第97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 hoshino@cc.u-tokyo.ac.jp

2018年4月18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、スパコンの使い方
- 13:30 - 14:20　GPUのアーキテクチャ
- 14:30 - 15:20　OpenACC入門
- 15:30 - 16:20　OpenACC演習
- 16:30 - 18:00　FDTD法による電磁波伝搬計算

スケジュール（改）

- 09:30 - 10:00　受付
- 10:00 - 12:00　Reedbush-Hログイン、スパコンの使い方
- 13:30 - 14:30　GPU・OpenACC入門　（座学）
- 14:45 - 18:00　OpenACC演習　（適宜休憩）
  - 簡単なプログラムのOpenACC並列化
  - 3次元拡散方程式
  - FDTD法による電磁波伝搬計算
  - 質問など

本講習会について

- 本講習会の趣旨
  - ✓ 主に自分のアプリケーションのGPU上での実行を考えている人を対象とした、GPUプログラミングの入門
  - ✓ 入門に相応しいOpenACCを用いたプログラミング技能の習得

- その他の講習会

  https://www.cc.u-tokyo.ac.jp/support/kosyu/
  - ✓ 5/31 にはディープラーニングに特化した講習会が開催されます。
  - ✓ 後期には複数GPUの利用に関する講習会も開催予定です。(時期未定)

- スパコンイベント情報メール配信サービス

  https://regist.cc.u-tokyo.ac.jp/announce/
  - ✓ 講習会や研究会の案内、トライアルユースの実施のお知らせなどを配信しています。

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

Reedbush 利用上の注意（1）

- 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/gt00/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/gt00/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
  - cdw コマンドでLustreファイルシステムに移動できます。

- 実行中のジョブの確認
  - ジョブ投入はqsubで行うが、ジョブ確認は qstat ではなく rbstat

Reedbush 利用上の注意（2）

- コンパイルおよび実行のための環境準備
  - ✓ コンパイルおよび実行のための環境を準備するために module コマンドを使用する。これによって様々な環境を簡単に切り替えて使用できる。

$ module load <*module_name*>

モジュール名 **<*module_name*>** のモジュールをロードして環境を準備。環境変数PATHなどが設定される。

$ module avail

使用可能なモジュール一覧を表示する。

$ module list

使用中のモジュールを表示する。

モジュールの切り替え

- PGIコンパイラ（OpenACCやCUDA Fortran）を使う場合
  ```
  $ module load pgi
  ```
- CUDA開発環境を使う場合
  ```
  $ module load cuda
  ```
- Intelコンパイラを使う場合
  ```
  $ module load intel
  ```
- MPIを使う場合
  ```
  $ module load openmpi/gdr/2.1.1/{gnu,intel,pgi}
  $ module load mvapich2/gdr/2.2/{gnu,intel,pgi}
  ```
  ✓ PGIなどのコンパイラに追加して load

- モジュールはジョブ実行時にもコンパイル時と同じものを load する。
- 組み合わせて利用できる

サンプルコードのコンパイル

- Reedbush へのログイン
  ```
  $ ssh -Y txxxxx@reedbush.cc.u-tokyo.ac.jp
  ```
  ✓ txxxxx 各自の利用者番号（アカウント）に置き換えてください。
  ✓ -Y をつけてください。
- cdw コマンドで Lustreファイルシステムへ移動する。
  ```
  $ cdw
  ```
- 自分のディレクトリにサンプルコードをコピーする。
  ```
  $ cp /lustre/gt00/share/openacc_samples.tar.gz .
  ```
- サンプルコードを展開する。
  ```
  $ tar zxvf openacc_samples.tar.gz
  ```
- サンプルコードへ移動する。
  ```
  $ cd openacc_samples
  ```
- モジュールをロードする。
  ```
  $ module load pgi
  ```
- コンパイルする。
  ```
  $ cd openacc_hello/01_hello_acc
  $ make
  ```
- 実行ファイルがてきていることを確認する。
  ```
  $ ls
  ```

プログラムの実行

- ジョブとして投入し、実行する。

```
$ qsub ./run.sh
```

- 投入されたジョブを確認する。

```
$ rbstat
```

- 実行が終了すると、以下のファイルが生成される。

```
run.sh.o??????
run.sh.e??????   （?????? は数字）
```

- 上記の標準出力ファイルの中身を確認する。

```
$ cat run.sh.o??????
```

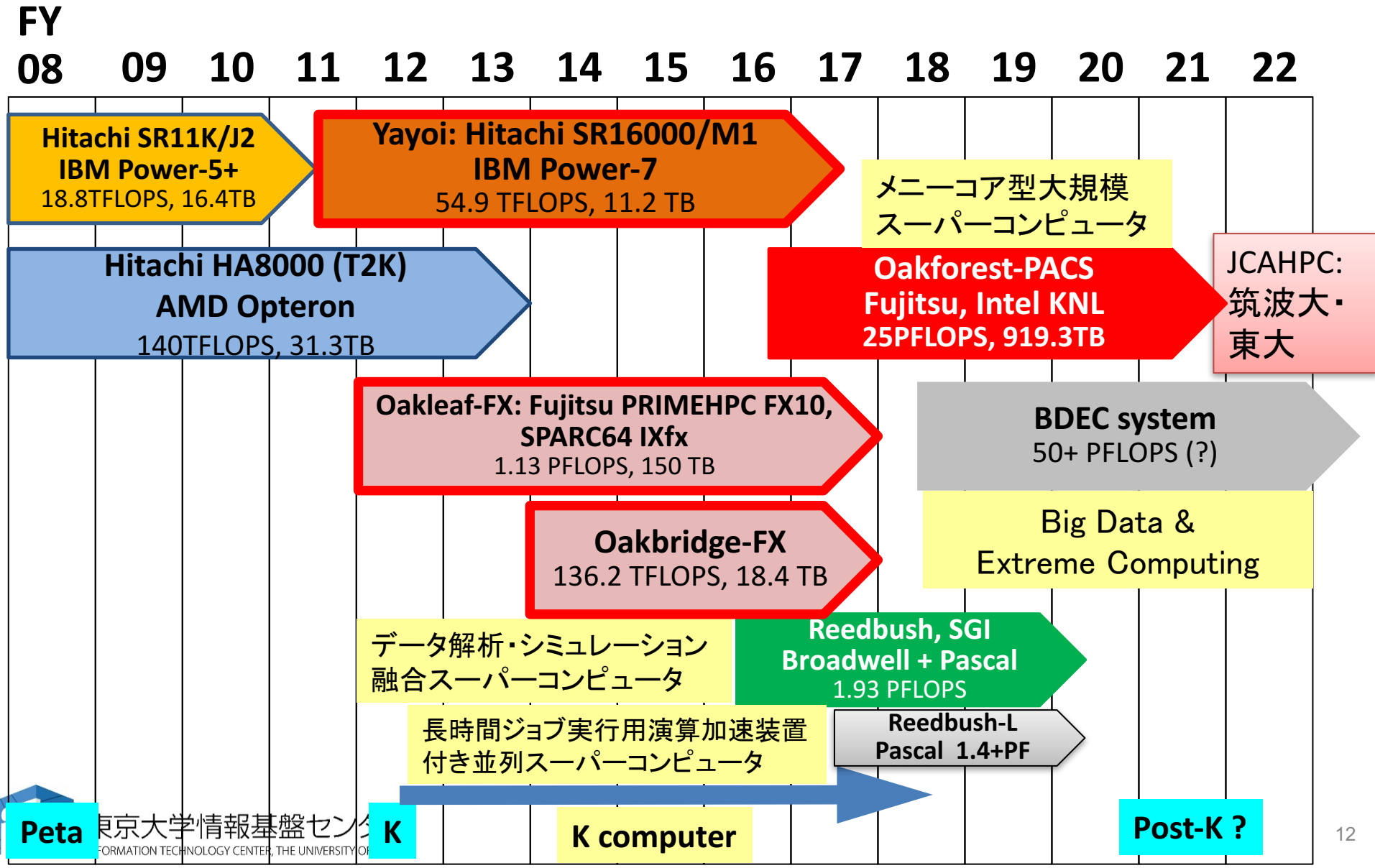
- 必要に応じて、上記のエラー出力ファイルの中身を確認する。

```
$ cat run.sh.e??????
```

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



2システム運用中

- 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9位 (2017年11月) (日本で2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（1/2）

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（1/2）

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

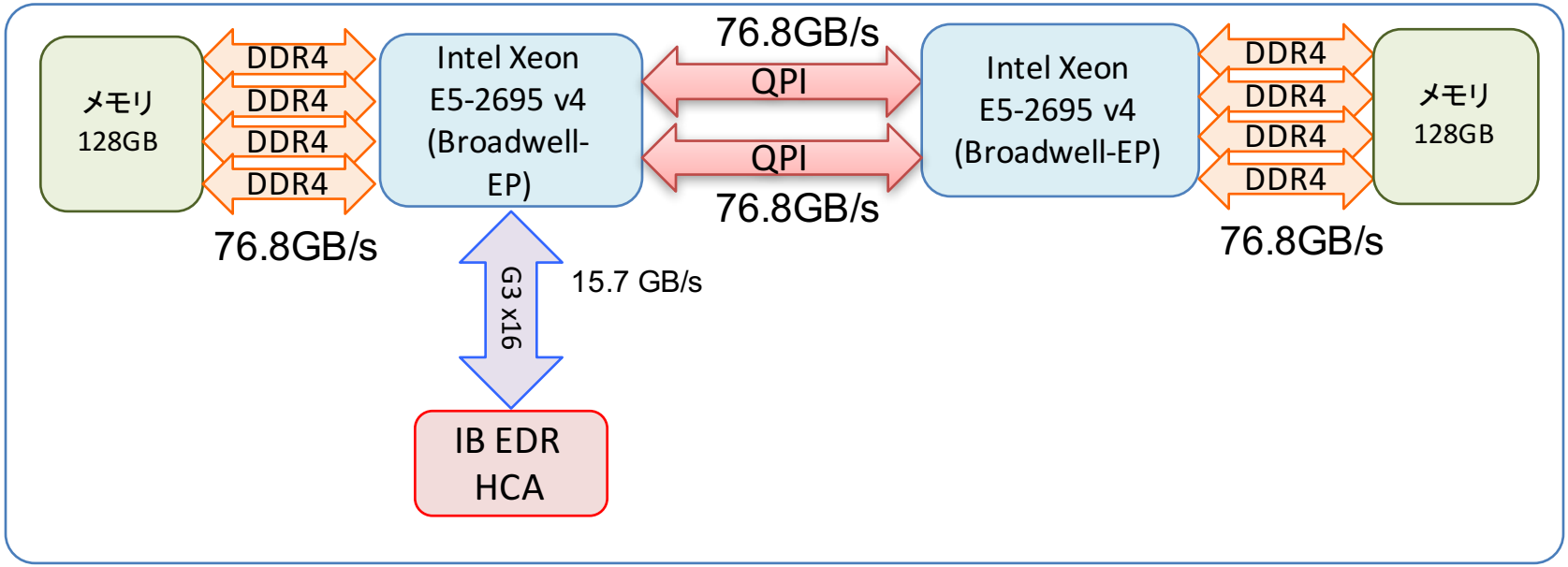
Intel Xeon Phi 7250 1.4 GHz 68 core x1 socket

2016年12月1日試験運転開始

2017年4月3日正式運用開始

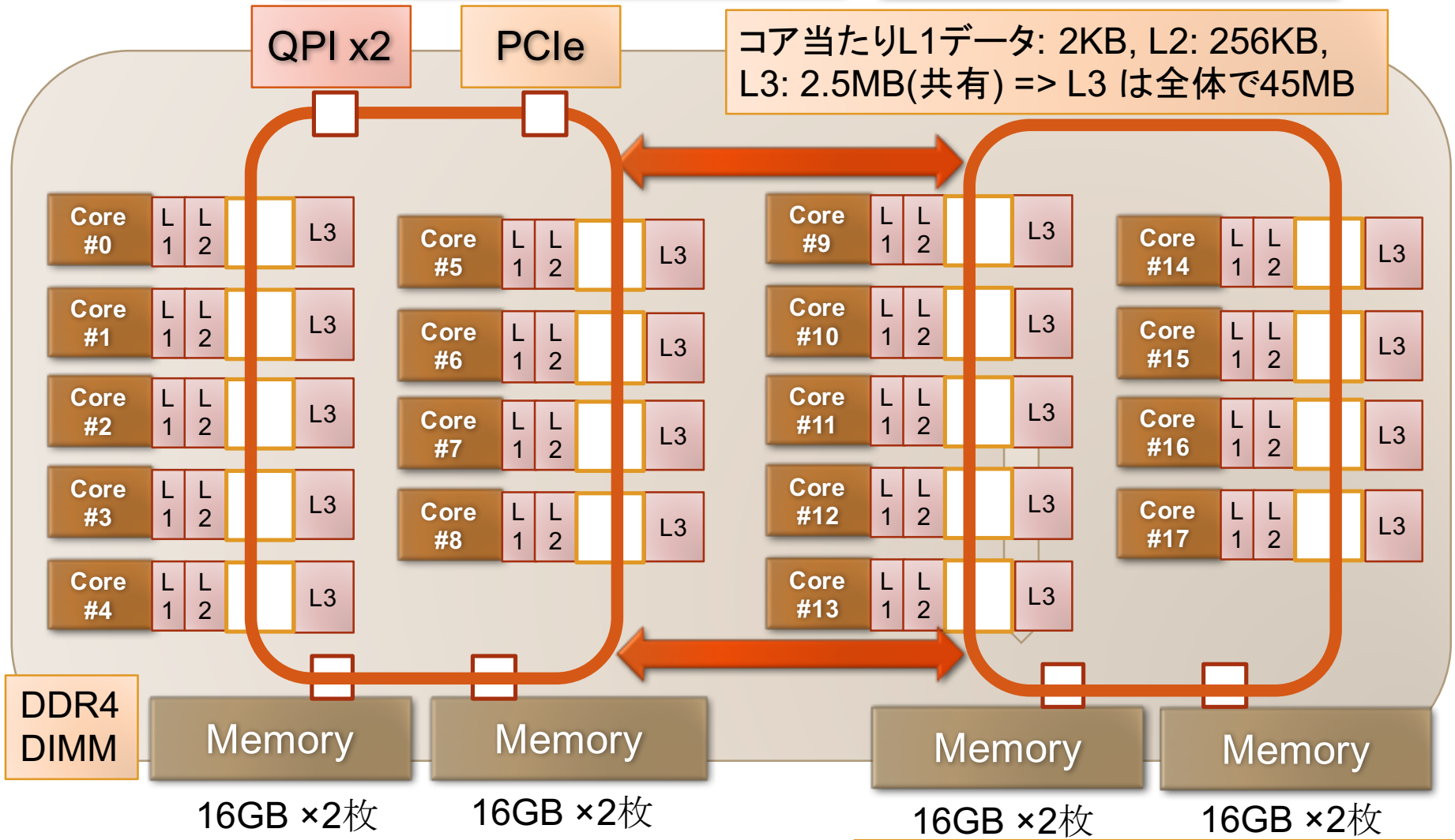
Reedbush-Uノードのブロック図

- 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)



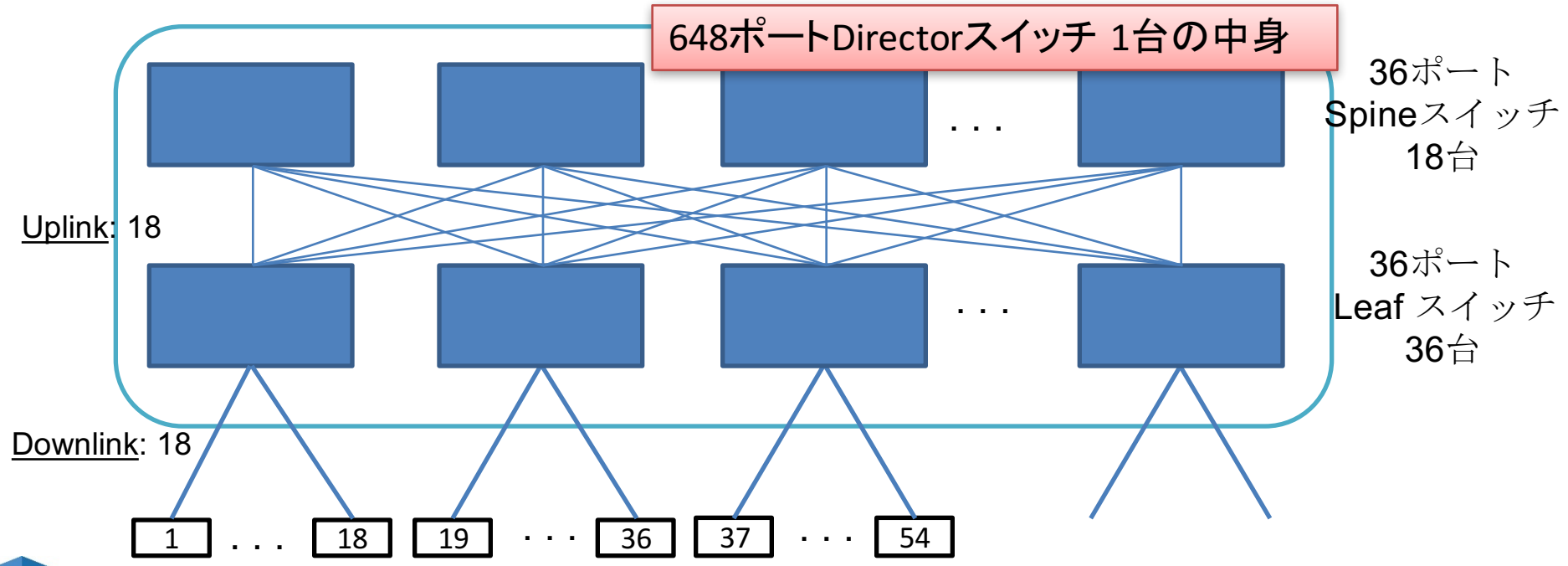
Broadwell-EPの構成

1ソケットのみを図示

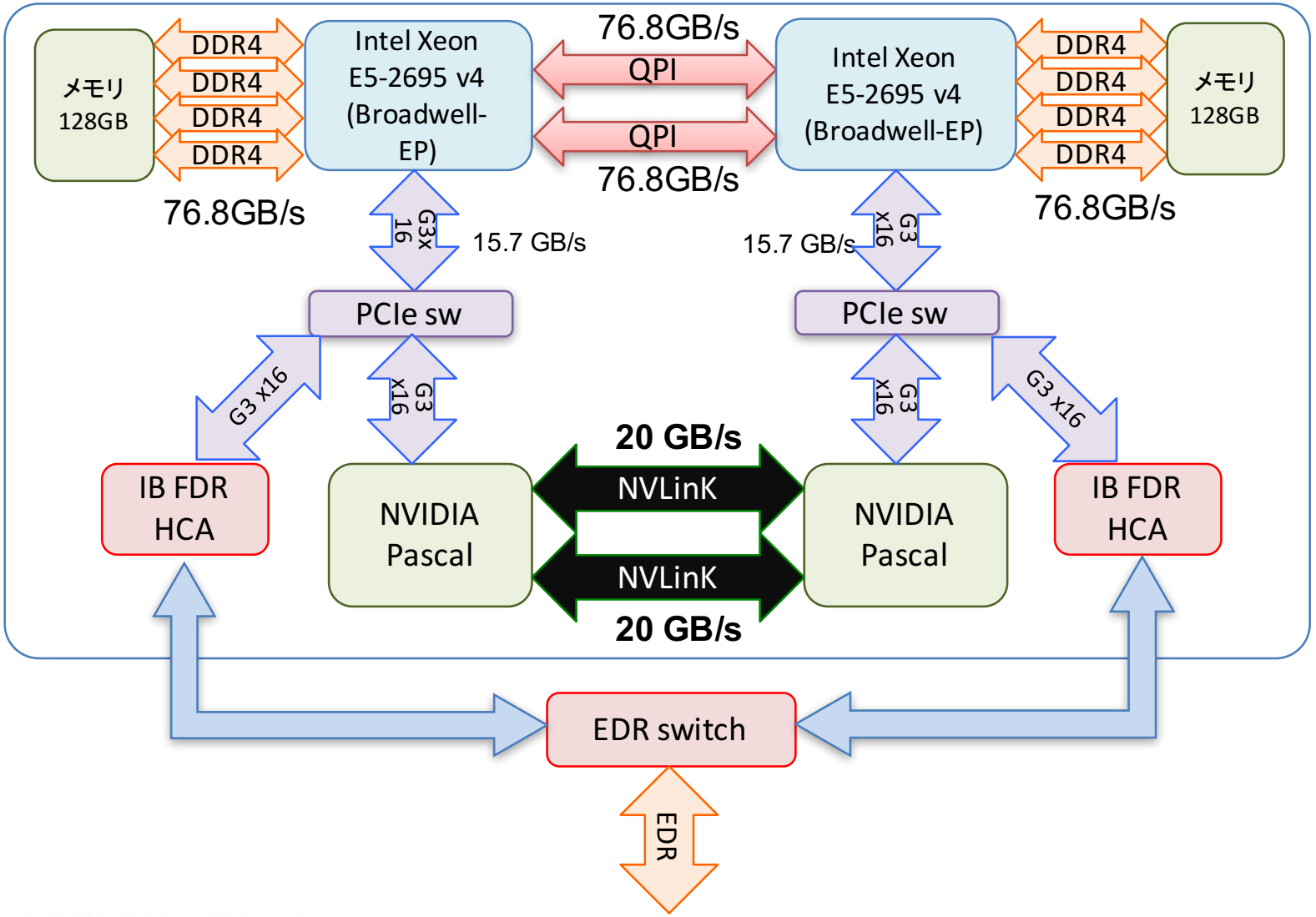


Reedbush-Uの通信網

- 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



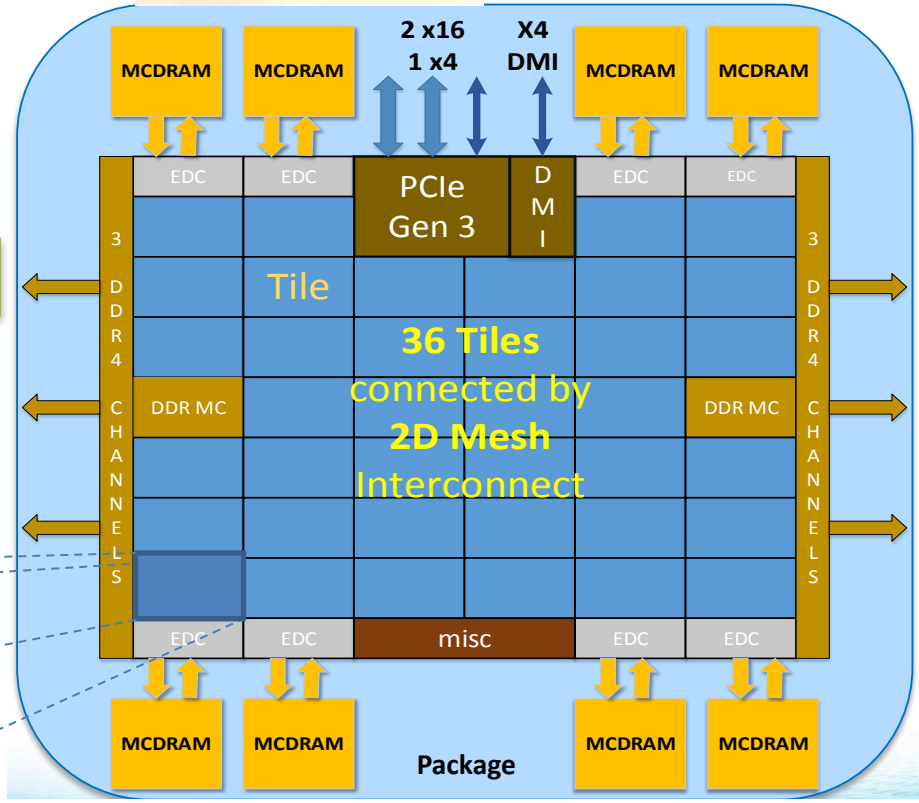
Oakforest-PACS 計算ノード

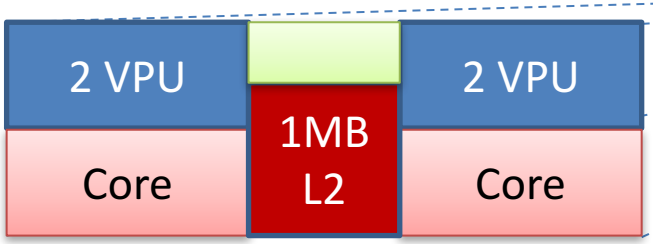
- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット
- MCDRAM: オンパッケージの高バンド幅メモリ16GB + DDR4メモリ

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（2018年4月1日）

- パーソナルコース（年間） 個人研究者（大学・公共機関）のみ
  - 1口 100,000円 = 17,280トークン(2ノード年相当)
  - 基準ノード数※ = 8ノード
  - 最大ノード数 = 2,048ノード
- グループコース
  - 1口 400,000円 (企業 480,000円) = 69,120トークン(8ノード年相当)
  - 基準ノード数※ = 口数×8ノード
  - 最大ノード数 = 2,048ノード

※トークン消費について

- 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（2018年4月1日 ）

- パーソナルコース（年間） 個人研究者（大学・公共機関）のみ
  - 1口 150,000円 = 17,280トークン
- グループコース
  - 1口 300,000円 (企業 480,000円) = 69,120トークン

	最大ノード数	基準ノード数	トークン消費係数 (基準ノード超分は2.0倍)
RB-U	128	口数×4	1.0
RB-H	32	口数×1	2.5
RB-L	16	口数×1	4.0

- 大学等のユーザはReedbushとの相互トークン移行も可能
- ノード固定もあり

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/

GPU入門

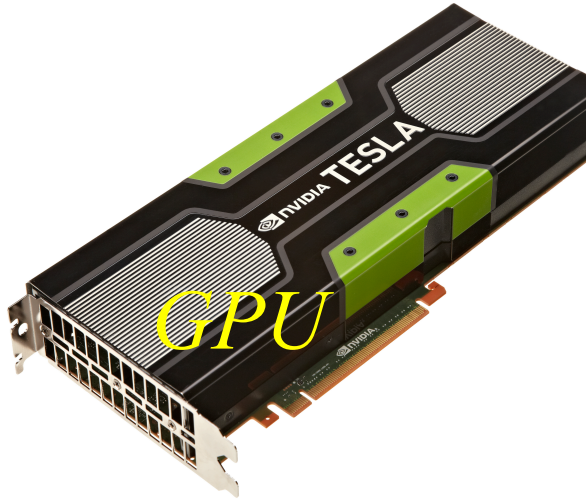
「GPUプログラミング入門」講習会の目的

GPUのアーキテクチャ

GPUの使い方

What's GPU ?

