九州大学情報基盤研究開発センター

2018.11.27 13:00-17:30

並列プログラミング “超” 入門 講習会

GPUコース：GPU入門

画像処理用の GPUを、多数の演算器で構成された並列計算機として使う技術が注目されています。そのための並列プログラミング手法について、記述が簡単なOpenACCという並列プログラミングモデルを中心に、紹介します。一部、OpenMPや MPIの知識を前提とした内容を含みますが、知らない方でも理解できます。

日程・内容

いずれも

OpenMPコース 11月1日（木）

1台の計算機に搭載された多数の「CPUコア」を使

MPIコース 11月 8日（木）、11月

（どちら

複数台の計算機による「クラスタ型」並列計算機向け ログラム

GPUコース 11月27日（火）

画像処理用のGPUを並列計算機として使うための並列プログラム

会　場 九州大学 伊都キャンパス
情報基盤研究開発センター 2階 多目的教室

時　間 13:00 ～ 17:30

本講習会の構成

- 第一部（本資料を用いる）
  - ITOの説明
  - 実際にログインしてみる、PGIコンパイラをインストールする
  - GPUと並列計算についての説明

- 第二部（別資料を用いる）
  - OpenACCの説明と実習
  - ～17:30：質疑・自由演習

- 配付したITOのアカウントは1週間有効です
  - 受講後にも色々と試してみてください

はじめに

- ITOへのログイン（別紙）
- PGIコンパイラ（Community Edition）の導入
  - OpenACCのために必要なソフトウェアを入手する
    - インストール済のPGIコンパイラは古い、利用者に制限がある
  - ログインノードにSSHログインし、以下のコマンドでWebアクセス
    - elinks pgroup.com
  - pgilinux-2018-1810-x86-64.tar.gz　を入手
    - エラーメッセージが出て終了してしまうが恐らくダウンロード自体はできている、md5sum で正しくDLできたかを確認
    - 正しいチェックサムは 9d2254112147d634e5fa422defd9e66b
  - インストール先を作成：mkdir opt
  - 作業場所を作成：mkdir tmp
  - 入手したファイルを展開：
    tar zxvf ./pgilinux-2018-1810-x86-64.tar.gz －C tmp
  - 実際のインストール作業：cd tmp　→　./install

PGIコンパイラインストール：手順例メモ

- ./install
  - Press enter to continue... enter
  - ライセンス情報を読む
    → Do you accept these terms? (accept,decline) accept
  - Please choose install option: 1
  - Installation directory? [/opt/pgi] ~/opt/pgi
  - ライセンス情報を読む → accept
  - Do you wish to update/create links in the 2018 directory? (y/n) y
  - Press enter to continue... enter
  - Do you want to install Open MPI onto you system? (y/n) y
  - Do you want to enable NVIDIA GPU support in Open MPI? (y/n) y
  - Do you wish to obtain permanent license key or configure license service? (y/n) n
  - Do you want the files in the install directory to be read-only? (y/n) n
    - y にしても構わないのだが、あとで更新・追加する際に引っかかる可能性あり

- 問題が起きたらCtrl+Cでキャンセルし、
  ~/optを消して（名前を変えて）やり直す

PGIコンパイラインストール：インストール後の確認

- ファイルの確認
  - ls ~/opt/pgi以下にlinux86-64, linux86-64-llvm, modulefilesディレクトリとlicense関係のファイルが存在することを確認

- modulefileの使用
  - module load ~/opt/pgi/modulefiles/pgi/18.10
  - pgccなどのコマンドが利用可能となることを確認
  - pgaccelinfoコマンドによりOpenACC対応GPUの情報などが得られる、が、ログインノードで実行してもCUDAドライバのバージョン情報しか得られない

動作確認

- 以下の内容を適当なファイルに保存する（仮にtest.shとする）

```
#!/bin/bash
#PJM -L "rscunit=ito-b"
#PJM -L "rscgrp=ito-b-lecture-4"
#PJM -L "vnode=1"
#PJM -L "vnode-core=9"
#PJM -L "elapse=00:05:00"
#PJM -o out.txt
#PJM -j
#PJM -S

module load ~/opt/pgi/modulefiles/pgi/18.10
pgaccelinfo
```

  - コピペする場合は記号が全角に化けるなどしないか気を付けること
  - 書かれている内容については後で説明する

- pjsub test.shを実行する
- しばし待つ（バックエンドでジョブが実行される）
- out.txtファイルが作られたら中身を確認する
  - （作られてすぐは中身が書き込まれていないことあり、もう少し待つ）

スーパーコンピュータITO

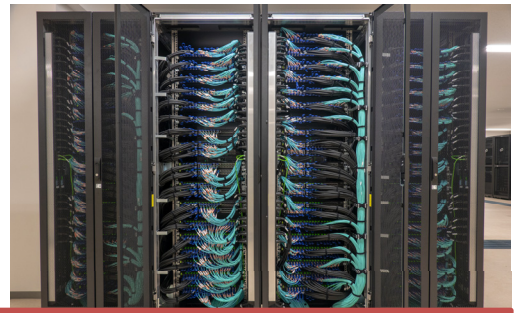
- この建物の地下1Fに設置されているスーパーコンピュータ


バックエンド サブシステムA


バックエンド サブシステムB


フロントエンド サブシステム


共有ストレージ


ノード間高速ネットワーク

- スーパーコンピュータ
  ＝「パソコンのようなもの」が大量に並べて
  つなげてある、ようなイメージ
  - どの程度「パソコンのようなもの」であるかは
    システム次第
  - ITOは「普通のパソコンに近い」部分が多い
    - 普通のパソコンで使っているものより
      「ちょっとスゴイ」ものを多数並べて
      全体として「とてもスゴイ」性能を得ている

数字で見るITO

より詳細な情報はWebを参照
https://www.cc.kyushu-u.ac.jp/scp/system/ITO/01_intro.html

- 主に計算に使う部分：バックエンドとフロントエンド
  - バックエンド　サブシステムA
    - 1CPUあたり18コア×3GHz
    - （2CPU＋192GBメモリ）×2000
    - 合計 6.91 PFLOPS
  - バックエンド　サブシステムB
    - 1CPUあたり18コア×2.3GHz
    - （2CPU＋384GBメモリ＋4GPU）×128
    - 合計 3.05 PFLOPS
  - フロントエンド
    - 1CPUあたり18コア×2.3GHz、（2CPU+384GBメモリ+1GPU）×160
    - 1CPUあたり22コア×2.2GHz、（16CPU+12TBメモリ+1GPU）×4

- バックエンド1台の性能はPCと比べて「ちょっとスゴイ」
- 多数並べると全体として「とてもスゴイ」

今回の利用対象

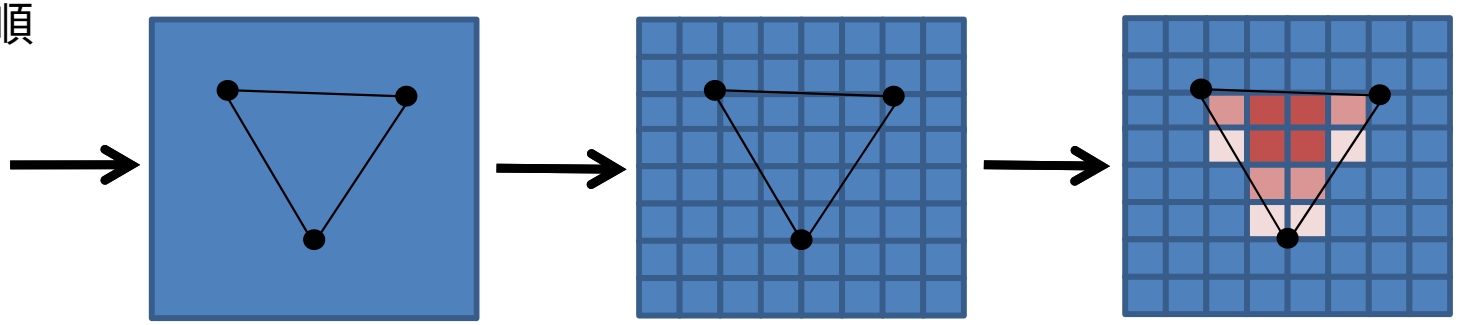
➢ 1PFLOPS＝ 1秒間に1000兆回の実数計算ができる
- 2018年のハイエンドAndroid端末は500GFLOPS程度
- 1000=1K, 1000K=1M, 1000M=1G, 1000G=T, 1000T=1P

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)

