

GPUプログラミング実践 (OpenACC)

プロメテック・ソフトウェア株式会社 長谷川 雄太 Version 2025.07.25

本資料は以下の資料をベースにプロメテック・ソフトウェア株式会社が本講習会に適した情報となるよう加筆修正等を行ったものです。

OpenACC Official Training Materials

- Below slides are released by NVIDIA Corporation under CC BY 4.0
- Slides: https://drive.google.com/open?id=1d_elwIRfScHxfJu6pnR28JrV3cMIwkIL
- CC BY 4.0: <https://creativecommons.org/licenses/by/4.0/deed.ja>

自己紹介

長谷川 雄太 (Hasegawa Yuta)

➤ プロメテック・ソフトウェア株式会社

AI/HPCプラットフォーム事業開発本部 シニアエンジニア

➤ 専門: CFD (数値流体力学) , HPC

GPU向けCFDコードの開発・研究 (CUDA, MPI, cuBLAS, cuSOLVER等)

デジタルツインソリューション開発

NVIDIA HPC SDKコンパイラサポート

➤ 経歴

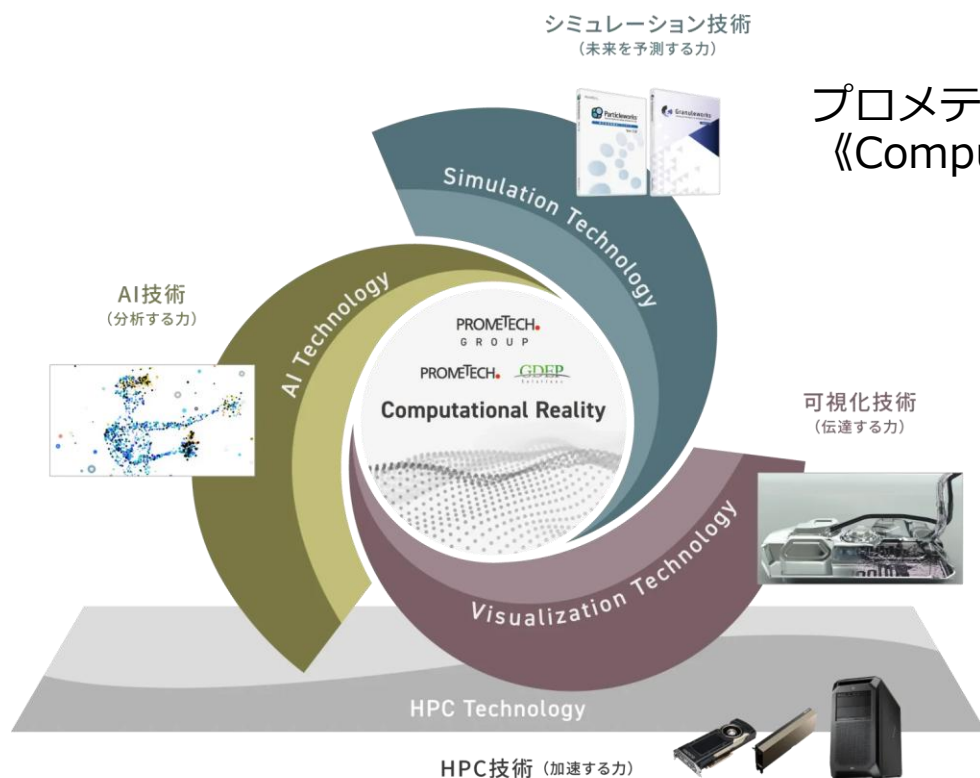
～2019年3月 東京工業大学 (現 東京科学大学) 工学院機械系 博士課程修了

～2024年9月 国立研究開発法人日本原子力研究開発機構

GPUスパコン向けCFD・DA (データ同化) コードの開発・研究に従事

2024年10月～ 現職

プロメテックグループ紹介



プロメテックグループはグループ各社の専門性を結集させて、
《Computational Reality》を社会に実装する挑戦を続けています。

プロメテック・ソフトウェア株式会社

計算科学技術をコアとするソフトウェア製品の開発を担当

GDEPソリューションズ株式会社

高度な科学計算を実行するハイパフォーマンスコンピューティング（HPC）の基盤構築を担当

事業内容

- 流体・粉体解析ソフトウェアの開発・販売・サポート



- 解析コンサルティングサービス
- 可視化・映像制作サービス
- NVIDIA HPCコンパイラサポートサービス

ソフテック社より継承した旧PGIコンパイラの技術活用によるコンパイラのサポート

前回の入門編講習会の内容

OpenACCの入門的な講習

1. GPUのプログラミングの概要
 2. OpenACCを使ったプログラムの並列化
 3. CPU-GPU間のデータ転送の最適化
 4. ループの並列化
 5. OpenACCによる並列化の進め方
- ✓ FortranまたはC言語の知識を前提とし、講習はFortranベース
 - ✓ OpenACCを用いるターゲットはNVIDIA CUDA GPU (NVIDIA A100)

今回の実践編講習会の内容

OpenACCの実践的な講習

1. ループの最適化
 2. 非同期処理
 3. CUDAとの連携 (Interoperability)
 4. マルチGPU計算の基礎
 5. 開発における実装方針のススメ
- ✓ 前回入門編の知識をベースに話を進める
 - ✓ FortranまたはC言語の知識を前提とし、講習はFortranベース
 - ✓ OpenACCを用いるターゲットはNVIDIA CUDA GPU (NVIDIA A100)

1. ループの最適化

ループの最適化はなぜ重要か

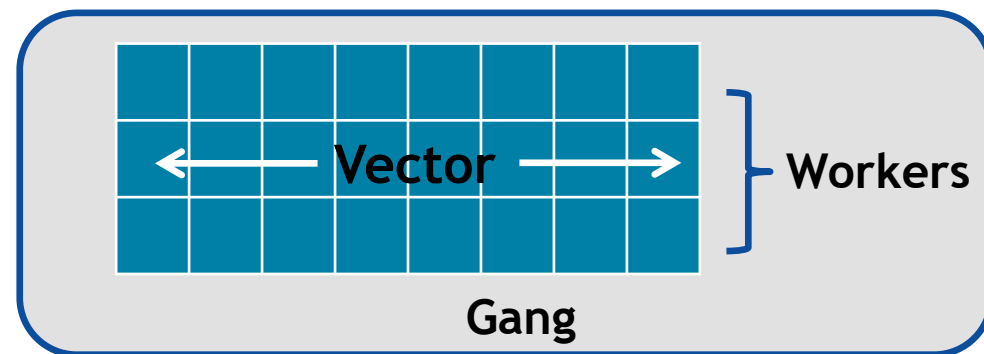
ループの並列化

- 多くのソフトウェアの実行時間は大部分がループ処理に費やされる
- 各ループは定型化されたものもあるが、非常に様々な方法で記述される
- OpenACCのループ最適化を使用することで、コード中の最も時間がかかる箇所を高速化することができる

ループの最適化

Gang - Worker - Vector について

- OpenACCにおける、複数階層の並列性のこと
Gang -> Worker -> Vector
- 多重ループ(階層的構造を持つ)を並列化する場合に、効率よく並列化を定義できる
- OpenACCは様々なハードウェアに適用できるようにGang / Worker / Vectorモデルを定義する
- (今回はNVIDIA GPUを想定して説明)



Gang - Worker - Vector について

- ループを並列化するときの最高位の並列性を **Gang-level parallelism** と呼ぶ

→ コンパイラは、典型的には最外(Outermost)ループを Gang として並列化する

- kernels/parallelディレクティブを指定すると複数のGangが生成され、ループがGangによって分散実行される
- 多くのアーキテクチャではGangは完全に独立、もしくはプライベートなメモリを持っている



Gang - Worker - Vector について

- 右図のコードでは最外ループにGang節を適用している
- 最外ループに任意のGangが割り当てられ、並行して互いに独立に計算を実行する
- kernels/parallelディレクティブによって並列領域が生成される度に、Gangが生成される
- ユーザーはいくつのGangを生成するかを指定することも可能
- 指定せずにコンパイラに任せた方が無難



```
!$acc parallel loop gang
do i = 1, N
  do j = 1, M
    < loop code >
  end do
end do
```


Gang - Worker - **Vector** について

- 最下層の並列性を**Vector-level parallelism**と呼ぶ

→ コンパイラは、典型的には最内(Innermost)ループを Vector として並列化する



- すべてのGangは最低でも**1つのVectorを持つ**
- また各Vectorは複数のデータ要素に対し1つの命令を実行可能

Gang - Worker - **Vector** について

- 右図のコードでは内部ループがVector並列化される
- 二重ループの並列化は互いに並行して実行されることになる
- Vector loopの内側にあるループはこれ以上並列化されず、**逐次処理が行われる**



```
!$acc parallel loop gang
do i = 1, N
  !$acc loop vector
  do j = 1, M
    < loop code >
  end do
end do
```

Gang - Worker - Vector について

- Worker節はGang内に複数のVectorを持つための機能
- Workerの用途は、主に1つの大きなVectorを複数の小さなVectorに分割すること
→ 後ほど具体例
- 内側ループが非常に小さく、大きなVectorを持つことにメリットがない場合に有効なことがある



Gang - Worker - Vector について

- 右図のコードは、Gang/Worker levelの並列性を最外ループに適用している
- 先ほどのコードとの違いは、最内ループを実行するVectorを小さくできること
→ 後ほど具体例
- 内側のループが比較的小さい場合に性能向上を狙える



```
!$acc parallel loop gang worker
do i = 1, N
  !$acc loop vector
  do j = 1, M
    < loop code >
  end do
end do
```

parallelディレクティブの場合

- parallelディレクティブの場合 **num_gangs(N)**, **num_workers(M)**, **vector_length(Q)** で Gang, Worker, Vectorの数を指定する
- それぞれがどのループに属するかを **gang**, **worker**, **vector** で指定する

```
!$acc parallel num_gangs(2) num_workers(2) vector_length(32)
  !$acc loop gang worker
    do x = 1, 4
      !$acc loop vector
        do y = 1, 32
          array(x,y) = array(x,y) + 1
        end do
      end do
    end do
```

kernelsディレクティブの場合

- kernelsディレクティブを使用する場合は処理が簡単化される
- **gang(N), worker(M), vector(Q)** でループ位置と数を同時に指定可能
- parallelディレクティブを同じ方法で指定することも可能
- 数を指定しない場合はコンパイラが自動的に決定する

```
!$acc kernels loop gang(2) worker(2)
do x = 1, 4
  !$acc loop vector(32)
  do y = 1, 32
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```

kernelsディレクティブの場合：複数の多重ループ

- kernelsディレクティブを使用する場合は処理が簡単化される
- **gang(N), worker(M), vector(Q)** でループ位置と数を同時に指定可能
- parallelディレクティブを同じ方法で指定することも可能
- 数を指定しない場合はコンパイラが自動的に決定する
- ループごとにgang, worker, vectorの値は変更可能

```
!$acc kernels
```

```
!$acc loop gang(2) worker(2)
```

```
do x = 1, 4
```

```
!$acc loop vector(32)
```

```
do y = 1, 32
```

```
array(x,y) = array(x,y) + 1
```

```
end do
```

```
end do
```

```
!$acc loop gang(4) worker(4)
```

```
do x = 1, 16
```

```
!$acc loop vector(16)
```

```
do y = 1, 16
```

```
array2(x,y) = array2(x,y) + 1
```

```
end do
```

```
end do
```

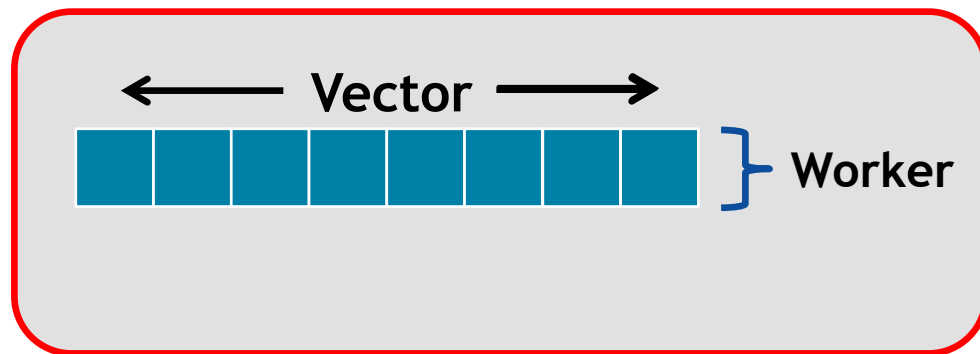
```
!$acc end kernels
```


Gang - Worker - Vector の動作例

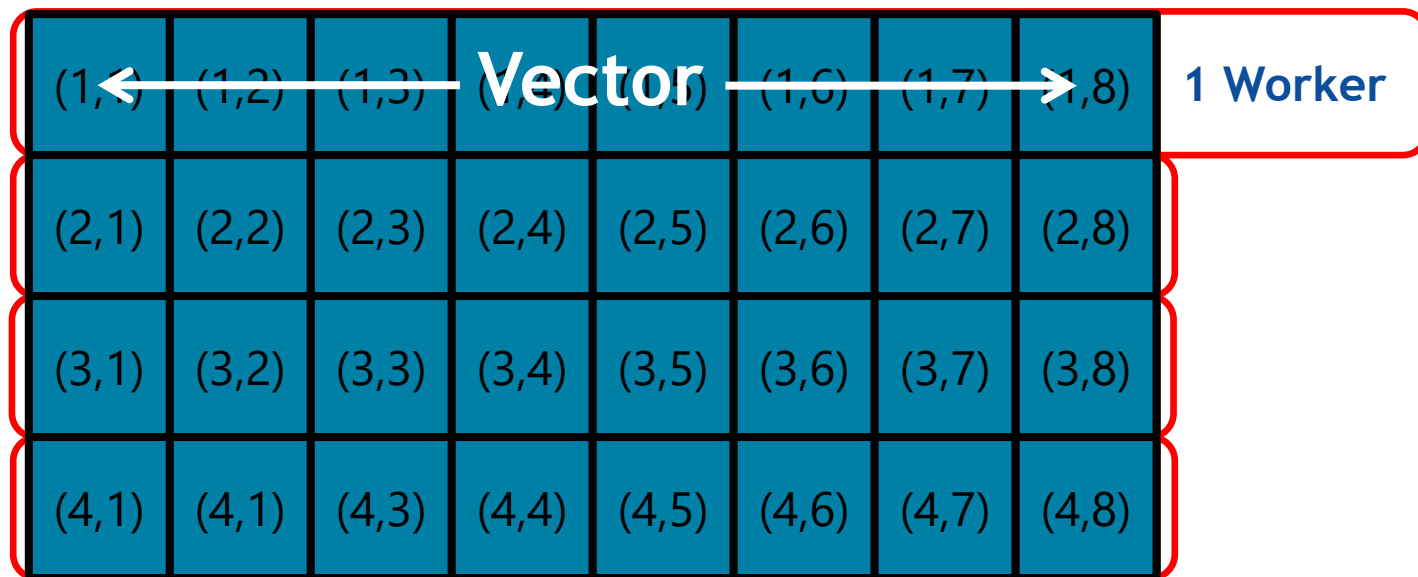
```
!$acc kernels loop gang worker(1)
do x = 1, 4
  !$acc loop vector(8)
  do y = 1, 8
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```

