GPUプログラミング入門（OpenACC）

プロメテック・ソフトウェア株式会社　阿部　光平，Version 2025.7.18

本資料は以下の資料をベースにプロメテック・ソフトウェア株式会社が本講習会に適した情報となるよう加筆修正等を行ったものです。

OpenACC Official Training Materials

- Below slides are released by NVIDIA Corporation under CC BY 4.0
- Slides: https://drive.google.com/open?id=1d_eIwIRfScHxfJu6pnR28JrV3cMIwkIL
- CC BY 4.0: https://creativecommons.org/licenses/by/4.0/deed.ja

OpenACC
More Science, Less Programming

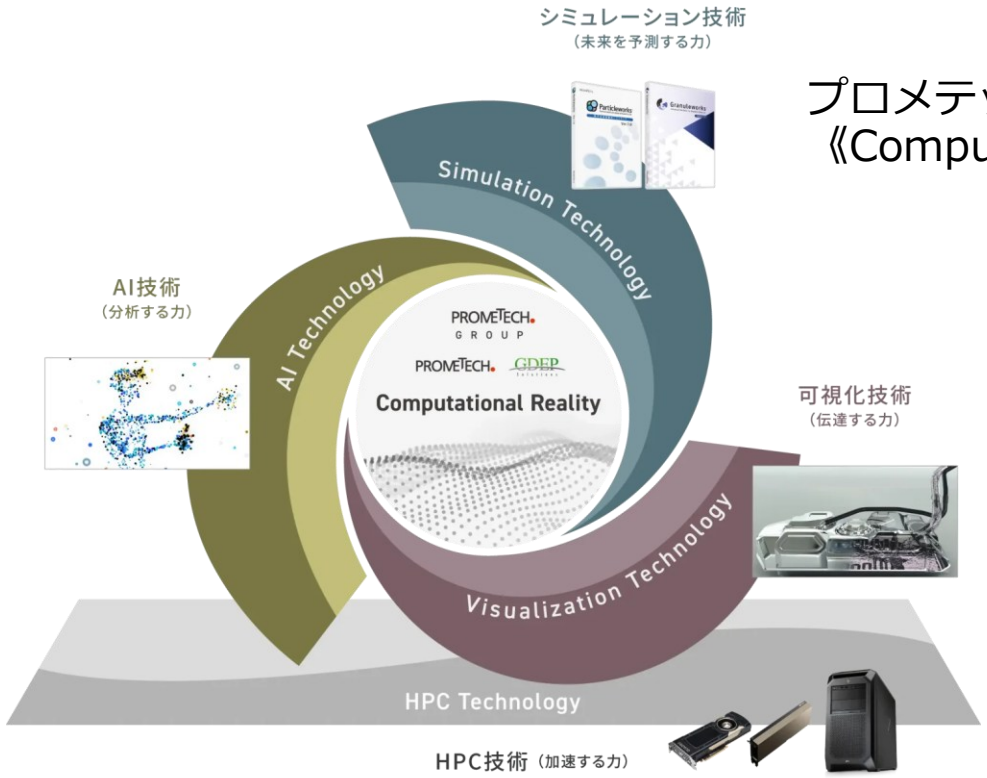
自己紹介

阿部 光平 (Kohei Abe)

- 所属：プロメテック・ソフトウェア株式会社
  AI/HPCプラットフォーム事業開発本部　エンジニア
- 業務経験：OpenACCを使ったアプリケーションのGPU高速化
  NVIDIA HPC SDKのインストールやコンパイラのサポート
  GPUを搭載したLLM/RAGソリューションセットの開発、評価
- 経歴：-2023年3月　千葉大学大学院 融合理工学府 博士後期課程修了
  2023年4月-　現職
- 学位：博士（理学）

OpenACC
More Science, Less Programming

プロメテックグループ紹介



プロメテックグループはグループ各社の専門性を結集させて、
《Computational Reality》を社会に実装する挑戦を続けています。

プロメテック・ソフトウェア株式会社
計算科学技術をコアとするソフトウェア製品の開発を担当

GDEPソリューションズ株式会社
高度な科学計算を実行するハイパフォーマンスコンピューティング（HPC）の基盤構築を担当

OpenACC
More Science, Less Programming

事業内容

- 流体・粉体解析ソフトウェアの開発・販売・サポート

- 解析コンサルティングサービス
- 可視化・映像制作サービス
- NVIDIA HPCコンパイラサポートサービス

  ソフテック社より継承した旧PGIコンパイラの技術活用によるコンパイラのサポート

講習会の内容（GPUプログラミング入門）

1. GPUプログラミングの概要
    i. アムダールの法則
    ii. GPUの特性
    iii. GPUプログラミング
    iv. NVIDIA HPC SDK
2. OpenACCを使ったプログラムの並列化
    i. OpenACCの概要
    ii. OpenACCの基本構文
    iii. Parallel指示文
    iv. Kernels指示文
    v. Loop指示文
    vi. OpenACCコードのコンパイル
3. CPU-GPU間のデータ転送の最適化
    i. メモリ管理の概要
    ii. Managed memory
    iii. Data節
    iv. 明示的なメモリ管理
    v. Data指示文
    vi. 暗黙 vs 明示的なデータ領域
    vii. 非構造データ指示文
    viii. CPU-GPU間のデータ同期
4. ループの並列化
5. OpenACCによる並列化の進め方
6. まとめ

OpenACC
More Science, Less Programming

講習会の内容

GPUプログラミング入門

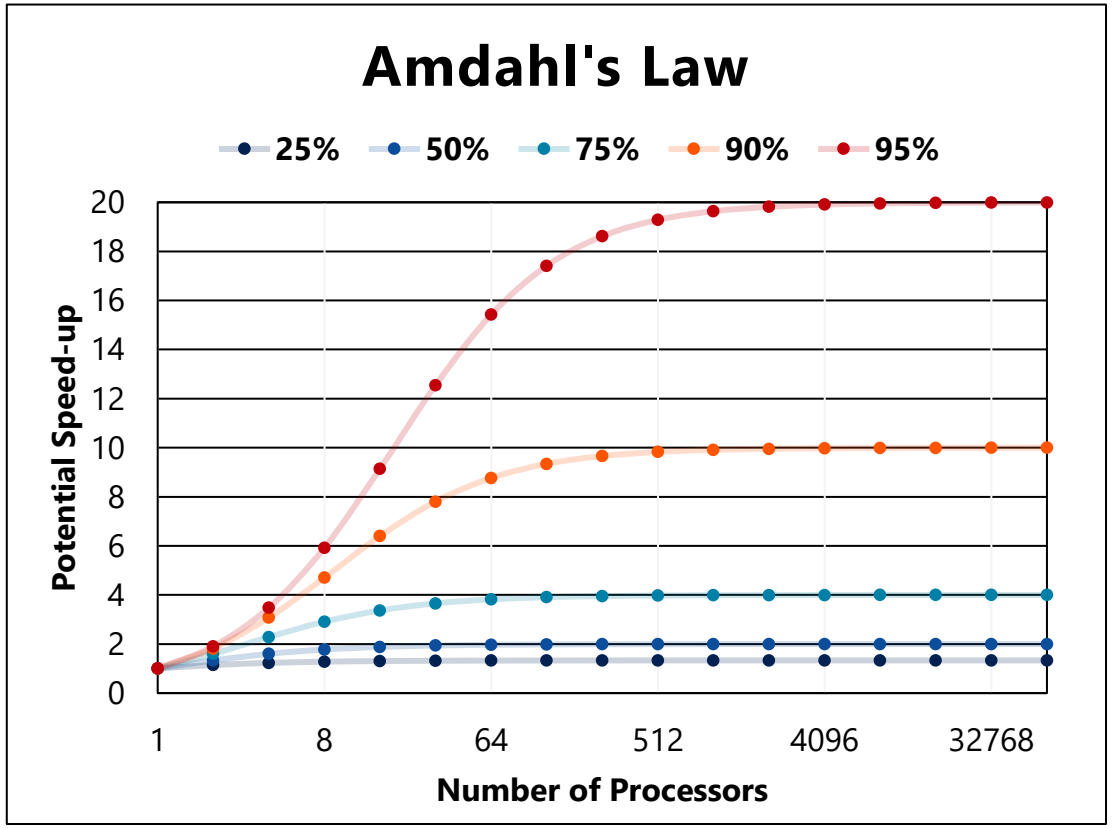
- ✓前提知識：Fortran、C言語（講習はFortranベース）
- ✓あるとよい知識：MPI、OpenMP
- ✓OpenACCを用いるターゲットは、NVIDIA CUDA GPU (NVIDIA A100)

1. GPUプログラミングの概要
i. アムダールの法則

OpenACC
More Science, Less Programming

アムダールの法則

性能は逐次処理によって律速される



- コードの並列化により達成可能な最高性能は、並列化不可能な逐次処理部分によって制限される
- 計算コストの高い処理から順に並列化する
- 例１：計算コストの50%を並列化した場合

  → 1/(1 - 0.5) = **2倍**
- 例２：計算コストの90%を並列化した場合

  → 1/(1 - 0.9) = **10倍**

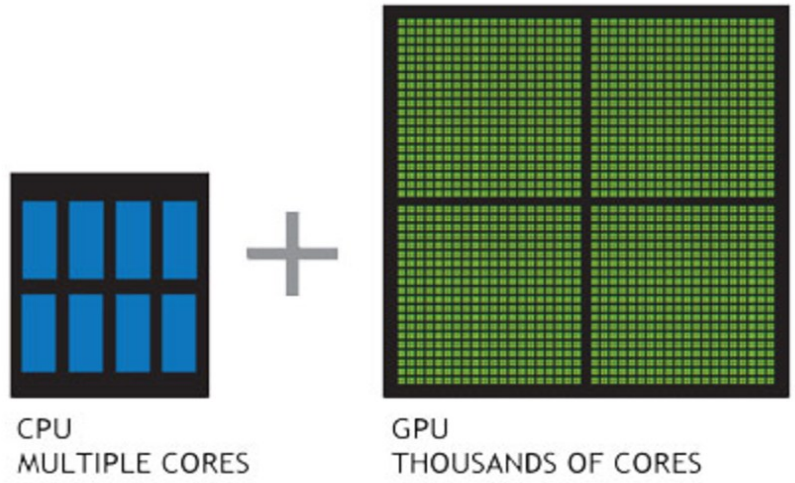
OpenACC
More Science, Less Programming

ii. GPUの特性

OpenACC
More Science, Less Programming

CPUとGPUの物理コアの違い

- CPU
  - 複雑かつ高性能な演算コアを数十個持つ
- GPU
  - シンプルかつ低性能な演算コアを1000個以上持つ
  - 並列化により高い性能を発揮する

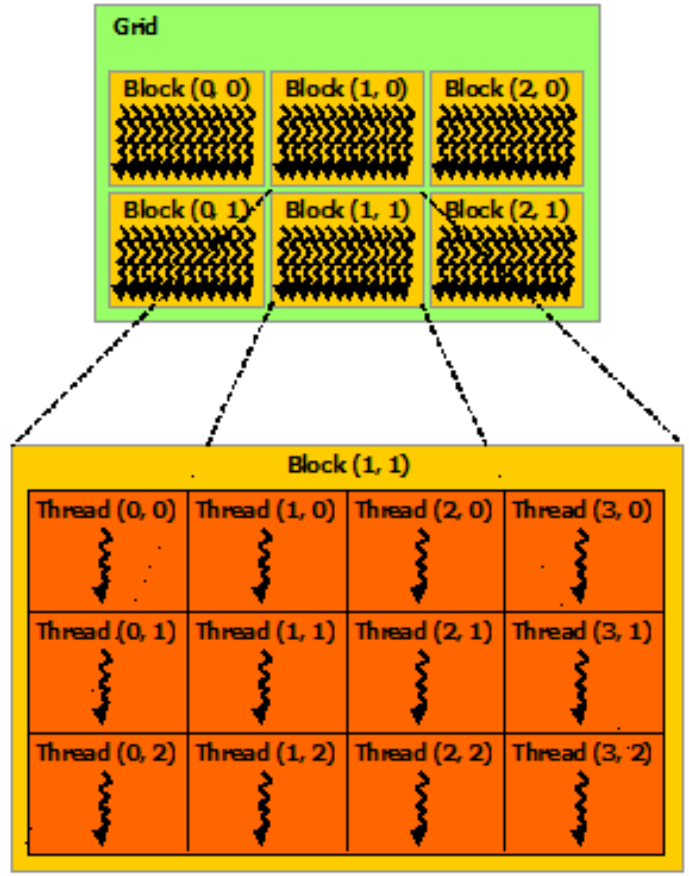


OpenACC
More Science, Less Programming

NVIDIA CUDA GPUの並列化階層

- GPUが持つ1000以上の演算コアを活用するために
  3つの並列化階層に分けられている
- **GRID**：最大3次元で複数のBLOCKを持つ
- **BLOCK**：最大3次元で複数のTHREADを持つ
- **THREAD**：並列化の最小単位

- 多重ループ並列化、SIMD処理などに割当



図：CUDA Programming Guideより

OpenACC
More Science, Less Programming

SIMTモデル

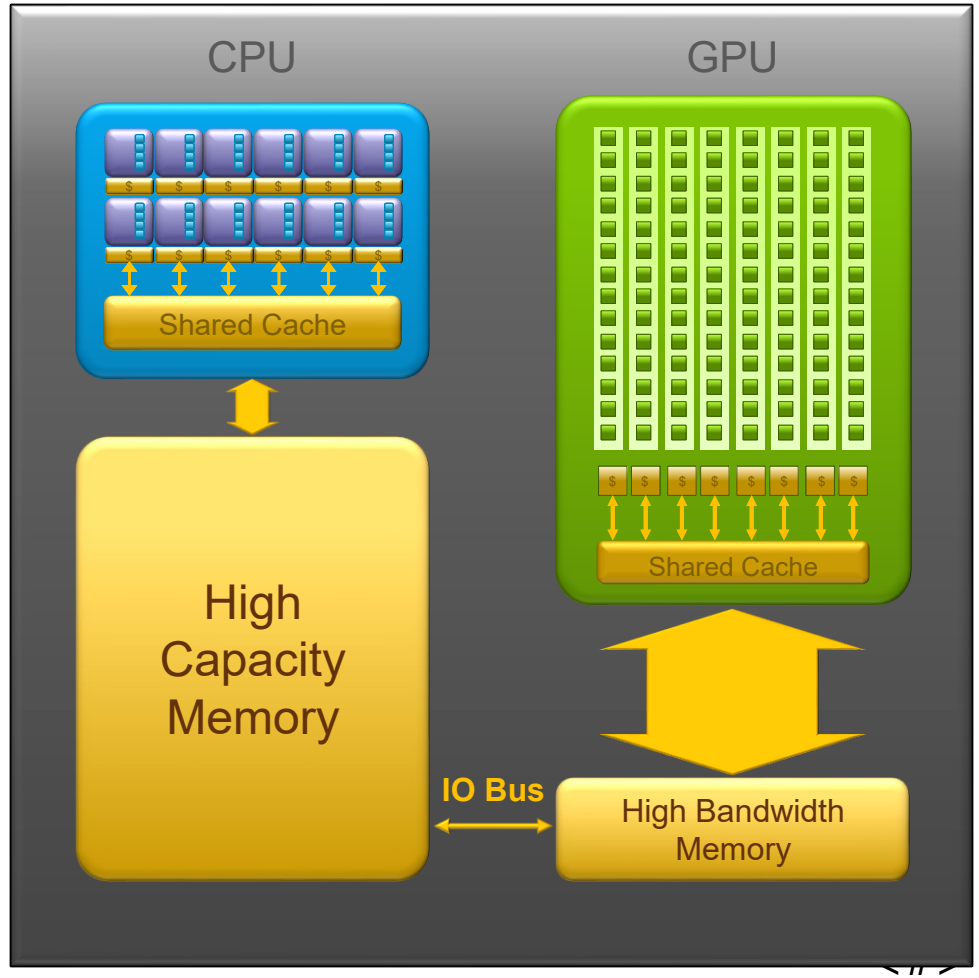
Single-Instruction Multiple-Threads

- NVIDIA CUDA GPUの並列化モデル
- SIMD (Single-Instruction Multiple-Data) と同一と捉えてよい
- 複数のスレッドが同じ命令（計算）を同時に実行する
- NVIDIA GPUはWarp単位で並列処理を実行する (1Warp = 32 Threads)

