2020年10月07日（水）10:00-17:30
第3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）

第3回スーパーコンピュータ「不老」利用型講習会 OpenACC（初級）

名古屋大学 NAGOYA UNIVERSITY　ITC　不老

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」へのログイン
  - 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

- 13:30 – 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）」

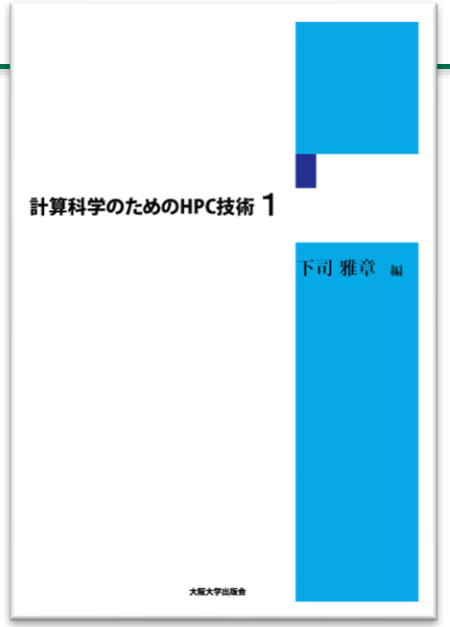
イントロダクション、端末設定など

- 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」へのログイン
- 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

- 別紙にて紹介

（日本語で書かれた）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
- 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



サンプルプログラム（ソースコード）について

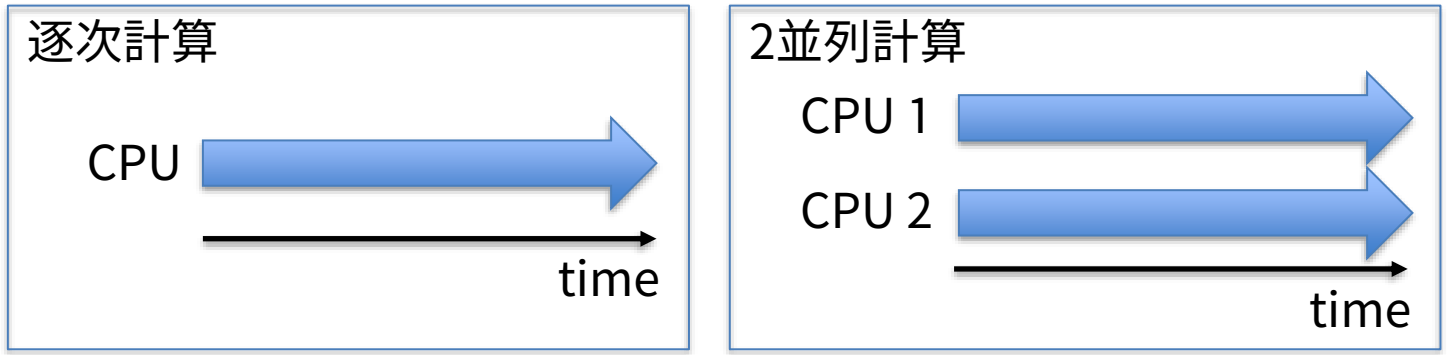
- 資料中で用いているソースコードは /home/center/a49979a/share/20201007.tgz にて公開している
- cpでコピーして tar zxvf 20201007.tgz で展開して使ってください
- 見た目の都合等から公開コードと資料中のコードが完全一致しているわけではない点に注意

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
      - 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

MacBookProの公式サイトの例。モデルごとにCPUのコア数が明確に示されている。正確には「コア数＝性能」ではないのだが、重要な性能の目安ではある。
画像引用元：https://www.apple.com/jp/macbook-pro/
（2020.06.05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	おおよその理論演算性能（FP64）	理論メモリバンド幅	備考
SPARC64VIIIfx	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	名大旧システムFX100に搭載
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、藤井聡太二冠の自作PCで話題に

- 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

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	動作周波数	理論演算性能 (FP64)	理論メモリバンド幅	製品発表時期など
Tesla P100	3,584 (64*56)	1,328 ~ 1,480 MHz	5.3 TFLOPS	720 GB/s	2016.04 製品発表
Tesla V100	5,120 (64*80)	~ 1,530 MHz	7.8 TFLOPS	900 GB/s	2017.06 製品発表 「不老」Type IIに搭載
NVIDIA A100	6,912 (64*108)	~ 1,410 MHz	9.7 TFLOPS	1,555 GB/s	2020.05 製品発表 普及はこれから

- 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

➢ 製品発表時期はイベント等で正式に発表した年月。提供開始時期については、一部ユーザへの先行提供などがあるためよくわからないことが多い。

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 A100）	Fire Stream Radeon 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 2020年6月版（最新版）

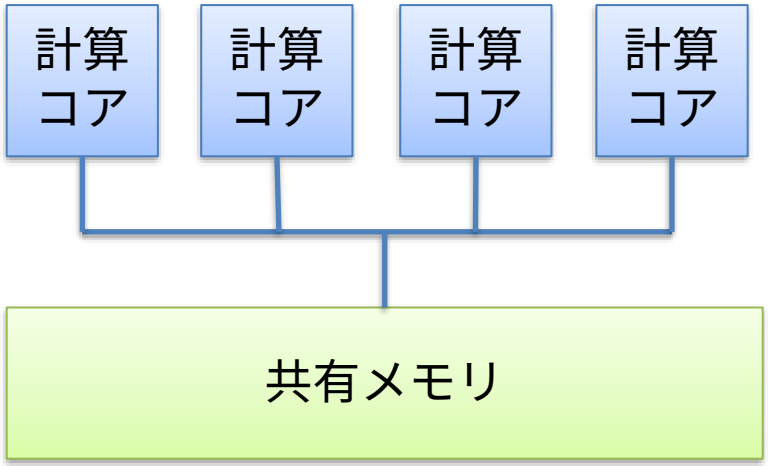
- 理研RCCSの「富岳」が2位以下を大きく引き離して1位で初登場
  - 日本の1位は2011年11月の「京」以来
  - 「富岳」はTOP500、HPCG、Graph500、HPL-AIの4つのベンチマークで世界一を達成
- 上位10システム中4システムが新システム（☆）
- 上位10システム中6システムがNVIDIA GPU搭載
  - 7位のSeleneのA100は5月に公式発表された最新GPU（NVIDIA A100）
  - TOP100中34、全500中141
- アメリカと中国のシステムが多い
- 上位システムは総コア数が100万を超える
- 消費電力も数十MW級
  - 10MWで一般家庭300世帯程度

画像引用元：https://www.top500.org/

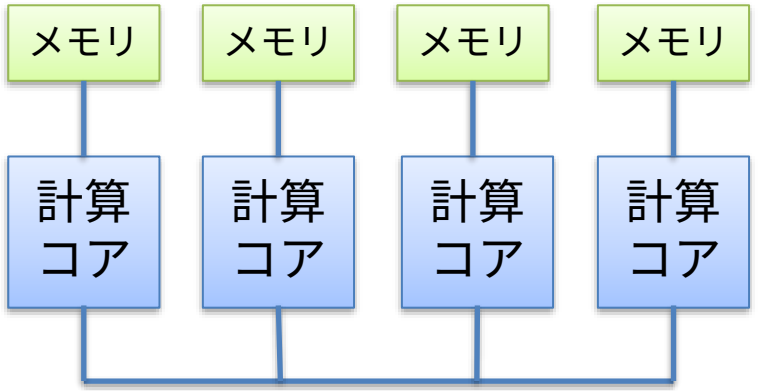
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1 ☆	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
6 ☆	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
7 ☆	**Selene** - DGX A100 SuperPOD, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	277,760	27,580.0	34,568.6	1,344
8	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
9 ☆	**Marconi-100** - IBM Power System AC922, IBM POWER9 16C 3GHz, Nvidia Volta V100, Dual-rail Mellanox EDR Infiniband, IBM CINECA Italy	347,776	21,640.0	29,354.0	1,476
10	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100, Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384

並列計算機の分類

- 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
  - 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];
      }
```

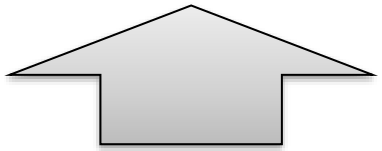
```
PE2   for(i=0; i<N; i++){
        D[i] = E[i] + F[i];
      }
```

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];
      }
```