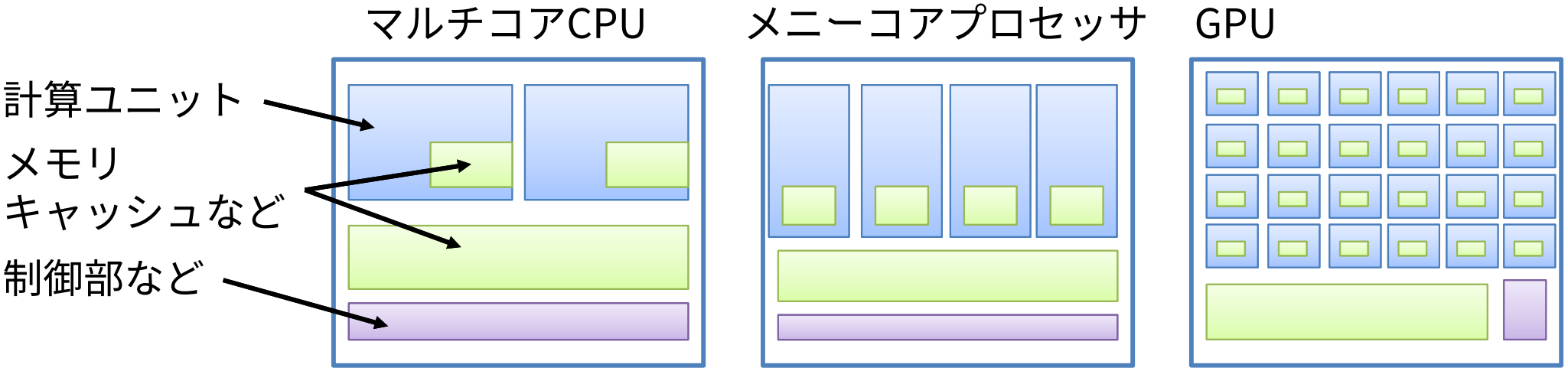


オブジェクト単位、頂点単位、ピクセル単位で同時（並列）処理が可能

  - 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

ハードウェア性能の比較

※名古屋大版

	Tesla P100	Xeon Gold 6140（サブシステムB）	Xeon Gold 6154（サブシステムA）	SPARC64 XIfx（「京」後継機）
アーキテクチャ名	Pascal	Skylake-SP		SPARC64 IXfx
コア数	3584 FP32/1792 FP64 (56 SMs)	18（HTにより36スレッド実行可能だが、HTは無効化している）		32（+2アシスタントコア）
動作周波数	1.328 GHz – 1.480 GHz	2.3 GHz - 3.7 GHz	3.0 GHz – 3.7 GHz	2.2 GHz
搭載メモリ	HBM2 16 GB	DDR4 192 GB/socket	DDR4 96 GB/socket	HMC 32GB
HPL ピボット選択付きLU分解	4 TFLOPS程度 ※NVIDIA提供バイナリ実測値	0.9 TFLOPS程度 ※MKL実測値	1.1 TFLOPS程度 ※MKL実測値	0.97 TFLOPS ※名古屋大提供値
STREAM Triad A[i]=B[i]+C[i]	550 GB/s	95 GB/s	95 GB/s	210 GB/s

↑非常に高い性能を持つ、使わなければもったいない！

P100のHW構成（コア数の詳細）

- 56のStreaming Multiprocessor (SM)
- SMあたり32のFP64コアと64のFP32コア、キャッシュなど
- コアのスケジューリングはWARP単位（SIMD長32のようなもの）

世界のスパコンで使われるGPU

- TOP500 1~10位中、5システムがNVIDIAのGPUを活用
  - 上位50中の13、上位100中の24、全500中の128

	Name	Site	Country	Total Cores	ACC Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	ACC
1	Summit	ORNL	US	2397824	2196480	143500	200794.88	Volta GV100
2	Sierra	LLNL	US	1572480	1382400	94640	125712	Volta GV100
3	Sunway TaihuLight	Wuxi	China	10649600		93014.594	125435.9	None
4	Tianhe-2A	Guangzhou	China	4981760	4554752	61444.5	100678.66	Matrix-2000
5	Piz Daint	CSCS	Switzerland	387872	319424	21230	27154.3	Tesla P100
6	Trinity	LANL/SNL	US	979072		20158.7	41461.15	None
7	ABCI	AIST	Japan	391680	348160	19880	32576.635	Tesla V100 SXM2
8	SuperMUC-NG	Leibniz	Germany	305856		19476.6	26873.856	None
9	Titan	ORNL	US	560640	261632	17590	27112.55	Tesla K20x
10	Sequoia	LLNL	US	1572864		17173.224	20132.659	None

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？
  - ITOにGPUが搭載されているから……というのも事実ではあるが
  - 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
- どうやって使う？
  - 並列化されたソフトウェア（アプリケーション）を使う
  - （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
  - 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

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書くことのできる並列化プログラミング言語
  - NVIDIAのGPUに限定されない
- 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
  - CPU向けの並列化に広く用いられているOpenMPを踏襲、理解も利用も容易
  - 最適化を行う上ではある程度GPUの知識があった方が良い
- 「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化を行う上で最も簡単でコストパフォーマンスの高い方法（だと思う）
  - コストパフォーマンスが高い＝労力に対して十分な性能が得られる

OpenACCに関する情報、ツール、etc.

- 規格などの情報
  - https://www.openacc.org/
- 主な対応コンパイラ
  - 商用：PGI（無償版あり）、Cray、国家超级计算无锡中心
  - 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
  - ソフテック社（PGIコンパイラの代理店）の資料
    - OpenACC ディレクティブによるプログラミング by PGI Compilers
      https://www.softek.co.jp/SPG/Pgi/OpenACC/

（GPUやOpenACCの前に）並列計算の基礎知識

- 並列計算とは何か？並列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：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
    - 平行：何らかの処理を同時並行的に行う
  - 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
- 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HW）の持つ能力をフル活用したいから
  - 現代のプロセッサは複数の計算コアを搭載
  - PC用CPUもスマートフォン向けCPUも4～8程度のコア数が主流
  - 大規模なスーパーコンピュータの総コア数は100万以上
  - →　HWの性能を十分に引き出すには並列計算が必須

並列化の基本的なイメージ：単純なループ並列化の例

- 元々の逐次計算
  - 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];
       }
```

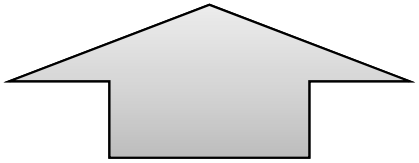
```
PE2    for(i=0; i<N; i++){
        D[i] = E[i] + F[i];
       }
```

PE: Processor Element、CPUやGPUなど計算をする「何か」

- 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];
       }
```

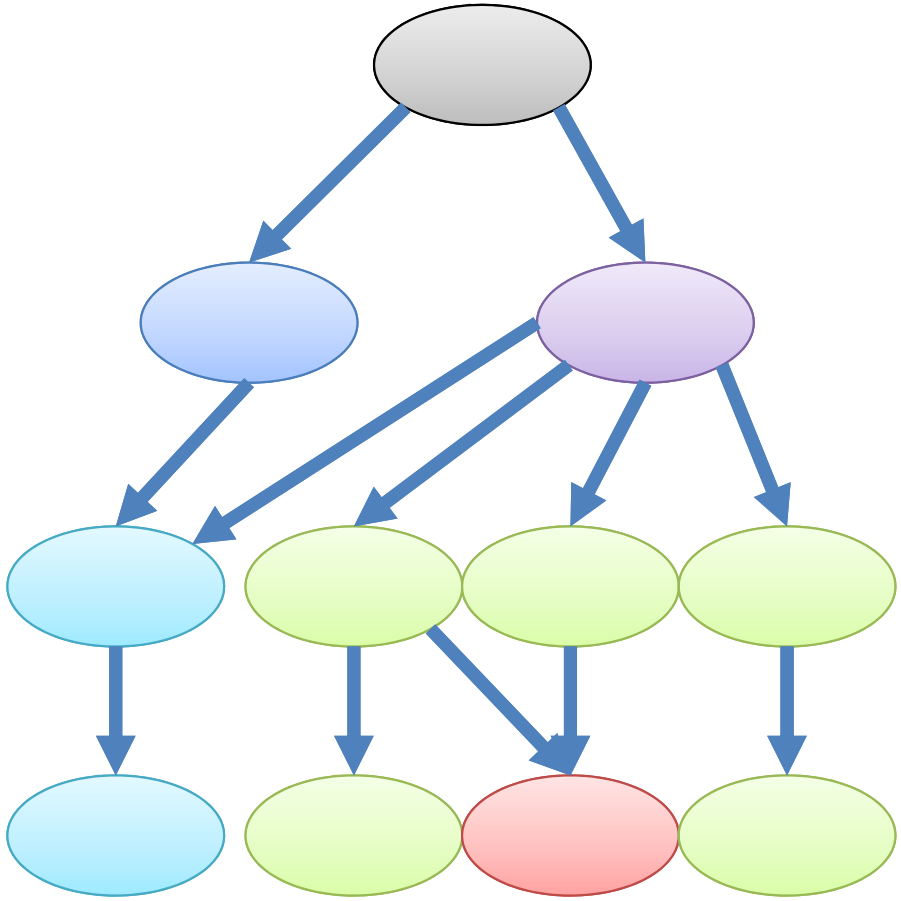
```
PE2    for(i=N/2; i<N; i++){
        A[i] = B[i] + C[i];
        D[i] = E[i] + F[i];
       }
```



