チュートリアル

科学技術計算の分野で始まったGPU（Graphic Processing Unit）による汎用計算は、製造業、医療、金融など様々な分野へ広まり、モバイル、組み込みの分野でも使われ始めています。当初は専用言語を使う必要がありGPUによる汎用計算は気軽に始められるものではありませんでしたが、用途が広がると同時に、従来の言語でGPU向けにプログラミングする環境が整ってきました。OpenACCはその中の一つで、既存のプログラムにディレクティブと呼ばれるヒント情報を追加するだけで、コンパイラが自動でGPUコードを生成する開発手法です。今回のチュートリアルでは、全4回のシリーズとして、このOpenACCを使ってアプリケーションを効率良くGPUで加速する方法を解説します。

OpenACCで始めるGPUコンピューティング：ループの並列化

成瀬 彰

1　はじめに

前回は、OpenACCを使ってどのようにアプリケーションを並列化するのがよいのか、我々が推奨する方法を紹介した。我々の推奨方法は、アプリケーションを一挙に並列化するのではなく、処理時間の長い部分から一つずつ順番に並列化する方法である（図1）。

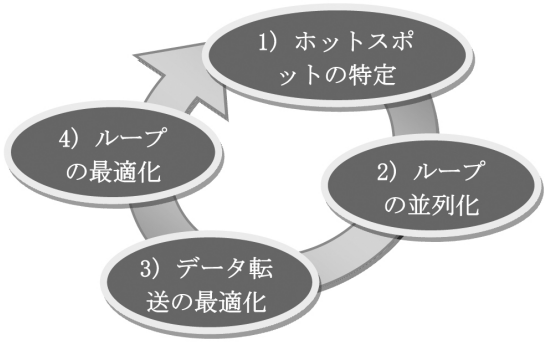

図1　OpenACCによるアプリケーション並列化のサイクル

この方法は、一見すると、回りくどい方法に見えるかもしれない。しかし、並列化作業の過程で混入したバグの修正が、いかに困難なものかを考慮すると、並列化に要する時間をトータルでは最も短くできる方法であると我々は確信している。実際、コード行数が十万行を越える大規模アプリであっても、この方法に基づいて、アプリケーション全体のGPU向け並列化を達成した事例も存在する。

今回は、このアプリケーション並列化サイクルの、1）ホットスポットの特定と、2）ループの並列化を説明してゆく。

2　ホットスポットの特定

コードの並列化を始める前に、プログラムのどの部分の処理に時間を要しているかを把握することは重要である。アプリケーション全体の実行時間に対して、相当の割合を占めるサブルーチンやループネストはホットスポットと呼ばれるが、並列化はこのホットスポットから始めるのが良い。

2.1　プロファイリング・ツール

GNU gprof[1]、Intel VTune[2]、Vampir[3]、TAU[4]等々、アプリケーション情報を採取・収集する、いわゆるプロファイリングツールは世の中に多数存在する。各ツールにはそれぞれ長所・短所があり、どのツールが良いとは一概には言えないが、より詳細なプロファイリングデータがあるほど、並列化の作業を効率良く進められる。

続きはWebで
日本計算工学会誌「計算工学（Vol.21, No.3）」HP:
http://www.jsces.org/Issue/Journal/

筆者紹介

なるせ あきら
1996年、名古屋大学大学院工学系研究科修士課程修了（情報工学専攻）。同年、富士通研究所に入社、大規模サーバ開発、HPCシステム開発など、様々なプロジェクトに参加、関連するハードウェアやソフトウェアの研究開発に従事。2009年、山下記念研究賞受賞。2010年、SACSIS2010最優秀論文賞受賞。2013年、NVIDIAに参加、シニアデベロッパーテクノロジーエンジニアとして様々なアルゴリズムとアプリケーションのGPU向け並列化に従事、GPUコンピューティングの普及に努めている。