T0	T1	T2	T3	T4	T5	T6	T7
1	2	3	4	5	6	7	8
+	+	+	+	+	+	+	+
9	10	11	12	13	14	15	16
=	=	=	=	=	=	=	=
10	12	14	16	18	20	22	24

CPUとGPUの物理ダイアグラム

- CPUは高容量なメモリを持ち、GPUのメモリは高いバンド幅を持つ
- CPUとGPUのメモリは物理的に分けられ、一般的にはPCI-eをI/Oバスとして接続される
- I/OバスによりCPUとGPUの間でデータ転送が制御される
- I/Oバスのデータ転送速度はメモリバンド幅に対し相対的に遅い
- 必要なデータがメモリになければGPUは計算を行うことはできない

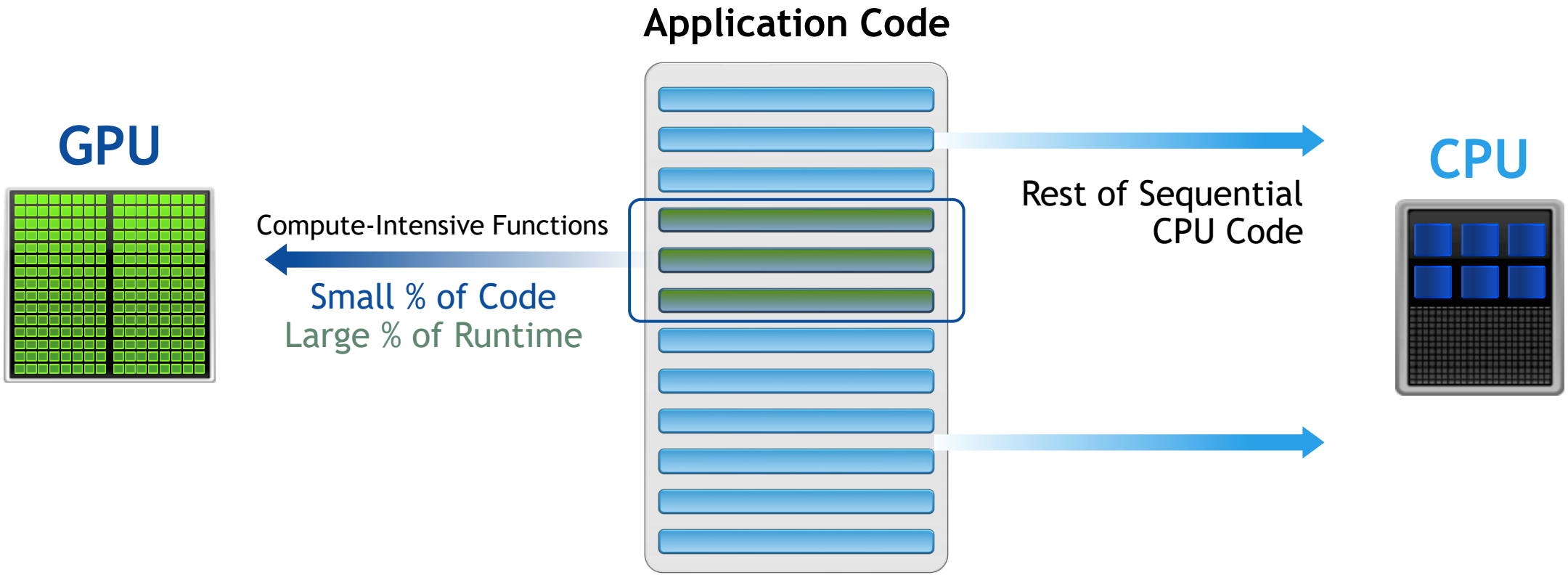


OpenACC
More Science, Less Programming

iii. GPUプログラミング

OpenACC
More Science, Less Programming

アプリケーションへのGPUの適用



GPU対応アプリケーションの実装方法

- **CUDA C++, CUDA Fortran**

  NVIDIA GPU用に開発された並列プログラミング言語

- **OpenMP accelerator model**

  OpenMP version 4.0から追加されたアクセラレータ向け拡張仕様

- **OpenACC**

  GPUを代表としたアクセラレータプログラムをディレクティブベースで実装

CUDA C++, CUDA Fortran

- どちらもGPU向けの並列言語
- CUDA C++：NVIDIAが実装した
- CUDA Fortran：PGIが実装した。その後、NVIDIAはPGIを買収し、NVIDIAネイティブな実装へ

```
module mymodule
contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x + (blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

nt = 128
call saxpy<<<(n+nt-1)/nt, nt>>>(n, a, x, y)
```

OpenMP accelerator model

- OpenMP version 4.0で導入された拡張仕様
- OpenMPはユーザーによる明示的な並列化を基本としているため、かなり細かく指示文を指定する必要がある
- 今回のターゲットであるNVIDIA HPC コンパイラも実装しているがOpenACCに比べて最適化が十分でない

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
!$omp target map(to:x(1:n)) &
              map(tofrom:y(1:n))
!$omp teams
!$omp distribute parallel do
  do i = 1, n
    y(i) = a*x(i) + y(i)
  end do
!$omp end teams
!$omp end target
end subroutine saxpy
```


OpenACC
More Science, Less Programming

OpenACC

- 2011年にversion 1.0が策定、OpenMPに近いインターフェイスでアクセラレータプログラミングが可能
- NVIDIA HPCコンパイラで利用できる
- 単一のコードによって、CPUとGPUの両方で実行でき、管理し易い
- CUDAやOpenMP accelerator modelに比べてハードウェア詳細を考えるコストが低い

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
!$acc kernels
  do i = 1, n
    y(i) = a*x(i) + y(i)
  end do
!$acc end kernels
end subroutine saxpy
```

CUDAとOpenACCの比較

- 性能：ふつうはOpenACCよりCUDAの方が良い

  （OpenACCで性能が出るかはアプリケーションに依る）

- 学習コスト： CUDAは高コストでOpenACCは低コスト
- 開発・管理コスト：
  - CUDA：NVIDIA GPUが搭載されていないシステムではCUDAで書かれたコードを利用できないため、CPUとGPU向けにコードが2つ必要で高コスト
  - OpenACC：コードは１つで良く、シンプルなAPIで低コスト
- 既存のMPI + OpenMP化されたコードがある場合の開発・管理コスト：
  - MPI + OpenMP + CUDAはプログラムが複雑になり高コスト
  - MPI + OpenMP + OpenACCは低コスト

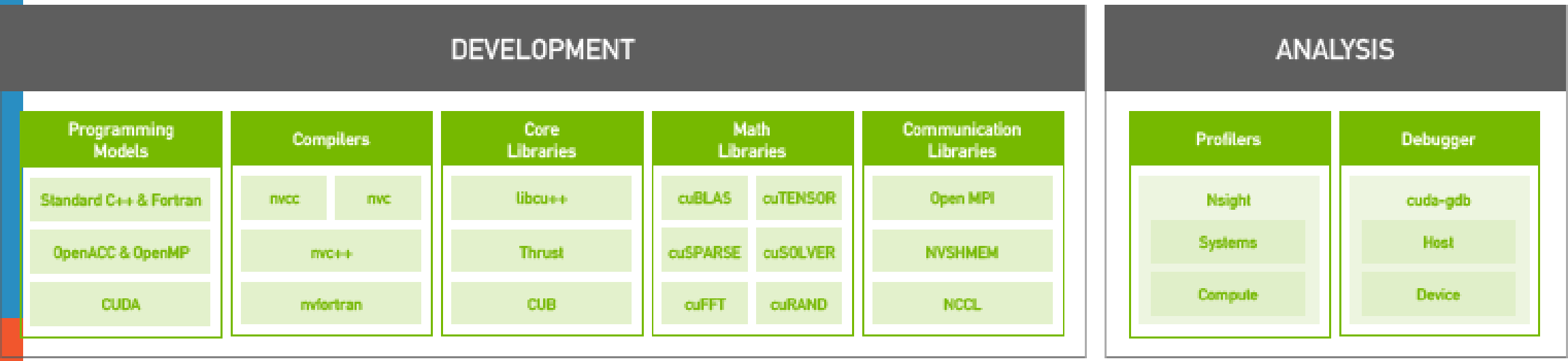
OpenACC
More Science, Less Programming

iv. NVIDIA HPC SDK

NVIDIA HPC SDK

A Comprehensive Suite of Compilers, Libraries and Tools for HPC

- コンパイラ、ライブラリ、プロファイラ、デバッガのセット
- CUDA SDK + Open MPI + NVIDIA HPCコンパイラ （旧PGIコンパイラ）



OpenACC
More Science, Less Programming

NVIDIA HPCコンパイラ

- 旧PGIコンパイラの新名称

旧PGIコンパイラ	バージョン20.4で開発終了
NVIDIA HPCコンパイラ	PGI 20.4をリネームし開発を継続、 バージョン25.5が最新

- 旧PGIコンパイラはNVIDIA V100 GPUまでをサポート
- NVIDIA A100 GPU以降の最適化はNVIDIA HPCコンパイラで実装される

PGI	pgc	pgc++	pgfortran
NVIDIA	nvc	nvc++	nvfortran

- サポートするCPU：x86_64, Arm

コンパイラがサポートする言語規格

nvc	ISO/ANSI C99/C11
nvc++	ISO/ANSI C++03/C++11/C++14/C++17/C++20/C++23
nvfortran	ISO/ANSI Fortran 2003 (2008の多くの機能)
OpenMP	Version 3.1, 4.5, 5.0 subsets
OpenACC	Version 2.7（一部を除く）

- Partial supportの場合もあるのでご注意ください

NVIDIA HPC SDKへの移行

- PGIはCommunity Edition (CE) を配布していた
- NVIDIA HPC SDKは基本的にフリーアクセス（ライセンスへの同意要）
- これまでのPGIコンパイラの技術を、最適化がさらに進んだ状態で使用できる
- ダウンロードURI：https://developer.nvidia.com/hpc-sdk

- 技術サポートは有償契約で提供。国内窓口は弊社：https://hpcworld.jp/support/

コンパイラの技術資料について

- 株式会社ソフテックが提供・管理されてきた旧PGIコンパイラの技術情報について弊グループが運営するHPC WORLDにてアーカイブを提供
- NVIDIA HPCコンパイラのオプションも基本的に旧PGIコンパイラと同じだが、一部変更・廃止されている
- 旧PGIコンパイラ技術情報のアーカイブ

  https://hpcworld.jp/pgi-compiler-archive/
- NVIDIA HPC SDKのドキュメント

  https://docs.nvidia.com/hpc-sdk/index.html
- SDKのリリースノートの訳

  https://hpcworld.jp/nvsdk_releasenotes/

2. OpenACCを使ったプログラムの並列化
i. OpenACCの概要

OpenACC
More Science, Less Programming

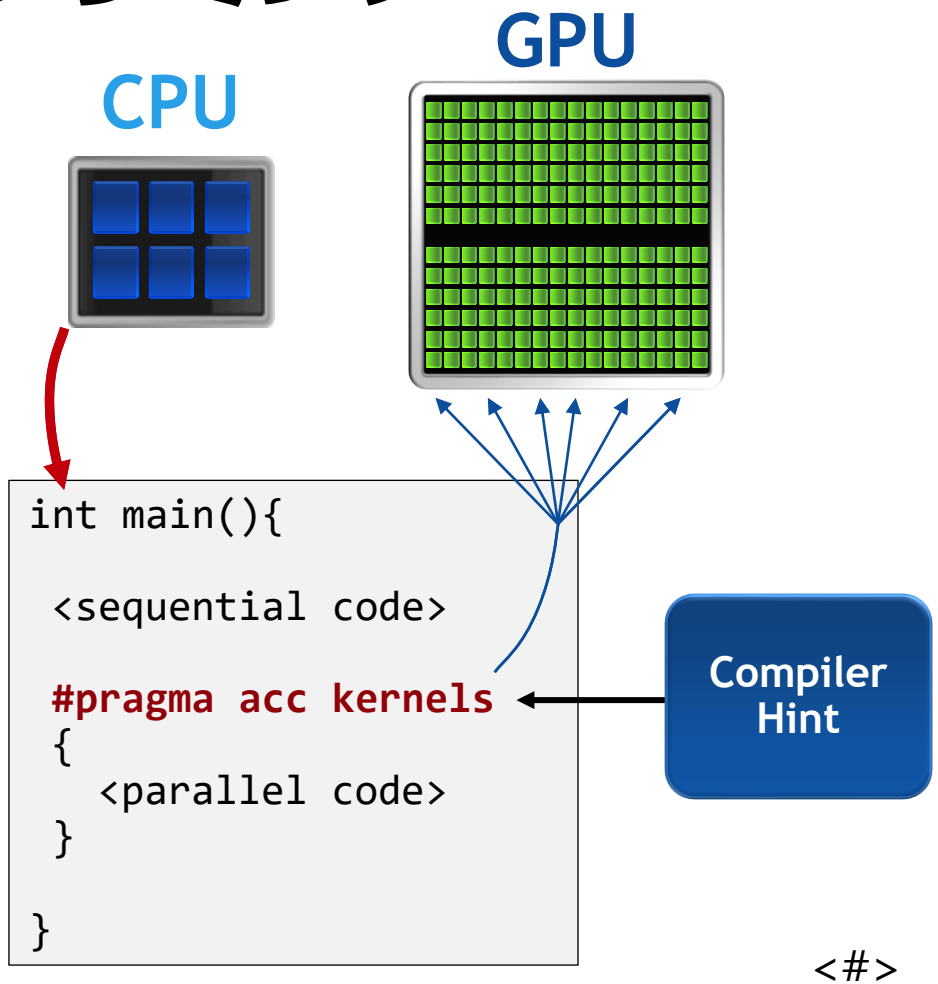
OpenACC is a directives-based programming approach to **parallel computing** designed for **performance** and **portability** on CPUs and GPUs for HPC.

