第86回お試しアカウント付き 並列プログラミング講習会 「GPUプログラミング入門」

星野哲也 (助教) hoshino@cc.u-tokyo.ac.jp

2017年10月4日 (水)

東京大学情報基盤センター

スケジュール

- 09:30 - 10:00　受付
- 10:00 - 12:00　Reedbush-Hログイン、GPU入門
- 13:30 - 15:00　OpenACC入門
- 15:15 - 18:00　OpenACC演習

事前準備

1. Reedbushにログインする（別資料を参照）
   ※ログインノードは-U/-H共通
2. 利用上の注意

2. Reedbush 利用上の注意

- ディレクトリの扱いについて
  - ログイン時のディレクトリ（/home/gt00h/txxxxx）にはログイン時に最低限必要なファイルのみを置く
  - 作業には/lustre以下のディレクトリ（/lustre/gt00h/txxxxx）を使う
  - /homeに置いたファイルは計算ノードから参照できない

- コンパイルの為の準備
  - moduleコマンドを使ってコンパイルのための準備をする
    - モジュールをロード：module load 対象モジュール名
      - 環境変数PATHなどが書き換わり、コンパイラやライブラリが利用可能に
      - バージョンを指定する場合：module load 対象モジュール名/バージョン
    - 使用可能なモジュールの一覧を表示：module avail
    - 使用中のモジュールを確認：module list

詳細は午後説明します

moduleの切替

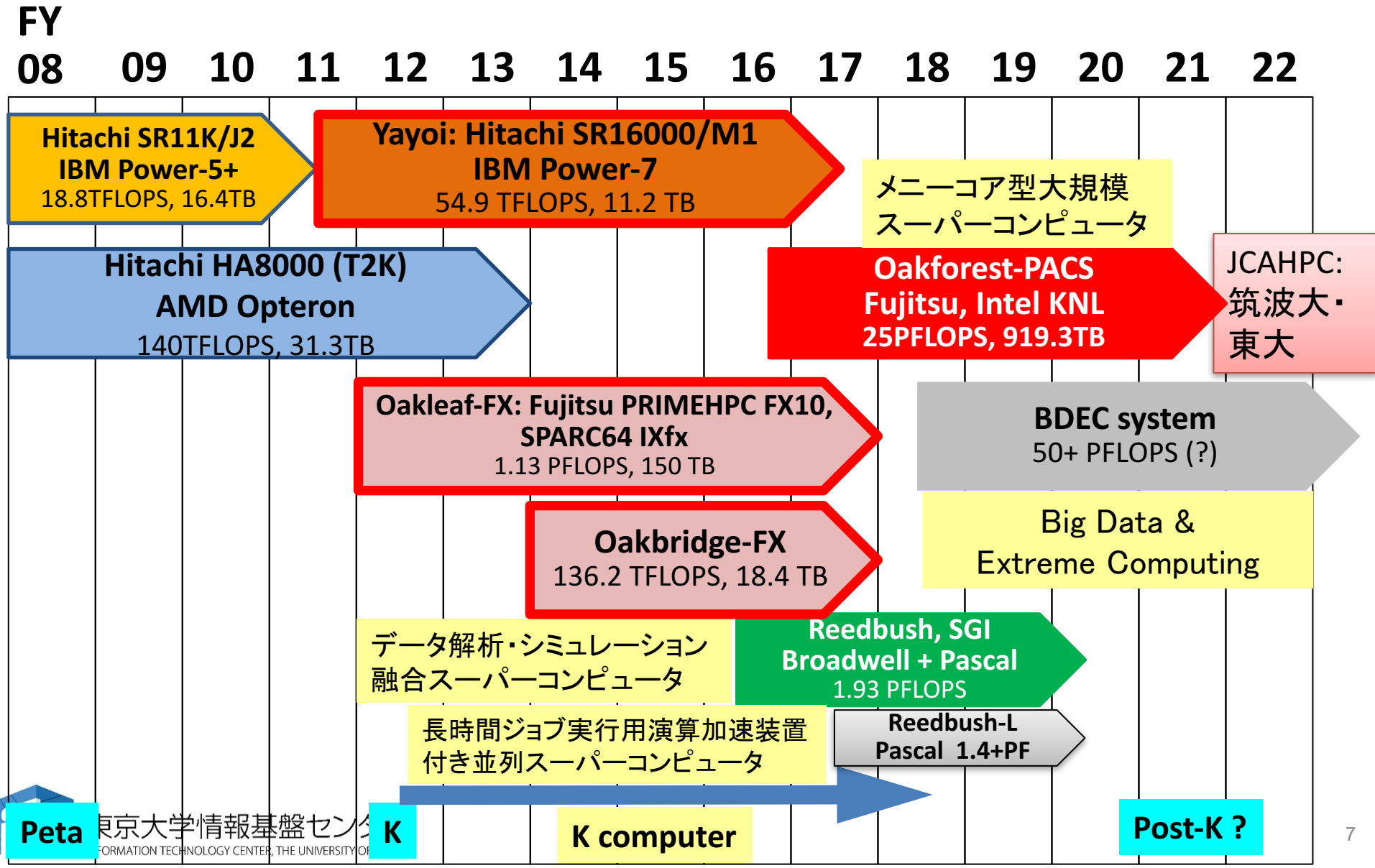
- CUDA開発環境を使う場合
  - module load cuda
- PGIコンパイラを使う場合
  - （OpenACCやCUDA Fortranを使う場合）
  - module load pgi
- Intelコンパイラを使う場合
  - module load intel
- MPIを使う場合（コンパイラに追加してloadする）
  - module load mvapich2-gdr/2.2/{gnu,intel,pgi}
  - module load openmpi-gdr/2.0.2/{gnu,intel,pgi}
- ジョブ実行時にも同じmoduleをloadすること
- 組み合わせて使用しても良い
  - 順序に注意、環境変数PATHやLD_LIBRARY_PATHなどを確認する

詳細は午後説明します

当センターの運用システム

東大センターのスパコン

2基の大型システム，6年サイクル



4システム運用中

- Oakleaf-FX (富士通 PRIMEHPC FX10)
  - 1.135 PF, 京コンピュータ商用版, 2012年4月 ～ 2018年3月

- Oakbridge-FX (富士通 PRIMEHPC FX10)
  - 136.2 TF, 長時間実行用（168時間）, 2014年4月 ～ 2018年3月
- Reedbush (SGI, Intel BDW + NVIDIA P100 (Pascal))
  - データ解析・シミュレーション融合スーパーコンピュータ
  - 3.361 PF, 2016年7月～ 2020年6月
  - 東大ITC初のGPUシステム (2017年3月より), DDN IME (Burst Buffer)

- Oakforest-PACS (OFP) (富士通、Intel Xeon Phi (KNL))
  - JCAHPC (筑波大CCS＆東大ITC)
  - 25 PF, TOP 500で6位 (2016年11月) (日本で1位)
  - Omni-Path アーキテクチャ, DDN IME (Burst Buffer)

東京大学情報基盤センター スパコン（1/3）

Fujitsu PRIMEHPC FX10 （FX10スーパーコンピュータシステム）

Total Peak performance	: 1.13 PFLOPS
Total number of nodes	: 4,800
Total memory	: 150TB
Peak performance per node	: 236.5 GFLOPS
Main memory per node	: 32 GB
Disk capacity	: 2.1 PB
SPARC64 IXfx 1.848GHz	

2012年7月~2018年3月（予定）

Oakbridge-FX

：長時間ジョブ用のFX10
ノード数：24～576
制限時間：最大168時間
（1週間）

東京大学情報基盤センター スパコン（2/3）

Reedbush（SGI Rackable クラスタシステム）

Reedbush-U (2016/7/1 ~)
- 理論性能：508TFlops
- ノード数：420
- ノード構成：Intel Xeon Broadwell x2

Reedbush-H (2017/3/1 ~)
- 理論性能：1418TFlops
- ノード数：120
- ノード構成：Intel Xeon Broadwell x2 + **NVIDIA P100 GPU x2**

Reedbush-L (**2017/10/1 ~**)
- 理論性能：1435TFlops
- ノード数：64
- ノード構成：Intel Xeon Broadwell x2 + **NVIDIA P100 GPU x4**

東京大学情報基盤センター スパコン（3/3）

筑波大学計算科学研究センターと共同運用

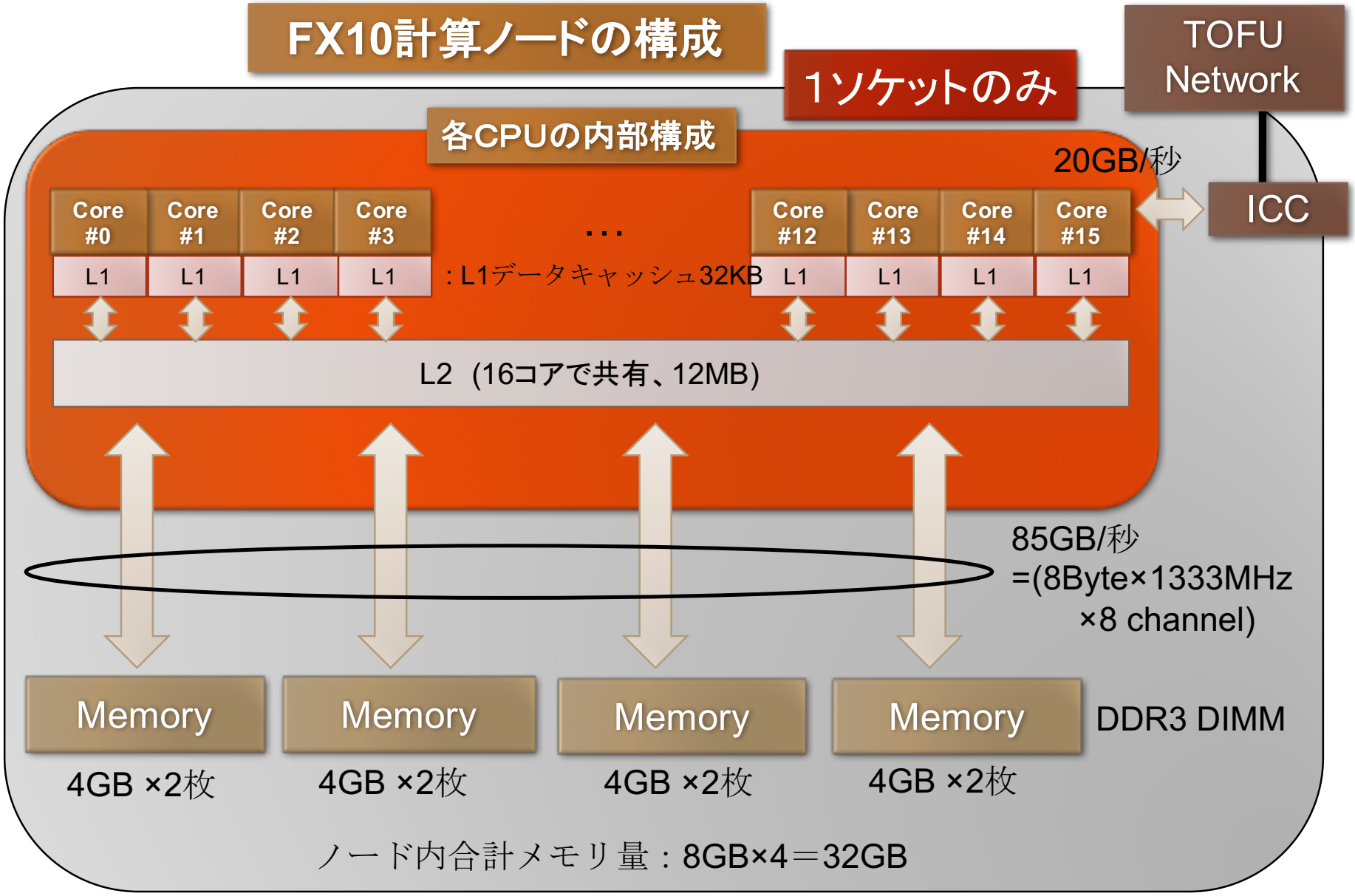
Oakforest-PACS （Fujitsu PRIMERGY CX600）

Total Peak performance	: 25 PFLOPS
Total number of nodes	: 8,208
Total memory	: 897.7 TB
Peak performance per node	: 3.046 TFLOPS
Main memory per node	: 96 GB (DDR4)
	+ 16 GB(MCDRAM)
Disk capacity	: 26.2 PB
File Cache system (SSD)	: 960 TB

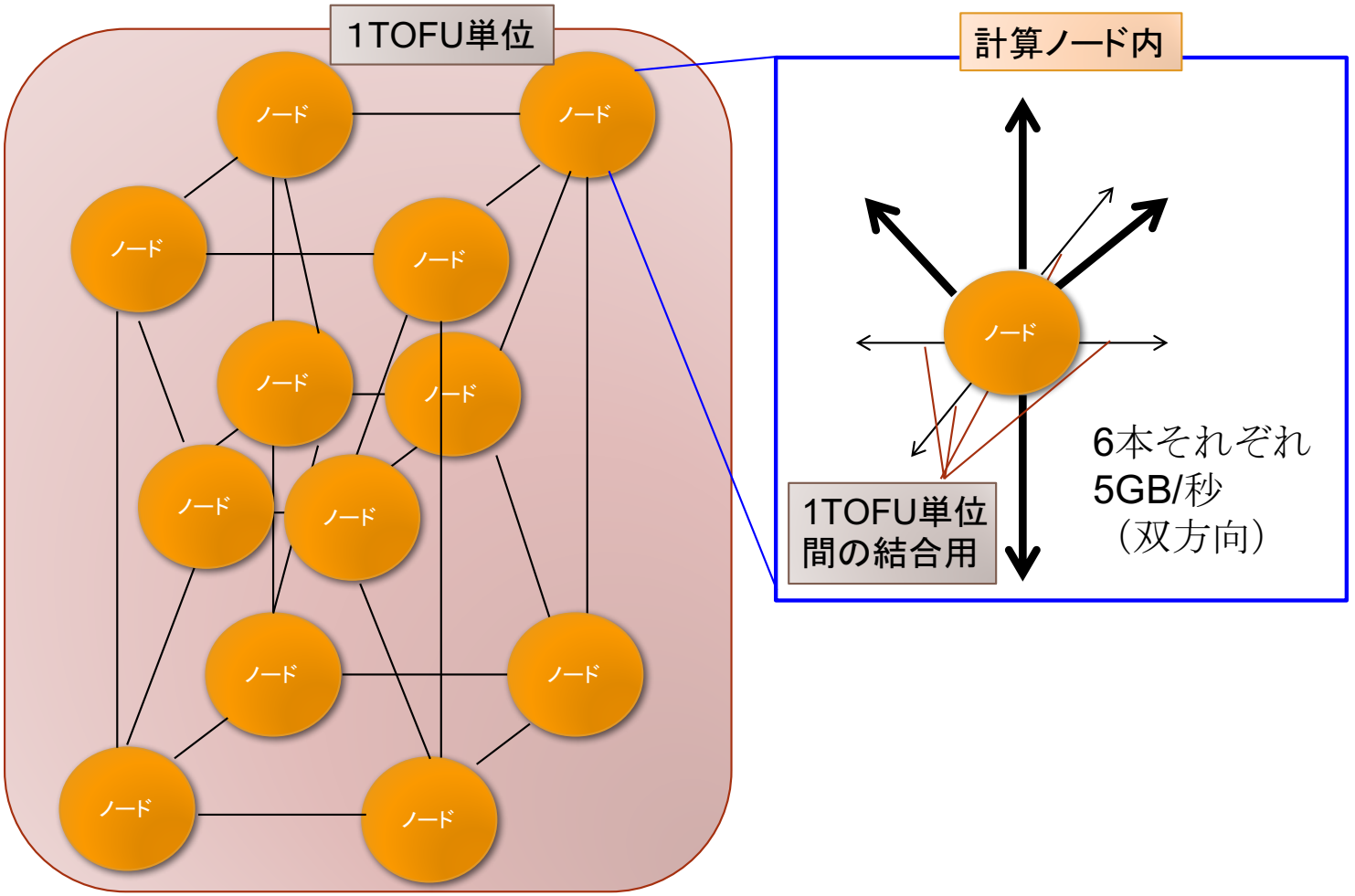
Intel Xeon Phi 7250 1.4 GHz 68 core x1 socket

2016年12月1日試験運転開始

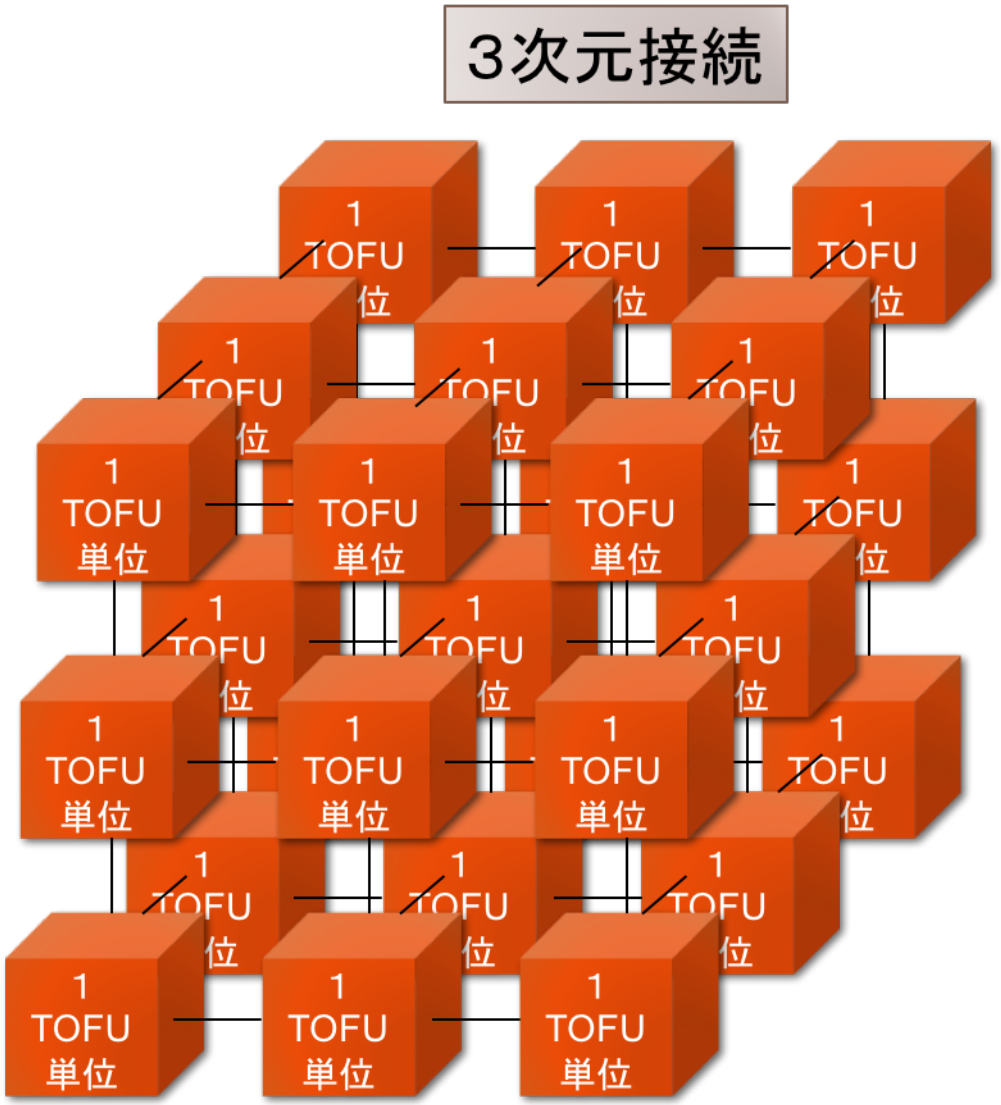
2017年4月3日正式運用開始


FX10計算ノードの構成
1ソケットのみ
TOFU Network
各CPUの内部構成
20GB/秒
ICC
Core #0
Core #1
Core #2
Core #3
...
Core #12
Core #13
Core #14
Core #15
L1
L1
L1
L1
: L1データキャッシュ32KB
L1
L1
L1
L1
L2（16コアで共有、12MB)
85GB/秒
=(8Byte×1333MHz
×8 channel)
Memory
Memory
Memory
Memory
DDR3 DIMM
4GB ×2枚
4GB ×2枚
4GB ×2枚
4GB ×2枚
ノード内合計メモリ量：8GB×4＝32GB

FX10の通信網


1TOFU単位
ノード
ノード
ノード
ノード
ノード
ノード
ノード
ノード
ノード
ノード
ノード
計算ノード内
ノード
1TOFU単位
間の結合用
6本それぞれ
5GB/秒
（双方向）

FX10の通信網（1TOFU単位間の結合）

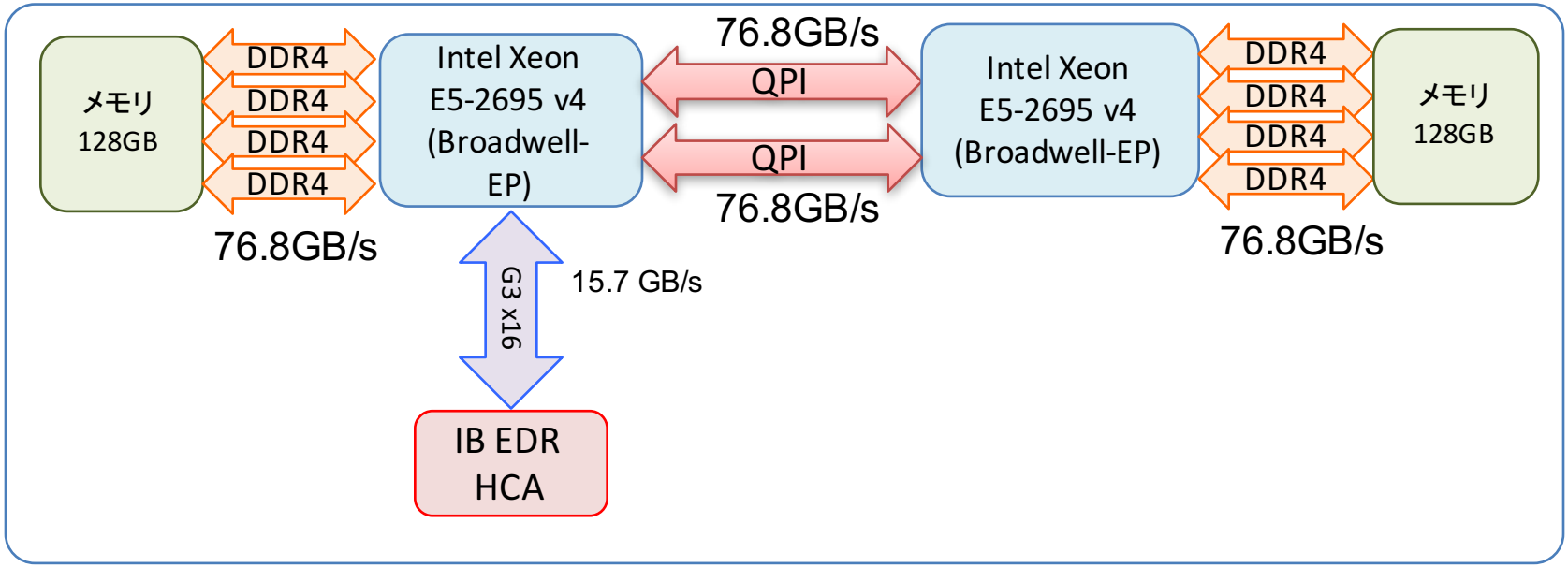


- ユーザから見ると、X軸、Y軸、Z軸について、奥の1TOFUと、手前の1TOFUは、繋がってみえます（3次元トーラス接続）
- ただし物理結線では
  - X軸はトーラス
  - Y軸はメッシュ
  - Z軸はメッシュまたは、トーラス

  になっています

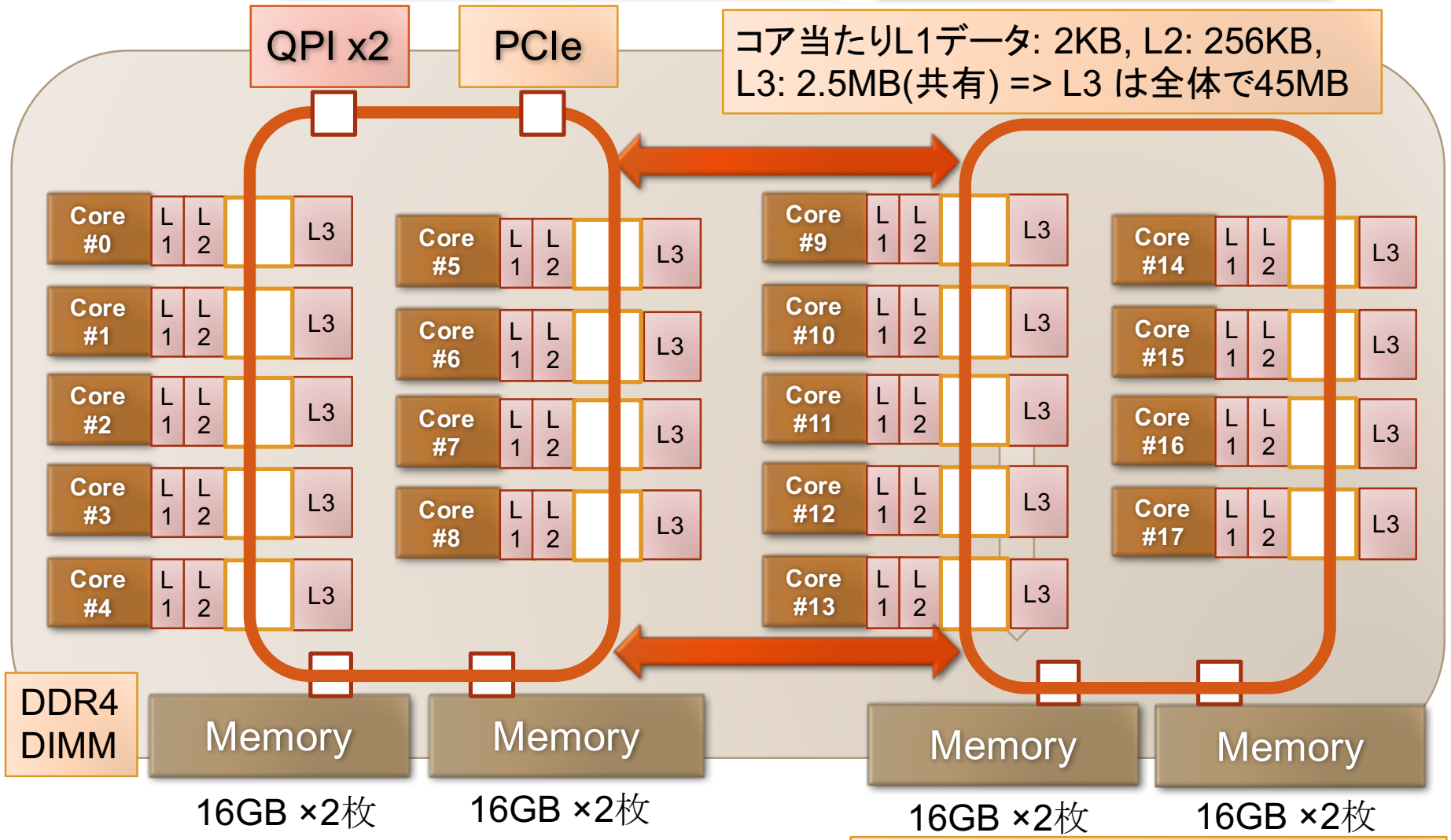
Reedbush-Uノードのブロック図

- メモリのうち、「近い」メモリと「遠い」メモリがある
  => NUMA (Non-Uniform Memory Access)
  (FX10はフラット)



Broadwell-EPの構成

1ソケットのみを図示

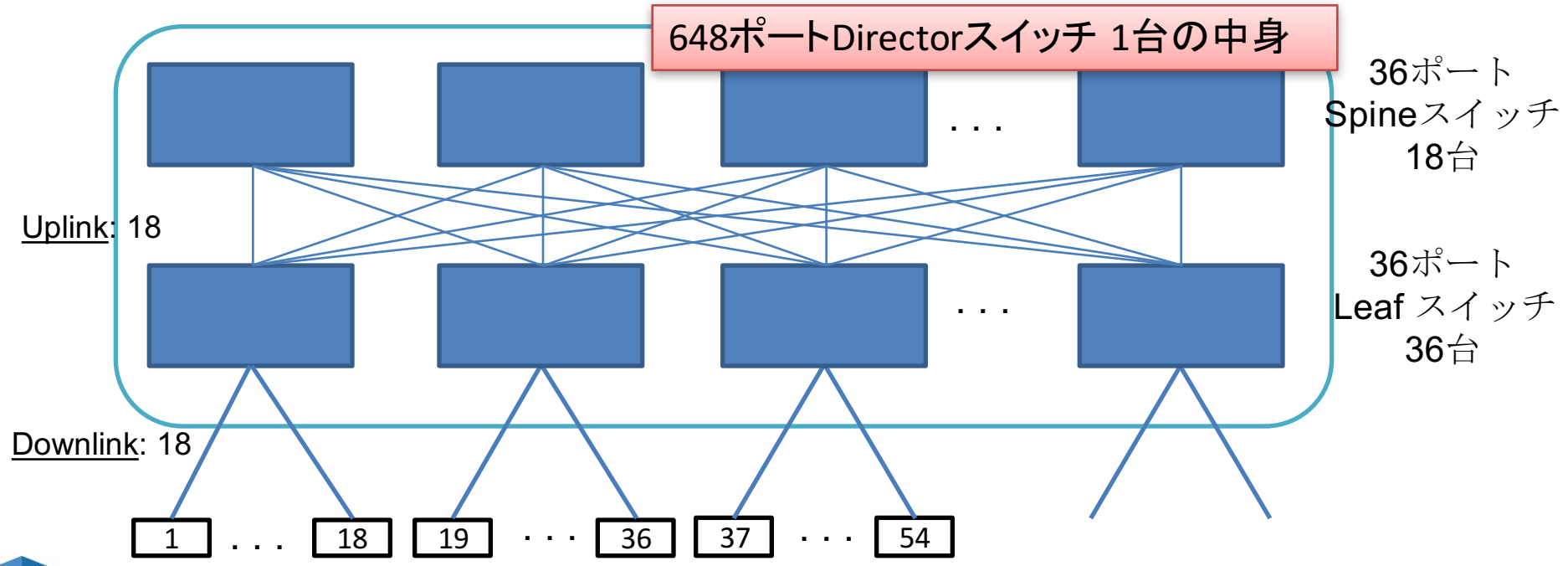


ソケット当たりメモリ量：16GB×8＝128GB

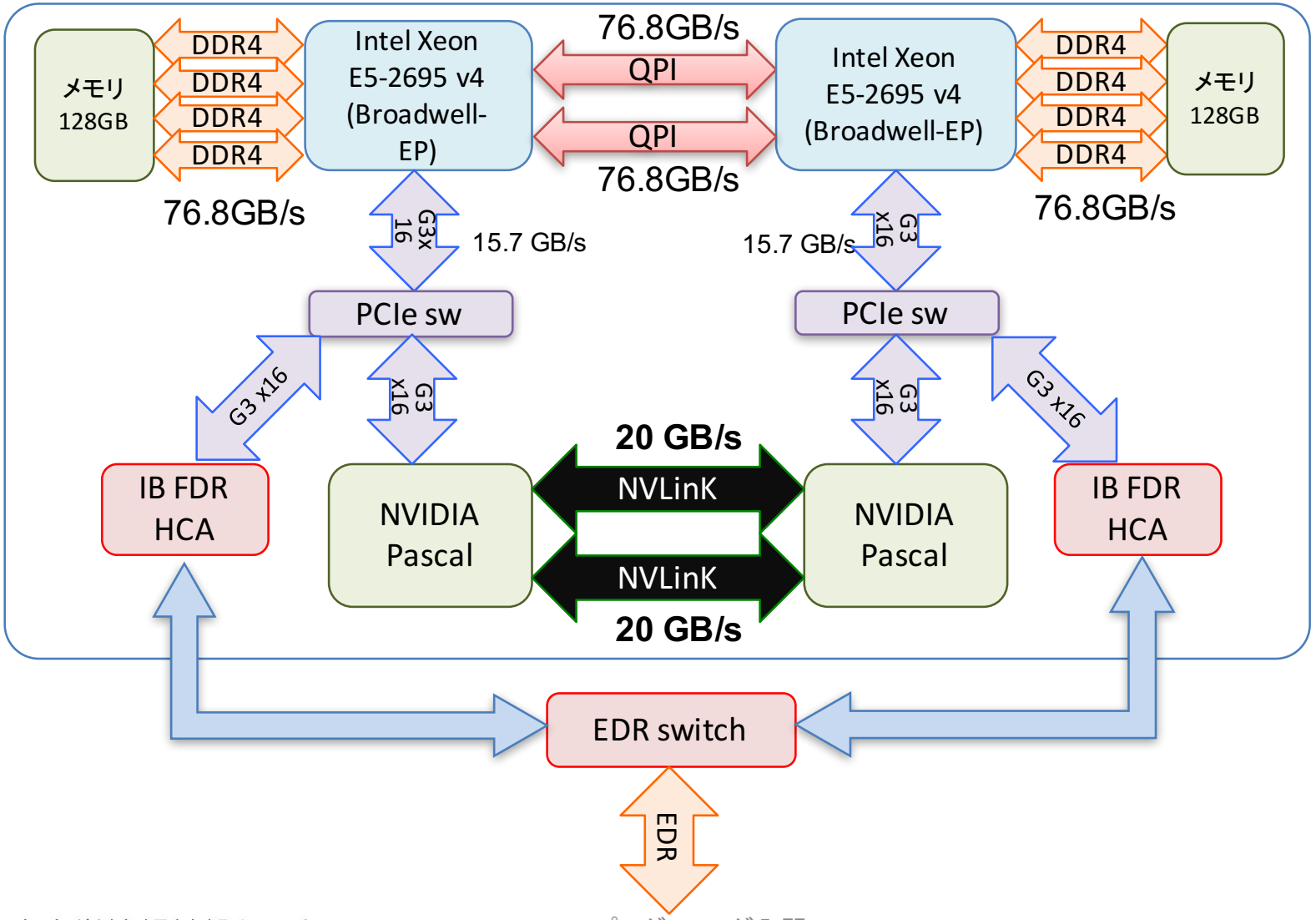
76.8 GB/秒
=(8Byte×2400MHz×4 channel)

