2019年11月14日（水）10:00-17:00
名古屋大学　2階演習室（201号室）

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

第19回
データサイエンス基礎入門OpenACC入門講習会

プログラムと時間の目安

※実際の時間は状況にあわせて調整します

- 9:30 – 10:00　受付

- 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
  - GPUサーバへのログイン
  - 実行環境の準備

- 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

講師について

- 名前：大島　聡史（おおしま　さとし）
  - 情報基盤研究センター　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-）

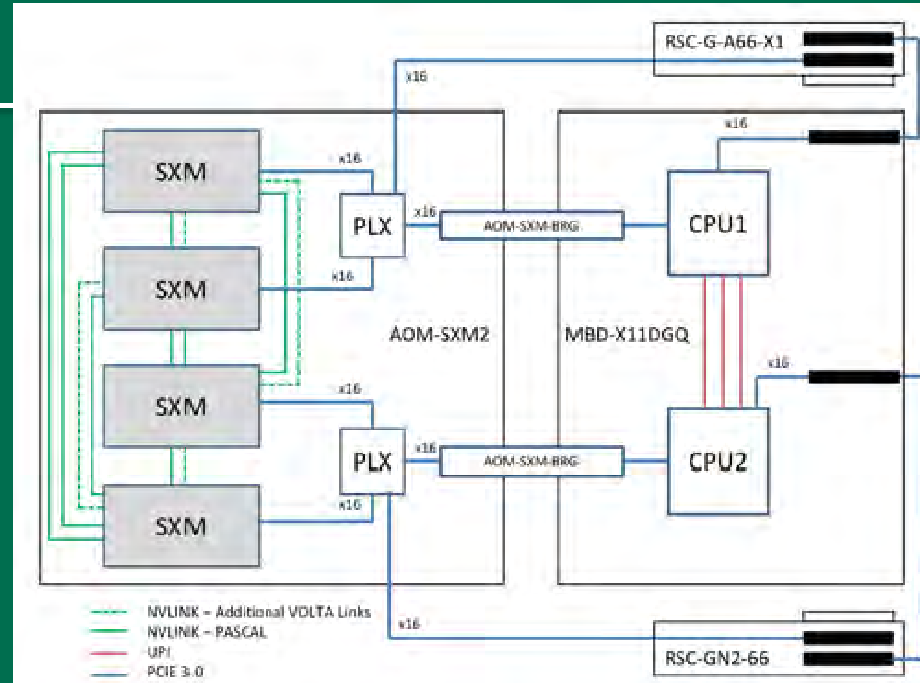
- スパコンの運用・調達に関わりながら最新の計算機環境の活用について研究
  - GPUコンピューティング（およびアプリケーションのGPU化）
  - 並列数値計算（行列計算、疎行列ソルバー、ライブラリ）
  - プログラミング環境（言語やライブラリ）

実習の準備

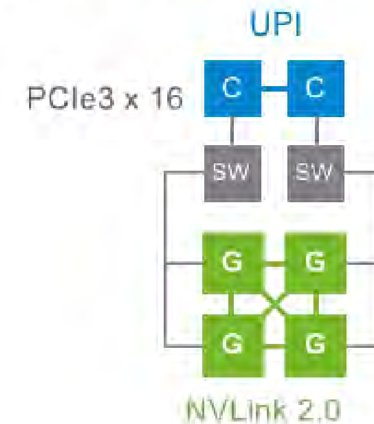
- GPUサーバにログインできることを確認する
- PGIコンパイラ Community Edition の導入（別紙）

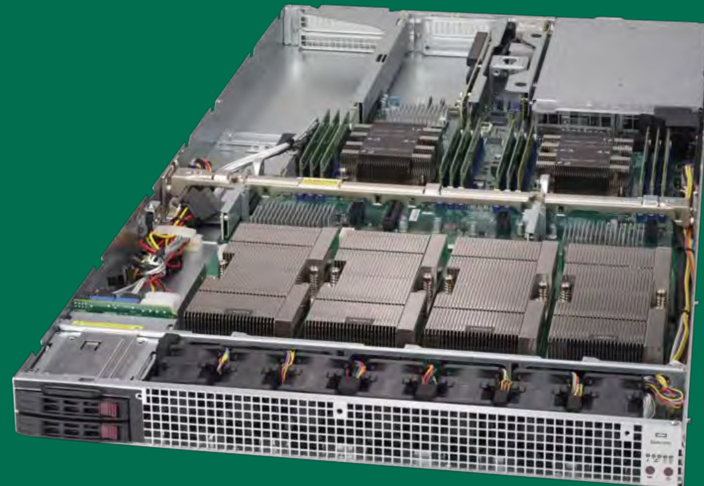
GPU実験環境について

- Skylake + Voltaを搭載した実験用サーバ
- 対象機器：HPE Apollo sx40
  - 情報基盤センター内に設置、ラックマウント式のサーバ×1
  - Intel Xeon Gold 5122 (3.6GHz, 4コア) ×2
  - NVIDIA V100 SXM2 ×4



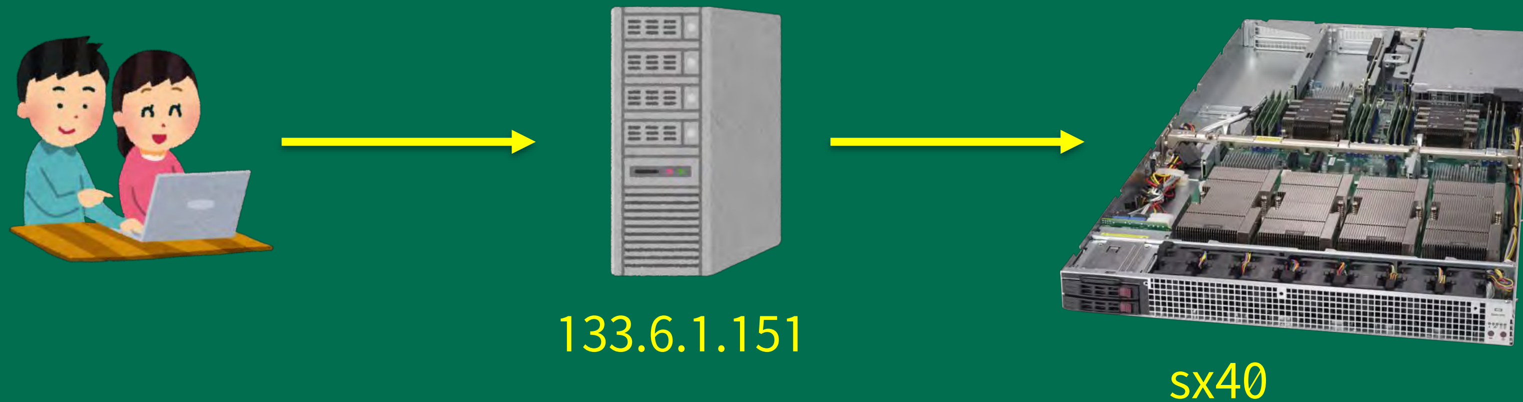
模式化したブロックダイアグラムは下記の通りです。





ログイン方法

- 基本手順：専用のログインノード（133.6.1.151）にsshでログインし、さらに計算ノードsx40にsshでログイン



- 具体的な手順
  - ログインノード「133.6.1.151」 にsshログインする
    - 学内から限定、現地配付したユーザ名と公開鍵を使う
  - ログインノードから計算ノード「10.80.0.15」にsshログインする

無線LANの設定

- 各自のパソコンにおいて、無線LANの設定をしてください
  - 詳細は会場にある無線LAN情報をご覧ください

SSH公開鍵アクセスの設定を行う

- 公開鍵
  - アクセスしたい相手（スパコン）に設置するもの、他人に見られても問題ない
  - 基本的に id_rsa.pub というファイル名が出てきたら公開鍵
- 秘密鍵
  - アクセスする元に置いておく必要があるもの、他人に絶対見せては（見られては）ならない
  - 基本的に id_rsa というファイル名が出てきたら秘密鍵
  - ファイルの読み書き権限にも注意すること
    - `chmod 600 id_rsa` で他人からは読めないようにしておくのが基本

ログイン：ターミナル版

- macOSのターミナル、Linux、Cygwinなどの場合
  - 以下の通り入力する
    `ssh 133.6.1.151 -l aYYxxx`
    または
    `ssh aYYxxx@133.6.1.151`
    - 「-l」はハイフンと小文字のＬ、「aYYxxx」は各自の利用者番号に置き換える
    - ~/.ssh/id_rsa以外のファイル名の場合には -i オプションでファイルを指定する
      例：`ssh -i ~/work/id_rsa aYYxxx@133.6.1.151`
- 接続するかどうかを確認されるので、yesと入力（初回のみ）
- 鍵を作成する際に決めたパスワード（パスフレーズ）を入力
- 正しく入力すればログインノードにログインできる
  - コンソール表示は　bash@vcl ~ $
- さらに `ssh sx40` を実行することでsx40にログインできる
  - 初回は接続先の確認がある→yesと入力してEnter、パスワードを入力してEnterでログイン完了
  - コンソール表示は　bash@sx40 ~ $

ログイン：Windows GUI版

- MobaXtermの場合
  - 「Settings」を選択→「SSH」を選択
  - 「SSH agents」で 「Use Internal SSH agent “MobAgent”」を選択
  - 「Load following keys in MobAgent」で、右側の “+”を選択し、保存しておいた秘密鍵ファイルを選択
  - メッセージに従いMobaXtermを再起動後、パスフレーズを入力
  - 「Session」を選択→「SSH」を選択
  - ログイン先（133.6.1.151）とユーザIDを指定して「OK」でログインノードにログイン
    - コンソール表示は　bash@vcl ~ $
  - さらに ssh sx40 コマンドでsx40にログイン
    - 初回は接続先の確認がある→yesと入力してEnter、パスワードを入力してEnterでログイン完了
    - コンソール表示は　bash@sx40 ~ $

ssh接続を用いたデータ（ファイル）転送：ターミナル版

- scpコマンドを使う
  - ダウンロード
    - `scp aYYxxx@接続先:~/a.f90 ./`
    - 「接続先のホームディレクトリにある“a.f90”を、ローカルPCのカレントディレクトリに取ってくる」という意味
    - ディレクトリごと得る場合は“-r”を追加　`scp -r aYYxxx@接続先:~/SAMP ./`
    - 接続先のホームディレクトリにあるSAMPフォルダを、その中身ごと、PCのカレントディレクトリに取ってくる
  - アップロード
    - `scp ./a.f90 aYYxxx@接続先:`
    - 「ローカルPCのカレントディレクトリにある“a.f90”を、接続先のホームディレクトリに置く」という意味
    - ディレクトリごと置く場合は“-r”を追加　`scp -r ./SAMP aYYxxx@接続先:`
    - PCのカレントディレクトリにあるSAMPフォルダを、その中身ごと、接続先のホームディレクトリに置く