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

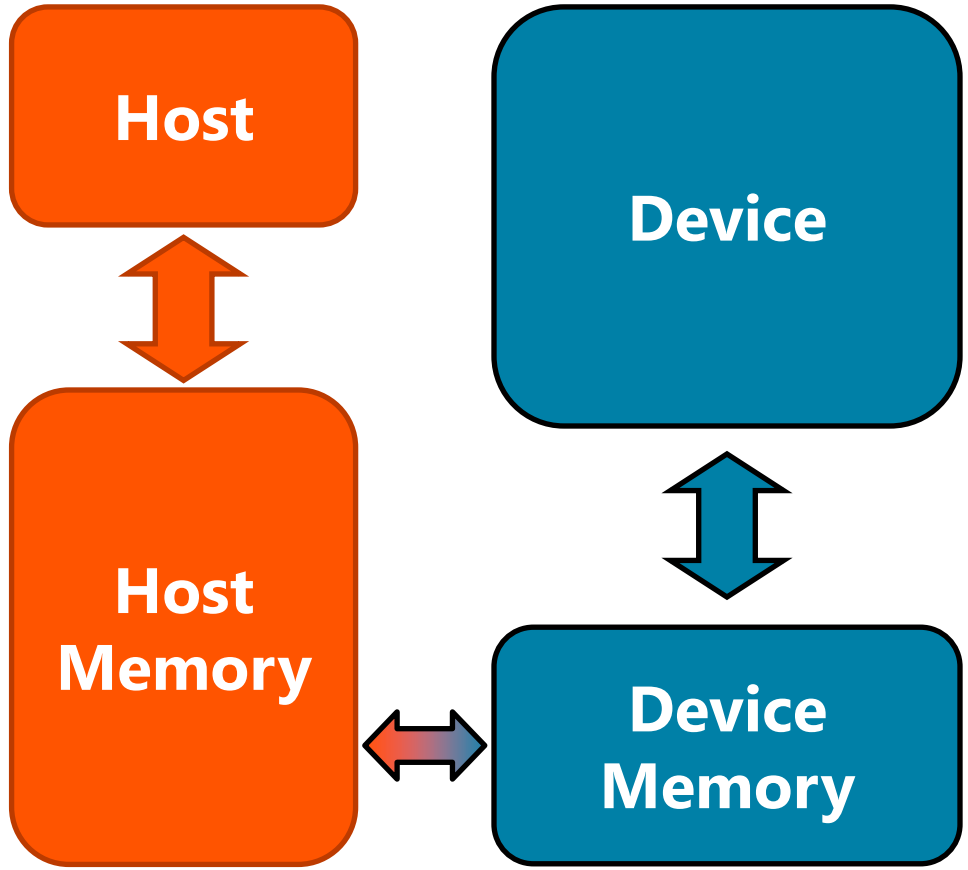
OpenACCを用いたGPUプログラミング

- プログラムは常にCPUで開始・終了する
- OpenACC指示文を用いて、Compute-intensiveなループの処理をGPUにオフロードする
- GPUへのオフロードに伴って、CPUとGPUの間でデータ転送が必要になる
- GPUの並列性および制御を活かせない処理は、CPUで実行する

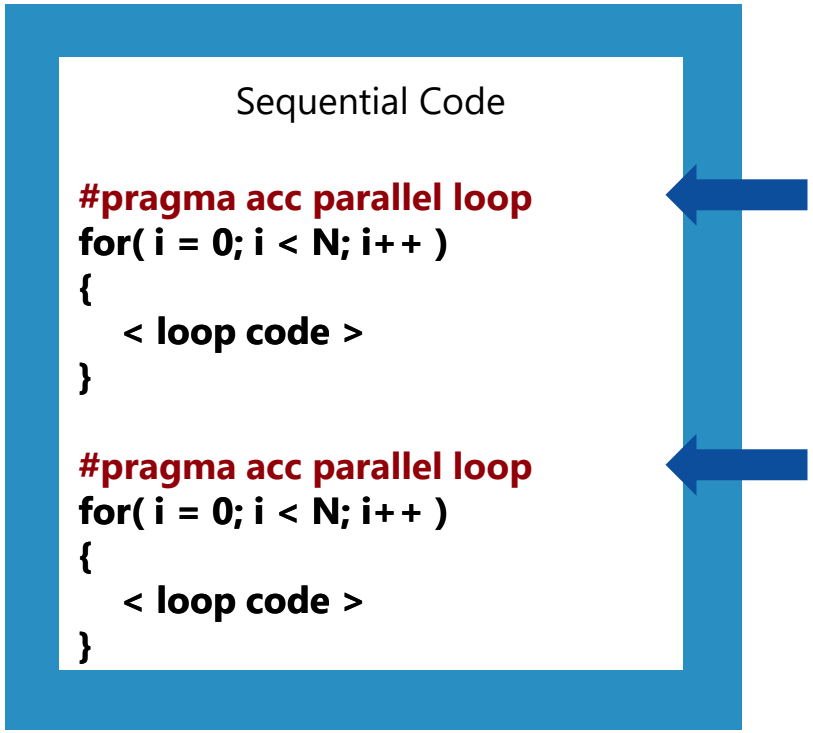


OpenACCの移植性

- GPUに限らず、多くの並列プラットフォームへの移植をサポートできるように設計されている
- ユーザーはハードウェア詳細を考えずに、汎化された言語で並列性を記述できる
- OpenACCはGPUなどの1つ以上の並列処理デバイス（GPU等）と、それを管理するホスト（通常はCPU）で実行される
- デバイスとホストは、論理的に分けられたメモリを別々に持っていると仮定される



OpenACCの利点 (incremental)



逐次処理コードを作成する

OpenACC指示文で並列化

コードが正しく動いたら、
必要に応じて修正し、最適化

OpenACCの利点 (single source)

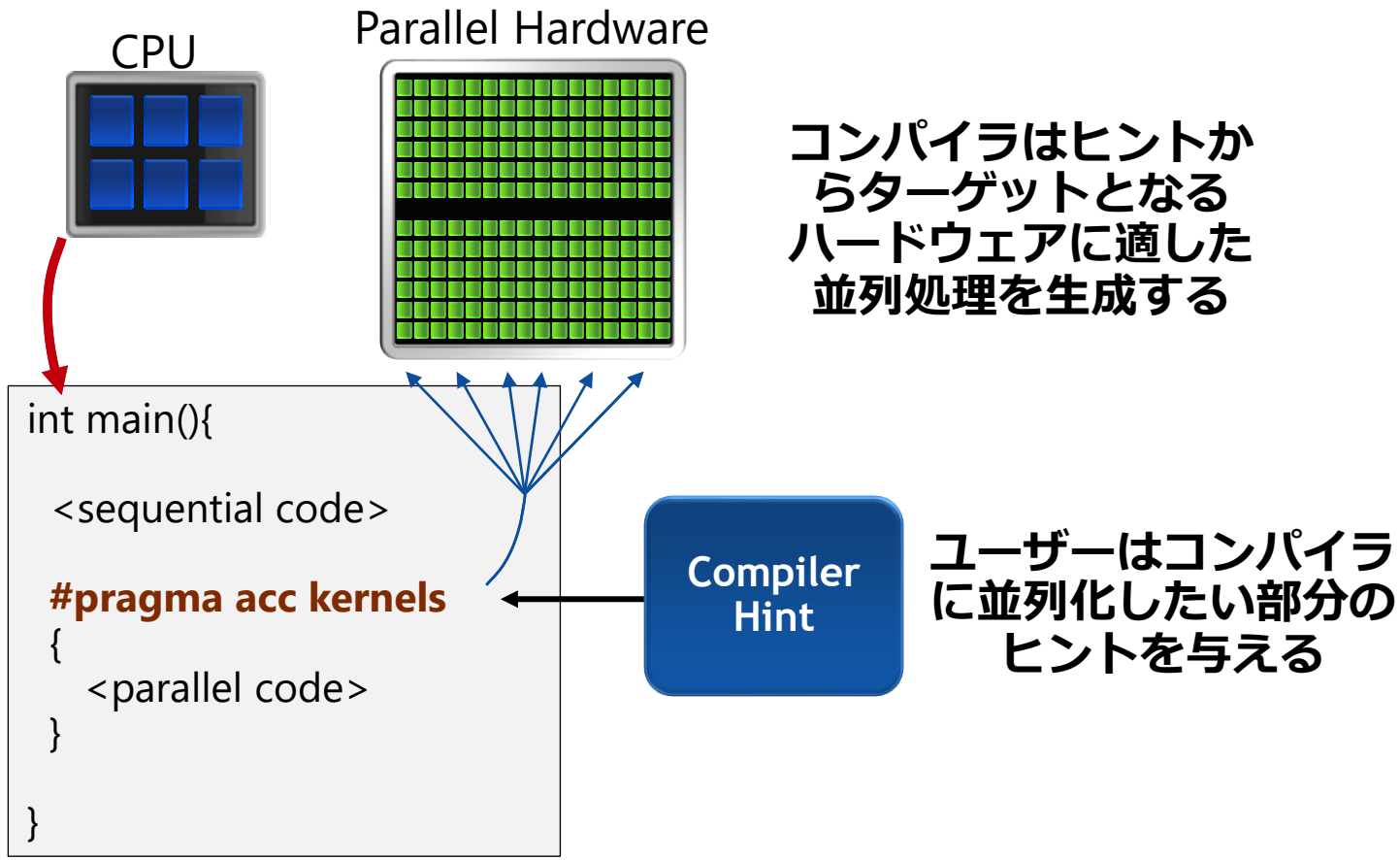
- コードを修正せず複数のハードウェアでリビルド
- コンパイラが目的のハードウェアに合わせて並列化方法を決定
- コンパイラは追加されたOpenACCコードを無視することもできるため、同じコードを並列処理にも逐次処理にも使える

```
int main(){

...

    #pragma acc parallel loop
    for(int i = 0; i < N; i++)
        < loop code >

}
```

OpenACC
More Science, Less Programming

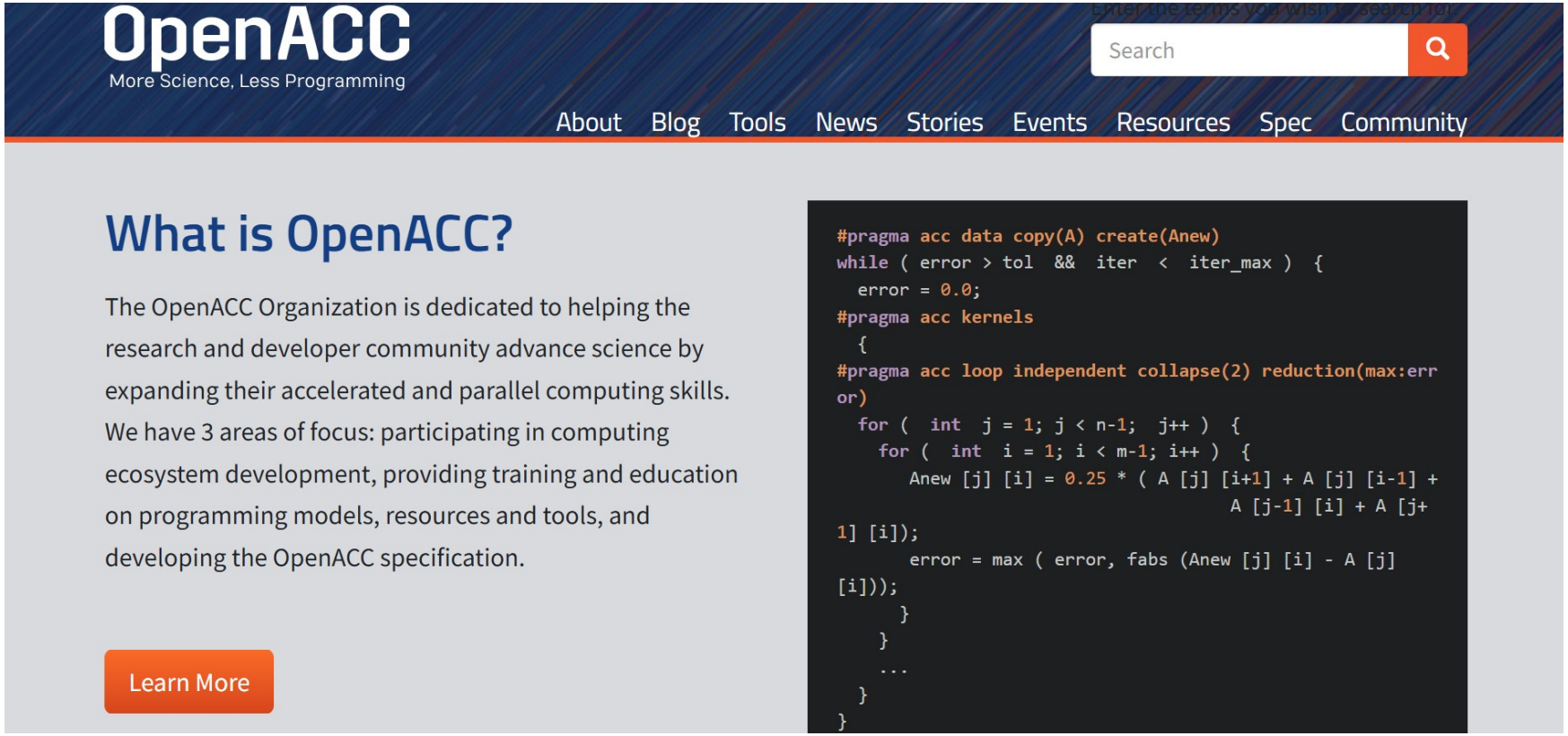
OpenACCの利点 (low learning curve)

- OpenACCは比較的容易に利用・学習できる
- ユーザーは使い慣れたC, C++, またはFortranをそのまま利用できる
- ハードウェアの深い知識を学ばずとも利用できる



OpenACC
More Science, Less Programming

Resources



https://www.openacc.org

- Tools
- Success Stories
- Events
- Resources
  - Guides
  - Tutorials
  - Courses
  - Code Samples
  - Talks
  - Books
- Teaching Materials
- Specification
- Community
  - Stack Overflow
  - Slack

ii. OpenACCの基本構文

OpenACCの基本構文

C, C++

```
#pragma acc directive clauses
<code>
```

Fortran

```
!$acc directive clauses
<code>
```

- **#pragma**: C, C++におけるプリプロセッサで、コードのコンパイル方法をコンパイラに指示する（コンパイラは自由に無視できる）
- **acc**: 後に続く文がOpenACC指示文であることをコンパイラに教える
- Fortranの場合は特別なフォーマットのコメントで**!$acc** のように書く（機能はCの #pragma accと同じ）
- **Directive（ディレクティブ、指示文）**: OpenACCでコードを変更するためのコマンド
- **Clauses（クローズ、節）**: directiveの指定や追加

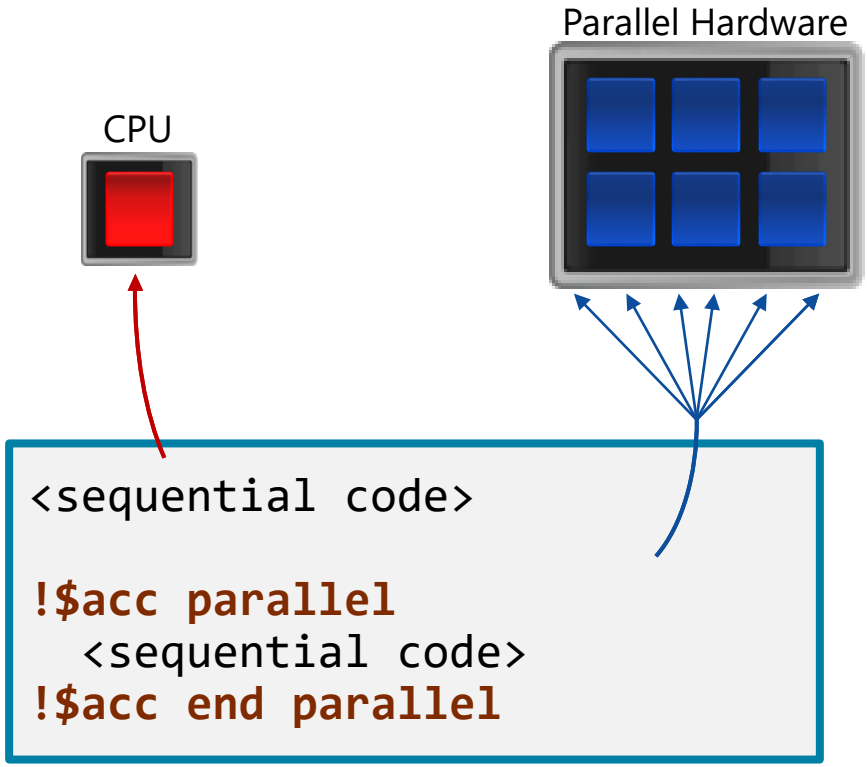
OpenACC
More Science, Less Programming

iii. Parallel指示文

OpenACC
More Science, Less Programming

Parallel指示文

明示的な並列化の指示



- **Parallel は、デバイス上にgang (並列集団) を生成するようにコンパイラに指示する**
- **gangはデバイス上のスレッド群の独立したグループ**
- **Parallel指示文に含まれるコードはすべてのgangによって冗長実行される**

OpenACC
More Science, Less Programming

Parallel指示文

単一ループの並列化

Fortran

```
!$acc parallel
!$acc loop
  do i = 1, N
    a(i) = 0
  end do
!$acc end parallel
```

C/C++

```
#pragma acc parallel
{
#pragma acc loop
  for(int i = 0; i < N; i++)
    a[i] = 0;
}
```

- 並列実行させたい領域についてparallel指示文を指定する
- 並列化される領域はC, C++では中括弧、Fortranでは!$accおよび!$acc endで指定する
- Loop指示文はすぐ下の行のループを並列化し、gangsをまたがって実行するようコンパイラに指示する

OpenACC
More Science, Less Programming

Parallel指示文

単一ループの並列化（略記方法）

Fortran

```
!$acc parallel loop
  do i = 1, N
    a(i) = 0
  end do
```

C/C++

```
#pragma acc parallel loop
  for(int i = 0; i < N; i++)
    a[i] = 0;
```