例えば、コード行数が数百～数千行のサブルーチンの場合、そのサブルーチンがホットスポットと分かるだけではあまり意味がない。具体的に、サブルーチン内のどの部分で時間がかかっているかを特定できないと、効率良く並列化を進められない。また、ホットスポットの性能律速要因が何なのかも重要である。もしボトルネックがディスクIOやプロセス間通信であれば、GPUのようなアクセラレータに出番はない。演算性能やメモリアクセス性能が性能リミッターであれば、GPUで加速可能だ。しかし、そこに十分なデータ並列性がなければ、大きな性能向上は望めない。そのホットスポットがループネストだとすると、ループ反復回数は何回なのか、多いのか、少ないのか、このような情報も、並列化の効果が見込めるかどうかの観点で、ホットスポットの重要度を決める上で有用である。

GPU向け並列化の過程では、CPUのプロファイルデータだけでなく、GPUのプロファイルデータも採取できるのが望ましい。PGIのプロファイリングツールpgprof[5]は、CPU向けプロファイリング機能とGPU向けプロファイリング機能の両方をサポートしている。現時点でOpenACCコンパイラとして普及しているのはPGIコンパイラであり、本チュートリアルでもPGIコンパイラを使用することもあり、本稿ではプロファイリング事例にpgprofを使用する。

2.2　正しいプロファイルデータ：

プロファイルデータはとても有用だが、正しいプロファイルデータの採取に注意する必要がある。アプリケーションには様々なパラメータやオプションが存在する。これらのちょっとした違いにより、アプリケーションは特性が変わることがある。例えば、実行の簡単な設定、例えば、小さな問題サイズの設定でアプリケーションを実行した場合と、本来の用途の問題サイズで実行した場合とで、ホットスポットが全く変わることもある。適切ではないプロファイルに基づいて並列化を進めると、本当のホットスポットを見逃す危険性がある。本来の用途に近いパラメータ設定でアプリケーションを実行したときのプロファイルデータを使うことに注意して欲しい。

2.3　プリファイリング事例

CPUのプロファイルデータの採取例を説明する。サンプルコードは、本チュートリアルの第一回目の最後に示したヤコビ反復法のFortranコードである。

プロファイリング実行ファイルの作成

PGIコンパイラでプロファイルデータを採取するには、コンパイル・オプションに-Mprofを追加して実行ファイルを生成する。以下の操作で、プロファイリング用の実行ファイルa.outが生成される。

```
$ pgf90 –Mprof jacobi.f90
```

プロファイルデータの採取

プロファイルデータの採取には、pgprofコマンドを使用する。オプション-oを付けて実行すると、このオプションで指定したファイルにプロファイルデータが記録される。以下の操作では、ファイルa.perfにデータが記録される。

```
$ pgprof –o a.prof ./a.out
```

プロファイルデータの表示

プロファイルデータの表示にもpgprofコマンドを使用する。表示の場合は、オプション-iでプロファイルデータが入ったファイルを指定する。

```
$ pgprof –i a.prof --cpu-profiling-mode flat
======== CPU profiling result (flat):
Time(%)      Time  Name
 100.00%    20.57s  MAIN_
```

このプログラムにはMAIN以外にサブルーチンがないので、MIANしか表示されないが、サブルーチンが多数ある場合は、各サブルーチンの実行時間と全実行時間に占める割合が、実行時間の長いものから表示される。なお、上記例のオプション--cpu-profiling-modeは表示形式を指定するオプションで、bottom-up、top-down、flatから選択できる。

プロファイルデータの表示（命令レベル）

このプログラムにはループネストが2つあるが、サブルーチン単位のプロファイルからでは、どちらのループネストがホットスポットか特定できない。pgprofは、オプション--cpu-profiling-scopeで表示する情報の粒度を、function、instructionから選択できる。デフォルトはfunctionだが、このオプションにinstructionを選択すると、命令レベルのプロファイルが表示される。

```
$ pgprof --cpu-profiling-scope instruction --cpu-profiling-
mode flat –i a.prof
======== CPU profiling result (flat):
Time(%)      Time  Name
 10.50%    2 . 16s  MAIN_ (./jacobi.f90:42 0x2af)
  5.49%    1 . 13s  MAIN_ (./jacobi.f90:52 0x335)
  4.67%    960ms  MAIN_ (./jacobi.f90:42 0x24d)
  4.38%    900ms  MAIN_ (./jacobi.f90:43 0x2c0)
  4.18%    860ms  MAIN_ (./jacobi.f90:41 0x238)
  4.18%    860ms  MAIN_ (./jacobi.f90:52 0x329)
  4.04%    830ms  MAIN_ (./jacobi.f90:42 0x2a2)
  3.99%    820ms  MAIN_ (./jacobi.f90:42 0x269)
  3.84%    790ms  MAIN_ (./jacobi.f90:51 0x300)
  3.84%    790ms  MAIN_ (./jacobi.f90:42 0x275)
...
```