- ✓ Graphics Processing Unit
- ✓ もともと PC の3D描画専用の装置
- ✓ パソコンの部品として量産されてる。
  = 非常に安価



http://www.nvidia.co.jp

GPUコンピューティング

- GPUはグラフィックスやゲームの画像計算のために進化を続けている。
- CPUがコア数が2-12個程度に対し、GPUは1000以上のコアがある。
- GPUを一般のアプリケーションの高速化に利用することを「GPUコンピューティング」「GPGPU (General Purpose computation on GPU)」などという。
- 2007年にNVIDIA社のCUDA言語がリリースされて大きく発展
- ここ数年、ディープラーニング（深層学習）、機械学習、AI（人工知能）などでも注目を浴びている。

参考：NVIDIA社のGPU

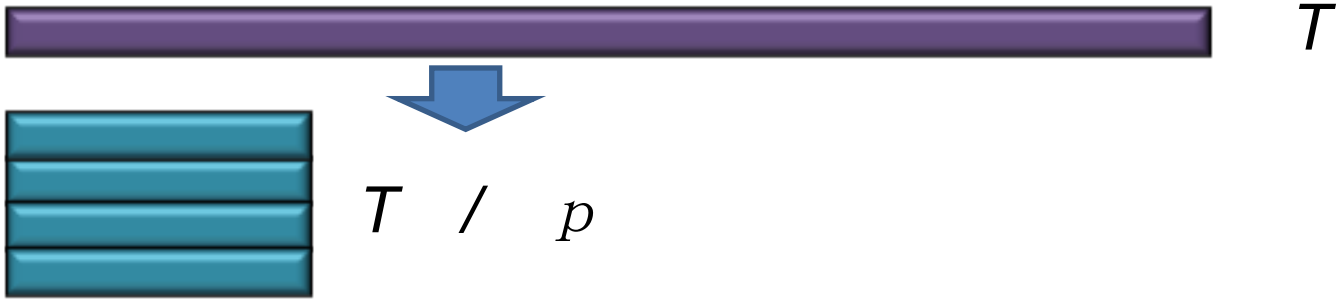
- 製品シリーズ
  - GeForce
    - コンシューマ向け。安価。
  - Tesla
    - HPC向け。倍精度演算器、大容量メモリ、ECCを備えるため高価。
- アーキテクチャ（世代）
  1. Tesla：最初のHPC向けGPU、TSUBAME1.2など
  2. Fermi：2世代目、TSUBAME2.0など
     - ECCメモリ、FMA演算、L1 L2 キャッシュ
  3. Kepler：現在HPCにて多く利用、TSUBAME2.5など
     - シャッフル命令、Dynamic Parallelism、Hyper-Q
  4. Maxwell：コンシューマ向けのみ
  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、アメリカの国立研究所で大規模運用予定
     - Tensor Coreなど

GPUの特徴

- なぜGPUが使われるのか？
  - ✓ 性能高い
    - ✓ NVIDIA P100 (Reedbush-H) 5,304 GFlops
    - ✓ Intel Broadwell (Reedbush-U) 604.8 GFlops
    - ✓ Intel Xeon Phi (Oakforest-PACS) 3,046.4 GFLops
  - ✓ 消費電力低い
    - ✓ スパコンに搭載されている
- コンピュータに取り付ける増設ボード
  - ✓ 単体では動作できず、**CPUからの指令が必要**。
  - ✓ CPUとGPUはメモリが異なるため、**メモリ間のデータ移動のプログラミングも必要**。
- 多数の小さなコア（1000以上）を搭載。これを有効に活用するため**「並列計算」が必須**。
  - ✓ CPUは大きなコア：分岐予測、パイプライン処理などできる。逐次処理が得意。
  - ✓ GPUは小さなコア：CPUの持つ機能がほとんどないか、全くない。

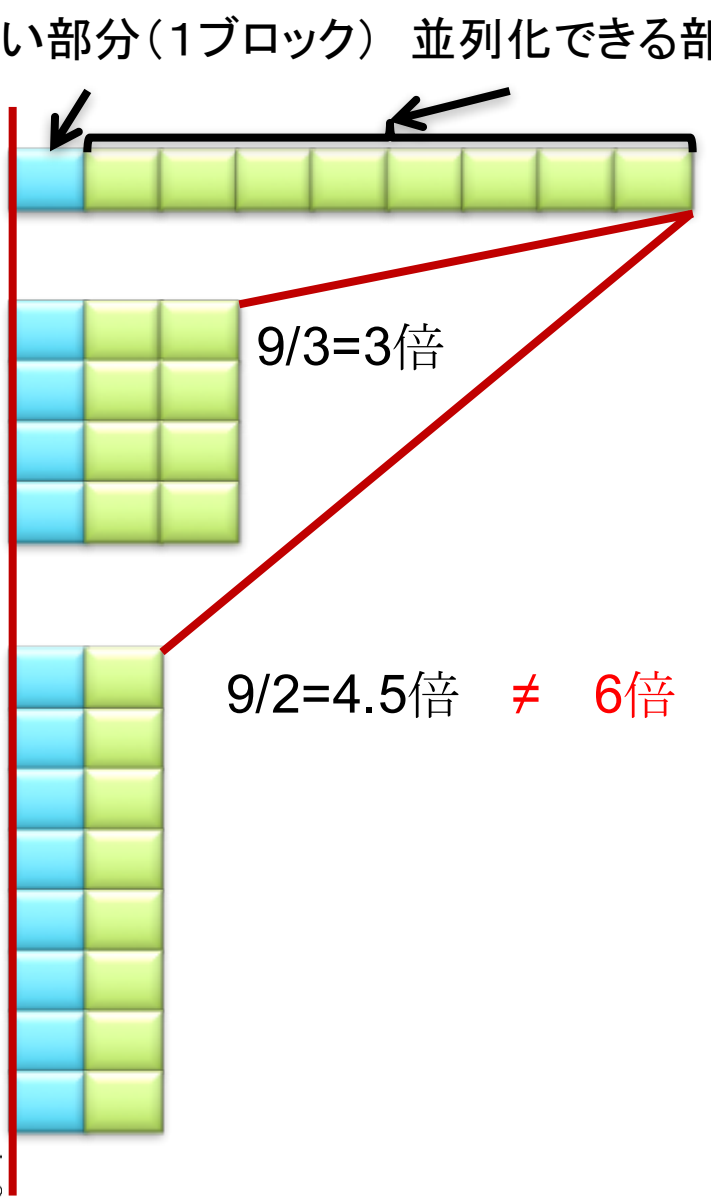
並列プログラミングとは何か？

- 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）

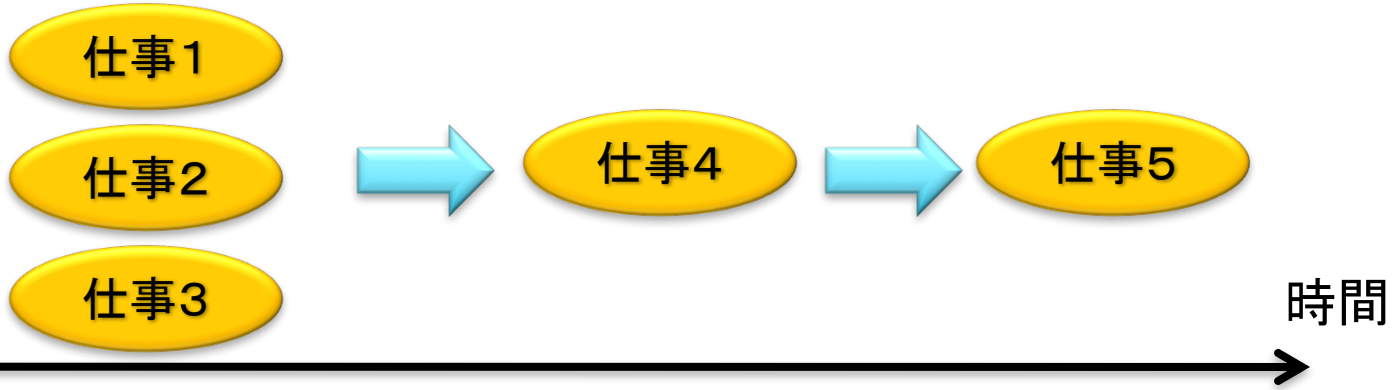
9/2=4.5倍　≠　6倍

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



データ並列

- データを分割することで並列化する。
  - ✓ データは異なるが計算の手続きは同じ。
- データ並列の例：手分けをして算数ドリルを解く
  - ✓ 数字だけ異なるが計算の手続きは同じ。

2 + 1 =	12 – 88 =	34 + 3 =	2 + 4 =
3 + 19 =	- 20 + 29 =	1 + 2 =	99 – 72 =
4 - 6 =	4 + 10 =	-3 + 2 =	2 + 10 =
5 + 3 =	1 + 10 =	-10 – 10 =	3 – 11 =
-2 + 10 =	1 + 13 =	2 + 1 =	12 – 34 =
1 + 2 =	0 + 0 =	1 -10	45 + 19 =

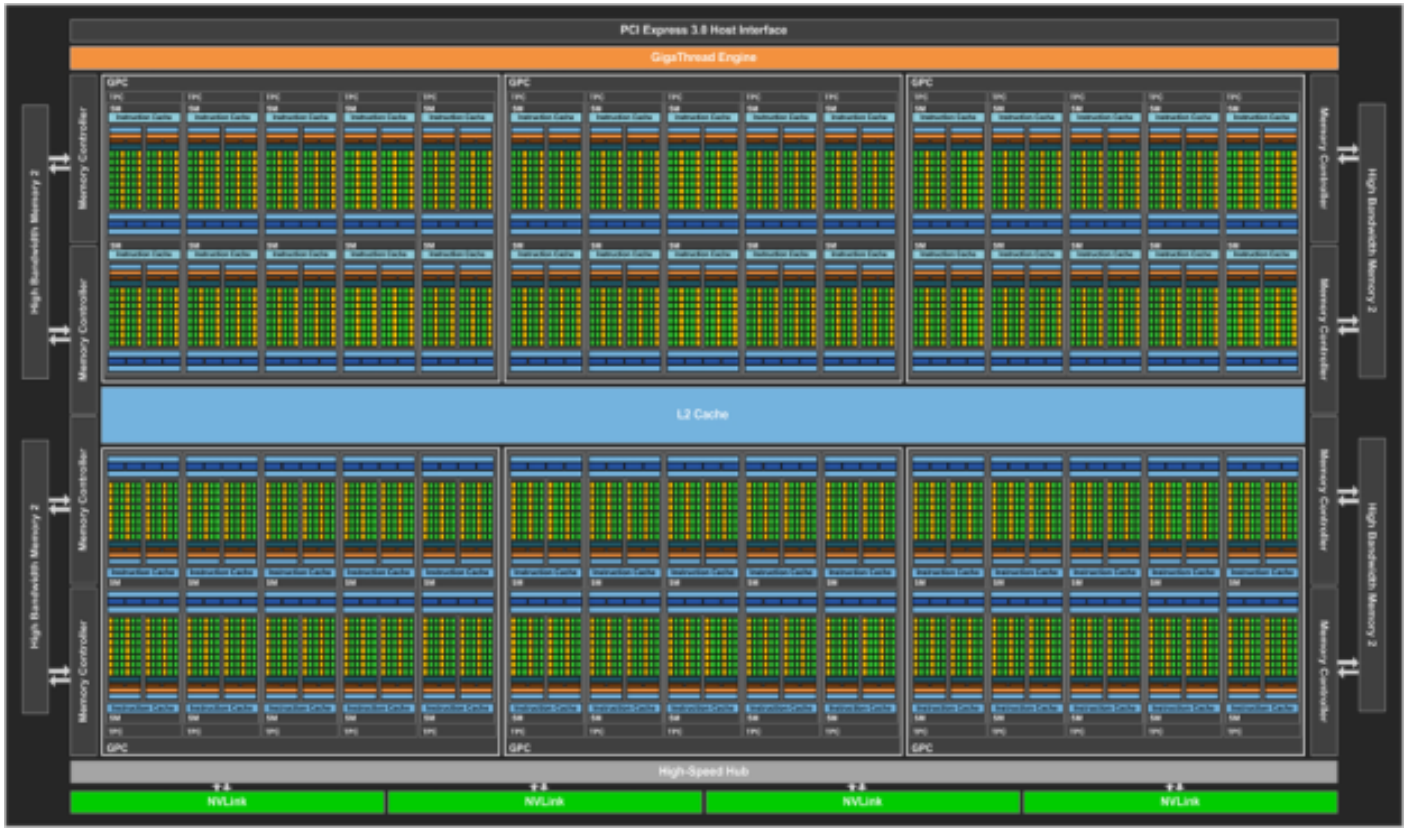
GPUではこれが原則！

抑えておくべきGPUの特徴

- 超並列計算が必須
- CPUと独立のGPUメモリ
- 階層的スレッド管理とコミュニケーション
- Warp 単位の実行
- コアレスドアクセス

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

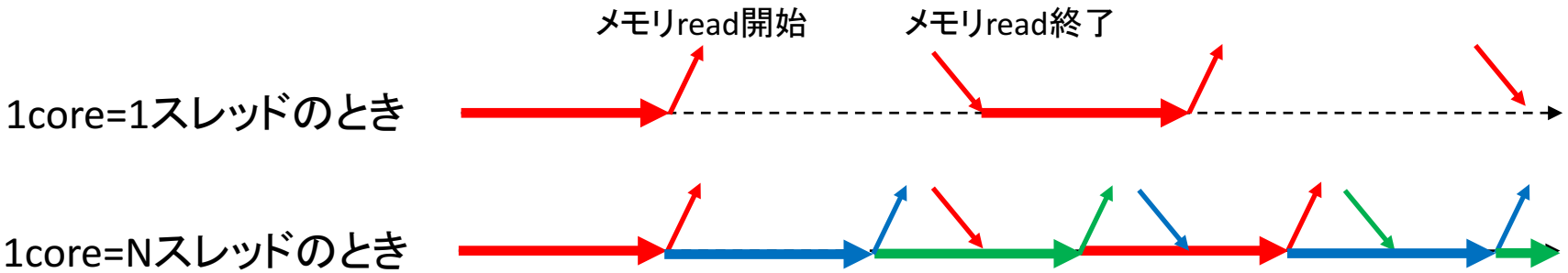
GP100 SM

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

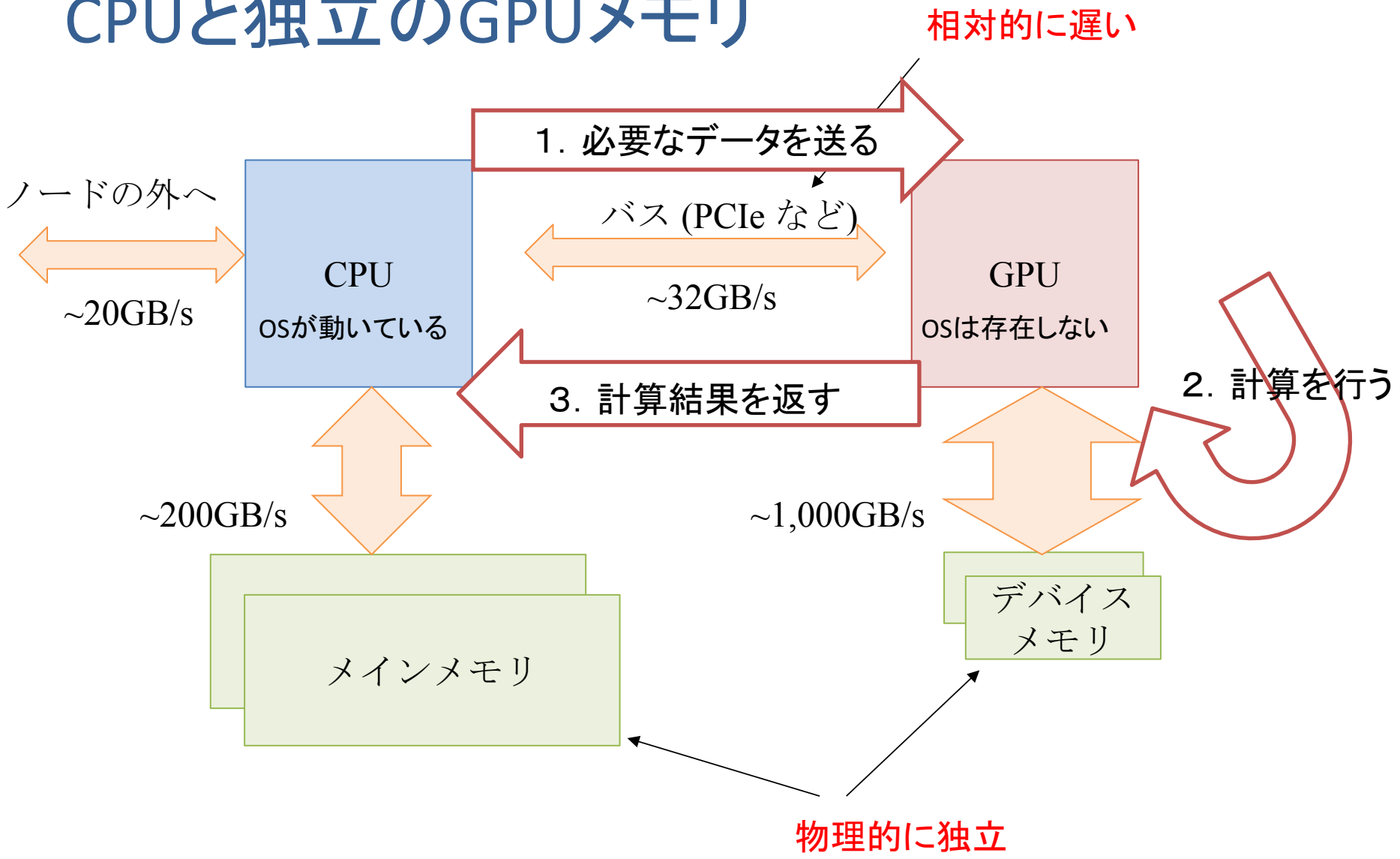
P100 whitepaper より

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

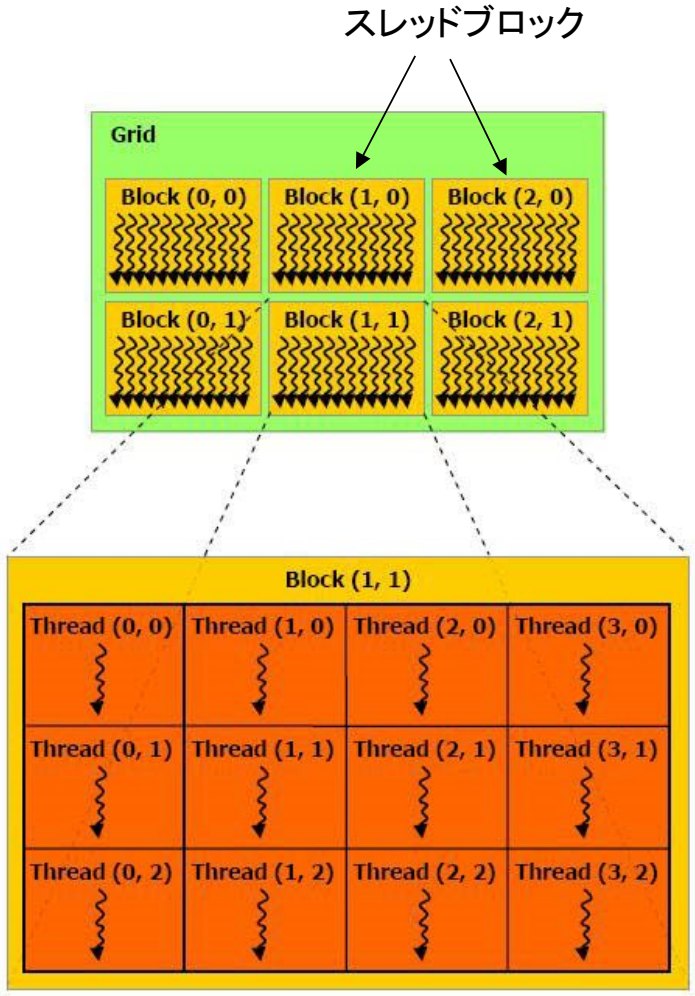


CPUと独立のGPUメモリ



階層的スレッド管理とコミュニケーション

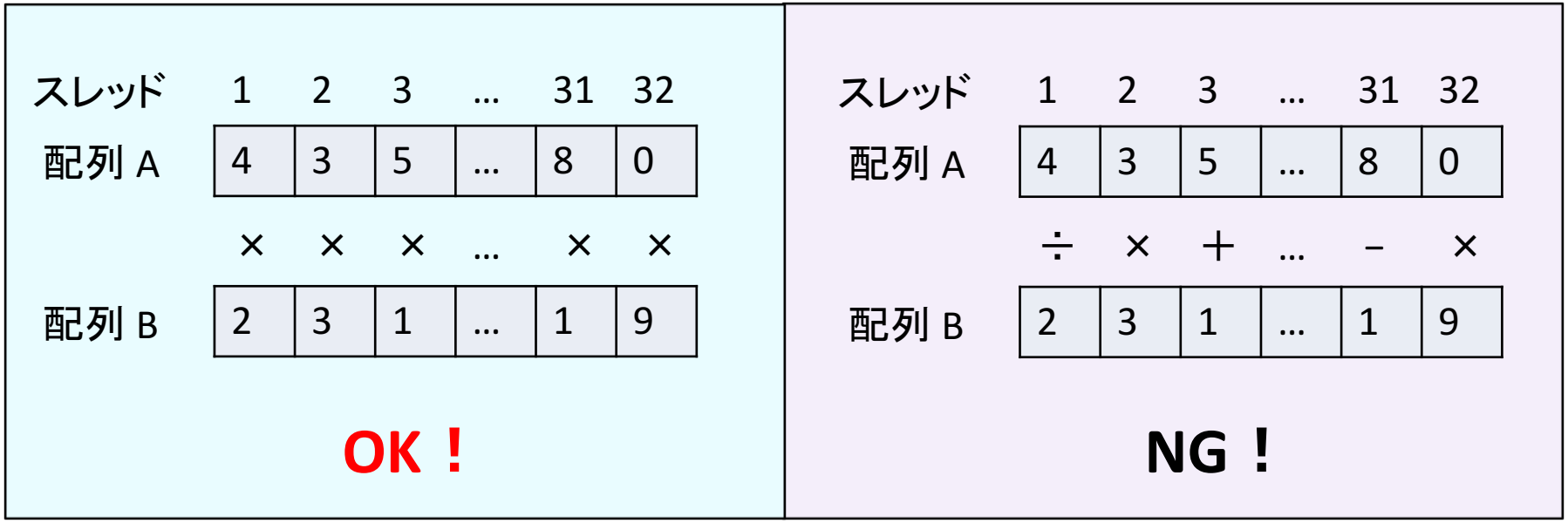
- 階層的なコア/スレッド管理
  - P100は56 SMを持ち、1 SMは64 CUDA coreを持つ。トータル3584 CUDA core
  - 1 SMが複数のスレッドブロックを担当し、1 CUDA core が複数スレッドを担当
- スレッド間のコミュニケーション
  - 同一スレッドブロック内のスレッドは高速コミュニケーション可能
  - 異なるスレッドブロックに属するスレッド間はコミュニケーションが低速
    - いったんメモリに書き出したり、CPUに処理を戻さなくてはならない



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

Warp 単位の実行

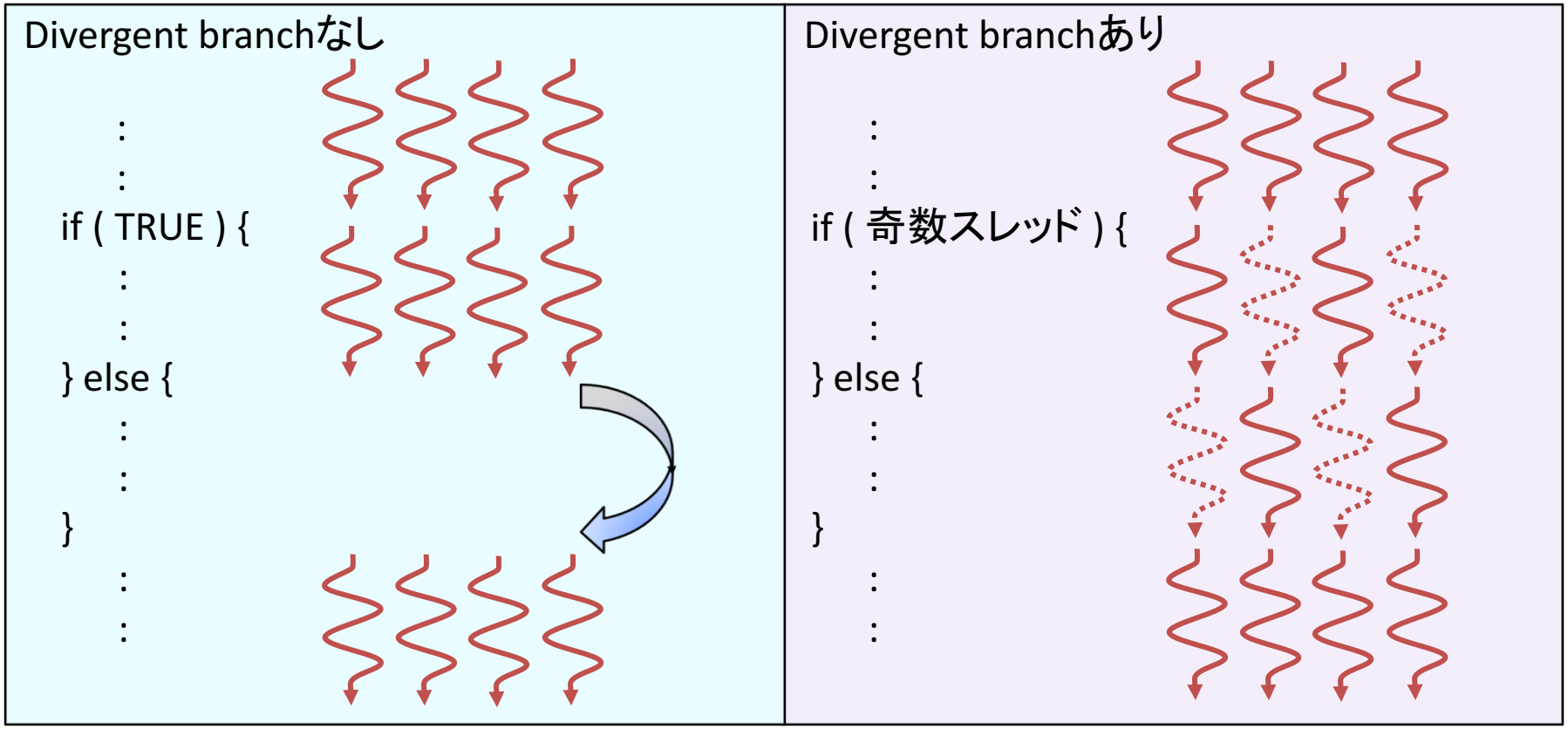
- 連続した32スレッドを1単位 = Warp と呼ぶ
- このWarpは足並み揃えて動く
  - 実行する命令は32スレッド全て同じ
  - データは違ってもいい



Warp内分岐

CUDA 8 以前のバージョン
（本講習会はCUDA 8 準拠）

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



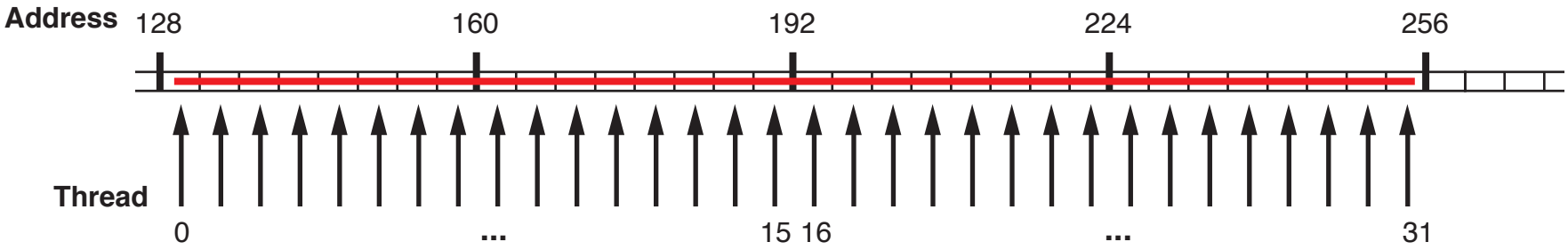
else 部分は実行せずジャンプ

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

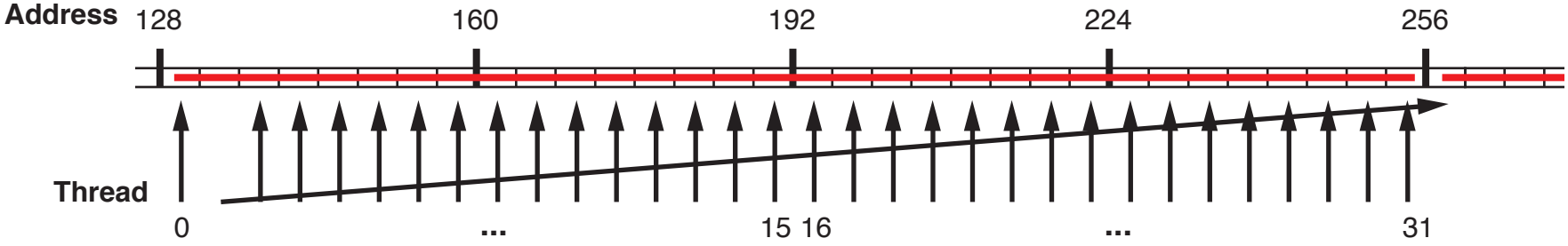
コアレスドアクセス

- 同じWarp内のスレッド（連続するスレッド）は近いメモリアドレスへアクセスすると効率的
  - ✓ コアレスドアクセス（coalesced access）と呼ぶ
  - ✓ メモリアクセスは128 Byte 単位で行われる。128 Byte に収まれば1回のアクセス、超えれば128 Byte アクセスをその分繰り返す。