- PCとログインノード間、ログインノードとsx40間で転送する必要がある
  - ポート転送などを使えば一気に転送できる
- 実際には公開鍵を使う際にパスフレーズを要求される
  - 省略するにはssh-agentを使う
- sftpコマンドを使っても良い

ssh接続を用いたデータ（ファイル）転送： Windows GUI版

- MobaXtermは画面左側のツリー状のウィンドウを使って転送が可能
- その他、scpやsftpに対応したソフトを使えば転送が可能
  - WinSCP http://winscp.net/eng/download.php
  - FileZilla https://filezilla-project.org
    - 「Download FileZilla Client」からダウンロード
  - いずれもsshログイン時同様にホストやユーザ名を入力して使う
    - ホスト名に、接続先
    - ユーザ名に、センター発行の利用者番号
    - 秘密鍵に、生成した秘密鍵ファイルを指定
  - 具体的な使い方は各自で調べてください
    - 接続先、ユーザ名、公開鍵の設定さえ正しければ動くはず

PGIコンパイラの導入

- 別紙の手順に従ってインストールする
- pgccなどのコマンドが利用できることを確認する

GPUと並列計算に関する基礎知識

この講習会の目的：GPUを活用する方法の基礎を学ぶ

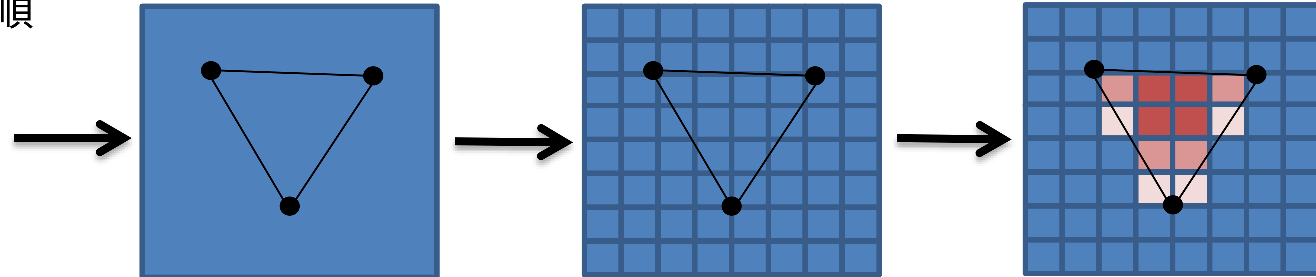
- 何故GPUを活用する必要があるのか？
  - GPUは非常に高い性能をもつハードウェアであり、うまく活用できれば大変強力な研究の道具となる：成果を得るための加速装置
  - GPUは国内外の様々な計算環境に導入されているため、利用スキルを得ておくことはきっとプラスになる
  - 名古屋大学情報基盤センターでも来年度中にGPUを搭載したスーパーコンピュータシステムが稼働開始する予定
- どうやって使う？
  1. 並列化されたソフトウェア（アプリケーション）を使う
  2. （並列化されていないプログラムから）並列化されたライブラリやフレームワークを使う
  3. 自分で並列計算を実装する
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

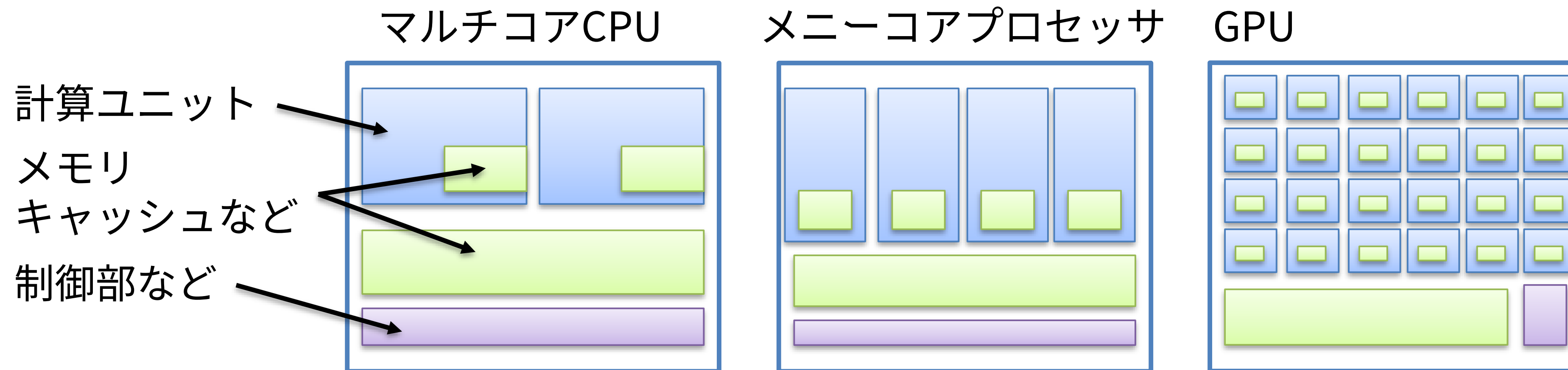


オブジェクト単位、頂点単位、ピクセル単位で同時（並列）処理が可能

  - 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

ハードウェア性能の比較

	名古屋大学FX100（SPARC64 XIfx、「京」後継機）	Tesla V100	Tesla P100	Xeon Gold 6140（九大ITO-B）	Xeon Gold 6154（九大ITO-A）
アーキテクチャ名	SPARC64 XIfx	Volta	Pascal	Skylake-SP	
コア数	32（＋2アシスタントコア）	5120 FP32/2560 FP64 (84 SMs)	3584 FP32/1792 FP64 (56 SMs)	18（HTにより36スレッド実行可能だが、HTは無効化している）	
動作周波数	2.2 GHz	1.455 GHz? - 1.530 GHz	1.328 GHz – 1.480 GHz	2.3 GHz - 3.7 GHz	3.0 GHz – 3.7 GHz
搭載メモリ	HMC 32GB	HBM2 16 or 32 GB	HBM2 16 GB	DDR4 192 GB/socket	DDR4 96 GB/socket
HPL（ピボット選択付きLU分解）	0.97 TFLOPS ※名古屋大提供値	5 TFLOPS程度 ※NVIDIA公称値	4 TFLOPS程度 ※NVIDIA提供バイナリ実測値	0.9 TFLOPS程度 ※MKL実測値	1.1 TFLOPS程度 ※MKL実測値
STREAM Triad A[i]=B[i]+C[i]	210 GB/s	820 GB/s	550 GB/s	95 GB/s	95 GB/s