各行の括弧内の「ファイル名:行番号」は、各命令に

対応するソースコード上の場所を示している。41-43行は1番目のループネスト、51-52行は2番目のループネストに対応しており、上記の情報から1番目のループネストがよりホットスポットと分かる。

ここではテキストベースのプロファイリング事例を説明したが、pgprofはGUI表示もサポートしている。pgprofには他にも様々な機能があるが、詳細は[5]を参照して頂きたい。

3　ループの並列化

アプリケーションのホットスポットが特定できたら、次はホットスポットの並列化である。アプリケーションによっては、ホットスポットが多数あるかもしれないが、それらを一挙に並列化する必要はない。優先度の高いホットスポットから順番にOpenACCの並列化ディレクティブを挿入、一つずつループを並列化していけばよい。

GPUコンピューティングの識者は、CPU・GPU間のデータ転送はどうなるのか、不安に思うかもしれない。ループを並列するディレクティブを入れただけでも、コンパイラはソースコードを解析して必要なCPU・GPU間のデータ転送処理を自動で追加するので、この段階ではデータ移動を考慮する必要はない。多くの場合、コンパイラが決定したデータ転送の方法は最適ではなく、CPU・GPU間のデータ転送が必要以上に行われる。そのため、この段階ではアプリケーション性能が低下することも珍しくない。OpenACCにはデータ転送を制御するディレクティブがあるが、それを用いたデータ転送の最適化は、次のステップで取り組む。このステップではループの並列化に集中する。

OpenACCは並列化領域を指定する方法として2つのディレクティブを用意している。Kernelsコンストラクトと、Parallelコンストラクトである。以下、それぞれの特徴や使い方を説明する。

3.1 Kernelsコンストラクト

Kernelsコンストラクトは、ソースコード内の並列化可能領域を指定するディレクティブの一つである。主に、指定領域内の処理を具体的にどう並列化するかを、コンパイラに決定させるときに使用する。指定領域内には、複数の処理ブロック、例えば、複数のループネストが含まれていても良い。コンパイラはループネスト毎にGPUカーネルを生成するかもしれないし、複数のループネストを一つにまとめてGPUカーネルを生成するかもしれない。また、各ループネストを並列化しても安全（＝正しい計算結果が得られる）か否かの判定はコンパイラに委ねられている。コンパイラが安全でないと判断して、並列化が行われないこともある。Kernelsコンストラクトは、「コンパイラおまかせ方式」とも言える。以下、FortranとCそれぞれの、Kernelsコンストラクトの使用例を示す。

```
------------------------------
[Fortran]
    !$acc kernels
    do i=1,n
         x(i) = 0
         y(i) = i
    enddo

    do i=1,n
        y(i)=y(i)+2.0*x(i)
    enddo
    !$acc end kernels
------------------------------
[C/C++]
    #pragma acc kernels
    {
      for (i=0;i<n;i++) {
          x[i] = 0.0;
          y[i] = i;
      }
      for (i=0;i<n;i++) {
          y[i] += 2.0*x[i];
      }
  }
------------------------------
```

このコードサンプルは2つのループで構成されており、1番目のループで配列値を設定し、2番目のループではその配列上で計算を行っている。どちらのループも、イタレーション間にデータ依存性が無く、並列化可能なので、コンパイラは2つのループのそれぞれにGPUカーネルを生成するだろう。Kernelsコンストラクトでは、内在する並列性を具体的にどうGPUリソースにマップするかはコンパイラに託されている。アクセラレータの種類に依存しない、抽象度の高い記述が可能な方法であり、ソースコードのポータビリティを重視するときに有用である。