- 二重ループの並列化
- **Worker数を1, Vector長を8**と定義
- 生成する Gang の数を指定しないため**ループをカバーするのに十分な数の Gang をコンパイラが生成する**

Gang - Worker - Vector の動作例



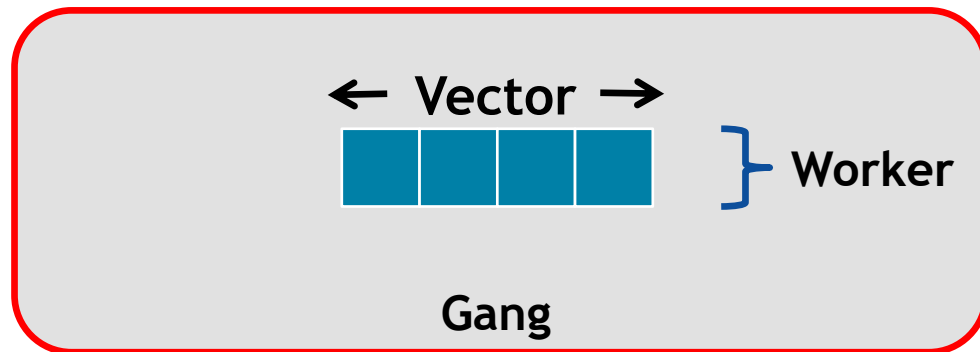
```
!$acc kernels loop gang worker(1)
do x = 1, 4
  !$acc loop vector(8)
  do y = 1, 8
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```



Gang

- ループサイズからコンパイラは理論的には**4 Gangs**を生成する

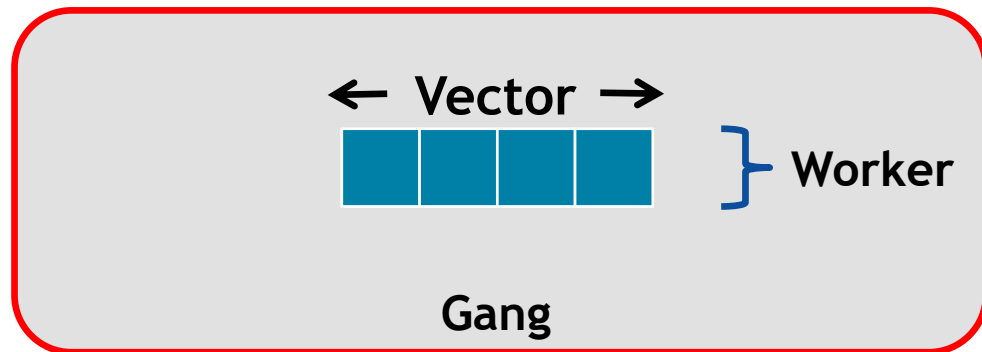
Gang - Worker - Vector の動作例: ベクトル長 < ループ長



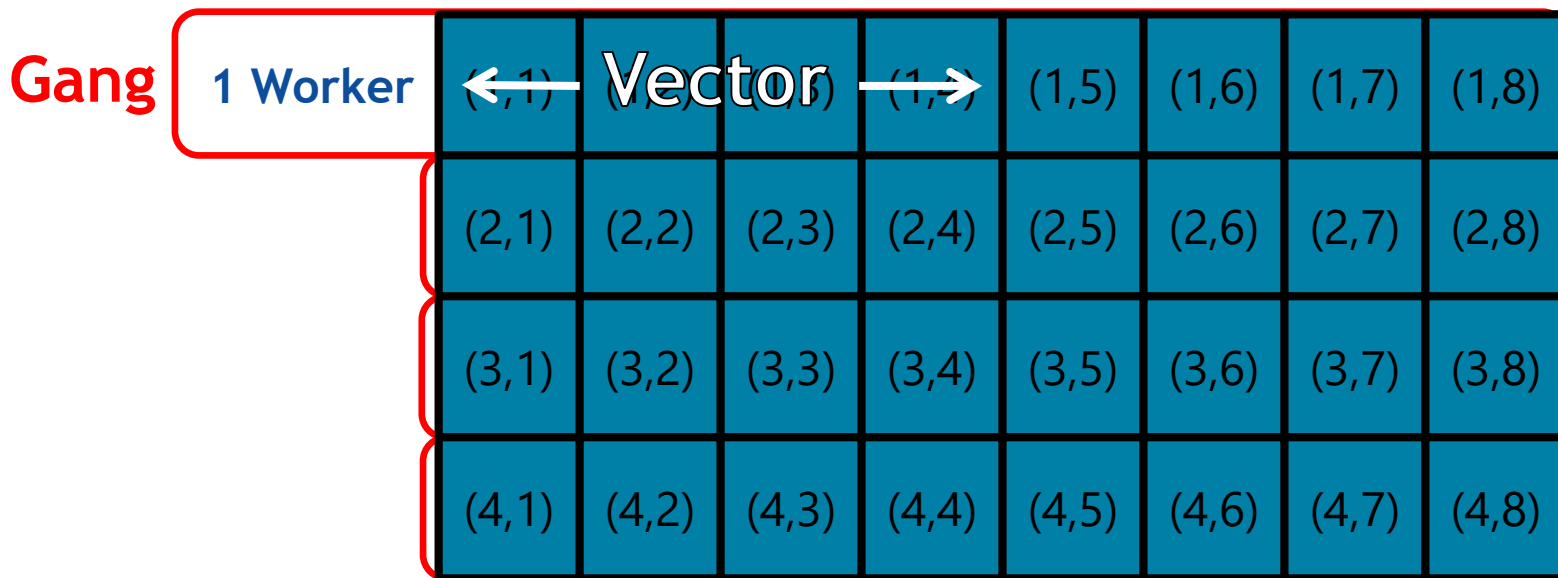
```
!$acc kernels loop gang worker(1)
do x = 1, 4
  !$acc loop vector(4)
  do y = 1, 8
    array(x,y) = array(x,y) + 1
  do
do
!$acc end kernels
```

- ベクトル長を短くした時の動作を
見てみる
→ 実際のプログラムで、ベクトル長よりも最
内ループ長が大きい場合と教えてください
- **Vector長を4**に減らす。その他は同
じ
- 最外ループの次元は変わらないの
で Gang の数は 4 のまま

Gang - Worker - Vector の動作例: ベクトル長 < ループ長

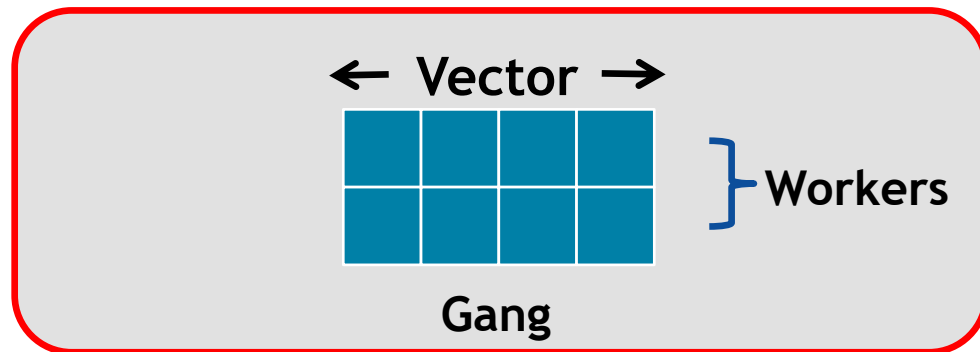


```
!$acc kernels loop gang worker(1)
do x = 1, 4
  !$acc loop vector(4)
  do y = 1, 8
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```



- 4つの Gang を生成したが各 Vector は2回のループ反復を行う
- より多くの Gang を生成するためには最外ループのサイズを大きくする必要がある

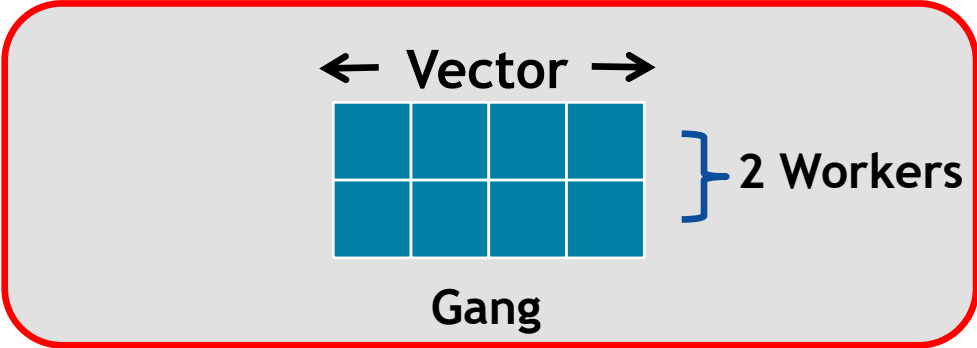
Gang - Worker - Vector の動作例: 複数ワーカー



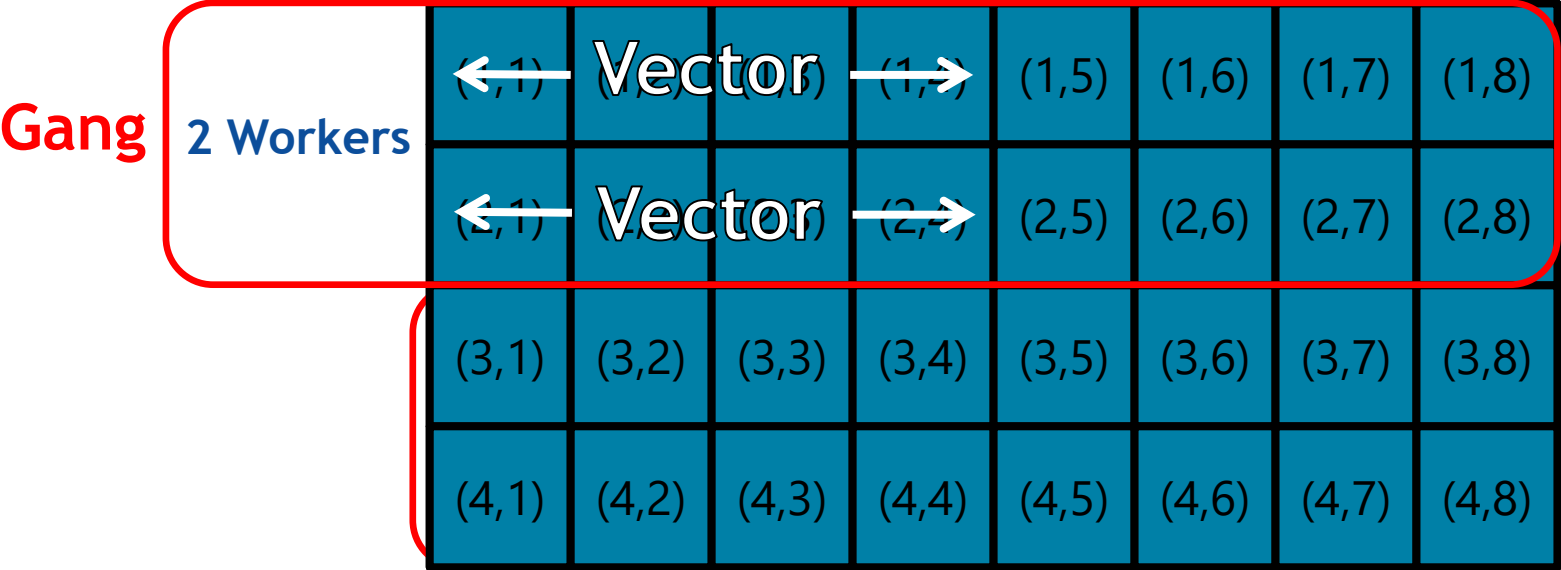
- **Worker数を2**に増やしてみる
- **Gang あたり Vector が2個**生成される。各 Vector のサイズは 4

```
!$acc kernels loop gang worker(2)
do x = 1, 4
  !$acc loop vector(4)
  do y = 1, 8
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```

Gang - Worker - Vector の動作例: 複数ワーカー

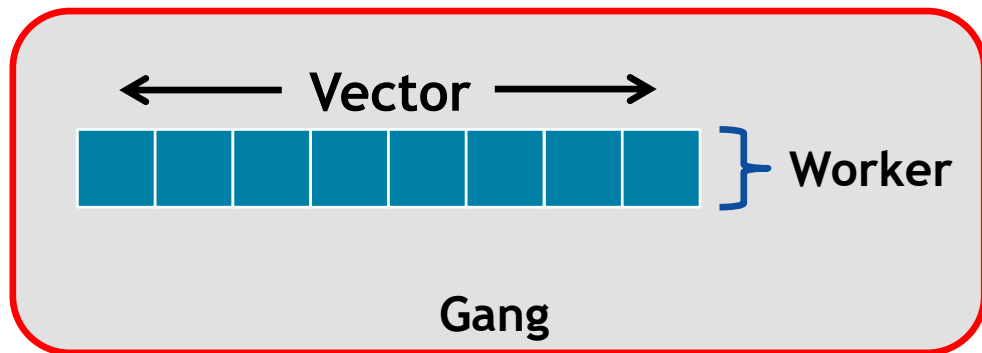


