

# OpenACCの紹介・ Reedbush-Hお試し

---

東京大学情報基盤センター 准教授 塙 敏博

2020年1月14日(火)10:25 – 12:10



レポートおよびコンテスト課題  
(締切:  
2020年2月3日(月)24時 厳守

# 講義日程(工学部共通科目)

- ~~1.~~ ~~9月24日: ガイダンス~~
- ~~2.~~ ~~10月1日~~
  - ~~● 並列数値処理の基本演算(座学)~~
- ~~3.~~ ~~10月8日: スパコン利用開始~~
  - ~~● ログイン作業、テストプログラム実行~~
- ~~4.~~ ~~10月15日~~
  - ~~● 高性能プログラミング技法の基礎1  
(階層メモリ、ループアンローリング)~~
- ~~5.~~ ~~10月29日~~
  - ~~● 高性能プログラミング技法の基礎2  
(キャッシュブロック化)~~
- ~~6.~~ ~~11月5日~~
  - ~~● 行列ベクトル積の並列化~~
- ~~7.~~ ~~11月12日~~
  - ~~● ベキ乗法の並列化~~
- ~~8.~~ ~~11月26日~~
  - ~~● 行列-行列積の並列化(1)~~
- ~~9.~~ ~~12月3日~~
  - ~~● 行列-行列積の並列化(2)~~
- ~~10.~~ ~~12月10日~~
  - ~~● LU分解法(1)~~
  - ~~● コンテスト課題発表~~
- ~~11.~~ ~~12月17日~~
  - ~~● LU分解法(2)~~
- ~~12.~~ ~~1月7日~~
  - ~~● LU分解法(3)、非同期通信~~
13. 1月14日
  - RB-Hお試し、研究紹介他

# GPUプログラミングの紹介

---

一部は本センターのGPU講習会資料から、です。

# GPUを使った汎用計算: GPGPU

- GPU: Graphic Processing Unit, 画像処理用のハードウェア
  - 高速な描画、3次元画像処理など
- 3次元画像処理などに用いる演算を汎用計算に応用
  - 多数のピクセル(画素)に対して高速に計算するために、**多数の演算器で並列処理** ➡ **数値計算に**
- 10年程度で歴史は浅いが、HPCでは広く使われている
  - 最近では機械学習(Deep Learning)の主役に

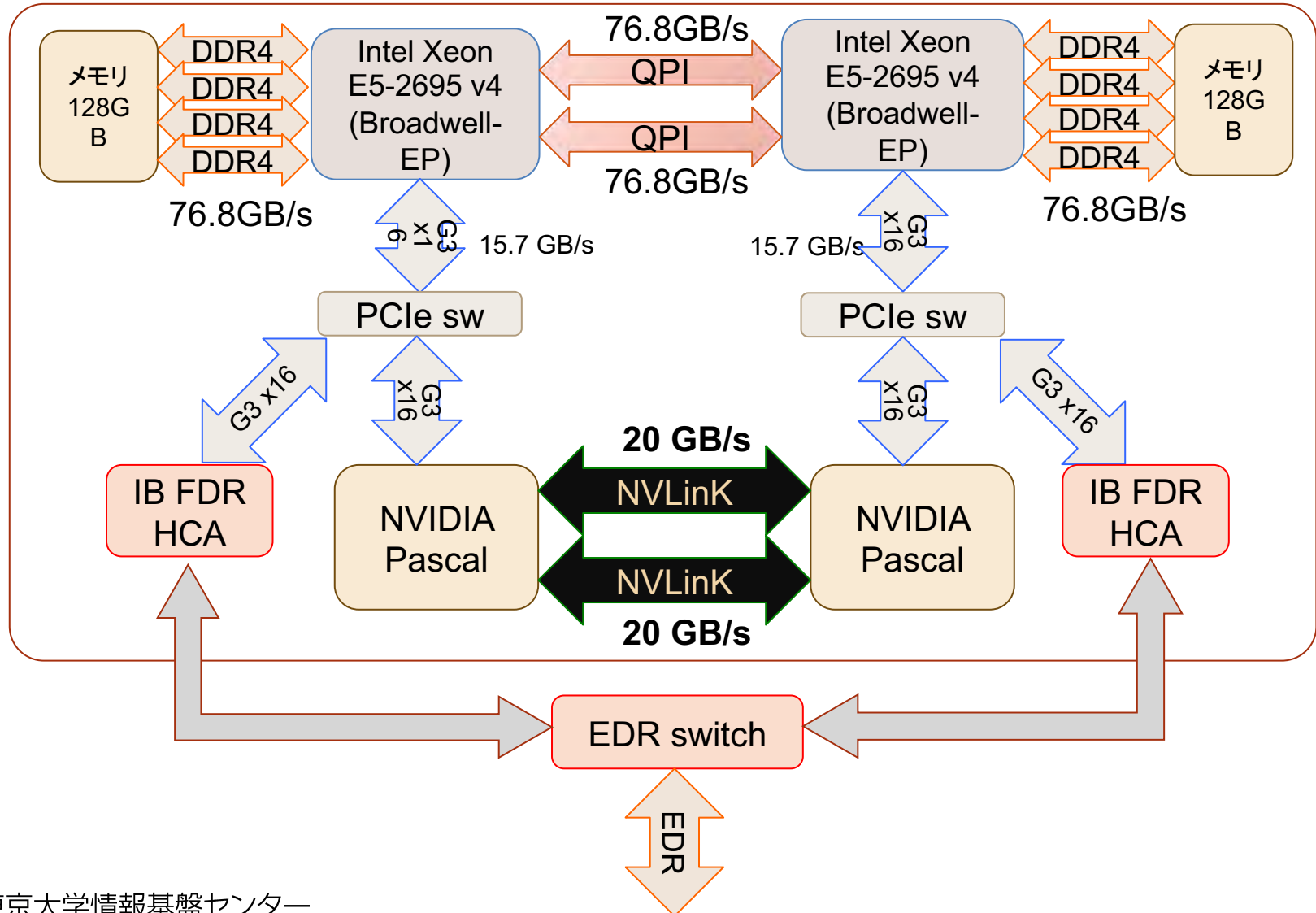
# Green 500 Ranking (Nov., 2019)

<http://www.top500.org/>

	TOP 500 Rank	System	Cores	HPL Rmax (Pflop/s)	Power (kW)	GFLOPS/W
1	159	A64FX Prototype, Fujitsu, Japan	36,864	1.999	118	16.876
2	420	NA-1, PEZY, Japan	1,271,040	1.303	80	16.256
3	24	AiMOS, USA	130,000	8.045	510	15.771
4	373	Satori, USA	23,040	1.464	94	15.574
5	1	Summit, USA	2,414,592	148.6	10,096	14.719
6	8	ABCI, Japan	391,680	19,880	1,649	14.423
7	494	MareNostrum P9 CTE, Spain	18,360	1,145	81	14.131
8	23	TSUBAME 3.0, Japan	135,828	8,125	792	13.704
9	11	PANGEA III, France	291,024	17,860	1,367	13.065
10	2	Sierra, USA	1,572,480	94,640	7,438	12.723
13	June'18	Reedbush-L, U.Tokyo, Japan	16,640	806	79	10.167
19		Reedbush-H, U.Tokyo, Japan	17,760	802	94	8.576



# Reedbush-Hノードのブロック図



# なぜGPUコンピューティング？

- 性能が高いから！

	P100	BDW	KNL
動作周波数(GHz)	1.480	2.10	1.40
コア数(有効スレッド数)	3,584	18 (18)	68 (272)
理論演算性能(GFLOPS)	<b>5,304</b>	604.8	3,046.4
主記憶容量(GB)	16	128	16
メモリバンド幅 (GB/sec., Stream Triad)	534	65.5	490
備考	Reedbush-Hの GPU	Reedbush-U/Hの CPU	Oakforest-PACSの CPU (Intel Xeon Phi)

# GPUプログラミングは何が難しい？

- CPU: **大きなコア**を**いくつか**搭載
  - Reedbush-H の CPU : 2.10 GHz 18コア
  - 大きなコア... 分岐予測、パイプライン処理、Out-of-Order
    - 要はなんでもできる
  - 逐次の処理が得意
- GPU: **小さなコア**を**たくさん**搭載
  - Reedbush-H の GPU: 1.48 GHz **3,584** コア
  - 小さなコア... 上記機能が弱い, またはない!
  - **並列処理が必須**

## GPUの難しさ

1. 並列プログラミング自体の難しさ
2. 多数のコアを効率良く扱う難しさ



# 参考: NVIDIA Tesla P100

- 56 SMs
- 3584 CUDA Cores
- 16 GB HBM2



P100 whitepaperより

# 参考: NVIDIA Tesla P100 の SM

GPU TECHNOLOGY  
CONFERENCE

## GP100 SM

### GP100

CUDA Cores	64
Register File	256 KB
Shared Memory	64 KB
Active Threads	2048
Active Blocks	32