Reedbush-Uの通信網

- フルバイセクションバンド幅を持つFat Tree網
  - どのように計算ノードを選んでも互いに無衝突で通信が可能
- Mellanox InfiniBand EDR 4x CS7500: 648ポート
  - 内部は36ポートスイッチ (SB7800)を (36+18)台組み合わせたものと等価



Reedbush-Hノードのブロック図



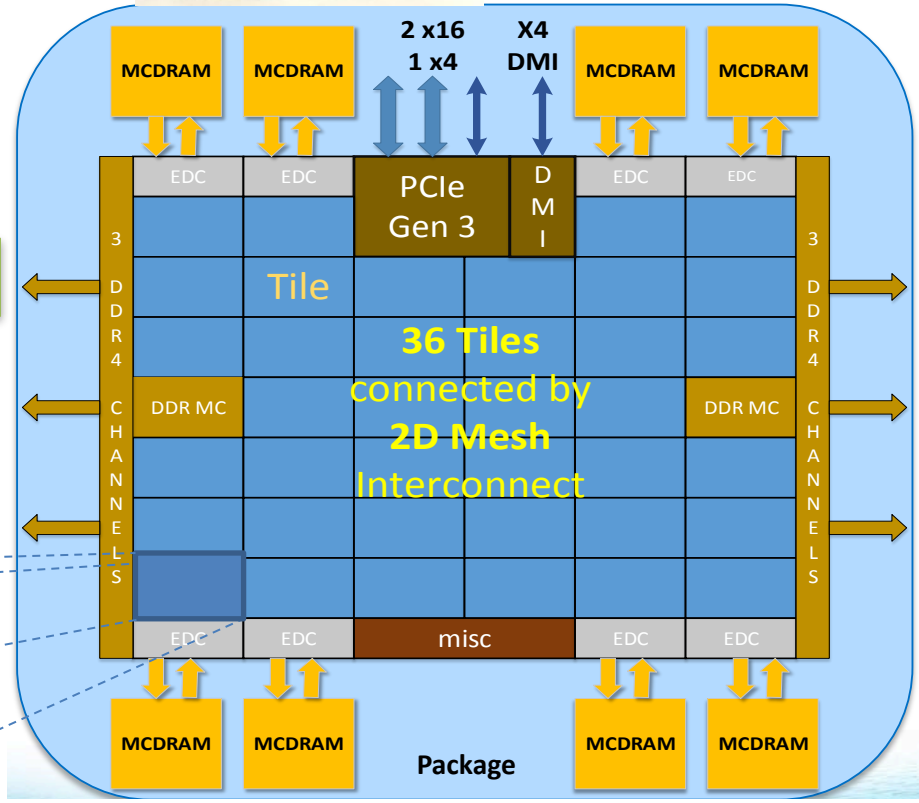
Oakforest-PACS 計算ノード

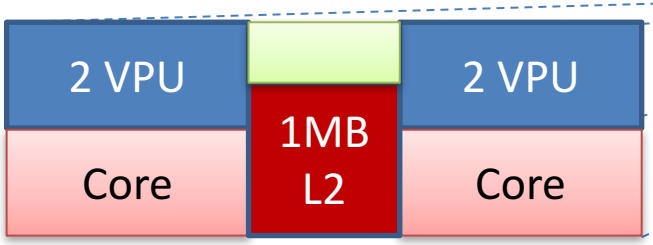
- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット
- MCDRAM: オンパッケージの高バンド幅メモリ16GB + DDR4メモリ

ソケット当たりメモリ量：16GB×6＝96GB

MCDRAM: 490GB/秒以上（実測）
DDR4: 115.2 GB/秒
=(8Byte×2400MHz×6 channel)

HotChips27
KNLスライドより





Oakforest-PACS: Intel Omni-Path Architecture によるフルバイセクションバンド幅Fat-tree網

コストはかかるがフルバイセクションバンド幅を維持
- システム全系使用時にも高い並列性能を実現
- 柔軟な運用：ジョブに対する計算ノード割り当ての自由度が高い

東大情報基盤センターOakforest-PACSスーパーコンピュータシステムの料金表（2017年4月1日 ）

- パーソナルコース（年間）
  - コース1： 100,000円 ： 8ノード(基準)、最大16ノードまで
  - コース2： 200,000円 ： 16ノード(基準)、最大64ノードまで
- グループコース
  - 400,000円 (企業 480,000円) ：1口 8ノード（基準）、最大128ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはFX10、Reedbushとの相互トークン移行も可能

東大情報基盤センターReedbushスーパーコンピュータシステムの料金表（2017年4月1日 ）

- パーソナルコース（年間）
  - 150,000円　：　RB-U: 4ノード（基準）、最大16ノードまで
    RB-H: 1ノード（基準）、最大2ノードまで
- グループコース
  - 300,000円：1口 4ノード（基準）、最大128ノードまで、
    RB-H：1ノード（基準）、最大32ノードまで（トークン係数はUの2.5倍）
  - RB-Uのみ 企業 360,000円　：1口 4ノード（基準）、最大128ノードまで
  - RB-Hのみ 企業 216,000円　：1口 1ノード（基準）、最大32ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはFX10, Oakforest-PACSとの相互トークン移行も可能
  - ノード固定もあり

東大情報基盤センターFX10スーパーコンピュータシステムの料金表（2017年4月1日）

- パーソナルコース（年間）
  - コース1： 90,000円 ： 12ノード(基準)、最大24ノードまで
  - コース2： 180,000円 ： 24ノード(基準)、最大96ノードまで
- グループコース
  - 360,000円 (企業 432,000円) ：1口、12ノード、最大1440ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはReedbush, Oakforest-PACSとの相互トークン移行も可能

トライアルユース制度について

- 安価に当センターのOakleaf/Oakbridge-FX, Reedbush-U/H, Oakforest-PACSシステムが使える「無償トライアルユース」および「有償トライアルユース」制度があります。
  - アカデミック利用
    - パーソナルコース、グループコースの双方（1ヶ月～3ヶ月）
  - 企業利用
    - パーソナルコース（1ヶ月～3ヶ月）（FX10: 最大24ノード、最大96ノード, RB-U: 最大16ノード, RB-H: 最大2ノード、OFP: 最大16ノード, 最大64ノード）
      講習会いずれかの受講が必須、審査無
    - グループコース
      - 無償トライアルユース：（1ヶ月～3ヶ月）：無料（FX10:最大1,440ノード、RB-U: 最大128ノード、RB-H: 最大32ノード、OFP: 最大2048ノード）
      - 有償トライアルユース：（1ヶ月～最大通算9ヶ月）、有償（計算資源は無償と同等）
      - スーパーコンピュータ利用資格者審査委員会の審査が必要（年2回実施）
    - 双方のコースともに、簡易な利用報告書の提出が必要

スーパーコンピュータシステムの詳細

- 以下のページをご参照ください
  - 利用申請方法
  - 運営体系
  - 料金体系
  - 利用の手引

  などがご覧になれます。

  http://www.cc.u-tokyo.ac.jp/system/ofp/
  http://www.cc.u-tokyo.ac.jp/system/reedbush/
  http://www.cc.u-tokyo.ac.jp/system/fx10/

GPU入門

「GPUプログラミング入門」講習会の目的

GPUのアーキテクチャ

GPUの使い方

GPUって何？

- GPU : Graphics Processing Unit
  - いわゆるグラフィックボード(グラボ)、ビデオカード

- 画像処理に特化したハードウェア
  - 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
  - ノートPC、ゲームハード、スマホなどに搭載

- GPUを汎用計算に用いる手法をGPGPU、GPUコンピューティングなどと呼ぶ

なぜGPUコンピューティング？

- 性能が高いから！

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

GPUの応用例

http://www.nvidia.com/object/gpu-applications.html (英語 情報量多い)
http://www.nvidia.co.jp/object/gpu-applications-domain-jp.html (日本語)

- 数値流体力学
- 数値分析
- 医療画像
- 天気と気候
- データサイエンス
- 機械学習
  - AlphaGo の学習にもGPU

線形代数演算が速いので、広い範囲で応用可能！

GPUプログラミングは何が難しい？

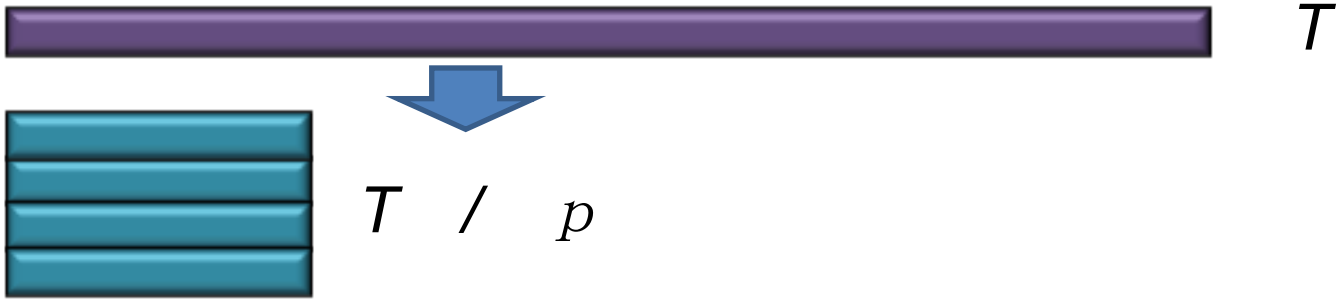
- CPU：大きなコアをいくつか搭載
  - Reedbush-H の CPU：2.10 GHz 18コア
  - 大きなコア… 分岐予測、パイプライン処理、Out-of-Order
    - 要はなんでもできる
  - 逐次の処理が得意
- GPU：小さなコアをたくさん搭載
  - Reedbush-H の GPU：1.48 GHz 3,584 コア
  - 小さなコア… 上記機能が**弱いまたはない！**
  - **並列処理が必須**

GPUの難しさ

1. 多数のコアを効率良く扱う難しさ
2. 並列プログラミング自体の難しさ

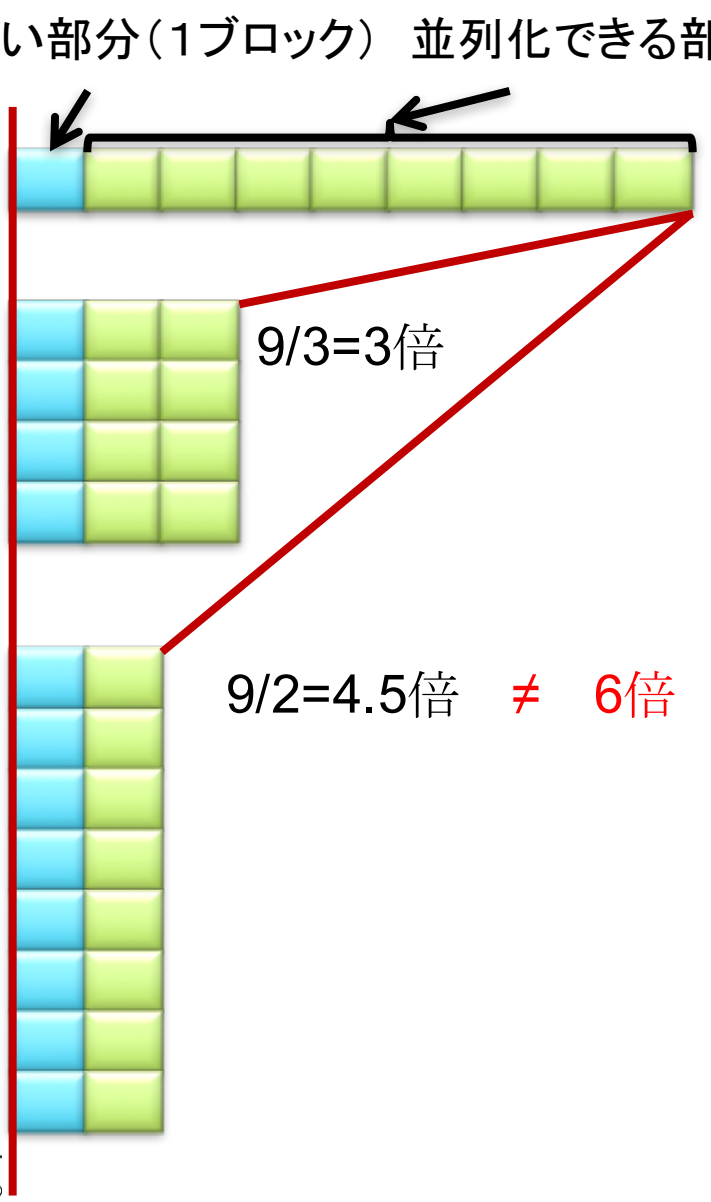
並列プログラミングとは何か？

- 逐次実行のプログラム（実行時間T）を、p台の計算機を使って、T / pにすること。



- 素人考えでは自明。
- 実際は、できるかどうかは、対象処理の内容（アルゴリズム）で **大きく** 難しさが違う
  - アルゴリズム上、絶対に並列化できない部分の存在
  - 通信のなどのオーバヘッドの存在
    - 通信立ち上がり時間
    - データ転送時間

アムダールの法則



- 逐次実行

＝88.8%が並列化可能

- 並列実行（４並列）

9/3=3倍

- 並列実行（８並列）

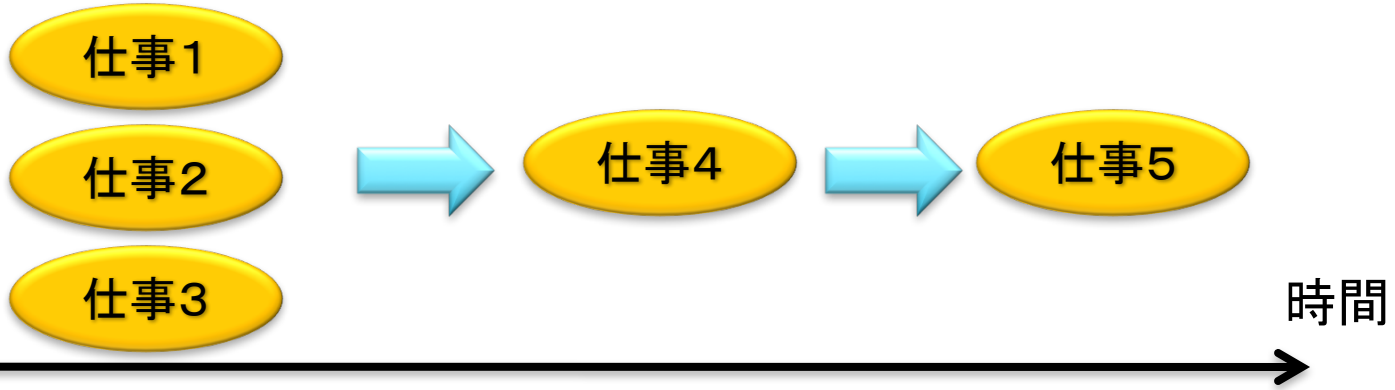
9/2=4.5倍　≠　6倍

並列処理の実行形態：タスク並列

- タスク（ジョブ）を分割することで並列化する。
- データの操作（＝演算）は異なるかもしれない。
- タスク並列の例：カレーを作る
  - 仕事１：野菜を切る
  - 仕事２：肉を切る
  - 仕事３：水を沸騰させる
  - 仕事４：野菜・肉を入れて煮込む
  - 仕事５：カレールゥを入れる

GPUは苦手

●並列化



並列処理の実行形態：データ並列

- データを分割することで並列化する。
- データの操作（＝演算）は同一となる。
- データ並列の例：行列ー行列積

GPUでは
基本これ！

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1*9+2*6+3*3 & 1*8+2*5+3*2 & 1*7+2*4+3*1 \\ 4*9+5*6+6*3 & 4*8+5*5+6*2 & 4*7+5*4+6*1 \\ 7*9+8*6+9*3 & 7*8+8*5+9*2 & 7*7+8*4+9*1 \end{pmatrix}$$

●並列化

全CPUで共有

CPU0
CPU1
CPU2

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1*9+2*6+3*3 & 1*8+2*5+3*2 & 1*7+2*4+3*1 \\ 4*9+5*6+6*3 & 4*8+5*5+6*2 & 4*7+5*4+6*1 \\ 7*9+8*6+9*3 & 7*8+8*5+9*2 & 7*7+8*4+9*1 \end{pmatrix}$$

並列に計算：初期データは異なるが演算は同一

本講習会の目的

並列アルゴリズムはひとまず置いといて…

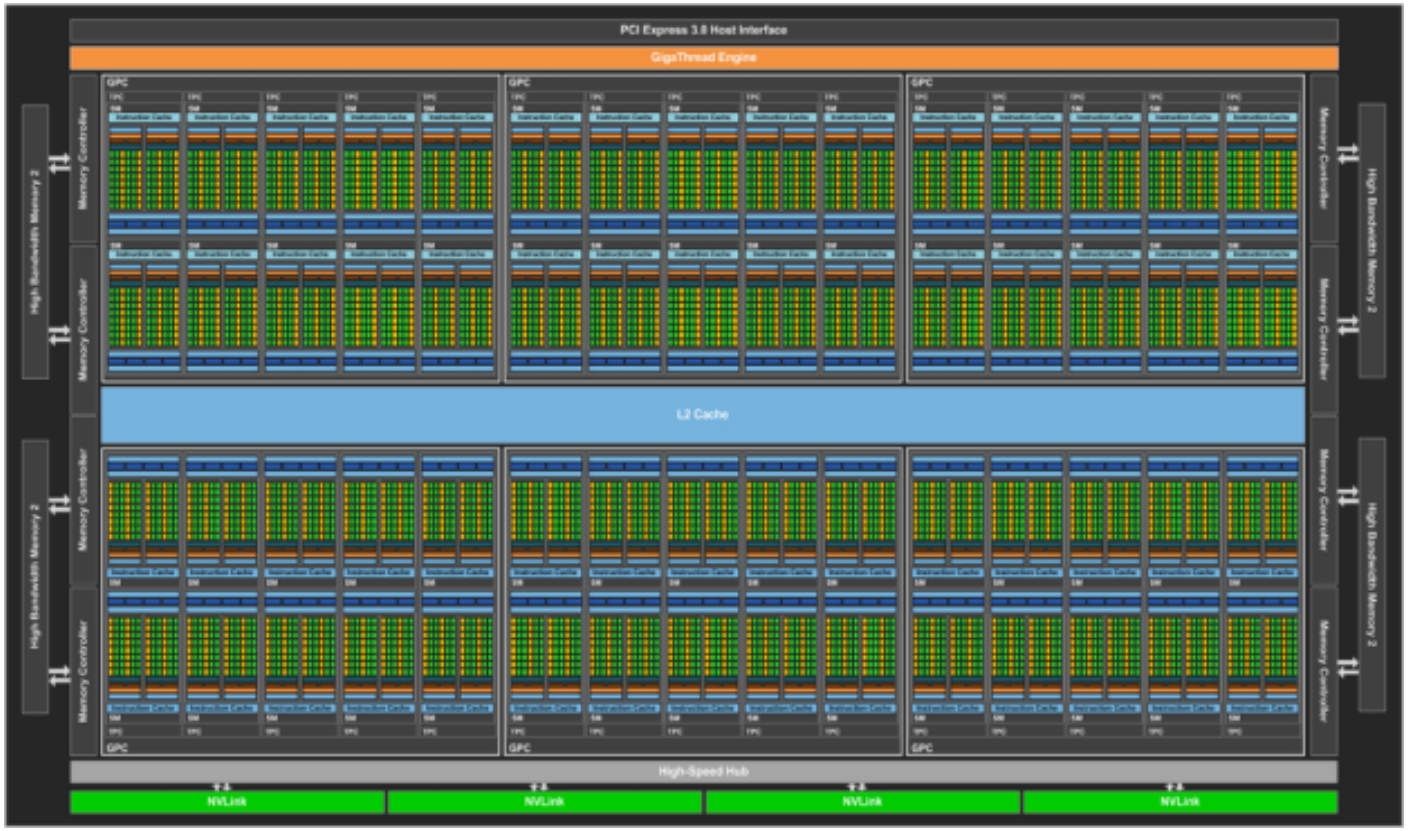
GPUの多数のコアを効率良く扱う手法を学ぼう！

- その他の講習会 (予定) http://www.cc.u-tokyo.ac.jp/support/kosyu
  - 10/31,11/1
    - OpenMP/OpenACCによるマルチコア・メニィコア並列プログラミング入門
  - 11/7, 8
    - 有限要素法で学ぶ並列プログラミングの基礎
  - 11/21
    - １日速習：三次元並列有限要素法とハイブリッド並列プログラミング
  - 11/24
    - KNL実践
  - 1月あたり
    - ディープラーニング講習会（仮）

「こんな講習会が欲しい！」という要望があればぜひアンケートへ！

参考 : NVIDIA Tesla P100

- 56 SMs
- 3584 CUDA Cores
- 16 GB HBM2



P100 whitepaper より

参考：NVIDIA Tesla P100 の SM

GPU TECHNOLOGY CONFERENCE

GP100 SM

	GP100
CUDA Cores	64
Register File	256 KB
Shared Memory	64 KB
Active Threads	2048
Active Blocks	32

P100 whitepaper より

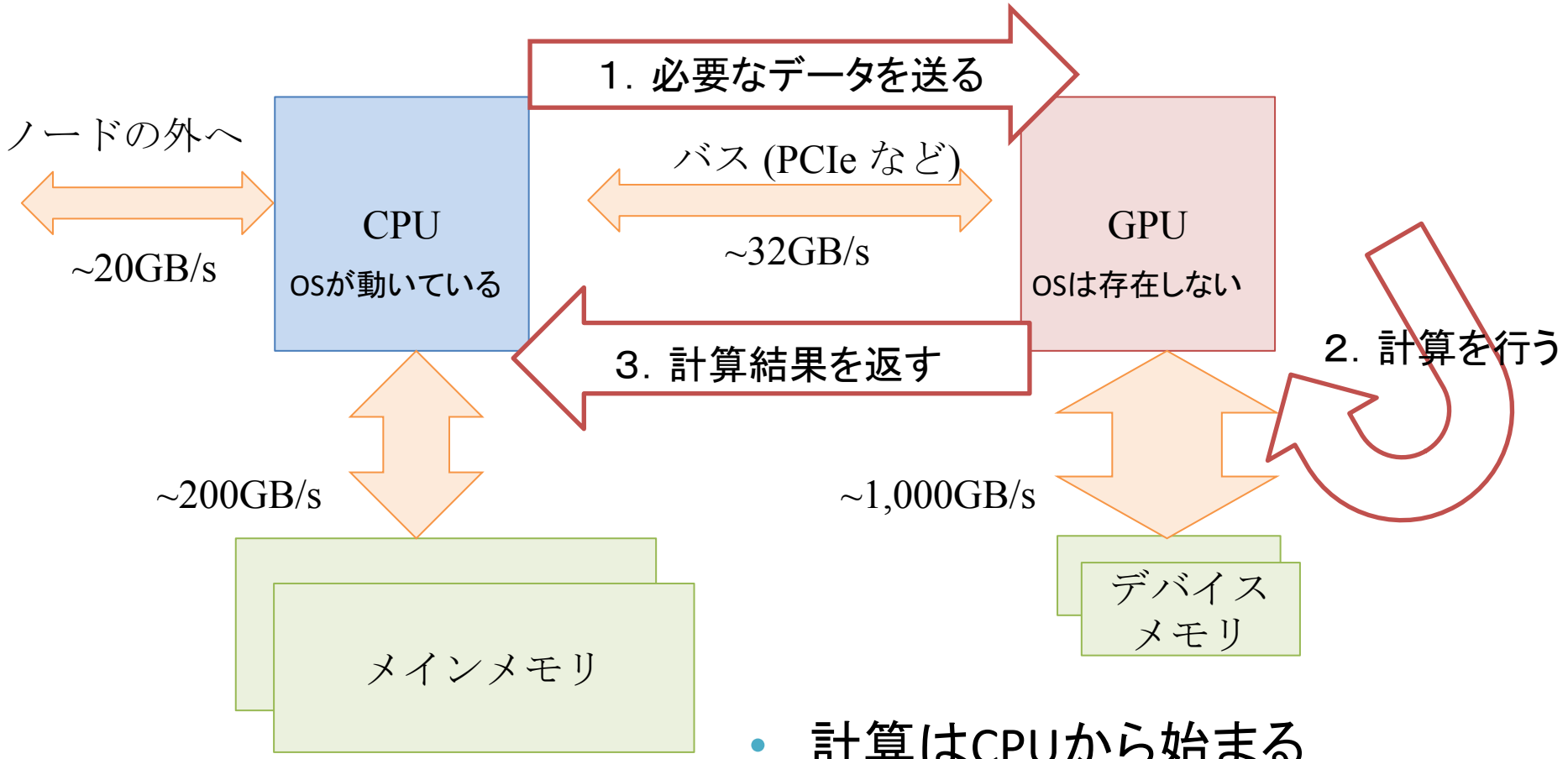
参考：NVIDIA社のGPU

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

押さえておくべきGPUの特徴

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

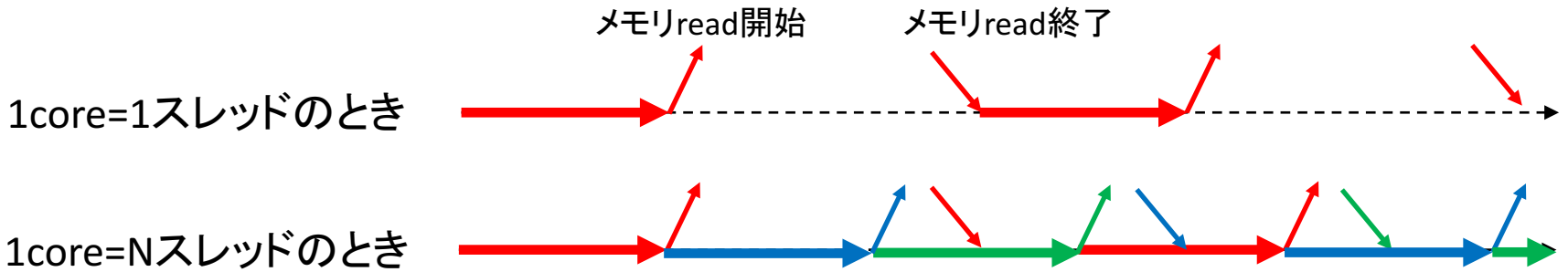
CPUと独立のGPUメモリ



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

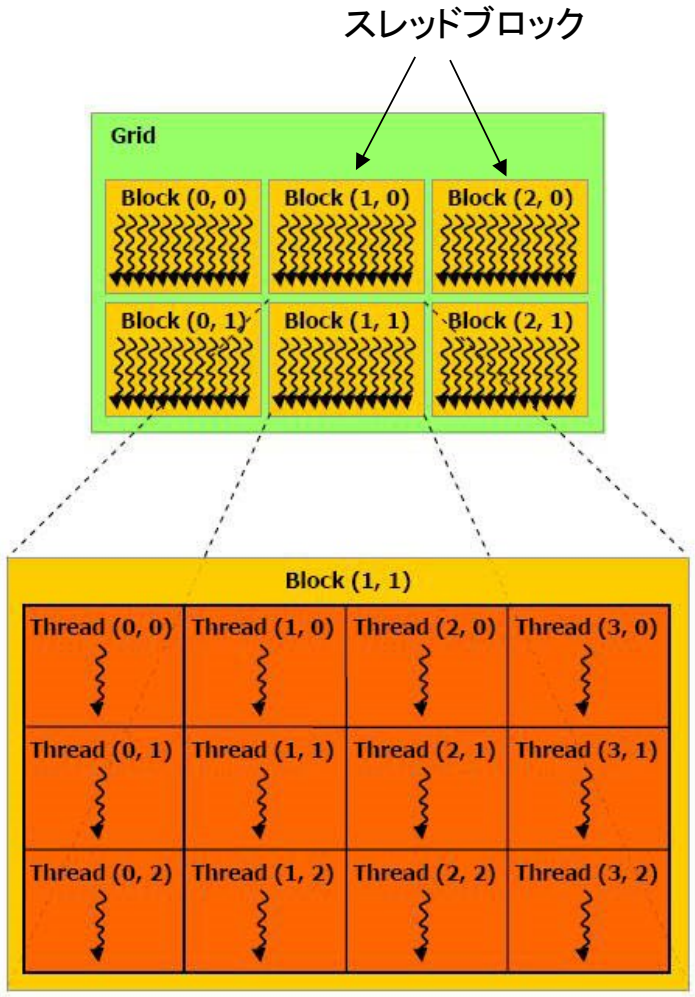
性能を出すためにはスレッド数>>コア数

- 推奨スレッド数
  - CPU: スレッド数=コア数 (高々数十スレッド)
  - GPU: スレッド数>=コア数*4~ (数万~数百万スレッド)
    - 最適値は他のリソースとの兼ね合いによる
- 理由:高速コンテキストスイッチによるメモリレイテンシ隠し
  - CPU:レジスタ・スタックの退避はOSがソフトウェアで行う(遅い)
  - GPU:ハードウェアサポートでコストほぼゼロ
    - メモリアクセスによる暇な時間(ストール)に他のスレッドを実行



階層的スレッド管理と同期

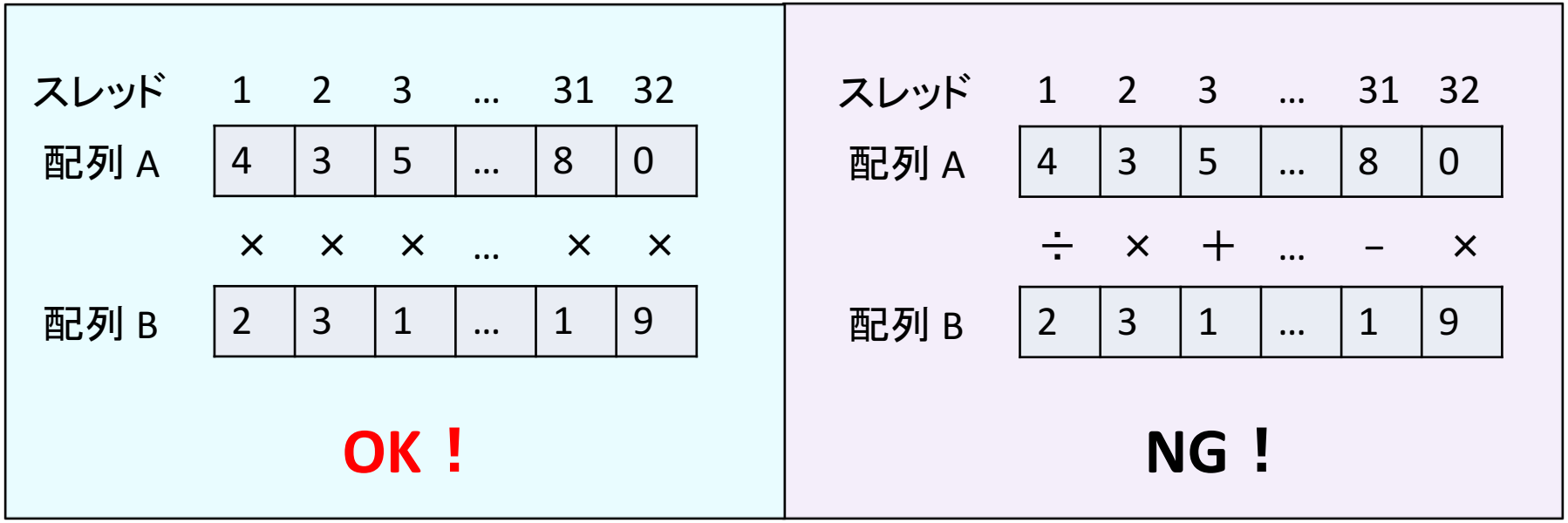
- 階層的なコア/スレッド管理
  - P100は56 SMを持ち、1 SMは64 CUDA coreを持つ。トータル3584 CUDA core
  - 1 SMが複数のスレッドブロックを担当し、1 CUDA core が複数スレッドを担当
- スレッド間の同期
  - 同一スレッドブロック内のスレッドは同期できる
  - 異なるスレッドブロックに属するスレッド間は同期できない
    - 同期するためにはGPUの処理を終了する必要あり
    - atomic 演算は可能



cited from : http://cuda-programming.blogspot.jp/2012/12/thread-hierarchy-in-cuda-programming.html