- 本講習会で扱うOpenACCによる並列化はこちらのイメージ
  - OpenMPもこちら

並列化の基本的なイメージ：タスク並列化

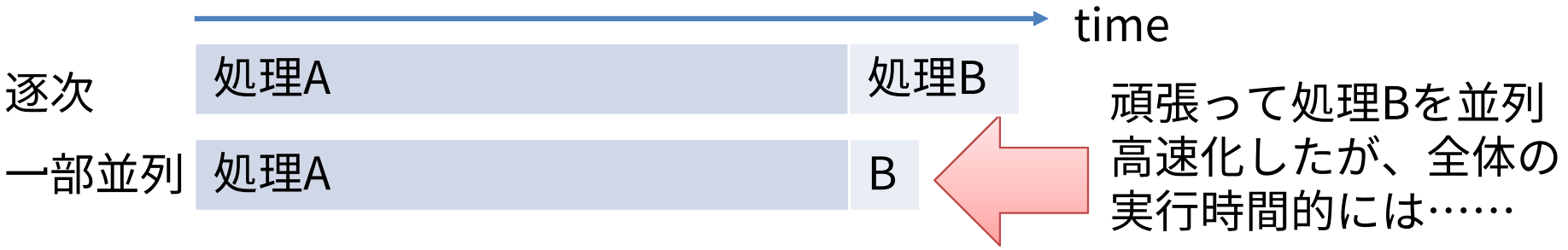
- 単純なループによる並列化とは異なり、
  やや大きい粒度の並列計算などでよく活用
- OpenMPにおいて近年サポートが活発
- GPU（OpenACC）にはあまり適して
  いないため今回は扱わない



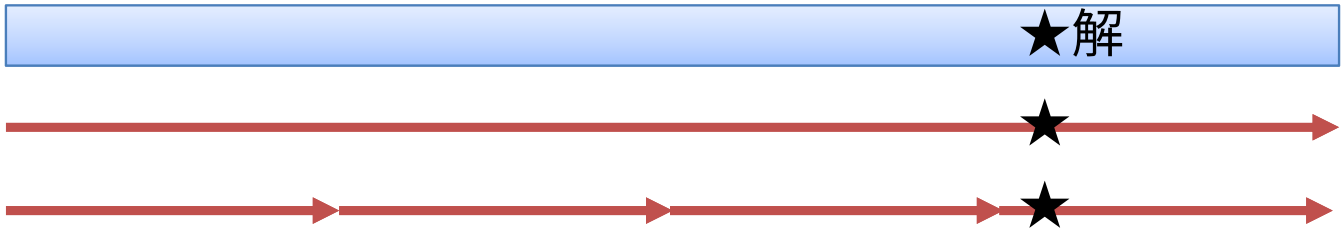
並列計算の注意点

- 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];
}
```



  - 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GPUに担当させるかをちゃんと考える必要がある

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
    - Voltaから変わってしまった
- 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

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・利用側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
  - コード記述者が明示することで適切な最適化を行うのが現在のトレンド

並列計算を行う方法（言語など）の例

- 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などの関数を使う
- pthread：スレッド並列処理のための関数群
  - pthread_create, pthread_……
  - 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
  - 現状では利用できる環境が少ないが、将来的にはOpenACCを吸収？
- CUDA：NVIDIA社のGPU向け、GPUを普及させた最大要因の一つ、NVIDIA GPUの能力をフル活用したければ必須
- OpenCL：「汎用版CUDA」のようなもの、FPGAなどでも利用可能
- 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

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 = gridDimd.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を元に計算する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
  - 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, ...
  - 行いたい処理が対応している場合は容易にGPUの性能を活用することができる

サンプルソースコードはITOの /home/tmp/gpu 以下に置いてあります
※実質的に「演習の答え」となるものもあるので注意

九州大学情報基盤研究開発センター
2018.11.27 13:00-17:30

並列プログラミング“超”入門 講習会
GPUコース：OpenACC入門

OpenACC

- GPUプログラムを簡単に記述するために開発された指示文規格
  - GPU向けのOpenMP、のようなもの
  - 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
    - 初登場が2011年、まだ10年経っていない
  - CUDAでしか書けない処理も多いが、「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十分高性能
  - 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
  - 基本的には単体GPUを駆動するための手段、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する

単純なOpenACCプログラムの例

- C (vector0.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
  int i, n=10;
  double v1[10], v2[10];

  for(i=0; i<n; i++){
    v1[i] = (double)(i+1);
    v2[i] = 0.0;
  }

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

  for(i=0;i<n;i++){ printf(" %.2f", v1[i]); }
  printf("¥n");
  for(i=0;i<n;i++){ printf(" %.2f", v2[i]); }
  printf("¥n");

  return 0;
}
```

- Fortran (vector0.f90)

```fortran
program main
  implicit none
  integer :: i, n=10
  double precision :: v1(10), v2(10)

  do i=1, n
     v1(i) = dble(i)
     v2(i) = 0.0d0
  enddo

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

  do i=1,n
     write(*,'(1H F8.2)',advance="NO")v1(i)
  enddo
  write(*,*)""
  do i=1,n
     write(*,'(1H F8.2)',advance="NO")v2(i)
  enddo
  write(*,*)""
end program main
```

OpenACC並列化対象
＝GPU上で実行される

- 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  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char **argv)
6  {
7    int i, n=10;
8    double v1[10], v2[10];
9
10   for(i=0; i<n; i++){
11     v1[i] = (double)(i+1);
12     v2[i] = 0.0;
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[
21   printf("¥n");
22   for(i=0;i<n;i++){ printf(" %.2f", v2[
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