128 byte x 1回のメモリアクセス



128 byte x 2回のメモリアクセス

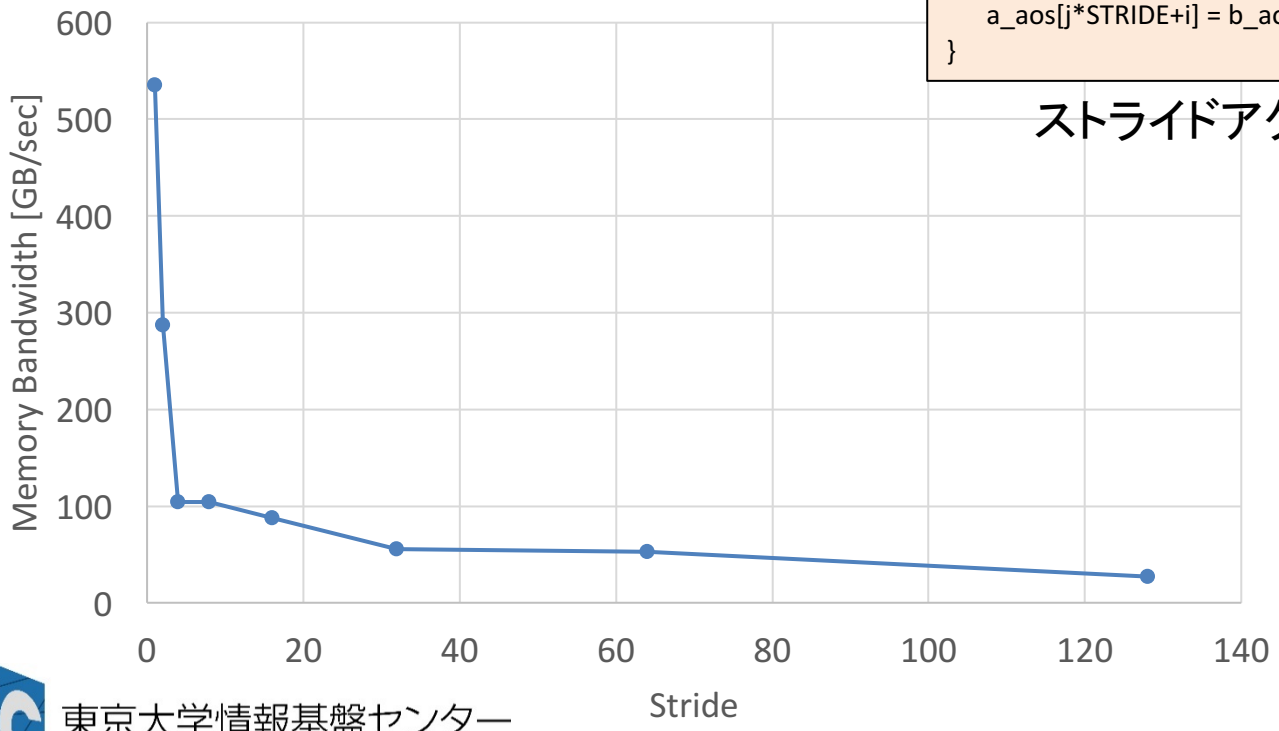


ストライドアクセスがあるとどうなるか

- 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コンピューティングの方法

- ライブラリの利用（CUFFT, CUBLAS など）
  - ✓ GPU用ライブラリを呼ぶだけで、すぐに利用できる。
  - ✓ ライブラリ以外の部分は高速化されない。
- 指示文ベース（OpenACC）
  - ✓ 指示文（ディレクティブ）を挿入するだけである程度高速化。
  - ✓ 既存のソースコードを活用できる。
- プログラミング言語（CUDA、OpenCLなど）
  - ✓ GPUの性能を最大限に活用。
  - ✓ プログラミングにはGPGPU用言語を使用する必要あり。

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

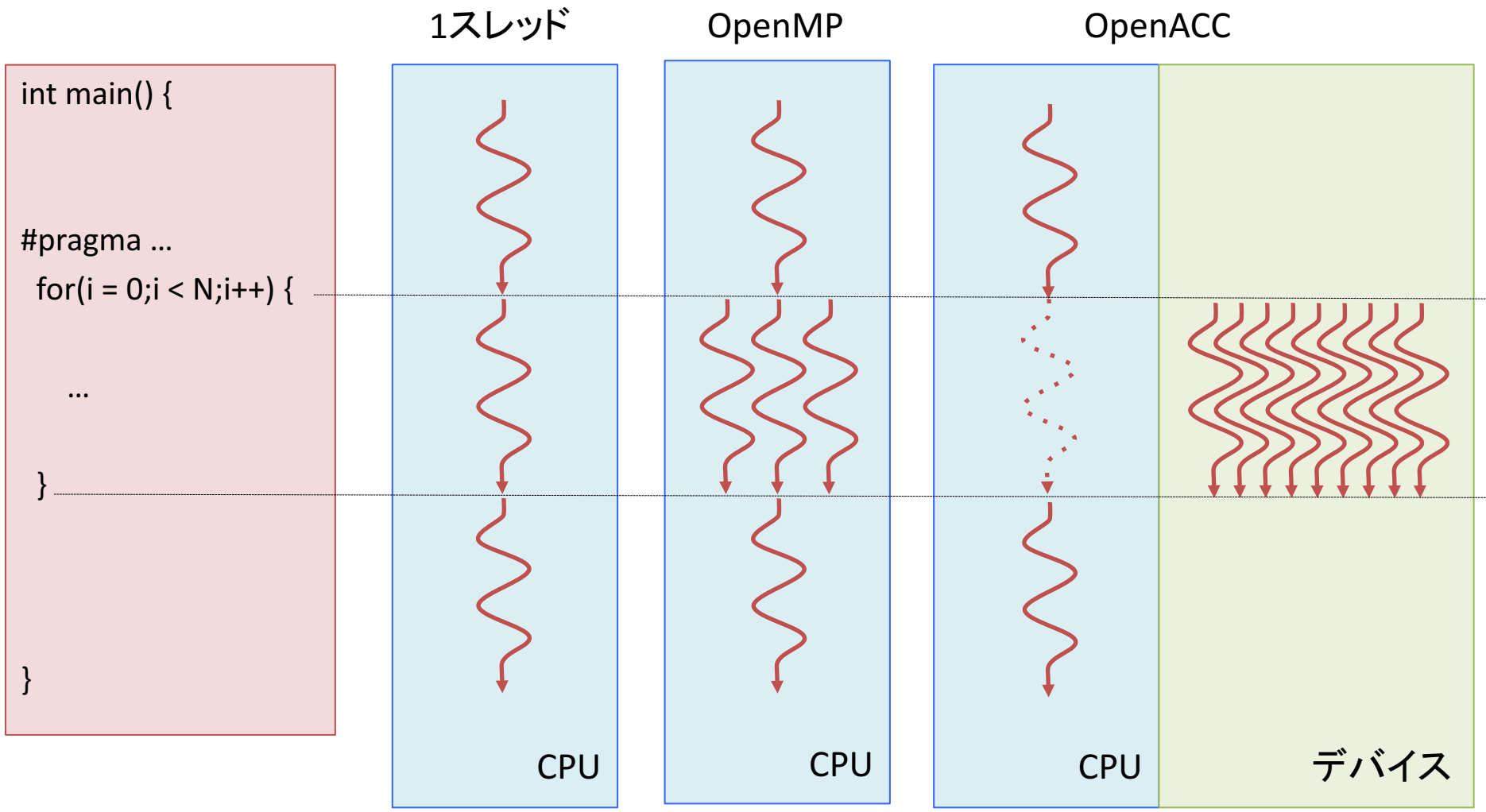
```
!$acc kernels
do i = 1, N
  ….
end do
!$acc end kernels
```

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

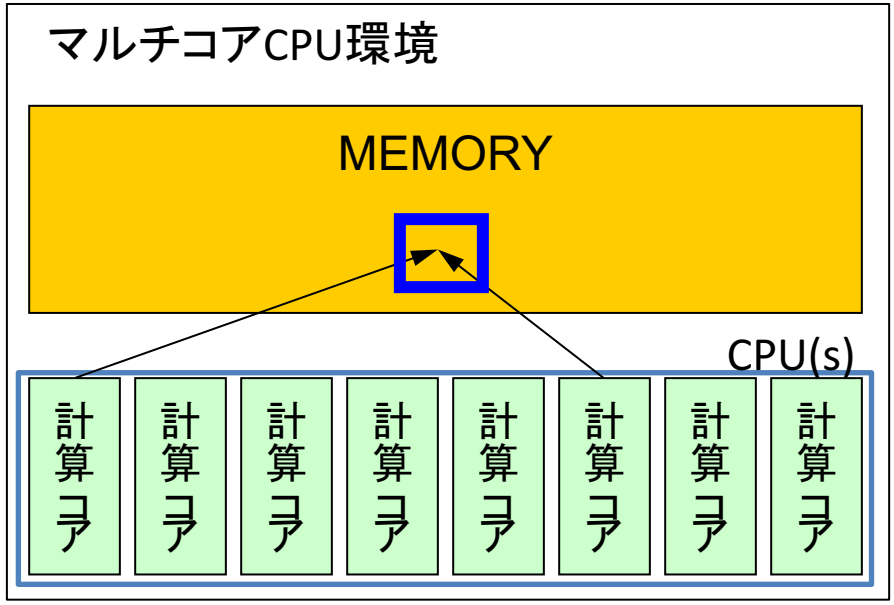
本講習会ではPGIコンパイラを用いる

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

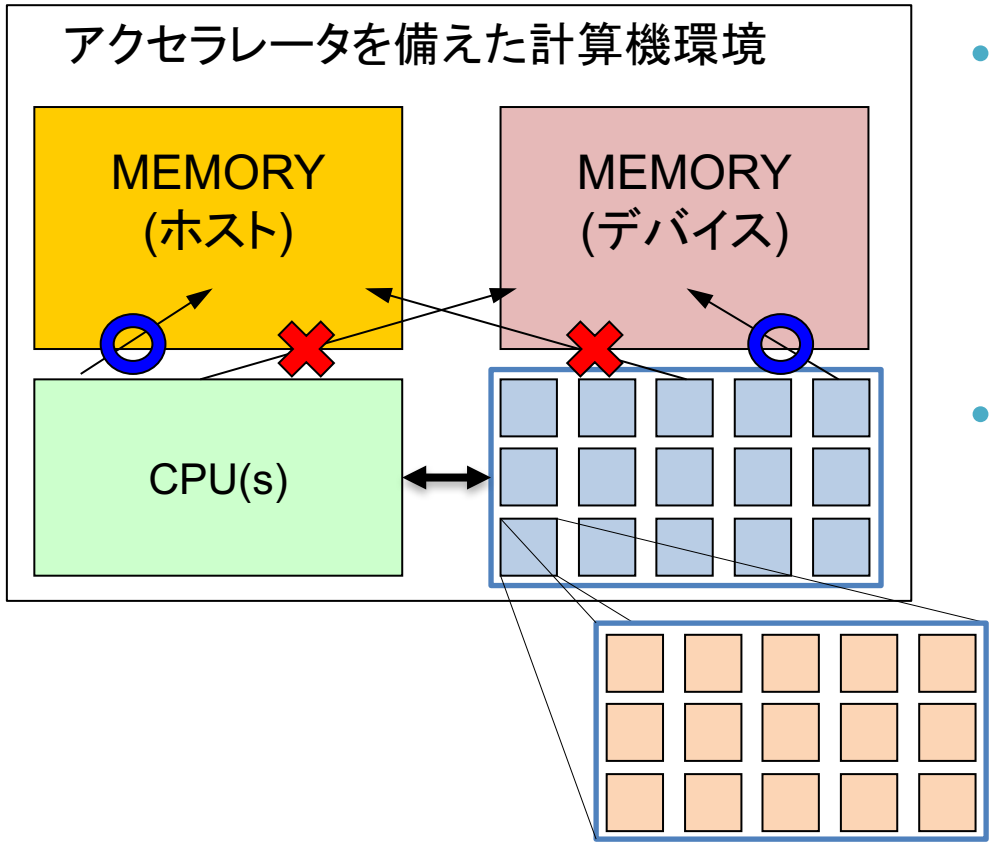


- 計算コアがN個
  - N < 100 程度 (Xeon Phi除く)
- 共有メモリ

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

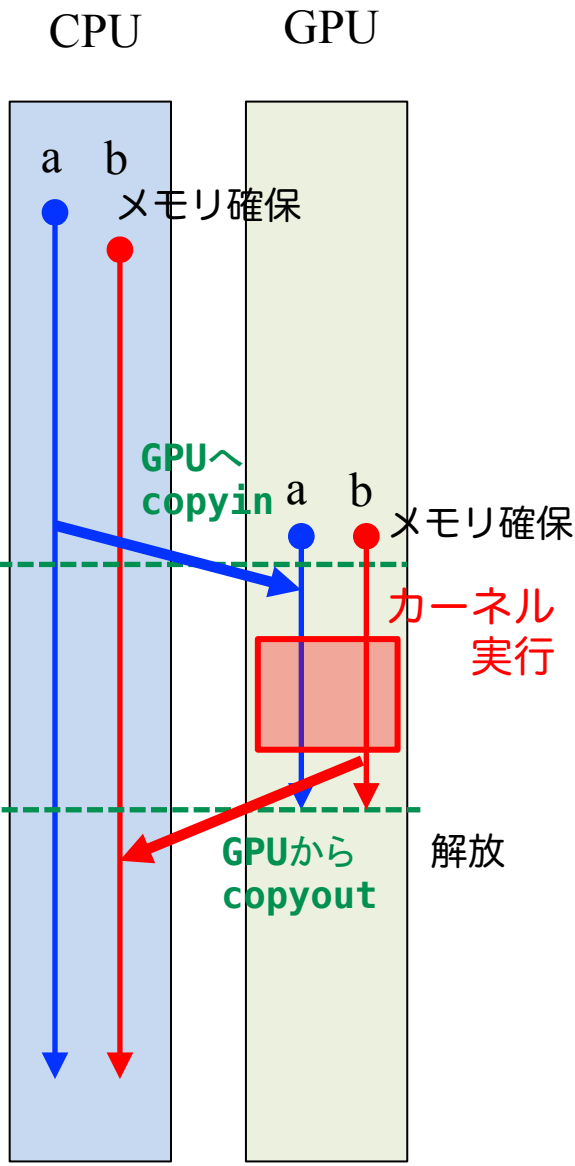
- OpenMP
  - 全部 shared
- OpenACC
  - スカラ変数：firstprivate or private
  - 配列：shared
    - プログラム上のparallel/kernels構文に差し掛かった時、OpenACCコンパイラは実行に必要なデータを自動で転送する
    - 正しく転送されないこともある。自分で書くべき
    - 構文に差し掛かるたびに転送が行われる(非効率)。後述のdata指示文を用いて自分で書くべき
    - 配列はデバイスに確保される (shared的振る舞い)
    - 配列変数をprivateに扱うためには private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

OpenACC の指示文

はじめてのOpenACCコード

openacc_hello/01_hello_acc

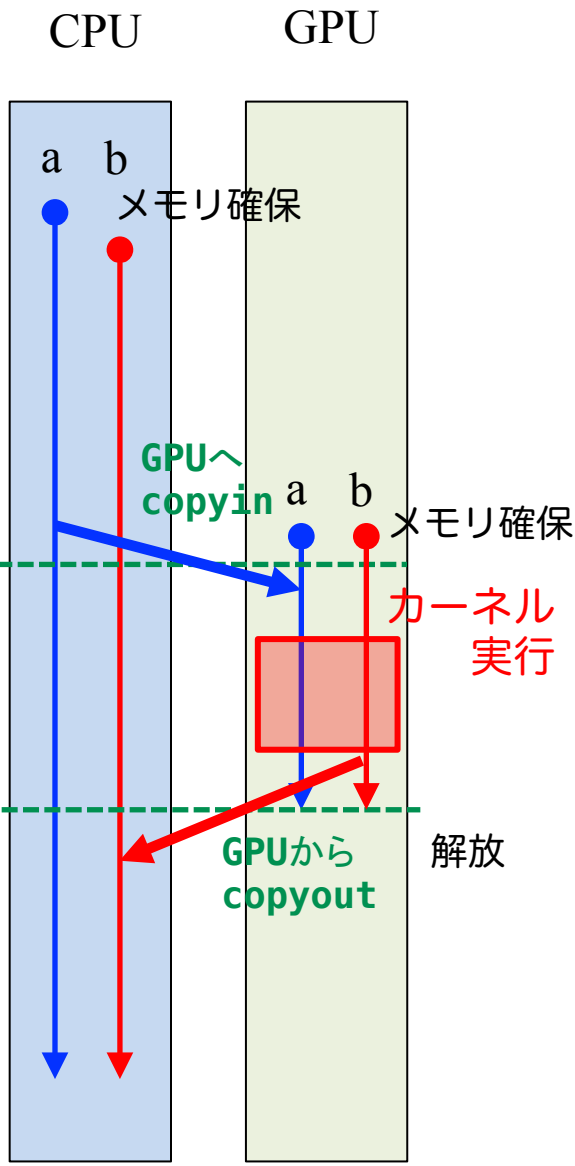
```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
    fprintf(stdout, "%f¥n", sum/n);
    free(a); free(b);
    return 0;
}
```



はじめてのOpenACCコード

openacc_hello/01_hello_acc

```
int main(){
    const int n = 1000;
    float *a = malloc(n*sizeof(float));
    float *b = malloc(n*sizeof(float));
    float  c = 2.0;
    for (int i=0; i<n; i++) {
        a[i] = 10.0;
    }

#pragma acc data copyin(a[0:n]), copyout(b[0:n])
#pragma acc kernels
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        b[i] = a[i] + c;
    }

    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += b[i];
    }
```



コード上同じ a, b であっても、原則として
ホストコードはホストメモリで確保された a, b 、GPUで実行される並列領域（カーネル）はデバイスメモリで確保された a, b を参照しにいく。

参考：OpenACC 化とCUDA化の比較

```
// OpenACC

void calc(int n, const float *a,
const float *b, float c, float *d)
{
#pragma acc kernels present(a, b, d)
#pragma acc loop independent
    for (int i=0; i<n; i++) {
        d[i] = a[i] + c*b[i];
    }
}

int main()
{
    ...
#pragma acc data copyin(a[0:n], b[0:n]) copyout(d[0:n])
    {
        calc(n, a, b, c, d);
    }
    ...
}
```

kernel

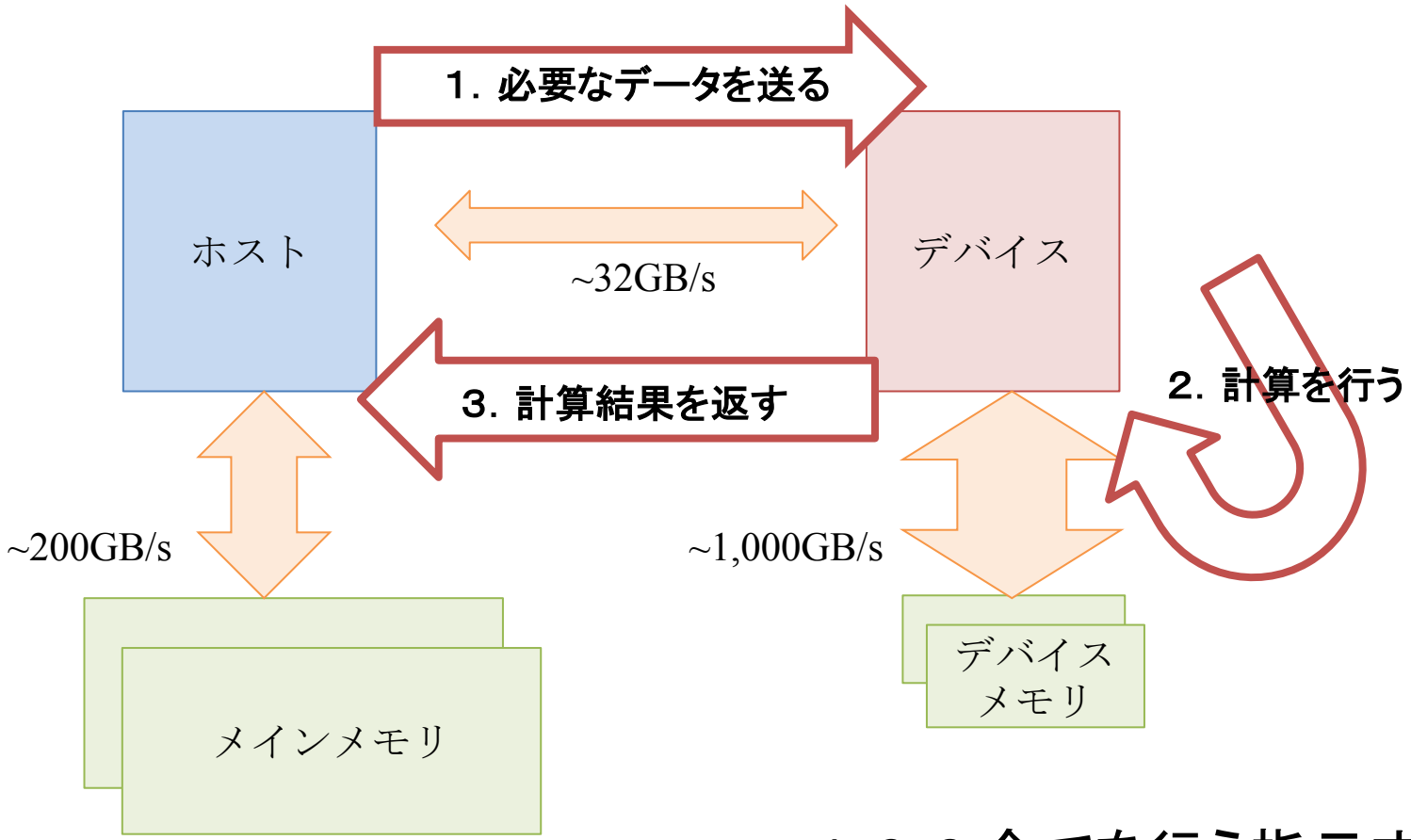
- ✓ kernels 指示文でGPUでの実行領域を指定。
- ✓ loop 指示文で並列処理の最適化。
- ✓ data 指示文でデータ転送を制御。kernels 指示文でデータ転送を自動的に行うこともできる。

```
// CUDA

__global__
void calc_kernel(int n, const float *a, const float *b,
float c, float *d)
{
    const int i   = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        d[i] = a[i] + c*b[i];
    }
}

void calc(int n, const float *a, const float *b, float c,
float *d)
{
    dim3 threads(128);
    dim3 blocks((n + threads.x - 1) / threads.x);

    calc_kernel<<<blocks, threads>>>(n, a, b, c, d);
    cudaThreadSynchronize();
}

int main()
{
    ...

    float *a_d, *b_d, *d_d;
    cudaMalloc(&a_d, n*sizeof(float));
    cudaMalloc(&b_d, n*sizeof(float));
    cudaMalloc(&d_d, n*sizeof(float));

    cudaMemcpy(a_d, a, n*sizeof(float), cudaMemcpyDefault);
    cudaMemcpy(b_d, b, n*sizeof(float), cudaMemcpyDefault);
    cudaMemcpy(d_d, d, n*sizeof(float), cudaMemcpyDefault);

    calc(n, a_d, b_d, c, d_d);

    cudaMemcpy(d, d_d, n*sizeof(float), cudaMemcpyDefault);
    ...
}
```

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
```

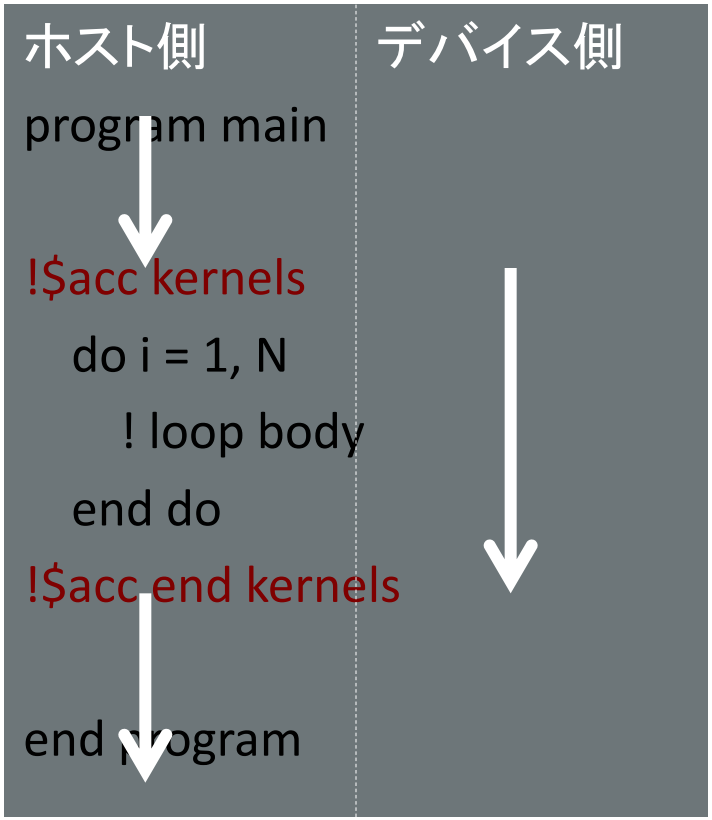
parallel

```
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

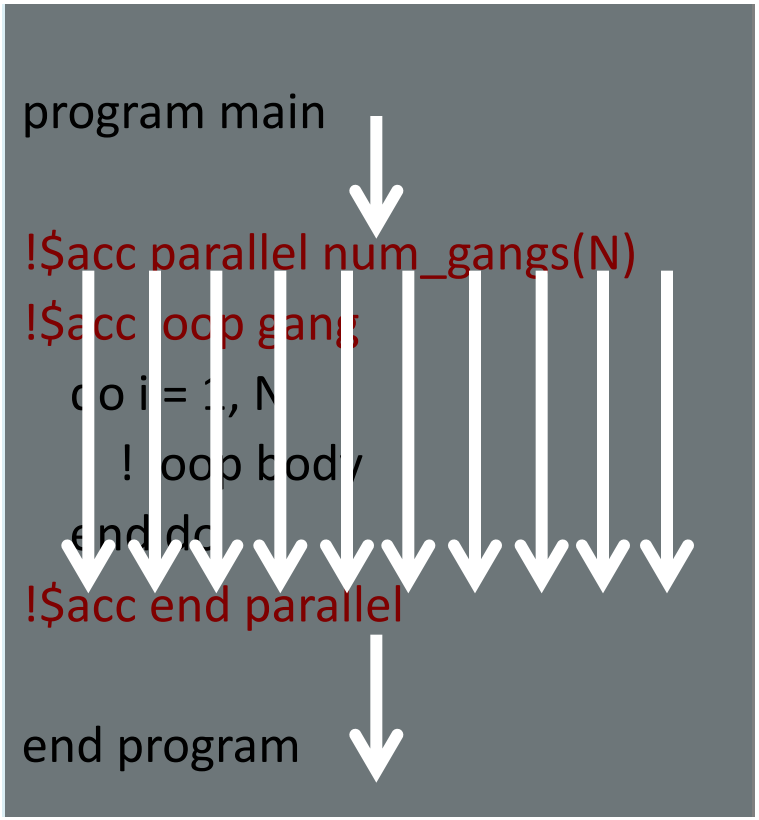
- ホスト-デバイスを意識するのがkernels
- 並列実行領域であることを意識するのがparallel

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
```

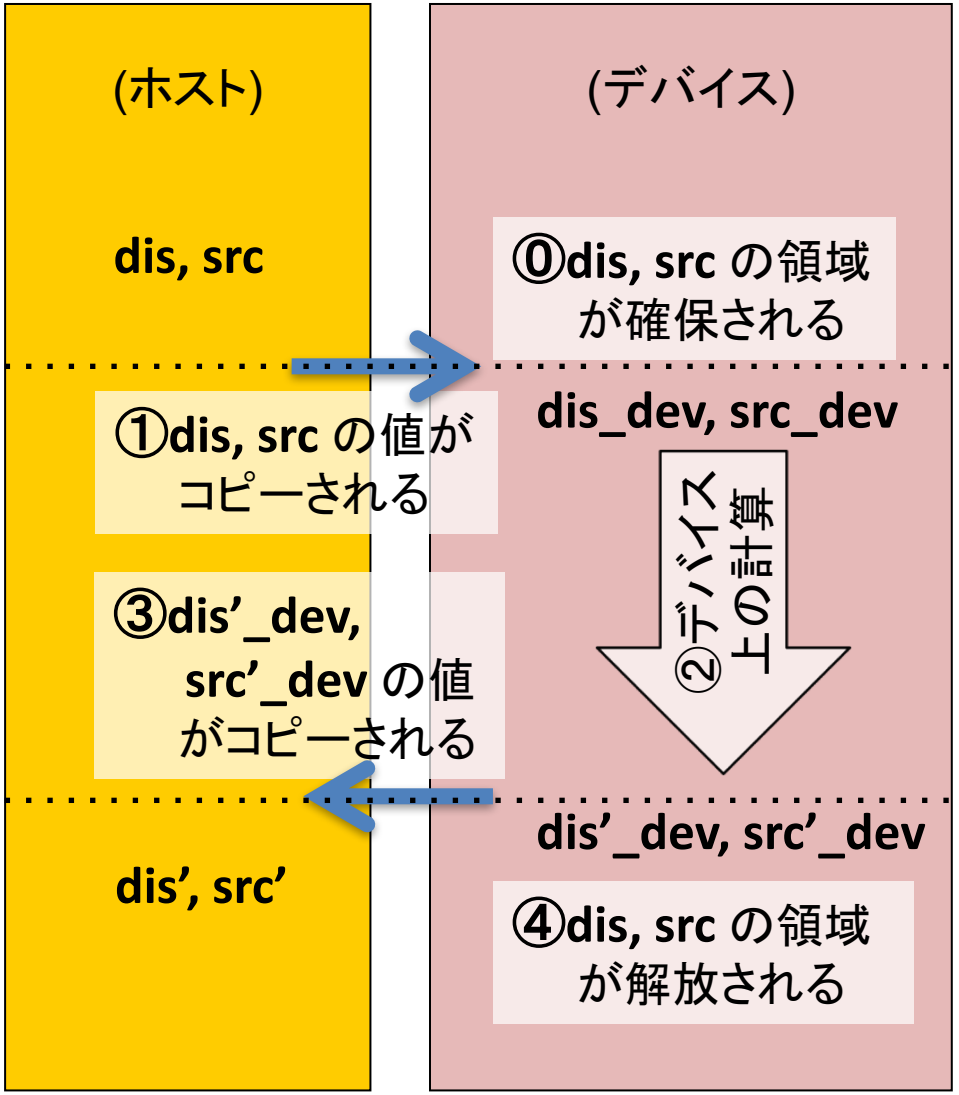
C言語

