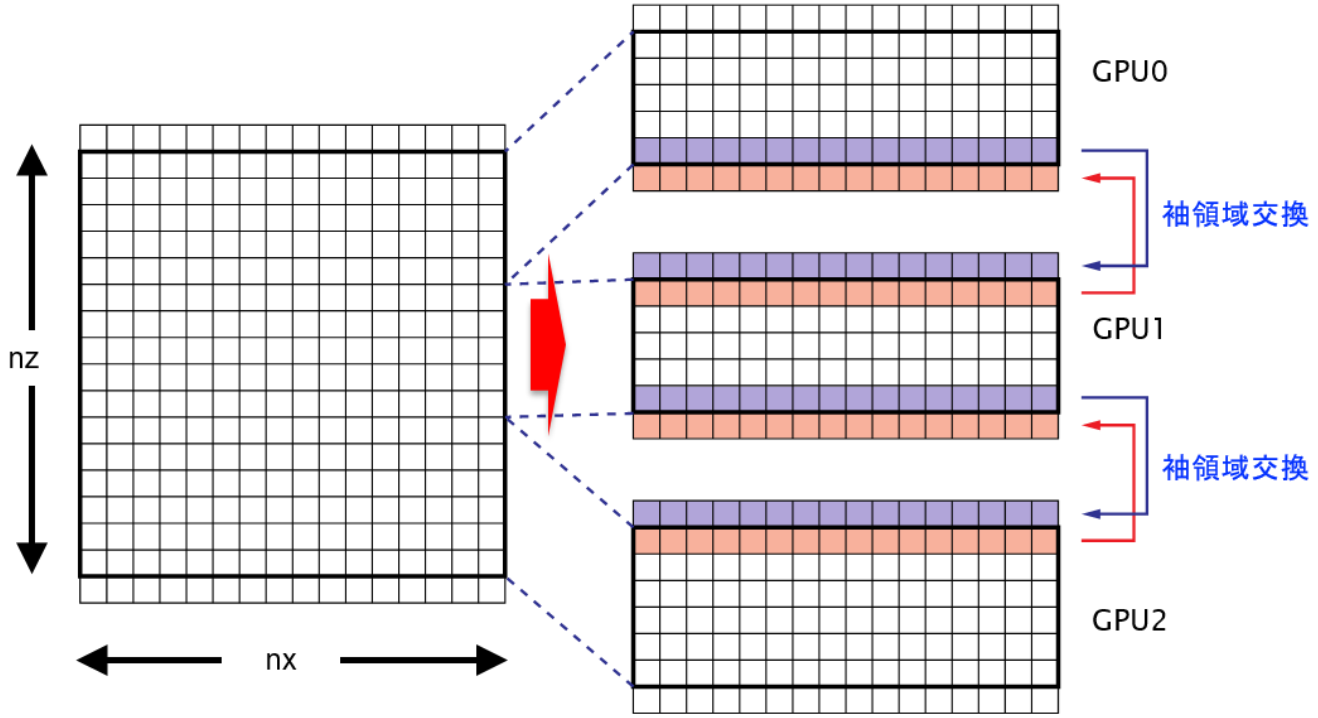
```
#!/bin/bash
GPU_UUID=(${CUDA_VISIBLE_DEVICES//,/ })
export CUDA_VISIBLE_DEVICES=${GPU_UUID[$OMPI_COMM_WORLD_LOCAL_RANK]}
$@
```

CUDA_VISIBLE_DEVICES: 使用可能なGPUの番号(0,1,2,…)またはUUIDをコンマ区切りで与える。Miyabi-Gでの初期値は、そのノード内で利用可能なGPUのUUIDが全て書かれている

OMPI_COMM_WORLD_LOCAL_RANK: MPIの処理系としてOpenMPIを使う場合、ノードローカルなランク番号がここに設定される。つまり上の場合、OMPI_COMM_WORLD_LOCAL_RANK番のGPUを使うことになる。

サンプルプログラム:拡散方程式カーネル

- 拡散方程式カーネル
  - ✓メモリ律速なカーネル
  - ✓7点ステンシルと呼ばれ、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