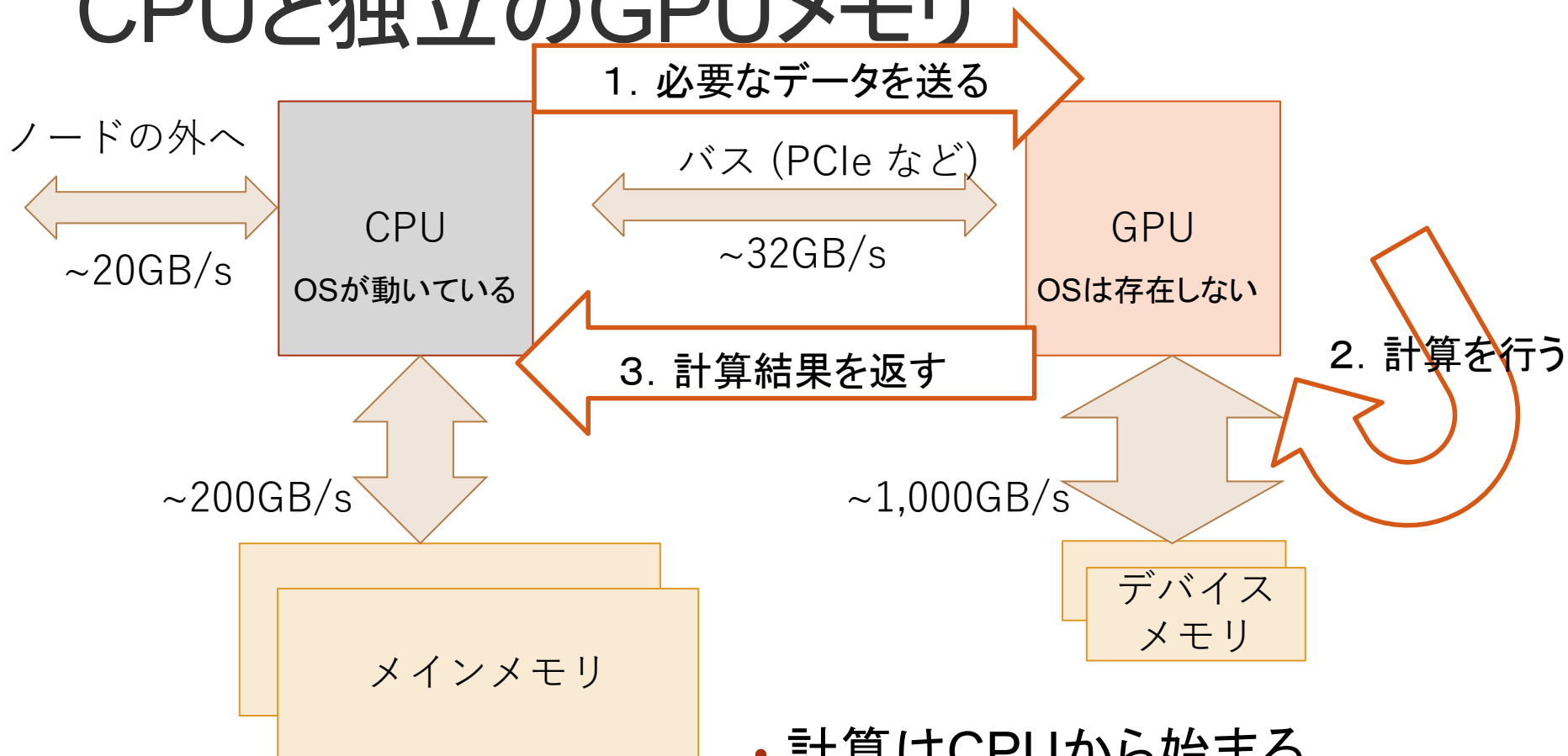
# 参考：NVIDIA社のGPU

- 製品シリーズ
  - GeForce
    - コンシューマ向け。安価。
  - Tesla
    - HPC向け。倍精度演算器、大容量メモリ、ECCを備えるため高価。
- アーキテクチャ(世代)
  1. Tesla: 最初のHPC向けGPU、TSUBAME1.2など
  2. Fermi: 2世代目、TSUBAME2.0など
    - ECCメモリ、FMA演算、L1 L2 キャッシュ
  3. Kepler: 現在HPCにて多く利用、TSUBAME2.5など
    - シャッフル命令、Dynamic Parallelism、Hyper-Q
  4. Maxwell: コンシューマ向けのみ
  5. Pascal: 最新GPU、Reedbush-HIに搭載
    - HBM2、半精度演算、NVLink、倍精度atomicAdd など
  6. Volta: 次世代GPU
    - Tensor Coreなど

# 押さえておくべきGPUの特徴

- CPUと独立のGPUメモリ
- 性能を出すためにはスレッド数 $\gg$ コア数
- 階層的スレッド管理と同期
- Warp 単位の実行
- やってはいけないWarp内分岐
- コアレスドアクセス

# CPUと独立のGPUメモリ



- 計算はCPUから始まる
- 物理的に独立のデバイスメモリとデータのやり取り必須

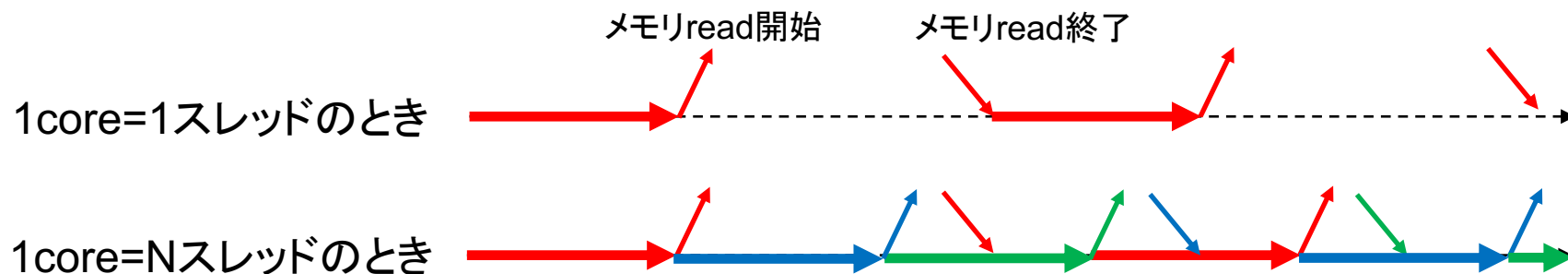
# 性能を出すためにはスレッド数>>コア数

## • 推奨スレッド数

- CPU: スレッド数=コア数 (高々数十スレッド)
- GPU: スレッド数>=コア数\*4~ (数万~数百万スレッド)
  - 最適値は他のリソースとの兼ね合いによる

## • 理由: 高速コンテキストスイッチによるメモリレイテンシ隠蔽

- CPU: レジスタ・スタックの退避はOSがソフトウェアで行う(遅い)
- GPU: ハードウェアサポートでコストほぼゼロ
  - メモリアクセスによる暇な時間(ストール)に他のスレッドを実行



# 階層的スレッド管理と同期

- コアの管理に対応
  - 1 SM の中に 64 CUDA core、56 SM で 3584 CUDA core
  - 1 CUDA core が複数スレッドを担当
- スレッド間の同期
  - 同一SM内のスレッド間は同期できる
    - 正確には同ースレッドブロック内
  - 異なるSMのスレッド間は同期できない
    - 同期するためにはGPUの処理を終了する必要あり
    - atomic 演算は可能
- メモリ資源の共有
  - L1 cache, shared memory, Instruction cache などはSM内で共有
  - L2 cache, Device memory などは全スレッドで共有

値はP100の場合

# Warp 単位の実行

- 連続した32スレッドを1単位 = Warp と呼ぶ
- このWarpは足並み揃えて動く
  - 実行する命令は32スレッド全て同じ
  - データは違っていい

Volta世代からは実装が変わったので注意:

