第74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 in 名古屋」

星野哲也 (助教) hoshino@cc.u-tokyo.ac.jp
大島聡史 (助教) ohshima@cc.u-tokyo.ac.jp

2016年3月14日 (火)
東京大学情報基盤センター

概要

- OpenACC とは
  - OpenACC について
  - OpenMP, CUDA との違い
- OpenACC の指示文
  - 並列化領域指定指示文 (kernels/parallel)
  - データ移動指示文
  - ループ指示文
- OpenACC の実用例
- 実習
  - コンパイラメッセージの見方
  - OpenACC プログラムの実装
  - 各種ツールの使い方
    - NVIDIA Visual Profilerなど

OpenACC とは

OpenACC

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

C/C++

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

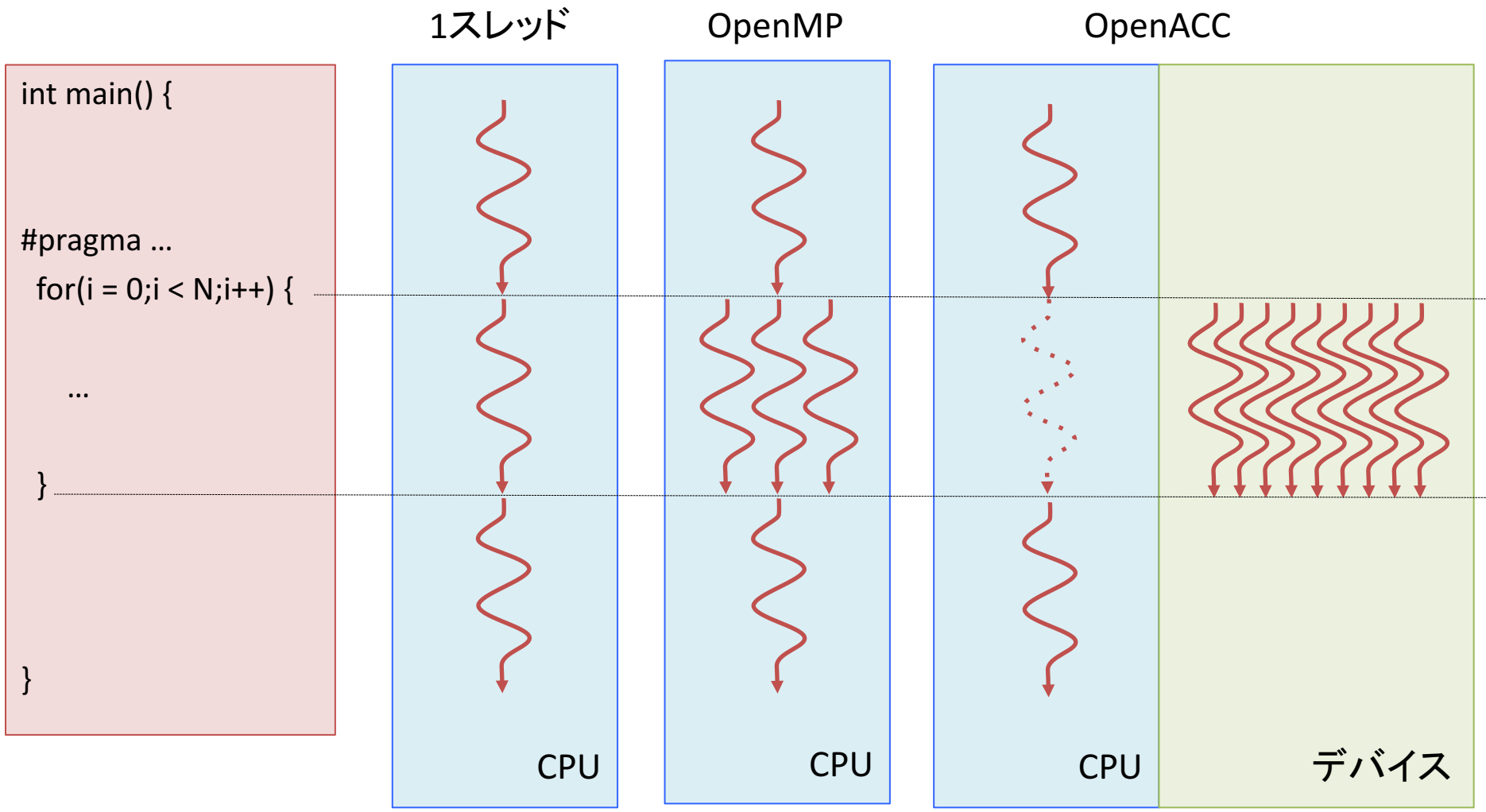
Fortran

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

OpenACC

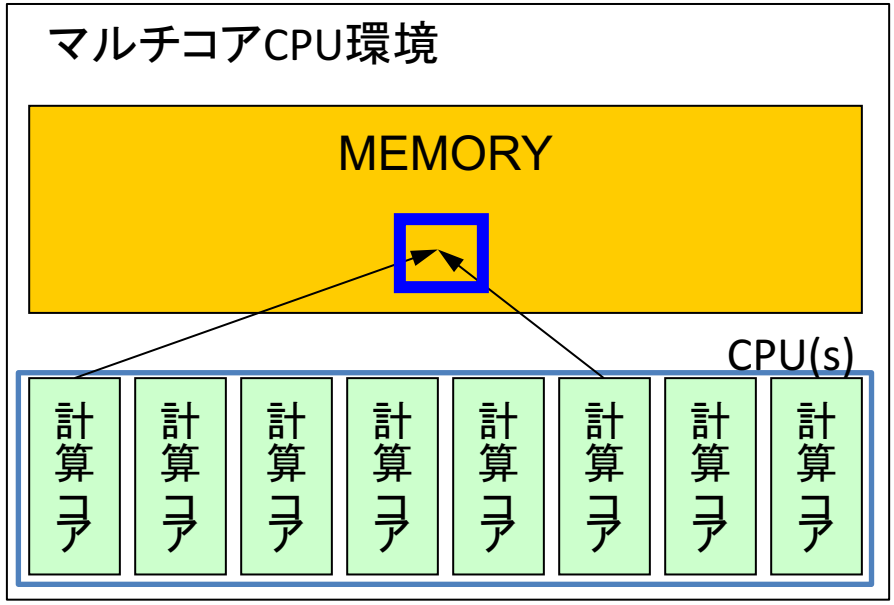
- 規格
  - 各コンパイラベンダ(PGI, Crayなど)が独自に実装していた拡張を統合し共通規格化 (http://www.openacc.org/)
  - 2011年秋にOpenACC 1.0 最新の仕様はOpenACC 2.5
- 対応コンパイラ
  - 商用：PGI, Cray, PathScale
  - 研究用：Omni (AICS), OpenARC (ORNL), OpenUH (U.Houston)
  - フリー：GCC 6.x
    - 開発中 (開発状況： https://gcc.gnu.org/wiki/Offloading)
    - 実用にはまだ遠い