```
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

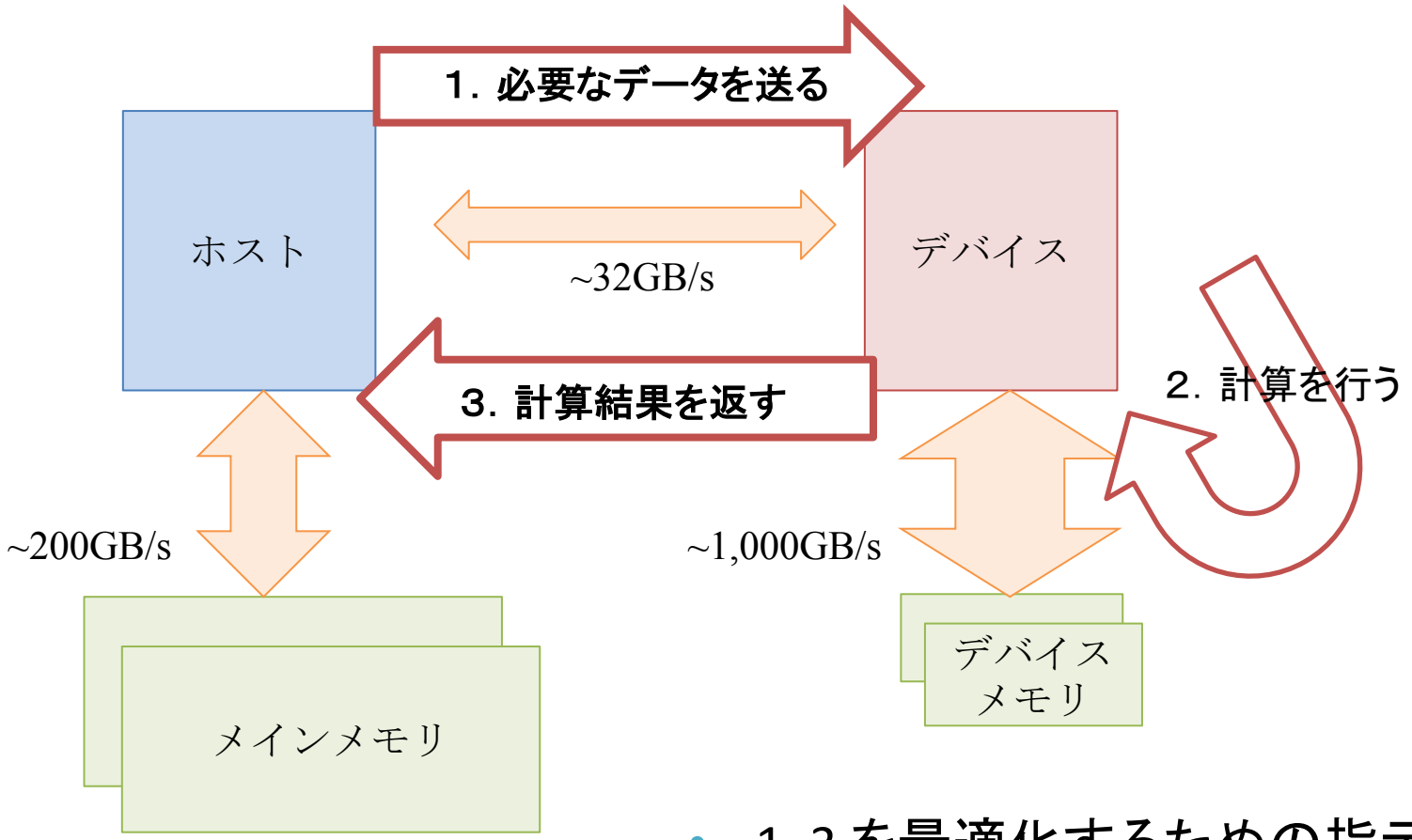
```
subroutine copy(dis, src)
  real(4), dimension(:) :: dis, src

!$acc kernels copy(src,dis)
  do i = 1, N
    dis(i) = src(i)
  end do
!$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 (Host ⇄ Device) + 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 (Host → Device)
  - 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 (Host ← Device) + 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

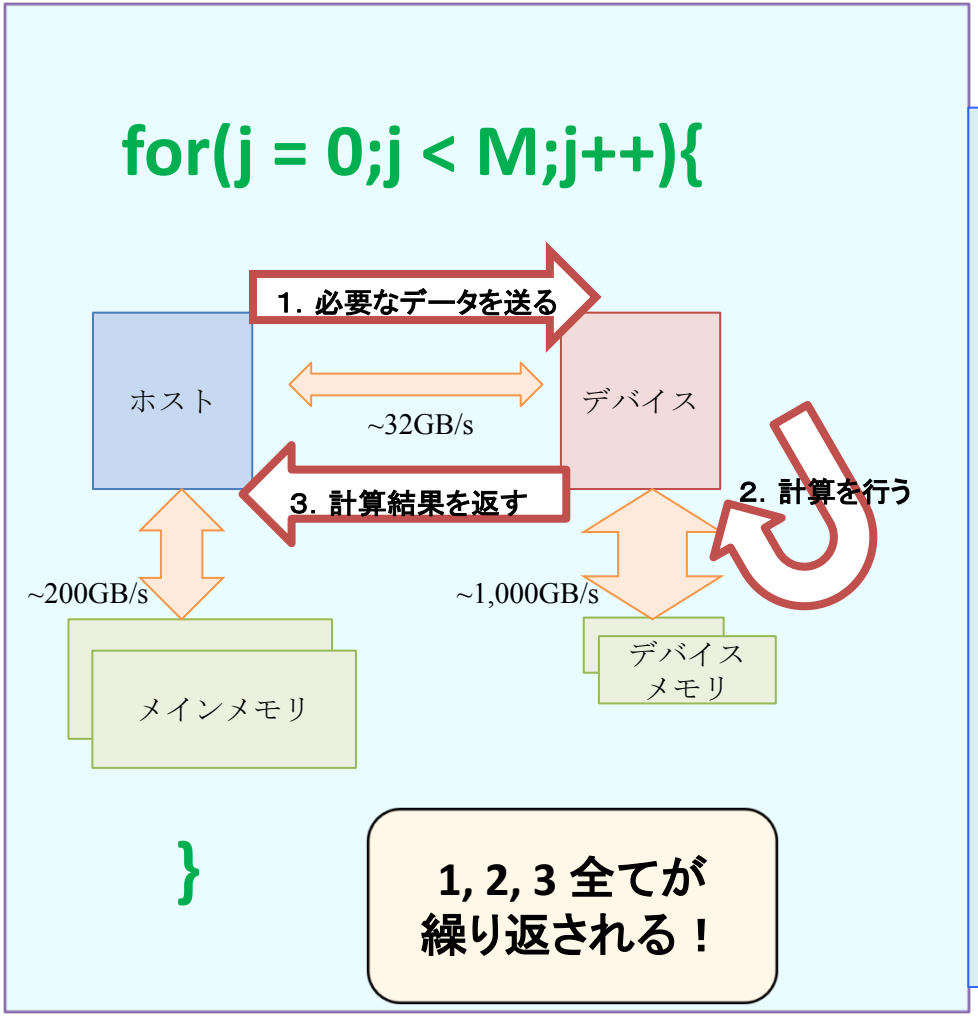
```
subroutine copy(dis, src)
  real(4), dimension(:) :: dis, src
  do j = 1, M
!$acc kernels copy(src,dis)
      do i = 1, N
         dis(i) = dis(i) + src(i)
      end do
!$acc end kernels
  end do
end subroutine copy
```

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
```

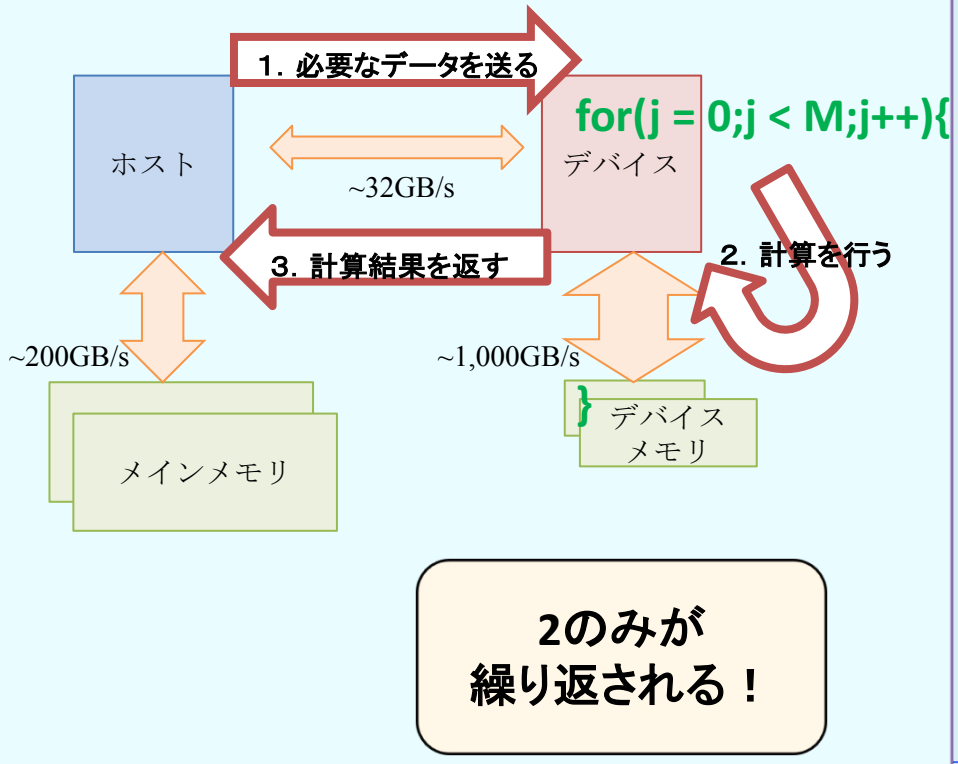
present: 既に転送済であることを示す
(OpenACC2.5の仕様以降、copyはpresent_or_copyとして扱われることになったので、実は書き換えなくても大丈夫になった。)

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

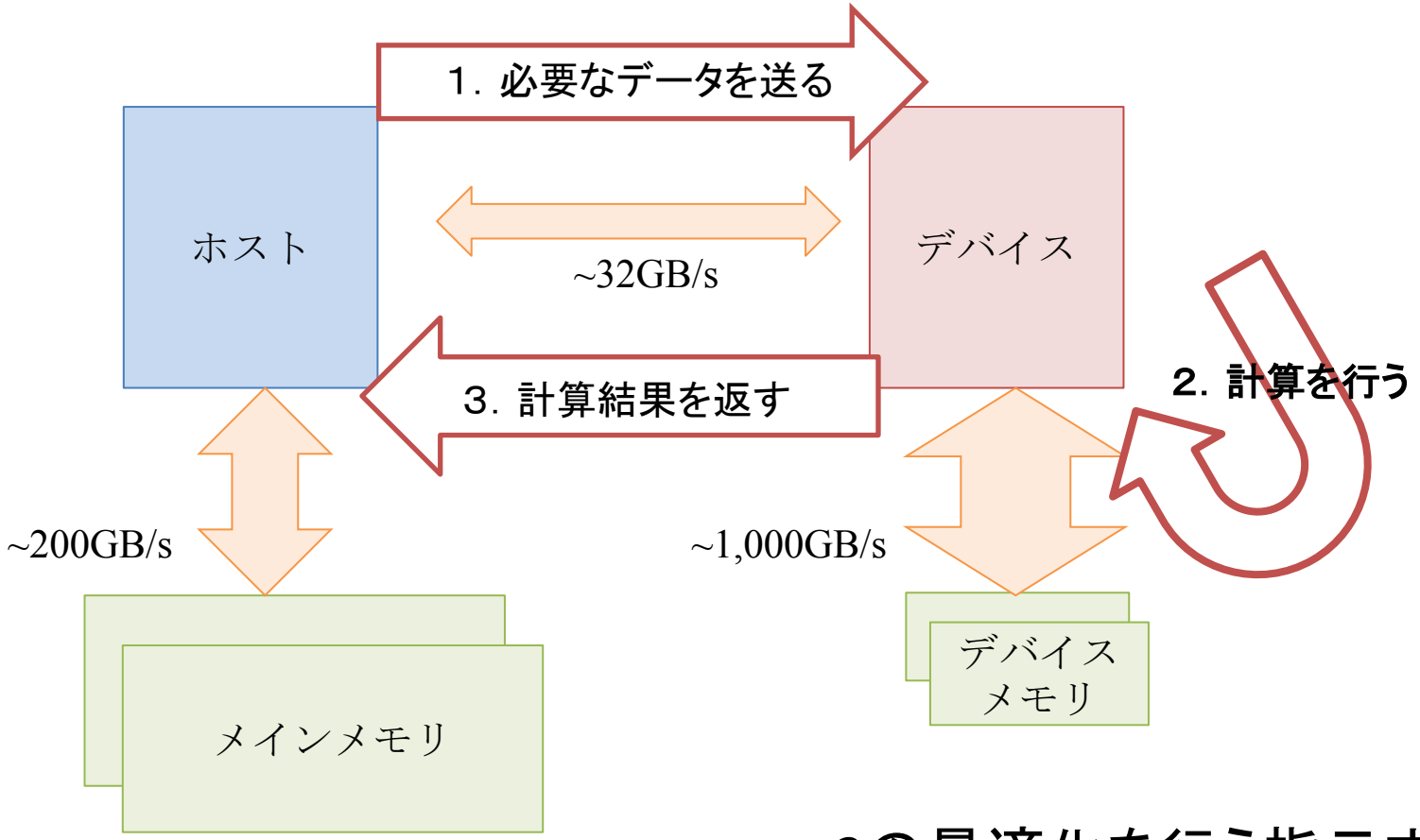
- データ指示文などで既にデバイス上に確保済みのデータを対象とする
  - Memcpy (H ⇄ D) の機能を持っていると思えば良い

```
!$acc data copy( A(:,:) )
do step = 1, N
  …
!$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

ループ指示文

- loop 指示文
  - parallel/kernels中でしか使えない
  - 並列化したいループに指定
  - ループマッピングのパラメータ調整
    - 粒度(gang, worker, vector, seq)の指定
    - ある程度はコンパイラが自動で決定してくれるため、さほど気にしなくていい
  - データの独立性の保証 (independent)
    - コンパイラは保守的に解析するため、並列性を見切れないことがある。C言語ではほぼ必須。
  - リダクション処理

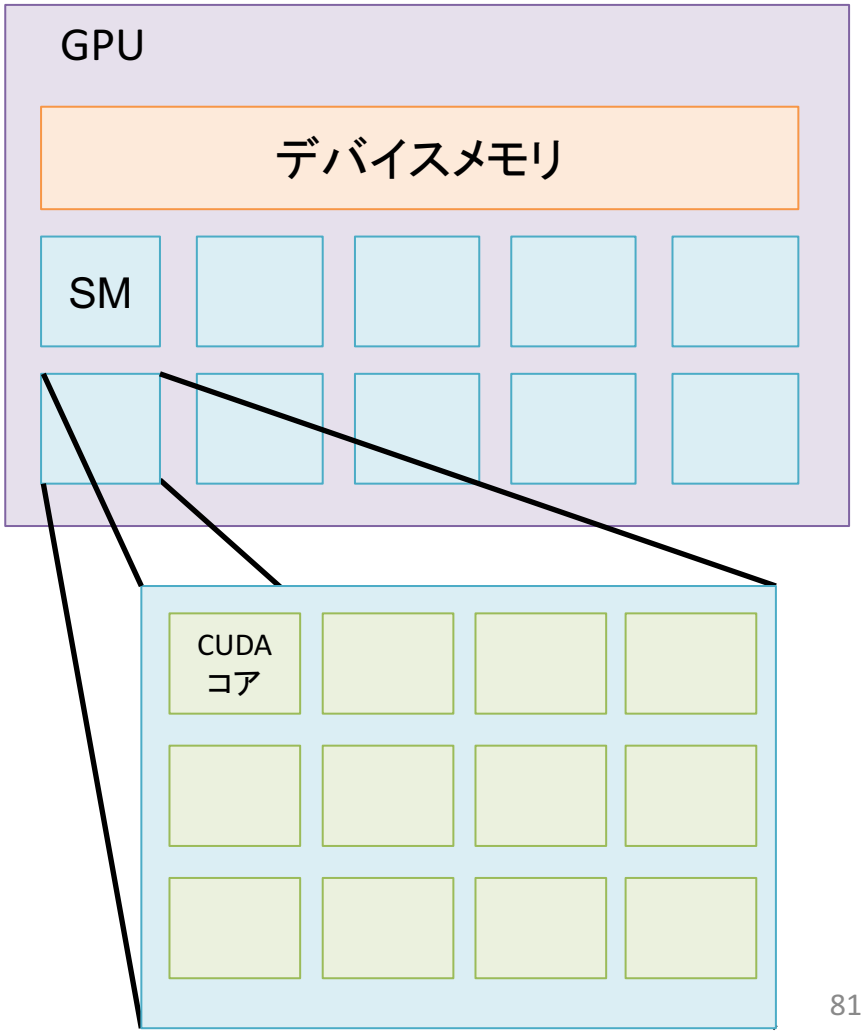
```
#pragma acc kernels
#pragma acc loop independent
   for (int i=0; i<n; i++) {
      b[i] = a[i] + c;
   }
```

配列の1要素が1スレッドで処理されるイメージ

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
  - gang：workerの集合 一番大きな単位
  - worker：vectorの集合
  - vector：スレッドに相当する一番小さい処理単位

- 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

ループがデータ独立であることを明示する
コンパイラが並列化できないと判断したときに使用

```
#pragma acc kernels
#pragma acc loop independent
   for (int i=0; i<n; i++) {
      b[i] = a[i] + c;
   }
```

並列化可能（データ独立）なので、independent を指定
（コンパイラは並列化可能とは判断してくれなかった）

■ データ独立でない（並列化可能でない）例

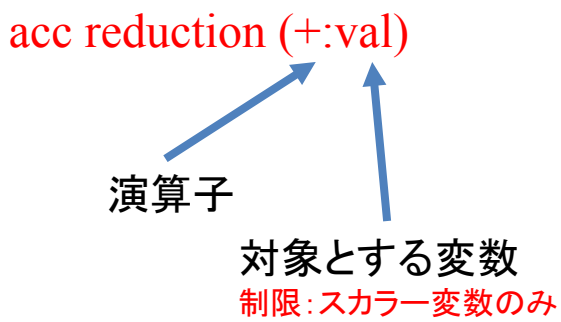
```
// これは正しくない

#pragma acc kernels
#pragma acc loop independent
   for (int i=1; i<n; i++) {
      d[i] = d[i-1];
   }
```

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)

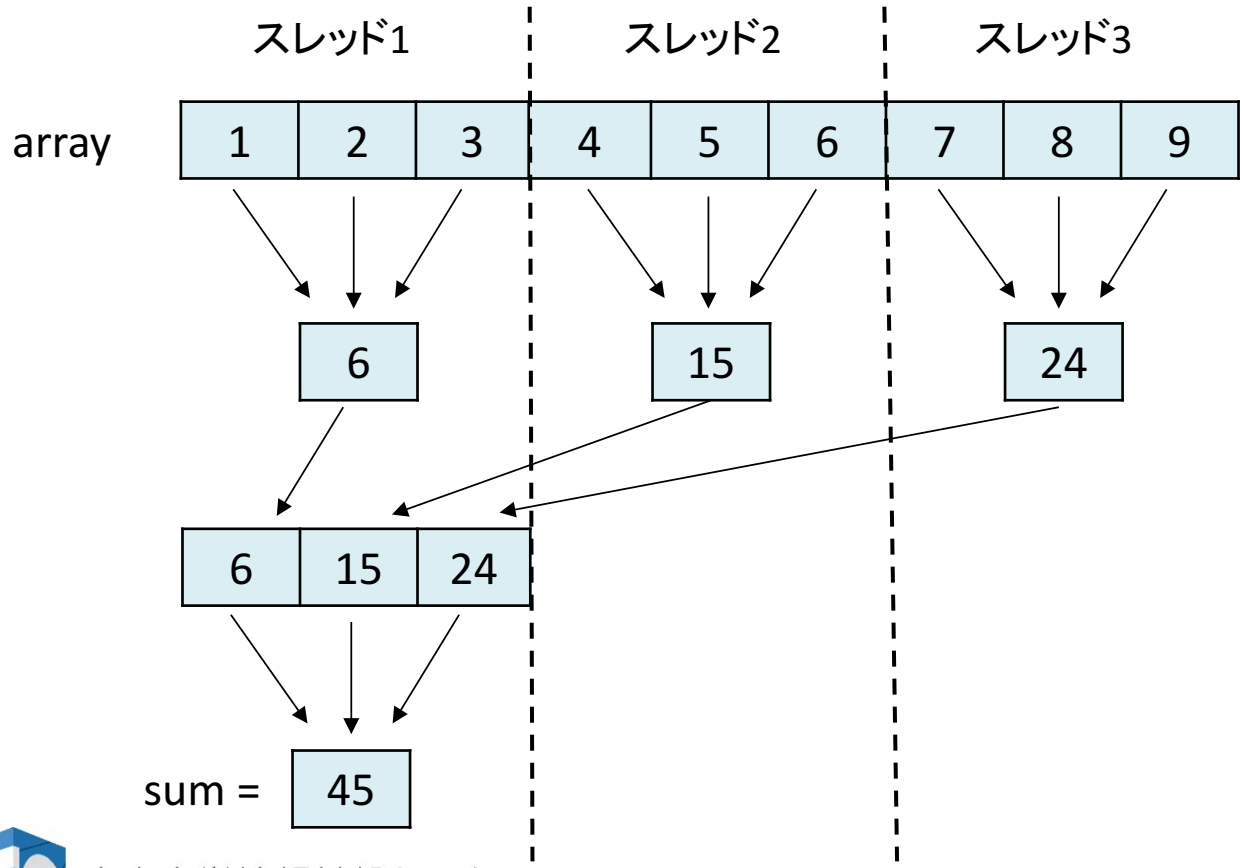
C and C++		Fortran	
operator	initialization value	operator	initialization value
+	0	+	0
*	1	*	1
max	least	max	least
min	largest	min	largest
&	~0	iand	all bits on
|	0	ior	0
^	0	ieor	0
&&	1	.and.	.true.
||	0	.or.	.false.
		.eqv.	.true.
		.neqv.	.false.

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[] ){
  ....
  double *tmp,*ans_d;
  cudaMalloc((void**)&tmp, sizeof(double) * 896);
  cudaMalloc((void**)&ans_d, sizeof(double) * 1);

  int chunk = (N+895)/896;
  dim3 dimGrid_L(896, 1, 1);
  dim3 dimBlock_L(128, 1, 1);
  dim3 dimGrid_G(1, 1, 1);
  dim3 dimBlock_G(1024, 1, 1);

  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);
  ....
}
```

ホスト側プログラム

一時配列の確保

使用するスレッド数の宣言

GPUカーネル呼び出し

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;
  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];
  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 == 0) tmp[bx] = sum + sum_tmp[1] + sum_tmp[2] + sum_tmp[3];
}
```

GPUカーネル1

Warp shuffle機能を使ったWarp内の縮約

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){
  __shared__ double sum_tmp[32];
  double sum;
  int tx = threadIdx.x;
  if(tx >= 896)
    sum = 0.0;
  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;
}
```

GPUカーネル2

一時配列も同様に縮約

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
  - 特にcaller, calleeでファイルが違う場合に必要とされる

プロトタイプ宣言にもつける

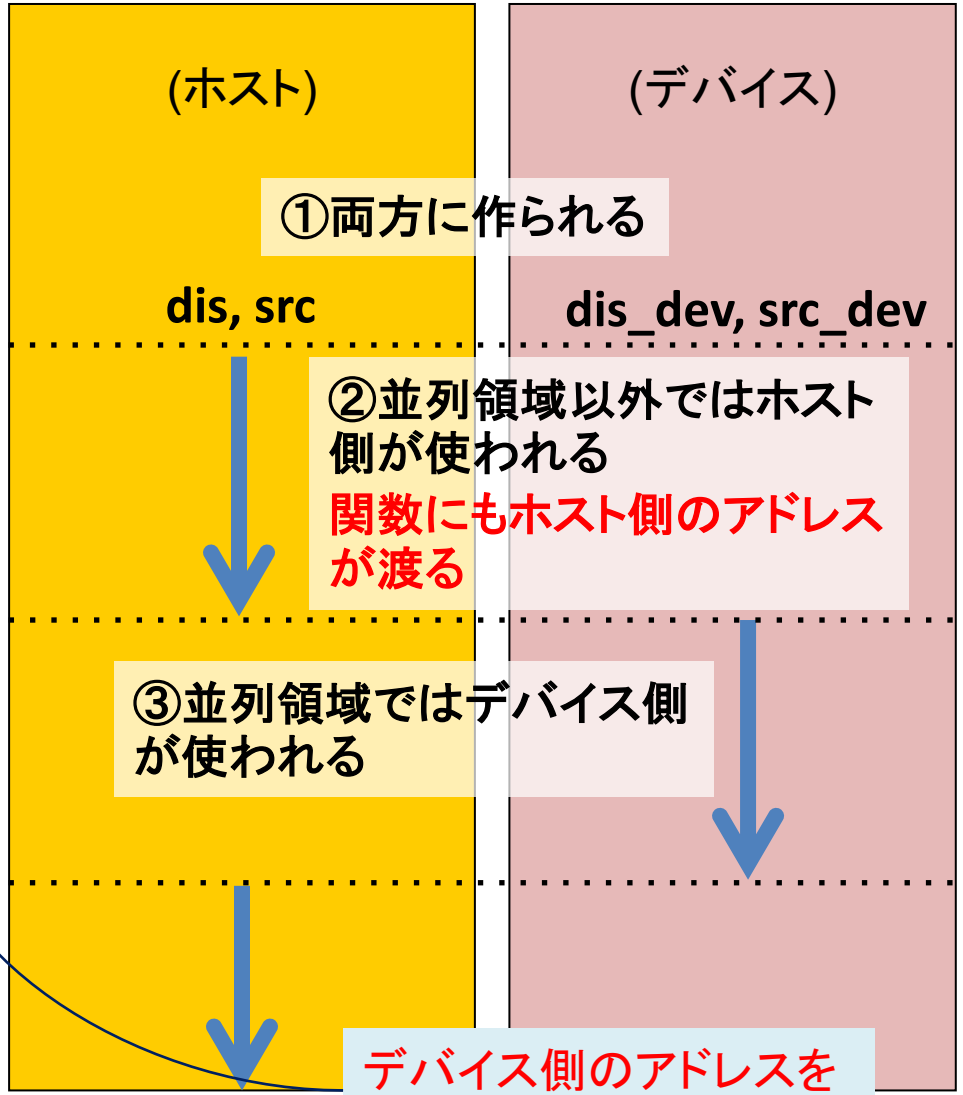
```
#pragma acc routine vector
extern double vecsum(double *A);
…
#pragma acc parallel num_gangs(N) vector_length(128)
for (int i = 0;i < N; i++){
   max = vecsum(A[i*N]);
}
```

```
#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、

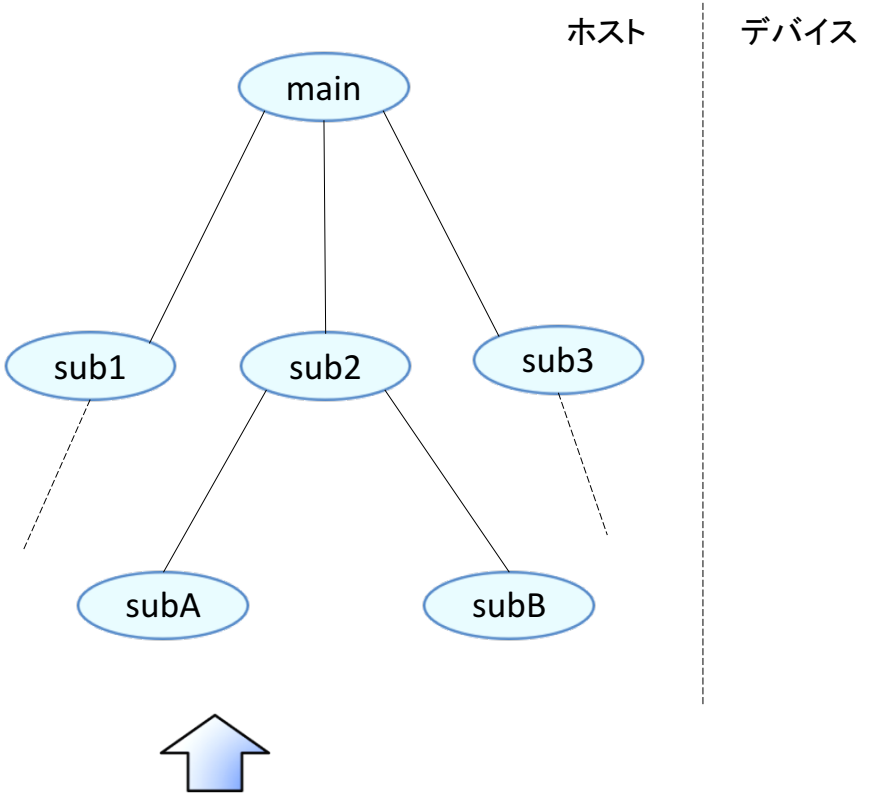
5. カーネルのCUDA化
   - スレッド間の相互作用が多いアプリケーションでは、shared memory や shuffle 命令を自由に使えるCUDAの方が圧倒的に有利

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

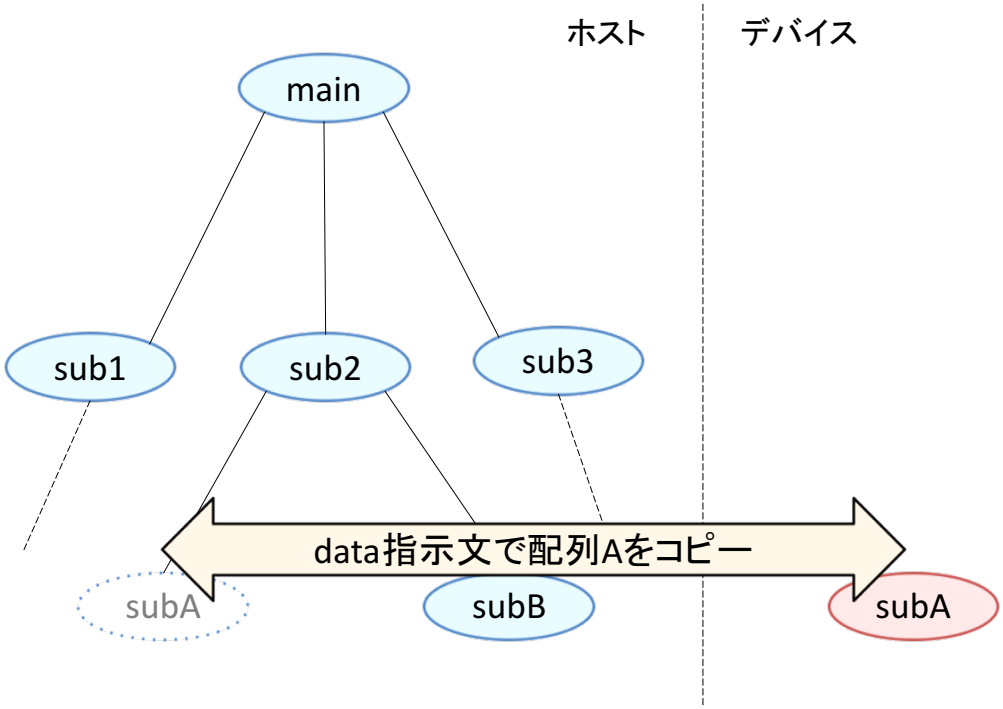
```
int main(){
  double A[N];
  sub1(A);
  sub2(A);
  sub3(A);
}