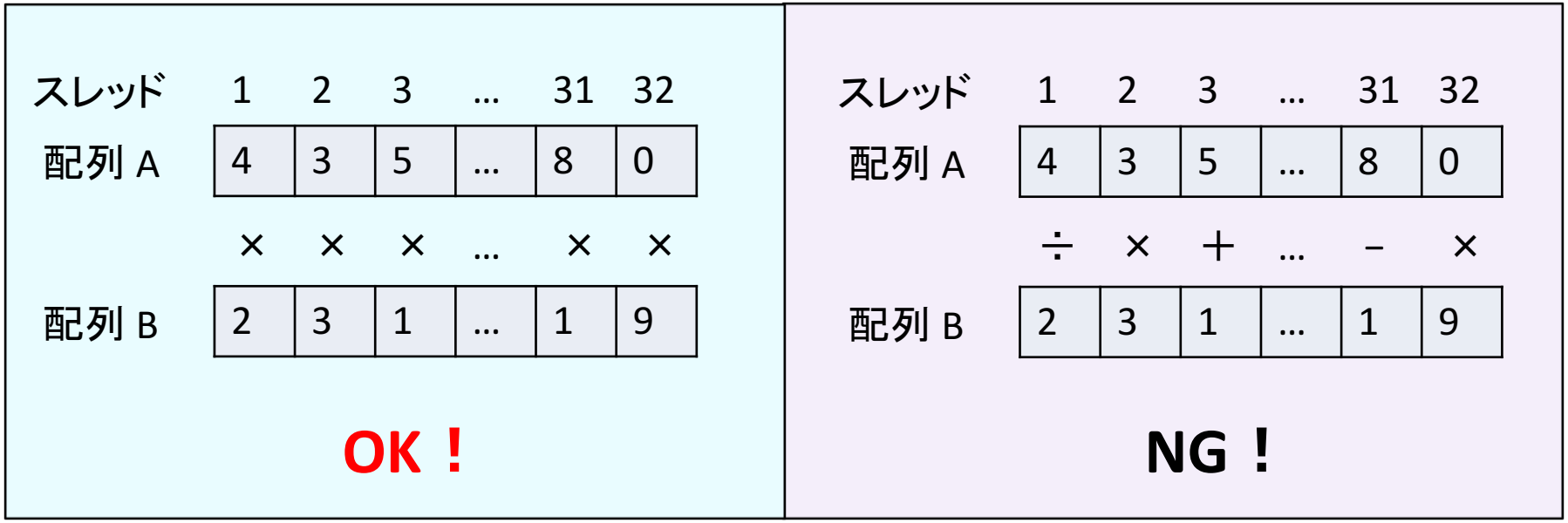
Warp 単位の実行

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



Warp 単位の実行

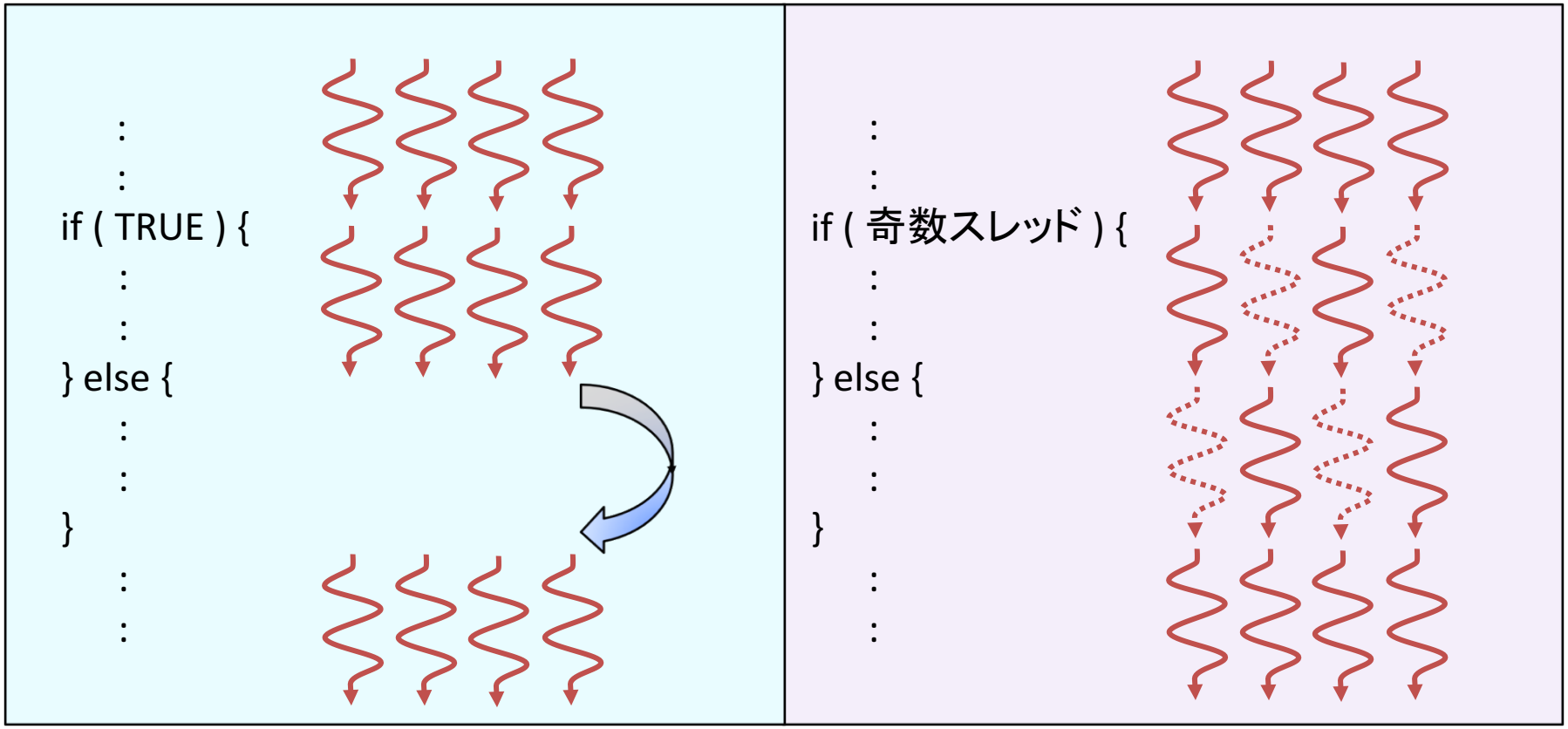
- 連続した32スレッドを1単位 = Warp と呼ぶ
- このWarpは足並み揃えて動く
  - 実行する命令は32スレッド全て同じ
  - データは違ってもいい



やってはいけないWarp内分岐

この仕様はCUDA9で少し変わります！（今はCUDA8）

- Divergent Branch
  - Warp 内で分岐すること。Warp単位の分岐ならOK。

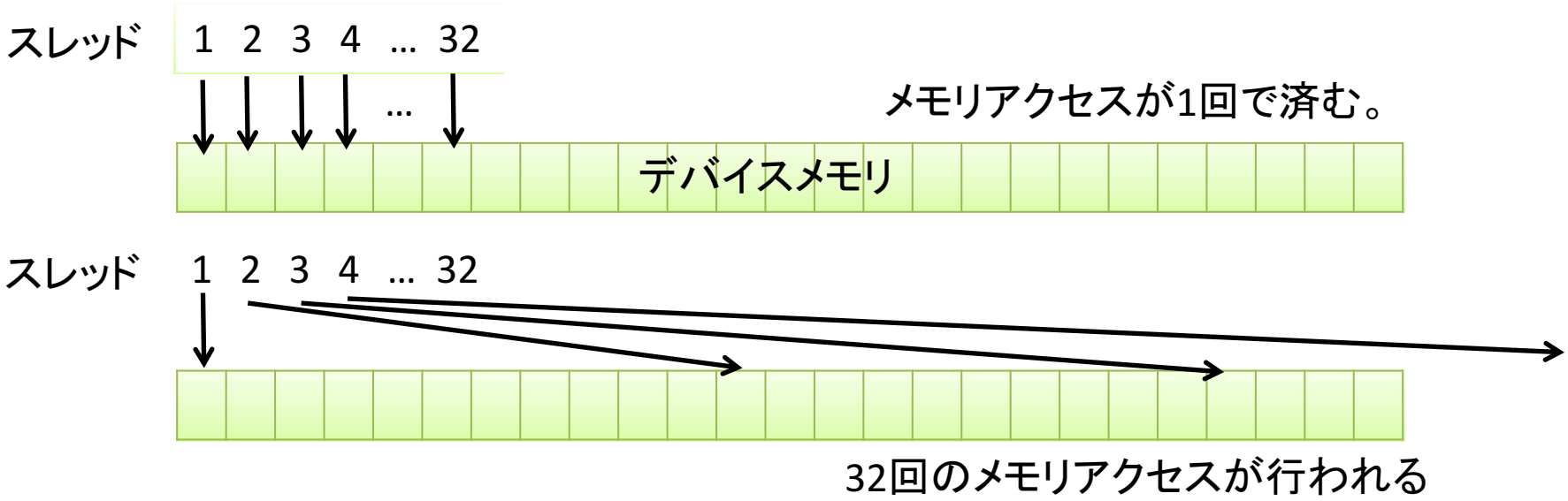


else 部分は実行せずジャンプ

一部スレッドを眠らせて全分岐を実行
最悪ケースでは32倍のコスト

コアレスドアクセス

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



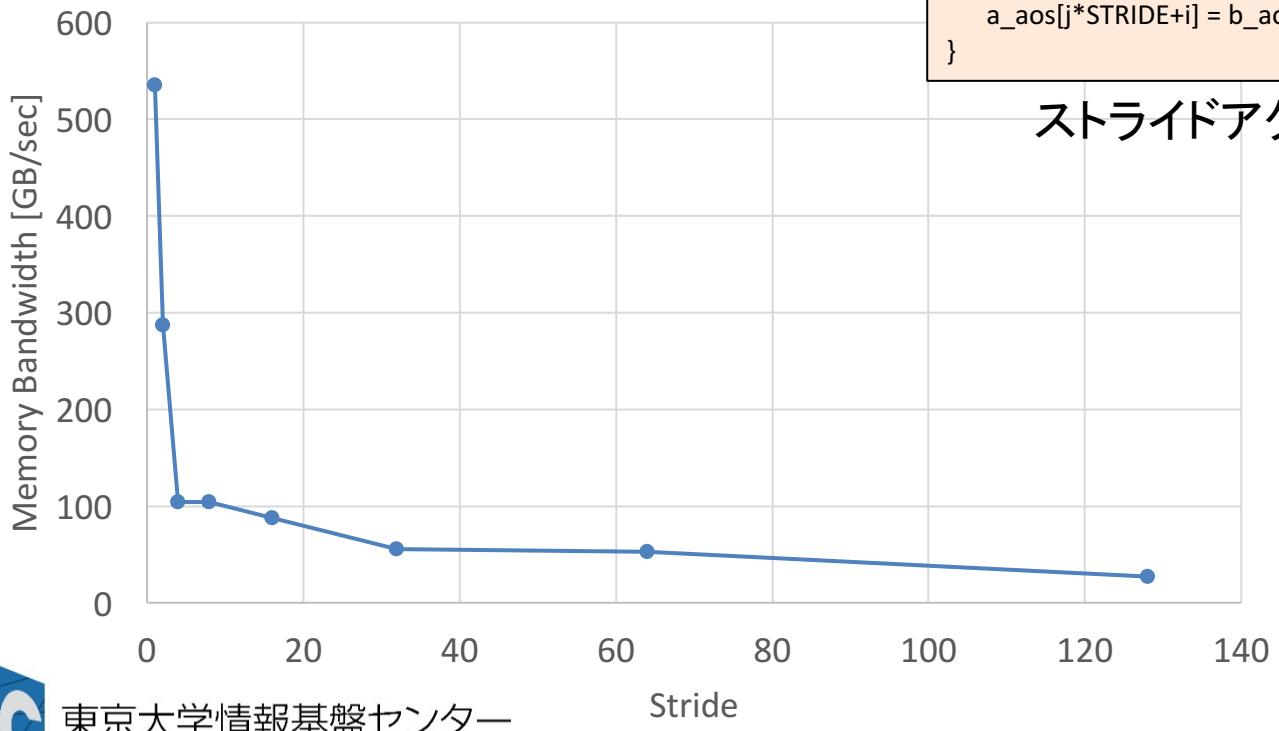
128バイト単位でメモリアクセス。Warp内のアクセスが128バイトに収まってれば1回。外れればその分だけ繰り返す。最悪ケースでは32倍のコスト

ストライドアクセスがあるとどうなるか

- GPUはストライドアクセスに弱い！

```
void AoS_STREAM_Triad(STREAM_TYPE scalar)
{
  ssize_t i,j;
#pragma omp parallel for private(i,j)
#pragma acc kernels present(a_aos[0:STREAM_ARRAY_SIZE] ¥
      ,b_aos[0:STREAM_ARRAY_SIZE],c_aos[0:STREAM_ARRAY_SIZE])
#pragma acc loop gang vector independent
  for (j=0; j<STREAM_ARRAY_SIZE/STRIDE; j++)
    for (i=0; i<STRIDE; i++)
      a_aos[j*STRIDE+i] = b_aos[j*STRIDE+i]+scalar*c_aos[j*STRIDE+i];
}
```

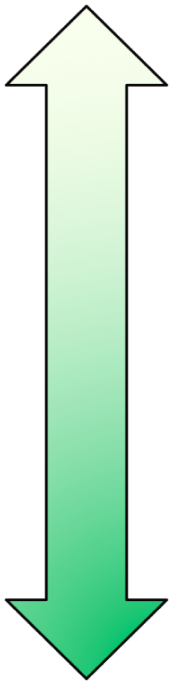
ストライドアクセス付き stream triad



OpenACC入門

- OpenACC とは
- OpenACC の指示文
- OpenACC の実用例
- OpenACC 演習

GPUを利用する方法



- GPU対応ライブラリ
  - ライブラリを呼び出すだけ
  - ライブラリとして存在するもののみ
- OpenACC
  - 既存のC/C++/Fortranプログラムに指示文を入れるだけ
  - 細かいチューニングは不可
- CUDA
  - 自由度が最も高い
  - プログラムの大幅な書き換えを必要とする

本講習会の主な対象

OpenACC

- OpenACCとは… **アクセラレータ(GPUなど)向けのOpenMPのようなもの**
  - 既存のプログラムのホットスポットに指示文を挿入し、計算の重たい部分をアクセラレータにオフロード
  - 対応言語：C/C++, Fortran
- 指示文ベース
  - 指示文：コンパイラへのヒント
  - 記述が簡便, メンテナンスなどをしやすい
  - コードの可搬性(portability)が高い
    - 対応していない環境では無視される

C/C++

```
#pragma acc kernels
for(i = 0;i < N;i++) {
  ….
}
```

Fortran

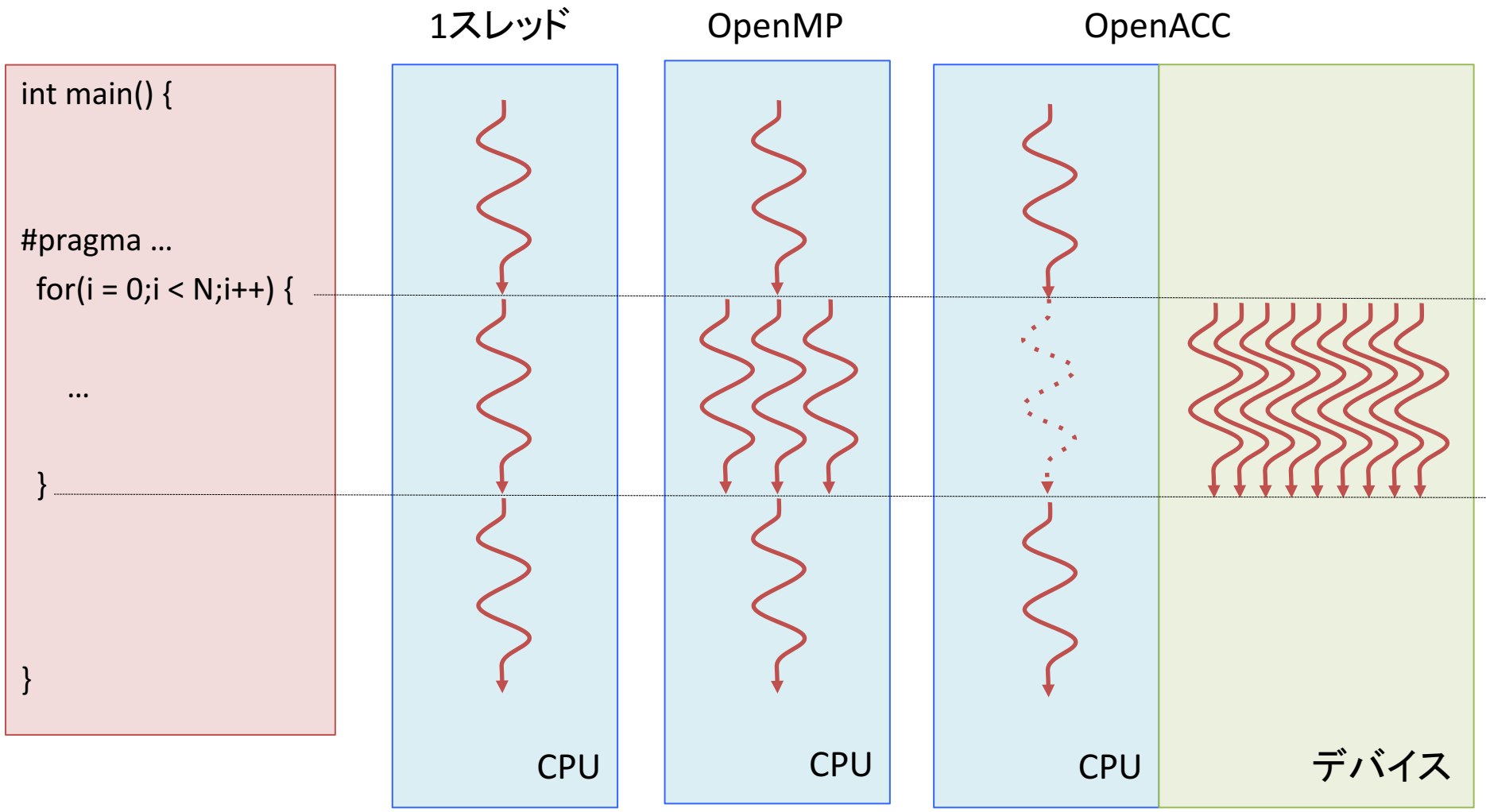
```
!$acc kernels
do i = 1, N
  ….
end do
!$acc end kernels
```

OpenACC

- 規格
  - 各コンパイラベンダ(PGI, Crayなど)が独自に実装していた拡張を統合し共通規格化 (http://www.openacc.org/)
  - 2011年秋にOpenACC 1.0 最新の仕様はOpenACC 2.5
- 対応コンパイラ
  - 商用：PGI, Cray, PathScale
    - PGI は無料版も出している
  - 研究用：Omni (AICS), OpenARC (ORNL), OpenUH (U.Houston)
  - フリー：GCC 6.x
    - 開発中 (開発状況： https://gcc.gnu.org/wiki/Offloading)
    - 実用にはまだ遠い

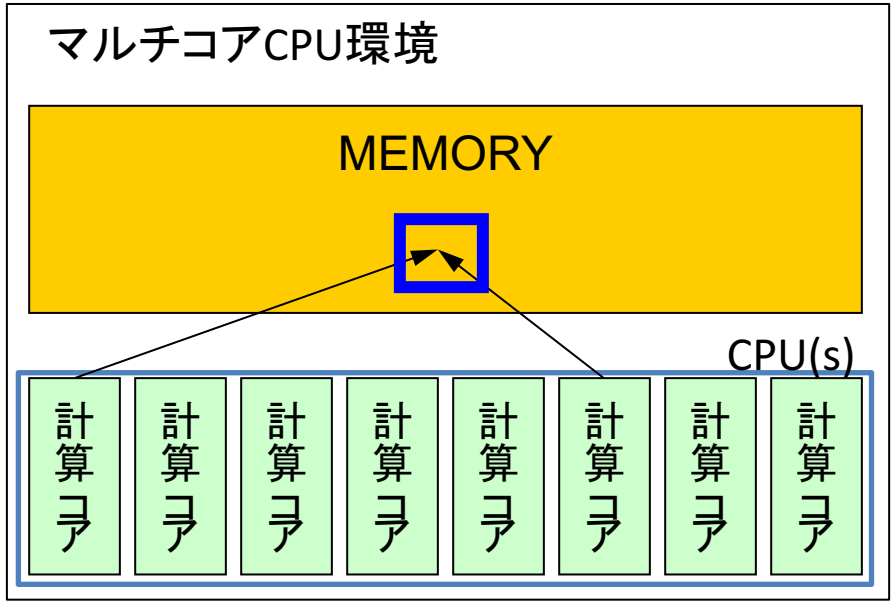
本講習会ではPGIコンパイラを用いる

OpenACC と OpenMP の実行イメージ比較



OpenACC と OpenMP の比較

OpenMPの想定アーキテクチャ

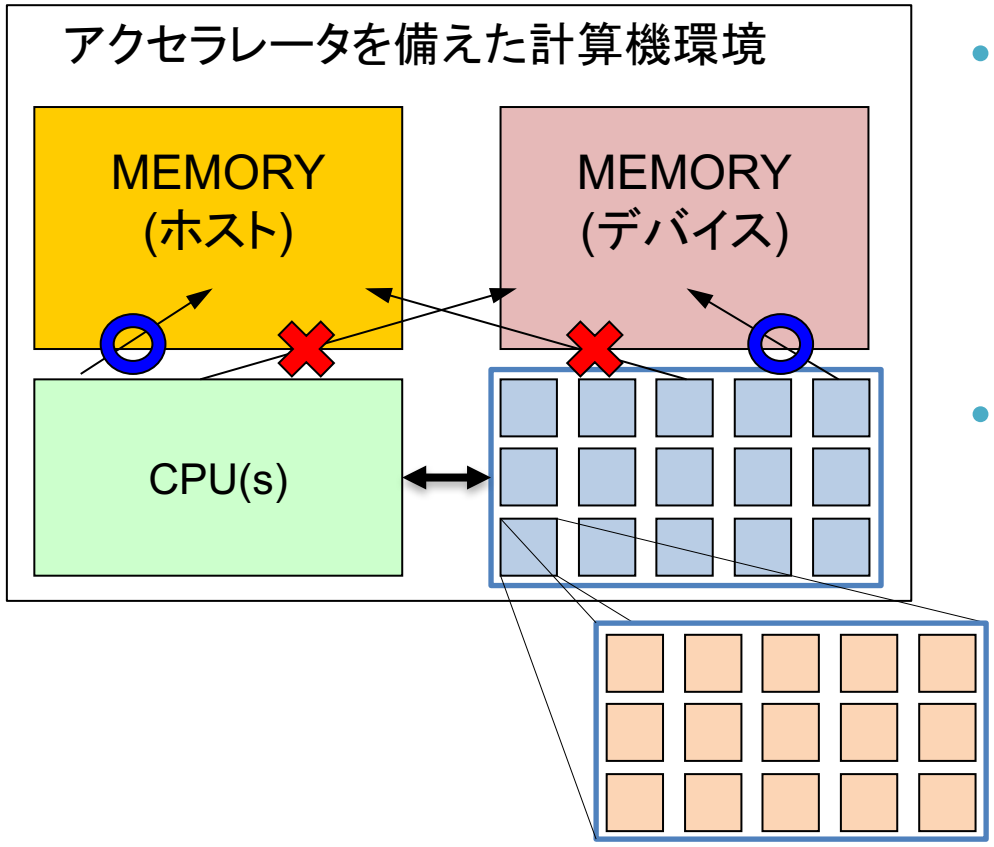


- 計算コアがN個
  - N < 100 程度 (Xeon Phi除く)
- 共有メモリ

一番の違いは対象アーキテクチャの複雑さ

OpenACC と OpenMP の比較

OpenACC の想定アーキテクチャ



- 計算コアN個をM階層で管理
  - N > 1000 を想定
  - 階層数Mはアクセラレータによる
- ホスト-デバイスで独立したメモリ
  - ホスト-デバイス間データ転送は低速

一番の違いは対象アーキテクチャの複雑さ

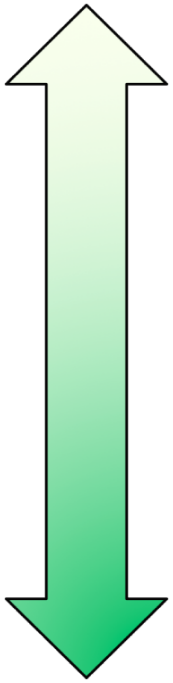
OpenACCとOpenMPの比較

- OpenMPと同じもの
  - Fork-Joinという概念に基づくループ並列化
- OpenMPになくてOpenACCにあるもの
  - ホストとデバイスという概念
    - ホスト-デバイス間のデータ転送
  - 多階層の並列処理
- OpenMPにあってOpenACCにないもの
  - スレッドIDを用いた処理など
    - OpenMPのomp_get_thread_num()に相当するものが無い
- その他、気をつけるべき違い
  - OpenMPと比べてOpenACCは勝手に行うことが多い
    - 転送データ、並列度などを未指定の場合は勝手に決定

OpenACC と OpenMP の比較
デフォルトでの変数の扱い

- OpenMP
  - 全部 shared
- OpenACC
  - スカラ変数：firstprivate or private
  - 配列：shared
    - プログラム上のparallel/kernels構文に差し掛かった時、OpenACCコンパイラは実行に必要なデータを自動で転送する
    - 正しく転送されないこともある。自分で書くべき
    - 構文に差し掛かるたびに転送が行われる(非効率)。後述のdata指示文を用いて自分で書くべき
    - 配列はデバイスに確保される (shared的振る舞い)
    - 配列変数をprivateに扱うためには private 指示節使う

GPUプログラミング難易度早見表（私見）



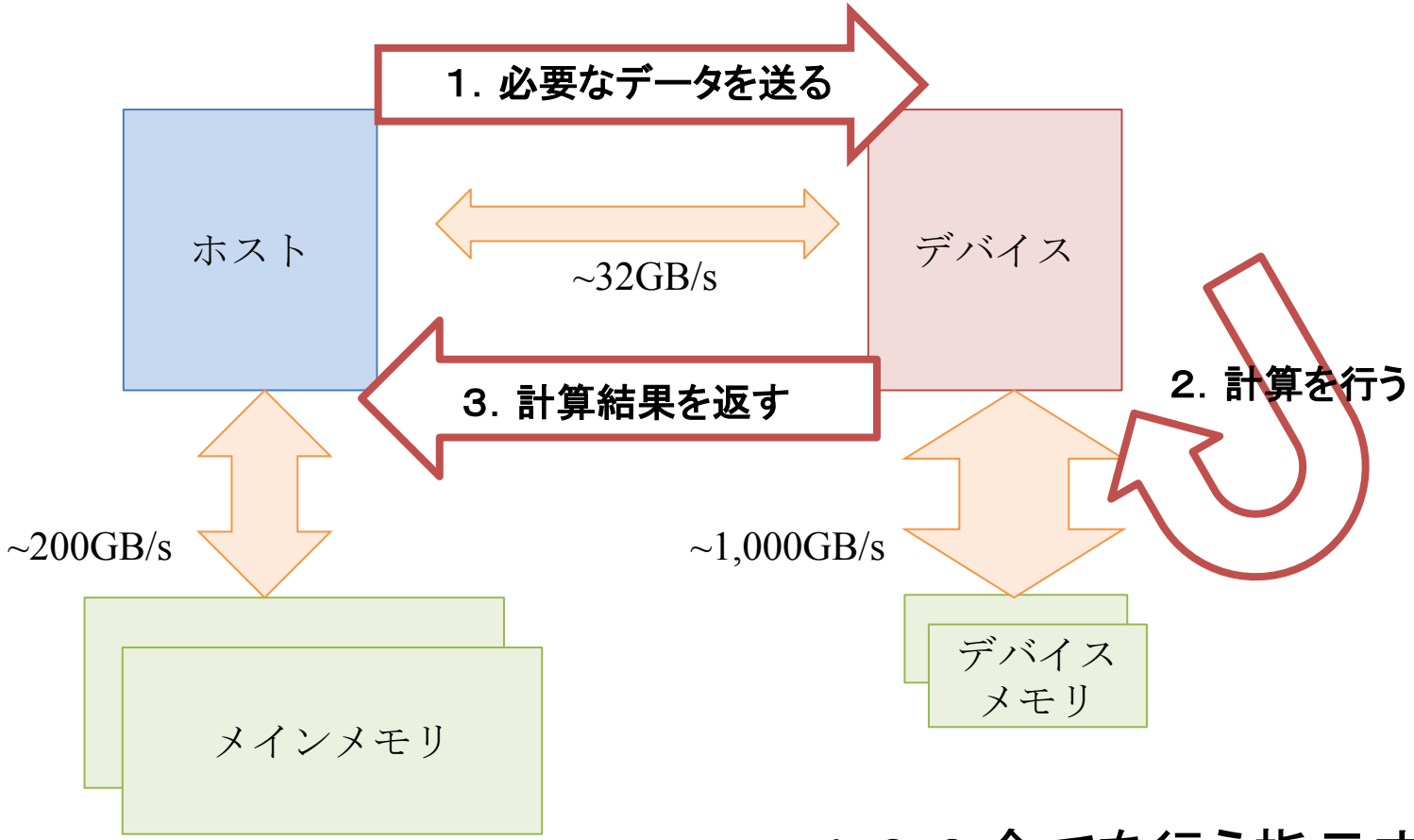
OpenACC	OpenMP
易	
OpenACC with Unified Memory	omp parallel do 書くだけ
データマネージメントの壁	
OpenACC with データ指示文	
スレッド制御の壁	
OpenACC のカーネルチューニング	指示文を用いたSIMD化
カーネルのCUDA化	
難	intrinsic を用いたSIMD化

OpenACC の指示文

OpenACC の主要な指示文

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

並列領域指定指示文



- 1, 2, 3 全てを行う指示文

並列領域指定指示文：parallel, kernels

- アクセラレータ上で実行すべき部分を指定
  - OpenMPのparallel指示文に相当
- 2種類の指定方法：parallel, kernels
  - parallel：(どちらかというと) マニュアル
    - OpenMP に近い
    - 「ここからここまでは並列実行領域です。並列形状などはユーザー側で指定します」
  - kernels：(どちらかというと) 自動的
    - 「ここからここまではデバイス側実行領域です。あとはお任せします」
  - 細かい指示子・節を加えていくと最終的に同じような挙動になるので、どちらを使うかは好み
    - 個人的にはkernels推奨

kernels/parallel 指示文

kernels

```
program main

!$acc kernels
   do i = 1, N
      ! loop body
   end do
!$acc end kernels

end program
```