OpenACC と OpenMP の実行イメージ



OpenACC と OpenMP の比較

OpenMPの想定アーキテクチャ

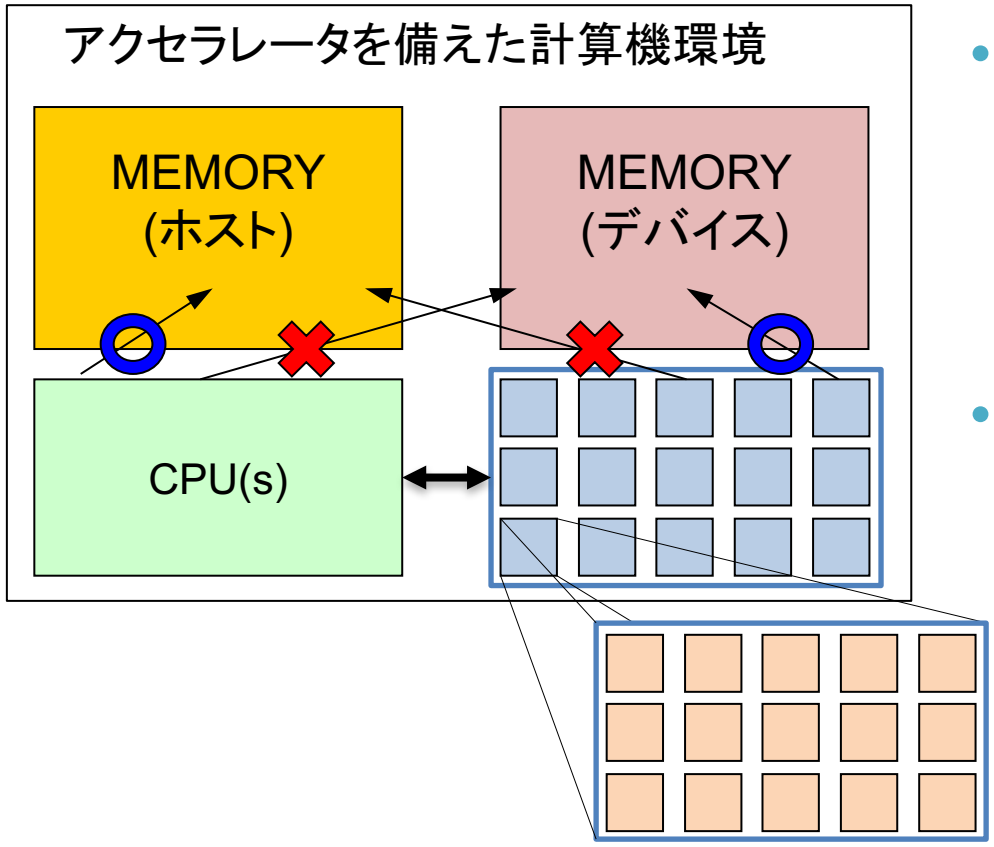


- 計算コアがN個
  - N < 100 程度 (Xeon Phi除く)
- 共有メモリ

一番の違いは対象アーキテクチャの複雑さ

OpenACC と OpenMP の比較

OpenACC の想定アーキテクチャ



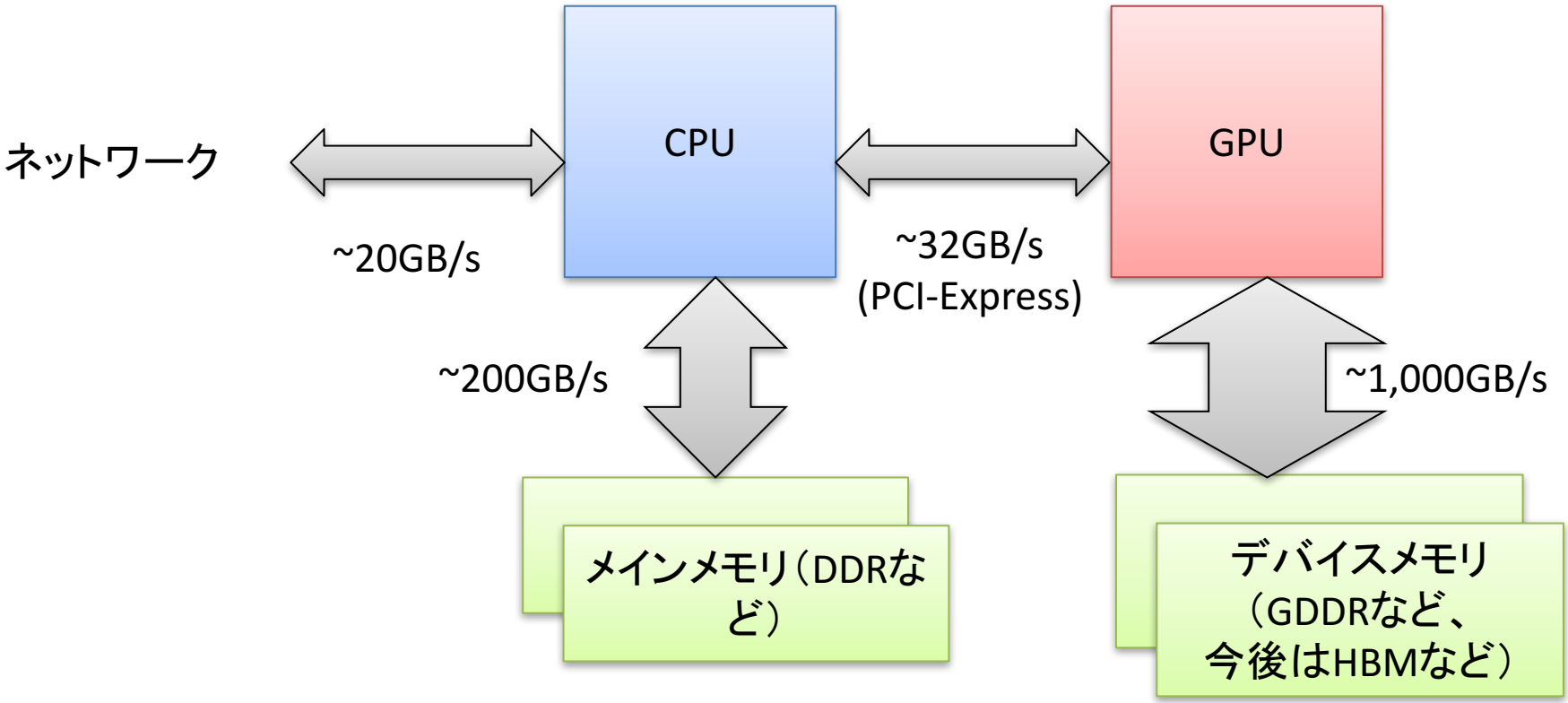
- 計算コアN個をM階層で管理
  - N > 1000 を想定
  - 階層数Mはアクセラレータによる
- ホスト-デバイスで独立したメモリ
  - ホスト-デバイス間データ転送は低速

一番の違いは対象アーキテクチャの複雑さ

OpenACCとOpenMPの比較

- OpenMPと同じもの
  - Fork-Joinという概念に基づく並列化
- OpenMPになくてOpenACCにあるもの
  - ホストとデバイスという概念
    - ホスト-デバイス間のデータ転送
  - 多階層の並列処理
- OpenMPにあってOpenACCにないもの
  - スレッドIDを用いた処理など
    - OpenMPのomp_get_thread_num()に相当するものが無い
- その他、気をつけるべき違い
  - OpenMPと比べてOpenACCは勝手に行うことが多い
    - 転送データ、並列度などを未指定の場合は勝手に決定

想定されるハードウェア構成



- デバイス内外のデータ転送速度差的にも、対象とするプロセッサ内で計算が完結していることが望ましい

OpenACC と CUDA の違い

OpenACC

- 指示文ベース
- 対象：アクセラレータ全般
- 記述の自由度
  - 高レベルな抽象化
  - ある程度勝手にやってくれる
  - デバイスに特化した機能は使えない
    - shuffle機能を使えない など

CUDA

- 言語拡張
- 対象：NVIDIA GPU のみ
- 記述の自由度
  - 低レベルな記述
  - 書いたようにしかやらない
  - デバイスの持つ性能を十分に引き出せる

プログラムの可搬性◎
可読性◎

性能◎

ただし簡単ではない！

最低限動くプログラムを作るには

1. オフロードする領域を決める

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc

  do j = 1, n
    do i = 1, n
      cc = 0
      do k = 1, n
        cc = cc + a(i,k) * b(k,j)
      end do
      c(i,j) = cc
    end do
  end do

end subroutine matmul
```

CUDA

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc

  do j = 1, n
    do i = 1, n
      cc = 0
      do k = 1, n
        cc = cc + a(i,k) * b(k,j)
      end do
      c(i,j) = cc
    end do
  end do

end subroutine matmul
```

最低限動くプログラムを作るには

2. オフロード領域の並列化、カーネルコードの記述

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
!$acc kernels
   do j = 1, n
      do i = 1, n
         cc = 0
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
end subroutine matmul
```

OpenACCはこの時点で実行可能！

CUDA

```
attribute(global) subroutine mm_cuda(a, b, c, n)
  integer, value :: n
  real(8), dimension(n, n) :: a, b, c
  integer :: i, j, k
  real(8) :: cc
  i = (blockidx%x-1) * blockdim%x + threadidx%x
  j = (blockidx%y-1) * blockdim%y + threadidx%y
  cc = 0.0
  do k = 1, n
    cc = cc + a(i, k) * b(k, j)
  end do
  c(i, j) = cc
end subroutine mm_cuda

subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
```

```
end subroutine matmul
```

最低限動くプログラムを作るには

3. GPU用のメモリを確保し、明示的にデータ転送する

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
!$acc data copyin(a, b) copyout(c)
!$acc kernels
   do j = 1, n
      do i = 1, n
         cc = 0
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
!$acc end data
end subroutine matmul
```

