The OpenACC™ Application Programming Interface

Version 1.0

November, 2011

日本語版(2012/10/19)

目次

1. はじめに

このドキュメントは C,C++,Fortran プログラムのコードをホスト CPU から接続されたアクセラレータデバイスへオフロードするための OpenACC アプリケーションプログラミングインターフェース(OpenACC API) を集合的に定義するコンパイラ指示文、ライブラリルーチン、環境変数について記述する。概説された方法はオペレーティングシステムや様々な種類のホスト CPU やアクセラレータで移植可能なアクセラレータプログラミングのためのモデルを提供する。指示文は ISO/ANSI 標準 C,C++,Fortran 基礎言語を拡張し、プログラマが標準ベースの C,C++,Fortran を使用してアクセラレータ目標にアプリケーションを追加的に移植できるようにする。

このドキュメントで定義されている指示文とプログラミングモデルにより、プログラマはホストとアクセラレータ間のデータやプログラムの転送の管理や、アクセラレータの起動と終了をする必要なしにアクセラレータを使用可能なアプリケーションを作成できる。それどころか、それらの詳細すべてはプログラミングモデルで暗黙で、OpenACC API を使用可能なコンパイラと実行環境により管理される。このプログラミングモデルにより、プログラマはアクセラレータのローカルに置くデータの指定、アクセラレータ上のループの配置の指示、同様のパフォーマンスに関連する詳細を含むコンパイラで利用可能な情報を増やすことができる。

1.1 スコープ

この OpenACC API ドキュメントはアクセラレータデバイスへのオフローディングの対象となるホストプログラムの領域をユーザーが指定するユーザー指示アクセラレータプログラミングのみを扱っている。プログラムの起動はホストで実行される。このドキュメントは全体としてホストプログラミング環境の機能や制限について記述しない。つまり、アクセラレータでオフロードされるコードのループと領域の指定に限られている。

このドキュメントはコンパイラや他のツールによりコードの領域を自動検出し、アクセラレータへオフロードすることについて記述しない。このドキュメントは単一ホストに接続された複数のアクセラレータへループやコード領域をオフロードすることは記述していない。将来のコンパイラは自動オフローディングや同種の複数アクセラレータや異種の複数アクセラレータを考慮にいれているかもしれないが、これらの機能がこのドキュメントで記述されることはない。

1.2 実行モデル

OpenACC API を利用可能なコンパイラによって対象とされている実行モデルは GPU のような接続されたアクセラレータデバイスを用いたホスト指示実行である。ユーザーアプリケーションの大部分はホストで実行する。計算の集中する領域はホストの管理下でアクセラレータデバイスにオフロードされる。デバイスは一般的に作業共有ループを含む *parallel 領域*や、一般的にカーネルとして実行される 1 つ、ないし複数のループを含む *kernels 領域*を実行する。アクセラレータを対象にした領域においてさえ、アクセラレータデバイス上のメモリ割当、データ転送の開始、アクセラレータへのコード送信、並列領域への引数渡し、デバイスコードのキューイング、完了の待機、ホストへの結果の転送、メモリの解放による実行をホストは指揮しなければならない。ほとんどの場合、ホストはデバイスで実行される一連の演算を次々とキューに入れることができる。

ほとんどの現在のアクセラレータは並列の 2 つまたは 3 つのレベルをサポートする。ほとんどのアクセラレータは実行ユニットで完全な並列実行する粗粒度並列をサポートする。粗粒度並

列演算間の同期のサポートは限られているだろう。多くのアクセラレータは同様に細粒度並列をサポートする。それはしばしば単一実行ユニット内の実行の複数スレッドとして実行され、一般的にレイテンシの長いメモリ操作を許容するために実行ユニット上ですばやく切り替えられる。最後に、ほとんどのアクセラレータは同様にそれぞれの実行ユニット内で SIMD かベクトル演算をサポートする。デバイス側の実行モデルは多様な並列レベルを見せ、プログラマはそれらの違いを理解することが求められる。例として完全並列ループと、ベクトル化できるが、文の間で同期が必要なループを挙げる。完全並列ループは粗粒度並列実行用にプログラムされることができる。依存のあるループは粗粒度並列実行を許すために分割されるか、または細粒度並列かベクトル並列か逐次を使う単一実行ユニット上で実行するようプログラムされるかしなければならない。

