

2022年6月27日（月）10:00-17:30

第33回スーパーコンピュータ「不老」利用型講習会 OpenACC（初級）

Zoomによるオンライン開催

名古屋大学 情報基盤センター 大島聡史 （問い合わせ先 [ohshima@cc.nagoya-u.ac.jp](mailto:ohshima@cc.nagoya-u.ac.jp)）

# 第33回スーパーコンピュータ「不老」利用型講習会 OpenACC（初級）



## お詫び（補足）

- hpc\_sdk 21.11からコンパイラの挙動が変更になり、動的な配列を扱う際は「tripletな指定」が推奨されるようになっていました。
- Cの動的に確保する配列を用いたプログラムでデータのコピーを明示する際は、data[N]のような長さだけの記述ではだめで、data[0:N]のように先頭と長さを明示する必要があります。
- Fortran版のallocatableな配列を用いたプログラムでデータのコピーを明示する際は、data(N)のような長さだけの記述ではだめで、data(1:N)のように上界・下界を明示する必要があります。
- いくつかのサンプルプログラムが影響を受けているため注意してください。
  - 動いてしまったものもあるため気がつきませんでした。
  - コンパイル時の-gpuオプションにimplicitsectionsオプションを追加すると従来の挙動になるようですが、今後もこれが引き継がれるのかはわかりません。
    - 参考情報：<https://docs.nvidia.com/hpc-sdk/archive/22.5/compiler/hpc-compilers-user-guide/index.html#gpu>

# プログラムと時間の目安

---

- 9:30 Zoom接続開始
- 10:00 – 12:00 イントロダクション、端末設定など
  - 名古屋大学情報基盤センターの計算機および利用形態
  - スーパーコンピュータ「不老」の使い方、ジョブの投入方法、実行確認
  - スーパーコンピュータ「不老」へのログイン
- 13:00 – 17:00 並列プログラミングの基本とOpenACCの学習
  - OpenACCの基礎：仕様と使い方
  - 並列計算の考え方とOpenACCプログラムの最適化
  - 並列化演習
- 17:00 – 17:30 自由演習、スパコン利用相談会

# はじめに

---

- 講師について
  - 名前：大島 聡史（おおしま さとし）
    - 情報基盤センター 准教授
  - 出身：栃木県塩谷郡（那須と日光の間）
  - 主な経歴
    - 出身大学 電気通信大学
    - → 東京大学 情報基盤センター 助教（2009.09-2017.03）
    - → 九州大学 情報基盤研究開発センター 助教（2017.04-2019.06）
    - → 名古屋大学 情報基盤センター 准教授（2019.07-）
- スパコンの運用・調達に関わりながら最新の計算機環境の活用について研究
  - 「GPUコンピューティング（およびアプリケーションのGPU化）」
  - 「並列数値計算（行列計算、疎行列ソルバー、ライブラリ）」
  - 「プログラミング環境（言語やライブラリ）」

# サンプルプログラム（ソースコード）について

- 資料中で用いているソースコードは /data/group1/a49979a/share/20220627.tgz にて公開しています
- コピーして使ってください（ディレクトリへのcdは許可していません）

```
$ cp /data/group1/a49979a/share/20220627.tgz ./
```

```
$ tar zxvf 20220627.tgz
```

→20220627というディレクトリが作られる

ディレクトリへのcdは許可していません。  
cpで持って行ってください。

- 見た目の都合等から公開コードと資料中のコードが完全一致しているわけではない点に注意してください
  - 資料の1ページに収まるように省略していることなどがあります
- 講習会用のアカウントは**今日から三日間（明後日の23:59まで）**有効です
- 必要なファイルはscpやsftpで手元にコピーしておいてください

## 参考書の例

- 「計算科学のためのHPC技術1」
  - 下司 雅章 (編集), 片桐 孝洋, 中田 真秀, 渡辺 宙志, 山本 有作, 吉井 範行, Jaewoon Jung, 杉田 有治, 石村 和也, 大石 進一, 関根 晃太, 森倉 悠介, 黒田 久泰, 著
  - 大阪大学出版会、ISBN-10: 4872595866、ISBN-13: 978-4872595864、  
発売日：2017年4月3日
- 「並列プログラミング入門：サンプルプログラムで学ぶOpenMPとOpenACC」
  - 片桐 孝洋 著
  - 東大出版会、ISBN-10: 4130624563、ISBN-13: 978-4130624565、  
発売日：2015年5月25日
- 「Parallel and High Performance Computing」
  - Robert Robey, Yuliana Zamora 著
  - Manning、ISBN-10: 1617296465、ISBN-13: 978-1617296468、  
発売日：2021年6月22日
- OpenACCの仕様書や参考Webサイトについては講習会の中で紹介する





# イントロダクション、端末設定など

---

- 「講習会資料-共通編\_20220421.pdf」 および録画済み動画にて紹介
  - スーパーコンピュータ「不老」の紹介
  - ログイン方法と基本的な操作の紹介
  - いくつか内容が古くなっている部分がありますがご了承ください
    - 動画作成時は「富岳」がTOP500ランキング世界一位でしたが、2022年6月に二位に後退しました
    - 「不老」の各ランキング情報もちろん資料中の順位より低下しています
    - (各種ランキングは基本的にシステム稼働開始時に測定・登録し、後発のより高性能なシステムが抜き去っていくもの)

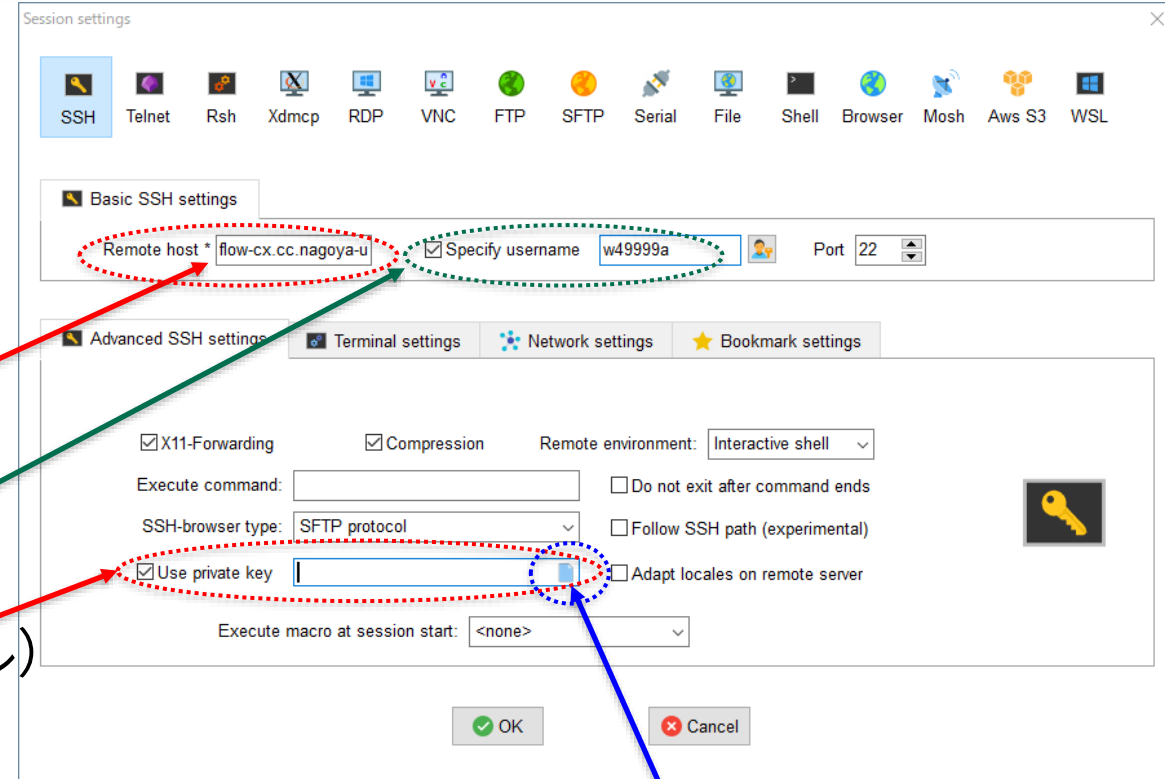
# 「不老」へのログイン

- 実際にログインしてみましょう
- 全体的な流れ
  1. 配布されたユーザIDとパスワード情報を用いて「HPCポータル」にログインする
    - <https://portal.cc.nagoya-u.ac.jp/>
  2. SSH公開鍵（秘密鍵と公開鍵のペア）を準備する
  3. HPCポータル上でSSH公開鍵を登録する
  4. SSHアクセスする
    - 今回の接続先は `flow-cx.cc.nagoya-u.ac.jp`
- 具体的な手順は以下のいずれかのファイルを参照（手順を飛ばさないこと！）
  - 不老へのログイン\_Windows\_MobaXterm.pdf Windows利用者向け、おすすめ
  - 不老へのログイン\_Windows\_Cygwin.pdf Windows利用者向け、Cygwinユーザ向け
  - 不老へのログイン\_Windows\_Putty.pdf Windows利用者向け、Puttyユーザ向け
  - 不老へのログイン\_MacLinux.pdf MacやLinuxを利用している人向け



# 講習会用アカウント（w49～）でSSH接続する際の詳細手順（補足）

- 接続先は **flow-cx.cc.nagoya-u.ac.jp**
- ユーザ名は w49～
- 秘密鍵はid\_rsa（MobaXtermではid\_rsa.ppk）
- MobaXtermを使う場合
  - 「Remote host」に接続先を記入する
  - 「Specify username」にw49～を指定する
  - 「Use private key」に秘密鍵（id\_rsa.ppkファイル）を指定する
- MacやLinux、Cygwinを使う場合
  - ssh w49～@**flow-cx.cc.nagoya-u.ac.jp** というコマンドを実行する
  - ~/.ssh/id\_rsa 以外の秘密鍵を使う場合は -i で明示する  
ssh -i 鍵ファイルのパス w49～@**flow-cx.cc.nagoya-u.ac.jp**



ここをクリックするとファイル選択画面が開くので、保存した秘密鍵（id\_rsa.ppkファイル）を選択する

# バッチジョブの実行と結果の確認

---

- プログラムの実行を伴わない（環境変数や時刻を表示して待機するだけ）のジョブを実行し、バッチジョブの実行と結果の確認方法を確認する
- 手順
  1. バッチジョブスクリプトを作成する
  2. ジョブを投入する
  3. ジョブの実行状況を確認する
  4. ジョブの実行結果を確認する

# ノードの使い分け（ログインノードと計算ノードの違い）

- ログインノード
  - 待たずに利用できる
  - 多ユーザがリソースを共有している（メモリをたくさん必要とするような処理はしないこと）
  - 外部ネットワークに直接（ではないが、近く）つながっている
  - PCI-Express版のTesla V100が搭載されているためGPUの動作確認程度はできる
  - プログラムのコンパイル・インストールや動作確認、データの整理といった用途を想定
  - ログインノードからバッチジョブを投入すると計算ノード上で計算が実行される
- 計算ノード
  - ノードが空いていないと利用できない
  - リソースを占有できる（例外：\*-shareの名前のリソースグループ）
  - 外部ネットワークへのアクセスはログインノード経由
  - バッチジョブ実行が基本（例外：\*-interactiveの名前のリソースグループ）
- ホットストレージはどちらからも同じように見える
  - ホームディレクトリも `/data/group1/${USER}` も利用可能（後者の利用を推奨）

## スパコン上でのプログラム（テキスト）編集

- sshアクセス時にも利用できるエディタとしてはEmacsやviが有名だが利用には慣れが必要
- 「いわゆる普通のテキストエディタ」的なものとしてはnanoが利用可能（次ページ参照）
- 手元で作成したファイルを転送しても良いが、スパコン上でもちょっとしたテキスト編集をしたいケースは多いと思うので、sshアクセス時に使えるエディタにも慣れておくが良い
- MobaXtermでemacsを起動しようとするするとX転送によりウィンドウが別途開かれてしまう
  - 便利だが動作は遅め、`emacs -nw`でCUI版のemacsを利用することを推奨
- ファイルの転送にはscp、sftp、MobaXtermの左側のパネルなどを使う
  - Visual Studio CodeのRemote SSHプラグインなども利用可能
  - 外部のgitリポジトリなどを経由するなどの方法もある
- 主にWindowsユーザ向けの注意
  - 利用するコンパイラなどによっては文字コードや改行コードが合わずにエラーすることがあるが、見た目では判別できないため注意が必要：文字コードUTF-8、改行コードLFなら安全（なはず）
  - 最近の「メモ帳」はUTF-8で保存できるが、LFで保存できないため変換が必要
  - バッチジョブスクリプトは改行コードLFでないとエラーしてしまうため特に注意が必要

# nanoの使い方

- 起動：nanoコマンドで起動
  - 実行時引数でファイル名を渡すと開いた状態で起動
  - 例：nano hello.c
- Shift+カーソルキーで文字列の選択ができる
- 画面の一番下にショートカットコマンドが書かれている

```

a49979a@flow-fx03:/data/group1/a49979a/work/nano/nano-5.2
GNU nano 5.2                               ./README
GNU nano -- a simple editor, inspired by Pico
Overview
The nano project was started because of a few "problems" with the
wonderfully easy-to-use and friendly Pico text editor.
First and foremost was its license: the Pine suite does not use
the GPL, and (before using the Apache License) it had unclear
restrictions on redistribution. Because of this, Pine and Pico
were not included in many GNU/Linux distributions. Furthermore,
some features (like go-to-line-number or search-and-replace) were
unavailable for a long time or require a command-line flag. Yuck.
Nano aimed to solve these problems by: 1) being truly free software
by using the GPL, 2) emulating the functionality of Pico as closely
as is reasonable, and 3) including extra functionality by default.
Nowadays, nano wants to be a generally useful editor with sensible
defaults (linewise scrolling, no automatic line breaking).
The nano editor is an official GNU package. For more information on
^G Help      ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location   ^M-U Undo     ^M-A Set Mark
^X Exit      ^R Read File  ^N Replace    ^U Paste      ^J Justify    ^G Go To Line ^M-E Redo     ^M-G Copy
  
```

^X Exit → Ctrl+Xで終了  
 ※保存が必要そうな場合は確認が出る

M-U Undo → Alt+U、またはEsc→Uでアンドゥ

# 改行コードの確認と変換

## 確認

- odコマンドを使うと簡単に確認できる
- 使い方: `od -c 確認したいファイル`

```
[a49979a@flow-fx05 tmp]$ od -c testjob.sh | head
0000000 # ! / b i n / b a s h - x \r \n
0000020 # P J M - L r s c g r p = f
0000040 x - w o r k s h o p \r \n # P J M
0000060                               P J M
0000100                               0 0 \r
0000120 \n # P J M - J \r \n # P J M -
0000140 S \r \n \r \n d a t e \r \n e n v \r \n
0000160 s l e e p 3 0 \r \n d a t e \r \n
0000200

[a49979a@flow-fx05 tmp]$ od -c testjob2.sh | head
0000000 # ! / b i n / b a s h - x \n #
0000020 P J M - L r s c g r p = f x
0000040 - w o r k s h o p \n # P J M -
0000060 L                               - L
0000100                               # P J
0000120 M                               \n d a
0000140 t e \n e n v \n s l e e p \n
0000160 d a t e \n
0000165
[a49979a@flow-fx05 tmp]$
```

Windows標準の改行コードCR+LFに対応

Linux標準の改行コードLFに対応

※ | head は先頭の数行（デフォルトで10行）だけ表示させたいときによく使う方法。確認したいファイルの先頭10行という意味ではなく、実行結果の先頭10行。

## 変換

- nkfコマンドを使うと簡単に変換できる
- 使い方
  - `nkf -Lu 変換元ファイル > 変換先ファイル`
- 左側の例は実際に以下のコマンドで変換した結果である
  - `nkf -Lu testjob.sh > testjob2.sh`

改行コード  
のせいで →  
エラーした例

```
[a49979a@flow-fx05 tmp]$ cat testjob.sh.484645.out
: invalid option
Usage: /bin/bash [GNU long option] [option] ...
       /bin/bash [GNU long option] [option] script-file ...
GNU long options:
  --debug
  --debugger
  --dump-po-strings
  --dump-strings
  --help
  --init-file
  --login
  --noediting
  --noprofile
  --norc
  --posix
  --rcfile
  --rpm-requires
  --restricted
  --verbose
  --version
Shell options:
  -ilrsD or -c command or -O shopt_option          (invocation only)
  -abefhkmptuvxBCHP or -o option
[a49979a@flow-fx05 tmp]$
```



# 計算ノードの使い方：バッチジョブとインタラクティブジョブ

- Type IIサブシステムの計算ノードは複数のリソースグループ（キュー）にわかれており、状況に応じて使いわけることでプログラムを効率良く実行できる
  - 空いているキューを積極的に利用して欲しい
  - SSDを使わないならcx-\*とcxgfs-\*に差はない
  - インタラクティブジョブは専用キュー（cx-interactive, cxgfs-interactive）
- バッチジョブ
  - ジョブスクリプトをエディタで作成する
  - pjsubでジョブを投入し、pjstat, pjstat2で実行状況を確認する
- インタラクティブジョブ（会話型バッチ処理）
  - バッチジョブ実行の仕組みを使って計算ノードにログインする仕組みであり、計算ノード上でインタラクティブにコマンドを実行できる
  - pjsub --interact -L rscgrp=cx-interactive
  - ログインしているだけでポイントを消費する
  - 主に環境構築・動作確認のための利用を想定、コンテナの構築にも有用



# バッチジョブスクリプトの記述例

<code>#!/bin/bash -x</code>	← -xを付けておくと、実行するコマンドとその結果の両方が出力される	
<code>#PJM -L rscgrp=cx-workshop</code>	← リソースグループ名：今回はcx-workshop	} 必須ではないが重要 } お好みで
<code>#PJM -L node=1</code>	← 利用ノード数：1	
<code>#PJM -L elapse=1:00</code>	← 利用時間制限：1分	
<code>#PJM -j</code>	← 標準出力と標準エラー出力を統合	
<code>#PJM -S</code>	← 統計情報を出力	
<code>date</code>	← 時刻を表示	
<code>env</code>	← 環境変数を表示（特に意味はない）	
<code>sleep 30</code>	← 30秒待つ	
<code>date</code>	← 時刻を再度表示	

- 赤字部分はバッチジョブの設定に関する情報
- 黒字部分は一般的なbashスクリプト処理

## 作業手順

1. `cd /data/group1/${USER}`で作業場所を変更  
(この通り入力してください。\${USER}は自動的にw49~に置き換わって解釈されます。)
2. `job_test.sh`として上記ファイルを作成→ `pjsub job_test.sh`でジョブ投入

# バッチジョブ実行と確認の例

赤文字が実行したコマンド、緑文字が実行結果として出力されたもの

```
[a49979a@flow-cx02 ~]$ pjsub job_test.sh
[INFO] PJM 0000 pjsub Job 75544 submitted.
[a49979a@flow-cx02 ~]$
[a49979a@flow-cx02 ~]$ pjstat2
JOB_ID      JOB_NAME    STATUS  GROUP   RSCGROUP      START_DATE      ELAPSE      NODE
75544      job_test   QUEUED  center  cx-single     --/-- --:--:--  00:00:00    1
[a49979a@flow-cx02 ~]$
[a49979a@flow-cx02 ~]$ pjstat2
JOB_ID      JOB_NAME    STATUS  GROUP   RSCGROUP      START_DATE      ELAPSE      NODE
75544      job_test   RUNNING center  cx-single     (01/05 14:15)  00:00:00    1
[a49979a@flow-cx02 ~]$
[a49979a@flow-cx02 ~]$ pjstat2
JOB_ID      JOB_NAME    STATUS  GROUP   RSCGROUP      START_DATE      ELAPSE      NODE
75544      job_test   RUNNING center  cx-single     01/05 14:15:29  00:00:02    1
[a49979a@flow-cx02 ~]$
[a49979a@flow-cx02 ~]$ pjstat2
No unfinished job found.
```

←pjsubでジョブを投入、ジョブIDが通知される  
(設定を間違えた場合などはpjdelにこのIDを指定して削除する)

←pjstat2でジョブの状況を確認 (pjstatでも確認可能詳細は利用手引書を参照)

↑QUEUEDは実行されていない状態、開始時刻が決まるとSTART\_DATEに表示される

↑RUNNINGは実行中、START\_DATEが括弧付きの場合は開始準備中

←実行中/実行開始待ちのジョブがなくなった状態

# 実行結果ファイルの確認

```
[a49979a@flow-cx02 ~]$ cat job_test.sh.76342.out
```

```
+ date
```

```
Thu Jan 7 17:46:31 JST 2021
```

```
+ env
```

```
PJM_DPREFIX=#PJM
```

```
PJM_JOBNAME=job_test.sh
```

中略

```
}
```

```
_=/usr/bin/env
```

```
+ sleep 30
```

```
+ date
```

```
Thu Jan 7 17:46:41 JST 2021
```

← catコマンド（またはlessなど）でファイルを表示

← dateコマンドが実行された時刻

← ここから環境変数

← 最後の時刻表示

補足

- 結果ファイル名は「ジョブスクリプト名.ジョブID.out」、#PJM -jを付けなかった場合は標準エラー出力が別途「ジョブスクリプト名.ジョブID.err」に出力される ※オプション指定によって変更可能
- +で始まる行はコマンドの実行結果ではなくコマンドそのもの（bashに-xオプションを付けると表示される）
- Type Iサブシステムでbash -xを指定すると冒頭に環境設定のための処理などがたくさん表示される

## リソースグループについての補足（消費ポイントについての注意点）

- 本講習会では`cx-workshop`を指定しているが、実際に利用する際にはリソースグループとGPU数の関係に注意が必要
- `cx-share`だけは常に1/4ノード、1GPUのみが確保される
- `cx-share`以外は常に1ノード単位で確保され、4GPUが利用される
- Type IIサブシステムの消費ポイント計算式は **利用GPU数** × 経過時間[s] × 0.007
- `cx-share`以外では常に **利用GPU数 = ノード数 × 4** であることに注意すること
  - `cx-single`の`single`は1ノードの意味での`single`
    - GPU数をベースにするかノード数をベースにするかの問題であり、他のサブシステムとの一貫性からノード数をベースにした

# インタラクティブジョブの実行確認

通常はcx-interactiveを使うが、講習会用にcx-workshopでもインタラクティブジョブが実行可能にしてあるのでcx-workshopを指定してください。30分を超えるインタラクティブジョブを実行したい場合はcx-interactiveを利用、特に60分を超える場合はelapsed指定を追加してください。

```
[a49979a@flow-cx02 ~]$ hostname
flow-cx04
[a49979a@flow-cx02 ~]$ pjsub --interact -L rscgrp=cx-workshop
[INFO] PJM 0000 pjsub Job 76345 submitted.
[INFO] PJM 0081 .connected.
[INFO] PJM 0082 pjsub Interactive job 76345 started.
[a49979a@cx120 ~]$ hostname
cx120
[a49979a@cx120 ~]$
[a49979a@cx120 ~]$ exit
logout
[INFO] PJM 0083 pjsub Interactive job 76345 completed.
[a49979a@cx120 ~]$
```

← ログインノードflow-cx04にいる

↑ インタラクティブジョブ用のオプションを付けてpjsub

← 計算ノードcx120にいる  
ここで何かコマンドを入力すれば計算ノード上で動作する

← Ctrl+Dやexitコマンドでログアウト

← インタラクティブジョブ終了

- インタラクティブジョブ実行時はpjsubに--interactが必要。
- -L オプションはジョブスクリプトのPJM行と同じ役割。インタラクティブではないバッチジョブの投入時にオプションを与えることも可能。
- elapsed指定をしていない状態と同じである点に注意。もっと長い時間実行したい場合は-L elapsedで指定する。cx-interactiveは24時間まで延長可能。cx-workshopは標準でも最長でも30分まで。

例：pjsub --interact -L rscgrp=cx-interactive -L elapsed=2:00:00

または pjsub --interact -L rscgrp=cx-interactive,elapsed=2:00:00

- 計算ノードに空きがないと実行できない。表示されるメッセージをよく見ておくこと。

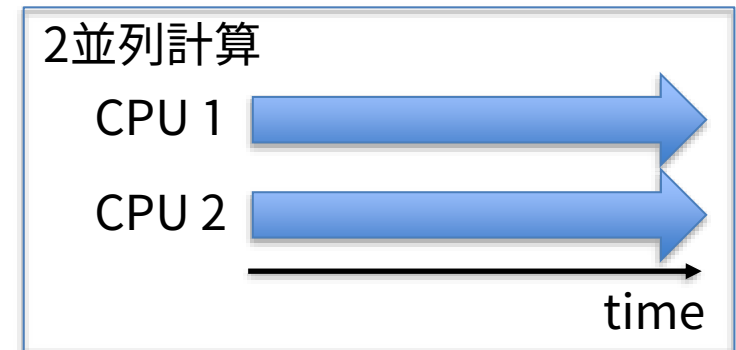
実際にやってみましょう

# 内容

- OpenACCを学ぶ前に
  - 並列計算の基礎
  - GPUとOpenACC
- 単純なベクトル計算を題材としてOpenACCの基本を学ぶ
  - コンパイルの仕方、実行の仕方
  - CPU-GPU間のデータ転送について
- 行列積を題材としてOpenACCの基本を学ぶ
  - 並列化ループの指定方法について
- その他、OpenACCに関する話題
  - 最適化のための一般的なヒント
  - デバッグとプロファイリング
  - 複数GPUの活用 など
- まとめ
- 参考（演習課題）：CG法のOpenACC化

## 並列計算とは？

- 並列計算とは何か？並列プログラミング（並列化プログラミング）とは何か？
  - 並列計算：何らかの計算処理を同時並行的に行うこと
  - 並列プログラミング（並列化プログラミング）：並列化・並列計算を行うためのプログラミング



- 並列？ 並行・平行？
  - 並列
    - 同じ処理を同時に行う、ある処理を幾つかのサブ処理に分けて同時並行的に行う
      - 1つのアプリケーションを高速に実行するために分割して同時実行するイメージ
  - 並行・平行
    - 何らかの処理を同時に行う、時分割多重のような疑似並列処理も含む
      - 直接関係のない複数の処理を同時に行うイメージ
      - 多数のプログラムの動作を調停するシステムソフトウェア（OSなど）の話をする場合はこちら
        - » 直接の関係がない処理同士だが、同じ資源を利用するために調整が必要



## なぜ並列計算を行うのか？

- 短時間で終わらせたい計算があるから、計算機（CPUやパソコンなど、計算を行うハードウェア）の持つ能力をフル活用したいから
  - 現代の計算機は並列計算により高い性能を達成、並列計算を行わなければ高い性能を得られない
    - PC用CPUもスマートフォン用CPUもマルチコアCPUが主流
      - マルチコアCPU：コア（＝計算をするユニットのかたまり）を複数搭載したCPU
    - GPUも「超」並列計算向けのプロセッサ
  - HWの性能を十分に引き出すには並列計算が必須
    - 逐次計算では搭載された全ての計算コアを使い切れず、性能を活かしきれない



Surface Pro 7 の技術仕様

あらゆる点でより強力な、軽量かつさまざまな用途に使える Surface Pro 7 についての必要な情報は、すべてここにあります。

バッテリー駆動時間*	通常のデバイス使用時間は最大 10.5 時間	メモリ	4GB、8GB、または 16GB LPDDR4x RAM
グラフィックス	Intel® UHD グラフィックス (i3) Intel® Iris™ Plus グラフィックス (i5, i7)	プロセッサ	デュアルコア 第 10 世代 Intel® Core™ i3-1005G1 プロセッサ クワッドコア 第 10 世代 Intel® Core™ i5-1035G4 プロセッサ クワッドコア 第 10 世代 Intel® Core™ i7-1065G7 プロセッサ

Surface Pro7の公式サイト の例。

プロセッサの選択肢が3つあり、コア数も明記されている。

「コア数＝性能」ではないが、重要な性能の目安ではある。

画像引用元：<https://www.microsoft.com/ja-jp/surface/devices/surface-pro-7/tech-specs>（2021.04.17の掲載内容）

## 最近のマルチコアCPUの性能の例

CPU名	コア数 (スレッド数)	クロック周波数 ×同時実行命令数	およその理論演算性能 (FP64)	理論メモリバンド幅	備考
SPARC64VIIIIfx	8 (8)	2.0 GHz × 8	128 GFLOPS	64 GB/s	「京」コンピュータに搭載
SPARC64XIfx	32 (32)	2.2 GHz × 16	1.1 TFLOPS	480 GB/s	FX100(名大旧システム)に搭載
Fujitsu A64FX	48 (48)	2.2 GHz × 32	3.3 TFLOPS	1,024 GB/s	「富岳」※、「不老」Type Iに搭載
Intel Xeon Gold 6230	20 (40)	3.00 - 3.70 GHz × 32	1.3 TFLOPS	140.75 GB/s	「不老」Type IIに搭載
IBM POWER9	22 (88)	3.07 GHz × 8	540 GFLOPS	170.7 GB/s	Summit (2018年11月時点の世界最速スパコン) に搭載
Intel Core i9 10900K	10 (20)	3.70 - 5.30 GHz × 16	592 GFLOPS	45.8 GB/s	最近のPC向けハイエンド
AMD Threadripper 3990X	64 (128)	2.90 - 4.30 GHz × 16	3.0 TFLOPS	102.4 GB/s	最近のPC向けハイエンド

- 1FLOPS=1秒間に (倍精度) 浮動小数点演算 (加算・乗算) を1回行える性能
  - 1K=1000, 1M=1000K, 1G=1000M, 1T=1000G, 1P=1000T, 1Exa=1000P
  - 複数のコアを搭載するのが当たり前、コア数は徐々に増加してきている
  - 実際のプログラムの性能はコア数や (理論) 演算性能だけでは決まらない点には注意が必要
- ※ 「富岳」は2.0 GHz

## GPUの性能の例

- HPC分野でよく利用される（された、されるであろう）GPUの例

	コア数(FP32 CUDAコア)	動作周波数	理論演算性能(FP64)*	理論メモリバンド幅	発表年月**
Tesla P100	3584 (64*56)	1328 ~ 1480 MHz	5.3 TFLOPS	720 GB/s	2016.04 製品発表
Tesla V100	5120 (64*80)	~ 1530 MHz	7.8 TFLOPS	900 GB/s	2017.06 製品発表
NVIDIA A100	6912 (64*108)	~ 1410 MHz	9.7 TFLOPS	1555 GB/s	2020.05 製品発表
NVIDIA H100	16,896 (128*132)	未定	30 TFLOPS	3,000 GB/s	2022.03 製品発表