性能は、Unified Memory版＜Separate Memory版＜Separate Memory オーバーラップ版

- 差はあまり大きくない
- OpenACC、OpenMP target、Do concurrentのどれも似たような性能

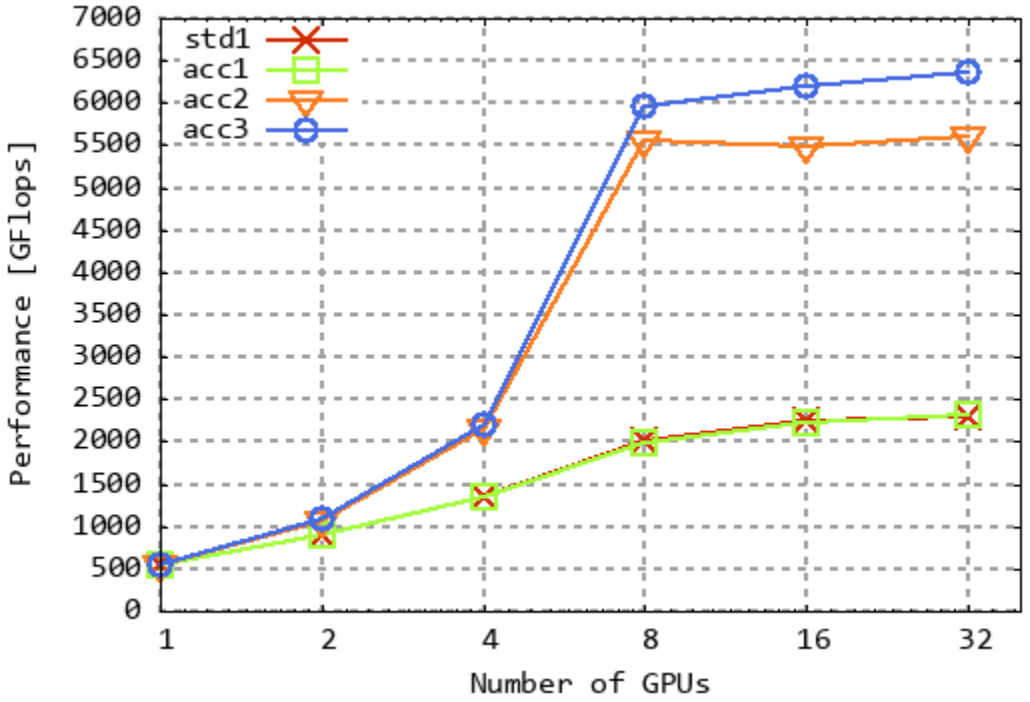
✓ Miyabi-GはCPU-GPU通信が速いため、Unified MemoryでのMPI通信の性能低下は軽微

CPU-GPU通信が遅いAquariusの場合

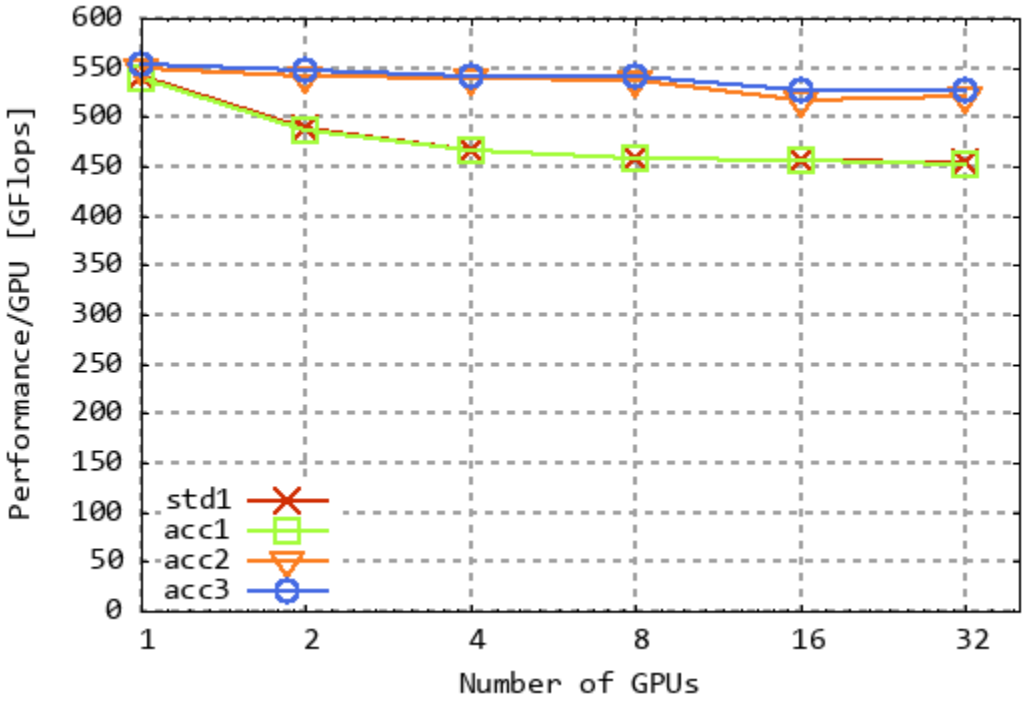
- Strong scalingでは特に、Unified memory使用時の性能低下が大きい
  - do concurrentを使う場合はどうしようもない

