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 2024.6.27

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：OpenACCを使ったアプリケーションのGPU高速化

  NVIDIA HPC SDKのインストールとコンパイラのサポート
- 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

弊グループ紹介

- 会社名　　プロメテック・ソフトウェア株式会社
- 設立年月日　2004年10月29日
- 所在地　　本社：東京都文京区本郷三丁目34番3号　本郷第一ビル8階
- 事業　　科学技術計算用ソフトウェア開発・販売とコンサルティング

PROMETECH

GDEP Solution

大学との共同研究開発協力体制

東京大学・東北大学・京都大学・東京理科大学・東洋大学・琉球大学・横浜国立大学　など

グループ企業

GDEPソリューションズ株式会社

CAD/CAEなどをターゲットとしたNVIDIA社GPU製品を活用したITソリューションの提供

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
- HPC関連サービス

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のコード例
    iii. OpenACCの基本構文
    iv. Parallel directive
    v. Kernels directive
    vi. Loop directive
    vii. 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. 基本的なデータ管理の考え方
    ii. CUDA managed memory
    iii. OpenACC data clauses
    iv. Explicit memory management
    v. OpenACC data directive
    vi. Implied data regions
    vii. Unstructured data directives
    viii. data synchronization
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)

OpenACC
More Science, Less Programming

1. GPUプログラミングの概要

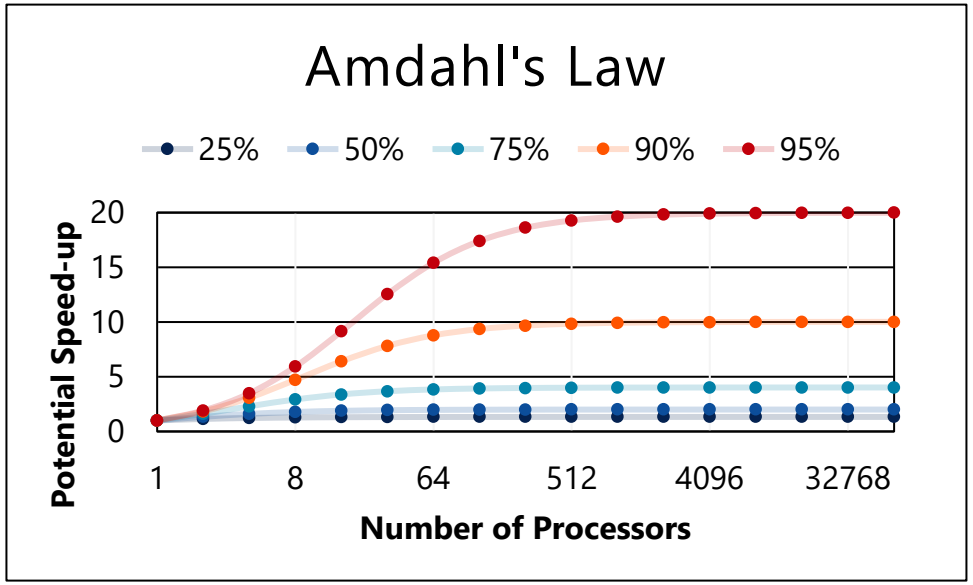
OpenACC
More Science, Less Programming

i. アムダールの法則

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
- 逐次処理が多ければ多いだけ並列化の恩恵が少なくなる
- 最も時間のかかる計算を中心に並列化することが極めて重要
- 簡単な計算であっても並列化が必要な場合がある（特にGPUでは）



OpenACC
More Science, Less Programming

アムダールの法則を使用する

並列化によって得られる最大性能向上率を推定できる

- コードの50%を並列化

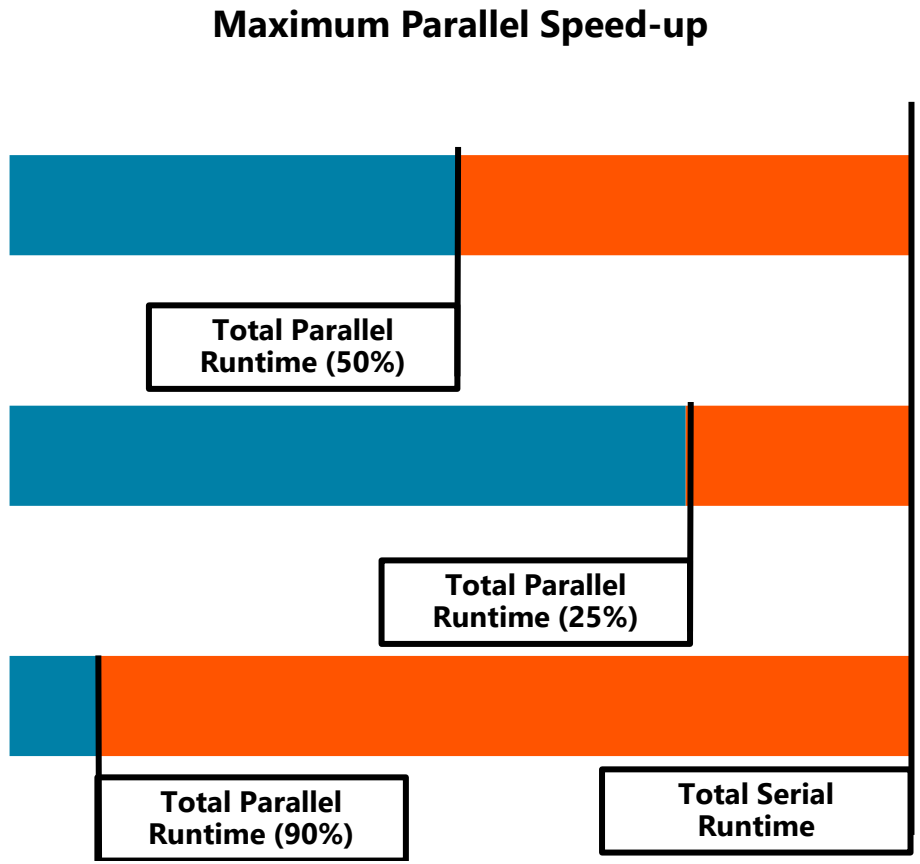
  1/(100% - 50%) = 1/(1.0 - 0.50) = **2.0倍**

- コードの25%を並列化

  1/(100% - 25%) = 1/(1.0 - 0.25) ≒ **1.3倍**

- コードの90%を並列化

  1/(100% - 90%) = 1/(1.0 - 0.90) = **10.0倍**



OpenACC
More Science, Less Programming

ii. GPUの特性

OpenACC
More Science, Less Programming

CPU vs GPU

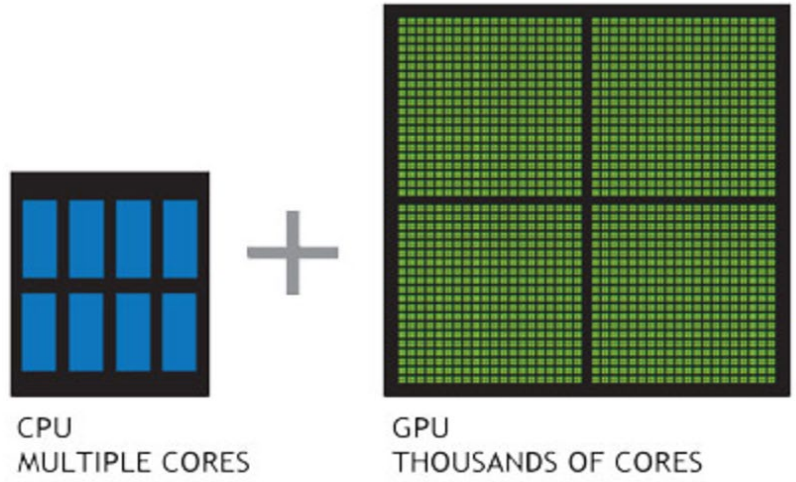
物理コアと並列化の違い

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1000個以上**持ち並列化により高い性能を発揮する



どちらにとっても並列化は重要

GPUはその構造から並列化をより強く求められる

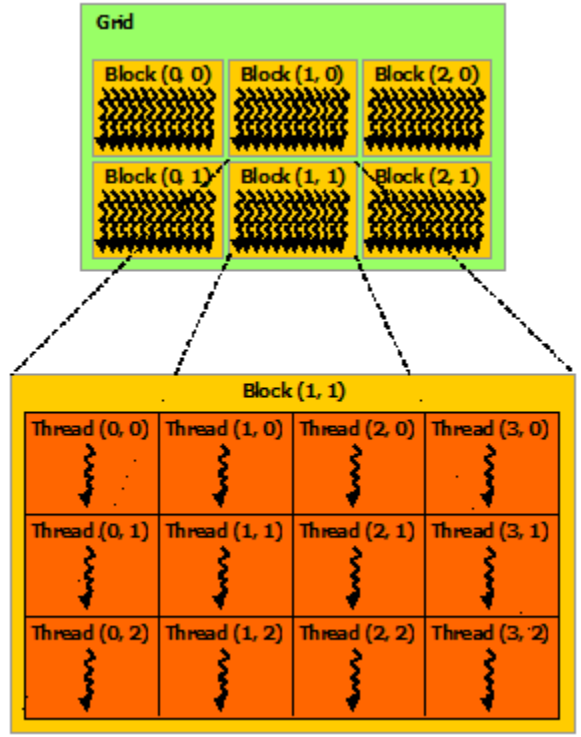
OpenACC
More Science, Less Programming

Grid, Block, Thread, Warp

NVIDIA CUDA GPUにおける並列化階層

- 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
- 多重ループ並列化、SIMD処理などに割当

- 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
  32スレッド単位で処理を実行



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 (Warp = 32 Threads)

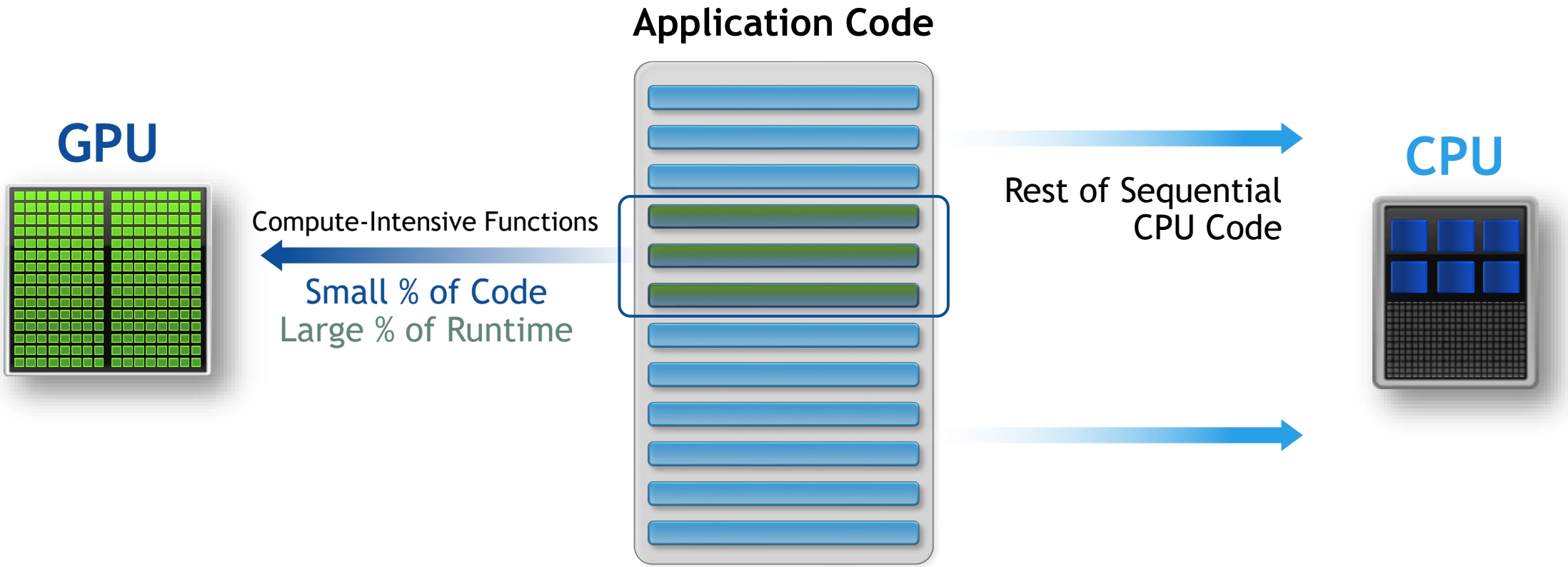
T0	T1	T2	T3	T4	T5	T6	T7
1	2	3	4	5	6	7	8
+	+	+	+	+	+	+	+
9	10	11	12	13	14	15	16
=	=	=	=	=	=	=	=
10	12	14	16	18	20	22	24