1.3 メモリモデル

ホストのみのプログラムとホスト+アクセラレータのプログラムの最も重要な違いはアクセラレータ上のメモリはホストメモリから完全に独立していることである。例としてこれはほとんどの現在の GPU の実状である。この場合、デバイスメモリはホストの仮想メモリにマップされていないため、ホストはデバイスメモリを直接読み書きできないだろう。ホストメモリとデバイスメモリ間のすべてのデータ移動は一般的にダイレクトメモリアクセス(DMA)を使って独立したメモリ間のデータを明白に移動するホスト経由ランタイムライブラリコールによって実行されるべきだ。同様に、アクセラレータがホストメモリを読み書きできると思うのは、一部のアクセラレータデバイスでサポートされているとしても妥当ではない。

独立したホストとアクセラレータメモリの概念は CUDA や OpenCL などの低レベルアクセラレータプログラミング言語で非常に明白で、メモリ間のデータ移動はユーザーコードを支配できる。OpenACC モデルではメモリ間のデータ移動は暗黙で、プログラマからの指示文に基づいてコンパイラにより管理される。しかしながら、プログラマは以下のような多くの理由のために独立したメモリの可能性を知っておくべきである。

- ホストメモリとデバイスメモリ間のメモリバンド幅は与えられたコード領域を効果的に加速するために必要な計算量のレベルを決める。
- 限定されたデバイスメモリサイズは巨大なデータ上で計算するコード領域のオフローディングを妨げる。

アクセラレータ側で、一部のアクセラレータ(現在の GPU のような)は weak memory model を提供する。特に、それらは異なる実行ユニットによって実行される処理間のメモリ一貫性に対応していない、即ち同じ実行ユニットにおいてさえ、メモリ一貫性はメモリ処理が明確なバリアで分けられているときにのみ保証される。もしそうでないなら、もしある処理がひとつのメモリ領域を更新し、もう一つが同じ領域を読む、または二つの処理が同じ領域に値を書き込むならば、ハードウェアは同じそれらの実行に対する同じ結果を保証しないであろう。コンパイラがこの実際には潜在的なエラーのいくつかを見つけられる一方、それにもかかわらず、一致しない数値結果を生成するアクセラレータ並列やカーネル領域を書くことができる。

一部の現在のアクセラレータはソフトウェア管理のキャッシュを持ち、一部はハードウェア管理のキャッシュを持ち、ほとんどは特定の状況でのみ使うことができ、読み込み専用データに限られたハードウェアキャッシュを持っている。CUDA や OpenCL 言語のような低レベルプログラミングモデルにおいて、それらのキャッシュの管理はプログラマに任されている。
OpenACC モデルではそれらのキャッシュは指示文の形でプログラマから与えられるヒントを頼りにコンパイラにより管理される。

1.4 このドキュメントの構成

このドキュメントの残りは次のように構成されている。

2 章 *ディレクティブアクセラレータ領域を示したり、コンパイラがループのスケジューリングやデータの分類のために利用可能な情報を増やしたりするために使われる* C,C++,Fortran 指示文について説明する。

3 章 *ランタイムライブラリルーチン* アクセラレータデバイス機能の問い合わせや実行時にアクセラレータ有効プログラムの動作を管理するためのユーザー呼び出し可能な関数とライブラリルーチンを定義する。

4 章 *環境変数* 実行時にアクセラレータ有効プログラムの動作をコントロールするために使われるユーザー定義可能な環境変数を定義する。

5 章 *用語解説* このドキュメントで使用される共通用語を定義する。

1.5 参考文献

- *American National Standard Programming Language C*, ANSI X3.159-1989 (ANSI C).
- ISO/IEC 9899:1999, *Information Technology–Programming Languages–C* (C99).
- ISO/IEC 14882:1998, *Information Technology–Programming Languages–C++*.
- ISO/IEC 1539-1:2004, *Information Technology–Programming Languages–Fortran–Part 1: Base Language*, (Fortran 2003).
- *OpenMP Application Program Interface*, version 3.1, July 2011
- *PGI Accelerator Programming Model for Fortran & C*, version 1.3, November 2011
- *NVIDIA CUDA™ C Programming Guide*, version 4.0, May 2011.
- *The OpenCL Specification*, version 1.1, Khronos OpenCL Working Group, June 2011.