GPUは非常に高い性能を持つ、使わなければもったいない！

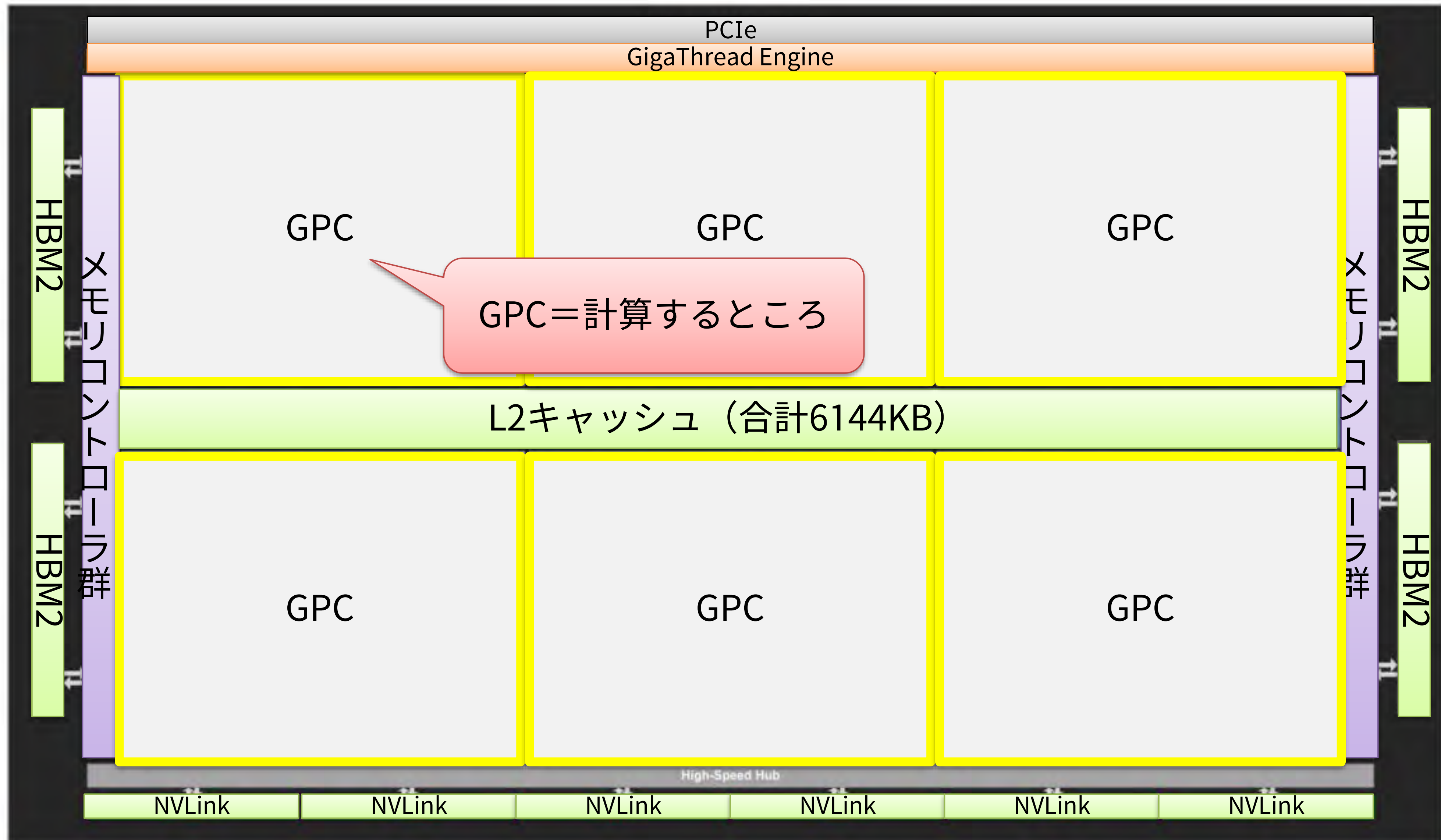
GPUの中身（もう少し詳しく）

- 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

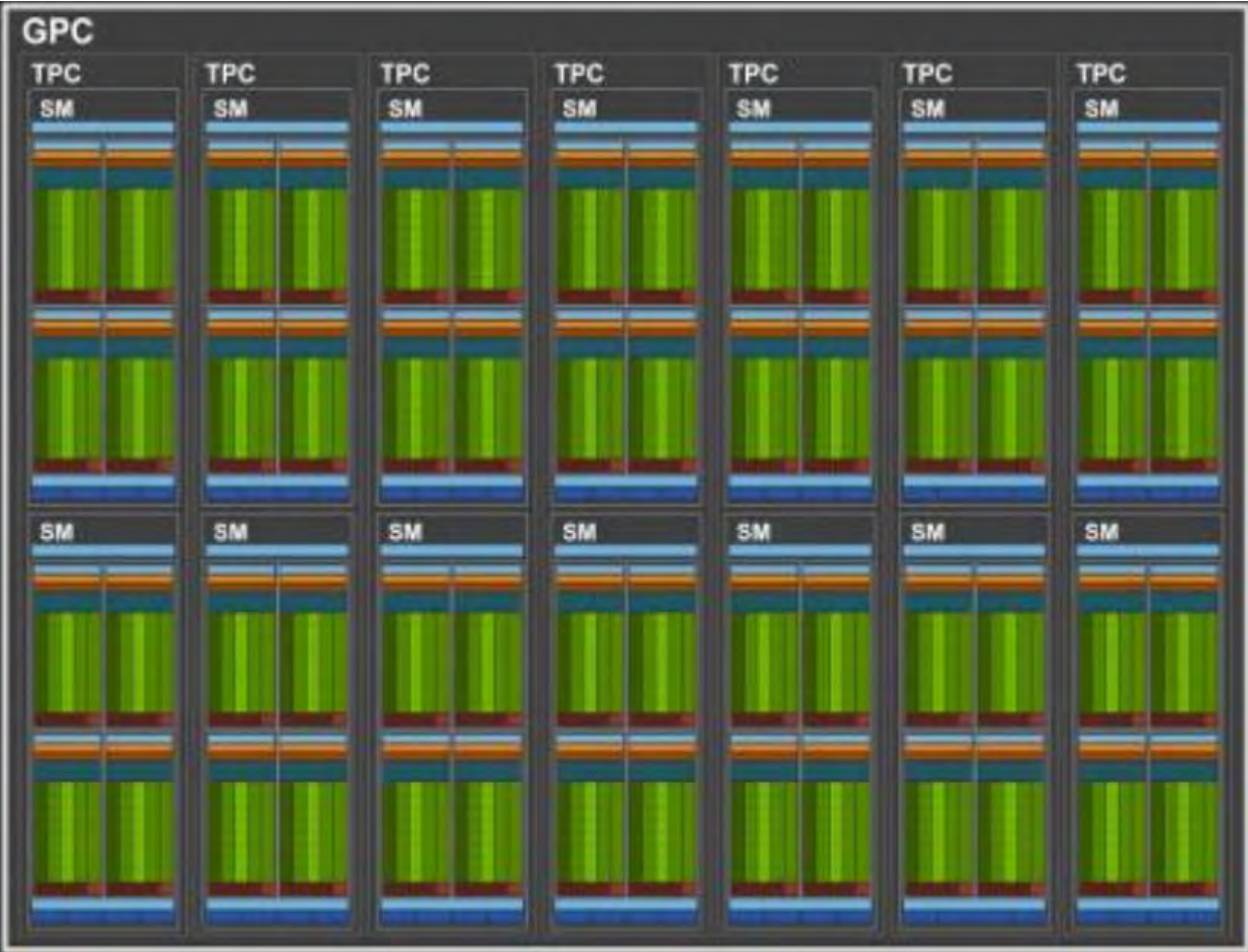
全体構成

全体構成



GPCの中身

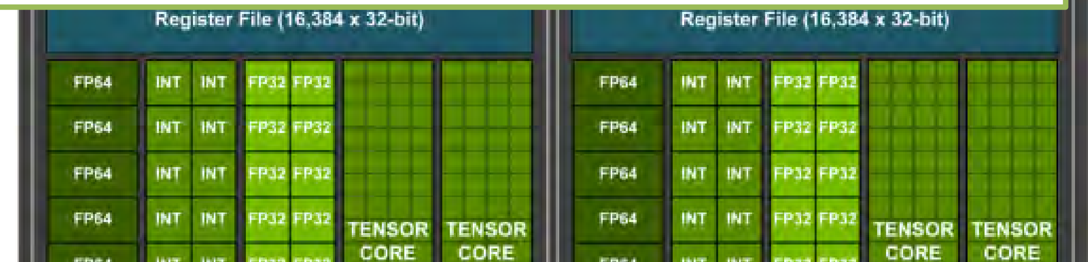
- GPC: GPU Processing Cluster
- TPC: Texture Processing Cluster
- SM: Streaming Multiprocessor



- 1GPUに6GPC搭載
- 1GPCに7TPCs搭載、各TPCに2SMs搭載
- 1SM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……
  - 5120 FP32 cores
  - 5120 INT32 cores
  - 2560 FP64 cores
  - 640 Tensor cores
  - 320 Texture units
- これら大量のコアが最大1530MHz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



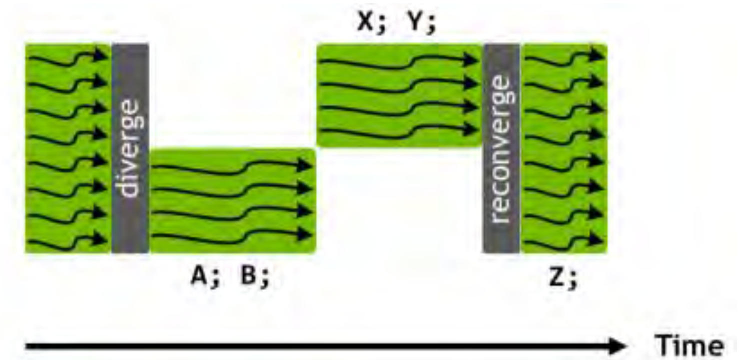
- Voltaで新しく搭載された計算コア
- 4x4行列[FP16or32] ＝ 4x4行列[FP16] × 4x4行列[FP16] ＋ 4x4行列[FP16or32]

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

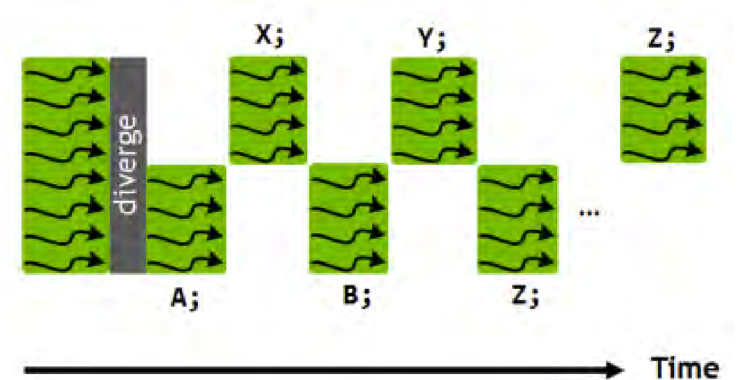
SIMT(Single Instruction Multiple Thread)

- 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
    - 実際にはデータ転送分が省かれるが

- NVIDIAの示している例
  - PascalまでのSIMTの例



  - Volta以降のSIMTの例



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

世界のスパコンで使われるGPU

- アメリカと中国が特に強い
- 最上位の総コア数は100万以上
- 理論性能に対して出ている性能はそれほど高くはない
  - 昔（主にベクトル計算機）は高かった
  - 現実のアプリではもっと下がる
- 消費電力も大きい
  - 大きなものは発電所を1つ占有するレベル
  - 10MW：一般家庭300世帯程度

- 下線部はNVIDIA社のGPU
  - TOP10内の5システムを含む多数のスパコンがGPUを主要な計算装置として活用
    - TOP100中29、全500中125

TOP500 2019年6月版

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**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
2	**Sierra** - IBM Power System S922LC, 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
3	**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
4	**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
5	**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	
6	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
7	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
8	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
9	**SuperMUC-NG** - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path , Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	
10	**Lassen** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100 , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	288,288	18,200.0	23,047.2	

並列計算の基礎知識

- 並列計算とは何か？並列プログラミングとは何か？
  - 並列計算：何らかの計算処理を同時並行的に行うこと
    - 並列：同じ処理を同時に行う、ある処理を幾つかのサブ処理に分けて同時並行的に行う
    - 平行：何らかの処理を同時並行的に行う
  - 並列プログラミング（並列化プログラミング）：並列化・並列計算を行うためのプログラミング
- なぜ並列計算を行うのか？
  - 短時間で終わらせたい計算があるから、計算機（CPUやパソコンなど計算を行うHW）の持つ能力をフル活用したいから
  - 現代の計算機は並列計算により高い性能を達成するシステム
    - 逐次計算性能を上げることができなくなっている
  - PC用CPUもスマートフォン向けCPUもマルチコアCPUが主流
    - 大規模なスーパーコンピュータの総コア数は100万以上
  - ➢ HWの性能を十分に引き出すには並列計算が必須

並列化の基本的なイメージ：ループ並列化

- 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];
}
```

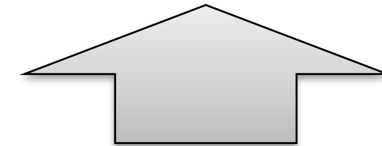
PE2
```
for(i=0; i<N; i++){
 D[i] = E[i] + F[i];
}
```

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];
}
```

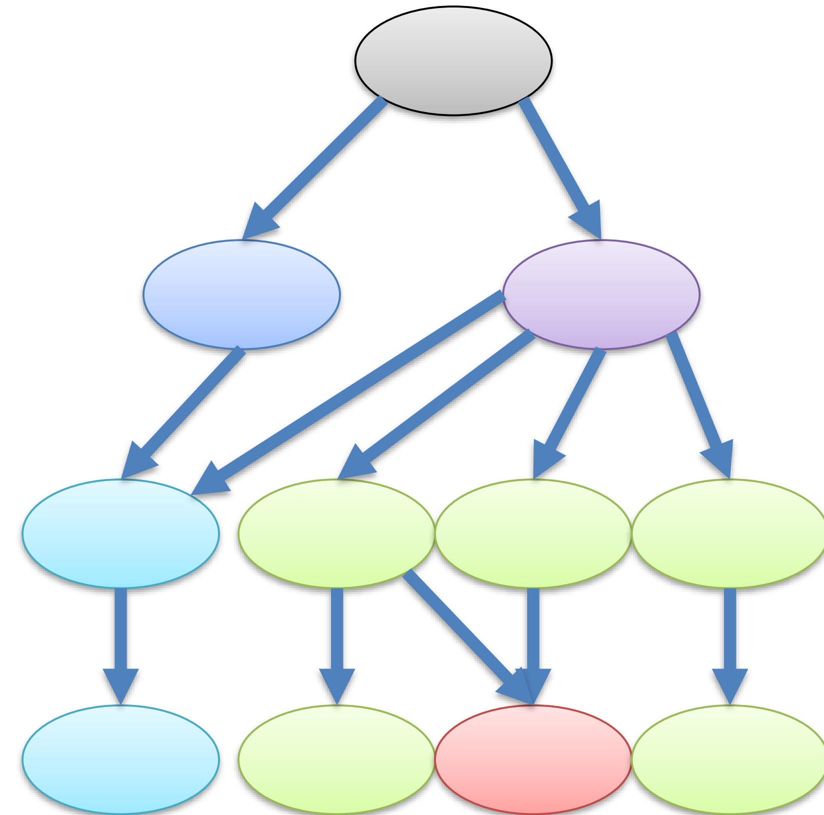
PE2
```
for(i=N/2; i<N; i++){
 A[i] = B[i] + C[i];
 D[i] = E[i] + F[i];
}
```