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

- **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

- **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

CUDA C++, CUDA Fortran

- GPGPU向けの並列言語
- NVIDIAによりCUDA C++が実装
- PGIによりCUDA FortranがPGIコンパイラで実装される
- その後、NVIDIAはPGIを買収しCUDA Fortranも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、かなり細かくディレクティブ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模様

```
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
- CUDAやOpenMP accelerator modelに比べてハードウェア詳細を考える時間が少ない
- NVIDIA HPCコンパイラで利用できる
- 単一のコードでCPU、GPU実行でき、管理し易い。

```
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
```

OpenACC
More Science, Less Programming

CUDAプログラミングの難しさ

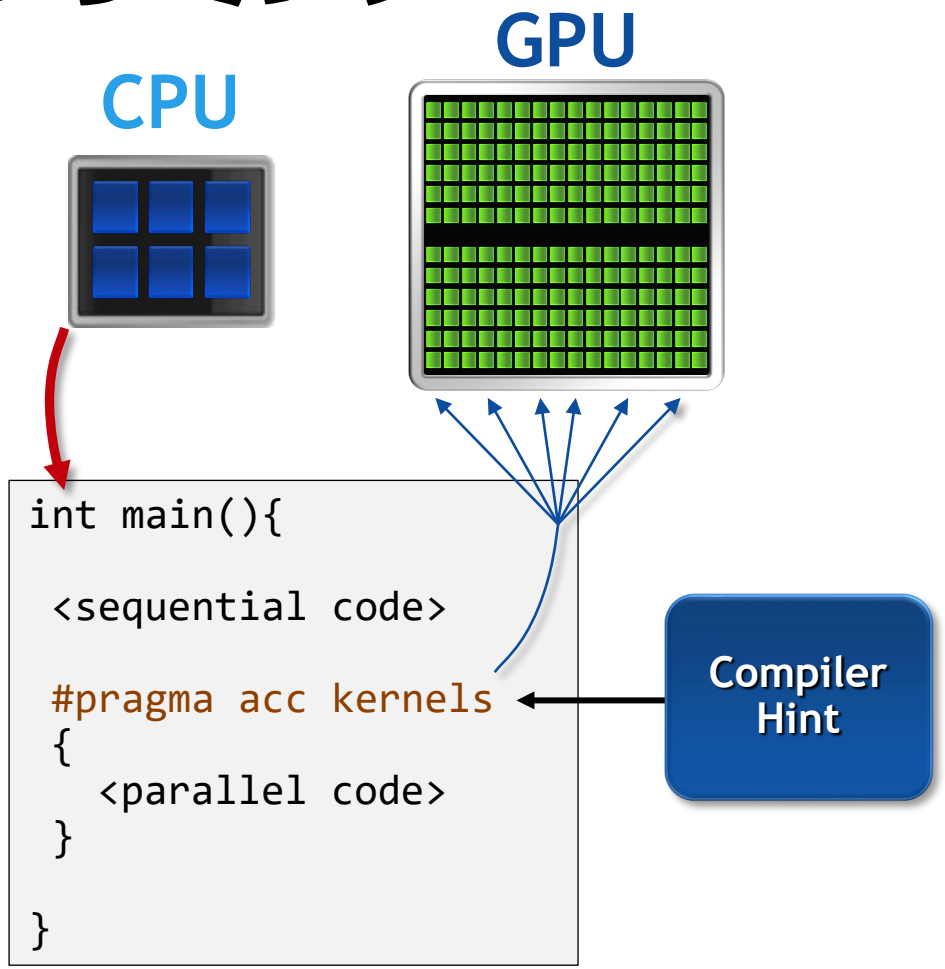
従来と異なるプログラミングパラダイム・並列モデルの導入

- NVIDIA GPUが搭載されていないシステムでは利用できない
- 従来システムとGPUシステムで2つの実装が必要になり、メンテナンス性が低下
- スーパーコンピュータ（大規模並列システム）においてはMPI + OpenMPのハイブリッド並列化がデファクトスタンダード、ここにCUDAを加えるとプログラムの複雑性が大幅に増加

- MPI + OpenMP + OpenACCならまだ簡単？

OpenACCを用いたGPUプログラミング

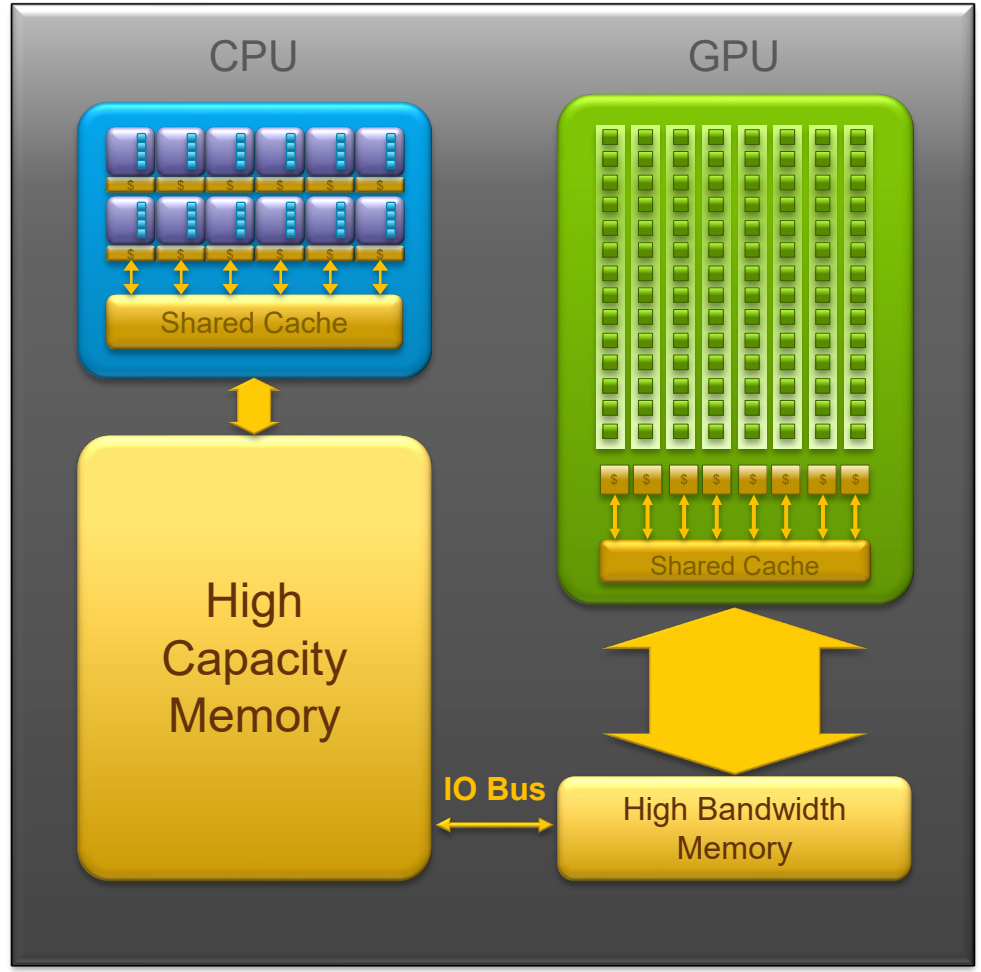
- プログラムは常にCPUで開始・終了する
- Compute-intensiveなループをOpenACCディレクティブを用いてGPUにオフロードする
- GPUへのオフロードは場合によりCPUとGPUの間でデータ転送が必要となる
- CPUはGPUの制御およびGPUの並列性を活かせないコードを実行する



OpenACC
More Science, Less Programming

CPU + GPU
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



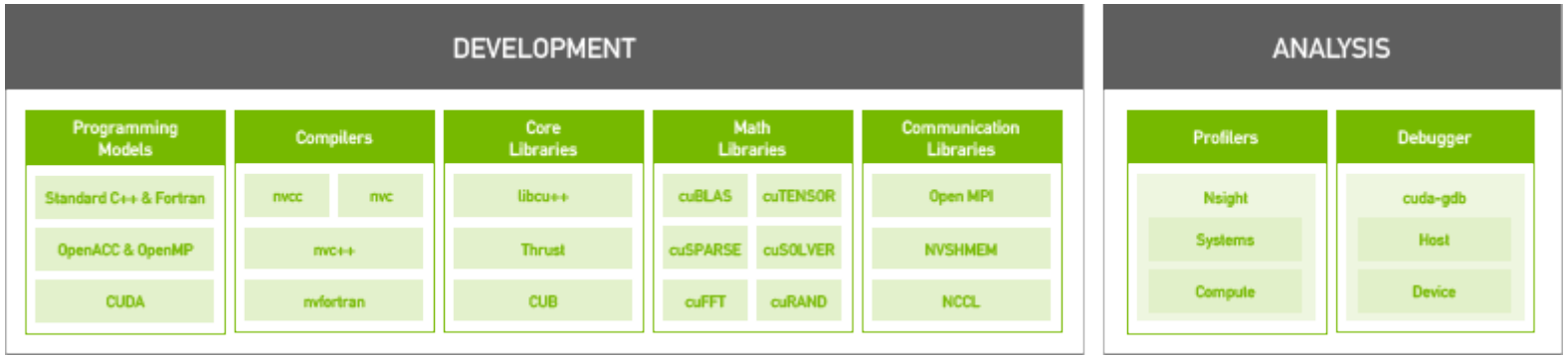
OpenACC
More Science, Less Programming

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）



NVIDIA HPCコンパイラ

x86_64, Armをサポート

- 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24.5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

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
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）

- 規格詳細については、それぞれの言語規格を参照すること
- Partial supportの場合もあるので注意

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
- 技術サポートが有償契約で提供、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
- 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/

OpenACC
More Science, Less Programming

2. OpenACCを使ったプログラムの並列化

OpenACC
More Science, Less Programming

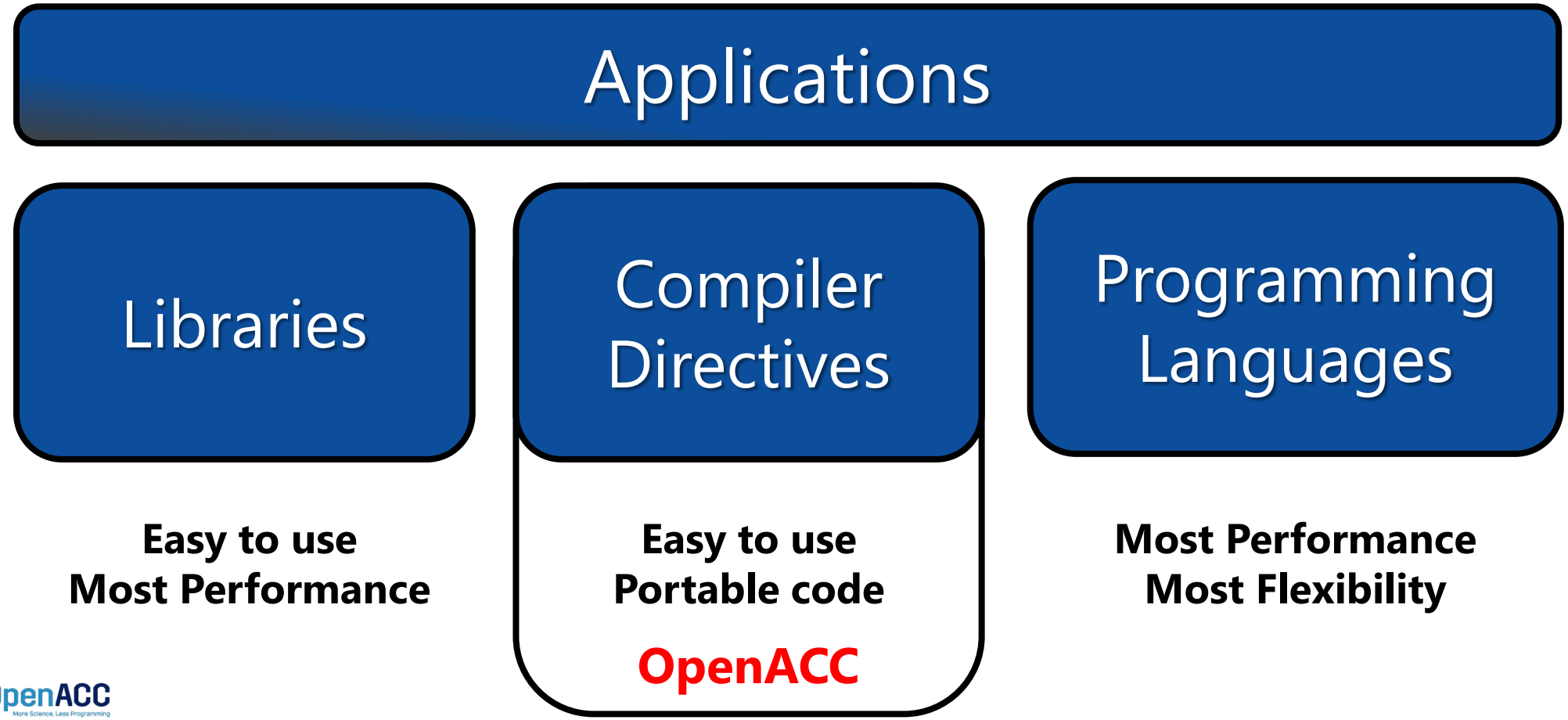
i. OpenACCの概要

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>
  }
}
```