2. 指示文

この章は OpenACC 指示文の構文と動作について記述する。C と C++では OpenACC の指示文は言語から提供される**#pragma** メカニズムを用いて記される。 Fortran では OpenACC 指示文はユニークなセンチネルにより識別される特別なコメントを用いて記される。もしサポートが無効かサポートされていないならば、一般的にコンパイラは OpenACC 指示文を無視する。

制約

- OpenACC 指示文は Fortran の PURE または ELEMENTAL プロシージャ中に現れてはいけない。

2.1 指示文形式

C と C++では OpenACC 指示文は**#pragma** メカニズムで記される。OpenACC 指示文の構文は

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

各々の指示文は**#pragma acc** から始まる。指示文の残りは C と C++の pragma の慣習に従う。空白が**#**の前後で使われてもよい。空白は指示文中で語を区切るために必要となる。**#pragma acc** に続く前処理トークンはマクロ置換を受ける。指示文は大文字と小文字が区別される。OpenACC 指示文は直後の文やブロックやループに適用する。

Fortran では OpenACC 指示文は自由形式のソースファイルで次のように記される。

!$acc *directive-name [clause [[,] clause] ...]*

コメント接頭辞(**!**)はどの列で現れてもよく、その前には空白(スペースとタブ)のみ置いてよい。センチネル(**!$acc**)は間に空白のない単語として現れなければならない。行の長さ、空白、継続ルールは指示文行にも適用される。最初の指示文行はセンチネルの後にスペースがなければならない。継続した指示文行は行の最後のブランク文字としてアンパサンド(**&**)がなければならず、それは指示文中のどのコメントよりも優先される。継続指示文行はセンチネルから始まらなければならず(空白が前にあるかもしれない)、センチネルの後の最初の空白でない文字としてアンパサンドがあるべきだ。感嘆符から始まり、行の最後を延ばすコメントは指示文と同じ行に現れてもよい。もしセンチネルより後の最初の非ブランク文字が感嘆符なら、この行は無視される。 Fortran の固定形式ソースファイルでは、OpenACC 指示文は次のいずれかで記述される。

!$acc *directive-name [clause [[,] clause] ...]*
c$acc *directive-name [clause [[,] clause] ...]*
***$acc** *directive-name [clause [[,] clause] ...]*

センチネル(**!$acc**, **c$acc**, ***$acc**)は 1~5 列で使用されなければならない。固定形式の行の長さ、空白、継続、列規則は指示文行にも適用される。最初の指示文行は 6 列目にスペースかゼロがなければならなず、継続指示文行は 6 列目にスペースかゼロ以外がなければならない。7 列目かその後に感嘆符で始まり、行の最後まで続くコメントは指示文と同じ行に現れてもよい。

Fortran では指示文は大文字小文字を区別しない。指示文は継続文の中に埋め込むことはできず、文は継続した指示文の中に埋め込んではならない。このドキュメントでは、すべての Fortran の OpenACC 指示文例に自由形式が用いられる。

指示文毎にただ 1 つの *directive-name* が記述できる。指示節の現れる順序に意味はなく、指示節は他で記述されない限り、繰り返されても良い。いくつかの指示節は*リスト*引数をもつ。*リスト*はコンマで区切られた変数名、配列名、ときには添字の範囲の部分配列のリストである。

2.2 条件付きコンパイル

_OPENACC マクロ名は実装がサポートする OpenACC のバージョンの年を yyyy、月を mm として yyyymm という値を持つと定義される。このマクロは OpenACC 指示文が有効なときのみコンパイラによって定義される必要がある。ここで記述されるバージョンは 201111 である。

2.3 内部制御変数

OpenACC 実装はプログラムの動作を制御する内部制御変数(ICVs)があるかのようにふるまう。これらの ICVs は実装により初期化され、環境変数や OpenACC API ルーチンの呼び出しを通して値を与えてよい。プログラムは OpenACC API ルーチンの呼び出しを通して値の検索が出来る。 ICVs は

- *acc-device-type-var* - どのタイプのアクセラレータデバイスが使用されるか制御する。
- *acc-device-num-var* - 選択されたタイプのアクセラレータデバイスのどれが使用されるか制御する。

