OpenACC in a Nutshell

Hideo Matsufuru（KEK）for HPC-Phys 勉強会
Version 1.0 [2018.12.01]

この文書は OpenACC を用いてコードを作成する際の、一つのスタイルに沿った必要事項をまとめる。一通りイメージが掴めたら[1], [2]などを参照して、自分に合ったスタイルを探して欲しい。

OpenACC とは

Fortran または C/C++のコードに指示文(directive)を挿入することにより、GPU などのアクセラレータに実行をオフロードする部分を指定する API(Application Programming Interface)規格 [1]。コンパイラがそれを解釈してオフロードのためのコードを生成する。指示文を無視してコンパイルすれば元のコードと同じ動作となる。独立な多数のスレッドによって処理を並列化できるようなループを、オフロードの対象とすることが多い。

考え方

- プログラムを開始する CPU をホストと呼び、オフロード先のアクセラレータをデバイスと呼ぶ。一般にデバイスはホストとは別のメモリ空間を持つ。ホストとデバイス間のデータ転送、デバイス上で動作するコード(カーネル)の実行はホスト側から制御される。OpenACC では変数はホストとデバイスで同じ名前で参照されるため、どちら側のデータか文脈によって区別する。
- コンパイラは指示文に従って元のコードからオフロードするコード部分を切り出し、カーネルコードを生成する。そのために必要な、ホストとデバイス間で通信が必要なデータ、カーネルとして切り出す部分（同期の取り方含む)、スレッド並列化するループとその仕方、を指定する必要がある。これらはそれぞれ OpenACC の data 構文、parallel（または kernels)構文、loop 構文に対応する。コンパイラになったつもりで考えると directive の意味するところが解りやすい（かもしれない)。
- 一般的にデバイスは階層構造を持つ。OpenACC では 3 レベル: gang, worker, vector を想定。Gang は粗粒度の実行単位(互いに同期しない単位、NVIDIA の Streaming Multiprocessor に対応)、worker は細粒度の実行単位(互いに同期可能な単位、NVIDIA の warp に対応)、vector は SIMD などベクトルユニットのレーンに対応。Gang 数、gang 毎の worker 数、worker 毎の vector 数は parallel/kernels 構文で指定できる。各ループをどの単位に割り当てて並列化するかは loop 構文で指定できる。

オフロード処理の流れ

(1) ホスト上のメモリ確保
(2) ホストからデバイスメモリへのデータ転送
(3) デバイス上のカーネル実行
(4) デバイスからホストへのデータ転送
(5) デバイス上のメモリ解放

メモリ確保やデータ転送の制御の仕方によって、2 つの処方が考えられる。
(A) カーネル実行の直前にデバイスメモリを確保し、実行後メモリを解放する。

data 構文で囲った領域の前後で(1), (5)を暗黙に行い、(2) (4)を data 構文の指示節(clause)を使って指定する。

(B) ホスト上のメモリは初めに確保しておき、できるだけ使いまわす。

(1)は enter data 構文で行い、(2), (4) は update 構文(device/host)で行う。(5) は exit data 構文で行う。このやり方は OpenACC 2.0 以降で使用可。(HPC-Phys 勉強会[2018.08.22]: 松古資料参照)

data 構文を使っても同じことはできるが、enter/exit data 構文を使うと見通しが良い。

指示文の一般形

C/C++: **#pragma acc** *directive [clause [[,] clause]...] new-line*

次の行に指示文を続けるときは ¥ (バックスラッシュ)。

指示文の最後には改行が必要。{ と } で囲った構造化ブロック(*)、もしくは直後の命令に適用される。

F77: **!$acc** *directive [clause [[,] clause]...]*

!の代わりに*, c を用いてもよい。次の行に続けるときは続きの行を !$acc+ で始める。

適用する構造化ブロックを

!$acc end *directive*

で囲む。

F90: 次の行に続けるときは行の最後に &

次行は !$acc で始める。

他は F77 と同じ。

以下では C/C++の場合の書式を引用する。

(*) 構造化ブロックとは、実行文のブロックで以下の特徴を持つものを言う。一つの入り口と一つの出口を持ち、ブロック内からブロック外への、またはブロック外からブロック内への分岐を持たないこと。

Data 構文

指定した変数について、対応するメモリ領域をデバイス上に確保し、支持節(clause)で指定した動作を行う。**この構文の始点と終点で、デバイスメモリの確保・解放とデータ転送が行われる**ことに注意。

#pragma acc data *[clause [[,] clause]...] new-line*

代表的な指示節(clause):