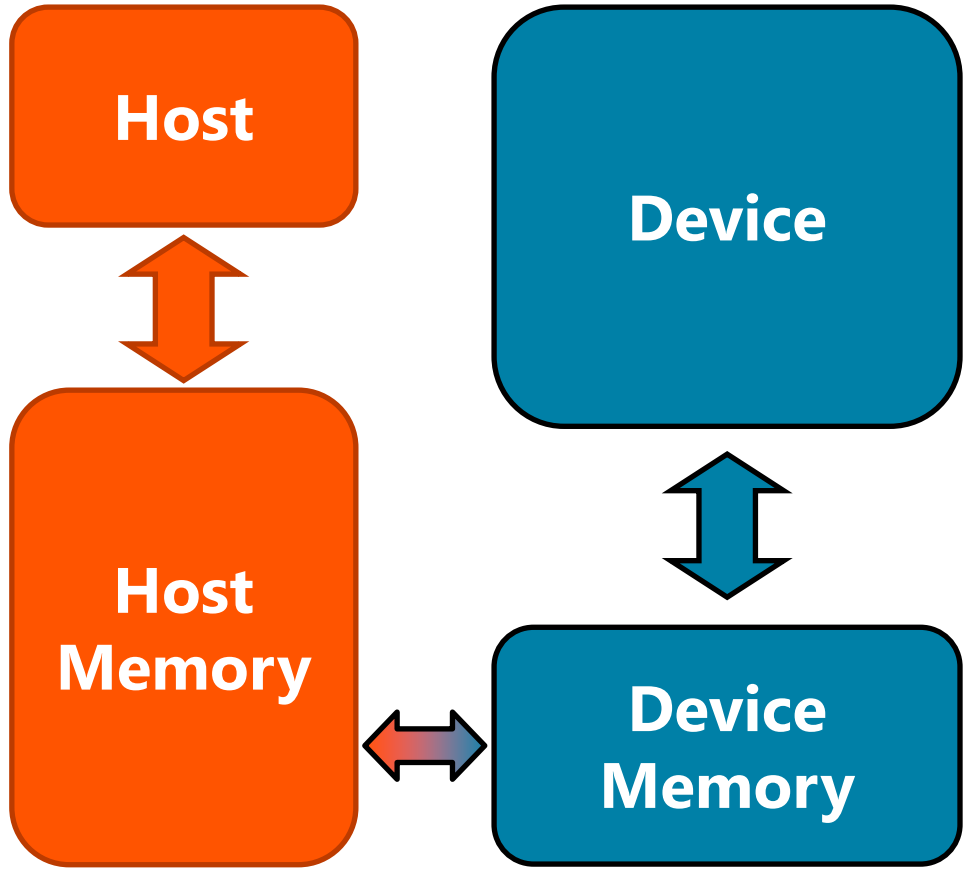
3 Ways to Accelerate Applications



OpenACC
More Science, Less Programming

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

- 既存の逐次処理コードを維持
- 並列性をディレクティブで記述
- 正しく動いたら、ディレクティブの詳細を記述し最適化する

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
- 逐次処理コードの維持

Low Learning Curve

- OpenACCは簡単に利用・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

OpenACCの利点 (incremental)

Incremental

- 既存の逐次処理コードを維持
- 並列性をディレクティブで記述
- 正しく動いたら、ディレクティブの詳細を記述し最適化する

Enhance Sequential Code

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

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

逐次処理コードを作成する

OpenACCで並列化

コードが正しく動いたら、必要に応じOpenACCコードを修正・改善する

OpenACC
More Science, Less Programming

OpenACCの利点 (single source)

Supported Platforms

x86 CPU

NVIDIA GPU

Arm CPU

Single Source

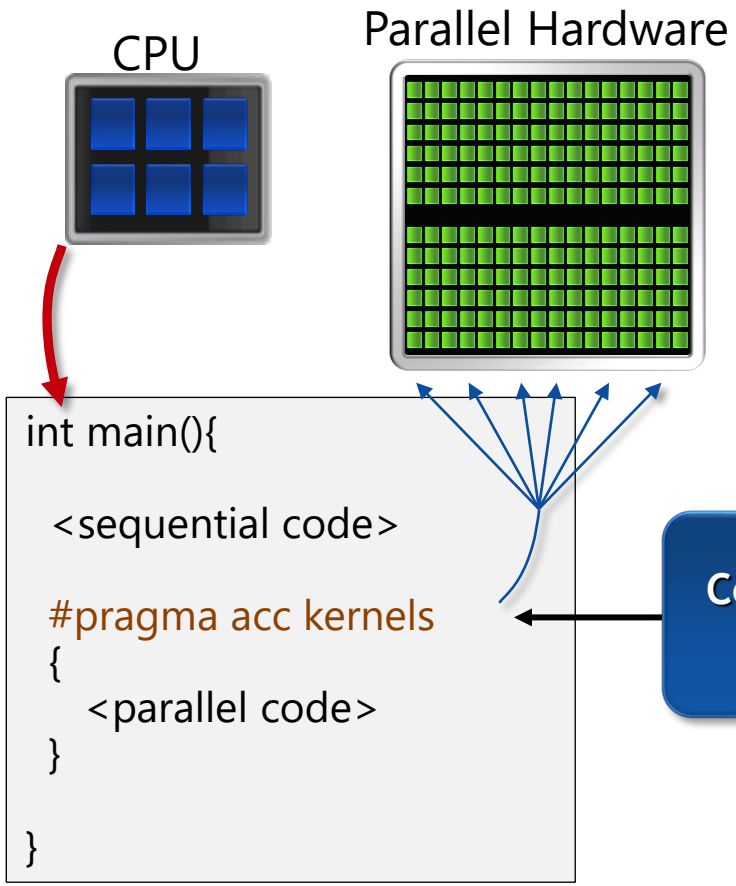
- コードを修正せず複数のハードウェアでリビルド
- コンパイラが目的のハードウェアに合わせて並列化方法を決定
- 逐次処理コードの維持

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用することができる

```
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)



ユーザーはコンパイラに並列化したい部分のヒントを与える

コンパイラはヒントからターゲットとなるハードウェアに適した並列処理を生成する

Low Learning Curve

- OpenACCは簡単に利用・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

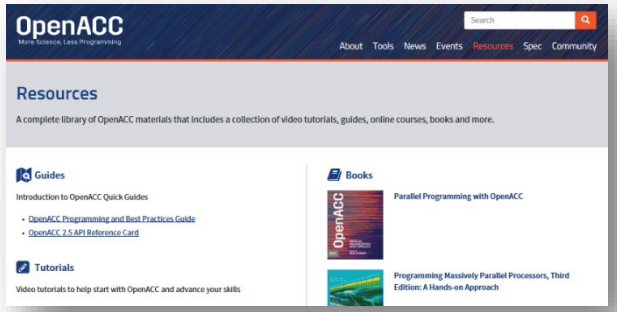
OpenACC
More Science, Less Programming

OpenACC Resources

Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

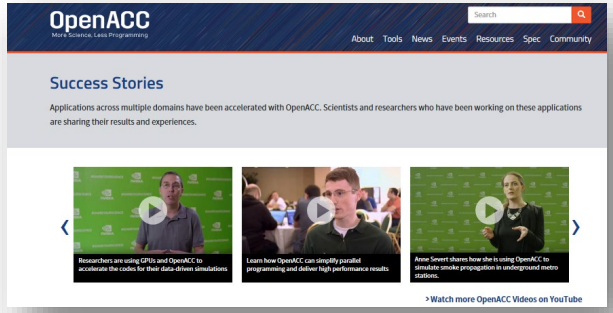
Resources

https://www.openacc.org/resources



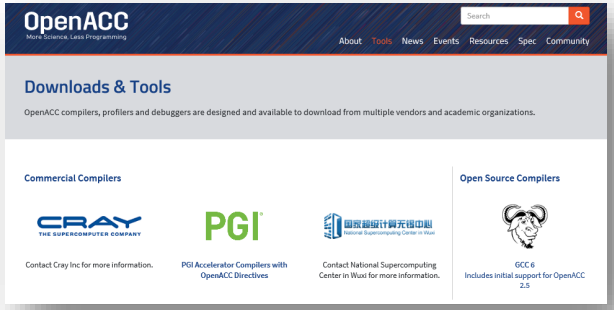
Success Stories

https://www.openacc.org/success-stories



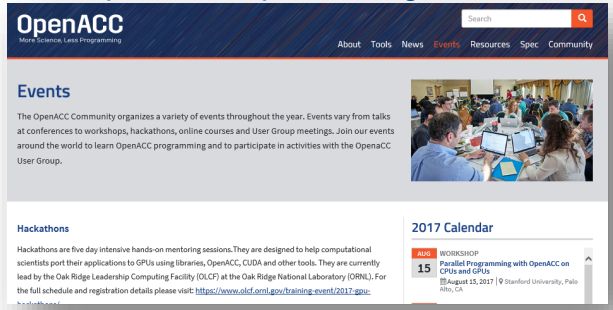
Compilers and Tools

https://www.openacc.org/tools



Events

https://www.openacc.org/events



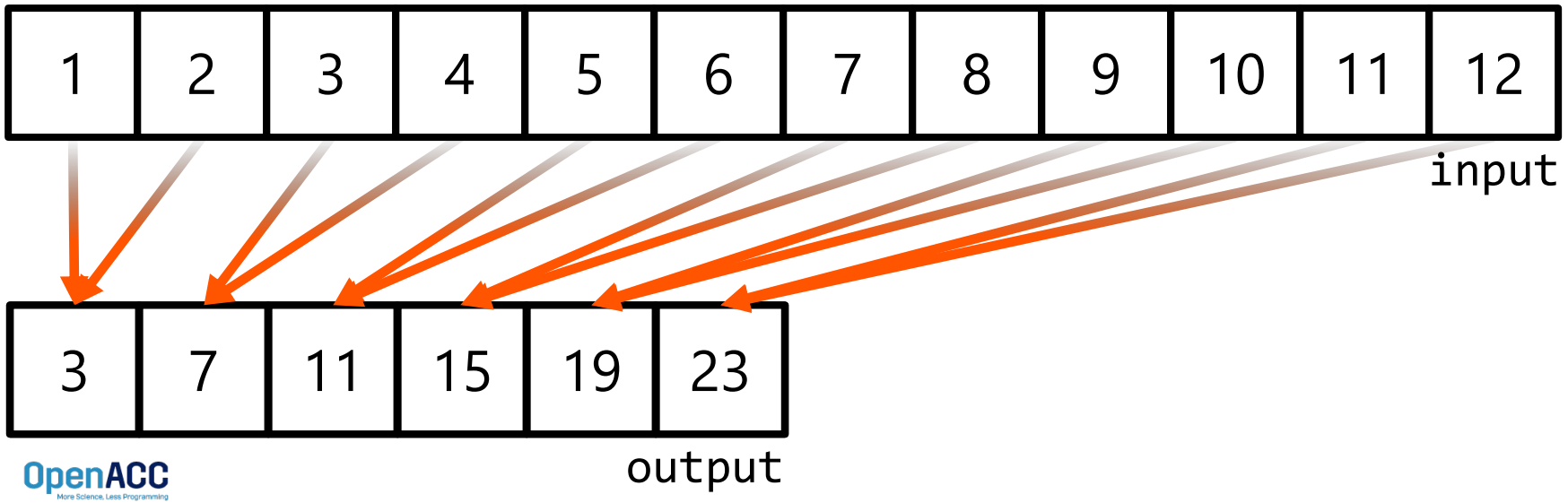
OpenACC
More Science, Less Programming

ii. OpenACCのコード例

OpenACCのコード例

Array pairing example