- GPUはCPUと比べて……
  - 大量の計算コア、低い動作周波数、高い理論演算性能、高いメモリバンド幅
  - 数字以外の違いについては後述

\* 倍精度(64bit)浮動小数点演算のFLOPS値。32bitなど異なるデータ型に対する演算性能に注目することもある。

\*\* イベント等で正式に発表した年月。一部のユーザに先行提供していたり、一般向けに販売開始したのがいつなのかわからないことも多い。

# スーパーコンピュータ（スパコン）とGPU

- そもそもスーパーコンピュータとは何か？
  - 一般的な計算機システムよりも大幅に高い性能をもつ計算機システム、ただし**明確な定義はない**
  - ある程度の基準になりそうなものはある：TOP500リストや外為法の規制対象など
  - 必要条件ではないが、現実的に多数の計算機を連結したシステム
- 近年ではGPUを大量に搭載したGPUスパコンが普及
  - GPUスパコンに搭載されているGPUは高価なため、個人用途・練習環境ではより安価なGPUを使うことも多い（機能や性能に違いはあるが、ある程度は有用に使える）

	NVIDIA社	AMD社	Intel社
オンボードグラフィックス	GeForce	Radeon	Intel HD Graphics
ゲーミング	GeForce	Radeon	
プロフェッショナルCG	Quadro	Fire GL Radeon PRO	
高性能計算	Tesla（ただし最新製品はNVIDIA）	Fire Stream Instinct	
組み込み、車載など	Tegra, Xavier, Jetson		

IntelもXeという新しいGPUをリリースしてより広い範囲をカバーしようとしているようだが、詳細はまだ不明。

# TOP500 2022年6月版

- 連立一次方程式の数値解法（巨大なLU分解）
- 初登場のFrontierが首位（ついに1EFLOPS到達）、理研RCCSの「富岳」はついに2位に後退
- 近年はアメリカと中国が上位の多くを占める
- 理論性能に対する性能比はあまり高くない
  - 昔は高かった。現実のアプリではもっと下がることが多い。
- 消費電力も大きい
  - 大きなものは発電所を1つ占有するレベル
  - 10MW：一般家庭300世帯程度
- 下線部はNVIDIA社とAMD社のGPU
  - TOP10内の7システムを含む多数のスパコンがGPUを主要な計算装置として活用
  - 特に今回はAMDの最新GPU Instinct MI250Xが活躍
- 基本的にシステム稼働直後に測定した値を登録、後発に抜かれていくシステム

画像引用元：<https://www.top500.org/>

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	<b>Frontier</b> - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 20Hz, <u>AMD Instinct MI250X</u> , Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	<b>Supercomputer Fugaku</b> - Supercomputer Fugaku, A64FX 48C 2.26GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	<b>LUMI</b> - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 20Hz, <u>AMD Instinct MI250X</u> , Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	<b>Summit</b> - IBM Power System AC922, IBM POWER9 22C 3.07GHz, <u>NVIDIA Volta GV100</u> , Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	<b>Sierra</b> - IBM Power System AC922, IBM POWER9 22C 3.16GHz, <u>NVIDIA Volta GV100</u> , Dual-rail Mellanox EDR Infiniband, IBM / <u>NVIDIA</u> / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438
6	<b>Sunway TaihuLight</b> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371
7	<b>Perlmutter</b> - HPE Cray EX235a, AMD EPYC 7763 64C 2.45GHz, <u>NVIDIA A100 SXM4 40 GB</u> , Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	761,856	70.87	93.75	2,589
8	<b>Selene</b> - NVIDIA DGX A100, AMD EPYC 7762 64C 2.25GHz, <u>NVIDIA A100</u> , Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63.46	79.22	2,644
9	<b>Tianhe-2A</b> - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.26GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61.44	100.68	18,482
10	<b>Adastra</b> - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 20Hz, <u>AMD Instinct MI250X</u> , Slingshot-11, HPE Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Supérieur (GENCI-CINES) France	319,072	46.10	61.61	921



## ちなみに

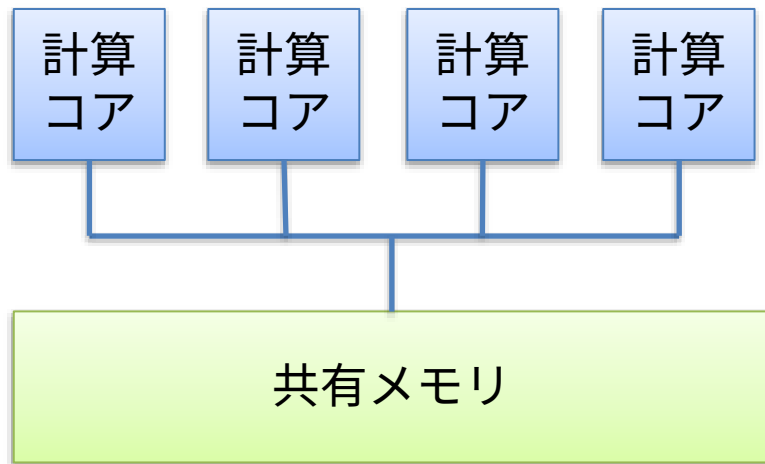
- 電力効率（性能を消費電力で割った値）を競う  
Green500の上位はGPUの独壇場に近い
  - 10システム中9システムがGPUスパコン
  - （上位4システムは同じHPE Crayのマシンだけど……）
  - 5位のMN-Coreは独自のアクセラレータ
- 「富岳」も超大規模システムとしては非常に高効率だが  
GPUがそれを上回っている

Rank	Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watt)
1	29	<b>Frontier TDS</b> - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, <u>AMD Instinct MI250X</u> , Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	120,832	19.20	309	62.684
2	1	<b>Frontier</b> - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, <u>AMD Instinct MI250X</u> , Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.08	21,100	52.227
3	3	<b>LUMI</b> - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, <u>AMD Instinct MI250X</u> , Slingshot-11, HPE EuroHPC/SC Finland	1,110,144	151.90	2,942	51.629
4	10	<b>Adastria</b> - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, <u>AMD Instinct MI250X</u> , Slingshot-11, HPE Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Supérieur (GENCI-CINES) France	319,072	48.10	921	50.028
5	324	<b>MN-3</b> - MN-Core Server, Xeon Platinum 8280M 24C 2.4GHz, Preferred Networks MN-Core, MN-Core DirectConnect, Preferred Networks Preferred Networks Japan	1,664	2.18	53	40.981
6	315	<b>SSC-21 Scalable Module</b> - Apollo 6500 Gen10 plus, AMD EPYC 7543 32C 2.8GHz, <u>NVIDIA A100 80GB</u> , Infiniband HDR200, HPE Samsung Electronics South Korea	16,704	2.27	101	33.983
7	319	<b>Tethys</b> - NVIDIA DGX A100 Liquid Cooled Prototype, AMD EPYC 7742 64C 2.25GHz, <u>NVIDIA A100 80GB</u> , Infiniband HDR, Nvidia NVIDIA Corporation United States	19,840	2.25	72	31.538
8	304	<b>Wilkes-3</b> - PowerEdge XE8545, AMD EPYC 7763 64C 2.45GHz, <u>NVIDIA A100 80GB</u> , Infiniband HDR200 dual port, DELL EMC University of Cambridge United Kingdom	26,880	2.29	76	30.797
9	105	<b>Athens</b> - FormatSecur THOR ER021, AMD EPYC 7742 64C 2.25GHz, <u>NVIDIA A100 80GB</u> , Infiniband HDR, Format Sp. z s.p. Cyfronet Poland	47,616	5.05	147	29.926
10	343	<b>Phoenix - 2022</b> - ThinkSystem SR670 V2, Xeon Platinum 8369Y 36C 2.4GHz, <u>NVIDIA A100</u> , Infiniband HDR, Lenovo University of Adelaide Australia	20,140	2.07	69	29.924

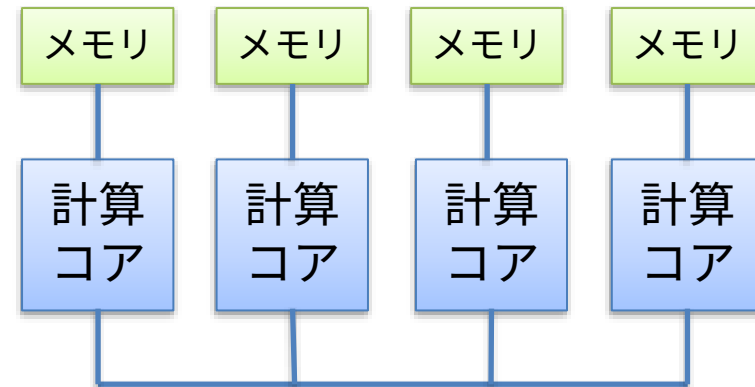
# 並列計算機の種類

- 並列計算機は共有メモリ型と分散メモリ型に大別される

- 共有メモリ型：メモリを共有している
  - 同じメモリ（データ）に各プロセッサが自由に直接アクセスできる、メモリアクセスへの競合（衝突）が起こる
  - 分散メモリ型として扱うこともできる



- 分散メモリ型：メモリを共有していない
  - 分散メモリモデル：各プロセッサが個別のメモリ（データ）を持つ、互いに直接アクセスできない、他のコアの持つメモリにアクセスするには通信が必要
  - 共有メモリ型として扱うためには、分散メモリを共有メモリとして扱う特別な仕組みが必要



- 大規模環境では混在した（階層的な）構成となる
- 物理的な構成とプログラミング手法は1対1の対応関係ではない
- ノードの概念とも1対1ではない（基本的にはノード内が共有、ノード間が分散ではある）



## 並列計算環境を利用するための手段

- 並列計算環境が普及した今日では様々な「並列化のための道具」が使われている
  - バラバラだと提供側・利用者ともに不便なため共通化されることが多い
    - ある環境向けに書いたものが別の環境でも利用できる（互換性）
    - 性能まで互換性があるとは限らない点には注意が必要（性能可搬性）
      - ある環境では有効であった最適化が別の環境では性能低下要因になることも
      - 環境が変わってもその環境ごとに常に最大の性能が得られるようにするための研究も行われている→自動チューニング
- 自動化はできないのか？
  - 全く不可能なわけではない、ある程度はコンパイラ等が行ってくれる
  - 「コンパイラ様のご機嫌を伺う」コードを書く必要がある、コンパイラによって差が大きい、簡単なコードでないとうまくいかないことが多い
    - 単純な行列積などコードの形状と最適化のパターンが決まっているものは人が書くよりコンパイラやライブラリの方が得意
  - OpenMPやOpenACCは簡単・少量の記述で効果的な並列高速化を可能とする（良いところ取り、規格化により汎用性・可搬性も担保）

## 並列計算を行う方法（言語など）の例

- コンパイラによる自動並列化：SIMD並列化など一部の処理で有用
  - pthread：スレッド並列処理のための関数群
    - pthread\_createなどのスレッド操作関数を使う
    - 現在ではアプリケーションコードで直接利用することはあまりない
  - OpenMP：スレッド並列処理、主にループ並列化向けの指示文規格
    - #pragma omp parallel for、!\$omp parallel do
    - 最近の規格ではタスク処理やGPU対応などを拡充
  - **OpenACC**：GPU向けのOpenMPのようなもの
  - CUDA：NVIDIA社のGPU向け、GPUを普及させた最大要因の一つ、NVIDIA GPUの能力をフル活用できる
  - OpenCL：「汎用版CUDA」のようなもの、FPGAなどでも利用可能
  - MPI：プロセス間の通信規格、特に複数ノード利用時に必須
    - MPI\_Send, MPI\_Recv, MPI\_Gatherなどの通信関数を使う
  - 必要に応じてこれらを組み合わせて用いる（OpenMP+MPIなど）
- 
- ノード内CPU内スレッド並列化
  - 共有メモリモデル
- 
- GPU内並列化
  - デバイス内では共有メモリモデル、外部とのやりとりは分散メモリモデル
- 
- ノード間並列化
  - 分散メモリモデル

## 並列計算の基本的なイメージ：ループ並列化（C版）

- 元となる逐次計算

– 単純な繰り返しループ計算

```
for(i=0; i<N; i++){
  A[i] = B[i] + C[i];
  D[i] = E[i] + F[i];
}
```

- ループ内の処理を分割し、同時に計算

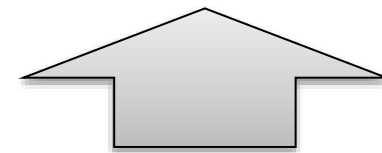
```
PE1 for(i=0; i<N; i++){
  A[i] = B[i] + C[i];
}
```

```
PE2 for(i=0; i<N; i++){
  D[i] = E[i] + F[i];
}
```

- ループそのものを分割し、同時に計算

```
PE1 for(i=0; i<N/2; i++){
  A[i] = B[i] + C[i];
  D[i] = E[i] + F[i];
}
```

```
PE2 for(i=N/2; i<N; i++){
  A[i] = B[i] + C[i];
  D[i] = E[i] + F[i];
}
```



- OpenMPやOpenACCはこちらのイメージ

## 並列計算の基本的なイメージ：ループ並列化（Fortran版）

- 元となる逐次計算

– 単純な繰り返しループ計算

```
do i=1, N
  A(i) = B(i) + C(i)
  D(i) = E(i) + F(i)
end do
```

- ループ内の処理を分割し、同時に計算

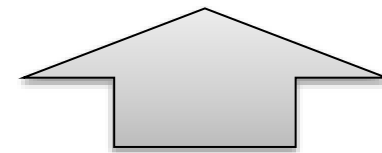
```
PE1 do i=1, N
     A(i) = B(i) + C(i)
end do
```

```
PE2 do i=1, N
     D(i) = E(i) + F(i)
end do
```

- ループそのものを分割し、同時に計算

```
PE1 do i=1, N/2
     A(i) = B(i) + C(i)
     D(i) = E(i) + F(i)
end do
```

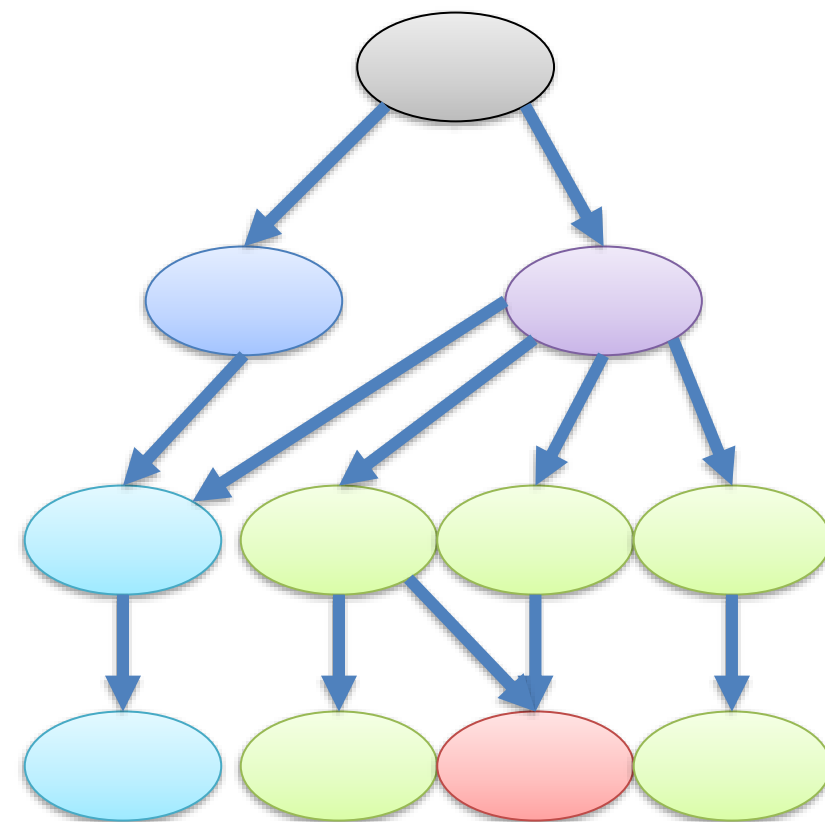
```
PE2 do i=N/2+1, N
     A(i) = B(i) + C(i)
     D(i) = E(i) + F(i)
end do
```



- OpenMPやOpenACCはこちらのイメージ

## 並列計算の基本的なイメージ：タスク並列化

- ループによる並列化よりも大きい粒度の並列計算に適する
- 大規模なプログラム・複雑なプログラムの並列化に有用
- OpenMPにおいても近年サポートが活発
  - task構文
- GPU (OpenACC) には向いていないため  
今回の講習会では扱わない

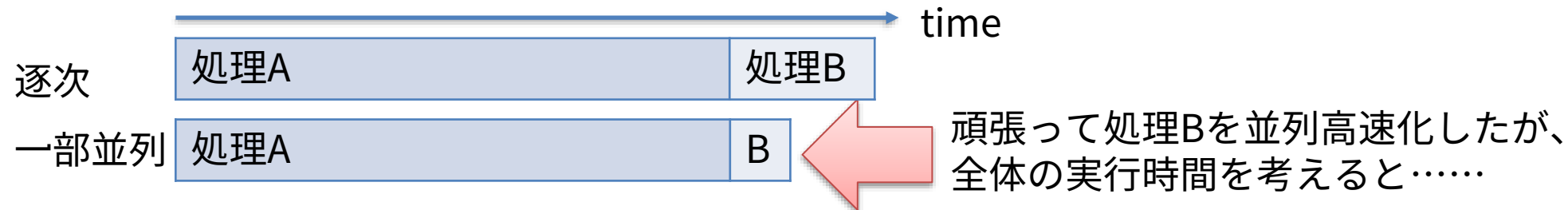


# 並列計算の限界

- 並列化できないプログラムも少なくない
  - 計算順序に制約がある場合は困難
    - 工夫により可能となることもある
- (プログラム高速化全般に言えることだが)  
速くした部分しか速くならない

```
for(i=1; i<N; i++){
  A[i] = A[i-1] + B[i];
  C[i] = A[i] + D[i];
}
```

例：ループイタレーション間にも、  
同じループイタレーション内にも、  
依存関係がある ⇒ 並列化が困難



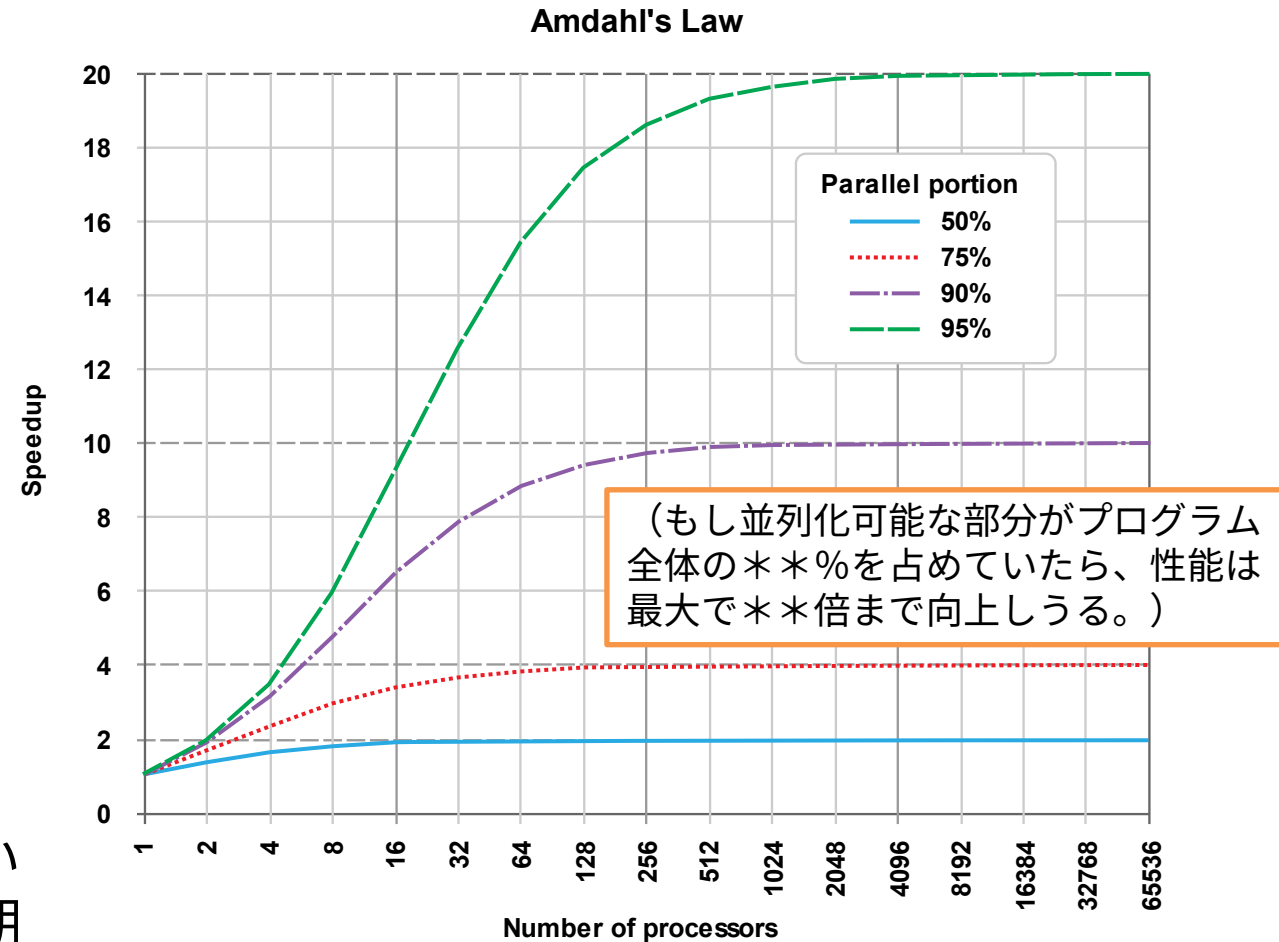
- 探索問題などでは分割数（並列度）よりずっと速くなることもある
  - 探索対象
  - 逐次探索
  - 並列探索

プログラムのどこを並列化させるか、GPUに担当させるかを  
しっかり考える必要がある



# 性能向上率と並列化の限界

- アムダールの法則
  - 並列化できる範囲の割合と、並列化により得られる性能向上の関係
- そもそも対象問題（プログラム）に十分な並列度がなければならない
- 並列化できない部分が大きいといくら並列化しても時間が短くならない
- 並列化に伴い通信などのオーバーヘッドが増える可能性がある
  - 逐次実行では不要であった通信が増える
  - 共有メモリモデルであるOpenMPに通信はないが、スレッドの生成・破棄やスレッド間の同期は必要でありオーバーヘッドとなる



※グラフはWikipedia「アムダールの法則」から引用



# この講習会の目的：GPUを活用する方法の基礎を学ぶ

- 何故GPUを活用する必要があるのか？
  - GPUは非常に高い性能をもつハードウェアであり、うまく活用できれば大変強力な研究の道具となる：成果を得るための加速装置
  - GPUは国内外の様々な計算環境に導入されているため、利用スキルを得ておくことはきっとプラスになる
    - 名古屋大学 情報基盤センターでもType IIサブシステムにGPUを搭載
- どうやって使う？
  1. 並列化されたソフトウェア（アプリケーション）を使う
  2. （並列化されていないプログラムから）並列化されたライブラリやフレームワークを使う
  3. **自分で並列計算を実装する**
    - 自分が扱いたい任意の問題（アプリ）をGPU化できる
    - 扱いたい問題が既にGPU化されているならそれを使っても良いが、GPUのことを理解しているかどうかでさらに高い性能が得られるかどうかが決まる、こともある
      - 例：特定の形状・大きさの行列しか扱わない

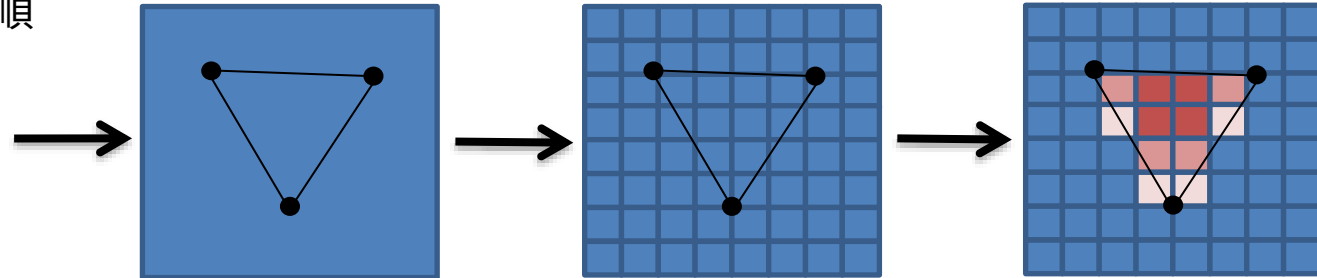
# GPU (Graphics Processing Unit)

- 画像処理用のハードウェア

- CPUやマザーボードに組み込まれたチップ、または拡張スロットに搭載するビデオカードとして広く普及
- 本来の役割：高速・高解像度描画、3D描画処理（透視変換、陰影・照明）、画面出力

3次元画像描画の手順

- ① (2, 2)
- ② (8, 3)
- ③ (5, 7)



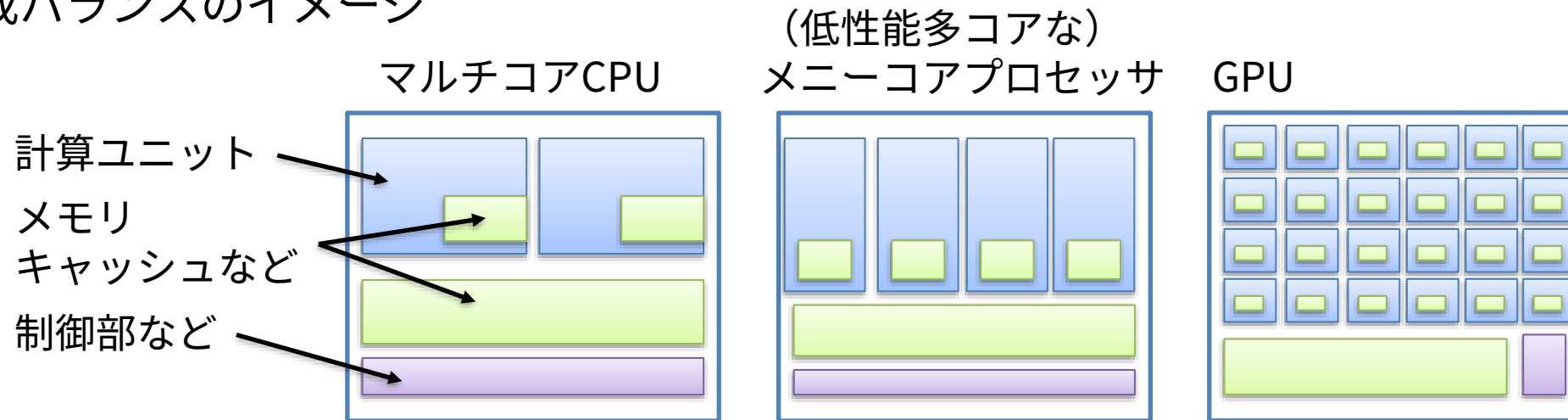
オブジェクト単位、頂点単位、ピクセル単位で同時（並列）処理が可能

- 同時にできることが多いため、それにあわせたハードウェアへと進化
- 数値シミュレーション（科学技術計算）を高速に行えるハードウェアと両立することがわかり、GPUの重要な市場の一つとなってきた
- 現在では特にビッグデータ、機械学習、AI処理などで性能を発揮

# CPUとGPUの違い

- GPUはCPUと比べて**単純な処理を（順序を問わず）たくさん行う**ことが求められるため、それに適したハードウェアへと進化してきた

– HW構成バランスのイメージ



- GPUの特徴：たくさんの計算ユニット、高速なメモリ、OSレス
  - OSを動かし様々な仕事をせねばならないCPUとは大きく異なる
  - 単体で利用できない、CPUによるサポートが必要
  - 現代の計算加速装置（アクセラレータ）の代表格
- CPUよりスゴイ、というよりも、**得意とする仕事が違う**ことが重要

## GPUの中身（もう少し詳しく）

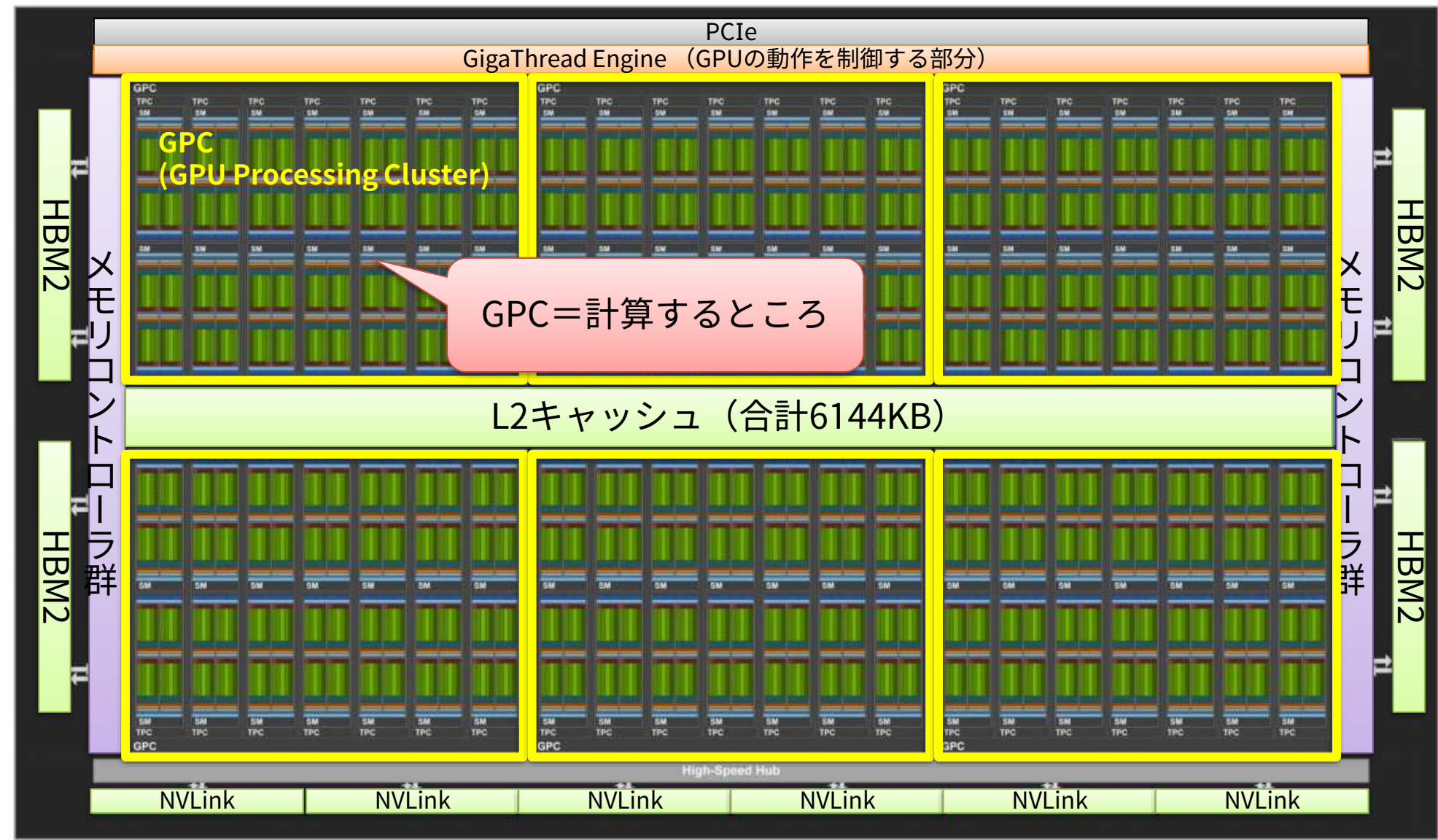
- GPUとはどのようなハードウェアなのだろうか？
  - CPUとはバランスが違う、たくさんの計算コアと高速なメモリを搭載、ということは既に紹介したが……
- 具体的な例（Tesla V100の例）
  - NVIDIA TESLA V100 GPU ARCHITECTUREより引用
    - 引用元
      - <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
    - 日本語版はこちら
      - <https://images.nvidia.com/content/pdf/tesla/Volta-Architecture-Whitepaper-v1.1-jp.pdf>
  - 名古屋大学情報基盤センターに設置されているスーパーコンピュータ「不老」のType IIサブシステムに搭載されているGPU
    - ノードあたり4枚、サブシステム全体で884枚搭載
    - 資料に出てくる様々な数字を全て覚える必要はない（最適化を行う際には気にする必要がある）