- 左の例はシンプルかつ一般的な並列化方法で、1行のparallel loop指示文により、すべての並列化指示が可能
- parallel loop指示文により、並列実行領域をマークすると同時に、ループの並列化指示を行うことが可能
- OpenMPと同じマナー（並列化の明示）
- データ依存関係があるループに適用した場合は誤った計算を行う場合がある

OpenACC
More Science, Less Programming

Parallel指示文

複数のループの並列化

Fortran

```
!$acc parallel loop
  do i = 1, N
    a(i) = 0

!$acc parallel loop
  do j = 1, M
    b(j) = 0
```

- 複数ループを並列化する場合、各ループにparallel指示文の記述が必要
- 各並列ループは異なるループサイズとループ最適化方法を指示できる
- 各並列ループは異なる方法で並列化を指示できる
- 1つの並列領域に1つの並列ループを書くのを推奨

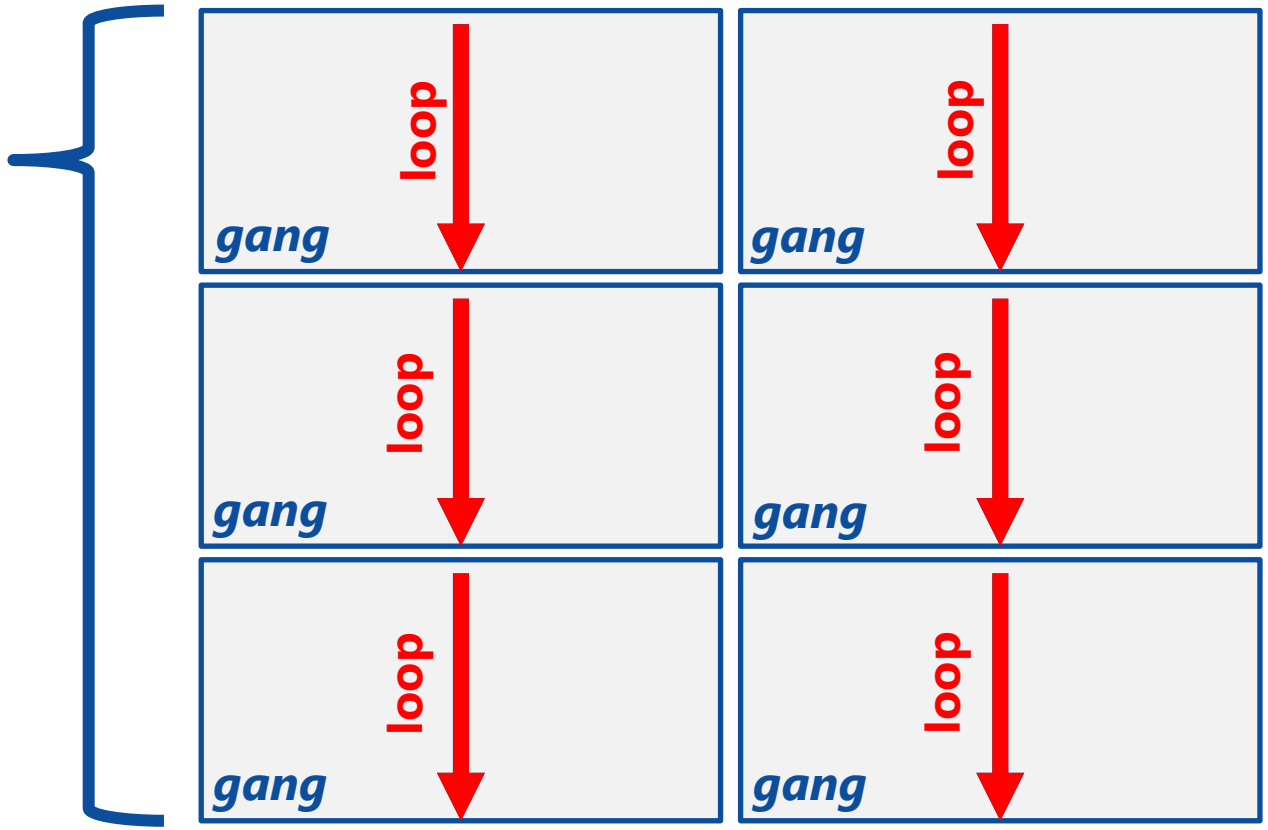
OpenACC
More Science, Less Programming

Parallel指示文

図を使った説明

Fortran

```
!$acc parallel

  do i = 1, N
    a(i) = 0
  end do
!$acc end parallel
```



Parallel指示文が記述されると、コンパイラは1つ以上のgangを生成し、各gangはループ全体を冗長実行する

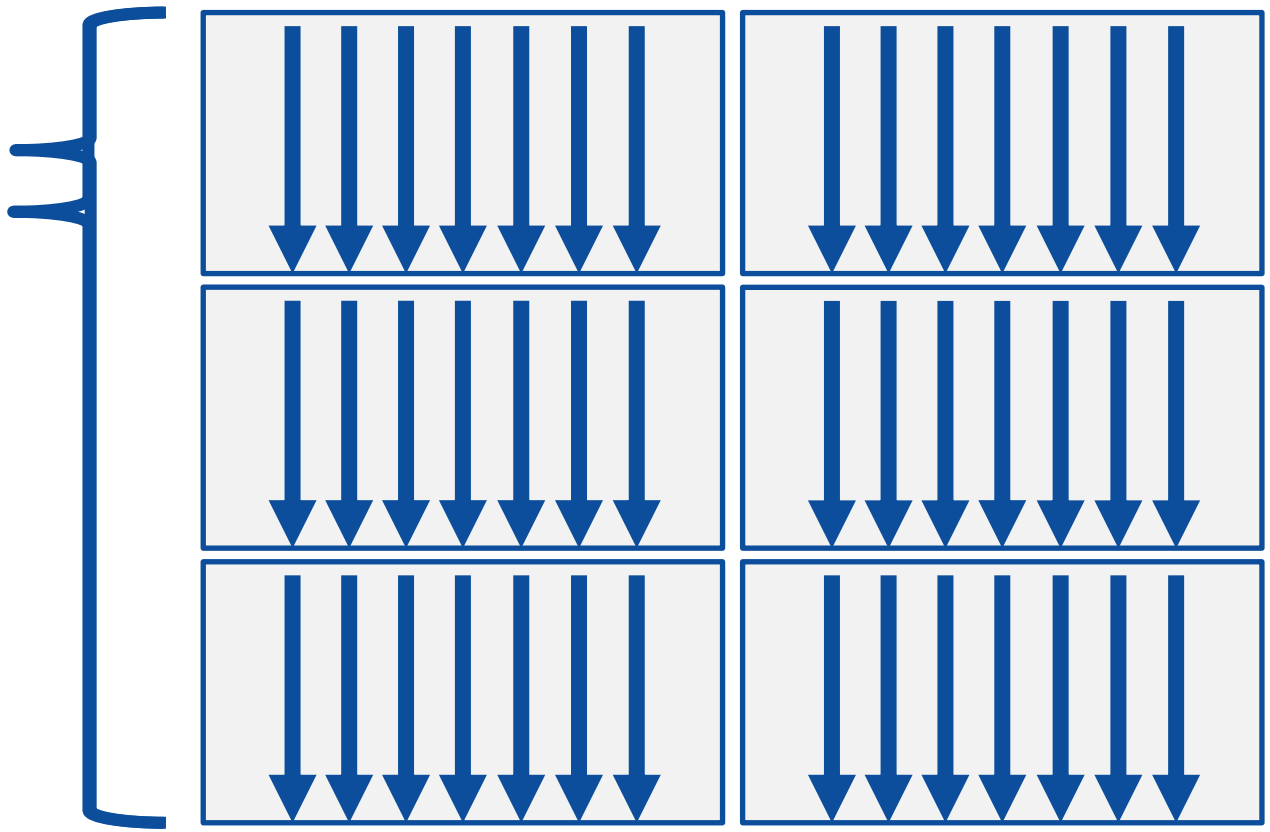
OpenACC
More Science, Less Programming

Parallel指示文

図を使った説明

Fortran

```
!$acc parallel
!$acc loop
  do i = 1, N
    a(i) = 0
  end do
!$acc end parallel
```



loop指示文を記述すると、ループの並列化をコンパイラへ指示し、ループが並列に実行される

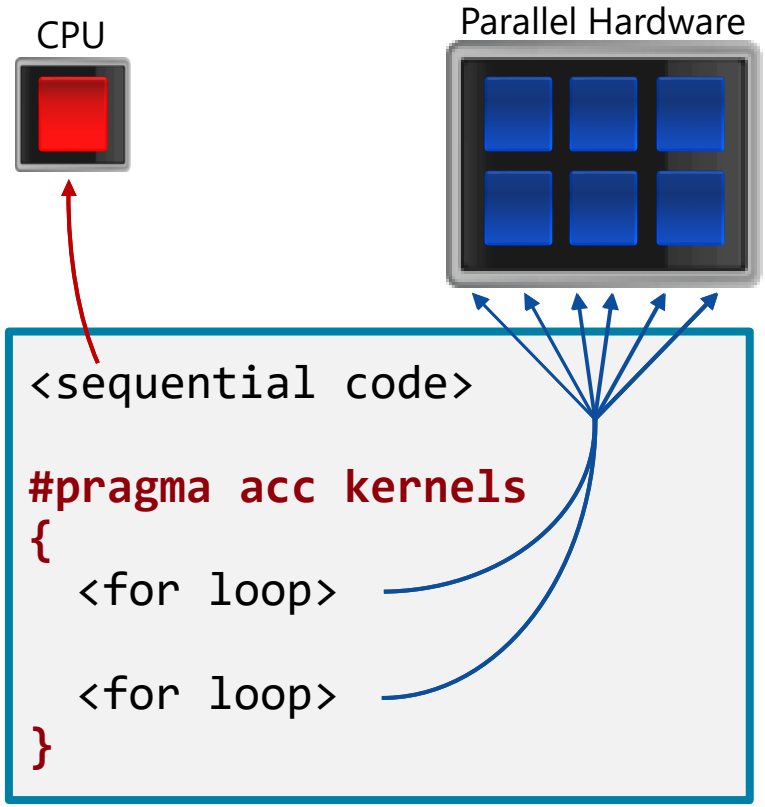
OpenACC
More Science, Less Programming

iv. Kernels指示文

OpenACC
More Science, Less Programming

Kernels指示文

コンパイラによる半自動的な並列化



- **kernels**指示文はコード中の並列ループを検索するようにコンパイラに指示する
- コンパイラは、ループを分析して安全かつ性能向上が見込まれるループを並列化する

OpenACC
More Science, Less Programming

Kernels指示文

単一ループの並列化

Fortran

```
!$acc kernels
  do i = 1, N
    a(i) = 0
  end do
!$acc end kernels
```

C/C++

```
#pragma acc kernels
  for(int i = 0; i < N; i++)
    a[i] = 0;
```

- kernels指示文をdo（またはfor） ループに適用
- コンパイラは、指示されたループに対して、並列リソース上での並列化とループ最適化を試みる
- コンパイラが並列化できないと判断した場合、並列化なしでコンパイルが行われる

OpenACC
More Science, Less Programming

Kernels指示文

複数ループの並列化

Fortran

```
!$acc kernels
  do i = 1, N
    a(i) = 0
  end do

  do j = 1, M
    b(j) = 0
  end do
!$acc end kernels
```

C/C++

```
#pragma acc kernels
{
  for(int i = 0; i < N; i++)
    a[i] = 0;

  for(int j = 0; j < M; j++)
    b[j] = 0;
}
```

- kernels指示文は複数ループがある計算領域（コードブロック）にまとめて適用することも可能
- コードブロックはparallel指示文同様、C, C++では中括弧、Fortranでは!$accおよび!$acc endで指定する
- 単一ループ指定と同様、コンパイラはコードブロック内のすべてのループに対して、個別に並列化と最適化を試みる

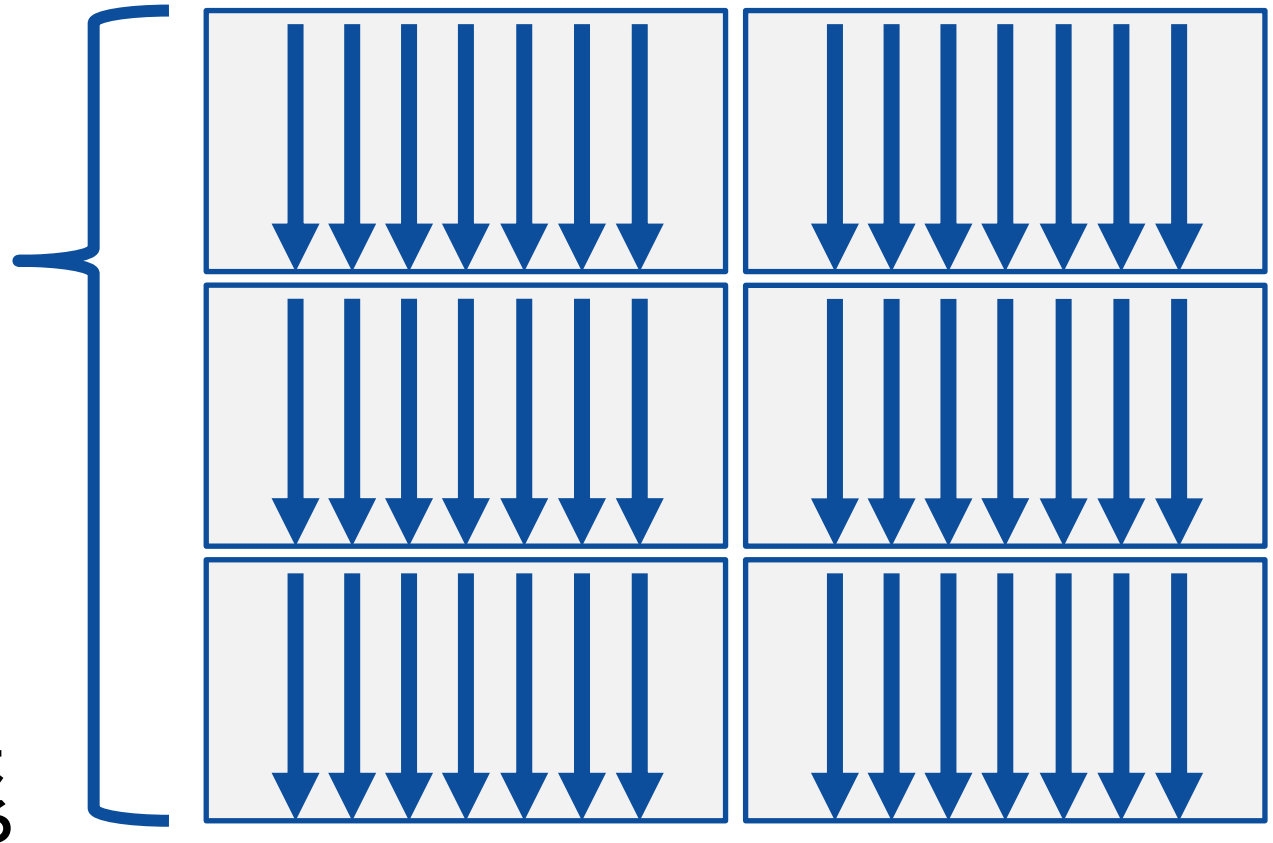
OpenACC
More Science, Less Programming

Kernels指示文

図を使った説明

Fortran

```
!$acc kernels
  do i = 1, N
    a(i) = 0
  end do

  do j = 1, M
    b(j) = 0
  end do
!$acc end kernels
```



- kernels指示文は暗黙的にloop指示文を指定する
- 各並列ループに異なるサイズのgangが割り当てられ、各gangは別々に並列化と最適化が行われる

OpenACC
More Science, Less Programming

Kernels指示文

Fortran配列構文の並列化

Fortran

```
!$acc kernels
a(:) = 1
b(:) = 2
c(:) = a(:) + b(:)
!$acc end kernels
```

Fortran

```
!$acc parallel loop
c(:) = a(:) + b(:)
```

- kernels指示文はparallel指示文と異なり、コードブロックを対象とできるので、Fortranのarray notation（配列構文）の並列化も可能
- parallel指示文は、loop指示文と対である必要があるが、loop指示文はFortran配列構文を並列化可能なループとして認識できない