```
subroutine pairing(input, output, N)

  do i = 1, N
   output(i) = input(i*2-1) + input(i*2);
  end do
end subroutine
```

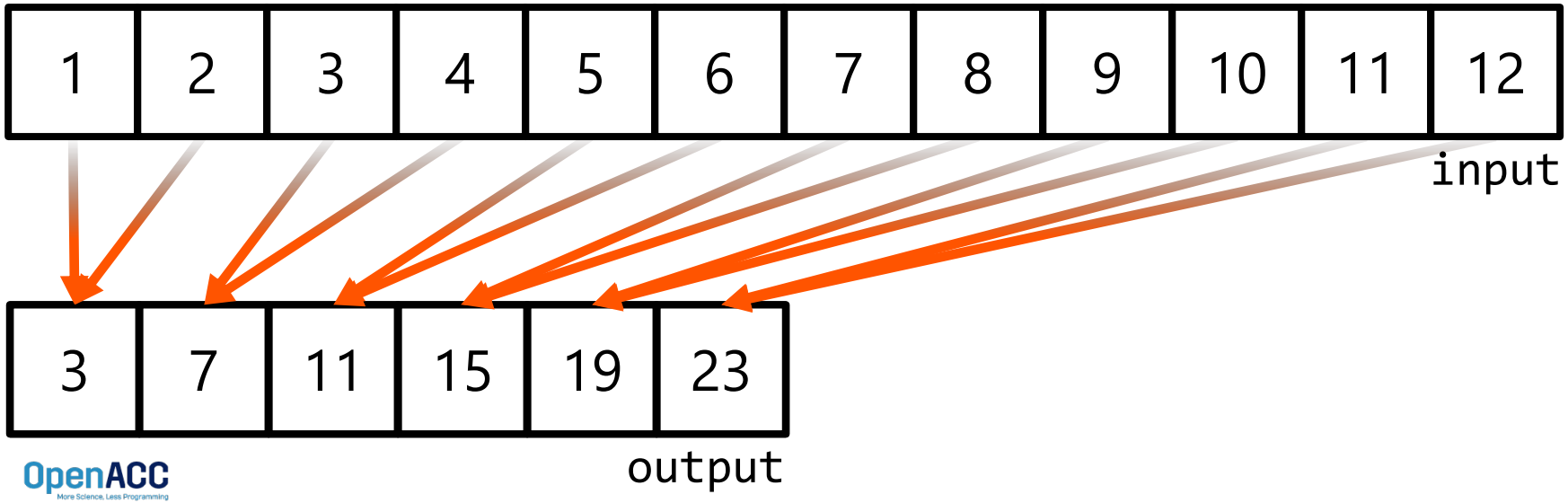


OpenACC
More Science, Less Programming

OpenACCのコード例

Array pairing example - parallel

```
subroutine pairing(input, output, N)
!$acc parallel loop
  do i = 1, N
    output(i) = input(i*2-1) + input(i*2);
  end do
end subroutine
```



OpenACC
More Science, Less Programming

データ依存で並列化できない例

Not all loops are parallel

```
subroutine pairing(a, N)
!$acc parallel loop
  do i = 2, N
    a(i) = a(i) + a(i-1)
  end do
end subroutine
```

逐次実行と並行実行で
計算結果が変わってしまう

iii. OpenACCの基本構文

OpenACC syntax

Syntax for using OpenACC directives in code

Fortran

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

C, C++

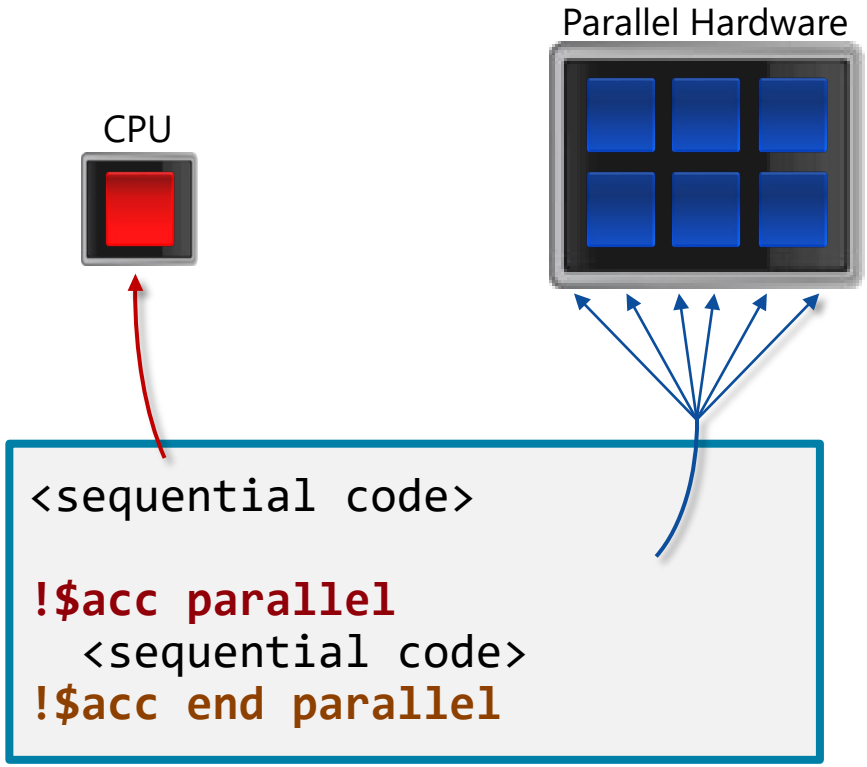
```
#pragma acc directive clauses
<code>
```

- directive: Fortranにおいて特別なフォーマットのコメントで、コードのコンパイル方法をコンパイラに指示し、pragmaと同様にコンパイラは自由に無視できる
- pragma: C, C++においてコードのコンパイル方法をコンパイラに教えるためのプリプロセッサ、記述されたpragmaが翻訳できない際にコンパイラは自由に無視できる
- “acc”: 後に続く文がOpenACCディレクティブであることをコンパイラに教える
- Directives: OpenACCでコードを変更するためのコマンド
- Clauses: directivesの指定や追加のこと

OpenACC
More Science, Less Programming

iv. Parallel directive

Parallel directive
明示的な並列化の指示



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

OpenACC
More Science, Less Programming

Parallel directive
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ディレクティブを指定する
- 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ディレクティブは次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 directive

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行のディレクティブによりすべての並列化指示が可能
- parallel loopディレクティブ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
- データ依存関係があるループに適用した場合はparallel loopは誤った計算を行うことがある
- OpenMPと同じマナー（explicit parallelization)

OpenACC
More Science, Less Programming

Parallel directive
複数のループの並列化

```
!$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ディレクティブの記述が必要
- 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

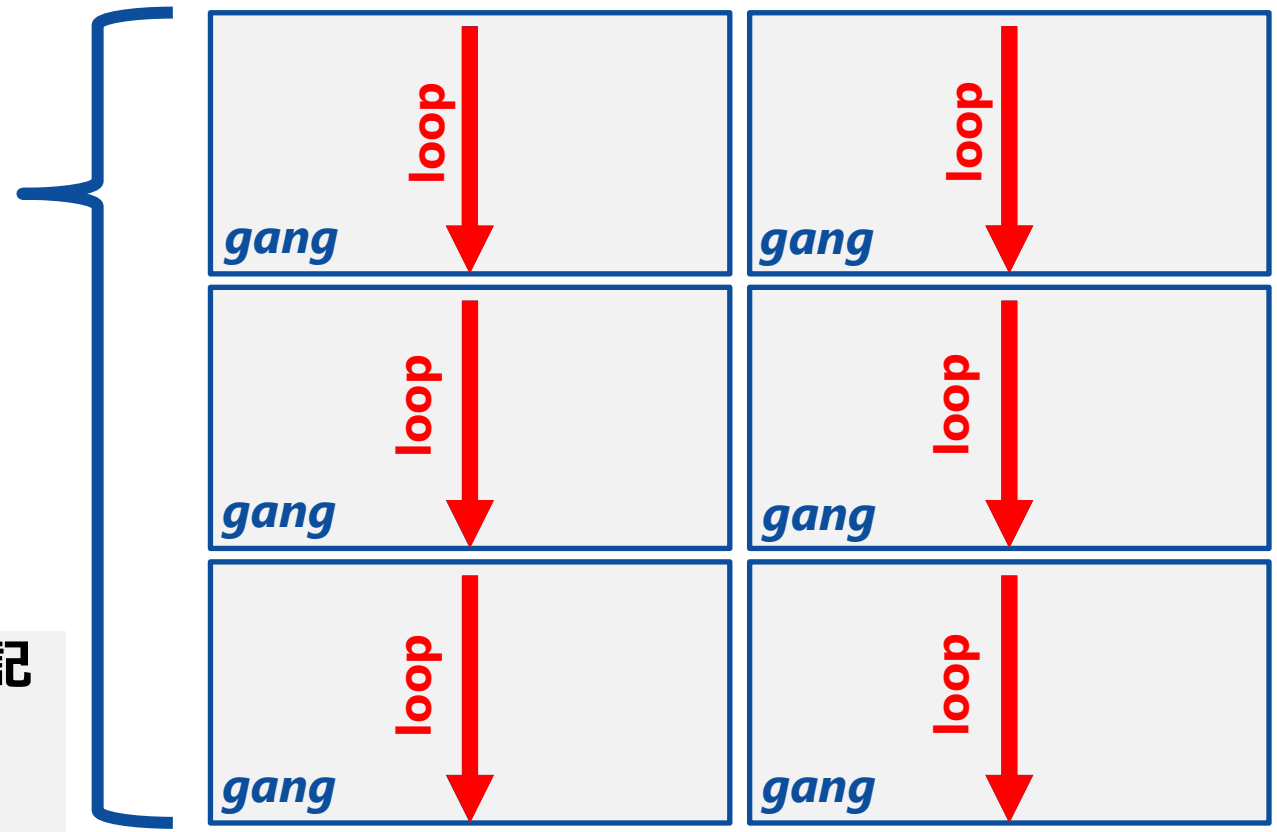
OpenACC
More Science, Less Programming

Parallel directive

Expressing parallelism

```
!$acc parallel


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

!$acc end parallel
```

parallelディレクティブが記述されたとき、コンパイラは1つ以上のgangsを生成し、冗長実行を行う



OpenACC
More Science, Less Programming

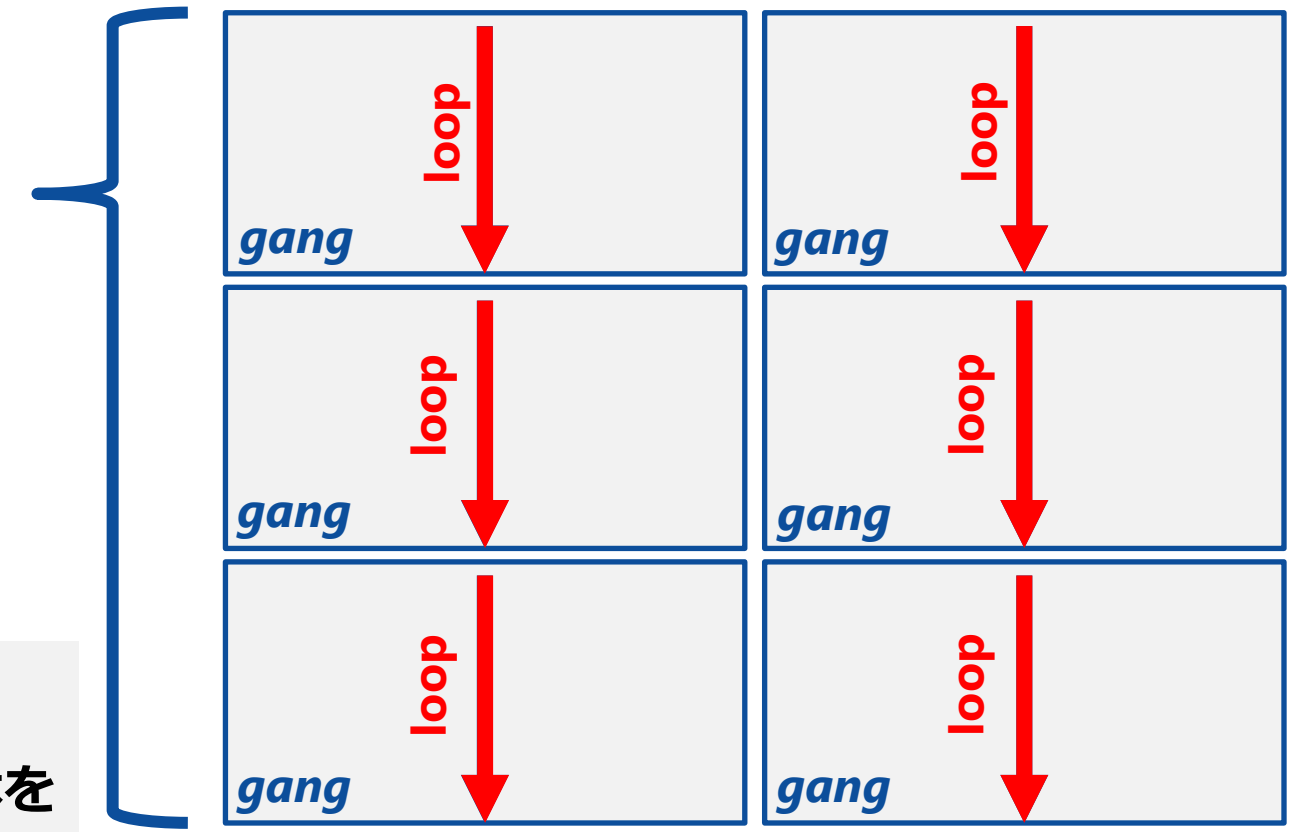
Parallel directive

Expressing parallelism

```
!$acc parallel

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

!$acc end parallel
```