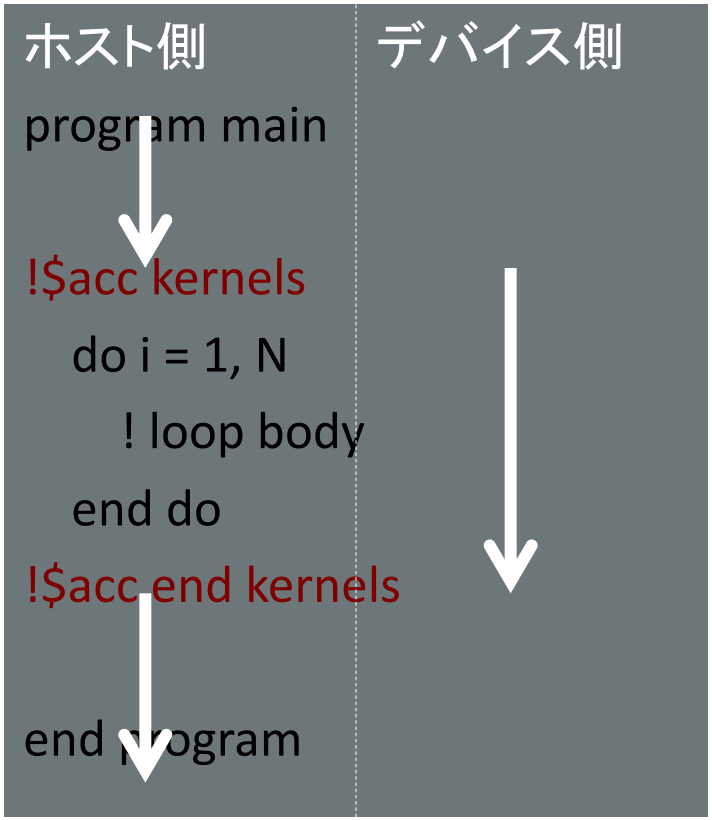
parallel

```
program main

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

end program
```

kernels/parallel 指示文

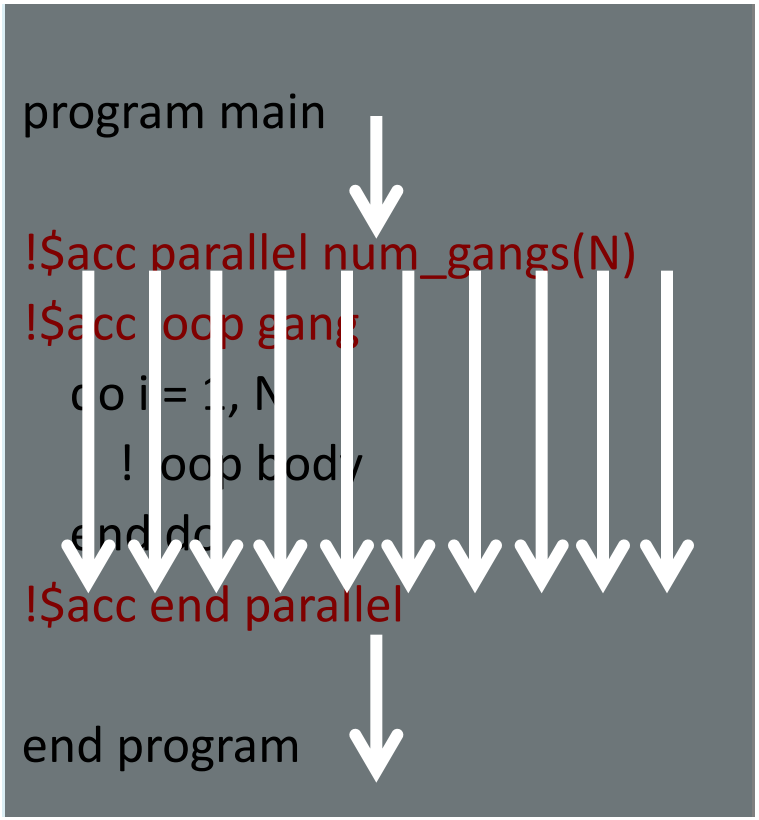
- ホスト-デバイスを意識するのがkernels
- 並列実行領域であることを意識するのがparallel

kernels



「並列数はデバイスに合わせてください」

parallel



「並列数Nでやってください」

kernels/parallel 指示文：指示節

kernels

- async
- wait
- device_type
- if
- default(none)
- copy…

parallel

- async
- wait
- device_type
- if
- default(none)
- copy…
- num_gangs
- num_workers
- vector_length
- reduction
- private
- firstprivate

kernels/parallel 指示文：指示節

kernels

非同期実行に用いる。

- async
- wait

実行デバイス毎にパラメータを調整

- device_type
- if
- default(none)

データ指示文の機能を使える

- copy…

parallelでは並列実行領域であることを意識するため、並列数や変数の扱いを決める指示節がある。

- num_gangs
- num_workers
- vector_length
- reduction
- private
- firstprivate

parallel

kernels/parallel 実行イメージ

Fortran

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

!$acc kernels copy(src,dis)
   do i = 1, N
      dis(i) = src(i)
   end do
!$acc end kernels

end subroutine copy
```

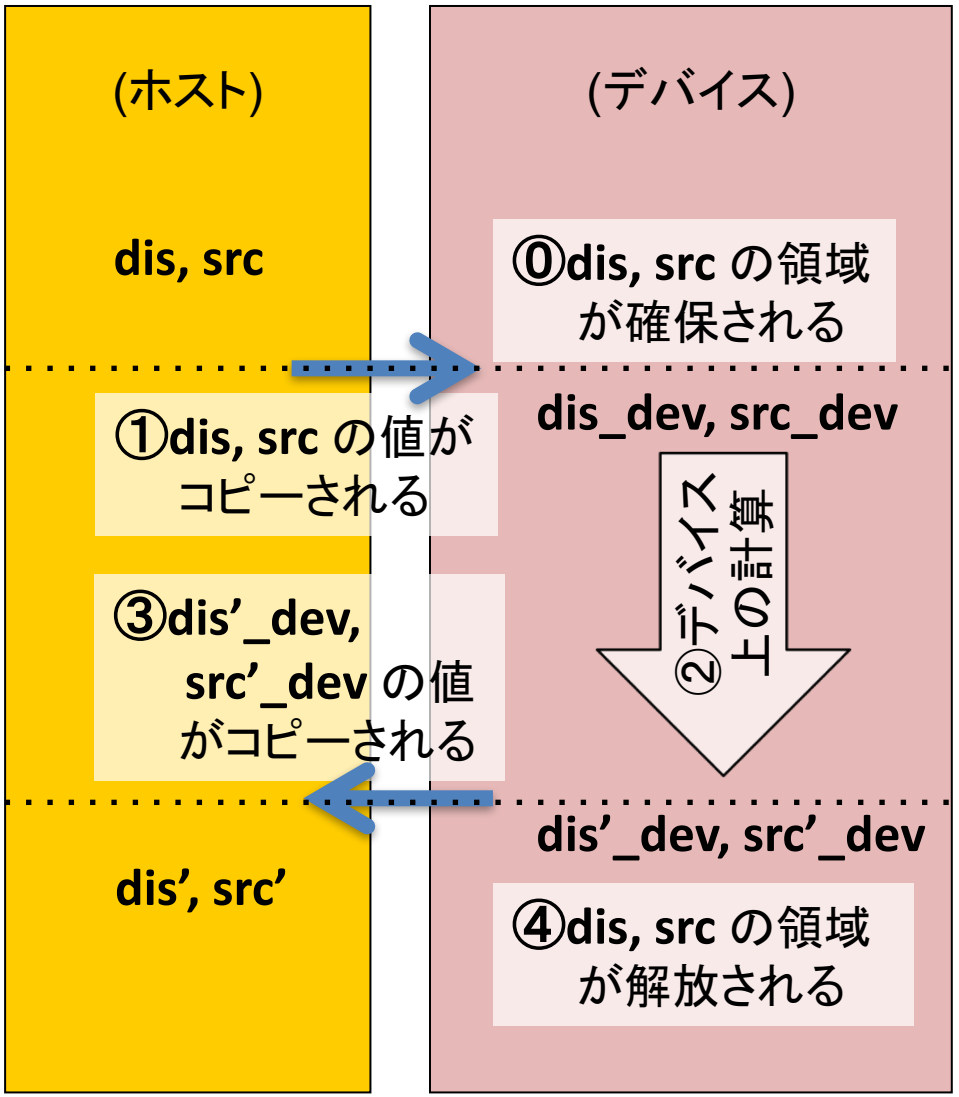
C言語

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

#pragma acc kernels copy(src[0:N] ¥
    dis[0:N])
     for(i = 0;i < N;i++){
        dis[i] = src[i];
     }

}
```

kernels/parallel 実行イメージ

Fortran

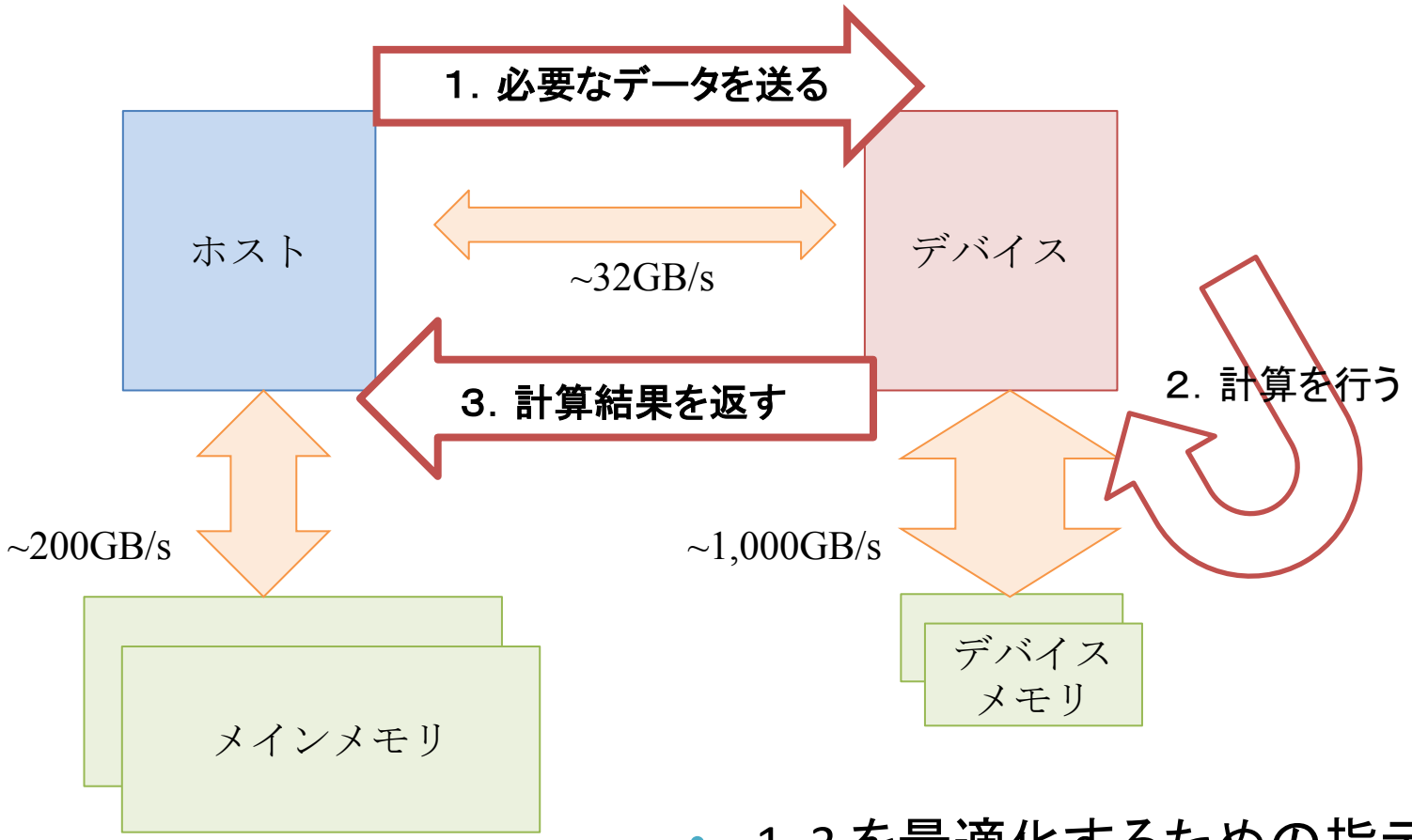
```
subroutine copy(dis, src)
  real(4), dimension(:) :: dis, src

!$acc kernels copy(src,dis)
  do i = 1, N
    dis(i) = src(i)
  end do
!$acc end kernels

end subroutine copy
```



デバイス上で扱われるべきデータについて

- プログラム上のparallel/kernels構文に差し掛かった時、OpenACCコンパイラは実行に必要なデータを自動で転送する
  - 正しく転送されないこともある。自分で書くべき
  - 構文に差し掛かるたびに転送が行われる(非効率)。後述のdata指示文を用いて自分で書くべき
  - 自動転送はdefault(none)で抑制できる
- スカラ変数は firstprivate として扱われる
  - 指示節により変更可能
- 配列はデバイスに確保される (shared的振る舞い)
  - 配列変数をスレッドローカルに扱うためには private を指定する

データ移動最適化指示文



- 1, 3 を最適化するための指示文
- ただしデータの一貫性を維持するのはユーザの責任

データ移動最適化指示文：data, enter/exit data

- デバイス側で必要なデータと範囲を指定
  - Allocate, Memcpy, Deallocate を行う
- data 指示文 (推奨)
  - Allocate + Memcpy (H ⇄ D) + Deallocate
  - 構造ブロックに対してのみ適用可
    - コードの見通しが良い
- enter data 指示文
  - Allocate + Memcpy (H → D)
  - exit data とセット。構造ブロック以外にも使える
- exit data 指示文
  - Memcpy (H ← D) + Deallocate
  - enter data とセット。構造ブロック以外にも使える

データ移動最適化指示文が必要なとき

Fortran

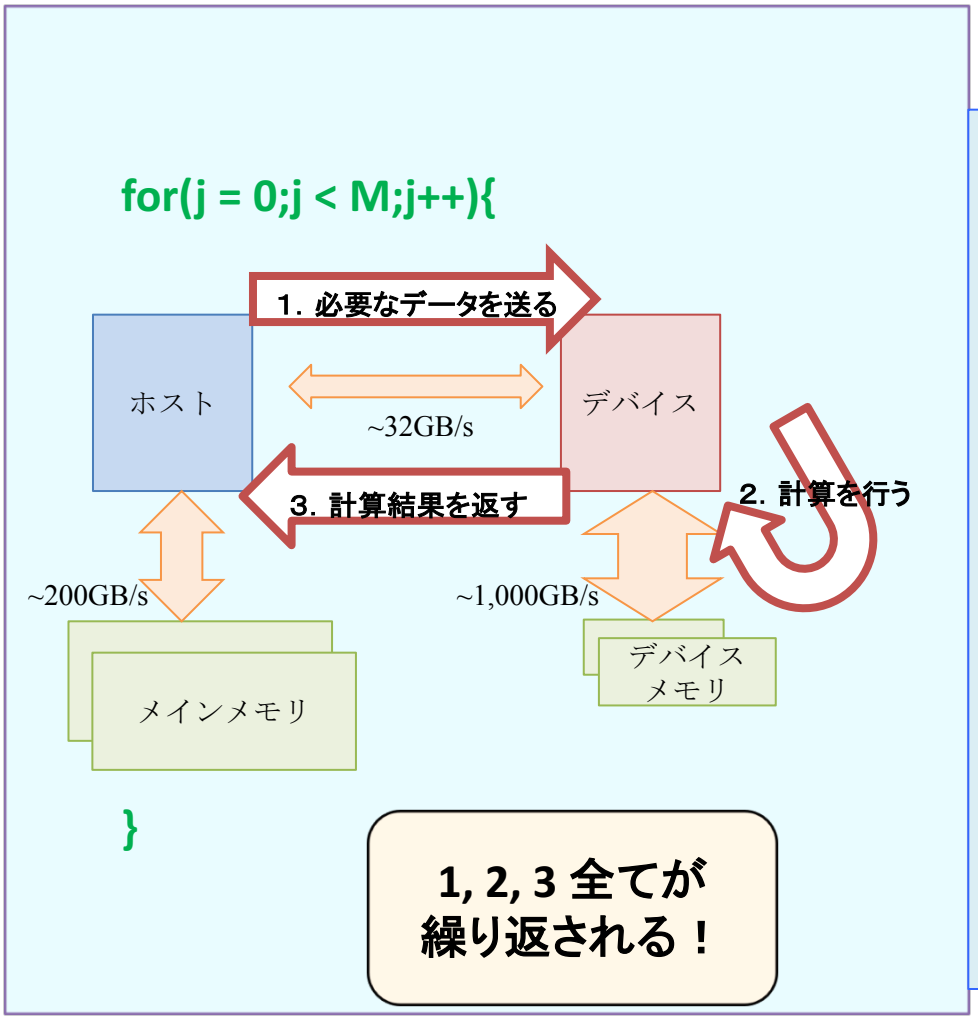
```
subroutine copy(dis, src)
  real(4), dimension(:) :: dis, src
  do j = 1, M
!$acc kernels copy(src,dis)
      do i = 1, N
         dis(i) = dis(i) + src(i)
      end do
!$acc end kernels
  end do
end subroutine copy
```

C言語

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

Kernels をループで囲むとどうなるか…

データ移動最適化指示文が必要なとき



C言語

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

Kernels をループで囲むとどうなるか…

data指示文

Fortran

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

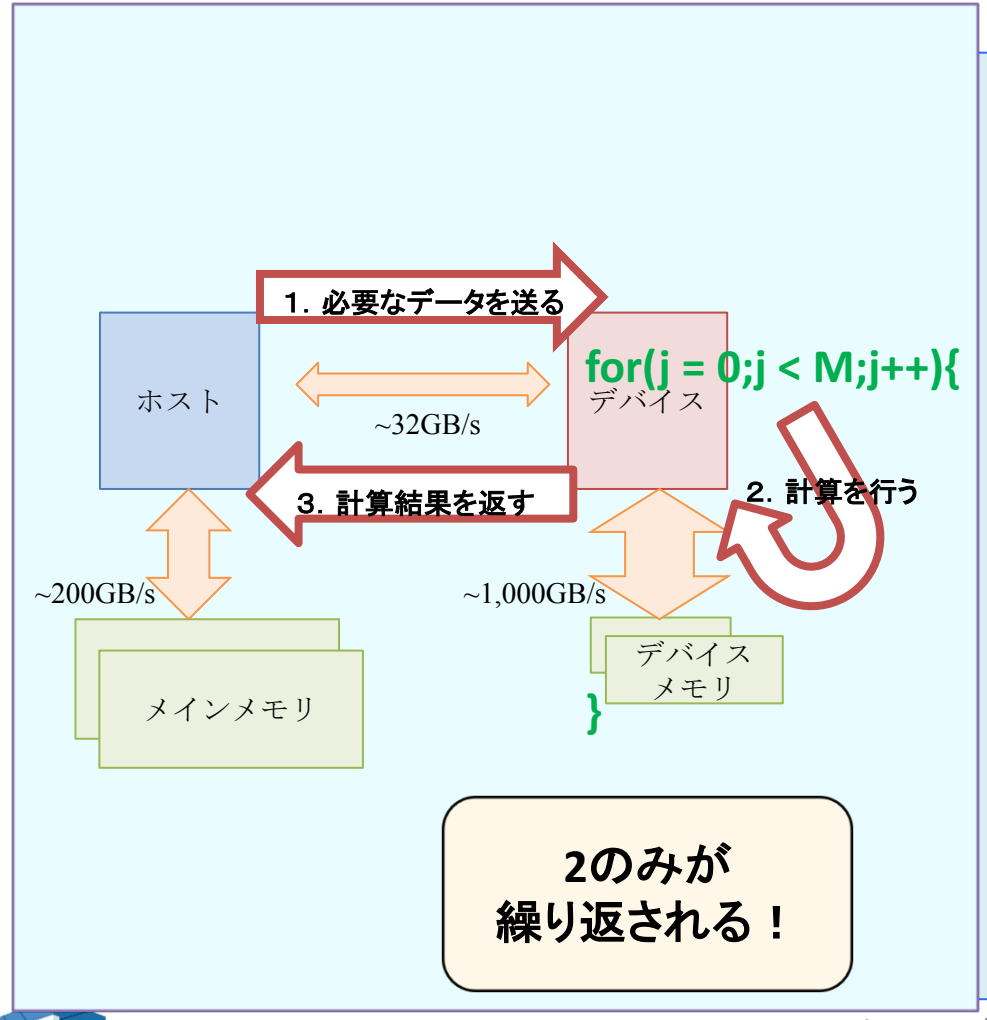
present: 既に転送済であることを示す
(OpenACC2.5の仕様以降、copyはpresent_or_copyとして扱われることになったので、実は書き換えなくても大丈夫になった。)

C言語

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

Cの場合、data指示文の範囲は{}で指定
(この場合はforが構造ブロックになってるのでなくても大丈夫だが)

data指示文



C言語

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

Cの場合、data指示文の範囲は{}で指定
(この場合はforが構造ブロックになってるのでなくても大丈夫だが)

enter data, exit data指示文

```
void main() {
  double *q;
  int step;
  for(step = 0;step < N;step++){
    if(step == 0) init(q);
    solverA(q);
    solverB(q);
     ….
    if(step == N) fin(q);
  }
}
```

```
void init(double *q) {
  q = (double *)malloc(sizeof(double)*M);
  q = … ; // 初期化
#pragma acc enter data copyin(q[0:M])
}
```

```
void fin(double *q) {
#pragma acc exit data copyout(q[0:M])
  print(q); //結果出力
  free(q);
}
```

data, enter/exit data 指示文の指示節

data

- if
- copy
- copyin
- copyout
- create
- present
- present_or_...
- deviceptr
  - CUDAなどと組み合わせる時に利用。cudaMallocなどで確保済みのデータを指定し、OpenACCで扱い可とする

enter data

- if
- async　　非同期転送用
- wait
- copyin
- create
- present_or_...

exit data

- if
- async
- wait
- copyout
- delete

data, enter/exit data 指示文の指示節

- **copy**
  - allocate, memcpy (H→D), memcpy (D→H), deallocate
- **copyin**
  - allocate, memcpy (H→D), deallocate　結果の出力を行わない
- **copyout**
  - allocate, memcpy (D→H), deallocate　データの入力を行わない
- **create**
  - allocate, deallocate コピーは行わない
- **present**
  - 何もしない。既にデバイス上にあることを教える
- **present_or_copy/copyin/copyout/create** (省略形：pcopy)
  - デバイス上になければ copy/copyin/copyout/create する。あれば何もしない

ただしOpenACC2.5以降では、
copy, copyin, copyout の挙動は
pcopy, pcopyin, pcopyout と同一

データ移動指示文：データ転送範囲指定

- 送受信するデータの範囲の指定
  - 部分配列の送受信が可能
  - 注意：FortranとCで指定方法が異なる
- 二次元配列Aを転送する例

Fortran版

```
!$acc data copy(A(lower1:upper1, lower2:upper2) )
  …
!$acc end data
```

fortranでは下限と上限を指定

C版

```
#pragma acc data copy(A[start1:length1][start2:length2])
  …
#pragma acc end data
```

Cでは先頭と長さを指定

データ移動指示文：update 指示文

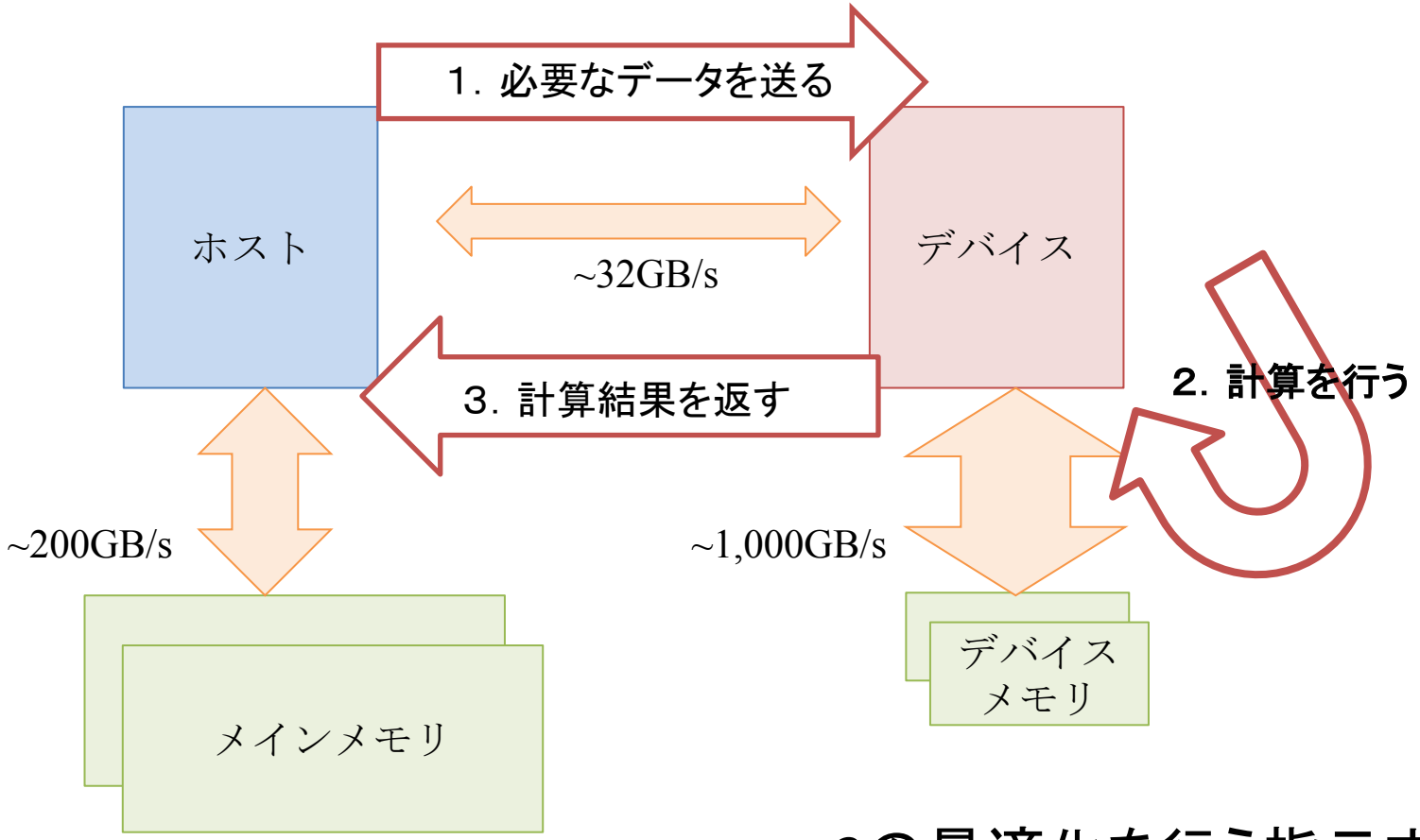
- データ指示文などで既にデバイス上に確保済みのデータを対象とする
  - Memcpy (H ⇄ D) の機能を持っていると思えば良い

```
!$acc data copy( A(:,:) )
do step = 1, N
  …
!$acc update host( A(1:2,:) )
  call comm_boundary( A )
!$acc update device( A(1:2,:) )
  …
end do
!$acc end data
```

update

- if
- async
- wait
- device_type
- self　#host と同義
- host　# H ← D
- device　# H → D

ループ最適化指示文



- 2の最適化を行う指示文

階層的並列モデルとループ指示文

- OpenACC ではスレッドを階層的に管理
  - gang, worker, vector の3階層
  - gang：workerの塊 一番大きな単位
  - worker：vectorの塊
  - vector：スレッドに相当する一番小さい処理単位
- loop 指示文
  - parallel/kernels中のループの扱いについて指示
    - パラメータの設定はある程度勝手にやってくれる
  - 粒度(gang, worker, vector)の指定
  - ループ伝搬依存の有無の指定

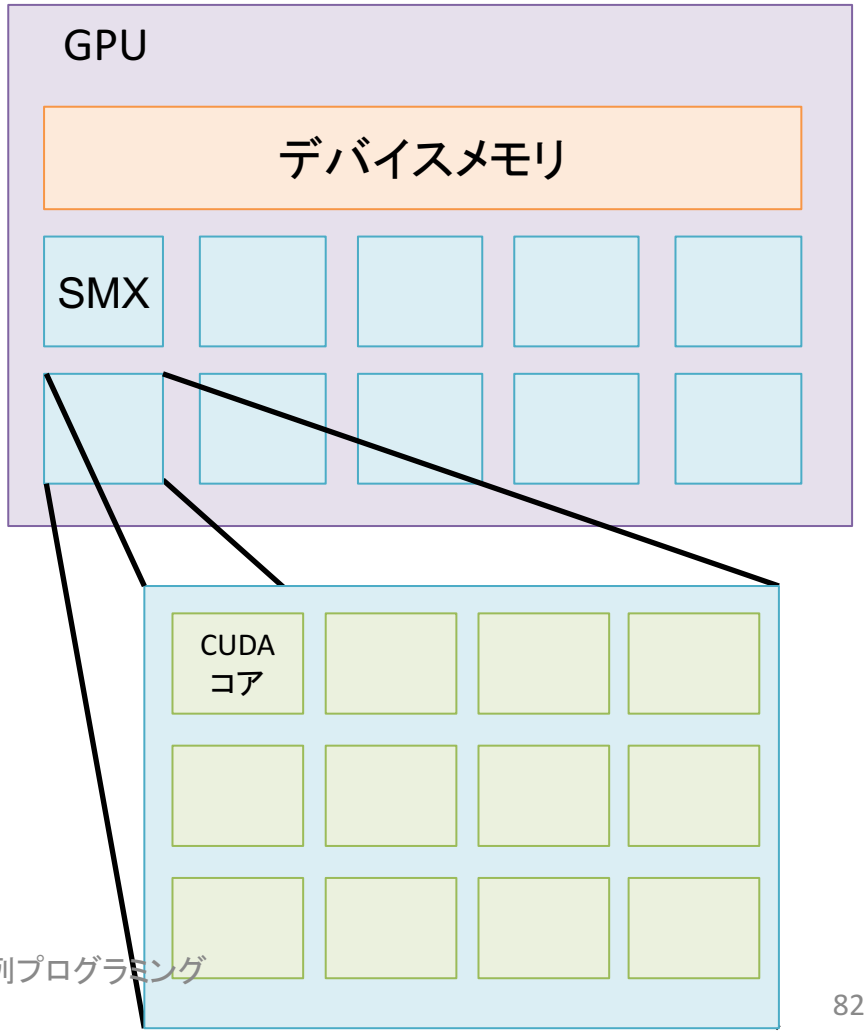
GPUでの行列積の例

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

階層的並列モデルとアーキテクチャ

- OpenMPは1階層
  - マルチコアCPUも1階層
  - 最近は2階層目(SIMD)がある
- CUDAは block と thread の2階層
  - NVIDA GPUも2階層
    - 1 SMX に複数CUDA coreを搭載
    - 各コアはSMXのリソースを共有
- OpenACCは3階層
  - 様々なアクセラレータに対応するため

- NVIDIA GPUの構成



ループ指示文:指示節

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

ループ指示文：指示節

loop

- **collapse**
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

3つのループが一重化される

```
!$acc kernels
!$acc loop collapse(3) gang vector
  do k = 1, 10
    do j = 1, 10
      do i = 1, 10

        ....

      end do
    end do
  end do
!$acc end kernels
```

並列化するにはループ長の短すぎるループに使う

ループ指示文：指示節

loop

- collapse
- **gang**
- **worker**
- **vector**
- seq
- auto
- tile
- device_type
- independent
- private
- reduction

```
!$acc kernels
!$acc loop gang(N)
   do k = 1, N
!$acc loop worker(1)
      do j = 1, N
!$acc loop vector(128)
         do i = 1, N

            ....

!$acc kernels
!$acc loop gang vector(128)
   do i = 1, N

      ....
```

vectorはworkerより内側
workerはgangより内側

ただし1つのループに
複数つけるのはOK

数値の指定は難しいので、最初は
コンパイラ任せでいい

ループ指示文：指示節

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- **independent**
- private
- reduction

Bに間接参照→

```
do j = 1, N
  do i = 1, N
    idxI(i) = i;  idxJ(j) = j
  end do
end do

!$acc kernels &
!$acc& copyin(A, idxI, idxJ) copyout(B)
!$acc loop independent gang
do j = 1, N
  !$acc loop independent vector(128)
  do i = 1, N
    B(idxI(i),idxJ(j)) = alpha * A(i,j)
  end do
end do
!$acc end kernels
```

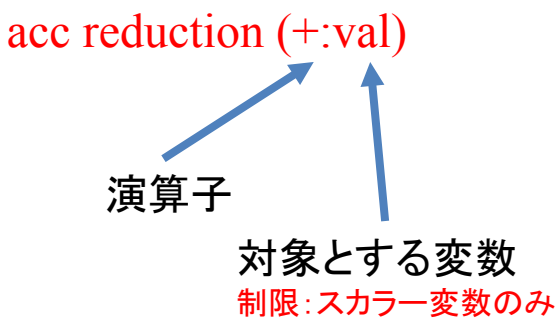
OpenACCコンパイラは保守的。依存関係が生じそうなら並列化しない。きちんと並列化されているかどうか、必ずコンパイラのメッセージを確認する。(やり方は後述)

ループ指示文：指示節

loop

- collapse
- gang
- worker
- vector
- seq
- auto
- tile
- device_type
- independent
- private
- **reduction**

```
!$acc kernels &
   !$acc loop reduction(+:val)
   do i = 1, N
      val = val + 1
   end do
!$acc end kernels
```



簡単なものであれば、PGIコンパイラは自動でreductionを入れてくれる

利用できる演算子
(OpenACC2.0仕様書より)

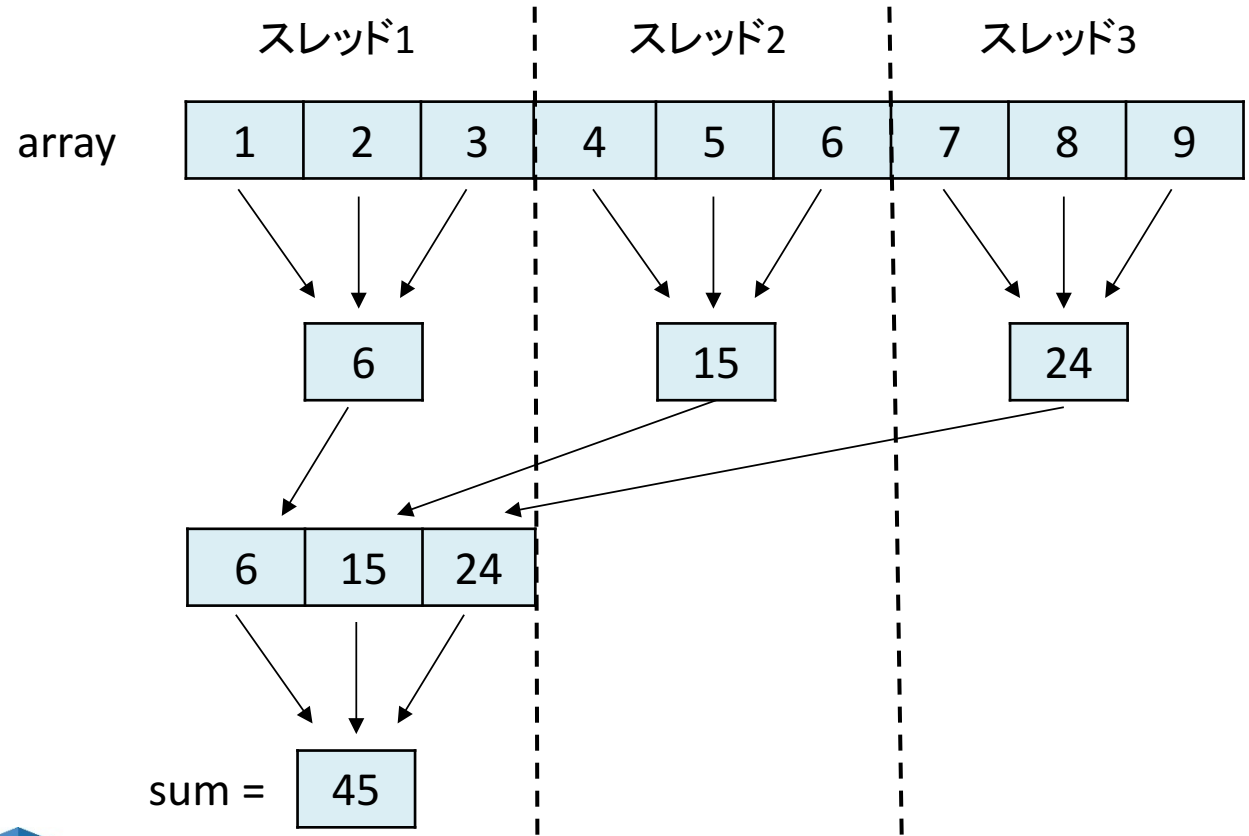
C and C++		Fortran	
operator	initialization value	operator	initialization value
+	0	+	0
*	1	*	1
max	least	max	least
min	largest	min	largest
&	~0	iand	all bits on
|	0	ior	0
^	0	ieor	0
&&	1	.and.	.true.
||	0	.or.	.false.
		.eqv.	.true.
		.neqv.	.false.