- CPUからGPUに対して計算指示が行われる
- GPU上のメモリを確保、CPUからGPUへ必要なデータを転送

GPU上の計算コアにより並列計算される

- CPUはGPUの計算終了を待つ
- GPUからCPUへ結果データを転送、GPU上のメモリを解放

コンパイル例

- pgccまたはpgfortranでコンパイルする
  - -acc　OpenACC指示文を有効化
  - -ta　OpenACCの対象ハードウェア（対象GPU）を指定
  - -Minfo=accel　OpenACC化に関する情報を出力
  - -tp　対象CPUを指定

```
pgcc -Minfo=accel -acc -ta=tesla:cc60 -tp=skylake -o vector0_c_acc vector0.c
main:
     16, Generating implicit copyout(v2[:])
         Generating implicit copyin(v1[:])
     17, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
pgfortran -Minfo=accel -acc -ta=tesla:cc60 -tp=skylake -cpp -o vector0_f_acc vector0.f90
main:
     13, Generating implicit copyout(v2(:))
         Generating implicit copyin(v1(:))
     14, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         14, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

データ転送やGPU上の計算コアの使い方は
コンパイラが適当に判断してくれた
※最適でない（問題が起きる）こともある

実行例

- ./a.out
  - CPU向けの実行可能ファイルと同様にそのまま実行できる

C版の場合

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

Fortran版の場合

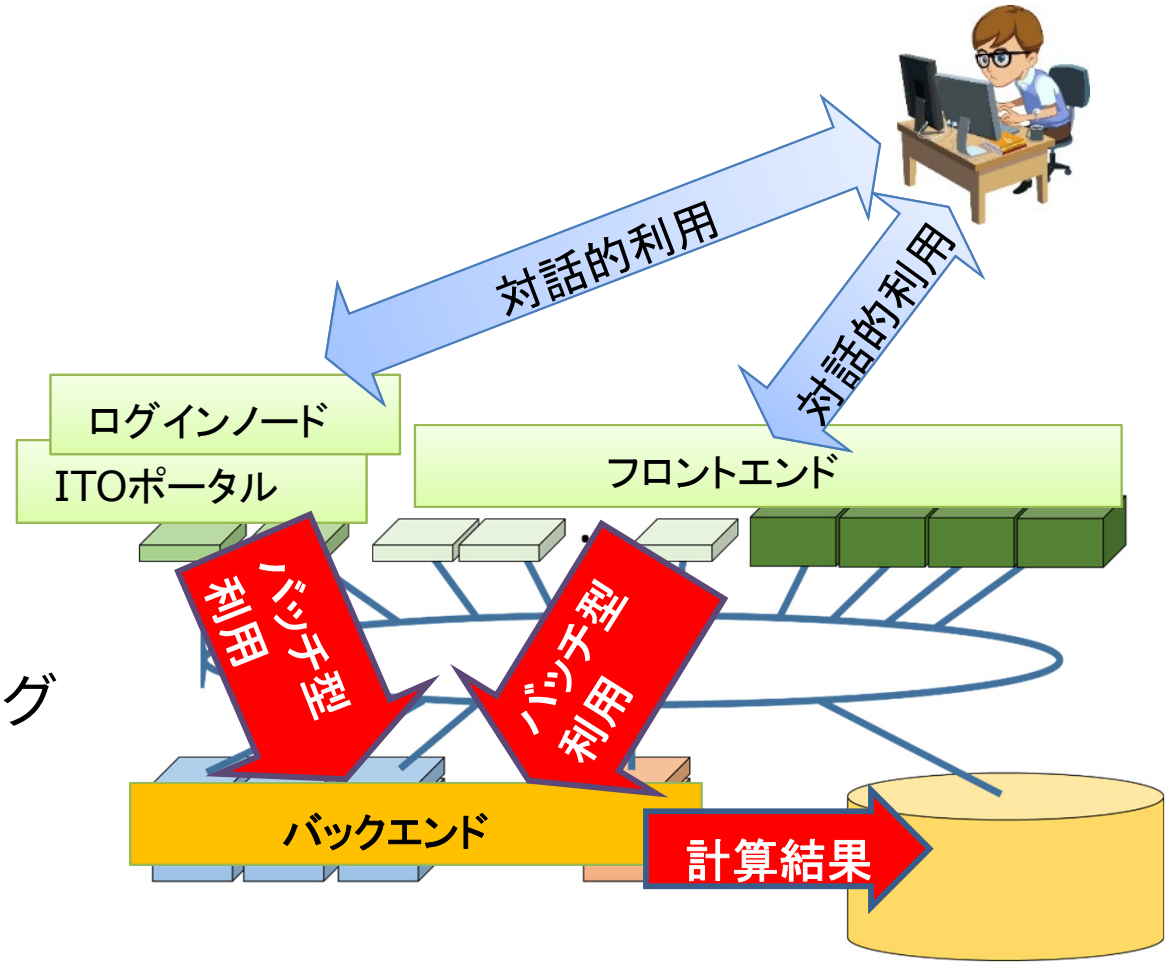
```
$ ./a.out
   1.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00  10.00
   2.00   4.00   6.00   8.00  10.00  12.00  14.00  16.00  18.00  20.00
```

- LD_LIBRARY_PATHなどの対応は必要
  - module loadにより解決可能
  - （envコマンドにより環境変数をみれば確認可能）
  - GPU向けにコンパイルしたものはGPUがないと実行時エラー

バッチジョブの利用

- ITOのように多人数で利用するシステムでは、計算ノードに直接ログインするのではなく「バッチジョブシステム」を利用すると便利
  - 計算機の稼動率を高める
  - 特定ユーザに占有させない
  - セキュリティを高める

- 利用手順
  1. ジョブスクリプトの作成
  2. ジョブの投入
  3. システムがスケジューリングして実行
  4. 結果の確認



バッチジョブシステムの仕組み

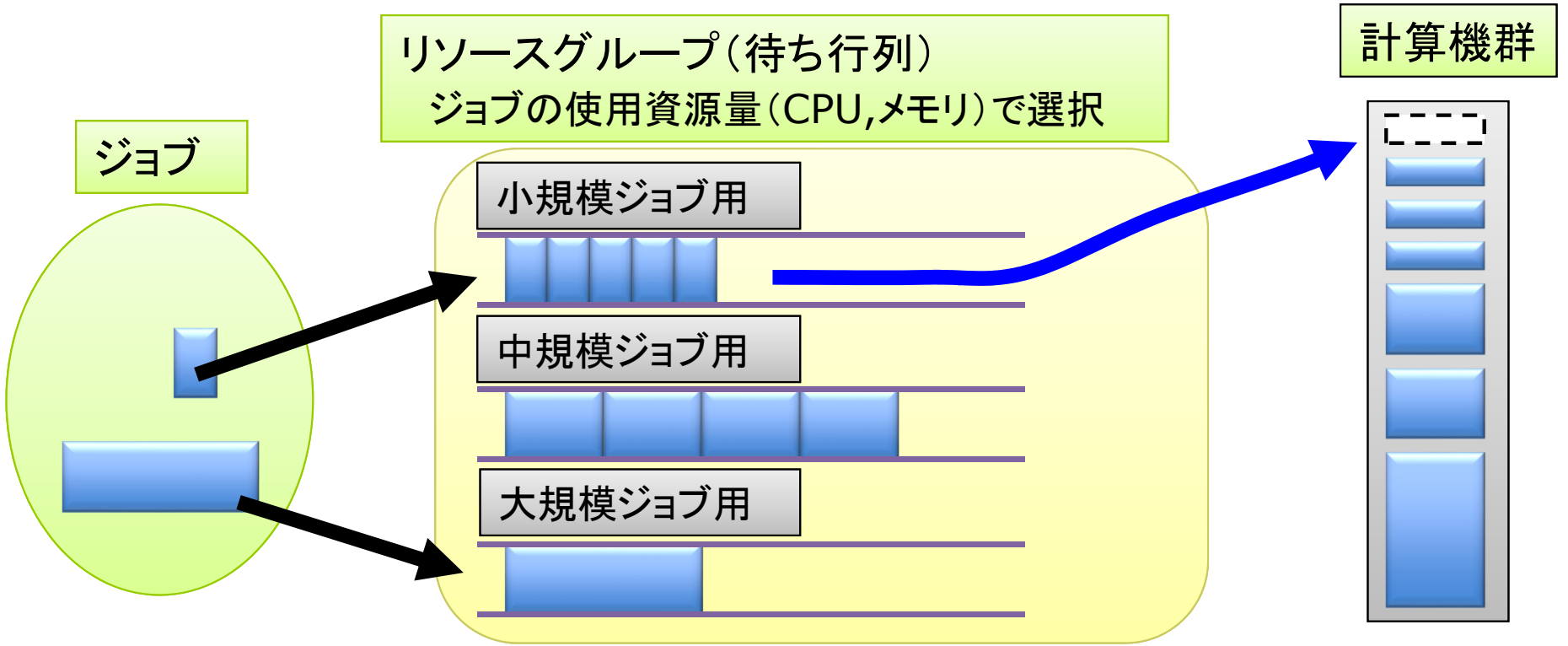
- 処理してほしい内容を記述したファイル（ジョブスクリプト）を作成しシステムに投入
  - ジョブとして受付
- 資源の空き状況に応じて順に処理される
  - 要求内容や空き状況によっては先を越される
  - 適切な資源量の指定が重要

```
#!/bin/sh

#PJM -L "vnode=4"
#PJM -L "vnode-core=36"
#PJM -L "rscunit=ito-b"
#PJM -L "rscgrp=ito-g-16"
#PJM -L "elapse=10:00"

mpiexec -np 16 ./a.out
```

ジョブスクリプトの例



バッチ処理に用いるコマンド

- バッチジョブの投入
  - pjsub　ジョブスクリプトファイル名
- バッチジョブの状況確認
  - pjstat
    - 終了済のジョブの情報を見る場合はpjstat –H
    - 特定ジョブの詳細情報をみる場合はpjstat –S ジョブID
    - –Hと–Sの組み合わせも可能（終了済の特定ジョブの詳細を確認）
- バッチジョブのキャンセル
  - pjdel　ジョブID

  - 各コマンドに追加できるオプションは--helpオプションを付けることで確認可能

pjsub バッチジョブの投入

- コマンド

```
$ pjsub　オプション　ジョブスクリプトファイル名
```

  - オプション：使用する資源等に関する指定
    - スクリプトファイル内に書いても良いし、オプションで与えても良い
    - オプションで与えた情報が優先される
  - ジョブスクリプトファイル：依頼する処理内容
    - シェルスクリプトとして記述

  - 例）ジョブスクリプトファイル test.sh を投入

