第58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」 OpenACC編

2016年6月8日 (水)

東京大学情報基盤センター
助教 星野哲也 (hoshino@cc.u-tokyo.ac.jp)
助教 大島聡史 (ohshima@cc.u-tokyo.ac.jp)

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 (web掲載なし)
- 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とは

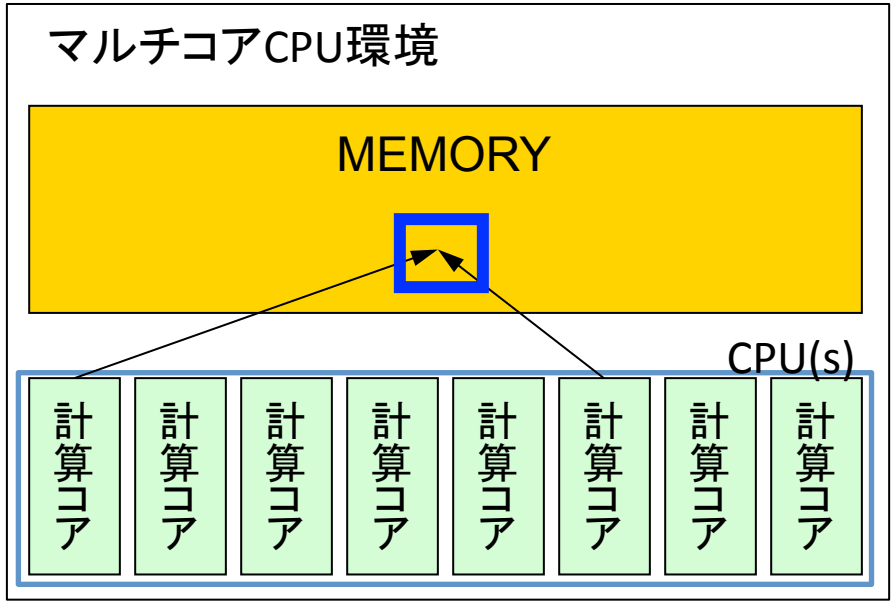
- アクセラレータ向けの指示文ベースのプログラミングモデル
  - C/C++, Fortran 対応
- アクセラレータとは
  - NVIDIA/AMD GPU や Intel Xeon Phi など
- 指示文ベースプログラミングモデルとは
  - 指示文:コンパイラへのヒント
  - マルチコア向けのOpenMP など
  - 記述が簡便, メンテナンスなどをしやすい
  - コードの可搬性(portability)が高い
    - 対応していない環境では無視される

OpenACCとは

- 2011年頃から具体的な規格化
  - 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/)
  - 最新の仕様はOpenACC 2.5
    - OpenACC 1.0 (2011年11月)
    - OpenACC 2.0 (2013年6月)　←本演習で利用
    - OpenACC 2.5 (2015年10月)　←サポートは今後
- 対応コンパイラ
  - 商用：PGI, Cray, PathScale
  - OSS、研究用：Omni (AICS), OpenARC (ORNL), OpenUH (U.Houston), GCC 5.x以上 など
    - FortranやGPU以外のHWへの対応はあまり進んでいない

OpenACC と OpenMP の比較

OpenMPの想定アーキテクチャ

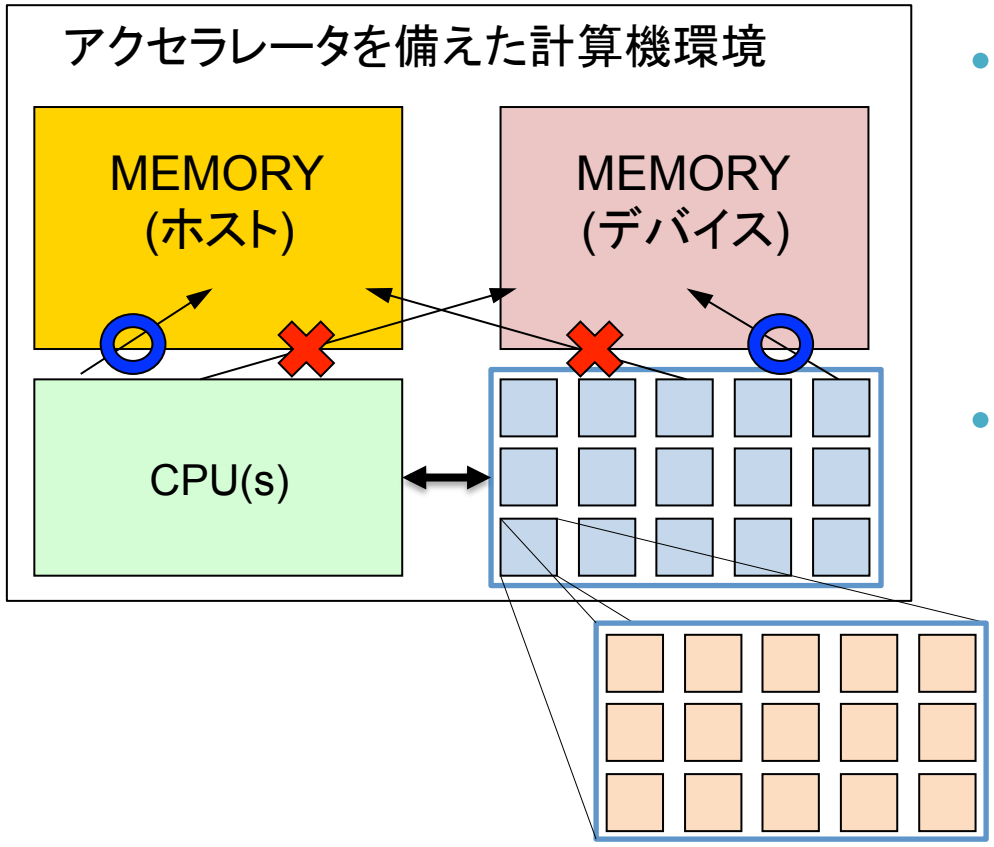


- 計算コアがN個
  - N < 100 程度 (Xeon Phi除く)
- 共有メモリ

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



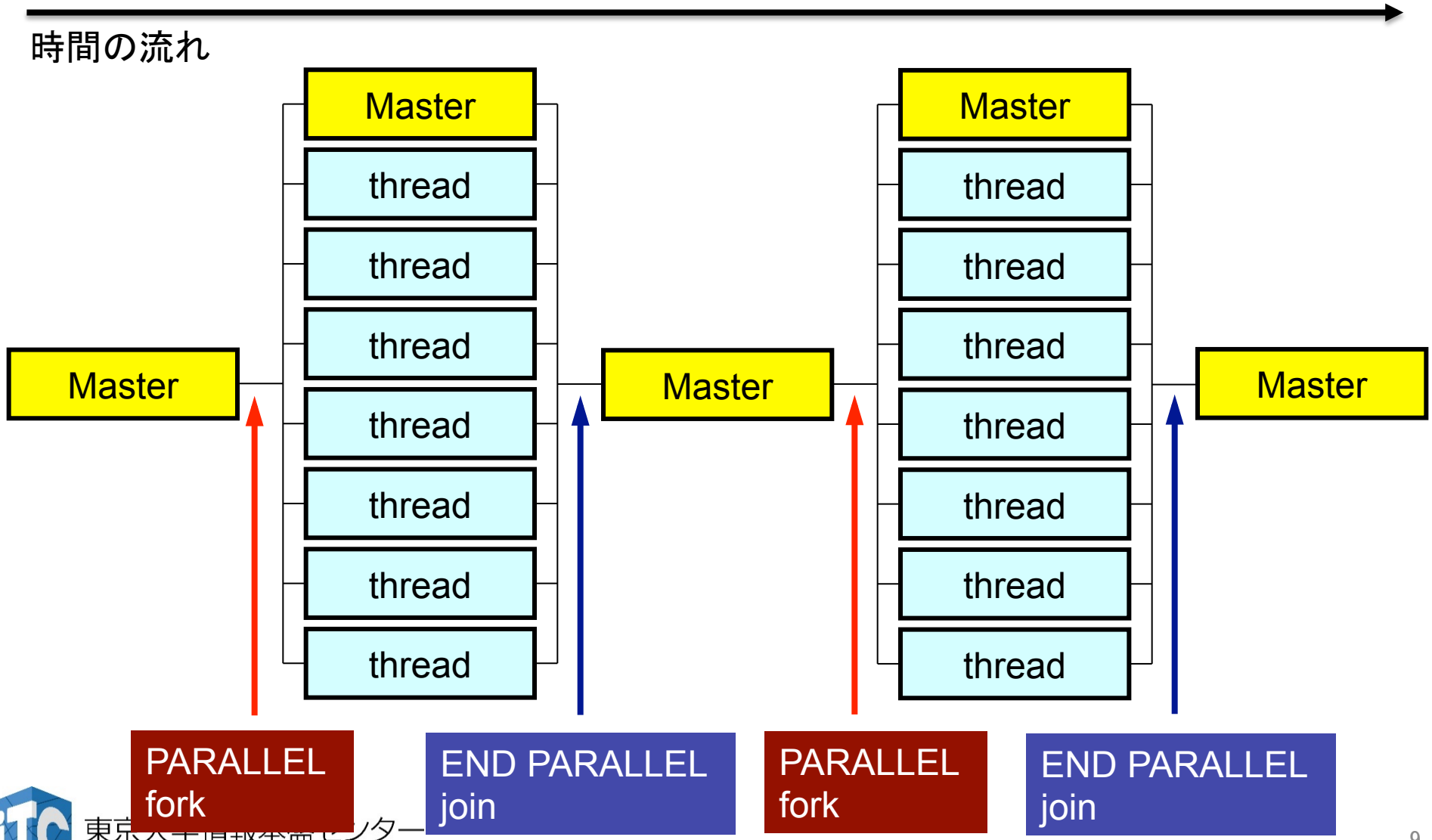
- 計算コアN個をM階層で管理
  - N > 1000 を想定
  - 階層数Mはアクセラレータによる
- ホスト-デバイスで独立したメモリ
  - ホスト-デバイス間データ転送は低速