各スレッドが独立して動作可能  
但し資源が競合すれば待つことには変わらない

スレッド 1 2 3 ... 31 32

配列 A

4	3	5	...	8	0
---	---	---	-----	---	---

× × × ... × ×

配列 B

2	3	1	...	1	9
---	---	---	-----	---	---

**OK !**

スレッド 1 2 3 ... 31 32

配列 A

4	3	5	...	8	0
---	---	---	-----	---	---

÷ × + ... - ×

配列 B

2	3	1	...	1	9
---	---	---	-----	---	---

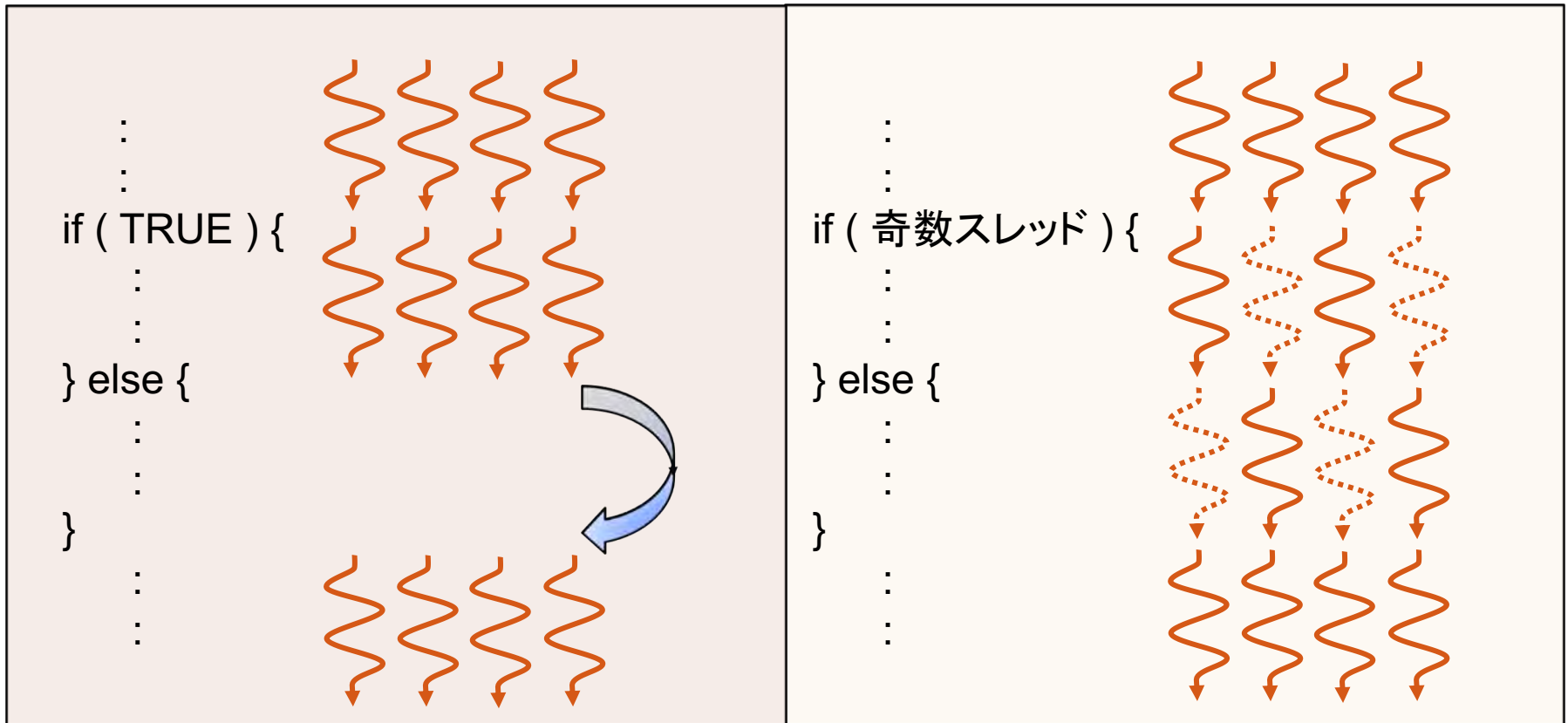
**NG !**



# やってはいけないWarp内分岐

- Divergent Branch

- Warp 内で分岐すること。Warp単位の分岐ならOK。

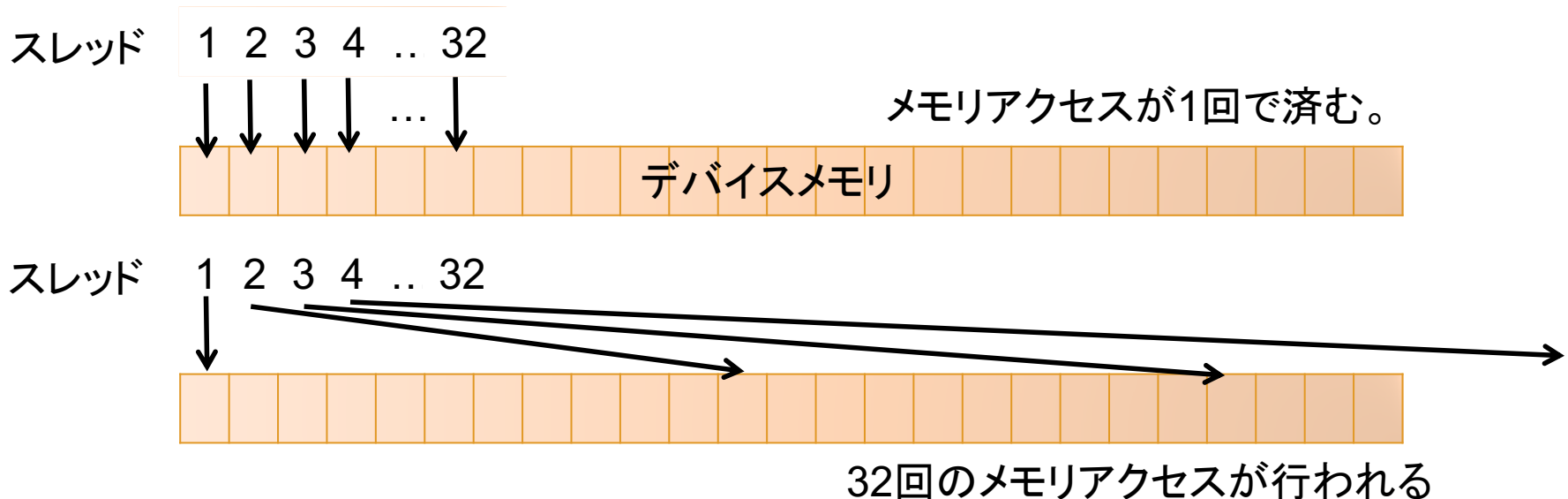


else 部分は実行せずジャンプ

一部スレッドを眠らせて全分岐を実行  
最悪ケースでは32倍のコスト

# コアレスドアクセス

- 同じWarp内のスレッドが近いアドレスに同時にアクセスするのがメモリの性質上効率的
  - これをコアレスドアクセス(coalesced access)と呼ぶ



128バイト単位でメモリアクセス。Warp内のアクセスが128バイトに収まってれば1回。外れればその分だけ繰り返す。**最悪ケースでは32倍のコスト**

# GPU向けプログラミング環境

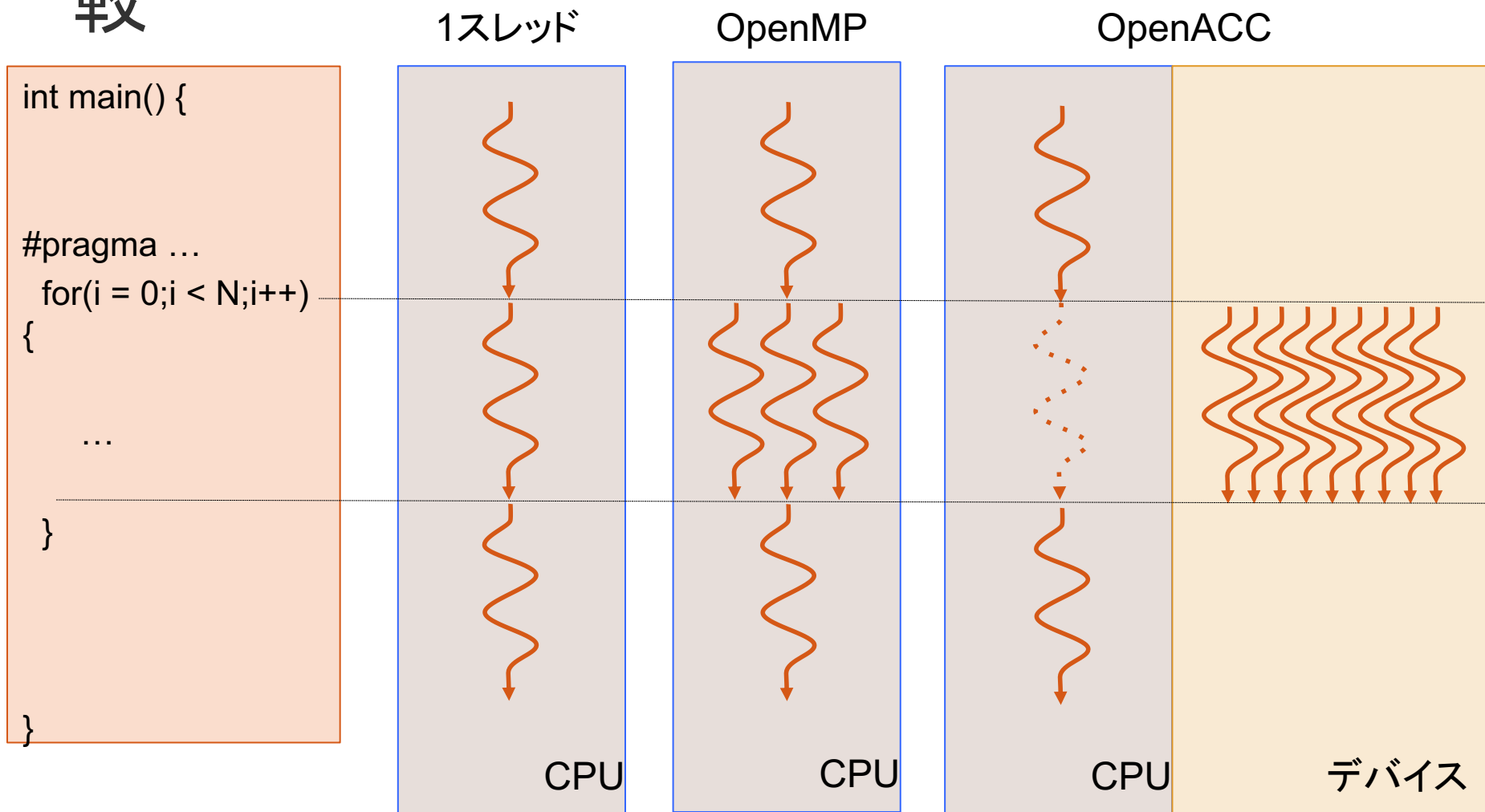
- **CUDA** (Compute Unified Device Architecture)
  - NVIDIAのGPU向け開発環境。C言語版は**CUDA C**としてNVIDIAから、Fortran版は**CUDA Fortran**としてPGI(現在はNVIDIAの子会社)から提供されている。
- **OpenACC**: 指示文を用いて並列化を行うプログラミング環境。C言語とFortranの両方の仕様が定められている。PGIコンパイラなど幾つかのコンパイラが対応。(GPUが主なターゲットだが)GPU専用言語ではない。
  - (特に単純なプログラムにおいては)OpenACCでもCUDAでも同様の性能が出ることもあるが、一般的にはCUDAの方が高速
  - レガシーコードがある場合はOpenACCで書く方がはるかに楽

# 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
    - PGI は無料版も出している
  - 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

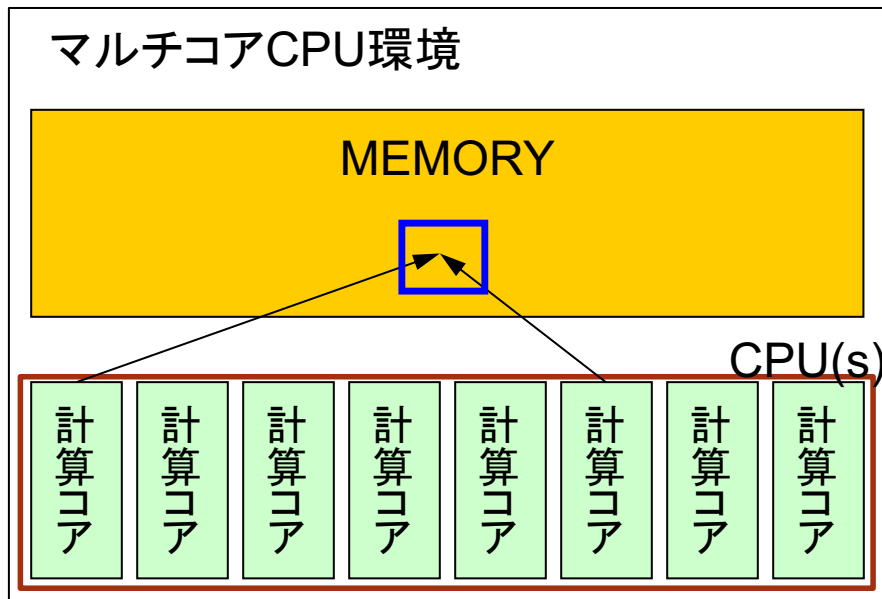
RB-HではPGIコンパイラを用いる

# OpenACC と OpenMP の実行イメージ比較



# OpenACC と OpenMP の比較

## OpenMPの想定アーキテクチャ



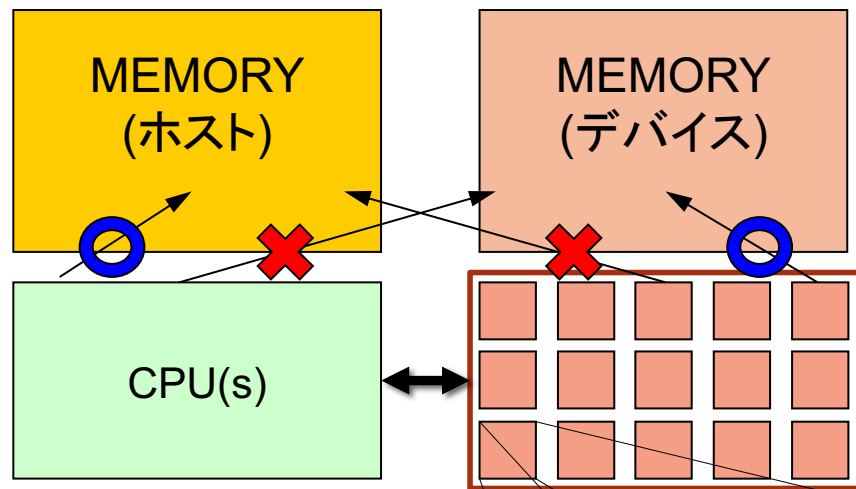
- 計算コアがN個
  - $N < 100$  程度 (Xeon Phi除く)
- 共有メモリ