2.3.1 ICV 値の変更と取得

下の表は内部制御変数の値を変更するための環境変数またはプロシージャと、値を取得するためのプロシージャを示している。

ICV	値を変更する手段	値を取得する手段
acc-device-type-var	ACC_DEVICE_TYPE acc_set_device_type	acc_get_device_type
acc-device-num-var	ACC_DEVICE_NUM acc_set_device_num	acc_get_device_num

初期値は実装で定義されている。初期値が与えられた後、OpenACC 構文や API ルーチンが実行される前に、ユーザが設定した環境変数の値が読まれ関連した ICVs はそれに応じて変更される。この時点の後、プログラムや外部による環境変数の変更は ICVs に影響しない。
OpenACC 構文上の指示節は ICV 値を変更しない。

2.4 アクセラレータ計算構文

2.4.1 Parallel 構文

概要

この基本的な構文はアクセラレータデバイス上で並列実行を開始する。

文法

C,C++での OpenACC parallel 指示文の文法は

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

structured block

Fortran での文法は

!$acc parallel *[clause [[,] clause] ...]*

structured block

!$acc end parallel

clause の部分は以下の 1 つである

- **if(** *condition* **)**
- **async** [**(** *scalar-integer-expression* **)**]
- **num_gangs(** *scalar-integer-expression* **)**
- **num_workers(** *scalar-integer-expression* **)**
- **vector_length(** *scalar-integer-expression* **)**
- **reduction(** *operator : list* **)**
- **copy(** *list* **)**
- **copyin(** *list* **)**
- **copyout(** *list* **)**
- **create(** *list* **)**
- **present(** *list* **)**
- **present_or_copy(** *list* **)**
- **present_or_copyin(** *list* **)**
- **present_or_copyout(** *list* **)**
- **present_or_create(** *list* **)**
- **deviceptr(** *list* **)**
- **private(** *list* **)**
- **firstprivate(** *list* **)**

説明

プログラムがアクセラレータ **parallel** 構文に出会うとき、アクセラレータ parallel 領域を実行するための複数の worker の複数の gang が生成される。一度 gang が生成されると、gang 数と各 gang 内の worker 数は parallel 領域の間で一定のままである。各 gang の 1 つの worker が構文の構造化ブロック内のコードの実行を開始する。

もし **async** 節が存在しないなら、アクセラレータ parallel 領域の終わりに暗黙のバリアがあり、ホストプログラムはすべての gang が実行を完了するまで待機する。**parallel** 構文の中で参照される配列や集合データ形式の変数で、その構文や囲んでいる **data** 構文のデータ指示節で現れないものは、あたかも **parallel** 構文の **present_or_copy** 節の中で現れるかのように扱われる。
parallel 構文の中で参照されるスカラー変数で、その構文や囲んでいる **data** 構文のデータ指示節の中で現れないものは、あたかも **parallel** 構文の **private** 節(もし live-in や live-out でないなら)もしくは **copy** 節で現れるかのように扱われる。

制約

- OpenACC parallel 領域はほかの parallel 領域や kernels 領域を含んではならない。
- プログラムは OpenACC**parallel** 構文の内側、または外側に分岐してはならない。
- プログラムは指示節の評価の順序や評価のいかなる副作用に依存してはならない。
- 高々1 つの **if** 節が現れてもよい。Fortran では条件はスカラー論理値を評価しなければならない。C,C++では条件はスカラー整数値を評価しなければならない。

copy, copyin, copyout, create, present, present_or_copy, present_or_copyin, present_or_copyout, present_or_create, deviceptr, firstprivate, private データ指示節はセクション 2.7 で説明される。

2.4.2 Kernels 構文

概要

この構文はアクセラレータデバイス上での実行のために一連のカーネルにコンパイルされるプログラムの領域を定める。

文法

C と C++での OpenACC kernels 指示文の文法は

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

structured block

Fortran での文法は

!$acc kernels *[clause [[,] clause] ...]*

structured block

!$acc end kernels

clause の部分は以下の 1 つである

- **if(** *condition* **)**
- **async** [**(** *scalar-integer-expression* **)**]
- **copy(** *list* **)**
- **copyin(** *list* **)**
- **copyout(** *list* **)**
- **create(** *list* **)**
- **present(** *list* **)**
- **present_or_copy(** *list* **)**
- **present_or_copyin(** *list* **)**
- **present_or_copyout(** *list* **)**
- **present_or_create(** *list* **)**
- **deviceptr(** *list* **)**