# 全体構成





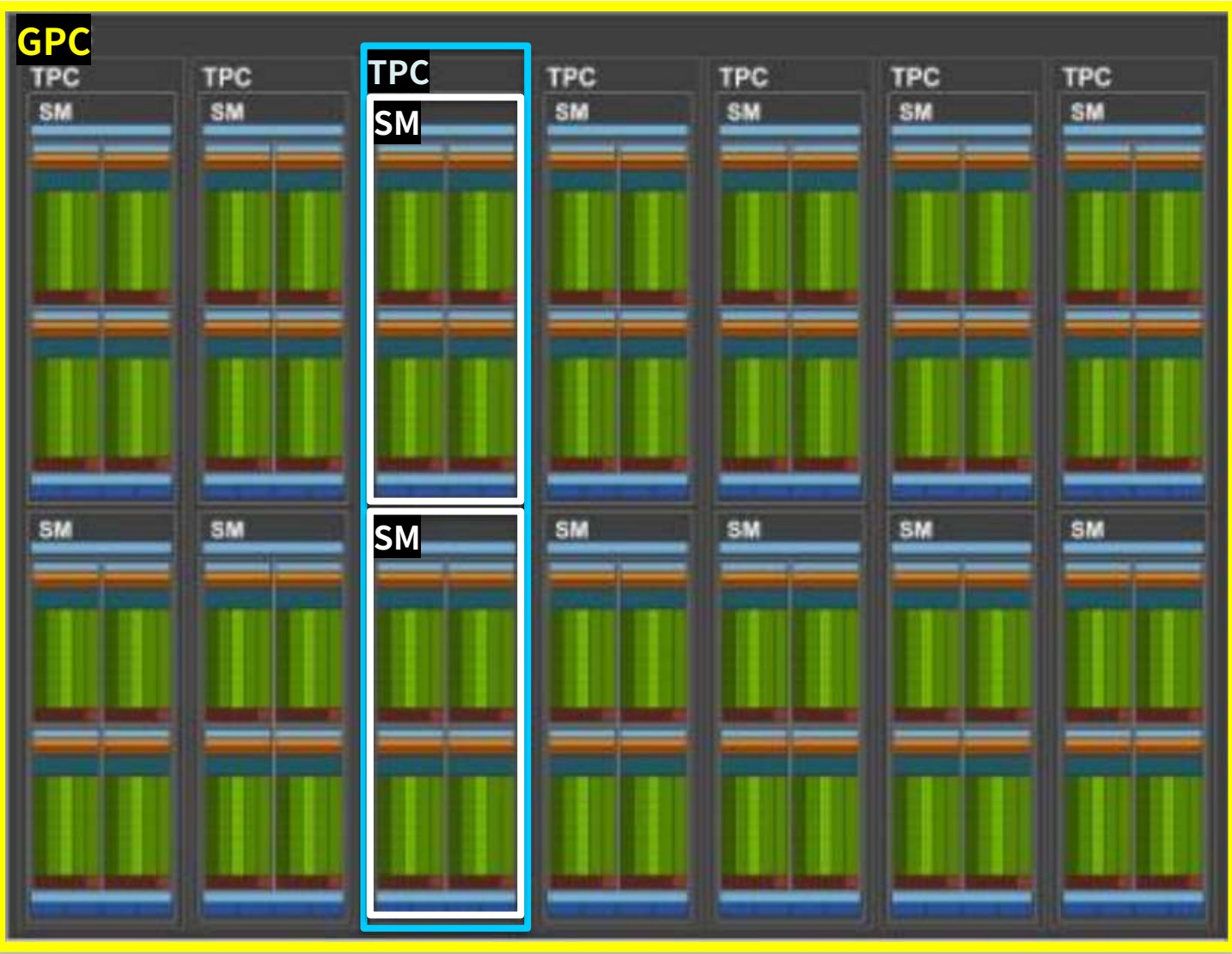
# 全体構成





# GPCの中身

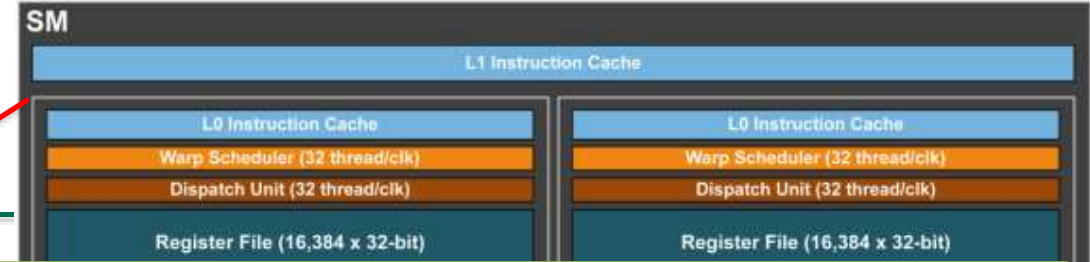
- GPC: GPU Processing Cluster
- TPC: Texture Processing Cluster
- SM: Streaming Multiprocessor



- 1GPUに6GPC搭載
- 1GPCに7TPCs搭載、各TPCに2SMs搭載
- 1SMあたり
  - 64 FP32 cores
  - 64 INT32 cores
  - 32 FP64 cores
  - 8 Tensor cores
  - 4 Texture units
- GPU全体では6\*14=84SMs搭載※、すなわち
  - 5,120 FP32 cores
  - 5,120 INT32 cores
  - 2,560 FP64 cores
  - 640 Tensor cores
  - 320 Texture units
- これら大量のコアが最大1,530MHzで動作
- GPU全体で32GBのHBM2を搭載
- SMあたり96KBまでの高速共有メモリも搭載
- PCIeやNVLinkでホストCPUや他のGPUなどと接続

※ Tesla V100では84SMs中の80SMsのみが有効。4個までは不良品が含まれてもOKとすることで歩留まりを向上させている。

# SMの中身



- 多数のコアを単一のスケジューラが調整している点も大きな特徴の一つ
- 同じタイミングで32スレッドが同じ計算を行う（異なるデータに対して同一の計算を実行）ことに注意が必要
- Voltaで変更が入り細かい挙動が変わったため、Pascal以前とVolta以降では最適化の仕方が異なることがある

- Voltaで新しく搭載された計算コア
- $4 \times 4$ 行列[FP16or32] =  $4 \times 4$ 行列[FP16] ×  $4 \times 4$ 行列[FP16] +  $4 \times 4$ 行列[FP16or32]  
 という計算だけを高速に行うことができる、低精度小密行列積演算加速装置  
 （後継のGPUでは他の精度の計算にも対応）

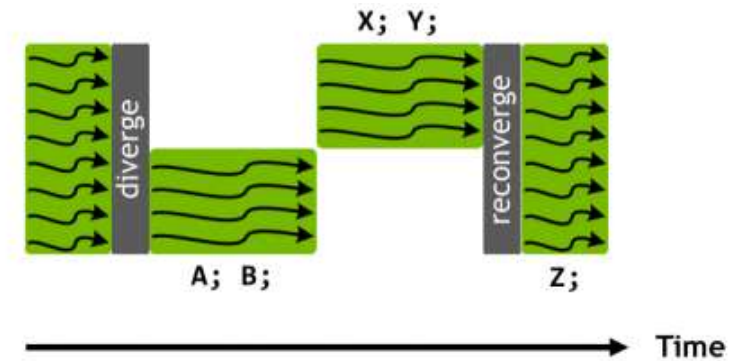


# SIMT(Single Instruction Multiple Thread)

- CPUで用いられているのはSIMD
  - Single Instruction Multiple Data
  - 1命令で複数のデータを扱う
- SIMTは1命令で複数のスレッドが動作する
  - 言い換えれば、複数のスレッドが同時に同じ命令を実行する
- 命令分岐はマスク処理される
  - 実行はしないがプログラムカウンタは遷移する、全スレッドが全分岐を実行するのと同程度の時間がかかる
    - データ転送分が省かれたりするだけマシンではあるが

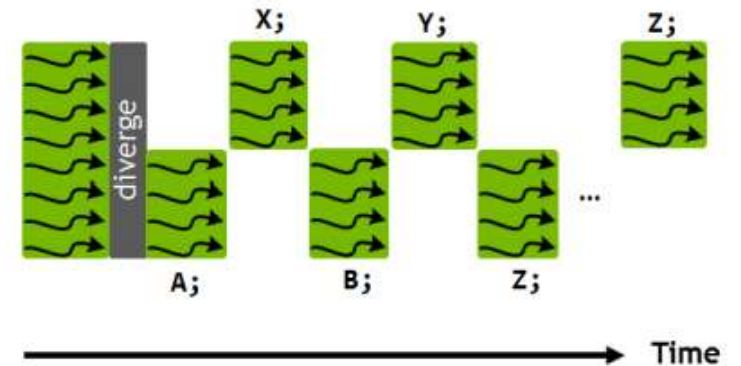
- NVIDIAの示している例
  - PascalまでのSIMTの例

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



- Volta以降のSIMTの例

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```





## もっと単純に言うと？

- 『多数の計算コアを搭載した「計算コア群」』が多数搭載されたハードウェア
- 合計で数千コアが搭載されている、ただし「計算コア群」の中の計算コアの動作には制限があり、完全にバラバラに計算できるわけではない
- スレッド並列化 + SIMD並列化をイメージするとわかりやすい、かもしれない
  - CPU：16コア・AVX512 → 16コアが個別に動作、各コア内で512bit (64bit\*8) がまとめて動作
  - GPU：84SMs・32FP64cores → 84SMsが個別に動作、各SM内で32のFP64coresがまとめて動作
  - (ただし細かい動作モデルなどは異なる)
- GPU全体の構成やSM内の構成はGPUの世代によって変わるため、具体的な数字を一生懸命覚える必要はない
- 実際のプログラミングにおいてはある程度抽象化して扱う
  - もちろん、最適化の際には具体的な数字を考えることになることもある

## GPUを用いた並列計算の注意点（性能を得るコツ）

- 多数のコア全てを満たすに十分な仕事を与える
  - 数千のコア全てをフル稼働させる並列度が必要
  - コア数の数倍以上の仕事があることが望ましい
    - メモリアクセスを待つ間に別の仕事ができる
- WARPを意識する
  - 内部的には32演算器単位で動作しており、分岐の際などに注意が必要
    - だったのだが、Voltaから変わってしまった
    - 変わってしまったが、まずはWARPの動作をイメージしてプログラムを作成すると良い性能が得られやすい
- 連続メモリアクセスについて考える
  - コアレスなメモリアクセスが高速
- 詳細は終盤（OpenACCプログラムの最適化）にて解説する
  - 基本的には、並列度が高くて分岐の少ない単純なプログラムが良い

# 参考：CUDAプログラム

## • 単純な配列コピーの例

```
__global__ void gpukernel
(int N, float* C, float* A){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nt = blockDim.x * blockDim.x;
    for(int i=tid; i<N; i+=nt){
        C[i] = A[i];
    }
}
```

GPU上で行われる処理  
(GPUカーネル)

- 同時にたくさん実行される
- 自分のIDを元に計算すべき範囲を特定する

```
int main(int argc, char **argv){
    int N = 100000;
    float *A, *C;
    float *d_A, *d_C;
    A = (float*)malloc(sizeof(float)*N);
    C = (float*)malloc(sizeof(float)*N);
    cudaMalloc((void**)&d_A, sizeof(float)*N);
    cudaMalloc((void**)&d_C, sizeof(float)*N);
    cudaMemcpy(d_A, A, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, C, sizeof(float)*N, cudaMemcpyHostToDevice);
    gpukernel<<<4,4>>>(N, d_C, d_A);
    cudaMemcpy(C, d_C, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    return 0;
}
```

CPU上で行われる処理

- 専用の関数などを用いて様々な処理を行う

慣れてしまえばそれほど難しいものではないが、記述量がそれなりにあり「カジュアルなGPU利用」にはあまり向かない



# GPU (CUDA) 向けライブラリ・フレームワーク

- GPUの性能を活用できる様々なライブラリが公開されている
  - <https://developer.nvidia.com/gpu-accelerated-libraries>
  - 例
    - Deep Learningライブラリ
      - cuDNN, TensorRT, DeepStream SDK
    - 数値計算・数学ライブラリ
      - cuBLAS, cuSPARSE, cuRAND, cuFFT, ...
    - 通信、C++クラス、etc.
      - NCCL, Thrust, OpenCV, MAGMA, ...
  - NVIDIA社自ら手がけるものは「CUDA-X」と呼ばれるようになった
  - 各自が行いたい処理とマッチしていれば容易にGPUの性能を活用することができる
  - OpenACCとの連携についてはとくに書かれていないものが大半を占めると思うが、ペナルティなしに連携して利用できるものもある（ユーザがcudaMallocで確保したメモリのポインタを引き渡すタイプだと簡単）

# 内容

- OpenACCを学ぶ前に
  - 並列計算の基礎
  - GPUとOpenACC
- 単純なベクトル計算を題材としてOpenACCの基本を学ぶ
  - コンパイルの仕方、実行の仕方
  - CPU-GPU間のデータ転送について
- 行列積を題材としてOpenACCの基本を学ぶ
  - 並列化ループの指定方法について
- その他、OpenACCに関する話題
  - 最適化のための一般的なヒント
  - デバッグとプロファイリング
  - 複数GPUの活用 など
- まとめ
- 参考（演習課題）：CG法のOpenACC化

## OpenACCとは？

- GPU（などのアクセラレータ）向けのプログラムを簡単に記述することができる並列化プログラミング言語（指示文規格）
  - GPU向けのOpenMPのようなもの、C/C++やFortranで書かれたプログラムに簡単な**指示文**を追加するだけでGPU対応が可能
    - コンパイラ向けのコメントを記述する
    - 最適化を行うにはある程度GPUの知識があった方がよい
    - マルチGPU・マルチノードについてはMPIなどと組み合わせて利用
    - 最近はOpenMPのGPU対応も進んでいる、今後どちらが主流になるかはわからない
      - OpenMPに集約されるという話はあるが、なかなか進んでいない
      - いますぐに試しやすいのはOpenACC
  - 幾つかの会社が独自に開発していたものが共通規格として集約されたもの
    - 初登場が2011年、ようやく10周年の新しいめな言語（規格）
  - OpenMPと似た部分も多いため、どちらかを学んであるともう一方の学習も容易
    - 少なくとも**指示文を用いた並列化**という基本的な部分は共通しているため馴染みやすいはず
  - AMDのGPUでも使えるという話はあるが、ちょうど良い実験環境を有していないため未調査

# OpenACCとCUDA

- OpenACCは「どんなプログラムでもGPU化できる」「どんなプログラムでも高速化できる」**わけではない**が、多くのプログラムに対して有用
  - 自作コードのGPU化を行うことができる上で最も簡単でコストパフォーマンスの高い（労力に対して十分な性能が得られる）方法
- CUDAでしか書けない処理も多い
  - 「とにかく高い並列度で一気に計算すれば良い」という典型的なGPU向けプログラムではOpenACCでも十分高い性能が得られる
  - CUDAを使うべきプログラム
    - GPU上の高速共有メモリやシャッフル命令を意識したアルゴリズム
    - インスタンスIDを意識したアルゴリズム
      - CUDAはThreadIDなど「ID」を意識して並列処理を記述する
      - OpenACCは「ID」の概念そのものがない
    - その他、最新のハードウェア機能をフル活用したい場合
      - Tensor core、RT core、半精度演算

## OpenACCに関する情報、ツール、etc.

---

- 規格などの情報
  - <https://www.openacc.org/>
    - 仕様書その他、学習用の資料へのリンクなどもあり
      - 例 <https://github.com/OpenACC/openacc-training-materials>
- 主な対応コンパイラ
  - 商用：PGI (→NVIDIA HPC SDKへ移行)、Cray、国家超級計算无锡中心
  - 研究：Omni、OpenARC、OpenUH、ROSEACC
  - OSS：GCC
- プロファイラやデバッガなど
  - allinea MAP/DDT、PGI pgprof、NVIDIA nvprof/nvvp
- 日本語で書かれた資料
  - 旧PGIコンパイラ技術情報 | HPC WORLD <https://hpcworld.jp/pgi-compiler-archive/>
    - 旧PGIコンパイラの代理店であるソフテック社のWebサイトに掲載されていた資料のアーカイブ
    - 言語仕様が大きく変わってしまわない限りは今後も有用だろう

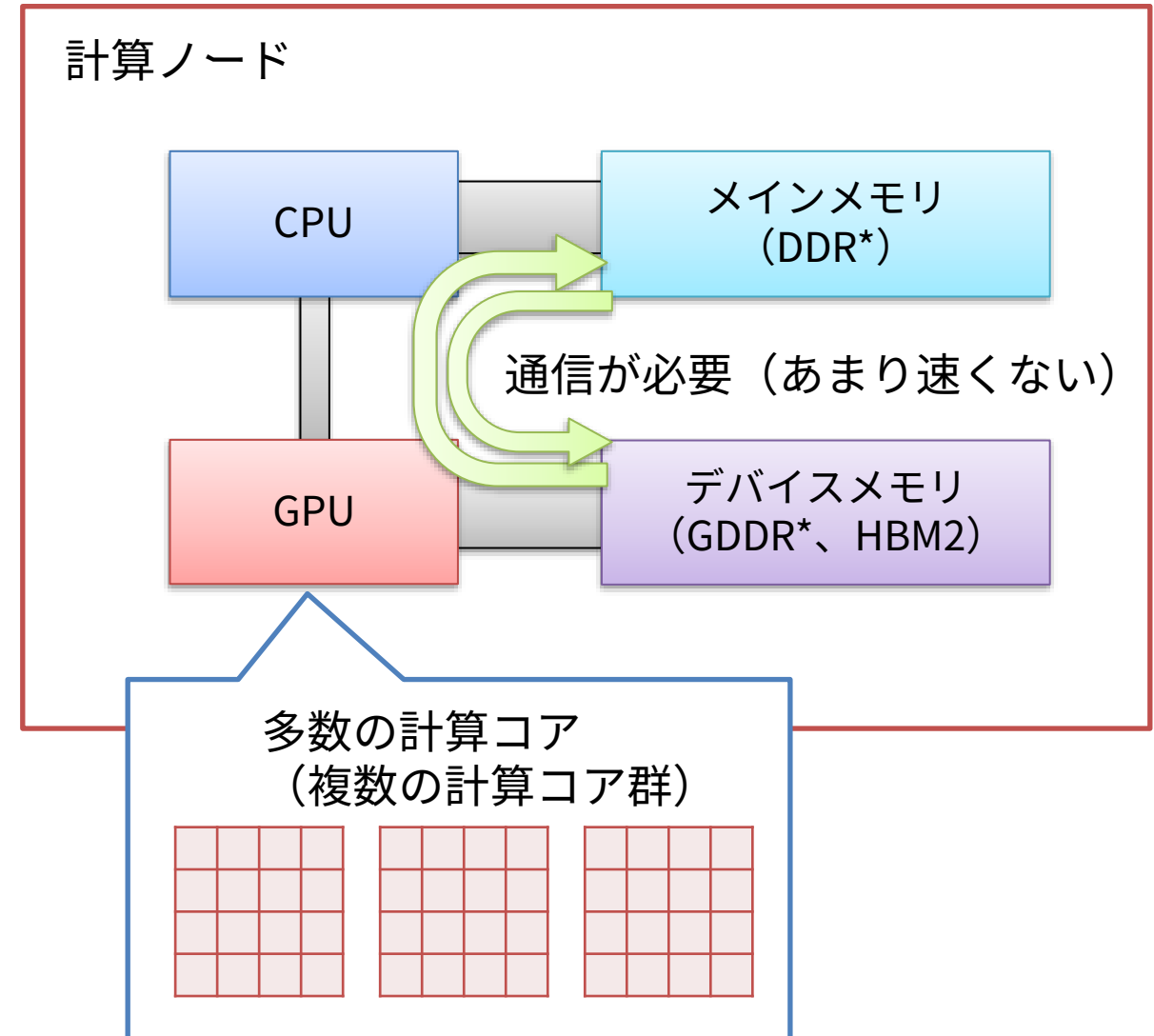
## PGIコンパイラとHPC SDK

- PGIは独立したコンパイラ開発会社であったが、2013年にNVIDIAに買収された。PGIコンパイラは継続して提供（販売、一部無償公開）されていたが、2020年6月末にNVIDIA HPC SDKに統合された。
  - なお、AMDのGPUへの対応はHPC SDKへ統合されるずっと前に打ち切られている
- これからOpenACCを始める場合にはHPC SDKを使うのが基本となると思われるため、本資料も全てHPC SDKの利用を前提として作成してある
- NVIDIA HPC SDK
  - Webページ <https://developer.nvidia.com/hpc-sdk>
    - 無料で誰でも利用できる
  - 「不老」には対応するmodulefileが用意されているため簡単に利用できる
  - (SDK自体はユーザ権限でもインストールできるため、最新版をはやく使いたい場合は自分で導入してください)



## OpenACCの想定する典型的なGPU計算環境

- 1つの計算ノードに1つのCPUと1つのGPUが搭載された環境
- OpenACCを用いてある程度まともな性能を得るために必要なGPUに対する最低限の理解
  - GPUは多数の計算コア（複数の計算コア群）が搭載されたプロセッサ
  - CPUとGPUは個別に計算用のメモリを持っており、相手側のメモリを読み書きするにはCPU-GPU間の通信が必要
    - （CPU-GPU間の通信はあまり速くないため、あまり頻繁に・大量に通信したくない）



# OpenACCの基本的な使い方

---

1. 専用の指示文を含むソースコードを用意する
  - 特別な拡張子などは定められていない
  
2. 専用のコンパイラでコンパイルする
  - OpenACC向けのオプションを指定する
  
3. 実行する
  - CPU向けの実行可能ファイルと同様にそのまま実行可能
    - ./a.out
  - (少なくともHPC SDKであれば) 特別なプログラムを介したりする必要はない
    - LD\_LIBRARY\_PATHなどの対応は必要
      - 「不老」ではmodule loadの時点で設定されている
  - 想定されているGPUが利用できないと実行時エラー
    - nvaccelinfoプログラムでGPU情報が見えていることが前提条件

## GPU環境の確認

- nvaccelinfoコマンドでGPUの存在が確認できる状態である必要がある

Type IIサブシステム用ログインノードで実行した場合の例 →

```
$ nvaccelinfo
```

```

CUDA Driver Version:      11040
NVRM version:            NVIDIA UNIX x86_64 Kernel Module  470.57.02  Tue Jul 13 16:14:05 UTC 2021

Device Number:          0
Device Name:            Tesla V100-PCI-E-32GB
Device Revision Number: 7.0
Global Memory Size:    34089730048
Number of Multiprocessors: 80
Concurrent Copy and Execution: Yes
Total Constant Memory: 65536
Total Shared Memory per Block: 49152
Registers per Block:   65536
Warp Size:              32
Maximum Threads per Block: 1024
Maximum Block Dimensions: 1024, 1024, 64
Maximum Grid Dimensions: 2147483647 x 65535 x 65535
Maximum Memory Pitch:  2147483647B
Texture Alignment:     512B
Clock Rate:             1380 MHz
Execution Timeout:      No
Integrated Device:      No
Can Map Host Memory:    Yes
Compute Mode:           default
Concurrent Kernels:     Yes
ECC Enabled:            Yes
Memory Clock Rate:      877 MHz
Memory Bus Width:       4096 bits
L2 Cache Size:          6291456 bytes
Max Threads Per SMP:    2048
Async Engines:          7
Unified Addressing:     Yes
Managed Memory:        Yes
Concurrent Managed Memory: Yes
Preemption Supported:   Yes
Cooperative Launch:    Yes
  Multi-Device:         Yes
Default Target:         cc70

```

1GPU分の情報  
(下にもう1つのGPUの分の情報がつづく)

OpenACCコンパイル時に  
指定すべきGPU情報

cc70

# 単純なOpenACCプログラムの例 (配列の定数倍計算)

## • C (vector1.c)

```

1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int i;
6      int n=10;
7      double v1[10], v2[10];
8
9      for(i=0; i<n; i++){
10         v1[i] = (double)(i+1);
11         v2[i] = 0.0;
12     }
13
14     #pragma acc kernels
15     for(i=0; i<n; i++){
16         v2[i] = v1[i] * 2.0;
17     }
18
19     for(i=0; i<n; i++)printf(" %.2f", v1[i]);
20     printf(" ¥n");
21     for(i=0; i<n; i++)printf(" %.2f", v2[i]);
22     printf(" ¥n");
23
24     return 0;
25 }
26 }

```

OpenACC並列化対象  
=GPU上で実行される

## • Fortran (vector1.f90)

```

1  program main
2      implicit none
3      integer :: i, n=10
4      double precision :: v1(10), v2(10)
5
6      do i=1, n
7          v1(i) = dble(i)
8          v2(i) = 0.0d0
9      enddo
10
11     !$acc kernels
12     do i=1, n
13         v2(i) = v1(i) * 2.0d0
14     enddo
15     !$acc end kernels
16
17     do i=1,n
18         write(*,'(1H F8.2)',advance="NO")v1(i)
19     enddo
20     write(*,*)""
21     do i=1,n
22         write(*,'(1H F8.2)',advance="NO")v2(i)
23     enddo
24     write(*,*)""
25     end program main
26

```

➤ 単純なプログラムであれば  
kernels指示文で対象を指定  
するだけでGPU化が可能

# OpenACCの実行モデル

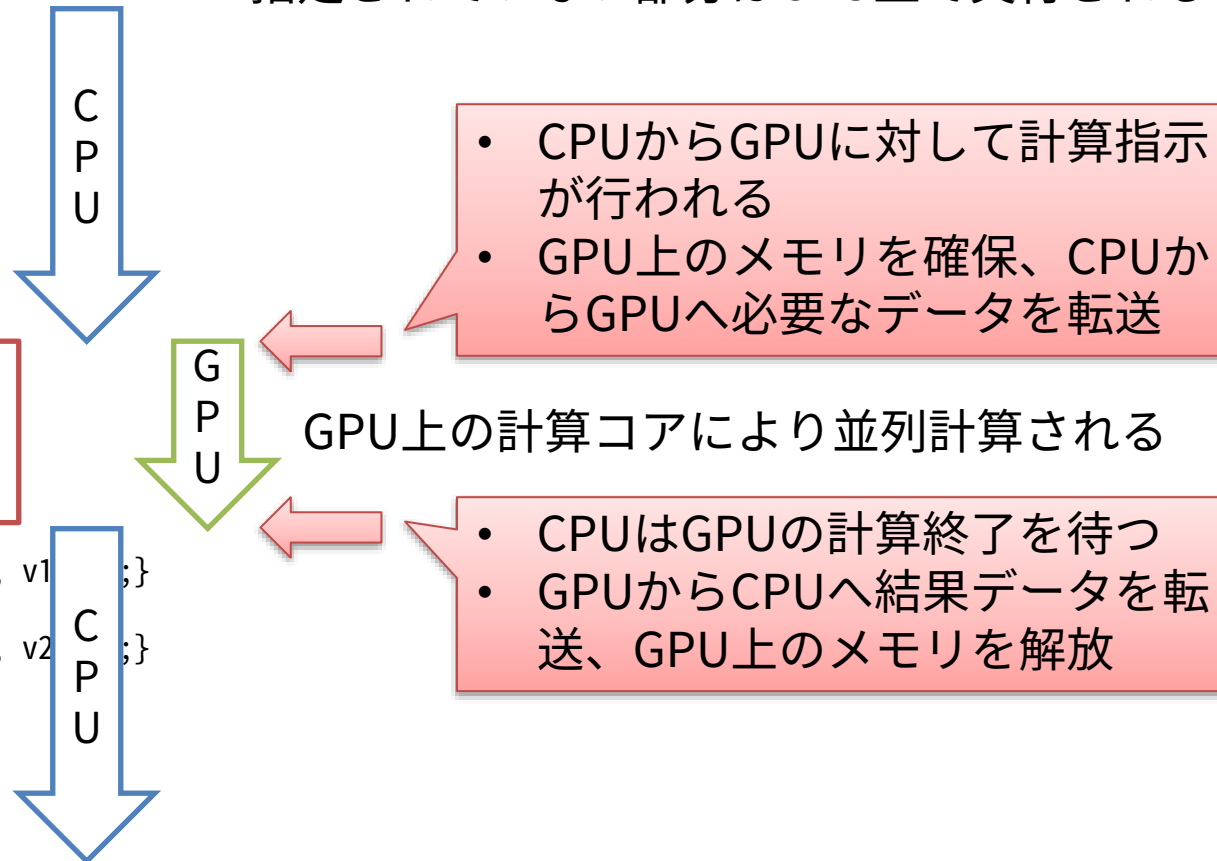
- 指定した部分だけがGPU上で実行される

```

1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     int i;
6     int n=10;
7     double v1[10], v2[10];
8
9     for(i=0; i<n; i++){
10         v1[i] = (double)(i+1);
11         v2[i] = 0.0;
12     }
13
14
15 #pragma acc kernels
16     for(i=0; i<n; i++){
17         v2[i] = v1[i] * 2.0;
18     }
19
20     for(i=0; i<n; i++){printf(" %.2f", v1[i]);}
21     printf(" ¥n");
22     for(i=0; i<n; i++){printf(" %.2f", v2[i]);}
23     printf(" ¥n");
24
25     return 0;
26 }

```

- 全体としてCPUが「主」、GPUが「従」の関係
- 指定されていない部分はCPU上で実行される



## 「不老」 Type IIサブシステムでOpenACCを使う方法（準備）

- ログインノード上でmodule loadコマンドを実行して準備を行う
- moduleコマンドの使い方（よく使うオプションとその意味）

module avail	利用可能なモジュールの一覧を表示
module load <modulename>	対象モジュールのload
module unload <modulename>	対象モジュールのunload
module list	load済みモジュールの一覧を表示
module purge	load済み全モジュールのunload