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

アクセラレータを備えた計算機環境



- 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ループ並列化
- 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

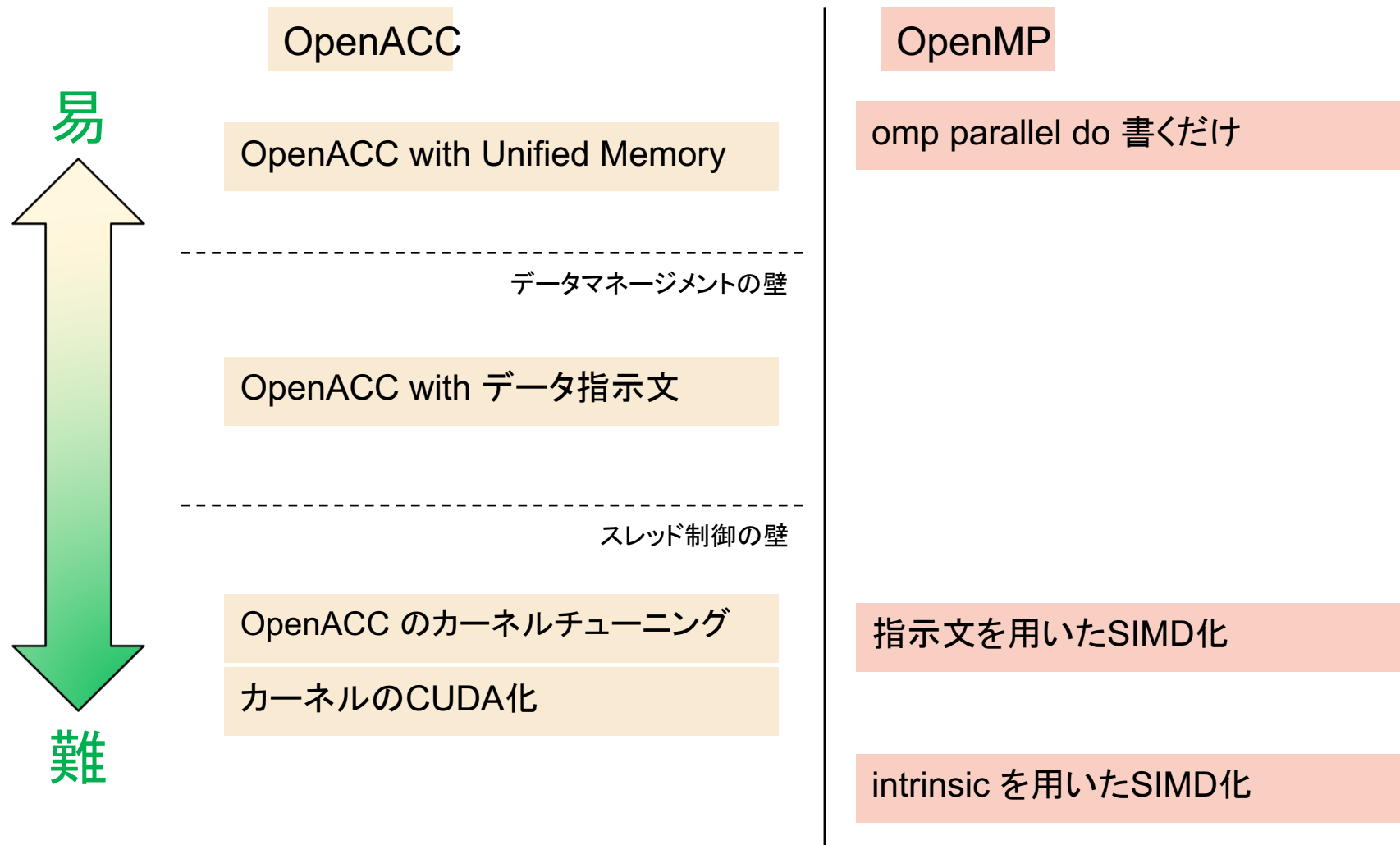


# OpenACC と OpenMP の比較

## デフォルトでの変数の扱い

- OpenMP
  - 全部 shared
- OpenACC
  - スカラ変数: firstprivate or private
  - 配列: shared
    - 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
    - 正しく転送されないこともある。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
    - 配列はデバイスに確保される (shared的振る舞い)
    - 配列変数をprivateに扱うためには private 指示節使う

# GPUプログラミング難易度早見表



# Reedbush-Hの利用開始

---

OFFPとの違いを中心に

# 鍵の登録(1/2)

1. ブラウザを立ち上げる
2. 以下のアドレスを入力する  
<https://reedbush-www.cc.u-tokyo.ac.jp/>
3. 「ユーザ名」にセンターから配布された、“利用者番号”をいれる。
4. 「パスワード」に、センターから配布された“パスワード”を入力する。

# Reedbushへログイン

- ターミナルから、以下を入力する  
`$ ssh reedbush.cc.u-tokyo.ac.jp -l tYYxxx`  
「-l」はハイフンと小文字のL、  
「tYYxxx」は利用者番号(数字)  
“tYYxxx”は、利用者番号を入れる
- 接続するかと聞かれるので、yes を入れる
- 鍵の設定時に入れた  
自分が決めたパスワード(パスフレーズ)  
を入れる
- 成功すると、ログインができる

# バッチキューの設定のしかた

- バッチ処理は、Altair社のバッチシステム PBS Professional で管理されています。
- 以下、主要コマンドを説明します。
  - ジョブの投入：  
`qsub <ジョブスクリプトファイル名>`
  - 自分が投入したジョブの状況確認：`rbstat`
  - 投入ジョブの削除：`qdel <ジョブID>`
  - バッチキューの状態を見る：`rbstat --rsc`
  - バッチキューの詳細構成を見る：`rbstat -rsc -x`
  - 投げられているジョブ数を見る：`rbstat -b`
  - 過去の投入履歴を見る：`rbstat -H`
  - 同時に投入できる数／実行できる数を見る：`rbstat --limit`

OFPとの対応:

pjsub => qsub  
pjstat => rbstat

# JOBスクリプトサンプルの説明(ピュアMPI)

(hello-pure.bash, C言語、Fortran言語共通)

```
#!/bin/bash
#PBS -q h-lecture
#PBS -Wgroup_list=gt35
#PBS -l select=8:mpiprocs=36
#PBS -l walltime=00:01:00
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh
mpirun ./hello
```

リソースグループ名  
: h-lecture

利用グループ名  
: gt35

利用ノード数

ノード内利用コア数  
(MPIプロセス数)

実行時間制限  
: 1分

MPIジョブを  $8 * 36 = 288$  プロセス  
で実行する。

カレントディレクトリ設定、環境変  
数設定(必ず入れておく)

# 本講義でのキュー名

- 本演習中のキュー名：
  - h-lecture5
  - 最大10分まで
  - 最大ノード数は2ノード(4GPU) まで
- 本演習時間以外(24時間)のキュー名：
  - h-lecture
  - 利用条件は演習中のキュー名と同様



# Reedbushにおける注意

- /home ファイルシステムは容量が小さく、ログインに必要なファイルだけを置くための場所です。
  - /home に置いたファイルは計算ノードから参照できません。ジョブの実行もできません。
- => ログイン後は /lustre ファイルシステムを使ってください。

- ホームディレクトリ: /home/gt35/t35XXX
  - cd コマンドで移動できます。
- Lustreディレクトリ: /lustre/gt35/t35XXX
  - cdw コマンドで移動できます。

# OpenACC の指示文

---

# OpenACC の主要な指示文

- 並列領域指定指示文
  - kernels, parallel
- データ移動最適化指示文
  - data, enter data, exit data, update
- ループ最適化指示文
  - loop
- その他、比較的よく使う指示文
  - host\_data, atomic, routine, declare

# 並列領域指定指示文: 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」
  - **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」
  - 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**
    - どちらかというto kernels推奨

# 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

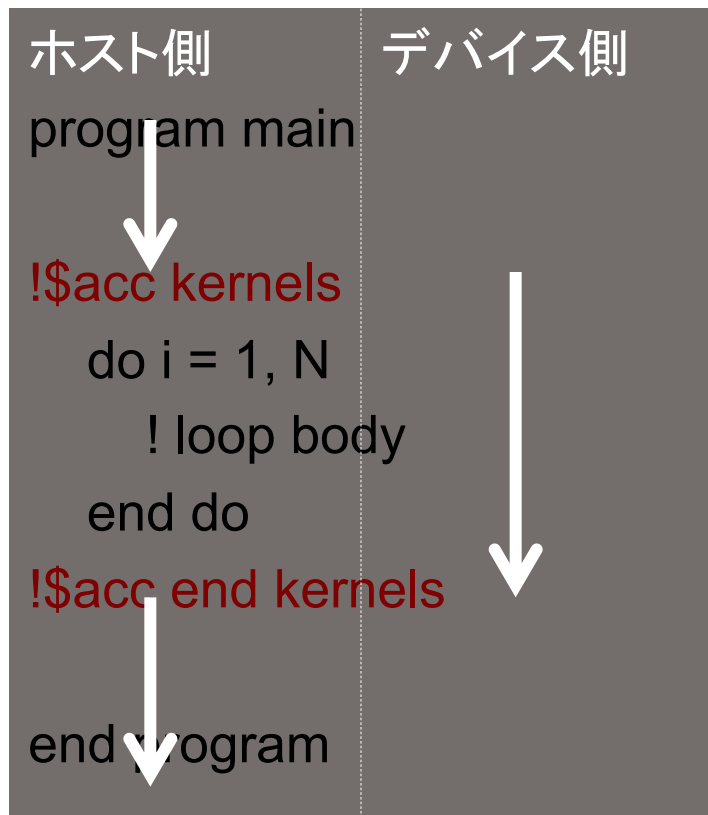
!$acc parallel num_gangs(N)
!$acc loop gang
  do i = 1, N
    ! loop body
  end do
!$acc end parallel