- 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
```

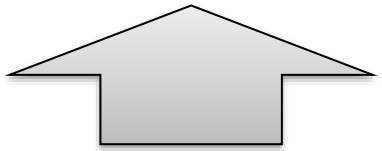
```
PE2   do i=1, N
        D(i) = E(i) + F(i)
      end do
```

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

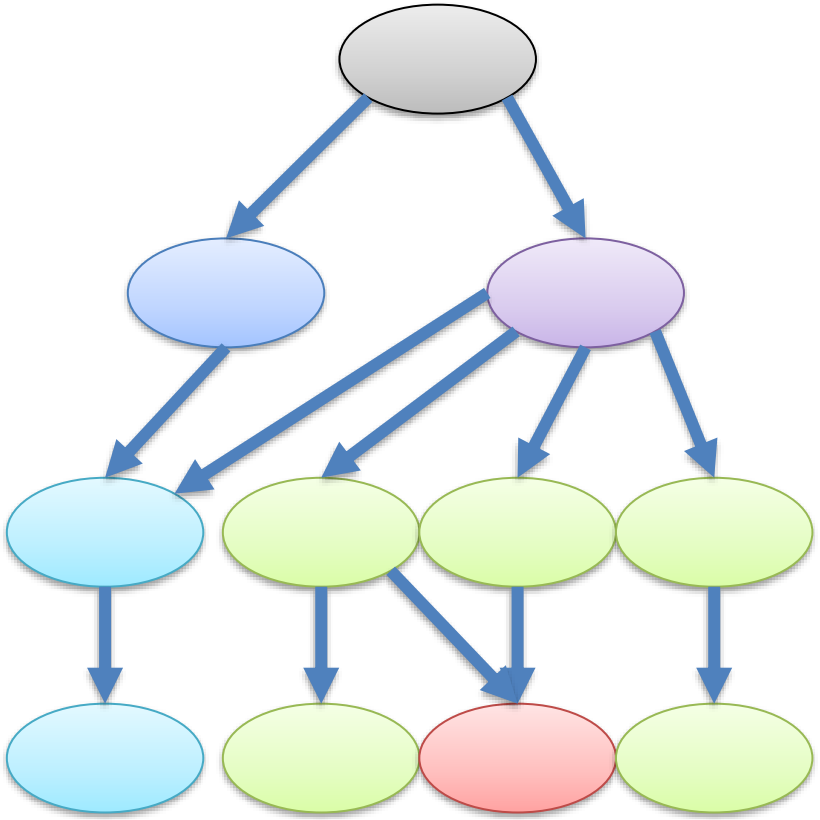
```
PE2   do i=N/2+1, N
        A(i) = B(i) + C(i)
        D(i) = E(i) + F(i)
      end do
```



- 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



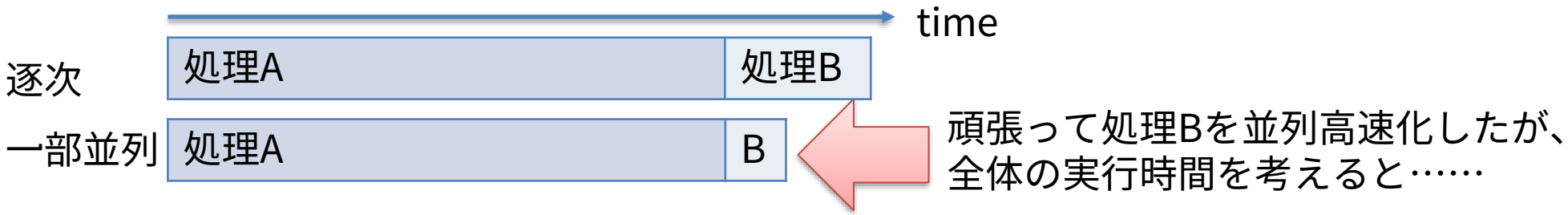
並列計算の限界

- 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

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

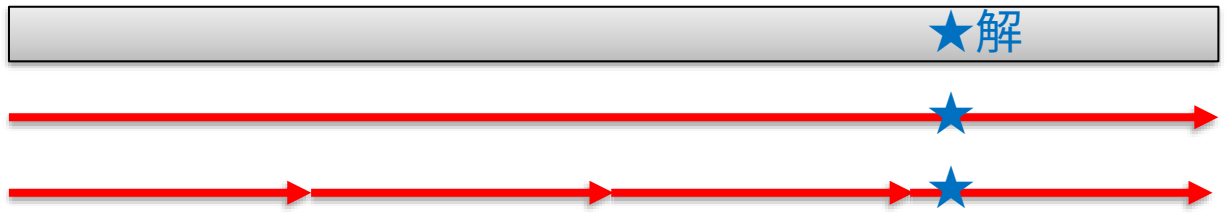
例：ループイタレーション間にも、
同じループイタレーション内にも、
依存関係がある ⇒ 並列化が困難

- （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



頑張って処理Bを並列高速化したが、
全体の実行時間を考えると……

  - 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

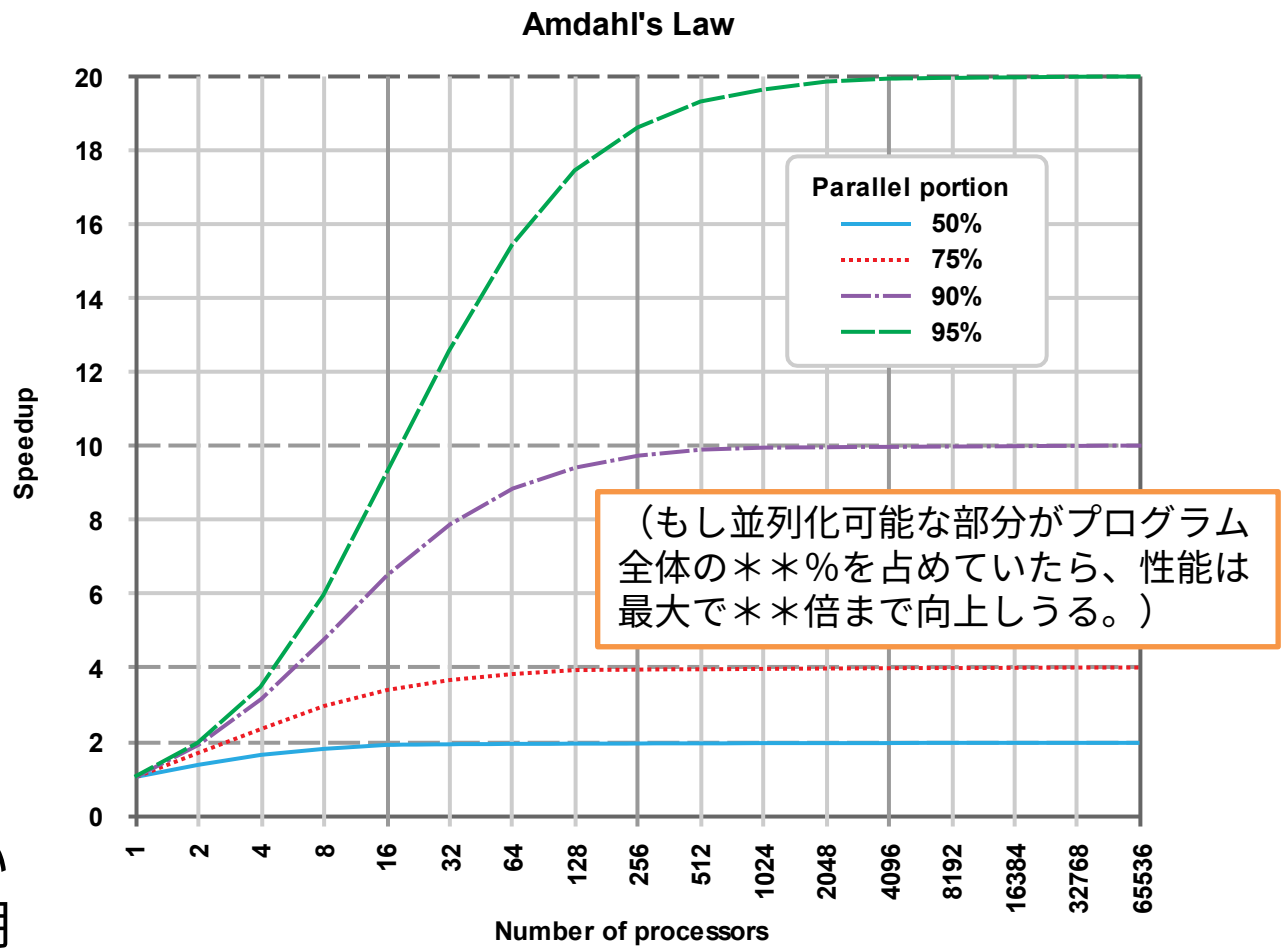


プログラムのどこを並列化させるか、GPUに担当させるかをしっかり考える必要がある

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

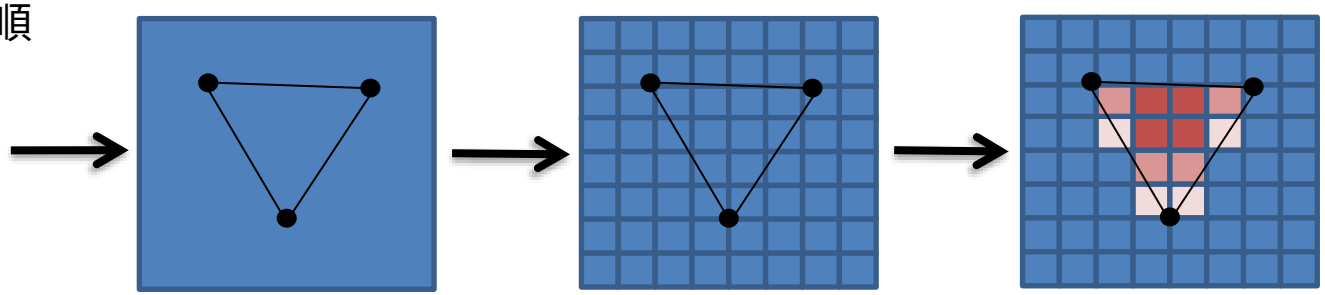
- 何故GPUを活用する必要があるのか？
  - GPUは非常に高い性能をもつハードウェアであり、うまく活用できれば大変強力な研究の道具となる：成果を得るための加速装置
  - GPUは国内外の様々な計算環境に導入されているため、利用スキルを得ておくことはきっとプラスになる
    - 名古屋大学 情報基盤センターでもType IIサブシステムにGPUを搭載
- どうやって使う？
  1. 並列化されたソフトウェア（アプリケーション）を使う
  2. （並列化されていないプログラムから）並列化されたライブラリやフレームワークを使う
  3. 自分で並列計算を実装する
     - 自分が扱いたい任意の問題（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)

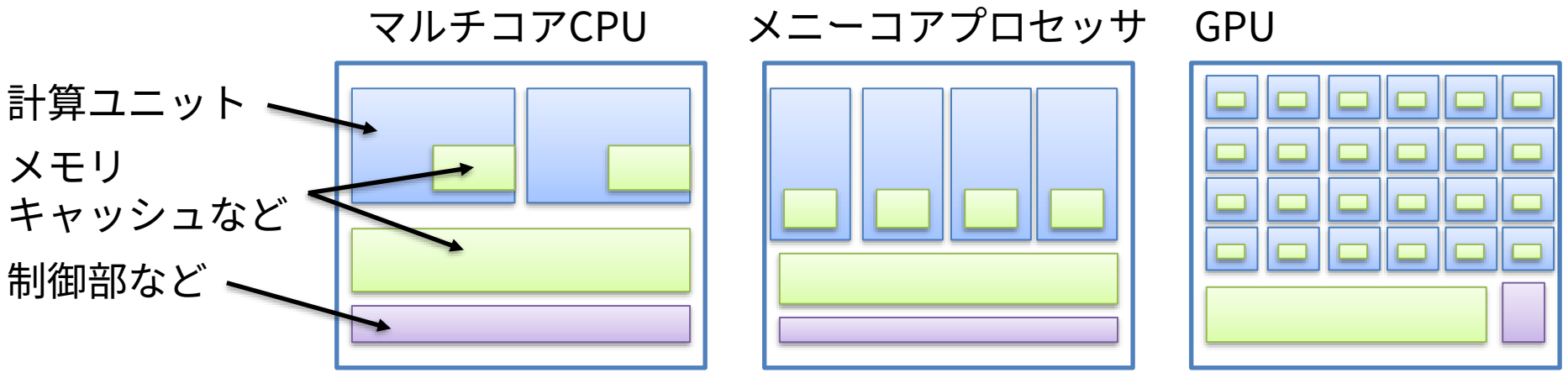