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

並列化の基本的なイメージ：タスク並列化

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



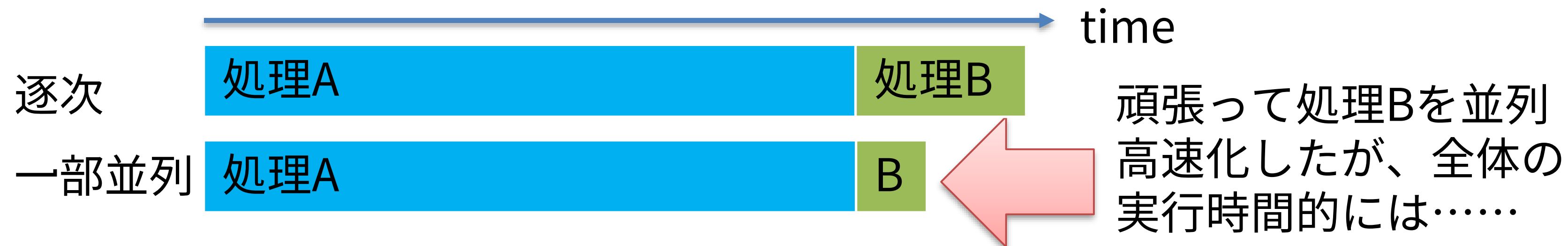
並列計算の注意点

- 並列化できないプログラムも少なくない
  - 計算順序に制約がある場合は困難
    - 工夫により可能となることもある

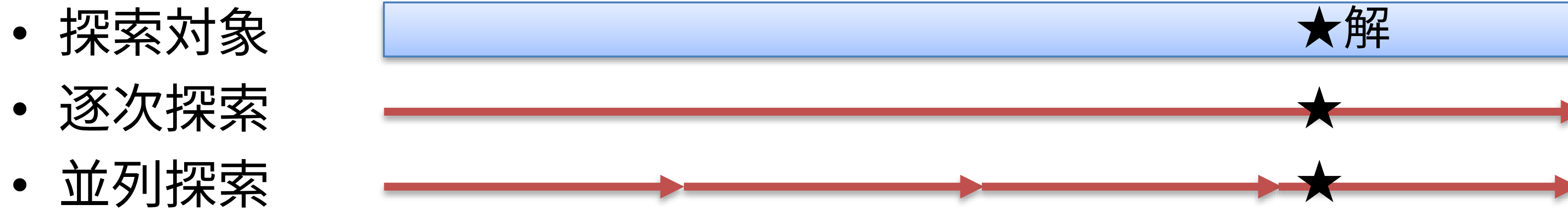
```
for(i=1; i<N; i++){
 A[i] = A[i-1] + B[i];
 C[i] = A[i] + D[i];
}
```

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、プログラム全体を見る必要もある



  - 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に担当させるかをちゃんと考える必要がある

GPUを用いた並列計算の注意点（性能を得るコツ）

- 多数のコア全てを満たすに十分な仕事を与える
  - 数千のコア全てをフル稼働させる並列度が必要
  - コア数の数倍以上の仕事があることが望ましい
    - メモリアクセスを待つ間に別の仕事ができる
- WARPを意識する
  - 内部的には32演算器単位で動作しており、分岐の際などに注意が必要
    - Voltaから変わってしまった
- 連続メモリアクセスについて考える
  - コアレスなメモリアクセスが高速

- 詳細は終盤（OpenACCプログラムの最適化）にて解説する
  - 基本的には、並列度が高くて分岐の少ない単純なプログラムが良い

並列計算環境を利用するための手段

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

- 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
- 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
    - GPUに対応した実装はまだ少ないが、将来的にはOpenACCを吸収？
- CUDA：NVIDIA社のGPU向け、GPUを普及させた最大要因の一つ、NVIDIA GPUの能力をフル活用したければ必須
- OpenCL：「汎用版CUDA」のようなもの、FPGAなどでも利用可能
- コンパイラによる自動並列化：SIMD並列化など一部の処理で有用

参考：CUDAプログラム

- 単純な配列コピーの例

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

GPU上で行われる処理（GPUカーネル）

- 同時にたくさん実行される
- 自分のIDを元に計算するべき範囲を特定する

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

CPU上で行われる処理

- 専用の関数などを用いて様々な処理を行う

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

GPU（CUDA）向けライブラリ・フレームワーク

- GPUの性能を活用できる様々なライブラリが公開されている
  - https://developer.nvidia.com/gpu-accelerated-libraries
  - 例
    - Deep Learningライブラリ
      - cuDNN, TensorRT, DeepStream SDK
    - 数値計算・数学ライブラリ
      - cuBLAS, cuSPARSE, cuRAND, cuFFT, ...
    - 通信、C++クラス、etc.
      - NCCL, Thrust, OpenCV, MAGMA, ...

  - NVIDIA社自ら手がけるものは「CUDA-X」と呼ばれるようになった
  - 各自が行いたい処理とマッチしていれば容易にGPUの性能を活用することができる

OpenACCハンズオン

内容

- 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
- CG法を題材としてOpenACCの基本を学ぶ
  - 少し複雑なコードのOpenACC化について
- 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

- 資料中で用いているソースコードは
  /home/a49979a/share/20191114/20191114.tgz
  で公開している。
- cpでコピーして tar zxvf 20191114.tgz で展開して使ってください。
- ただし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。

OpenACC

- 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

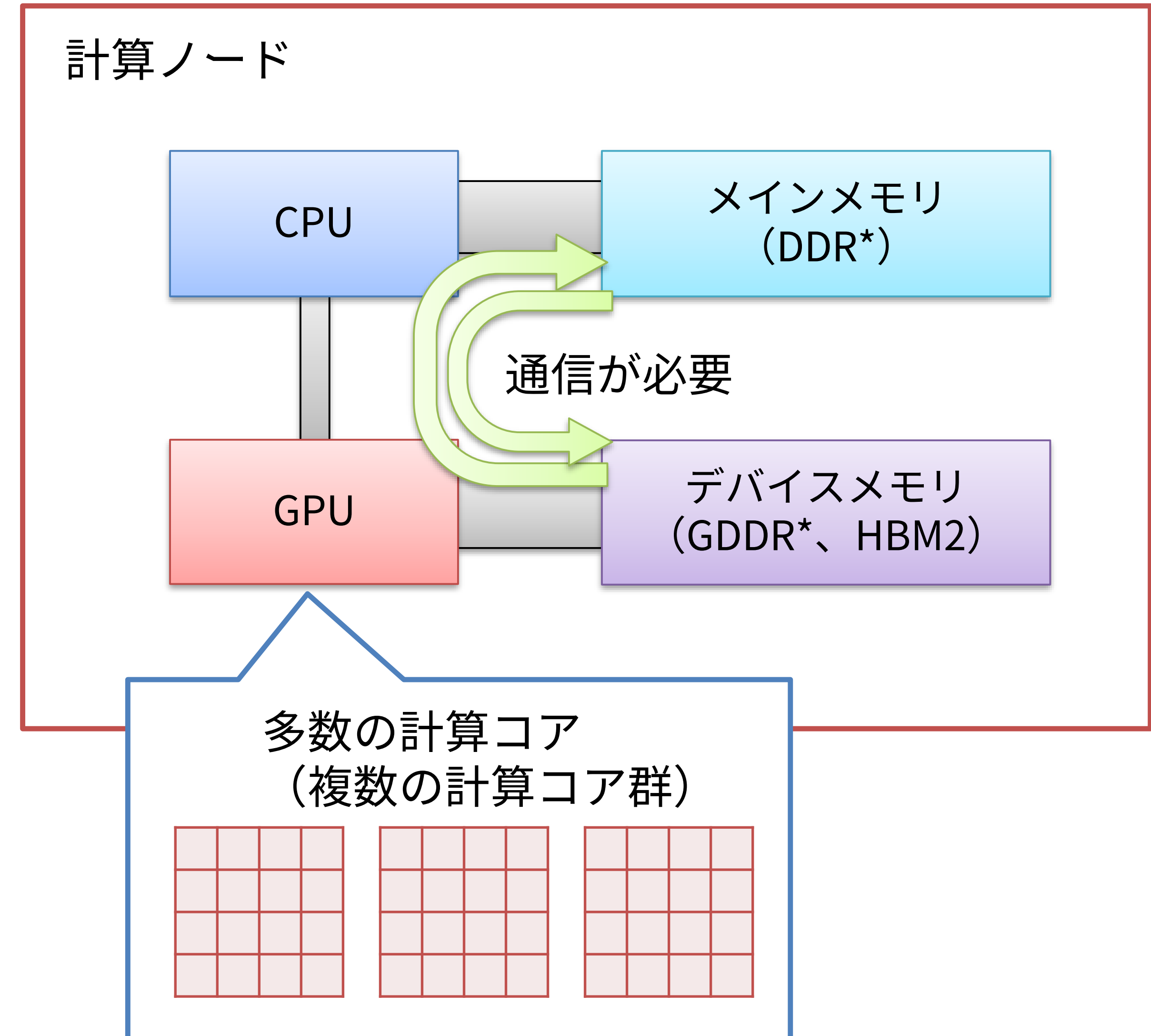
- OpenACCは「どんなプログラムでもGPU化できる」「どんなプログラムでも高速化できる」わけではないが、多くのプログラムに対して有用
  - 自作コードのGPU化を行うことができる上で最も簡単でコストパフォーマンスの高い（労力に対して十分な性能が得られる）方法
- CUDAでしか書けない処理も多い
  - 「とにかく高い並列度で一気に計算すれば良い」という典型的なGPU向けプログラムではOpenACCでも十分高い性能が得られる
  - CUDAを使うべきプログラム
    - GPU上の高速共有メモリやシャッフル命令を意識したアルゴリズム
    - インスタンスIDを意識したアルゴリズム
      - 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）、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/

OpenACCの想定する典型的なGPU計算環境

- 「とりあえず」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（あまり速くない）が必要



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向けの実行ファイルと同様にそのまま実行可能
     - ./a.out
   - 特別なプログラムを介したりする必要はない
     - LD_LIBRARY_PATHなどの対応は必要（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

単純なOpenACCプログラムの例（配列の定数倍計算）

- C (vector0.c)

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

- Fortran (vector0.f90)