CUDA

```
attribute(global) subroutine mm_cuda(a, b, c, n)
  integer, value :: n
  real(8), dimension(n, n) :: a, b, c
  integer :: i, j, k
  real(8) :: cc
  i = (blockidx%x-1) * blockdim%x + threadidx%x
  j = (blockidx%y-1) * blockdim%y + threadidx%y
  cc = 0.0
  do k = 1, n
    cc = cc + a(i, k) * b(k, j)
  end do
  c(i, j) = cc
end subroutine mm_cuda

subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
  real(8), device, allocatable, dimension(:, :) ::  &
     a_dev, b_dev, c_dev


  allocate(a_dev(n, n), b_dev(n, n), c_dev(n, n))
  a_dev(:, :) = a(:, :)
  b_dev(:, :) = b(:, :)



  c(:, :) = c_dev(:, :)
end subroutine matmul
```

最低限動くプログラムを作るには

4. スレッドを割り当てる

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
!$acc data copyin(a, b) copyout(c)
!$acc kernels
!$acc loop gang
   do j = 1, n
!$acc loop vector
      do i = 1, n
         cc = 0
!$acc loop seq
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
!$acc end data
end subroutine matmul
```

CUDA

```
attribute(global) subroutine mm_cuda(a, b, c, n)
  integer, value :: n
  real(8), dimension(n, n) :: a, b, c
  integer :: i, j, k
  real(8) :: cc
  i = (blockidx%x-1) * blockdim%x + threadidx%x
  j = (blockidx%y-1) * blockdim%y + threadidx%y
  cc = 0.0
  do k = 1, n
    cc = cc + a(i, k) * b(k, j)
  end do
  c(i, j) = cc
end subroutine mm_cuda

subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
  real(8), device, allocatable, dimension(:, :) ::  &
     a_dev, b_dev, c_dev
  type(dim3) :: dimGrid, dimBlcok

  allocate(a_dev(n, n), b_dev(n, n), c_dev(n, n))
  a_dev(:, :) = a(:, :)
  b_dev(:, :) = b(:, :)
  dimGrid = dim3( n/16, n/16, 1)
  dimBlock = dim3( 16, 16, 1)
  call mm_cuda<<<dimGrid, dimBlock>>>(a, b, c, n)
  c(:, :) = c_dev(:, :)
end subroutine matmul
```

OpenACC の指示文

並列化領域指定指示文：parallel, kernels

- アクセラレータ上で実行すべき部分を指定
  - OpenMPのparallel指示文に相当
- 2種類の指定方法：parallel, kernels
  - parallel：(どちらかというと) マニュアル
    - OpenMP に近い
    - 「ここからここまでは並列実行領域です。並列形状などはユーザー側で指定します」的な概念
  - kernels：(どちらかというと) 自動的
    - 「ここからここまではデバイス側実行領域です。あとはお任せします」的な概念
  - 細かい指示子・節を加えていくと最終的に同じような挙動になるので、どちらを使うかは好み

kernels/parallel 指示文

kernels

```
program main

!$acc kernels
   do i = 1, N
      ! loop body
   end do
!$acc end kernels

end program
```

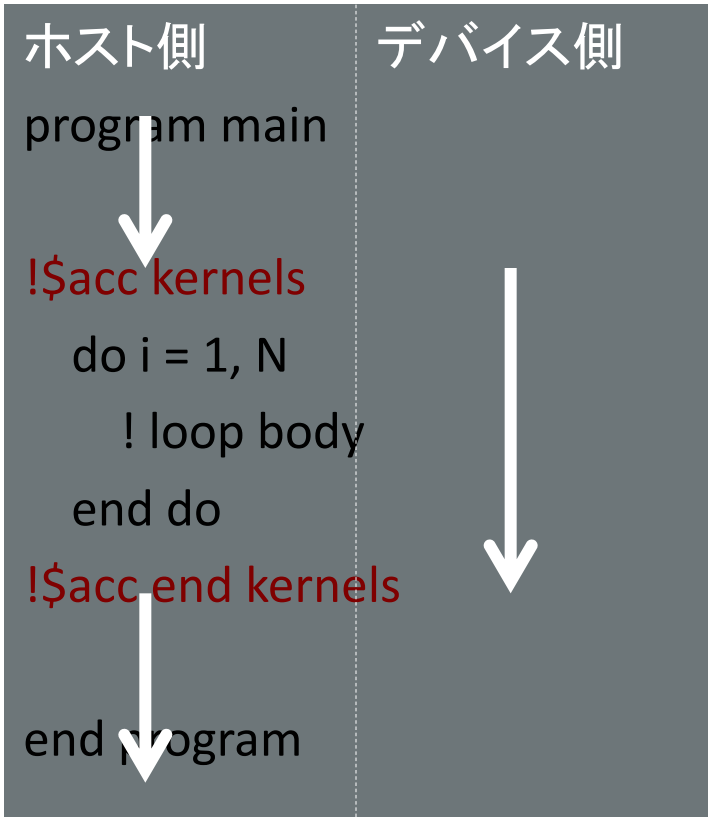
parallel

```
program main

!$acc parallel num_gangs(N)
!$acc loop gang
   do i = 1, N
      ! loop body
   end do
!$acc end parallel