参考:リダクションでOpenACCが好きになる

- そもそもリダクションって？

```
sum = 0.0
for(i = 0;i < N;i++)
    sum += array[i]
```

リダクションが必要な例



1. 各スレッドが担当領域で縮約

2. 一時配列に書き込む

3. 一時配列を縮約

参考：リダクションでOpenACCが好きになる

- CUDAで実装しようと思うと…
  - 全部自分でやる
  - これはshuffle機能を使ったリダクション

```
int main( int argc, char* argv[] ){                    ホスト側プログラム
  ....
  double *tmp,*ans_d;
  cudaMalloc((void**)&tmp, sizeof(double) * 896);      一時配列の確保
  cudaMalloc((void**)&ans_d, sizeof(double) * 1);

  int chunk = (N+895)/896;
  dim3 dimGrid_L(896, 1, 1);                           使用するスレッド数の宣言
  dim3 dimBlock_L(128, 1, 1);
  dim3 dimGrid_G(1, 1, 1);
  dim3 dimBlock_G(1024, 1, 1);
                                                       GPUカーネル呼び出し
  reduction_L <<<dimGrid_L, dimBlock_L>>> (N,A_d,tmp,chunk);
  reduction_G <<<dimGrid_G, dimBlock_G>>> (tmp,ans_d);

  cudaMemcpy(&sum,ans_d,sizeof(double),cudaMemcpyDeviceToHost);
  ....                                                 結果の書き戻し
}
```

```
__global__ void reduction_L(int N, double *A, double *tmp, int chunk){
  __shared__ double sum_tmp[4];
  int tx = threadIdx.x;  int bx = blockIdx.x;                    GPUカーネル1
  int st = bx*chunk + tx;  int en = min(N,(bx+1)*chunk);
  double sum = 0.0;
  for(i = st;i < en;i+=128)
    sum += A[i];                                                 Warp shuffle機能を
  sum += __shfl_xor(sum,16);  sum += __shfl_xor(sum,8);          使ったWarp内の縮約
  sum += __shfl_xor(sum,4);  sum += __shfl_xor(sum,2);
  sum += __shfl_xor(sum,1);
  if(tx % 32 == 0)    sum_tmp[tx/32] = sum;
  __syncthreads();
  if(tx == 0) tmp[bx] = sum + sum_tmp[1] + sum_tmp[2] + sum_tmp[3];
}                                                    shared memoryを使ったWarp間の縮約
                                                     syncthreads()による同期必須
```

```
__global__ void reduction_G(double *tmp, double *ans){          GPUカーネル2
  __shared__ double sum_tmp[32];
  double sum;
  int tx = threadIdx.x;
  if(tx >= 896)
    sum = 0.0;                                                   一時配列も同様に縮約
  else
    sum = tmp[tx];
  sum += __shfl_xor(sum,16);  sum += __shfl_xor(sum,8);
  sum += __shfl_xor(sum,4);  sum += __shfl_xor(sum,2);
  sum += __shfl_xor(sum,1);
  if(tx % 32 == 0)    sum_tmp[tx/32] = sum;
  __syncthreads();
  if(tx < 32){
    sum = sum_tmp[tx];
    sum += __shfl_xor(sum,16);    sum += __shfl_xor(sum,8);
    sum += __shfl_xor(sum,4);    sum += __shfl_xor(sum,2);
    sum += __shfl_xor(sum,1);
  }
  __syncthreads();
  if(tx == 0) ans[0] = sum;
}
```

参考：リダクションでOpenACCが好きになる

- OpenACC なら…

```
sum = 0.0
#pragma acc kernels copyin(A[0:N])
#pragma acc loop reduction(+:sum)
for(i = 0;i < N;i++){
    sum += array[i]
}
```

これだけ！

- 性能も…
  1. OpenACC のリダクション
  2. 一旦CPUに書き戻して1スレッドで計算
  3. CPUに書きもどさず、GPUの1スレッドで計算
  4. CUDA の shuffleを使ったリダクション

array(倍精度)のサイズ100000000の時 (Fortran)

1. reduction acc time	:	0.00148 (s)
2. copyback and CPU time	:	0.63831 (s)
3. cuda 1 thread time	:	9.63381 (s)
4. reduction cuda time	:	0.00140 (s)

下手なプログラムを書くくらいなら
OpenACCに任せた方が速い！

関数呼び出し指示文：routine

- parallel/kernels領域内から関数を呼び出す場合、routine指示文を使う

```
#pragma acc routine vector
extern double vecsum(double *A);
…
#pragma acc parallel num_gangs(N) vector_length(128)
for (int i = 0;i < N; i++){
   max = vecsum(A[i*N]);
}
```

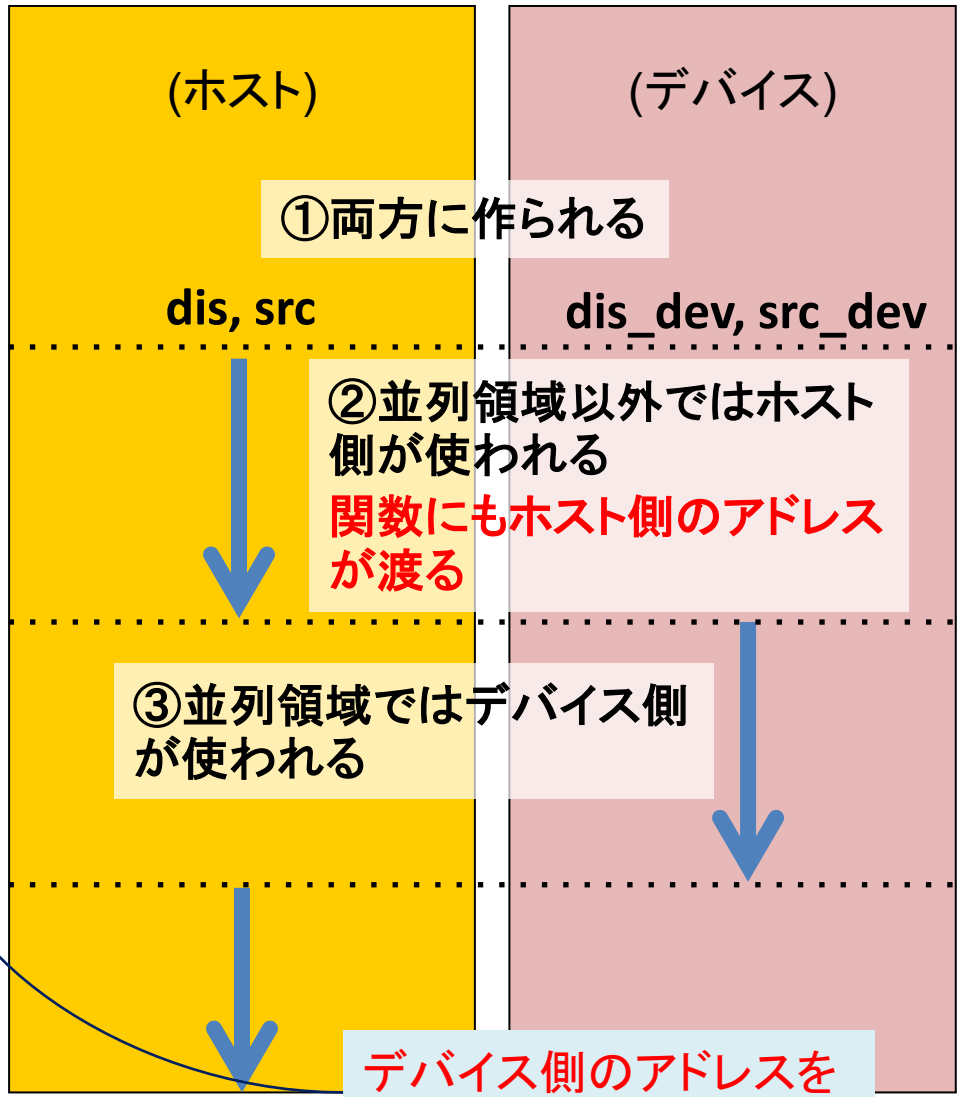
プロトタイプ宣言にもつける

```
#pragma acc routine vector
double vecsum(double *A){
   double x = 0;
#pragma acc loop reduction(+:x)
  for(int j = 0;j < N;j++){
      x += A[j];
  }
   return x;
}
```

host_data 指示文

- OpenACC のデータ指示文を使うと…

```
subroutine copy(dis, src)
  real(4), dimension(:) :: dis, src
!$acc data copy(src,dis)
  !-------------------------
  ! CPU上の処理
  !-------------------------
   call hoge(dis,src)

!$acc kernels present(src,dis)
     do i = 1, N
        dis(i) = dis(i) + src(i)
     end do
!$acc end kernels

!$acc end data
end subroutine copy
```



host_data 指示文

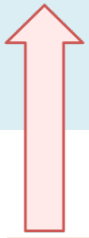
- 並列領域の外でデバイス側のアドレスを使うための指示文
  - データ指示文で確保済の配列が対象
- デバイス側のアドレスを使いたいケースって？
  - CUDAで書いた関数の呼び出し
  - GPU用のライブラリの呼び出し
  - GPU Direct によるMPI通信
    - GPU Direct：ホスト側のメモリを介さず、GPU間で直接MPIによるデータ通信をするもの

Fortran

```
!$acc data create(tmp) copy(val) copyin(A)
  ...
!$acc host_data use_device(A,tmp,val)
   call reduction_cuda_shuffle(val,N,A,tmp)
!$acc end host_data
  ...
!$acc end data
```

C

```
#pragma acc data create(tmp[0:896]) copy(val)
  {
    ...
#pragma acc host_data use_device(A,tmp,val)
    {
      reduction_cuda_shuffle(&val,N,A,tmp);
    }
    ...
  }
```



領域内ではデバイス側のアドレスが使われる

atomic 指示文

- 並列化領域内で、どうしても並列化できない部分が存在する場合に使う
  - 全体的には並列に計算できるが、書き込み先が衝突するようなケース
- Pascal GPU は倍精度のatomicAddをハードウェアサポートしてるので、それなりに速い

一文ずつ囲む

```
!$acc kernels async(0)
!$acc loop independent gang vector
    do k= inls,inle
      i= IAL(k)
      X1= Xin(3*i-2)
      X2= Xin(3*i-1)
      X3= Xin(3*i  )
      WVAL1= AL(k,1)*X1 + AL(k,2)*X2 + AL(k,3)*X3
      WVAL2= AL(k,4)*X1 + AL(k,5)*X2 + AL(k,6)*X3
      WVAL3= AL(k,7)*X1 + AL(k,8)*X2 + AL(k,9)*X3
      i = INL_G(k)
!$acc atomic
      tmpL(i,1) = tmpL(i,1) + WVAL1
!$acc end atomic
!$acc atomic
      tmpL(i,2) = tmpL(i,2) + WVAL2
!$acc end atomic
!$acc atomic
      tmpL(i,3) = tmpL(i,3) + WVAL3
!$acc end atomic
    enddo
!$acc end kernels
```

i が関節参照なので、書き込み先が衝突する可能性がある

OpenACC と Unified Memory

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

Unified Memoryのメリット・デメリット

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

```
!$acc kernels
!$acc loop independent
 do il=1,kt
!$acc loop independent
   do it=1,ndt; itt=it+nstrtt-1
    zbu(il)=zbu(il)+st_leafmtxp%st_lf(ip)%a1(it,il)*zu(itt)
   enddo
 enddo
!$acc end kernels
```

↑こう書くだけで正しく動くのは、従来のCUDAユーザーからすると革命的

- デメリット
  - ページ単位で転送するため、細かい転送が必要な場合には遅くなる
  - CPU側のメモリ管理を監視しているので、allocate, deallocateを繰り返すアプリではCPU側が極端に遅くなる
    - 今研究で使っているコードでは20倍近く遅くなった

アプリケーションの移植方法

アプリケーションのOpenACC化手順

1. プロファイリングによるボトルネック部位の導出
2. ボトルネック部位のOpenACC化
   1. 並列化可能かどうかの検討
   2. (OpenACCの仕様に合わせたプログラムの書き換え)
   3. parallel/kernels指示文適用
3. data指示文によるデータ転送の最適化
4. OpenACCカーネルの最適化

1 ~ 4 を繰り返し適用。それでも遅ければ、

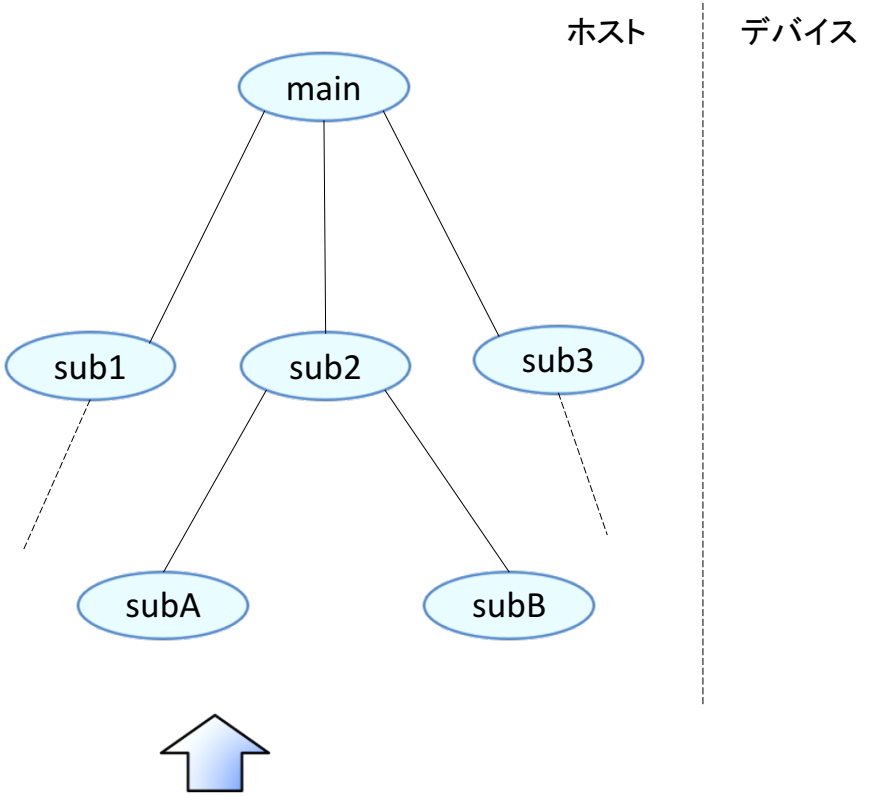
5. カーネルのCUDA化
   - スレッド間の相互作用が多いアプリケーションでは、shared memory や shuffle 命令を自由に使えるCUDAの方が圧倒的に有利

既にOpenMP化されているアプリケーションのOpenACC化手順

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

データ指示文による最適化手順

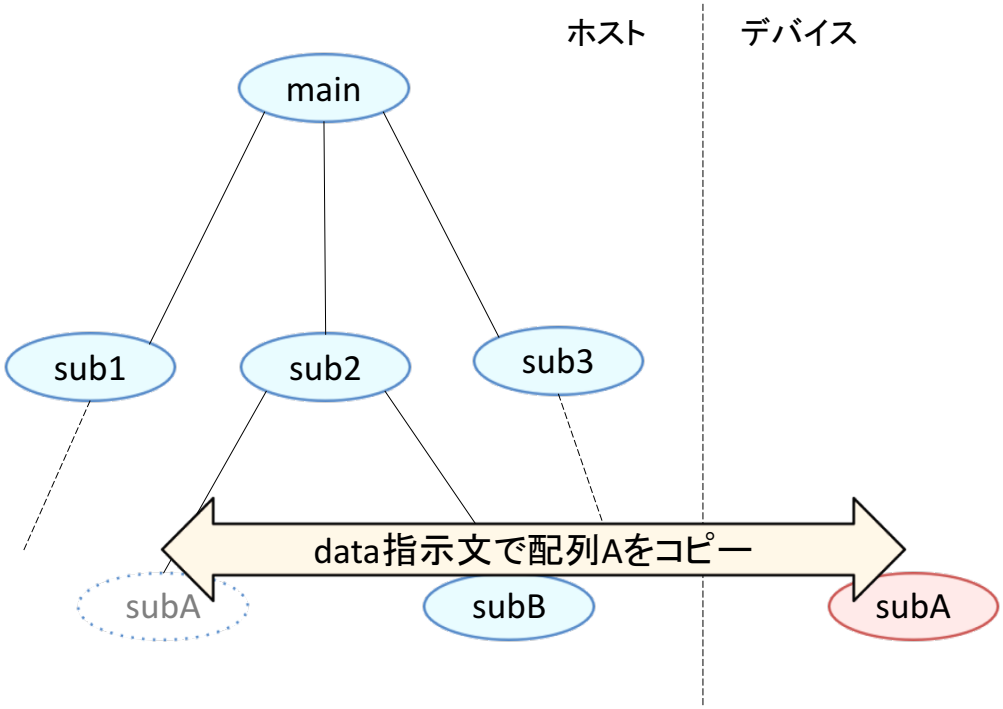
```
int main(){
  double A[N];
  sub1(A);
  sub2(A);
  sub3(A);
}

sub2(double A){
  subA(A);
  subB(A);
}

subA(double A){

  for( i = 0 ~ N ) {
    …
  }
}
```



葉っぱの部分から
OpenACC化を始める

データ指示文による最適化手順

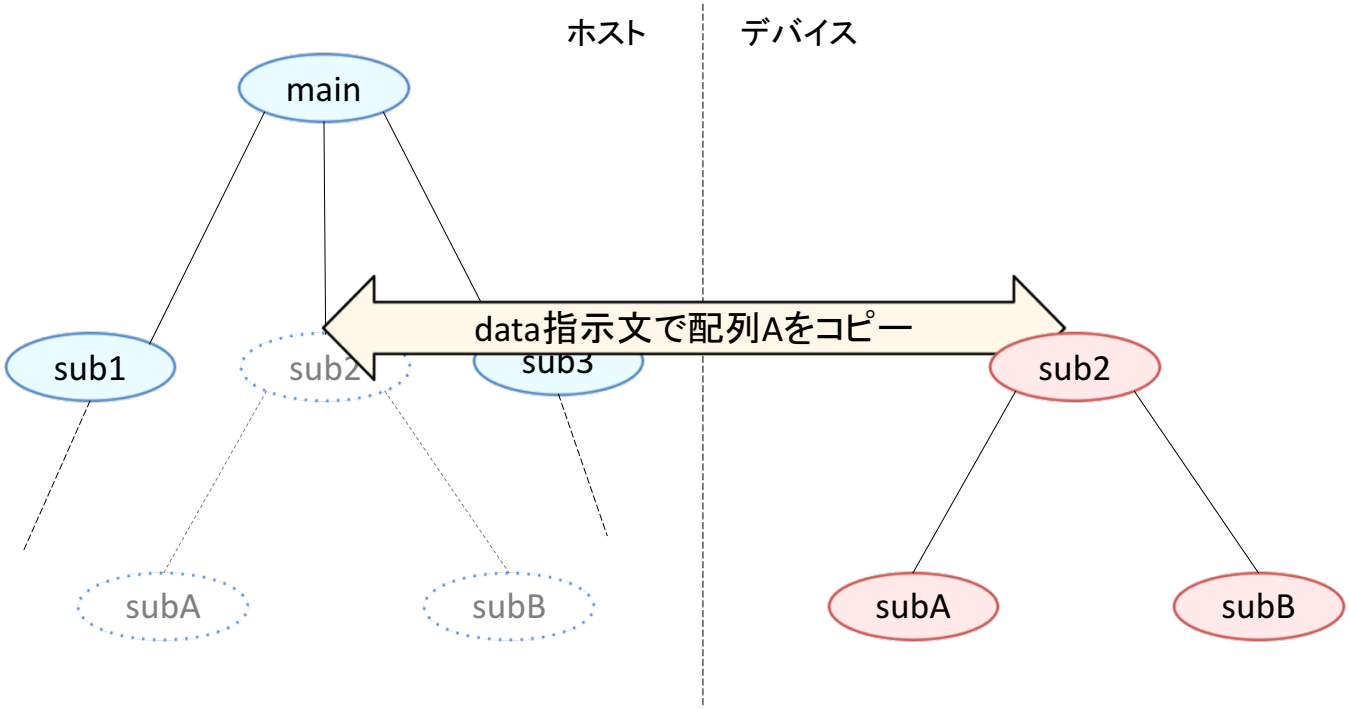
```
int main(){
  double A[N];
  sub1(A);
  sub2(A);
  sub3(A);
}

sub2(double A){
  subA(A);
  subB(A);
}

subA(double A){
#pragma acc …
  for( i = 0 ~ N ) {
    …
  }
}
```



この状態でも必ず正しい結果を得られるように作る！
この時、速度は気にしない！

データ指示文による最適化手順

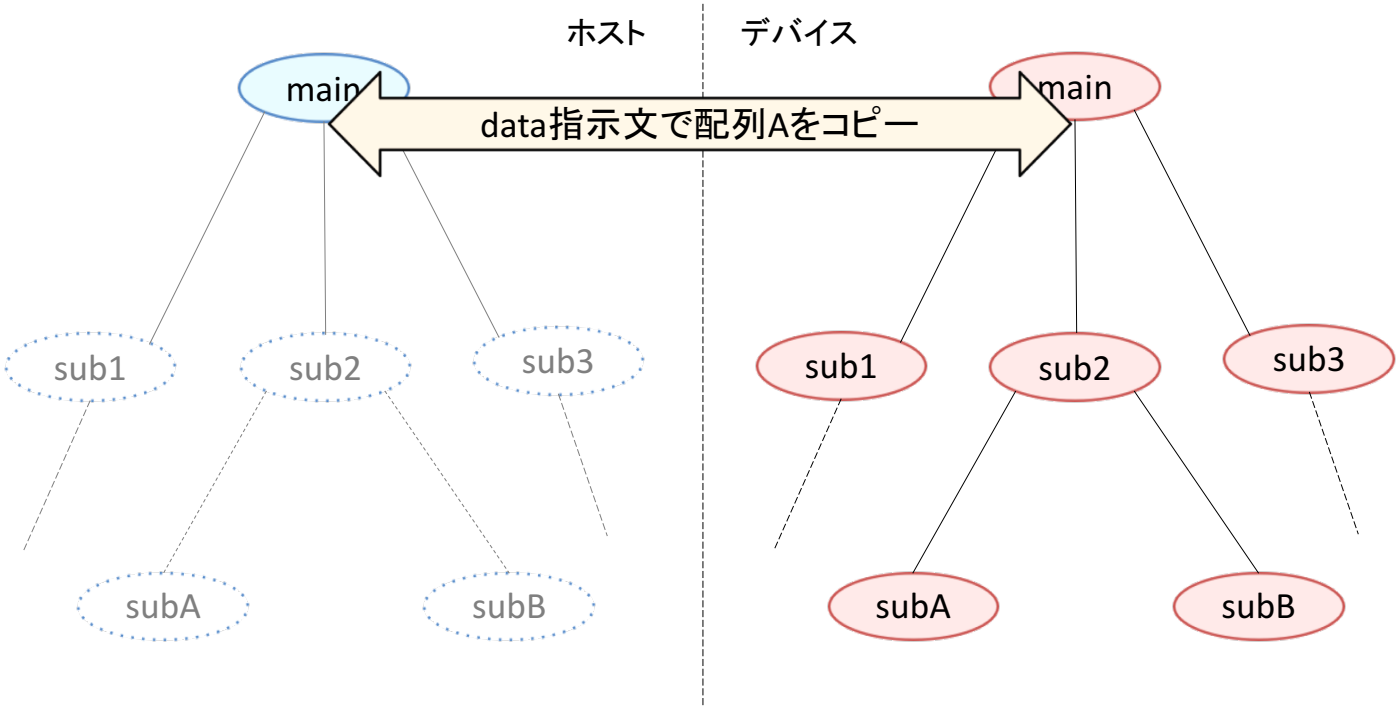
```
int main(){
  double A[N];
  sub1(A);
#pragma acc data
  {
    sub2(A);
  }
  sub3(A);
}

sub2(double A){
  subA(A);
  subB(A);
}

subA(double A){
#pragma acc …
  for( i = 0 ~ N ) {
    …
  }
}
```



徐々にデータ移動を上流に移動する

データ指示文による最適化手順

```
int main(){
  double A[N];
#pragma acc data
  {
    sub1(A);
    sub2(A);
    sub3(A);
  }
}

sub2(double A){
  subA(A);
  subB(A);
}

subA(double A){
#pragma acc …
  for( i = 0 ~ N ) {
    …
  }
}
```



ここまで来たら、ようやく個別のカーネルの最適化を始める。
※データの転送時間が相対的に十分小さくなればいいので、かならずしも最上流までやる必要はない

上で説明した指示文を用いてOpenACC化したもの

OpenACC の適用例

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

OpenACCを用いたICCG法ソルバーの Pascal GPUにおける性能評価

星野哲也・大島聡史・塙 敏博・中島研吾・伊田明弘

1) 東京大学情報基盤センター・最先端共同HPC基盤施設（JCAHPC）

2) 第158回 HPC研究会@熱海

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

メニーコアプロセッサと消費電力

- Green500 Top10 (2016/11)は全部メニーコア
- メニーコア向けのアルゴリズムや最適化手法の開発が重要

Green500 List for November 2016

Listed below are the November 2016 The Green500's energy-efficient supercomputers ranked from 1 to 10.