- **copy**(*list*) 最初にホストからデバイスにデータをコピー、最後にデバイスからホストにコピー
- **copyin**(*list*) 最初にホストからデバイスにデータをコピー
- **copyout**(*list*) 最後にデバイスからホストへデータをコピー
- **create**(*list*) デバイスメモリ上に変数領域を作成、データのコピーは行わない
- **present**(*list*) デバイス上に既に変数領域が存在することを示す

変数リストは、配列の場合、例えば次のように指定する。部分配列も可能。多次元配列も可能。

C/C++: a[0:n] 起点とサイズ。この例では a[0]から n 個の要素。

Fortran: a(1:n) 起点と終点。この場合は a(1)から a(n)までの n 個の要素。

起点とサイズを省略した場合、Fortran では配列全体とみなされる。C/C++ではポインタとして扱われるので転送の際には必ずサイズを指定する。

暗黙のデータ属性

data 構文で明示的にデータ転送属性を指定していない場合、配列あるいは集合データ型には present_or_copy（データが既にあれば present, なければ copy), スカラ変数には firstprivate(ホストの値で初期化してスレッドに private, parallel 構文の clause)が設定される（data 構文で明示的に指定するのがベター。Cf. **default(none)** clause)。ループ変数は thread に private となる。

Parallel 構文

この指示文で囲われた領域が、スレッド並列に実行される。同様の指示文として kernels 構文があり、よりコンパイラの解釈に依存した処理が行われる（kernels 構文を用いた開発については[2]を参照)。Parallel 構文を使用する場合は、依存関係の確認などはユーザの責任となり、より意図に沿った動作が期待できる。**この指示文で囲った領域を出る際にスレッド間の同期が取られる。**内部に複数の並列化対象 loop がある場合でも、loop 毎に同期は取られないことに注意。

#pragma acc parallel *[clause [[,] clause]...] new-line*

代表的な指示節：

- **num_gangs**（ *expression* ） gang 数を設定。省略すると実装依存のデフォルト値。
- **num_workers**（ *expression* ） worker 数を設定。
- **vector_length**（ *expression* ） vector レーン数を設定。
- **async**（ *expression-list* ） 非同期に実行したい場合、数字などで区別して指定。wait 指示文で同期をとる。

  #pragma acc wait（ *expression-list* ）

Loop 構文

この指示文の直後にあるループをスレッド並列化する。

#pragma acc loop *[clause [[,] clause]...] new-line*

代表的な指示節：

- **gang, worker, vector** どのレベルに割り当てるかを指定する。
- **reduction**(*operator:list*) reduction 処理を行う。Operation としては、+, * など。
- **collapse**(n) 直後の n 個のループをまとめる。

以上は上記の処方(A)で必要な構文。処方(B)で必要になるものとして以下を追加する。

exter data 構文

デバイスメモリ上に変数領域をアロケートする。以下の形で使うことが多い（create 指示節は data 構文の場合と同じ意味)。copyin 指示節も利用可。

#pragma acc enter data create（ *list* ）*new-line*

exit data 構文

enter data 構文とペアで使う。以下の形で使うことが多い。

#pragma acc exit data delete（ *list* ）*new-line*

update 構文

既にデバイス上に確保されている変数、配列に対して、データ転送を行う。

#pragma acc update *[clause [[,] clause]...] new-line*

代表的な指示節：

- **device** (*list*)　host → device のデータ転送
- **host** (*list*)　device → host のデータ転送。**self** clause は host と同じ。
- **async** (*expression*)　parallel 構文の場合と同様、データ転送を非同期にできる。Parallel 構文によるデバイスカーネルの実行とも非同期に実行可能。wait 構文で同期をとる。

C 言語コードによる例（based on Himeno benchmark）

```
#pragma acc data copy(pp[0:size]),  ¥
              copyin(size, mrows, mcols, mdeps, omega, pwrk1[0:size],  ¥
                     pbnd[0:size], pa[0:4*size], pb[0:3*size], pc[0:3*size]), ¥
              create(pwrk2[0:size])   // 転送する配列データは起点とサイズで指定する
 {                                                  // 構造化ブロックを囲む
  for(n=0 ; n<nn ; n++){
    gosa = 0.0;

#pragma acc parallel num_workers(1) vector_length(256)
 {                                                  // 構造化ブロックを囲む
#pragma acc loop gang worker vector reduction(+ : gosa)
  for(int site = 0 ; site < size; site++){    // 3-level のループを 1 次元化済み
    int k = site % mdeps;
    if(k == 0 || k == mdeps-1) continue;
    … (略)
    float s0 =  pa[IDX(0,i,j,k)]*pp[IDX(0,i+1,j,  k)]
           + pa[IDX(1,i,j,k)]*pp[IDX(0,i,  j+1,k)]
    … (略)
    float ss = …
    gosa += ss*ss;
    pwrk2[IDX(0,i,j,k)] = pp[IDX(0,i,j,k)] + omega * ss;
  }
 } //acc parallel                                  // thread 間の同期

#pragma acc parallel num_workers(2) vector_length(32)
 {
#pragma acc loop gang worker vector
  for(int site = 0 ; site < size; site++){
    … (略)
  }
 } // acc parallel                                  // thread 間の同期

} /* end n loop */

} // acc data        // copy, copyout のデータがホストに送られる。メモリ領域解放
```