3.2　Parallelコンストラクト

Parallelコンストラクトも、並列化可能領域を指定するディレクティブである。Parallelコンストラクトは、後述するLoopコンストラクトと組み合わせたparallel loopディレクティブとして使われることが多い。以下、FortranとCそれぞれの、Parallelコンストラクトの使用例を示す。

```
------------------------------
[Fortran]
    !$acc parallel loop
    do i=1,n
         x(i) = 0
         y(i) = i
    enddo
    !$acc end parallel
```

```
!$acc parallel loop
do i=1,n
    y(i)=y(i)+2.0*x(i)
enddo
!$acc end parallel
```

[C/C++]

```
#pragma acc parallel loop
for (i=0;i<n;i++) {
    x[i] = 0.0;
    y[i] = i;
}
#pragma acc parallel loop
for (i=0;i<n;i++) {
    y[i] += 2.0*x[i];
}
```

Parallelコンストラクトでは、Kernelsコンストラクトとは違い、各処理ブロック（≒ループ or ループネスト）に対してディレクティブを挿入する必要がある。上記の例では、2つのループの各々にParallel loopディレクティブが挿入されいている。1つのParallelコンストラクトから、1つのGPUカーネルが生成されると捉えると分かりやすいだろう。これは、コード開発者が明示的に並列化方法を指示する場合に使うディレクティブとして、Parallelコンストラクトが設計されているからである。従って、GPUのタイプや種類に特化した並列化方法や最適化方法を指示する場合には、Parallelコンストラクトが適している。

3.3　KernelsとParallelの違い

並列領域を指定するディレクティブとして、何故、KernelsとParallelの2種類のディレクティブがOpenACCには存在するのか、戸惑う人は多いだろう。実際、これらはとても良く似ているが、微妙な違いが存在する。Kernelsディレクティブが並列化できそうなところをコンパイラに知らせる「ヒント」だとすると、Parallelディレクティブは並列化できるところをコンパイラに指示する一種の「命令」である。

Kernelsディレクティブの場合、コンパイラにはある種の自由度が与えられる。あるループを並列化するか否か、ターゲットアクセラレータに対してどのように並列性をマッピングするのかは、コンパイラの責任で行われる。コンパイラが安全と判断すれば、ループは並列化されるが、コンパイラが安全と判断するに足る十分な情報がないと、ループは並列化されない。従って、Kernelsディレクティブでは、コンパイラを変えると、挙動や性能が大きく変わることがある。

Parallelディレクティブはプログラマーからコンパイラへの指示であり、コンパイラはその命令に従って、ループを並列化する。プログラマーからの指示が十分でない部分はコンパイラが補うが、原則としては、あるループを並列化しても安全か否かの判断は、プログラマーの責任となる。従って、並列化できないループに対して誤ってParallelディレクティブを適用すると、計算結果が不正となることもある。

KernelsディレクティブとParallelディレクティブのどちらを使えば良いのか。それぞれ長所・短所があるので、その選択は開発者次第だが、プログラム並列化の経験の少ないプログラマーにはKernelsディレクティブ、並列処理の経験豊富なプログラマーにはParallelディレクティブの使用が推奨される。これらディレクティブは併用可能なので、例えば、どう並列したらよいか分からない、そもそも並列化してよいかどうか分からない処理ブロックに対してはKernelsディレクティブを使用し、並列化の具体的なイメージのある部分にはParallelディレクティブを使用するといった様に、同一プログラム内でも場所によって使い分けるても良い。