loop

ループは各gangsで冗長的に実行される。（各gangsは、ループ全体を実行する。）



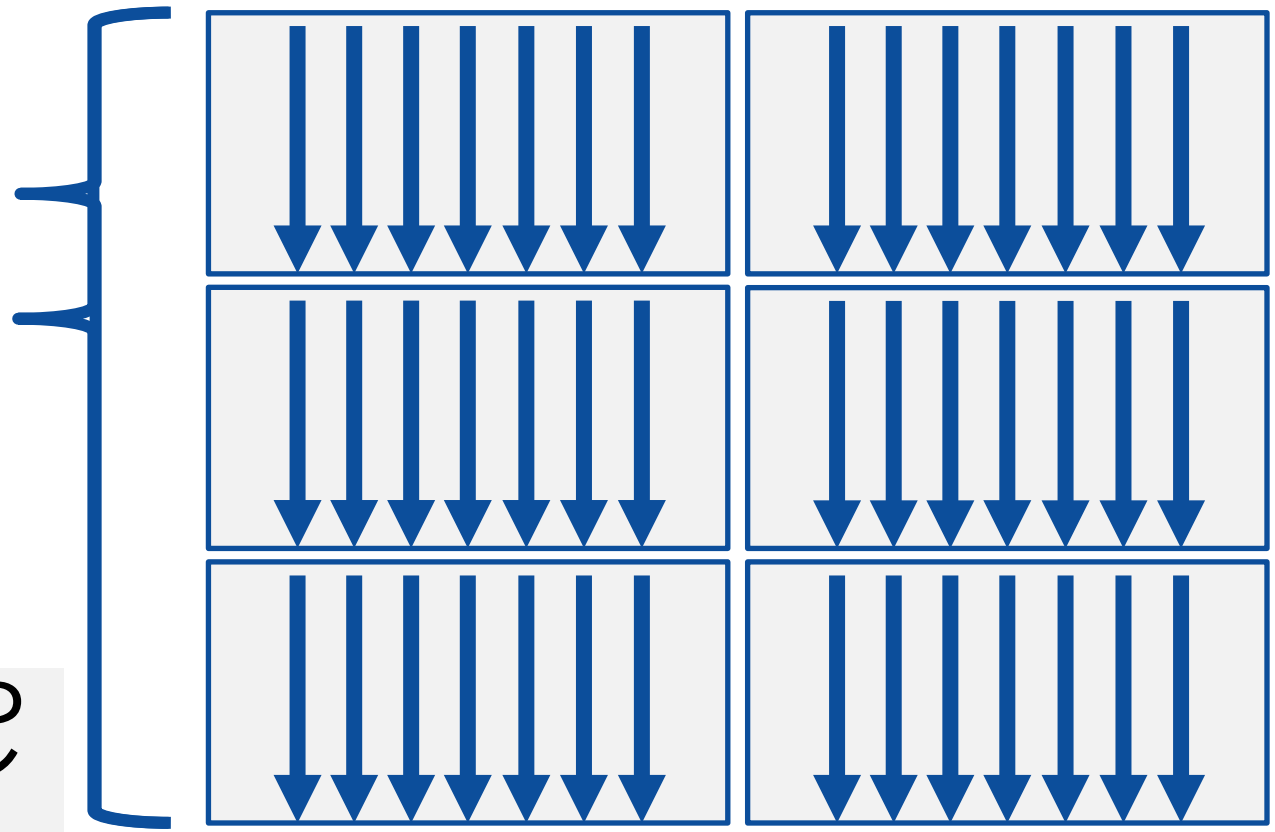
OpenACC
More Science, Less Programming

Parallel directive

Expressing parallelism

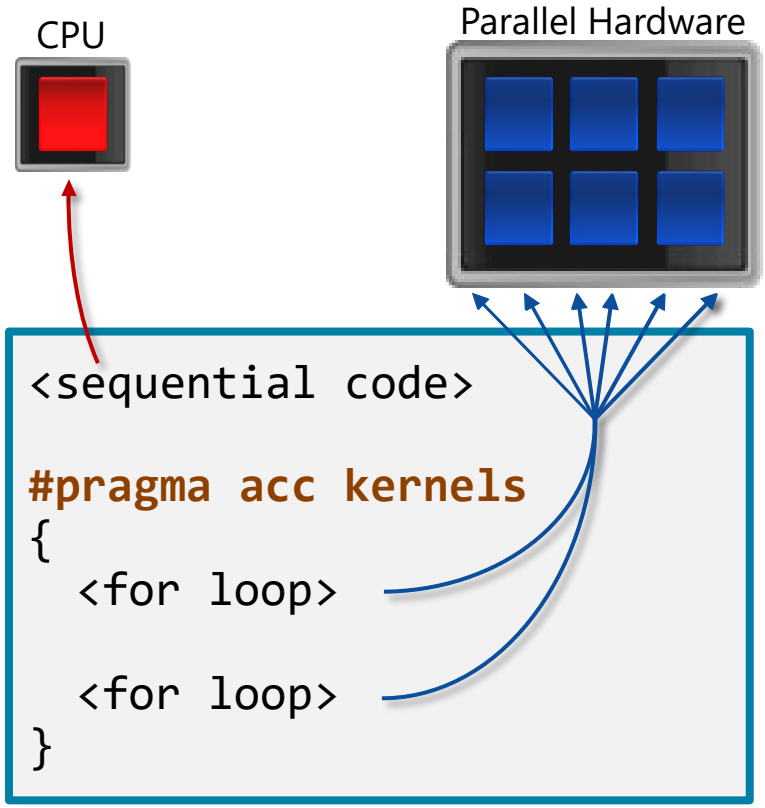
```
!$acc parallel

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

!$acc end parallel
```

loopディレクティブはどのループを並列化するかコンパイラへ指示



v. Kernels directive

Kernels directive
コンパイラによる並列化



- **kernels**ディレクティブ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
- kernelsディレクティブは複数ループがある計算領域にまとめて適用することも可能
- 半自動的な並列化

OpenACC
More Science, Less Programming

Kernels directive
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;
```

- 左のコード例では、kernelsディレクティブを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ついて、並列リソース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は並列化なしでコンパイルが行われる

OpenACC
More Science, Less Programming

Kernels directive

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;
}
```

- 次の例ではコードブロックの並列化をkernelsディレクティブで指示している
- コードブロックはC, C++では中括弧を使い、Fortranでは!$accおよび!$acc endで定義する
- 単一ループ指定と同様にコンパイラはコードブロック内のすべてのループについて個別に並列化と最適化を試みる

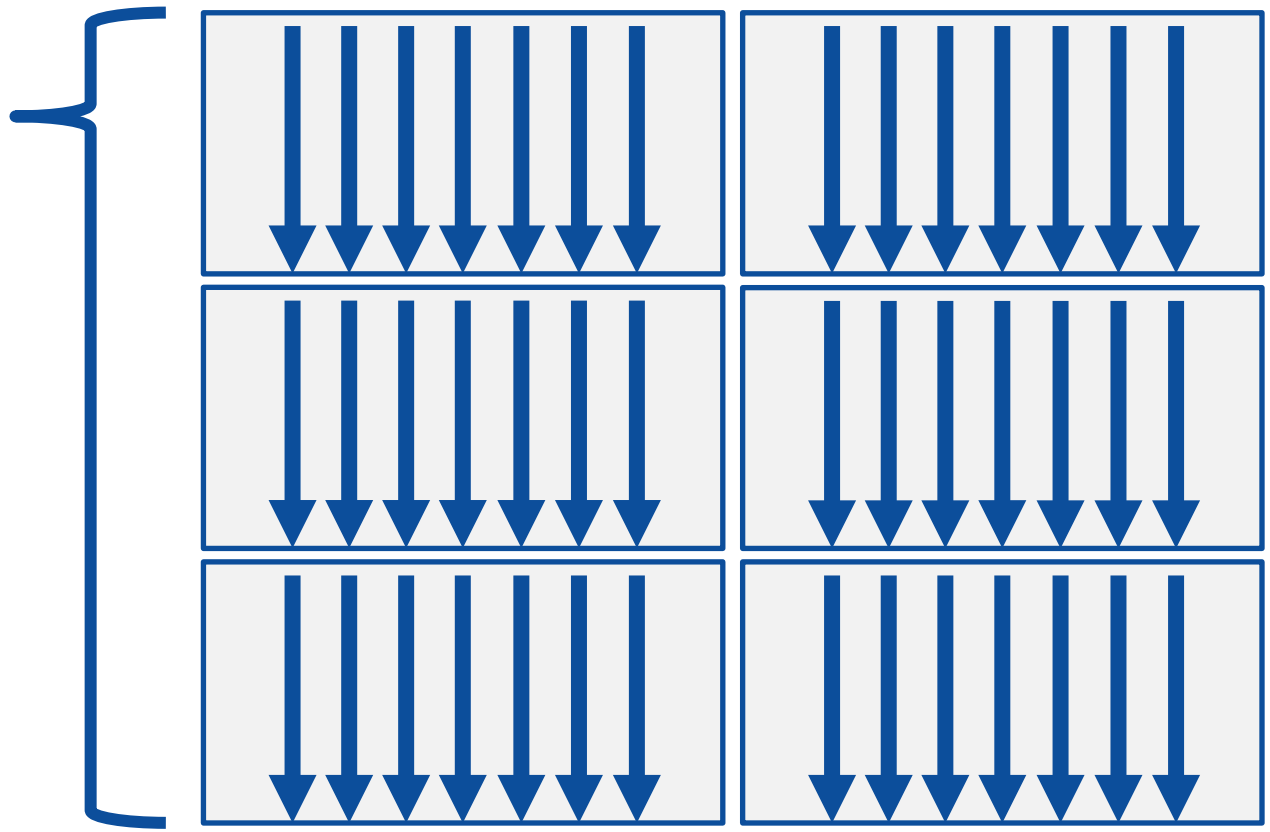
OpenACC
More Science, Less Programming

Kernels directive

Expressing parallelism

```
!$acc kernels

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

  do i = 1, M
    b(j) = 0
  end do

!$acc end kernels
```

kernelsは暗黙的にloop ディレクティブを指定



OpenACC
More Science, Less Programming

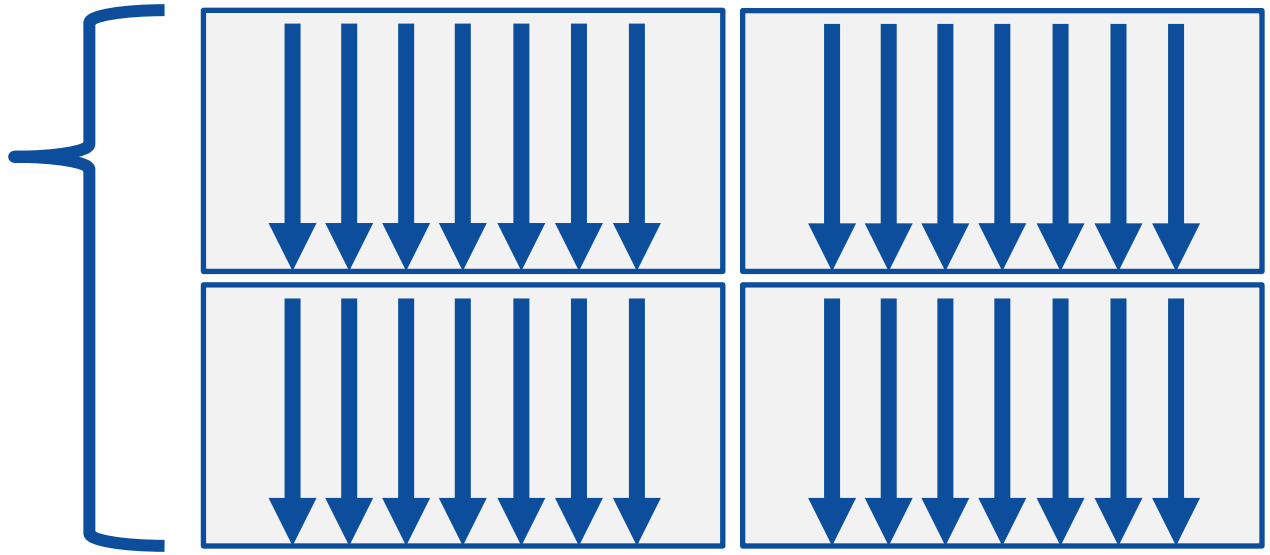
Kernels directive

Expressing parallelism

```
!$acc kernels

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

  do i = 1, M
    b(j) = 0
  end do

!$acc end kernels
```

並列化・最適化プロセスは kernels内で複数実行される

各並列ループには異なるサイズのgangsが割り当てられ、各gangsは別々に並列化と最適化が行われる



OpenACC
More Science, Less Programming

Kernels directive

Fortran array syntax

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

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

- kernelsディレクティブはparallelと異なりFortranのarray notation（配列構文）の並列化も可能
- parallelはloopディレクティブと対である必要があるが、loopディレクティブはarray notationを並列化可能なループとして認識できない
- kernelsはコードブロックを対象とできるので、正しく並列化が可能