一番の違いは対象アーキテクチャの複雑さ

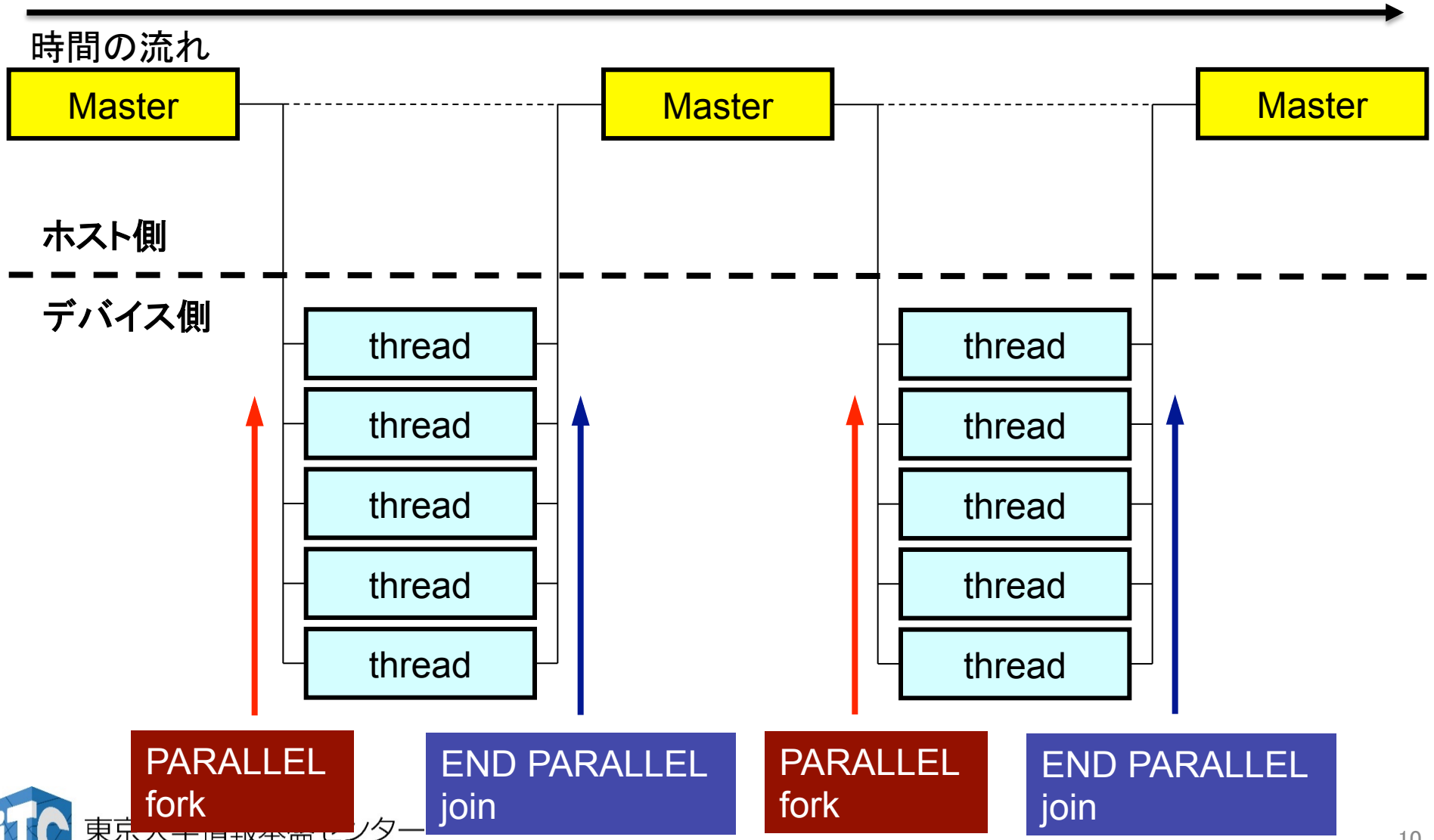
OpenACC と OpenMP の比較

- 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、並列度などをほっとくと勝手に決定

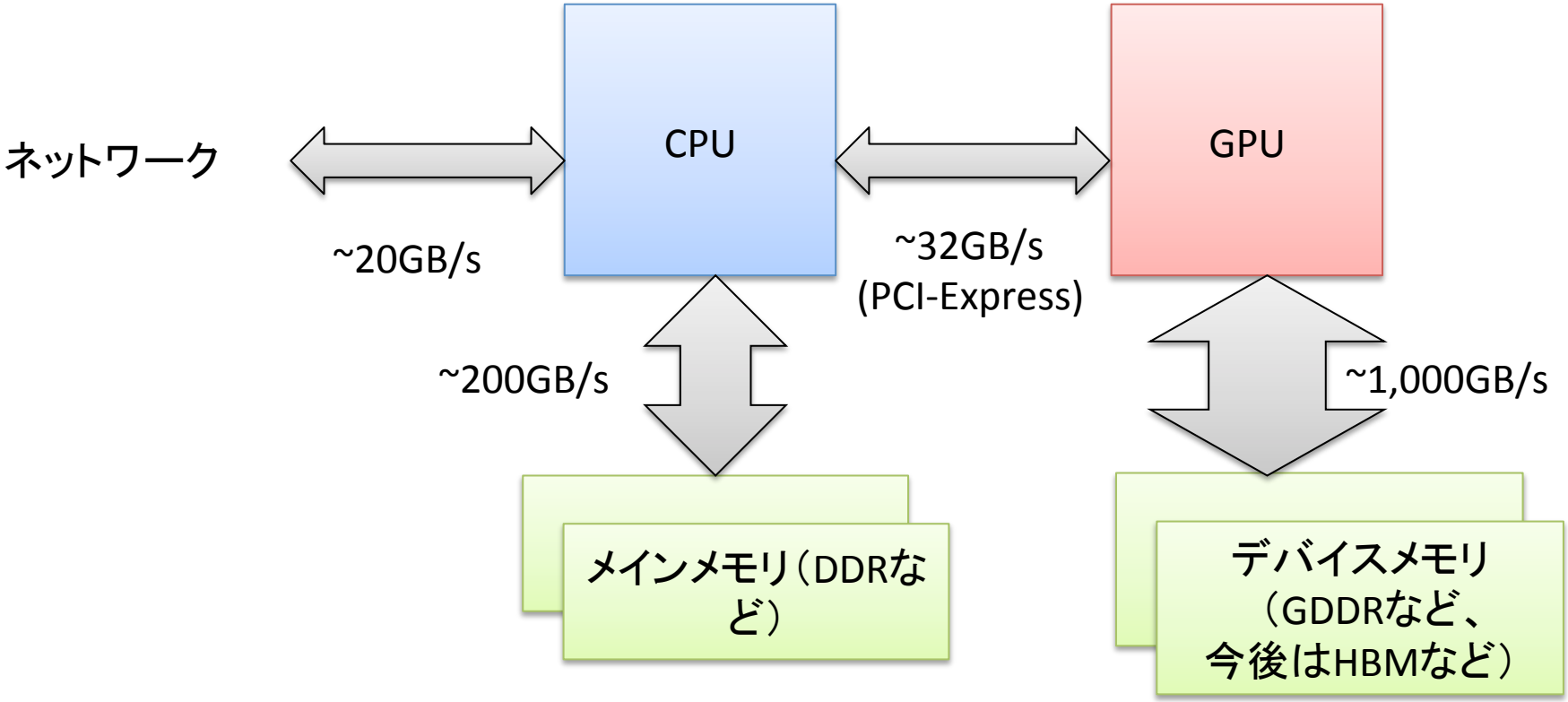
OpenMPにおけるFork-Joinのイメージ



OpenACCにおけるFork-Joinのイメージ



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

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
```

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
```