end program
```

kernels/parallel 指示文

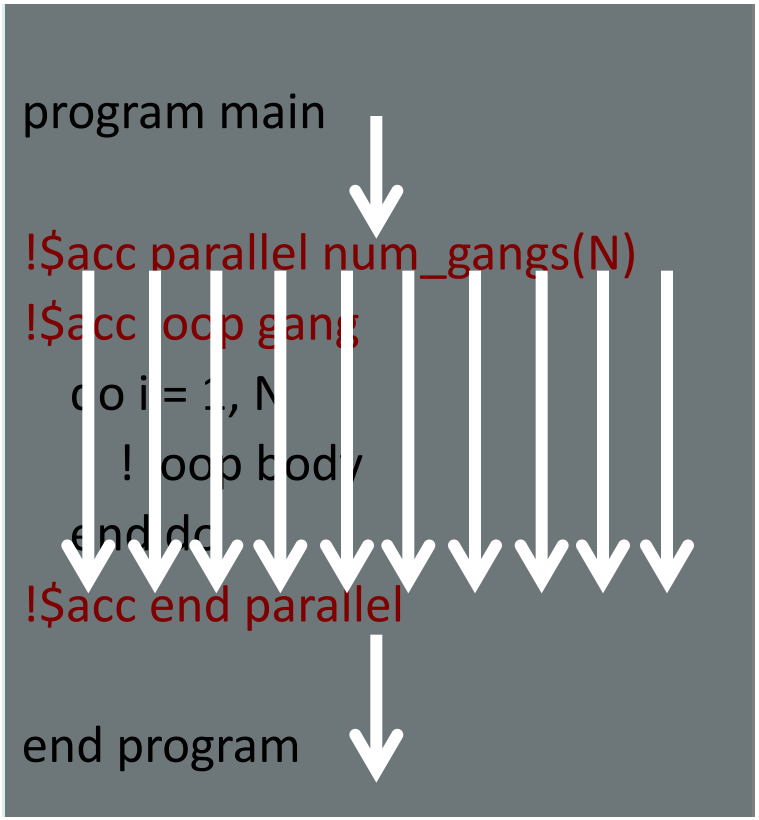
- ホスト-デバイスを意識するのがkernels
- 並列実行領域であることを意識するのがparallel

kernels



「並列数はデバイスに合わせてください」

parallel



「並列数Nでやってください」

kernels/parallel 指示文：指示節

kernels

- async
- wait
- device_type
- if
- default(none)
- copy…

parallel

- async
- wait
- device_type
- if
- default(none)
- copy…
- num_gangs
- num_workers
- vector_length
- reduction
- private
- firstprivate

kernels/parallel 指示文：指示節

kernels

非同期実行に用いる。
今回は扱わない。

実行デバイス毎にパラメータを調整

if(0)/if(.false.)などとするとホスト側で実行される

データの自動転送を行わないようにする

データ指示文の機能を使える（後述）

parallel

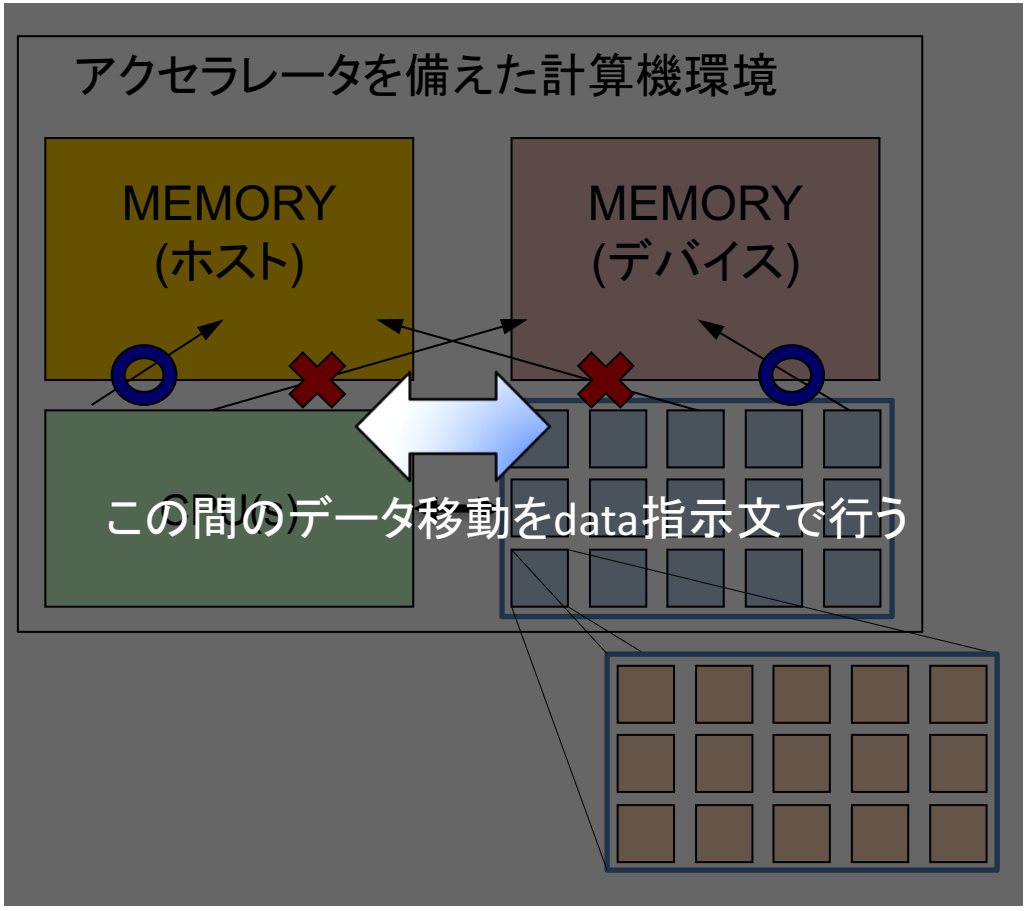
- async
- wait
- device_type
- if
- default(none)
- copy…
- num_gangs
- num_workers
- vector_length
- reduction
- private
- firstprivate

parallelでは並列実行領域であることを意識するため、並列数や変数の扱いを決める指示節がある。

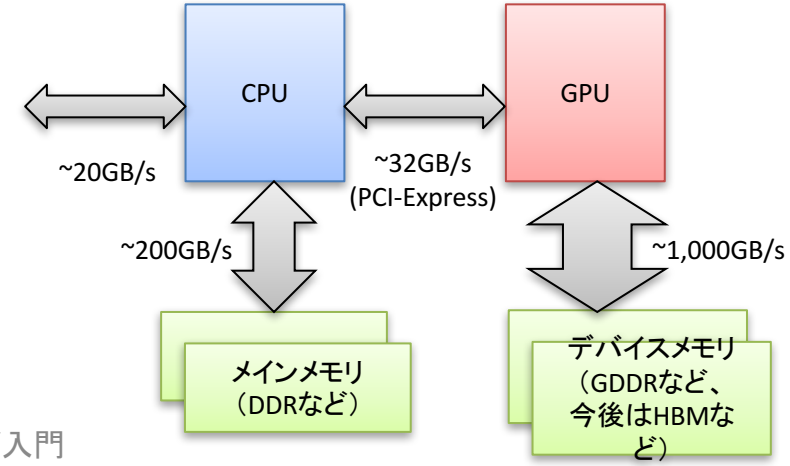
デバイス上で扱われるべきデータについて

- プログラム上のparallel/kernels構文に差し掛かった時、OpenACCコンパイラは実行に必要なデータを自動で転送する
  - 往々にして正しく転送されない。自分で書くべき
  - 構文に差し掛かるたびに転送が行われる(非効率)。後述のdata指示文を用いて自分で書くべき
  - 自動転送はdefault(none)で抑制できる
- スカラ変数は firstprivate として扱われる
  - 指示節により変更可能
- 配列はデバイスに確保される (shared的振る舞い)
  - 配列変数をスレッドローカルに扱うためには private を指定する

データ関連指示文



- ホスト-デバイス間のデータ移動を行う
- データの一貫性を保つのはユーザーの責任
- ホスト-デバイス間のデータ転送は相対的に遅いので要最適化



データ関連指示文：data, enter/exit data

- デバイス側で扱われるべきデータとその領域を指定
  - CUDAでいう、cudaMalloc, cudaMemcpy, cudaFree を行う
- data 指示文 (推奨)
  - cudaMalloc + cudaMemcpy (H ⇄ D) + cudaFree
  - 構造ブロックに対してのみ適用可
    - コードの見通しが良い
- enter data 指示文
  - cudaMalloc + cudaMemcpy (H → D)
  - exit data とセット。構造ブロック以外にも使える
- exit data 指示文
  - cudaMemcpy (H ← D) + cudaFree
  - enter data とセット。構造ブロック以外にも使える

データ関連指示文：data 指示文

Fortran

```
subroutine copy(dis, src)
  real(4), dimension(:) :: dis, src

!$acc data copy(src,dis)
!$acc kernels
   do i = 1, N
      dis(i) = src(i)
   end do
!$acc end kernels
!$acc end data
end subroutine copy
```

C言語

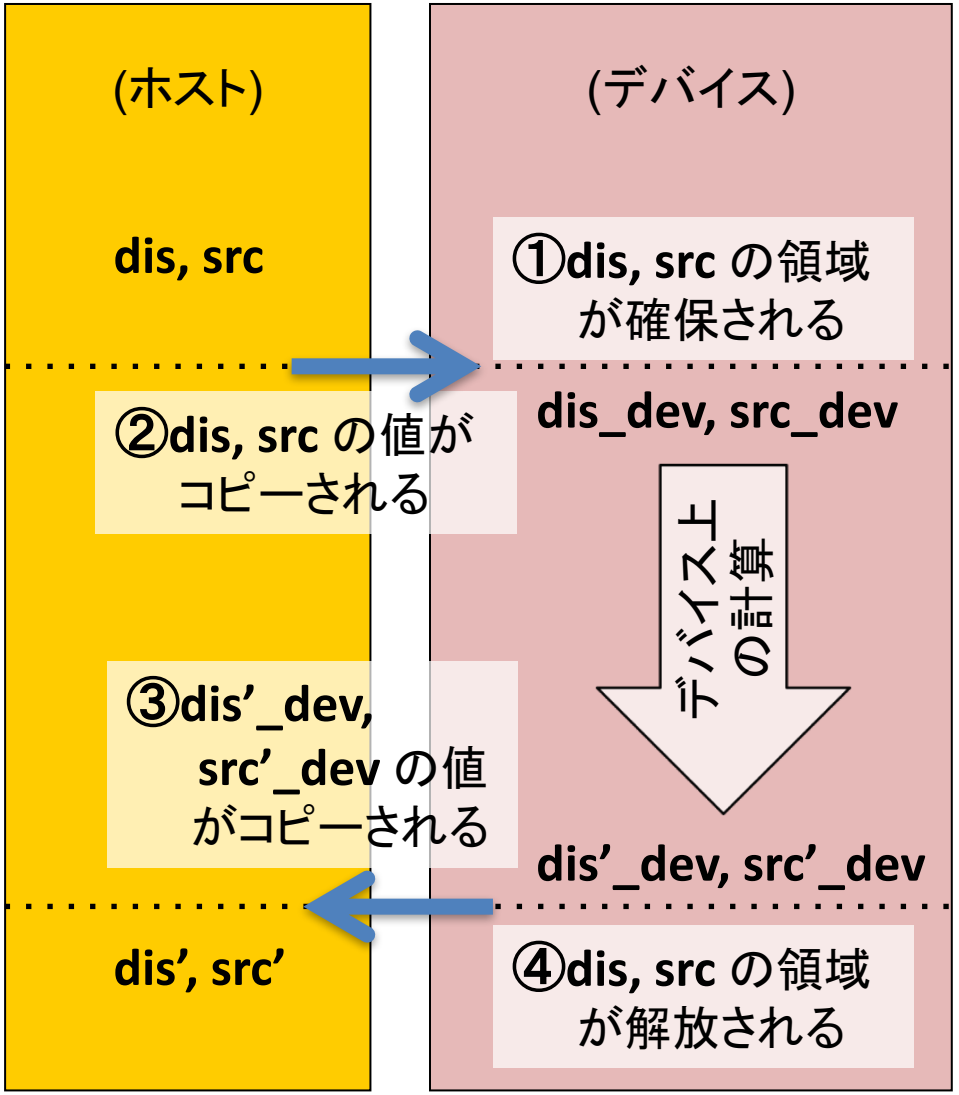
```
void copy(float *dis, float *src) {
  int i;
#pragma acc data copy(src[0:N] ¥
    dis[0:N])
  {
#pragam acc kernels
      for(i = 0;i < N;i++){
         dis[i] = src[i];
      }
  }
}
```

構造ブロックにのみ適用可
C言語なら {} で囲める部分

データ関連指示文：data指示文イメージ

Fortran