```
!$acc kernels loop gang worker(2)
do x = 1, 4
  !$acc loop vector(4)
  do y = 1, 8
    array(x,y) = array(x,y) + 1;
  end do
end do
!$acc end kernels
```



- Workerを増やした結果
Gangは2個しか生成
されなくなる

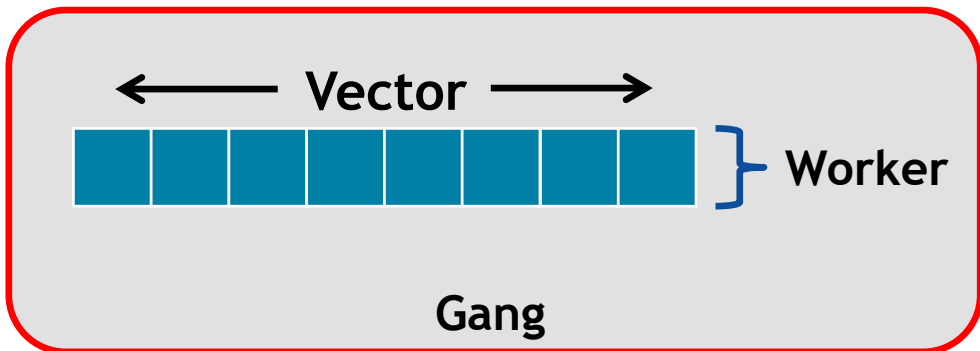
Gang - Worker - Vector の動作例: ベクトル長 > ループ長



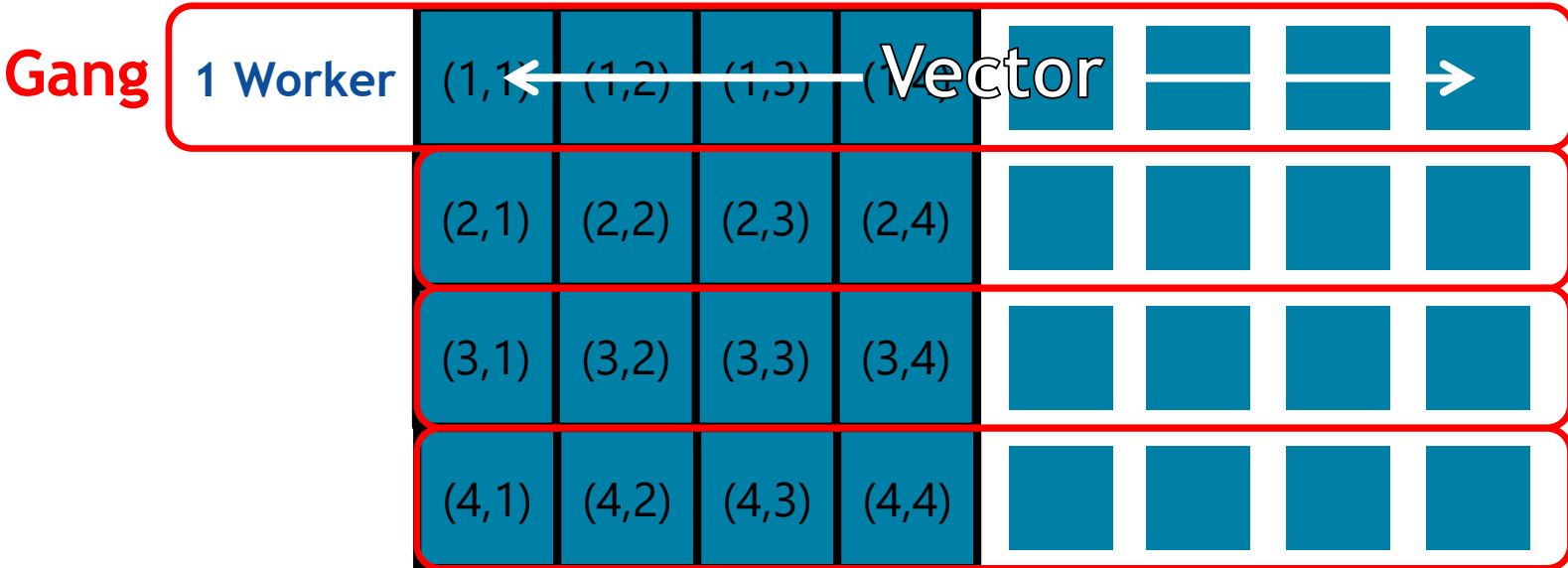
- 最内ループ長が 4 の場合
- ベクトル長 を 8 にすると？

```
!$acc kernels loop gang worker(1)
do x = 1, 4
  !$ acc loop vector(8)
  do y = 1, 4
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```

Gang - Worker - Vector の動作例: ベクトル長 > ループ長

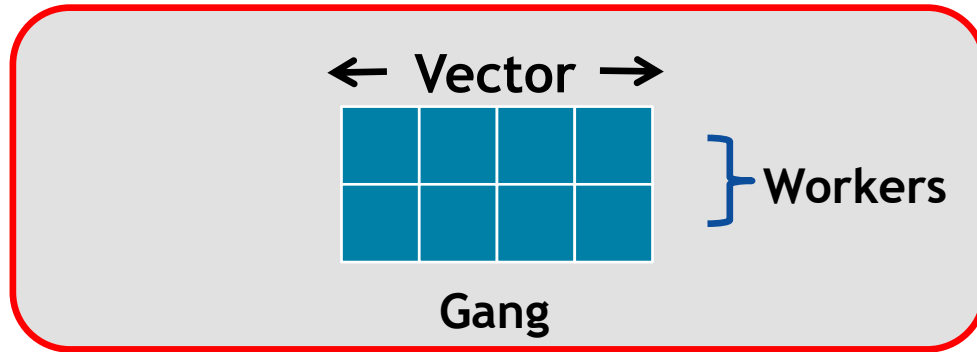


```
!$acc kernels loop gang worker(1)
do x = 1, 4
  !$acc loop vector(8)
  do y = 1, 4
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```



- **Vector** の長さが最内ループより大きい
→ 最内ループが小さい
- Vector の半分を無駄にしており、半分の性能しか出せない
- うまく性能を出すには？

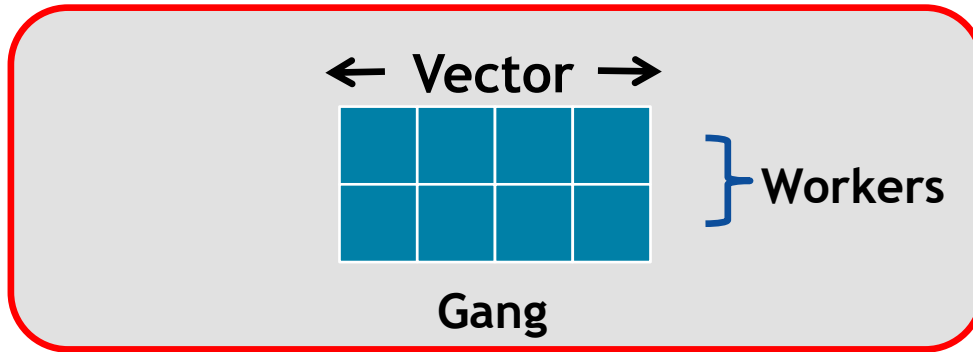
Gang - Worker - Vector の動作例: 最内ループが小さい場合



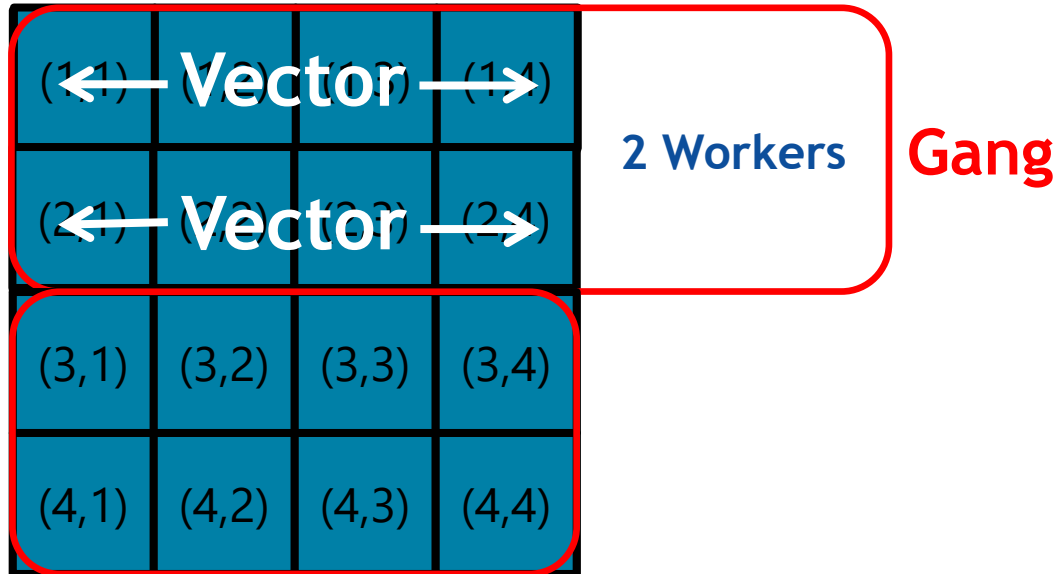
- 性能を引き出すためには Vector を 2 Workers に分割する必要がある
- 1つの長い Vector から、2つの短い Vector へ
- ループ構成に適した設定にできた

```
!$acc kernels loop gang worker(2)
do x = 1, 4
  !$acc loop vector(4)
  do y = 1, 4
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```

Gang - Worker - Vector の動作例: 最内ループが小さい場合

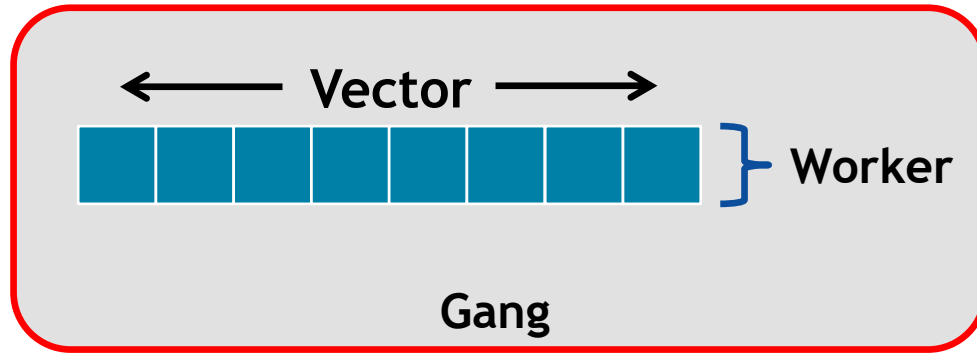


```
!$acc kernels loop gang worker(2)
do x = 1, 4
  !$acc loop vector(4)
  do y = 1, 4
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```



- Vector が小さくなったことでループサイズにマッチし無駄を省けた
- Gang, Worker, Vectorサイズの選択 = ループ長にマッチした値を考える必要がある

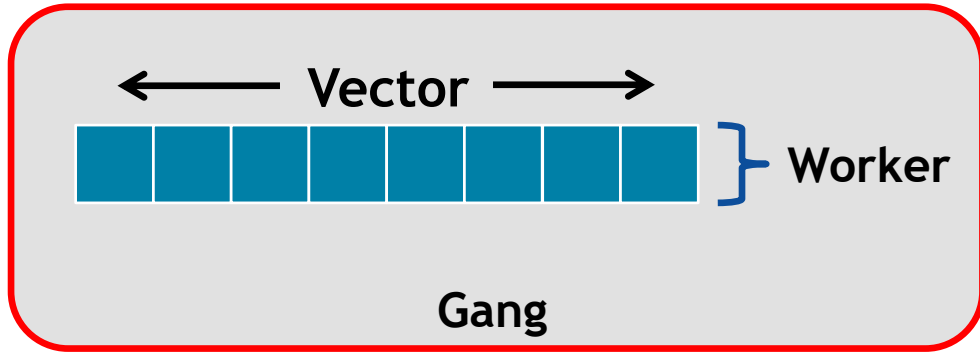
Gang - Worker - Vector の動作例: collapse句を使用する場合



- 多重ループには **collapse句** というループ一重化の機能がある

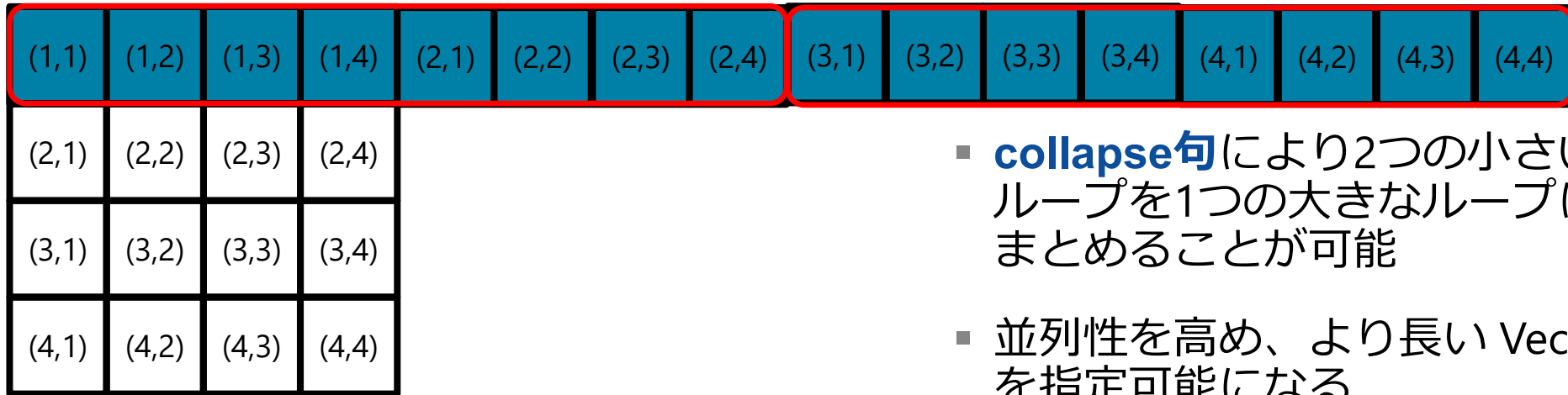
```
!$acc kernels loop collapse(2) gang worker(1) vector(8)
do x = 1, 4
  do y = 1, 4
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```

Gang - Worker - Vector の動作例: collapse句を使用する場合



```
!$acc kernels loop collapse(2) gang worker(1) vector(8)
do x = 1, 4
  do y = 1, 4
    array(x,y) = array(x,y) + 1
  end do
end do
!$acc end kernels
```

collapse(2)



- **collapse句**により2つの小さいループを1つの大きなループにまとめることが可能
- 並列性を高め、より長い Vector を指定可能になる

Warp: GPUスレッドの実行単位

- 実際にはGang - Worker - Vectorの数はより大きいものが割り当てられる
- NVIDIA GPU向けにプログラムする場合、 **Warps** を十分に活用できるように常にVectorの長さを32の倍数に固定する
- 簡単には32スレッドのグループのことを Warp と呼ぶ

device_type 節: 複数種類のターゲット対応

- **device_type (<type>)**
- これ以降に指定された節は実行が type で指定した device の場合にのみ適用される
- 例えばNVIDIA GPU向けに最適化したとしても、CPUなど別の device の性能を損なうことがなくなる
- 1つのディレクティブで複数の device について個別の最適化節を記述可能

```
!$acc parallel loop collapse(3) device_type(nvidia) vector_length(256)
do i = 1, size
  do j = 1, size
    do k = 1, size
      c(i,j) = c(i,j) + a(i,k) * b(k,j);
    end do
  end do
end do
```