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

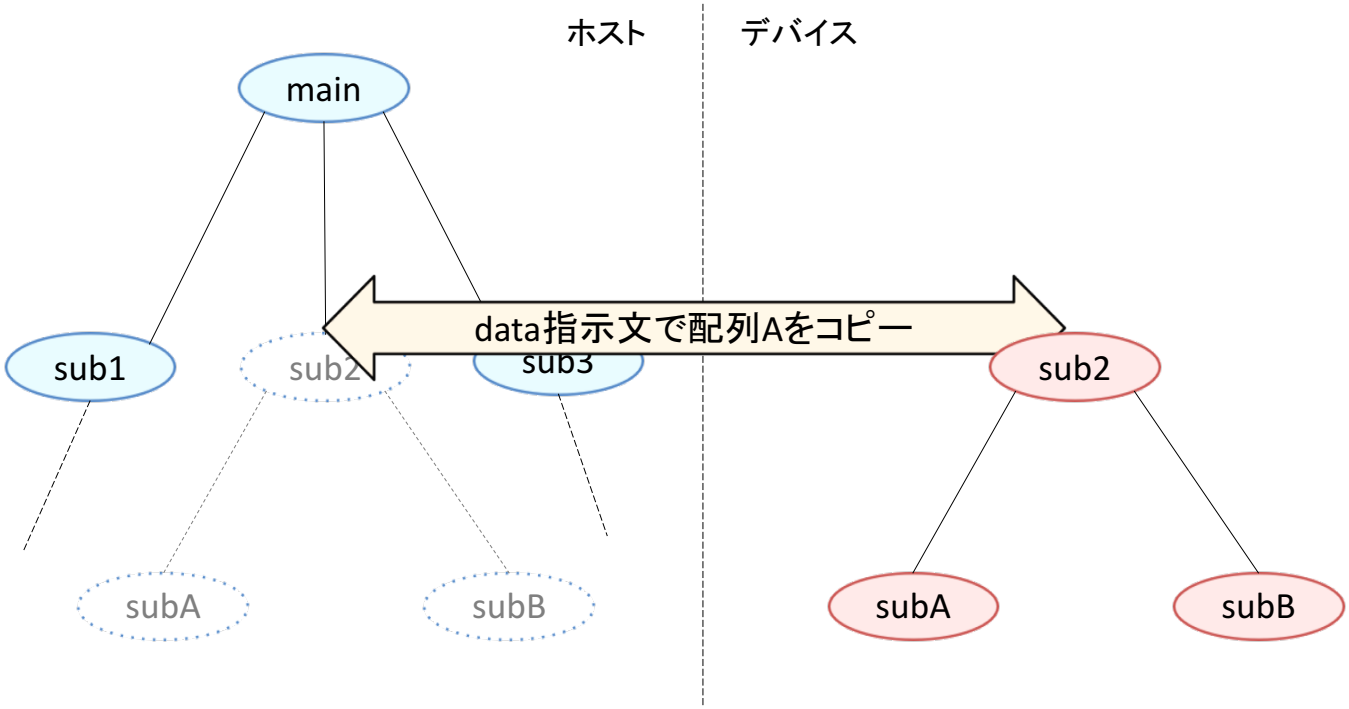
```
int main(){
  double A[N];
  sub1(A);
  sub2(A);
  sub3(A);
}

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

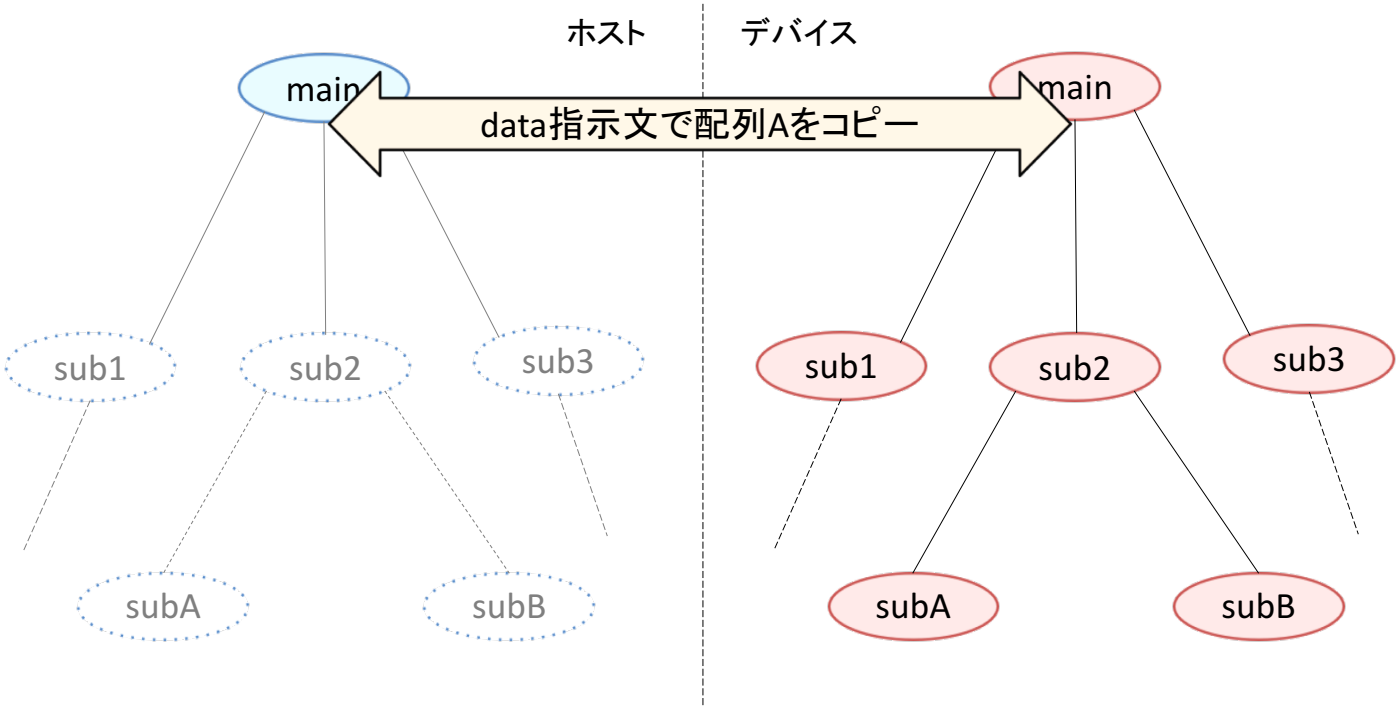
```
int main(){
  double A[N];
  sub1(A);
#pragma acc data
  {
    sub2(A);
  }
  sub3(A);
}

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

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

- 簡単なOpenACC プログラムの実行
  - コンパイラのメッセージの読み方を学ぼう
  - OpenACCのプログラムを実行しよう
- 3次元拡散方程式のOpenACC化
  - 自分でOpenACCの指示文を挿入してみよう
- FDTD法による電磁波伝搬計算
  - より実際のアプリケーションに近いプログラムをOpenACC化しよう

OpenACCコードのコンパイル

- PGIコンパイラによるコンパイル
  - ✓ ReedbushではOpenACCはPGIコンパイラで利用できます。

```
$ module load pgi/17.1
$ pgcc -O3 -acc -Minfo=accel -ta=tesla,cc60 -c main.c
```

-acc: OpenACCコードであることを指示

-Minfo=accel:

OpenACC指示文からGPUコードが生成できたかどうか等のメッセージを出力する。このメッセージがOpenACC化では大きなヒントになる。

-ta=tesla,cc60:

ターゲット・アーキテクチャの指定。NVIDIA GPU Teslaをターゲットとし、compute capability 6.0 (cc60) のコードを生成する。

- Makefileでコンパイル

講習会のサンプルコードには Makefile がついているので、コンパイルするためには、単純に下記を実行すれば良い。

```
$ module load pgi/17.1
$ make
```

簡単なOpenACCコード

- サンプルコード: openacc_basic/
  - ✓ OpenACC指示文 kernels, data, loop を利用したコード
  - ✓ 計算内容は簡単な四則演算

- ソースコード

openacc_basic/01_original	CPUコード。
openacc_basic/02_kernels	OpenACCコード。上にkernels指示文を追加。
openacc_basic/03_loop	OpenACCコード。上にloop指示文を追加
openacc_basic/04_data	OpenACCコード。上にdata指示文を明示的に追加。
openacc_basic/05_present	OpenACCコード。上でpresent指示節を使用。
openacc_basic/06_reduction	OpenACCコード。上にreduction指示節を使用。

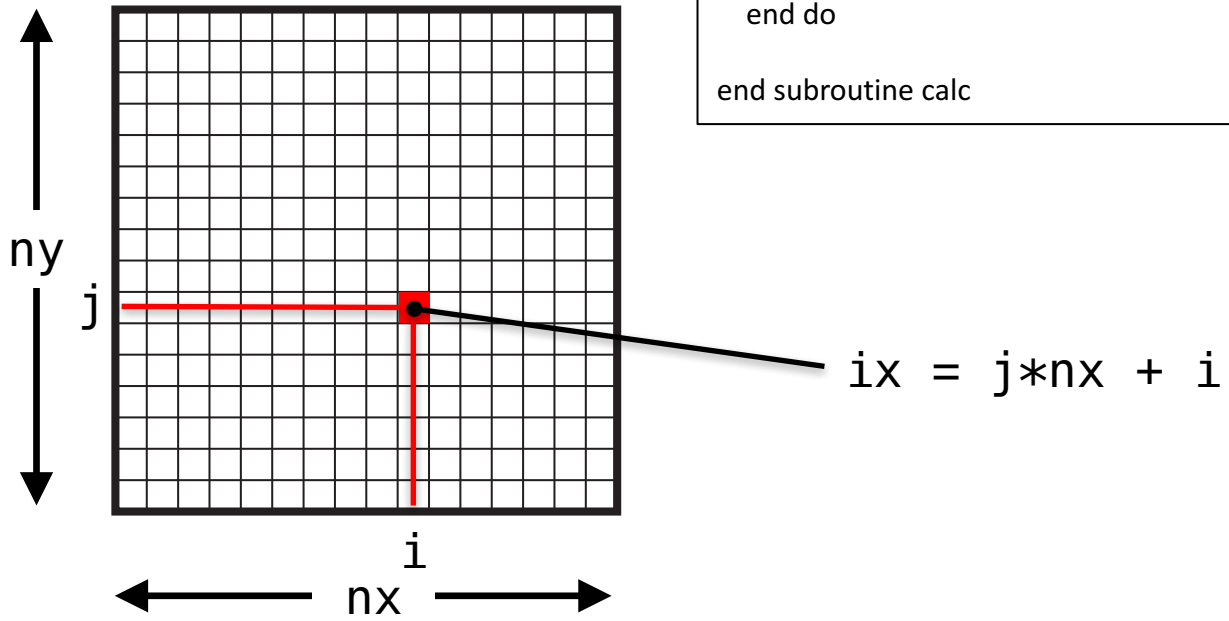
配列のインデックス計算

calc関数（C言語）

```
void calc(unsigned int nx, unsigned int ny, const float
*a, const float *b, float *c){
   for (unsigned int j=0; j<ny; j++) {
      for (unsigned int i=0; i<nx; i++) {
         const int ix = i + j*nx;
         c[ix] += a[ix] + b[ix];
      }
   }
}
```

calcサブルーチン（Fortran）

```
subroutine calc(nx, ny, a, b, c)
   integer,intent(in) :: nx,ny
   real,dimension(:),intent(in)  :: a,b
   real,dimension(:),intent(out) :: c
   integer :: i,j,ix

   do j = 1, ny
     do i = 1, nx
       ix = i + (j-1) * nx
       c(ix) = c(ix) + a(ix) + b(ix)
     end do
   end do

end subroutine calc
```



簡単なOpenACC: CPUコード

- CPUコードのコンパイルと実行
  - ✓ 配列の平均値と実行時間が出力されています。

```
$ cd openacc_basic/01_original
$ make
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =   19.886 [sec]
```

？の数字はジョブごとに変わります。

答えは常に3000.0

openacc_basic/01_original

- 計算内容
  - ✓ 配列 a、b、cをそれぞれ 1.0, 2.0, 0.0 で初期化
  - ✓ calc関数内で c += a *b を nt(=1000)回実行。
  - ✓ この実行時間を測定

簡単なOpenACC: kernels 指示文（1）

- 02_kernelsコード: calc関数

openacc_basic/02_kernels

  - ✓ CPUコードにkernels 指示文の追加

C言語

```
void calc(unsigned int nx, unsigned int ny, const float
*a, const float *b, float *c){
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])
   for (unsigned int j=0; j<ny; j++) {
      for (unsigned int i=0; i<nx; i++) {
         const int ix = i + j*nx;
         c[ix] += a[ix] + b[ix];
      }
   }
}
```

Fortran

```
subroutine calc(nx, ny, a, b, c)
   integer,intent(in) :: nx,ny
   real,dimension(:),intent(in)  :: a,b
   real,dimension(:),intent(out) :: c
   integer :: i,j,ix
!$acc kernels copy(a,b,c)          allocate, H -> D
   do j = 1, ny
      do i = 1, nx
         ix = i + (j-1) * nx
         c(ix) = c(ix) + a(ix) + b(ix)
      end do
   end do
!$acc end kernels                  D->H, deallocate
end subroutine calc
```

- ✓ kernels 指示文では data 指示文の機能が使える
- ✓ 上の場合は、copy を指定
  - ✓ カーネル前後でGPUとCPU間のメモリ転送が行われる。

簡単なOpenACC: kernels 指示文(2)

- 02_kernelsコード:初期化
  - ✓ CPUコードにkernels 指示文の追加

openacc_basic/02_kernels

C言語

```
int main(int argc, char *argv[])
{
...
#pragma acc kernels copyout(b[0:n], c[0:n])
  {
    for (unsigned int i=0; i<n; i++) {
      b[i] = b0;
    }
    for (unsigned int i=0; i<n; i++) {
      c[i] = 0.0;
    }
  }

...
}
```

b0 はスカラ変数のため自動的に各スレッドへコピーが渡される。

Fortran

```
program main
...
 !$acc kernels copyout(b,c)
 do i = 1, n
   b(i) = b0
 end do

 do i = 1, n
   c(i) = 0.0
 end do
 !$acc end kernels
...
end program main
```

allocate

D->H, deallocate

Fortranは配列のサイズ情報を持つので、大抵の場合サイズ指定はいらない。

簡単なOpenACC: kernels 指示文(3)

- コンパイル
  - ✓ データの独立性がコンパイラにはわからず、並列化されない。

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
calc:
     13, Generating copy(a[:n],c[:n],b[:n])
     14, Complex loop carried dependence of a-> prevents parallelization
         Loop carried dependence due to exposed use of c[:n] prevents parallelization
         Complex loop carried dependence of c->,b-> prevents parallelization
         Accelerator scalar kernel generated
         Accelerator kernel generated
         Generating Tesla code
         14, #pragma acc loop seq
         15, #pragma acc loop seq
     15, Complex loop carried dependence of a->,c->,b-> prevents parallelization
         Loop carried dependence due to exposed use of c[:i1+n] prevents parallelization
main:
     43, Generating copyout(c[:n],b[:n])
     45, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         45, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     48, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         48, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60   main.o   -o run
```

簡単なOpenACC: loop 指示文（1）

- 03_loopコード
  - ✓ 02_kernelsコードにloop independent の追加

openacc_basic/03_loop

```
void calc(unsigned int nx, unsigned int ny, const float *a,
const float *b, float *c){
   const unsigned int n = nx * ny;
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])
#pragma acc loop independent
   for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
      for (unsigned int i=0; i<nx; i++) {
         const int ix = i + j*nx;
         c[ix] += a[ix] + b[ix];
      }
   }
}
```

```
// main 関数内
#pragma acc kernels copyout(b[0:n], c[0:n])
   {
#pragma acc loop independent
      for (unsigned int i=0; i<n; i++) {
         b[i] = b0;
      }
#pragma acc loop independent
      for (unsigned int i=0; i<n; i++) {
         c[i] = 0.0;
      }
   }
```

```
subroutine calc(nx, ny, a, b, c)
   integer,intent(in) :: nx,ny
   real,dimension(:),intent(in)  :: a,b
   real,dimension(:),intent(out) :: c
   integer :: i,j,ix
!$acc kernels copy(a,b,c)
!$acc loop independent
   do j = 1, ny
!$acc loop independent
      do i = 1, nx
         ix = i + (j-1) * nx
         c(ix) = c(ix) + a(ix) + b(ix)
      end do
   end do
!$acc end kernels
end subroutine calc
```

```
!$acc kernels copyout(b,c)
!$acc loop independent
 do i = 1, n
   b(i) = b0
 end do
!$acc loop independent
 do i = 1, n
   c(i) = 0.0
 end do
!$acc end kernels
```

簡単なOpenACC: loop 指示文（2）

- コンパイル
  - ✓ ループが並列化され、カーネルが生成された。

openacc_basic/03_loop

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
calc:
     13, Generating copy(a[:n],c[:n],b[:n])
     15, Loop is parallelizable
     17, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         15, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         17, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
main:
     45, Generating copyout(c[:n],b[:n])
     48, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         48, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     52, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         52, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

簡単なOpenACC: loop 指示文（3）

- 03_loopコードの実行

openacc_basic/03_loop

  - ✓ 答えは正しいが、実行時間が大変長い。

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =   70.414 [sec]
```

  - ✓ ソースコードをみると、calc関数でカーネル前後にGPUとCPU間のデータ転送が発生する。この関数はmain関数から1000回呼ばれる。これが性能低下させている。

```
// main 関数内
 const unsigned int nt = 1000;
 …
 for (unsigned int icnt=0; icnt<nt; icnt++) {
      calc(nx, ny, a, b, c);
   }
```

1000回呼ばれる

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b,
float *c){
   const unsigned int n = nx * ny;
#pragma acc kernels copy(a[0:n], b[0:n], c[0:n])
#pragma acc loop independent
   for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
      for (unsigned int i=0; i<nx; i++) {
         const int ix = i + j*nx;
         c[ix] += a[ix] + b[ix];
      }
   }
}
```

allocate, H -> D

D->H, deallocate

簡単なOpenACC: data指示文（1）

- 04_dataコード

  ✓ 03_loopにdata指示文追加

openacc_basic/04_data

```
// main関数内
#pragma acc data copyin(a[0:n]) create(b[0:n]) copyout(c[0:n])
  {
#pragma acc kernels copyout(b[0:n], c[0:n])
    {
#pragma acc loop independent
      for (unsigned int i=0; i<n; i++) {
        b[i] = b0;
      }
#pragma acc loop independent
      for (unsigned int i=0; i<n; i++) {
        c[i] = 0.0;
      }
    }

    for (unsigned int icnt=0; icnt<nt; icnt++) {
      calc(nx, ny, a, b, c);
    }
  }
```

present として振舞う。

```
! mainプログラム内
!$acc data copyin(a) create(b) copyout(c)
!$acc kernels copyout(b,c)
!$acc loop independent
 do i = 1, n
   b(i) = b0
 end do
!$acc loop independent
 do i = 1, n
   c(i) = 0.0
 end do
!$acc end kernels

 do icnt = 1, nt
   call calc(nx, ny, a, b, c)
 end do
!$acc end data
```

a: allocate, H -> D
b: allocate
c: allocate

a: deallocate
b: deallocate
c: D->H, deallocate

  ✓ copy/copyin/copyout/create は既にデバイス上確保されているデータに対しては何もしない。present として振舞う。（OpenACC2.5以降）

  ✓ 配列 a, b, c は利用用途に合わせた指示節を指定。

簡単なOpenACC: data指示文（2）

openacc_basic/04_data

- 04_dataコードの実行
  - ✓ 答えは正しく、速度が上がった。

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =    1.174 [sec]
```

簡単なOpenACC: present指示節

- 05_presentコード
  - ✓ 04_dataコードで present 指示節を使用

openacc_basic/05_present

```
void calc(unsigned int nx, unsigned int ny, const float *a, const float *b, float *c){
   const unsigned int n = nx * ny;
#pragma acc kernels present(a, b, c)
#pragma acc loop independent
   for (unsigned int j=0; j<ny; j++) {
#pragma acc loop independent
      for (unsigned int i=0; i<nx; i++) {
         const int ix = i + j*nx;
         c[ix] += a[ix] + b[ix];
      }
```

present へ変更

```
// main 関数内
#pragma acc kernels
   {
#pragma acc loop independent
      for (unsigned int i=0; i<n; i++) {
         b[i] = b0;
      }
```

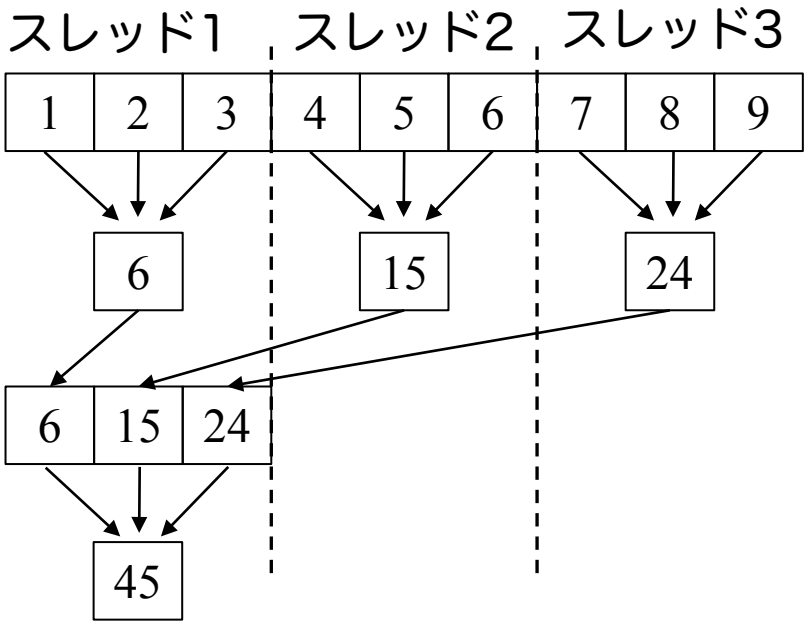
指示節を削除

- ✓ データ転送の振る舞いは変化しないため、性能変化はなし。
- ✓ present ではメモリ確保、データ転送をしないため、配列サイズの指定は不要。
- ✓ コードとしては見通しがよい。

リダクション計算（1）

- リダクション計算
  - ✓ 配列の全要素から一つの値を抽出
  - ✓ 総和、総積、最大値、最小値など
  - ✓ 出力が一つのため、並列化に工夫が必要（CUDAでの実装は煩雑）

```
double sum = 0.0;
for (unsigned int i=0; i<n; i++) {
   sum += array[i];
}
```



1. 各スレッドが担当する領域をリダクション
2. 一時配列に移動
3. 一時配列をリダクション
4. 出力を得る

リダクション計算(2)

- loop 指示文に reduction 指示節を指定
  - ✓ reduction 演算子と変数を組み合わせて指定

```
double sum = 0.0;
#pragma acc kernels
#pragma acc loop reduction(+:sum)
for (unsigned int i=0; i<n; i++) {
   sum += array[i];
}
```

- Reduction 指示節
  - ✓ **acc loop reduction(+:sum)**
    - ✓ 演算子と対象とする変数(スカラー変数)を指定する。
- 利用できる主な演算子と初期値
  - ✓ 演算子: +, 初期値: 0
  - ✓ 演算子: *, 初期値: 1
  - ✓ 演算子: max, 初期値: least
  - ✓ 演算子: min, 初期値: largest

簡単なOpenACC: reduction指示節（1）

- 06_reductionコード
  - ✓ 05_presentコードで reductionを使用

openacc_basic/06_reduction

```
// main 関数内
      for (unsigned int icnt=0; icnt<nt; icnt++) {
         calc(nx, ny, a, b, c);
      }

#pragma acc kernels
#pragma acc loop reduction(+:sum)
      for (unsigned int i=0; i<n; i++) {
         sum += c[i];
      }
```

```
do icnt = 1, nt
   call calc(nx, ny, a, b, c)
end do

sum = 0
!$acc kernels
!$acc loop reduction(+:sum)
do i = 1, n
   sum = sum + c(i)
end do
!$acc end kernels
!$acc end data
```

  - ✓ data 指示文で c を create に変更。
  - ✓ data指示文の範囲に注意。
  - ✓ リダクションコードが生成された。

```
$ make
pgcc -O3 -acc -Minfo=accel  -ta=tesla,cc60  -c main.c
(省略)
main:
(省略)
67, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         67, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:sum)
```

簡単なOpenACC: reduction指示節（2）

openacc_basic/06_reduction

- 06_reductionコードの実行
  - ✓ 答えは正しく、速度が上がった。
  - ✓ 配列 c の転送が削減されたこと、リダクションがGPU上で行われることによる性能向上。

```
$ qsub ./run.sh
$ cat run.sh.o??????
mean = 3000.00
Time =    1.089 [sec]
```

OpenACC化のステップのまとめ

- OpenACC化のための3つの指示文の適用
  - ✓ kernels 指示文を用いてGPUで実行する領域を指定
  - ✓ data 指示文を用い、ホスト-デバイス 間の通信を最適化
  - ✓ loop 指示文を用い、並列処理の指定

```
#pragma acc data copyin(a[0:n]) create(b[0:n], c[0:n])
  {

#pragma acc kernels
    {
#pragma acc loop independent
      for (unsigned int i=0; i<n; i++) {
        b[i] = b0;
      }
#pragma acc loop independent
      for (unsigned int i=0; i<n; i++) {
        c[i] = 0.0;
      }
    }

    for (unsigned int icnt=0; icnt<nt; icnt++) {
      calc(nx, ny, a, b, c);
    }

#pragma acc kernels
#pragma acc loop reduction(+:sum)
    for (unsigned int i=0; i<n; i++) {
      sum += c[i];
    }
  }
```

```
!$acc data copyin(a) create(b) copyout(c)
!$acc kernels
!$acc loop independent
do i = 1, n
  b(i) = b0
end do
!$acc loop independent
do i = 1, n
  c(i) = 0.0
end do
!$acc end kernels

do icnt = 1, nt
  call calc(nx, ny, a, b, c)
end do

!$acc kernels
!$acc loop reduction(+:sum)
do i = 1, n
  sum = sum + c(i)
end do
!$acc end kernels
!$acc end data
```

openacc_basic/06_reduction

3次元拡散方程式のOpenACC化

- 3次元拡散方程式のCPUコードにOpenACC の kernels, data, loop 指示文を追加し、GPUで高性能で実行しましょう。
  - サンプルコード: openacc_diffusion/01_original