```
$ pjsub test.sh
[INFO] PJM 0000 pjsub Job 28246 submitted.
```

ジョブ ID

ジョブスクリプト例

```
#!/bin/bash                                  （スクリプト記述にbashを使用）
#PJM -L "rscunit=ito-b"                      サブシステムBを利用
#PJM -L "rscgrp=ito-b-lecture-4"             講習会用のリソースグループを指定
#PJM -L "vnode=1"                            1GPU使えれば良いので1/4ノード
#PJM -L "vnode-core=9"                       対応するコア数は9
#PJM -L "elapse=00:05:00"                    5分で打ち切り
#PJM -j                                      標準出力とエラー出力を統合
#PJM -S                                      ノード利用情報を出力

module load ~/opt/pgi/modulefiles/pgi/18.10    modulefileを用いて環境設定
pgaccelinfo                                  OpenACC対応デバイス情報の確認
./a.out                                      プログラムの実行
```

「動作確認」で書いていた
#PJM -o out.txt
は、標準出力の内容を指定のファイルへ書き出すというオプション。
エラー出力についても同様に-eで指定可能。
-o –e -j 全て指定すると-oで指定したファイルに全て書き出される。

演習

- サンプルプログラム（vector0.c または vector.f90）をコンパイルし、バッチジョブとして実行してみる
- 実行結果が確認できたら、次は環境変数PGI_ACC_TIMEをセットして実行してみる
  - export PGI_ACC_TIME=1　をプログラム実行よりも前の行に書く
  - GPUがどのような仕事をしたのかが確認できるようになる
    - 例

```
Accelerator Kernel Timing data
/home/usr0/m70000a/work/gitprojects/testprograms/openacc/vector0/vector0.c
  main  NVIDIA  devicenum=0
    time(us): 36
    16: compute region reached 1 time                    ←GPU上での計算
        17: kernel launched 1 time
            grid: [1]  block: [32]
             device time(us): total=4 max=4 min=4 avg=4
            elapsed time(us): total=459 max=459 min=459 avg=459
    16: data region reached 2 times                      ←CPU-GPU間の通信
        16: data copyin transfers: 1
             device time(us): total=13 max=13 min=13 avg=13
        21: data copyout transfers: 1
             device time(us): total=19 max=19 min=19 avg=19
```

補足

- もっと細かく確認したい場合には環境変数 PGI_ACC_NOTIFY も有効
  - 1,2,4,8のビット組み合わせで指定、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
  - 例：export PGI_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

OpenACCプログラムの構成

- 指示文（directive）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
- 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

  - どちらを用いても良いが、本講習会ではkernelsを用いる

再掲：単純なOpenACCプログラムの例

- C (vector0.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
  int i, n=10;
  double v1[10], v2[10];

  for(i=0; i<n; i++){
    v1[i] = (double)(i+1);
    v2[i] = 0.0;
  }

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

  for(i=0;i<n;i++){ printf(" %.2f", v1[i]); }
  printf("¥n");
  for(i=0;i<n;i++){ printf(" %.2f", v2[i]); }
  print

  retur
}
```

- Fortran (vector0.f90)

```fortran
program main
  implicit none
  integer :: i, n=10
  double precision :: v1(10), v2(10)

  do i=1, n
     v1(i) = dble(i)
     v2(i) = 0.0d0
  enddo

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

  do i=1,n
     write(*,'(1H F8.2)',advance="NO")v1(i)
  enddo
  write(*,*)""
  do i=1,n
     write(*,'(1H F8.2)',advance="NO")v2(i)
```

（OpenMPと同様に）

- C/C++では{}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

```
pgcc -Minfo=accel -acc -ta=tesla:cc
main:
   16, Generating implicit copyout(v2[:])
       Generating implicit copyin(v1[:])
   17, Loop is parallelizable
       Accelerator kernel generated
       Generating Tesla code
       17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

- Fortran

```
pgfortran -Minfo=accel -a
main:
   13, Generating implicit copyout(v2(:))
       Generating implicit copyin(v1(:))
   14, Loop is parallelizable
       Accelerator kernel generated
       Generating Tesla code
       14, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

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

アクセラレータ(GPU)向けのカーネルが生成された
Tesla向けのコードが生成された
ループの並列化が行われた

- 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

手動による適用：C版

- C (vector0.c)

```
 4  int main(int argc, char **argv)
 5  {
 6    int i, 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
```

- C (vector1.c)

```
 4  int main(int argc, char **argv)
 5  {
 6    int i, 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 copyout(v2[:]) copyin(v1[:])
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、「copyout」「copyin」
「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 (vector0.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
23    enddo
24    write(*,*)""
25  end program main
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  !$acc kernels copyout(v2(:)) copyin(v1(:))
12  !$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(*,*)""
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、「copyout」「copyin」
「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 copyout(v2[:]) copyin(v1[:])
#pragma acc loop gang, vector(32)
  for(i=0; i<n; i++){
    v2[i] = v1[i] * 2.0;
  }
```

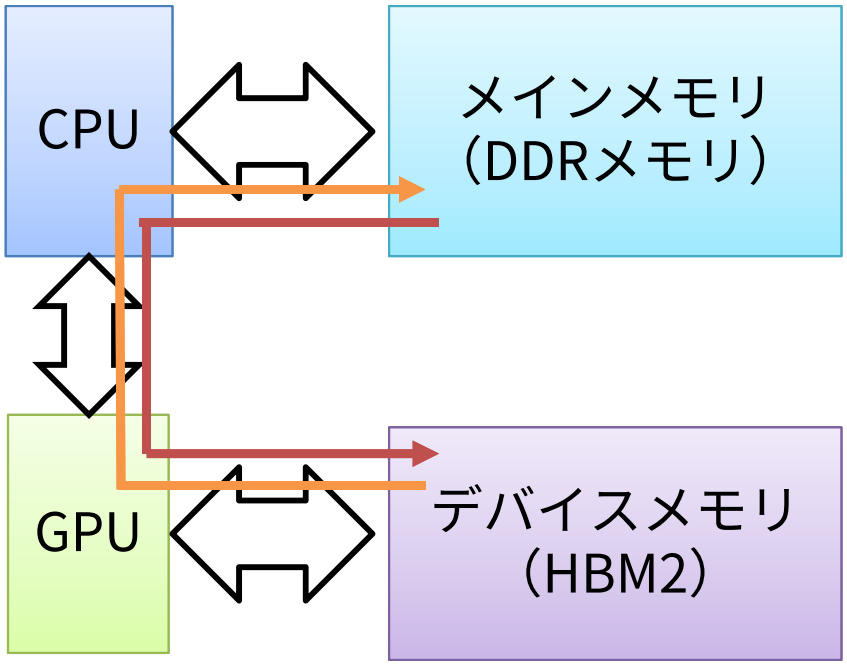
```
#pragma acc kernels ¥
copyout(v2[:]) copyin(v1[:])
#pragma acc loop gang, vector(32)
  for(i=0; i<n; i++){
    v2[i] = v1[i] * 2.0;
  }
```

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

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

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
  - 例外については後述
- 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HBM2へのデータ転送
  - GPUカーネル終了後：HBM2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++ではコンパイラが長さを認識できない配列を扱うことが多いため注意が必要
  - 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PGI_ACC_NOTIFYを用いて確認すること。

※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

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))
```

  - :N　のような省略表記も可能（先頭からN要素が送られる）

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？

```c
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);
  }

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

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

  return 0;
}
```

(vector2.c)