OpenACC
More Science, Less Programming

Kernels vs parallel

Kernels

- ユーザーが指示し、コンパイラが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

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
- 各ループを別々に並列化しなければならない

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

vi. Loop directive

Loop directive

- 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
- ループをどのように並列化するかを様々な方法で指示できる (OpenMP + α)
- ループを並列化するためにはOpenACCの並列領域 (parallel or kernels region) に含まれている必要がある

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;
```

Loop directive

parallelディレクティブ内のループ並列化

```
!$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
```

- 左のコードの場合、最初のループはloop並列化されない
- 最初のループは冗長並列化が行われ、ループ全体が並列リソース（例えばスレッド）によって複数回実行される
- 2つ目のループはloop並列化によってループの各反復処理が並列リソース間で適切に分割実行される

OpenACC
More Science, Less Programming

Loop directive

kernelsディレクティブ内のループ並列化

```
!$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ディレクティブではloopディレクティブが暗黙に定義される
- ユーザーはloopディレクティブ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ディレクティブは不要だが、ループ自体の最適化はユーザーができる

OpenACC
More Science, Less Programming

Loop directive
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で、各ループにloopディレクティブを与えることができる
- 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ディレクティブを無視してもよいことになっている

OpenACC
More Science, Less Programming

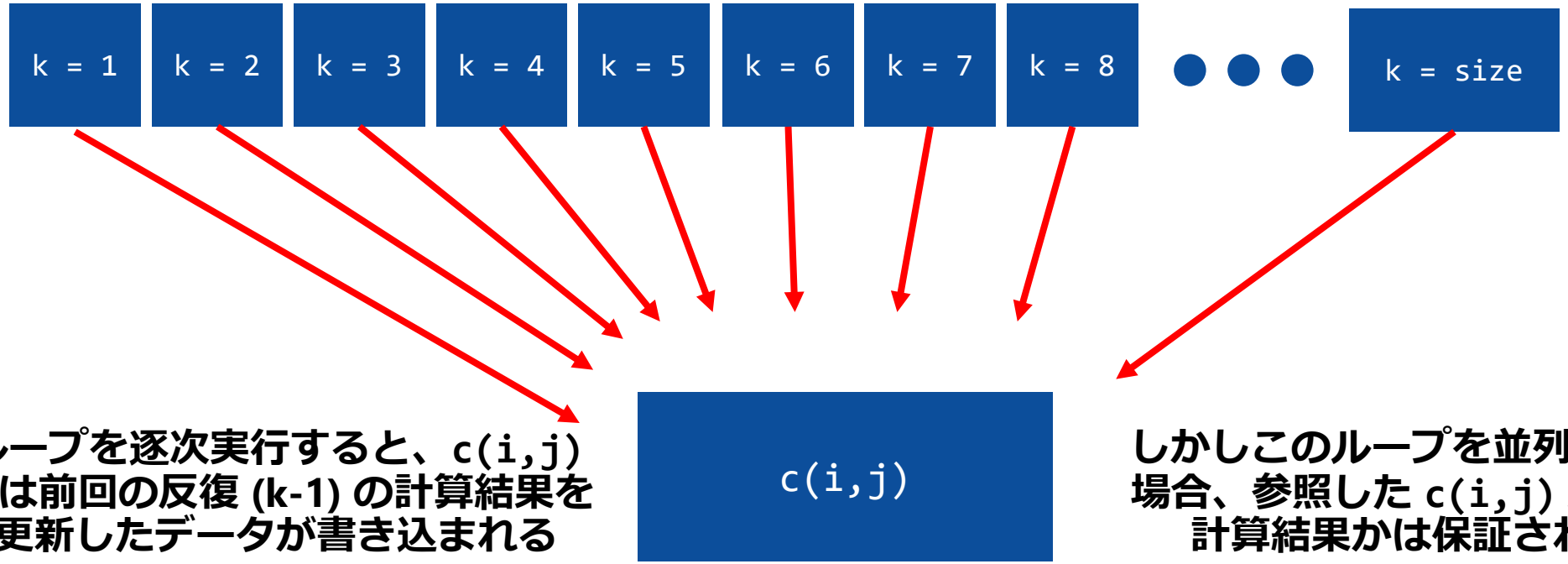
Reduction clause

- 右のコードにおいて最内ループは並列化不可
- 三重ループをそのまま並列化すると複数スレッドが c(i,j) に同時に書き込みを行うかもしれない
- 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節によりこの問題を解決する

```
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
```

Without a reduction

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



ループを逐次実行すると、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

しかしこのループを並列実行した場合、参照した c(i,j) が k-1 の計算結果かは保証されない

OpenACC
More Science, Less Programming

Reduction clause

- **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

```
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
```

```
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
```

OpenACC
More Science, Less Programming

Reduction clause

- 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
- コンパイラによってはkernelsよりparallelディレクティブのほうが最適な並列化方法を選択することもある

```
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
```

Reduction clause
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
```

Reduction clause operators

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

vii. 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
```

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
```

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
```

並列化できないループ

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
```

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
```

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

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
```

OpenACC
More Science, Less Programming

3. CPU-GPU間のデータ転送の最適化

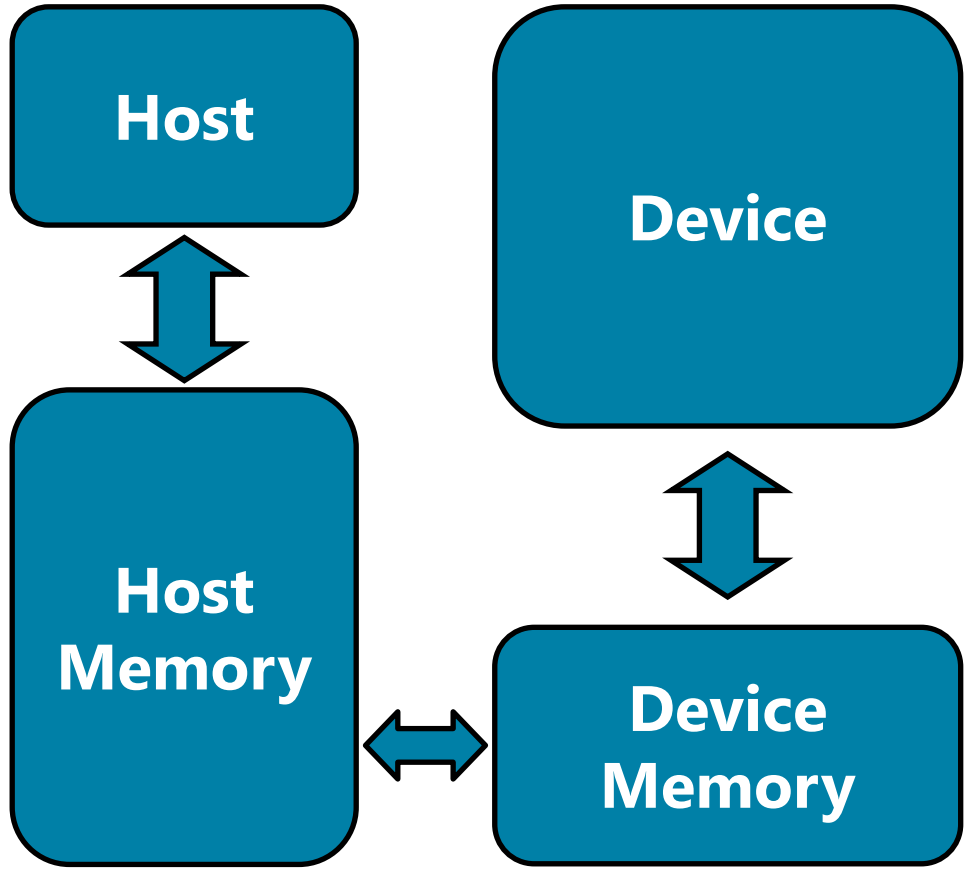
OpenACC
More Science, Less Programming

i. 基本的なデータ管理の考え方

基本的なデータ管理の考え方

ホストとデバイス間のやりとり

- 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
- 共有メモリ型のアクセラレータを使う限り、明示的なデータ管理は不要

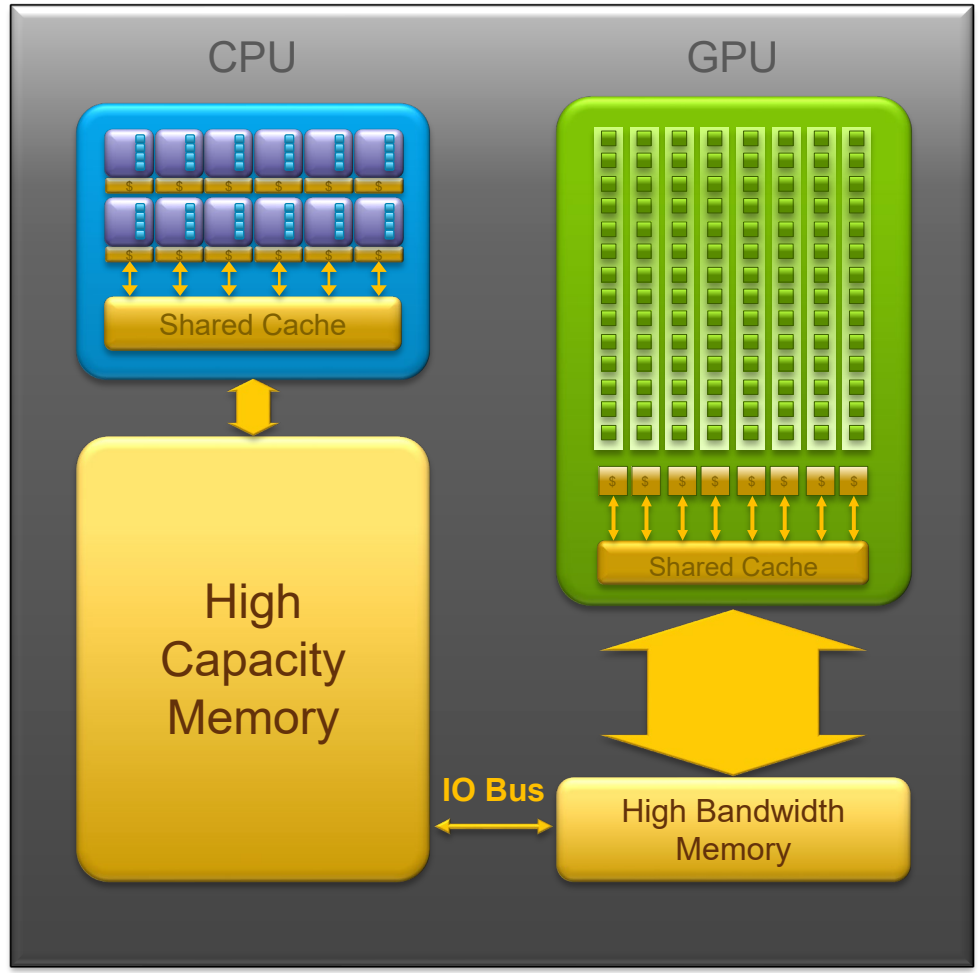


OpenACC
More Science, Less Programming

基本的なデータ管理の考え方

ホストとデバイス間のやりとり

- 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
- まず最初は、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を使い、データのやりとりを自動化する



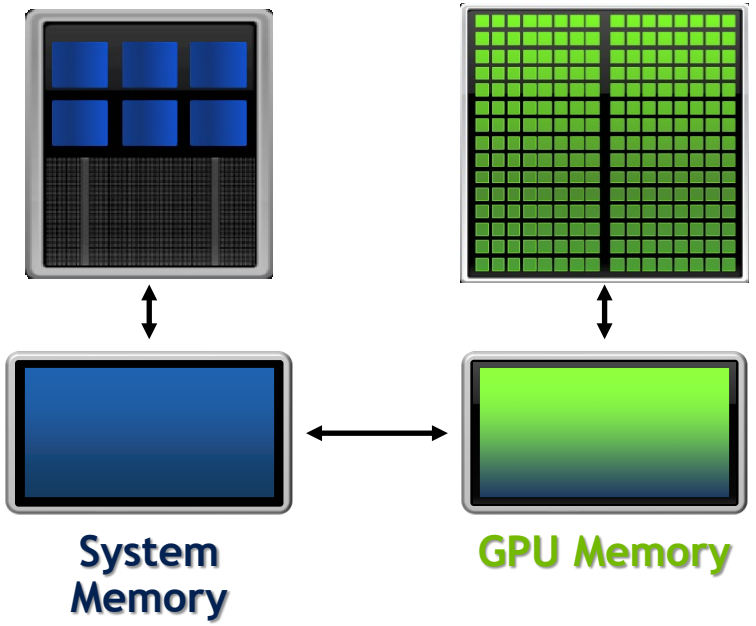
OpenACC
More Science, Less Programming

ii. CUDA Managed memory

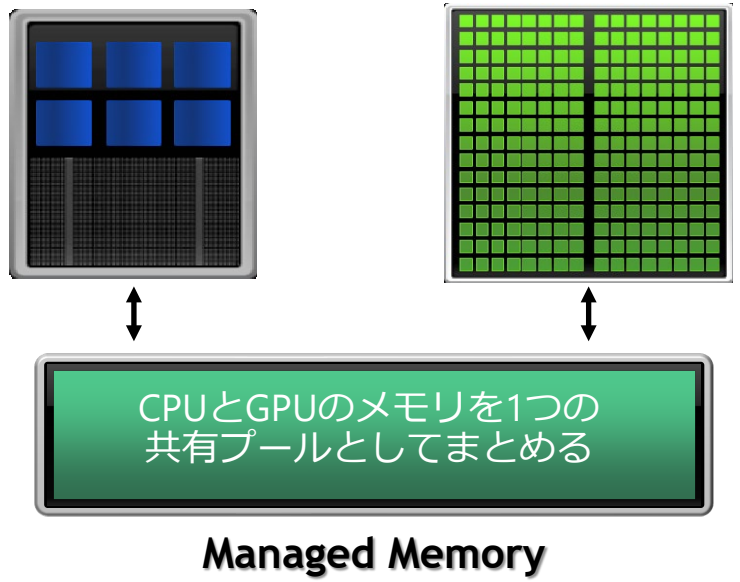
CUDA managed memory

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

CUDA managed memory
なぜ有用なのか

- Host-Device (CPU-GPU) 間の明示的なデータ転送の管理は複雑で困難
- NVIDIA (PGI) コンパイラはCUDA Managed MemoryをOpenACCで利用可能
- データ管理を「一旦」忘れることができる
- ユーザーは並列化に集中、データ管理を最適化フェーズで考えることが可能になる

Fortran

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

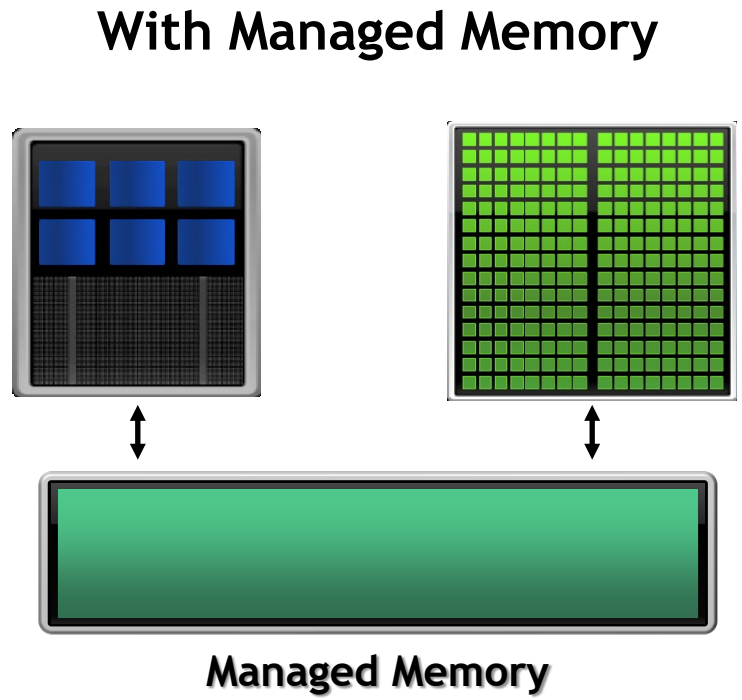
C/C++

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

OpenACC
More Science, Less Programming

Managed memory
制限事項

- ユーザーはほとんどの場合、データ管理を手動処理することでより高い性能を得られる
- メモリの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
- NVIDIA (PGI) コンパイラのみの機能
- Allocateやmallocでヒープ領域に確保されたメモリだけが利用可



OpenACC with managed memory

課題