オブジェクト単位、頂点単位、ピクセル単位で同時（並列）処理が可能

  - 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）

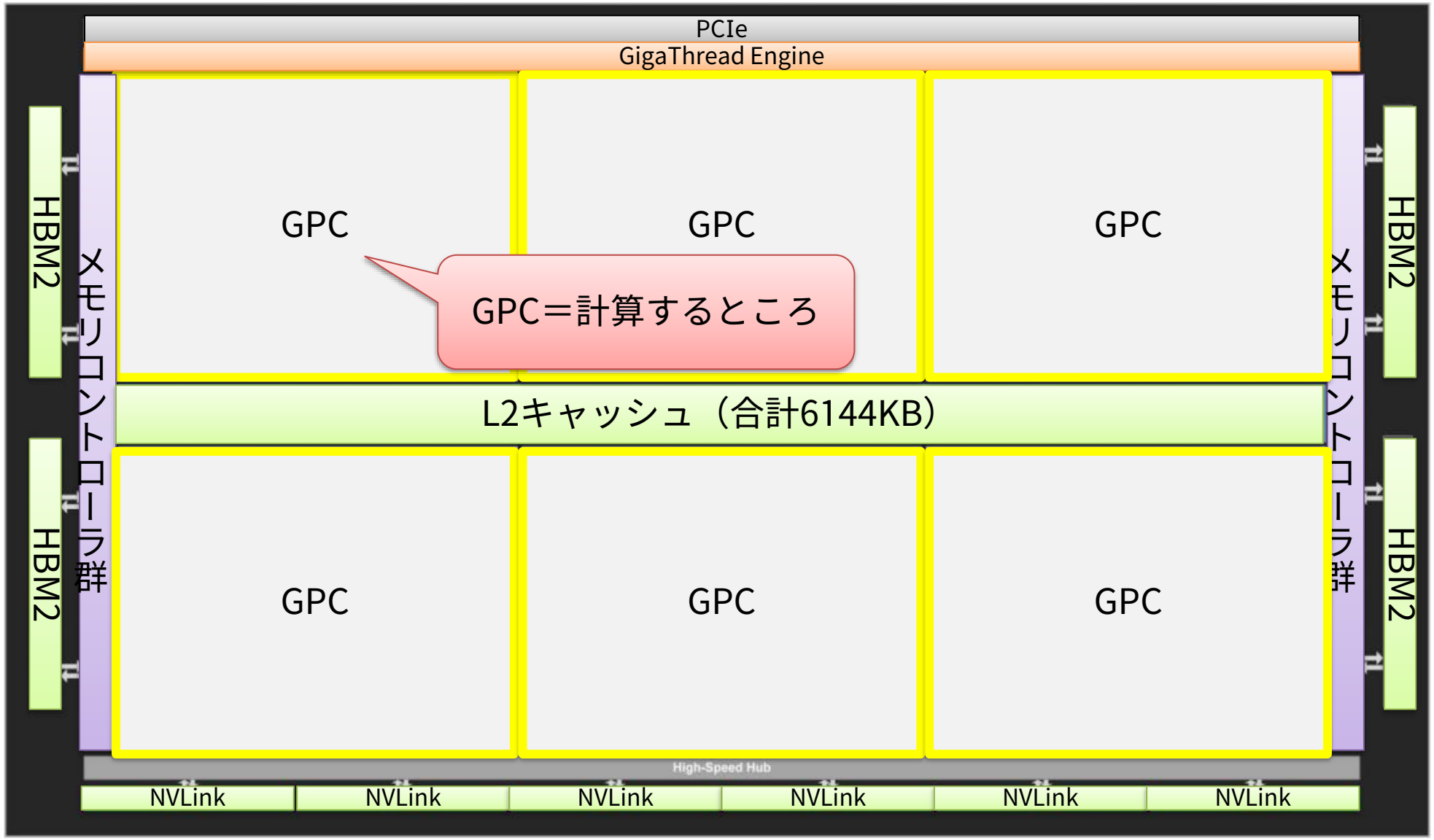
- GPUとはどのようなハードウェアなのだろうか？
  - （既に述べたように）CPUとはバランスが違う、たくさんの計算コアと高速なメモリを搭載

- 具体的な例（Tesla V100の例）
  - 以下、NVIDIA TESLA V100 GPU ARCHITECTUREより引用
  - https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
    - 日本語版の資料はこちら
    - https://images.nvidia.com/content/pdf/tesla/Volta-Architecture-Whitepaper-v1.1-jp.pdf

全体構成

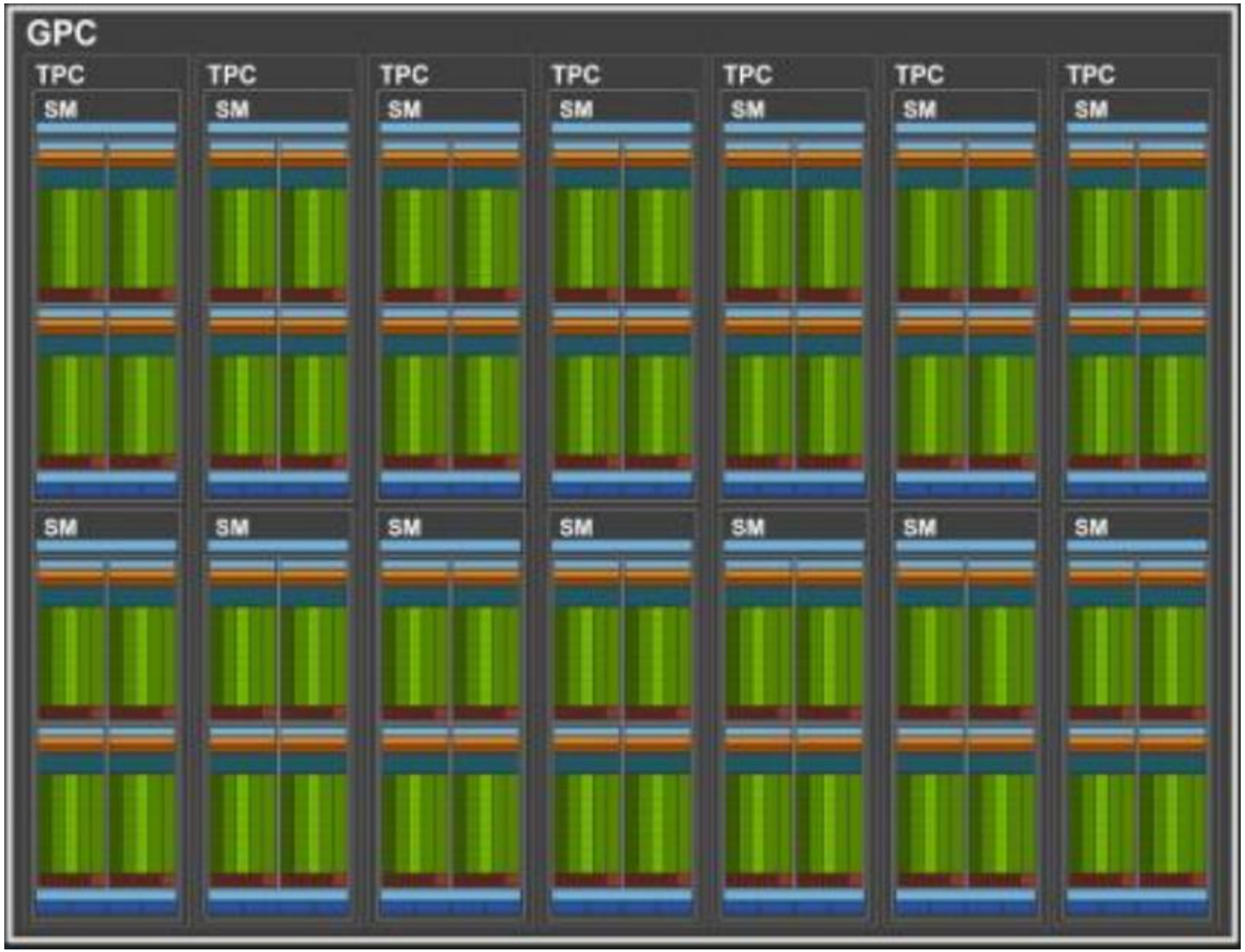

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

全体構成


PCIe
GigaThread Engine
HBM2
メモリコントローラ群
GPC
GPC
GPC
GPC＝計算するところ
L2キャッシュ（合計6144KB）
GPC
GPC
GPC
High-Speed Hub
NVLink
NVLink
NVLink
NVLink
NVLink
NVLink

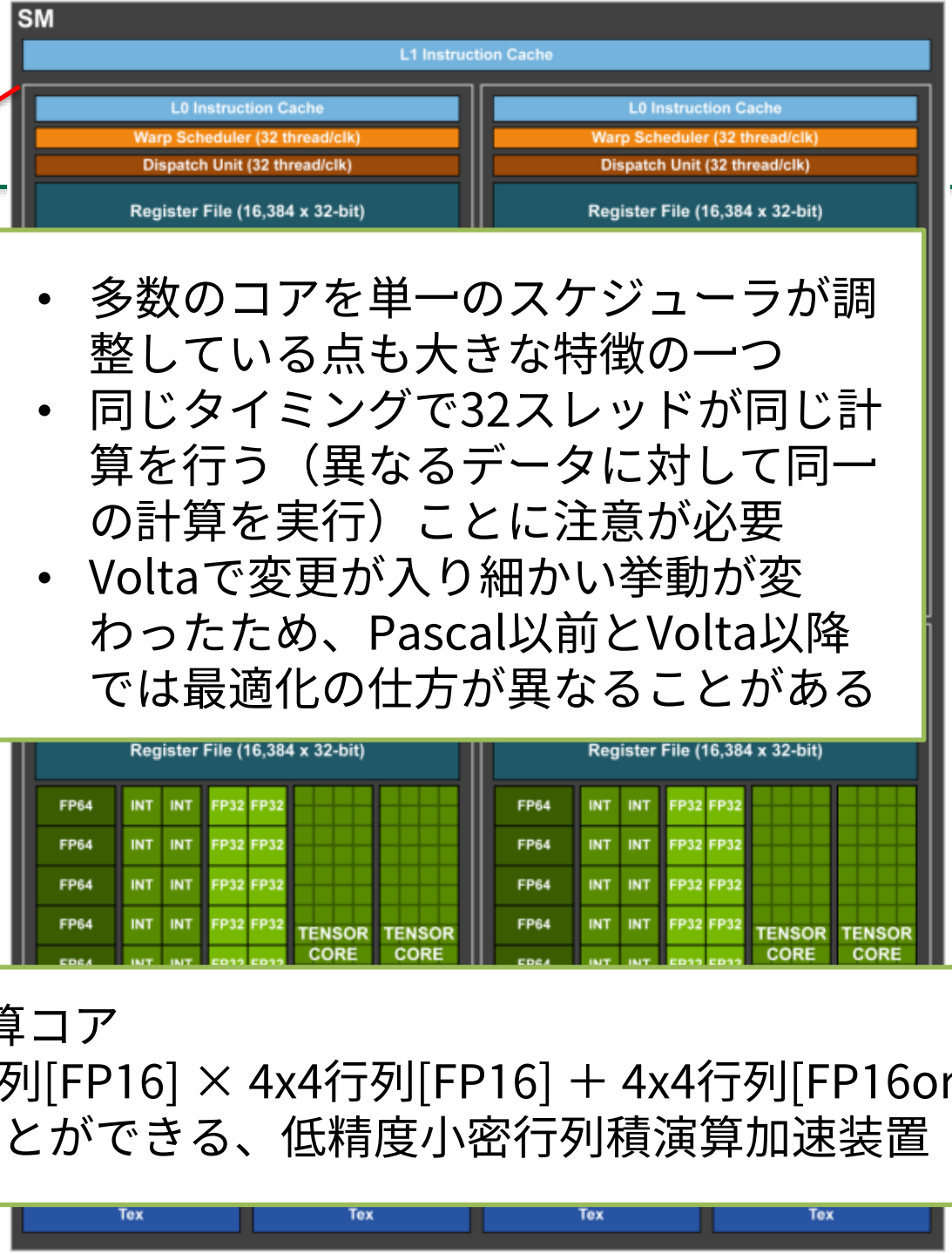
- GPC: GPU Processing Cluster
- TPC: Texture Processing Cluster
- SM: Streaming Multiprocessor

GPCの中身



- 1GPUに6GPC搭載
- 1GPCに7TPCs搭載、各TPCに2SMs搭載
- 1SMに……
  - 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、合計で……
  - 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

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

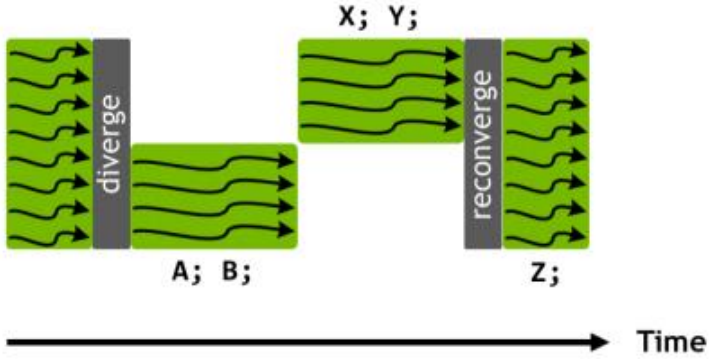
- Voltaで新しく搭載された計算コア
- 4x4行列[FP16or32] ＝ 4x4行列[FP16] × 4x4行列[FP16] ＋ 4x4行列[FP16or32]という計算だけを高速に行うことができる、低精度小密行列積演算加速装置

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

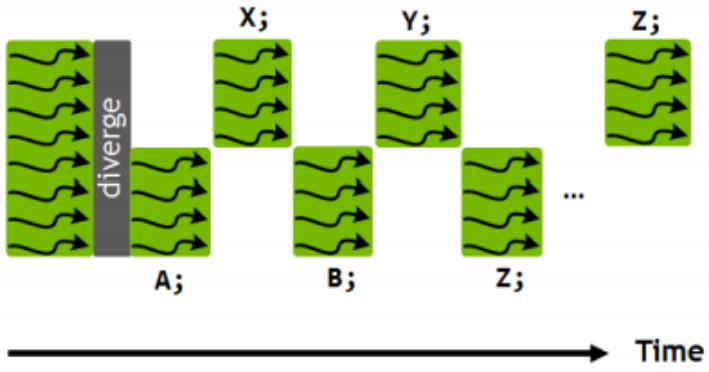
- NVIDIAの示している例
  - PascalまでのSIMTの例