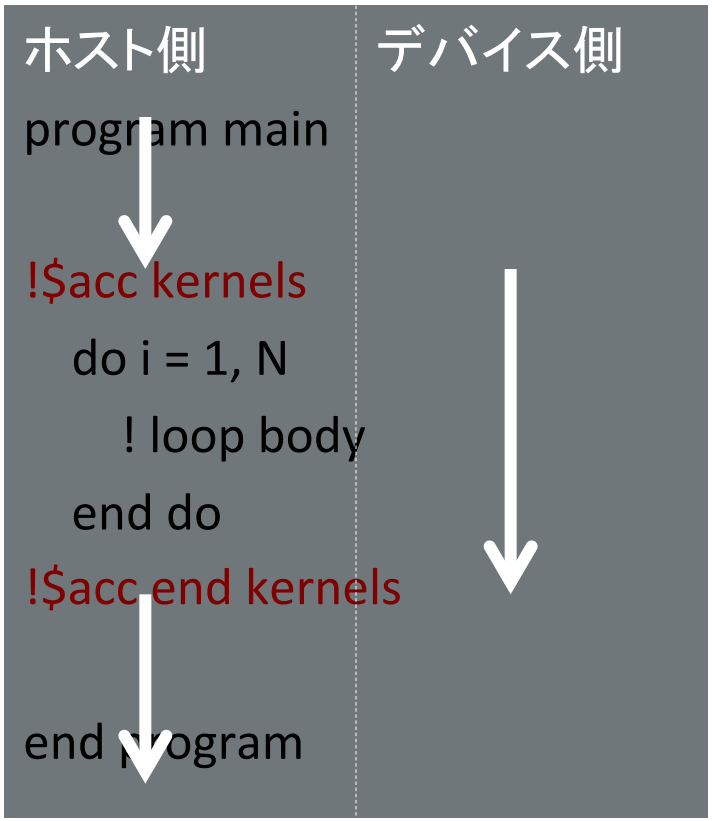
parallel

```
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

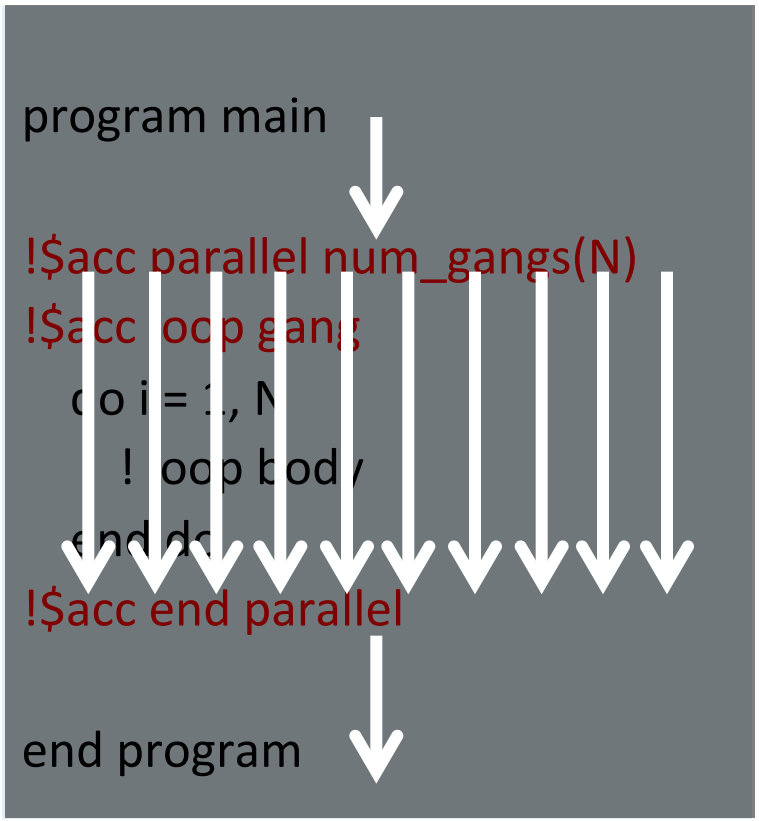
- ホスト-デバイスを意識するのがkernels
- 並列実行領域であることを意識するのがparallel

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…
  - データディレクティブのclauseが使える (詳細は後述)

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(.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

parallel

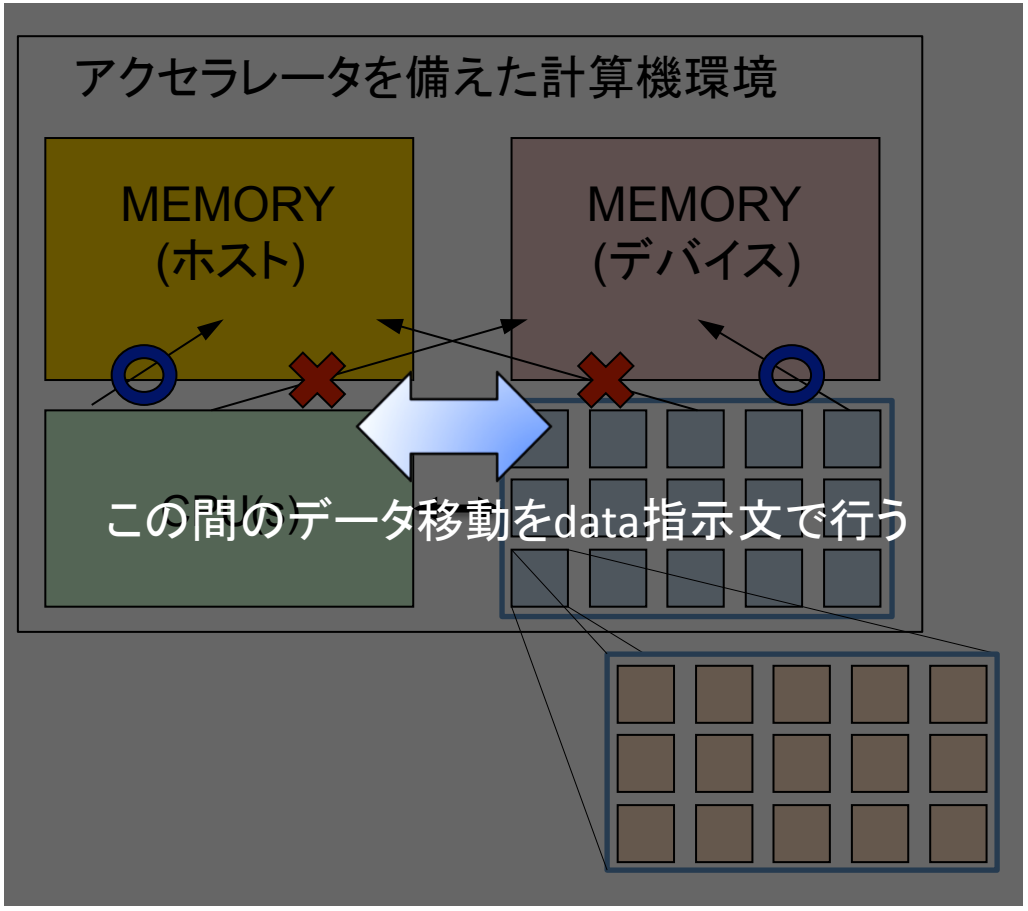
- async
- wait
- device_type
- if
- default(none)
- copy…
- num_gangs
- num_workers
- vector_length
- reduction
- private