Fortran コードによる例（based on Himeno benchmark）

```
!$acc data copy(p),
!$acc+     copyin(kmax, jmax, imax, omega, wrk1, bnd, a, b, c),
!$acc+     create(wrk2)       // 配列データは起点と終点で指定、省略すると全体を転送

    DO loop=1,nn
      gosa=0.0

!$acc parallel num_workers(1) vector_length(128)
!$acc loop gang worker vector collapse(3) reduction(+ : GOSA)
      DO K=2,kmax-1                          ! collapse(3) で 3 重ループをまとめる
        DO J=2,jmax-1
          DO I=2,imax-1
            S0=a(I,J,K,1)*p(I+1,J,K)+a(I,J,K,2)*p(I,J+1,K)
               +a(I,J,K,3)*p(I,J,K+1)
               … (略)
            SS=(S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
            GOSA=GOSA+SS*SS
            wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
          enddo
        enddo
      enddo
!$acc end parallel     ! thread 間の同期

!$acc parallel num_workers(2) vector_length(32)
!$acc loop gang worker vector collapse(3)
       DO K=2,kmax-1
          DO J=2,jmax-1
             DO I=2,imax-1
                p(I,J,K)=wrk2(I,J,K)
             enddo
          enddo
       enddo
!$acc end parallel     ! thread 間の同期

    enddo
 CC End of iteration
!$acc end data       !copy, copyout のデータがホストに転送される。メモリ領域解放
```

高速化のポイント

- ホスト-デバイス間のデータ転送の最小化
- Gang, worker, vector サイズの最適化
- 各スレッドのタスクの最適化
- 非同期なデータ転送、カーネル実行
- データレイアウトの最適化（coalesced access）

Device の指定について

複数のアクセラレータを持つシステムの場合、実行プロセスによって異なるデバイスを使用する必要が生じる場合がある(例： MPI 並列で各プロセス毎に GPU を割り当てる、複数個のプログラムを同時に実行、など)。OpenACC のランタイムライブラリを使って設定できる。

- **acc_get_num_devices**(*devicetype*) 利用できるデバイス数を返す。
- **acc_set_device_num**(*devicenum, devicetype*) そのプロセスで使用するデバイスを数値で指定。

C/C++では #include “openacc.h”、Fortran では use openacc を実行。*devicetype* は header ファイルで定義されている。NVIDIA のデバイスには acc_device_nvidia が定義されている(C では enum, Fortran では Integer)。

プリプロセッサマクロ

条件コンパイルには、次のプリプロセッサマクロが使える。OpenACC 使用時は _OPENACC が(バージョンに従って yyyymm のように)セットされる。

コンパイル (PGI)

> pgcc -fast -acc -ta=tesla:cc30 -Minfo=accel -o a.out sample.c

> pgfortran -fast -acc -ta=tesla:cc30 -Minfo=accel -o a.out sample.f

PGI コンパイラでは、OpenACC コードでは **-acc** オプションを指定する。更に、

- **-ta=tesla:cc30** ターゲットアーキテクチャを指定（cc は compute capability でアーキテクチャによって決まっている。GTX 680 (Kepler) は 3.0 → cc30）
- **-Minfo=accel** コンパイル時にアクセラレータ用コードの詳細を出力

その他の便利なオプション:

- **-fast** 一般的な最適化セットを適用
- **-Minline=level:3** 関数のインライン化を 3 重に行う（ネスト階数を指定できる）

例えば https://www.softek.co.jp/SPG/Pgi/TIPS/option1.html を参照。

便利なコマンド

- **pgaccelinfo** (PGI compiler 環境のみ)
  ハードウェア情報を表示する。-ta オプションに指定する compute capability も表示される。
- **nvprof** (NVIDIA 環境のみ)
  簡易プロファイラ。 > nvprof ./a.out のように使う。

Reference

[1] OpenACC Home http://www.openacc.org/
OpenACC specification (最新 2.7)など
[2] PGI「OpenACC ディレクティブによるプログラミング」(日本語版はソフテックが提供)
http://www.softek.co.jp/SPG/Pgi/OpenACC/
[3] John Cheng「CUDA C プロフェッショナルプログラミング」（インプレス, 2015）