- ログイン時は何もloadされていない状態
- hpc\_sdk/22.2 をloadするとコンパイラ等が利用可能になる
  - module load hpc\_sdk/22.2
  - /22.2 は省略可能。省略した場合、システムのデフォルト設定のものがloadされる。
  - バージョンを省略しておけば今後さらに新しいものをインストールして提供する場合は新しいものが自動的に読み込まれるようになる。ただしコンパイル時と実行時でバージョンが異なると問題が起きることがある点には注意が必要。今回の講習会ではデフォルトの22.2を使用する。



# コンパイル例

- nvcまたはnvfortranでコンパイル

- -Minfo=accel OpenACC化に関する情報を出力
- -acc OpenACC指示文を有効化
- -gpu OpenACCの対象ハードウェア（対象GPUの種類）を指定
- -tp 対象CPUを指定

- -gpu（対象GPU）
  - nvaccelinfoの「Default Target」に対応
  - 例：Tesla P100ならcc60、V100ならcc70
  - 搭載されているGPUがV100であるためcc70を指定
- -tp（対象CPU）
  - hostにすればコンパイル環境が参照される
- 互換性のある古い環境を指定しても動くはずだが、コンパイラによる最適化の度合いが下がる。

- コンパイル例

```
$ nvc -Minfo=accel -acc -gpu=cc70 -tp=host -o v1_c_acc vector1.c
```

```
main:
```

```
16, Loop is parallelizable
```

```
Generating Tesla code
```

```
16, #pragma acc loop gang, vector(32)
```

```
16, Generating implicit copyout(v2[:]) [if not already present]
```

```
Generating implicit copyin(v1[:]) [if not already present]
```

```
$ nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -o v1_f_acc vector1.f90
```

```
main:
```

```
12, Generating implicit copyin(v1(:)) [if not already present]
```

```
Generating implicit copyout(v2(:)) [if not already present]
```

```
13, Loop is parallelizable
```

```
Generating Tesla code
```

```
13, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

※出力情報の具体的な読み方は後述する

データ転送やGPU上の計算コアの使い方はコンパイラがある程度判断してくれるが、最適でない（問題が起きる）こともある

# OpenACCプログラム向けのジョブスクリプトの書き方

```
#!/bin/bash
```

```
#PJM -L rscgrp=cx-workshop
```

```
#PJM -L node=1
```

```
#PJM -L elapse=1:00
```

```
#PJM -j
```

```
#PJM -S
```

```
module load hpc_sdk/22.2
```

```
export NVCOMPILER_ACC_TIME=1
```

```
./v1_c_acc
```

← リソースグループ名：cx-workshop

← 利用ノード数：1

← 利用時間制限：1分

← 標準出力と標準エラー出力を統合

← 統計情報を出力

← module loadでhpc\_sdkを使えるようにする

← GPUの実行時間内訳を確認するための設定（必須ではない）

← プログラムを実行

} 必須ではないが重要  
} お好みで

※module loadはコンパイル時のみではなく実行時にも必要（ライブラリなどを参照するため）

job\_vector1.sh として作成→ pjsub job\_vector1.sh でジョブ投入

- 赤字部分はバッチジョブの設定に関する情報
- 黒字部分は一般的なbashスクリプト処理

## リソースグループについての補足（消費ポイントについての注意点）

- 本講習会では`cx-workshop`を指定しているが、実際に利用する際にはリソースグループとGPU数の関係に注意が必要
- `cx-share`だけは常に1/4ノード、1GPUのみが確保される
- `cx-share`以外は常に1ノード単位で確保され、4GPUが利用される
- Type IIサブシステムの消費ポイント計算式は **利用GPU数** × 経過時間[s] × 0.007
- `cx-share`以外では常に **利用GPU数 = ノード数 × 4** であることに注意すること
- `cx-single`の`single`は1ノードの意味での`single`
  - GPU数をベースにするかノード数をベースにするかの問題、他のサブシステムとの一貫性からノード数をベースにした

# バッチジョブ実行と確認の例

赤文字が実行したコマンド、緑文字が実行結果として出力されたもの

```

$ pjsub job_vector1.sh
[INFO] PJM 0000 pjsub Job 75544 submitted.
$
$ pjstat2
JOB_ID      JOB_NAME    STATUS    GROUP    RSCGROUP    START_DATE    ELAPSE    NODE
75544      job_vector  QUEUED   center   cx-single    --/-- --:--:--  00:00:00    1
$
$ pjstat2
JOB_ID      JOB_NAME    STATUS    GROUP    RSCGROUP    START_DATE    ELAPSE    NODE
75544      job_vector  RUNNING  center   cx-single    (01/05 14:15)  00:00:00    1
$
$ pjstat2
JOB_ID      JOB_NAME    STATUS    GROUP    RSCGROUP    START_DATE    ELAPSE    NODE
75544      job_vector  RUNNING  center   cx-single    01/05 14:15:29  00:00:02    1
$
$ pjstat2
No unfinished job found.

```

←pjsubでジョブを投入、ジョブIDが通知される  
(設定を間違えた場合などはpjdelにこのIDを指定して削除する)

←pjstat2でジョブの状況を確認 (pjstatでも確認可能詳細は利用手引書を参照)

↑QUEUEDは実行されていない状態、開始時刻が決まるとSTART\_DATEに表示される

↑RUNNINGは実行中、START\_DATEが括弧付きの場合は開始準備中

←実行中/実行開始待ちのジョブがなくなった状態

## 実行例

- 正しく実行されている限り、GPU上で計算されていることは意識できない
  - あえていうなら性能

C版の場合

```
$ ./v1_c_acc
```

```
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
```

```
2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00
```

←元データ

←計算結果

Fortran版の場合

```
$ ./v1_f_acc
```

```
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
```

```
2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00
```

←元データ

←計算結果

# GPUが仕事をしていることを確認する 1/2

- 幾つかの環境変数を用いることでGPUの利用状況を確認可能
- 環境変数 NVCOMPILER\_ACC\_TIME
  - export NVCOMPILER\_ACC\_TIME=1 等と指定してから実行すると、計算や通信の回数や時間が確認できる

```
$ export NVCOMPILER_ACC_TIME=1
```

```
$ ./v1_c_acc
```

```
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
```

```
2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00
```

- ログインノード上での実行例 →

```
Accelerator Kernel Timing data
```

```
/home/center/a49979a/share/20211004/vector1.c
```

```
main NVIDIA devicenum=0
```

```
time(us): 30
```

```
16: compute region reached 1 time
```

```
16: kernel launched 1 time
```

```
grid: [1] block: [32]
```

```
elapsed time(us): total=26 max=26 min=26 avg=26
```

```
16: data region reached 2 times
```

```
16: data copyin transfers: 1
```

```
device time(us): total=14 max=14 min=14 avg=14
```

```
18: data copyout transfers: 1
```

```
device time(us): total=16 max=16 min=16 avg=16
```

GPU上での計算に関する情報  
計算回数、計算時間

CPU-GPU間の通信に関する情報  
通信回数、通信時間



## GPUが仕事をしていることを確認する 2/2

---

- NVCOMPILER\_ACC\_NOTIFYも有効
  - 1, 2, 4, 8, 16のビット組み合わせ（論理和）で指定、以下の情報が出力される
    - 1: GPUカーネル起動
    - 2: データ転送
    - 4: regionのentry/exit
    - 8: wait/sync
    - 16: デバイスメモリの確保と破棄
  - 例：export NVCOMPILER\_ACC\_NOTIFY=3
    - 3=1と2の論理和、GPUカーネル起動情報とデータ転送情報が出力される

# NVCOMPILER\_ACC\_NOTIFY情報の例

## 1: GPUカーネル起動

launch CUDA kernel file=/path/to/source.c function=main line=16 device=0 threadid=1 num\_gangs=1

num\_workers=1 vector\_length=32 grid=1 block=32

並列実行形状（後述）も確認できる

## 2: データ転送

upload CUDA data file=/path/to/source.c function=main line=16 device=0 threadid=1 variable=v1 bytes=80

download CUDA data file=/path/to/source.c function=main line=18 device=0 threadid=1 variable=v2 bytes=80

## 4: regionのentry/exit

Enter enter data construct file=/path/to/source.c function=main line=16 device=0 threadid=1

Leave enter data construct file=/path/to/source.c function=main line=16 device=0 threadid=1

Enter compute region file=/path/to/source.c function=main line=16 device=0 threadid=1

Leave compute region file=/path/to/source.c function=main line=16 device=0 threadid=1

Enter exit data construct file=/path/to/source.c function=main line=16 device=0 threadid=1

Leave exit data construct file=/path/to/source.c function=main line=18 device=0 threadid=1

## 8: wait/sync

Implicit wait file=/path/to/source.c function=main line=16 device=0 threadid=1

Implicit wait file=/path/to/source.c function=main line=16 device=0 threadid=1

Implicit wait file=/path/to/source.c function=main line=18 device=0 threadid=1

16のデバイスメモリ確保/破棄は省略

## 演習

- サンプルプログラム (vector1.c または vector1.f90) をコンパイルし、まずはそのままログインノードで実行してみる
  - ログインノードにもPCI Express版のV100が搭載されているためGPU向けプログラムを実行可能
  - (実行前に`export NVCOMPILER_ACC_TIME=1`しておく、GPUが本当に動いていることが確認できる)
- さらに、バッチジョブとして実行してみる
- バッチジョブとして実行できたら、さらに環境変数NVCOMPILER\_ACC\_TIMEやNVCOMPILER\_ACC\_NOTIFYをセットして実行してみる
- 注意
  - ログインノードは数が限られる (誰かがGPUを使っていると実行できない) ため、最初の動作確認など一部の場を除いてはバッチジョブとして実行すること

実際にやってみましょう

# OpenACCプログラムの構成

- 指示文：directive
- 指示節（指示句）：clause  
あまり厳密に考えなくてもよい

- 指示文により全てを記述する
  - 指示文：コンパイラに対して指示を行う特殊なコメント
    - C/C++：`#pragma acc ~`
    - Fortran：`!$acc ~`
    - 基本的には「無視してしまっても問題が起きない文」
      - コンパイラが対応していない場合もコンパイルと実行自体は可能、もちろんGPUは使えない
      - 対応環境向けコードとそれ以外を別のコードに分けなくても良いので管理が楽
        - » 対応の有無によって別のアルゴリズムを使い時などはどうにもならないが
- 具体的な指示文の例
  - 並列計算の方法を指示するもの
    - kernels, parallel
    - loop, seq, collapse
    - gang/num\_gangs, worker/num\_workers, vector/vector\_length
  - データの移動について指示するもの
    - data, enter/exit data, copy{in,out}, present, update, create, delete

## 並列化対象範囲の指定：kernelsとparallel

- 「この範囲内をGPU上で並列実行したい」ことを示す
  - kernelsとparallelではコンパイラによる解釈の仕方が異なる
    - parallel：基本的に利用者が細かく指定する
    - kernels：ある程度コンパイラが判断、手動で調整（上書き）可能
    - 最適化をしていくと結局同じようなコードになる、はずである
    - 範囲の途中で離脱するような構造は不可（forループのbreakなど）
  - 利用する指示節にも違いが生じる

### kernelsと共に利用するもの

- async / wait
- device\_type
- if
- default
- copy系

### parallelと共に利用するもの

- async / wait
- device\_type
- if
- default
- copy系
- num\_gangs / num\_workers / vector\_length
- reduction
- private

- OpenACCの仕様の元となった指示文規格の違いに起因しているようだ
- kernelsとparallelのどちらを用いても良いが、本講義ではkernelsを用いる

# 再掲：単純なOpenACCプログラムの例

- C (vector1.c)

```

1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     int i;
6     int n=10;
7     double v1[10], v2[10];
8
9     for(i=0; i<n; i++){
10         v1[i] = (double)(i+1);
11         v2[i] = 0.0;
12     }
13
14
15 #pragma acc kernels
16     for(i=0; i<n; i++){
17         v2[i] = v1[i] * 2.0;
18     }
19
20     for(i=0; i<n; i++)printf(" %.2f", v1[i]);
21     printf(" ¥n");
22     for(i=0; i<n; i++)printf(" %.2f", v2[i]);
23     printf(" ¥n");
24
25     return 0;
26 }

```

- Fortran (vector1.f90)

```

1 program main
2     implicit none
3     integer :: i, n=10
4     double precision :: v1(10), v2(10)
5
6     do i=1, n
7         v1(i) = dble(i)
8         v2(i) = 0.0d0
9     enddo
10
11
12 !$acc kernels
13     do i=1, n
14         v2(i) = v1(i) * 2.0d0
15     enddo
16 !$acc end kernels
17
18     do i=1,n
19         write(*,'(1H F8.2)',advance="NO")v1(i)
20     enddo
21     write(*,*)"

```

(OpenMPと同様に)

- C/C++では指示文直後のループや {} で括った部分（構造化ブロック）が指示文の適用対象となる
- Fortranではendで閉じる必要がある



# コンパイル時のメッセージ再確認

- C `$ nvc -Minfo=accel -acc -gpu=cc70 -tp=host -o v1_c_acc vector1.c`  
main:

16, Loop is parallelizable  
Generating Tesla code

16, #pragma acc loop gang, vector(32) /\* blockIdx.x threadIdx.x \*/

16, Generating implicit copyout(v2[:]) [if not already present]  
Generating implicit copyin(v1[:]) [if not already present]

コンパイラの判断によって  
copyin, copyoutという命令が生成された

- Fortran

`$ nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -o v1_f_`  
main:

12, Generating implicit copyin(v1(:)) [if not already present]  
Generating implicit copyout(v2(:)) [if not already present]

13, Loop is parallelizable  
Generating Tesla code

13, !\$acc loop gang, vector(32) ! blockIdx%x threadIdx%x

Tesla(NVIDIA GPU)向けのコードが生成された、  
ループの並列化が行われた

※implicit : 暗黙的な  
プログラムには書かれていなかったが、  
コンパイラが判断した

- どのように判断・処理されたのかを確認することは非常に重要

※ if not already present とblockidx や threadIdxについては後述

# 手動による適用：C版

- C (vector1.c)

```

3 int main(int argc, char **argv)
4 {
5     int i;
6     int n=10;
7     double v1[10], v2[10];
8
9     for(i=0; i<n; i++){
10         v1[i] = (double)(i+1);
11         v2[i] = 0.0;
12     }
13
14     #pragma acc kernels
15     for(i=0; i<n; i++){
16         v2[i] = v1[i] * 2.0;
17     }
18
19
20     for(i=0; i<n; i++)printf(" %.2f", v1[i]);
21     printf(" ¥n");
22     for(i=0; i<n; i++)printf(" %.2f", v2[i]);
23     printf(" ¥n");
24
25     return 0;
26 }

```



- C (vector2.c)

```

3 int main(int argc, char **argv)
4 {
5     int i;
6     int n=10;
7     double v1[10], v2[10];
8
9     for(i=0; i<n; i++){
10         v1[i] = (double)(i+1);
11         v2[i] = 0.0;
12     }
13
14     #pragma acc kernels copyin(v1[:]) copyout(v2[:])
15     #pragma acc loop gang, vector(32)
16     for(i=0; i<n; i++){
17         v2[i] = v1[i] * 2.0;
18     }
19
20     for(i=0; i<n; i++)printf(" %.2f", v1[i]);
21     printf(" ¥n");
22     for(i=0; i<n; i++)printf(" %.2f", v2[i]);
23     printf(" ¥n");
24 }

```

コンパイラの判断により、「copyin」「copyout」  
「loop gang, vector(32)」が自動的に挿入されていたと思えば良い

# 手動による適用：Fortran版

## Fortran (vector1.f90)

```

1 program main
2   implicit none
3   integer :: i, n=10
4   double precision :: v1(10), v2(10)
5
6   do i=1, n
7     v1(i) = dble(i)
8     v2(i) = 0.0d0
9   enddo
10
11  !$acc loop kernels
12  do i=1, n
13    v2(i) = v1(i) * 2.0d0
14  enddo
15  !$acc end kernels
16
17  do i=1,n
18    write(*,'(1H F8.2)',advance="NO")v1(i)
19  enddo
20  write(*,*)""
21  do i=1,n
22    write(*,'(1H F8.2)',adv
23  enddo
24  write(*,*)""
25
26 end program main

```



## Fortran (vector2.f90)

```

1 program main
2   implicit none
3   integer :: i, n=10
4   double precision :: v1(10), v2(10)
5
6   do i=1, n
7     v1(i) = dble(i)
8     v2(i) = 0.0d0
9   enddo
10
11  !$acc kernels copyin(v1(:)) copyout(v2(:))
12  !$acc loop gang, vector(32)
13  do i=1, n
14    v2(i) = v1(i) * 2.0d0
15  enddo
16  !$acc end kernels
17
18  do i=1,n
19    write(*,'(1H F8.2)',advance="NO")v1(i)
20  enddo
21  write(*,*)""
22  do i=1,n
23    write(*,'(1H F8.2)',adv
24  enddo
25  write(*,*)""
26 end program main

```

※end loopは不要  
(書いてもエラーには  
ならないようだ)

コンパイラの判断により、「copyin」「copyout」  
「loop gang, vector(32)」が自動的に挿入されていたと思えば良い

## 参考：指示文の継続行

- 指示文行を次の行に継続させることも可能
  - 長くなってしまったときなどに
- C/C++とFortranで少し違うので注意

```
#pragma acc kernels copyin(v1[:]) copyout(v2[:])
#pragma acc loop gang, vector(32)
  for(i=0; i<n; i++){
    v2[i] = v1[i] * 2.0;
  }
```



```
#pragma acc kernels ¥
copyin(v1[:]) copyout(v2[:])
#pragma acc loop gang, vector(32)
  for(i=0; i<n; i++){
    v2[i] = v1[i] * 2.0;
  }
```

- 末尾の¥（バックスラッシュ）で継続
- 継続行の先頭にはpragmaは不要

※これはOpenACCの都合というより  
CとFortranの仕様の問題  
(OpenMPでも同様に違いがある)

```
!$acc kernels copyin(v1(:)) copyout(v2(:))
!$acc loop gang, vector(32)
  do i=1, n
    v2(i) = v1(i) * 2.0d0
  enddo
!$acc end kernels
```



```
!$acc kernels &
!$acc copyin(v1(:)) copyout(v2(:))
!$acc loop gang, vector(32)
  do i=1, n
    v2(i) = v1(i) * 2.0d0
  enddo
!$acc end kernels
```

- 末尾の&で継続
- 継続行の先頭にも!\$accが必要

## データ転送に関する指示節

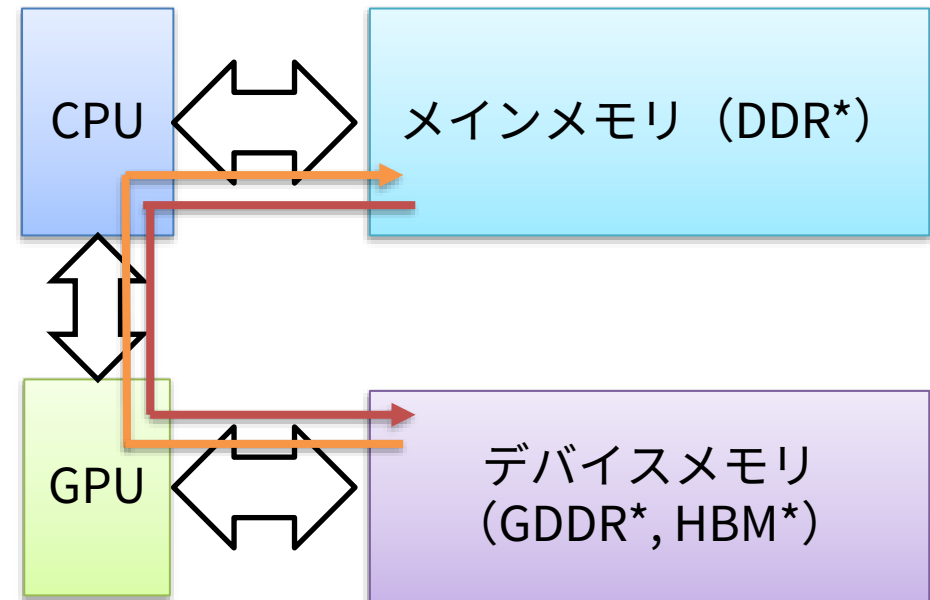
- kernels指示文に追加して配列のデータ転送を明示する
  - GPUカーネル実行前に、デバイスメモリを確保し、ホストからデバイスへコピーする
    - copyin
  - GPUカーネル終了後に、デバイスからホストへ書き戻し、デバイスメモリを破棄する
    - copyout
  - copyin + copyout
    - copy
  - デバイスメモリを確保するのみ
    - create
  - 既にデバイスメモリに存在していることをコンパイラに伝える
    - present
  - 存在していない場合のみcopy{,in,out}する
    - present\_or\_copy{,in,out}

※OpenACC2.5からは常に「present\_or\_\*」の挙動となり、存在していれば使い回してくれる。実際にどう扱われるかはコンパイル時のメッセージやNVCOMPILER\_ACC\_NOTIFYを用いれば確認できる。コンパイル時に[if not already present]と出ていたのはこのため。

※データを使い回す、該当するものが無ければ実行時エラー

## CPU-GPU間のデータ転送

- CPUとGPUは個別のメモリを持っており、直接相手側のメモリにアクセスできない
  - 例外あり、詳しくは後述
- 適切なデータ送受信を行わねば正しい計算が行えない
  - GPUカーネル起動時：メインメモリからデバイスメモリへのデータ転送
  - GPUカーネル終了後：デバイスメモリからメインメモリへのデータ転送
- 単純なプログラムでは自動的にデータ転送を行ってくれるが、ある程度複雑な場合には明示する必要がある
  - 特にC/C++では、コンパイラが配列の長さを把握できないために生じる実行時エラーをくらいやすいため注意が必要
  - GPUカーネルが生成されなかったり、実行時にエラーしたりする原因となる

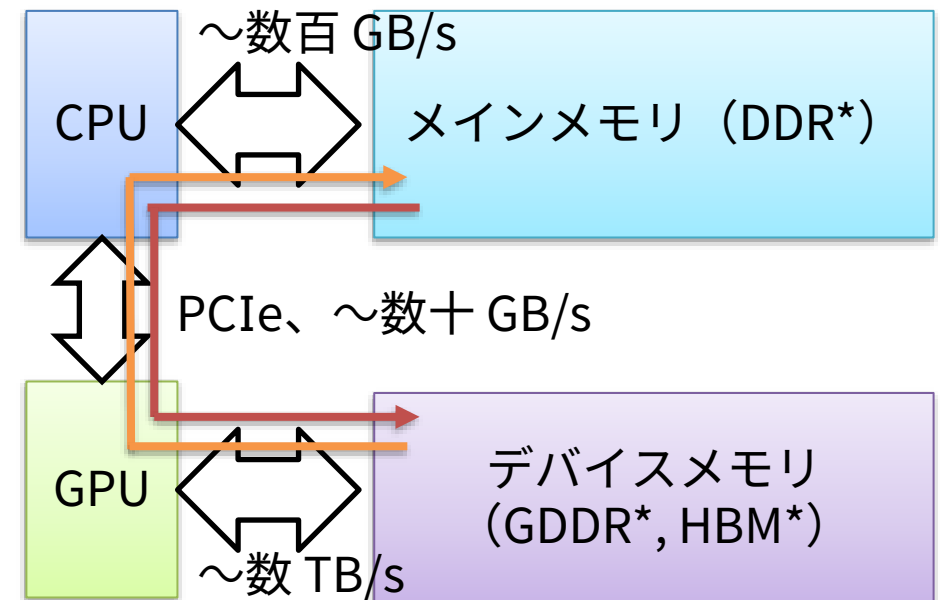




# データ転送速度

- メインメモリやデバイスメモリの転送速度に対してCPU-GPU間のデータ転送速度はずっと低速、頻繁な通信は避けたたい
  - CPU - メインメモリ
    - DDR4、100GB/s/socket程度 (STREAM Triad実測)
      - より高速なCPUでも理論性能で200GB/s/socket程度
  - GPU - デバイスメモリ
    - HBM2、P100で550GB/s程度、V100では800GB/s弱 (STREAM Triad実測)
  - CPU - GPU 他より低速
    - PCI Express Gen.3 x16
      - 理論性能でも最大16GB/s (×双方向)
        - Gen.4 でさらに2倍、それでも32GB/s (×双方向)
    - GPU - GPU
      - NVLink、20(Pascal) or 25(Volta)GB/s×双方向
        - 1GPUあたり4(Pascal) or 6(Volta)本のNVLink、GPU数によって最適なつなぎ方が変わる
        - Power系CPUではCPU-GPU間でもNVLinkが使える

※ A64FXなどHBM系を搭載したCPUであれば、同様のメモリを搭載したGPUと同程度の速度が出る



## データ転送範囲の指定

- 配列全体ではなく一部のみを送受信することも可能
- 注意：C/C++とFortranでは部分配列の指定方法が異なる
  - C/C++：先頭と長さを指定する

```
#pragma acc kernels copy(A[head:length])
```
  - Fortran：開始点と終了点を指定する

```
!$acc kernels copy(A(begin:end))
```
  - A[:N], A(:N) のような省略表記も可能（先頭からN要素が送られる）
- 単純なプログラムであればコンパイラが勝手に正しい長さを認識してくれることが多いが、長さの認識が間違っているためにプログラムがエラーすることも少なくない。
- コンパイラの実出力を確認し、認識が正しくない場合は手動で指定するよう気を付けること。

# 繰り返しデータ転送を行う場合の問題

- GPUカーネルを何度も実行する場合はどうなるだろうか？

```
int main(int argc, char **argv)
{
    int i, j, n=10;
    double v1[10];

    for(i=0; i<n; i++){
        v1[i] = (double)(i+1);
    }
}
```

```
program main
    implicit none
    integer :: i, j, n=10
    double precision :: v1(10)

    do i=1, n
        v1(i) = dble(i)
    enddo
```

```
for(j=0; j<10; j++){
#pragma acc kernels
    for(i=0; i<n; i++){
        v1[i] = v1[i] * 2.0;
    }
}
```

(vector3.c)

```
do j=1, 10
!$acc kernels
    do i=1, n
        v1(i) = v1(i) * 2.0d0
    enddo
!$acc end kernels
enddo
```

(vector3.f90)

# 繰り返しデータ転送を行う場合の問題

- GPUカーネルを何度も実行する場合はどうなるだろうか？

```
int main(int argc, char **argv)
{
    int i, j, n=10;
    double v1[10];

    for(i=0; i<n; i++){
        v1[i] = (double)(i+1);
    }
}
```

```
program main
    implicit none
    integer :: i, j, n=10
    double precision :: v1(10)

    do i=1, n
        v1(i) = dble(i)
    enddo
```

デバイスメモリの生成と破棄を繰り返してしまうため、本来は不要なはずの余計な時間がかかる

```
for(j=0; j<10; j++){
#pragma acc kernels
    for(i=0; i<n; i++){
        v1[i] = v1[i] * 2.0;
    }
}
```

(vector3.c)

```
do j=1, 10
!$acc kernels
    do i=1, n
        v1(i) = v1(i) * 2.0d0
    enddo
!$acc end kernels
enddo
```

(vector3.f90)

繰り返し

デバイスメモリの生成  
計算  
デバイスメモリの破棄

# データ転送のみを行う指示文

- data指示文

- ループ並列化のタイミング以外で、データのみを操作できる

```
#pragma acc data copyin(A) copyout(B)           !$acc data copyin(A) copyout(B)
  構造化ブロック                               構造化ブロック
                                               !$acc end data
```

- enter/exit data指示文

- 構造化ブロックを囲まずに自由な位置で送受信を行うことも可能

```
#pragma acc enter data copyin(A)                !$acc enter data copyin(A)

#pragma acc exit data copyout(B)                 !$acc exit data copyout(B)
```

- どちらを使っても良い

- enter/exit data指示文の方が便利だが、プログラムの見通しが悪くならないように注意が必要
  - GPU化範囲の前でとにかく全部送信したいとき？
  - 複数ソースコードにプログラムが分割されているとき？

- 配列長（転送範囲）の指定方法はkernelsの場合と同様

# data指示文によるデバイスメモリの生存期間の最適化

- GPUカーネルを何度も実行する場合などにdata指示文が有効

```
int main(int argc, char **argv)
{
    int i, j, n=10;
    double v1[10];

    for(i=0; i<n; i++){
        v1[i] = (double)(i+1);
    }
}
```

```
#pragma acc data copy(v1[:])
for(j=0; j<10; j++){
#pragma acc kernels
    for(i=0; i<n; i++){
        v1[i] = v1[i] * 2.0;
    }
}
```

(vector4.c)

```
program main
    implicit none
    integer :: i, j, n=10
    double precision :: v1(10)

    do i=1, n
        v1(i) = dble(i)
    enddo
enddo
```

```
!$acc data copy(v1(:))
do j=1, 10
!$acc kernels
    do i=1, n
        v1(i) = v1(i) * 2.0d0
    enddo
!$acc end kernels
enddo
!$acc end data
```

(vector4.f90)

デバイスメモリの生成  
繰り返し

計算

デバイスメモリの破棄

生成と破棄は最初と最後に  
のみ行われ、無駄がない



# data指示文によるデバイスメモリの生存期間の最適化

- GPUカーネルを何度も実行する場合などにdata指示文が有効

```
int main(int argc, char **argv)
{
    int i, j, n=10;
    double v1[10];

    for(i=0; i<n; i++){
        v1[i] = (double)(i+1);
    }
}
```

```
#pragma acc data copy(v1[:])
for(j=0; j<10; j++){
#pragma acc kernels present(v1)
    for(i=0; i<n; i++){
        v1[i] = v1[i] * 2.0;
    }
}
```

```
program main
    implicit none
    integer :: i, j, n=10
    double precision :: v

    do i=1, n
        v1(i) = dble(i)
    enddo
enddo
```

```
!$acc data copy(v1[:])
do j=1, 10
!$acc kernels present(v1)
    do i=1, n
        v1(i) = v1(i) * 2.0d0
    enddo
!$acc end kernels
enddo
!$acc end data
```

- 配列のアクセスに間接参照がある場合などはコンパイラが判断に失敗して繰り返しデータコピーをしてしまうこともある
- kernelsにpresent節を加えるとコンパイラの判断を上書きできる
- acc kernels present(v1)

デバイスメモリの生成  
繰り返し

計算

デバイスメモリの破棄

生成と破棄は最初と最後にのみ行われ、無駄がない

## 実習

- vector3.cとvector4.c（もしくはvector3.f90とvector4.f90）を実行して動作を比較してみよう
  - NVCOMPILER\_ACC\_TIMEやNVCOMPILER\_ACC\_NOTIFY=2を設定して実行し、出力を比較してみよう（通信回数や時間を比較してみよう）
- プログラムの動作を理解できたら、配列v1の値を2倍するのではなく、結果を配列v2に格納したり、配列v1とv2の加算結果を配列v3に格納するなど、少し異なる動作をするプログラムも作成してみよう
  - 最適と思われる指示文の入れ方は？
  - コンパイラは最適な指示文を挿入してくれるだろうか？

実際にやってみましょう

# 実習：実行例