- firstprivate

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。

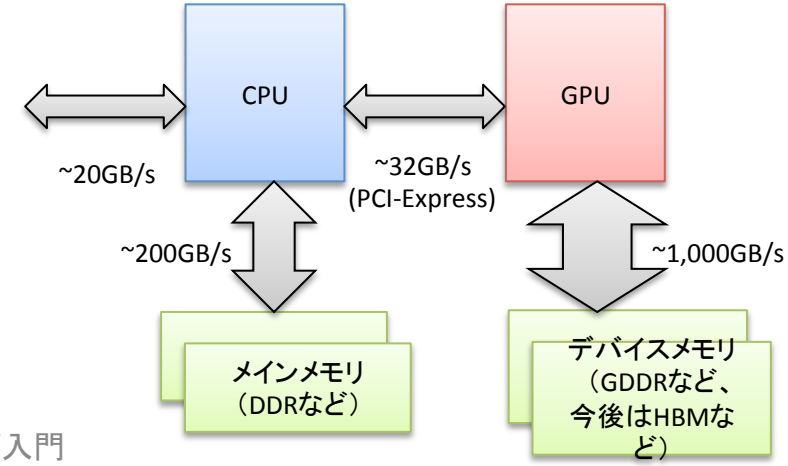
デバイス上で扱われるべきデータについて

- 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

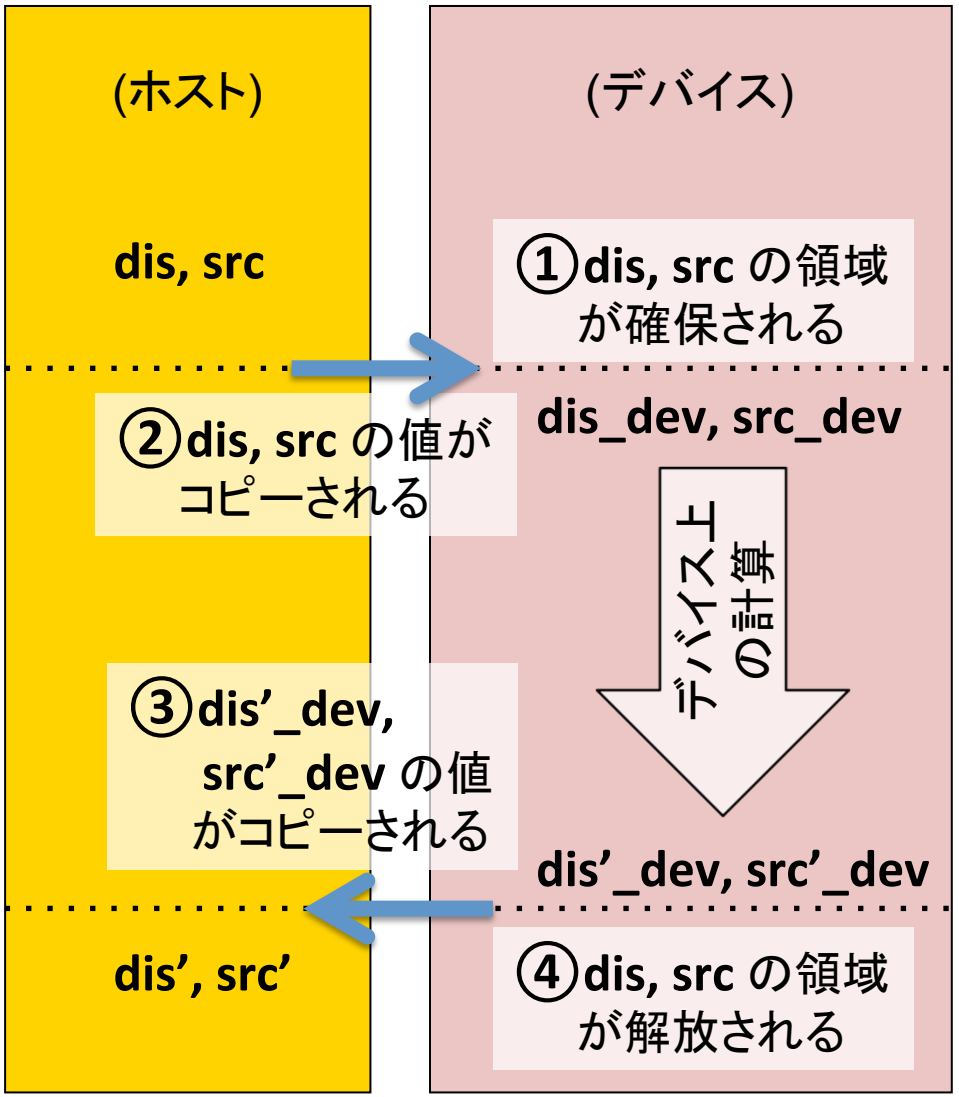
```
void copy(float *dis, float *src) {
  int i;
#pragma acc data copy(src[0:N] \
    dis[0:N])
  {
#pragam acc kernels
      for(i = 0;i < N;i++){
         dis[i] = src[i];
      }
  }
}
```

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

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

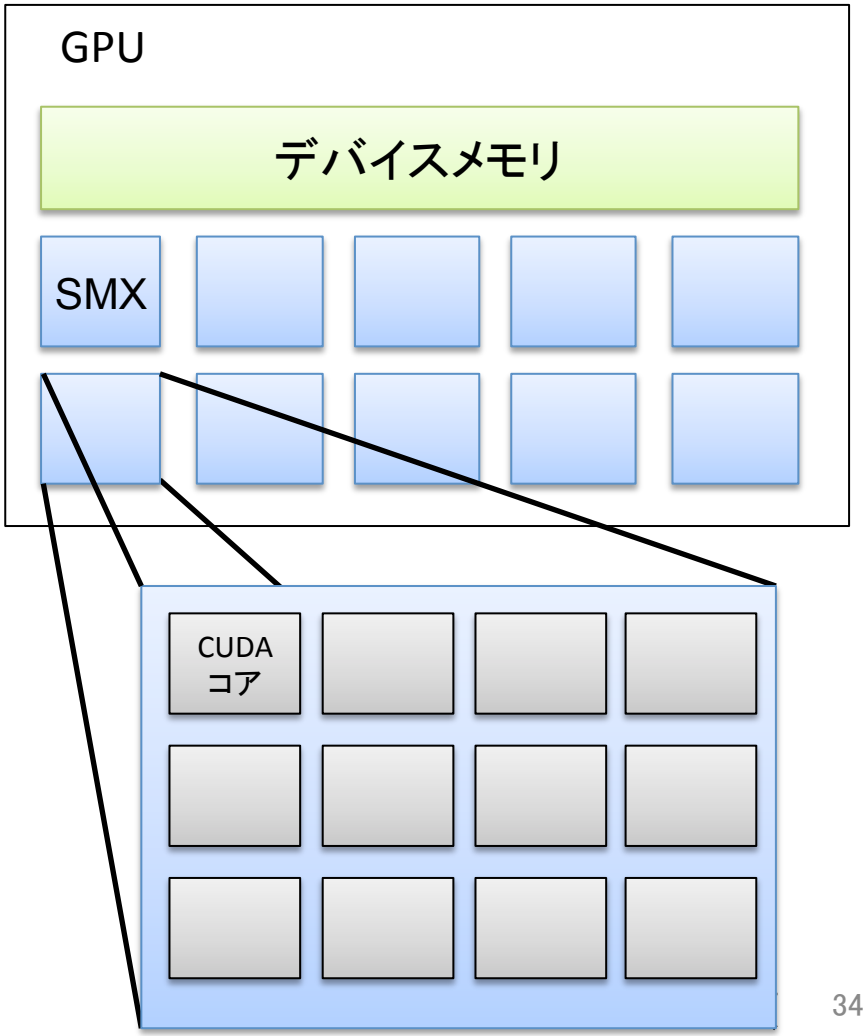
GPUでの行列積の例