end program
```

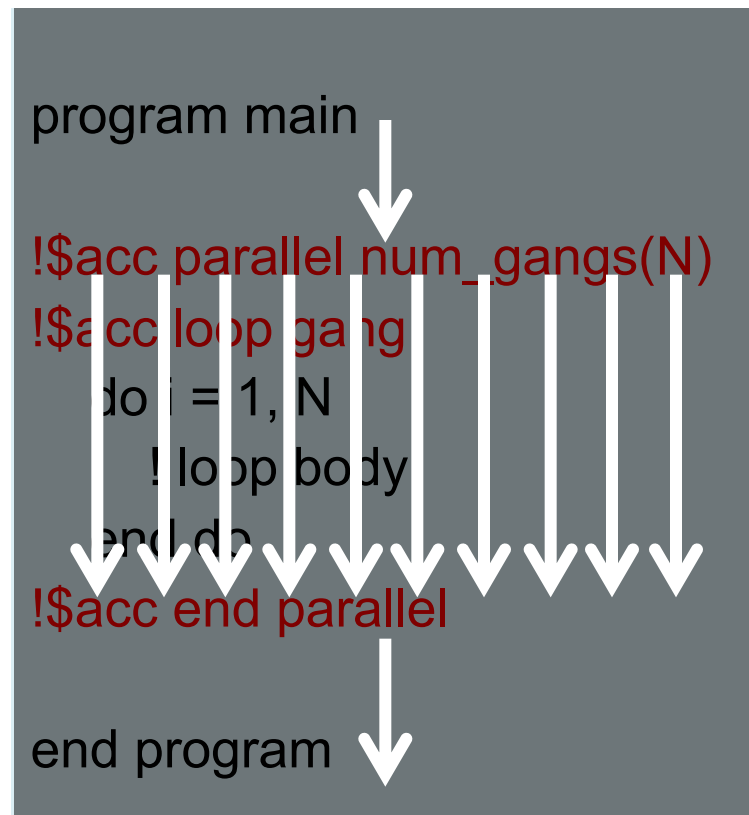
# kernels/parallel 指示文

kernels

- 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 parallel



「並列数はデバイスに合わせてください」



「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

parallel

非同期実行に用いる。

実行デバイス毎にパラメータを調整

データ指示文の機能を使える

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。

- async
- wait
- device\_type
- if
- default(none)
- copy...
- num\_gangs
- num\_workers
- vector\_length
- reduction
- private
- firstprivate



# kernels/parallel 実行イメージ

Fortran

C言語

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

```
!$acc kernels copy(src,dst)
```

```
  do i = 1, N
    dst(i) = src(i)
  end do
```

```
!$acc end kernels
```

```
end subroutine copy
```

```
void copy(float *dst, float *src) {
  int i;
```

```
#pragma acc kernels copy(src[0:N] ¥  
dst[0:N])
```

```
  for(i = 0; i < N; i++){
    dst[i] = src[i];
  }
```

```
}
```

# kernels/parallel 実行イメージ

Fortran

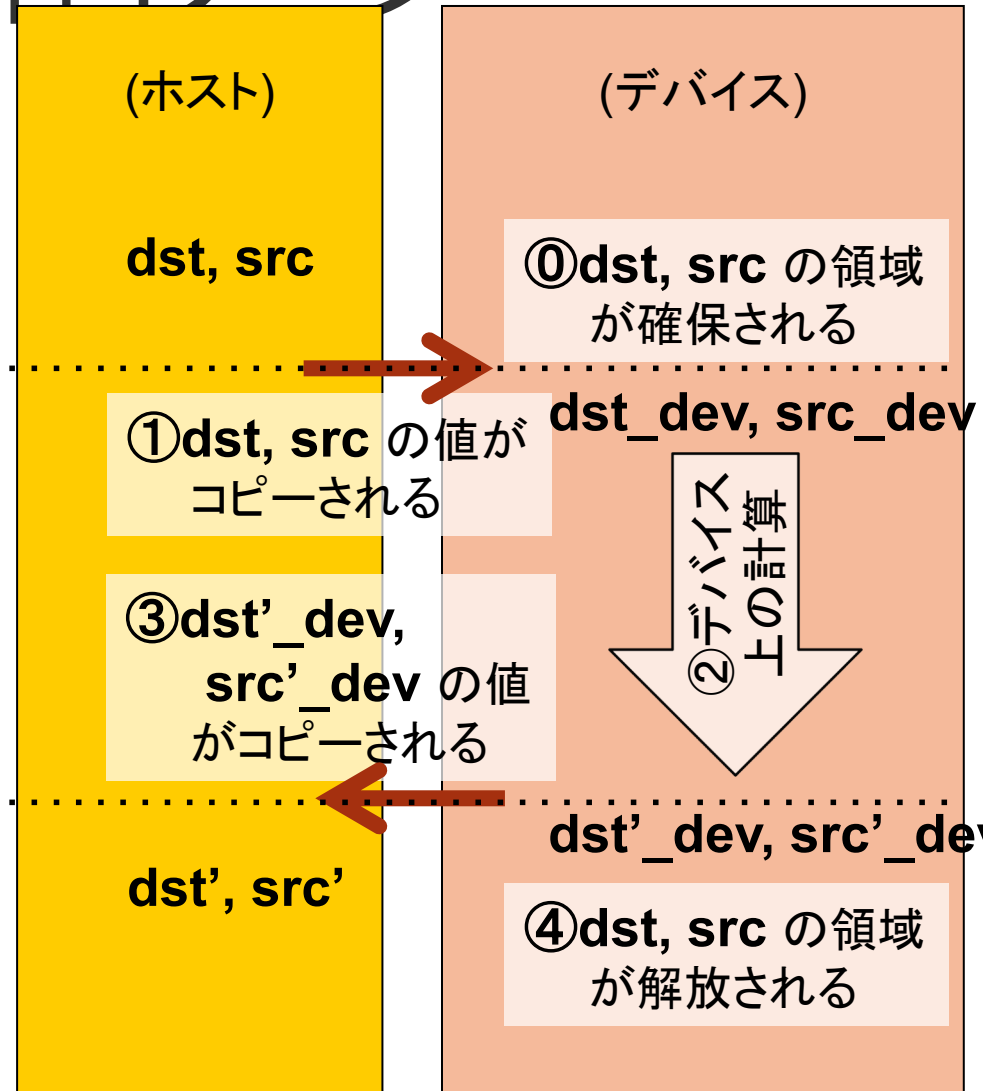
```
subroutine copy(dst, src)
  real(4), dimension(:) :: dst, src
```

```
!$acc kernels copy(src,dst)
```

```
do i = 1, N
  dst(i) = src(i)
end do
```

```
!$acc end kernels
```

```
end subroutine copy
```



# デバイス上で扱うデータについて

- 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
  - 正しく転送されないこともある。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

# データ移動最適化指示文が必要なとき

Fortran

```
subroutine copy(dst, src)
  real(4), dimension(:) :: dst, src
  do j = 1, M
    !$acc kernels copy(src,dst)
    do i = 1, N
      dst(i) = dst(i) + src(i)
    end do
    !$acc end kernels
  end do
end subroutine copy
```

C言語

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

Kernels をループで囲むと、  
HtoD転送=>計算=>DtoH転送  
の繰り返し...

# data指示文

Fortran

```

subroutine copy(dst, src)
  real(4), dimension(:) :: dst, src
  !$acc data copy(src,dst)
  do j = 1, M
  !$acc kernels present(src,dst)
    do i = 1, N
      dst(i) = dst(i) + src(i)
    end do
  !$acc end kernels
  end do
  !$acc end data
end subroutine copy
  
```

**present:** 既に転送済であることを示す

C言語

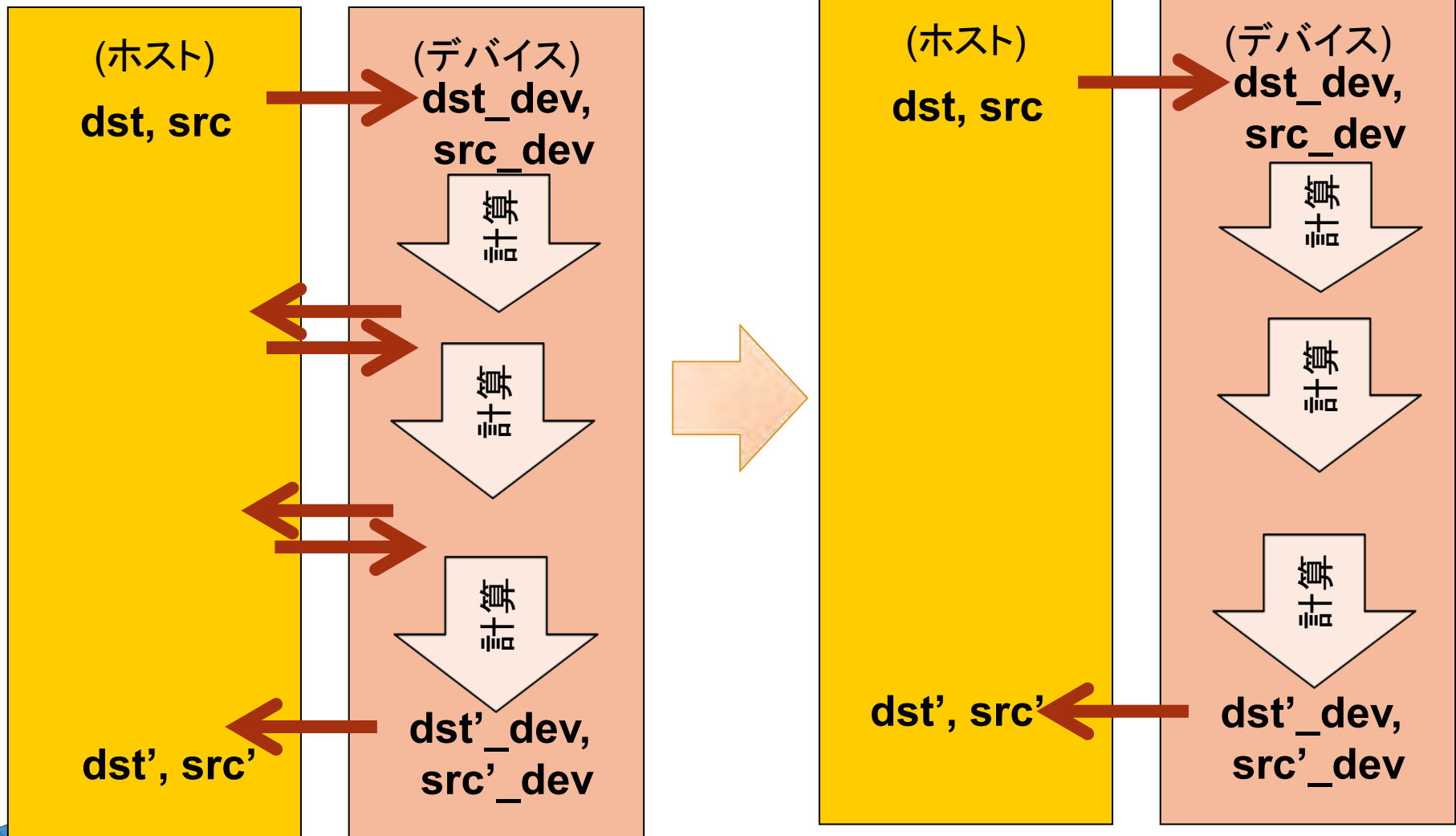
```