```
Accelerator Kernel Timing data
/home/center/a49979a/share/20211004/vector3.c
main NVIDIA devicenum=0
time(us): 117
15: compute region reached 10 times
15: kernel launched 10 times
grid: [1] block: [32]
elapsed time(us): total=131 max=28 min=11 avg=13
15: data region reached 20 times
15: data copyin transfers: 10
device time(us): total=55 max=14 min=4 avg=5
17: data copyout transfers: 10
device time(us): total=62 max=16 min=5 avg=6
```



```
Accelerator Kernel Timing data
/home/center/a49979a/share/20211004/vector4.c
main NVIDIA devicenum=0
time(us): 30
13: data region reached 2 times
13: data copyin transfers: 1
device time(us): total=13 max=13 min=13 avg=13
18: data copyout transfers: 1
device time(us): total=17 max=17 min=17 avg=17
15: compute region reached 10 times
15: kernel launched 10 times
grid: [1] block: [32]
elapsed time(us): total=116 max=26 min=9 avg=11
```

```
Accelerator Kernel Timing data
/home/center/a49979a/share/20211004/vector3.f90
main NVIDIA devicenum=0
time(us): 110
12: compute region reached 10 times
13: kernel launched 10 times
grid: [1] block: [32]
elapsed time(us): total=127 max=26 min=10 avg=12
12: data region reached 20 times
12: data copyin transfers: 10
device time(us): total=53 max=13 min=4 avg=5
16: data copyout transfers: 10
device time(us): total=57 max=13 min=4 avg=5
```



```
Accelerator Kernel Timing data
/home/center/a49979a/share/20211004/vector4.f90
main NVIDIA devicenum=0
time(us): 29
10: data region reached 2 times
10: data copyin transfers: 1
device time(us): total=14 max=14 min=14 avg=14
18: data copyout transfers: 1
device time(us): total=15 max=15 min=15 avg=15
12: compute region reached 10 times
13: kernel launched 10 times
grid: [1] block: [32]
elapsed time(us): total=148 max=34 min=10 avg=14
12: data region reached 20 times
```

## 動的な配列のデータ送受信

- 部分転送時の範囲指定には変数を用いても良い
- 動的に確保した配列などコンパイル時に配列の長さが分からないものは長さを明示する必要がある
  - 間接参照をしているときなどに明示的な指定が必要となりやすい

(allocate1.c)

```
double *v1, *v2;
int *index;
v1 = (double*)malloc(sizeof(double)*n);
v2 = (double*)malloc(sizeof(double)*n);
index = (int*)malloc(sizeof(int)*n);
#pragma acc kernels
for(i=0; i<n; i++){
  v2[i] = v1[index[i]] * 2.0;
}
```

(allocate1.f90)

```
double precision, allocatable :: v1(:), v2(:)
integer, allocatable :: index(:)
allocate(v1(n), v2(n), index(n))
!$acc kernels
  do i=1, n
    v2(i) = v1(index(i)) * 2.0d0
  enddo
!$acc end kernels
```

- v1の範囲（長さ）がうまく認識できず、コンパイルはできるが実行時エラーや正しくない結果になることがある（特にC言語版）
- `copyin(v1[n])` および `copyin(v1(n))` を加えると正しく動作する

## 動的な配列のデータ送受信

- 部分転送時の範囲指定には変数を用いても良い
- 動的に確保した配列などコンパイル時に配列の長さが分からないものは長さを明示する必要がある
  - 間接参照をしているときなどに明示的な指定が必要となりやすい

(allocate2.c)

```
double *v1, *v2;
int *index;
v1 = (double*)malloc(sizeof(double)*n);
v2 = (double*)malloc(sizeof(double)*n);
index = (int*)malloc(sizeof(int)*n);
#pragma acc kernels copyin(v1[n])
for(i=0; i<n; i++){
    v2[i] = v1[index[i]] * 2.0;
}
```

(allocate2.f90)

```
double precision, allocatable :: v1(:), v2(:)
integer, allocatable :: index(:)
allocate(v1(n), v2(n), index(n))
!$acc kernels copyin(v1(n))
    do i=1, n
        v2(i) = v1(index(i)) * 2.0d0
    enddo
!$acc end kernels
```

- v1の範囲（長さ）がうまく認識できず、コンパイルはできるが実行時エラーや正しくない結果になることがある（特にC言語版）
- `copyin(v1[n])` および `copyin(v1(n))` を加えると正しく動作する

# コンパイル時の出力の比較

## allocate1.c

```
$ nvc -Minfo=accel -acc -gpu=cc70 -tp=host allocate1.c
```

```
main:
```

- ```
21, Accelerator restriction: size of the GPU copy of v1 is unknown
Complex loop carried dependence of v1-> prevents parallelization
Loop carried dependence of v2-> prevents parallelization
Loop carried backward dependence of v2-> prevents vectorization
Accelerator serial kernel generated
Generating Tesla code
21, #pragma acc loop seq
21, Generating implicit copyin(index[:10]) [if not already present]
Generating implicit copyout(v2[:10]) [if not already present]
Generating implicit copyin(v1[:]) [if not already present]
Complex loop carried dependence of v1-> prevents parallelization
Loop carried dependence of v2-> prevents parallelization
Loop carried backward dependence of v2-> prevents vectorization
```

## allocate2.c

```
$ nvc -Minfo=accel -acc -gpu=cc70 -tp=host allocate2.c
```

```
main:
```

- ```
21, Generating copyin(v1[:n]) [if not already present]
Complex loop carried dependence of v1-> prevents parallelization
Loop carried dependence of v2-> prevents parallelization
Loop carried backward dependence of v2-> prevents vectorization
Accelerator serial kernel generated
Generating Tesla code
21, #pragma acc loop seq
21, Generating implicit copyout(v2[:10]) [if not already present]
Generating implicit copyin(index[:10]) [if not already present]
Loop carried dependence of v2-> prevents parallelization
Loop carried backward dependence of v2-> prevents vectorization
```

## allocate1.f90

```
$ nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host allocate1.f90
```

```
main:
```

- ```
14, Generating implicit copyin(v1(:)) [if not already present]
Generating implicit copyout(v2(1:10)) [if not already present]
Generating implicit copyin(index(1:10)) [if not already present]
15, Loop is parallelizable
Generating Tesla code
15, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

配列v1の長さがわからない模様  
※ただしFortranについてはこのままでも  
(結果的に) 問題ない

## allocate2.f90

```
$ nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host allocate2.f90
```

```
main:
```

- ```
14, Generating implicit copyin(index(1:10)) [if not already present]
Generating implicit copyout(v2(1:10)) [if not already present]
Generating copyin(v1(:n)) [if not already present]
15, Loop is parallelizable
Generating Tesla code
15, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```



## 実行結果の比較

配列v1初期値	:	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
配列v2初期値	:	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
index (C言語版)	:	0.00	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00
index (F90版)	:	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.00
正しいv2の結果	:	2.00	4.00	6.00	8.00	10.00	12.00	14.00	16.00	18.00	20.00
allocate1.cのv2結果	:	2.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
allocate1.f90のv2結果	:	2.00	4.00	6.00	8.00	10.00	12.00	14.00	16.00	18.00	20.00
allocate2.cのv2結果	:	2.00	4.00	6.00	8.00	10.00	12.00	14.00	16.00	18.00	20.00
allocate2.f90のv2結果	:	2.00	4.00	6.00	8.00	10.00	12.00	14.00	16.00	18.00	20.00

- allocate1.cでは配列長をうまく認識できていないためか計算結果がおかしい。
- どの時点で問題が起きているのか詳細を確認することはできるだろうか？

# NVCOMPILER\_ACC\_NOTIFYをセットして動作を確認

```
$ export NVCOMPILER_ACC_NOTIFY=2
```

←2に設定するとCPU-GPU間の通信の情報が出力される

allocate1.cではv1が8byte、つまり1要素しか送信されていないことがわかる

```
$ ./a1_c_acc
upload CUDA data file=/home/center/a49979a/share/20211004/allocate1.c function=main line=21 device=0 threadid=1 variable=index bytes=40
upload CUDA data file=/home/center/a49979a/share/20211004/allocate1.c function=main line=21 device=0 threadid=1 variable=v1 bytes=8
download CUDA data file=/home/center/a49979a/share/20211004/allocate1.c function=main line=23 device=0 threadid=1 variable=v2 bytes=80
 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
 2.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

allocate1.f90は正しく10要素(80byte)転送できたようだ

```
$ ./a1f_f_acc
upload CUDA data file=/home/center/a49979a/share/20211004/allocate1.f90 function=main line=14 device=0 threadid=1 variable=v1 bytes=80
upload CUDA data file=/home/center/a49979a/share/20211004/allocate1.f90 function=main line=14 device=0 threadid=1 variable=index bytes=40
download CUDA data file=/home/center/a49979a/share/20211004/allocate1.f90 function=main line=18 device=0 threadid=1 variable=v2 bytes=80
 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
 2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00
```

allocate2.cも正しく10要素(80byte)転送できたようだ

```
$ ./a2c_c_acc
upload CUDA data file=/home/center/a49979a/share/20211004/allocate2.c function=main line=21 device=0 threadid=1 variable=v1 bytes=80
upload CUDA data file=/home/center/a49979a/share/20211004/allocate2.c function=main line=21 device=0 threadid=1 variable=index bytes=40
download CUDA data file=/home/center/a49979a/share/20211004/allocate2.c function=main line=23 device=0 threadid=1 variable=v2 bytes=80
 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
 2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00
```

```
$ ./a2f_f_acc
upload CUDA data file=/home/center/a49979a/share/20211004/allocate2.f90 function=main line=14 device=0 threadid=1 variable=v1 bytes=80
upload CUDA data file=/home/center/a49979a/share/20211004/allocate2.f90 function=main line=14 device=0 threadid=1 variable=index bytes=40
download CUDA data file=/home/center/a49979a/share/20211004/allocate2.f90 function=main line=18 device=0 threadid=1 variable=v2 bytes=80
 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
 2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00
```

# 多次元配列の送受信

- 多次元配列の送受信も可能
  - ただし、連続したメモリの範囲しか扱えない
    - 低次元分は全て転送する必要がある
  - C/C++ `#pragma acc data copyin(A[head:length][0:N])`
    - C/C++は**右側の次元が低次元**
  - Fortran `!$acc data copyin(A(1:N, begin:end))`
    - Fortranは**左側の次元が低次元**

➤ 物理的なメモリアドレスは一次元

0x**00	0x**01	0x**02	0x**03	0x**04	0x**05	0x**06	0x**07	0x**08
--------	--------	--------	--------	--------	--------	--------	--------	--------

➤ C言語の二次元（多次元）配列は行方向優先

0x**00 [0,0]	0x**01 [0,1]	0x**02 [0,2]
0x**03 [1,0]	0x**04 [1,1]	0x**05 [1,2]
0x**06 [2,0]	0x**07 [2,1]	0x**08 [2,2]

➤ Fortranの二次元（多次元）配列は列方向優先

0x**00 [0,0]	0x**03 [1,0]	0x**06 [2,0]
0x**01 [0,1]	0x**04 [1,1]	0x**07 [2,1]
0x**02 [0,2]	0x**05 [1,2]	0x**08 [2,2]

このような形状で送ること**はできない**


## ただし……

- C言語ではよく二次元配列を2段階のmallocで確保するが、これをやるとメモリの連続性が失われてしまう

```
a = (double**)malloc(sizeof(double*)*n);
for(i=0; i<n; i++){
    a[i] = (double*)malloc(sizeof(double)*n);
}
```

- OpenACCでどう扱われるのか（気を利かせて転送してくれるのか）よくわからないため、本講習会ではC言語で二次元配列（行列）を動的確保する際は一次元配列で実装している
  - 送ってくれるようなのだが、必ず送れるのか不安が……

```
a = (double*)malloc(sizeof(double)*n*n);
for(i=0; i<n; i++){
    for(j=0; j<n; j++){
        a[i*n+j] = (double)(i+1) + (double)(j+1)*0.01;
    }
}
```

 a[i][j] に相当

※ Fortranは一命令で二次元配列をallocateできるから大丈夫だろうと判断し、二次元配列をそのまま用いている

# データの更新

- data指示文内のGPUカーネル間でデータの確認や更新を行いたい場合はどうすれば良いだろうか？

```
#pragma acc data copy(v1)
{
  #pragma acc kernels
  for(i=0; i<n; i++){
    v1[i] = v1[i] * 2.0;
  }
}
```

- 並列化ループの途中で値を確認したい・更新したい
- たとえばここでv1の値を出力したら何が出力されるだろうか？

```
#pragma acc kernels
for(i=0; i<n; i++){
  v1[i] = v1[i] * 2.0;
}
}
```

```
!$acc data copy(v1)
!$acc kernels
do i=1, n
  v1(i) = v1(i) * 2.0d0
enddo
!$acc end kernels
```

```
!$acc kernels
do i=1, n
  v1(i) = v1(i) * 2.0d0
enddo
!$acc end kernels
!$acc end data
```

- data範囲のあとであれば計算結果が全てメインメモリに書き戻されているのだが……？

```
for(i=0; i<n; i++){
  printf(" %.2f", v1[i]);
}
printf(" ¥n");
```

(update1.c)

```
do i=1,n
  write(*,'(1H F8.2)',advance="NO")v1(i)
enddo
write(*,*)" "
```

(update1.f90)

# データの更新

- data指示文内のGPUカーネル間でデータの確認や更新を行いたい場合はどうすれば良いだろうか？

```
#pragma acc data copy(v1)
{
  #pragma acc kernels
  for(i=0; i<n; i++){
    v1[i] = v1[i] * 2.0;
  }
}
```

- 並列化ループの途中で値を確認したい・更新したい
- たとえばここでv1の値を出力したら何が出力されるだろうか？

```
#pragma acc kernels
for(i=0; i<n; i++){
  v1[i] = v1[i] * 2.0;
}
}
```

```
!$acc data copy(v1)
!$acc kernels
do i=1, n
  v1(i) = v1(i) * 2.0d0
enddo
!$acc end kernels
```

⇒ 計算前の値が出力されてしまう

```
!$acc kernels
do i=1, n
  v1(i) = v1(i) * 2.0d0
enddo
!$acc end kernels
!$acc end data
```

- data範囲のあとであれば計算結果が全てメインメモリに書き戻されているのだが……？

```
for(i=0; i<n; i++){
  printf(" %.2f", v1[i]);
}
printf(" ¥n");
```

(update1.c)

```
do i=1,n
  write(*,'(1H F8.2)',advance="NO")v1(i)
enddo
write(*,*)" "
```

(update1.f90)



## データの更新：update指示文

- デバイスメモリの生成・破棄を伴わないデータ転送（更新）にはupdateを用いる
  - ホストからデバイス：update device
  - デバイスからホスト：update host または update self
  - 一部のみの更新も可能、範囲の指定方法はdata指示文と同様

```
#pragma acc data copy(v1)
{
  #pragma acc kernels
  for(i=0; i<n; i++){
    v1[i] = v1[i] * 2.0;
  }

  #pragma acc update host(v1)
  for(i=0; i<n; i++){
    printf(" %.2f", v1[i]);
  }
  printf(" ¥n");

  #pragma acc kernels
  for(i=0; i<n; i++){
    v1[i] = v1[i] * 2.0;
  }
}
```

(update2.c)

```
!$acc data copy(v1)
!$acc kernels
do i=1, n
  v1(i) = v1(i) * 2.0d0
enddo
!$acc end kernels

!$acc update host(v1)
do i=1,n
  write(*, '(1H F8.2)', advance="NO")v1(i)
enddo
write(*,*)""

!$acc kernels
do i=1, n
  v1(i) = v1(i) * 2.0d0
enddo
!$acc end kernels
!$acc end data
```

(update2.f90)

# 内容

- OpenACCを学ぶ前に
  - 並列計算の基礎
  - GPUとOpenACC
- 単純なベクトル計算を題材としてOpenACCの基本を学ぶ
  - コンパイルの仕方、実行の仕方
  - CPU-GPU間のデータ転送について
- 行列積を題材としてOpenACCの基本を学ぶ
  - 並列化ループの指定方法について
- その他、OpenACCに関する話題
  - 最適化のための一般的なヒント
  - デバッグとプロファイリング
  - 複数GPUの活用 など
- まとめ
- 参考（演習課題）：CG法のOpenACC化

## ループ並列化方法の指定

---

- ここまではkernels節のみを用いて並列化対象の指定を行った
- しかし現実のプログラム（複雑なプログラム）ではコンパイラにはループ並列化が可能であるかを判断できないことがある
- kernels指示文でループを囲んでも、コンパイラが並列化できないと判断することがある
- プログラマが並列化の判断を明示すれば並列化させることができる
- 逆に、並列化させないこともできる

# loop指示文による並列化判断の指定

- loop指示文
  - 並列化対象ループを指定する
- loop指示文に加えて以下の指示節を使うことで判断・動作を制御・修正することができる
  - **independent**指示節と**seq**指示節
    - 対象ループを並列実行するか逐次実行するかを明示する
    - 強制力があり、コンパイラによる判断は行われなくなる
  - **collapse(n)**指示節：collapse(2), collapse(3)など
    - 多重ループをまとめて並列化する
    - 並列度の低い階層ループの並列化などに極めて重要
  - **reduction**指示節：reduction(+:a) など
    - 計算結果の集約などを行う
    - 多くの場合はコンパイラが正しく判断してくれるため書く必要はないが、コンパイラがどのように判断したか確認できるように読み方を覚えておくと良い

# 三重ループ構造による単純な行列積プログラムの例

## • C (matmul1.c)

```

double a[N*N], b[N*N], c[N*N];
初期化等は省略
27 #pragma acc kernels
28 for(i=0; i<N; i++){
29     for(j=0; j<N; j++){
30         for(k=0; k<N; k++){
31             c[i*N+j] += a[i*N+k] * b[k*N+j];
32         }
33     }
34 }

```

n=10で実行して計算時間を比較してみた

C

```

time(us): 62
28: compute region reached 1 time
30: kernel launched 1 time
    grid: [1] block: [32]
    device time(us): total=13 max=13 min=13 avg=13
    elapsed time(us): total=549 max=549 min=549 avg=549

```

Fortran

```

time(us): 45
29: compute region reached 1 time
32: kernel launched 1 time
    grid: [1] block: [128]
    device time(us): total=4 max=4 min=4 avg=4
    elapsed time(us): total=491 max=491 min=491 avg=491

```

## • Fortran (matmul1.f90)

```

double precision :: a(N,N), b(N,N), c(N,N)
初期化等は省略
29 !$acc kernels
30 do i=1, N
31     do j=1, N
32         do k=1, N
33             c(j,i) = c(j,i) + a(k,i) * b(j,k)
34         enddo
35     enddo
36 enddo
37 !$acc end kernels

```

- Cがやや遅いが、もの凄く遅いという程ではない (blockの数を見ると妥当性があるのだが、そのあたりは後ほど)
- (そもそも問題サイズが小さすぎる、もっと大きなサイズでチェックすべきではある)

# 三重ループ構造による単純な行列積プログラムの例：動的確保の場合

- C (matmul1a.c)

```

double *a=NULL, *b=NULL, *c=NULL;
mallocなどは省略
34 #pragma acc kernels
35 for(i=0; i<N; i++){
36     for(j=0; j<N; j++){
37         for(k=0; k<N; k++){
38             c[i*N+j] += a[i*N+k] * b[k*N+j];
39         }
40     }
41 }

```

- Fortran (matmul1a.f90)

```

double precision, allocatable :: a(:,,:), b(:,,:), c(:,,:)
allocateなどは省略
36 !$acc kernels
37 do i=1, N
38     do j=1, N
39         do k=1, N
40             c(j,i) = c(j,i) + a(k,i) * b(j,k)
41         enddo
42     enddo
43 enddo
44 !$acc end kernels

```

- 実行して計算時間を比較しようとする、  
そもそもC言語版は実行時にエラーしてしまう

```

$ ./mm1a_c_acc
N = 10
out = 1

```

```

Failing in Thread:1
call to cuStreamSynchronize returned error 700:
Illegal address during kernel execution

```

```

$ ./mm1a_f_acc

```

```

n = 10

```

```

result:

```

58.96	59.06	59.17	59.27	59.38	59.48	59.59	59.69	59.80	59.91
114.06	114.26	114.47	114.67	114.88	115.08	115.29	115.49	115.70	115.91
169.16	169.46	169.77	170.07	170.38	170.68	170.99	171.29	171.60	171.91
中略									
444.66	445.46	446.27	447.07	447.88	448.68	449.49	450.29	451.10	451.91
499.76	500.66	501.57	502.47	503.38	504.28	505.19	506.09	507.00	507.90
554.86	555.86	556.87	557.87	558.88	559.88	560.89	561.89	562.90	563.90

これらはエラーではないので無視して良い

Warning: ieee\_inexact is signaling  
FORTRAN STOP



# コンパイラの出力するメッセージの確認

## • C (matmul1a.c)

```
$ nvc -Minfo=accel -acc -gpu=cc70 -tp=host -o mm1a_c_acc matmul1a.c
main:
```

```
35, Accelerator restriction: size of the GPU copy of c,a is unknown
Complex loop carried dependence of b->a-> prevents parallelization
Loop carried dependence of c-> prevents parallelization
Loop carried backward dependence of c-> prevents vectorization
Complex loop carried dependence of c-> prevents parallelization
Accelerator serial kernel generated
Generating NVIDIA GPU code
35, #pragma acc loop seq
36, #pragma acc loop seq
37, #pragma acc loop seq
35, Generating implicit copyin(a[:]) [if not already present]
Generating implicit copy(c[:]) [if not already present]
Loop carried backward dependence of c-> prevents vectorization
36, Complex loop carried dependence of b->,a->,c-> prevents parallelization
37, Accelerator restriction: size of the GPU copy of b is unknown
Complex loop carried dependence of b->,a->,c-> prevents parallelization
Loop carried dependence due to exposed use of c[(N)*i1+i2] prevents parallelization
Generating implicit copyin(b[:]) [if not already present]
Complex loop carried dependence of c-> prevents parallelization
```

seq = sequential  
ループが逐次実行される  
ことを意味する

- C言語版は配列のサイズが認識できていない  
(データの転送以前にそもそもメモリの確保が正しく行  
えていないと考えられる)
- カーネルの並列化もできていない (できても困るが)

## • Fortran (matmul1a.f90)

```
$ nvfortran -g -Minfo=accel -acc -gpu=cc70 -tp=host -o mm1a_f_acc matmul1a.f90
main:
```

```
36, Generating implicit copyin(b(1:N,1:N)) [if not already present]
Generating implicit copy(c(1:N,1:N)) [if not already present]
Generating implicit copyin(a(1:N,1:N)) [if not already present]
37, Loop is parallelizable
38, Loop is parallelizable
39, Complex loop carried dependence of c prevents parallelization
Loop carried dependence of c prevents parallelization
Loop carried backward dependence of c prevents vectorization
Inner sequential loop scheduled on accelerator
Generating NVIDIA GPU code
37, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
38, ! blockidx%x threadidx%x auto-collapsed
39, !$acc loop seq
39, Complex loop carried dependence of c prevents parallelizationz
```

- Fortran言語版は配列のサイズが正しく認識できており、  
データ転送も問題なく行われているように見える
- カーネルの並列化もできているようだ
- 3重ループの外側2つは並列化され (auto-collapsedについては後述)、最内ループは逐次実行。cへの書き込みは並列化すると競合するため、妥当な判断。

# 動的確保 + copy指示文の追加

## • C (matmul2a.c)

```

34 #pragma acc kernels copy(c[N*N]) copyin(a[N*N],b[N*N])
35 for(i=0; i<N; i++){
36     for(j=0; j<N; j++){
37         for(k=0; k<N; k++){
38             c[i*N+j] += a[i*N+k] * b[k*N+j];
39         }
40     }
41 }

```

```

$ nvc -Minfo=accel -acc -gpu=cc70 -tp=host -o mm2a_c_acc matmul2a.c
main:

```

```

35, Generating copyin(b[:N*N]) [if not already present]
Generating copy(c[:N*N]) [if not already present]
Generating copyin(a[:N*N]) [if not already present]
Complex loop carried dependence of b->,a-> prevents parallelization
Loop carried dependence of c-> prevents parallelization
Loop carried backward dependence of c-> prevents vectorization
Complex loop carried dependence of c-> prevents parallelization
Accelerator serial kernel generated
Generating NVIDIA GPU code

```

```

35, #pragma acc loop seq
36, #pragma acc loop seq
37, #pragma acc loop seq

```

- C言語版は配列サイズの認識問題が解決
- カーネルの並列化はできていないまま

```

35, Loop carried dependence of c-> prevents parallelization
Loop carried backward dependence of c-> prevents vectorization
Complex loop carried dependence of c-> prevents parallelization
36, Complex loop carried dependence of b->,a->,c-> prevents parallelization
37, Complex loop carried dependence of b->,a->,c-> prevents parallelization
Loop carried dependence due to exposed use of c[(N)*i1+i2] prevents parallelization

```

## • Fortran (matmul2a.f90)

```

36 !$acc kernels copy(c(N,N)) copyin(a(N,N),b(N,N))
37 do i=1, N
38     do j=1, N
39         do k=1, N
40             c(j,i) = c(j,i) + a(k,i) * b(j,k)
41         enddo
42     enddo
43 enddo
44 !$acc end kernels

```

```

$ nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -o mm2a_f_acc matmul2a.f90
main:

```

```

36, Generating copy(c(N,N)) [if not already present]
Generating copyin(b(N,N),a(N,N)) [if not already present]
37, Loop is parallelizable
38, Loop is parallelizable
39, Complex loop carried dependence of c prevents parallelization
Loop carried dependence of c prevents parallelization
Loop carried backward dependence of c prevents vectorization
Inner sequential loop scheduled on accelerator
Generating NVIDIA GPU code
37, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
38, ! blockidx%x threadidx%x auto-collapsed
39, !$acc loop seq
39, Loop carried dependence of c prevents parallelization
Loop carried backward dependence of c prevents vectorization

```

- Fortran版は特に変わらず

## 動的確保 + copy指示文の追加：実行時間の確認

- C (matmul2a.c)

```
$ NVCOMPILER_ACC_TIME=1 ./mm2a_c_acc
N = 10
out = 1
result:
省略
```

Accelerator Kernel Timing data

/data/group1/a49979a/share/20220627/matmul2a.c

main NVIDIA devicenum=0

time(us): 108

35: compute region reached 1 time

35: kernel launched 1 time

grid: [1] block: [1]

device time(us): total=65 max=65 min=65 avg=65

elapsed time(us): total=530 max=530 min=530 avg=530

35: data region reached 2 times

35: data copyin transfers: 3

device time(us): total=31 max=16 min=4 avg=10

41: data copyout transfers: 1

device time(us): total=12 max=12 min=12 avg=12

- Fortran (matmul2a.f90)

```
$ NVCOMPILER_ACC_TIME=1 ./mm2a_f_acc
n = 10
result:
省略
```

Warning: ieee\_inexact is signaling

FORTRAN STOP

Accelerator Kernel Timing data

/data/group1/a49979a/share/20220627/matmul2a.f90

main NVIDIA devicenum=0

time(us): 62

36: compute region reached 1 time

39: kernel launched 1 time

grid: [1] block: [128]

device time(us): total=6 max=6 min=6 avg=6

elapsed time(us): total=510 max=510 min=510 avg=510

36: data region reached 2 times

36: data copyin transfers: 3

device time(us): total=40 max=19 min=10 avg=13

44: data copyout transfers: 1

device time(us): total=16 max=16 min=16 avg=16

- C版はFortran版と比べてかなり遅い
- GPUカーネルの並列化ができていないので当然ではある

# loop指示文の追加

## • C (matmul3a.c)

```

34 #pragma acc kernels copy(c[N*N]) copyin(a[N*N],b[N*N])
35 #pragma acc loop independent
36   for(i=0; i<N; i++){
37 #pragma acc loop independent
38   for(j=0; j<N; j++){
39 #pragma acc loop seq
40     for(k=0; k<N; k++){
41       c[i*N+j] += a[i*N+k] * b[k*N+j];
42     }
43   }
44 }

```

並列実行する

→ 並列実行しない

- データ転送もGPUによる計算も問題なさそう

```
$ nvc -Minfo=accel -acc -gpu=cc70 -tp=host -o mm3a_c_acc matmul3a.c
main:
```

```

36, Generating copyin(b[:N*N]) [if not already present]
Generating copy(c[:N*N]) [if not already present]
Generating copyin(a[:N*N]) [if not already present]
Loop is parallelizable
38, Loop is parallelizable
40, Generating NVIDIA GPU code
36, #pragma acc loop gang, vector(128) collapse(2)
/* blockIdx.x threadIdx.x */
38, /* blockIdx.x threadIdx.x auto-collapsed */
40, #pragma acc loop seq

```

## • Fortran (matmul3a.f90)

```

36 !$acc kernels copy(c(N,N)) copyin(a(N,N),b(N,N))
37 !$acc loop independent
38   do i=1, N
39 !$acc loop independent
40     do j=1, N
41 !$acc loop seq
42       do k=1, N
43         c(j,i) = c(j,i) + a(k,i) * b(j,k)
44       enddo
45     enddo
46   enddo
47 !$acc end kernels

```

並列実行する

→ 並列実行しない

```
$ nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -o mm3a_f_acc matmul3a.f90
main:
```

```

36, Generating copyin(b(N,N)) [if not already present]
Generating copy(c(N,N)) [if not already present]
Generating copyin(a(N,N)) [if not already present]
38, Loop is parallelizable
40, Loop is parallelizable
42, Generating NVIDIA GPU code
38, !$acc loop gang, vector(128) collapse(2) ! blockIdx%x threadIdx%x
40, ! blockIdx%x threadIdx%x auto-collapsed
42, !$acc loop seq

```

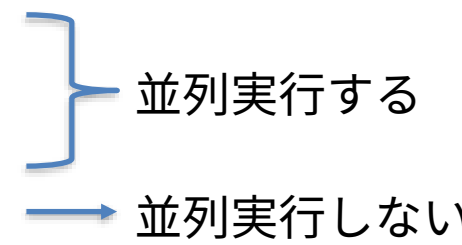
# loop指示文の追加（実行時間の比較）

## • C (matmul3a.c)

```

34 #pragma acc kernels copy(c[N*N]) copyin(a[N*N],b[N*N])
35 #pragma acc loop independent
36   for(i=0; i<N; i++){
37 #pragma acc loop independent
38   for(j=0; j<N; j++){
39 #pragma acc loop seq
40     for(k=0; k<N; k++){
41       c[i*N+j] += a[i*N+k] * b[k*N+j];
42     }
43   }
44 }