```
!$acc kernels
!$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

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・実行
  - PGIコンパイラのメッセージの読み方
  - PGI_ACC_TIMEによるGPU実行の確認
- OpenACC プログラムの作成
  - 行列積、拡散方程式
- OpenACC プログラムの最適化
  - NVIDIA visual profiler の使い方など

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知るため
    - 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るのか知るため
  - メッセージを見て、指示文・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
    - 例： pgfortran -acc -Minfo=accel

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

```
subroutine acc_kernels()
  double precision :: A(N,N), B(N,N)
  double precision :: alpha = 1.0
  integer :: i, j
  A(:,:) = 1.0; B(:,:) = 0.0
  !$acc kernels
  do j = 1, N
    do i = 1, N
      B(i,j) = alpha * A(i,j)
    end do
  end do
  !$acc end kernels
end subroutine acc_kernels
```

コンパイラメッセージ(fortran)

```
pgfortran  -O3 -acc -Minfo=accel -ta=tesla,cc20 -Mpreprocess acc_compute.f90 -o acc_compute
acc_kernels:
    15, Generating copyin(a(:,:))
        Generating copyout(b(:,:))
    16, Loop is parallelizable
    17, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
        16, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
        17, !$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

```
subroutine acc_kernels()
  double precision :: A(N,N), B(N,N)
  double precision :: alpha = 1.0
  integer :: i, j
  A(:,:) = 1.0; B(:,:) = 0.0
  !$acc kernels
  do j = 1, N
    do i = 1, N
      B(i,j) = alpha * A(i,j)
    end do
  end do
  !$acc end kernels
end subroutine acc_kernels
```

ソースコード

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,cc20 -Mpreprocess acc_compute.f90 -o acc_compute
acc_kernels:
    15, Generating copyin(a(:,:))
        Generating copyout(b(:,:))
    16, Loop is parallelizable
    17, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
        16, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
        17, !$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

16, 17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)

```
void acc_kernels(double *A, double *B){
  double alpha = 1.0;
  int i,j;
  / * A と B 初期化 */
#pragma acc kernels
  for(j = 0;j < N;j++){
    for(i = 0;i < N;i++){
      B[i+j*N] = alpha * A[i+j*N];
    }
  }
}
```

コンパイラメッセージ(C)

```
pgcc -O3 -acc -Minfo=accel -ta=tesla,cc20  acc_compute.c -o acc_compute
acc_kernels:
    50, Generating copyin(A[:1000000])
        Generating copy(B[:1000000])
    51, Complex loop carried dependence of A-> prevents parallelization
        Loop carried dependence of B-> prevents parallelization
        Loop carried backward dependence of B-> prevents vectorization
        Complex loop carried dependence of B-> prevents parallelization
        Accelerator scalar kernel generated
        Accelerator kernel generated
        51, #pragma acc loop seq
        52, #pragma acc loop seq
    52, Complex loop carried dependence of A->,B-> 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)

```
void acc_kernels(double *restrict A, double
*restrict B){
  double alpha = 1.0;
  int i,j;
  / * A と B 初期化 */
#pragma acc kernels
  for(j = 0;j < N;j++){
    for(i = 0;i < N;i++){
      B[i+j*N] = alpha * A[i+j*N];
    }
  }
}
```

コンパイラメッセージ(C)

```
pgcc -O3 -acc -Minfo=accel -ta=tesla,cc20  acc_compute.c -o acc_compute
acc_kernels_restrict:
    69, Generating copyin(A[:1000000])
        Generating copy(B[:1000000])
    71, Loop carried dependence of B-> prevents parallelization
        Loop carried backward dependence of B-> prevents vectorization
    73, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
        71, #pragma acc loop seq
        73, #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、インデックスi+j*N が実は同じ場所を更新する可能性を考慮し、71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)

```
void acc_kernels(double *restrict A, double
*restrict B){
  double alpha = 1.0;
  int i,j;
  / * A と B 初期化 */
#pragma acc kernels
#pragma acc loop independent
  for(j = 0;j < N;j++){
#pragma acc loop independent
    for(i = 0;i < N;i++){
      B[i+j*N] = alpha * A[i+j*N];
    }
  }
}
```

コンパイラメッセージ(C)

```
pgcc -O3 -acc -Minfo=accel -ta=tesla,cc20  acc_compute.c -o acc_compute
acc_kernels_independent:
    90, Generating copyin(A[:1000000])
        Generating copy(B[:1000000])
    92, Loop is parallelizable
    94, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
        92, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
        94, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

ようやく並列化

実習1

- 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)
- 今回はHA-PACS 環境なので、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

実習2

- 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

- 行列積のOpenACC化
- matmul.f90 または matmul.c を用いる。
  - 他のサンプルプログラムを参考に、acc_matmul ルーチンに OpenACC指示文を加えてください。

出力例

```
====== OpenACC matmul program ======

 1024 * 1024 matrix
check result...OK
elapsed time[sec] :    0.07179
FLOPS[GFlops]     :   29.88292
```

実習4

- 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化されているので、実行してみてください

出力例

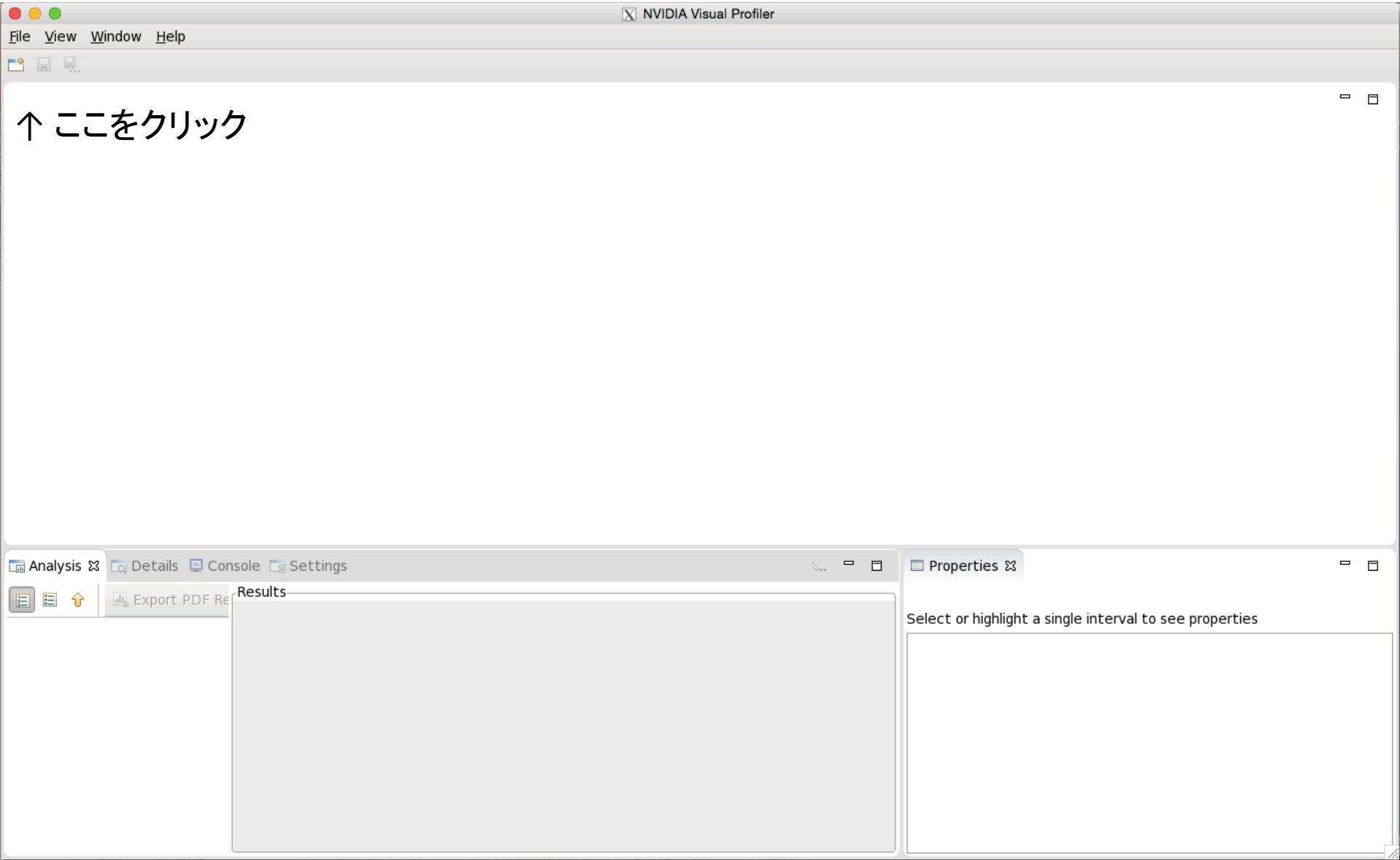
```
====== OpenACC diffusion program ======

 128 *  128 *  128
accuracy       :  0.444E-05
elapsed time[sec] :   24.79527
throughput[GB/sec]:    2.21664
FLOPS[GFLOPS]    :    1.80102
```