ループ最適化の進め方

- Gang の数を設定する必要はほぼないのでコンパイラに任せる
- ほとんどの場合は Vector の長さ指定だけで並列ループを効果的に最適化可能
- Workerループを使うことは稀。Vector が非常に短い場合に Worker で全体の並列性を高めることに役立つ
- Vectorループでは、可能な限り配列の連続した要素を順に処理すべきである (SIMD)
- device_type句により、あるアーキテクチャへの最適化が他のアーキテクチャに影響しないように配慮する

2. 非同期プログラミング

非同期プログラミング

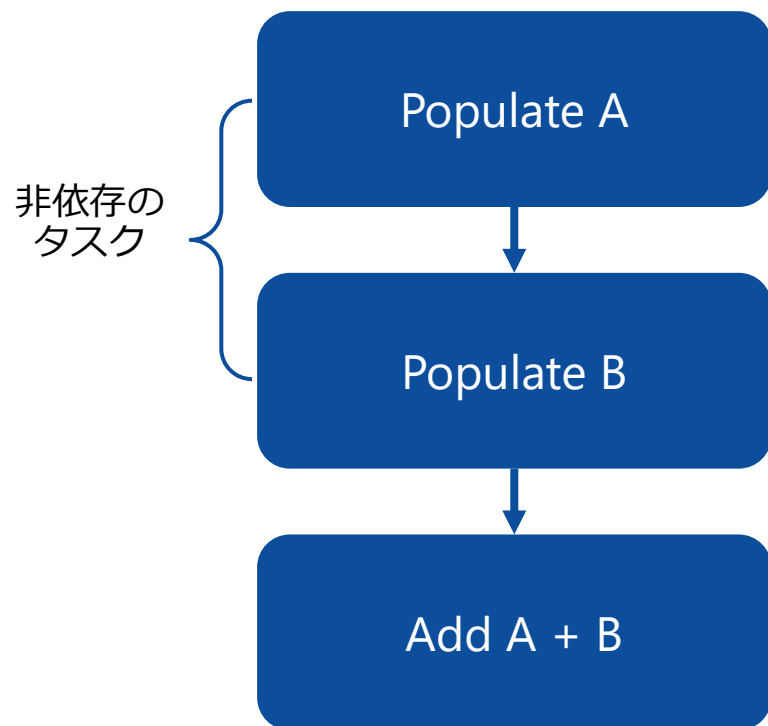
2つ以上の関連性のない操作が独立、または同時に行われても、すぐには同期が取る必要のないプログラミングのこと

現実世界での例：

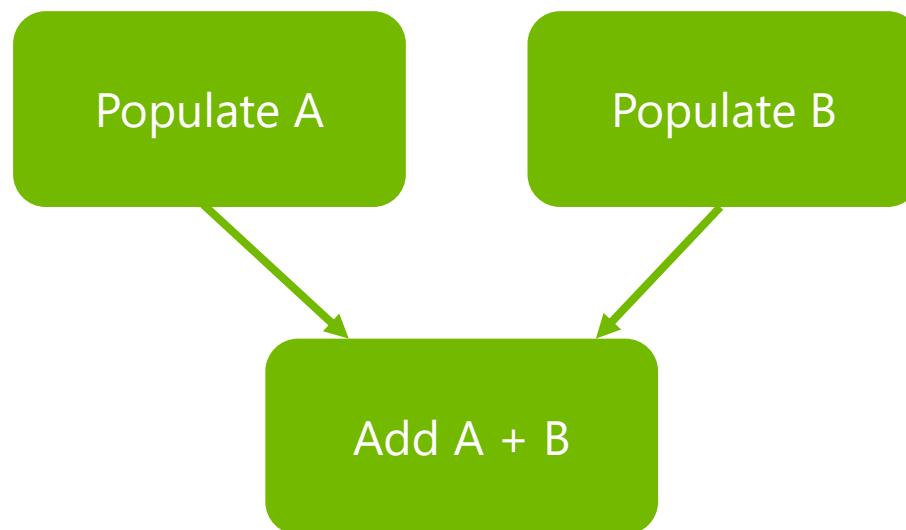
- 料理：他の食材の準備をしながらじゃがいもを茹でる
- プロジェクト：ある歴史上の有名な人物の研究に取り組む3人の学生がそれぞれ幼少期、青年期、壮年期について研究する
- 自動車の組み立て：各作業領域毎に独立して部品を作成し、最後にマージすることで車が完成するようにする

非同期の例

SYNCHRONOUS



ASYNCHRONOUS



タスクを非同期に行うためには、お互いに完全に独立な処理でなければならない

OpenACC async および wait 節

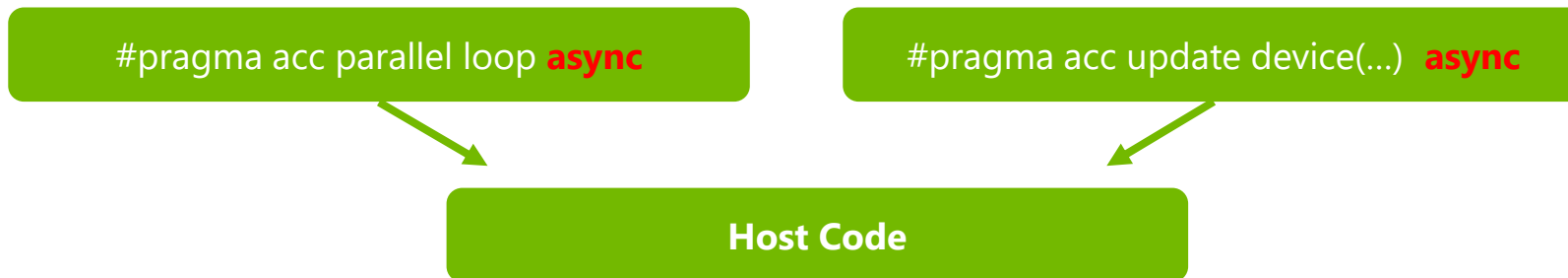
async 節の文法

C/C++: `#pragma acc <directive> <clauses> async(n)`

Fortran: `!$acc <directive> <clauses> async(n)`

Async launches work asynchronously in queue n, then returns to host

- OpenACCはasyncによりいくつかの処理を同時に実行可能
- Device上の計算とHost上の計算を同時に実行、あるいはDevice上で計算すると同時にデータ転送を行うことが可能
- キューn が指定されない場合はデフォルトキュー（後述）が指定される



非同期処理の実現: キューについて

- 非同期処理の実行は、キューと呼ばれる待ち行列を複数用意して実行する
- 非同期処理しないコードは1つのキュー（デフォルトキュー）しか使用しない

→ これまで紹介したプログラムはデフォルトキューを使用

- 必要に応じキューをいくつも生成できるが、生成にはオーバーヘッドが生じる
- 通常は必要最小限のキューを生成するのが良い

Default
Queue

Queue 1

Queue 2

非同期処理を使用しない場合

```
!$acc parallel loop
do i = 1, 100
    X(i) = ...
end do
!$acc update self(X[1:100])

!$acc parallel loop
do i = 101, 200
    X(i) = ...
end do
!$acc update self(X[101:200])

call print_array(X)
```

Default
Queue

Loop 1

Update 1

Loop 2

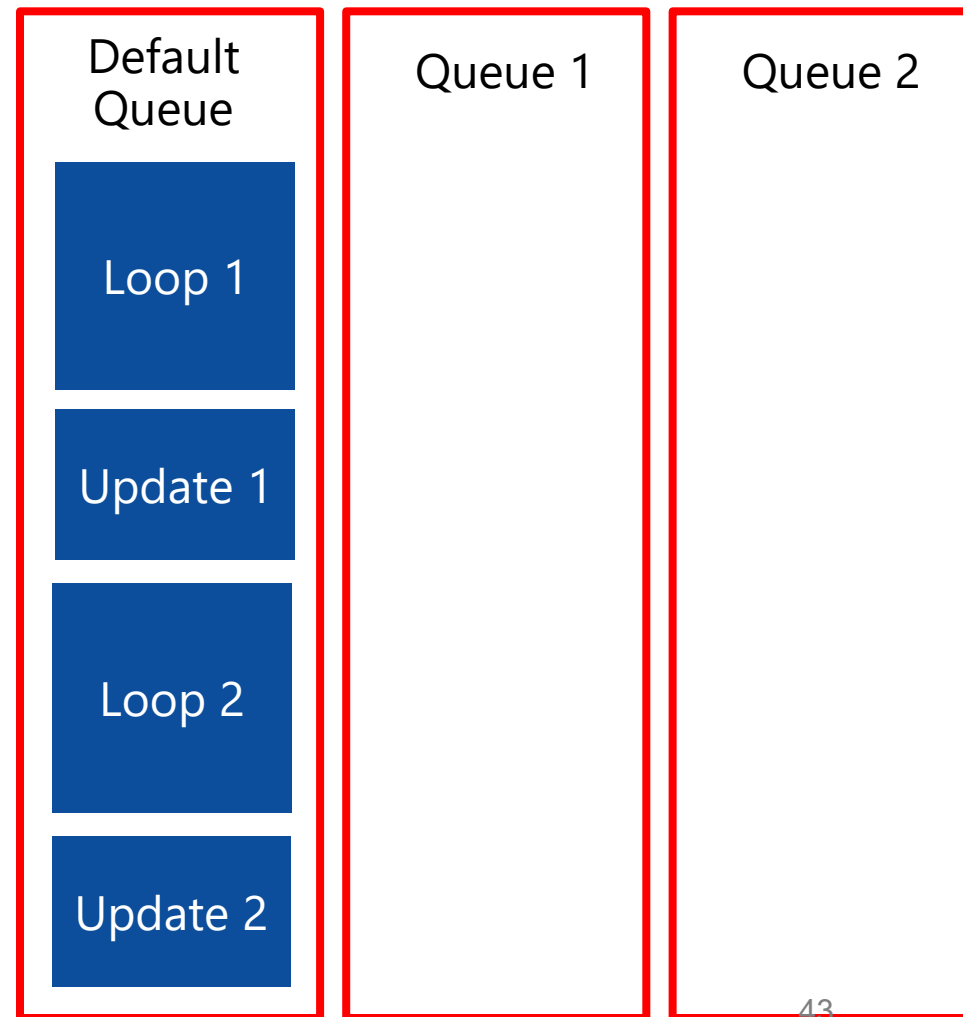
Update 2

Queue 1

Queue 2

非同期処理を使用しない場合

- 各処理は前の処理が終了するのを待つ
- OpenACCのデフォルト動作

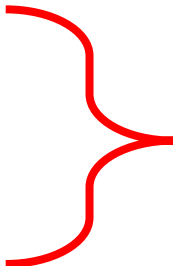


非同期処理の例：二つを並行処理

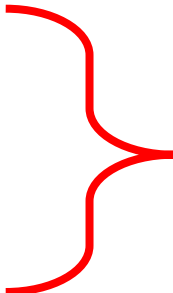
```
!$acc parallel loop async(1)
do i = 1, 100
  X(i) = ...
end do
!$acc update self(X[1:100]) async(1)

!$acc parallel loop async(2)
do i = 101, 200
  X(i) = ...
end do
!$acc update self(X[101:200]) async(2)

call print_array(X)
```



キュー-1へ投入



キュー-2へ投入

非同期処理の例：二つを並行処理

```
!$acc parallel loop async(1)
do i = 1, 100
  X(i) = ...
end do
!$acc update self(X[1:100]) async(1)

!$acc parallel loop async(2)
do i = 101, 200
  X(i) = ...
end do
!$acc update self(X[101:200]) async(2)

call print_array(X)
```

Default
Queue

Queue 1

Loop 1

Update 1

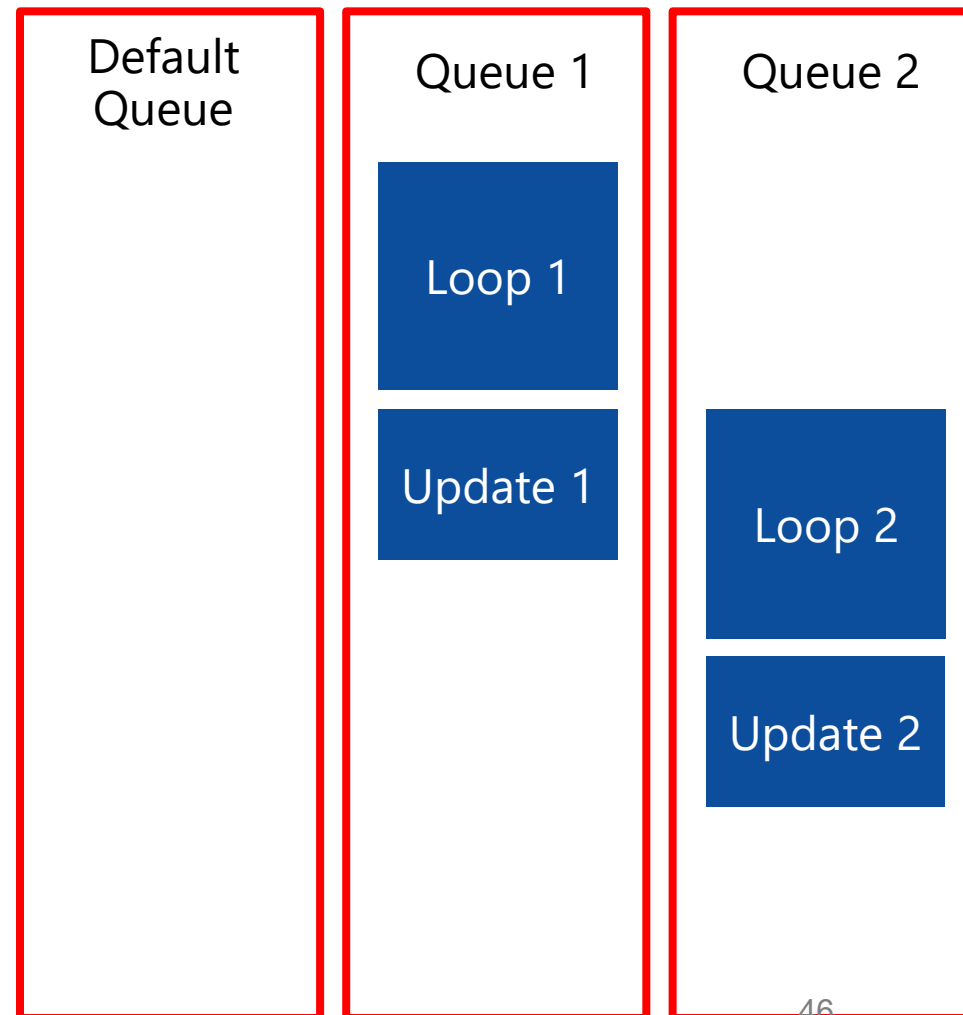
Queue 2

Loop 2

Update 2

非同期処理の例：二つを並行処理

- `async`節により、Loop 1とLoop 2を同時に実行できた
- `async`を使用しない場合に比べ、Loop 2をすぐに開始できたが...