```

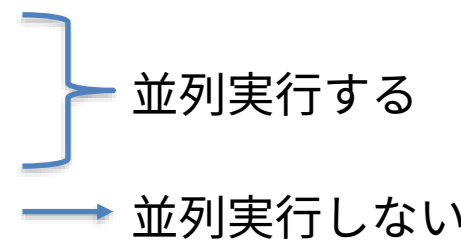


## • Fortran (matmul3a.f90)

```

36 !$acc kernels copy(c(N,N)) copyin(a(N,N),b(N,N))
37 !$acc loop independent
38   do i=1, N
39 !$acc loop independent
40     do j=1, N
41 !$acc loop seq
42       do k=1, N
43         c(j,i) = c(j,i) + a(k,i) * b(j,k)
44       enddo
45     enddo
46   enddo
47 !$acc end kernels

```



n=10で実行してみた → C版が劇的に高速化、Fortran版とほぼ同じ性能

time(us): 57

36: compute region reached 1 time

40: kernel launched 1 time

grid: [1] block: [128]

device time(us): total=6 max=6 min=6 avg=6

elapsed time(us): total=544 max=544 min=544 avg=544

time(us): 59

36: compute region reached 1 time

42: kernel launched 1 time

grid: [1] block: [128]

device time(us): total=6 max=6 min=6 avg=6

elapsed time(us): total=559 max=559 min=559 avg=559

- 一般的に、Fortranプログラムの方がコンパイラによる並列化判断が適切に働きやすい
- C/C++はポインタ参照の都合で不具合が起きないように保守的な判断がされやすい
- loop指示文で指定すればコンパイラの判断を上書きできる（妥当な性能を引き出せる）

# collapse版

## • C (matmul4a.c)

```

34 #pragma acc kernels copy(c[N*N]) copyin(a[N*N],b[N*N])
35 #pragma acc loop independent collapse(2)
36   for(i=0; i<N; i++){
37     for(j=0; j<N; j++){
38 #pragma acc loop seq
39       for(k=0; k<N; k++){
40         c[i*N+j] += a[i*N+k] * b[k*N+j];
41       }
42     }
43 }

```

- ループを融合してから並列化する
- 短いループがネストしている際に有用
- 実行時間が短くなるかはループの構造と長さ次第

```
$ nvc -Minfo=accel -acc -gpu=cc70 -tp=host -o mm4a_c_acc matmul4a.c
```

```
main:
```

```

36, Generating copyin(b[:N*N]) [if not already present]
   Generating copy(c[:N*N]) [if not already present]
   Generating copyin(a[:N*N]) [if not already present]
   Loop is parallelizable
37, Loop is parallelizable
39, Generating NVIDIA GPU code
   36, #pragma acc loop gang, vector(128) collapse(2)
       /* blockIdx.x threadIdx.x */
   37, /* blockIdx.x threadIdx.x collapsed */
   39, #pragma acc loop seq

```

## • Fortran (matmul4a.f90)

```

36 !$acc kernels copy(c(N,N)) copyin(a(N,N),b(N,N))
37 !$acc loop independent collapse(2)
38   do i=1, N
39     do j=1, N
40 !$acc loop seq
41       do k=1, N
42         c(j,i) = c(j,i) + a(k,i) * b(j,k)
43       enddo
44     enddo
45   enddo
46 !$acc end kernels

```

```
$ nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -o mm4a_f_acc matmul4a.f90
```

```
main:
```

```

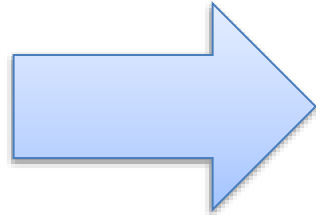
36, Generating copyin(b(N,N)) [if not already present]
   Generating copy(c(N,N)) [if not already present]
   Generating copyin(a(N,N)) [if not already present]
38, Loop is parallelizable
39, Loop is parallelizable
41, Generating NVIDIA GPU code
   38, !$acc loop gang, vector(128) collapse(2) ! blockIdx%x threadIdx%x
   39, ! blockIdx%x threadIdx%x collapsed
   41, !$acc loop seq

```



# collapseのイメージ

```
#pragma acc loop independent collapse(2)
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    for(k=0; k<N; k++){
      mat[i][j] = ...;
    }
  }
}
```



```
#pragma acc loop independent
for(ij=0; ij<N*N; ij++){
  i = ij/N; j = ij%N;
  for(k=0; k<N; k++){
    mat[i][j] = ...;
  }
}
```

- 短いループでも多数の計算コアを使い切れるようになる
- OpenMPのcollapseと同様の効果

# kernels loop

- kernelsの中にloopがひとつあるだけであれば、kernelsとloopをまとめて書くことも可能
  - (OpenMPでいうところのparallel for/parallel doと同様)

```
#pragma acc kernels
#pragma acc loop
  for(i=0; i<N; i++){
    v2[i] = v1[i] * 2.0;
  }
```

(loop1.c)

```
#pragma acc kernels loop
  for(i=0; i<N; i++){
    v2[i] = v1[i] * 2.0;
  }
```

```
!$acc kernels
!$acc loop
  do i=1, N
    v2(i) = v1(i) * 2.0d0
  enddo
!$acc end kernels
```

(loop1.f90)

```
!$acc kernels loop
  do i=1, N
    v2(i) = v1(i) * 2.0d0
  enddo
!$acc end kernels loop
```

end側は書かなくてもOK (対象が1つのloopであることが自明なため省略できる)

## OpenACCにおける変数や配列の共有・非共有の扱い

- これまでのコードは多重ループを含むが、そのループカウンタなどが競合する可能性については特に気にしていなかった
- OpenACCにおける変数や配列の扱い
  - スカラ変数はfirstprivateとなる
    - 並列化範囲外の値を引き継ぎ、互いに干渉しあわない
  - 配列はデバイスメモリにて共有される
    - 互いに干渉する、private指示節により挙動を変更することも可能
- 以上の仕様により、何も考えずに並列化指示文を入れても正しく動いてくれることも多いが、注意は必要である
- 参考：OpenMPの場合
  - private/shared指示節で指定
  - 何も指定しないとsharedとなりスレッド間で競合・干渉する
  - Cでは並列化範囲内で宣言される局所変数はprivate扱い
  - Fortranのみ、並列化範囲内の逐次ループのループカウンタはprivate扱い

## 並列実行形状の指定

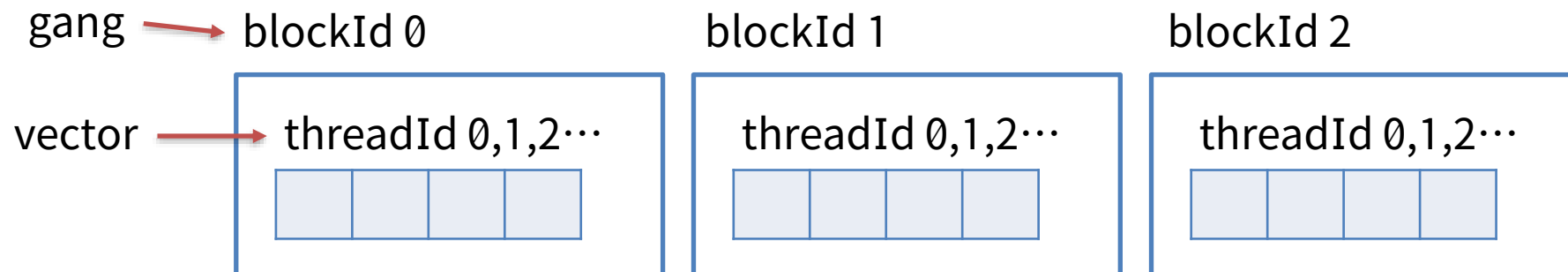
- ここまで、ループをどのようにGPU上の計算コアに割り当てるかは「おまかせ」だった
- 簡単なプログラムでは特に問題ないことが多いが、明示的に調整したい場合もある
  - ネストしたループ（多重ループ）はどのように計算コアに割り当たっている？
  - ハードウェアの特徴にあわせたループ構造にしたつもりだが、コンパイラはそれに合わせた実行をしてくれているのか？
- 実はコンパイル時のメッセージにどのように割り当てるかが出力されていた
  - gang, vector と blockIdx, threadIdx という概念が存在することが確認できる

```
nvc -Minfo=accel -acc -gpu=cc70 -tp=host -o v1_c_acc vector1.c
      16, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */

nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -o v1_f_acc vector1.f90
      13, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```
  - 数字を指定することもできるようだ
    - vector(32)

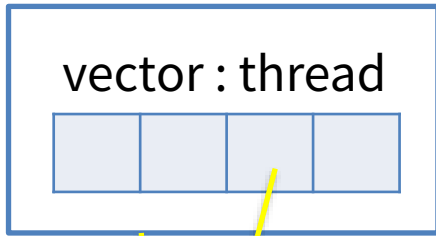
## gang, worker, vectorとblockIdx, threadIdx

- OpenACCではハードウェアに階層的な並列性があることを想定しており、上位階層から順にgang, worker, vectorとなっている
- これらをどのように組み合わせて実行するかを指定できる
- CUDAの並列実行モデルが階層的になっているため、それにあわせて言語設計された、と考えるのが適切
  - CUDAはgrid, threadblock, threadの三階層構造
    - 最近はThread Block Clustersという概念もある
  - OpenACCのおおまかな並列実行モデルの対応付け



# ハードウェアとのおおまかな対応付け

gang : block



- Streaming Multiprocessor(SM)内の並列性はvector、SM単位の並列性はgang
- おおまかには
  - 連続メモリアクセスする最内側のループはvector
  - より外側のループはgang
- くらいのイメージ
- HWの制約上実際には32コアなどの単位で動作していることを覚えておくと良い性能が出ることもある
- ()で数字を与えた場合はその数単位で割り当てられる
  - ループ長やプロセッサ数にあわせて調整する余地がある、程度に考えておこう

もう少し詳細に言えば……

- gangはHWレベルで同期できない単位の粗粒度並列性 (SM単位)
- workerはHWレベルで同期できる単位の細粒度並列性 (SM内のWARP群)
- vectorはworker内部でのSIMDやベクトル並列処理 (WARPの中)
- ※ 外側のループほど上位 (gang側) でなければならない : OpenACC 2.0以降で厳密化



## 演習：単純な行列積のコンパイル確認と性能確認

- 配布している行列積プログラムをコンパイルして実行する
  - コンパイルメッセージを見てデータの転送と並列化の状況を確認する
  - 問題サイズを大きくしてみて実行時間を確認する
- ソースコード

matmul1	3重ループ行列積
matmul1a	1の配列動的確保版
matmul2	copy指示節あり
matmul2a	2の配列動的確保版
matmul3	copy指示節+loop指示文あり
matmul3a	3の配列動的確保版
matmul4	copy指示節+loop指示文+collapse指示節あり
matmul4a	4の配列動的確保版

## 演習：行列積の繰り返し実行

- 行列積を単純に2回連続で行うと、その間で全データの転送が発生するはずである。data指示文で通信をなくしてみよう。（もちろん、independentなども指定したうえで。）

```
#pragma acc kernels copy(c[N*N]) copyin(a[N*N],b[N*N])
#pragma acc loop independent collapse(2)
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
#pragma acc loop seq
      for(k=0; k<N; k++){
        c[i*N+j] += a[i*N+k] * b[k*N+j];
      }
    }
  }
```

```
#pragma acc kernels copy(c[N*N]) copyin(a[N*N],b[N*N])
#pragma acc loop independent collapse(2)
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
#pragma acc loop seq
      for(k=0; k<N; k++){
        c[i*N+j] += a[i*N+k] * b[k*N+j];
      }
    }
  }
```

(matmul5.c, matmul5a.c)

```
!$acc kernels copy(c(N,N)) copyin(a(N,N),b(N,N))
!$acc loop independent collapse(2)
  do i=1, N
    do j=1, N
!$acc loop seq
      do k=1, N
        c(j,i) = c(j,i) + a(k,i) * b(j,k)
      enddo
    enddo
  enddo
!$acc end kernels
```

```
!$acc kernels copy(c(N,N)) copyin(a(N,N),b(N,N))
!$acc loop independent collapse(2)
  do i=1, N
    do j=1, N
!$acc loop seq
      do k=1, N
        c(j,i) = c(j,i) + a(k,i) * b(j,k)
      enddo
    enddo
  enddo
!$acc end kernels
```

(matmul5.f90, matmul5a.f90)

```
NVCOMPILER_ACC_TIME=1
や
NVCOMPILER_ACC_NOTIFY=2
を設定して通信の状況を確認してみよう
```

実際にやってみましょう

# 内容

- OpenACCを学ぶ前に
  - 並列計算の基礎
  - GPUとOpenACC
- 単純なベクトル計算を題材としてOpenACCの基本を学ぶ
  - コンパイルの仕方、実行の仕方
  - CPU-GPU間のデータ転送について
- 行列積を題材としてOpenACCの基本を学ぶ
  - 並列化ループの指定方法について
- その他、OpenACCに関する話題
  - 最適化のための一般的なヒント
  - デバッグとプロファイリング
  - 複数GPUの活用 など ※やや難しめ、初心者向けとは言い難い話題
- まとめ
- 参考（演習課題）：CG法のOpenACC化

## 最適化のためのヒント (1/2)

- GPUは**大量の計算コアをフル活用することで高い合計性能を得る**ハードウェアであることを忘れてはならない
  - GPUは多数の計算コアによる並列計算によって高性能を得ているため、並列化対象ループに十分な長さがあるようにする
    - `vector(threadIdx)`は32以上、`gang(blockIdx)`はSM数以上
    - 短いループはcollapseで結合させるなどする
    - 問題サイズを大きくしてみる
  - とにかく、単純な計算を大量に行うことを心がける
- CPU-GPU間のデータ通信は最小限にする
  - 転送回数も転送量も減らす
- 実はGPUカーネルの起動と終了にも少し時間がかかっている
  - 大規模なプログラムでは実行時間に影響する、まとめられる計算カーネルはまとめた方が良い

## 最適化のためのヒント (2/2)

- 連続メモリアクセスできるようにする
  - 最内ループでvectorによる連続メモリアクセスを行う
  - コアレスなメモリアクセスを狙う
  - (CPUプログラミングでも連続メモリアクセス自体は常識だが)

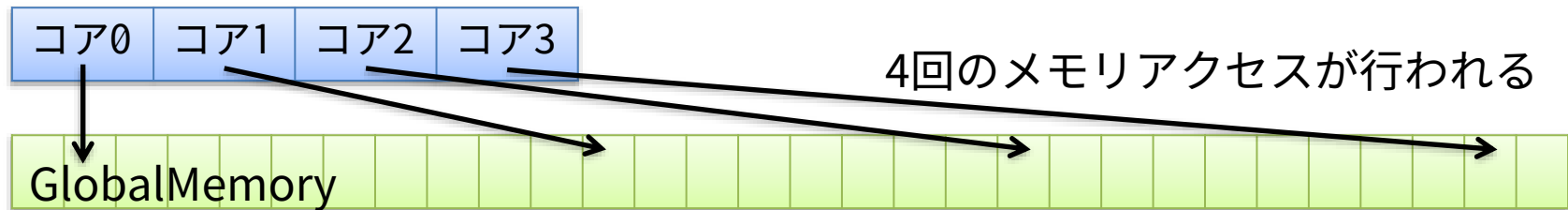
```
#pragma acc kernels
#pragma acc loop gang
for(i=0;i<N;i++){
#pragma acc loop vector
  for(j=0; j<N; j++){
    ○ a[i][j] = b[i][j] + c[i][j];
    × a[j][i] = b[j][i] + c[j][i];
  }
}
```

```
!$acc kernels
!$acc loop gang
do i=1, N
!$acc loop vector
  do j=1, N
    × a(i,j) = b(i,j) + c(i,j);
    ○ a(j,i) = b(j,i) + c(j,i);
  end do
end do
!$acc end kernels
```

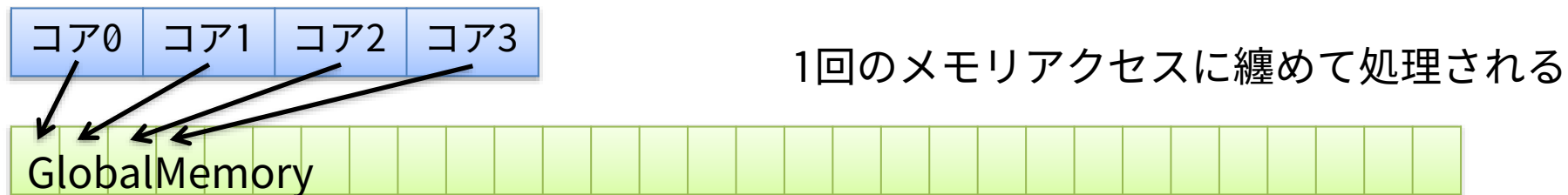
- 無理なものは無理：演算性能・メモリ転送性能を超える性能は出ない、CPUに勝てるとは限らない

## コアレスなメモリアクセス

- 同一SM内の複数CUDAコアによる同時メモリアクセスのアクセス先が近い場合にまとめてアクセスしてくれる機能
  - 詳細な条件はGPUの世代によって異なる、最新世代ほど条件が緩い
- アクセスがバラバラな（遠い）場合



- アクセスが揃っている（近い）場合



- 基本的にはvectorループが連続メモリアクセスになるようにすれば良い



# Array of Structure(AoS)とStructure of Array(SoA)

- 構造体の配列 (Array of Structure, AoS) と配列の構造体 (Structure of Array, SoA) を GPU で扱うことを考えてみる

構造体の配列 (AoS)

```
struct myStruct
{
    double a;
    double b;
    double c;
};
struct myStruct data[N];
```

// メンバ変数の値を2倍にする

```
#pragma acc kernels loop copy(data[N])
for(i=0; i<N; i++){
    data[i].a *= 2.0;
    data[i].b *= 2.0;
    data[i].c *= 2.0;
}
```

配列の構造体 (SoA)

```
struct myStruct
{
    double a[N];
    double b[N];
    double c[N];
};
struct myStruct data;
```

// メンバ変数の値を2倍にする

```
#pragma acc kernels loop copy(data)
for(i=0; i<N; i++){
    data.a[i] *= 2.0;
    data.b[i] *= 2.0;
    data.c[i] *= 2.0;
}
```

(aos.c および soa.c)

どちらの実装がGPUに  
適しているのだろうか？

# Array of Structure(AoS)とStructure of Array(SoA) : Fortran版

- 構造体の配列 (Array of Structure, AoS) と配列の構造体 (Structure of Array, SoA) を GPUで扱うことを考えてみる

構造体の配列 (AoS)

```

type MyType
  double precision :: a
  double precision :: b
  double precision :: c
end type MyType

type(myType) :: data(n)

// メンバ変数の値を2倍にする
!$acc kernels loop copy(data(n))
do i=1, n
  data(i)%a = data(i)%a * 2.0d0
  data(i)%b = data(i)%b * 2.0d0
  data(i)%c = data(i)%c * 2.0d0
end do

```

配列の構造体 (SoA)

```

type MyType
  double precision :: a(n)
  double precision :: b(n)
  double precision :: c(n)
end type MyType

type(myType) :: data

// メンバ変数の値を2倍にする
!$acc kernels loop copy(data)
do i=1, n
  data%a(i) = data%a(i) * 2.0d0
  data%b(i) = data%b(i) * 2.0d0
  data%c(i) = data%c(i) * 2.0d0
end do

```

(aos.f90 および soa.f90)

どちらの実装がGPUに適しているのだろうか？

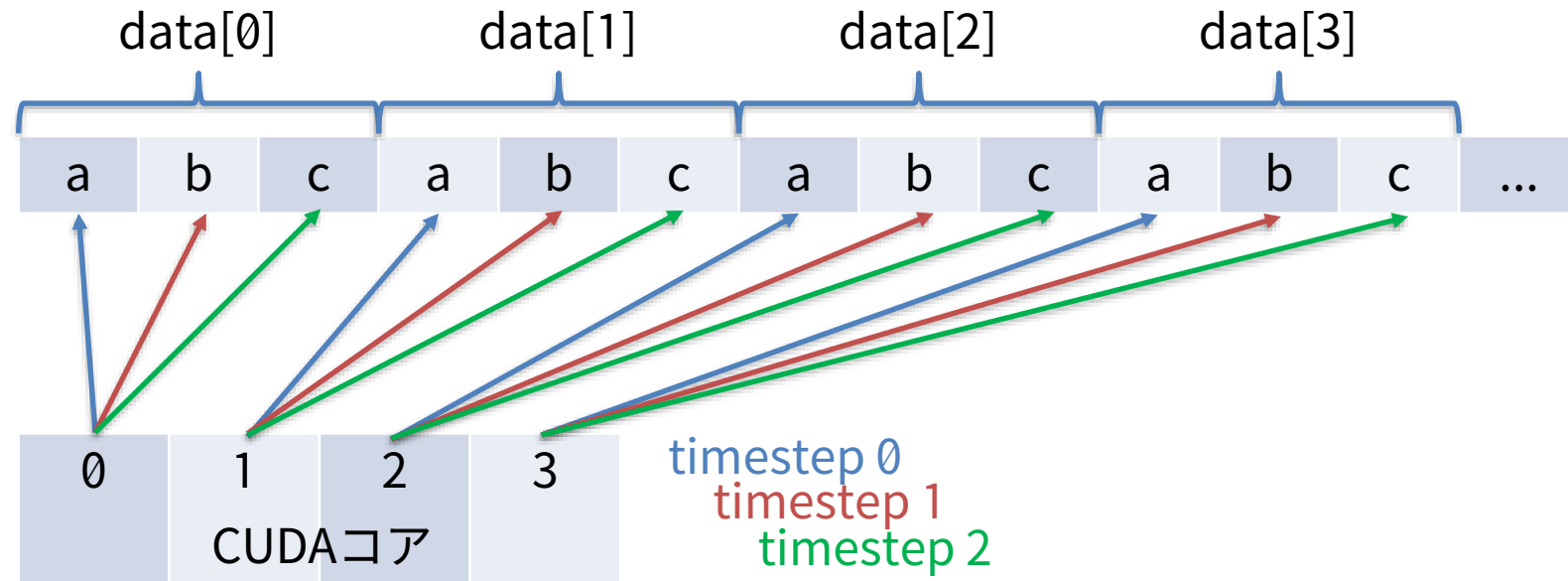
## 構造体の配列 (AoS) の挙動

- vector(4)による並列計算を考えてみる
  - (より長いvectorの場合も同様に考えれば良い)

```
struct myStruct
{
    double a;
    double b;
    double c;
};
struct myStruct data[N];
```

```
#pragma acc kernels loop
for(i=0; i<N; i++){
    data[i].a *= 2.0;
    data[i].b *= 2.0;
    data[i].c *= 2.0;
}
```

メモリ上のdata[N]の配置イメージ



CUDAコアへの計算割り当てのイメージ

- あるタイミングにおける連続した計算コアのアクセス先は不連続
- メモリアクセスがまとめて行われなため低速

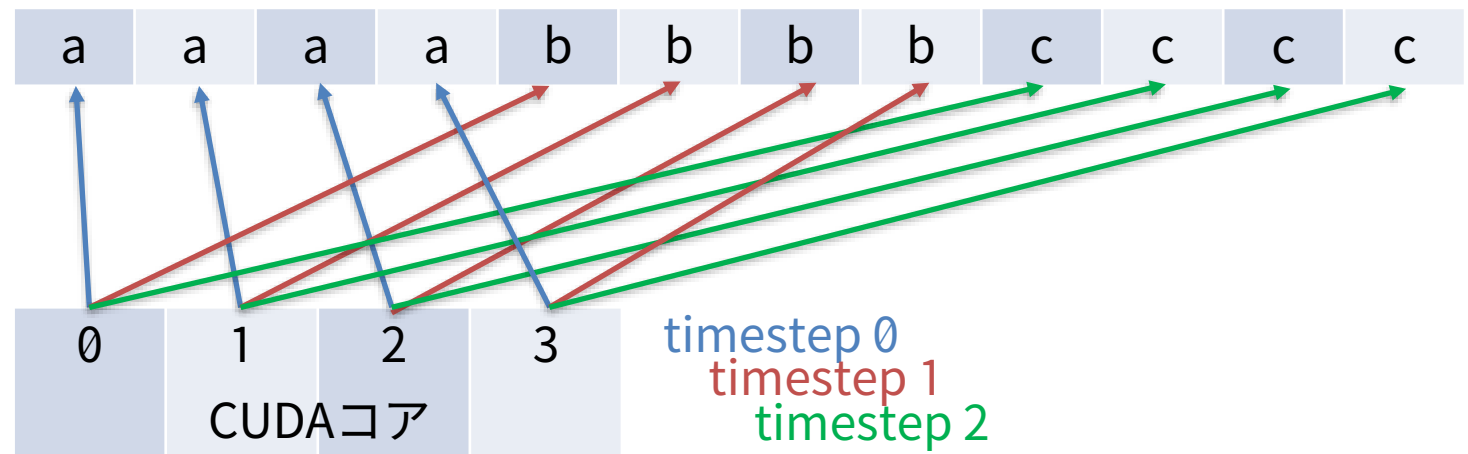
## 配列の構造体 (SoA) の挙動

- vector(4)による並列計算を考えてみる
  - (より長いvectorの場合も同様に考えれば良い)

```
struct myStruct
{
    double a[N];
    double b[N];
    double c[N];
};
struct myStruct data;
```

```
#pragma acc kernels loop
for(i=0; i<N; i++){
    data.a[i] *= 2.0;
    data.b[i] *= 2.0;
    data.c[i] *= 2.0;
}
```

メモリ上のdataの配置イメージ (N=4)



CUDAコアへの計算割り当てのイメージ

- あるタイミングにおける連続した計算コアのアクセス先は連続
- メモリアクセスがまとめて行われるため高速
- マルチコアCPUで並列計算を行う場合はアクセス競合の都合でAoSの方が高速になりやすい (CPUとGPUで適したコードが異なる)

## 動的な要素を持つ集合的な要素の転送には注意が必要

- 完全なDeep copyができない（動的な要素を持つ集合的な要素をまとめて転送できない）
  - C/C++：動的に確保された配列をメンバとして持つ構造体やクラスを一括コピーできない
  - Fortran：allocatable属性やpointer属性を持つメンバを含む派生型を一括コピーできない
  - 複雑なデータ構造を扱う場合に注意が必要
- 解決方法
  - 従来からの解決方法（手間はかかるが確実）
    - 必要な分だけ手動でcopyする
  - 最近の解決方法（簡単だが新しめの機能のため注意して使う必要あり）
    - コンパイル時の `-gpu` オプションに `deepcopy` を追加
      - Fortranのみ対応、完全なDeep copyができる（はず）
    - Unified memoryを使う
      - メインメモリとデバイスメモリを同一に扱う技術
      - コンパイル時の `-gpu` オプションに `managed` を追加、制限もあるため注意が必要
  - シンプルなデータ構造にコードを書き換える

## Deep copy問題 (1/2)

- 動的な要素を持つ集合的な要素を扱う際には注意が必要
  - C/C++ : 動的に確保された配列をメンバとして持つ構造体やクラスを一括コピーする場合
  - Fortran : allocatable属性やpointer属性を持つメンバを含む派生型を一括コピーする場合
  - ポインタ・アドレスのみがコピーされてその中身がコピーされず、中身を読めないために実行時エラー（エラーしないとしても計算結果があわない）という問題が生じる
  - メンバ長が固定であれば問題は起きない



```

struct Sobj{
  double values[10];
};
struct Sobj obj;

for(i=0; i<10; i++){
  obj.values[i] = (double)(i+1);
}

#pragma acc kernels
for(i=0; i<10; i++){
  obj.values[i] = obj.values[i] * 2.0;
}

```

(deepcopy1.c)



```

type :: Tobj
  double precision :: values(10)
end type Tobj
type(Tobj) :: obj

do i=1, n
  obj%values(i) = dble(i)
enddo

!$acc kernels
do i=1, n
  obj%values(i) = obj%values(i) * 2.0d0
enddo
!$acc end kernels

```

(deepcopy1.f90)



## Deep copy問題 (2/2)

- メンバに動的要素を持つコードの例

```

struct Sobj{
  double *values; ←コンパイル時にサイズが未定
};
struct Sobj obj; 実行時の引数からnを取得してmalloc
if(argc<2)n=10; else n=atoi(argv[1]);

obj.values = (double*)malloc(sizeof(double)*n);
for(i=0; i<n; i++){
  obj.values[i] = i+1;
}
                                     (deepcopy2.c)

#pragma acc kernels
for(i=0; i<n; i++){
  obj.values[i] = obj.values[i] * 2.0;
}

```

```

type :: Tobj
  double precision, allocatable :: values(:)
end type Tobj
type(Tobj) :: obj
                                     ↑コンパイル時にサイズが未定

```

! 実行時引数からnを得る処理は省略

```

allocate(obj%values(n))
do i=1, n
  obj%values(i) = dble(i)
enddo

!$acc kernels
do i=1, n
  obj%values(i) = obj%values(i) * 2.0d0
enddo
!$acc end kernels
                                     (deepcopy2.f90)

```

- Fortran版についてはdeepcopyオプションをつけると正しく転送してくれるようになる
- ……はずだったのだが、現在はオプションなしでも正しく動いてしまうようだ
  - 複雑なプログラムでもうまくいくかは不明

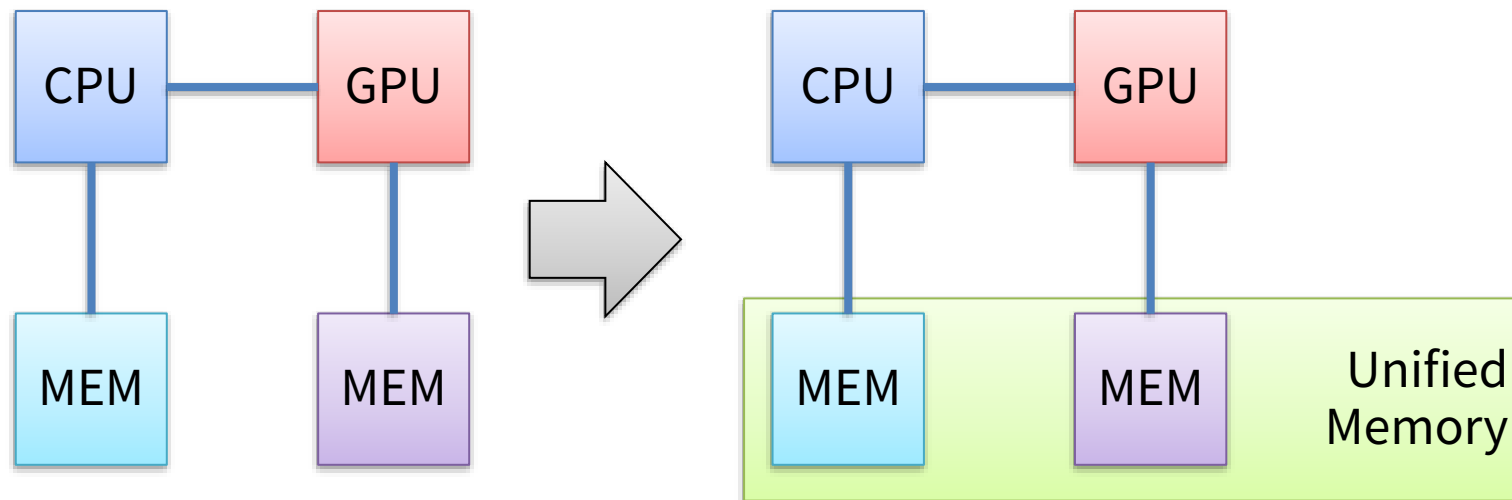
```

$ nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -o d2f_f_acc deepcopy2.f90
$ nvfortran -Minfo=accel -acc -gpu=cc70,deepcopy -tp=host -o d2f_f_acc_dc deepcopy2.f90

```

## Deep copy問題の解決方法：Unified Memoryの活用

- managedオプションを付けるとC/C++, Fortranいずれも正しく動作する
- Unified MemoryというCPU(ホスト)とGPUのメモリをまとめて1つに見せる仕組みを用いる
  - 便利だが転送効率は下がるため、Unified Memoryを使わず手動で実装した場合より長い時間がかかる可能性がある
  - 対象プログラムによってはあまり性能が落ちないこともあるため、プログラム移植の際には、ひとまずmanagedで実装してみてからデータ転送の最適化を行う、という手順は検討に値する



# 実行例

## • managedオプションを付けた場合

C

```
$ nvc -Minfo=accel -acc -gpu=cc70,managed -tp=host -o d2_c_acc_m deepcopy2.c
main:
  25, Loop is parallelizable
  Generating Tesla code
  25, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
  25, Generating implicit copy(obj.values[:n]) [if not already present]
  Generating implicit copyin(obj) [if not already present]
```

```
$ ./d2_c_acc_m
```

```
upload CUDA data file=/home/center/a49979a/share/20211004/deepcopy2.c function=main line=25 device=0 threadid=1 variable=obj bytes=8
before
  1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
after
  2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00
```

Fortran

```
$ nvfortran -Minfo=accel -acc -gpu=cc70,managed -tp=host -o d2f_acc_m deepcopy2.f90
main:
  32, Generating implicit copyin(obj) [if not already present]
  Generating implicit copy(obj%values(1:n)) [if not already present]
  33, Loop is parallelizable
  Generating Tesla code
  33, !$acc loop gang, vector(128) ! blockIdx%x threadIdx%x
```

```
$ ./d2_f_acc_m
```

```
upload CUDA data file=/home/center/a49979a/share/20211004/deepcopy2.f90 function=main line=32 device=0 threadid=1 variable=obj bytes=144
upload CUDA data file=/home/center/a49979a/share/20211004/deepcopy2.f90 function=main line=32 device=0 threadid=1 variable=.attach. bytes=8
upload CUDA data file=/home/center/a49979a/share/20211004/deepcopy2.f90 function=main line=32 device=0 threadid=1 variable=descriptor bytes=128
before
  1.00    2.00    3.00    4.00    5.00    6.00    7.00    8.00    9.00   10.00
after
  2.00    4.00    6.00    8.00   10.00   12.00   14.00   16.00   18.00   20.00
```

計算結果は正しいが、  
転送状況を見てもよく  
わからなくなった……

## Deep copy問題の解決方法：手動でフルコピーする

- 大規模なプログラムでは手間がかかるが、全て手動で対応することも可能
  - enter dataとupdateの組み合わせ（updateについては次頁以降で説明）

```
#pragma acc enter data copyin(obj) copyin(obj.values[n])
#pragma acc kernels
  for(i=0; i<n; i++){
    obj.values[i] = obj.values[i] * 2.0;
  }
#pragma acc update host(obj.values[n])
#pragma acc exit data delete(obj.values[n], obj)
```

(deepcopy3.c)

```
!$acc enter data copyin(obj) copyin(obj%values(n))
!$acc kernels
  do i=1, n
    obj%values(i) = obj%values(i) * 2.0d0
  enddo
!$acc end kernels
!$acc update host(obj%values)
!$acc exit data delete(obj%values, obj)
```

(deepcopy3.f90)

- data節で対応する場合はcreateとpresentを使うと良い
  - (createをcopyにするとNG)

```
#pragma acc data create(obj) copy(obj.values[n])
{
#pragma acc kernels present(obj.values[n])
  for(i=0; i<n; i++){
    obj.values[i] = obj.values[i] * 2.0;
  }
}
```

(deepcopy4.c)

```
!$acc data create(obj) copy(obj%values(n))
!$acc kernels present(obj%values(n))
  do i=1, n
    obj%values(i) = obj%values(i) * 2.0d0
  enddo
!$acc end kernels
!$acc end data
```

(deepcopy4.f90)

# OpenACC並列化部における関数の呼び出し

- kernels/parallel内部（OpenACC並列化対象内部＝GPU上）で関数を実行したい場合
  - 呼び出される関数にroutine指示文（acc routine）を書いておく必要がある
  - 例（スペースの都合上、変数の宣言などは省略）

```
#pragma acc routine
void calc(int n, double *v);

int main(int argc, char **argv)
{
  #pragma acc data copy(v1)
  {
    #pragma acc kernels
    {
      calc(n, v1);
    }
  }
}
```

どちらか片方でも良い  
(両方書いても問題はない)

```
#pragma acc routine
void calc(int n, double *v)
{
  int i;
  for(i=0; i<n; i++){
    v[i] = v[i] * 2.0;
  }
}
```

(routine0.c)

```
module mod
contains
subroutine calc(n, v)
!$acc routine
  do i=1, n
    v(i) = v(i) * 2.0d0
  enddo
end subroutine calc
end module mod
```

```
program main
  use mod
!$acc data copy(v1)
!$acc kernels
  call calc(n,v1)
!$acc end kernels
!$acc end data
end program main
```

(routine0.f90)

- Cでは関数名の直前に、Fortranでは関数名の直後に指示文を挿入
- **しかし、これらの書き方では呼び出された関数内のループは逐次実行になってしまう（calc関数内のループにloop指示文を加えても逐次実行になってしまう）**

# 関数内のループを並列実行する方法

- routineにgangなどの並列実行形状情報を追加する

```
void calc(int n, double *v);

int main(int argc, char **argv)
{
    #pragma acc data copy(v1)
    {
        #pragma acc kernels
        {
            calc(n, v1);
        }
    }
}
```

どちらに  
書いても  
良い

```
#pragma acc routine gang
void calc(int n, double *v)
{
    int i;
    #pragma acc loop gang vector
    for(i=0; i<n; i++){
        v[i] = v[i] * 2.0;
    }
}
```

(routine1.c)

```
#pragma acc routine gang
void calc(int n, double *v);

int main(int argc, char **argv)
{
    #pragma acc data copy(v1)
    {
        #pragma acc kernels
        {
            calc(n, v1);
        }
    }
}
```

```
void calc(int n, double *v)
{
    int i;
    #pragma acc loop gang vector
    for(i=0; i<n; i++){
        v[i] = v[i] * 2.0;
    }
}
```

(routine2.c)

関数内部の繰り返し部にloop指定がない場合はコンパイラ任せになるため、確実に並列化したい場合は指定すること

```
module mod
contains
subroutine calc(n,v)
!$acc routine gang
    integer :: n
    double precision :: v(*)
!$acc loop gang vector
    do i=1, n
        v(i) = v(i) * 2.0d0
    enddo
end subroutine calc
end module mod
```

```
program main
    use mod
    implicit none
    integer :: i, j, n=10
    double precision :: v1(10)
!$acc data copy(v1)
!$acc kernels
    call calc(n,v1)
!$acc end kernels
!$acc end data

end program main
```

(routine1.f90)

## 関数内を並列実行する方法：複数ソースファイルの場合（C）

- routine指示文を適切に使う必要がある

routine3a.c

```
#pragma acc routine gang
void calc(int n, double *v)
{
    int i;
    #pragma acc loop
    for(i=0; i<n; i++){
        v[i] = v[i] * 2.0;
    }
}
```

routine3b.c

```
#pragma acc routine gang
void calc(int n, double *v);

int main(int argc, char **argv)
{

    #pragma acc data copy(v1)
    {
        #pragma acc kernels
        {
            calc(n, v1);
        }
    }
}
```

- C言語版ではプロトタイプと実体の両方にroutine指示文が必要
- コンパイル・リンクは普通に行えばよい  
（それぞれのコードに-gpu=cc70等を付ける）
- nvc -Minfo=accel -acc -gpu=cc70 -tp=host -c routine3a.c
- nvc -Minfo=accel -acc -gpu=cc70 -tp=host -c routine3b.c
- nvc -Minfo=accel -acc -gpu=cc70 -tp=host  
-o r3\_c\_acc routine3a.o routine3b.o
- （最後はリンクするだけなので-Minfo=accelをつけても得られる情報はないはず、つけたままでも特に問題はない）



# 関数内を並列実行する方法：複数ソースファイルの場合（Fortran）

- routine指示文を適切に使う必要がある

routine3a.f90

```
module mod
contains
subroutine calc(n,v)
!$acc routine gang
integer :: n
double precision :: v(*)
!$acc loop
do i=1, n
v(i) = v(i) * 2.0d0
enddo
end subroutine calc
end module mod
```

- Fortran版ではただ分割するだけで良い
- コンパイルも普通に行える  
（それぞれのコードに-gpu=cc70等を付ける）

- nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -c routine3a.f90
- nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -c routine3b.f90
- nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host  
-o r3\_f\_acc routine3a.o routine3b.o

routine3b.f90

```
program main
use mod

!$acc data copy(v1)
!$acc kernels
call calc(n,v1)
!$acc end kernels
!$acc end data

end program main
```

## OpenACC API関数の利用

- OpenACCは指示文を用いて使うものであるが、その多くの機能は関数によっても提供されている
  - OpenMPのomp\_で始まる関数のようなものと思えば良い
  - 例
    - 利用可能なGPUの数を得る
      - acc\_get\_num\_devices
    - CPU-GPU間のデータコピー
      - acc\_copyin, acc\_copyout
  - C言語では#include “openacc.h”、Fortranではuse openaccして使う
  - 使いたいコンパイラの組み合わせ等によっては有益なこともあるかもしれない
    - OpenACCに対応していないコンパイラと組み合わせたい場合など？（未検証）

## 外部連携

- OpenACCは配列（メモリ）がホスト上にあるのかデバイス上にあるのかをあまり意識しなくて良い作りになっている
  - 例外：update指示文による更新
  - プログラム中で用いる配列はホストとデバイス両方を指すような概念になっている
- 一方で、明示的に一方のメモリを指したくなるときもあるため、そのための記述方法が提供されている
  - host\_data, use\_device, deviceptr
  - うまく使うことでCUDAやMPIとの連携が便利に行える
    - ホストメモリかデバイスメモリかを具体化することで、そのままcudaMemcpyやMPI通信関数の引数に与えられるようになる（少し後でOpenACC + MPIのより具体的な利用例を示す）

```
#pragma acc data copy(buf)
{
    .....
    #pragma acc host_data use_device(buf)
    MPI_Send(buf, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD)
```

# デバuggとプロファイラ

- デバugg、プロファイラ：基本的にCUDAと共通のツールが利用できる
  - HPC SDKに含まれるcuda-gdbやnvprof、nsysなどが利用できる
  - GUIが表示されるものはX転送が必要
    - MovaXtermは標準対応、Putty単体では対応不能、CygwinはX Window関係のパッケージが必要
    - sshコマンド実行に-Yオプションを付けておく必要がある
  - 必要に応じて、コンパイラオプション -gを付けたり、-taオプションにdebugやlineinfoを付ける（はずなのだが、付けなくても普通に情報を見られる気もする……）

```
$ nvc -Minfo=accel -g -acc -gpu=cc70,lineinfo -tp=host -o m2_c_acc_gl matmul2.c
```

※実際には-gとlineinfoは同時に指定できない、指定するとlineinfoが無視される

- デバugg
  - 単純なプログラムのOpenACC化でデバuggに困ることはあまりないと思われる
  - 意図した通りにGPU化やデータ転送ができていないか、コンパイラの出力に注意が必要
- プロファイラ
  - NVIDIA社が提供：nvprof, nvvp, Nsight
  - NVIDIA社以外が提供するツールもいくつかあるようだ

# nvprof の利用例

```
$ nvprof ./a1c_c_acc
==96336== NVPROF is profiling process 96336, command: ./a1_c_acc
 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
 2.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

```
==96336== Profiling application: ./a1_c_acc
```

```
==96336== Profiling result:
```

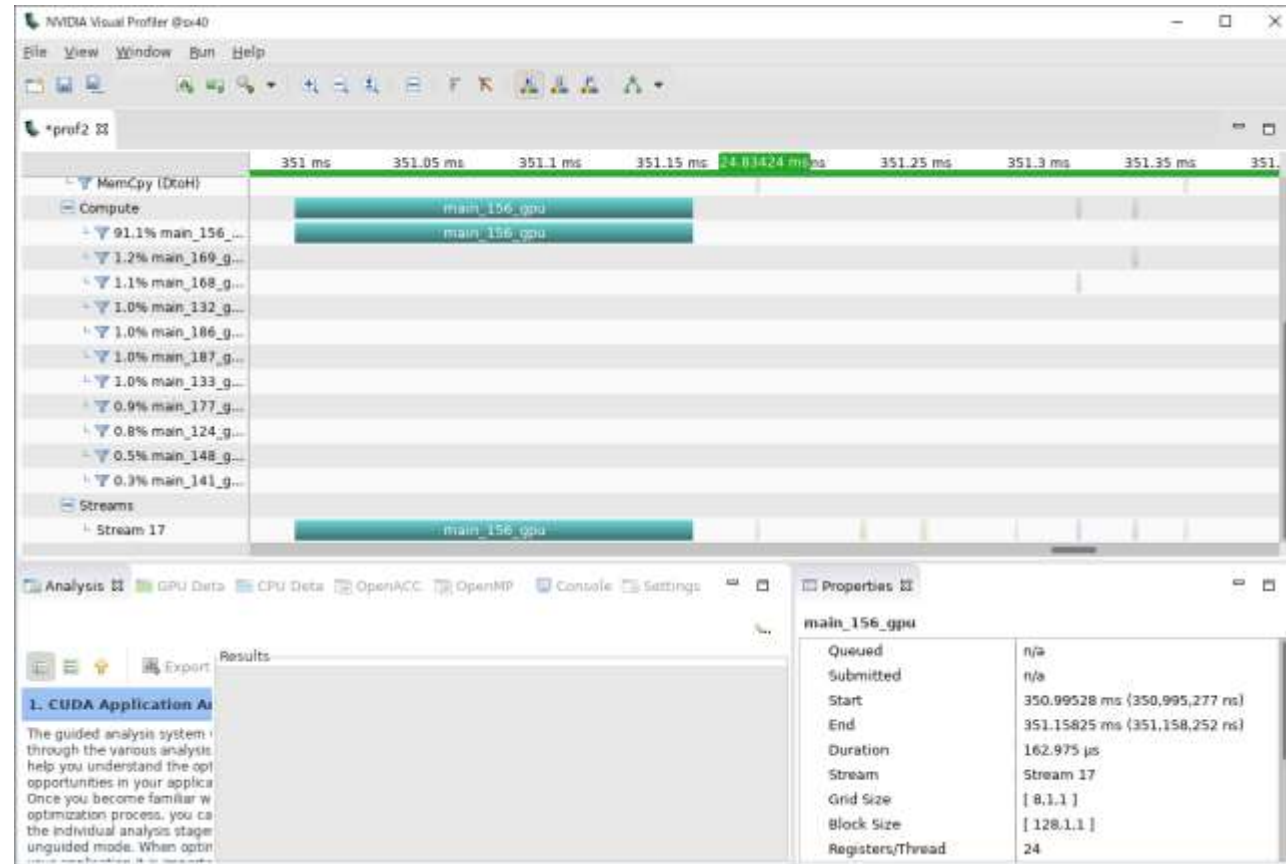
Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	42.33%	4.0640us	2	2.0320us	1.6000us	2.4640us	[CUDA memcpy HtoD]
	35.67%	3.4240us	1	3.4240us	3.4240us	3.4240us	main_21_gpu
	22.00%	2.1120us	1	2.1120us	2.1120us	2.1120us	[CUDA memcpy DtoH]
API calls:	80.70%	353.32ms	1	353.32ms	353.32ms	353.32ms	cuDevicePrimaryCtxRetain
	12.26%	53.667ms	1	53.667ms	53.667ms	53.667ms	cuDevicePrimaryCtxRelease
	5.14%	22.489ms	1	22.489ms	22.489ms	22.489ms	cuMemHostAlloc
	1.59%	6.9831ms	1	6.9831ms	6.9831ms	6.9831ms	cuMemFreeHost
	0.19%	817.59us	1	817.59us	817.59us	817.59us	cuMemAllocHost
	0.06%	245.00us	4	61.249us	4.2880us	223.16us	cuMemAlloc
～中略～							
OpenACC (excl):	98.08%	22.609ms	1	22.609ms	22.609ms	22.609ms	acc_enter_data@allocate1.c:20
	0.59%	135.03us	1	135.03us	135.03us	135.03us	acc_device_init@allocate1.c:20
	0.38%	87.861us	1	87.861us	87.861us	87.861us	acc_enqueue_launch@allocate1.c:21 (main_21_gpu)
	0.28%	64.660us	1	64.660us	64.660us	64.660us	acc_enqueue_upload@allocate1.c:20
	0.12%	28.616us	1	28.616us	28.616us	28.616us	acc_enqueue_download@allocate1.c:25
	0.11%	25.423us	1	25.423us	25.423us	25.423us	acc_enqueue_upload@allocate1.c:21
	0.10%	23.474us	1	23.474us	23.474us	23.474us	acc_exit_data@allocate1.c:21
	0.07%	17.253us	1	17.253us	17.253us	17.253us	acc_enter_data@allocate1.c:21

実はCUDAに関する関数が呼ばれており、OpenACCがCUDAの上で動いているような関係にあることを確認することができる

OpenACCコードの何行目にどれだけ時間がかかったかがわかる

# nvvp (NVIDIA Visual Profiler)

- 測定方法（バッチジョブ内で）：`nvprof -o filename ./a.out`
  - oで出力先ファイル名を指定、これをnvvpで読む
- 閲覧方法
  - nvvpコマンドで起動
  - メニューのfileからimportを選び、Select an import source:でNvprofを選びNext、Single processでNext、Timeline data file:で出力された結果ファイルを選びFinishを選択すれば結果を見ることができる
  - ログインノード上で実行可能  
（レスポンスがあまり良くないためローカルPCにインストールして実行した方が快適かもしれないが、現在HPC SDKはLinux向けしか存在しない）
- 画面例 →



# Nsight

- 新しいプロファイリングツール
  - Nsight Systems, Nsight Compute, Nsight Graphicsから構成される
  - CUIとGUIそれぞれで利用できる
- CUIによるプロファイル情報の取得と表示
  - `nsys profile ./a.out`
  - `--stats=true`オプションをつけておくとその場で統計情報も得られる
    - 統計情報の一部 →
  - デフォルトの出力ファイル名は `report1.qdstrm`
  - Nsight Systemsで閲覧するには`-o`で `qdrep`ファイルを指定しておく必要があるようだ

Generating CUDA API Statistics...  
CUDA API Statistics (nanoseconds)

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
77.1	23989788	1	23989788.0	23989788	23989788	cuMemHostAlloc
13.2	4123835	1	4123835.0	4123835	4123835	cuModuleLoadDataEx
3.8	1181563	10	118156.3	2924	1110527	cuMemAlloc_v2
2.9	898283	1	898283.0	898283	898283	cuMemAllocHost_v2
1.4	423594	20	21179.7	5039	52341	cuLaunchKernel
0.6	181128	28	6468.9	2476	20178	cuStreamSynchronize
0.3	94849	7	13549.9	4444	25066	cuMemcpyDtoHAsync_v2
0.3	82336	6	13722.7	3396	27629	cuMemsetD32Async
0.2	53309	7	7615.6	3497	29734	cuMemcpyHtoDAsync_v2
0.1	37604	7	5372.0	1915	13222	cuEventRecord
0.1	28981	1	28981.0	28981	28981	cuStreamCreate
0.1	22506	7	3215.1	1366	5176	cuEventSynchronize
0.0	6956	2	3478.0	1764	5192	cuEventCreate

Generating CUDA Kernel Statistics...

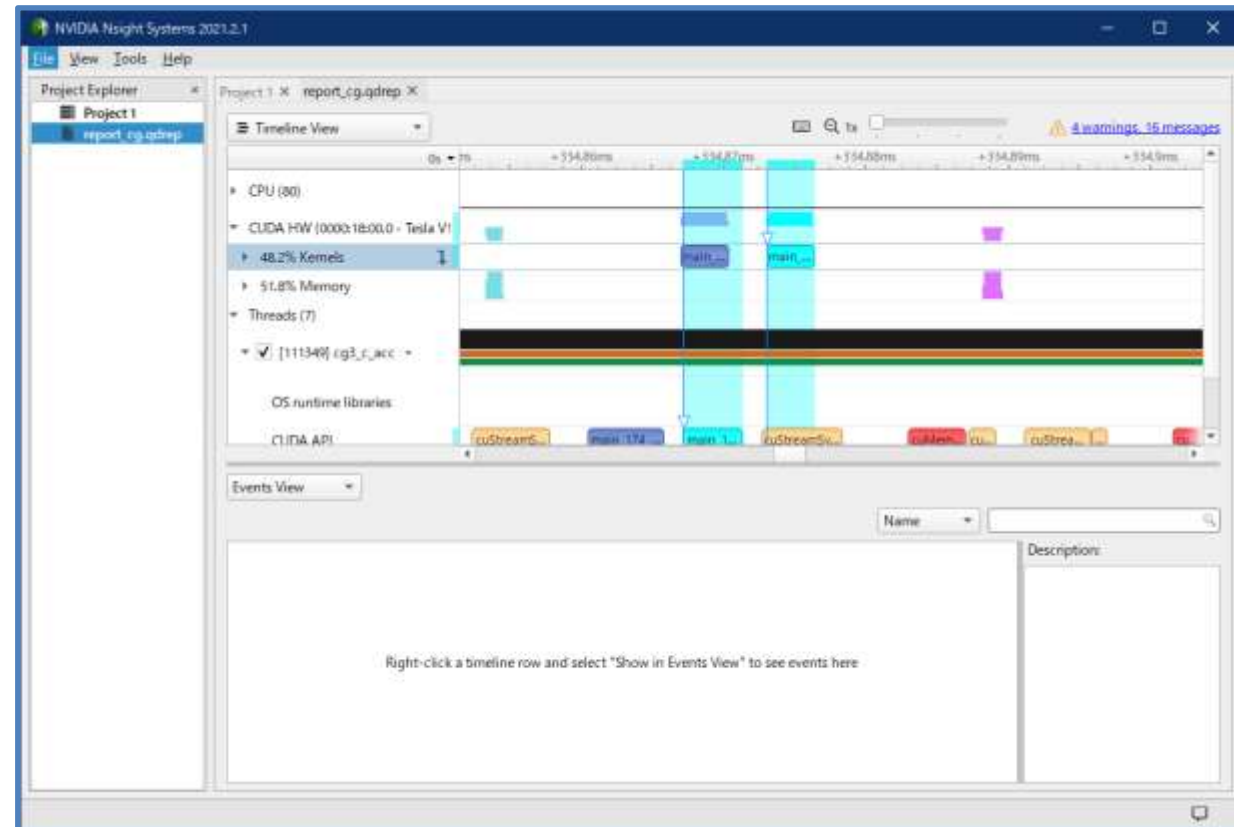
Generating CUDA Memory Operation Statistics...  
CUDA Kernel Statistics (nanoseconds)

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
12.2	4448	2	2224.0	1856	2592	main_159_gpu
11.2	4064	2	2032.0	1696	2368	main_132_gpu
10.8	3936	2	1968.0	1760	2176	main_132_gpu__red
10.3	3744	2	1872.0	1760	1984	main_171_gpu__red
10.2	3712	2	1856.0	1760	1952	main_191_gpu__red
9.8	3584	2	1792.0	1664	1920	main_171_gpu
9.5	3456	2	1728.0	1312	2144	main_123_gpu
9.4	3424	2	1712.0	1664	1760	main_191_gpu
9.1	3296	2	1648.0	1504	1792	main_181_gpu
3.8	1376	1	1376.0	1376	1376	main_142_gpu
3.8	1376	1	1376.0	1376	1376	main_150_gpu



## Nsight SystemsのGUIによる結果の表示

- nsysで出力したqdrepファイルを読み込んでGUI表示する
- WindowsやMacのクライアントがあるため、qdrepファイルをダウンロードして閲覧するのがおすすめ
- 例：`nsys profile -o report1.qdrep --stats=true ./a.out`
- Nsight Systemsを起動し  
File - Openでqdrepファイルを開いて閲覧
- カーネル内の性能調査にはさらにNsight Computeとの連携も必要
  - クライアントからログインノードに接続して使えると良いのだが、いまのところうまく動かせていない



## 複数GPUの活用

- 1ノードに複数のGPUを搭載した環境も増えている
  - 「不老」Type IIサブシステムも1ノードに4GPU
- 複数のGPUを使うことで、より高速・より大規模なプログラムの実行が可能になる可能性がある
- OpenACC内部で利用するGPUを切り替えるには`acc_set_device_num`関数を使う
- OpenACCプログラムが認識できるGPUを調整・制御するには環境変数`CUDA_VISIBLE_DEVICES`を使う
  - その名の通りCUDAでも有効（NVIDIA GPUを使ったGPUプログラム全般で有効な可能性が高い）
- GPUスパコンの構成（たとえば各ノードに搭載されたCPUとGPUの数）は色々考えられるが、1MPIプロセスが1GPUを担当するような実行形態を考えておくと汎用性がある使いやすい

# ノード内複数GPUの活用：OpenACCだけを利用する例

- 4つの配列を用意し、それぞれ異なるGPUに処理させることを考える

```
#include <stdio.h>
#include <openacc.h>
int main(int argc, char **argv){
    int i, j;
    int n=10;
    double v1[4][10], v2[4][10];
    for(j=0; j<4; j++){
        for(i=0; i<n; i++){
            v1[j][i] = (double)(j+1)+(double)(i+1)/100.0;
            v2[j][i] = 0.0;
        }
    }
    for(j=0; j<4; j++){
        acc_set_device_num(j, acc_device_nvidia);
#pragma acc kernels
        for(i=0; i<n; i++){
            v2[j][i] = v1[j][i] * 2.0;
        }
    }
    for(j=0; j<4; j++){
        printf("GPU %d ¥n", j);
        for(i=0; i<n; i++)printf(" %8.3f", v1[j][i]);
        printf(" ¥n");
        for(i=0; i<n; i++)printf(" %8.3f", v2[j][i]);
        printf(" ¥n");
    }
    return 0;
}
```

計算対象の配列を二次元化

GPUカーネルの直前で対象GPUを切り替え

multigpu1.c

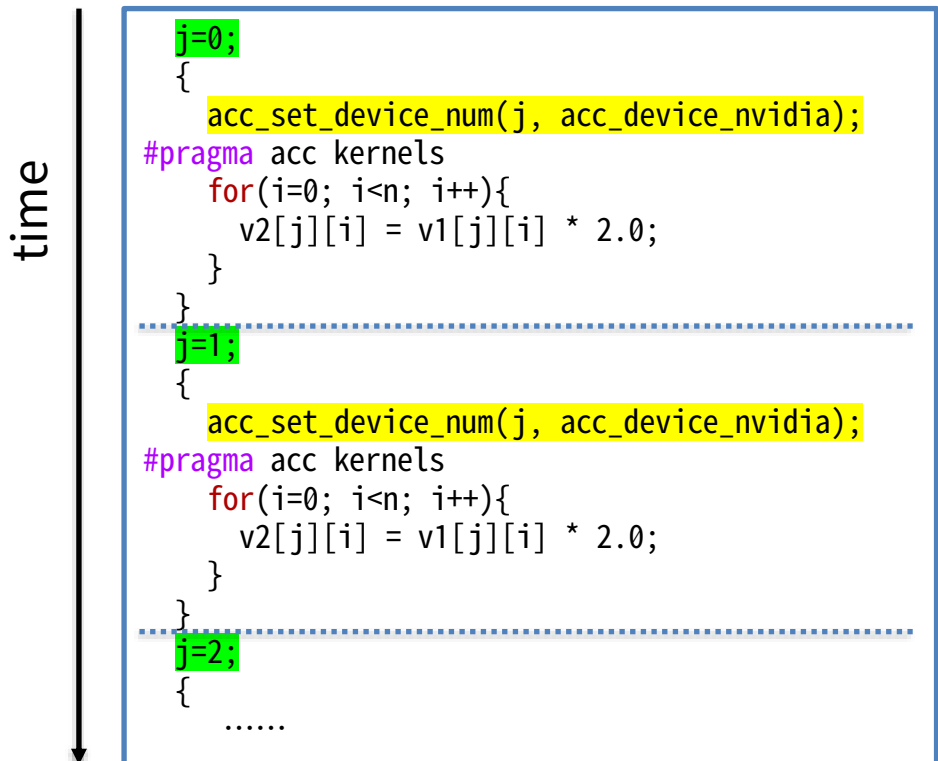
```
program main
    use openacc
    integer :: i, j, n=10
    double precision :: v1(10,4), v2(10,4)
    do j=1, 4
        do i=1, n
            v1(i,j) = dble(j) + dble(i)/100.0
            v2(i,j) = 0.0d0
        enddo
    enddo
    do j=1, 4
        call acc_set_device_num(j, acc_device_nvidia)
!$acc kernels
        do i=1, n
            v2(i,j) = v1(i,j) * 2.0d0
        enddo
!$acc end kernels
    enddo
    do j=1, 4
        write(*,*)"GPU", j
        do i=1, n
            write(*, '(1H F8.3)', advance="NO")v1(i,j)
        enddo
        write(*,*)""
        do i=1, n
            write(*, '(1H F8.3)', advance="NO")v2(i,j)
        enddo
        write(*,*)""
    enddo
end program main
```