説明

コンパイラは kernels 領域のコードを一連のアクセラレータカーネルに分解する。一般的に、各ループネストは全く異なるカーネルになる。プログラムが **kernels** 構文に出会うとき、デバイス上で順に一連のカーネルを開始する。worker の gang の数と設定、そしてベクトル長はカーネルごとに異なってよい。

もし **async** 節が存在しないなら、kernels 領域の最後に暗黙のバリアがありホストプログラムはすべてのカーネルが実行を完了するまで待機する。

kernels 構文の中で参照される配列か集合データ形式の変数で、その構文や囲まれた **data** 構文のデータ指示節で現れないものは、あたかも **kernels** 構文の **present_or_copy** 節で現れるかのように扱われる。**kernels** 構文で参照されるスカラーで、その構文や囲まれた **data** 構造のデータ指示節で現れないものは、あたかも **kernels** 構文の **private** 節(もし live-in や live-out でないなら)または **copy** 節で現れるかのように扱われる。

制約

- OpenACC kernels 領域は他の parallel 領域や kernels 領域を含んではならない。
- プログラムは OpenACC**kernels** 構文の内側または外側に分岐してはならない。
- プログラムは節の評価の順序やいかなる評価の副作用に依存してはならない。
- 高々1 つの **if** 節が現れてもよい。Fortran では条件はスカラー論理値を評価しなければならず、C や C++では条件はスカラー整数値を評価しなければならない。

copy, copyin, copyout, create, present, present_or_copy, present_or_copyin, present_or_copyout, present_or_create, deviceptr データ指示節はセクション 2.7 で説明される。

2.4.3 if 節

if 節は **parallel** や **kernels** 構文上で任意である。すなわち、もし **if** 節がなければ、コンパイラはアクセラレータデバイス上でその領域を実行するためのコードを生成する。 **if** 節が現れるときは、コンパイラはその領域の 2 つのコピーを生成する。1 つはアクセラレータで実行するための、もう 1 つはホストで実行するためのものである。**if** 節の **condition** の評価が C,C++でゼロ、Fortran で**.false.**のときはホストコピーが実行される。**condition** の評価が C,C++で非ゼロ、Fortran で**.true.**のときはアクセラレータコピーが実行される。

2.4.4 async 節

async 節は **parallel** や **kernels** 構文上で任意である。すなわち、もし **async** 節がなければ、ホストプロセスはその領域に続くいかなるコードの実行の前に、その並列またはカーネル領域が完了するまで待機する。**async** 節があるときは、並列またはカーネル領域はアクセラレータデバイスにより非同期的に実行され、同時にホストプロセスはその領域に続くコードを続ける。

もし存在するなら、**async** 節の引数は整数式でなければならない(C,C++では **int**, Fortran では **integer**)。ホストプロセスがその領域の完了のテストや待機をするために同じ整数式値が **wait** 指示文や様々なランタイムルーチンで使われてもよい。**async** 節は引数なしで使われてもよく、その場合、実装はプログラム中のすべての明示的な **async** 引数と異なる値を使う。

同じ引数値の 2 つの非同期処理はホストプロセスがそれらに出会う順でデバイス上で実行される。異なるハンドル値の 2 つの非同期処理は互いに相対的に任意の順序でデバイス上で実行さ

れる。もし 2 つ以上のホストスレッドが実行していて同じアクセラレータデバイスを共有しているなら、相対的な順序は決定されないが、同じ引数値の 2 つの非同期処理は順に実行される。

2.4.5 num_gangs 節

num_gangs 節は **parallel** 構文で許されている。その整数式の値はその領域で実行する並列ギャングの数を定める。もしその指示節が記されていないなら、実装で定義されたデフォルト値が使われる。

2.4.6 num_workers 節

num_workers 節は **parallel** 構文で許されている。その整数式の値はその領域を実行する各ギャング内のワーカー数を定める。もしその指示節が記されていないなら、実装で定義されたデフォルト値が使われる。そのデフォルト値は 1 だろう。

2.4.7 vector_length 節

vector_length 節は **parallel** 構文で許されている。その整数式の値はギャングの各ワーカー内のベクトルまたは SIMD 演算のために使うベクトル長を定める。もしその指示節が記されていないなら、実装で定義されたデフォルト値が使われる。このベクトル長は **loop** 指示文上の **vector** 節で注釈の付いたループや、コンパイラにより自動でベクトル化されたループのために使われる。ベクトル長の式の許容値には実装で定義された制限があるだろう。