OpenACC
More Science, Less Programming

ParallelとKernelsの比較

Parallel

- ユーザーが明示的に並列化内容を決定し、コンパイラに指示する
- ユーザーが結果を保証
- 複数ループは別々に並列化する
- Fortran配列構文は並列化できない

Kernels

- ユーザーが指示し、コンパイラが半自動的に並列化対象を決める
- コンパイラが結果を保証
- 複数ループをまとめて並列化できる
- Fortran配列構文も並列化できる

十分な最適化が行われていれば、どちらも同程度の性能が得られる

OpenACC
More Science, Less Programming

v. Loop指示文

Loop指示文

Fortran

```
!$acc loop
do i = 1, N
  a(i) = 0
end do
```

C/C++

```
#pragma acc loop
for(int i = 0; i < N; i++)
  a(i) = 0;
```

- 単一ループの並列化を指示
- ループに関する情報や最適化をコンパイラに与えることができる
- OpenMP同様、ループの並列化方法（reduction等）を節で指定する
- ループを並列化するためにはOpenACCの並列領域 (parallelまたはkernels領域) に含まれている必要がある

OpenACC
More Science, Less Programming

Loop指示文

parallel指示文内のループの並列化

Fortran

```
!$acc parallel

  do i = 1, N
    a(i) = 0
  end do

  !$acc loop
  do j = 1, M
    b(j) = 0
  end do

!$acc end parallel
```

- 最初のループは、ループ全体が並列リソース（例えばスレッド）上で冗長に並行実行される
- 2つ目のループは、ループの各反復処理が適切に分割され並列リソース上で並行実行される

OpenACC
More Science, Less Programming

Loop指示文

kernels指示文内のループ並列化

Fortran

```
!$acc kernels

  !$acc loop
  do i = 1, N
    a(i) = 0
  end do

  !$acc loop
  do j = 1, M
    b(j) = 0
  end do

!$acc end kernels
```

- kernels指示文ではloopディレクティブが暗黙に定義される
- ユーザーはloop指示文によりループ並列化を明示できるが、コンパイラの最適化に影響を与えることがある
- loop指示文は不要だが、ループ自体の最適化はユーザーができる

OpenACC
More Science, Less Programming

Loop指示文

多重ループの並列化

Fortran

```
!$acc parallel loop
do j = 1, M
  !$acc loop
  do i = 1, N
    a(i,j) = 0
  end do
end do
```

C/C++

```
#pragma acc parallel loop
for(int i = 0; i < N; i++){
  #pragma acc loop
  for(int j = 0; j < M; j++){
    a[i][j] = 0;
  }
}
```

- 多重ループの並列化も可能。各ループに対してloop指示文を書く
- CUDA GPUでは多次元の並列性を持つため、この機能により性能向上が期待される
- （NVIDIA GPUでない）1次元の並列性しか持たない従来のマルチコアプロセッサをターゲットとする場合、内側のloop指示文を無視してもよいことになっている

OpenACC
More Science, Less Programming

vi. OpenACCコードのコンパイル

OpenACCコードのコンパイル (NVIDIA, PGI)

CODE

```
7 : !$acc parallel loop
8 : do i = 1, N
9 :   a(i) = 0
10: end do
```

CPUスレッド並列でのコンパイル方法

COMPILING

```
$ nvfortran -fast -acc=multicore -Minfo=accel main.f90
$ pgfortran -fast -acc -ta=multicore -Minfo=accel main.f90
```

FEEDBACK

```
main:
     7,  Generating Multicore code
         8, !$acc loop gang
```

OpenACC
More Science, Less Programming

OpenACCコードのコンパイル (NVIDIA, PGI)

CODE

```
7 : !$acc kernels loop
8 : do i = 1, N
9 :   a(i) = 0
10: end do
```

CPUスレッド並列でのコンパイル方法

COMPILING

```
$ nvfortran -fast -acc=multicore -Minfo=accel main.f90
$ pgfortran -fast -acc -ta=multicore -Minfo=accel main.f90
```

FEEDBACK

```
main:
     8,  Loop is parallelizable
         Generating Multicore code
          8, !$acc loop gang
```

OpenACC
More Science, Less Programming

OpenACCコードのコンパイル (NVIDIA, PGI)

CODE

```
7 : !$acc parallel loop
8 : do i = 2, N
9 :   a(i) = a(i) + a(i-1)
10: end do
```

※逐次実行と並列実行で結果が変わってしまう（並列化できない）ループの場合

COMPILING

```
$ nvfortran -fast -acc=multicore -Minfo=accel main.f90
$ pgfortran -fast -acc -ta=multicore -Minfo=accel main.f90
```

FEEDBACK

```
main:
     7,  Generating Multicore code
         8, !$acc loop gang
```

並列化されてしまっている！

OpenACC
More Science, Less Programming

OpenACCコードのコンパイル (NVIDIA, PGI)

CODE

```
7 : !$acc kernels loop
8 : do i = 2, N
9 :   a(i) = a(i) + a(i-1)
10: end do
```

※逐次実行と並列実行で結果が変わってしまう（並列化できない）ループの場合

COMPILING

```
$ nvfortran -fast -acc=multicore -Minfo=accel main.f90
$ pgfortran -fast -acc -ta=multicore -Minfo=accel main.f90
```

FEEDBACK

```
main:
     8,  Loop carried dependence of a prevents parallelization
         Loop carried backward dependence of a prevents vectorization
```

並列化されない

OpenACC
More Science, Less Programming

NVIDIA A100 GPUで動作させる場合

CODE

```
7 : !$acc parallel loop
8 : do i = 1, N
9 :   a(i) = 0
10: end do
```

COMPILING

```
$ nvfortran -fast -acc -gpu=cc80 -Minfo=accel main.f90
$ pgfortran -fast -acc -ta=tesla,cc80 -Minfo=accel main.f90
```

FEEDBACK

```
main:
     7,  Generating Tesla code
         8, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
     7,  Generating implicit copyout(a(1:1024)) [if not already present]
```

OpenACC
More Science, Less Programming

補足: NVIDIA P100, V100で動作させる場合

COMPILING

```
# for NVIDIA P100
$ nvfortran -fast -acc -gpu=cc60 -Minfo=accel main.f90
$ pgfortran -fast -acc -ta=tesla,cc60 -Minfo=accel main.f90

# for NVIDIA V100
$ nvfortran -fast -acc -gpu=cc70 -Minfo=accel main.f90
$ pgfortran -fast -acc -ta=tesla,cc70 -Minfo=accel main.f90
```

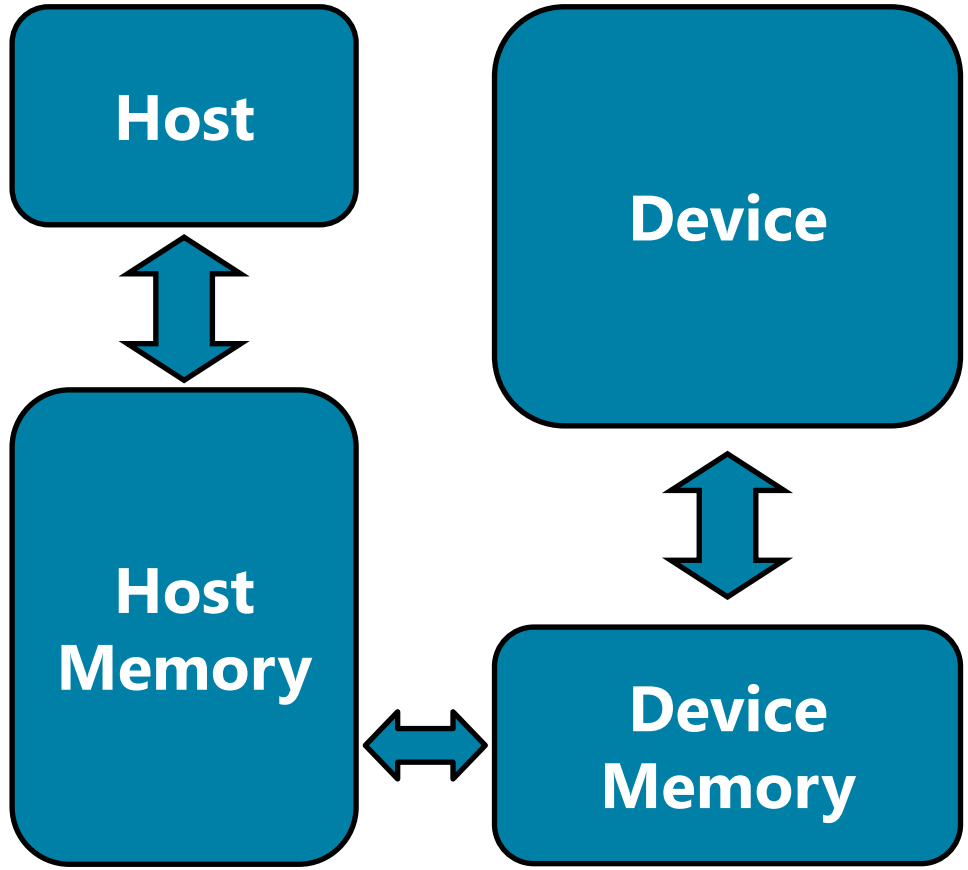
3. CPU-GPU間のデータ転送の最適化
i. メモリ管理の概要

OpenACC
More Science, Less Programming

基本的なメモリ管理の考え方

ターゲットがマルチコアの場合

- **Host**: 一般的にCPU
- **Device**: GPUを代表とする何らかのアクセラレータ
- OpenACCのターゲットとなるハードウェアがマルチコアの場合、Host/Deviceは同一、当然メモリも同一で、共有メモリ型のアクセラレータを使う限り、明示的なメモリ管理は不要

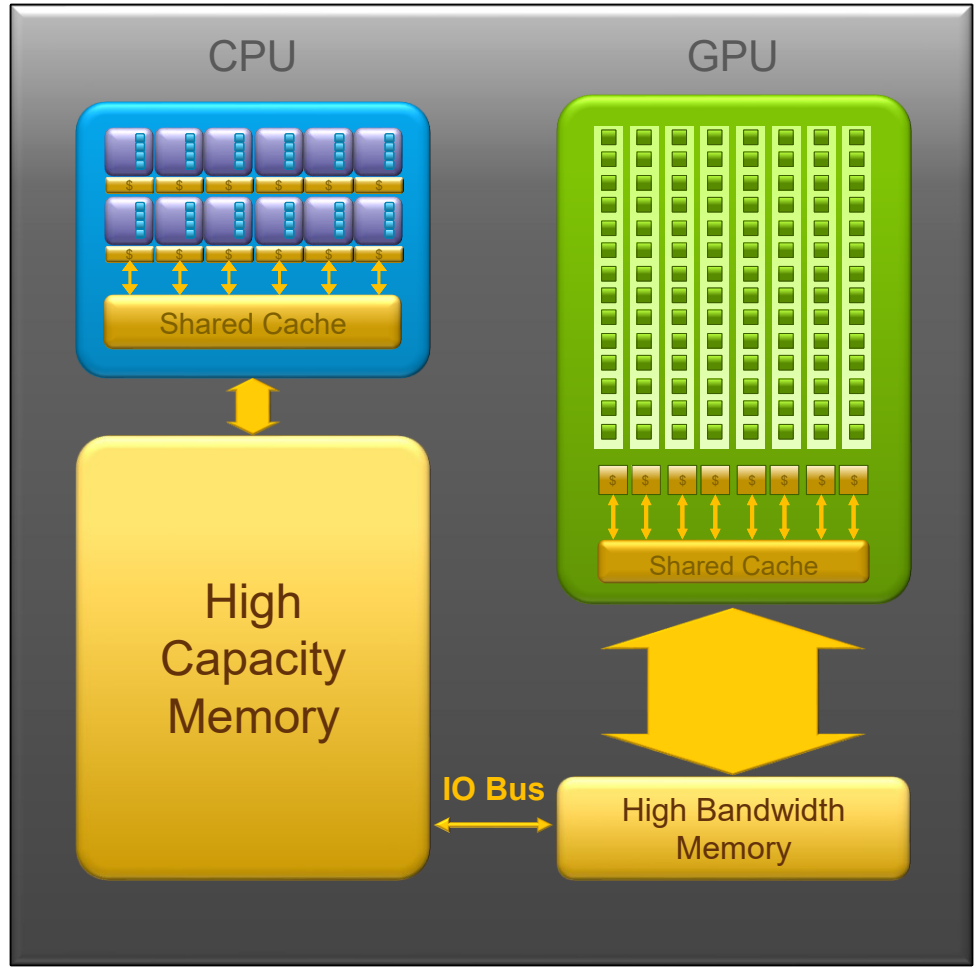


OpenACC
More Science, Less Programming

基本的なメモリ管理の考え方

ターゲットがGPUの場合

- OpenACCのターゲットがGPUの場合、計算に必要なデータをCPUとGPUのメモリ間でやりとりする必要がある



メモリ管理の種類

	V24.5より前	V24.5以降
Unified memory (+GH200等のアーキテクチャ)	-gpu=unified	-gpu=mem:unified
Managed memory	-gpu=managed	-gpu=mem:managed
Pinned memory	-gpu=pinned	-gpu=mem:pinned

- Parallel/Kernels指示文（＋data節）：暗黙的なメモリ領域を持つ
- data指示文：メモリ領域を明示する構造的な指示文
- Enter data/exit data指示文：非構造的な指示文
- Update指示文：データ同期に使う

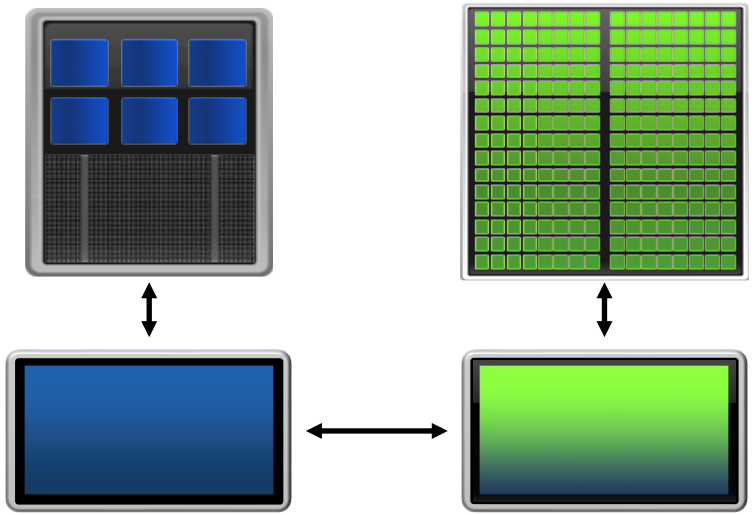
OpenACC
More Science, Less Programming

ii. Managed memory

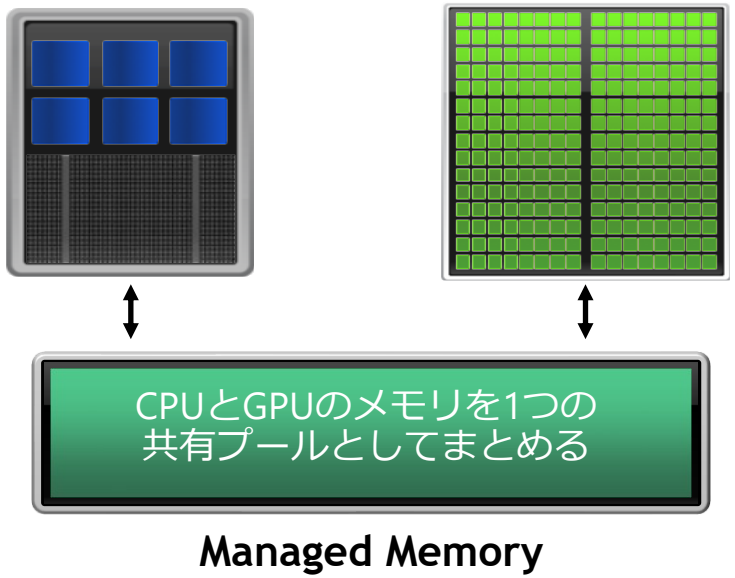
CUDA managed memory

開発コストを軽減する仕組みとして

Without Managed Memory



With Managed Memory



OpenACC
More Science, Less Programming

Managed memory

- CPU/GPUのメモリが統一されているようかのように振る舞う機能（managed memory）を使い、CPU-GPU間のデータ転送を自動化できる
- CUDA Managed MemoryはOpenACCでも利用できる
- Host-Device (CPU-GPU) 間の明示的なメモリ管理は複雑だが、それを一旦忘れることができるため、ループの並列化の作業に集中できる
- Managed memoryはdeep copyのデータも扱える