C/C++での注意点：

C/C++コードでは、配列ポインターが、関数パラメータとして渡される場合、Parallelディレクティブでは並列化できるのに、Kernelsディレクティブでは並列化できないといった現象がしばしば発生する。これは、ポインタ・エイリアス[6]が原因であることが多い。ポインタ・エイリアスとは、別々のポインタを経由するメモリアクセスが、メモリ上の同じ場所をアクセスする問題であるが、ポインタ・エイリアスの可能性があるループの並列化は安全ではない。このようなループであってもParallelディレクティブを使うと、コンパイラはプログラマーの指示に従ってそのループを並列化するが（Parallelディレクティブは、コンパイラへの命令）、Kernelsディレクティブを使うと、コンパイラは安全ではないと判断してループを並列化しない。その配列ポインターでエイリアスが発生しないことが確かな場合は、C/C++ではrestrictキーワードを使って、それをコンパイラに通知することができる。もしKernelsディレクティブでループが並列化されない現象が発生したら、配列ポインターにrestrictキーワードを付けることを試すと良いだろう。

Fortranでの注意点：

Fortranコードでは、例えば array(:,:,:) = 0 のような配列記述で、doループを記述することなく、全部もしくは指定の配列要素に対して、同じ処理を適用できる。この部分にKernelsディレクティブを使うと、コンパイラは自動で暗黙的にdoループを作成して並列化するが、Parallelディレクティブを使う場合には並列化が行われない。このような配列記述にParallelディレクティブを適用するには、プログラマーが明示的にプログラムを書き換えてdoループを入れる必要がある。

3.4　Loopコンストラクト

Loopコンストラクトは、それが挿入された次の行のループに関するヒント・指示を、コンパイラに渡すときに使用する。Loopコンストラクトは、例えばparallel

loopディレクティブのように、parallelコンストラクトと組み合わせて使用できるが、kernelsコンストラクトで指定した並列領域内でも使うことができる。

Loopディレクティブを使うと、様々なヒントや指示をコンパイラに渡すことができる。その指示の種類は、性能を最適化するため指示と、正しい結果を得るための指示に、大別される。以下では、正しい結果を得るためのLoopディレクティブの使い方を一部説明する。

3.4.1 Private節

Private節は、ループイタレーション毎に独立して使用すべき変数・配列（private変数・配列）をコンパイラに伝えるのに使われる。例えば、ループ内で、配列tmpが計算過程の値を保持する一時配列として使われていたとする。もし、この配列tmpをイタレーション間で共用する状態で、全イタレーションを並列実行すると、配列tmp上でデータ競合が発生し、正しい結果が得られない。多くの場合、private化すべき変数・配列は、コンパイラが自動で識別するので、常にprivate節を使用せねばならない、ということはない。しかし、コンパイラが正しく識別できないケースもある。その場合は、以下のように、Private節を使って、privateな変数・配列を指示する必要がある。

```
------------------------------
[Fortran]
    !$acc parallel loop private(tmp)
    do i=1,n
        tmp(1) = ...
        tmp(2) = ...
        x(i) = x(i)+tmp(1)*tmp(2)
    enddo
    !$acc end parallel
------------------------------
[C/C++]
    #pragma acc parallel loop private(tmp)
    for (i=0;i<n;i++) {
        tmp[0] = ...;
        tmp[1] = ...;
        x[i] += tmp[0]*tmp[1];
    }
------------------------------
```

3.4.2 Reduction節

Reduction節は、各イタレーションの計算結果をまとめて一つの値を計算する処理、いわゆる縮約計算に使われる。縮約計算とは、例えば、総和、総乗、最大値、最小値を計算する処理である。例えば、イタレーション毎に計算した結果の合計（sum）を求める処理では、以下の様にReduction節を追加することで、ループを並列化できる。

```
------------------------------
[Fortran]
    !$acc parallel loop reduction(+:sum)
    do i=1,n
        x(i) = ...
        sum = sum+x(i)
    enddo
    !$acc end kernels
------------------------------
[C/C++]
  #pragma acc parallel loop reduction(+:sum)
    for (i=0;i<n;i++) {
        x[i] = ...;
        sum += x[i];
    }
------------------------------
```

上の例は総和を求めているので、縮約操作として+を指定している。縮約計算の内容が、総乗、最大値、最小値の場合は、それぞれ、*、max、minを縮約操作として指定すればよい。なお、縮約対象として指定できるのは変数のみである。Reduction節では、配列を縮約対象に指定できない。