```
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 = 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
  - 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

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良い
    - 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、まだ10年経っていない

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/
- 主な対応コンパイラ
  - 商用：PGI（無償版あり）→NVIDIA HPC SDK、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/

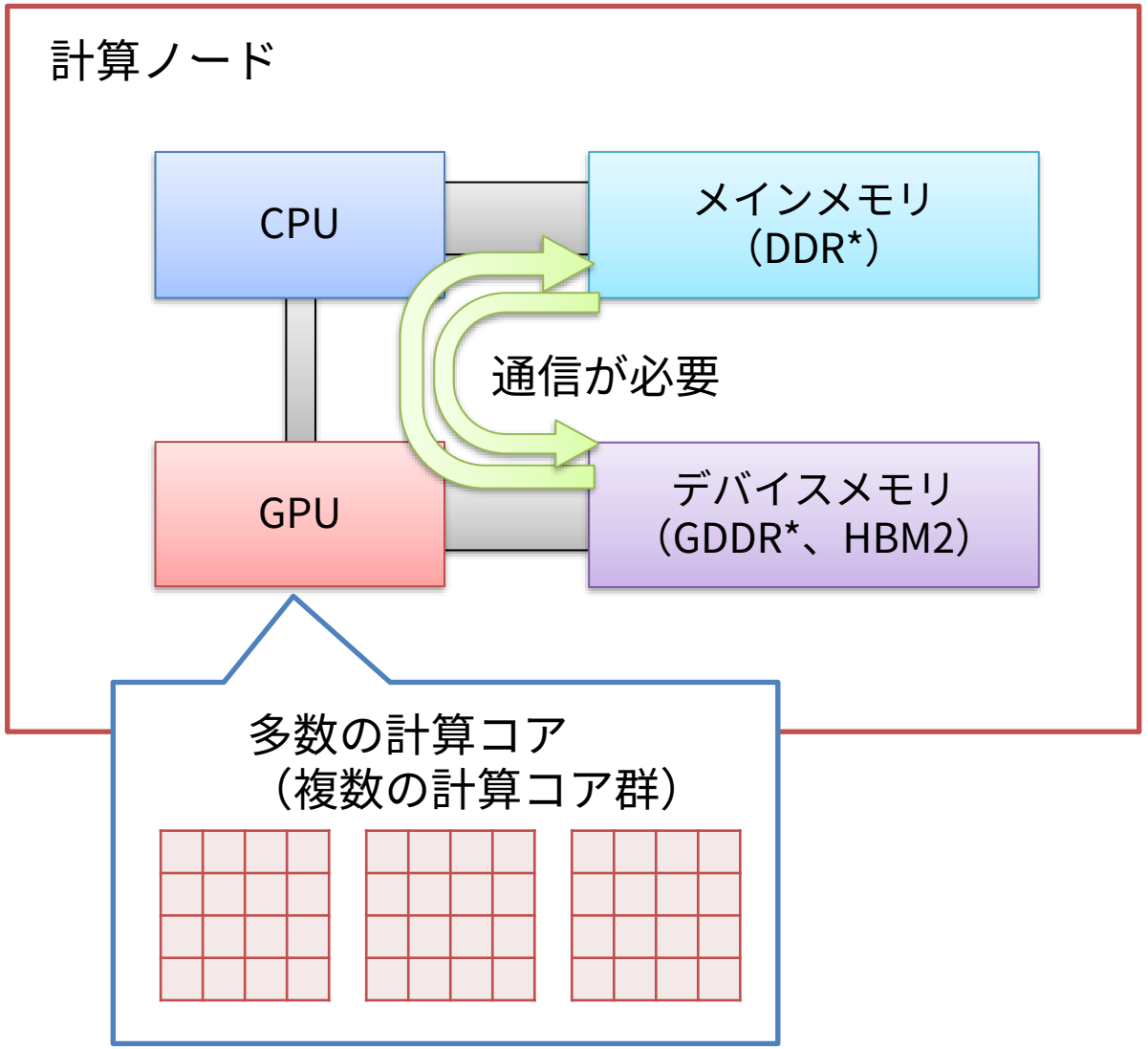
PGIコンパイラとHPC SDK

- 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。
- そのため、最新のOpenACCコンパイラとしてはHPC SDKを使うのが妥当だが、「不老」にはPGIコンパイラがインストールされているため、本講習会ではこれを利用する。
  - まだ両者にはバージョンに大きな差がないため、ほぼ同じように使えるはずである

- NVIDIA HPC SDKは無償で利用できるため、各自でダウンロードして「不老」上で利用しても良い
  - Web上の情報を探すとPGIコンパイラ時代の情報がたくさん発見されるが、基本的にはその情報に従って使えばよい
  - もちろん、更新されて変わってしまっている挙動などもあると思うので確認は必要

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PGIコンパイラの場合は特別なプログラムを介したりする必要はない
     - LD_LIBRARY_PATHなど環境変数の対応は必要
       - 提供されているmodulefileで簡単に設定できる
     - 想定されているGPUが利用できないと実行時エラー
       - pgaccelinfo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

- pgaccelinfo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　→

```
$ pgaccelinfo

CUDA Driver Version:           10020
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  440.64.00  Wed Feb 26
16:26:08 UTC 2020

Device Number:                 0
Device Name:                   Tesla V100-PCIE-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
PGI Default Target:            -ta=tesla:cc70

Device Number:                 1
```

1GPU分の情報

OpenACCコンパイル時に
指定する情報

ここから下にもう1つのGPUの分の情報

単純なOpenACCプログラムの例（配列の定数倍計算）

- C (vector1.c)

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;
  int 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;
}
```

OpenACC並列化対象
＝GPU上で実行される

- Fortran (vector1.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
```

- 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 }
```

CPU

GPU

CPU

- 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

「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 list`	load済みモジュールの一覧を表示
`module load <modulename>`	対象モジュールのload
`module avail`	利用可能なモジュールの一覧を表示
`module unload <modulename>`	対象モジュールのunload
`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
- pgi/20.4をloadするとPGIコンパイラが利用可能になる
  - `module load pgi/20.4`

コンパイル例

- -ta
  - Tesla P100ならtesla:cc60、V100ならtesla:cc70
  - pgaccelinfoの「PGI Default Target」に対応
- -tp
  - skylakeなど
- 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。

- pgccまたはpgfortran（pgf90）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
  - -ta　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

- コンパイル例

※出力情報の具体的な読み方は後述する

```
$ pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -o v1c_c_acc vector1.c
main:
     16, Loop is parallelizable
         Generating Tesla code
         16, #pragma acc loop gang, vector(32) /* bl
     16, Generating implicit copyout(v2[:]) [if not al
         Generating implicit copyin(v1[:]) [if not already present]

$ pgfortran -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -o v1f_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
```

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る
※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 pgi/20.4
export PGI_ACC_TIME=1
./v1c_c_acc
```

job_vector1.sh として作成→ pjsub job_vector1.sh でジョブ投入

← 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

必須ではないがなるべく意識して書く

← module loadでPGIコンパイラ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

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

- 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

バッチジョブ実行の例（pjsubによるジョブ実行とpjstatによる確認）

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
※見やすさの都合でコマンド実行ごとに改行していますが、本来は改行されません

```
bash@flow-cx03 vector $ pjsub job_vector1.sh
[INFO] PJM 0000 pjsub Job 29546 submitted.
```

←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）

```
bash@flow-cx03 vector $ pjstat
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM          NODE_REQUIRE    VNODE  CORE V_MEM
29546      job_vector NM RNA a49979a  (09/22 11:03)   0001:00:00          1               -      -    -
```

←pjstatでジョブの状況を確認

↑RNAは実行開始直前の状態

```
bash@flow-cx03 vector $ pjstat
JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM          NODE_REQUIRE    VNODE  CORE V_MEM
29546      job_vector NM RUN a49979a  09/22 11:03:13  0001:00:00          1               -      -    -
```

↑RUNは実行中、この他に待ち状態のQUEや終了処理中のRNEなどがある

```
bash@flow-cx03 vector $ pjstat
bash@flow-cx03 vector $
```

←終了済のジョブだけになると何も出力されない

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

```
$ ./ v1c_c_acc
 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00                    ←元データ
 2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00               ←計算結果
```

Fortran版の場合

```
$ ./v1f_f_acc
   1.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00  10.00     ←元データ
   2.00   4.00   6.00   8.00  10.00  12.00  14.00  16.00  18.00  20.00     ←計算結果
```

GPUが仕事をしていることを確認する　1/2

- PGIコンパイラの場合、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 PGI_ACC_TIME
  - export PGI_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
  - ログインノード上での実行例 →

```
bash@flow-cx03 vector $ export PGI_ACC_TIME=1
bash@flow-cx03 vector $ ./v1c_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

Accelerator Kernel Timing data
/home/center/a49979a/share/20201007/vector/vector1.c
  main  NVIDIA  devicenum=0
    time(us): 34
    16: compute region reached 1 time
        16: kernel launched 1 time
            grid: [1]  block: [32]
             device time(us): total=4 max=4 min=4 avg=4
            elapsed time(us): total=6,234 max=6,234 min=6,234 avg=6,234
    16: data region reached 2 times
        16: data copyin transfers: 1
             device time(us): total=15 max=15 min=15 avg=15
        18: data copyout transfers: 1
             device time(us): total=15 max=15 min=15 avg=15
```

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

- 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

PGI_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 queue=0
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
```

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
- 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PGI_ACC_TIMEやPGI_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合を除いてはバッチジョブとして実行すること

OpenACCプログラムの構成

- 指示文：directive
- 指示節（指示句）：clause

この資料中ではまとめて指示文と呼ぶことにする

- 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)

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;
  int 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 (vector1.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(*,*)""
```

（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

```
pgcc -Minfo=accel -acc -ta=tesla:cc70 以下省略
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]
```

- Fortran

```
pgfortran -Minfo=accel -acc -ta=tesla:cc70 以下省略
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
```

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

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
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  }
```

- C (vector2.c)

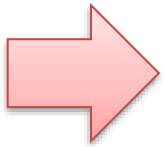
```
 3  int main(int argc, char **argv)
 4  {
 5    int i;
 6    int n=10;
 7    double v1[10], v2[10];
 8
 9    for(i=0; i<n; i++){
10      v1[i] = (double)(i+1);
11      v2[i] = 0.0;
12    }
13
14  #pragma acc kernels copyin(v1[:]) copyout(v2[:])
15  #pragma acc loop gang, vector(32)
16    for(i=0; 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");
```

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
12 !$acc loop kernels
13   do i=1, n
14      v2(i) = v1(i) * 2.0d0
15   enddo
16 !$acc end kernels
17
18   do i=1,n
19      write(*,'(1H F8.2)',advance="NO")v1(i)
20   enddo
21   write(*,*)""
22   do i=1,n
23      write(*,'(1H F8.2)',ad
24   enddo
25   write(*,*)""
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(*,*)""
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

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