Fortran

```
$ nvfortran -fast -acc -gpu=mem:managed -Minfo=accel main.f90
```

C/C++

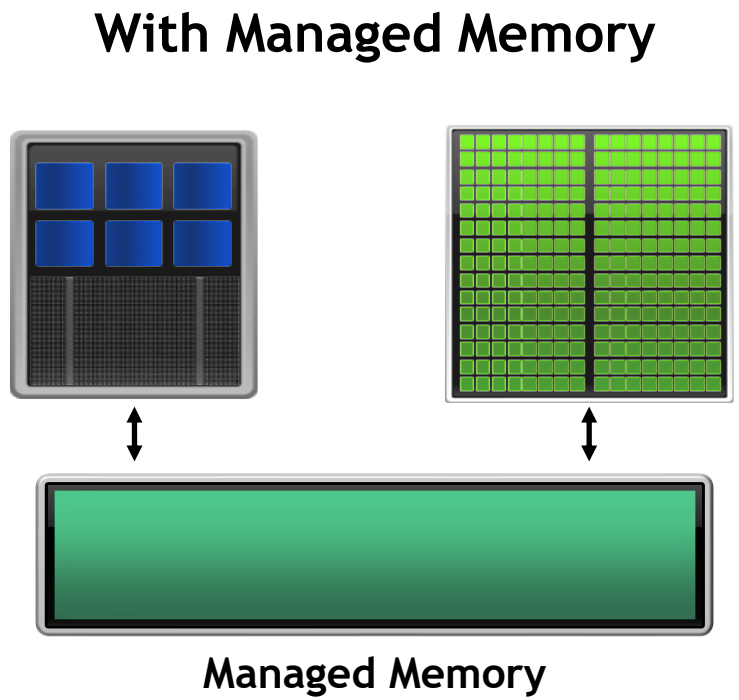
```
$ nvc -fast -acc -gpu=mem:managed -Minfo=accel main.c
```

※NVIDIA HPC SDK 24.5から、-gpu=mem:managedが導入されました。
最新バージョン25.5では、-gpu=managedは非推奨ですのでご注意ください。

OpenACC
More Science, Less Programming

Managed memory
制限事項

- NVIDIA (PGI) コンパイラのみの機能
- Allocate（やmalloc）でヒープ領域に動的確保されたメモリのみ使える
- メモリ割当と解放のオーバーヘッドが大きい
- データ転送が自動化されるため、明示的な非同期データ転送はできない
- 後述の明示的なメモリ管理をすることにより、より高い性能を得られる



Managed memory→データ転送コスト削減へ

C/C++

```
while ( error > tol && iter < iter_max )
{
  error = 0.0;
  #pragma acc kernels
  { // memcpy Host To Device (without Managed memory)
    for( int j = 1; j < n-1; j++)
    {
      for( int i = 1; i < m-1; i++ )
      {
        Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                            + A[j-1][i] + A[j+1][i]);
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));
      }
    }

    for( int j = 1; j < n-1; j++)
    {
      for( int i = 1; i < m-1; i++ )
      {
        A[j][i] = Anew[j][i];
      }
    }
  } // memcpy Device To Host (without Managed memory)
}
```

- Managed memoryを使わない場合、コンパイラは配列A, Anewのサイズを指定し、コード整合性のため、while文の各反復でCPU-GPU間のデータコピーを行う
- Managed memoryが有効な場合、システムプロセスのレベルで必要な場合にデータコピーを行う
- →データ転送のコストを減らす最適化へ

OpenACC
More Science, Less Programming

iii. Data節

基本的なメモリ管理

Host/Device間の明示的なデータコピー

- **Data節**でコンパイラに移動させたいデータとそのタイミングを指示する
- Data節はkernels/parallel指示文内だけでなく、後述の**data, enter data/exit data指示文**にも追加できる

Fortran

```
!$acc parallel loop
do i = 1, N
  a(i) = 0
end do
```

OpenACC
More Science, Less Programming

基本的なメモリ管理

Host/Device間の明示的なデータコピー

- **Data節**でコンパイラに移動させたいデータとそのタイミングを指示する
- Data節はkernels/parallel指示文内だけでなく、後述の**data, enter data/exit data指示文**にも追加できる

Fortran

```
!$acc parallel loop copyout(a(1:N))
do i = 1, N
  a(i) = 0
end do
```

配列aの初期値は必要ないため、最後にDeviceからHostにコピーするだけ

OpenACC
More Science, Less Programming

基本的なメモリ管理

Host/Device間の明示的なデータコピー

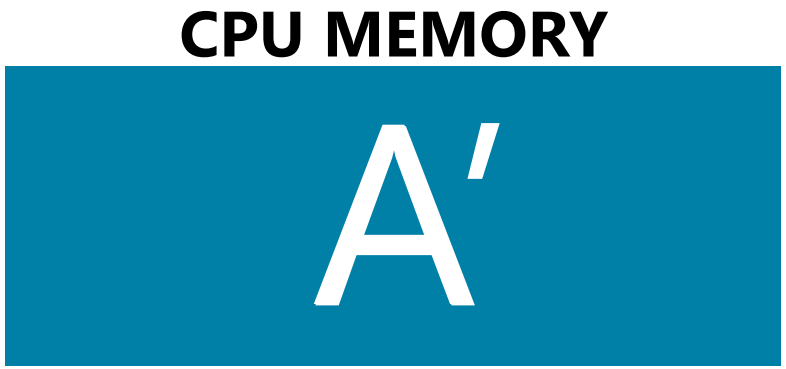
Fortran

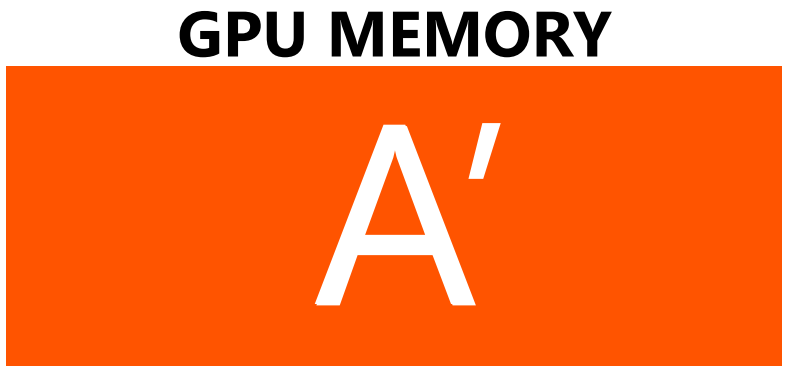
```
!$acc parallel loop copy(a(1:N))
do i = 1, N
  a(i) = 2 * a(i)
end do
```

OpenACC
More Science, Less Programming

基本的なメモリ管理

Host/Device間の明示的なデータコピー





OpenACC
More Science, Less Programming

Data節

`copy( list )`	Deviceにメモリを割り当て、並列領域に入るときはHostからDeviceにデータをコピー、出るときはHostにデータをコピーする 主な用途: コード中の重要なデータ構造について、Deviceに対するデータの入力・変更・返却を行うための最も基本的な処理節
`copyin( list )`	並列領域に入るときに、Deviceにメモリを割り当てHostからDeviceへのデータコピーを行う 主な用途: サブルーチンの入力配列のようなものと考える
`copyout( list )`	Deviceにメモリを割り当て、並列領域を出るときにデータをHostにコピーする 主な用途: 入力データを上書きせずにDeviceの計算結果を受け取る
`create( list )`	Deviceにメモリを割り当てるが、Hostとのコピーは行わない 主な用途: 一時的な作業用配列の確保

OpenACC
More Science, Less Programming

配列形状の指定方法

- コンパイラが配列形状を認識できるように情報を与える
- 最初の数字は配列の開始位置（インデックス）
- C/C++: 2番目の数字にはデータサイズを記載
- Fortran: 2番目の数字には配列の終了位置（インデックス）を記載

Fortran

```
copy(array(starting_index:ending_index))
```

C/C++

```
copy(array[starting_index:length])
```

OpenACC
More Science, Less Programming

配列形状の指定方法

多次元の場合の配列形状指定の例

Fortran

```
copy(array(1:N, 1:M))
```

C/C++

```
copy(array[0:N][0:M])
```

どちらも二次元配列をDeviceにコピーしている

OpenACC
More Science, Less Programming

配列形状の指定方法

部分配列の指定の例

Fortran

```
copy(array(N/4:N/4+N/4))
```

C/C++

```
copy(array[N/4:N/4])
```

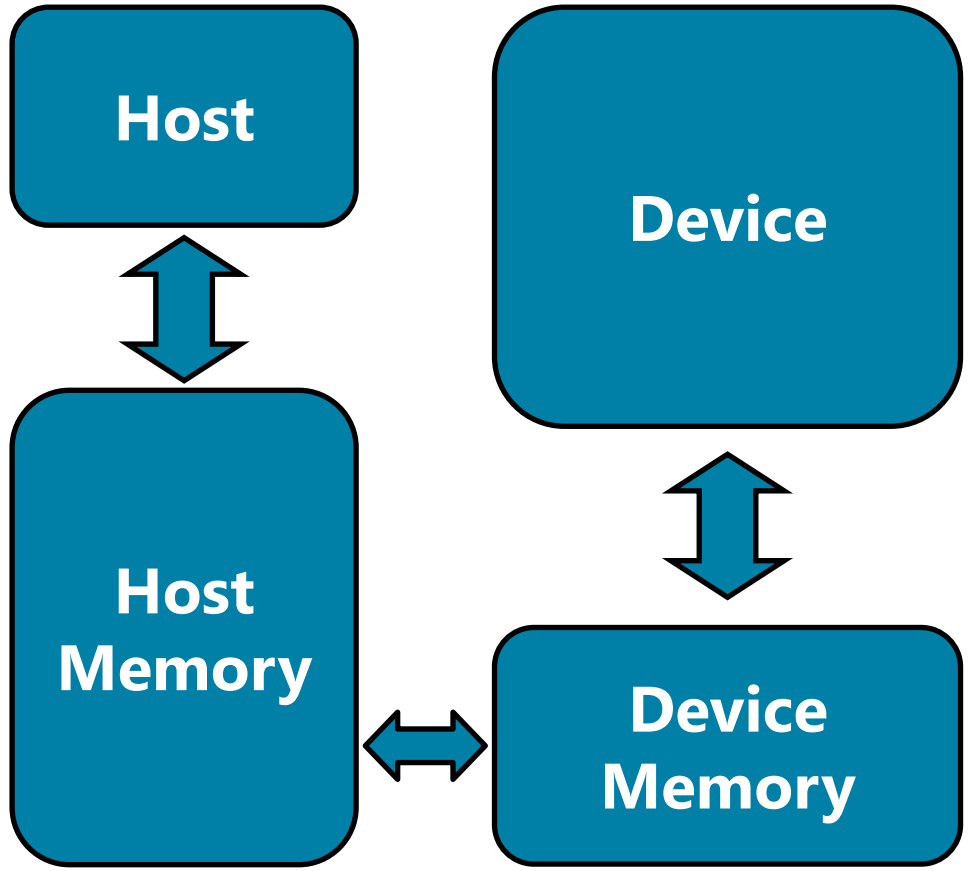
どちらも配列全体の1/4しかコピーしていない

iv. 明示的なメモリ管理

明示的なメモリ管理

メモリ管理の考え方

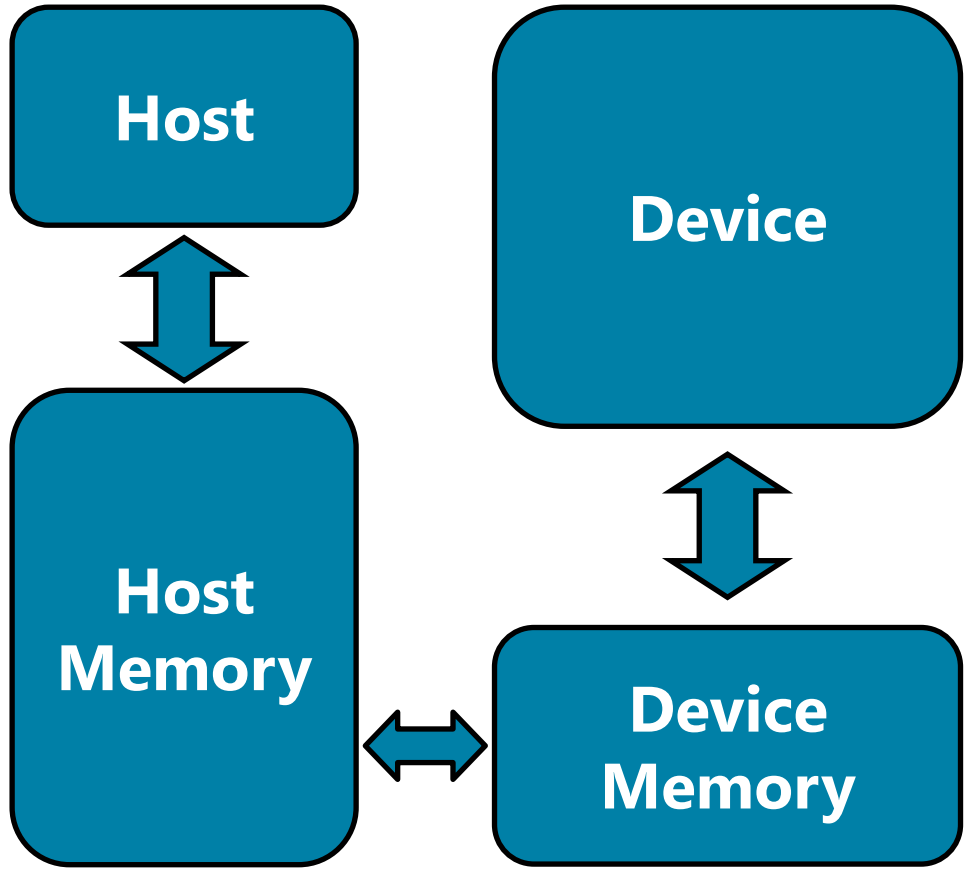
- 並列処理の実行時
  Device上でデータが見える必要がある
- 逐次処理の実行時
  Host上でデータが見える必要がある
- Host/Deviceがメモリを共有する必要がある場合、データのコピーが必要となる
- 性能を最大化するためには、不要なデータコピーを避ける