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）

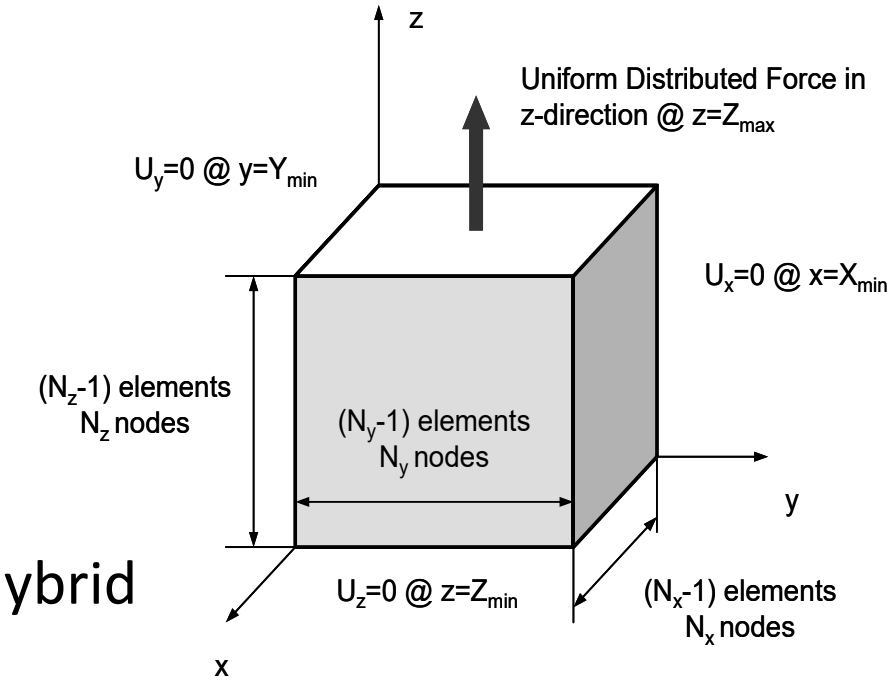


※問題サイズ 512^3、倍精度
※WisteriaのAquariusノード群を使用。1ノードあたりA100 GPUを8枚搭載
※A100 コンパイルコマンド: nvfortran (version 22.5) (-stdpar=gpu -acc=gpu) -Minfo=acc,stdpar -gpu=cc80(,managed)
※OpenMPI 4.1.4 を使用。

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



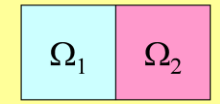
GeoFEM ICCGソルバー

- 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

do iterPRE= 1, iterPREmax (=2)

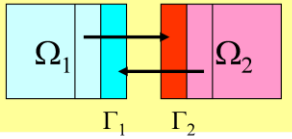
Local Operation (Forward/Backward Substitution)

$$calc.\quad M_{\Omega_1}^{-1}(r_{\Omega_1} - M_{\Omega_1} z_{\Omega_1}^{n-1} - M_{\Gamma_1} z_{\Gamma_1}^{n-1})$$
$$calc.\quad M_{\Omega_2}^{-1}(r_{\Omega_2} - M_{\Omega_2} z_{\Omega_2}^{n-1} - M_{\Gamma_2} z_{\Gamma_2}^{n-1})$$



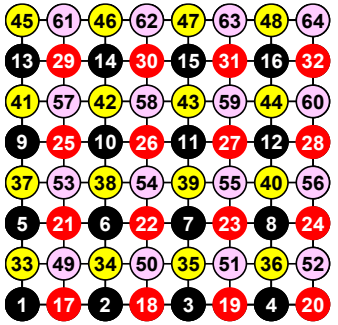
Global Nesting Correction: Iter's for Stable Convergence

$$z_{\Omega_1}^{n} = z_{\Omega_1}^{n-1} + M_{\Omega_1}^{-1}(r_{\Omega_1} - M_{\Omega_1} z_{\Omega_1}^{n-1} - M_{\Gamma_1} z_{\Gamma_1}^{n-1})$$
$$z_{\Omega_2}^{n} = z_{\Omega_2}^{n-1} + M_{\Omega_2}^{-1}(r_{\Omega_2} - M_{\Omega_2} z_{\Omega_2}^{n-1} - M_{\Gamma_2} z_{\Gamma_2}^{n-1})$$