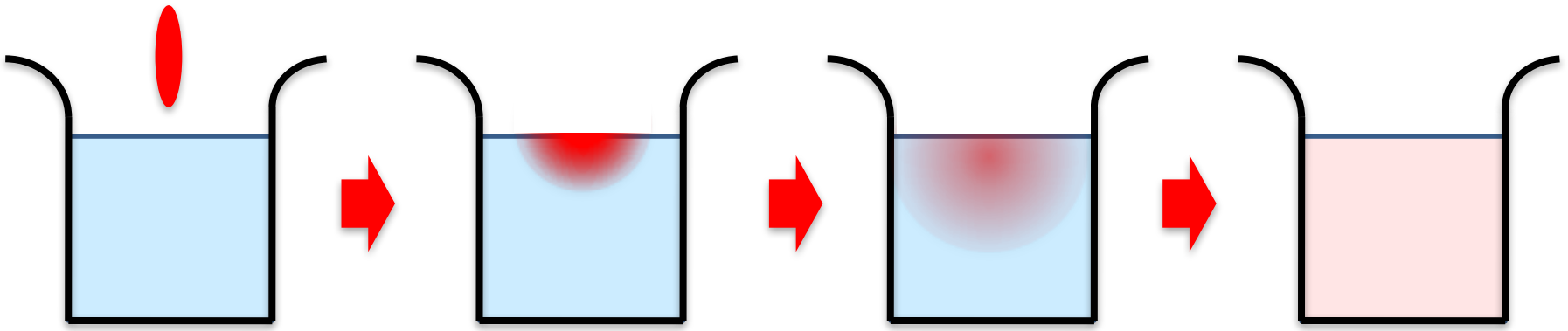
diffusion.c, diffusion3d 関数内

```
for(int k = 0; k < nz; k++) {
    for (int j = 0; j < ny; j++) {
        for (int i = 0; i < nx; i++) {
            const int ix = nx*ny*k + nx*j + i;
            const int ip = i == nx - 1 ? ix : ix + 1;
            const int im = i == 0      ? ix : ix - 1;
            const int jp = j == ny - 1 ? ix : ix + nx;
            const int jm = j == 0      ? ix : ix - nx;
            const int kp = k == nz - 1 ? ix : ix + nx*ny;
            const int km = k == 0      ? ix : ix - nx*ny;

            fn[ix] = cc*f[ix]
                   + ce*f[ip] + cw*f[im]
                   + cn*f[jp] + cs*f[jm]
                   + ct*f[kp] + cb*f[km];
        }
    }
}
```

openacc_diffusion/01_original

拡散現象シミュレーション（1）

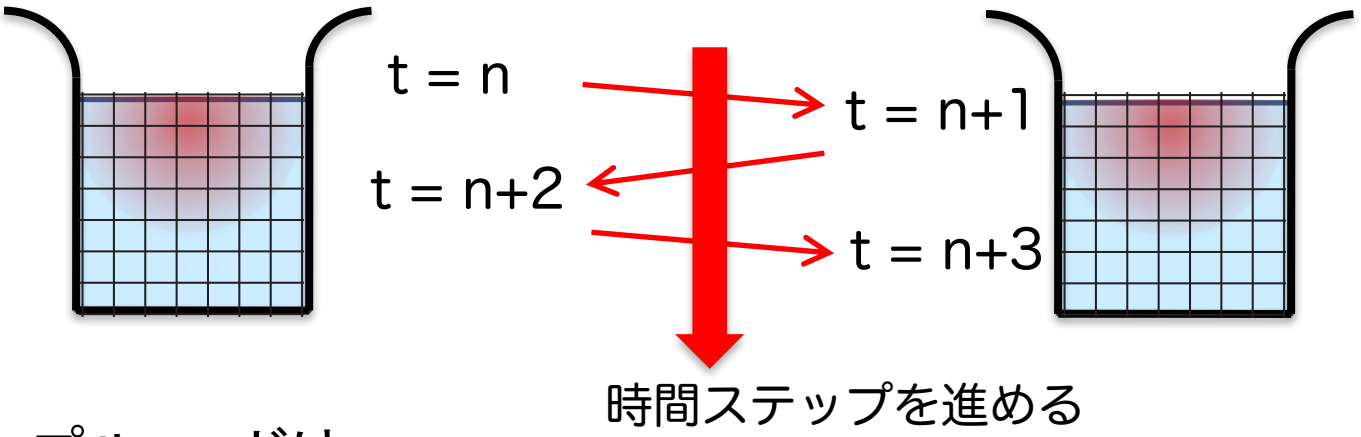
- 拡散現象
  - ✓ コップの中に赤インクを落とすと水中で拡がる
  - ✓ 次第に拡散し赤インクは拡がり、最後は均一な色になる。



- 拡散方程式のシミュレーション
  - ✓ 各点のインク濃度の時間変化を計算する

拡散現象シミュレーション（2）

- データ構造
  - ✓ 計算したい空間を格子に区切り、一般に配列で表す。
  - ✓ 計算は3次元であるが、C言語では1次元配列として確保することが一般的。
  - ✓ 2ステップ分の配列を使い、タイムステップを進める（ダブルバッファ）。



- サンプルコードは、
  - ✓ 計算領域: nx * ny * nz（3次元）
  - ✓ 最大タイムステップ: nt

  となっている。

拡散現象シミュレーション（3）

- 2次元拡散方程式の離散化の一例

$$f_{i,j}^{n+1} = (f_{i-1,j}^{n} + f_{i+1,j}^{n} + f_{i,j-1}^{n} + f_{i,j+1}^{n} + 4f_{i,j}^{n})/8$$

平均後の
自分自身の値

上下左右の値

自分自身の値の4倍

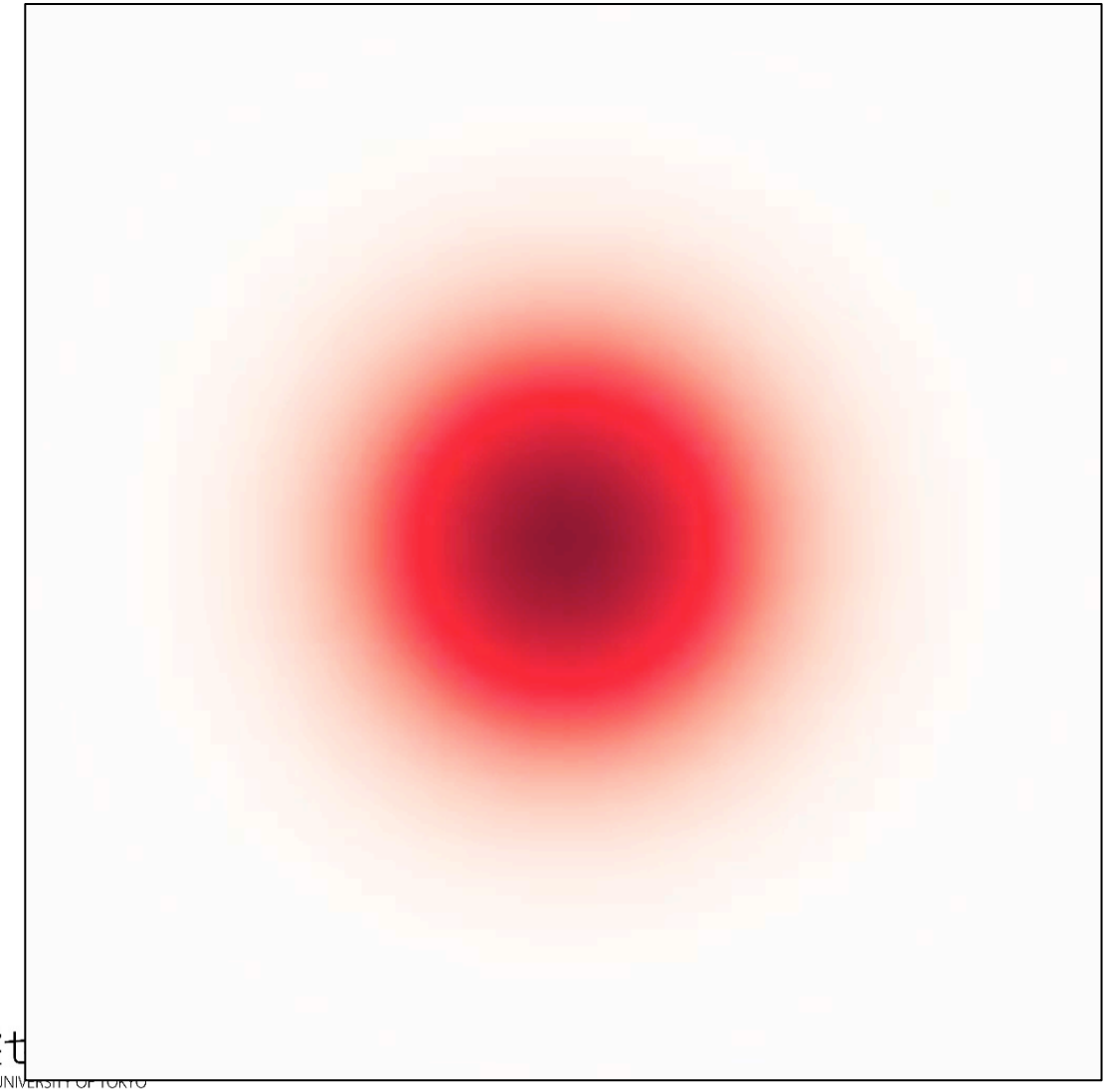
j \ i	1	2	3	4	5
1	0	0	1	0	0
2	0	2	8	2	0
3	1	8	20	8	1
4	0	2	8	2	0
5	0	0	1	0	0

2回目の平均後

繰り返し平均化を行うと、インクが拡散します。

拡散現象シミュレーション（4）

- 2次元拡散方程式の計算例



CPUコード

- CPUコードのコンパイルと実行

```
$ cd openacc_diffusion/01_original
$ make
$ qsub ./run.sh
# cat run.sh.o??????
time(   0) = 0.00000
time( 100) = 0.00610
time( 200) = 0.01221
...
time(1000) = 0.06104
time(1100) = 0.06714
time(1200) = 0.07324
time(1300) = 0.07935
time(1400) = 0.08545
time(1500) = 0.09155
time(1600) = 0.09766
Time =   20.564 [sec]
Performance=    2.17 [GFlops]
Error[128][128][128] = 4.556413e-06
```

- OpenACCコードでは、どのくらいの実行性能が達成できるでしょうか？

OpenACC化(0): Makefile の修正

- Makefile に OpenACC をコンパイルするよう –acc などを追加しましょう

```
CC   = pgcc
CXX  = pgc++
GCC  = gcc
RM   = rm -f
MAKEDEPEND = makedepend

CFLAGS    = -O3 -acc -Minfo=accel -ta=tesla,cc60
GFLAGS    = -Wall -O3 -std=c99
CXXFLAGS  = $(CFLAGS)
LDFLAGS   =
...
```

OpenACC化(1): kernels

- diffusion3d関数に kernelsを追加しましょう

```
#pragma acc kernels copyin(f[0:nx*ny*nz]) copyout(fn[0:nx*ny*nz])
    for(int k = 0; k < nz; k++) {
        for (int j = 0; j < ny; j++) {
            for (int i = 0; i < nx; i++) {
                const int ix = nx*ny*k + nx*j + i;
                const int ip = i == nx - 1 ? ix : ix + 1;
                const int im = i == 0      ? ix : ix - 1;
                const int jp = j == ny - 1 ? ix : ix + nx;
                const int jm = j == 0      ? ix : ix - nx;
                const int kp = k == nz - 1 ? ix : ix + nx*ny;
                const int km = k == 0      ? ix : ix - nx*ny;

                fn[ix] = cc*f[ix]
                       + ce*f[ip] + cw*f[im]
                       + cn*f[jp] + cs*f[jm]
                       + ct*f[kp] + cb*f[km];
            }
        }
    }

    return (double)(nx*ny*nz)*13.0;
}
```

diffusion.c, diffusion3d 関数内

make して実行してみましょう。

OpenACC化(2): loop

- diffusion3d関数に loopを追加しましょう

```
#pragma acc kernels copyin(f[0:nx*ny*nz]) copyout(fn[0:nx*ny*nz])
#pragma acc loop independent
    for(int k = 0; k < nz; k++) {
#pragma acc loop independent
        for (int j = 0; j < ny; j++) {
#pragma acc loop independent
            for (int i = 0; i < nx; i++) {
                const int ix = nx*ny*k + nx*j + i;
                const int ip = i == nx - 1 ? ix : ix + 1;
                const int im = i == 0      ? ix : ix - 1;
                const int jp = j == ny - 1 ? ix : ix + nx;
                const int jm = j == 0      ? ix : ix - nx;
                const int kp = k == nz - 1 ? ix : ix + nx*ny;
                const int km = k == 0      ? ix : ix - nx*ny;

                fn[ix] = cc*f[ix]
                       + ce*f[ip] + cw*f[im]
                       + cn*f[jp] + cs*f[jm]
                       + ct*f[kp] + cb*f[km];
            }
        }
    }

    return (double)(nx*ny*nz)*13.0;
}
```

diffusion.c, diffusion3d 関数内

高速化よりも、まずは正しい計算を行うコードを保つことが大事です。
末端の関数から修正を進めます。

make してジョブ投入 qsub ./run.shしてみましょう。
遅いですが実行できます。

OpenACC化(3): データ転送の最適化(1)

- diffusion3d関数で present とし、main関数で data を追加

diffusion.c, diffusion3d 関数内

```
#pragma acc kernels present(f, fn)
#pragma acc loop independent
    for(int k = 0; k < nz; k++) {
#pragma acc loop independent
        for (int j = 0; j < ny; j++) {
#pragma acc loop independent
            for (int i = 0; i < nx; i++) {
                const int ix = nx*ny*k + nx*j + i;
                const int ip = i == nx - 1 ? ix : ix + 1;
                const int im = i == 0      ? ix : ix - 1;
                const int jp = j == ny - 1 ? ix : ix + nx;
                const int jm = j == 0      ? ix : ix - nx;
                const int kp = k == nz - 1 ? ix : ix + nx*ny;
                const int km = k == 0      ? ix : ix - nx*ny;

                fn[ix] = cc*f[ix]
                       + ce*f[ip] + cw*f[im]
                       + cn*f[jp] + cs*f[jm]
                       + ct*f[kp] + cb*f[km];
            }
        }
    }

    return (double)(nx*ny*nz)*13.0;
}
```

なお、 present にしなくても期待通りに動作します。

OpenACC化(4): データ転送の最適化(2)

- diffusion3d関数で present とし、main関数で data を追加

main.c, main 関数内

```
#pragma acc data copy(f[0:n]) create(fn[0:n])
    {
        start_timer();

        for (; icnt<nt && time + 0.5*dt < 0.1; icnt++) {
            if (icnt % 100 == 0)
                fprintf(stdout, "time(%4d) = %7.5f¥n", icnt, time);

            flop += diffusion3d(nx, ny, nz, dx, dy, dz, dt, kappa, f, fn);

            swap(&f, &fn);

            time += dt;
        }

        elapsed_time = get_elapsed_time();
    }
```

copy/create など適切なものを選びます。

make して実行してみましょう。 どのくらいの実行性能が出ましたか？

OpenACC化の例は、 openacc_diffusion/02_openacc

PGI_ACC_TIME によるOpenACC 実行の確認

- PGIコンパイラを利用する場合、OpenACCプログラムがどのように実行されているか、環境変数PGI_ACC_TIMEを設定すると簡単に確認することができる。
- Linuxなどでは、環境変数PGI_ACC_TIME を1に設定し、プログラムを実行する。

```
$ export PGI_ACC_TIME=1
$ ./run
```

- Reedbush でジョブに環境変数PGI_ACC_TIME を設定する場合は、ジョブスクリプト中に記載する。

```
$ cat run.sh
...
 . /etc/profile.d/modules.sh
module load pgi/17.1
export PGI_ACC_TIME=1

./run
```

サンプルコードは、 openacc_diffusion/03_openacc _pgi_acc_time

PGI_ACC_TIME によるOpenACC 実行の確認

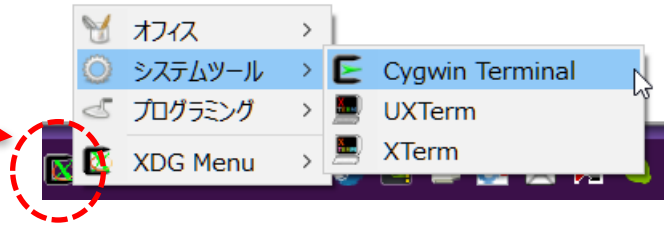
- ジョブ実行が終わると、標準エラー出力にメッセージが出力される。

```
$ cat run.sh.e??????
Accelerator Kernel Timing data
/lustre/pz0115/z30115/lecture/lecture_samples/openacc_diffusion/03_open
acc_pgi_acc_time/main.c
  main  NVIDIA  devicenum=0
    time(us): 6,359
    38: data region reached 2 times
        38: data copyin transfers: 1
             device time(us): total=3,327 max=3,327 min=3,327 avg=3,327
        55: data copyout transfers: 1
             device time(us): total=3,032 max=3,032 min=3,032 avg=3,032
/lustre/pz0115/z30115/lecture/lecture_samples/openacc_diffusion/03_open
acc_pgi_acc_time/diffusion.c
  diffusion3d  NVIDIA  devicenum=0
    time(us): 101,731
    19: compute region reached 1638 times
        25: kernel launched 1638 times
            grid: [4x128x32]  block: [32x4]
             device time(us): total=101,731 max=64 min=62 avg=62
            elapsed time(us): total=136,255 max=540 min=81 avg=83
    19: data region reached 3276 times
```

参考：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



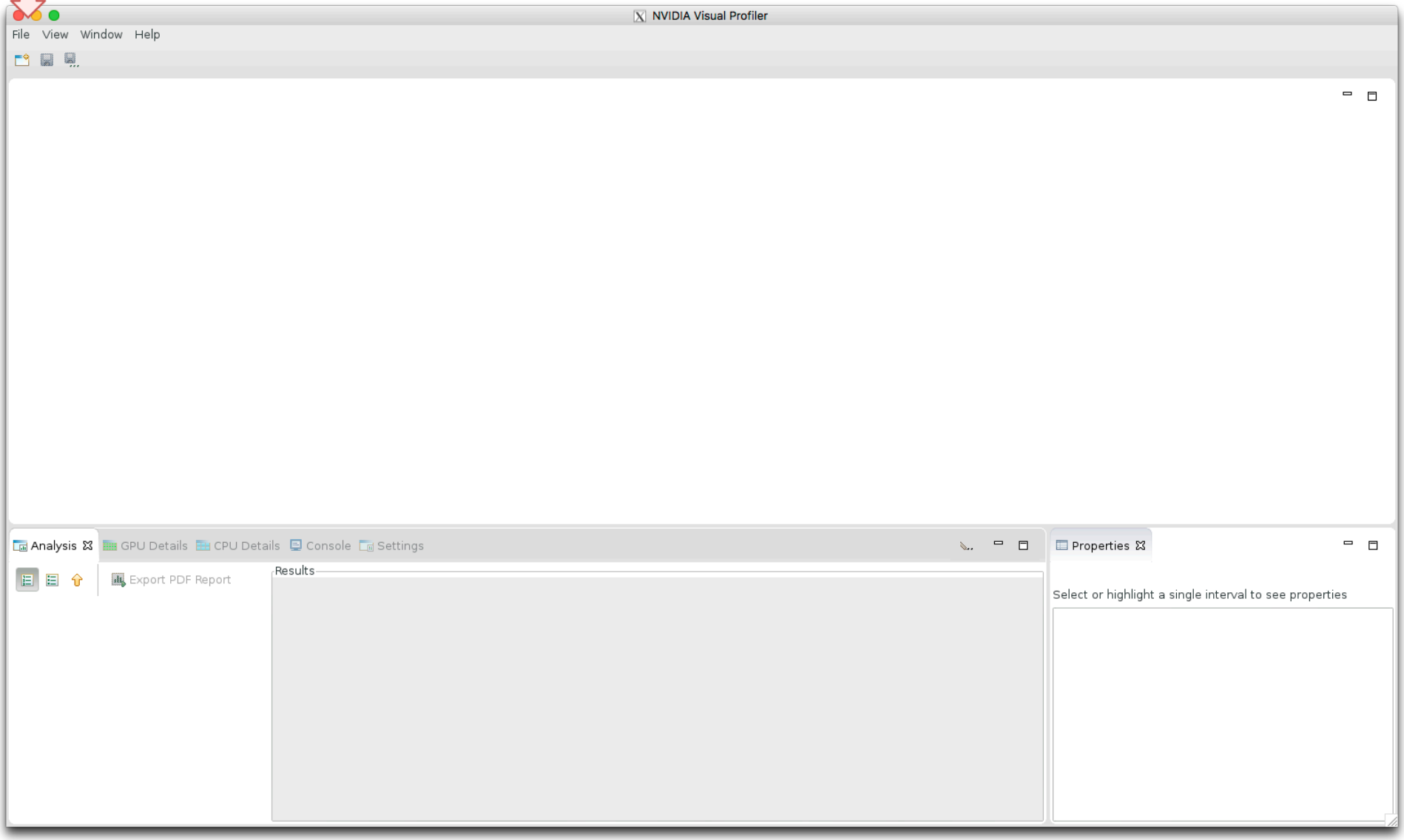
- 使用
  - Reedbushへssh接続する際に-Yオプションを付けておく
    - ssh -Y reedbush.cc.u-tokyo.ac.jp -l txxxxx
  - module設定を行い、nvvpを起動する
    - module load cudaを設定したあとでnvvpコマンドを実行