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

OpenACC並列化対象
＝GPU上で実行される

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

OpenACCの実行モデル

- 指定した部分だけがGPU上で実行される

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

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

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

GPU上の計算コアにより並列計算される

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

コンパイル例

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

- -ta
  - Pascalならcc60、Voltaならcc70
  - sx40ではcc70
- -tp
  - sx40ではskylake

指定が合わないと実行できない。

- -cpp
  - プリプロセッサ処理の有効化
  - あとで#ifdefなどを使う際に必要

```
pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -o vector0_c_acc vector0.c
main:
     16, Generating implicit copyout(v2[:])
         Generating implicit copyin(v1[:])
     17, Loop is parallelizable
         Generating Tesla code
         17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */

pgfortran -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -cpp -o vector0_f_acc vector0.f90
main:
     13, Generating implicit copyout(v2(:))
         Generating implicit copyin(v1(:))
     14, Loop is parallelizable
         Generating Tesla code
         14, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

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

※出力情報の具体的な読み方は後述する

実行例

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

C版の場合

```
$ ./a.out
 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00                ←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

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

  - コンパイルオプションの想定環境と実行環境が異なると実行時エラーする
  - 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

GPUが仕事をしていることを確認する 1

- 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

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

←GPU上での計算に関する情報

←CPU-GPU間の通信に関する情報

GPUが仕事をしていることを確認する　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=17 device=0 threadid=1 num_gangs=1 num_workers=1 vector_length=32 grid=1 block=32

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=21 device=0 threadid=1 variable=v2 bytes=80

4: regionのentry/exit
- 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=17 device=0 threadid=1
- Implicit wait file=/path/to/source.c function=main line=21 device=0 threadid=1

8: wait/sync
- 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=17 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=21 device=0 threadid=1

演習

- サンプルプログラム（vector0.c または vector0.f90）をコンパイルして実行してみる
- 実行結果が確認できたら、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
  - export PGI_ACC_TIME=1 を実行してからプログラムを実行する
  - unset PGI_ACC_TIME により解除できる

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（directive）により全てを記述する
  - 指示文：コンパイラに対して指示を行う特殊なコメント
    - C/C++：#pragma acc ~
    - Fortran：!$acc ~
    - 基本的には「無視してしまっても問題が起きない文」
      - コンパイラが対応していない場合もコンパイルと実行自体は可能、もちろんGPUは使えない
- 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 (vector0.c)

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

- Fortran (vector0.f90)

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

（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 -tp=skylake 以下省略
main:
   16, Generating implicit copyout(v2[:])
       Generating implicit copyin(v1[:])
   17, Loop is parallelizable
       Generating Tesla code
       17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

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

アクセラレータ(GPU)向けのカーネルが生成された
Tesla(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

- Fortran

```
pgfortran -Minfo=accel -acc -ta=tesla:cc70 以下省略
main:
   13, Generating implicit copyout(v2(:))
       Generating implicit copyin(v1(:))
   14, Loop is parallelizable
       Generating Tesla code
       14, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

※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

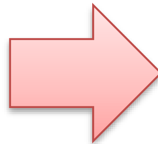
- どのように判断・処理されたのかを確認することは非常に重要

※blockidx や threadidxについては後述

手動による適用：C版

- C (vector0.c)

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



- C (vector1.c)

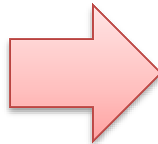
```c
 4 int main(int argc, char **argv)
 5 {
 6   int i, n=10;
 7   double v1[10], v2[10];
 8
 9   for(i=0; i<n; i++){
10     v1[i] = (double)(i+1);
11     v2[i] = 0.0;
12   }
13
14 #pragma acc kernels copyout(v2[:]) copyin(v1[:])
15 #pragma acc loop gang, vector(32)
16   for(i=0; i<n; i++){
17     v2[i]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

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

手動による適用：Fortran版

- Fortran (vector0.f90)

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



- Fortran (vector1.f90)

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

※end loopは不要（書いてもエラーにはならないようだ）

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

参考：指示文の継続行

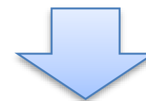
- 指示文行を次の行に継続させることも可能
  - 長くなってしまったときなどに
- C/C++とFortranで少し違うので注意

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

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

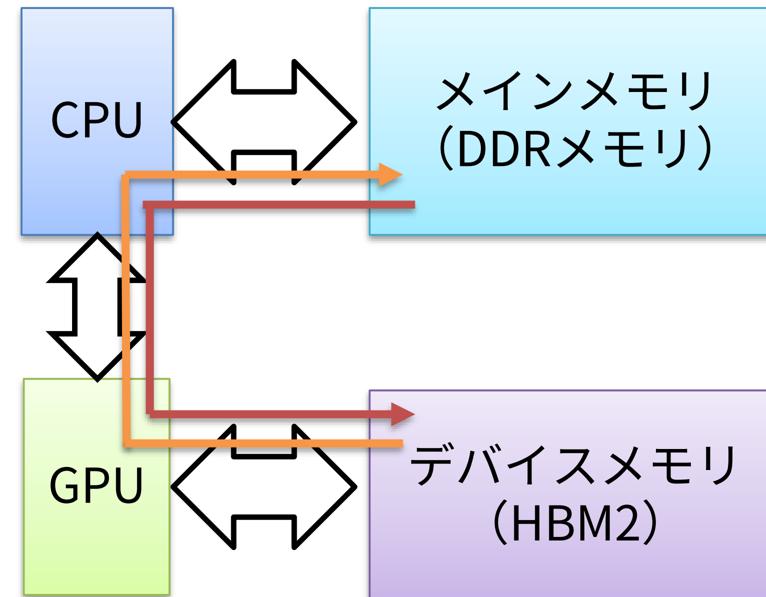
```
!$acc kernels copyout(v2(:)) copyin(v1(:))
!$acc loop gang, vector(32)
  do i=1, n
     v2(i) = v1(i) * 2.0d0
  enddo
!$acc end kernels
```



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

CPU-GPU間のデータ転送

- CPUとGPUは個別のメモリを持っており、直接相手側のメモリにアクセスできない
  - 例外あり、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++ではコンパイラが長さを認識できない配列を扱うことが多いため注意が必要
  - GPUカーネルが生成されなかったり、実行時にエラーしたりする原因となる



データ転送に関する指示節

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

※OpenACC2.5からは常に「present_or_*」の挙動となり、存在していれば使い回してくれる。
実際にどう扱われるかはコンパイル時のメッセージやPGI_ACC_NOTIFYを用いれば確認できる。

※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;
    }
  }

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

  return 0;
}
```

(vector2.c)

```fortran
program main

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

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

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

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

end program main
```

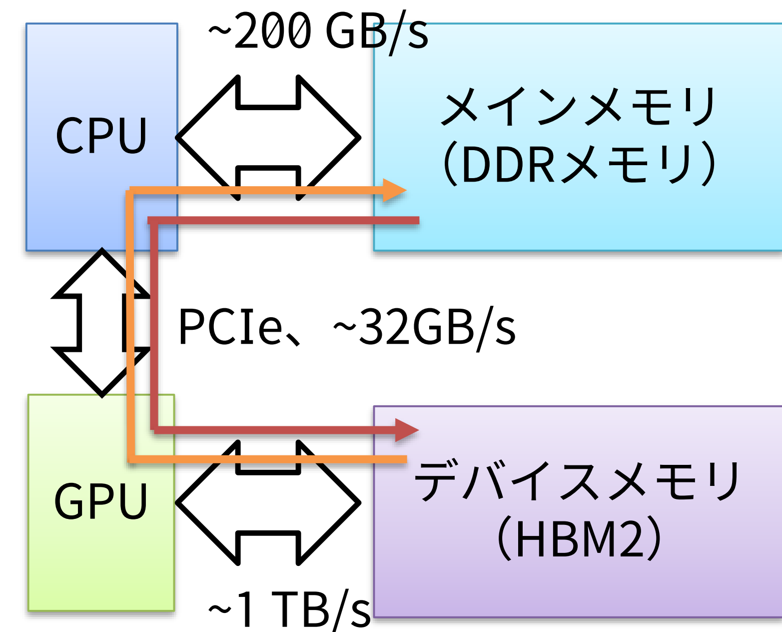
(vector2.f90)

繰り返し
- デバイスメモリの生成・転送
- 計算
- デバイスメモリの破棄・転送

デバイスメモリの生成・破棄と転送を繰り返してしまう（本当は不要な処理、余計な時間がかかる）

データ転送速度

- メインメモリやデバイスメモリの転送速度に対してCPU-GPU間のデータ転送速度はずっと低速、頻繁な通信は避けたい
  - CPU - メインメモリ
    - DDR4の場合、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超（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);
  }

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

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

  return 0;
}
```

(vector3.c)

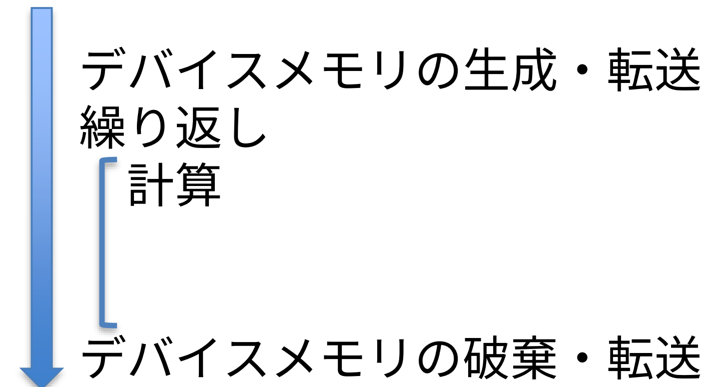
```fortran
program main

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

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

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

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

end program main
```

(vector3.f90)

※コンパイラが判断に失敗して内側ループでもデータを転送しようとする場合はkernelsにpresent節を加えると良い

acc kernels present(v1)

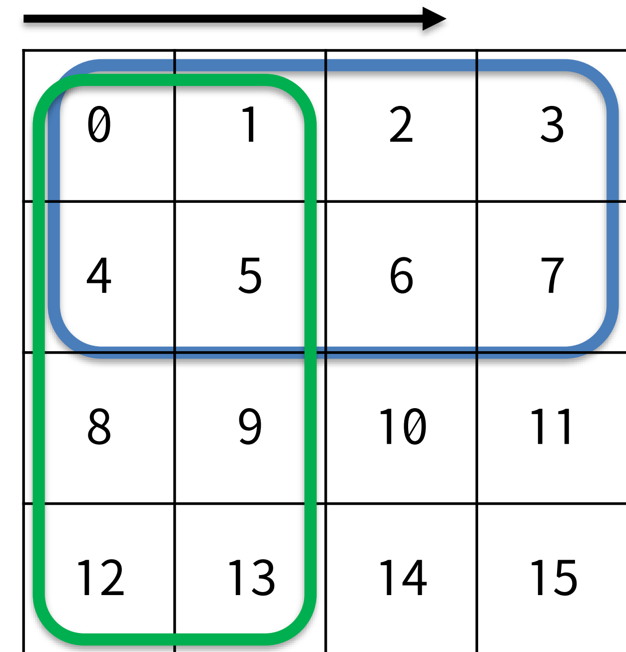
（v1のアクセスに間接参照がある場合などに効果的）



生成・破棄と転送は最初と最後にのみ行われ、無駄がない

多次元配列の送受信

- 多次元配列の送受信も可能
  - ただし、連続したメモリの範囲しか扱えない
    - 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