- 性能の確認
  - NVIDIA Visual Profilerを用いる
  - 性能の最適化を行う
    - 特にデータ転送に注意

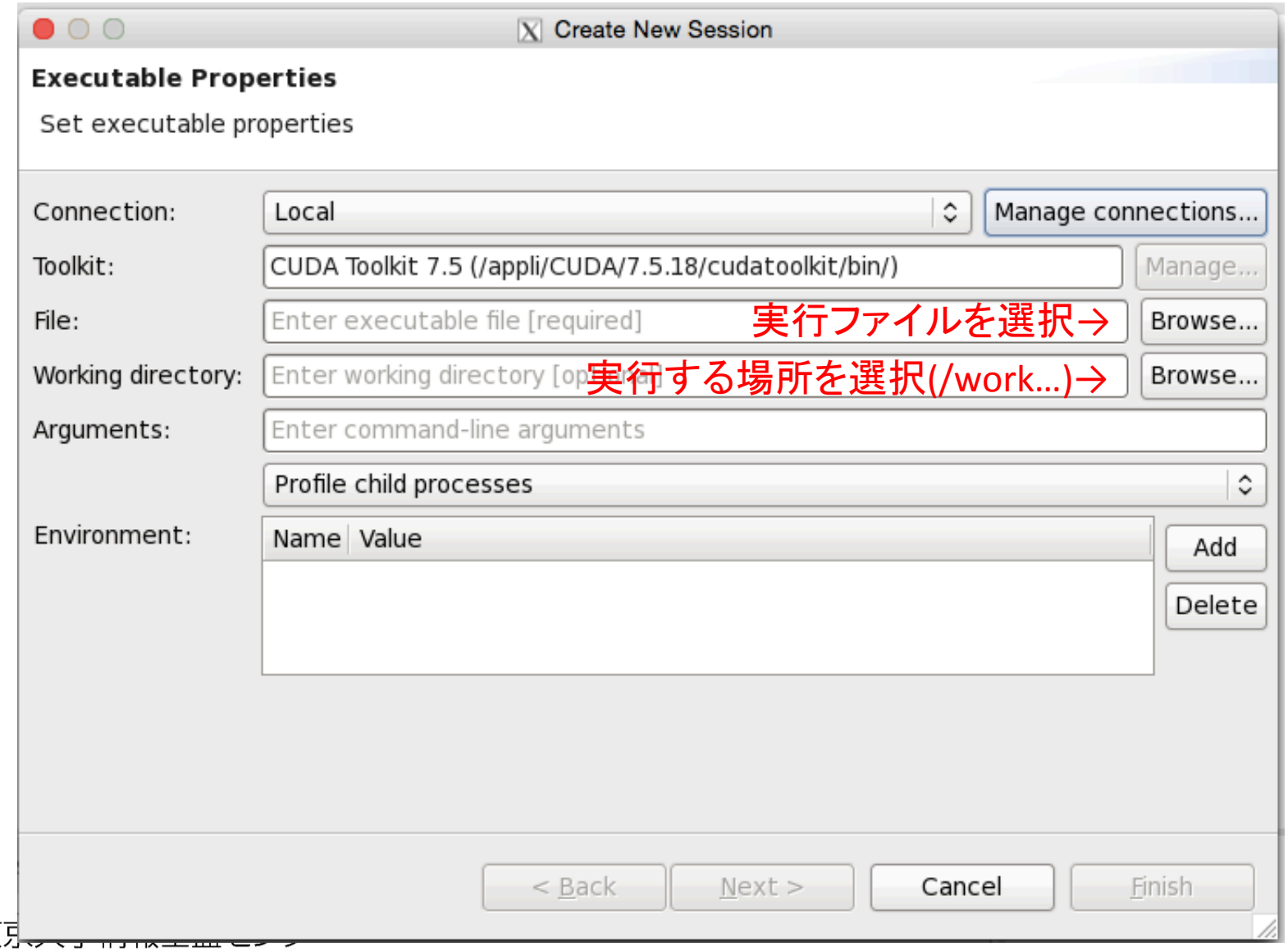
NVIDIA Visual Profiler を用いた最適化

- NVIDIA Visual Profiler を起動
  - インタラクティブジョブを実行
    - $ ./interactive-job-**x**.sh # HA-PACSへのssh時に-Yをつけていないと失敗する
  - 起動
    - $ nvvp