Green500 Rank	MFLOPS/W	Site	System	Total Power(kW)	
1	9462.1	NVIDIA Corporation	NVIDIA DGX-1, Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla P100	349.5	P100
2	7453.5	Swiss National Supercomputing Centre (CSCS)	Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100	1312	P100
3	6673.8	Advanced Center for Computing and Communication, RIKEN	ZettaScaler-1.6, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband FDR, PEZY-SCnp	150.0	PEZY
4	6051.3	National Supercomputing Center in Wuxi	Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	15371	Sunway
5	5806.3	Fujitsu Technology Solutions GmbH	PRIMERGY CX1640 M1, Intel Xeon Phi 7210 64C 1.3GHz, Intel Omni-Path	77	KNL
6	4985.7	Joint Center for Advanced High Performance Computing	PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path	2718.7	KNL
7	4688.0	DOE/SC/Argonne National Laboratory	Cray XC40, Intel Xeon Phi 7230 64C 1.3GHz, Aries interconnect	1087	KNL
8	4112.1	Stanford Research Computing Center	Cray CS-Storm, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, Nvidia K80	190	KNL
9	4086.8	Academic Center for Computing and Media Studies (ACCMS), Kyoto University	Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect	748.1	KNL
10	3836.6	Thomas Jefferson National Accelerator Facility	KOI Cluster, Intel Xeon Phi 7230 64C 1.3GHz, Intel Omni-Path	111	KNL

本研究の概要

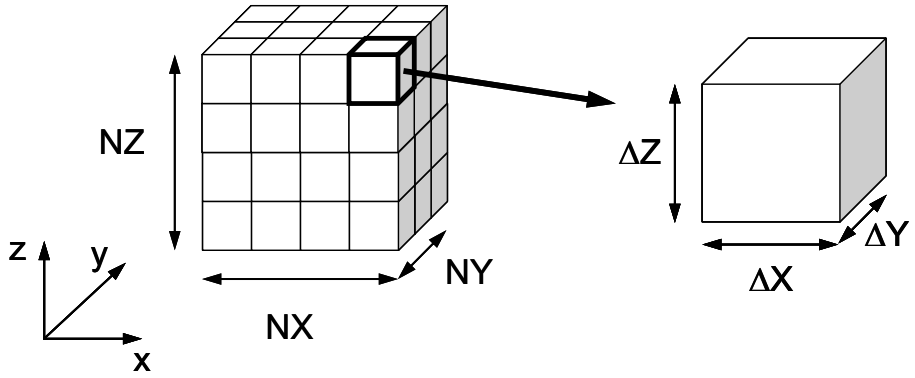
- 研究目的
  - 最新世代のGPU，Xeon Phiである、P100とKNLの性能評価
- 評価手法
  - OpenACC, OpenMPにより並列化したICCG法ソルバーを使用
  - P100とKNLの性能を比較
  - 前世代 (Kepler, KNC) と比較
  - それぞれへの最適化の適用
- 成果
  - P100において、指示文の最適化などにより1.21倍の性能向上を達成
  - KNLにおいて、同期の削減などにより1.20倍の性能向上を達成

発表概要

- 研究背景
- 対象アプリケーション
- ICCG法・行列格納手法
- OpenACCによる並列化
- P100 vs KNL (vs BDW, K20, KNC)
- P100・KNL向け最適化
- まとめ・今後の課題

対象アプリケーション

- 有限体積法，一様場ポアソン方程式ソルバー
  - [大島他 SWoPP 2014], [中島 HPC-139], [中島 HPC-147], [中島 HPC-157]
  - 差分格子：データ構造は非構造，7点ステンシル，問題サイズ128^3
  - 対称正定な疎行列を係数とする連立一次方程式
  - ICCG法，上下三角成分を別々に格納
- 色付け・リオーダリング
  - CM-RCM + Coalesced/Sequential
- 行列格納手法
  - CRS, Sliced-ELL, Sell-C-σ



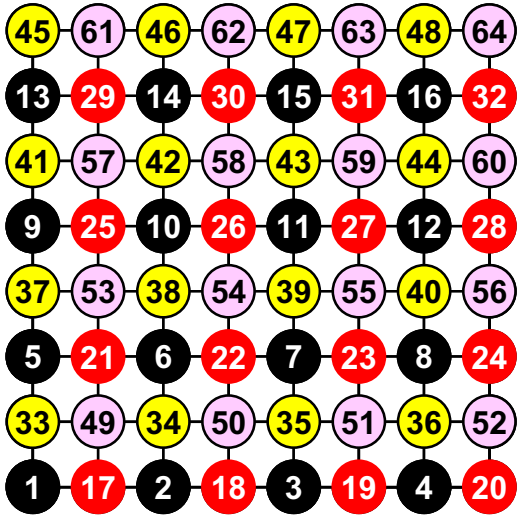
ICCG法

- 共役勾配法(Conjugate Gradient, CG)
- 前処理
  - 不完全コレスキー分解 (Incomplete Cholesky Factorization, IC)
  - 並列化にはカラーリングが必要

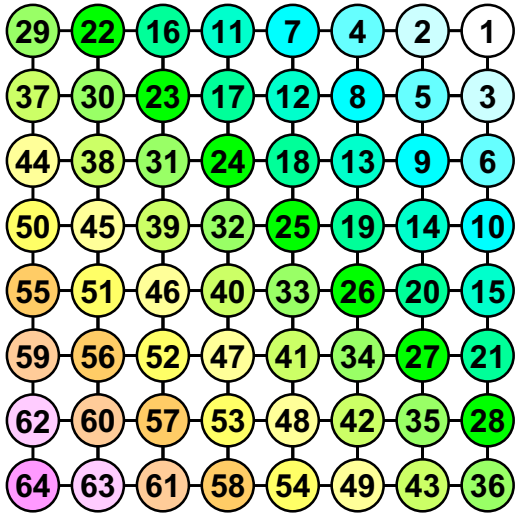
```
Compute r(0)= b-[A]x(0)
for i= 1, 2, …
    solve [M]z(i-1)= r(i-1)
    ρi-1= r(i-1) z(i-1)
    if i=1
      p(1)= z(0)
    else
      βi-1= ρi-1/ρi-2
      p(i)= z(i-1) + βi-1 p(i-1)
    endif
    q(i)= [A]p(i)
    αi  = ρi-1/p(i)q(i)
    x(i)= x(i-1) + αip(i)
    r(i)= r(i-1) - αiq(i)
    check convergence |r|
end
```

ICCG法並列化：データ依存性の解決

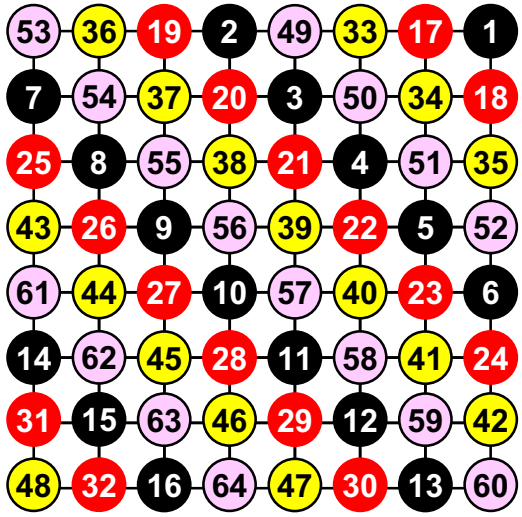
- 「色づけ＋リオーダリング」による並列性抽出
- 同色内は並列に実行可能
- 色数↑→ 収束性↑ 並列性↓ 同期コスト↑



MC (Color#=4)
Multicoloring



RCM
Reverse Cuthill-Mckee



CM-RCM (Color#=4)
Cyclic MC + RCM

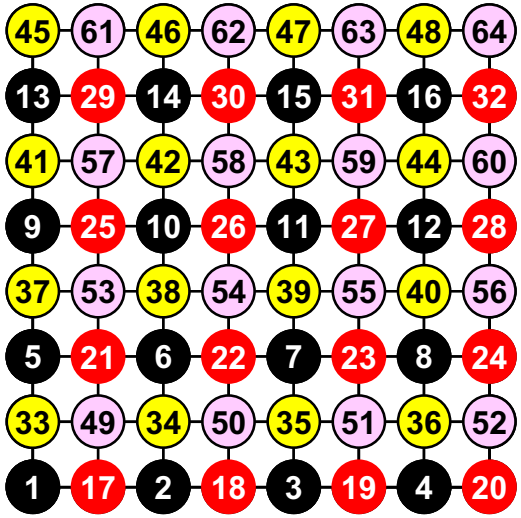
ICCG法並列化：データ依存性の解決

MC
- ● 並列性良
- ✕悪条件問題に弱い

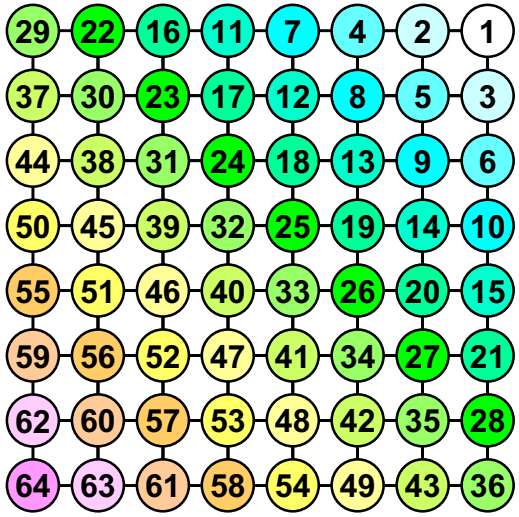
RCM
- ● 収束性良
- ✕並列性悪、同期コスト高

CM-RCM
- ● 並列性・収束性

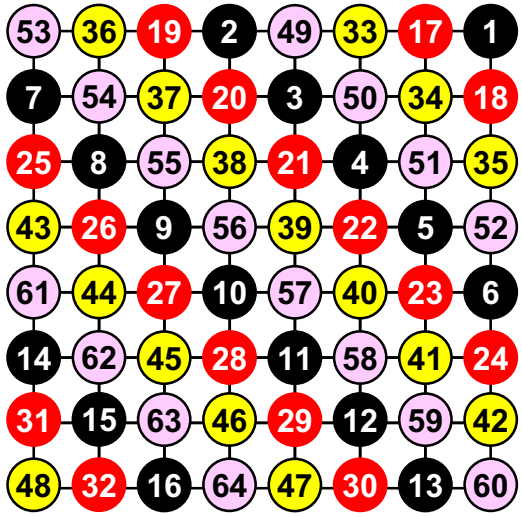
本発表で使用



MC (Color#=4)
Multicoloring



RCM
Reverse Cuthill-Mckee

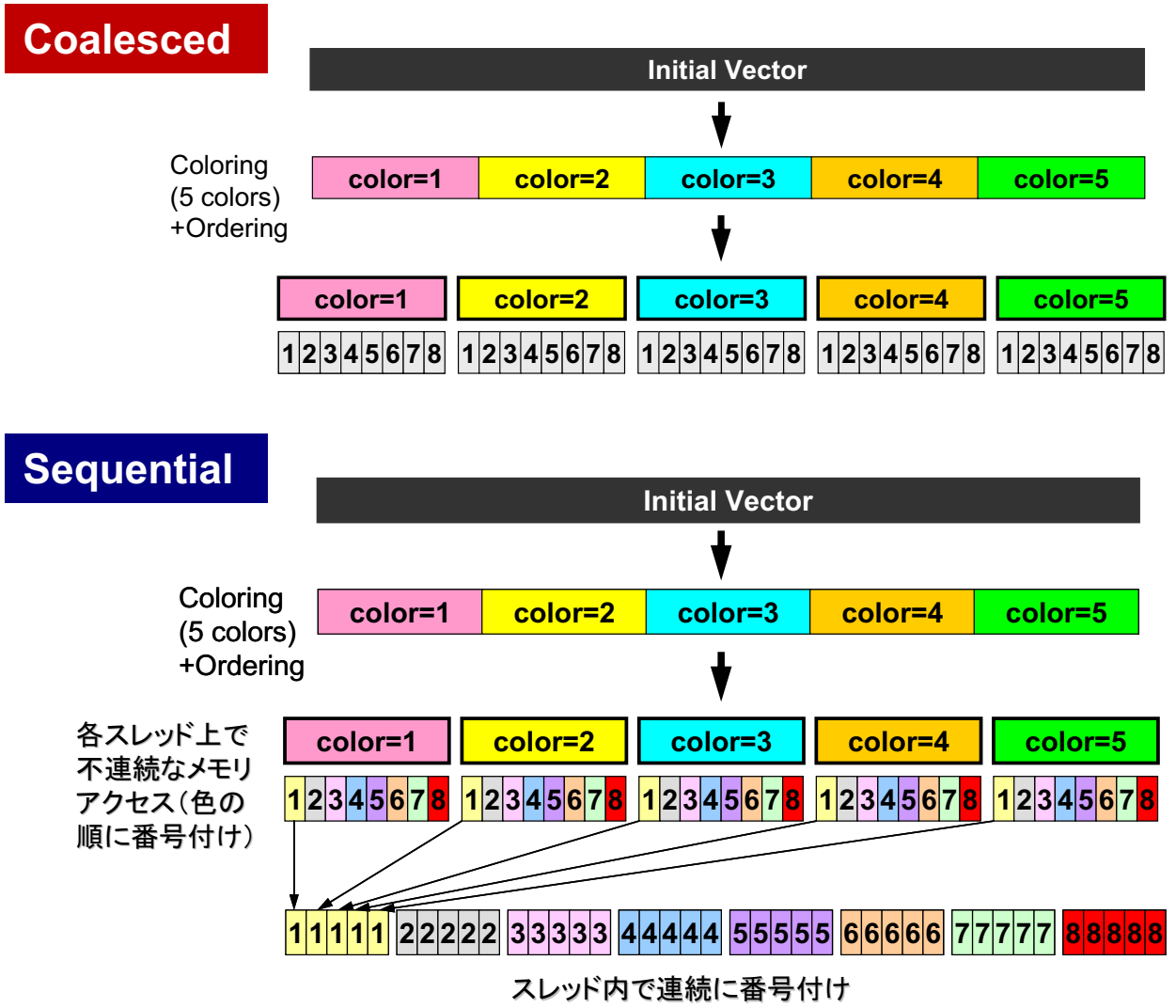


CM-RCM (Color#=4)
Cyclic MC + RCM

Coalesced, Sequential オーダリング

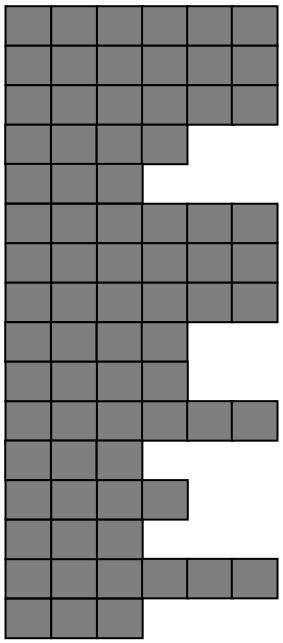
色順に並べ替えた後、

- Coalesced
  - データの順はそのまま
  - 各スレッドは不連続なメモリアクセス
- Sequential
  - データ順を再オーダリング
  - 計算単位(スレッドやコア)内で連続アクセス



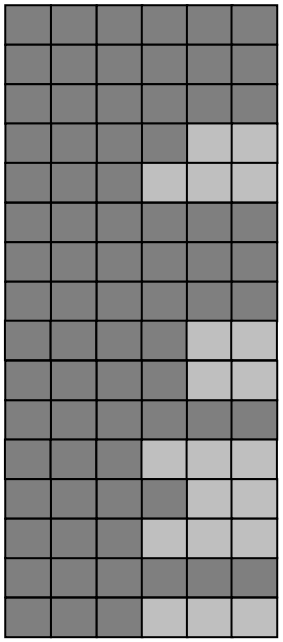
行列格納方法

一番一般的な格納方法



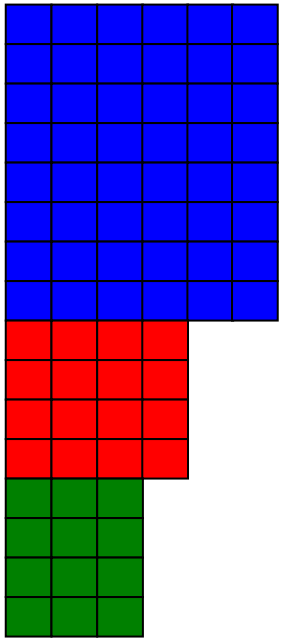
CRS

0でpadding



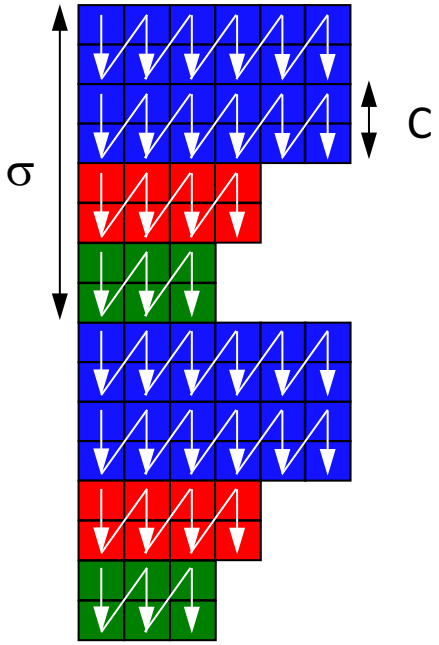
ELL

非零要素数でソート。複数配列で表現することで無駄な計算を削減



Sliced ELL

CをSIMD長等に合わせ、適宜0 padding で整形



SELL-C-σ
(SELL-2-8)

実施ケース

略称	Numbering	格納形式
c-CRS	Coalesced	CRS
c-Sliced-ELL		Sliced-ELL (ブロッキングあり)
c-SELL-C-σ		SELL-C-σ
s-CRS	Sequential	CRS
s-Sliced-ELL		Sliced-ELL (ブロッキングあり)
s-SELL-C-σ		SELL-C-σ

カラーリングはいずれもCM-RCM
問題サイズは128^3

OpenACC/OpenMP実装

- 同一ソースコードにそれぞれの指示文を挿入
  - HPC157から、不要な一時配列の除去などの最適化
  - HPC157から、一部最適化を評価のため除外(OpenMP)

- OpenACC
  - 全並列化ループに **!$acc kernels** を適用
  - データ転送は計測外で行う
  - 予稿から最適化レベルを1段階ダウン
    - async 指示節を除外
- OpenMP
  - 全並列化ループに **!$omp parallel do** を適用

```
!$omp parallel do private(... )
!$acc kernels
!$acc loop independent gang
    do ip= 1, PEsmpTOT
      ip1= (ip-1)*NCOLORtot + ic
!$omp simd
!$acc loop independent vector
      do i= index(ip1-1)+1, index(ip1)
        ...
      enddo
    enddo
!$omp end parallel do
!$acc end kernels
```

Hardware

- **P100**: NVIDIA Tesla P100 (予稿はPCI-E版)
  - Reedbush-H
- **K20**: NVIDIA Tesla K20Xm
  - TSUBAME2.5
- **KNC**: Intel Xeon Phi 5110P (Knights Corner)
  - KNSC
- **KNL**: Intel Xeon Phi 7250 (Knights Landing)
  - Oakforest-PACS
- **BDW**: Intel Xeon E5-2695 v4 (Intel Broadwell-EP)
  - Reedbush-U

Hardware spec

略　称	P100	K20	KNL	KNC	BDW
動作周波数（GHz）	1.480	0.732	1.40	1.053	2.10
コア数（最大有効スレッド数）	1,792	896	68 (272)	60 (240)	18 (18)
理論演算性能（GFLOPS）	5,304	1,311.7	3,046.4	1,010.9	604.8
主記憶容量（GB）	16	6	16	8	128
メモリバンド幅（GB/sec., Stream Triad）	**534**	**179**	**490**	**159**	**65.5**
System	Reedbush-H	TSUBAME2.5	Oakforest-PACS	KNSC	Reedbush-U

コンパイラ･環境変数
P100

- コンパイラ&オプション
  - pgfortran 17.1 (予稿は16.10)
  - -O3 –ta=tesla:cc60

- 環境変数
  - 特になし

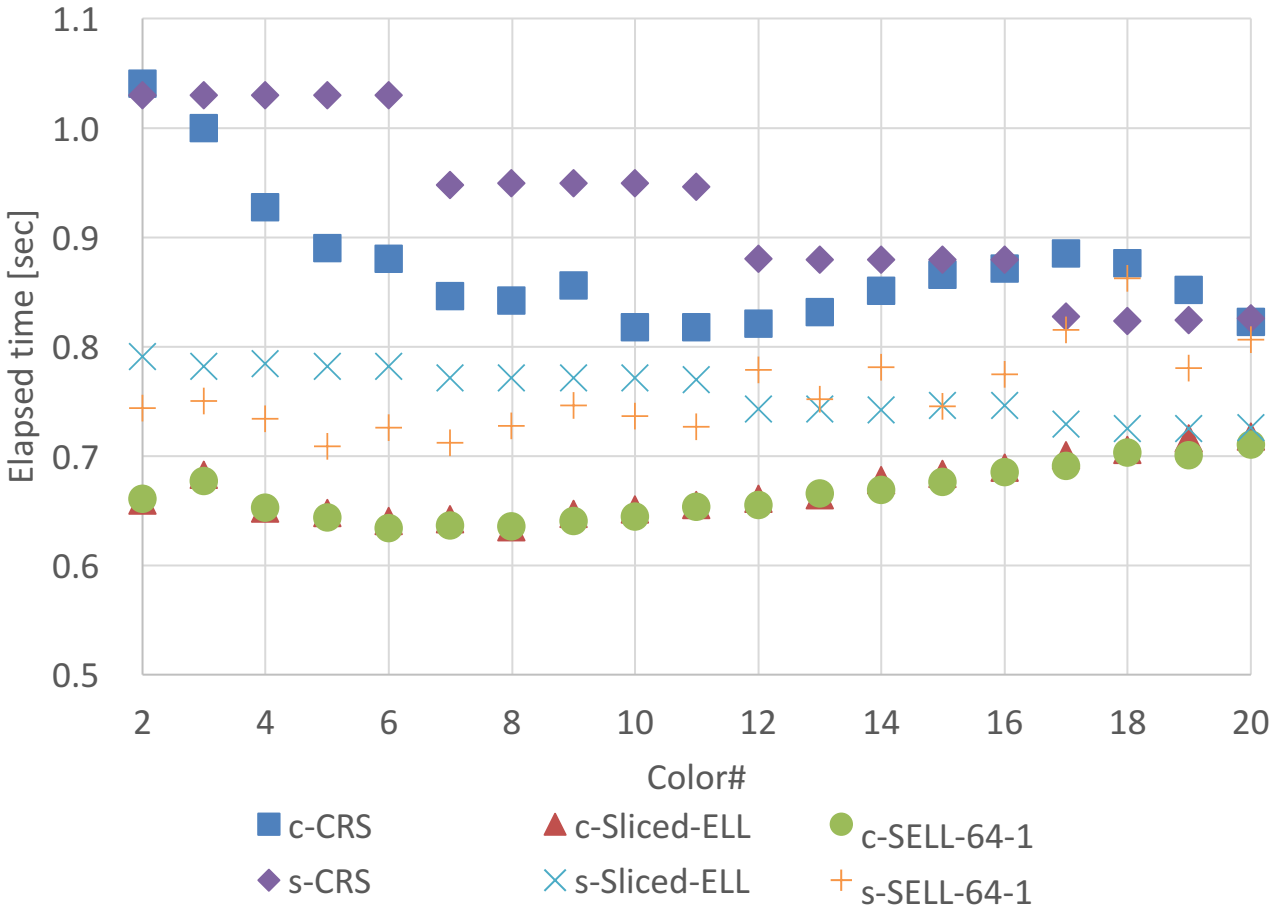
コンパイラ・環境変数
KNL

その他
・numactl –membind=1 で MCDRAMを使用
・メモリモデル：Flat
・サブNUMAモード：Quadrant

- コンパイラ&オプション
  - ifort (IFORT) 17.0.0 20160721
  - -align array64byte -O3 -xMIC-AVX512 -qopenmp -qopt-streaming-stores=always -qopt-streaming-cache-evict=0
    - ただしCRSを使うときは-qopt-streaming-stores=never (予稿ではalways)
- 環境変数
  - export OMP_STACKSIZE=1G
  - ulimit -s 1000000
  - 66 スレッドの時
    - export OMP_NUM_THREADS=66
    - export KMP_AFFINITY=granularity=fine,proclist=[2-67],explicit
  - 132/198/264 スレッドの時
    - export OMP_NUM_THREADS=132/198/264
    - export KMP_AFFINITY=compact
    - export KMP_HW_SUBSET=66c@2,2t/3t/4t

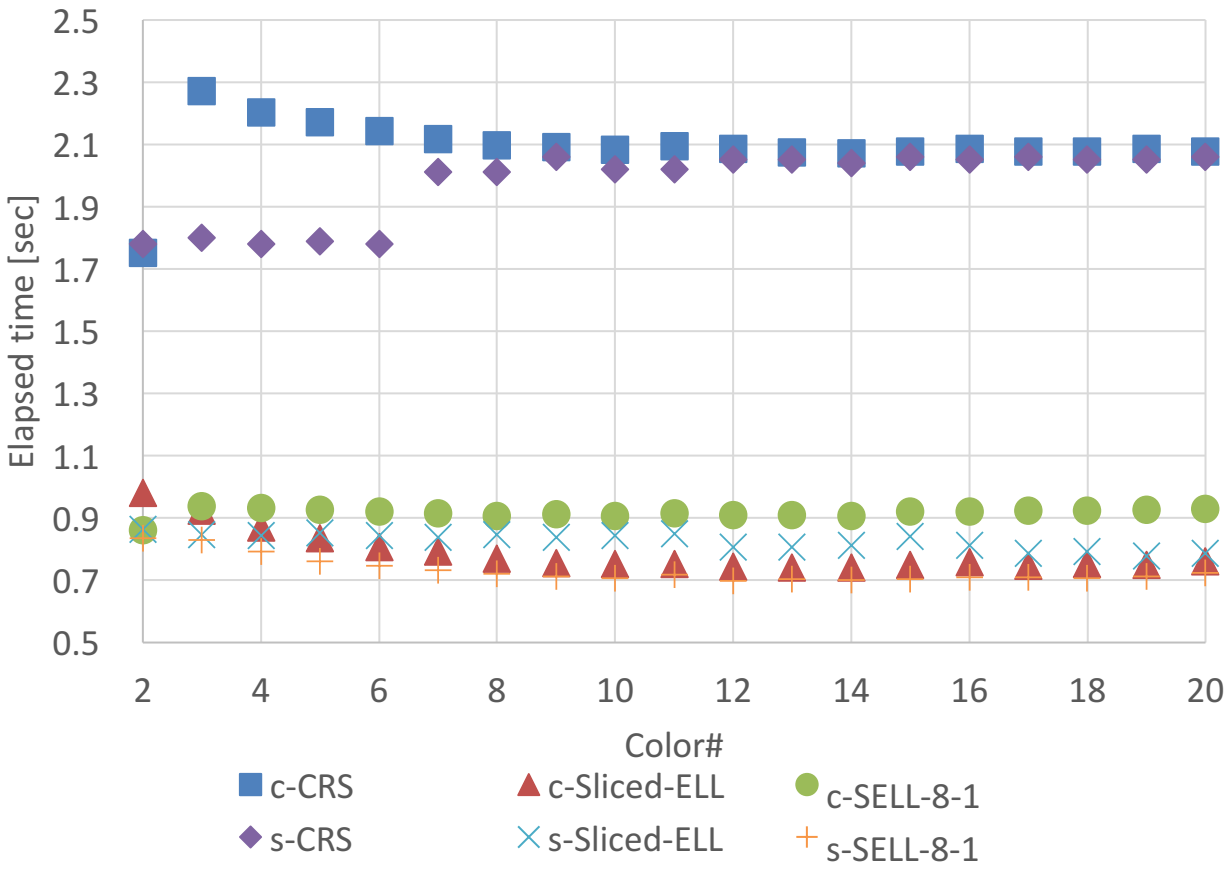
P100

- Coalesced版が速い
- Sell-C-σ(C=64, σ=1)とSliced-ELLが同程度で最速
- CRSが3割程度遅い
- 色数6程度が最速
  - 原因：同期コスト(カーネル呼び出し)
  - 予稿と違うのはasync外したため



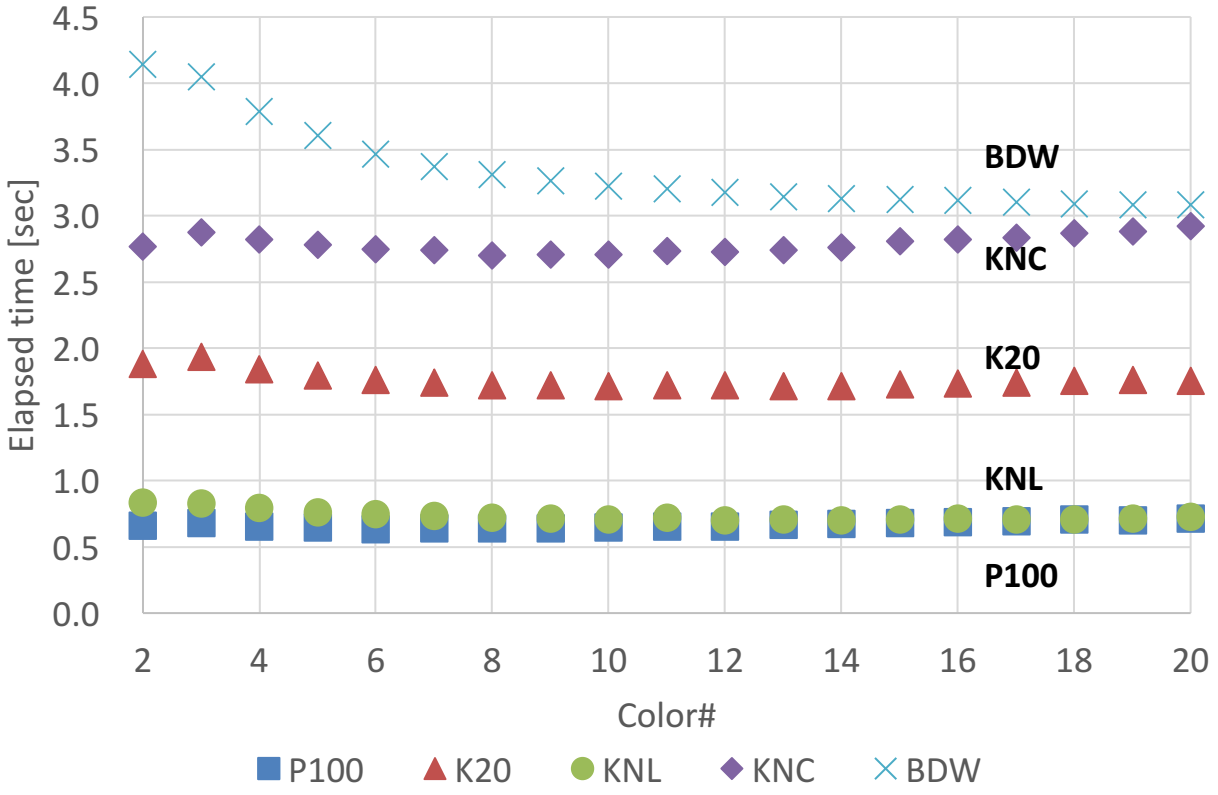
KNL

- Sequential版が速い
- Sell-C-σ(C=8, σ=1)が最速
- CRSが2.5倍程度遅い
- 色数14程度が最速
- 66/132/198/264 スレッド中 66 スレッドが最速



各プロセッサの性能比較

- 各プロセッサ最速の疎行列形状の結果
- K20 vs KNC
  - KNCが1.57倍も遅い
- P100 vs KNL
  - KNLが1.10倍しか遅くない
- P100 vs K20
  - P100はK20の2.71倍程度



実行効率 (vs. BDW)

- 色数10の時の比較
- メモリバンド幅X倍で性能もX倍になると仮定すると、P100・KNLの実行効率はBDWの61%程度

略　称	P100	K20	KNL	KNC	BDW
メモリバンド幅（GB/sec., Stream Triad）	534	179	490	159	65.5
相対メモリバンド幅	8.152	2.733	7.481	2.427	1
実行時間	0.6448	1.714	0.7061	2.708	3.224
相対性能	5.00	1.88	4.57	1.19	1
(相対性能)/(相対メモリバンド幅)	**0.613**	**0.688**	**0.610**	**0.490**	**1**

最適化(OpenACC)

1. Baseline
   - 全並列ループに!$acc kernels
2. Async (予稿のBaseline)
   - !$acc kernels を !$acc kernels async(0) と置き換え
3. Thread
   - gang, vector などのパラメータを調整

↑指示文のみの変更

↓コードの変更

4. Fusion
   - SPMV部分のカーネルを融合

最適化 3 Thread

- リダクション部分のスレッド数を調整
  - 目的：ローカルリダクションで生成される一時配列のサイズを小さくする
- デフォルト設定
  - ローカルリダクション
    - gang = (N-1)/128+1
    - vector = 128
  - グローバルリダクション
    - gang = 1
    - vector = 256

```
!$omp parallel do private(i) reduction(+:RHO)
!$acc kernels async(0)
!$acc loop independent gang(448) vector reduction(+:RHO)
    do i= 1, N
      RHO= RHO + W(i,R)*W(i,Z)
    enddo
!$acc end kernels
```

↑ 448要素の一時配列ができ、
その配列を1 gangでリダクション

← 16384要素の一時配列ができ、
その配列を1 gangでリダクション

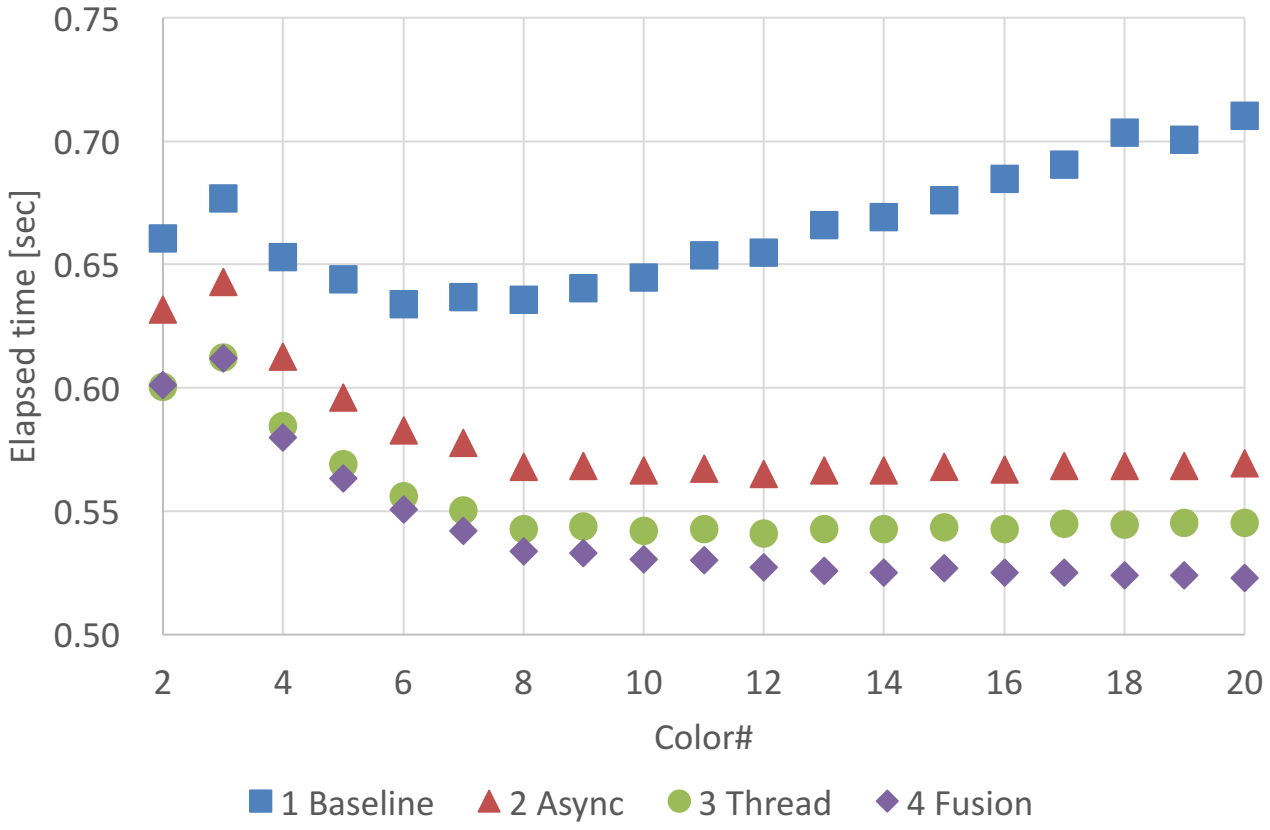
最適化 4 Fusion

- カーネルを融合し、呼び出しコストを削減
- SpMV部分では、色1, 色2~N-1, 色N の時で処理内容が異なる
  - 色2~N-1のカーネルをN-2回呼び出す
- 色2~N-1の処理を、複数のカーネルから1カーネルに融合

```
!C
!C   SpMV
!C

do ic = 1, NCOLORtot        ! 色ループ
   if( ic == 1 ) then

      ! 色1の時の処理

   else if ( 2 <= ic, ic < NCOLORtot) then

      ! 色2 ~ 色NCOLORtot-1の時の処理

   else

      ! 色NCOLORtot の時の処理

   endif
enddo
```