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

Cの場合、data指示文の範囲は{}で指定

(この場合はforが構造ブロックになっているのでなくても大丈夫だが)

# data指示文の効果



# データ移動指示文: 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では下限と上限を指定

C版

```
#pragma acc data copy(A[start1:length1][start2:length2])  
{  
...  
}
```

Cでは先頭と長さを指定

# 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
    - パラメータの設定はある程度勝手にやってくれる
  - 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

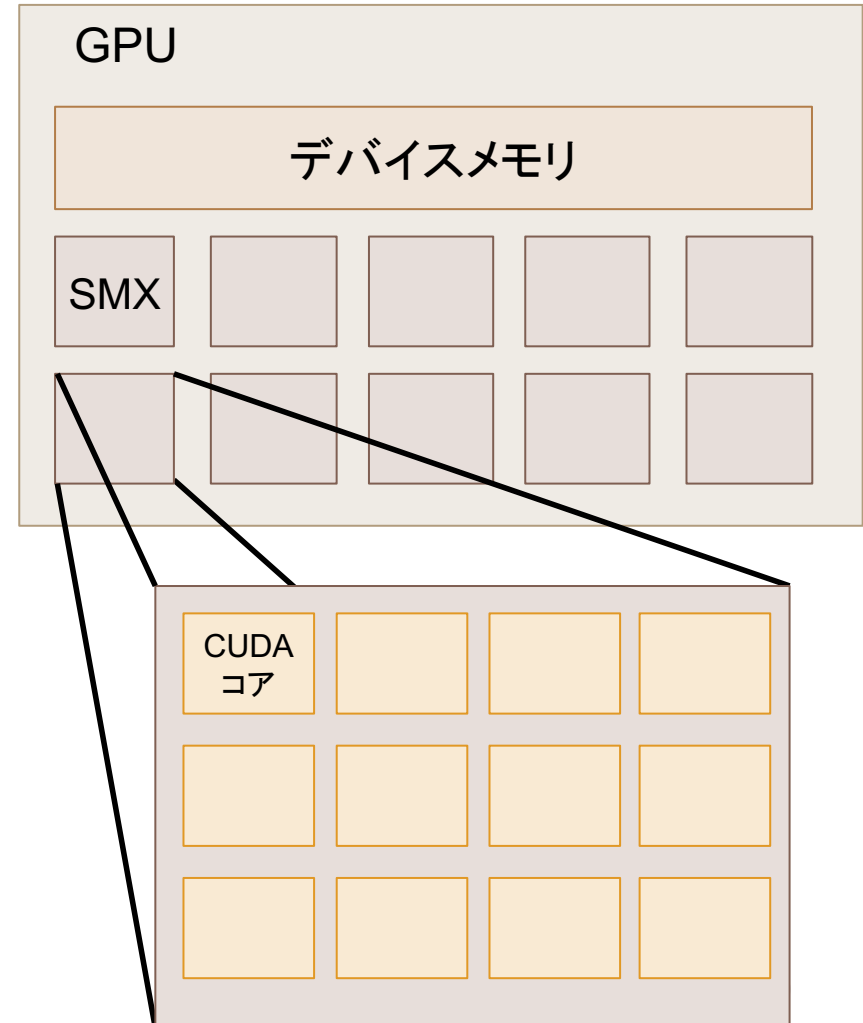
```
!$acc kernels
!$acc loop gang
  do j = 1, n
!$acc loop vector
  do i = 1, n
    cc = 0
!$acc loop seq
    do k = 1, n
      cc = cc + a(i,k) * b(k,j)
    end do
    c(i,j) = cc
  end do
end do
!$acc end kernels
```



# 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
  - 最近では2階層目(SIMD)がある
- CUDAは block と thread の2階層
- NVIDIA 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
  - 様々なアクセラレータに対応するため

- NVIDIA GPUの構成



# OpenACC と Unified Memory

- Unified Memory とは...
  - 物理的に別物のCPUとGPUのメモリをあたかも一つのメモリのように扱う機能
  - Pascal GPUでは**ハードウェアサポート**
    - ページフォルトが起こると勝手にマイグレーションしてくれる
    - Kepler以前も使えるが、ソフトウェア処理なのでひどく遅い
- OpenACC と Unified Memory
  - OpenACCにUnified Memoryを直接使う機能は**ない**
  - PGIコンパイラに managed オプションを与えることで使える
    - `pgfortran -acc -ta=tesla,managed`
  - 使うと**データ指示文が無視され**、代わりにUnified Memoryを使う

# Unified Memoryのメリット・デメリット

- メリット

- データ移動の管理を任せられる
- **ポインタなどの複雑なデータ構造を簡単に扱える**
  - 本来はメモリ空間が分かれているため、ディープコピー問題が発生する

- デメリット

- ページ単位で転送するため、細かい転送が必要な場合には遅くなる
- CPU側のメモリ管理を監視しているので、allocate, deallocateを繰り返すアプリではCPU側が極端に遅くなる

# OpenACCへの アプリケーション移植方法

---

# 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。それでも遅ければ、

5. カーネルのCUDA化
  - スレッド間の相互作用が多いアプリケーションでは、shared memory や shuffle 命令を自由に使えるCUDAの方が圧倒的に有利

# 既にOpenMP化されているアプリケーションのOpenACC化手順

1. !\$omp parallel を !\$acc kernels に機械的に置き換え
2. Unified Memory を使い、とりあえずGPU上で実行
3. コンパイラのメッセージを見ながら、OpenACCカーネルの最適化
4. データ指示文を用いて転送の最適
5. カーネルのCUDA化
  - スレッド間の相互作用が多いアプリケーションでは、shared memory や shuffle 命令を自由に使えるCUDAの方が圧倒的に有利

# データ指示文による最適化手順

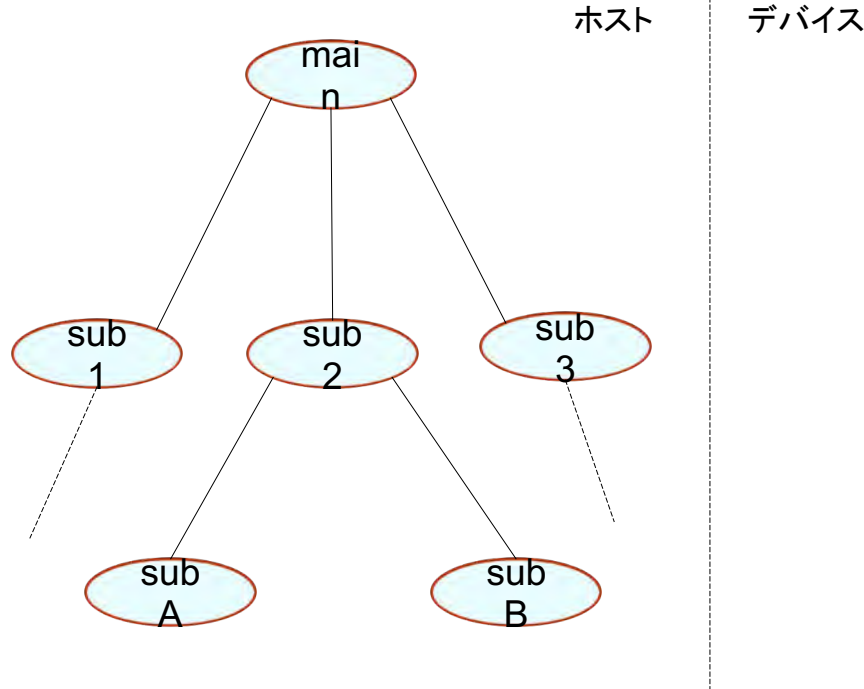
```

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 ) {
    ...
  }
}