```f90
program main

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

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

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

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

end program main
```

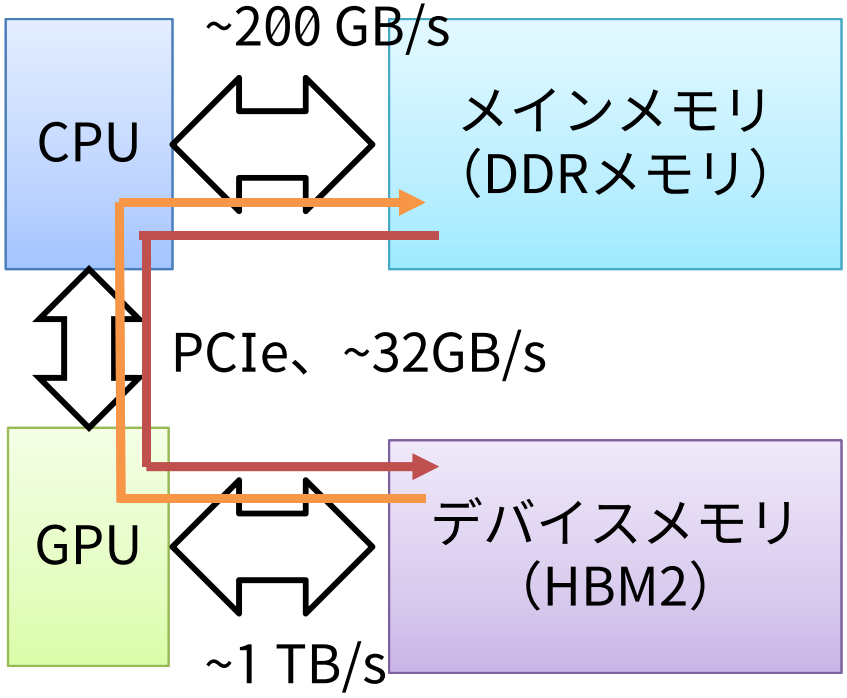
(vector2.f90)

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

デバイスメモリの生成と
破棄を繰り返してしまう
（余計な時間がかかる）

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い
  - CPU-メインメモリ
    - ITOの場合はDDR4で100GB/s/socket弱（STREAM Triad実測）
  - GPU-デバイスメモリ
    - HBM2、550GB/s程度（STREAM Triad実測）
      - 後継機のV100では800GB/sを超える
  - CPU-GPU
    - PCI Express Gen.3 x16
      - 理論性能でも最大16GB/s（×双方向）
  - GPU-GPU間
    - NVLink、20GB/s*1or2（×双方向）
      - 1GPUあたり4本のNVLink
      - Power系CPUではCPU-GPU間でもNVLinkが使えるため高速



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)
構造化ブロック
```

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

  - data指示文とは好みで使い分けてよい
    - プログラム全体の見通しが悪くならないように注意
      - 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？

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

```c
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;
    }
  }

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

  return 0;
}
```

(vector3.c)

```f90
program main

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

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

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

end program main
```

(vector3.f90)

※コンパイラが判断に失敗する場合はkernelsにpresent節を加えると良い
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

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
  - C/C++　#pragma acc data copyin(A[head:length][0:N])
    - C/C++は右側の次元が低次元、低次元分は全て転送する必要がある
  - Fortran　!$acc data copyin(A(1:N, begin:end))
    - Fortranは左側の次元が低次元、低次元分は全て転送する必要がある

データ送受信時の注意点・補足事項

- 部分転送時の範囲には変数を用いても良い
- 配列長については注意が必要
  - 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しておく必要がある
    - 範囲は変数による指定で良い、間接参照をしているときなどに注意

```
double *v1, *v2;
int *index;
v1 = (double*)malloc(sizeof(double)*n));
v2 = (double*)malloc(sizeof(double)*n));
index = (int*)malloc8sizeof(int)*n);
#pragma acc kernels
  for(i=0; i<n; i++){
    v2[i] = v1[index[i]] * 2.0;
  }
```

```
  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
```

具体例は vector12.c/f90 vector13.c/f90 を参照

  - ➢ v1の範囲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
  - ➢ copyin(v1[n]) および copyin(v1(n)) を加えると正しく動作する

- Deep copyはできない（可能となりつつある）
  - 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
  - Fortran：allocatable属性やpointer属性を持つメンバを含む派生型

演習

- サンプルプログラム（vector2.c, vector2.f90）をコンパイルし、バッチジョブとして実行してみる
- さらに、data指示文を挿入したプログラム（vector3.c, vector3.f90）も実行し、比較してみる
  - 環境変数PGI_ACC_TIMEをセットして実行すると容易に比較が可能

```
Accelerator Kernel Timing data
/home/usr0/m70000a/work/gitprojects/testprograms/openacc/lecture201811/vector2.c
  main  NVIDIA  devicenum=0
    time(us): 522
    14: compute region reached 10 times
        15: kernel launched 10 times
            grid: [1]  block: [32]
             device time(us): total=22 max=4 min=2 avg=2
            elapsed time(us): total=734 max=532 min=21 avg=73
    14: data region reached 20 times
        14: data copyin transfers: 10
             device time(us): total=93 max=34 min=6 avg=9
        18: data copyout transfers: 10
             device time(us): total=407 max=351 min=6 avg=40
```

回数を示す値が変化することに気がつくはずである

  - 余裕がある人は、配列の長さや本数を増やすなどして比較してみる

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;
    }
```

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

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見えるのだろうか？ ⇨ 計算前の値が見える

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

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

(vector20.c)

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

(vector20.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

```c
#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;
    }
  }
```

(vector21.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
```

(vector21.f90)

OpenACCにおける変数や配列の共有・非共有の扱い

- 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指示節により変更することも可能

- 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干渉する
  - 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

ループ並列化方法の指定

- 現実のプログラムではコンパイラが全てのループの並列化の判断を行うのは難しい、プログラマが並列化の判断をする必要がある
- loop指示文
  - 並列化対象ループを指定する
  - さらに以下の指示節と組み合わせることで動作の制御が可能
  - ➢ 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

単純な行列積プログラムの例

- C (matmul0.c)

```c
double **a=NULL, **b=NULL, **c=NULL;
// mallocでa,b,cを確保
#pragma acc kernels
  for(i=0; i<n; i++){
    for(j=0; j<n; j++){
      for(k=0; k<n; k++){
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
```

- Fortran (matmul0.f90)

```fortran
double precision, allocatable :: a(:,:), b(:,:), c(:,:)
allocate(a(n,n), b(n,n), c(n,n)
!$acc kernels
  do i=1, n
     do j=1, n
        do k=1, n
           c(j,i) = c(j,i) + a(k,i) * b(j,k)
        enddo
     enddo
  enddo
!$acc end kernels
```

n=10で実行してみた

C

```
37: kernel launched 1 time
        grid: [1]  block: [1]
         device time(us): total=108 max=108 min=108 avg=108
        elapsed time(us): total=130 max=130 min=130 avg=130
```

Fortran

```
40: kernel launched 1 time
        grid: [1x10]  block: [128]
         device time(us): total=5 max=5 min=5 avg=5
        elapsed time(us): total=418 max=418 min=418 avg=418
```

どうやら実行時間に大きな差があるようだ、何故だろう？

単純な行列積プログラムの例

- Fortran (matmul0.f90)

```
double precision, allocatable :: a(:,:), b(:,:), c(:,:)
allocate(a(n,n), b(n,n), c(n,n)
!$acc kernels
  do i=1, n
     do j=1, n
        do k=1, n
           c(j,i) = c(j,i) + a(k,i) * b(j,k)
        enddo
     enddo
  enddo
!$acc end kernels
```

```
pgfortran -Minfo=accel -acc -ta=tesla:cc60 ¥
 -tp=skylake -o m0f_f_acc matmul0.f90
main:
37, Generating implicit copyin(a(1:n,1:n))
    Generating implicit copy(c(1:n,1:n))
    Generating implicit copyin(b(1:n,1:n))
38, Loop is parallelizable
39, Loop is parallelizable
40, 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
    Accelerator kernel generated
    Generating Tesla code
    38, !$acc loop gang ! blockidx%y
    39, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
    40, !$acc loop seq
```

- copy関係はコンパイラの判断で特に問題はない
- 配列cの依存関係に関するメッセージは出ているが、3重ループの外側2つが並列化された

単純な行列積プログラムの例

- C (matmul0.c)

```
double **a=NULL, **b=NULL, **c=NULL;
// mallocでa,b,cを確保
#pragma acc kernels
  for(i=0; i<n; i++){
    for(j=0; j<n; j++){
      for(k=0; k<n; k++){
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
```

```
pgcc -Minfo=accel -acc -ta=tesla:cc60 -tp=skylake -o m0c_c_acc matmul0.c
main:
    36, Generating implicit copyin(b[:n][:n])
        Generating implicit copy(c[:n][:n])
        Generating implicit copyin(a[:n][:n])
    37, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
        Accelerator serial kernel generated
        Accelerator kernel generated
        Generating Tesla code
        34, #pragma acc loop seq
        35, #pragma acc loop seq
        36, #pragma acc loop seq
    38, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
    39, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
        Loop carried dependence due to exposed use of c[i1][i2] prevents parallelization
```

- copy関係はコンパイラの判断で特に問題はない
- 依存関係があり並列化できない、逐次コードが生成された旨が出力されている
- 正しく実行はできるが、逐次実行のため低速

単純な行列積プログラムの例：loop指示文の追加

- C (matmul1.c)

```
double **a=NULL, **b=NULL, **c=NULL;
// mallocでa,b,cを確保
#pragma acc kernels
#pragma acc loop independent
 for(i=0; i<n; i++){
#pragma acc loop independent
    for(j=0; j<n; j++){
#pragma acc loop seq
      for(k=0; k<n; k++){
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
```