- 青は連続している：まとめて一度に送れる
- 緑は連続していない：まとめて一度に送れない

データ送受信時の注意点・補足事項

- 部分転送時の範囲には変数を用いても良い
- 配列長については注意が必要
  - 動的に確保した配列などコンパイル時に配列の長さが分からないものは長さを明示する必要がある
    - 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

```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;
  }
```

```fortran
  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
- ➢ copyin(v1[n]) および copyin(v1(n)) を加えると正しく動作する
- ➢ 具体例は vector12.c/f90 および vector13.c/f90 を参照（コンパイル時に出力される情報を比較してみよう）

注意：Deep copyについて

- 動的な要素を持つ集合的な要素をまとめて転送しようとする問題が起きることがある

  これをDeep copyと呼ぶ

  – 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Deep copy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Deep copyできない

- 解決方法
  – 必要な分だけ手動でcopyする
  – 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、色々と制限があるため注意が必要
- この講習会では問題が起きないような単純なデータのみ扱う

演習

- 単純なサンプルプログラム（vector2.c, vector2.f90）と、data指示文を挿入したプログラム（vector3.c, vector3.f90）を実行し、比較してみよう
  - 以下は九大ITOサブシステムB（Tesla P100）にて実行した例
  - 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、回数が違うこと、実行時間が違うことがわかるはずである

```
Accelerator Kernel Timing data
/home/usr0/m70000a/work/201906/vector2.c
  main  NVIDIA  devicenum=0
    time(us): 134
    14: compute region reached 10 times
        15: kernel launched 10 times
            grid: [1]  block: [32]
            elapsed time(us): total=188 max=39 min=16 avg=18
    14: data region reached 20 times
        14: data copyin transfers: 10
             device time(us): total=66 max=12 min=5 avg=6
        18: data copyout transfers: 10
             device time(us): total=68 max=14 min=6 avg=6
```

```
Accelerator Kernel Timing data
/home/usr0/m70000a/work/201906/vector3.c
  main  NVIDIA  devicenum=0
    time(us): 25
    15: compute region reached 10 times
        16: kernel launched 10 times
            grid: [1]  block: [32]
            elapsed time(us): total=174 max=40 min=14 avg=17
    13: data region reached 2 times
        13: data copyin transfers: 1
             device time(us): total=12 max=12 min=12 avg=12
        21: data copyout transfers: 1
             device time(us): total=13 max=13 min=13 avg=13
```

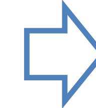
データの更新

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

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

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

並列化ループの途中で値を確認したい・更新したい
たとえばここでv1の値を出力したら何が見えるのだろうか？



計算前の値が見える

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

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

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

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

(vector4.c)

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

(vector4.f90)

データの更新：update指示文

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

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

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

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

(vector5.c)

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

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

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

(vector5.f90)

OpenACCにおける変数や配列の共有・非共有の扱い

- スカラ変数はfirstprivateとなる
  - 並列化範囲外の値を引き継ぎ、互いに干渉しあわない
- 配列はデバイスメモリにて共有される
  - 互いに干渉する
  - private指示節により変更することも可能

- 参考：OpenMPの場合
  - private/shared指示節で指定
  - 何も指定しないとsharedとなりスレッド間で干渉する
  - Fortranのみ並列化範囲内の逐次ループのループカウンタはprivate扱い

内容

- 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
- CG法を題材としてOpenACCの基本を学ぶ
  - 少し複雑なコードのOpenACC化について
- 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

ループ並列化方法の指定

- 現実のプログラムではコンパイラが全てのループの並列化の判断を行うのは難しい、プログラマが並列化の判断をする必要がある
- loop指示文
  - 並列化対象ループを指定する
  - さらに以下の指示節と組み合わせることで動作の制御が可能
  - independent指示節とseq指示節
    - 対象ループを並列実行するか逐次実行するかを明示する
    - 強制力があり、コンパイラによる判断は行われなくなる
  - collapse(n)指示節：collapse(2), collapse(3)など
    - 多重ループをまとめて並列化する
    - 並列度の低い階層ループの並列化などに極めて重要
  - reduction指示節：reduction(+:a）など
    - 計算結果の集約などを行う
    - 多くの場合はコンパイラが正しく判断してくれるため書く必要はない

三重ループ構造による単純な行列積プログラムの例

- C (matmul0.c)

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

- Fortran (matmul0.f90)

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

n=10で実行してみた

C

```
36: compute region reached 1 time
    37: kernel launched 1 time
        grid: [1]  block: [1]
        elapsed time(us): total=184 max=184 min=184 avg=184
```

Fortran

```
37: compute region reached 1 time
    40: kernel launched 1 time
        grid: [1x10]  block: [128]
        elapsed time(us): total=37 max=37 min=37 avg=37
```

どうやら実行時間に大きな差があるようだ、何故だろう？

三重ループ構造による単純な行列積プログラムの例

- Fortran (matmul0.f90)

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

➤ copy関係はコンパイラの判断で特に問題はない

```
pgfortran -Minfo=accel -acc -ta=tesla:cc70 ¥
 -tp=skylake -o m0f_f_acc matmul0.f90
main:
37, Generating implicit copyin(a(1:n,1:n))
    Generating implicit copy(c(1:n,1:n))
    Generating implicit copyin(b(1:n,1:n))
38, Loop is parallelizable
39, Loop is parallelizable
40, Complex loop carried dependence of c prevents parallelization
    Loop carried dependence of c prevents parallelization
    Loop carried backward dependence of c prevents vectorization
    Inner sequential loop scheduled on accelerator
    Generating Tesla code
    38, !$acc loop gang ! blockidx%y
    39, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
    40, !$acc loop seq
```

sequential
ループが逐次実行されることを意味する

➤ 配列cの依存関係に関するメッセージは出ているが、3重ループの外側2つが並列化された

三重ループ構造による単純な行列積プログラムの例

- C (matmul0.c)

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

```
pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -o m0c_c_acc matmul0.c
main:
36, Generating implicit copyin(b[:n][:n])
    Generating implicit copy(c[:n][:n])
    Generating implicit copyin(a[:n][:n])
37, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
    Accelerator serial kernel generated
    Generating Tesla code
    37, #pragma acc loop seq
    38, #pragma acc loop seq
    39, #pragma acc loop seq
38, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
39, Complex loop carried dependence of a->->,c->->,b->-> prevents parallelization
    Loop carried dependence due to exposed use of c[i1][i2] prevents parallelization
```

- copy関係はコンパイラの判断で特に問題はない
- 依存関係があり並列化できず、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

loop指示文の追加

- C (matmul1.c)

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

- Fortran (matmul1.f90)

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

- 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

```
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
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
```

- C言語、loop independent指定あり

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

collapse版

- C (matmul2.c)

```
double **a=NULL, **b=NULL, **c=NULL;
// mallocでa,b,cを確保
#pragma acc kernels
#pragma acc loop independent collapse(2)
 for(i=0; i<n; i++){
    for(j=0; j<n; j++){
#pragma acc loop seq
      for(k=0; k<n; k++){
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
```

- Fortran (matmul2.f90)

```
double precision, allocatable :: a(:,:), b(:,:), c(:,:)
allocate(a(n,n), b(n,n), c(n,n))
!$acc kernels
!$acc loop independent collapse(2)
  do i=1, n
     do j=1, n
!$acc loop seq
        do k=1, n
           c(j,i) = c(j,i) + a(k,i) * b(j,k)
        enddo
     enddo
  enddo
!$acc end kernels
```

- ループを融合してから並列化する
- 短いループが多重になっている（ネストしている）際に特に有用
- 実行時間的にはそれぞれのループを並列化した場合とほとんど変わらないことが多い

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

演習：単純な行列積プログラム

- C (matmul1.c)

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

- Fortran (matmul1.f90)

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

実際にコンパイルして実行してみよう
- 並列化されたか？
- 実行時間は短くなったか？
- independentやseqを変更してみるとどうか？
- collapse指定を変更するとどうか？

演習：単純な行列積プログラム：実行例

- C (matmul1.c)

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

- Fortran (matmul1.f90)

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

n=10で実行して比較してみた

C

```
36: compute region reached 1 time
    42: kernel launched 1 time
        grid: [1x10]  block: [128]
        elapsed time(us): total=22 max=22 min=22 avg=22
```

Fortran

```
37: compute region reached 1 time
    43: kernel launched 1 time
        grid: [1x10]  block: [128]
        elapsed time(us): total=34 max=34 min=34 avg=34
```

C版が遅すぎるという問題は解消された（Cの方が高速になった明確な理由まではわからず）

補足：行列（二次元配列）の確保の仕方について

- サンプルソースコードでは見た目をわかりやすくするためにdouble ** 型で配列を確保しmallocを2段階で行いa[i][j]によるアクセスを行ったが、実はこの確保の仕方だと配列のコピー回数が増えてしまう
  - a[i] ごとに転送されてしまう
- double * 型で確保してa[i*N+j]アクセスをするようにすれば配列のコピー回数はまとめて1回になり転送時間が削減できる

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？

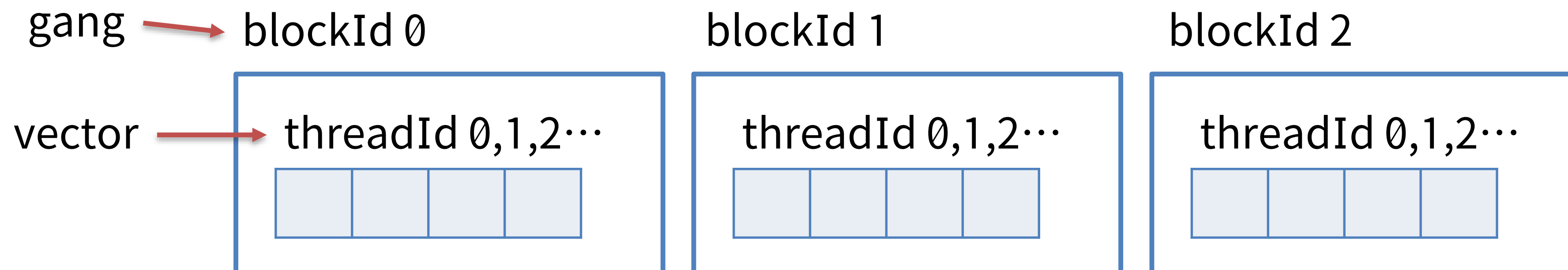
- 実はコンパイル時のメッセージにどのように割り当てるかが出力されていた
  - gang, vector と blockIdx, threadIdx という概念が存在するようだ