```



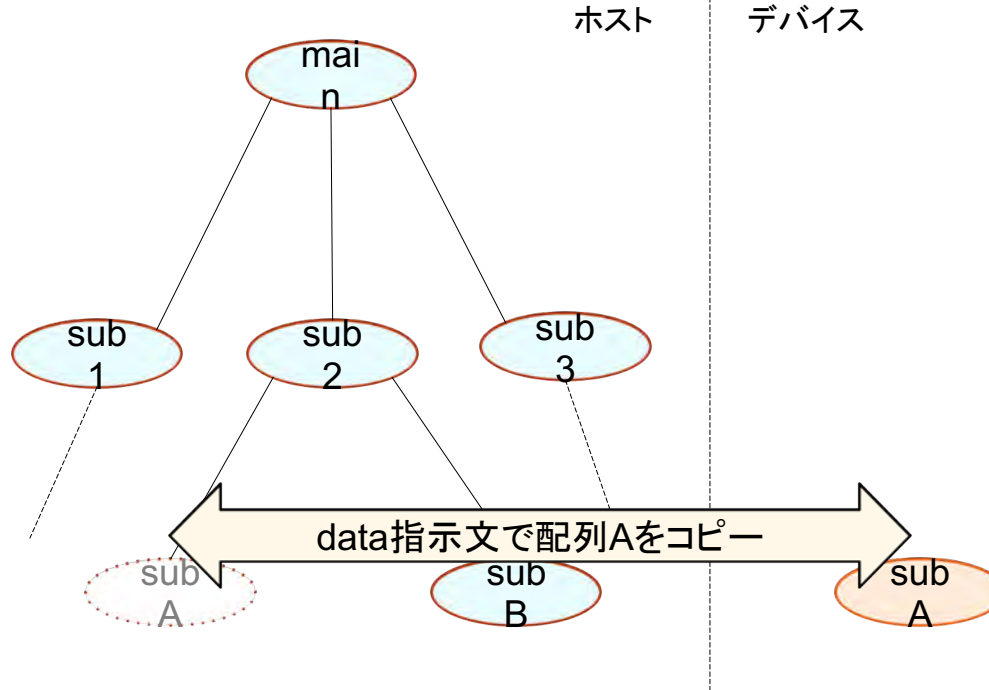
葉っぱの部分から  
OpenACC化を始める

# データ指示文による最適化手順

```
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！



# データ指示文による最適化手順

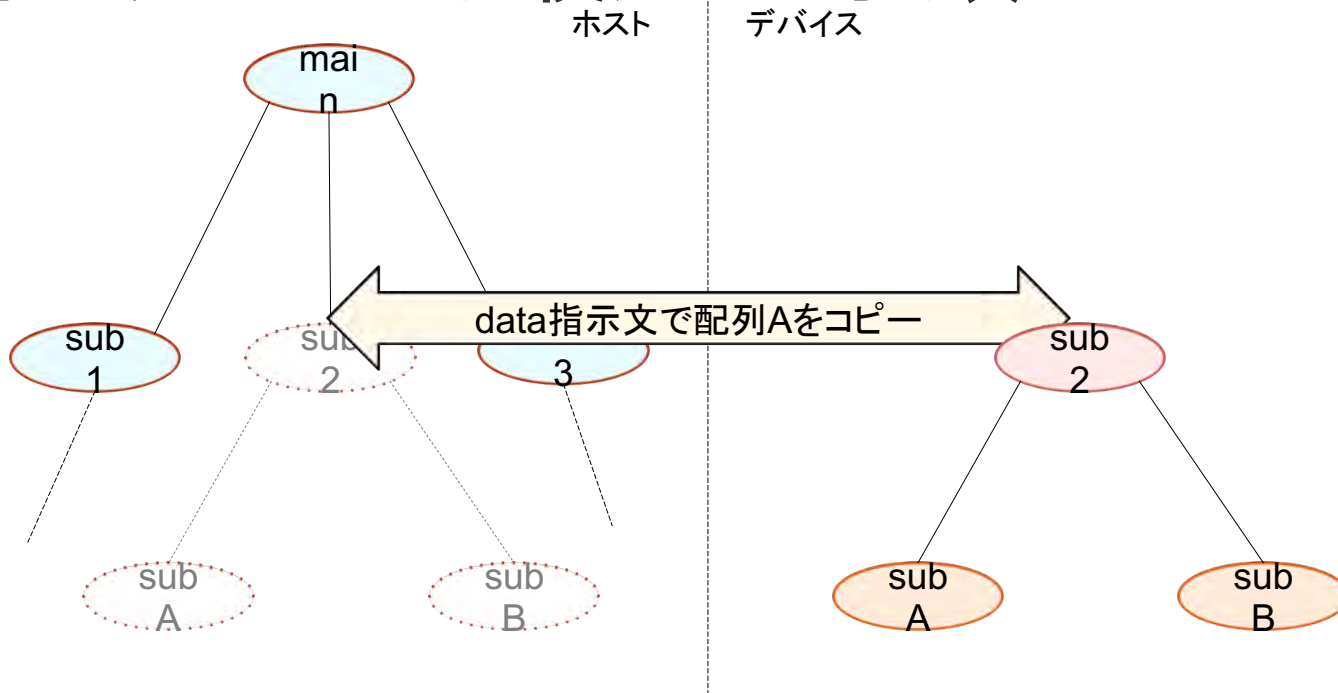
```

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

# データ指示文による最適化手順

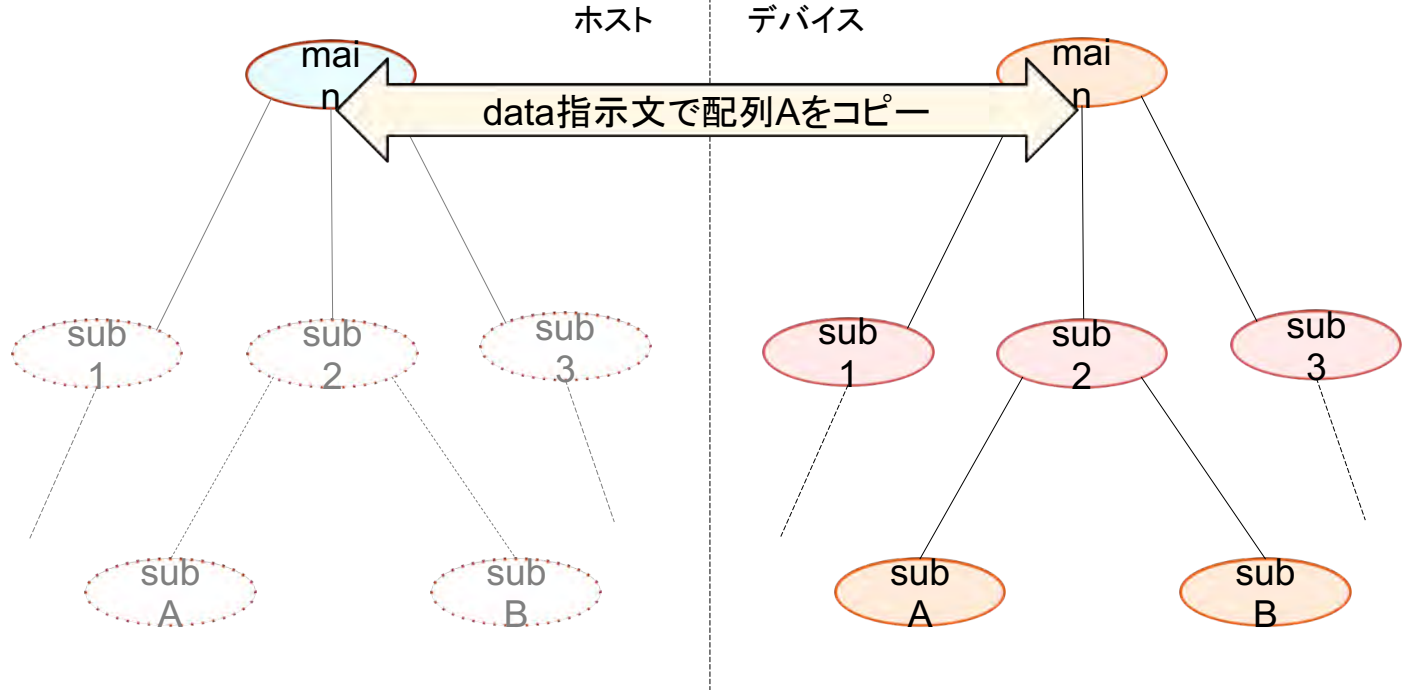
```

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

# PGIコンパイラによるメッセージの確認方法

- 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ったか知るため
  - 感覚としては、Intelコンパイラの最適化レポートを見ながらのSIMD化に近い
  - 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

# よく使うツール群

- PGIコンパイラが出力するレポート
  - pgfortran -Minfo=accel
- 環境変数 PGI\_ACC\_TIME
  - export PGI\_ACC\_TIME=1 で、標準エラーに実行情報が出力される
- NVIDIA Visual Profiler
- cuda-gdb

# PGIコンパイラによるメッセージの確認

- コンパイラオプションとして `-Minfo=accel` を付ける

サブルーチン名

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:

```

配列aはcopyin, bはcopyoutとして扱われます

```

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

```

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\_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

## 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] ← 起動したスレッド数
    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
```

↓カーネル実行時間

← データ移動  
の回数・時間

```
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. }
```

# 参考: moduleコマンドの使い方

- 様々なコンパイラ, MPI環境などを切り替えるためのコマンド
- パスや環境変数など必要な設定が自動的に変更される
- ジョブ実行時にもコンパイル時と同じmoduleをloadすること
  
- 使用可能なモジュールの一覧を表示: **module avail**
- 使用中のモジュールを確認: **module list**
- モジュールのload: **module load** モジュール名
- モジュールのunload: **module unload** モジュール
- モジュールの切り替え: **module switch** 旧モジュール 新モジュール
- モジュールを全てクリア: **module purge**

# コンパイラ等の切替: moduleコマンド

- デフォルトでは, Intelコンパイラ+Intel MPI
  - cf. module list

Currently Loaded Modulefiles:

1) intel/18.1.163      2) intel-mpi/2018.1.163      3) pbsutils

- PGIコンパイラを使う場合: (OpenACCやCUDA Fortran)
    - module switch intel pgi/17.5
  - CUDA開発環境を使う場合
    - module load cuda
    - 別途Cコンパイラも必要
  - MPIを使う場合(コンパイラに追加してload, コンパイラにあったものを選ぶ)
    - module load mvapich2/gdr/2.3a/{gnu,pgi}
    - module load openmpi/gdr/2.1.2/{gnu,intel,pgi}
- ジョブ実行時にも同じmoduleをload
  - 複数組み合わせても良いが, 順序に注意
    - 環境変数PATHやLD\_LIBRARY\_PATHなどを確認



# サンプルプログラムの実行 (行列-行列積OpenACC)

---

# 行列-行列積のサンプルプログラム (OpenACC版)の注意点

- C言語版およびFortran言語版のファイル名  
**Mat-Mat-acc.tar.gz**
- ジョブスクリプトファイル**mat-mat-acc.bash**  
中の  
キュー名を **h-lecture** から **h-lecture5**  
グループ名を **gt35**  
に変更し、qsub してください。
  - **h-lecture** : 実習時間外のキュー
  - **h-lecture6**: 実習時間内のキュー
  - **Reedbush-H**では、キュー名は“h-”で始まる

# 行列-行列積のサンプルプログラムの実行

- 以下のコマンドを実行する

```
$ cdw
```

```
$ cp /lustre/gt35/z30105/Mat-Mat-acc.tar.gz ./
```

```
$ tar xvfz Mat-Mat-acc.tar.gz
```

```
$ cd Mat-Mat-acc
```

- 以下のどちらかを実行

```
$ cd C : C言語を使う人
```

```
$ cd F : Fortran言語を使う人
```

- 以下は共通

```
$ module switch intel pgi/19.10
```

```
$ make
```

```
$ qsub mat-mat-acc.bash
```

- 実行が終了したら、以下を実行する

```
$ cat mat-mat-acc.bash.oXXXXXX
```

# 行列-行列積のコードのOpenMP化の解答 (C言語)

- 以下のようなコードになる

```
#pragma omp parallel for private (j, k)
for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
        for(k=0; k<n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

# 行列-行列積のコードのOpenACC化

- すべてGPU上で実行

```
#pragma acc kernels copyin(A[0:N][0:N], B[0:N][0:N]) copyout(C[0:N][0:N])
#pragma acc loop independent gang
    for(i=0; i<n; i++) {
#pragma acc loop independent vector
        for(j=0; j<n; j++) {
            double dtmp = 0.0;
#pragma acc loop seq
                for(k=0; k<n; k++) {
                    dtmp += A[i][k] * B[k][j];
                }
            C[i][j] = dtmp;
        }
    }
}
```

# 行列-行列積のコードのOpenMP化の解答 (Fortran言語)

- 以下のようなコードになる

```
!$omp parallel do private (j, k)
do i=1, n
  do j=1, n
    do k=1, n
      C(i, j) = C(i, j) + A(i, k) * B(k, j)
    enddo
  enddo
enddo
```

# 行列-行列積のコードのOpenACC化 (Fortran言語)

- すべてGPU上で実行

```
!$acc kernels copyin(A,B) copyout(C)
!$acc loop independent gang
do i=1, n
!$acc loop independent vector
  do j=1, n
    dtmp = 0.0d0
!$acc loop seq
    do k=1, n
      dtmp = dtmp + A(i, k) * B(k, j)
    enddo
    C(i,j) = dtmp
  enddo
enddo
!$acc end kernels
```

# 行列-行列積のサンプルプログラムの実行

- 以下のような結果が見えれば成功

N = 8192

Mat-Mat time = 8.184022 [sec.]