```
subroutine copy(dis, src)
  real(4), dimension(:) :: dis, src

!$acc data copy(src,dis)
!$acc kernels
  do i = 1, N
    dis(i) = src(i)
  end do
!$acc end kernels
!$acc end data

end subroutine copy
```



データ関連指示文：enter/exit 指示文

```
void main() {
  double *q;
  int step;
  for(step = 0;step < N;step++){
    if(step == 0) init(q);
    solverA(q);
    solverB(q);
     ….
    if(step == N) fin(q);
  }
}
```

```
void init(double *q) {
  q = (double *)malloc(sizeof(double)*M);
  q = … ; // 初期化
#pragma acc enter data copyin(q[0:M])
}
```

```
void fin(double *q) {
#pragma acc exit data copyout(q[0:M])
  print(q); //結果出力
  free(q);
}
```

データ関連指示文：指示節

data

- if
- copy
- copyin
- copyout
- create
- present
- present_or_...
- deviceptr
  - CUDAなどと組み合わせる時に利用。cudaMallocなどで確保済みのデータを指定し、OpenACCで扱い可とする

enter data

- if
- async　　非同期転送用
- wait
- copyin
- create
- present_or_...

enter data

- if
- async
- wait
- copyout
- delete

データ関連指示文：指示節

- **copy**
  - data 指示文へ差し掛かった時、ホスト側からデバイス側へデータをコピーし、data 指示文終了時にデバイス側からホスト側へコピー
- **copyin/copyout**
  - ホスト/デバイスからの入力/出力のみ行う
- **create**
  - デバイス上に配列を作成、コピーは行わない
- **present**
  - デバイス上に既に存在することを知らせる
- **present_or_copy/copyin/copyout/create** (省略形：pcopy) など
  - デバイス上に既にあれば copy/copyin/copyout/create せず、なければする

ただしOpenACC2.5以降では、
copy, copyin, copyout の挙動は
pcopy, pcopyin, pcopyout と同一

データ関連指示文：データ転送範囲指定

- 送受信するデータの範囲の指定
  - 部分配列の送受信が可能
  - 注意：FortranとCで指定方法が異なる
- 二次元配列Aを転送する例

Fortran版

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

fortranでは開始点と終了点を指定

C版

```
#pragma acc data copy(A[start1:length1][start2:length2])
  …
#pragma acc end data
```

Cでは先頭と長さを指定

データ関連指示文：update 指示文

- 既にデバイス上に確保済みのデータを対象とする
  - cudaMemcpy (H ⇄ D) の機能を持っていると思えば良い

```
!$acc data copy( A(:,:) )
do step = 1, N
  …
!$acc update host( A(1:2,:) )
  call comm_boundary( A )
!$acc update device( A(1:2,:) )
  …
end do
!$acc end data
```

update

- if
- async
- wait
- device_type
- self　#host と同義
- host　# H ← D
- device　# H → D

階層的並列モデルとループ指示文

- OpenACC ではスレッドを階層的に管理
  - gang, worker, vector の3階層
  - gang：workerの塊 一番大きな単位
  - worker：vectorの塊
  - vector：スレッドに相当する一番小さい処理単位
- loop 指示文
  - parallel/kernels中のループの扱いについて指示
  - 粒度(gang, worker, vector)の指定
  - ループ伝搬依存の有無の指定

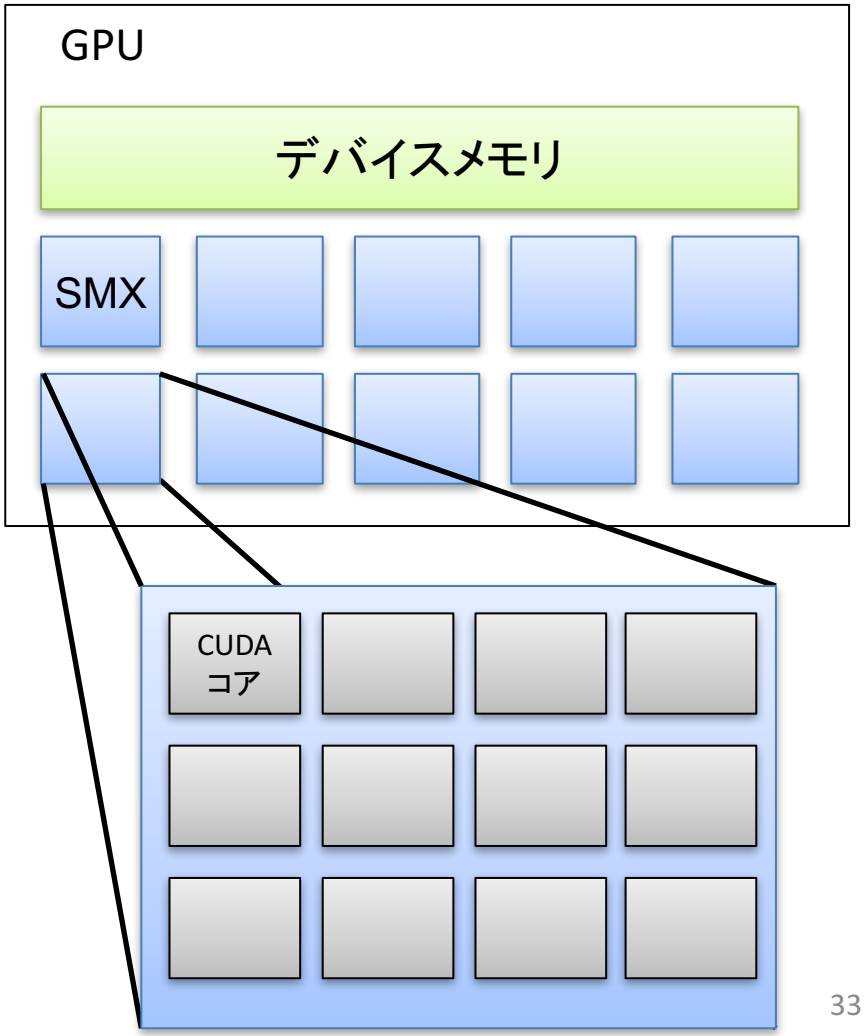
GPUでの行列積の例

```
!$acc kernels
!$acc loop gang
   do j = 1, n
!$acc loop vector
      do i = 1, n
         cc = 0
!$acc loop seq
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
```

階層的並列モデルとアーキテクチャ

- OpenMPは1階層
  - マルチコアCPUも1階層
- CUDAは block と thread の2階層
  - NVIDA GPUも2階層
    - 1 SMX に複数CUDA coreを搭載
    - 各コアはSMXのリソースを共有
- OpenACCは3階層
  - 今後出てくる様々なアクセラレータに対応するため

- NVIDIA Kepler GPUの構成



ループ指示文:指示節

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

ループ指示文:指示節

loop

- **collapse**
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

3つのループが
一重化される

```
!$acc kernels
!$acc loop collapse(3) gang vector
  do k = 1, 10
    do j = 1, 10
      do i = 1, 10

        ....