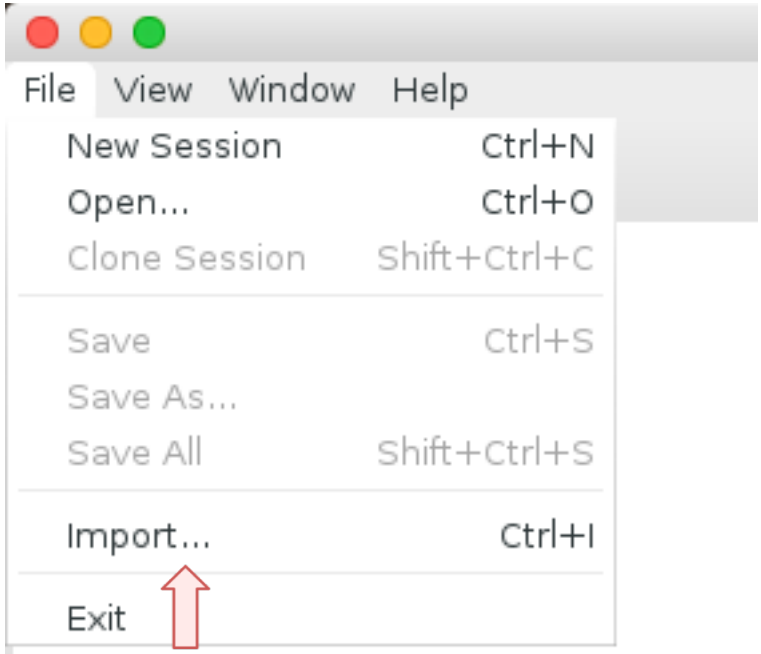
参考：Reedbush での NVVP の使い方

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…

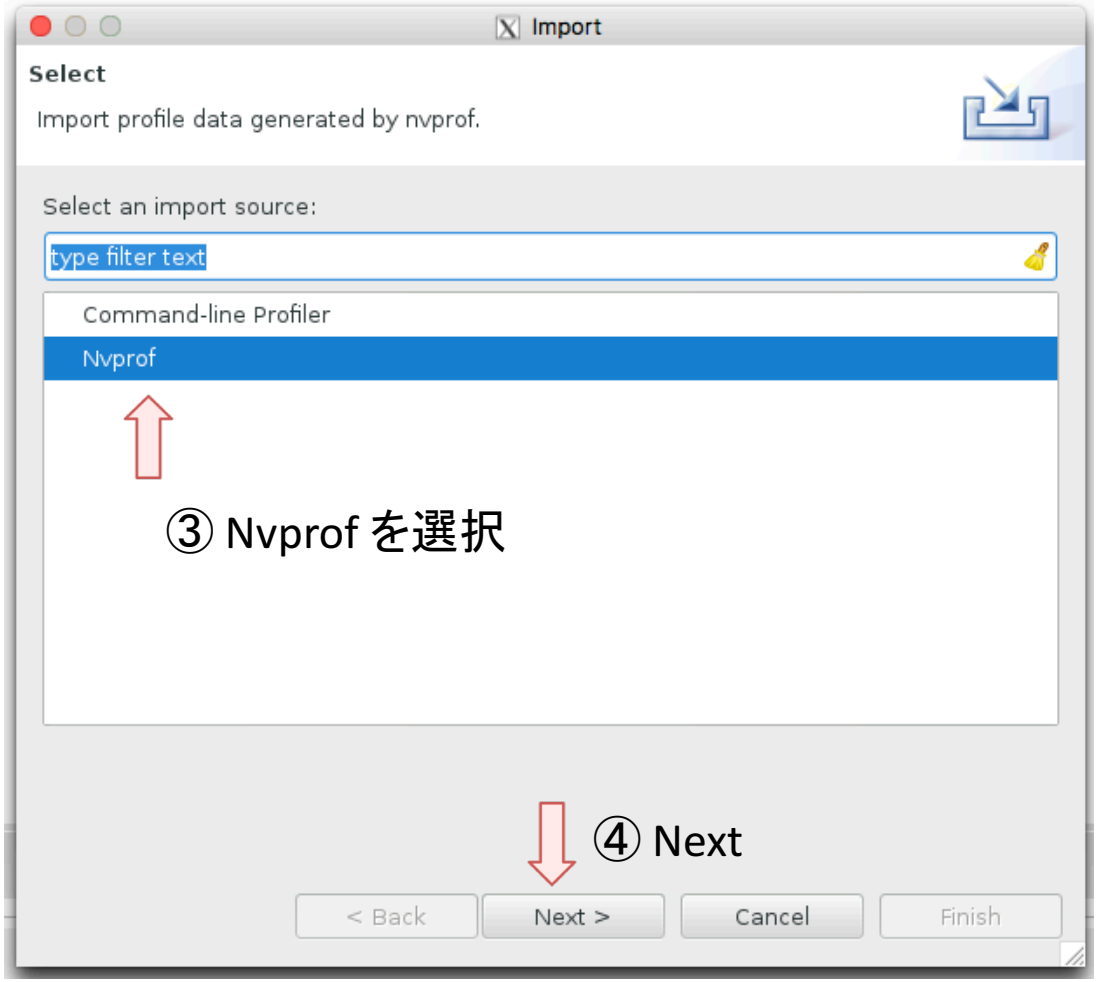
①Fileをクリック

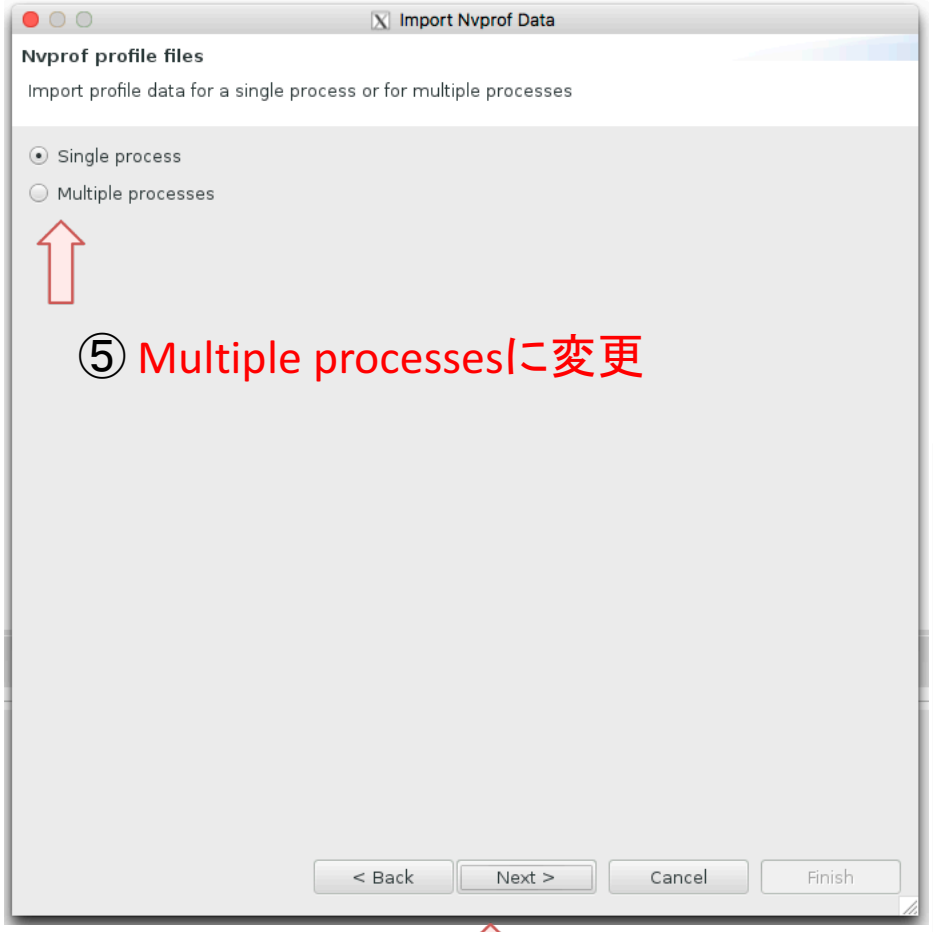
⓪ nvvp コマンドにより起動後の画面





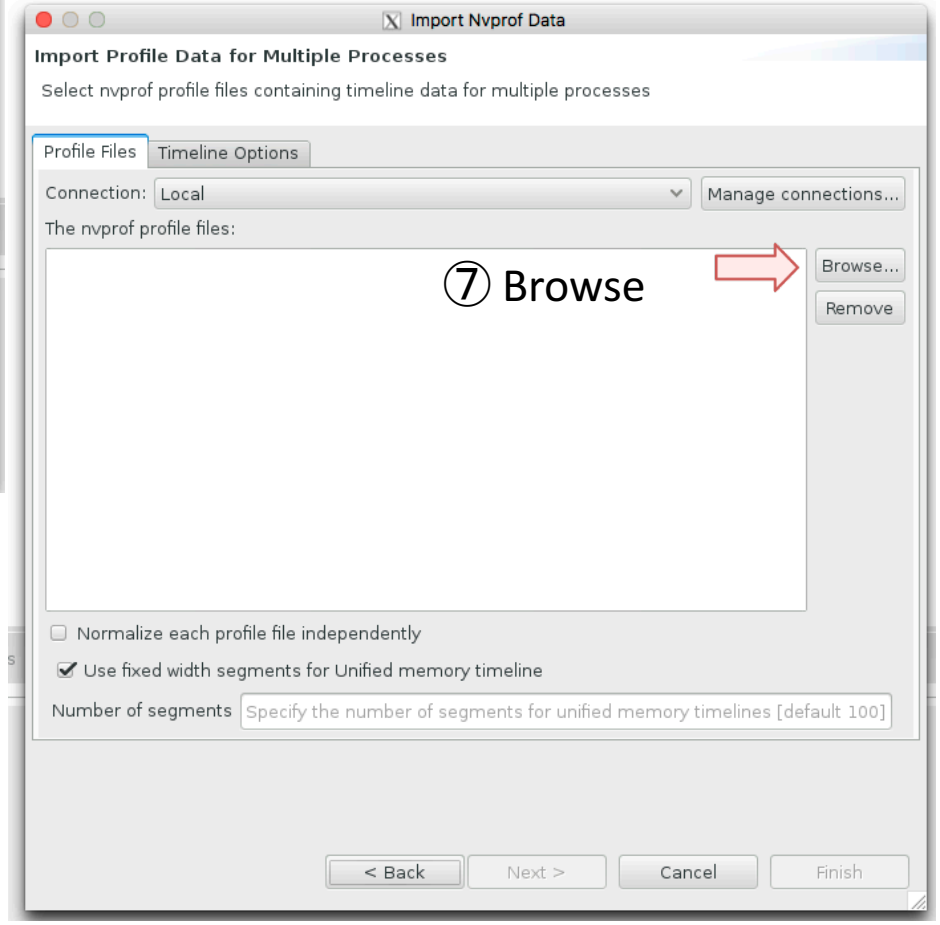
②Importをクリック



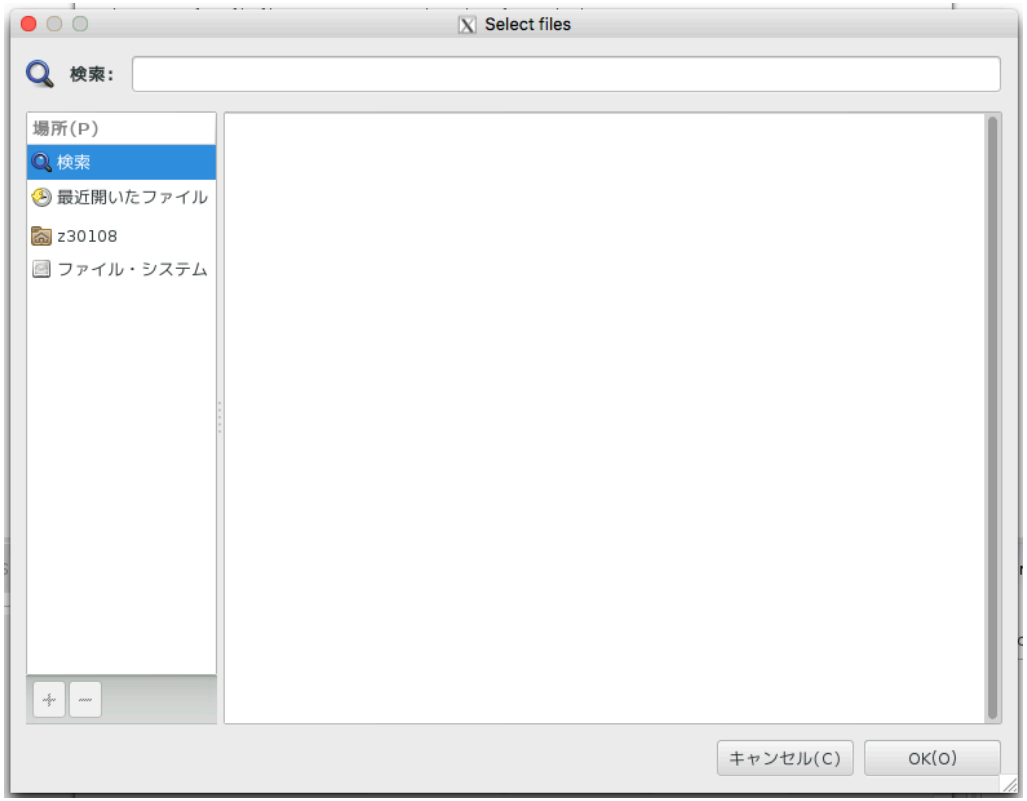


⑤ Multiple processesに変更

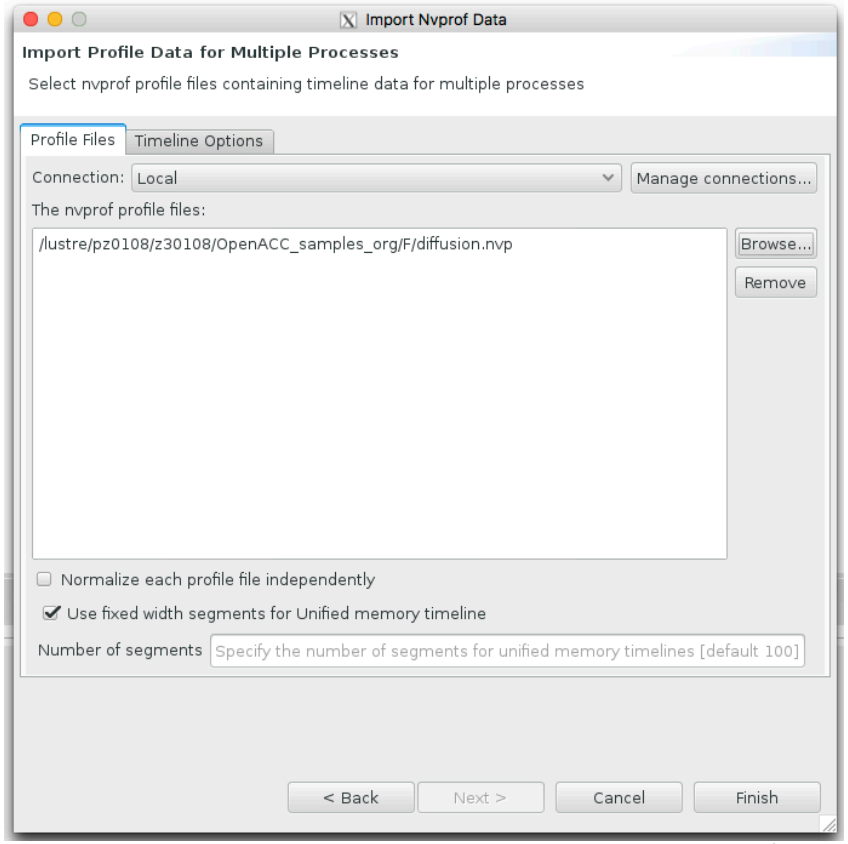
⑥ Next



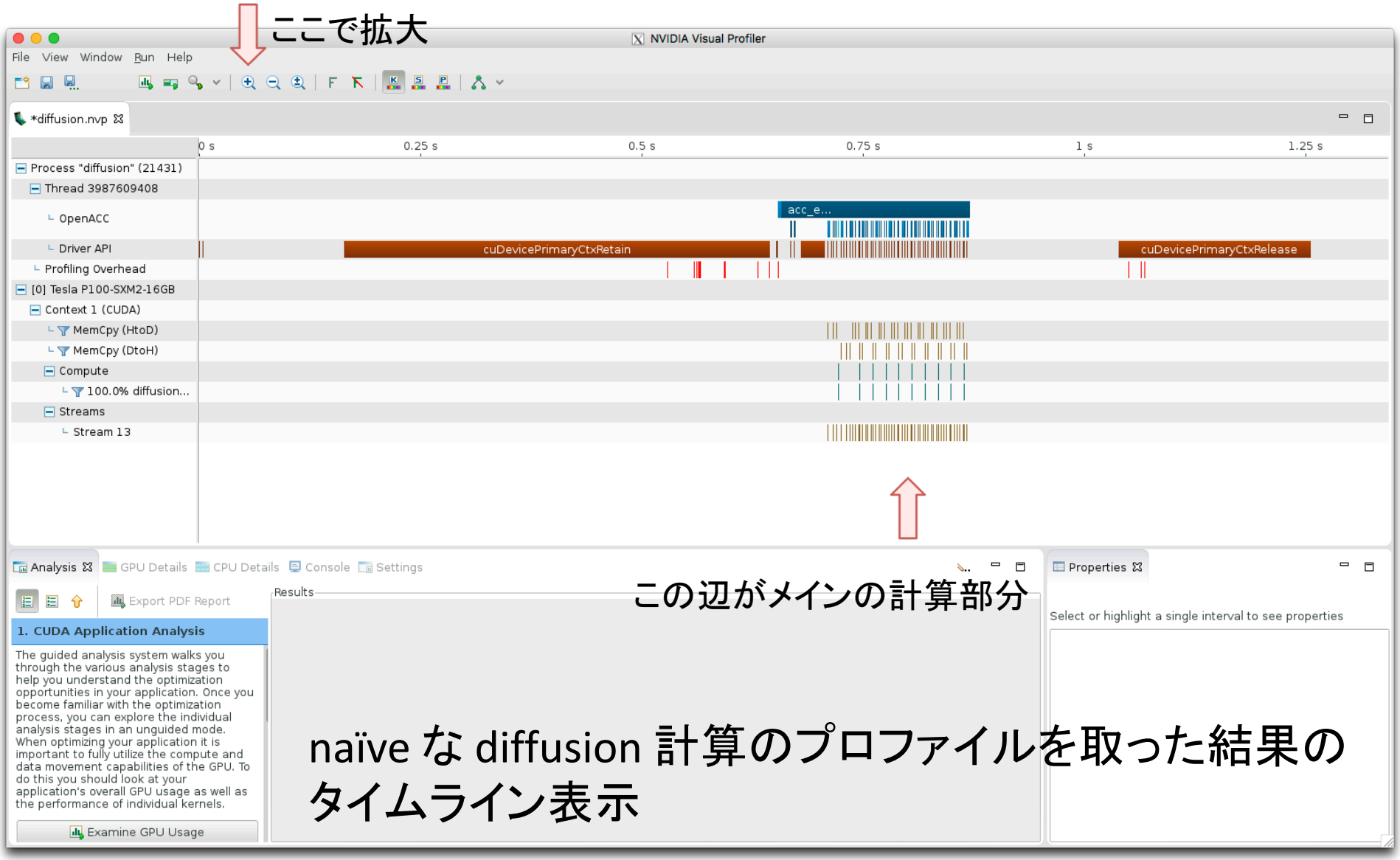
⑦ Browse



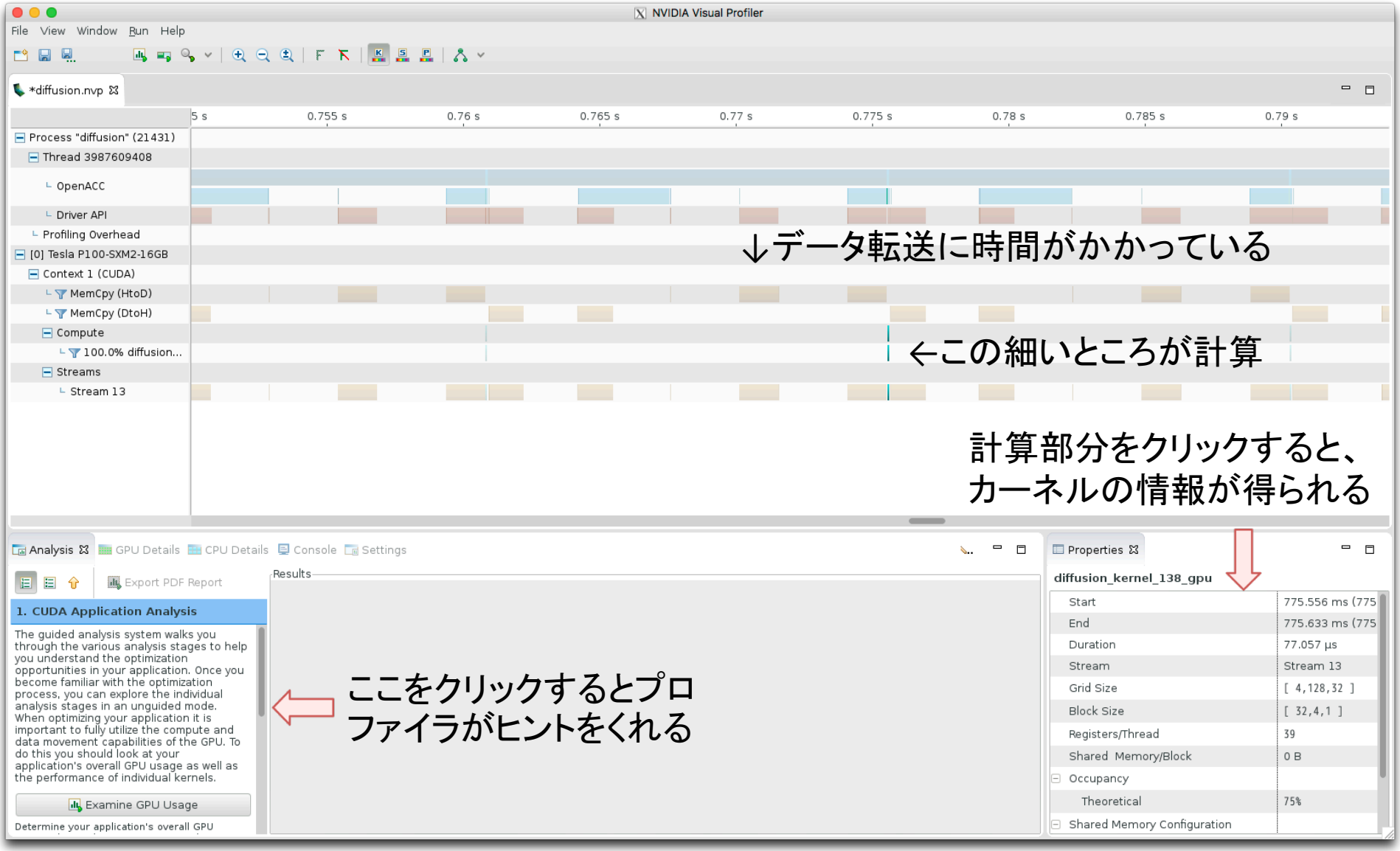
⑧ ファイル選択画面が開くので、nvvpで作成したプロファイリングデータを選択し、OK

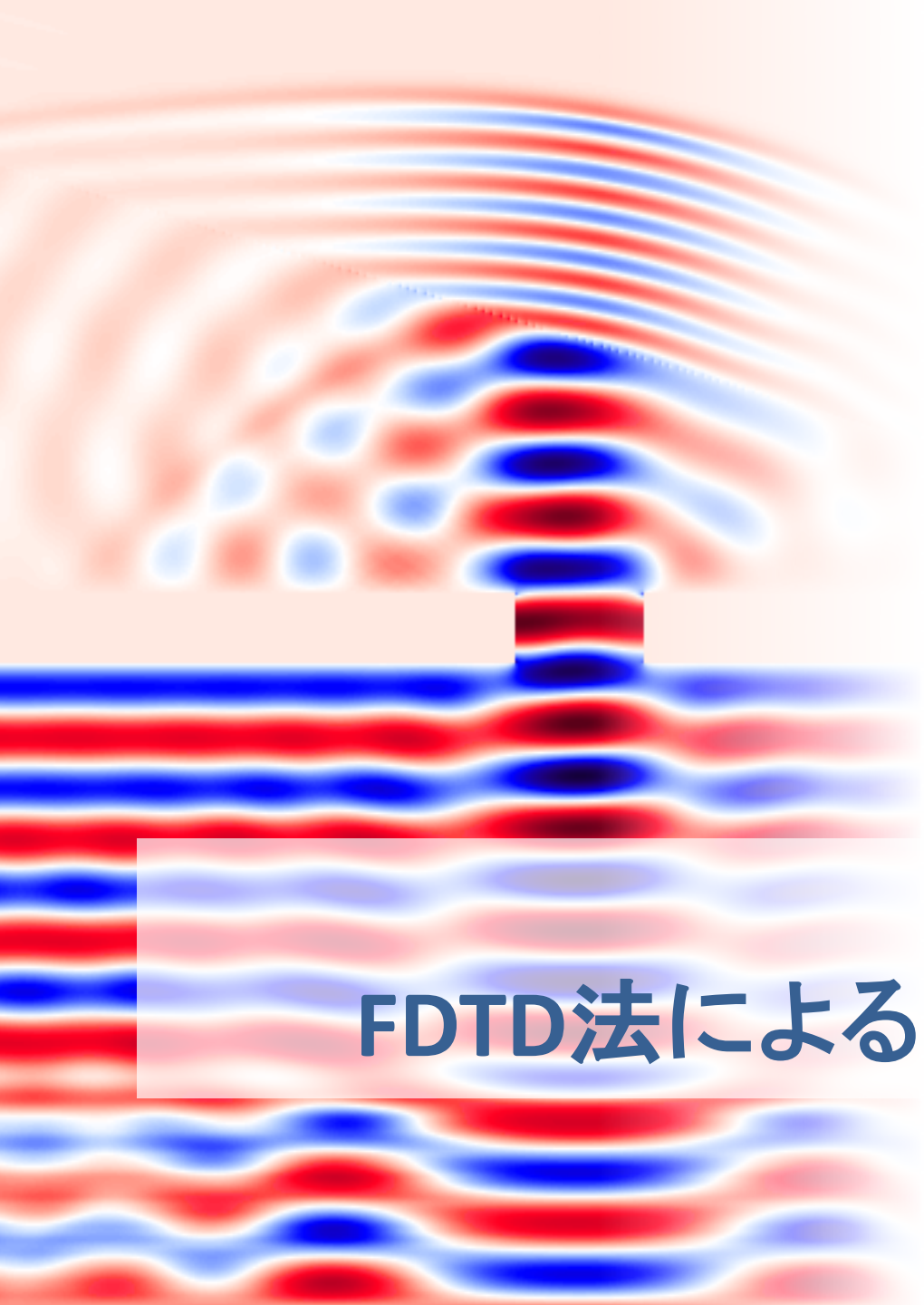


⑨ 選択できたら Finish



naïve な diffusion 計算のプロファイルを取った結果のタイムライン表示


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
↓データ転送に時間がかかっている
←この細いところが計算
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



GPUを用いた FDTD法による電磁波伝搬計算

電磁波の方程式

- 真空での電場Eと磁場Hの時間発展

  Maxwell 方程式の一部

$$\frac{\partial \boldsymbol{E}}{\partial t} = \frac{1}{\varepsilon}\nabla \times \boldsymbol{H} \qquad \frac{\partial \boldsymbol{H}}{\partial t} = -\frac{1}{\mu}\nabla \times \boldsymbol{E}$$

(ε：誘電率)　　(μ：透磁率)

この方程を、2次元FDTD法（Finite-difference time-domain 法）*を用いて解いて行きます。

* K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," IEEE Trans. on Antennas and Propagat., vol. 14, pp. 302-307, May 1966.

FDTD法（1）

- EとHの時間発展

$$\frac{\boldsymbol{E}^n - \boldsymbol{E}^{n-1}}{\Delta t} = \frac{1}{\varepsilon}\nabla \times \boldsymbol{H}^{n-\frac{1}{2}}$$

$$\frac{\boldsymbol{H}^{n+\frac{1}{2}} - \boldsymbol{H}^{n-\frac{1}{2}}}{\Delta t} = -\frac{1}{\mu}\nabla \times \boldsymbol{E}^{n}$$

変形して、

$$\boldsymbol{E}^n = \boldsymbol{E}^{n-1} + \frac{\Delta t}{\varepsilon}\nabla \times \boldsymbol{H}^{n-\frac{1}{2}}$$

$$\boldsymbol{H}^{n+\frac{1}{2}} = \boldsymbol{H}^{n-\frac{1}{2}} - \frac{\Delta t}{\mu}\nabla \times \boldsymbol{E}^{n}$$

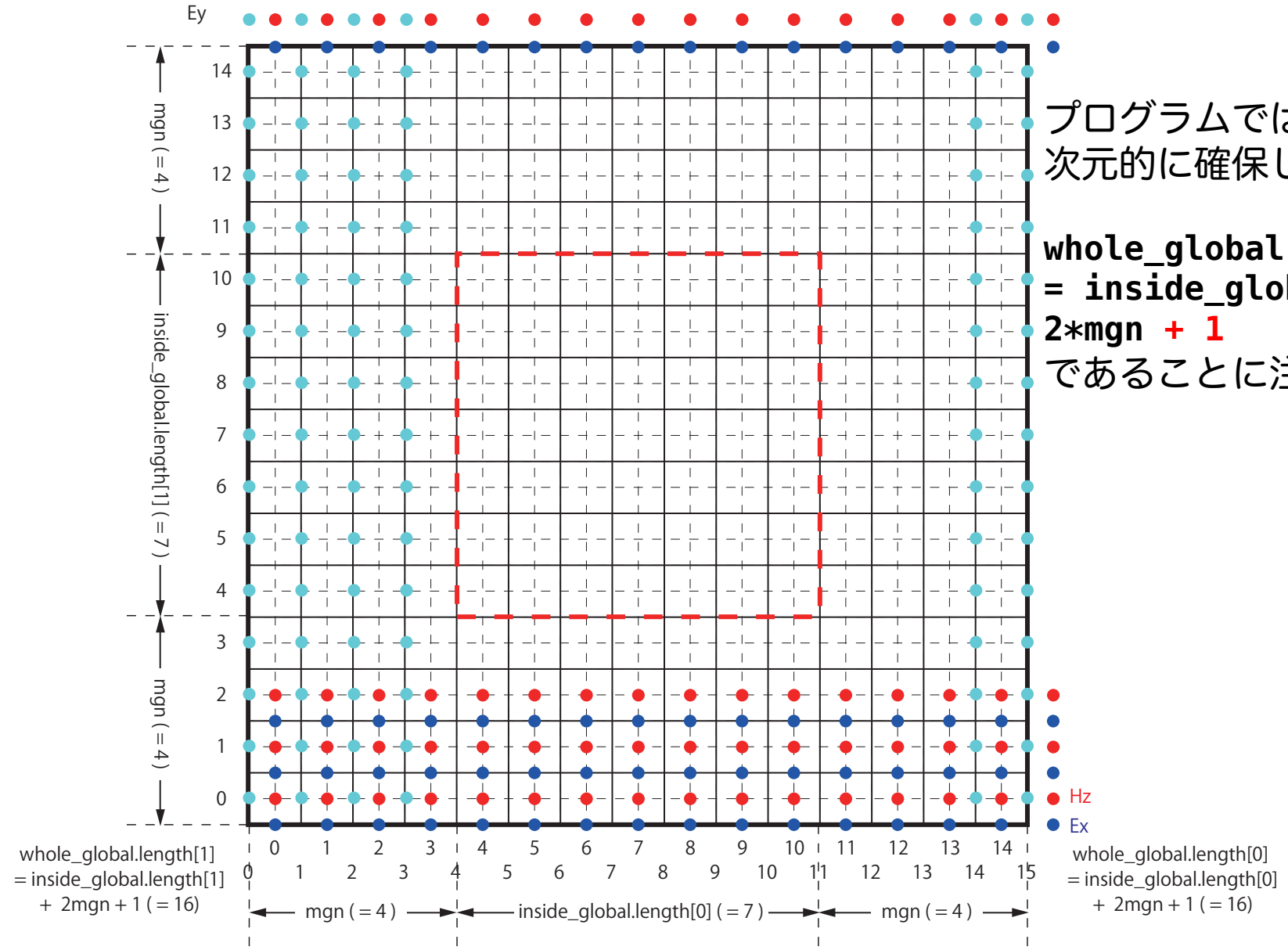
FDTD法（2）

- 例えば、

$$E_x^n(i+\frac{1}{2},j) = E_x^{n-1}(i+\frac{1}{2},j) + \frac{\Delta t}{\varepsilon(i+\frac{1}{2},j)}\left(\frac{H_z^{n-\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2}) - H_z^{n-\frac{1}{2}}(i+\frac{1}{2},j-\frac{1}{2})}{\Delta y}\right)$$

$$H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2}) = H_z^{n-\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2}) - \frac{\Delta t}{\mu(i+\frac{1}{2},j+\frac{1}{2})}\left(\frac{E_y^n(i+1,j+\frac{1}{2}) - E_y^n(i,j+\frac{1}{2})}{\Delta x} - \frac{E_x^n(i+\frac{1}{2},j+1) - E_x^n(i+\frac{1}{2},j)}{\Delta y}\right)$$

2次元FDTD法の変数配置



プログラムでは、配列は1次元的に確保します。

whole_global.length[0] = inside_global[0] + 2*mgn + 1
であることに注意

ソースコード（1）

- サンプルコード: openacc_fdtd/
  - ✓ OpenACCを利用したFDTD法（電磁波解析）

openacc_fdtd/01_original	MPI並列化されたCPUコード。
openacc_fdtd/02_openacc1	calc_ex_ey, pml_boundary_ex, pml_boundary_ey, がOpenACC。
openacc_fdtd/03_openacc2	時間更新ループ全体が OpenACC。
openacc_fdtd/04_openacc3	初期化を含め OpenACC。
openacc_fdtd/05_openacc4	データ移動の最適化。