明示的なメモリ管理

重要な課題

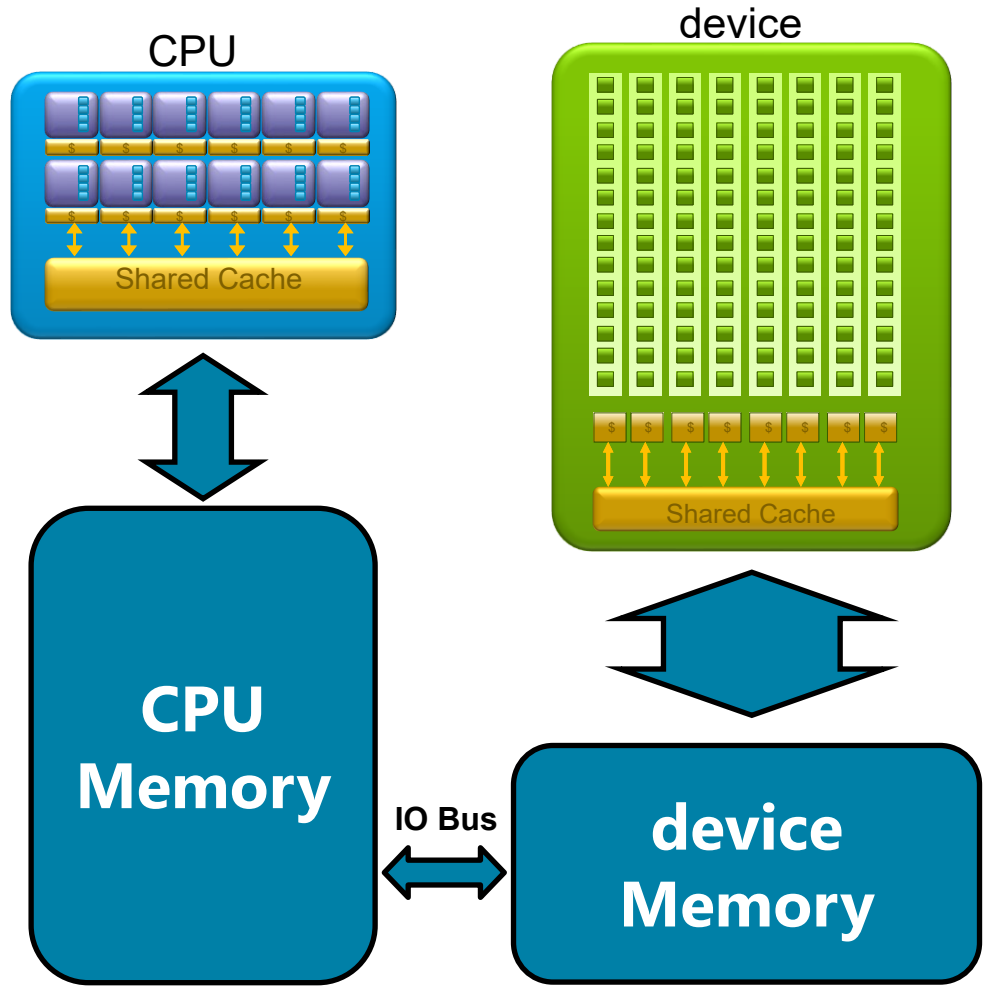
- 多くのDeviceは、Hostと別のメモリ空間を持つ
- 個々のメモリには異なるデータが格納され、自動では同期されないことが多い
- これらのメモリ間のデータ転送・共有は、非常に時間がかかる処理となる



明示的なメモリ管理

具体的なイメージに置き換えると

- 多くの並列ハードウェア（Device）はHostとは別のメモリプールを持つ
- 個々のメモリには異なるデータが格納され、自動では同期されないことが多い
- メモリはIO Bus（一般的にはPCIe）を経由してデータのコピーが行われるが、転送速度はメモリ性能に比べて遅い
- これらのメモリ間のデータ転送・共有は非常に時間がかかる処理となる



OpenACC
More Science, Less Programming

v. Data指示文

Data指示文

定義

- **Data指示文**はDevice上のデータの寿命を定義する
- 並列領域にいる間、データはDevice上に存在する
- Data指示文はユーザーがDevice上のデータの割り当てとコピーを制御するために使用する

Fortran

```
!$acc data clauses

  < Sequential and/or Parallel code >

!$acc end data
```

C/C++

```
#pragma acc data clauses
{

  < Sequential and/or Parallel code >

}
```

OpenACC
More Science, Less Programming

Data節

`copy( list )`	Deviceにメモリを割り当て、並列領域に入るときはHostからDeviceにデータをコピー、出るときはHostにデータをコピーする 主な用途: コード中の重要なデータ構造について、Deviceに対するデータの入力・変更・返却を行うための最も基本的な処理節
`copyin( list )`	並列領域に入るときに、Deviceにメモリを割り当てHostからDeviceへのデータコピーを行う 主な用途: サブルーチンの入力配列のようなものと考える
`copyout( list )`	Deviceにメモリを割り当て、並列領域を出るときにデータをHostにコピーする 主な用途: 入力データを上書きせずにDeviceの計算結果を受け取る
`create( list )`	Deviceにメモリを割り当てるが、Hostとのコピーは行わない 主な用途: 一時的な作業用配列の確保

OpenACC
More Science, Less Programming

Data指示文

例

Fortran

```
!$acc data copyin(a(1:N),b(1:N)) copyout(c(1:N))

  !$acc parallel loop
  do i = 1, N
    c(i) = a(i) + b(i)
  end do
!$acc end data
```

この並列領域はDevice上で実行されるので、 a, b, cはDevice上で見えていなければならない

OpenACC
More Science, Less Programming

Data指示文

例

Fortran

```
!$acc data copyin(a(1:N),b(1:N)) copyout(c(1:N))

  !$acc parallel loop
  do i = 1, N
    c(i) = a(i) + b(i)
  end do
!$acc end data
```

Action

OpenACC
More Science, Less Programming

vi. 暗黙 vs 明示的なデータ領域

OpenACC
More Science, Less Programming

データ領域

定義

- 暗黙的なデータ領域：kernels/parallel指示文が持っているデータ領域で、データはkernels/parallel指示文内でのみ存在できる

暗黙的

```
!$acc kernels copyout(a(1:100))
  do i = 1, 100
    a(i) = 0
  end do
!$acc end kernels
```

- 明示的なデータ領域：data指示文によって定義したデータ領域

OpenACC
More Science, Less Programming

暗黙 vs 明示的なデータ領域

機能的に同じ処理を行っている２つのコード例

明示的

```
!$acc data copyout(a(1:100))
!$acc kernels
  do i = 1, 100
    a(i) = 0
  end do
!$acc end kernels
!$acc end data
```

暗黙的

```
!$acc kernels copyout(a(1:100))
  do i = 1, 100
    a(i) = 0
  end do
!$acc end kernels
```

OpenACC
More Science, Less Programming

暗黙 vs 明示的なデータ領域

Data指示文を使うと性能向上が期待できる例

明示的

```
!$acc data copyout(a(1:N))

  !$acc kernels
    do i = 1, N
      a(i) = i
    end do
  !$acc end kernels

  !$acc kernels
    do i = 1, N
      a(i) = 2 * a(i)
    end do
  !$acc end kernels

!$acc end data
```

1 Data Copy

暗黙的

3 Data Copies

```
!$acc kernels copyout(a(1:N))
  do i = 1, N
    a(i) = i
  end do
!$acc end kernels

!$acc kernels copy(a(1:N))
  do i = 1, N
    a(i) = 2 * a(i)
  end do
!$acc end kernels
```

OpenACC
More Science, Less Programming

vii. 非構造データ指示文

OpenACC
More Science, Less Programming

非構造データ指示文

Enter data指示文

- Enter data指示文はDeviceのメモリ割り当てを行う
- メモリの割り当てにはcreate節またはcopyin節を使用できる
- 複数のenter data指示文を記述できるため、data指示文の開始点と異なる

Fortran

```
!$acc enter data clauses

  < Sequential and/or Parallel code >

!$acc exit data clauses
```

C/C++

```
#pragma acc enter data clauses

  < Sequential and/or Parallel code >

#pragma acc exit data clauses
```

OpenACC
More Science, Less Programming

非構造データ指示文

Exit data指示文

- Exit data指示文はDeviceのメモリ解放を行う
- メモリ解放にはdelete節またはcopyout節を使用できる
- Enter data指示文で与えた配列に対して同数のexit data指示文が必要
- Enter data/exit data指示文は異なる関数に記述可能

Fortran

```
!$acc enter data clauses

  < Sequential and/or Parallel code >

!$acc exit data clauses
```

C/C++

```
#pragma acc enter data clauses

  < Sequential and/or Parallel code >

#pragma acc exit data clauses
```

OpenACC
More Science, Less Programming

非構造データ指示文のdata節

`copyin ( list )`	enter dataディレクティブでDeviceにメモリを割り当てデータをHostからDeviceにコピーする
`copyout ( list )`	Deviceにメモリを割り当て、exit dataディレクティブでデータをHostにコピーする
`create ( list )`	enter dataディレクティブでDeviceにメモリ割り当てのみを行う
`delete ( list )`	exit dataディレクティブでDeviceのメモリを解放する

OpenACC
More Science, Less Programming

非構造データ指示文

基本的な例

Fortran

```
!$acc enter data copyin(a(1:N),b(1:N)) create(c(1:N))

  !$acc parallel loop
  do i = 1, N
    c(i) = a(i) + b(i)
  end do

!$acc exit data copyout(c(1:N))
```

非構造データ指示文

複数の関数にまたがったデータ転送の例

Fortran

```
subroutine allocate_array(A, N)
  allocate(A(1:N))
  !$acc enter data create(A[1:N])
end subroutine

subroutine deallocate_array(A)
  !$acc exit data delete(A)
  deallocate(A)
end subroutine

program main
  integer, allocatable :: A(:)
  allocate_array(A,100)
  !$acc kernels
    A(:) = 0
  !$acc end kernels
  deallocate_array(A)
end program
```

- 左記の例ではenter data/exit dataを別々の関数で定義
- ユーザーは、Deviceメモリの割り当てと解放をHostコードに合わせて配置できる
- クラス機能を持つC++において特に有効

OpenACC
More Science, Less Programming

非構造と構造の比較

非構造	構造
・複数の開始と終了点を持てる ・複数の関数に分岐可能 ・メモリは明示的に解放されるまで存在	・明示的な開始と終了点が必要 ・開始点と終了点が1つの関数内であること ・メモリは開始点と終了点の間でのみ存在

非構造

```
!$acc enter data copyin(a(1:N),b(1:N)) create(c(1:N))

  !$acc parallel loop
  do i = 1, N
    c(i) = a(i) + b(i)
  end do

!$acc exit data copyout(c(1:N)) delete(a,b)
```

構造

```
!$acc data copyin(a(1:N),b(1:N)) copyout(c(1:N))

  !$acc parallel loop
  do i = 1, N
    c(i) = a(i) + b(i)
  end do

!$acc end data
```

OpenACC
More Science, Less Programming

viii. CPU-GPU間のデータ同期

OpenACC
More Science, Less Programming

Update指示文

Update指示文: HostとDeviceの間の明示的なデータコピーを行う

データ領域の途中でデータを同期させたい場合に有効な指示文

節:

- **self**: memcpy Device to Host
- **device**: memcpy Host to Device

Fortran

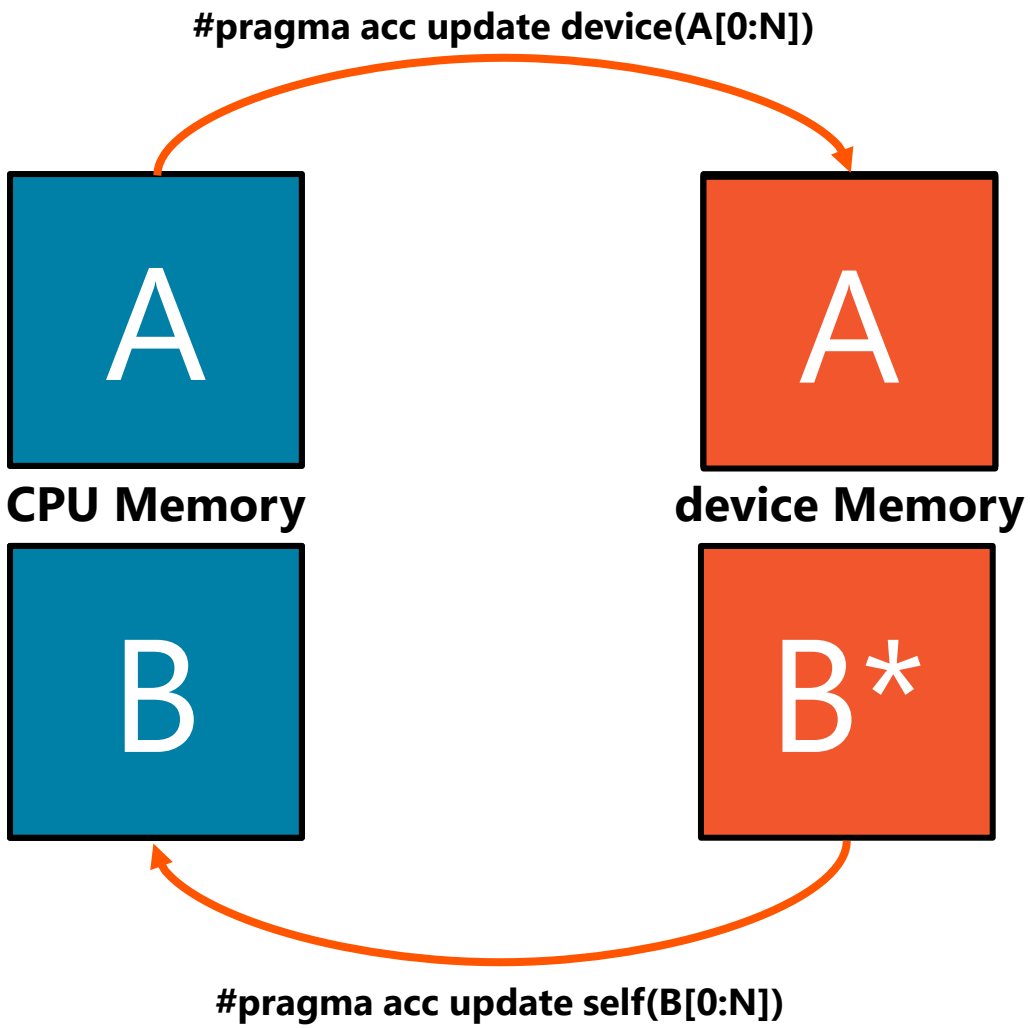
```
!$acc update self(x(1:end_index))
!$acc update device(x(1:end_index))
```

C/C++

```
#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])
```

OpenACC
More Science, Less Programming

Update指示文



Update指示文が機能するためには、データがHostとDeviceの両方に存在していなければならない

OpenACC
More Science, Less Programming

Update指示文

データ同期の例

Fortran

```
subroutine allocate_array(A, N)
  allocate(A(1:N))
  !$acc enter data create(A[1:N])
end subroutine

subroutine deallocate_array(A)
  !$acc exit data delete(A)
  deallocate(A)
end subroutine

subroutine initialize_array(A, N)
  do i = 1, N
    A(i) = i
  end do
  !$acc update device(A[1:N])
end subroutine
```

- initialize_array関数の中で配列AのHostデータを変更する
- HostとDeviceの間で配列Aのデータを同期するために、update deviceを実行する
- Update指示文が無いと、後々の計算コードにおいてデータ不整合が発生し結果がおかしくなる

OpenACC
More Science, Less Programming

4. ループの並列化

OpenACC
More Science, Less Programming

Reduction節

Fortran

```
do i = 1, size
  do j = 1, size
    do k = 1, size
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
```

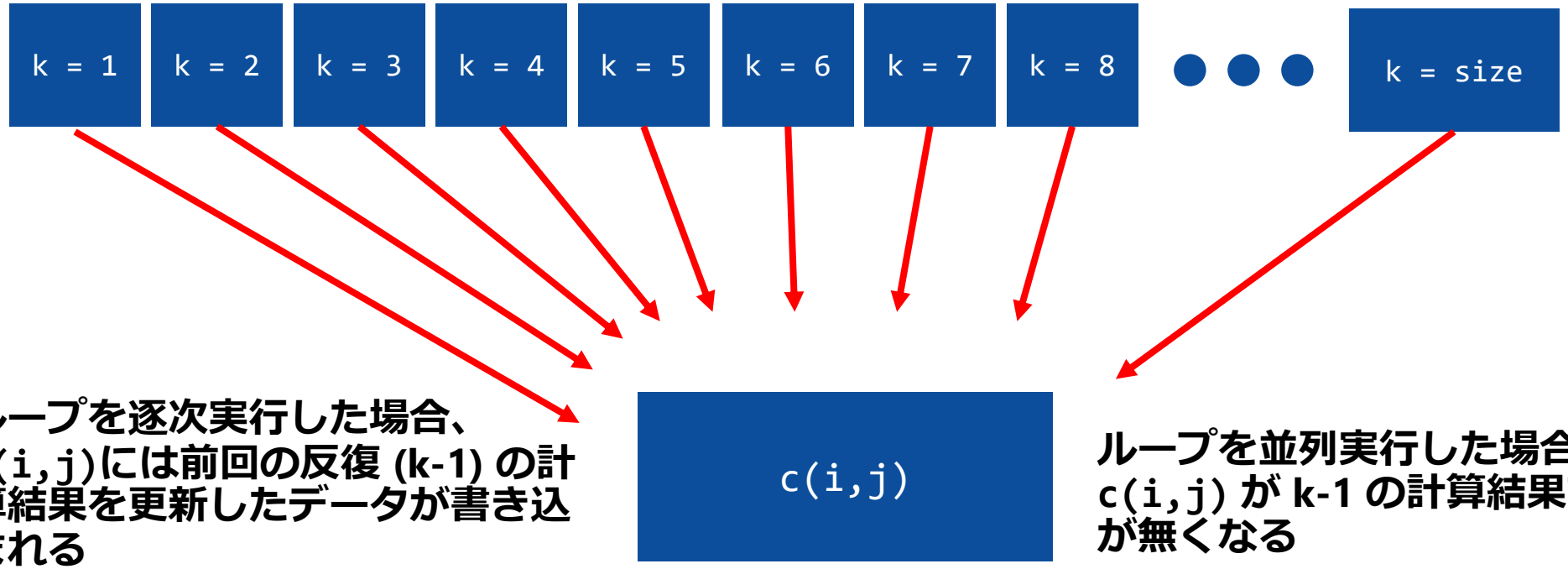
- 最内ループは並列化不可
- 三重ループをそのまま並列化した場合、複数スレッドが c(i,j) に同時に書き込みを行う可能性がある
- 複数スレッドが同時に同じメモリに書き込みを行うと、結果に誤りが生じる（データ競合）
- Reduction clauseによりこの問題を解決する

OpenACC
More Science, Less Programming

Reductionしないと...