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

並列化するにはループ長の短すぎる
ループに使う

ループ指示文：指示節

loop

- collapse
- **gang**
- **worker**
- **vector**
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

```
!$acc kernels
!$acc loop gang(N)
   do k = 1, N
!$acc loop worker(1)
      do j = 1, N
!$acc loop vector(128)
         do i = 1, N

           ....

!$acc kernels
!$acc loop gang vector(128)
   do i = 1, N

      ....
```

vectorはworkerより内側
workerはgangより内側

ただし1つのループに
複数つけるのはOK

数値の指定は難しいので、最初は
コンパイラ任せでいい

ループ指示文:指示節

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- **independent**
- private
- reduction

Bに間接参照→

```
do j = 1, N
  do i = 1, N
    idxI(i) = i;  idxJ(j) = j
  end do
end do

!$acc kernels &
!$acc& copyin(A, idxI, idxJ) copyout(B)
!$acc loop independent gang
do j = 1, N
  !$acc loop independent vector(128)
  do i = 1, N
    B(idxI(i),idxJ(j)) = alpha * A(i,j)
  end do
end do
!$acc end kernels
```

OpenACCコンパイラは保守的。
依存関係が生じそうなら並列化しない。

ループ指示文：指示節

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- **reduction**

```
!$acc kernels &
   !$acc loop reduction(+:val)
   do i = 1, N
      val = val + 1
   end do
!$acc end kernels
```

acc reduction (+:val)

演算子

対象とする変数

簡単なものであれば、PGIコンパイラは自動でreductionを入れてくれる

利用できる演算子
(OpenACC2.0仕様書より)

C and C++		Fortran	
operator	initialization value	operator	initialization value
+	0	+	0
*	1	*	1
max	least	max	least
min	largest	min	largest
&	~0	iand	all bits on
|	0	ior	0
^	0	ieor	0
&&	1	.and.	.true.
||	0	.or.	.false.
		.eqv.	.true.
		.neqv.	.false.

関数呼び出し指示文：routine

- parallel/kernels領域内から関数を呼び出す場合、routine指示文を使う

```
#pragma acc routine vector
extern double vecsum(double *A);
…
#pragma acc parallel num_gangs(N) vector_length(128)
for (int i = 0;i < N; i++){
   max = vecsum(A[i*N]);
}
```

プロトタイプ宣言にもつける

```
#pragma acc routine vector
double vecsum(double *A){
   double x = 0;
#pragma acc loop reduction(+:x)
  for(int j = 0;j < N;j++){
      x += A[j];
  }
   return x;
}
```

その他知っておくと便利なもの

- #ifdef _OPENACC
  - OpenACC API の呼び出し時などに使う
- if clause の使い方
  - kernels/parallel, data 指示文などに使える
  - fortranならif(.false.), Cならif(0) を指定すると、ホスト側で実行される。デバッグに便利
- atomic
  - 並列領域内のatomic領域を囲むことで利用
- declare 指示文
  - fortranのmodule内変数、Cのglobal変数などを使う時に用いる
- CUDA 関数の呼び出し
  - CUDA関数の呼び出しインターフェースも存在する
  - 基本はOpenACCで作成し、どうしても遅い部分だけCUDAという実装が可能

アプリケーションの移植方法

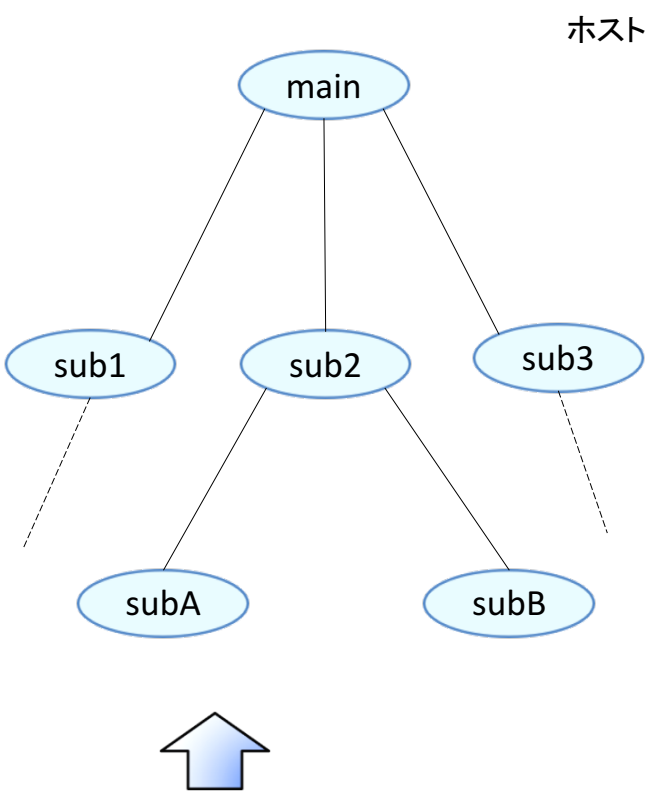
アプリケーションのOpenACC化手順

1. プロファイリングによるボトルネック部位の導出
2. ボトルネック部位のOpenACC化
    1. 並列化可能かどうかの検討
    2. (OpenACCの仕様に合わせたプログラムの書き換え)
    3. parallel/kernels指示文適用
3. data指示文によるデータ転送の最適化
4. OpenACCカーネルの最適化

1 ~ 4 を繰り返し適用

アプリケーションのOpenACC化手順

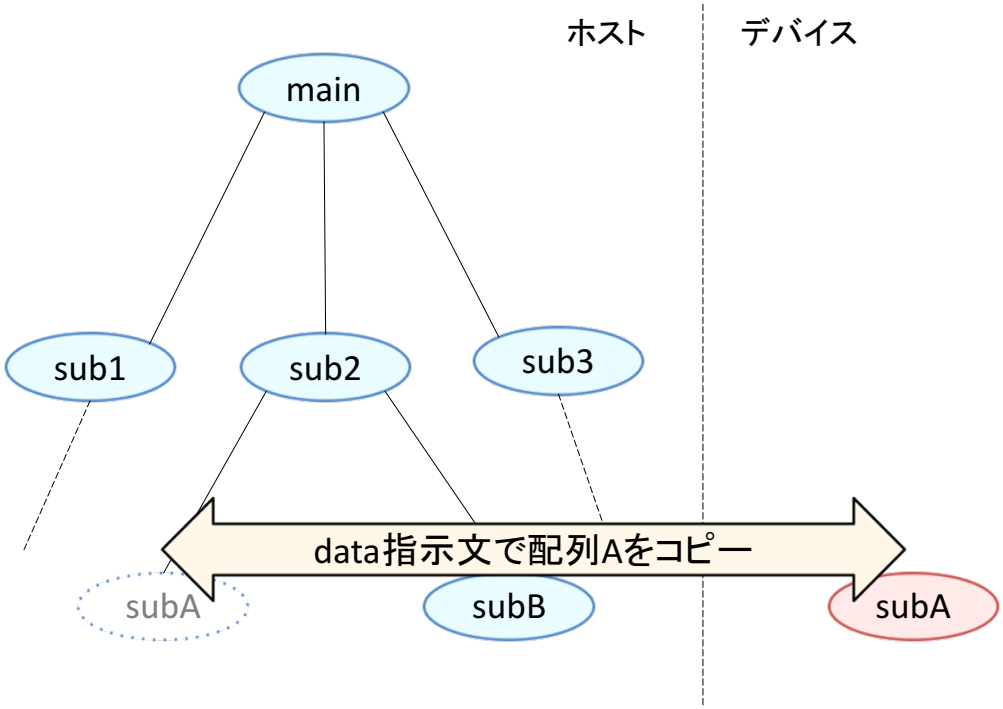
```
int main(){
  double A[N];
  sub1(A);
  sub2(A);
  sub3(A);
}

sub2(double A){
  subA(A);
  subB(A);
}

subA(double A){

  for( i = 0 ~ N ) {
    …
  }
}
```