```
pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -o vector0_c_acc vector0.c
         17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */

pgfortran -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -o vector0_f_acc vector0.f90
         14, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

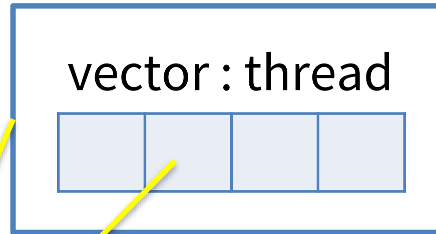
gang, worker, vectorとblockIdx, threadIdx

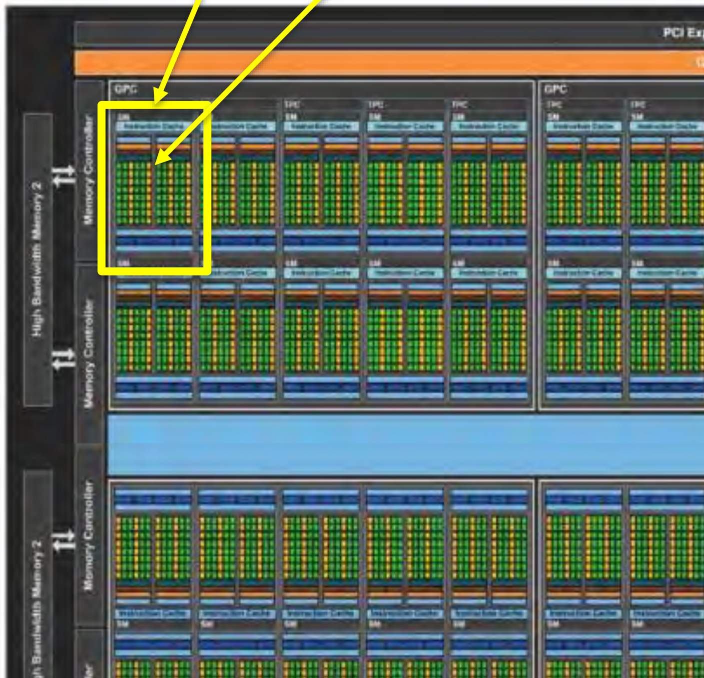
- OpenACCではハードウェアに階層的な並列性があることを想定しており、上位階層から順にgang, worker, vectorとなっている
- これらをどのように組み合わせて実行するかを指定できる

- CUDAの並列実行モデルが階層的になっているため、それにあわせて言語設計された、という方が正しい
  - grid, threadblock, threadの三階層構造
  - OpenACCのおおまかな並列実行モデルの対応付け



ハードウェアとのおおまかな対応付け





- Streaming Multiprocessor(SM)内の並列性はvector、SM単位の並列性はgang
- 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コア単位で動作していることを覚えておくと良い性能が出ることもある
- ()で数字を与えた場合はその数単位で割り当てられる

もう少し詳細に言えば……
- gangはHWレベルで同期できない単位の粗粒度並列性（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

- とりあえず、gangとvectorを意識しておくと高い性能が出やすくなることもあるかも知れない、程度に考えておくと良い

内容

- 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
- CG法を題材としてOpenACCの基本を学ぶ
  – 少し複雑なコードのOpenACC化について
- 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

CG法プログラムのOpenACC化

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

$r_0 = b - Ax_0$
$p_0 = r_0$

for (k = 0; ; k++)

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

if r_{k+1} が十分に小さい then
break

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

結果は x_{k+1}

CG法の計算手順

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

※上付き文字は反復回数に対応

- Ax=b：Aは行列、xとbはベクトル
- z,r,p,qはベクトル
- α・β・ρはスカラー（ベクトルのリダクション結果）

主要コード

- 行列やベクトルに対する単純な計算ばかりで構成されているため、並列化・OpenACC化は容易
  - ★：行列要素同士のコピーや四則演算
  - ★：集約演算（reduction、dot product）
  - 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化手順例： 1.各計算部の並列化

- 並列化できることがわかっているループに指示文を挿入してみる
  - 前頁の各行列計算・ベクトル計算ごとに指示文を挿入

- 行列ベクトル積を並列化する例

```c
    // {q} = [A]{p}
#pragma acc kernels copyin(A[N*N])
#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];
      }
    }
```

```fortran
! {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
```

- 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（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では自動的に判断されてcopyされる

CG法のOpenACC化手順例： 1.各計算部の並列化（つづき）

- 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節が必要となることが多い
    - 配列の参照先アドレスが重複する可能性を考慮するためか、並列化してくれない
  - 配列長が認識できずにデータ転送に躓くことがあるため適切にcopy指示節を追加する
    - 特に、C言語版の行列Aに要注意
    - in/outを意識してcopyin/copyoutを行えば無用な配列コピーをなくすことができる
  - （reductionが適切に生成されているかを確認する）
    - （問題なく生成されると思って良いが、念のため）

- 各計算部に指示文を入れれば正しくGPU上で実行できるようになる
  - OpenACCの便利なところ
    - 全てのループに指示文を入れていない状態でもコンパイル・実行が可能（1ループだけでもOK）
    - step by stepで少しずつ並列化できるためデバッグや性能評価が行いやすい

実習：CG法のOpenACC化： 1.各計算部の並列化

- 実際にOpenACC化してみよう
  - サンプルコード cg2.c または cg2.f90 に指示文を挿入する
    - Fortran版ではコンパイル時に-cppオプションを加える必要あり
      - 出力部の調整のために #if を使っているため
  - PGI_ACC_TIME環境変数を設定し、実行時間を確認してみよう
    - 余裕がある人はOpenMP版も作って時間を比較してみよう
  - コンパイル例
    - C言語版

CPU向け（OpenMP並列化）

```
pgcc -tp=skylake -mp -o cg2c_c cg2.c
```

GPU向け（OpenACC並列化）

```
pgcc -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -mp -cpp -o cg2c_c_acc cg2.c
```

- Fortran版

CPU向け（OpenMP並列化）

```
pgfortran -tp=skylake -mp -cpp -o cg2f_f cg2.f90
```

GPU向け（OpenACC並列化）

```
pgfortran -Minfo=accel -acc -ta=tesla:cc70 -tp=skylake -mp -cpp -o cg2f_f_acc cg2.f90
```

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

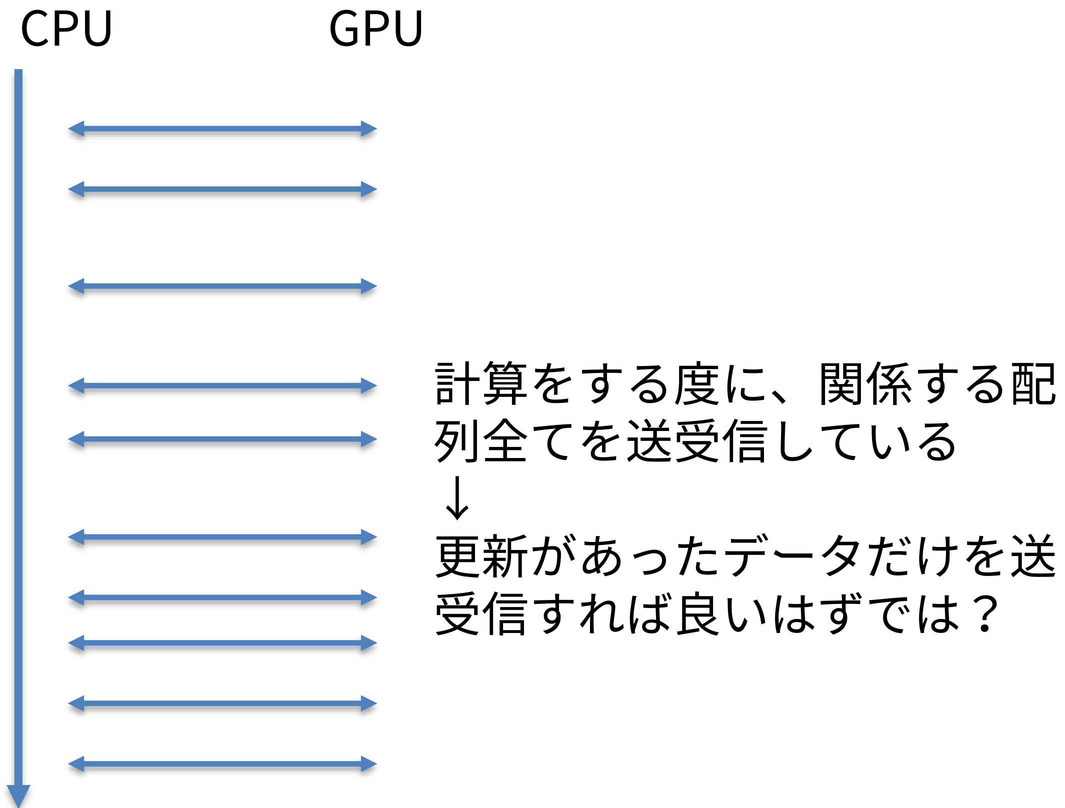
←計算結果ベクトルXと最初に設定したベクトルXの比較

- cg2.cとcg2.f90はOpenACC指示文を挿入していないソースコード
- kernelsとloopを適切に指定したのがcg3.cとcg3.f90
- cg3.cではコンパイラが行列Aの大きさに迷わないようcopyinも追加している
- #define DEBUGOUTをコメントアウトすれば行列やベクトルのデバッグ出力は消える、大きな問題サイズで試す場合は消すと良い

正しい結果は得られているが……

- PGI_ACC_TIME等で確認すると、データ転送回数が多いことがわかるはず
  - 全体の実行時間に対してデータ転送時間が無視できない長さになる可能性がある
- 現在のデータ転送状況イメージ

```
Compute r^(0) = b-[A]x^(0)
for i= 1, 2, …
  solve [M]z^(i-1) = r^(i-1)
  ρ_(i-1) = r^(i-1)  z^(i-1)
  if i=1
    p^(1) = z^(0)
  else
    β_(i-1) = ρ_(i-1) / ρ_(i-2)
    p^(i) = z^(i-1)  + β_(i-1) z^(i)
  endif
  q^(i) = [A]p^(i)
  α_i = ρ_(i-1) /p^(i) q^(i)
  x^(i) = x^(i-1)  + α_i p^(i)
  r (i) = r^(i-1)  - α_i q^(i)
  check convergence |r|
end
```