3.5 ヤコビ反復法の並列化

ここまで説明したディレクティブを使って、ヤコビ反復法のループを並列化してみよう。ヤコビ反復法のサンプルコードのメインループには、2つの処理ブロック（ループネスト）が含まれている。1番目のループネストは、配列Aに記録されている現時点の配列値から、各配列要素の次の値を計算し、それを配列Anewに記録する処理である。2番目のループネストでは、配列Anewの値が配列Aにコピーバックされる。どちらのループネストも、基本的には、全ての配列要素を独立して並列に処理可能だが、1番目のループネストでは、各配列要素の現在の値と次の値の差分の絶対値の最大値の計算（変数errorに関わる計算）、いわゆる縮約計算が行われていることに注意が必要である。

3.5.1 Parallelコンストラクト使用

Parallelコンストラクトで並列化する場合、ディレクティブの入れ方は以下の通りとなる。

```
------------------------------
[Fortran]
  do while ( error > tol .and. iter < iter_max )
    error = 0.0
    !$acc parallel loop reduction(max:error)
    do j=2, N-1
      !$acc loop reduction(max:error)
      do i=2, M-1
        Anew(i,j) = 0.25 * &
          (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
        error = max(error, abs(Anew(i,j) – A(i,j) ))
      end do
    end do
    !$acc end parallel
    !$acc parallel loop
    do j=2, N-1
      !$acc loop
      do i=2, M-1
        A(i,j) = Anew(i,j)
      end do
    end do
    !$acc end parallel
    if (mod(iter,100)==0) write(*,*) iter, error
    iter = iter + 1
  end do
------------------------------
[C/C++]
  while ( error > tol && iter < iter_max ) {
    error = 0.0;
    #pragma acc parallel loop reduction(max:error)
    for (int j = 1; j < N-1; j++ ) {
      #pragma acc loop reduction(max:error)
      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] ));
      }
    }
    #pragma acc parallel loop
    for (int j = 1; j < N-1; j++ ) {
      #pragma acc loop
      for (int i = 1; i < M-1; i++ ) {
        A[j][i] = Anew[j][i];
      }
    }
    if (iter % 100 == 0) printf("%d, %f\n", iter, error)
    iter++;
  }
------------------------------
```

C/C++コード、Fortranコード、どちらもループネストの外側のループにparallel loopディレクティブを挿入し、内側のループの前にloopディレクティブを挿入する。多重ループの場合、内側のループに対してもloopディレクティブの挿入が必要なことに注意されたい。Parallelコンストラクトでは、loopコンストラクトが付けられたループは、並列化可能であることを意味する。従って、これでループネスト内の全てのループが並列化可能であることをコンパイラに指示したことになる。

1番目のループネストでは、変数errorに対して縮約計算（最大値の算出）が行われているので、reduction節を使って、これを指示する。なお、コンパイラによっては、reduction節がなくても、自動で縮約計算処理を認識して、暗黙的に縮約計算を行うコードを生成することがある。コンパイラの自動認識機能に依存してもよいが、コードのポータビリティを重視するなら明示的にreduction節を入れるのがよいだろう。

OpenACCコードから、PGIコンパイラを使ってGPU用実行ファイルを生成するには、オプション-acc –ta=teslaを付ける。追加でオプション-Minfo=accelを付けると、コンパイラのコード解析結果に関する情報が表示される。特に開発過程では、このオプションの使用を推奨する。以下に、このサンプルコードをPGI Fortranコンパイラ（v16.4）でビルドしたときの出力を示す。コンパイラメッセージの詳細は、ここでは説明しないが、このメッセージを見ると、どのループが並列化されたのか、どのように並列化されたのか、縮約計算は指示通り行われているか、計算に必要なデータは正しく転送されているか、などを確認することができる。

```
------------------------------
$ pgf90 -acc -ta=tesla -Minfo=accel jacobi.f90
jacobi:
     37, Accelerator kernel generated
         Generating Tesla code
         37, Generating reduction(max:error)
         38, !$acc loop gang ! blockidx%x
         40, !$acc loop vector(128) ! threadidx%x
             Generating reduction(max:error)
     37, Generating copyout(anew(2:4095,2:4095))
         Generating copyin(a(1:4096,1:4096))
     40, Loop is parallelizable
     47, Accelerator kernel generated
         Generating Tesla code
         48, !$acc loop gang ! blockidx%x
         50, !$acc loop vector(128) ! threadidx%x
     47, Generating copyin(anew(2:4095,2:4095))
         Generating copyout(a(2:4095,2:4095))
     50, Loop is parallelizable
------------------------------
```