```
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がない場合、コンパイラはA, Anewのサイズを指定し、かつコード整合性のために各反復でGPU-CPU間のデータコピーが必要

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

OpenACC
More Science, Less Programming

iii. OpenACC data clauses

Basic data management

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ディレクティブ内だけでなく、後述の**data, enter/exit dataディレクティブ**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

Basic data management

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ディレクティブ内だけでなく、後述のdata, enter/exit dataディレクティブにも追加できる

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

Basic data management

Host/Device間の明示的なデータコピー

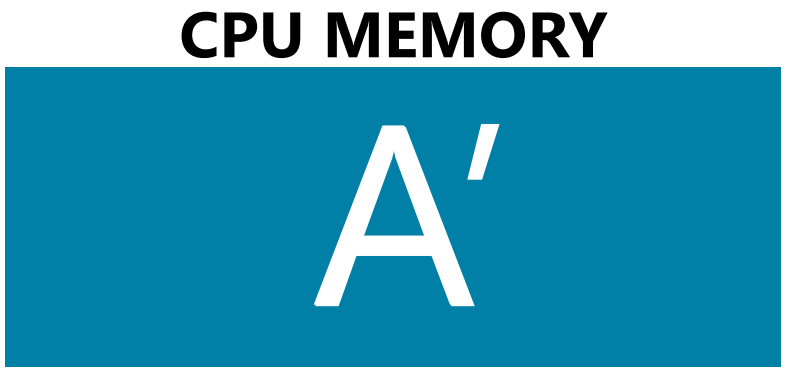
Fortran

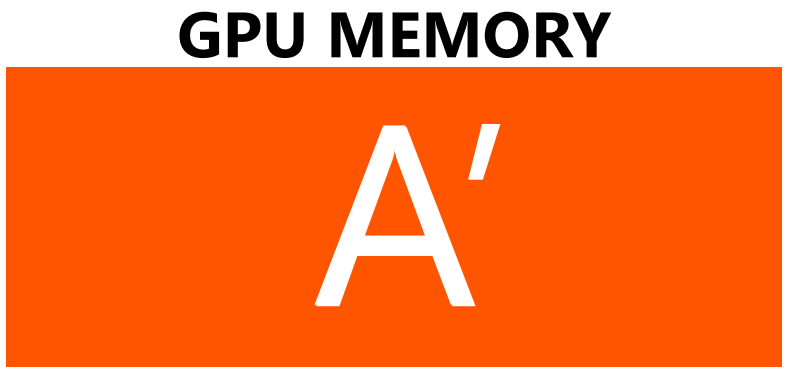
```
!$acc parallel loop copy(a(1:N))
do i = 1, N
 a(i) = 2 * a(i)
end do
```

OpenACC
More Science, Less Programming

Basic data management

Host/Device間の明示的なデータコピー





OpenACC
More Science, Less Programming

Data clauses

`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）
- 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
- C/C++: 2番目の数字にはデータサイズを記載

Fortran

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

C/C++

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

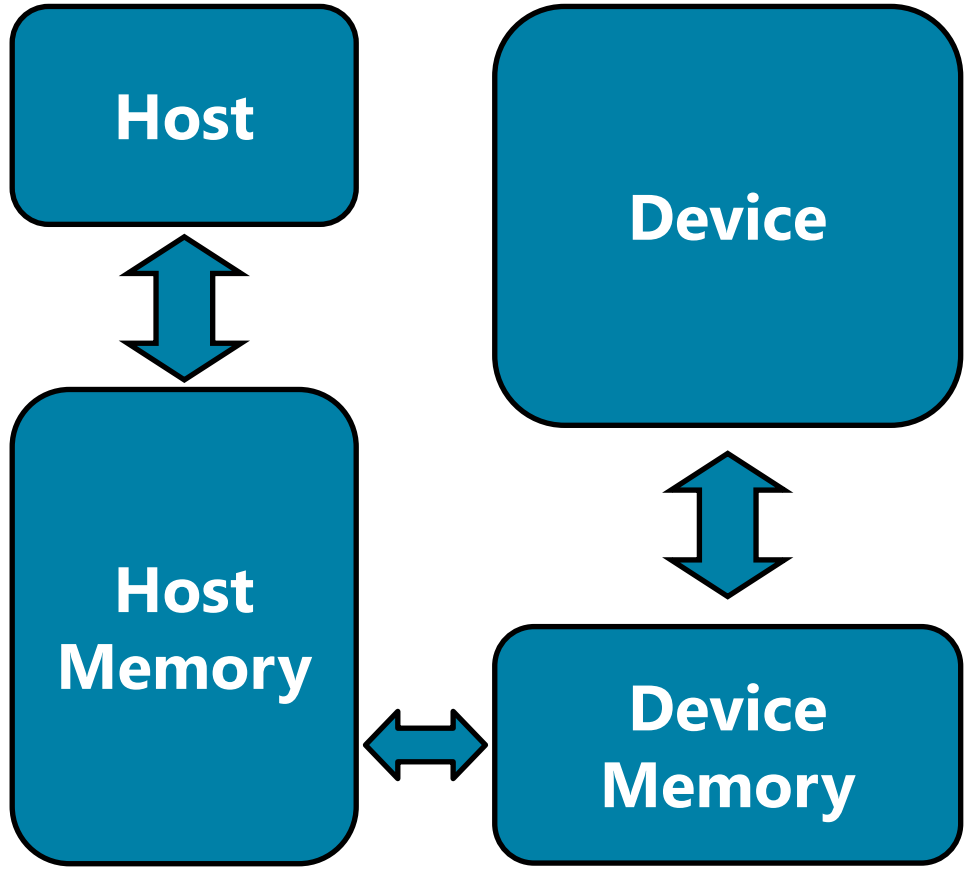
OpenACC
More Science, Less Programming

iv. Explicit memory management

Explicit memory management
データ管理の考え方

- 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

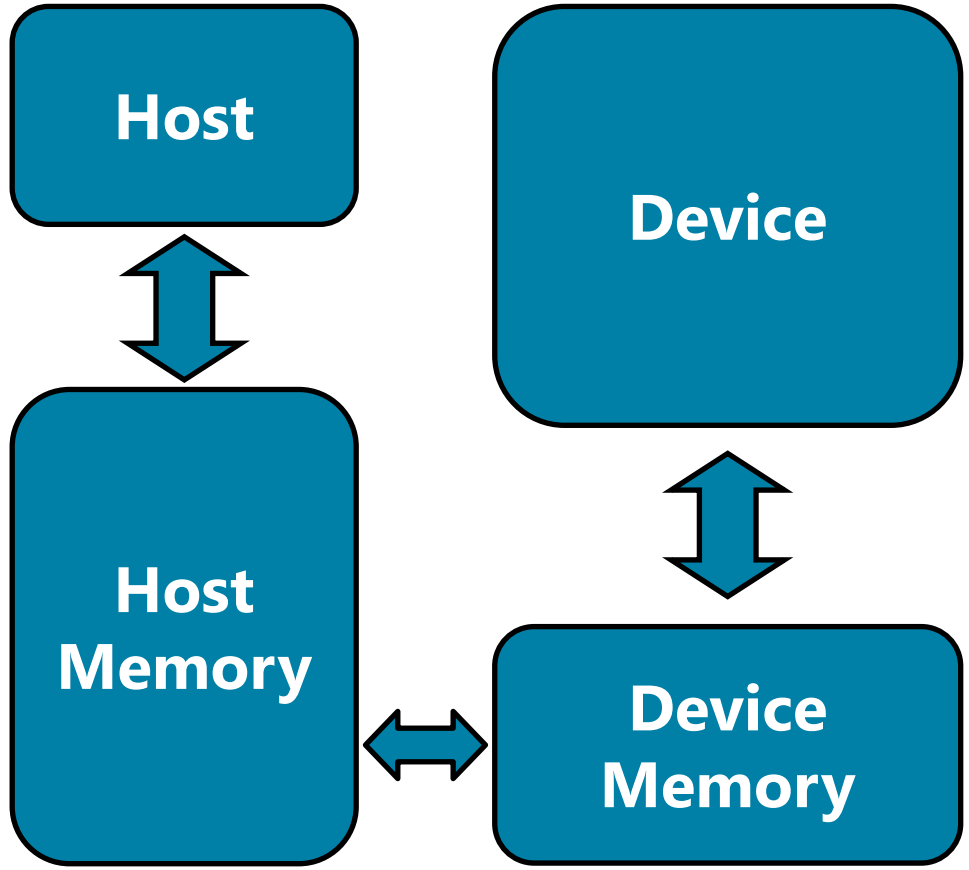
- Host/Deviceがメモリを共有していない場合、データのコピーが必要となる
- 性能を最大化するために、ユーザーは不要なデータコピーを避ける必要がある