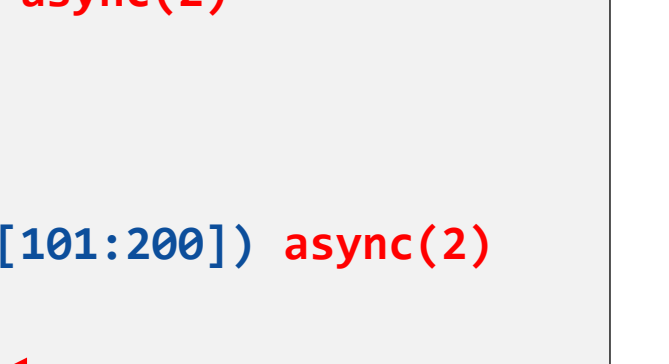


非同期処理の例：ホスト処理の落とし穴

```
!$acc parallel loop async(1)
do i = 1, 100
  X(i) = ...
end do
!$acc update self(X[1:100]) async(1)

!$acc parallel loop async(2)
do i = 101, 200
  X(i) = ...
end do
!$acc update self(X[101:200]) async(2)

call print_array(X)
```



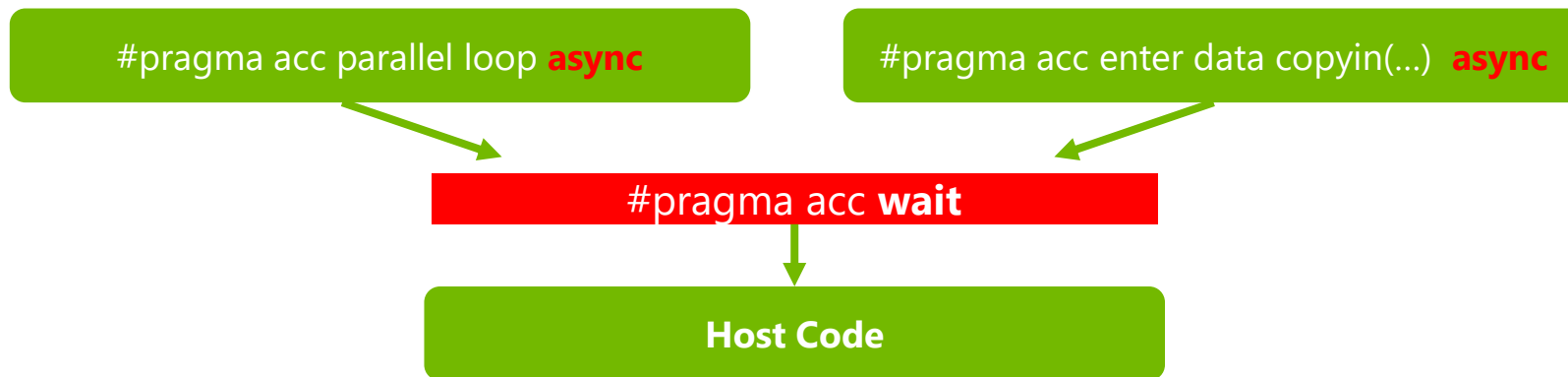
- `async`節により処理を異なるキューに入るが、その後のコードはすべて連続して実行される
- 左図のコードでは、ループや更新が終わる前に **`print_array`**関数が呼び出されてしまう
- これは **`wait`** ディレクティブで回避できる

wait ディレクティブ

C/C++: `#pragma acc wait(n)`

Fortran: `!$acc wait(n)`

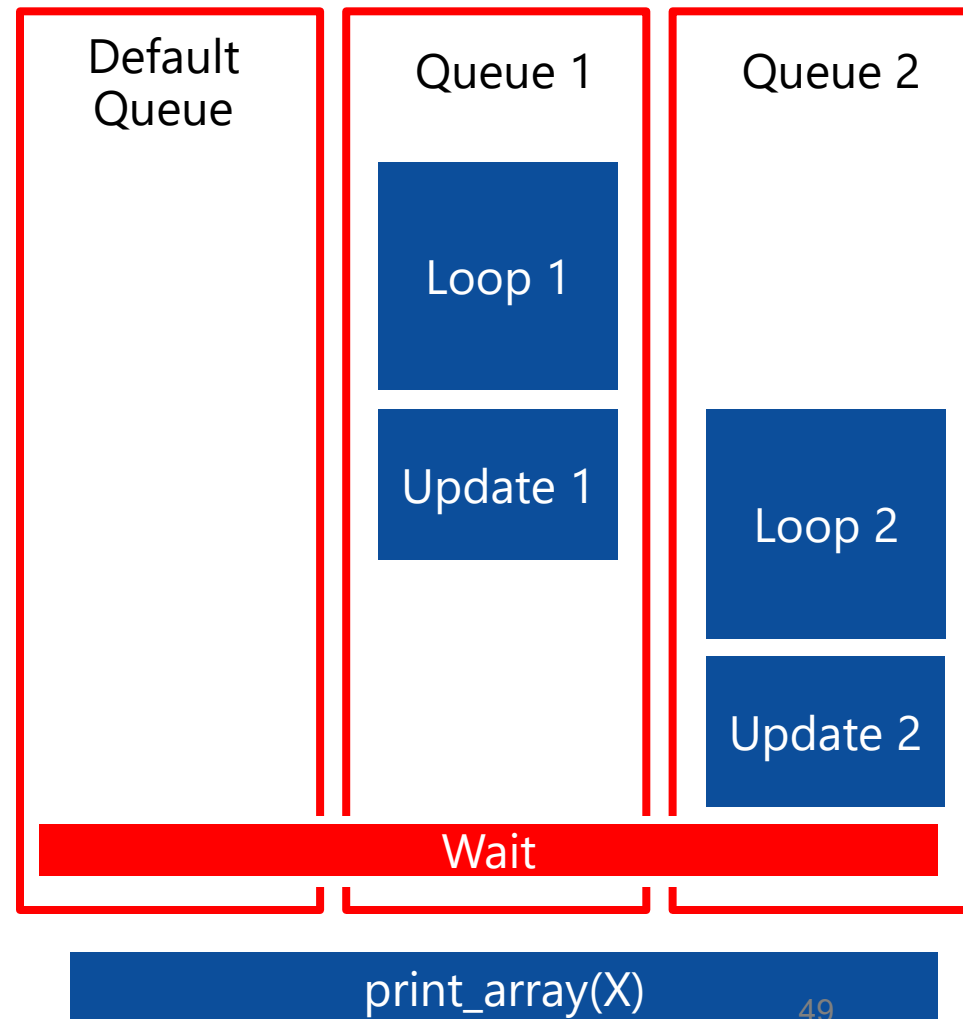
- wait はそれまでの非同期処理がすべて完了するまでプログラムを待機状態にする
- n を指定しない場合は、その時点のすべてのキューが完了するまで待機する



非同期処理の例：wait による同期

```
!$acc parallel loop async(1)
do i = 1, 100
  X(i) = ...
end do
!$acc update self(X[1:100]) async(1)

!$acc parallel loop async(2)
do i = 101, 200
  X(i) = ...
end do
!$acc update self(X[101:200]) async(2)
!$acc wait
call print_array(X)
```



3. CUDAとの連携 (Interoperability)

Interoperabilityの重要性

既存のCUDA資産の流用や、性能向上を考える場合に

OpenACCは、

- 自己完結のプログラミングモデル
- メインのプログラミングモデルとしての使用
- ネイティブのCUDAデータの直接操作 ← CUDAプログラムへ素早く機能追加する手段
- CUDAデバイス関数の呼び出し ← 既存資産の流用
- CUDAからOpenACC関数の呼び出し ← CUDAプログラムへ素早く機能追加する手段

を可能とする

Interoperability の考え方

データから見たとき

データシェアリング

- 複数のプログラミングモデル間でデバイス上のデータをやり取りする

カーネルシェアリング

- 別のプログラミングモデルで割り当てられたデータを使ってカーネルを呼び出す

具体的な相互利用のシチュエーション

OpenACCを用いて開発しているアプリケーションで

- CUDAでさらなる計算の最適化を行いたい ⇒ OpenACC管理のデータを共有

CUDAを用いて開発しているアプリケーションで

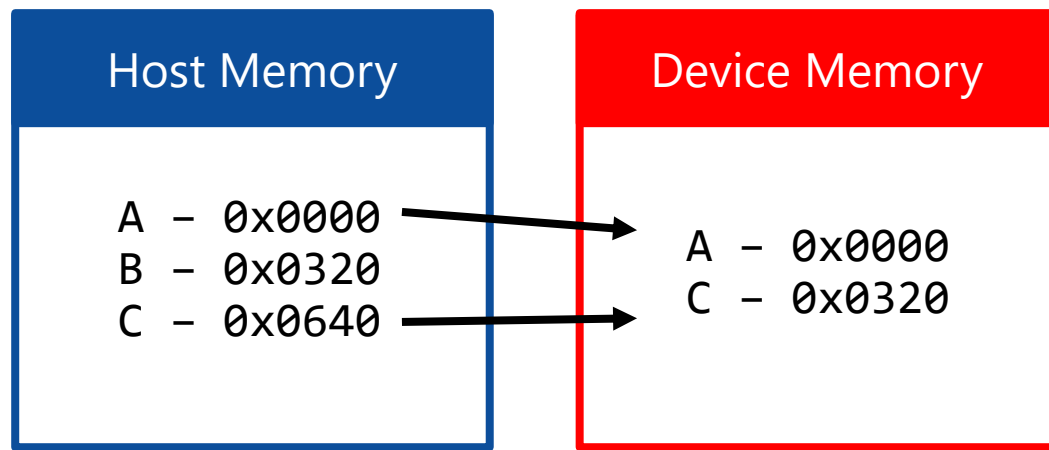
- OpenACCで新たな計算カーネルを追記したい ⇒ CUDA管理のデータを共有

OpenACC管理のデータを共有

- OpenACCからCUDAを呼び出す

OpenACCのデータを共有する

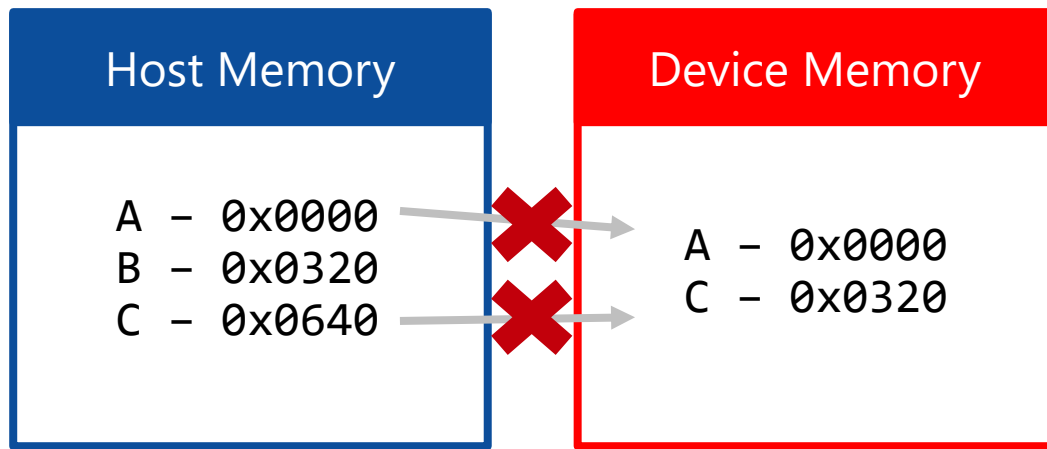
- OpenACC はプログラム上でHostとDeviceのメモリは1つのポインタで管理されるため、異なるポインタを持つ必要がない（変数は1つ）
- Device上にデータを割り当てたとき、HostとDeviceのアドレスのマッピングが作成される



- HostとDeviceどちらのメモリプールが使われるかはポインタが参照されるコンテキストによって変わる
- Hostコードでポインタ参照した場合はHostのアドレスが、
- Deviceコードでポインタ参照した場合はDeviceのアドレスが使用される

OpenACCのデータを共有する

- CUDAの場合、Host・Deviceで二つの変数があり、このようなマッピングは存在しない。したがって、CUDAを呼び出す場合、Deviceのアドレスを明示的に使用する必要がある
- OpenACCで確保したデータを使ってOpenACCコード内からCUDAカーネルを起動する場合、Deviceのアドレスを明示的に取得してCUDAカーネルに渡さなければならない



- **host_data** ディレクティブはOpenACCのDeviceデータへのアドレスを取得するために使用する
- **host_data** ディレクティブはHostの代わりにDeviceのポインタを使用するようコンパイラに指示する

呼び出したいCUDAコードの例: saxpy

- 右のような $y = ax + y$ を計算するCUDAコードをOpenACCから呼び出したい
- CUDAから呼び出す場合、この関数にはDeviceポインタ(~GPUメモリに確保した変数)を渡す
- OpenACCから呼び出す場合、dataディレクティブによりCPUからGPUにデータ転送し、host_dataディレクティブを利用し、CUDAカーネルにDeviceポインタを渡す

```
__global__
void saxpy_kernel(int n, float a,
                  float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

void saxpy(int n, float a,
           float *dx, float *dy)
{
    // Launch CUDA Kernel
    saxpy_kernel<<<4096,256>>>>(n, a, dx, dy);
}
```

Interface: FortranからCUDAを呼び出すために

- Fortran モジュールインターフェースを記載することでFortranプログラムからCの関数を呼び出せる
- 引数 n や a は C言語バインディングで変数の値が渡される
- インターフェース文によりFortran配列dx, dy のメモリアドレスが Cの関数の引数として受け渡される

```
module saxpy_c_mod
interface
  subroutine saxpy(n, a, dx, dy) bind(c)
    use iso_c_binding
    implicit none
    integer(C_INT) :: n
    real(C_FLOAT)  :: a
    type(C_PTR)    :: dx, dy
  end subroutine saxpy
end interface
end module saxpy_c_mod
```

host_data ディレクティブの使用例

Fortran からCUDAを呼ぶ

```
program main
  use saxpy_c_mod
  integer, parameter :: n = 2**20
  real, dimension(N) :: x, y
  real                :: a = 2.0
  x = 1.0
  y = 0.0

  !$acc data copy(y) copyin(x)
  ...
  !$acc host_data use_device(y,x)
  call saxpy(n, a, x, y)
  !$acc end host_data
  !$acc end data

end program
```

- `!$acc data copy(y) copyin(x)` Deviceにデータのコピーを作成する
- `!$acc host_data use_device(y,x)` – ランタイムにxとyのDeviceアドレスをsaxpy関数で使うように指示する
- “`!$acc host_data`” および “`!$acc end host_data`” の領域内で追加された他の関数呼び出しは、同様にDeviceアドレスを受け取る