3.5.2　Kernelsコンストラクト使用

次に、Kernelsコンストラクトを使用してOpenACC化する。Kernelsコンストラクトの基本は、コンパイラにおまかせ、である。その考えに従うと、kernelsディレクティブの入れ方は以下の通りとなる。

```
------------------------------
```

```
[Fortran]
    do while ( error > tol .and. iter < iter_max )
      error = 0.0
      !$acc kernels
      do j=2, N-1
        do i=2, M-1
          Anew(i,j) = 0.25 * &
            (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
          error = max(error, abs(Anew(i,j) – A(i,j) ))
        end do
      end do
      do j=2, N-1
    do i=2, M-1
      A(i,j) = Anew(i,j)
    end do
    end do
    !$acc end kernels
    if (mod(iter,100)==0) write(*,*) iter, error
    iter = iter + 1
  end do
-------------------------------
[C/C++]
   while ( error > tol && iter < iter_max ) {
     error = 0.0;
     #pragma acc kernels
     {
     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];
       }
     }
     }
     if (iter % 100 == 0) printf("%d, %f\n", iter, error)
     iter++;
   }
-------------------------------
```

Kernelsディレクティブを使うと、上記のように、ディレクティブ挿入量を少なくすることができる。ソースコードをポータビリティの高い状態にできると言ってよいだろう。一方で、コンパイラがループをどう認識したか、ループを正しく並列化できているか、並列化可能なところで逐次コードが生成されていないか、その確認は開発者の責任となる。以下にkernelsディレクティブでOpenACC化したFortranコードをPGIコンパイラ(v16.4)でビルドしたときのコンパイラ出力を示す。このメッセージを見ると、全てのループが並列化されており、集約計算も正しく対処されていることが分かる。

```
-------------------------------
$ pgf90 -acc –ta=tesla -Minfo=accel jacobi-k.f90
jacobi:
     37, Generating copyout(anew(2:4095,2:4095))
         Generating copyin(a(1:4096,1:4096))
         Generating copyout(a(2:4095,2:4095))
     38, Loop is parallelizable
     39, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
          38, !$acc loop gang, vector(4) ! blockidx%y
threadidx%y
          39, !$acc loop gang, vector(32) ! blockidx%x
threadidx%x
        41, Generating implicit reduction(max:error)
     45, Loop is parallelizable
     46, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
          45, !$acc loop gang, vector(4) ! blockidx%y
threadidx%y
          46, !$acc loop gang, vector(32) ! blockidx%x
threadidx%x
-------------------------------
```

4　おわりに

今回は、我々が推奨するOpenACCによるアプリケーション並列化サイクルの前半、1)ホットスポットの特定、2)ループの並列化、この2ステップを具体的な事例を用いて説明した。サンプルコードに用いたヤコビ反復法で時間のかかる処理は、この2ステップを適用することで、既にGPU上で実行できる状態になっている。しかし、実はこの状態では、CPU実行よりもGPU実行の方が遅い。その理由は、CPUとGPU間のデータ転送が最適化されておらず、データ転送が必要以上に行われているためである。

次回は、OpenACCによるアプリケーション並列化サイクルの後半、データ転送を最適化する方法、そして、ループをより良く並列化する方法について説明する。

■参考文献

[1] GNU gprof: https://sourceware.org/binutils/docs/gprof/
[2] Intel VTune: https://software.intel.com/en-us/intel-vtune-amplifier-xe
[3] Vampir: https://www.vampir.eu/
[4] TAU: https://www.cs.uoregon.edu/research/tau/home.php
[5] PGI Profiler User's Guide: http://www.pgroup.com/doc/pgprofug.pdf
[6] Pointer aliasing: https://en.wikipedia.org/wiki/Pointer_aliasing