Fortran

```
!$acc parallel loop
do k = 1, size
  c(i,j) = c(i,j) + a(i,k) * b(k,j)
end do
```



ループを逐次実行した場合、c(i,j)には前回の反復 (k-1) の計算結果を更新したデータが書き込まれる

ループを並列実行した場合、参照したc(i,j) が k-1 の計算結果である保証が無くなる

OpenACC
More Science, Less Programming

Reduction節

- **reduction**は例えば配列の総和のような多数の値を1つの値に「縮約」する場合に使う
- 各スレッドはreduction節に指定した変数のプライベートなコピーを作成し、自身が計算したループの結果に対し部分的な縮約を行う
- ループ計算後、reduction節は各スレッドの結果を畳み込み、最終的な結果が得られる
- reduction節が必要かどうかはコンパイラにとって判断が容易なため、省略可能

Fortran

```
do i = 1, size
  do j = 1, size
    do k = 1, size
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
    end do
  end do
end do
```

Fortran

```
do i = 1, size
  do j = 1, size
    tmp = 0.0
    !$acc parallel loop reduction(+:tmp)
    do k = 1, size
      tmp = tmp + a(i,k) * b(k,j)
    end do
    c(i,j) = tmp
  end do
end do
```

スカラ変数tmpを新たに定義

OpenACC
More Science, Less Programming

Reduction節

制限事項

- reduction節の対象には配列の要素は指定できない

```
a(0) = 0
!$acc parallel loop reduction(+:a[0])
do i = 1, 100
  a(0) = a(0) + i
end do
```

- reduction節の対象となる変数は、C言語のstruct、C++のclass、Fortranの派生型のメンバであってはならない

```
v%val = 0
!$acc parallel loop reduction(+:v%val)
do i = 1, v%size
  v%val(0) = v%val(0) + i
end do
```

OpenACC
More Science, Less Programming

Reduction節の演算子

Operator	Description	Example
`+`	Addition/Summation	`reduction(+:sum)`
`*`	Multiplication/Product	`reduction(*:product)`
`max`	Maximum value	`reduction(max:maximum)`
`min`	Minimum value	`reduction(min:minimum)`
`&`	Bitwise and	`reduction(&:val)`
`|`	Bitwise or	`reduction(|:val)`
`&&`	Logical and	`reduction(&&:val)`
`||`	Logical or	`reduction(||:val)`

OpenACC
More Science, Less Programming

Auto節

- **auto節は指定したループが並列化可能かコンパイラに判断させる**
- **ループを並列化しても問題ないか、判断が難しい場合に便利**

Fortran

```
!$acc parallel loop auto
do i = 1, size
  do j = 1, size
    do k = 1, size
       c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
```

Auto節

- Kernels指示文は暗黙的にauto節が付与されるため、auto節を明示する必要はない
- auto節はparallel指示文を利用する際に非常に有用

Fortran

```
!$acc kernels loop auto
do i = 1, size
  do j = 1, size
    do k = 1, size
       c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
!$acc end kernels
```

OpenACC
More Science, Less Programming

Independent節

- independent節は指定したループが並列化可能であることを明示し、コンパイラがループについて判断した内容を上書きする
- Kernels指示文を使用する際、コンパイラが本来できるはずの並列化を様々な要因で諦めてしまうことがある
- プログラマはindependent節を使ってコンパイラが非並列と判断したループの並列化を強制させることが可能

Fortran

```
!$acc kernels loop independent
do i = 1, size
  do j = 1, size
    do k = 1, size
       c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
!$acc end kernels
```

OpenACC
More Science, Less Programming

Independent節

- **Parallel指示文は暗黙的にindependent節が付与されるため、明示する必要はない**

Fortran

```
!$acc parallel loop independent
do i = 1, size
  do j = 1, size
    do k = 1, size
       c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
```

OpenACC
More Science, Less Programming

Seq節

- **seq (sequential) 節はループを逐次実行するように指示**
- **右のサンプルコードは、コンパイラは外の2重ループを複数スレッドで並列化するが、各スレッドは最内のループを逐次処理する**
- **コンパイラは次元数が多すぎるループに対し、ターゲットデバイスに合わせて自動的にseq節を付与する場合がある**

Fortran

```
!$acc parallel loop
do i = 1, size
  !$acc loop
  do j = 1, size
    !$acc loop seq
    do k = 1, size
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
```

Private and firstprivate節

- **private節は指定した変数リストをthread-private変数として定義**
- **各スレッドはカンマ区切りで与えられたリストのすべての変数のプライベートなコピーを持つ**
- **firstprivate節は、private節の処理に加えHostメモリの値でDeviceメモリが初期化される**

Fortran

```
real :: tmp(3)

!$acc kernels loop private(tmp(1:3))
do i = 1, size
   tmp(1) = <value>
   tmp(2) = <value>
   tmp(3) = <value>
end do
!$acc end kernels

! note that the host value of “tmp”
! remains unchanged.
```

OpenACC
More Science, Less Programming

Private and firstprivate節

- private/firstprivate節は記述されたループの中でプライベートな変数として定義される
- 多重ループの最内以外で定義された場合、プライベート変数は内側のループで共有される

Fortran

```
real :: tmp(3)

!$acc kernels loop private(tmp(1:3))
do i = 1, size
   ! the tmp array is private to each iteration
   ! of the outer loop
   tmp(1) = <value>
   tmp(2) = <value>
   tmp(3) = <value>
   !$acc loop
   do j = 1, size2
      ! but tmp is shared amongst the threads
      ! in the inner loop
      array(i,j) = tmp(1)+tmp(2)+tmp(3)
   end do
end do
!$acc end kernels
```

OpenACC
More Science, Less Programming

Scalars and private節

- デフォルト動作では、スカラ変数（配列でないただの数）はparallel指示文内では firstprivate 、kernels指示文内ではprivateが指定される
- いくつかのケースを除いて、スカラ変数はprivate指定する必要はない。以下はその状況だが、これに限定されず必要な場合もある

1. スカラ変数が、C/C++のグローバル変数、Fortranのモジュール変数のように、グローバルで共有される場合
2. スカラ変数が、Deviceのサブルーチンに参照として渡される場合
3. スカラ変数が、計算後にreturn-valueとして返却される場合

- 注意：スカラ変数をprivate指定するとかえって性能が低下する場合がある

OpenACC
More Science, Less Programming

Collapse節

- collapse(N)
- N個のtightly nested loopをマージ
- 多重ループを1次元ループに変換できる

- メモリの局所性を高める
- 大きなループを生成し並列性を高める

...などに非常に有効

Fortran

```
!$acc parallel loop collapse(2)
do i = 1, size
  do j = 1, size
    tmp = 0
    !$acc loop reduction(+:tmp)
    do k = 1, size
       tmp = tmp + a(i,k) * b(k,j)
    end do
    c(i,j) = tmp
  end do
end do
```

OpenACC
More Science, Less Programming

Collapse節

collapse(2)

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

Fortran

```
!$acc parallel loop collapse(2)
do j = 1, 4
  do i = 1, 4
    array(i,j) = 0
  end do
end do
```

OpenACC
More Science, Less Programming

Tile節

- `tile ( x , y , z, ...)`
- 多重ループを “Tile” or “Block に分割
- コードによってはdata locality (データ局所性)の向上が可能
- 複数の “Tile” を同時に実行可能

Fortran

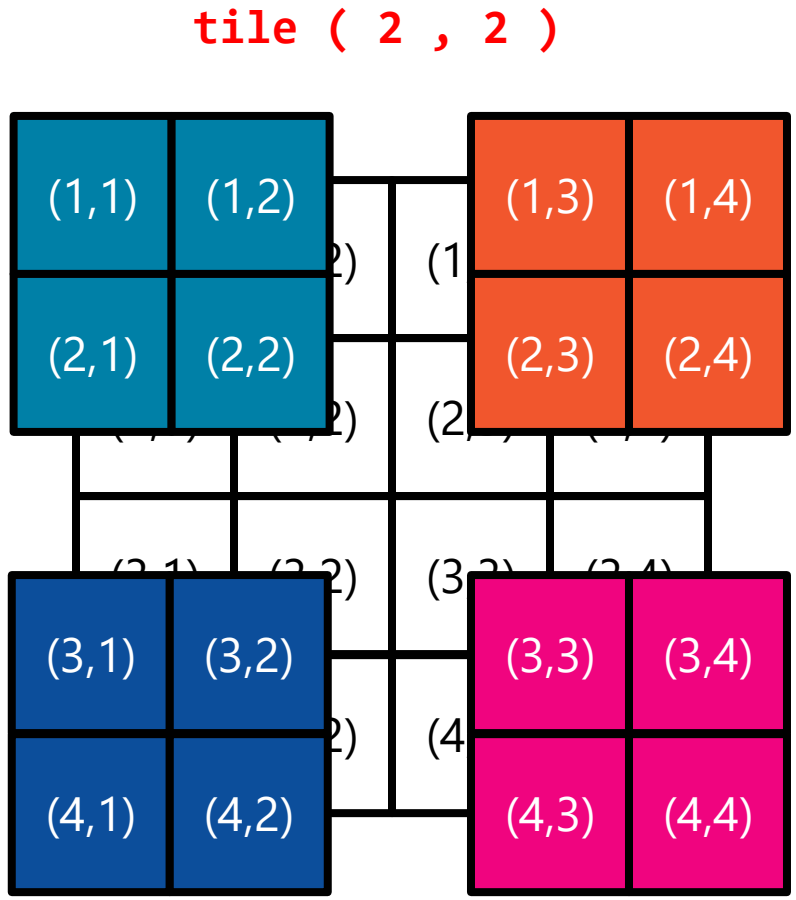
```
!$acc kernels loop tile(32, 32)
do i = 1, size
  do j = 1, size
    do k = 1, size
       c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
!$acc end kernels
```

OpenACC
More Science, Less Programming

Tile節

Fortran

```
!$acc kernels loop tile(2,2)
do j = 1, 4
  do i = 1, 4
    array(i,j) = array(i,j) + 1
  end do
end do
!$acc end kernels
```



OpenACC
More Science, Less Programming

5. OpenACCによる並列化の進め方

コードの例 (1)

daxpy: a * x + y

Fortran

```
subroutine daxpy(n, a, x, y)
  integer :: n
  double precision :: a, x(n), y(n)

  !$acc kernels loop copyin(x(1:n)) copy(y(1:n))
  do i = 1, n
    y(i) = a * x(i) + y(i)
  end do
  !$acc end kernels
end subroutine
```

1. **Kernels指示文で並列化できるか試す**
2. **data節をつけてデータのコピーを明示的に処理**

コードの例 (2)

ddot: x・y

Fortran

```
function ddot(n, x, y) result(s)
  integer :: n
  double precision :: s, x(n), y(n)

  s = 0
  !$acc kernels loop reduction(+:s) copyin(x(1:n),y(1:n))
  do i = 1, n
    s = s + x(i) * y(i)
  end do
  !$acc end kernels
end function
```

1. **Kernels指示文で並列化を試みる（reductionはコンパイラが自動付与）**
2. **data節をつけてデータのコピーを明示的に処理**

OpenACC
More Science, Less Programming

理化学研究所情報システム部：
https://i.riken.jp/supercom/documents/himenobmt/

コードの例 (3)

3次元ステンシル計算 (himenoBMTxp.f90)

```
subroutine jacobi(nn, gosa)
  ...
  !$acc data copyin(a,b,c,bnd,wrk1) create(wrk2) copy(p)   ! コンパイラが把握可能なら範囲省略可
  do loop = 1, nn
    gosa = 0.0
    !$acc kernels loop reduction(+:gosa)
    do k = 2, kmax-1
      do j = 2, jmax-1
        do i = 2, imax-1
          s0 = a(i,j,k,1)*p(i+1,j,k) &
          ...
          gosa = gosa + ss*ss
          ...
    end do
    !$acc end kernels
    !$acc kernels
    p(2:imax-1, 2:jmax-1, 2:kmax-1) = wrk2(2:imax-1, 2:jmax-1, 2:kmax-1)
    !$acc end kernels
  end do
  !$acc end data
end subroutine
```

OpenACC
More Science, Less Programming

OpenACCを使ったアプリGPU化の進め方

1. NVIDIAコンパイラを使って、既存のアプリ（CPUコード）の動作確認・結果確認
2. 高コストなループにparallel/kernels指示文を入れる
3. コンパイル（オプション-acc –Minfo=accelを付ける）
4. コンパイルメッセージを読んで、ループがGPU上で並列化されていることを確認
5. プログラムの実行・結果確認
6. 実行時間の確認（timer、プロファイリングツールNsight Systemsを使う等）
7. 必要に応じて、2-6の作業を繰り返し、コードの最適化を進める

OpenACC
More Science, Less Programming

OpenACCを使ったアプリGPU化と工数対効果

- 結論：工数対効果の出しやすさは、アプリケーションに強く依存する
- 理想：高コストループにkernels指示文を入れるだけでアプリ性能が劇的に向上する
- 工数対効果を出すのが容易でないパターン：逐次処理に律速してしまう、コスト分布が分散している、高コストループの演算負荷が低い、高コストループの並列度が大きくない、データ転送のコストが大きくコスト削減も容易でない
- データ転送のコストがボトルネックになる場合：
  - Unified memory機能（+GH200等のアーキテクチャ）を使う
  - メモリ管理の指示文を入れる（使うGPUは限定されない）
- メモリ管理の指示文を入れる場合、コード管理できるように設計することが大事
- 高度なチューニングの方針：アプリケーション全体の初期化と終了処理でだけenter/exit dataを記述し、update self/deviceでHost/Device間データコピーのみを行う

OpenACC
More Science, Less Programming

6. まとめ

本講習会で紹介した内容

1. GPUプログラミングの概要
    i. アムダールの法則
    ii. GPUの特性
    iii. GPUプログラミング
    iv. NVIDIA HPC SDK
2. OpenACCを使ったプログラムの並列化
    i. OpenACCの概要
    ii. OpenACCの基本構文
    iii. Parallel指示文
    iv. Kernels指示文
    v. Loop指示文
    vi. OpenACCコードのコンパイル
3. CPU-GPU間のデータ転送の最適化
    i. メモリ管理の概要
    ii. Managed memory
    iii. Data節
    iv. 明示的なメモリ管理
    v. Data指示文
    vi. 暗黙 vs 明示的なデータ領域
    vii. 非構造データ指示文
    viii. データ同期
4. ループの並列化
5. OpenACCによる並列化の進め方
6. まとめ

OpenACC
More Science, Less Programming

弊グループのHPCサポート

- 弊グループ（プロメテック・ソフトウェア、GDEPソリューションズ）にて下記のHPC関連のサポートをしております。

- GPUサーバ https://www.gdep-sol.co.jp/
- NVIDIA HPCコンパイラサポートサービス https://hpcworld.jp/support/

THANK YOU

OpenACC
More Science, Less Programming