ライブラリ: cuBLAS

host_dataを用いた手段は既存のGPUに最適化されたライブラリとのインタフェースとのインターフェイスとしても使用できる

例えば:

- cuBLAS
- cuFFT
- MAGMA
- Thrust(後述します)
- Libsci_acc

OpenACC Main Calling cuBLAS

```
program main
  use cublas_v2
  integer, parameter :: n = 2**20
  real*4, dimension(N) :: x, y
  type(cublasHandle)   :: cublas_H
  integer               :: status

  x = 1.0
  y = 0.0

  status = cublasCreate(cublas_H)
  !$acc data copy(y) copyin(x)
  !$acc host_data use_device(y,x)
  status = cublasSaxpy(cublas_H, n, 2.0, x, 1, y, 1)
  !$acc end host_data
  !$acc end data
  status = cublasDestroy(cublas_H)

end program
```

nvfortran -acc -cudalib=cublas saxpy_v2.f90

CUDA管理のデータを共有

- CUDAからOpenACCを呼び出す

CUDAのデータを共有する

Using CUDA addresses in OpenACC compute constructs

- CUDAのデータをOpenACCと共有する場合は？
- CUDAに割り当てられたデータをOpenACCで参照したい
- CUDAアドレスを使用する場合は以下の方法でマークをする必要がある：
 1. **DeviceポインタをOpenACCに明示する**
 2. **DeviceポインタをOpenACCランタイムオブジェクトと関連付ける**

呼び出したいOpenACCコードの例: saxpy

OpenACC Version

```
void saxpy(int n, float a,
           float *x, float *y)
{

#pragma acc parallel loop default(present)
  for(int i = 0; i < n; i++) {
    y[i] = a*x[i] + y[i];
  }

}
```

- 左のような $y = ax + y$ を計算するOpenACCコードをCUDAから呼び出したい
- OpenACCから呼び出す場合は、この関数にHostポインタを与える。OpenACCランタイムがマップされたDeviceポインタに変換する
- しかしCUDAデバイスメモリを使用する場合、Deviceポインタしか使用できない
- **deviceptr**を用いることで、xとyがDeviceポインタであり変換が必要ないことをランタイムに伝えることが可能

deviceptr 節

deviceptr節はオブジェクトがすでにDevice上にあり変換は必要ないことをコンパイラに伝える

- deviceptrはparallel, kernels, dataディレクティブで利用できる

```
program main
.....
  cudaMallocManaged((void*)&x,(size_t)n*sizeof(float));
  cudaMallocManaged((void*)&y,(size_t)n*sizeof(float));
  call saxpy(n, a, x, y)
.....
end program
```

```
subroutine saxpy(n, a, x, y)
  integer :: n
  real    :: a, x(*), y(*)
  !$acc parallel loop deviceptr(y,x)
  do i = 1, n
    y(i) = y(i) + a * x(i)
  end do
  !$acc end parallel
end subroutine
```

XとYはすでにDevice
ポインタとして使える
ので直接使用する

OpenACC & Thrust

ThrustはC++のためのSTLライクなアクセラレータライブラリ

- High-level interface
- Host/Device container classes
- Common parallel algorithms

Thrustのvector型をDeviceポインタにキャストしてOpenACCで 사용할 수 있는 것이 가능

```
void saxpy(int n, float a, float *x, float *y)
{
    #pragma acc parallel loop deviceptr(y,x)
        for(int i = 0; i < n; i++) {
            y[i] = a*x[i] + y[i];
        }
}
```



What is Thrust?

Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL). Thrust's **high-level** interface greatly enhances programmer **productivity** while enabling performance portability between GPUs and multicore CPUs. **Interoperability** with established technologies (such as CUDA, TBB, and OpenMP) facilitates integration with existing software. Develop **high-performance** applications rapidly with Thrust!

Examples

Thrust is best explained through an example. It first generates a random vector, then transfers them to a parallel device.

```
#include <thrust/host_vector>
#include <thrust/device_vector>
#include <thrust/generate>
#include <thrust/sort>
#include <thrust/copy>
#include <algorithm>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers
    thrust::host_vector<int> h_vec(32768);
    std::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec(h_vec);

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to the host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

This code sample computes the sum of the squares of the elements in the vector.

Thrust 1.17.2

Latest

Summary

Thrust 1.17.2 is a minor bugfix release that provides an updated version of CUB.

Assets

Source code (zip)

15 hours ago

Source code (tar.gz)

15 hours ago

Join discussion

Thrust 2.0.0

Pre-release

Summary

The Thrust 2.0.0 major release adds a dependency on libcu++ and contains several breaking changes. These include new diagnostics when inspecting device-only lambdas from the host, removal of the `libcub` symlink in the Thrust repository root, and removal of the deprecated `THRUST_*` macros. It also includes several minor bugfixes and cleanups.

Breaking Changes

- #1605: Add libcu++ dependency.
 - A suitable version of libcu++ is provided through the `$(THRUST_ROOT)/dependencies/libcudacxx/` submodule.
 - Non-cmake users may need to add the libcu++ include path to their builds (`-I $(THRUST_ROOT)/dependencies/libcudacxx/include/`).
 - The Thrust CMake packages have been updated to add this include path.
- #1605: The following macros are no longer defined by default. They can be re-enabled by defining

OpenACC & Thrust

ThrustはC++のためのSTLライクなアクセラレータライブラリ

- High-level interface
- Host/Device container classes
- Common parallel algorithms

Thrustのvector型をDeviceポインタにキャストしてOpenACCで 사용할 수 있습니다

```
subroutine saxpy(n, a, x, y) bind(c)
  integer :: n
  real    :: a, x(*), y(*)
  !$acc kernels deviceptr(y,x)
  do i = 1, n
    y(i) = y(i) + a * x(i)
  end do
  !$acc end kernels
end subroutine
```

```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
for(int i=0; i<N; i++)
{
  x[i] = 1.0f;
  y[i] = 0.0f;
}

// Copy to Device
thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

saxpy(N, 2.0, d_x.data().get(),
      d_y.data().get());

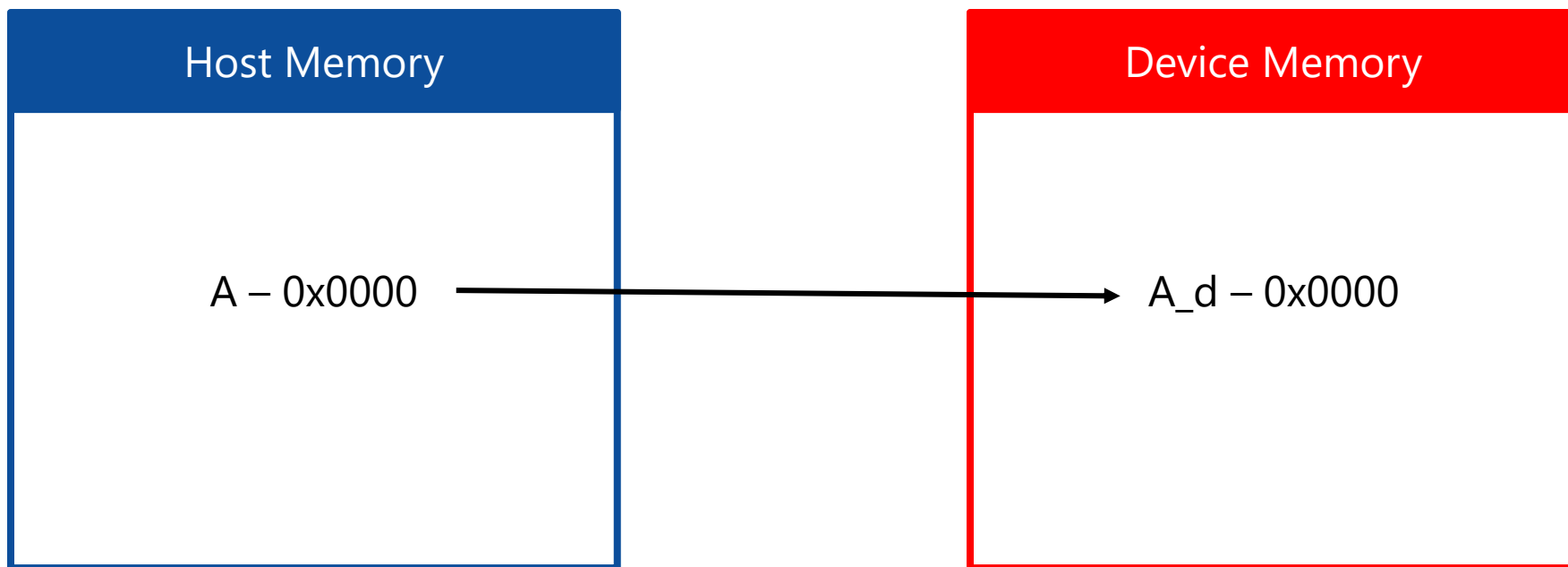
// Copy back to host
y = d_y;
```

デバイスポインタのマッピング

- OpenACCはホストとデバイスのメモリアドレスのマッピングを行う
- 一方でCUDAはホストとデバイスのメモリアドレスのマッピングは行わない
- OpenACCとCUDAを相互に利用するために、CUDAのデバイスデータを、OpenACCでホストアドレス(ポインタ)と対応付けることができる

OpenACCの acc_map_data 関数

```
float A[100];  
cudaMalloc((void*)&A_d, (size_t)100*sizeof(float));  
acc_map_data(A, A_d, 100*sizeof(float));
```



OpenACCの acc_map_data 関数

- **acc_map_data** (**acc_unmap_data**) によりデバイスのメモリ割り当てをOpenACCの変数と対応付ける(map)または対応の解除(unmap)ができる
- データの対応付けをするために双方のポインタが有効である必要がある
- さらにメモリを解放する場合はデータの対応付けを解除する必要がある

対応付けにより、このループは自動的にxとyをそれぞれのデバイスアドレスに変換します

```
float x[n], y[n];

cudaMalloc((void*)&x_d, (size_t)n*sizeof(float));
cudaMalloc((void*)&y_d, (size_t)n*sizeof(float));

acc_map_data(x, x_d, n*sizeof(float));
acc_map_data(y, y_d, n*sizeof(float));

#pragma acc parallel loop
for(int i = 0; i < n; i++) {
    x[i] = x[i] * y[i];
}
```

device routinesの利用

acc routineによるCUDAとの連携

CUDA Code

```
extern "C" __device__ void  
f1dev(float* a, float* b)  
{  
    *a = *b + 1.0;  
}
```

nvcc -c -rdc true testsub.cu

同様にCUDA **__device__**関数を**acc routine**で宣言すればOpenACCから呼び出し可能

OpenACC Code

```
interface  
subroutine f1dev( x, y )  
!$acc routine seq  
bind(c)  
    use iso_c_binding  
    implicit none  
    type(C_PTR):: x, y  
end subroutine f1dev  
end interface  
  
...  
!$acc parallel loop copy(a(1:n))  
copyin(b(1:n))  
do i, 1, n  
    call f1dev( a(i), b(i) )  
end do
```

nvfortran -acc -cuda test.f90 testsub.o

acc routineによるCUDAとの連携

CUDA Code

```
extern "C" __device__ void  
f1dev(float* a, float* b, int i)  
{  
    a[i] = .... b[i] .... ;  
}
```

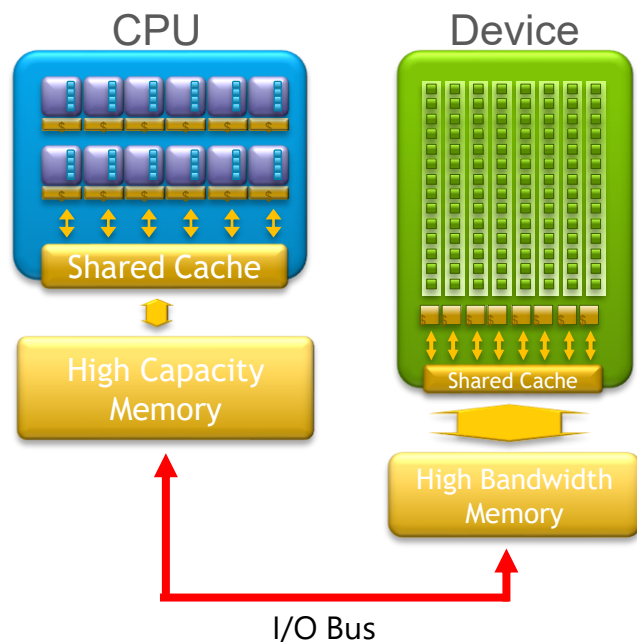
同様にCUDA **__device__**関数を**acc routine**
で宣言すればOpenACCから呼び出し可能

OpenACC Code

```
interface  
subroutine f1dev( a, b, i )  
!$acc routine seq  
bind(c)  
    use iso_c_binding  
    implicit none  
    integer(C_INT) :: i  
    type(C_PTR):: a, b  
end subroutine f1dev  
end interface  
...  
!$acc parallel loop present(  
a(1:n), b(1:n) )  
do i, 1, n  
    call f1dev( a, b, i )  
end do
```

4. マルチGPU計算の基礎

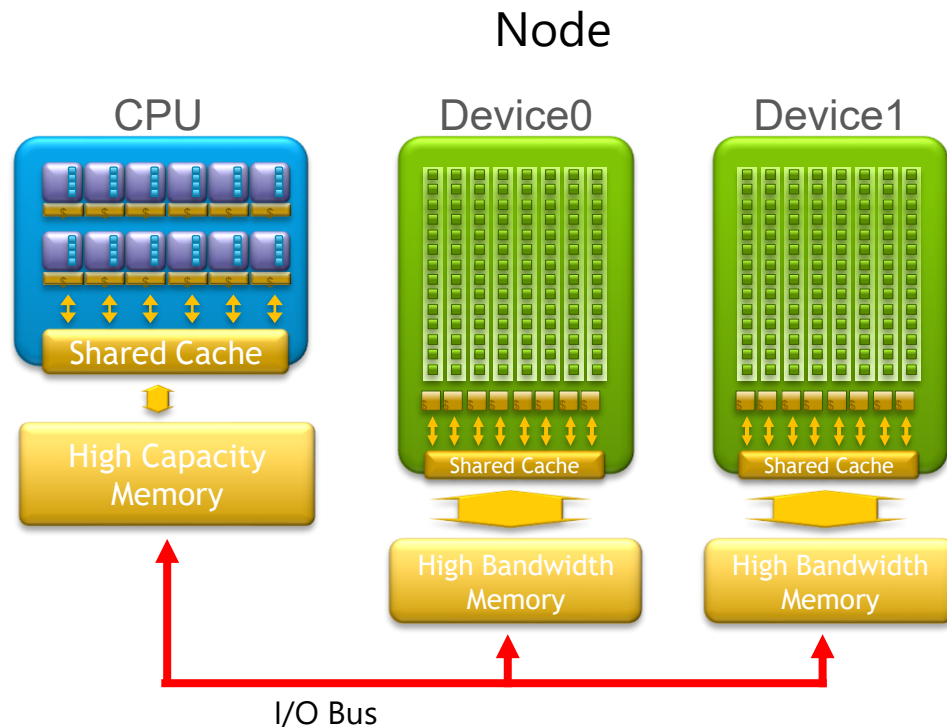
複数デバイスを持つ計算機を利用する



- ここまでは1台のDeviceを想定、複数Device、複数計算機は考慮していなかった
- しかし昨今、計算機は複数のDeviceが持つことが一般的となった
- いくつかの異なる方法でのMulti-Deviceなプログラミング手法を説明する

複数デバイスを持つ計算機を利用する

- **Multi-Device**を実現する3つの選択肢
- OpenACCの組み込み関数とasyncを用いたMulti-Device操作
- OpenMPで各スレッドに別のDeviceを割り当てる
- MPIを使用しMPIランクごとに別のデバイスを割り当てる（**弊社推奨**）



OpenACC のみでマルチGPUを使う

OpenACC のみでマルチGPUを使う

- OpenACCの組み込み関数を用いて実装する
- 非同期プログラミング (async-wait) も使用する必要がある