※本プログラムはMPI化されていますが、本講習会では扱いません

ソースコード（2）

- それぞれのファイルの内容

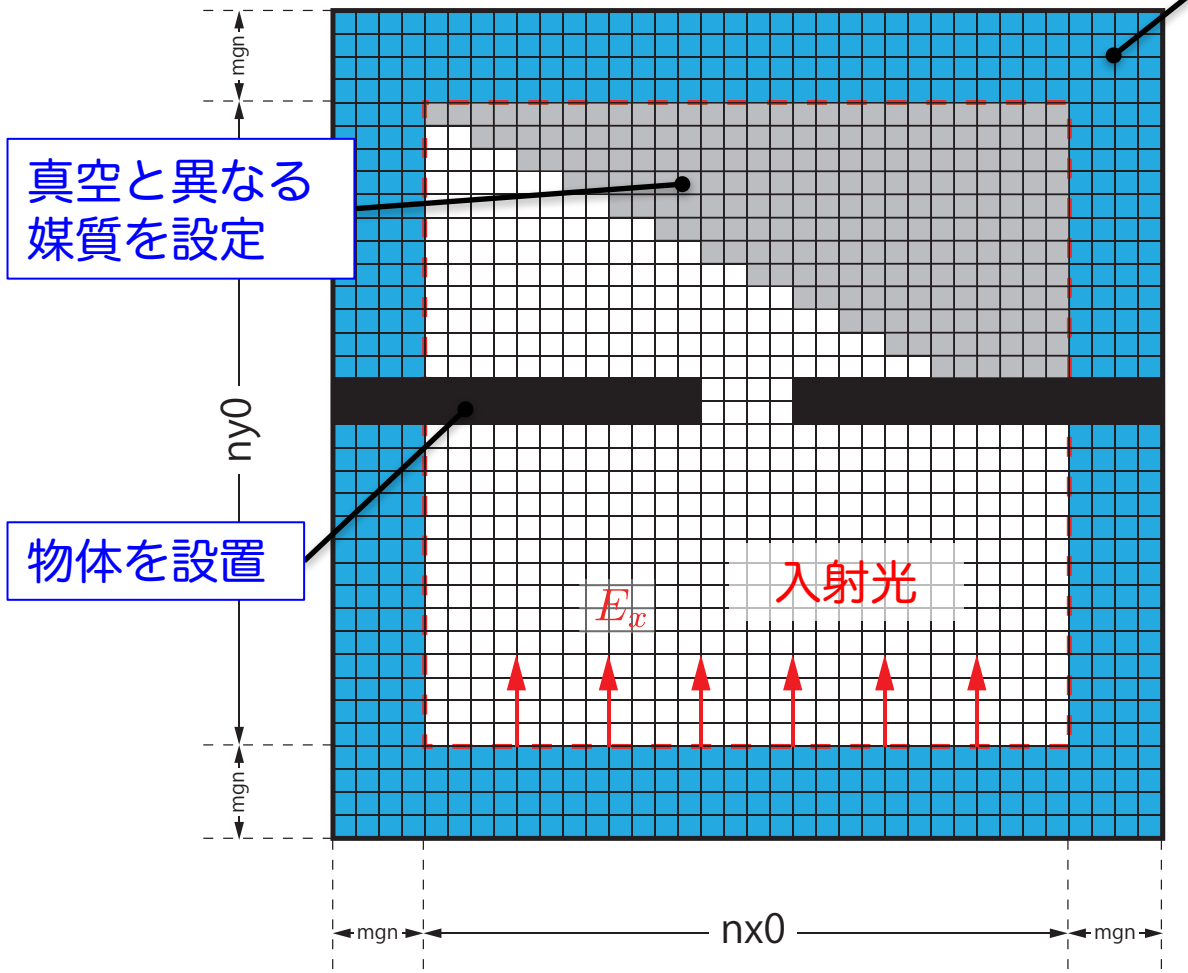
main.c	プログラムのメインコード
fdtd2d.{c, h}	2次元 FDTD の 計算コード
fdtd2d_sources.{c, h}	入射光設定のための関数
setup.c	計算条件の設定と変数の初期化
config.{c, h}	物理定数の定義
output.{cc, h}	計算結果出力のための関数
bitmap*	BMPファイル作成のための関数

本講習では、“main.c”、“fdtd2d.c”、” fdtd2d_sources.c”、”setup.c” のソースコードを追記・修正していきます。

計算条件

- 2次元波動伝搬
  - ✓ 成分: Ex、Ey、Hz
  - ✓ y 方向下側から平面波を入射

電磁波の境界での非物理的な反射を防ぐための吸収境界条件 (PML)




```
dx = lx/nx
dy = ly/ny
```

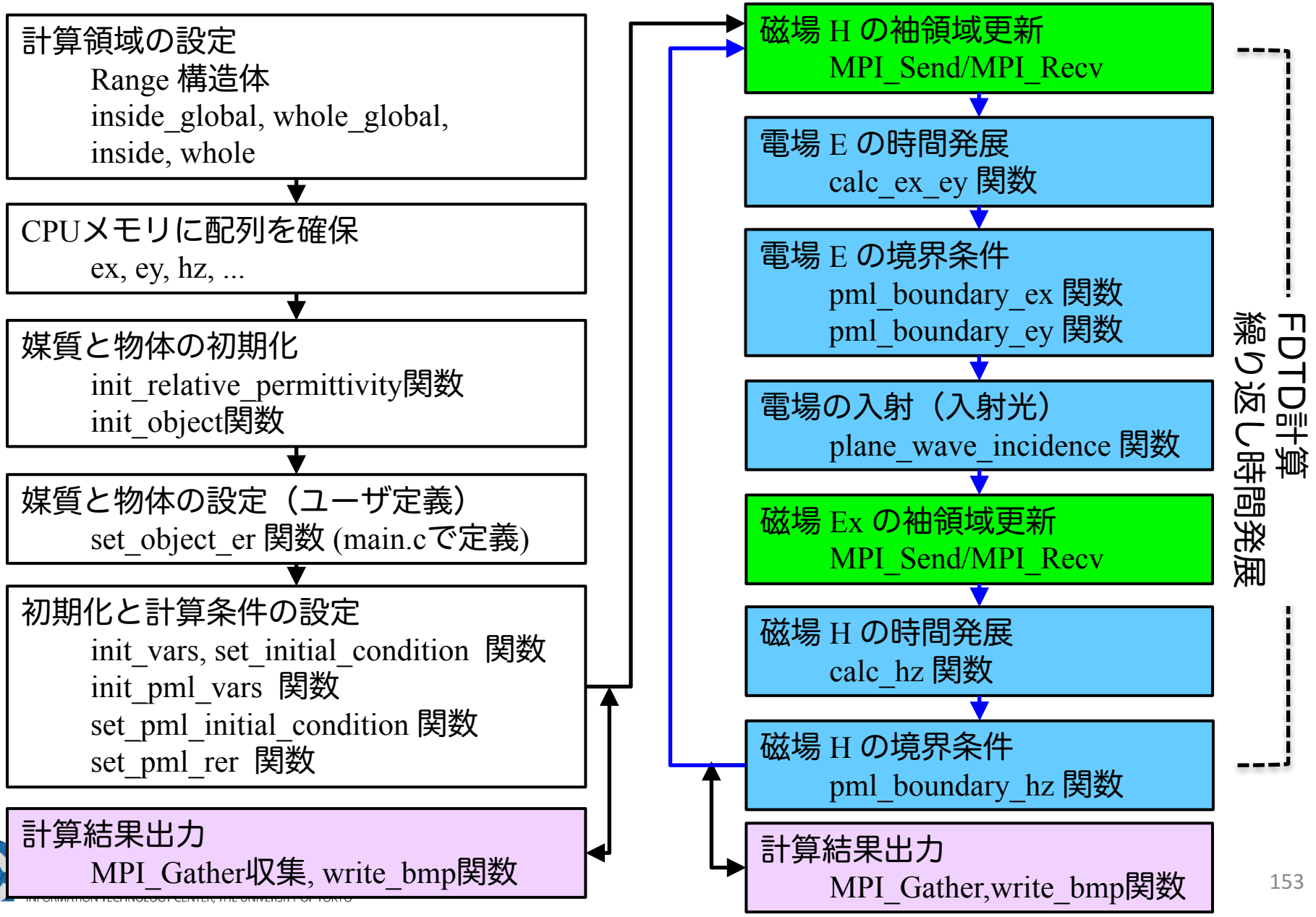
デフォルト設定:

```
nx = 512
ny = 512
mgn = 8
lnx = 529
lny = 529
```

プログラム中では下記の変数が使われているので注意

```
inside_global.length[0] = nx0
inside_global.length[1] = ny0
whole_global.length[0]
            = nx0 + 2*mgn + 1
whole_global.length[1]
            = ny0 + 2*mgn + 1
```

コード全体の流れ（main.c 内）



計算領域の設定(1)

- Range 構造体
  - ✓ 計算領域の始点と大きさを保持

```
// config.h
struct Range {
    int length[2];
    int begin [2];
};

// main.c
const struct Range inside_global = { { atoi(argv[1]), atoi(argv[2]) },
                                     { 0, 0 } };
const struct Range whole_global  = { { inside_global.length[0] + 2*mgn + 1,
                                       inside_global.length[1] + 2*mgn + 1},
                                     { inside_global.begin[0]  - mgn      ,
                                       inside_global.begin[1]  - mgn   } };

const struct Range inside        = { { inside_global.length[0],
                                       inside_global.length[1]/nsubdomains },
                                     { 0,
                                       inside_global.length[1]/nsubdomains * rank } };
const struct Range whole         = { { inside.length[0] + 2*mgn + 1,
                                       inside.length[1] + 2*mgn + 1},
                                     { inside.begin[0]  - mgn      ,
                                       inside.begin[1]  - mgn   } };
```

全領域の中心領域

全領域の全体領域

分割領域の中心領域

分割領域の全体領域

計算領域の設定(2)

- Range 構造体
  - ✓ 計算領域の始点と大きさを保持

```
struct Range {
    int length[2];
    int begin [2];
};

const struct Range inside        = { { inside_global.length[0],
                                       inside_global.length[1]/nsubdomains },
                                     { 0,
                                       inside_global.length[1]/nsubdomains * rank } };
const struct Range whole         = { { inside.length[0] + 2*mgn + 1,
                                       inside.length[1] + 2*mgn + 1},
                                     { inside.begin[0]  - mgn      ,
                                       inside.begin[1]  - mgn   } };
```

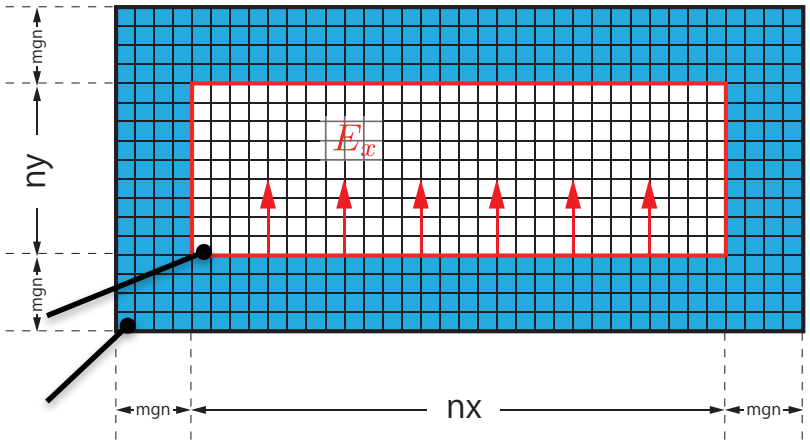
分割領域の
中心領域

分割領域の
全体領域

プログラムでは
下記の通り

```
inside.length[0] = nx
inside.length[1] = ny

whole.length[0] = nx + 2*mgn + 1
whole.length[1] = ny + 2*mgn + 1
```

座標(inside.begin[0], inside.begin[1])
座標(whole.begin[0], whole.begin[1])



配列の確保

- 物理変数配列は main.c で確保

```
// main.c
const int    nelems      = whole.length[0] * whole.length[1];
const int    nelems_x    = whole.length[0];
const int    nelems_y    = whole.length[1];
const size_t size        = sizeof(FLOAT)*nelems;
const size_t size_x      = sizeof(FLOAT)*nelems_x;
const size_t size_y      = sizeof(FLOAT)*nelems_y;
const size_t size_global = sizeof(FLOAT)* whole_global.length[0] * whole_global.length[1];

FLOAT *ex    = (FLOAT *)malloc(size); // 電場 Ex
FLOAT *ey    = (FLOAT *)malloc(size); // 電場 Ey
FLOAT *hz    = (FLOAT *)malloc(size); // 磁場 Hz
...
// For output
FLOAT *ex_global = (FLOAT *)malloc(size_global);
FLOAT *ey_global = (FLOAT *)malloc(size_global);
FLOAT *hz_global = (FLOAT *)malloc(size_global);
```

  - 多くの配列は whole.length[0] * whole.length[1]
  - ex_global, ey_global, hz_global はファイル出力に使うため、**whole_global.length[0] * whole_global.length[1]**

時間発展(1)

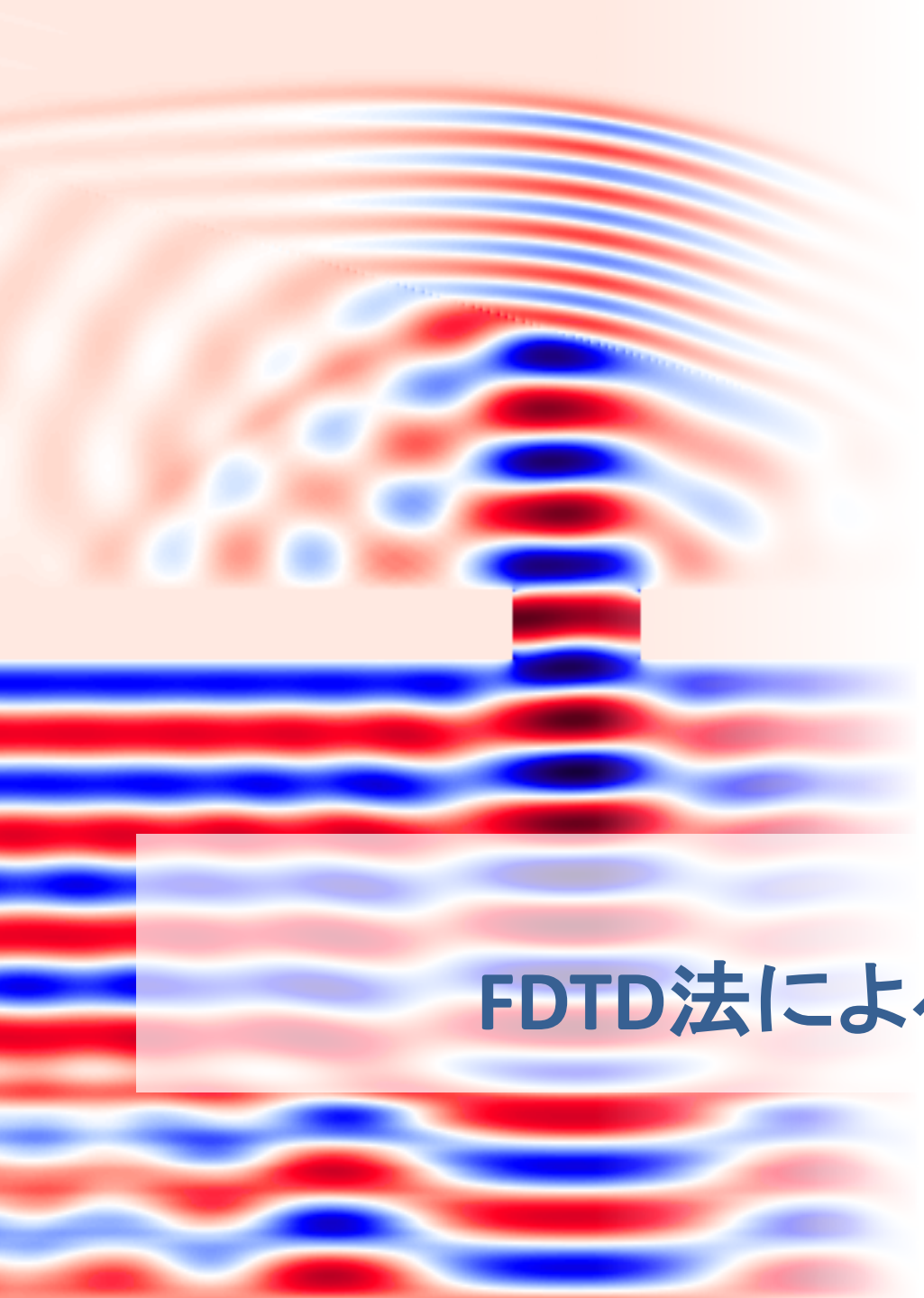
- 前半
  - ✓ 電場Eの時間発展(calc_ex_ey)、境界条件(pml_boundary_...)
  - ✓ 入射光(plane_wave_incidence)

```
while (icnt < nt) {

    MPI_Status status;
    const int tag = 0;
    const int nhalo       = whole.length[0];
    const int inside_end1 = inside.begin[1] + inside.length[1];

    const int src_hz      = whole.length[0] * (inside_end1     - whole.begin[1] - 1);
    const int dst_hz      = whole.length[0] * (inside.begin[1] - whole.begin[1] - 1);

    MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up  , tag, MPI_COMM_WORLD);
    MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD, &status);

    calc_ex_ey(&whole, &inside, hz, cexly, ceylx, ex, ey);
    pml_boundary_ex(&whole, &inside, hz, cexy, cexyl, rer_ex, ex, exy);
    pml_boundary_ey(&whole, &inside, hz, ceyx, ceyxl, rer_ey, ey, eyx);

    const int j_in = 0;
    plane_wave_incidence(&whole, &inside, time, j_in, wavelength, ex, ey);
    time += 0.5*dt;

    (後半へ)
```

時間発展（2）

- 後半
  - ✓ 磁場Hの時間発展（calc_hz）、境界条件(pml_boundary_hz)

```
    （前半から）

    const int src_ex      = whole.length[0] * (inside.begin[1] - whole.begin[1]);
    const int dst_ex      = whole.length[0] * (inside_end1     - whole.begin[1]);

    MPI_Send(&ex[src_ex], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD);
    MPI_Recv(&ex[dst_ex], nhalo, MPI_FLOAT_T, rank_up  , tag, MPI_COMM_WORLD, &status);

    calc_hz(&whole, &inside, ey, ex, chzlx, chzly, hz);
    pml_boundary_hz(&whole, &inside, ey, ex, chzx, chzxl, chzy, chzyl, hz, hzx, hzy);
    time += 0.5*dt;

    icnt++;

    (出力など)
}
```



GPUを用いた FDTD法による電磁波伝搬計算の実習

プログラムのコンパイルと実行（1）

- CPUコードのコンパイルと実行

openacc_fdtd/01_original

```
$ cd openacc_mpi_fdtd/01_original
$ make
$ qsub ./run.sh
$ cat run.sh.o??????
Rank 0: hostname = a090
Rank 1: hostname = a090
Rank 2: hostname = a091
Rank 3: hostname = a091
Calculation condition
  nx_global     =   512

（省略）

icnt =  4900, time = 2.3115e-14 [sec]
icnt =  5000, time = 2.3587e-14 [sec]
-----------------------------
Domain      = 512 x 512
nsubdomains = 4
output_file = 1
Time        =   4.103535 [sec]
-----------------------------
```

？の数字はジョブごとに変わります。

利用したノード

計算領域サイズ、領域分割数、出力の有無、計算時間

なお、qsub ./run_no_out.sh すると出力なしで実行する。性能測定用。

プログラムのコンパイルと実行（2）

- プログラムの実行時オプション

openacc_fdtd/01_original

```
$ cat run.sh
#! /bin/sh
#PBS -q h-tutorial
#PBS -l select=1:mpiprocs=1:ompthreads=0

（省略）

mkdir -p sim_run
cd sim_run

nprocs=1
mpirun -np $nprocs ../run 512 512 $nprocs 5000 50
```

```
mpirun -np <nprocs> ../run <nx> <ny> <nprocs> <nt> <nout>
```

nprocs: 全ランク数（=分割数）※今回は1

nx, ny: 計算領域サイズ

nt: 全時間ステップ

nout: 出力を行うタイムステップ数。50 の場合、50ステップに1回出力する。0 を指定すると出力しない。

計算結果の表示

- 計算結果は sim_run に BMP として出力される

```
$ cd sim_run/
```

openacc_mpi_fdtd/01_original

- 計算結果の表示
  - ✓ 1枚のBMPを見る

```
$ display e05000.bmp
```

  - ✓ 複数のBMPファイルをアニメーションで表示

```
$ animate *.bmp
```

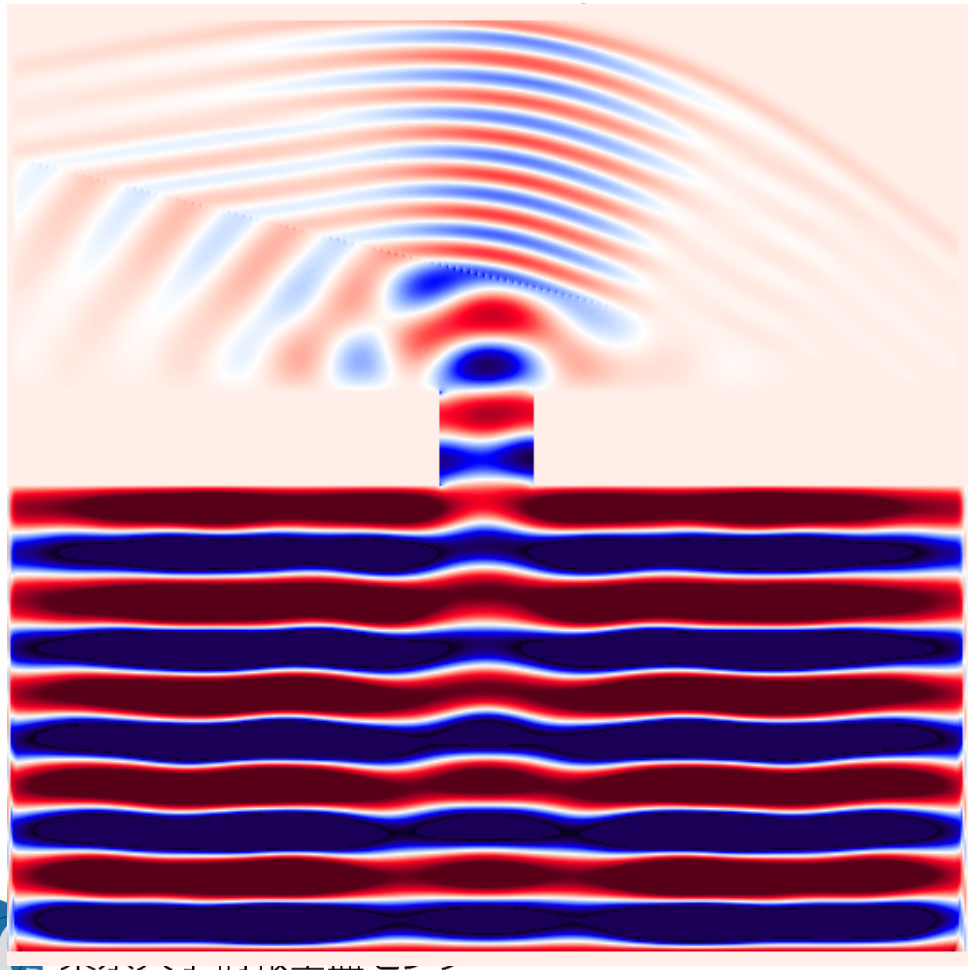
なお

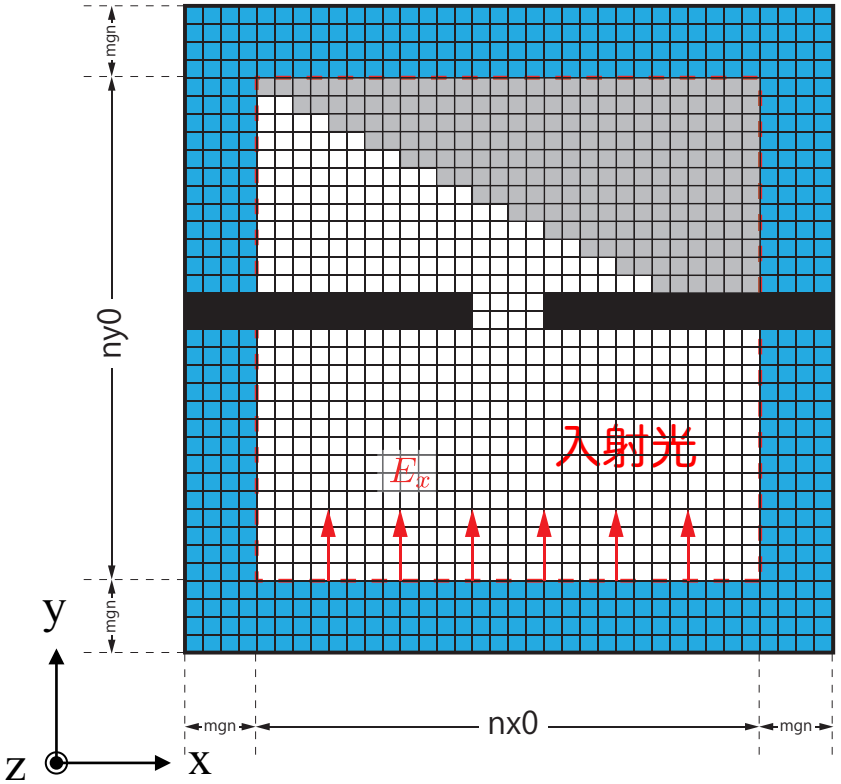
```
ssh -Y txxxxx@reedbush.cc.u-tokyo.ac.jp
```

と **-Y** をつけていないと表示されない。うまく表示できない場合は画像を手元にコピーして表示してください。

計算結果の例

- 出力されたBMPファイルの一例





実習1

- calc_ex_ey, pml_boundary_ex, pml_boundary_ey を OpenACC化しましょう。

- Makefile
  - ✓ コンパイルオプションの修正
- main.c
  - ✓ OpenACCヘッダーの追加
  - ✓ data 指示文の追加
- fdtd2d.c
  - ✓ kernels 指示文、loop 指示文の追加

実行速度が遅くても、動くプログラムである状態を保ちながらOpenACC化します。末端の関数からOpenACC化するのがよいでしょう。

解答例は、 openacc_fdtd/02_openacc1

kernels, loop指示文

- fdtd2d.c 内の関数

```
void calc_ex_ey(const struct Range *whole, const struct Range *inside,
          const FLOAT *hz, const FLOAT *cexly, const FLOAT *ceylx, FLOAT *ex, FLOAT *ey)
{
    const int nx    = inside->length[0];
    const int ny    = inside->length[1];
    const int mgn[] = { inside->begin[0] - whole->begin[0],
                        inside->begin[1] - whole->begin[1] };
    const int lnx   = whole->length[0];

#pragma acc kernels present(hz, cexly, ex)
#pragma acc loop independent
    for (int j=0; j<ny+1; j++) {
#pragma acc loop independent
        for (int i=0; i<nx; i++) {
            const int ix = (j+mgn[1])*lnx + i+mgn[0];
            const int jm = ix - lnx;
            //ex[ix] += cexly[ix]*(hz[ix]-hz[jm]) - cexlz[ix]*(hy[ix]-hy[km]);
            ex[ix] += cexly[ix]*(hz[ix]-hz[jm]);
        }
    }

(省略)

}
```

実習2

- main 関数内の while 内をすべて OpenACCにしましょう。

- main.c
  - ✓ data 指示文の移動と copyin などの最適化
- fdtd2d.c
  - ✓ 残りの関数にkernels 指示文、loop 指示文の追加
- fdtd2d_sources.c
  - ✓ kernels 指示文、loop 指示文の追加

解答例は、 openacc_fdtd/03_openacc2

data 指示文

- main関数のwhile 外に data を移動

```
#pragma acc data ¥
    copyin(ex[0:nelems], ey[0:nelems], hz[0:nelems])                    ¥
    copyin(cexly[0:nelems], ceylx[0:nelems], chzlx[0:nelems], chzly[0:nelems]) ¥
    copyin(exy[0:nelems], eyx[0:nelems], hzx[0:nelems], hzy[0:nelems])  ¥
    copyin(cexy[0:nelems_y], ceyx[0:nelems_x], chzx[0:nelems_x], chzy[0:nelems_y]) ¥
    copyin(cexyl[0:nelems_y], ceyxl[0:nelems_x], chzxl[0:nelems_x], chzyl[0:nelems_y]) ¥
    copyin(obj[0:nelems], er[0:nelems]) ¥
    copyin(rer_ex[0:nelems], rer_ey[0:nelems])
    {

    while (icnt < nt) {

        MPI_Status status;
        const int tag = 0;
        const int nhalo       = whole.length[0];
        const int inside_end1 = inside.begin[1] + inside.length[1];

        const int src_hz      = whole.length[0] * (inside_end1     - whole.begin[1] - 1);
        const int dst_hz      = whole.length[0] * (inside.begin[1] - whole.begin[1] - 1);

#pragma acc host_data use_device(hz)
        {
        MPI_Send(&hz[src_hz], nhalo, MPI_FLOAT_T, rank_up  , tag, MPI_COMM_WORLD);
        MPI_Recv(&hz[dst_hz], nhalo, MPI_FLOAT_T, rank_down, tag, MPI_COMM_WORLD, &status);
        }
```

実習3

- 初期化を含めて全てOpenACCにします。ただし、set_object_er がCPU上のユーザ定義関数のため、これ以降の初期化関数をOpenACCにします。
- main.c
  - ✓ data 指示文の移動と最適化（多くが create になるはずです）
- setup.c
  - ✓ kernels 指示文、loop 指示文の追加

解答例は、 openacc_fdtd/04_openacc3

実習4

- 計算領域のサイズなどを変更して性能測定してみましょう。
- OpenACCコードをさらに最適化しましょう。
  - ✓ PGI_ACC_TIMEも活用しましょう。
  - ✓ 実は単純に fdtd2d.c に kernels と loop を入れても、いくつかの関数で暗黙の copyin が発生します。これも修正していきましょう。

```
$ make
calc_ex_ey:
     25, Generating present(ex[:],cexly[:])
         Generating implicit copyin(mgn[:])
         Generating present(hz[:])
     27, Loop is parallelizable
     29, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         27, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         29, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     37, Generating present(ey[:],ceylx[:])
         Generating implicit copyin(mgn[:])
```

解答例は、openacc_fdtd/05_openacc4

Q & A

- アカウントは1ヶ月間有効です。
- 資料のPDF版はWEBページにございます。
  - https://www.cc.u-tokyo.ac.jp/events/lectures/97/lec097.php
- アンケートへの協力をお願いします。