- Fortran (matmul1.f90)

```
double precision, allocatable :: a(:,:), b(:,:), c(:,:)
allocate(a(n,n), b(n,n), c(n,n))
!$acc kernels
!$acc loop independent
  do i=1, n
!$acc loop independent
     do j=1, n
!$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
```

- 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く
- C/C++はポインタ参照の都合で不具合が起きないよう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

コンパイラによる判断の比較

- loop independent指定なし

```
37, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
    Accelerator serial kernel generated
    Accelerator kernel generated
    Generating Tesla code
    34, #pragma acc loop seq
    35, #pragma acc loop seq
    36, #pragma acc loop seq
38, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
39, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
    Loop carried dependence due to exposed use of c[i1][i2] prevents parallelization
```

- loop independent指定あり

```
38, Loop is parallelizable
40, Loop is parallelizable
42, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
    Loop carried dependence of c->-> prevents parallelization
    Loop carried backward dependence of c->-> prevents vectorization
    Accelerator kernel generated
    Generating Tesla code
    38, #pragma acc loop gang /* blockIdx.y */
    40, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    42, #pragma acc loop seq
```

単純な行列積プログラムの例：collapse版

- C (matmul2.c)

```c
double **a=NULL, **b=NULL, **c=NULL;
// mallocでa,b,cを確保
#pragma acc kernels
#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][j] += a[i][k] * b[k][j];
      }
    }
  }
```

- Fortran (matmul2.f90)

```fortran
double precision, allocatable :: a(:,:), b(:,:), c(:,:)
allocate(a(n,n), b(n,n), c(n,n))
!$acc kernels
!$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
```

```
38, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
    39,   /* blockIdx.x threadIdx.x collapsed */
    41, #pragma acc loop seq
39, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
    40,   ! blockidx%x threadidx%x collapsed
    42, !$acc loop seq
```

- collapseを指定すると、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
- 実行時間的にはそれぞれのループを並列化した場合と変わらないことが多い
  - 並列実行形状（後述）を細かく指定する場合などにうまく使い分けると良い

演習：単純な行列積プログラム

- C (matmul1.c)

```c
double **a=NULL, **b=NULL, **c=NULL;
// mallocでa,b,cを確保
#pragma acc kernels
#pragma acc loop independent
 for(i=0; i<n; i++){
#pragma acc loop independent
    for(j=0; j<n; j++){
#pragma acc loop seq
      for(k=0; k<n; k++){
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
```

- Fortran (matmul1.f90)

```fortran
double precision, allocatable :: a(:,:), b(:,:), c(:,:)
allocate(a(n,n), b(n,n), c(n,n))
!$acc kernels
!$acc loop independent
  do i=1, n
!$acc loop independent
     do j=1, n
!$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
```

実際にコンパイルして実行してみよう

- 並列化されたか？
- 実行時間は短くなったか？
- independentやseqを変更してみるとどうか？
- collapse指定を変更するとどうか？

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）を題材として、プログラムのOpenACC化を考えてみる
  - 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
    - 実際の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
    - 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++)

$\alpha_k = \frac{r_k^{\mathrm{T}} p_k}{p_k^{\mathrm{T}} A p_k}$
$x_{k+1} = x_k + \alpha_k p_k$
$r_{k+1} = r_k - \alpha_k A p_k$

if r_{k+1} が十分に小さい then
break

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

結果は x_{k+1}

CG法の計算手順

Compute $r^{(0)} = b-[A]x^{(0)}$	• 初期値、x(0)は適当な値（今回は0ベクトル）
for i= 1, 2, …	• 収束するまで繰り返す
solve $[M]z^{(i-1)} = r^{(i-1)}$	• 前処理、今回はrを対角要素で割るだけ
$\rho_{i-1} = r^{(i-1)}\ z^{(i-1)}$	• リダクション
if i=1	
$p^{(1)} = z^{(0)}$	• コピー
else	
$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$	• リダクション
$p^{(i)} = z^{(i-1)} + \beta_{i-1} z^{(i)}$	• ベクトル積和
endif	
$q^{(i)} = [A]p^{(i)}$	• 行列ベクトル積
$\alpha_i = \rho_{i-1} / p^{(i)} q^{(i)}$	• リダクション
$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$	• ベクトル積和
$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$	• ベクトル積和
check convergence $\|r\|$	• 収束判定（中身はリダクションと平方根）
end	

※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
- α・β・ρ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
    - reduction、dot product
  - 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？

```
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
  dnrm = 0.0;
  for(i=0;i<N;i++){    dnrm += r[i]*r[i];  }
  resid = sqrt(dnrm/bnrm);
  if(resid <= cond){break;}
  if(iter == maxiter){break;}
  rho1 = rho;
}
```

★行列ベクトル積
行ごとの計算を並列に行える

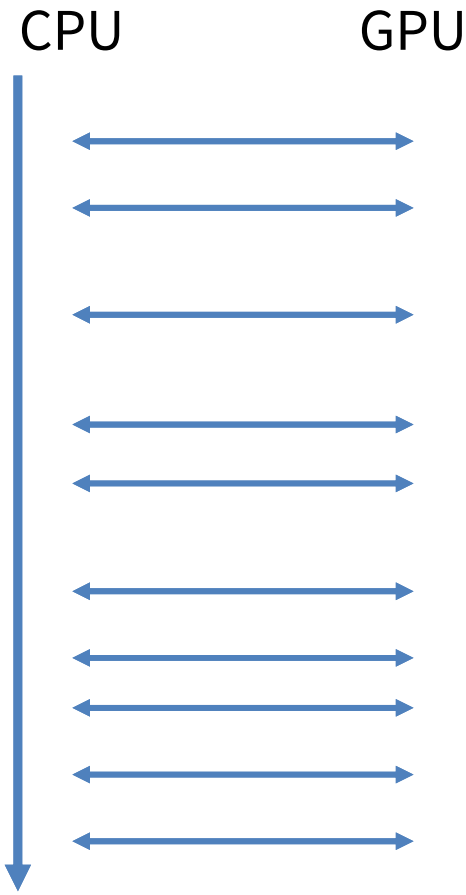
実習：CG法のOpenACC化： 1.各計算部の並列化

- 並列化できることがわかっているループに指示文を挿入
  - サンプルコードcg2.cおよびcg2.f90に指示文を挿入する
    - Fortran版ではコンパイル時に-cppオプションを加える必要あり
      - 出力部の調整のために #if を使っているため
  - はじめはループ1つだけに指示文を挿入して実行してみよう
    - step by stepで少しずつ並列化できるのはOpenACCの強みの一つ
- 注意点（コンパイラのメッセージも参考に）
  - 特にCの場合はindependent節も活用する必要あり
    - 主に、配列の参照先アドレスが重複する可能性を考慮するためか
  - （既に例示したように）配列長が認識できずにデータ転送に躓くことがあるので適切にcopy指示節を追加する
    - C言語版の行列Aに要注意
    - もちろん、in/outを意識してcopyin/copyoutとしても良い
  - reductionが適切に生成されているかを確認
    - （問題なく生成されると思って良いが、念のため）

（CG法のOpenACC化）

- 正しく計算できているはずだが、調べて見るとデータ転送が多い
  - PGI_ACC_TIMEなどで確認できる
- 現在のデータ転送状況イメージ

```
Compute r (0) = b-[A]x (0)
for i= 1, 2, …
  solve [M]z (i-1) = r (i-1)
  ρ i-1 = r (i-1)  z (i-1)
  if i=1
    p (1) = z (0)
  else
    β i-1 = ρ i-1 / ρ i-2
    p (i) = z (i-1)  + β i-1 z (i)
  endif
  q (i) = [A]p (i)
  α i = ρ i-1 /p (i) q (i)
  x (i) = x (i-1)  + α i p (i)
  r (i) = r (i-1)  - α 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？

実習：CG法のOpenACC化： 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 $[M]z^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)} \ z^{(i-1)}$
  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
  $q^{(i)} = [A]p^{(i)}$
  $\alpha_i = \rho_{i-1} / p^{(i)} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r(i) = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence |r|
end

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

  - 余裕がある人は、update節を用いて途中の配列データを覗いてみよう
    - デバッグの際などに役立つことがある

（CG法のOpenACC化）

- 実は最初に全てのデータを転送し、結果ベクトルのみ回収すれば良かった
  - もちろん、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い
  - 実際に実行時間レベルで意味がある（GPUを使うことでCPUより大幅な高速化が得られる）のは大規模な行列の場合
    - コードを書き換えたり実行時に引数を与えれば問題サイズを変更できるようになっている
      - データを出力している部分をなんとかしないと（コメントアウトなど）大量のデータが書き出されてしまうため注意
    - 問題が簡単過ぎてすぐに反復計算が終わってしまうため、もっと長時間実行したい場合は行列をいじったり収束条件を厳しくしたりする必要がある

関数の呼び出し

- kernels/parallel内部（OpenACC並列化対象内部＝GPU上）で関数を実行する場合はroutine指示文が必要