```
int num_devices = acc_get_num_devices(acc_device_default);
```

- | | | | | | | | | |
|--|--|--|--|----------|--|--|--|--|
| +-----+-----+-----+-----+-----+-----+-----+-----+-----+ | | | | | | | | |
| NVIDIA-SMI 515.43.04 Driver Version: 515.43.04 CUDA Version: 11.7 | | | | | | | | |
| +-----+-----+-----+-----+-----+-----+-----+-----+-----+ | | | | | | | | |
| GPU Name Persistence-M Bus-Id Disp.A Volatile Uncorr. ECC | | | | | | | | |
| Fan Temp Perf Pwr:Usage/Cap Memory-Usage GPU-Util Compute M. | | | | | | | | |
| | | | | MIG M. | | | | |
| +-----+-----+-----+-----+-----+-----+-----+-----+-----+ | | | | | | | | |
| 0 NVIDIA A100-SXM... On 00000000:01:00.0 Off 0 | | | | | | | | |
| N/A 43C P0 140W / 400W 4812MiB / 40960MiB 82% Default | | | | Disabled | | | | |
| +-----+-----+-----+-----+-----+-----+-----+-----+-----+ | | | | | | | | |
| 1 NVIDIA A100-SXM... On 00000000:41:00.0 Off 0 | | | | | | | | |
| N/A 31C P0 55W / 400W 0MiB / 40960MiB 0% Default | | | | Disabled | | | | |
| +-----+-----+-----+-----+-----+-----+-----+-----+-----+ | | | | | | | | |
| 2 NVIDIA A100-SXM... On 00000000:81:00.0 Off 0 | | | | | | | | |
| N/A 28C P0 53W / 400W 0MiB / 40960MiB 0% Default | | | | Disabled | | | | |
| +-----+-----+-----+-----+-----+-----+-----+-----+-----+ | | | | | | | | |
| 3 NVIDIA A100-SXM... On 00000000:C1:00.0 Off 0 | | | | | | | | |
| N/A 28C P0 52W / 400W 0MiB / 40960MiB 0% Default | | | | Disabled | | | | |
| +-----+-----+-----+-----+-----+-----+-----+-----+-----+ | | | | | | | | |

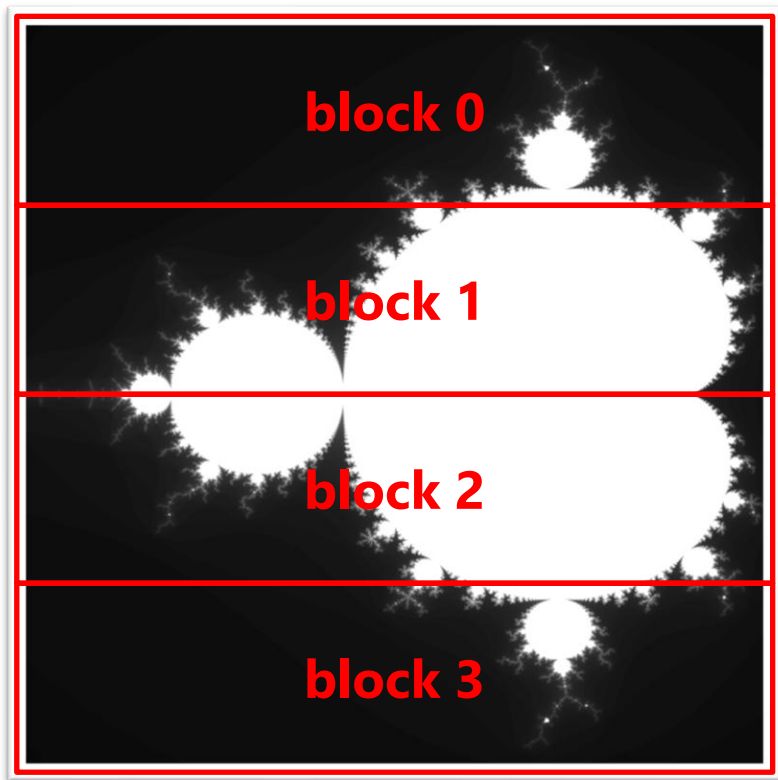
Deviceの切り替え

```
acc_set_device_num(device_num, acc_device_default);
```

```
acc_set_device_num(0, acc_device_default);  
#pragma acc parallel loop async  
for(int i = 0; i < size; i++) {  
    ...  
}  
  
acc_set_device_num(1, acc_device_default);  
#pragma acc parallel loop async  
for(int j = 0; j < size; j++) {  
    ...  
}
```

- `acc_set_device_num`でアクティブなDeviceを切り替える
- Deviceカーネルをキューに登録してから、別のDeviceに切り替えられる
- `async`ループにすることで他のDeviceの処理を待たずに各Deviceの処理を起動することが可能

Mandelbrot集合のマルチGPU化



- マンデルブロ集合を例にMulti-Device化を考える
- コードの目的は画像をいくつかのブロックへ分割し、各ブロックを別々のデバイスで計算し高速化
- シンプルにMulti-Device化が可能

Mandelbrot集合のマルチGPU化: データ

acc enter/exit dataを使用

全Deviceの計算用メモリを確保

全Deviceのメモリを解放

```
int main() {  
    ...  
    int ndevices = acc_get_num_devices(acc_device_default);  
    int block_size = WIDTH*HEIGHT/ndevices;  
    for(int device=0; device < ndevices; device++) {  
        acc_set_device_num(device, acc_device_default);  
        #pragma acc enter data create(image[block_size*device:block_size])  
    }  
  
    ... // Compute  
  
    for(int device=0; device < ndevices; device++) {  
        acc_set_device_num(device, acc_device_default);  
        #pragma acc exit data delete(image)  
    }  
}
```

Mandelbrot集合のマルチGPU化: データ

acc enter/exit dataを使用

- Deviceにより非同期にメモリの割当と解放が行われるとは限らない
- 事前にメモリを割り当てる
- この計算では、各Deviceは配列全体ではなく部分配列を持てば良い

```
int main() {  
    ...  
    int ndevices = acc_get_num_devices(acc_device_default);  
    int block_size = WIDTH*HEIGHT/ndevices;  
    for(int device=0; device < ndevices; device++) {  
        acc_set_device_num(device, acc_device_default);  
        #pragma acc enter data create(image[block_size*device:block_size])  
    }  
  
    ... // Compute  
  
    for(int device=0; device < ndevices; device++) {  
        acc_set_device_num(device, acc_device_default);  
        #pragma acc exit data delete(image)  
    }  
}
```

Mandelbrot集合のマルチGPU化: 計算部分

全Deviceループ処理

計算開始 (async)

データ転送開始
(async)

次のDeviceへ

```
int main() {  
    int ndevices = acc_get_num_devices(acc_device_default);  
    int block_size = WIDTH*HEIGHT/ndevices;  
    ... // Allocate Data  
    for(int device=0; device<ndevices; device++) {  
        int yStart = device*(HEIGHT/ndevices);  
        int yEnd    = yStart+(HEIGHT/ndevices);  
        acc_set_device_num(device, acc_device_default);  
        #pragma acc parallel loop async  
        for(int y=yStart;y<yEnd;y++) {  
            for(int x=0;x<WIDTH;x++) {  
                image[y*WIDTH+x]=mandelbrot(y,x);  
            }  
        }  
        #pragma acc update ¥  
        host(image[yStart*WIDTH:block_size]) async  
  
    }  
    ... // Wait  
    ... // Deallocate Data  
}
```

Mandelbrot集合のマルチGPU化: 計算部分

- Multi-Devicesで非同期処理を行う場合は各Deviceが個別に *async queue*を持つ
- `#pragma acc parallel loop async`
- 各Deviceにとってデフォルトの *async queue*となる
- 最後に、Multi-Deviceの場合には次の処理を行う前にすべてのDeviceが処理を終了したことを確かめる必要がある

```
int main() {  
    int ndevices = acc_get_num_devices(acc_device_default);  
    int block_size = WIDTH*HEIGHT/ndevices;  
    ... // Allocate Data  
    for(int device=0; device<ndevices; device++) {  
        int yStart = device*(HEIGHT/ndevices);  
        int yEnd    = yStart+(HEIGHT/ndevices);  
        acc_set_device_num(device, acc_device_default);  
        #pragma acc parallel loop async  
        for(int y=yStart;y<yEnd;y++) {  
            for(int x=0;x<WIDTH;x++) {  
                image[y*WIDTH+x]=mandelbrot(y,x);  
            }  
        }  
        #pragma acc update ¥  
        host(image[yStart*WIDTH:block_size]) async  
  
    }  
    ... // Wait  
    ... // Deallocate Data  
}
```


Mandelbrot集合のマルチGPU化: waitによる同期

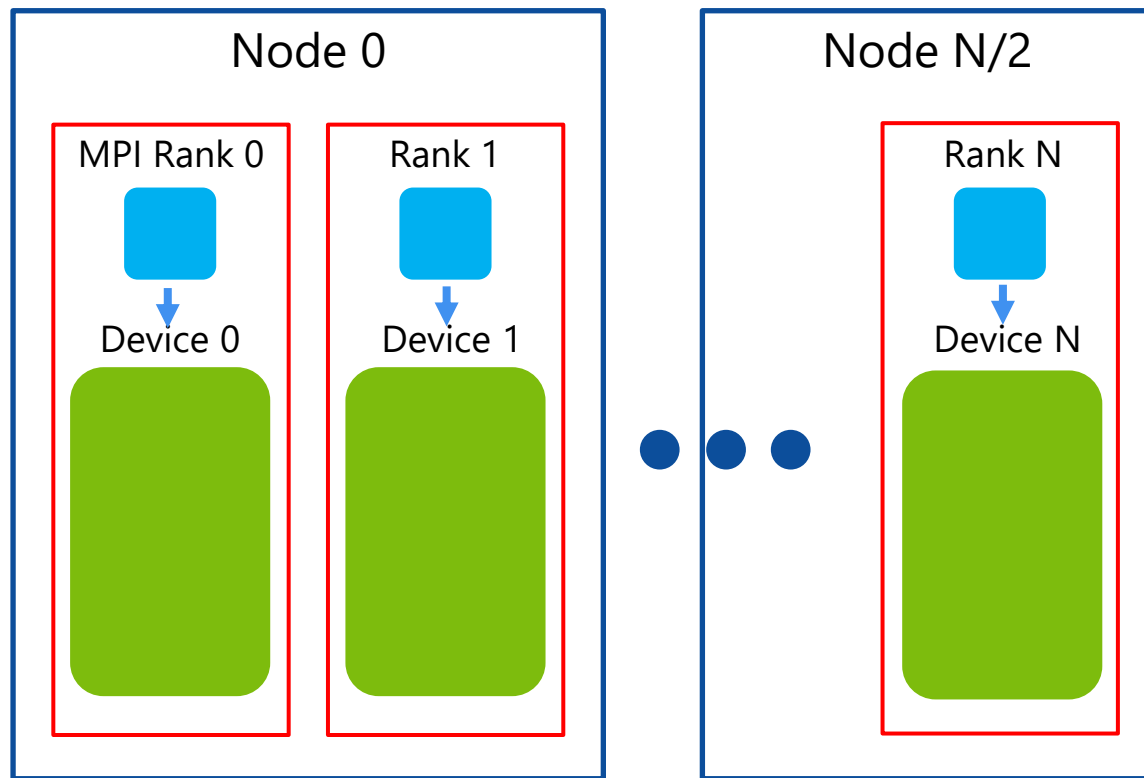
- 単体のwaitでは最後のDeviceの終了のみを待機してしまう
- 全Deviceの計算終了を確実に調べるために全*async queue*について**wait**を発行

全デバイスで**wait**を
呼び出す

```
int main() {  
    int ndevices = acc_get_num_devices(acc_device_default);  
    int block_size = WIDTH*HEIGHT/ndevices;  
    ... // Allocate Data  
  
    ... // Compute  
  
    for(int device=0; device < ndevices; device++) {  
        acc_set_device_num(device, acc_device_default);  
        #pragma acc wait  
    }  
  
    ... // Deallocate Data  
}
```

OpenACCとMPIを用いたマルチGPU化

MPIによるマルチGPU化



- MPIは実装は難しいが、柔軟性と拡張性に優れる
- OpenACC単体でのマルチGPU化コードは制御が複雑化
- MPI + OpenACCは、**標準的なMPIプログラム上でSingle-Deviceコードを制御する**。OpenACC単体よりもシンプルな構造になると期待される
- 物理的な計算機に何台のGPUが乗っていても、**仮想的にSingle-Deviceの計算機が複数あるものとして扱える**

MPIによるマルチGPU化: 簡単な例

```
MPI_Init(&argc,&argv);
int rank, nranks;
MPI_Comm_size(MPI_COMM_WORLD, &nranks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

acc_set_device_num(rank, acc_device_default);

MPI_Scatter(...);

#pragma acc parallel loop
for(int i = rank*N; i < rank*(N+1); i++)
    // Loop

MPI_Gather(...);
```

- 各MPIランクはループの一部または全く異なるループを実行できる
- 複数のDeviceを使用する場合には各MPIランクに1つずつDeviceを割り当てることができる（最もベーシックなやり方）
- Deviceの計算リソースが十分活用されていない場合は、複数のMPIランクでDeviceを共有することもできる

MPIを用いたMandelbrot集合のマルチGPU化

MPIは分散メモリモデルのため、各MPIランクは計算領域の*sub-image*を持つ

計算後、*sub-image*を集めて*single-image*とする