```
#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

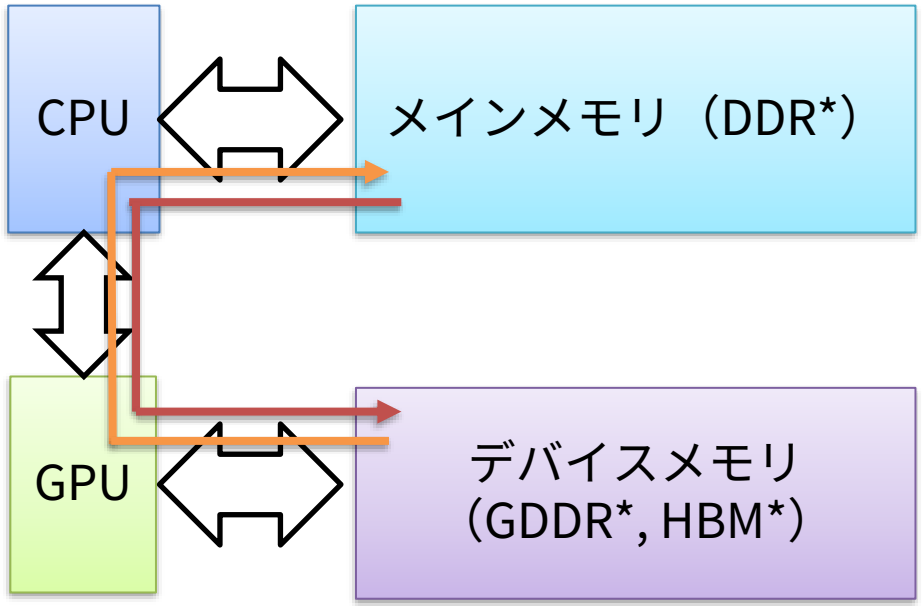
```
!$acc kernels copyin(v1(:)) copyout(v2(:))
!$acc loop gang, vector(32)
  do i=1, n
     v2(i) = v1(i) * 2.0d0
  enddo
!$acc end kernels
```

```
!$acc kernels &
!$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

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

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）

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;
    }
  }
```

(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

  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？

```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;
    }
  }
```

(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

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

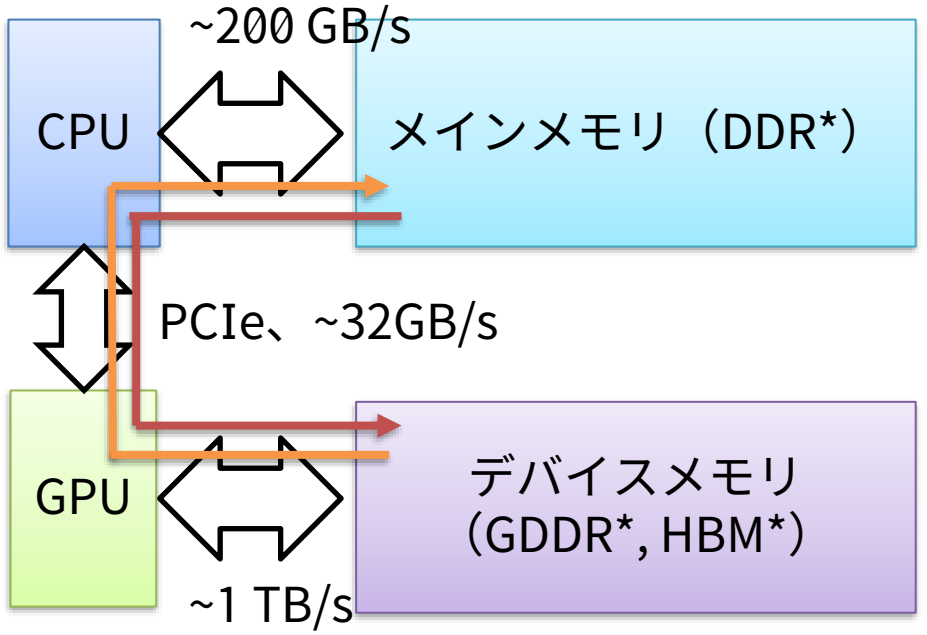
(vector3.f90)

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

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
    - DDR4、100GB/s/socket程度（STREAM Triad実測）
  - 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
    - PCI Express Gen.3 x16
      - 理論性能でも最大16GB/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数によってつなぎ方はかわる
      - Power系CPUではCPU-GPU間でもNVLinkが使える



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)

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

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

!$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？

data指示文によるデバイスメモリの生存期間の最適化

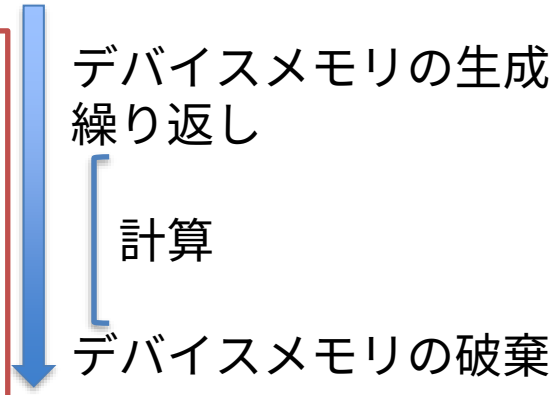
- GPUカーネルを何度も実行する場合などにdata指示文が有効

```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);
  }
```

```c
#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;
    }
  }
```

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

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

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



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
```

```
!$acc data copy(v1(:))
  do j=1, 10
!$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, vector3.f90を元に、data指示文を挿入したプログラムvector4.c, vector4.f90を作成し、実行時間を比較する
  - 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

よく見ると、実行回数も実行時間も大きく異なることが分かるはずである

実習：実行例

```
Accelerator Kernel Timing data
/home/center/a49979a/share/20201007/vector/vector3.c
  main  NVIDIA  devicenum=0
    time(us): 181
    15: compute region reached 10 times
        15: kernel launched 10 times
            grid: [1]  block: [32]
             device time(us): total=31 max=4 min=3 avg=3
            elapsed time(us): total=721 max=510 min=21 avg=72
    15: data region reached 20 times
        15: data copyin transfers: 10
             device time(us): total=70 max=16 min=5 avg=7
        17: data copyout transfers: 10
             device time(us): total=80 max=21 min=6 avg=8
```

```
Accelerator Kernel Timing data
/home/center/a49979a/share/20201007_bak/vector/vector4.c
  main  NVIDIA  devicenum=0
    time(us): 66
    13: data region reached 2 times
        13: data copyin transfers: 1
             device time(us): total=15 max=15 min=15 avg=15
        18: data copyout transfers: 1
             device time(us): total=20 max=20 min=20 avg=20
    15: compute region reached 10 times
        15: kernel launched 10 times
            grid: [1]  block: [32]
             device time(us): total=31 max=4 min=3 avg=3
            elapsed time(us): total=684 max=494 min=20 avg=68
```

```
Accelerator Kernel Timing data
/home/center/a49979a/share/20201007/vector/vector3.f90
  main  NVIDIA  devicenum=0
    time(us): 175
    12: compute region reached 10 times
        13: kernel launched 10 times
            grid: [1]  block: [32]
             device time(us): total=31 max=4 min=3 avg=3
            elapsed time(us): total=686 max=481 min=22 avg=68
    12: data region reached 20 times
        12: data copyin transfers: 10
             device time(us): total=66 max=15 min=5 avg=6
        16: data copyout transfers: 10
             device time(us): total=78 max=21 min=6 avg=7
```

```
Accelerator Kernel Timing data
/home/center/a49979a/share/20201007_bak/vector/vector4.f90
  main  NVIDIA  devicenum=0
    time(us): 65
    10: data region reached 2 times
        10: data copyin transfers: 1
             device time(us): total=16 max=16 min=16 avg=16
        18: data copyout transfers: 1
             device time(us): total=18 max=18 min=18 avg=18
    12: compute region reached 10 times
        13: kernel launched 10 times
            grid: [1]  block: [32]
             device time(us): total=31 max=4 min=3 avg=3
            elapsed time(us): total=670 max=482 min=20 avg=67
```

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]

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

```
main:
     21, 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(v1[index->],index[:10]) [if not
already present]
```

配列v1の長さの認識がおかしい……

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]
```

allocate1.f90

```
main:
     14, Generating implicit copyin(v1(index)) [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
```

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
                        ↑ PGI 20.4ではエラー終了してしまうようだ
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では配列長をうまく認識できておらず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？

PGI_ACC_NOTIFYをセットして動作を確認

```
$ export PGI_ACC_NOTIFY=2

$ ./a1c_c_acc
upload CUDA data  file=/home/center/a49979a/share/20201007/vector/allocate1.c function=main line=21 device=0 threadid=1 variable=v1 bytes=8
upload CUDA data  file=/home/center/a49979a/share/20201007/vector/allocate1.c function=main line=21 device=0 threadid=1 variable=index bytes=40
download CUDA data  file=/home/center/a49979a/share/20201007/vector/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

$ ./a2c_c_acc
upload CUDA data  file=/home/center/a49979a/share/20201007/vector/allocate2.c function=main line=21 device=0 threadid=1 variable=v1 bytes=80
upload CUDA data  file=/home/center/a49979a/share/20201007/vector/allocate2.c function=main line=21 device=0 threadid=1 variable=index bytes=40
download CUDA data  file=/home/center/a49979a/share/20201007/vector/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

$ ./a1f_f_acc
upload CUDA data  file=/home/center/a49979a/share/20201007/vector/allocate1.f90 function=main line=14 device=0 threadid=1 variable=index bytes=40
Failing in Thread:1
call to cuStreamSynchronize returned error 700: Illegal address during kernel execution

$ ./a2f_f_acc
upload CUDA data  file=/home/center/a49979a/share/20201007/vector/allocate2.f90 function=main line=14 device=0 threadid=1 variable=v1 bytes=80
upload CUDA data  file=/home/center/a49979a/share/20201007/vector/allocate2.f90 function=main line=14 device=0 threadid=1 variable=index bytes=40
download CUDA data  file=/home/center/a49979a/share/20201007/vector/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
```

←2に設定するとCPU-GPU間の通信の情報が出力される

v1が8byte、つまり1要素しか送信されていないのが原因だとわかる

やはりv1の転送で躓いている？

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 -ta オプションに deepcopy を追加
      - PGI 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 -ta 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
  - メンバの長が固定であれば問題は起きない

```c
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)

```f90
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;
};
struct Sobj obj;

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;
}

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

←コンパイル時にサイズが未定

実行時の引数からnを取得してmalloc

(deepcopy2.c)

```
type :: Tobj
   double precision,allocatable :: values(:)
end type Tobj
type(Tobj) :: obj

! 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
```

↑コンパイル時にサイズが未定

(deepcopy2.f90)

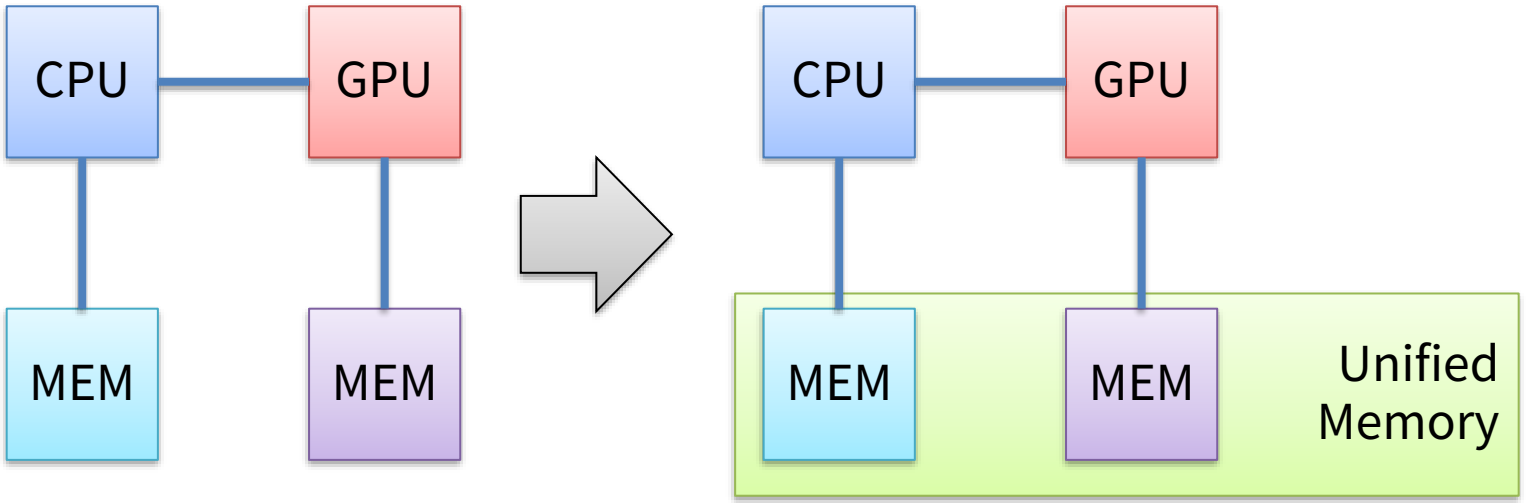
- コンパイラのバージョンとコードの書き方によって正しく動いたり動かなかったりする
- Fortran版についてはdeepcopyオプションをつけることで正しく動く、はずなので試してみると良い
  - 実際に試してみたが、まだ成熟していないためかPGI 19.1, 20.4, HPC SDKで結果がまちまちだった