葉っぱの部分から
OpenACC化を始める

アプリケーションのOpenACC化手順

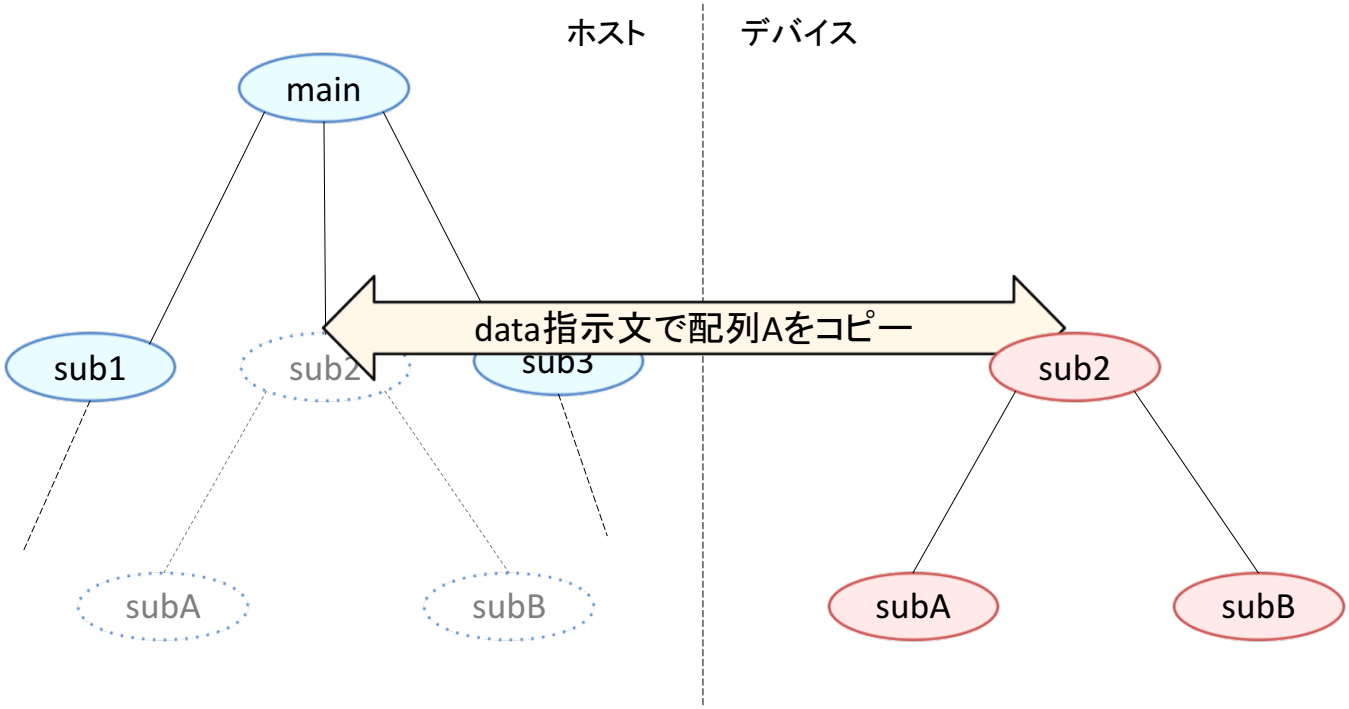
```
int main(){
  double A[N];
  sub1(A);
  sub2(A);
  sub3(A);
}

sub2(double A){
  subA(A);
  subB(A);
}

subA(double A){
#pragma acc …
  for( i = 0 ~ N ) {
    …
  }
}
```



この状態でも必ず正しい結果を得られるように作る！
この時、速度は気にしない！

アプリケーションのOpenACC化手順

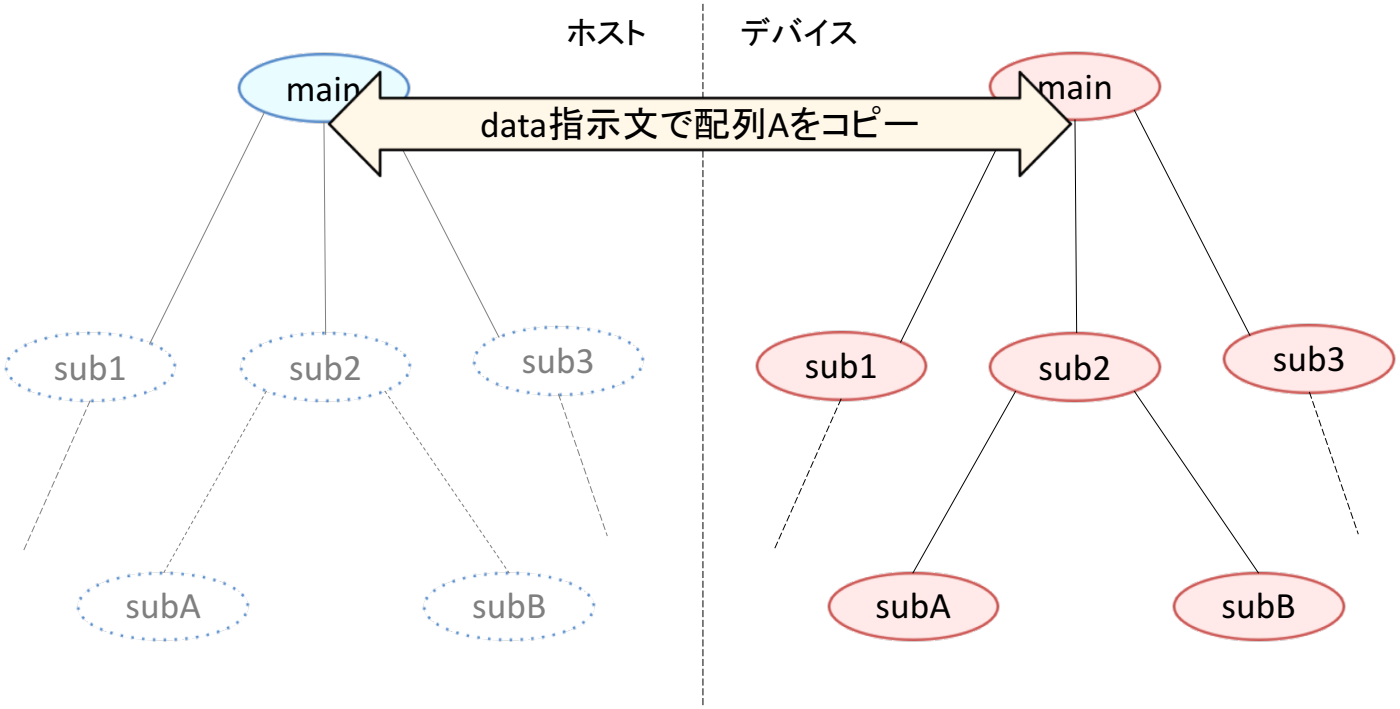
```
int main(){
  double A[N];
  sub1(A);
#pragma acc data
  {
    sub2(A);
  }
  sub3(A);
}

sub2(double A){
  subA(A);
  subB(A);
}

subA(double A){
#pragma acc …
  for( i = 0 ~ N ) {
    …
  }
}
```



徐々にデータ移動を上流に移動する

アプリケーションのOpenACC化手順

```
int main(){
  double A[N];
#pragma acc data
  {
    sub1(A);
    sub2(A);
    sub3(A);
  }
}

sub2(double A){
  subA(A);
  subB(A);
}

subA(double A){
#pragma acc …
  for( i = 0 ~ N ) {
    …
  }
}
```



ここまで来たら、ようやく個別のカーネルの最適化を始める。

※データの転送時間が相対的に十分小さくなればいいので、かならずしも最上流までやる必要はない

Q & A

実習

※今回の実習の例は、全てPGIコンパイラ16.4を使った際の例です

実習概要

- OpenACC プログラムのコンパイル
  - PGIコンパイラのメッセージの読み方
- OpenACC プログラムの作成
  - 行列積、diffusion
- OpenACC プログラムの最適化
  - NVIDIA visual profiler の使い方など

OpenACCサンプル集

- Reedbush へログイン
  - $ ssh **-Y** reedbush.cc.u-tokyo.ac.jp –l txxxxx
- module のロード
  - $ module load pgi/17.1
  - $ module load cuda/8.0.44
- ワークディレクトリに移動
  - $ cdw
- OpenACC_samples のコピー
  - $ cp /home/pz0108/z30108/OpenACC_samples.tar.gz .
  - $ tar zxvf OpenACC_samples.tar.gz
- OpenACC_samples へ移動
  - $ cd OpenACC_samples
  - $ ls

    C/　F/　　#CとFortran好きな方を選択

PGIコンパイラによるメッセージの確認

- コンパイラメッセージの確認はOpenACCでは極めて重要
  - OpenMP と違い、
    - 保守的に並列化するため、本来並列化できるプログラムも並列化されないことがある
    - 並列化すべきループが複数あるため、どのループにどの粒度(gang, worker, vector)が割り付けられたかしるため
    - ターゲットデバイスの性質上、立ち上げるべきスレッド数が自明に決まらず、スレッドがいくつ立ち上がったか知るため
  - メッセージを見て、プログラムを適宜修正する
- コンパイラメッセージ出力方法
  - コンパイラオプションに -Minfo=accel をつける

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード

```
8.    subroutine acc_kernels()
9.      double precision :: A(N,N), B(N,N)
10.     double precision :: alpha = 1.0
11.     integer :: i, j
12.     A(:,:) = 1.0
13.     B(:,:) = 0.0
14.     !$acc kernels
15.     do j = 1, N
16.       do i = 1, N
17.         B(i,j) = alpha * A(i,j)
18.       end do
19.     end do
20.     !$acc end kernels
21.   end subroutine acc_kernels
```