```
int main(int argc, char** argv ) {
    MPI_Init(&argc,&argv);
    int rank, nranks;
    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int block_size = WIDTH*HEIGHT/nranks;
    double *subimage = new double[block_size];
    ...
    acc_set_device_num(rank, acc_device_default);
#pragma acc parallel loop
    for(int y=0; y<HEIGHT/nranks; y++) {
        for(int x=0; x<WIDTH; x++) {
            subimage[y*WIDTH+x]=mandelbrot(y,x*(HEIGHT/rank));
        }
    }
#pragma acc update host(subimage[0:block_size])
    MPI_Gather(subimage, ..., image, ...);
}
```

MPIを用いたMandelbrot集合のマルチGPU化

各MPIランクはIDに応じて異なるDeviceが使用可能

各MPIランクはyについて部分領域を計算するためランクに応じてyの値を変更する

```
int main(int argc, char** argv ) {
    MPI_Init(&argc,&argv);
    int rank, nranks;
    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int block_size = WIDTH*HEIGHT/nranks;
    double *subimage = new double[block_size];
    ...
    acc_set_device_num(rank, acc_device_default);
#pragma acc parallel loop
    for(int y=0; y<HEIGHT/nranks; y++) {
        for(int x=0; x<WIDTH; x++) {
            subimage[y*WIDTH+x]=mandelbrot(y,x*(HEIGHT/rank));
        }
    }
#pragma acc update host(subimage[0:block_size])
    MPI_Gather(subimage, ..., image, ...);
}
```

Device間のMPIデータ通信

- MPIは`host_data`ディレクティブと`use_device`句を使用してDevice間のデータ転送を簡単化できる
- MPI_GatherにDeviceメモリのポインタを渡すだけで、Device間のデータ転送が可能になる
(Device memory supportがあるMPI処理系によって)

OpenACCの`host_data`と`use_device`によりDevice間の直接データ転送を実現

```
int main(int argc, char** argv ) {
    MPI_Init(&argc,&argv);
    int rank, nranks;
    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int block_size = WIDTH*HEIGHT/nranks;
    double *image = new double[WIDTH*HEIGHT];
    double *subimage = new double[block_size];
    if(rank == 0) {
        #pragma acc enter data create(image[0:WIDTH*HEIGHT])
    }
    #pragma acc enter data create(subimage[0:block_size])

    ... // GPU Computation on subimage

    #pragma acc host_data use_device(subimage, image)
    MPI_Gather(subimage, block_size, image, WIDTH*HEIGHT,
               MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

MPI処理系によるDeviceメモリ転送

CUDA-aware MPI, GDRについて

CUDA-aware MPI

- MPI処理系の内部でHostメモリを転送バッファとして使用し仮想的なDeviceメモリ転送を実現 (Traditional)

GDR (GPU Direct RDMA)

- 各計算ノードに搭載された通信デバイス (e.g. InfiniBand) の機能を用いて、Hostメモリを介在せずにDeviceメモリを直接転送する

サポートしている代表的な処理系

- Open MPI, MVAPICH2, MVAPICH2-GDR
- HPC-X

* Open MPIを含むNVIDIAの独自パッケージ。NVIDIA HPC SDKに同梱されている。 <https://developer.nvidia.com/networking/hpc-x>

CUDA-aware MPI, GDRの使い方

Open MPI

```
# CUDA-aware MPI (Open MPIビルド時に設定されていれば自動的に有効化)
```

```
$ mpirun ... ./a.out
```

```
# GDR
```

```
$ mpirun --mca btl_openib_want_cuda_gdr 1 ... ./a.out
```

[FAQ: Running CUDA-aware Open MPI \(open-mpi.org\)](https://www.open-mpi.org/faq/?category=runcuda) <https://www.open-mpi.org/faq/?category=runcuda>

MVAPICH2

```
# CUDA-aware MPI
```

```
$ export MV2_USE_CUDA=1
```

```
$ mpirun_rsh ... ./a.out
```

MVAPICH2-GDR

```
# GDR
```

```
$ export MV2_PATH=$(basename $(which mpirun_rsh))
```

```
$ export MV2_USE_CUDA=1
```

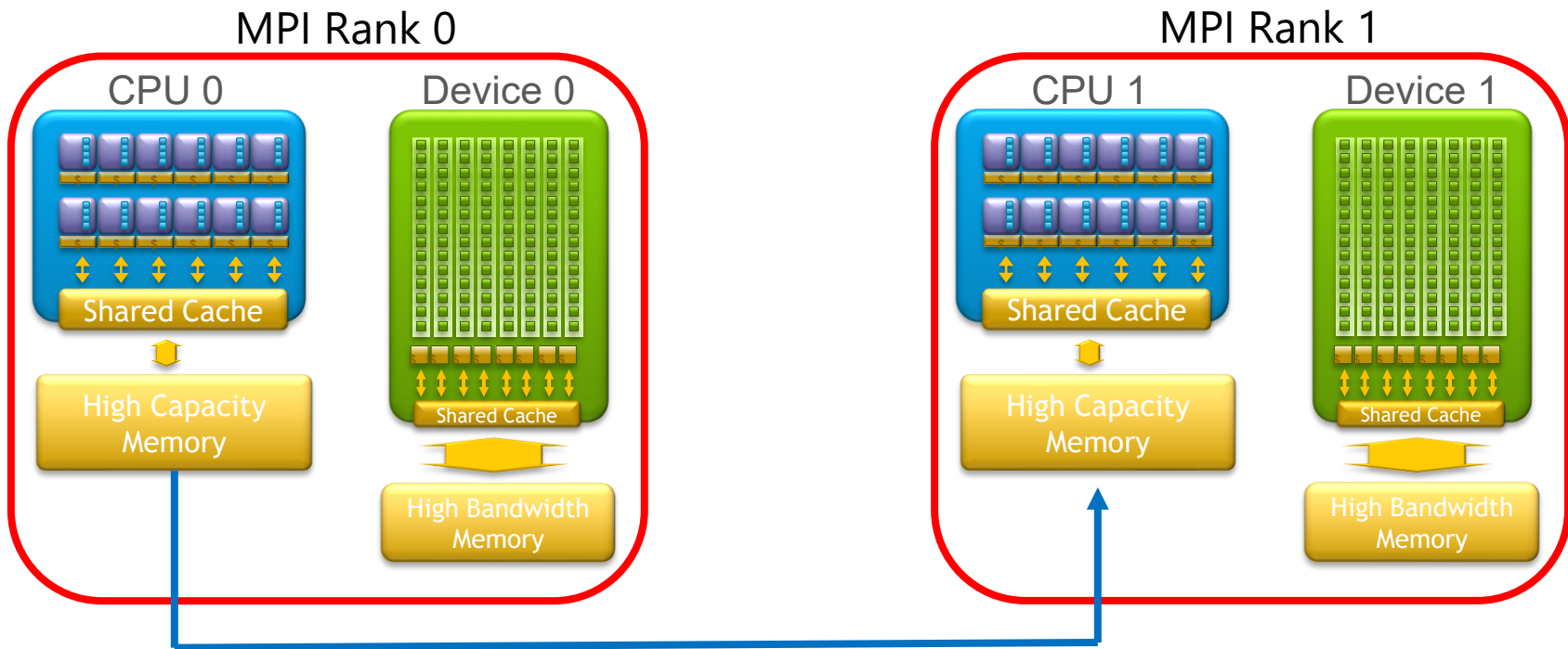
```
$ mpirun_rsh -export ... ./a.out
```

[MVAPICH :: GDR Userguide \(ohio-state.edu\)](http://mvapich.cse.ohio-state.edu/userguide/gdr/) <http://mvapich.cse.ohio-state.edu/userguide/gdr/>

※スーパーコンピュータなどシステムにより実行方法が
指定されている場合はそちらに従ってください

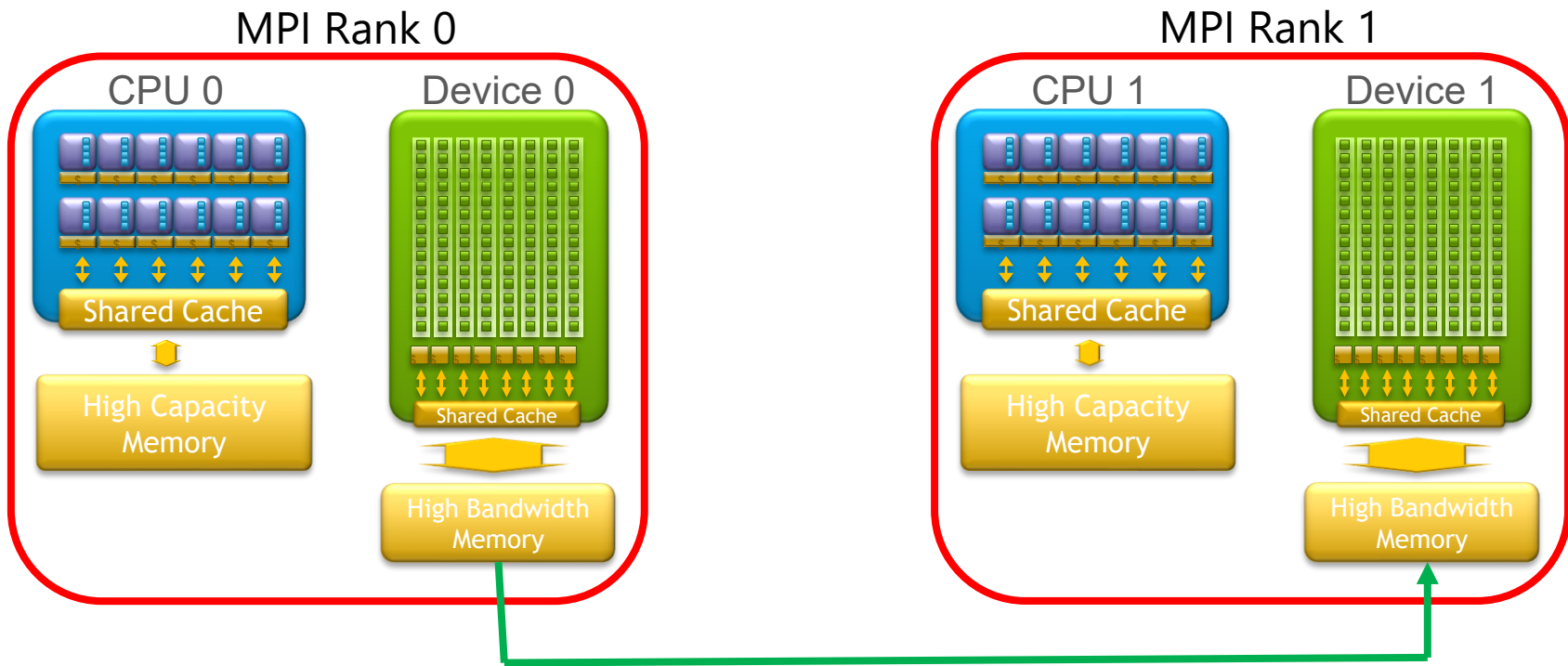
MPIによるホストデータの通信

```
MPI_Send( /* From Rank 0 -> Rank 1 */ );
```



MPIによるデバイスデータの通信

```
#pragma acc host_data use_device(array)  
MPI_Send( /* From Rank 0 -> Rank 1 */ );
```



5. 開発における実装方針のススメ

注意事項

- あくまでもプロメテックの見解です
- 共同研究者・チームメンバーらと相談の上、進めてください
- 例えばHPC専任の開発者と相談するのも良いと思われます

前提条件

- 基本的には書きやすさ（保守性）優先、性能はもちろん欲しいがそのために極端に最適化する必要はない
- アプリケーション全体をOpenACCで実装することを推奨
- どちらがより皆さんの目的に沿うでしょうか？
 1. 計算機が持つ100%の性能を得られる、ただしコードが複雑で保守できない
 2. 計算機が持つ70%の性能を得られる、ただしコードは慣れた言語で実装

0. コンパイラの動作確認・結果検証

- 大半はIntel Fortranないしベンダ提供の Fortran と思います
- NVIDIA Fortran で計算結果・誤差に問題がないか検証が必要
- また、最適化オプション等は各コンパイラで書き方がまちまち

(一例)	ifort	nvfortran	nfort	firt/frtpx
最適化	-Ofast or -fast	-fast	-O3	-Kfast
OpenMP	-qopenmp	-mp	-fopenmp	-Kopenmp
アーキテクチャ	(-march=icelake)	(-march=skylake or -tp=skylake)	(Aurora TSUBASA)	(armv8.2+SVE)

1. OpenACCオプションの付与

- -accで計算結果が正しいか
- -mp -acc両方をつけて計算結果が正しいか
- -gpu=managed (managed memory) を使ってプログラムが動作するか
- 動作するなら、-gpu=managedを指定した状態で次へ

2. OpenACC化

- まずは比較的重そうな単純ループについてacc kernelsで並列化していく、明示的なデータ転送が必要な場合がある
- Fortranの場合はshaped-arrayであれば自動的に転送されると考えられる
- cuBLASやcuFFTが使えるような場合はacc host_data/use_deviceを使ってCUDAライブラリを呼び出す
- 行列積（cuBLAS）, 疎行列ベクトル積（cuSPARSE）, 1-3D FFT（cuFFT）

3. データ転送の最適化

- acc dataやacc enter/exit dataを使ってデータが本当に必要になる場合にHost-Device間転送を有効化していく
- managed memoryを使っている場合はオプションを外す
- いつどのタイミングでデータ転送が走っているかは、NVCOMPILER_ACC_NOTIFY環境変数に3を指定するとわかる
- <https://docs.nvidia.com/hpc-sdk/compiler/openacc-gs/index.html#fortran-examples>

4. CUDAカーネルの実装

- 本当に性能が必要なときのみ使う
- 最適化はできるが保守性が極端に落ちる
- 誰も手を加えられなくても大丈夫（枯れている）場合は問題ないが、頻繁に書き換える場合だと手がつけられなくなる

まとめ

本講習会で紹介した内容

- ループの最適化
- 非同期処理
- CUDAとの連携（Interoperability）
- マルチGPU計算の基礎
- 開発における実装方針のススメ

➤ OpenACCを用いたプログラムの開発イメージはつかめてきたでしょうか？

弊グループのHPCサポート

- 弊グループ（プロメテック・ソフトウェア、GDEPソリューションズ）にて下記のHPC関連のサポートをしています。
- GPUサーバ <https://www.gdep-sol.co.jp/>
- NVIDIA HPCコンパイラサポートサービス <https://hpcworld.jp/support/>

THANK YOU

The free lunch is over. - Herb Sutter