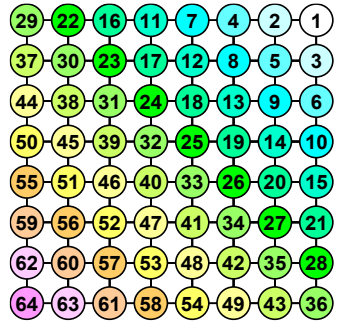


enddo

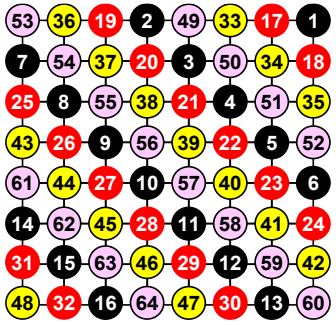
Ω_i: Internal Nodes (i≦N)
Γ_i: External Nodes (i>N)



MC (Color#=4)
Multicoloring



RCM
Reverse Cuthill-Mckee



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

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

 96  96  96
  1   1   1
  1   1
2000
 -8
```

全プロセス合計の
問題サイズ (NX*NY*NZ)

MPI プロセス数
例えば 1 2 2 にすると、
96 * 96 * 96 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

小さくすると、収束していなくても反復を打ち切る

```
96      96      96
      1       1       1
      1
     -8
### NORMAL
### CM-RCM
color number:       8
color number:       8
   2.267555E-02   coloring of elements
   6.146324E+00   matrix assembling
  100   1.985639E-02
  200   6.754801E-05
  300   3.330452E-07
  344   9.357312E-09
elapsed    344   1.953827E+02   1.953827E+02   1.953827E+02
iter       344   5.679730E-01   5.679730E-01   5.679730E-01
  884736  -2.850000E+01  -2.850000E+01   9.500000E+01
```

収束履歴（多少ブレる）

CG法全体の時間(elapsed),
反復あたりの時間(iter)

収束後のある要素の値。
96^3 の時はこの値になる。

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     = mpif90
F90OPT  = -Mpreprocess -fast -mp
LOPT    = -Mpreprocess -fast -mp
TARGET  = ../run/01_naive
```

OpenACCの場合

```
F90     = mpif90
F90OPT  = -Mpreprocess -fast -acc -gpu=cc90,mem:unified:nomanagedalloc -Minfo=acc
LOPT    = -Mpreprocess -fast -acc -gpu=cc90,mem:unified:nomanagedalloc -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。
-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（デフォルトでも設定される）
    - $ module load nvidia nv-hpcx

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

```
!$acc kernels
!$acc loop independent
      do i= 1, NP
        in= OtoN(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、unified:nomanagedallocをseparateに変更します

```
F90     = mpif90
F90OPT  = -Mpreprocess -fast -acc -gpu=cc90,mem:separate -Minfo=acc
LOPT    = -Mpreprocess -fast -acc -gpu=cc90,mem:separate -Minfo=acc
TARGET  = ../run/00_work
```

- このコードはgoto文が含まれるため、通常のdata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)
- enter dataやexit data指示文に与える変数名を間違えると、不可解な実行時エラーになることがあります。スペルミスに注意しましょう

ステップ5：MPIへの対応

- solver_SR_3.f の中身を書き換えて、MPIへ対応しましょう
  - OpenACC ならhost_data use_device指示文、OpenMPならtarget data use_device_ptr 指示文を使います
- MPIプロセスとGPU IDの対応は、ジョブスクリプトに記載済みです
  - wrapper.sh 内でOpenMPIとCUDAの環境変数を用いて設定しています
- MPIのプロセス数を変更してみましょう
  - mesh.inpの2行目と、run_gpu.shのgpu数及びMPIプロセス数を変更する必要があります

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	書き込み競合	実装名
do concurrent (stdpar)	Unified Memory	coloring	Std_U_c
		atomic	N/A
	Separate	coloring	N/A
		atomic	N/A
OpenMP for GPU	Unified Memory	coloring	Omp_U_c
		atomic	Omp_U_a
	Separate	coloring	-
		atomic	-
OpenACC	Unified Memory	coloring	ACC_U_f_c
		atomic	ACC_U_f_a
	Separate	coloring	ACC_D_f_c
		atomic	ACC_D_f_a

ICCG部分は共通。実装例として参考にしてください。

N/A：言語の制約で実装できない
-：未実装