```
pgfortran -Minfo=accel -acc -ta=tesla:cc70,deepcopy -tp=skylake -o d2f_f_acc_deepcopy deepcopy2.f90
```

（コンパイル時の出力的には特に変わった情報は出ない）

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

```
pgcc -Minfo=accel -acc -ta=tesla:cc70,managed -tp=skylake -o d2c_c_acc_managed 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],obj) [if not already present]

$ ./d2c_c_acc_managed
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

```
pgfortran -Minfo=accel -acc -ta=tesla:cc70,managed -tp=skylake -o d2f_f_acc_managed deepcopy2.f90
main:
     32, Generating implicit copy(obj,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

$ ./d2f_f_acc_managed
 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
```

Deep copy問題の解決方法：手動でフルコピーする

- いずれもPGI 19.10, 20.4で正しく動作した例
  - 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を使うと良い模様
    - （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)

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？

```c
#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;
    }
  }
```

```fortran
!$acc data copy(v1)
!$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……？

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

(update1.c)

```fortran
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？

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

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

```fortran
!$acc data copy(v1)
!$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
```

- 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？ ⇨ 計算前の値が出力されてしまう

- 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……？

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

(update1.c)

```fortran
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

```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;
    }
  }
```

(update2.c)

```f90
!$acc data copy(v1)
!$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

- 前回はkernels節を用いて並列化対象の指定を行った
- 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=NULL, **b=NULL, **c=NULL;
   // mallocでa,b,cを確保
36 #pragma acc kernels
37   for(i=0; i<n; i++){
38     for(j=0; j<n; j++){
39       for(k=0; k<n; k++){
40         c[i][j] += a[i][k] * b[k][j];
41       }
42     }
43   }
```

- Fortran (matmul1.f90)

```
   double precision, allocatable :: a(:,:), b(:,:), c(:,:)
   allocate(a(n,n), b(n,n), c(n,n)
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
```

n=10で実行してみた

C

```
37: compute region reached 1 time
    37: kernel launched 1 time
        grid: [1]  block: [1]
         device time(us): total=88 max=88 min=88 avg=88
        elapsed time(us): total=108 max=108 min=108 avg=108
```

Fortran

```
36: compute region reached 1 time
    39: kernel launched 1 time
        grid: [1x10]  block: [128]
         device time(us): total=6 max=6 min=6 avg=6
        elapsed time(us): total=504 max=504 min=504 avg=504
```

どうやらCとFortranで実行時間に大きな差があるようだ、何故だろうか？

三重ループ構造による単純な行列積プログラムの例

- Fortran (matmul1.f90)

```
   double precision, allocatable :: a(:,:), b(:,:), c(:,:)
   allocate(a(n,n), b(n,n), c(n,n)
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
```

```
pgfortran -Minfo=accel -acc -ta=tesla:cc70 ……
main:
     36, Generating implicit copyin(a(1:n,1:n)) [i
         Generating implicit copy(c(1:n,1:n)) [if
         Generating implicit copyin(b(1:n,1:n)) [i
     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 Tesla code
         37, !$acc loop gang ! blockidx%y
         38, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
         39, !$acc loop seq
```

- copy関係はコンパイラの判断で特に問題はない
- 37, 38, 39行目が3重ループ

sequential
ループが逐次実行されることを意味する

- 3重ループの外側2つは並列化され、最内ループは逐次実行。c(j,i)が競合しない妥当な判断。

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=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:cc70 -tp=skylake -o m1c_c_acc matmul1.c
main:
     37, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
         Accelerator serial kernel generated
         Generating Tesla code
         37, #pragma acc loop seq
         38, #pragma acc loop seq
         39, #pragma acc loop seq
     37, Generating implicit copyin(b[:n][:n]) [if not already present]
         Generating implicit copy(c[:n][:n]) [if not already present]
         Generating implicit copyin(a[:n][:n]) [if not already present]
     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
```

- 37, 38, 39行目が3重ループ
- 依存関係があり並列化できず、逐次実行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
- copy関係はコンパイラの判断で特に問題はない

loop指示文の追加

- C (matmul2.c)

```
   double **a=NULL, **b=NULL, **c=NULL;
   // mallocでa,b,cを確保
36 #pragma acc kernels
37 #pragma acc loop independent
38   for(i=0; i<n; i++){
39 #pragma acc loop independent
40     for(j=0; j<n; j++){
41 #pragma acc loop seq
42       for(k=0; k<n; k++){
43         c[i][j] += a[i][k] * b[k][j];
44       }
45     }
46   }
```

並列実行する

並列実行しない

n=10で実行してみた

```
    38: compute region reached 1 time
        42: kernel launched 1 time
            grid: [1x10]  block: [128]
             device time(us): total=7 max=7 min=7 avg=7
            elapsed time(us): total=27 max=27 min=27 avg=27
```

C版が劇的に高速化

- Fortran (matmul2.f90)

```
   double precision, allocatable :: a(:,:), b(:,:), c(:,:)
   allocate(a(n,n), b(n,n), c(n,n)
36 !$acc kernels
37 !$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

```
    36: compute region reached 1 time
        42: kernel launched 1 time
            grid: [1x10]  block: [128]
             device time(us): total=6 max=6 min=6 avg=6
            elapsed time(us): total=484 max=484 min=484 avg=484
```

- 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

- C言語、loop independent指定なし(matmul1.c)

```
37, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
    Accelerator serial kernel generated
    Generating Tesla code
    37, #pragma acc loop seq
    38, #pragma acc loop seq
    39, #pragma acc loop seq
37, Generating implicit copyin(b[:n][:n]) [if not already present]
    Generating implicit copy(c[:n][:n]) [if not already present]
    Generating implicit copyin(a[:n][:n]) [if not already present]
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
```

各ループに依存性があり並列化できないことが示されている

- C言語、loop independent指定あり(matmul2.c)

```
38, Loop is parallelizable
    Generating implicit copy(c[:n][:n]) [if not already present]
    Generating implicit copyin(b[:n][:n],a[:n][:n]) [if not already present]
40, Loop is parallelizable
42, 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
```

指示に従って各ループを並列化したことが示されている

collapse版

- C (matmul3.c)

```
   double **a=NULL, **b=NULL, **c=NULL;
   // mallocでa,b,cを確保
36 #pragma acc kernels
37 #pragma acc loop independent collapse(2)
38   for(i=0; i<n; i++){
39     for(j=0; j<n; j++){
40 #pragma acc loop seq
41       for(k=0; k<n; k++){
42         c[i][j] += a[i][k] * b[k][j];
43       }
44     }
45   }
```

- Fortran (matmul3.f90)

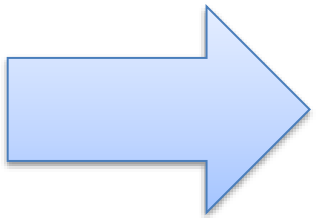
```
   double precision, allocatable :: a(:,:), b(:,:), c(:,:)
   allocate(a(n,n), b(n,n), c(n,n)
36 !$acc kernels
37 !$acc loop independent collapse(2)
38   do i=1, n
39     do j=1, n
40 !$acc loop seq
41       do k=1, n
42         c(j,i) = c(j,i) + a(k,i) * b(j,k)
43       enddo
44     enddo
45   enddo
46 !$acc end kernels
```

- 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
  （n=10のmatmul2とmatmul3ではほとんど変わらなかった）

```
C       38, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
        39,   ! blockidx%x threadidx%x collapsed
        41, !$acc loop seq

Fortran 38, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
        39,   /* blockIdx.x threadIdx.x collapsed */
        41, #pragma acc loop seq
```

collapseのイメージ

```
#pragma acc kernels
#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 kernels
#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] = ...;
      }
    }
  }
```

- OpenMPのcollapseと同じ効果
- 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

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特に気にしていなかった
  - 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

- OpenACCの場合
  - 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

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ようだ

```
pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -o v0c_c_acc vector0.c
         16, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */

pgfortran -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -o v0f_f_acc vector0.f90
         13, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

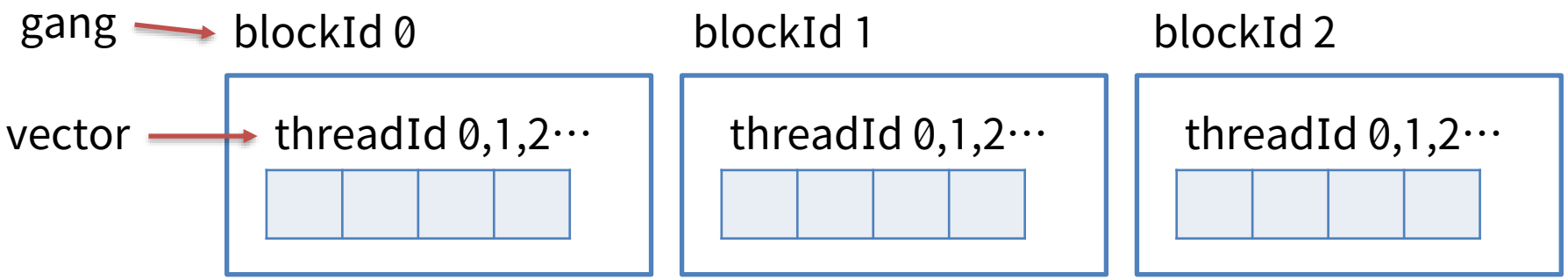
  - 数字を指定することもできるようだ
    - vector(32)

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

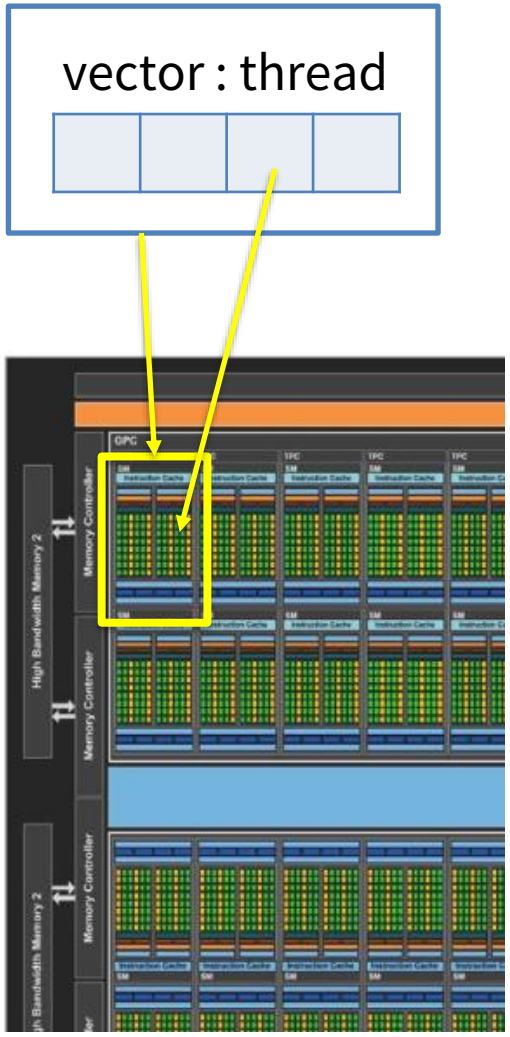
- CUDAの並列実行モデルが階層的になっているため、それにあわせて言語設計された、と考えれば良い
  - CUDAはgrid, threadblock, threadの三階層構造