コンパイラメッセージ(fortran)

```
pgfortran -O3 -acc -Minfo=accel -ta=tesla,cc60 -Mpreprocess  acc_compute.f90
-o acc_compute
acc_kernels:
     14, Generating implicit copyin(a(:,:))
         Generating implicit copyout(b(:,:))
     15, Loop is parallelizable
     16, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         16, !$acc loop gang, vector(32) ! blockidx%x threadidx%x


    ....
```

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード

```
8.    subroutine acc_kernels()
9.      double precision :: A(N,N), B(N,N)
10.     double precision :: alpha = 1.0
11.     integer :: i, j
12.     A(:,:) = 1.0
13.     B(:,:) = 0.0
14.     !$acc kernels
15.     do j = 1, N
16.       do i = 1, N
17.         B(i,j) = alpha * A(i,j)
18.       end do
19.     end do
20.     !$acc end kernels
21.   end subroutine acc_kernels
```

サブルーチン名

コンパイラメッセージ(fortran)

```
pgfortran -O3 -acc -Minfo=accel -ta=tesla,cc60 -Mpreprocess  acc_compute.f90
-o acc_compute
acc_kernels:
     14, Generating implicit copyin(a(:,:))
         Generating implicit copyout(b(:,:))
     15, Loop is parallelizable
     16, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         16, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
    ....
```

配列aはcopyin, bはcopyoutとして扱われます

15, 16行目の2重ループは(32x4)のスレッドでブロック分割して扱います。

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード(C)

```
40.  void acc_kernels(double *A, double *B){
41.   double alpha = 1.0;
42.   int i,j;
          / * A と B 初期化 */
50.  #pragma acc kernels
51.   for(j = 0;j < N;j++){
52.    for(i = 0;i < N;i++){
53.     B[i+j*N] = alpha * A[i+j*N];
54.    }
55.   }
56.  }
```

コンパイラメッセージ(C)

```
pgcc -O3 -acc -Minfo=accel -ta=tesla,cc60 -Mcuda acc_compute.c -o acc_compute
acc_kernels:
     50, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     51, Loop carried dependence of B-> prevents parallelization
         Loop carried backward dependence of B-> prevents vectorization
         Complex loop carried dependence of B->,A-> prevents parallelization
         Accelerator scalar kernel generated
         Accelerator kernel generated
         Generating Tesla code
         51, #pragma acc loop seq
         52, #pragma acc loop seq
     52, Complex loop carried dependence of B->,A-> prevents parallelization
```

配列aはcopyin,
bはcopyとして扱われます

ループ伝搬依存が見つかったので並列化しませんの意。ポインタAとBが同じ領域を指していることを警戒して、並列化しない。

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード(C)

```
59.  void acc_kernels(double *restrict A,
     double *restrict B){
60.   double alpha = 1.0;
61.   int i,j;
          / * A と B 初期化 */
69.  #pragma acc kernels
70.   for(j = 0;j < N;j++){
71.    for(i = 0;i < N;i++){
72.     B[i+j*N] = alpha * A[i+j*N];
73.    }
74.   }
75.  }
```

コンパイラメッセージ(C)

```
acc_kernels_restrict:
     69, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     70, Loop carried dependence of B-> prevents parallelization
         Loop carried backward dependence of B-> prevents vectorization
     71, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         70, #pragma acc loop seq
         71, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

2次元配列を1次元化して扱っているため、i+j*Nが実は同じ場所を指す可能性を考慮し、70行目を並列化していない。

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード(C)

```
78.  void acc_kernels(double *restrict A,
     double *restrict B){
79.   double alpha = 1.0;
80.   int i,j;
       / * A と B 初期化 */
88.  #pragma acc kernels
89.  #pragma acc loop independent
90.   for(j = 0;j < N;j++){
91.  #pragma acc loop independent
92.    for(i = 0;i < N;i++){
93.     B[i+j*N] = alpha * A[i+j*N];
94.    }
95.   }
96.  }
```

コンパイラメッセージ(C)

```
acc_kernels_independent:
     88, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     90, Loop is parallelizable
     92, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         90, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         92, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

ようやくまともに並列化

実習1
コンパイラのメッセージを確認しよう

- acc_compute.f90 or acc_compute.c のソースと、コンパイルメッセージを見比べてください
  - コンパイル：$ make acc_compute
- 注目点
  - parallel / kernels での違い
  - 間接参照があった際のコンパイラメッセージ
    - acc_kernels_BAD_indirect_reference と acc_kernels_indirect_reference の比較

PGI_ACC_TIME によるOpenACC 実行の確認

- PGI環境の場合、OpenACC プログラムが実行されているかを確認するには、環境変数PGI_ACC_TIME を使うのが簡単
- 使い方 (一般的なLinux環境、またはインタラクティブジョブ実行時)
  - $ export PGI_ACC_TIME=1
  - $ (プログラムの実行)
- 一般的なスパコン環境では、ジョブの中で環境変数を設定する必要が有る
  - ジョブスクリプト中に書いてある

PGI_ACC_TIME による OpenACC 実行の確認

- OpenACC_samples を利用
- $ qsub acc_compute.sh
  - 実行が終わると以下ができる
    - acc_compute.sh.eXXXXX (標準エラー出力)
    - acc_compute.sh.oXXXXX (標準出力)
- $ less acc_compute.sh.eXXXXX

```
40.  void acc_kernels(double *A, double *B){
41.   double alpha = 1.0;
42.   int i,j;
          / * A と B 初期化 */
50.  #pragma acc kernels
51.   for(j = 0;j < N;j++){
52.    for(i = 0;i < N;i++){
53.     B[i+j*N] = alpha * A[i+j*N];
54.    }
55.   }
56.  }
```

PGI_ACC_TIME による出力メッセージ

```
Accelerator Kernel Timing data
/lustre/pz0108/z30108/OpenACC_samples/C/acc_compute.c
  acc_kernels  NVIDIA  devicenum=0
    time(us): 149,101
    50: compute region reached 1 time
        51: kernel launched 1 time
            grid: [1]  block: [1]
             device time(us): total=140,552 max=140,552 min=140,552 avg=140,552
            elapsed time(us): total=140,611 max=140,611 min=140,611 avg=140,611
    50: data region reached 2 times
        50: data copyin transfers: 2
             device time(us): total=3,742 max=3,052 min=690 avg=1,871
        56: data copyout transfers: 1
             device time(us): total=4,807 max=4,807 min=4,807 avg=4,807
```

← 起動したスレッド数

↓カーネル実行時間

← データ移動の回数・時間

実習2
OpenACCプログラムを実行してみよう

- acc_data.f90 or acc_data.c のソースと、コンパイルメッセージを見比べてください
  - コンパイル：$ make acc_data
- プログラムを実行し、PGI_ACC_TIMEの出力を確認してください
  - 実行
    - $ qsub acc_data.sh
  - バッチジョブ実行が終了すると acc_data.sh.oXXXXXX (標準出力), acc_data.sh.eXXXXXX (標準エラー出力) の2ファイルが出来る
  - PGI_ACC_TIME の出力確認
    - $ less acc_data.sh.eXXXXXX　#標準エラーの方
- 注目点
  - acc_data_copy, acc_data_copyinout の実行時間の違い

実習3
OpenACCプログラムを作ろう

- 行列積のOpenACC化
- matmul.f90 または matmul.c を用いる。
  - acc_matmul ルーチンにOpenACC指示文を加えてください。
  - 注意：コンパイラの出力をよく見てください
    - よく見るエラー例(C)：
      - Accelerator restriction: size of the GPU copy of C,B is unknown

出力例

```
====== OpenACC matmul program ======
1024 * 1024 matrix
check result...OK
elapsed time[sec] :    0.00708
FLOPS[GFlops]     :  302.85417
```

実習4
OpenACCプログラムを速くしよう

- 拡散方程式のプログラムの高速化
- diffusion.f90 または diffusion.c を用いる。
  - 既にOpenACC化されていますが、実行してみると…

出力例(C)

```
(nx, ny, nz) = (128, 128, 128)
elapsed time : 574.247 (s)
flops        : 0.078 (GFlops)
throughput   : 0.096 (GB/s)
accuracy     : 4.443942e-06
count        : 1638
```

すごく遅い(特にC)

- 確認手順
  1. きちんと並列化されているか？ → コンパイラメッセージ
  2. 無駄にデータ転送してないか？ → PGI_ACC_TIME
  3. どっちもOKだけどなお遅い？ → NVIDIA Visual Profiler (nvvp)
     - https://reedbush-www.cc.u-tokyo.ac.jp/session_login.cgi に詳細資料あり