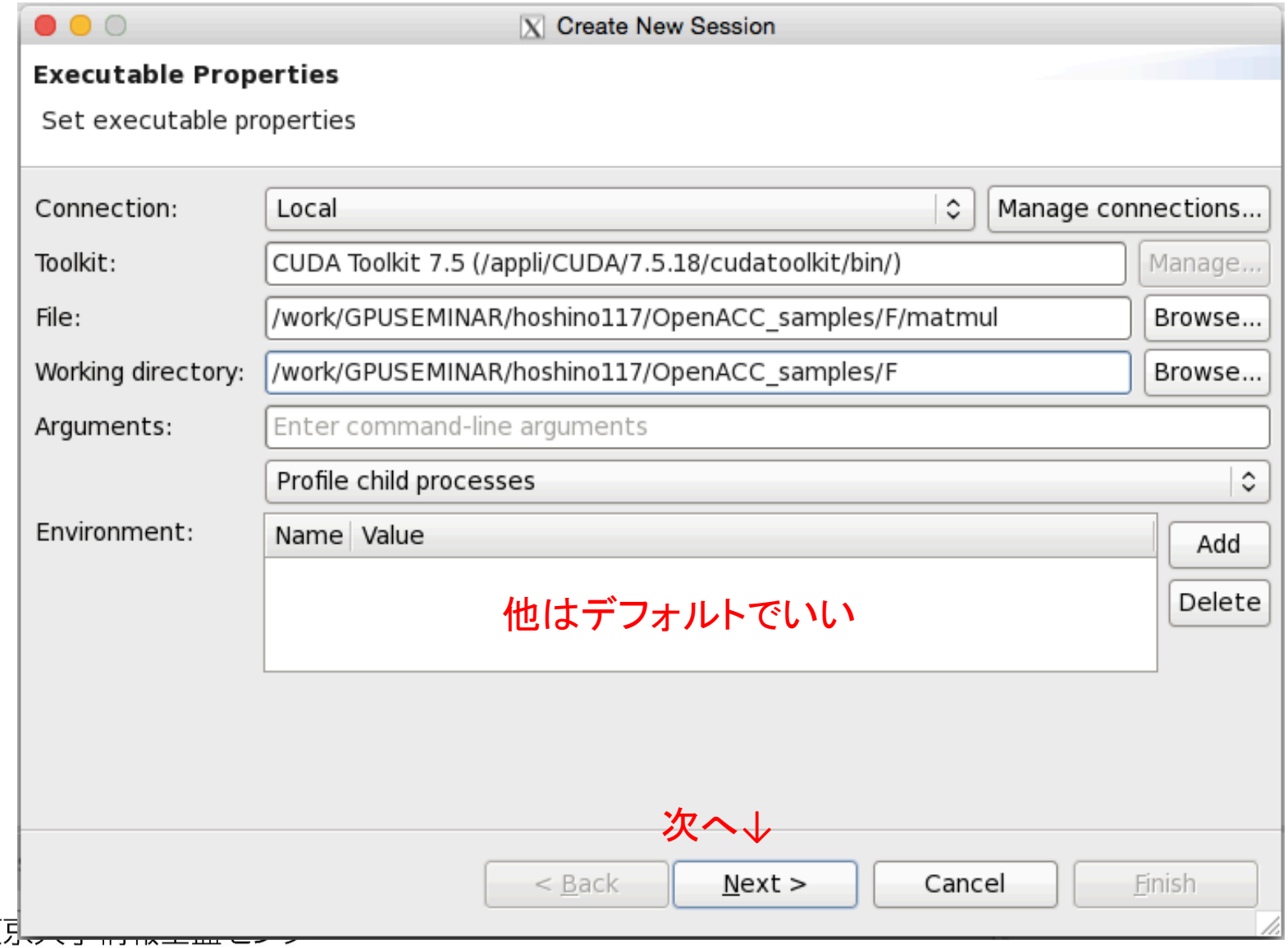
NVIDIA Visual Profiler を用いた最適化



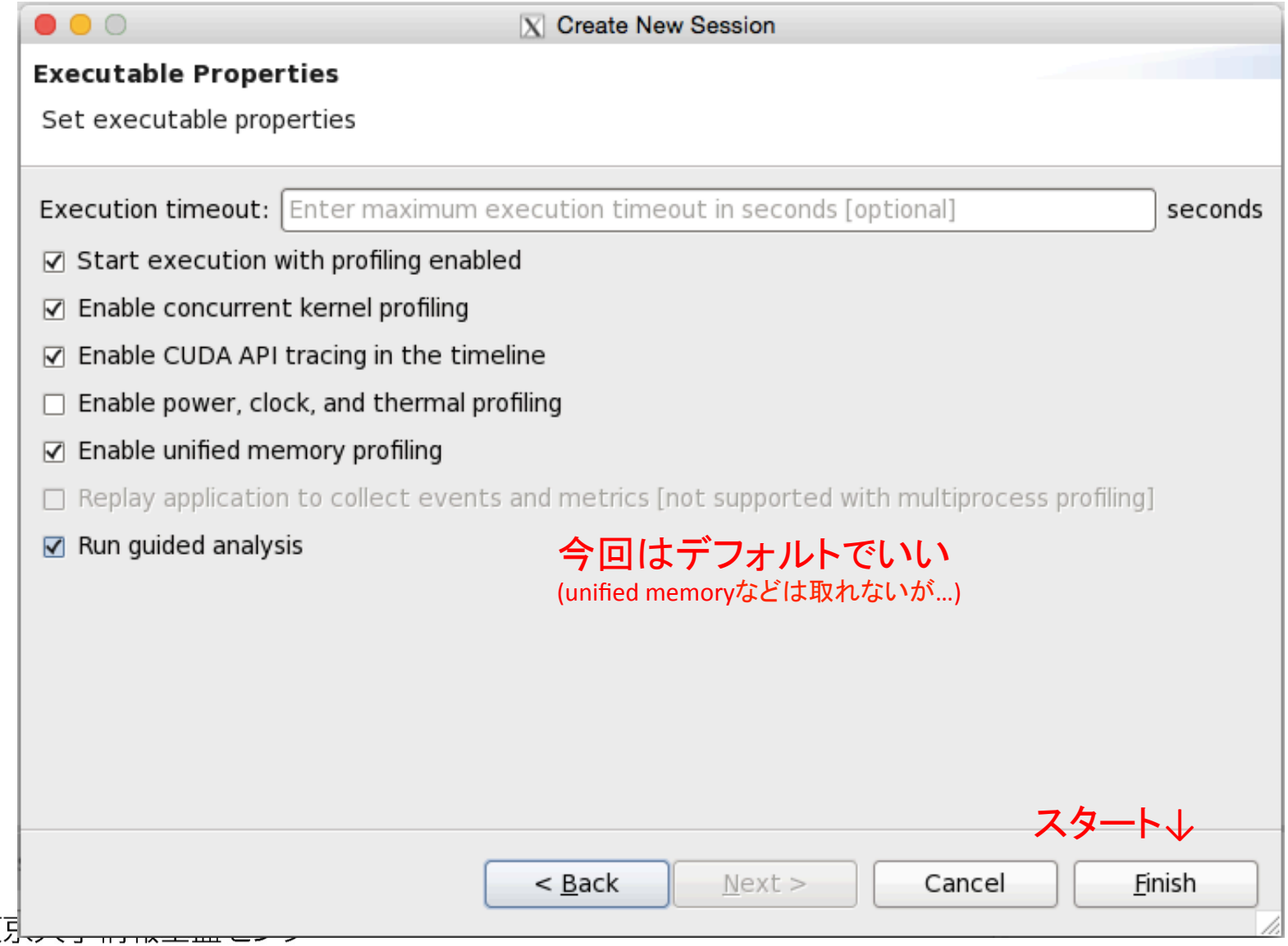
NVIDIA Visual Profiler を用いた最適化



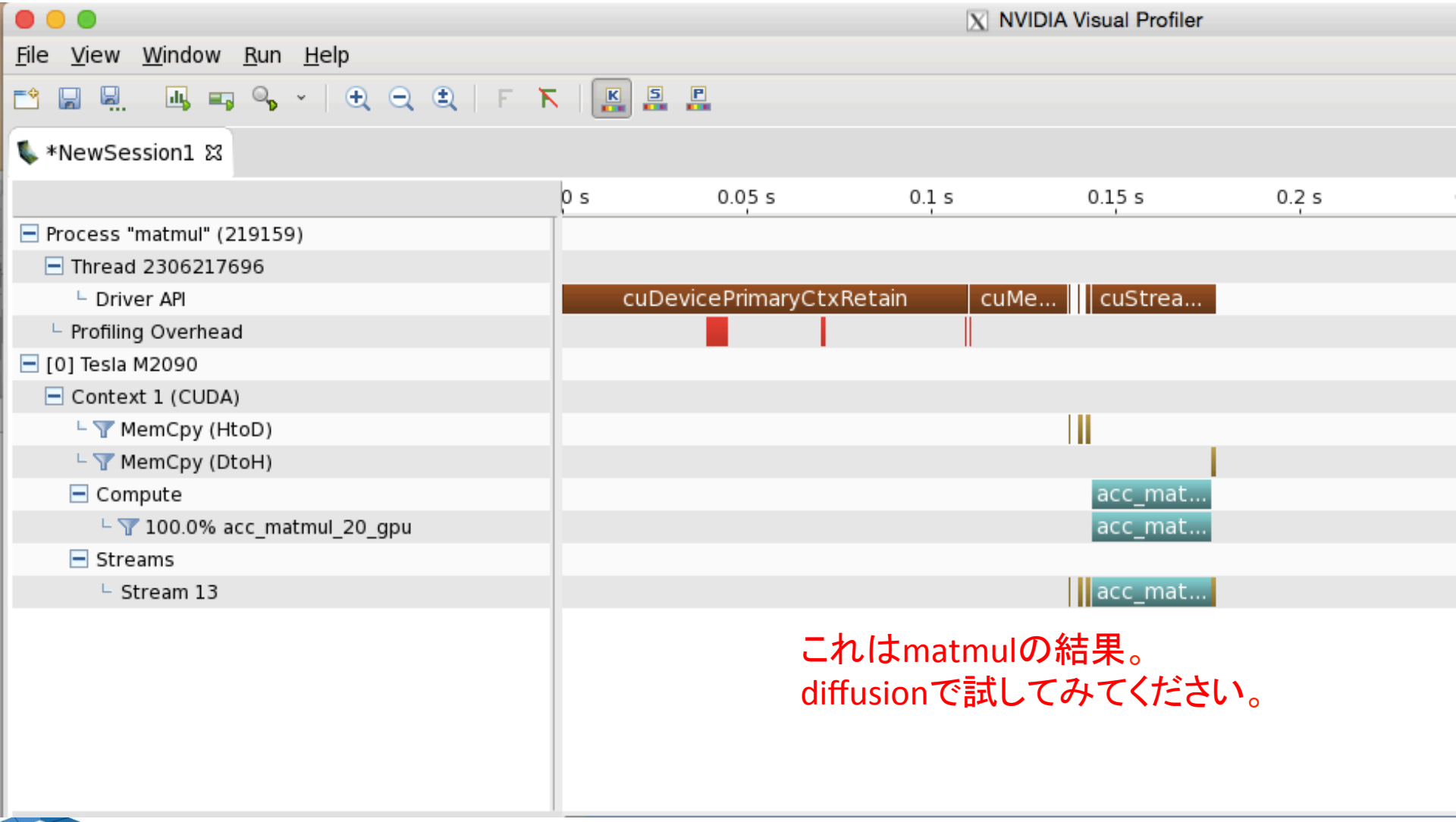
NVIDIA Visual Profiler を用いた最適化



NVIDIA Visual Profiler を用いた最適化