P100 最適化結果

- asyncの効果大
  - カーネル起動オーバーヘッドを隠せる
- Baselineから1.21倍の性能向上
- BDW比実行効率 61.3%→74.5%



最適化(OpenMP)

1. Baseline
   - 全並列ループに !$omp parallel do
2. mvparallel1
   - !$omp parallel を色ループの外に
3. nowait
   - !$omp end do nowait を使う(SpMV部分)
4. mvparallel2
   - !$omp parallelを収束判定ループの外に

↑指示文のみの変更

↓コードの変更

5. rmompdo
   - !$omp doを使わず自力で(機械的に)ループ分割 (reduction以外)
6. rmreduction
   - reductionを自分で書く
7. loopschedule
   - ループスケジューリングの変更

Move !$omp paralell

Baseline

```
do L= 1, ITR   ! 収束判定ループ

  do ic = 1, NCOLORtot   !色ループ
!$omp parallel do
        do ...
        end do
!$omp end parallel do
  end do

enddo
```

2 mvparallel

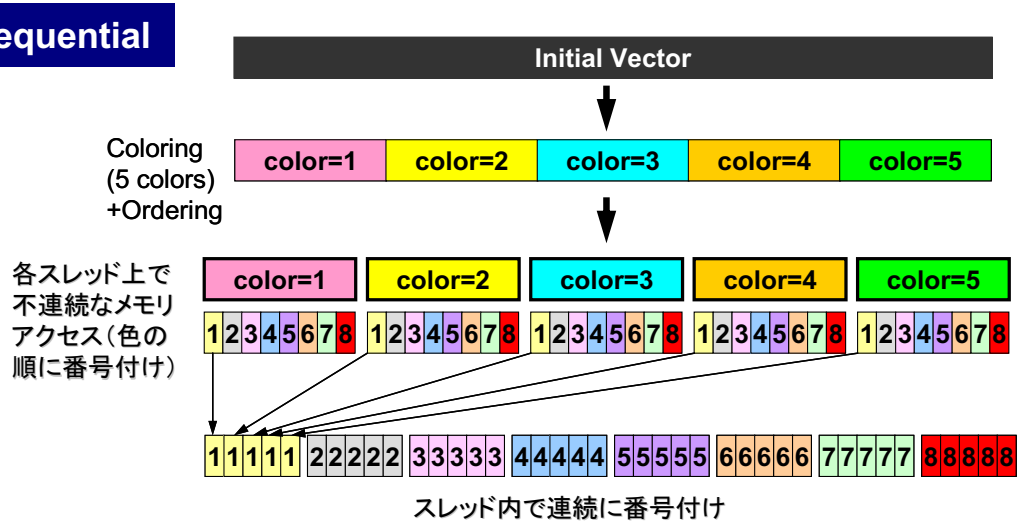
```
do L= 1, ITR   ! 収束判定ループ
!$omp parallel
  do ic = 1, NCOLORtot   !色ループ
!$omp do
        do ...
        end do
!$omp end do
  end do
!$omp end parallel
enddo
```

4 mvparallel

```
!$omp parallel
do L= 1, ITR   ! 収束判定ループ

do ic = 1, NCOLORtot   !色ループ
!$omp do
        do ...
        end do
!$omp end do
  end do

enddo
!$omp end parallel
```

ループスケジューリング

- 5 rmompdo
  - 自力で機械的にstatic分割
- 7 loopscheduling
  - リオーダリングの知識を使い、前進後退代入などと同一領域を担当



Baseline

```
!$omp do
    do i= 1, N
      W(i) = ....
    enddo
!$omp end do
!この後、W(i)を前進後退代入の入力
!として使う
```

5 rmompdo

```
    ip  = omp_get_thread_num()+1
    nth = omp_get_num_threads()
    ls  = (N+nth-1)/nth
    do i= (ip-1)*ls+1, min(ip*ls,N)
      W(i) = ....
    enddo
!$omp barrier
```

7 loopscheduling

```
do i = 自スレッドの担当部分
   W(i) = ....
enddo
! バリアはいらない
```

自力リダクション

```
    C1= 0.d0
!$omp do reduction(+:C1)
    do i= 1, N
      C1= C1 + W(i,P)*W(i,Q)
    enddo
!$omp end do
    ALPHA= RHO / C1
```

- オリジナルは**バリア2回**
  - ローカルリダクションとグローバルリダクションの間
  - !$omp end do の時
- nowaitは不可

```
!$omp parallel private(i, ALPHA, C1, ....)
    ip  = omp_get_thread_num()+1
    nth = omp_get_num_threads()
    ls  = (N+nth-1)/nth
    ...
    C1S(ip)= 0.0d0
    do i= (ip-1)*ls+1, min(ip*ls,N)
      C1S(ip)= C1S(ip) + W(i,P)*W(i,Q)
    enddo
    C1= 0.d0
!$omp barrier
    do i = 1, PEsmpTOT
      C1= C1 + C1S(i)
    end do
    ALPHA= RHO / C1
```

- **バリア1回**
- **各自グローバルリダクション**

最適化(OpenMP)

1. Baseline
2. mvparallel1
3. nowait
4. mvparallel2
5. rmompdo
6. rmreduction
7. loopschedule

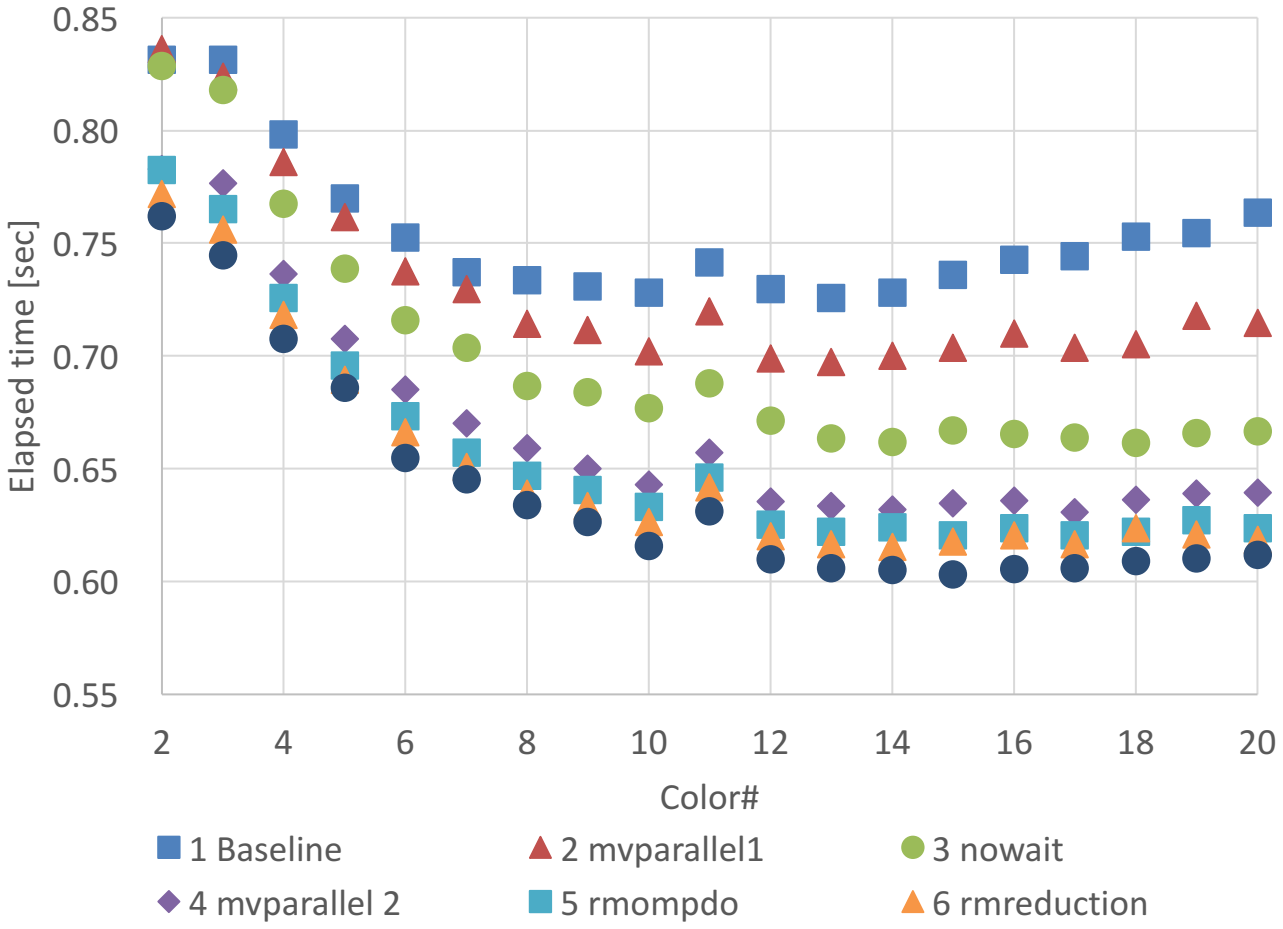
1イテレーションあたりのOMP指示文数 (12色の時)

	1	2	3	4	5	6	7
parallel do	40	5	5	0	0	0	0
parallel	0	3	3	0	0	0	0
do	0	35	23	28	3	0	0
do (nowait)	0	0	12	12	0	0	0
reduction clause	3	3	3	3	3	0	0
barrier (explicit)	**0**	**0**	**0**	**1**	**26**	**29**	**26**
barrier (implicit)	**43**	**46**	**34**	**31**	**6**	**0**	**0**
barrier (合計)	**43**	**46**	**34**	**32**	**32**	**29**	**26**

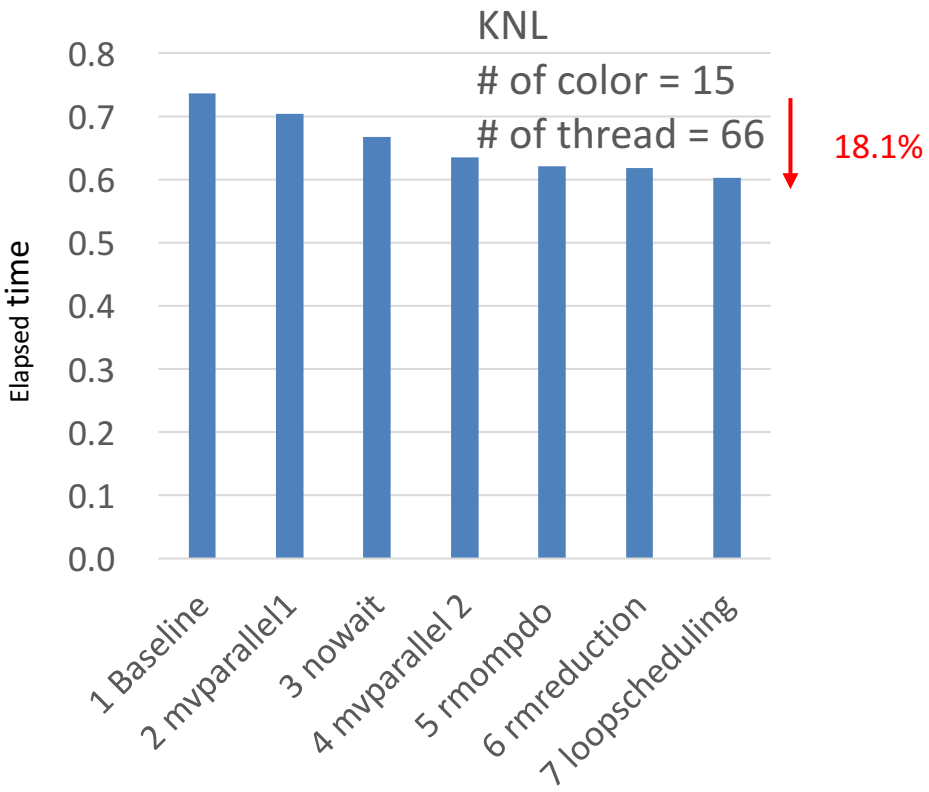
最終的に、1回の!$omp parallel と!$omp barrierだけのプログラムになる

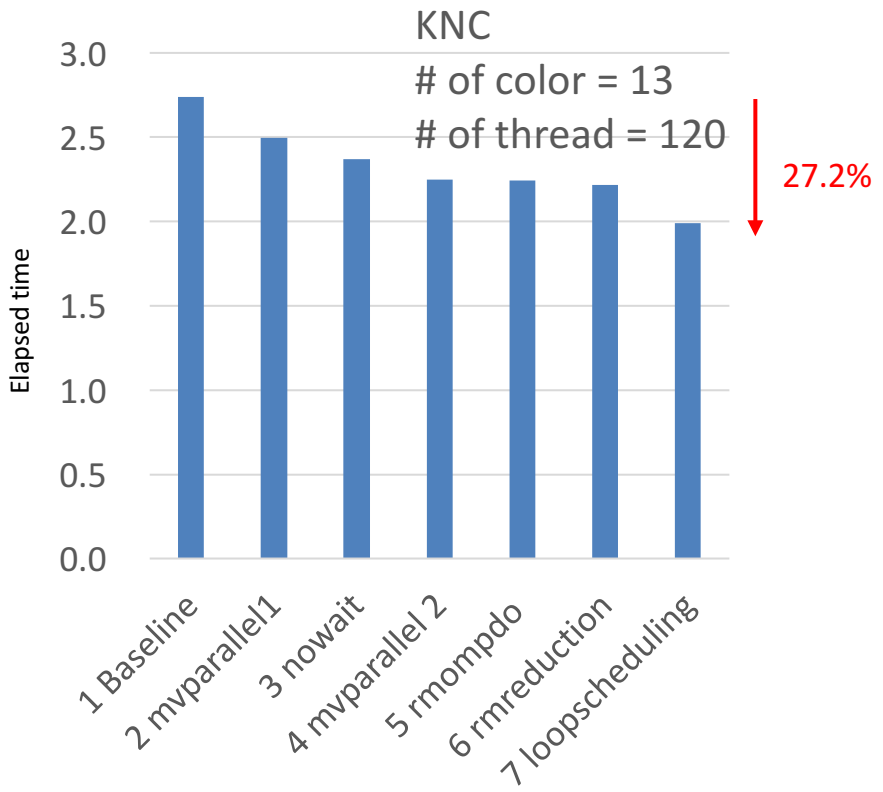
KNL最適化結果

- !$omp parallel の移動、バリアフリー どちらも効果大
  - KNLのバリアは遅い？
- Baselineから1.20倍の性能向上
- BDW比実行効率 61.0%→70.0%



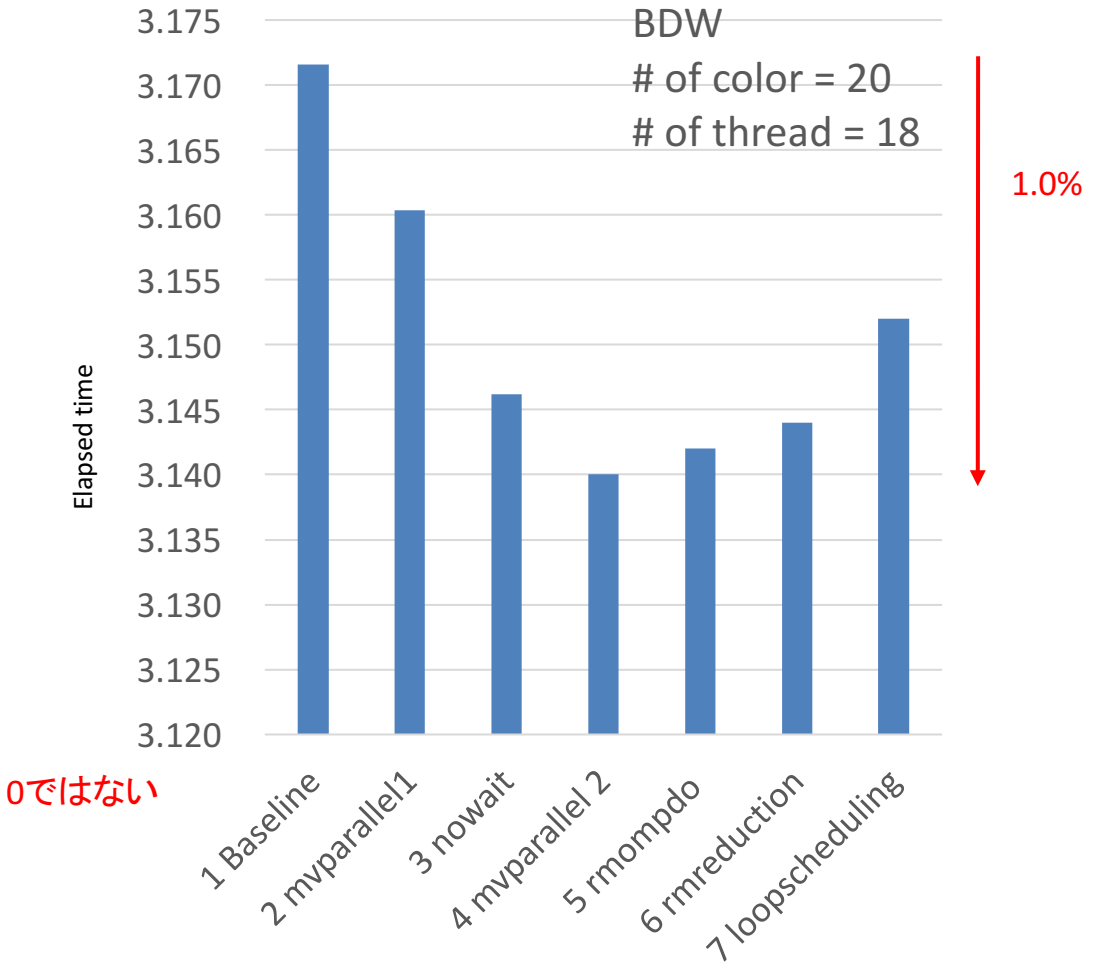
バリアフリーの効果(KNL, KNC)





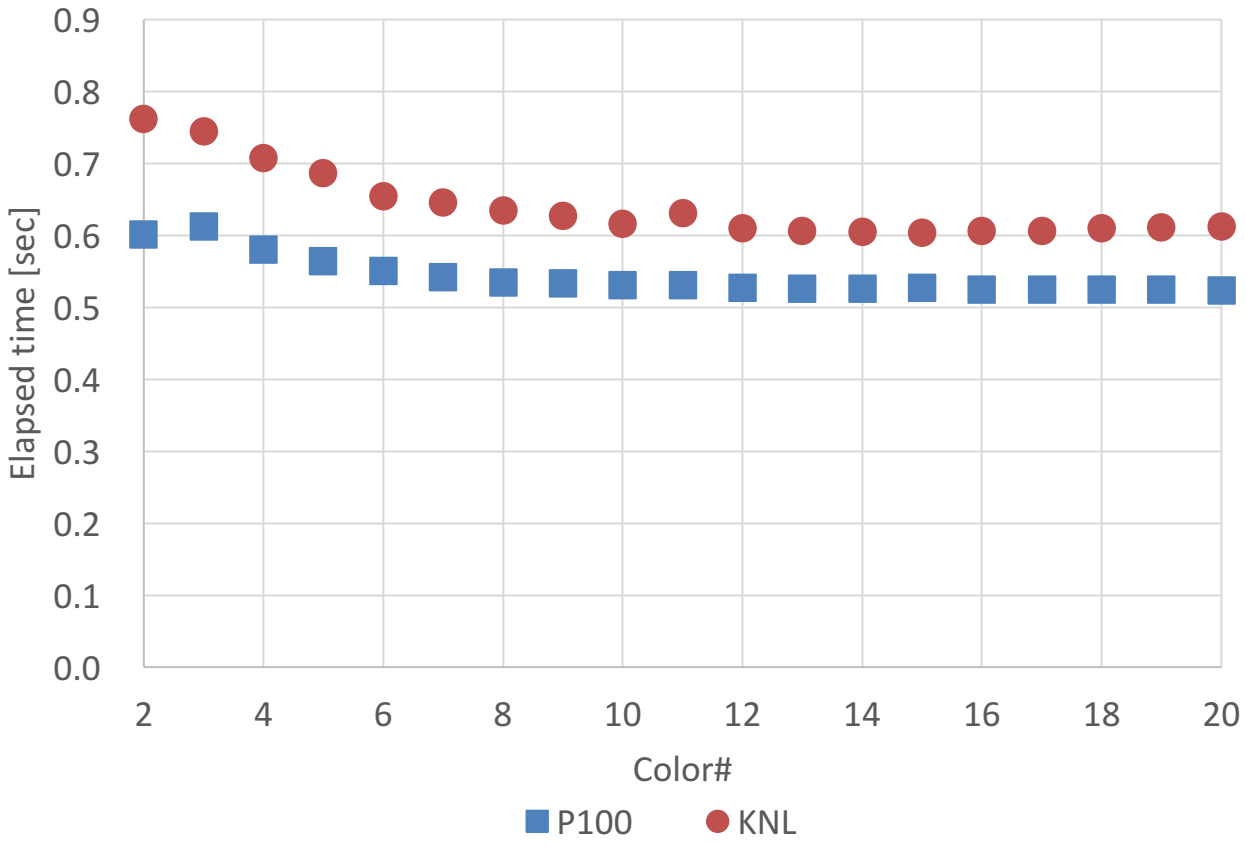
バリアフリーの効果(BDW)

- BDWではほとんど効果が得られない
  - 最外に!$omp parallelを追いやることで1.0%性能向上
  - “7”では”4”から0.3%性能低下



P100 vs KNL

- バンド幅比を考えると同程度
  - P100: 534 GB/s
  - KNL: 490 GB/s



まとめ

- ICCGソルバーにより、P100・KNLを評価
- P100
  - OpenACCの async 節を適切につけることが必須
    - CUDAはdefaultで非同期
  - 指示文の最適化などにより1.21倍の高速化
- KNL
  - バリアフリー化が効果大
  - 一連の最適化により1.20倍の高速化
- 計算速度の向上、コアの増加による相対的な同期コスト増により、バリアフリー化が重要
  - ただし、既存のCPUで性能低下を引き起こす可能性あり
- 今後の課題
  - MPIなどの通信を含むアプリケーションでの評価
  - 演算律速なアプリケーションでの評価
  - 同期コストについての詳細な評価

Q & A

実習

※今回の実習の例は、全てPGIコンパイラ17.1を使った際の例です

実習概要

- OpenACC プログラムのコンパイル
  - PGIコンパイラのメッセージの読み方
- OpenACC プログラムの作成
  - 行列積、diffusion
- OpenACC プログラムの最適化
  - NVIDIA visual profiler の使い方など

OpenACCサンプル集

- Reedbush へログイン
  - $ ssh **-Y** reedbush.cc.u-tokyo.ac.jp –l txxxxx
- module のロード
  - $ module load pgi/17.1
  - $ module load cuda/8.0.44
- ワークディレクトリに移動
  - $ cdw
- OpenACC_samples のコピー
  - $ cp /lustre/gt00h/z30108/OpenACC_samples.tar.gz .
  - $ tar zxvf OpenACC_samples.tar.gz
- OpenACC_samples へ移動
  - $ cd OpenACC_samples
  - $ ls

    C/　F/　　#CとFortran好きな方を選択

PGIコンパイラによるメッセージの確認

- コンパイラメッセージの確認はOpenACCでは極めて重要
  - OpenMP と違い、
    - 保守的に並列化するため、本来並列化できるプログラムも並列化されないことがある
    - 並列化すべきループが複数あるため、どのループにどの粒度(gang, worker, vector)が割り付けられたかしるため
    - ターゲットデバイスの性質上、立ち上げるべきスレッド数が自明に決まらず、スレッドがいくつ立ち上がったか知るため
  - メッセージを見て、プログラムを適宜修正する
- コンパイラメッセージ出力方法
  - コンパイラオプションに -Minfo=accel をつける

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード

```
8.    subroutine acc_kernels()
9.      double precision :: A(N,N), B(N,N)
10.     double precision :: alpha = 1.0
11.     integer :: i, j
12.     A(:,:) = 1.0
13.     B(:,:) = 0.0
14.     !$acc kernels
15.     do j = 1, N
16.       do i = 1, N
17.         B(i,j) = alpha * A(i,j)
18.       end do
19.     end do
20.     !$acc end kernels
21.   end subroutine acc_kernels
```

コンパイラメッセージ(fortran)

```
pgfortran -O3 -acc -Minfo=accel -ta=tesla,cc60 -Mpreprocess  acc_compute.f90
-o acc_compute
acc_kernels:
     14, Generating implicit copyin(a(:,:))
         Generating implicit copyout(b(:,:))
     15, Loop is parallelizable
     16, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         16, !$acc loop gang, vector(32) ! blockidx%x threadidx%x


    ....
```

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

```
8.    subroutine acc_kernels()
9.      double precision :: A(N,N), B(N,N)
10.     double precision :: alpha = 1.0
11.     integer :: i, j
12.     A(:,:) = 1.0
13.     B(:,:) = 0.0
14.     !$acc kernels
15.     do j = 1, N
16.       do i = 1, N
17.         B(i,j) = alpha * A(i,j)
18.       end do
19.     end do
20.     !$acc end kernels
21.   end subroutine acc_kernels
```

ソースコード

サブルーチン名

コンパイラメッセージ(fortran)

```
pgfortran -O3 -acc -Minfo=accel -ta=tesla,cc60 -Mpreprocess  acc_compute.f90
-o acc_compute
acc_kernels:
     14, Generating implicit copyin(a(:,:))
         Generating implicit copyout(b(:,:))
     15, Loop is parallelizable
     16, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         16, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
     ....
```

配列aはcopyin, bはcopyoutとして扱われます

15, 16行目の2重ループは(32x4)のスレッドでブロック分割して扱います。

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード(C)

```
40.  void acc_kernels(double *A, double *B){
41.   double alpha = 1.0;
42.   int i,j;
          / * A と B 初期化 */
50.  #pragma acc kernels
51.   for(j = 0;j < N;j++){
52.    for(i = 0;i < N;i++){
53.     B[i+j*N] = alpha * A[i+j*N];
54.    }
55.   }
56.  }
```

コンパイラメッセージ(C)

```
pgcc -O3 -acc -Minfo=accel -ta=tesla,cc60 -Mcuda acc_compute.c -o acc_compute
acc_kernels:
     50, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     51, Loop carried dependence of B-> prevents parallelization
         Loop carried backward dependence of B-> prevents vectorization
         Complex loop carried dependence of B->,A-> prevents parallelization
         Accelerator scalar kernel generated
         Accelerator kernel generated
         Generating Tesla code
         51, #pragma acc loop seq
         52, #pragma acc loop seq
     52, Complex loop carried dependence of B->,A-> prevents parallelization
```

配列aはcopyin,
bはcopyとして扱われます

ループ伝搬依存が見つかったので並列化しませんの意。ポインタAとBが同じ領域を指していることを警戒して、並列化しない。

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード(C)

```
59.  void acc_kernels(double *restrict A,
     double *restrict B){
60.   double alpha = 1.0;
61.   int i,j;
          / * A と B 初期化 */
69.  #pragma acc kernels
70.   for(j = 0;j < N;j++){
71.    for(i = 0;i < N;i++){
72.     B[i+j*N] = alpha * A[i+j*N];
73.    }
74.   }
75.  }
```

コンパイラメッセージ(C)

```
acc_kernels_restrict:
     69, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     70, Loop carried dependence of B-> prevents parallelization
         Loop carried backward dependence of B-> prevents vectorization
     71, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         70, #pragma acc loop seq
         71, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

2次元配列を1次元化して扱っているため、i+j*N が実は同じ場所を指す可能性を考慮し、70行目を並列化していない。

PGIコンパイラによるメッセージの確認

- OpenACC_samples を利用
- $ make acc_compute

ソースコード(C)

```
78.  void acc_kernels(double *restrict A,
     double *restrict B){
79.   double alpha = 1.0;
80.   int i,j;
       / * A と B 初期化 */
88.  #pragma acc kernels
89.  #pragma acc loop independent
90.   for(j = 0;j < N;j++){
91.  #pragma acc loop independent
92.    for(i = 0;i < N;i++){
93.     B[i+j*N] = alpha * A[i+j*N];
94.    }
95.   }
96.  }
```

コンパイラメッセージ(C)

```
acc_kernels_independent:
     88, Generating implicit copy(B[:1000000])
         Generating implicit copyin(A[:1000000])
     90, Loop is parallelizable
     92, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         90, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         92, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

ようやくまともに並列化

実習1
コンパイラのメッセージを確認しよう

- acc_compute.f90 or acc_compute.c のソースと、コンパイルメッセージを見比べてください
  - コンパイル：$ make acc_compute
- 注目点
  - parallel / kernels での違い
  - 間接参照があった際のコンパイラメッセージ
    - acc_kernels_BAD_indirect_reference と acc_kernels_indirect_reference の比較

PGI_ACC_TIME によるOpenACC 実行の確認

- PGI環境の場合、OpenACC プログラムが実行されているかを確認するには、環境変数PGI_ACC_TIME を使うのが簡単
- 使い方 (一般的なLinux環境、またはインタラクティブジョブ実行時)
  - $ export PGI_ACC_TIME=1
  - $ (プログラムの実行)
- 一般的なスパコン環境では、ジョブの中で環境変数を設定する必要が有る
  - ジョブスクリプト中に書いてある

PGI_ACC_TIME による OpenACC 実行の確認

- OpenACC_samples を利用
- $ qsub acc_compute.sh
  - 実行が終わると以下ができる
    - acc_compute.sh.eXXXXX (標準エラー出力)
    - acc_compute.sh.oXXXXX (標準出力)
- $ less acc_compute.sh.eXXXXX

```
40.  void acc_kernels(double *A, double *B){
41.   double alpha = 1.0;
42.   int i,j;
          / * A と B 初期化 */
50.  #pragma acc kernels
51.   for(j = 0;j < N;j++){
52.    for(i = 0;i < N;i++){
53.     B[i+j*N] = alpha * A[i+j*N];
54.    }
55.   }
56.  }
```

PGI_ACC_TIME による出力メッセージ

```
Accelerator Kernel Timing data
/lustre/pz0108/z30108/OpenACC_samples/C/acc_compute.c
  acc_kernels  NVIDIA  devicenum=0
    time(us): 149,101
    50: compute region reached 1 time
        51: kernel launched 1 time
            grid: [1]  block: [1]
             device time(us): total=140,552 max=140,552 min=140,552 avg=140,552
            elapsed time(us): total=140,611 max=140,611 min=140,611 avg=140,611
    50: data region reached 2 times
        50: data copyin transfers: 2
             device time(us): total=3,742 max=3,052 min=690 avg=1,871
        56: data copyout transfers: 1
             device time(us): total=4,807 max=4,807 min=4,807 avg=4,807
```

← 起動したスレッド数

↓カーネル実行時間

← データ移動の回数・時間

実習2
OpenACCプログラムを実行してみよう

- acc_data.f90 or acc_data.c のソースと、コンパイルメッセージを見比べてください
  - コンパイル：$ make acc_data
- プログラムを実行し、PGI_ACC_TIMEの出力を確認してください
  - 実行
    - $ qsub acc_data.sh
  - バッチジョブ実行が終了すると acc_data.sh.oXXXXXX (標準出力), acc_data.sh.eXXXXXX (標準エラー出力) の2ファイルが出来る
  - PGI_ACC_TIME の出力確認
    - $ less acc_data.sh.eXXXXXX　#標準エラーの方
- 注目点
  - acc_data_copy, acc_data_copyinout の実行時間の違い

実習3
OpenACCプログラムを作ろう

- 行列積のOpenACC化
- matmul.f90 または matmul.c を用いる。
  - acc_matmul ルーチンにOpenACC指示文を加えてください。
  - 注意：コンパイラの出力をよく見てください
    - よく見るエラー例(C)：
      - Accelerator restriction: size of the GPU copy of C,B is unknown