  - 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

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：単純な行列積プログラム

- C (matmul1.c)

```c
double **a=NULL, **b=NULL, **c=NULL;
a = (double**)malloc(sizeof(double*)*n);
b = (double**)malloc(sizeof(double*)*n);
c = (double**)malloc(sizeof(double*)*n);
#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 (matmul1.f90)

```fortran
double precision, allocatable :: a(:,:), b(:,:), c(:,:)
allocate(a(n,n))
allocate(b(n,n))
allocate(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
```

matmul1, matmul2, matmul3を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？
- 問題サイズを大きくしてみるとどうか？

演習：行列積を2回実行してみる

- 単純に2回記述すると、その間で全データの転送が発生する。data指示文で通信をなくそう。

```
double **a=NULL, **b=NULL, **c=NULL;
a = (double**)malloc(sizeof(double*)*n);
b = (double**)malloc(sizeof(double*)*n);
c = (double**)malloc(sizeof(double*)*n);
#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];
      }
    }
  }
#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];
      }
    }
  }
```

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

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など
- 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）

- 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

- 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

- 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）

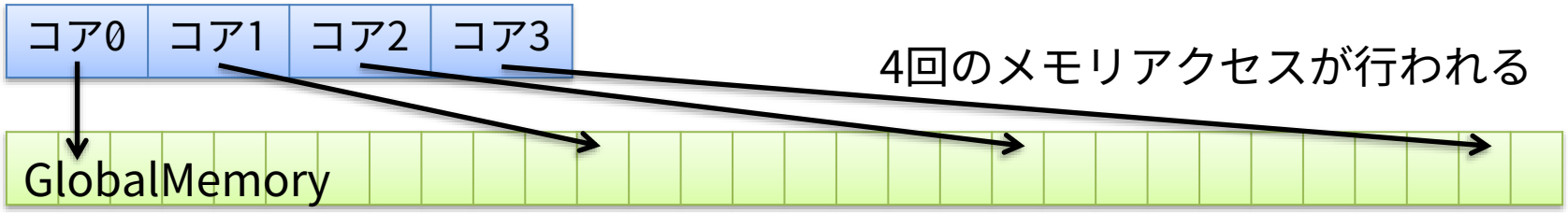
```
#pragma acc kernels
#pragma acc loop gang
for(i=0;i<N;i++){
#pragma acc loop vector
  for(j=0; j<N; j++){
○   a[i][j] = b[i][j] + c[i][j];
×   a[j][i] = b[j][i] + c[j][i];
  }
}
```

```
!$acc kernels
!$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
```

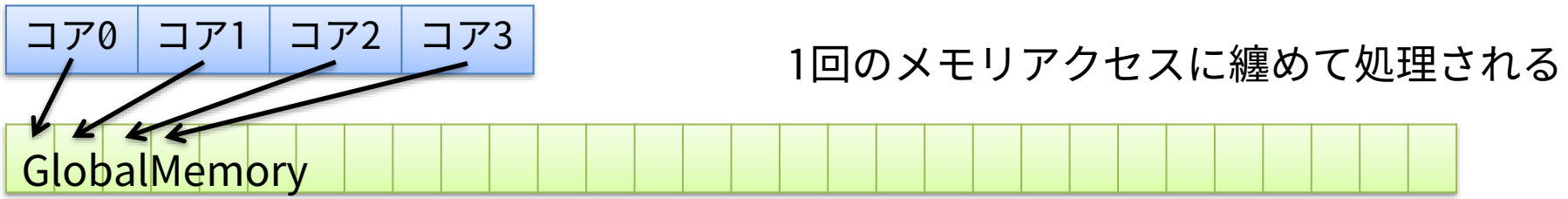
- 無理なものは無理：演算性能・メモリ転送性能を超える性能は出ない、CPUに勝てるとは限らない

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



- 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

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）

```c
#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);
    }
  }
}

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

どちらか片方でも良い
（両方書いても問題はない）

(routine0.c)

```f90
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GPUカーネルは逐次実行になってしまう
- （calc内のループにloop指示文を加えても逐次になってしまう）

関数内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

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

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

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

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

}

#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)

```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)

```f90
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-ta=tesla～を付ける）

- pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -c routine3a.c
- pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -c routine3b.c
- pgcc -ta=tesla:cc70 -tp=skylake -o r3c_c_acc routine3a.o routine3b.o

（最後のリンクにはもう-Minfo=acelや-accは不要、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
```

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
```

- 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-ta=tesla～を付ける）

- `pgfortran -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -c routine3a.f90`
- `pgfortran -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -c routine3b.f90`
- `pgfortran -ta=tesla:cc70 -tp=skylake -o r3f_f_acc routine3a.o routine3b.o`

デバッグと性能解析

- デバッグ
  - 単純なプログラムのOpenACC化でデバッグ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るかは注意して確認する必要がある
- 性能解析
  - NVIDIA社の提供するnvprofやnvvp、nsys(Nsight Systems)などが利用可能
    - nvprofとnvvpが使われてきたが、今後はnsysに移行するとNVIDIAが宣言している
  - 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lineinfoを付ける
    - （付けなくてもしっかり情報が得られるようで、よくわからない……）

```
pgcc -g -Minfo=accel -acc -ta=tesla:cc70,lineinfo -tp=skylake -o a1c_c_acc allocate1.c
```

    - nvprofやnvvpはCUDA toolkitに含まれているため、使うときはmodule load cudaも必要

nvprof の利用例

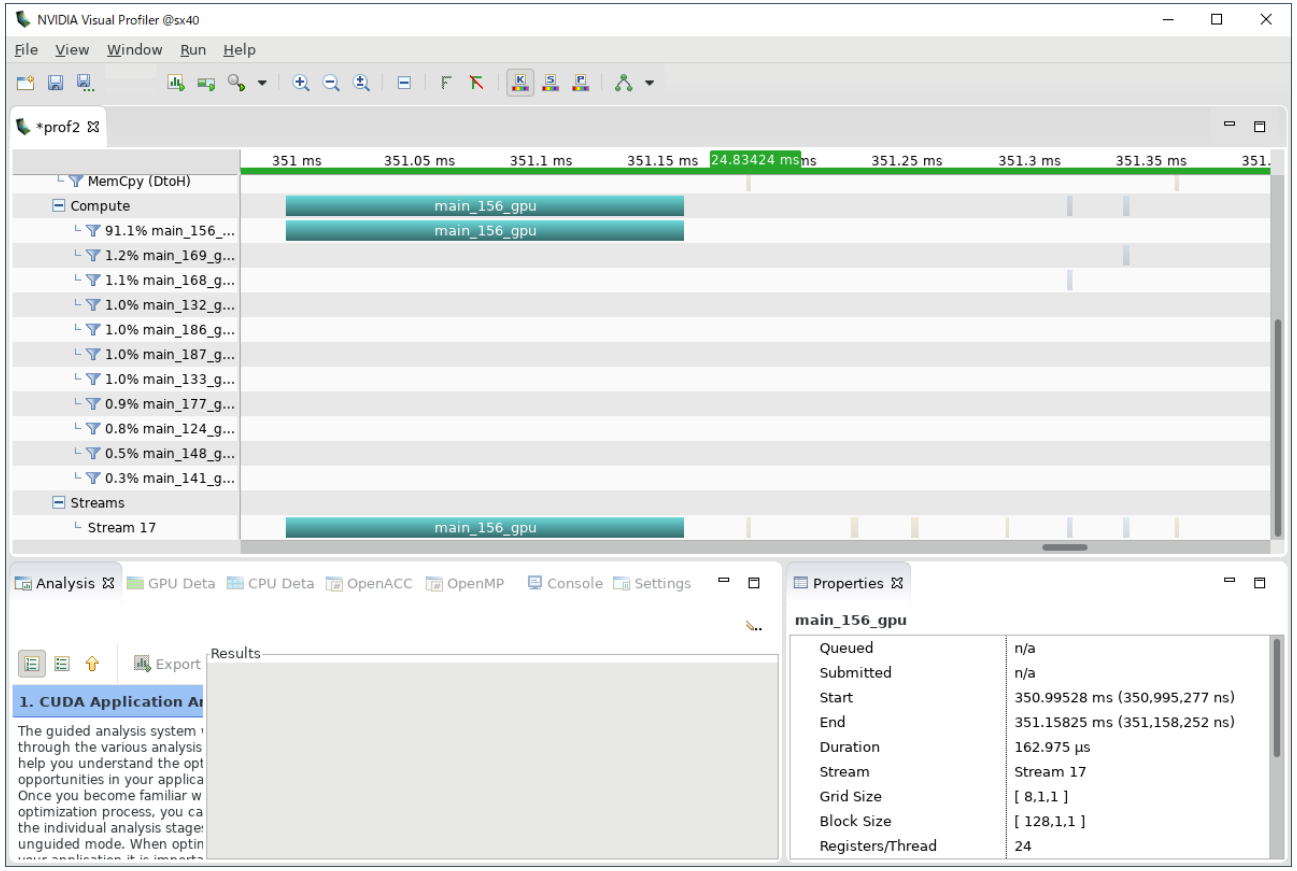
```
$ nvprof ./a1c_c_acc
==74== NVPROF is profiling process 74, command: ./a1c_c_acc
==74== Profiling application: ./a1c_c_acc
 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
==74== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   80.91%  24.960us         1  24.960us  24.960us  24.960us  main_21_gpu
                   11.51%  3.5520us         2  1.7760us  1.7280us  1.8240us  [CUDA memcpy HtoD]
                    7.57%  2.3360us         1  2.3360us  2.3360us  2.3360us  [CUDA memcpy DtoH]
      API calls:   89.63%  241.14ms         1  241.14ms  241.14ms  241.14ms  cuDevicePrimaryCtxRetain
                    8.89%  23.924ms         1  23.924ms  23.924ms  23.924ms  cuMemHostAlloc
                    0.72%  1.9300ms         1  1.9300ms  1.9300ms  1.9300ms  cuModuleLoadDataEx
                    0.37%  1.0034ms         4  250.84us  4.0220us  982.67us  cuMemAlloc
                    0.32%  873.38us         1  873.38us  873.38us  873.38us  cuMemAllocHost
                    0.02%  44.316us         1  44.316us  44.316us  44.316us  cuLaunchKernel
                    0.01%  36.630us         3  12.210us  3.9850us  25.765us  cuStreamSynchronize
                    0.01%  28.569us         2  14.284us  4.5230us  24.046us  cuMemcpyHtoDAsync
                    0.00%  12.803us         4  3.2000us  1.6110us  4.8280us  cuDeviceGetPCIBusId
                    0.00%  10.356us         1  10.356us  10.356us  10.356us  cuStreamCreate
                    0.00%  8.7470us         1  8.7470us  8.7470us  8.7470us  cuMemcpyDtoHAsync
 (中略)
 OpenACC (excl):   91.74%  24.066ms         1  24.066ms  24.066ms  24.066ms  acc_enter_data@allocate1.c:21
                    7.47%  1.9586ms         1  1.9586ms  1.9586ms  1.9586ms  acc_device_init@allocate1.c:21
                    0.19%  50.374us         1  50.374us  50.374us  50.374us  acc_enqueue_launch@allocate1.c:21 (main_21_gpu)
                    0.15%  39.490us         2  19.745us  5.3320us  34.158us  acc_enqueue_upload@allocate1.c:21
```

- 実は内部ではCUDAに関係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（再）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、ただし
    レスポンスの良さを考えるとローカルPCに
    インストールして実行する方がおすすめ）
- 画面例　→



Nsight

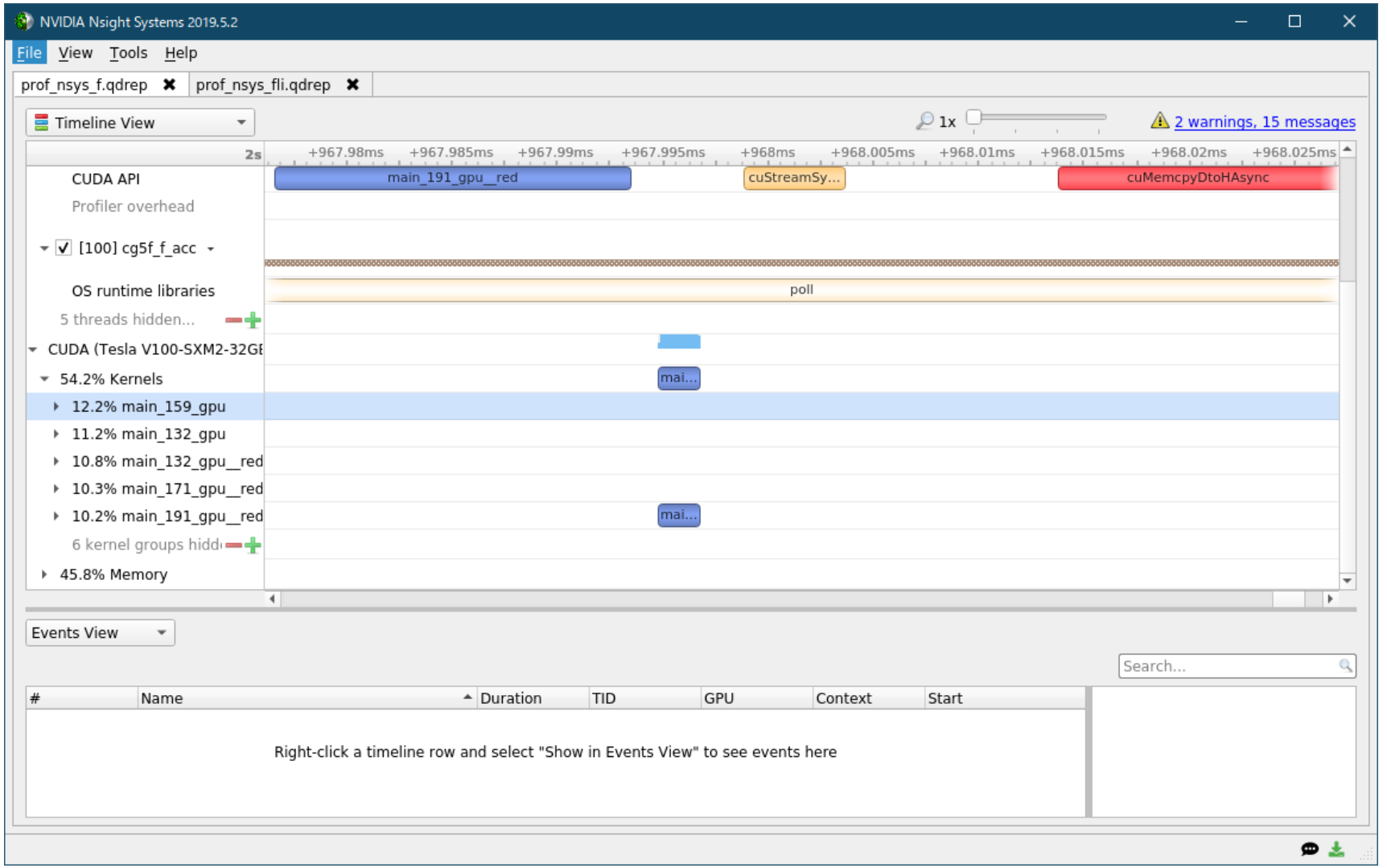
- 新しいプロファリングツール
  - Nsight Systems, Nsight Compute, Nsight Graphicsから構成される
  - CUIとGUIそれぞれで利用できる

- CUIによるプロファイル情報の取得と表示
  - nsys profile ./a.out
    → report1.qdrepファイルが作られる
  - 引数-oで生成ファイル名を指定
  - --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　→