実習：CG法のOpenACC化： 2.データ転送の最適化

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

```
#pragma acc data copyin(？) copyout(？)          !$acc data copyin(？) copyout(？)
Compute r^(0) = b-[A]x^(0)
for i= 1, 2, …
  solve [M]z^(i-1) = r^(i-1)
  ρ_i-1 = r^(i-1)  z^(i-1)
  if i=1
    p^(1) = z^(0)
  else
    β_i-1 = ρ_i-1 / ρ_i-2
    p^(i) = z^(i-1) + β_i-1 z^(i)
  endif
  q^(i) = [A]p^(i)
  α_i = ρ_i-1 /p^(i) q^(i)
  x^(i) = x^(i-1) + α_i p^(i)
  r (i) = r^(i-1) - α_i q^(i)
  check convergence |r|
end
                                                  !$acc end data
```

※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、特に気にしなくてよい

  - 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
  - 少しずつ最適化、を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
  - もちろん、わかっていればいきなりデータ通信の最適化を行っても良い
  - まずはimplicitで転送されていたものを手動で転送することを考えてみよう
  - ※実際にGPUを使うことでCPUより大幅な高速化が得られるのは大規模な行列の場合
    - 小規模な行列では時間が短すぎてあまり違いがわからない

```c
#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];
      }
    }
```

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

```fortran
!$acc data copyin(z(N), dd(N), r(N), p(N), q(N), A(N,N)) copy(x(N))
  do iter=1, maxiter
    ……
   ! {q} = [A]{p}
   !$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.cとcg4.f90はcg3.cとcg3.f90にdata指示文を加えて最初に全てのデータをGPUへ転送したもの
- cg5.cとcg5.f90はさらにkernels指示文にpresent節を加えてデータ転送が不要であることを明示したもの

内容

- 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
- CG法を題材としてOpenACCの基本を学ぶ
  – 少し複雑なコードのOpenACC化について
- 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

OpenACCプログラム最適化のための一般的なヒント

- 正しく動作するようになったら、速く動作させることを考えてみよう
  - 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
  - CPU-GPU間のデータ送受信は少なくする
  - 最内ループでvectorによる連続メモリアクセスを行うようにする
    - コアレスなメモリアクセスになりメモリアクセス性能を発揮できる

```
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];
  }
}
```

```
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);
  }
}
```

  - 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

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）で関数を実行する場合はroutine指示文が必要

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

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

```c
// 呼び出し対象の関数
#pragma acc routine
void calc(int n, double *v)
{
  int i;

  for(i=0; i<n; i++){
    v[i] = v[i] * 2.0;
  }
}
```

(vector6.c)

```fortran
! 呼び出し対象の関数
module mod
contains
subroutine calc(n,v)
!$acc routine
  integer :: n
  double precision :: v(*)

  do i=1, n
     v(i) = v(i) * 2.0d0
  enddo
end subroutine calc
end module mod
```

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

(vector6.f90)

- ➢ Cでは関数名の前に、Fortranでは関数名の次に指示文を挿入

OpenACC並列化部における関数の呼び出し：関数内での並列実行

- 関数内のループを並列化する場合はさらに指示文を追加する必要がある

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

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

```c
// 呼び出し対象の関数
#pragma acc routine 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;
  }
}
```

(vector7.c)

```fortran
! 呼び出し対象の関数
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
```

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

(vector7.f90)

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

デバッガやプロファイラの活用について

- OpenACC用のデバッガやプロファイラは提供されていないが、PGIやNVIDIAのツールがある程度利用できる
  - pgdbg, pgdebugを使うとCPUが処理している部分のデバッグができる
    - pgiコンパイラと同時にインストールされており、module loadしてpgiコンパイラが使える状態にしてあれば使える
  - cuda-gdbを使うとGPUが処理している部分のデバッグ……をするのは少し難しいが、プログラムに問題がある場合に具体的にどの部分に問題があるかがわかることもある
    - PGIコンパイラではなくシステムにインストールしてあるCUDAのディレクトリから起動する
    - 使い方：`/opt/nvidia/cuda10.1/bin/cuda-gdb ./a.out`

デバッガやプロファイラの活用について：nvprof

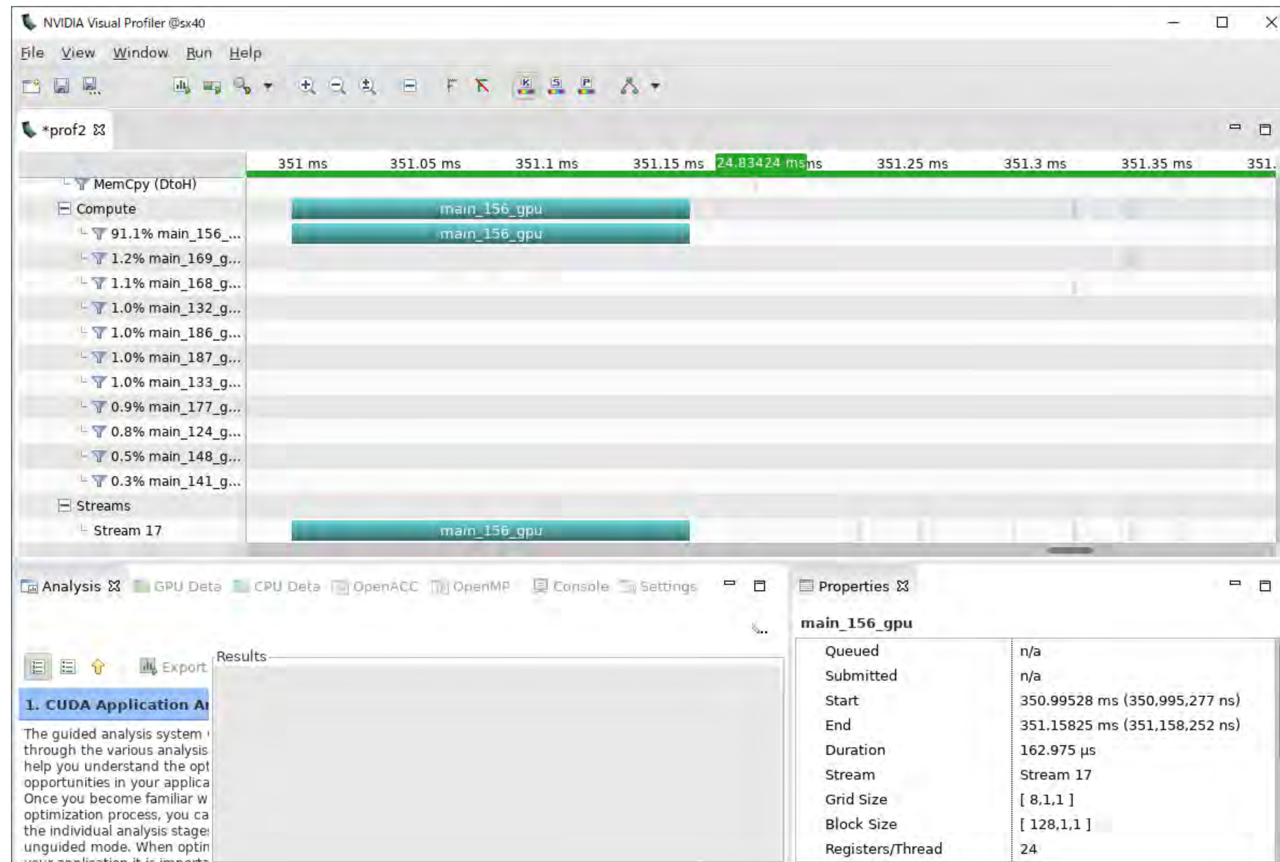
- CUDA用のプロファイラnvprofを使うとOpenACCカーネルのプロファイルが取れる
  – 内部ではCUDAコードが動いているということ
  – 起動例：~/opt/pgi/linux86-64-llvm/2019/cuda/10.1/bin/nvprof ./cg3c_c_acc
  – 出力例：

```
==102071== Profiling application: ./cg3c_c_acc
==102071== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.46%  113.57us        53  2.1420us  1.3760us  2.8480us  [CUDA memcpy HtoD]
                   19.31%  38.848us        24  1.6180us  1.5360us  2.2720us  [CUDA memcpy DtoH]
                    3.44%  6.9120us         3  2.3040us  2.2720us  2.3680us  main_156_gpu
                    2.56%  5.1520us         3  1.7170us  1.5360us  2.0800us  main_133_gpu__red
                    省略
      API calls:   66.50%  175.94ms         1  175.94ms  175.94ms  175.94ms  cuDevicePrimaryCtxRetain
                   21.11%  55.862ms         1  55.862ms  55.862ms  55.862ms  cuDevicePrimaryCtxRelease
                    8.31%  21.984ms         1  21.984ms  21.984ms  21.984ms  cuMemHostAlloc
                    2.64%  6.9922ms         1  6.9922ms  6.9922ms  6.9922ms  cuMemFreeHost
                    省略
 OpenACC (excl):   87.46%  22.124ms         3  7.3748ms  5.6300us  22.113ms  acc_enter_data@cg3.c:122
                    1.72%  434.15us         1  434.15us  434.15us  434.15us  acc_device_init@cg3.c:122
                    0.93%  234.31us        12  19.525us  18.573us  21.648us  acc_enqueue_upload@cg3.c:175
                    0.70%  177.40us         9  19.710us  18.031us  22.402us  acc_enqueue_upload@cg3.c:130
                    0.69%  174.21us         6  29.035us  18.786us  66.957us  acc_enqueue_upload@cg3.c:122
                    省略
```

末尾の数字がソースコードの
行番号に対応している

デバッガやプロファイラの活用について：nvvp（Visual Profiler）

- 測定方法：~/opt/pgi/linux86-64-llvm/2019/cuda/10.1/bin/nvprof -o prof ./cg3c_c_acc
  - -oで指定したファイルに結果が出力される、これをnvvpで読む
- 起動方法：
  ~/opt/pgi/linux86-64-nollvm/2019/cuda/10.1/bin/nvvp -vm ~/opt/pgi/linux86-64-llvm/2019/java/jre1.8.0_112/bin/java
- 画面例



- 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べば結果を見ることができる

- 左図に表示されているmain_156_gpuは、156行目にある行列ベクトル積部分に対応

その他、今回は触れなかった情報

- CUDAやMPIとの連携
  - OpenACCと外部とでデータ（ポインタ）をやりとりする方法を提供
  - host_data, use_device, deviptr指示子などを活用する
  - GPUデバイスメモリを直接使えるMPIと組み合わせると楽
  - CUDAやCUDA向けライブラリとOpenACCを組み合わせると生産性と性能の両立がしやすい
    - ほんのちょっとした処理にCUDAを使うのはメンドクサイ
- CUDA Unified Memory
  - CPUとGPUが連続したメモリアドレス空間を利用する技術
  - データ転送を記述しなくても必要に応じてCPU-GPU間のデータ転送が勝手に行われるため、プログラミングが容易になる
  - 幾つかの条件（制限）があり、性能にもペナルティが生じる
  - コンパイル時に-ta=tesla:managedオプションをつけるだけで良い
  - Deep copyについて気にしなくてもよくなる
  - （性能を追求したい高性能計算分野の研究者はあまり興味が無い？）