➤ NVCOMPILER\_ACC\_TIME=1を設定して実行するとGPU別に情報が表示されるため、複数GPUが使えていることが簡単に確認できる

multigpu1.f90

# ノード内複数GPUの活用：OpenACC + OpenMPの例

- 前頁の例では4つのGPUに対する処理が逐次的になってしまう
- 各GPUを並列に動作させたい場合はOpenMPやMPIと組み合わせるのが良い
- 動作イメージ



multigpu2.c

```

#pragma omp parallel for private(i)
  for(j=0; j<4; j++){
    acc_set_device_num(j, acc_device_nvidia);
#pragma acc kernels
  for(i=0; i<n; i++){
    v2[j][i] = v1[j][i] * 2.0;
  }
}

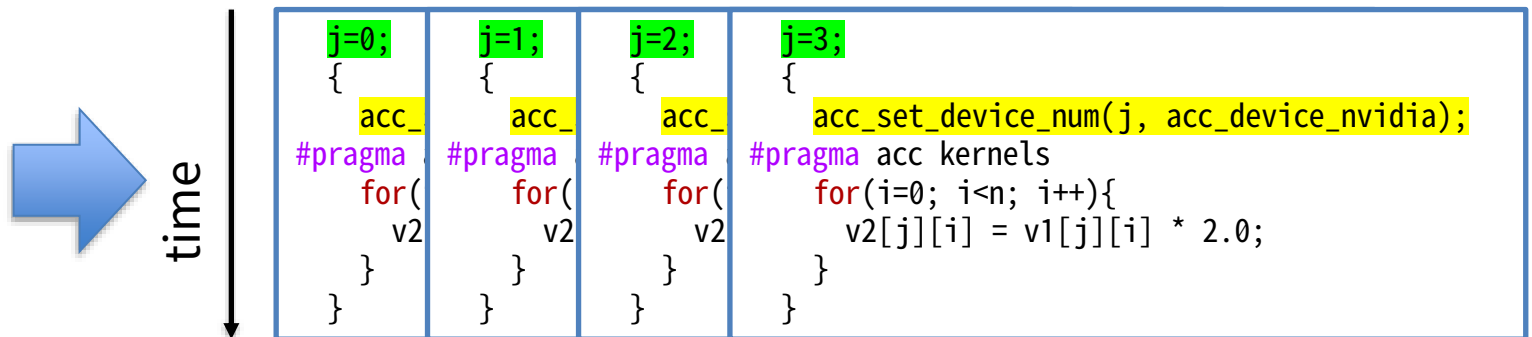
```

multigpu2.f90

```

!$omp parallel do
  do j=1, 4
    call acc_set_device_num(j, acc_device_nvidia)
!$acc kernels
    do i=1, n
      v2(i,j) = v1(i,j) * 2.0d0
    enddo
!$acc end kernels
  enddo

```



## ノード内複数GPUの活用：OpenACC + MPIの例（実行方法の例）

- MPIプログラム内で利用するGPUを指定するには環境変数CUDA\_VISIBLE\_DEVICESを使う
  - CUDA\_VISIBLE\_DEVICESはプロセスから見えるGPUとその順序を定める環境変数
  - acc\_set\_device\_num関数でも指定が可能、組み合わせて利用できるが混乱しないように注意
  - 使用例1

job1.sh

```
export CUDA_VISIBLE_DEVICES=0,1
./a.out
```

この実行方法ではa.outの中では2GPUのみ確認できるようになる。ちなみにCUDA\_VISIBLE\_DEVICES=3,2と指定するとGPU3とGPU2が順番に見えることになるが、NVCOMPILER\_ACC\_TIMEの出力上のdevicenumは0と1になるので注意が必要。

- 使用例2

job2.sh

```
mpirun -n 4 ./run.sh
```

1ノード内の4GPUを利用する例。OpenMPIを使う際は環境変数OMPI\_COMM\_WORLD\_RANKにランク番号がセットされるので、これをCUDA\_VISIBLE\_DEVICESに与えて各プロセスが異なるGPUを叩くように仕向けている。各プロセスには個別の1GPUのみが見える状態になるため、プログラム内でacc\_set\_device\_numを指定する必要はない。

run.sh

```
#!/bin/bash
export CUDA_VISIBLE_DEVICES=${OMPI_COMM_WORLD_RANK}
./a.out
```

# ノード内複数GPUの活用：OpenACC + MPIの例（プログラムの例1）

## 元となるMPIプログラムの概形

```
// 配列rbufとsbufは0.0で初期化されているものとする

// プロセスごとに送信用データを生成
for(i=0; i<10; i++)sbuf[i] = (double)rank + (double)(i+1)/100.0;

// ここでsbufの値をprintf出力（コードは省略）

// 各プロセスのsbufをMPI_Reduceでプロセス0のrbufに集約（加算）
MPI_Reduce(sbuf, rbuf, 10, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

// ここでrbufの値をprintf出力（コードは省略）
```

プロセス0のrbufに全プロセスが生成したsbufの総和が格納されているため、**最後のprintでは正しく総和が表示される。**

参考：multigpu\_mpi0.c

## OpenACC+MPIプログラムの概形1

```
// 配列rbufとsbufは0.0で初期化されているものとする

// GPUへ配列をコピー
#pragma acc data copy(sbuf[10], rbuf[10])
{
#pragma acc kernels
{
// 各プロセスはGPU上でsbufを生成
#pragma acc loop
    for(i=0; i<10; i++)sbuf[i] = (double)rank + (double)(i+1)/100.0;
}
// MPI_Reduceで値を集約
MPI_Reduce(sbuf, rbuf, 10, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}

// ここでrbufの値をprintf出力（コードは省略）
```

MPI\_Reduceを実行する時点でメインメモリ上のsbufには生成した値が入っていない（GPU上のsbufにのみ生成した値が入っている）ため、MPI\_Reduceで総和を求めても0.0であり、**最後のprintfでは0.0が表示される。**

参考：multigpu\_mpi1.c



# ノード内複数GPUの活用：OpenACC + MPIの例（プログラムの例2）

## OpenACC+MPIプログラムの概形2

```
// 配列rbufとsbufは0.0で初期化されているものとする

// GPUへ配列をコピー
#pragma acc data copy(sbuf[10], rbuf[10])
{
#pragma acc kernels
{
// 各プロセスはGPU上でsbufを生成
#pragma acc loop
    for(i=0; i<10; i++)sbuf[i] = (double)rank + (double)(i+1)/100.0;
}
#pragma acc update host(sbuf[10])
// MPI_Reduceで値を集約
MPI_Reduce(sbuf, rbuf, 10, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
#pragma acc update device(rbuf[10])
}

// ここでrbufの値をprintf出力（コードは省略）
```

MPI\_Reduceを実行する前にsbufをupdateでホストに持ってくれば、最後のprintでは正しく総和が表示される。（rbufをupdate deviceしておかないとdata copyによりGPU上のrbufが送られてきて破壊されてしまう点に注意。）

参考：multigpu\_mpi2.c  
MPI通信の度にホストメモリに書き戻すのは手間、性能的にも悪影響。

## OpenACC+MPIプログラムの概形3

```
// 配列rbufとsbufは0.0で初期化されているものとする

// GPUへ配列をコピー
#pragma acc data copy(sbuf[10], rbuf[10])
{
#pragma acc kernels
{
// 各プロセスはGPU上でsbufを生成
#pragma acc loop
    for(i=0; i<10; i++)sbuf[i] = (double)rank + (double)(i+1)/100.0;
}
#pragma acc host_data use_device(sbuf,rbuf)
// MPI_Reduceで値を集約
MPI_Reduce(sbuf, rbuf, 10, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}

// ここでrbufの値をprintf出力（コードは省略）
```

host\_data, use\_deviceを使えば、MPI\_Reduceを実行する際にメインメモリ上の配列ではなくGPU上の配列を使ってMPI通信が行われる（CUDA-Aware MPI）ため、最後のprintfでは正しく総和が表示される。

性能も向上することが期待される  
(が、向上するとは限らず、調査中……)

参考：multigpu\_mpi3.c  
同様にNCCLにも応用可能→参考：multigpu\_mpi4.c



## 複数ノード複数GPUの活用：OpenACC + MPI

- プログラムとしては「OpenACC+MPIプログラムの概形3」の例のままでよい
- ジョブスクリプトに多少の工夫が必要
- 2ノード×4GPU（合計8MPIプロセス）の例

job.sh

```
#!/bin/bash -x
#PJM -L rscgrp=cx-workshop
#PJM -L node=2
#PJM -L elapse=00:5:00
#PJM -j
#PJM -S
```

module load hpc\_sdk

mpirun -n 8 -machinefile \$PJM\_O\_NODEINF -map-by ppr:2:socket ./run.sh ./a.out

複数ノードにプロセスを適切に配置するためのオプション

（詳しくはWebの「スーパーコンピュータ「不老」基本マニュアルおよび関連資料」に掲載している「Type IIサブシステム向けのプロセス・スレッド配置方法」を参照）

run.sh（chmodで実行権を与えておくこと）

```
#!/bin/bash
export CUDA_VISIBLE_DEVICES=${OMPI_COMM_WORLD_LOCAL_RANK}
numactl -l $1
```

## 複数ノード実行に関するソースコードとジョブスクリプト例一覧

- `job_multigpu_mpi1.sh`, `job_multigpu_mpi1_a.sh`
  - 1ノード4GPUの実行例 (`multigpu_mpi0.c`から`multigpu_mpi4.c`まで比較)
- `job_multigpu_mpi2.sh`, `job_multigpu_mpi2_a.sh`
  - `multigpu_mpi2`の2ノード×4GPU版
- `job_multigpu_mpi3.sh`, `job_multigpu_mpi3_a.sh`
  - `multigpu_mpi3`の2ノード×4GPU版
- `job_multigpu_mpi4.sh`, `job_multigpu_mpi4_a.sh`
  - `multigpu_mpi4`の2ノード×4GPU版
- 各ソースコードの冒頭にコンパイルの例も記載
  - 動作確認はしましたが、最低限度の簡単なプログラムのみで、性能の良し悪しは見ていません。十分に高い性能を得るためにはシステム側の設定や環境変数の追加が必要かも知れません。新たな情報はセンターのWebにて紹介していきます。

## その他の指示文について

- ここまでに扱ってきた指示文だけでも多くのプログラムをOpenACCによりGPU化できる
- まだ扱っていない指示文についても少しだけ紹介しておく

### – async, wait

- 通信や計算が終了するのを待たずに次の処理を行うasyncと、その終了を待つためのwait
- うまく使えば様々な待ち時間を削減し全体の実行時間を短縮させることができる

- 例：
 

```
for(i=...){
    #pragma acc update device(array) async(i)
    #pragma acc parallel loop async(i)
    for(...){...}
    #pragma acc update host(array) async(i)
}
```

  - ループカウンタ*i*ごとに更新と計算を組にして処理を発行
  - 終了を待たずに次の*i*の発行を行い、最後に待ち合わせる

### – tile #pragma acc wait

- 二次元タイル分割を行うための指示節
- キャッシュ利用効率が向上することで性能向上することがある
- 指定したサイズが適していないと性能低下することがある
- 例：#pragma acc kernels loop tile(64,4)

# 内容

- OpenACCを学ぶ前に
  - 並列計算の基礎
  - GPUとOpenACC
- 単純なベクトル計算を題材としてOpenACCの基本を学ぶ
  - コンパイルの仕方、実行の仕方
  - CPU-GPU間のデータ転送について
- 行列積を題材としてOpenACCの基本を学ぶ
  - 並列化ループの指定方法について
- その他、OpenACCに関する話題
  - 最適化のための一般的なヒント
  - デバッグとプロファイリング
  - 複数GPUの活用 など
- **まとめ**
- 参考（演習課題）：CG法のOpenACC化

## まとめ

- OpenACCの仕様や使い方について紹介した
  - わずか数行の指示文でGPUが利用できる
    - 触れなかった機能・指示文もいくつかあります：tileやasyncなど
  - OpenMPと似ているため、両方使えると色々と活用できる
  - 並列計算環境（プログラミング言語など）を取り巻く状況は流動的であり、今後何十年もOpenACCが生き残るかは誰にもわからないが、OpenACCは簡単に高い性能が得られることが多いため、是非活用してみたい
    - OpenMP, oneAPI, SYCL, DPC++などによるGPU利用が進んでくるとどうなるかわからない状況
- 残りの時間は演習時間とします
  - 演習問題としてCG法の逐次コードを提供しているので、指示文を挿入して並列化してみましょう
  - その他、任意のプログラムのGPU化やプロファイラのテストなどをしてもらって構いません

# 内容

- OpenACCを学ぶ前に
  - 並列計算の基礎
  - GPUとOpenACC
- 単純なベクトル計算を題材としてOpenACCの基本を学ぶ
  - コンパイルの仕方、実行の仕方
  - CPU-GPU間のデータ転送について
- 行列積を題材としてOpenACCの基本を学ぶ
  - 並列化ループの指定方法について
- その他、OpenACCに関する話題
  - 最適化のための一般的なヒント
  - デバッグとプロファイリング
  - 複数GPUの活用 など
- まとめ
- **参考（演習課題）：CG法のOpenACC化**

# CG法プログラムのOpenACC化

- 単純なCG法の計算カーネル（反復計算部）を題材に、プログラムのOpenACC化を考える
  - 簡単にするため、行列は密行列、前処理は対角スケーリング
- CG法（共役勾配法、Conjugate Gradient Method）
  - 対称正定値行列を係数とする連立一次方程式 $Ax=b$ を解く手法
    - 行列A、既知のベクトルb、未知のベクトルx
  - 基本アルゴリズム
    - Wikipediaから引用、前処理なし
    - 利用するコードは計算順序が変更されている版
    - 単純な行列Aとランダム行列xxからbを求めておき $Ax=b$ を解いてxとxxが（ほぼ）一致することを確認する、という構造にしてある（すぐに収束してしまうため、固定回数ループで時間を測定すると良い）
    - 時間測定を簡単に書くためOpenMP関数を利用
      - コンパイル時に-mpオプションを加える必要あり

```


$$r_0 = b - Ax_0$$


$$p_0 = r_0$$


for (k = 0; ; k++)
  
$$\alpha_k = \frac{r_k^T p_k}{p_k^T A p_k}$$

  
$$x_{k+1} = x_k + \alpha_k p_k$$

  
$$r_{k+1} = r_k - \alpha_k A p_k$$


  if  $r_{k+1}$  が十分に小さい then
    break

  
$$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$$

  
$$p_{k+1} = r_{k+1} + \beta_k p_k$$

結果は  $x_{k+1}$ 

```



## 今回用いるCG法コードの計算手順

```

Compute  $r^{(0)} = b - [A]x^{(0)}$ 
for  $i = 1, 2, \dots$ 
  solve  $[M]z^{(i-1)} = r^{(i-1)}$ 
   $\rho_{i-1} = r^{(i-1)} \cdot z^{(i-1)}$ 
  if  $i=1$ 
     $p^{(1)} = z^{(0)}$ 
  else
     $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ 
     $p^{(i)} = z^{(i-1)} + \beta_{i-1} z^{(i)}$ 
  endif
   $q^{(i)} = [A]p^{(i)}$ 
   $\alpha_i = \rho_{i-1} / p^{(i)} \cdot q^{(i)}$ 
   $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$ 
   $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$ 
  check convergence  $|r|$ 
end

```

- 初期値、 $x(0)$ は適当な値（今回は0ベクトル）
- 収束するまで繰り返す
- 前処理、今回は $r$ を対角要素で割るだけ
- リダクション
- コピー
- リダクション
- ベクトル積和
- 行列ベクトル積＝一番時間がかかる処理
- リダクション
- ベクトル積和
- ベクトル積和
- 収束判定（中身はリダクションと平方根）

※上付き文字は反復回数に対応

- $Ax=b$ ： $A$ は行列、 $x$ と $b$ はベクトル
- $z, r, p, q$ はベクトル
- $\alpha \cdot \beta \cdot \rho$ はスカラー（ベクトルのリダクション結果）

# 主要コード

- 行列やベクトルに対する単純な計算ばかりで構成されているため、並列化・OpenACC化は容易
  - ★：行列要素同士のコピーや四則演算
  - ★：集約演算 (dot product, reduction)
  - これらを全てGPUカーネルにしていれば良さそうである
  - ※複雑な前処理を適用する場合は難易度が上がる
  - 具体的にはどのような手順でOpenACC化すれば良いだろうか？
    - ※ すぐに収束してしまうため、ここに妨害用のコードを追加してある
  - 各説明に対応するコードを提供しているが、まずはcg1.cまたはcg1.f90だけを見て並列高速化を目指してみよう

```

for(iter=1; iter<=maxiter; iter++){
  printf("iter %d ", iter);
  // {z} = [Minv]{r}
  for(i=0; i<N; i++){ z[i] = dd[i]*r[i]; } ★
  // {rho} = {r}{z} ※対角要素分の1だけのベクトルddを用意済
  rho = 0.0;
  for(i=0; i<N; i++){ rho += r[i]*z[i]; } ★
  // {p} = {z} if iter=1
  // beta = rho/rho1 otherwise
  if(iter==1){
    for(i=0; i<N; i++){ p[i] = z[i]; } ★
  }else{
    beta = rho/rho1;
    for(i=0; i<N; i++){ p[i] = z[i] + beta*p[i]; } ★
  }
  // {q} = [A]{p}
  for(i=0; i<N; i++){
    q[i] = 0.0;
    for(j=0; j<N; j++){
      q[i] += A[i*N+j]*p[j];
    }
  }
  // alpha = rho / {p}{q}
  pq = 0.0;
  for(i=0; i<N; i++){ pq += p[i]*q[i]; } ★
  alpha = rho / pq;
  // {x} = {x} + alpha*{p}
  // {r} = {r} - alpha*{q}
  for(i=0; i<N; i++){
    x[i] += + alpha*p[i]; ★
    r[i] += - alpha*q[i];
  }
  // check converged
  dnrn = 0.0;
  for(i=0; i<N; i++){ dnrn += r[i]*r[i]; } ★
  resid = sqrt(dnrn/bnrn);
  if(resid <= cond){break;}
  if(iter == maxiter){break;}
  rho1 = rho;
}

```

# 初期データ

行列A (サイズN×N)



ベクトルx (サイズN)

全て0.0

ベクトルxx (サイズN)

ランダム

ベクトルb (サイズN)

- A, x, xxを初期化し、 $b = A \times xx$ を計算しておいたうえで、cg法で $Ax = b$ を計算しxを求めるというプログラムにしてある。
- これなら前処理のない単純なCG法でも簡単に収束する。
- むしろ簡単過ぎてすぐに収束して終了してしまうため、意図的に値を破壊して収束しないようにしてある。(行列サイズN\*10回の反復を行い、収束しないまま終わる。)
- 正しく実装できているかを確認したい場合は妨害コードをコメントアウトすると良い。

# 実行例

- 第一引数は問題サイズN
- 第二引数を0にすると初期データや計算結果を出力しなくなる
- 第三引数に0以外を指定すると収束しても計算を続行する  
(すぐに終わってしまうのを防ぐもう1つの策)

```
$ ./cg5_c_acc
N=10, check=1, nocond=0
```

```
A:
9.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000
```

```
x:
8.400000
8.700000
7.800000
1.600000
9.400000
3.600000
8.700000
9.300000
5.000000
2.200000
```

```
b:
131.900000
134.300000
127.100000
77.500000
139.900000
93.500000
134.300000
139.100000
104.700000
82.300000
```

↑ 既知の行列A

←ベクトルX (求める答え)  
=前ページのベクトルXX

←既知のベクトルb

(右上に続く)

```
iter 1 1.075989e-01 1.000000e-16
iter 2 1.328273e-03 1.000000e-16
(省略)
iter 99 9.776105e-01 1.000000e-16
iter 100 9.874046e-01 1.000000e-16
time: 1.848180e-01 sec , 1.848180e-03 sec/iter
result:
```

↑ 反復計算の履歴、  
反復計算終了時に所要時間も表示

←計算結果ベクトルX

```
9.185754
9.742296
8.072670
-3.429193
11.040893
0.281085
9.742296
10.855379
2.878280
-2.316110
diff:
1 8.400000 9.185754: -7.857539e-01
2 8.700000 9.742296: -1.042296e+00
3 7.800000 8.072670: -2.726703e-01
4 1.600000 -3.429193: 5.029193e+00
5 9.400000 11.040893: -1.640893e+00
6 3.600000 0.281085: 3.318915e+00
7 8.700000 9.742296: -1.042296e+00
8 9.300000 10.855379: -1.555379e+00
9 5.000000 2.878280: 2.121720e+00
10 2.200000 -2.316110: 4.516110e+00
```

↑ 計算結果ベクトルXと最初に設定したベクトルXXの比較 (測定のため収束を妨害してあるので結果が合わなくて良い)

# CG法のOpenACC化： 1.各計算部の並列化：行列ベクトル積（1／2）

- まずは一番実行時間が長い処理である行列ベクトル積を並列化してみる

<pre> 151 // {q} = [A]{p} 152 #pragma acc kernels loop 153     for(i=0;i&lt;N;i++){ 154         q[i] = 0.0; 155         for(j=0;j&lt;N;j++){ 156             q[i] += A[i*N+j]*p[j]; 157         } 158     }                 cg1.c </pre>	<pre> 143 ! {q} = [A]{p} 144 !\$acc kernels loop 145 do i=1, N 146     q(i) = 0.0 147     do j=1, N 148         q(i) = q(i) + A(j,i) * p(j) 149     end do 150 end do                 cg1.f90 </pre>	<p>※cg0.c, cg0.f90は指示文が まったく入っていないコード</p>
--	--	---

- 実はkernelsとloopはまとめて1行で指定することができる
- Fortran版の閉じる指示文も不要

- コンパイルしたところ、Fortran版は外側ループが並列化されたが、C言語版は両ループともseq扱いされ、配列Aの長さも不明と判断されてしまった

```

153, Accelerator restriction: size of
the GPU copy of A is unknown

```

```

153, #pragma acc loop seq
155, #pragma acc loop seq

```

```

145, Loop is parallelizable

```

```

Generating Tesla code

```

```

145, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
147, !$acc loop seq

```

# CG法のOpenACC化： 1.各計算部の並列化：行列ベクトル積（2/2）

- independent/seqとcopyinを指定

```

151 // {q} = [A]{p}
152 #pragma acc kernels copyin(A[N*N])
153 #pragma acc loop independent
154     for(i=0;i<N;i++){
155         q[i] = 0.0;
156 #pragma acc loop seq
157         for(j=0;j<N;j++){
158             q[i] += A[i*N+j]*p[j];
159         }
160     }

```

cg2.c

```

143 ! {q} = [A]{p}
144 !$acc kernels
145 !$acc loop independent
146 do i=1, N
147     q(i) = 0.0
148 !$acc loop seq
149     do j=1, N
150         q(i) = q(i) + A(j,i) * p(j)
151     end do
152 end do
153 !$acc end kernels

```

cg2.f90

- C言語の場合はFortranと比べてcopyやindependentを指定せねば適切に並列化されないことが多いことに気を付けること
- コンパイラの出力をしっかり確認することが重要

```

154, Loop is parallelizable
    Generating Tesla code
154, #pragma acc loop gang, vector(128) /* 省略 */
157, #pragma acc loop seq

```

```

146, Loop is parallelizable
    Generating Tesla code
146, !$acc loop gang, vector(128) ! 省略
149, !$acc loop seq

```

## CG法のOpenACC化： 1.各計算部の並列化：各計算の並列化

- 同様にして、反復計算部の全ての行列・ベクトル計算を並列化していく
- コンパイラのメッセージを確認し、狙い通りに並列化されているかを確認する
  - 特にCの場合はindependent/copy\*/present節をしっかりと挿入する
  - reductionが適切に生成されているかを確認する
    - (このような簡単なプログラムで失敗することはないと思って良いが)

```
cg3.c 134 // {rho} = {r}{z}
      135 rho = 0.0;
      136 #pragma acc kernels
      137 #pragma acc loop independent
      138     for(i=0;i<N;i++){
      139         rho += r[i]*z[i];
      140     }
```

137, Loop is parallelizable

Generating Tesla code

```
137, #pragma acc loop gang, vector(128) /* 省略 */
    Generating implicit reduction(+:rho)
```

```
cg3.f90 127 ! {rho} = {r}{z}
        128 rho = 0.0d0
        129 !$acc kernels
        130 !$acc loop
        131 do i=1, N
        132     rho = rho + r(i) * z(i)
        133 end do
        134 !$acc end kernels
```

131, Loop is parallelizable

Generating Tesla code

```
131, !$acc loop gang, vector(128) ! 省略
    Generating implicit reduction(+:rho)
```



## 状況の確認

- 反復計算部の全ての行列・ベクトル計算を並列化してみた
- 簡単な問題ではあるが  
(問題サイズが小さければ時間もとても短い)  
環境変数 `NVCOMPILER_ACC_TIME=1`  
を設定して実行してみると  
頻繁にデータ転送していることがわかる
  - `NVCOMPILER_ACC_NOTIFY=2`を設定すると  
kernelsの度に配列全体をコピーしている  
こともわかる
- 本当にこんなに頻繁に転送する  
必要はあるのだろうか？

cg3.cの例

```

130: data region reached 20 times
    130: data copyin transfers: 20
        device time(us): total=91 max=15 min=3 avg=4
    132: data copyout transfers: 10
        device time(us): total=62 max=15 min=5 avg=6
138: compute region reached 10 times
    138: kernel launched 10 times
        grid: [1] block: [128]
        device time(us): total=30 max=3 min=3 avg=3
        elapsed time(us): total=178 max=20 min=17 avg=17
    138: reduction kernel launched 10 times
        grid: [1] block: [256]
        device time(us): total=30 max=3 min=3 avg=3
        elapsed time(us): total=166 max=19 min=16 avg=16
138: data region reached 20 times
    138: data copyin transfers: 30
        device time(us): total=126 max=12 min=3 avg=4
    140: data copyout transfers: 10
        device time(us): total=68 max=20 min=5 avg=6
147: compute region reached 1 time
    147: kernel launched 1 time
        grid: [1] block: [128]
        device time(us): total=2 max=2 min=2 avg=2
        elapsed time(us): total=18 max=18 min=18 avg=18

```

## cg3コンパイル時のメッセージの再確認

- 確かに頻繁にコピーが行われている可能性がある

```
nvc -Minfo=accel -acc -gpu=cc70 -tp=host -o cg3_c_acc cg3.c
```

```
main:
```

```
130, Loop is parallelizable  
Generating Tesla code  
130, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */  
130, Generating implicit copyout(z[:N]) [if not already present]  
Generating implicit copyin(r[:N],dd[:N]) [if not already present]  
138, Loop is parallelizable  
Generating Tesla code  
138, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */  
Generating implicit reduction(+:rho)
```

```
nvfortran -Minfo=accel -acc -gpu=cc70 -tp=host -o cg3_f_acc cg3.f90
```

```
main:
```

```
120, Generating implicit copyin(dd(1:n),r(1:n)) [if not already present]  
Generating implicit copyout(z(1:n)) [if not already present]  
122, Loop is parallelizable  
Generating Tesla code  
122, !$acc loop gang, vector(128) ! blockidx%x threadidx%x  
129, Generating implicit copyin(z(1:n)) [if not already present]  
Generating implicit copy(rho) [if not already present]  
Generating implicit copyin(r(1:n)) [if not already present]  
131, Loop is parallelizable  
Generating Tesla code  
131, !$acc loop gang, vector(128) ! blockidx%x threadidx%x  
Generating implicit reduction(+:rho)
```

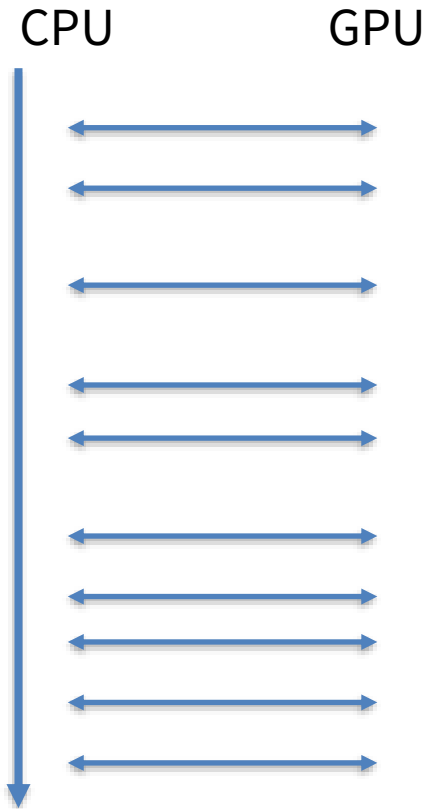
# データ転送の最適化を考える

- 現在のデータ転送状況のイメージ

```

Compute  $r^{(0)} = b - [A]x^{(0)}$ 
for  $i = 1, 2, \dots$ 
  solve  $[M]z^{(i-1)} = r^{(i-1)}$ 
   $\rho_{i-1} = r^{(i-1)} \cdot z^{(i-1)}$ 
  if  $i=1$ 
     $p^{(1)} = z^{(0)}$ 
  else
     $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ 
     $p^{(i)} = z^{(i-1)} + \beta_{i-1} z^{(i)}$ 
  endif
   $q^{(i)} = [A]p^{(i)}$ 
   $\alpha_i = \rho_{i-1} / p^{(i)} \cdot q^{(i)}$ 
   $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$ 
   $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$ 
  check convergence  $|r|$ 
end

```



計算をする度に、関係する配  
列全てを送受信している

↓

更新があったデータだけを送  
受信すれば良いはずでは？

## CG法のOpenACC化手順例：2.データ転送の最適化

- data節を用いてデータ転送を削減してみる
  - 送受信が必要なデータはどれだろうか？

```
#pragma acc data copyin(?) copyout(?)
```

```
!$acc data copyin(?) copyout(?)
```

```

Compute  $r^{(0)} = b - [A]x^{(0)}$ 
for i= 1, 2, ...
  solve  $[M]z^{(i-1)} = r^{(i-1)}$ 
   $\rho_{i-1} = r^{(i-1)} \cdot z^{(i-1)}$ 
  if i=1
     $p^{(1)} = z^{(0)}$ 
  else
     $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ 
     $p^{(i)} = z^{(i-1)} + \beta_{i-1} z^{(i)}$ 
  endif
   $q^{(i)} = [A]p^{(i)}$ 
   $\alpha_i = \rho_{i-1} / p^{(i)} \cdot q^{(i)}$ 
   $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$ 
   $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$ 
  check convergence  $|r|$ 
end

```

※リダクション結果のスカラー変数は自動的にCPU側に送られるため気にしなくてよい

- 途中の計算結果がおかしくないかを確認したい場合には、update節を用いて途中の配列データを覗いてみると良い

## 実は……

- 最初に全てのデータをGPUへ転送し、最後に結果ベクトルを回収するだけで良かった
  - わかっていればいきなりデータ通信の最適化を行っても良かった
  - 少しずつの最適化が行いやすい点はプログラム最適化においてとても助かる

```
#pragma acc data copyin(z[N], dd[N], r[N],
p[N], q[N], A[N*N]) copy(x[N])
```

```
{
  for(iter=1; iter<=maxiter; iter++){
```

```
.....
```

```
  // {q} = [A]{p}
```

```
#pragma acc kernels
```

```
#pragma acc loop independent
```

```
  for(i=0; i<N; i++){
```

```
    q[i] = 0.0;
```

```
    for(j=0; j<N; j++){
```

```
      q[i] += A[i*N+j]*p[j];
```

```
    }
```

```
  }
```

cg4.c

A, p, q全てGPU上にある状態で行列ベクトル積が開始される

```
!$acc data copyin(z(N), dd(N), r(N), p(N),
q(N), A(N,N)) copy(x(N))
```

```
  do iter=1, maxiter
```

```
.....
```

```
! {q} = [A]{p}
```

```
!$acc kernels
```

```
!$acc loop
```

```
do i=1, N
```

```
  q(i) = 0.0
```

```
  do j=1, N
```

```
    q(i) = q(i) + A(j,i) * p(j)
```

```
  end do
```

```
end do
```

```
!$acc end kernels
```

cg4.f90

for自体を  
囲む必要  
あり

(cg5.c と cg5.f90 はさらに念のためpresent節を加えたもの)