出力例

```
====== OpenACC matmul program ======
1024 * 1024 matrix
check result...OK
elapsed time[sec] :    0.00708
FLOPS[GFlops]     :  302.85417
```

実習4
OpenACCプログラムを速くしよう

- 拡散方程式のプログラムの高速化
- diffusion.f90 または diffusion.c を用いる。
  - 既にOpenACC化されていますが、実行してみると…

出力例(C)

```
(nx, ny, nz) = (128, 128, 128)
elapsed time : 574.247 (s)
flops        : 0.078 (GFlops)
throughput   : 0.096 (GB/s)
accuracy     : 4.443942e-06
count        : 1638
```

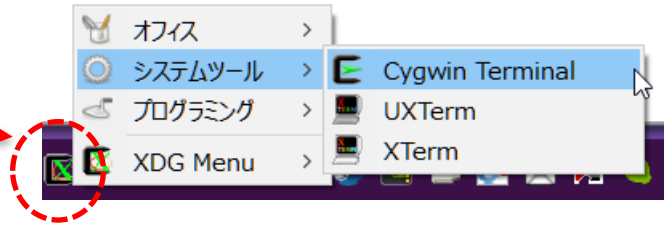
すごく遅い(特にC)

- 確認手順
  1. きちんと並列化されているか？ → コンパイラメッセージ
  2. 無駄にデータ転送してないか？ → PGI_ACC_TIME
  3. どっちもOKだけどなお遅い？ → NVIDIA Visual Profiler (nvvp)
     - https://reedbush-www.cc.u-tokyo.ac.jp/session_login.cgi に詳細資料あり

4. プロファイラ（nvvp）を使う

NVIDIA CUDA Visual Profiler

- （Windowsユーザのみ）準備
  - Cygwinインストール時にxorg-serverとxinitをインストールしておく
  - 「XWin Server」を起動する
    - タスクトレイにアイコンが2つ増える
  - 緑の線の入った方のアイコンから「Cygwin Terminal」を起動する



- 使用
  - Reedbushへssh接続する際に-Yオプションを付けておく
    - ssh -Y reedbush.cc.u-tokyo.ac.jp -l txxxxx
  - module設定を行い、nvvpを起動する
    - module load cudaを設定したあとでnvvpコマンドを実行

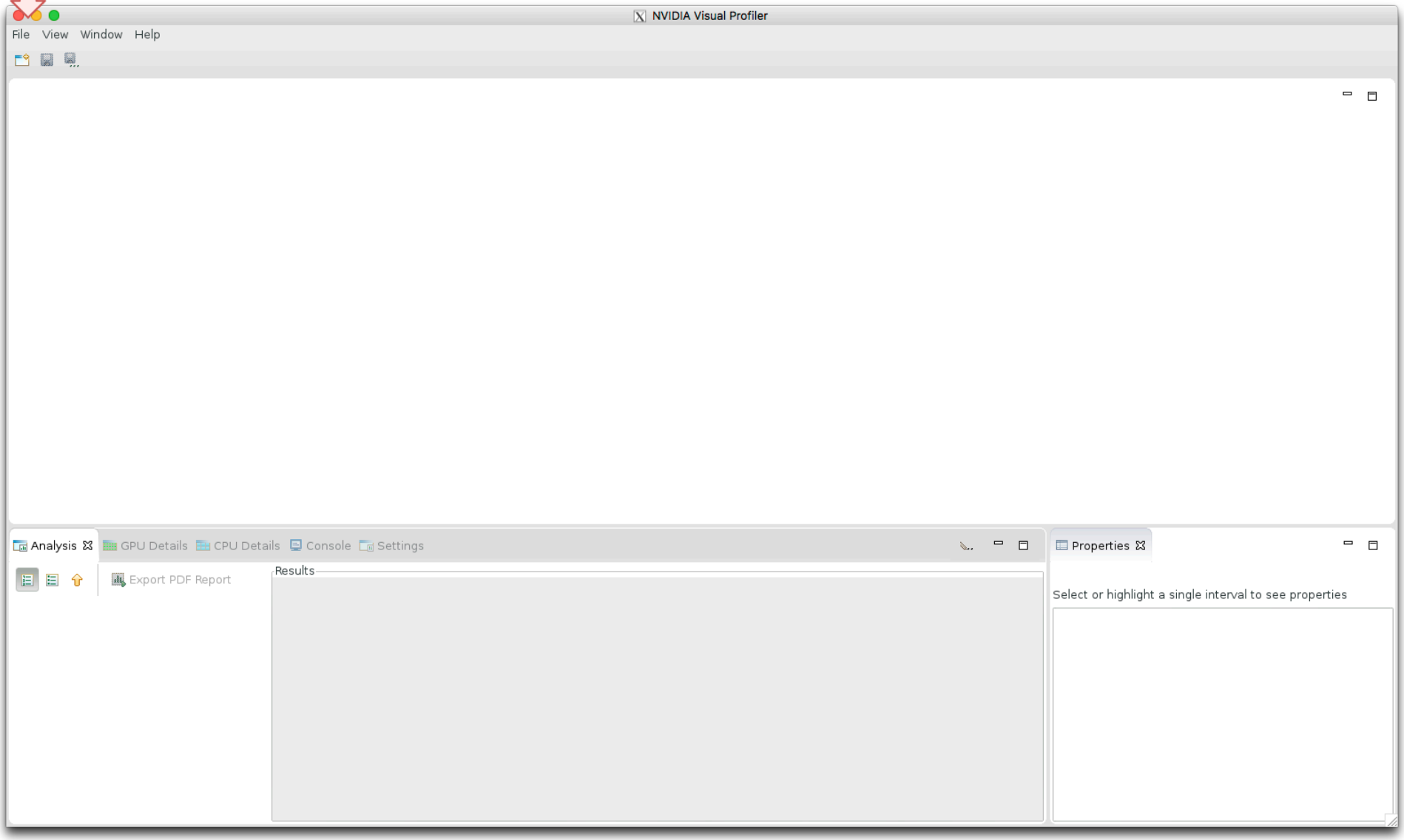
Reedbush での NVVP の使い方

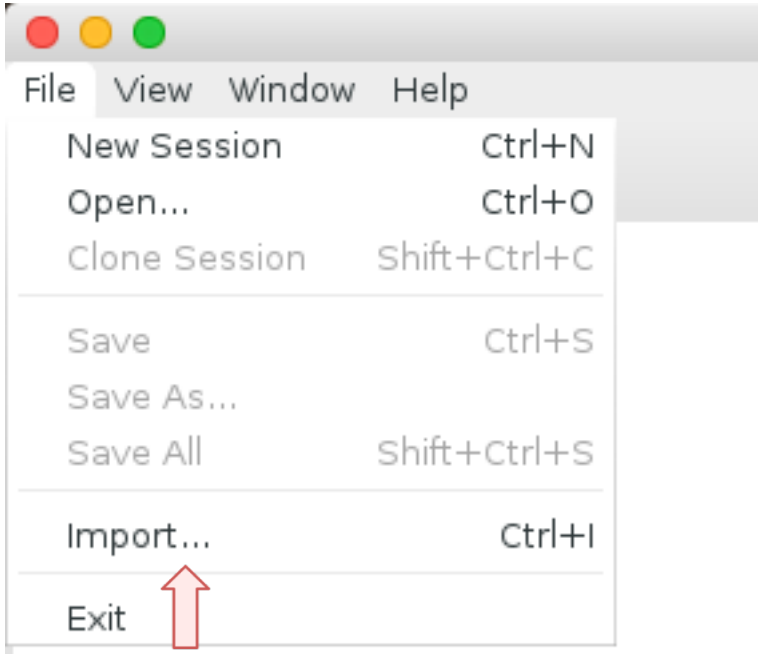
1. ~~インタラクティブノードを使う~~
   - ~~nvvpを起動し、nvvpの中からアプリケーションを起動~~
   - 講習会アカウントでは使えません
2. 計算ノードでデータ収集、ログインノードでnvvp起動
   - スパコン環境で一般的なやり方。今回はこれ
   - ネットワーク越しに画面転送するので遅い
3. 計算ノードでデータ収集、自分の端末にnvvpインストールして起動
   - 速い。よく使う人にオススメ
   - scp で収集データを自分の端末に持って来て起動
   - https://developer.nvidia.com/nvidia-visual-profiler
     - ただし1GB以上あるので、今やるとおそらくネットワークがパンクします…

Reedbush での nvvp の使い方

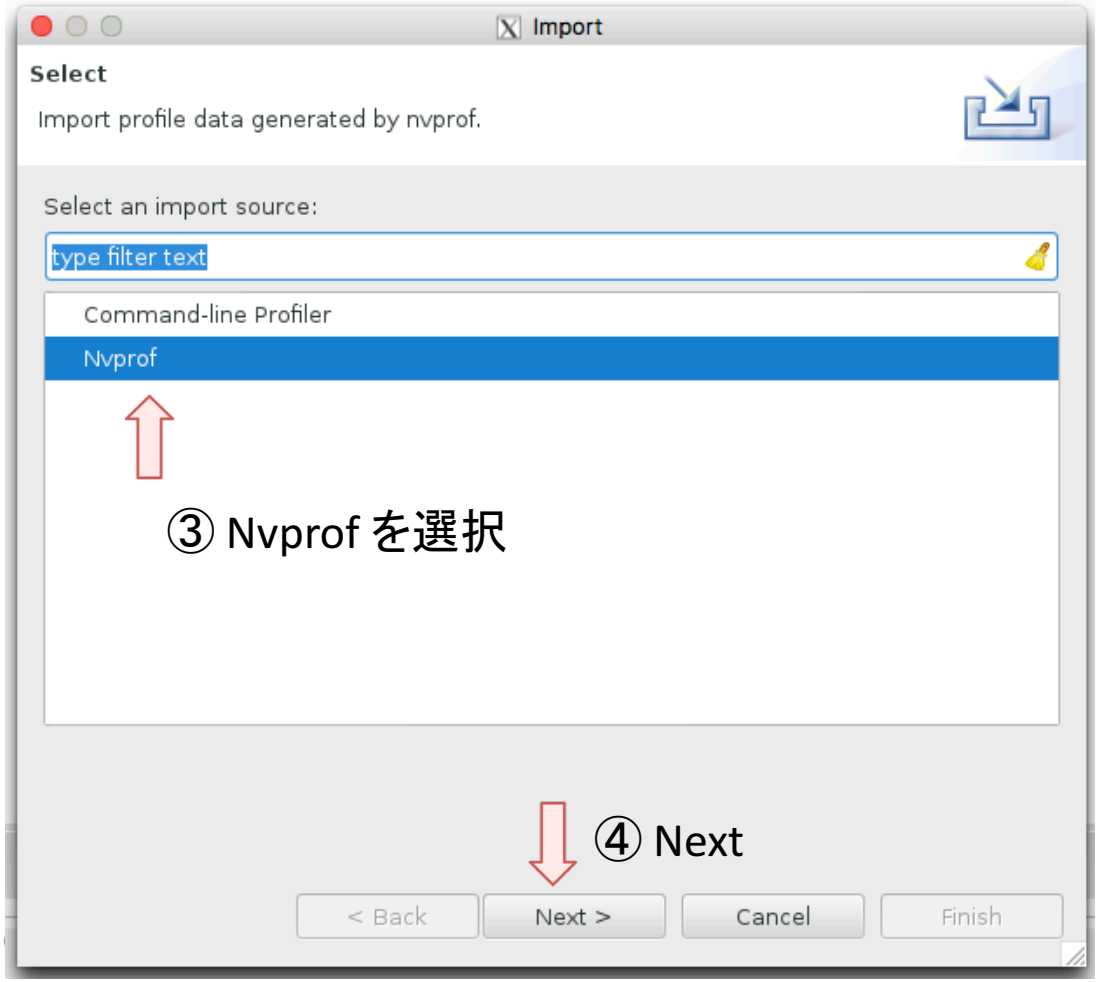
- ログインして module をロード
  - $ ssh -Y reedbush.cc.u-tokyo.ac.jp -l txxxxx
  - $ module load cuda pgi/17.1
  - $ cd OpenACC_samples/C または F
- nvprof コマンドを使ってデータを収集
  - ジョブスクリプトの中に書く
    - nvvp.sh 参照
  - 今回は予め取っておいたデータを使う
    - diffusion.nvp (diffusionを10ステップだけ実行したもの)
- nvvp の起動
  - $ nvvp

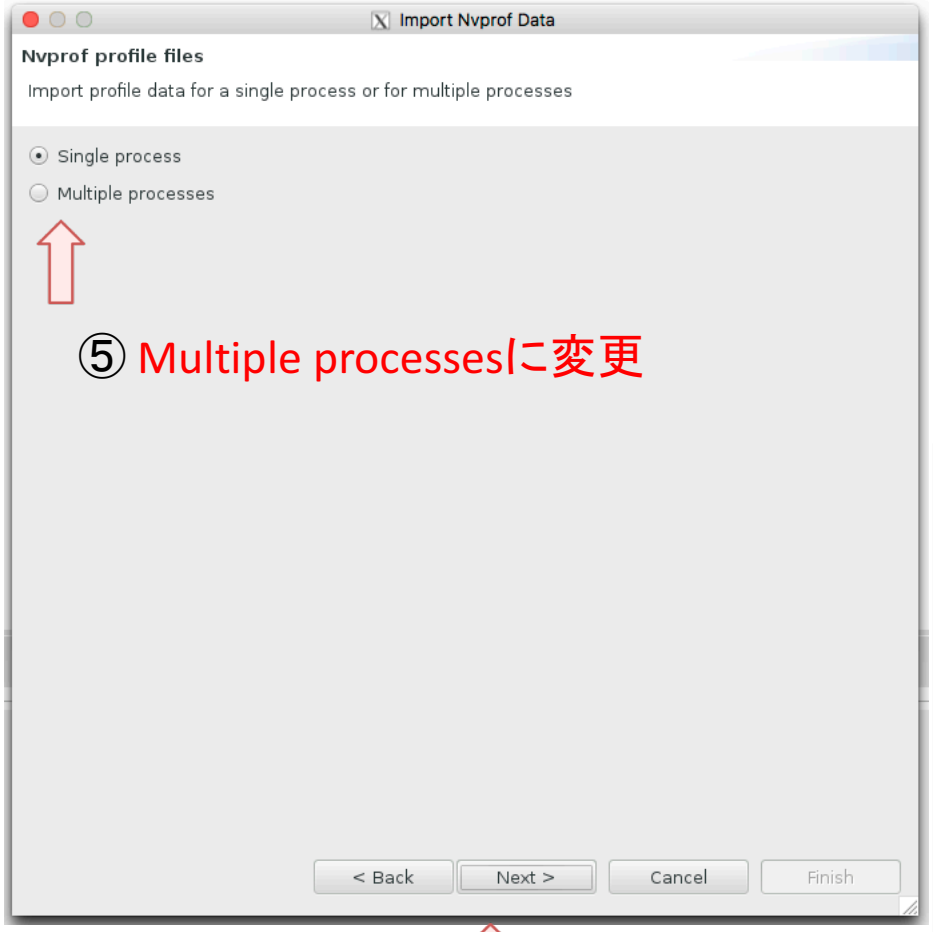
①Fileをクリック


NVIDIA Visual Profiler
File View Window Help
Analysis
GPU Details
CPU Details
Console
Settings
Export PDF Report
Results
Properties
Select or highlight a single interval to see properties



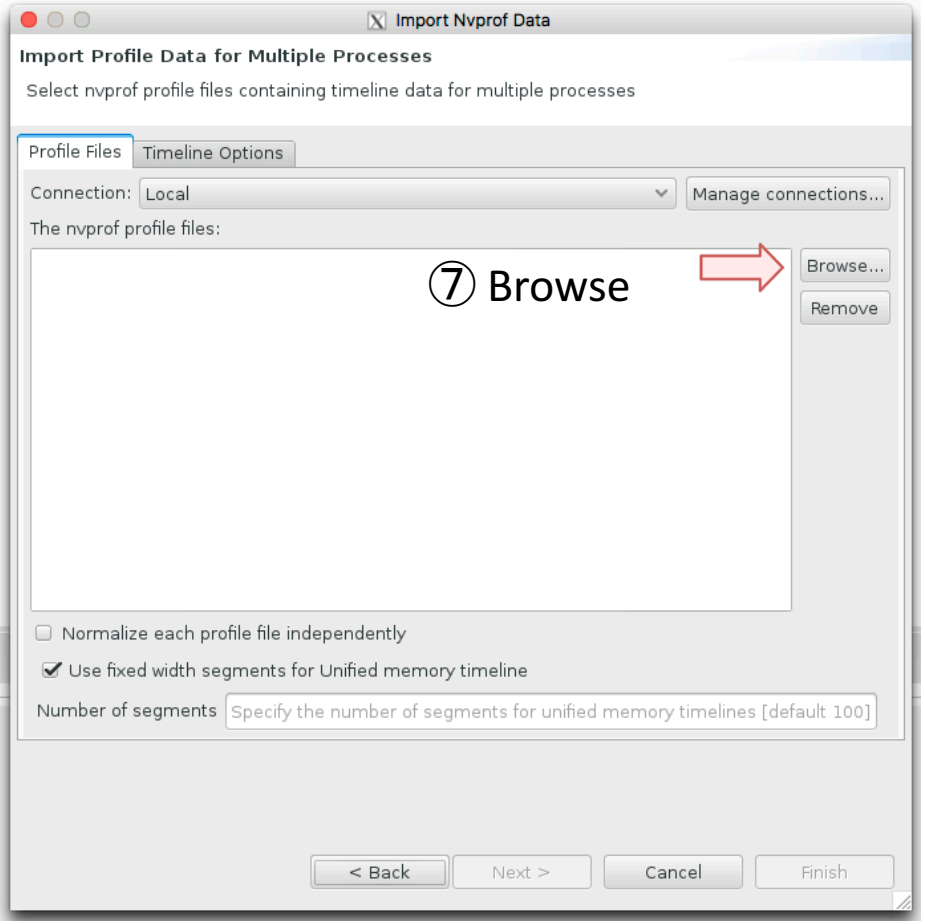
②Importをクリック



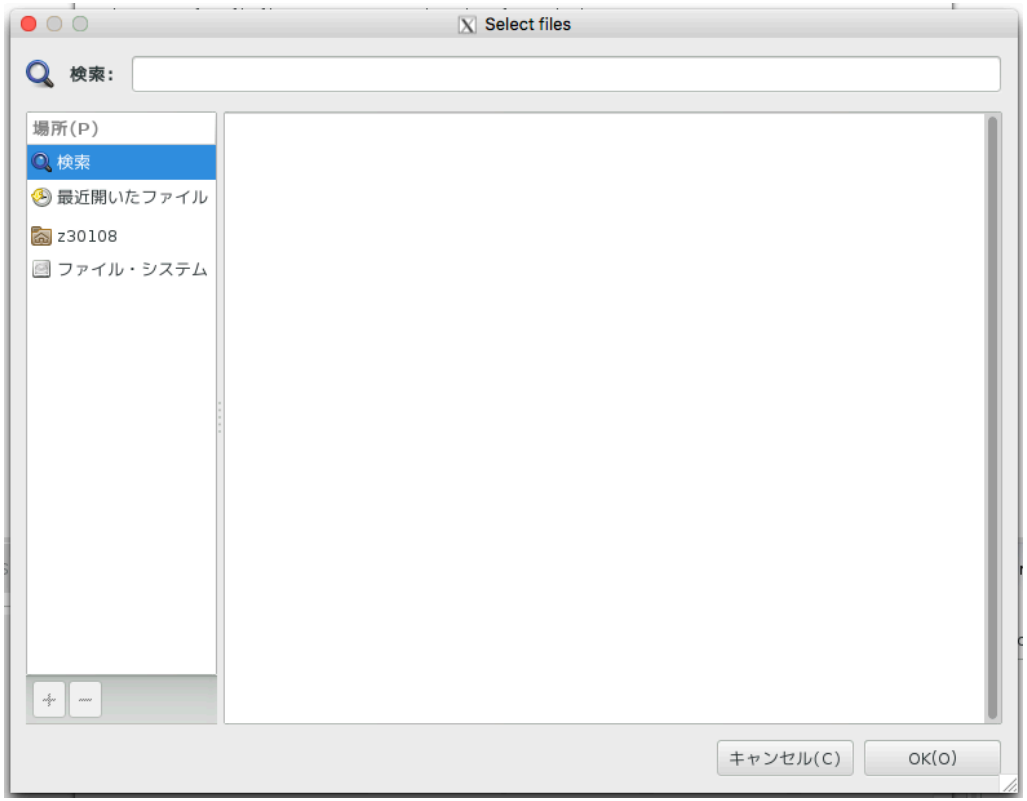


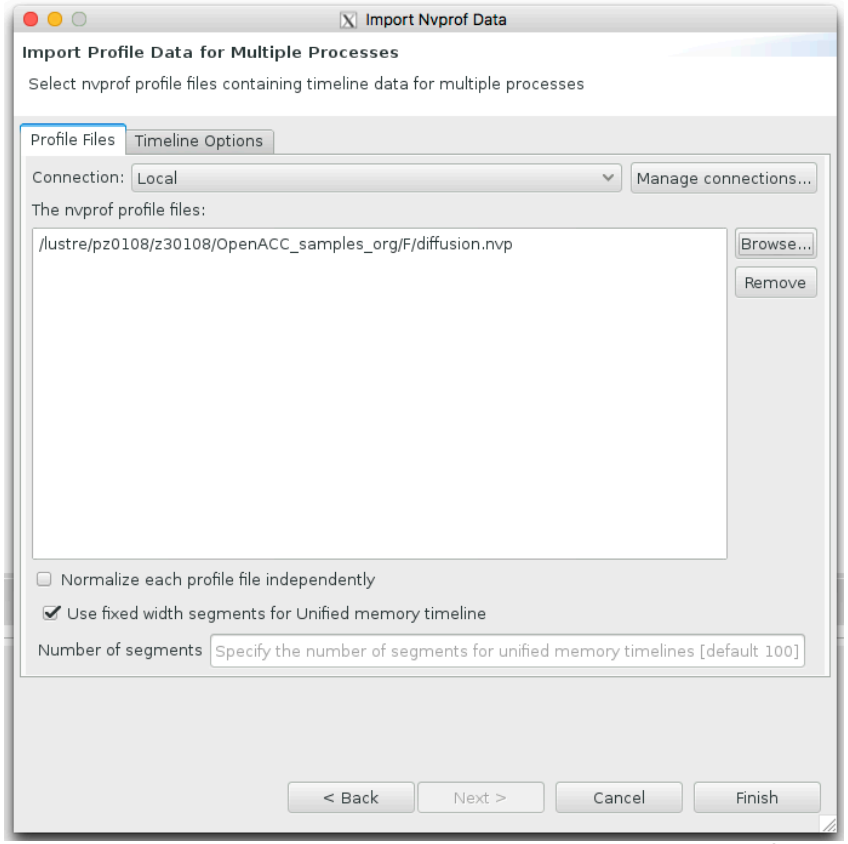
⑤ Multiple processesに変更

⑥ Next



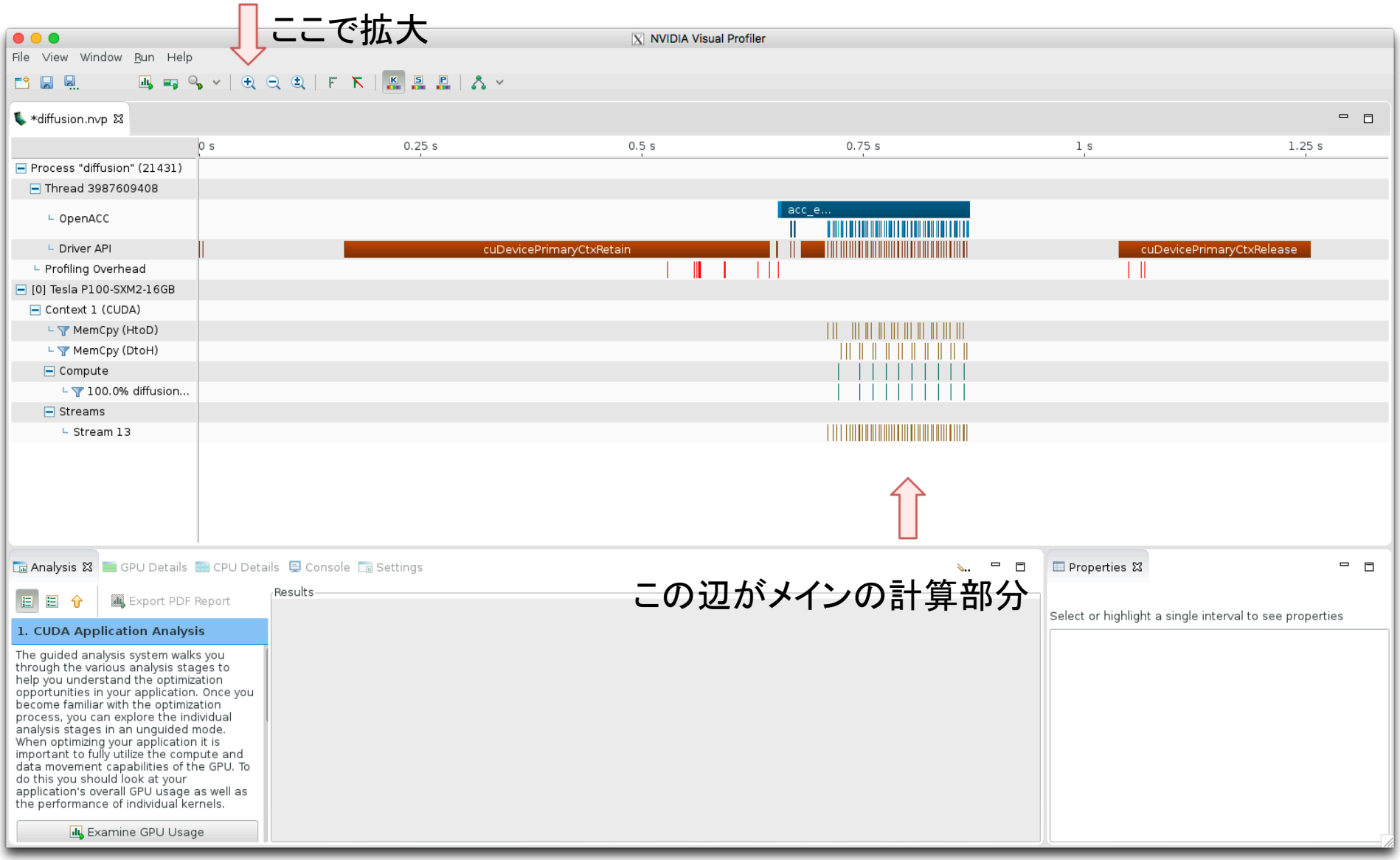
⑦ Browse

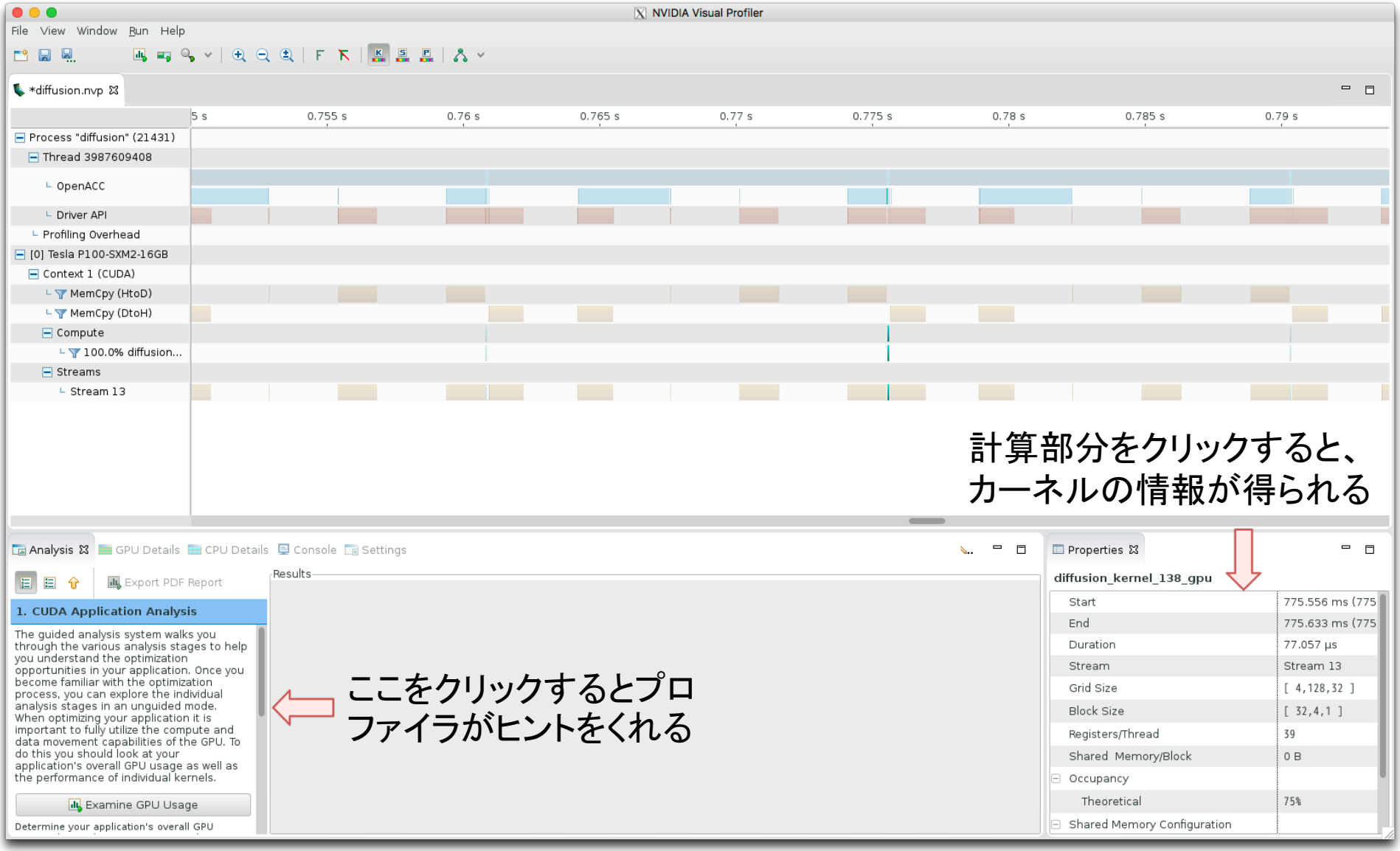




⑧ ファイル選択画面が開くので、nvvpで作成したプロファイリングデータを選択し、OK
ファイルシステム→上の「場所」欄に
/lustre/gt00h/ユーザー名/OpenACC_samples/…
とするのが速いか…？

⑨ diffusion.nvpを選択できたら Finish


ここで拡大
NVIDIA Visual Profiler
File View Window Run Help
*diffusion.nvp
0 s
0.25 s
0.5 s
0.75 s
1 s
1.25 s
Process "diffusion" (21431)
Thread 3987609408
OpenACC
acc_e...
Driver API
cuDevicePrimaryCtxRetain
cuDevicePrimaryCtxRelease
Profiling Overhead
[0] Tesla P100-SXM2-16GB
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
100.0% diffusion...
Streams
Stream 13
この辺がメインの計算部分
Analysis
GPU Details
CPU Details
Console
Settings
Export PDF Report
Results
1. CUDA Application Analysis
The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.
Examine GPU Usage
Properties
Select or highlight a single interval to see properties


NVIDIA Visual Profiler
File View Window Run Help
*diffusion.nvp
5 s
0.755 s
0.76 s
0.765 s
0.77 s
0.775 s
0.78 s
0.785 s
0.79 s
Process "diffusion" (21431)
Thread 3987609408
OpenACC
Driver API
Profiling Overhead
[0] Tesla P100-SXM2-16GB
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
100.0% diffusion...
Streams
Stream 13
計算部分をクリックすると、
カーネルの情報が得られる
Analysis
GPU Details
CPU Details
Console
Settings
Export PDF Report
1. CUDA Application Analysis
The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.
Examine GPU Usage
Determine your application's overall GPU
Results
ここをクリックするとプロ
ファイラがヒントをくれる
Properties
diffusion_kernel_138_gpu
Start 775.556 ms (775
End 775.633 ms (775
Duration 77.057 µs
Stream Stream 13
Grid Size [4,128,32]
Block Size [32,4,1]
Registers/Thread 39
Shared Memory/Block 0 B
Occupancy
Theoretical 75%
Shared Memory Configuration

Q & A

- アカウントは1週間有効です。
- 資料のPDF版はWEBページにございます。
  - http://www.cc.u-tokyo.ac.jp/support/kosyu/75/
- アンケートへの協力をお願いします。