134348.567798 [MFLOPS]

OK!

実際にはデータ転送の時間が含まれている。  
正味の計算時間は `mat-mat-acc.bash.e*` にある

MyMatMat NVIDIA devicenum=0

time(us): 6,864,586

つまり160GFLOPS



# Reedbushを用いた成果例

---

# ChainerMN

- Chainer



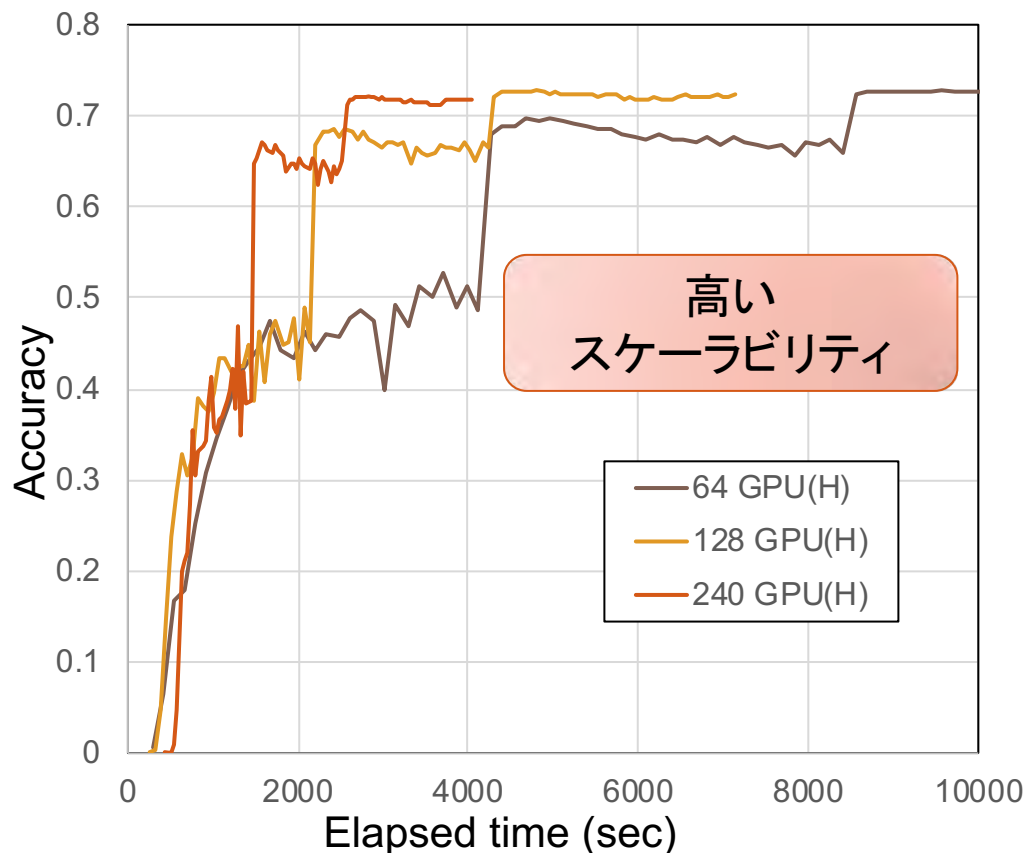
- Preferred Networksによって開発されているニューラルネットワークのためのフレームワーク
  - Open Source Software
  - Python
  - GPU向けに内部でCUDAやcuDNNを使用

- ChainerMN



- Chainerのマルチノード拡張
- MPI (Message Passing Interface)
- NCCL (NVIDIA Collective Communications Library)の活用
  - クラスタ内のGPU間における集団通信を最適化
  - 必要な通信の大半は集団通信の Allreduce通信

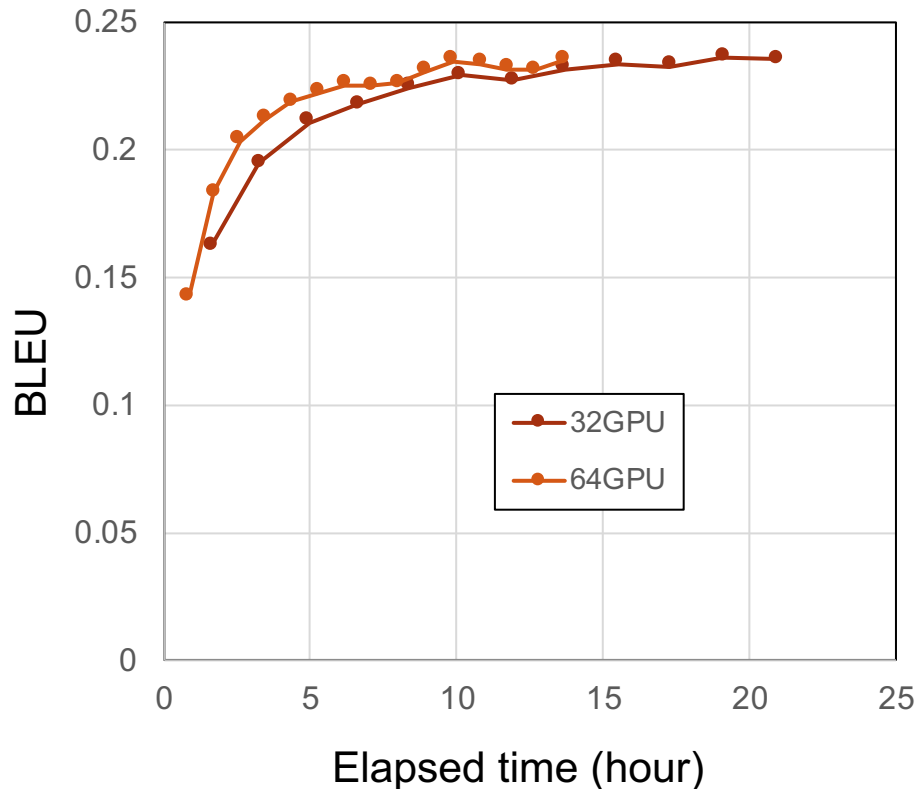
## Reedbush-HでのImageNet学習



# GPUs	100 epoch実行時間	精度
32	6時間で終了せず	(72%)
64	3時間58分20秒	72.0%
128	1時間59分02秒	72.2%
240	1時間7分24秒	71.7%

- ResNet-50
- 100エポック実行
- 64, 128, 240 GPU (RB-H)
- ChainerMN 1.0.0
  - OpenMPI 2.1.1, NCCLv2
  - Chainer 3.1.0
  - python 3.6.1, CUDA 8, cuDNN7, cupy 2.1.0

# Seq2seq学習結果



# GPUs	15 epoch実行時間	BLEU
32	24時間で終了せず (12 epoch)	(~23.6 %)
64	13.6時間	23.5 %

- 15エポック実行
- 32, 64 GPU (RB-H)
- ChainerMN 1.0.0
  - OpenMPI 2.1.1, NCCLv2
  - Chainer 3.1.0
  - python 3.6.1, CUDA 8, cuDNN7, cupy 2.1.0

# ReedbushにおけるPython環境構築

1. Moduleコマンドでインストール済みのものをロード
  - module availでモジュール名を確認しロードする
    - module load chainer/2.0.0
    - module load chainermn/1.3.0 openmpi/gdr/2.1.2/intel
    - module load horovod/0.15.2 (keras 2.2.4, tensorflow 1.8.0込み)
  - 随時更新(リクエスト可), しかし残念ながら更新頻度には限界
2. 半分自力で構築
  - **インストール済みのAnacondaを使う**
  - ある物は極力使いつつ, 最新を追いかける
3. 基本的に自力で構築
  - Anacondaも自分で入れたい場合
  - 等々

# Anacondaを利用したChainerMN環境構築

## 1. CUDAモジュールをロード

```
$ module load cuda/8.0.44-cuDNN7
```

## 2. MPIモジュールをOpen MPIに切り替え

- CUDA AwareなMPIが必要
- GPU Directを使いたい
- MVAPICH2ではエラーになる

```
$ module switch intel-mpi openmpi-gdr/2.1.1/intel
```

## 3. Anacondaモジュールをロード

```
$ module load anaconda3
```

## 4. HOMEを/lustreに差し替え

- 計算ノードは /lustre以下を使用

```
$ export HOME=/lustre/gi99/i12345
```

## 5. Anacondaの環境をcreate, activate

```
$ conda create -n chainerMN python=3  
$ source activate chainerMN
```

## 6. cupyをインストール

```
$ pip install -U cupy --no-cache-dir -vv
```

## 7. cythonをインストール

## 8. chainerをインストール

## 9. chainermnをインストール

スパコンニュース7月号  
に新しい情報を執筆

# ChainerMN実行

- 構築の際使ったのと同じモジュールをロード, 環境変数を設定

利用ノード数  
RB-H 32ノード=64GPU

- ジョブ登録

```
$ qsub train_imagenet.sh
```

- ジョブ実行状況

```
$ rstat
```

実行時間制限  
:4時間

- 実行中の出力確認

```
$ tail -f log-ジョブ番号.reedbush-  
pbsadmin0  
(Ctrl+C入力)
```

利用グループ名  
:gi99

- ジョブスクリプト例: train\_imagenet.sh

```
#!/bin/sh
#PBS -q h-regular
#PBS -l select=32:mpiprocs=2
#PBS -l walltime=04:00:00
#PBS -W group_list=gi99

cd $PBS_O_WORKDIR
module load cuda/8.0.44-cuDNN7 anaconda3
module switch intel-mpi openmpi-gdr/2.1.1/intel
export HOME=/lustre/gi99/i12345
source activate chainerMN

mpirun --mca mtl ^mxm --mca coll_hcoll_enable 0 ¥
--mca btl_openib_want_cuda_gdr 1 ¥
--mca mpi_warn_on_fork 0 ./get_local_rank_ompi ¥
python train_imagenet.py ... >& log-${PBS_JOBID}
```

スパコンニュース7月号  
に新しい情報を執筆

# レポート課題

1. [L10] GPU搭載スパコンについて、なるべく最新の3機種について詳細を調査せよ。Top500, HPCG, Green500などを参考にすること。
2. [L30~40] これまで実施した演習問題についてOpenACCを用いた記述に変更し、GPU上で実行し性能を評価せよ。(ベースの問題に応じて変動)

問題のレベルに関する記述:

- L00: きわめて簡単な問題。
- L10: ちょっと考えればわかる問題。
- L20: 標準的な問題。
- L30: 数時間程度必要とする問題。
- L40: 数週間程度必要とする問題。複雑な実装を必要とする。
- L50: 数か月程度必要とする問題。未解決問題を含む。

※L40以上は、論文を出版するに値する問題。



半年間お疲れ様でした。

コンテスト・レポートを頑張ってください。