OpenACC
More Science, Less Programming

Explicit memory management
重要な課題

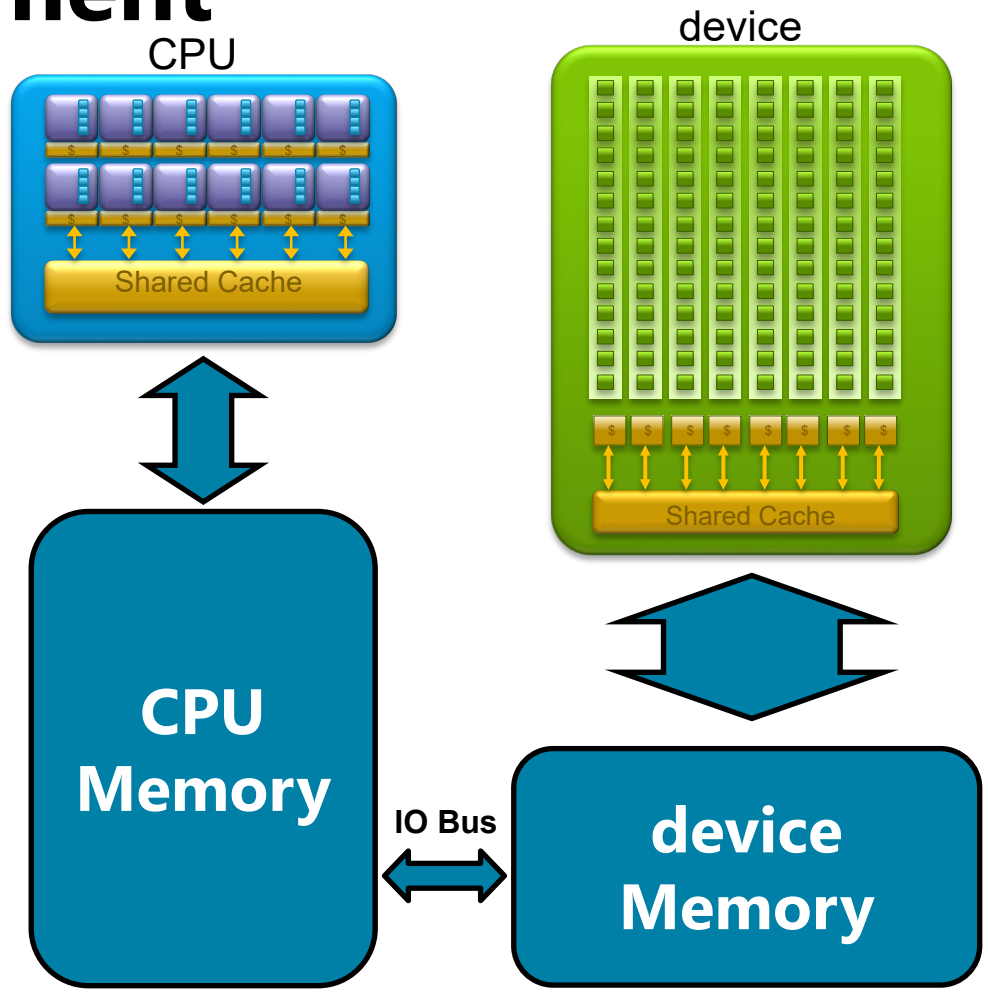
- 多くのDeviceは、Hostと別のメモリ空間を持つ
- 個々のメモリには異なるデータが格納され、自動では同期されないことが多い
- これらのメモリ間のデータ転送・共有は非常に時間がかかる処理となる



Explicit memory management

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. OpenACC data directive

OpenACC data directive

定義

- dataディレクティブは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節はユーザーが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 clauses

`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

Array shaping (cont.)

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

Array shaping (cont.)
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

Structured data directive

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

Structured data directive

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

Structured data directive

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. Explicit and implicit data regions

Implied data regions

定義

- すべてのkernelsとparallelディレクティブは、暗黙のデータ領域を持つ
- 暗黙の定義によりデータはそのディレクティブ内でのみ存在できる
- dataディレクティブで利用できるすべてのdata節は、kernels/parallelディレクティブでも使用できる

Fortran

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

OpenACC
More Science, Less Programming

Explicit vs. Implicit data regions

Explicit

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

Implicit

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

2つのコードは機能的に同じ処理を行っている

OpenACC
More Science, Less Programming

Explicit vs. Implicit data regions

制限事項

この場合、ExplicitなデータコピーはImplicitに比べ性能が高くなる

Explicit

```
!$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

Implicit

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. Unstructured data directives

Unstructured data directives

Enter Data Directive

- enter dataディレクティブは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ディレクティブを記述できるため、dataディレクティブの開始点と同じではない

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

Unstructured data directives

Exit Data Directive

- exit dataディレクティブは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で与えた配列に対して同数のexit dataが必要
- enter/exit data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

Unstructured data clauses

`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

Unstructured data directives
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))
```

Unstructured data directives

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)
  !$acc kernels
    A(1) = 0
  !$acc end kernels
  deallocate_array(A)
end program
```

- 左記の例ではenter/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UnstruHostコードに合わせて配置できる
- ctured dataディレクティブはクラス機能を持つC++において特に有効

OpenACC
More Science, Less Programming

Unstructured vs structured

Unstructured

- 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

```
!$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)
```

Structured

- 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dataでのみ存在

```
!$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. Data synchronization

OpenACC update directive

updateディレクティブ: 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ディレクティブ

Clauses:

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))
```

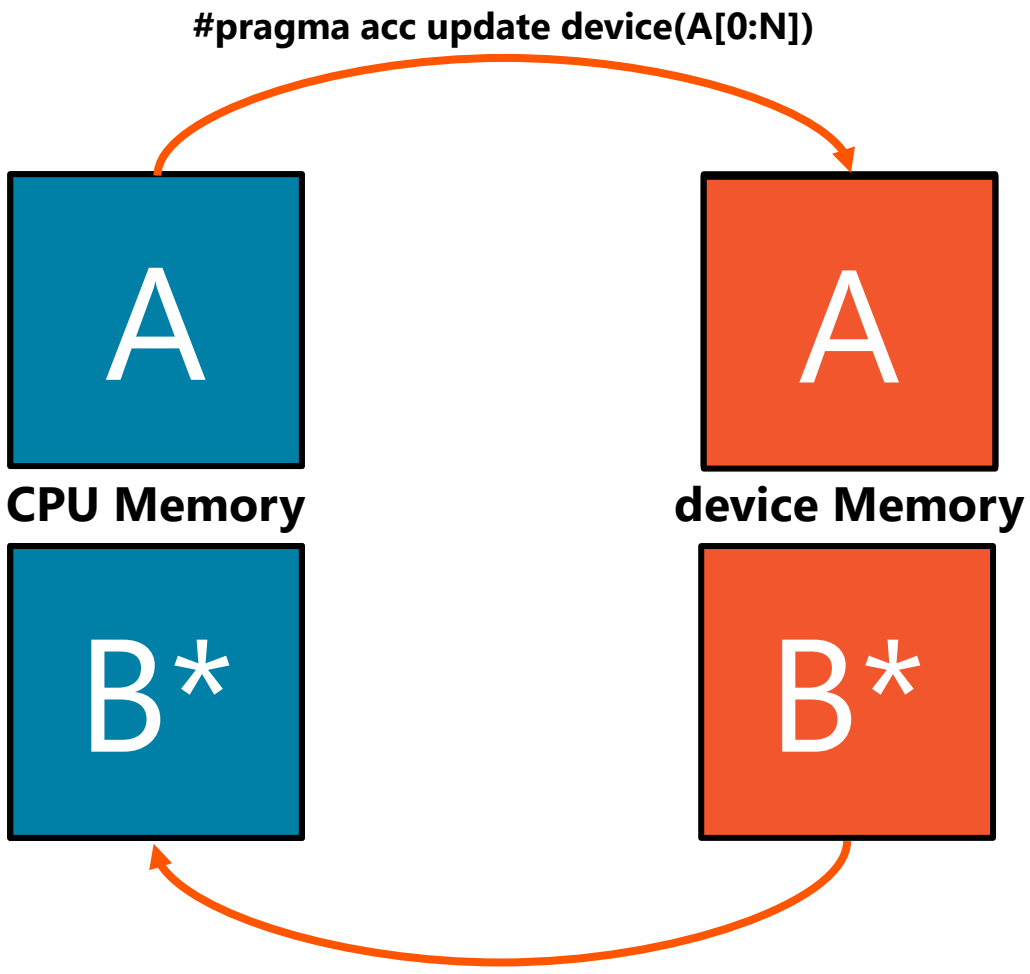
C/C++

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

OpenACC
More Science, Less Programming

OpenACC update directive

updateディレクティブ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



Synchronize data with update

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'A'のHostデータを変更する
- HostとDeviceの間で'A'のデータを同期するためにupdate deviceを実行する
- updateディレクティブがない場合、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

Auto clause

- **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
```

OpenACC
More Science, Less Programming

Auto clause

- kernelsディレクティブは暗黙的に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ディレクティブ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 clause

- 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 clause

- parallelディレクティブは暗黙的にindependent節が付与されるため、明示する必要はない
- parallelディレクティブではどのループが並列化可能かプログラマが判断し明示するため、independent節は不要

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 clause

- **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

```
!$acc parallel loop
do k = 1, size
  !$acc loop
  do j = 1, size
    !$acc loop seq
    do i = 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

Private and firstprivate clauses

- **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 clauses

- 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 clause

- 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ディレクティブ内ではfirstprivate 、kernelsディレクティブ内では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、以下はそのシチュエーションだが、これに限定されず必要な場合もある

1. 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スカラ変数
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 clause

- `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 clause

`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 clause

- `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

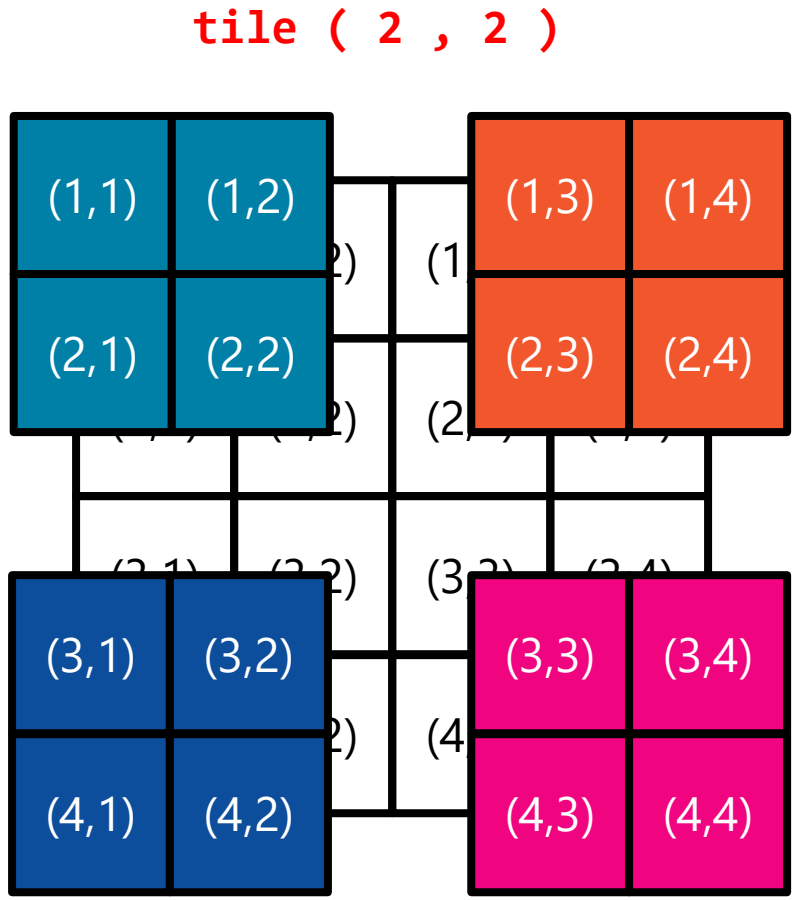
```
!$acc kernels loop tile(32, 32)
do i = 1, size
  do j = 1, size
    do k = 1, size
       c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
!$acc end kernels
```

OpenACC
More Science, Less Programming

Tile clause

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
end subroutine
```

1. **kernelsディレクティブ**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

OpenACC
More Science, Less Programming

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

end function
```

1. **kernelsディレクティブで並列化を試みる（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

    !$acc kernels
    p(2:imax-1,2:jmax-1,2:kmax-1)= wrk2(2:imax-1,2:jmax-1,2:kmax-1)
    !$acc end kernels
  end do
  !$acc end data
end subroutine
```

OpenACC
More Science, Less Programming

Dataディレクティブの最小化

メモリの確保・解放処理を最小限にする

- dataディレクティブは1つの関数で閉じる必要がある
- 実際のアプリケーションは複数のファイル・関数が定義され変数をやりとりするので、dataディレクティブでは不便
- どこでも呼び出せるenter/exit dataは使い勝手が良い
- 理想的には、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
- 最適化の戦略は、CUDAとOpenACCでほぼ同じだが、OpenACCは低工数でシンプルなAPIを用いてプログラムをGPU化できるのが大きく異なる

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のコード例
    iii. OpenACCの基本構文
    iv. Parallel directive
    v. Kernels directive
    vi. Loop directive
    vii. 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. 基本的なデータ管理の考え方
    ii. CUDA managed memory
    iii. OpenACC data clauses
    iv. Explicit memory management
    v. OpenACC data directive
    vi. Implied data regions
    vii. Unstructured data directives
    viii. data synchronization
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）にて複数のHPC関連のサポートを展開
- 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