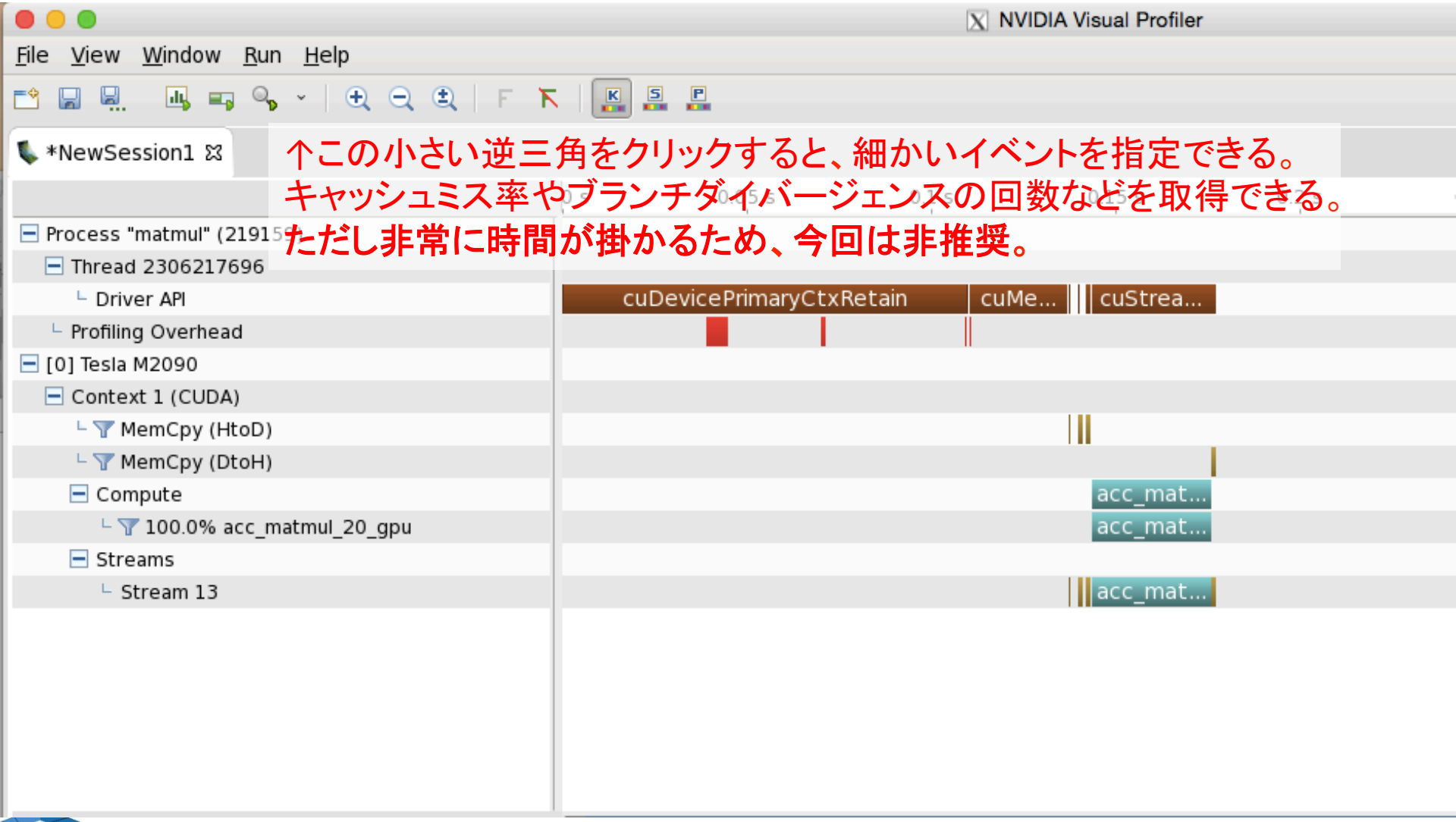


NVIDIA Visual Profilerを用いた最適化

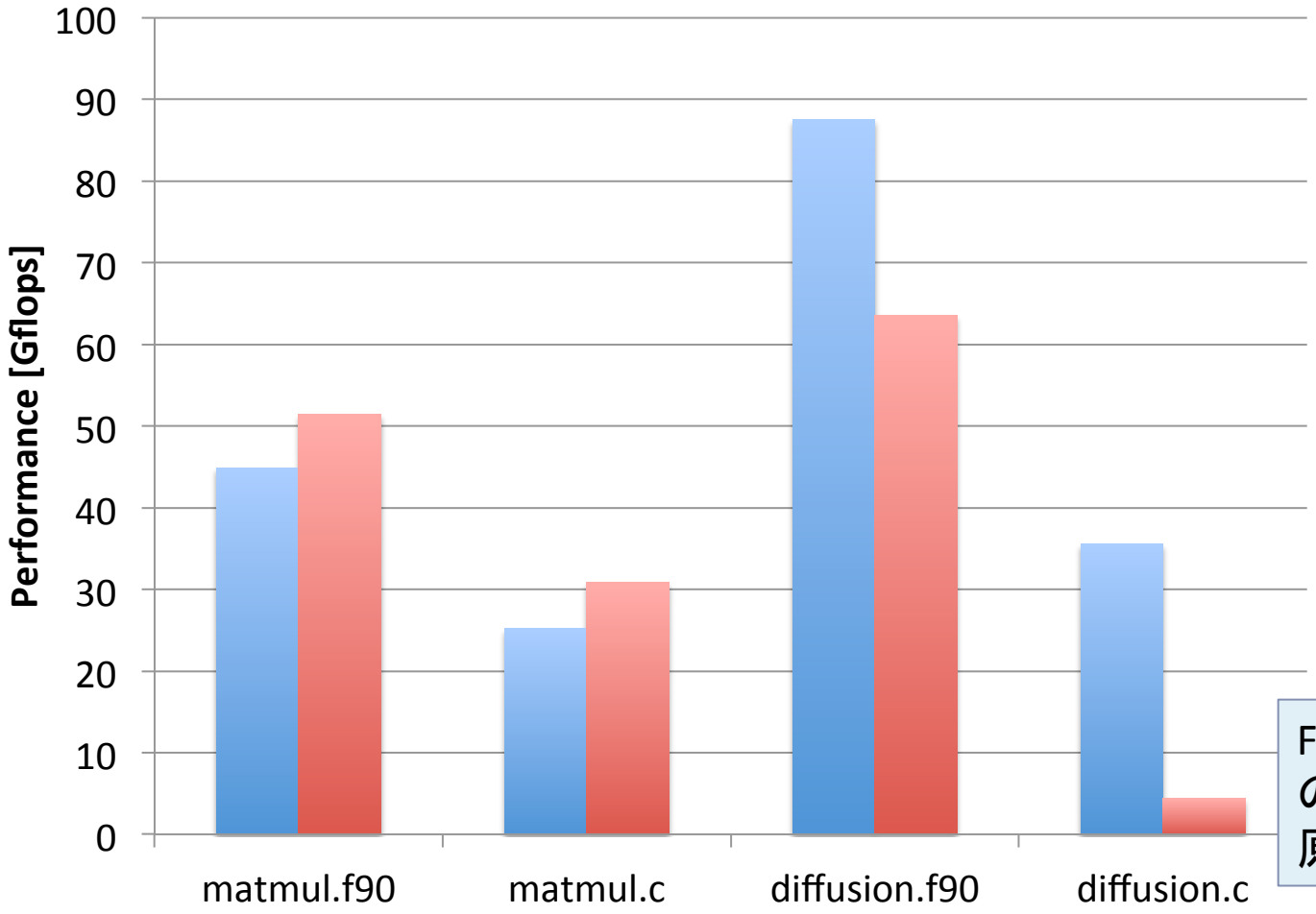


これはmatmulの結果。
diffusionで試してみてください。

NVIDIA Visual Profiler をさらに使いこなす



追加資料：diffusion.c が遅い？



FermiでのdiffusionのcのCの実行が非常に遅く、原因は調査中