```
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

- File – Openでqdrepファイルを開いて閲覧



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て使いたい場合など？（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

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

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

- 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を担当するような構成にしておけば汎用性があって使いやすい

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
  - OpenMPと似ているため、両方使えると色々と活用できる
  - GPUが利用できる環境がある場合は是非使ってみて欲しい

- 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化の練習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
    - 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
    - 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}

今回用いるCG法コードの計算手順

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＝一番時間がかかる処理
$\alpha_i = \rho_{i-1} / p^{(i)} q^{(i)}$	• 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（dot product, reduction）
  - これらを全てGPUカーネルにしてしまえば良さそうである
  - ※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？

```
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++){          ★行列ベクトル積
    q[i] = 0.0;                行ごとの計算を並列に行える
    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;
}
```

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

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

(cg1.c)

```
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)

※cg0.c, cg0.f90は指示文がまったく入っていないコード

- 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

```
152, Accelerator restriction: size of
the GPU copy of A is unknown
152, #pragma acc loop seq
154, #pragma acc loop seq
```

```
145, !$acc loop gang, vector(128)
147, !$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

```
150  // {q} = [A]{p}
151 #pragma acc kernels copyin(A[N*N])
152 #pragma acc loop independent
153     for(i=0;i<N;i++){
154       q[i] = 0.0;
155 #pragma acc loop seq
156       for(j=0;j<N;j++){
157         q[i] += A[i*N+j]*p[j];
158       }
159     }
```
(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copyやindependentを指定せねば適切に並列化されないことが多い（今後コンパイラがバージョンアップしたら不要となるかも知れない？）
- Fortranでは自動的に適切に並列化されることが多い
- いずれにしてもしっかり確認することは重要

```
153, #pragma acc loop gang, vector(128)
156, #pragma acc loop seq
```

```
146, !$acc loop gang, vector(128)
149, !$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）

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

(cg3.c)

```
137, Loop is parallelizable
     Generating Tesla code
    137, #pragma acc loop gang, vector(128)
         Generating implicit reduction(+:rho)
```

```
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
```

(cg3.f90)

```
131, Loop is parallelizable
     Generating Tesla code
    131, !$acc loop gang, vector(128)
         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た
- 問題サイズが小さすぎる（しかも行列が簡単過ぎる）ため実行時間を比較しても差はほとんどない
  - CPUでもGPUでもあっという間に終わってしまう
- しかし、PGI_ACC_TIME=1を設定して実行してみると、データ転送回数が多いことがわかる
  - PGI_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
  - 本当に毎回転送する必要はあるのだろうか

cg3.c

```
    129: data region reached 6 times
        129: data copyin transfers: 6
        131: data copyout transfers: 3
    137: data region reached 6 times
        137: data copyin transfers: 9
        139: data copyout transfers: 3
    146: data region reached 2 times
        146: data copyin transfers: 1
        148: data copyout transfers: 1
    153: data region reached 4 times
        153: data copyin transfers: 4
        155: data copyout transfers: 2
    161: data region reached 12 times
        161: data copyin transfers: 6
        167: data copyout transfers: 3
    173: data region reached 6 times
        173: data copyin transfers: 9
        175: data copyout transfers: 3
    182: data region reached 6 times
        182: data copyin transfers: 12
        185: data copyout transfers: 6
    191: data region reached 6 times
        191: data copyin transfers: 6
```

cg3.f90

```
    120: data region reached 4 times
        120: data copyin transfers: 4
        125: data copyout transfers: 2
    129: data region reached 4 times
        129: data copyin transfers: 6
        134: data copyout transfers: 2
    139: data region reached 2 times
        139: data copyin transfers: 1
        144: data copyout transfers: 1
    147: data region reached 2 times
        147: data copyin transfers: 2
        152: data copyout transfers: 1
    156: data region reached 4 times
        156: data copyin transfers: 4
        164: data copyout transfers: 2
    168: data region reached 4 times
        168: data copyin transfers: 6
        173: data copyout transfers: 2
    178: data region reached 4 times
        178: data copyin transfers: 8
        184: data copyout transfers: 4
    188: data region reached 4 times
        188: data copyin transfers: 4
```

cg3コンパイル時のメッセージの再確認

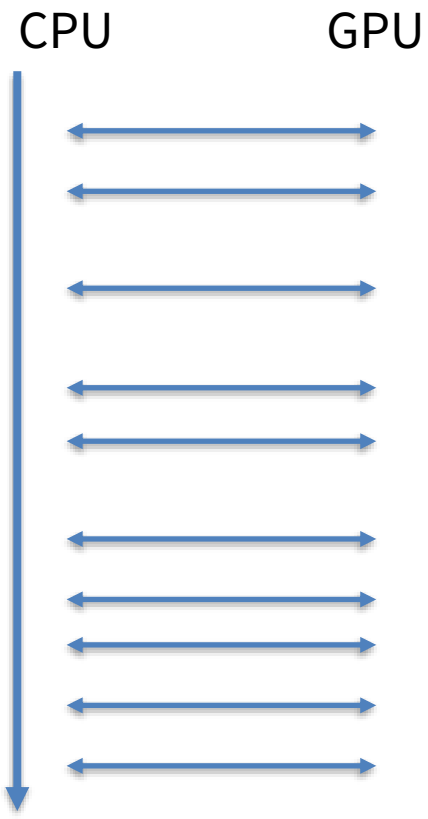
- 確かに毎回コピーが行われている可能性がある

```
pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -mp -o cg3c_c_acc cg3.c
main:
    129, Loop is parallelizable
         Generating Tesla code
        129, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    129, Generating implicit copyout(z[:N]) [if not already present]
         Generating implicit copyin(r[:N],dd[:N]) [if not already present]
    137, Loop is parallelizable
         Generating Tesla code
        137, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating implicit reduction(+:rho)

pgfortran -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -mp -o cg3f_f_acc cg3.f90
main:
    120, Generating implicit copyin(dd(1:n)) [if not already present]
         Generating implicit copyout(z(1:n)) [if not already present]
         Generating implicit copyin(r(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, …
  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？

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)
  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
```

※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

  - 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)

for自体を囲む必要あり

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)

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

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×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。（むしろ簡単過ぎてしまうので収束までの回数を固定化して実験した方が良いかもしれない。）

実行例

```
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
iter 1  1.089293e-01 1.000000e-16
iter 2  2.331098e-16 1.000000e-16
iter 3  5.450800e-18 1.000000e-16
time: 0.000010 sec , 0.000003 sec/iter
result(x):
 8.400000
 8.700000
 7.800000
 1.600000
 9.400000
 3.600000
 8.700000
 9.300000
 5.000000
 2.200000
1 8.400000 8.400000: 3.552714e-15
2 8.700000 8.700000: 5.329071e-15
3 7.800000 7.800000: -8.881784e-16
4 1.600000 1.600000: -8.881784e-16
5 9.400000 9.400000: 0.000000e+00
6 3.600000 3.600000: -1.776357e-15
7 8.700000 8.700000: 0.000000e+00
8 9.300000 9.300000: 0.000000e+00
9 5.000000 5.000000: -8.881784e-16
10 2.200000 2.200000: -1.776357e-15
```

←既知の行列A

←ベクトルX（求める答え）

←既知のベクトルb

←反復計算の履歴

←計算結果ベクトルX

←計算結果ベクトルXと最初に設定した
ベクトルXの比較

- 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）