```c
// プロトタイプ宣言にも指示文が必要
#pragma acc routine
void calc(int n, double *v);

// 呼び出し元
#pragma acc data copy(v1)
  {
#pragma acc kernels
    {
      calc(n, v1);
    }
  }
```

```c
// 呼び出し対象の関数
#pragma acc routine
void calc(int n, double *v)
{
  int i;
  for(i=0; i<n; i++){
    v[i] = v[i] * 2.0;
  }
}
```

```fortran
! 呼び出し対象の関数
module mod
contains
subroutine calc(n,v)
!$acc routine
  integer :: n
  double precision :: v(*)
!$acc loop
  do i=1, n
     v(i) = v(i) * 2.0d0
  enddo
end subroutine calc
end module mod
```

```fortran
! 呼び出し元
!$acc data copy(v1)
!$acc kernels
  call calc(n,v1)
!$acc end kernels
!$acc end data
```

- Cでは関数名の前に、Fortranでは関数名の次に指示文を挿入
- 関数内でも並列計算を行わせるにはroutineの後にさらにgangなどの並列実行形状の指定（後述）も必要

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？
  - WARP長(32)にあわせたループ構造にしたが、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ようだ

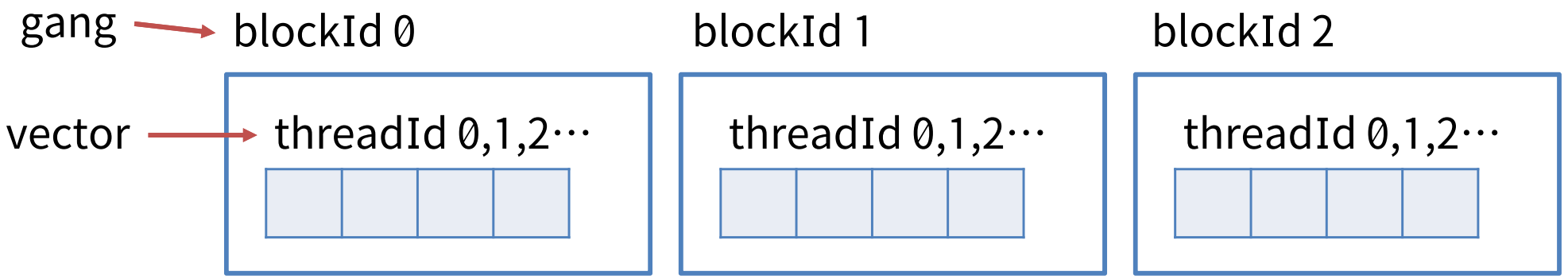
```
pgcc -Minfo=accel -acc -ta=tesla:cc60 -tp=skylake -o vector0_c_acc vector0.c
         17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
pgfortran -Minfo=accel -acc -ta=tesla:cc60 -tp=skylake -o vector0_f_acc vector0.f90
         14, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

gang, worker, vectorとblockIdx, threadIdx

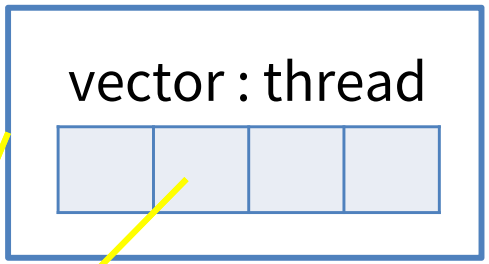
- OpenACCではハードウェアに階層的な並列性があることを想定しており、上位階層から順にgang, worker, vectorとなっている
- これらをどのように組み合わせて実行するかを指定できる

- 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、という方が正しい
  - grid, threadblock, threadの三階層構造

  - OpenACCのおおまかな並列実行モデルの対応付け

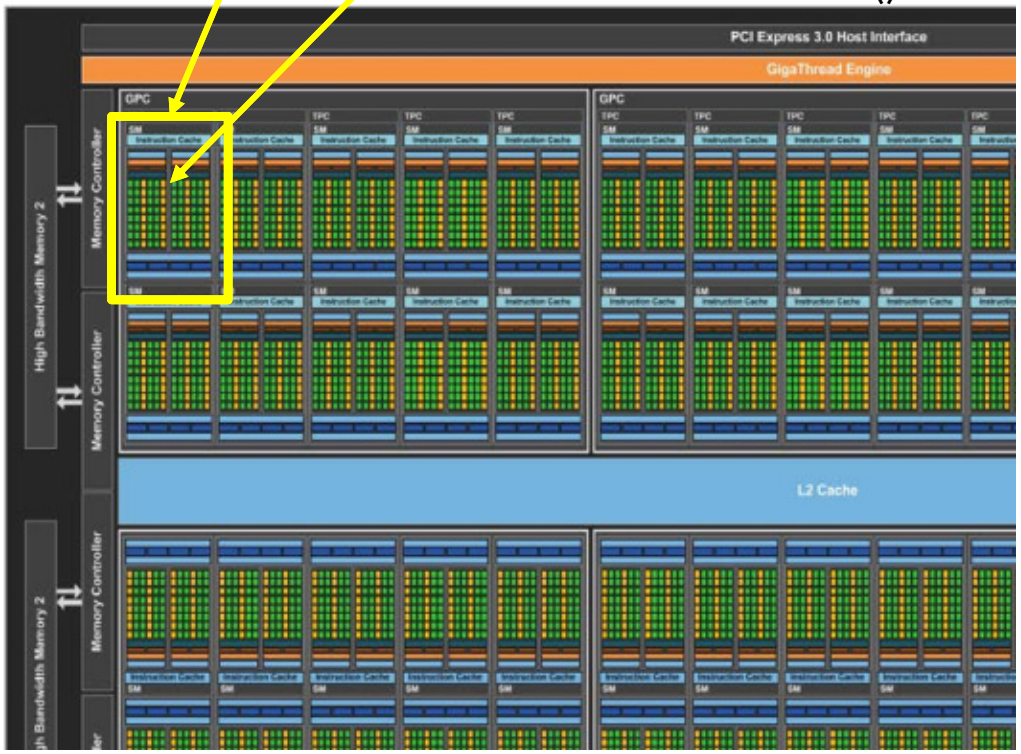


ハードウェアとのおおまかな対応付け



- Streaming Multiprocessor(SM)内の並列性はvector、SM単位の並列性はgang
- 基本的には
  - 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



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（CUDAではSM単位）
- workerはHWレベルで同期できる単位の細粒度並列性（CUDAではSM内のWARP群）
- vectorはworker内部でのSIMDやベクトル並列処理（CUDAではWARP）
- 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

最適化のためのヒント

- GPUの特徴にあわせた適切な実装を行うことで性能が向上する
- 特に以下の点は性能への影響が大きいため気を付けたい
  - 並列度
    - 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
  - 連続メモリアクセス
    - vectorループでは連続メモリアクセスを心がける
      - GPUの得意なメモリアクセス方式となる

```
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
○   a[i][j] = b[i][j] + c[i][j];
×   a[j][i] = b[j][i] + c[j][i];
  }
}
```

```
do i=1, N
  do j=1, N
×   a(i,j) = b(i,j) + c(i,j);
○   a(j,i) = b(j,i) + c(j,i);
  }
}
```

その他、今回は触れなかった情報

- ややアドバンスドな話であり今回は扱いませんでしたが、今後の講習会では扱うかも知れません
  - デバッガ、プロファイラ
    - PGIコンパイラ付属のpgdbgやpgprof、NVIDIA社の提供するnvvpやnvprof、その他サードパーティ製のソフトの幾つかが利用可能
    - GUIが表示されるものはX転送が必要
  - CUDAやMPIとの連携
    - OpenACCと外部とでデータ（ポインタ）をやりとりする方法を提供
    - host_data, use_device, deviptrなどを活用
  - CUDA Unified Memory
    - CPUとGPUが連続したメモリアドレス空間を利用する技術
    - データ転送を記述しなくても必要に応じてCPU-GPU間のデータ転送が勝手に行われるため、プログラミングが容易になる
    - 幾つかの条件（制限）があり、性能にもペナルティが生じる
    - コンパイル時に-ta=tesla:managedオプションをつけるだけで良い