2.4.8 private 節

private 節は **parallel** 構文で許されていて、リスト上の各項目のコピーが各並列ギャングに作られることを宣言する。

2.4.9 firstprivate 節

firstprivate 節は **parallel** 構文で許されていて、リスト上の各項目のコピーが各並列ギャングに作られ、そのコピーは **parallel** 構文に出会った時のホスト上の項目の値で初期化されることを宣言する。

2.4.10 reduction 節

reduction 節は **parallel** 構文で許されている。リダクション演算子と 1 つ以上のスカラー変数を指定する。各変数のために各並列ギャングにプライベートコピーが作られ、その演算子のために初期化される。その領域の最後で、各ギャングの値はリダクション演算子を使って結合され、その結果は元の変数の値と結合されて元の変数に格納される。リダクション結果はその領域の後で使用可能である。

以下の表は有効な演算子と初期値の一覧である。各場合で初期値は変数の型にキャストされる。**max** と **min** リダクションのための初期値はそれぞれ変数のデータ型の表現可能な最小値と表現可能な最大値である。サポートされているデータ型は数値データ型で C・C++では(int, float, double, complex)、Fortran では(integer, real, double, precision, complex)である。

C・C++	Fortran

演算子	初期値	演算子	初期値
+	0	+	0
*	1	*	1
max	最小値	max	最小値
min	最大値	min	最大値
&	~0	iand	全ビット ON
|	0	ior	0
^	0	ieor	0
&&	1	.and.	.true.
||	0	.or.	.false.
		.eqv.	.true.
		.neqv.	.false.

2.5 データ構文

概要

data 構文はその領域間のためにデバイスメモリで確保されるスカラー・配列・部分配列が、領域の入口でホストからデバイスメモリにコピーされるか、領域の出口でデバイスからホストメモリにコピーされるかを定義する。

文法

C,C++での OpenACC データ指示文の文法は

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

structured block

Fortran での文法は

!$acc data *[clause [[,] clause] ...]*

structured block

!$acc end data

clause の部分は以下の 1 つである

- **if(** *condition* **)**
- **copy(** *list* **)**
- **copyin(** *list* **)**
- **copyout(** *list* **)**
- **create(** *list* **)**
- **present(** *list* **)**
- **present_or_copy(** *list* **)**
- **present_or_copyin(** *list* **)**

- **present_or_copyout(** *list* **)**
- **present_or_create(** *list* **)**
- **deviceptr(** *list* **)**

説明

データはデバイスメモリに確保され、必要に応じてホストメモリからデバイスへコピーまたはコピーバックされる。データ指示節はセクション 2.7 で説明される。

2.5.1 if 節

if 節は任意で、**if** 節がなければコンパイラはアクセラレータデバイス上のメモリを確保し、ホストから、またはホストへデータを移動するコードを生成する。**if** 節が現れるときは、プログラムは条件的にメモリを確保し、デバイスへ、またはデバイスからデータを移動する。**if** 節の *condition* が C・C++でゼロ、Fortran で**.false.**のとき、デバイスメモリは確保されずデータは移動されない。*condition* が C・C++で非ゼロ、Fortran で**.true.**のとき、データは確保され指定されたように移動される。

2.6 Host_Data 構文

概要

host_data 構文はデバイスデータのアドレスをホスト上で利用可能にする。 **文法**

C,C++での OpenACC データ指示文の文法は

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

structured block

Fortran での文法は

!$acc host_data *[clause [[,] clause] ...]*

structured block

!$acc end host_data

clause の部分は 1 つだけ有効で

- **use_device(** *list* **)**

説明 この構文はデータのデバイスアドレスをホストコード上で利用可能にするために使用される。

2.6.1 use_device 節

use_device はその構文内のコードで、*list* 内の変数や配列のデバイスアドレスを使うようにコンパイラに伝える。特に、変数や配列のデバイスアドレスを低レベル API で書かれた最適化され

たプロシージャに渡すために使われるだろう。この構文を含むデータ領域のために、*list* 内の変数や配列はアクセラレータメモリに存在している必要がある。

2.7 データ節

これらのデータ節は **parallel** 構文、**kernels** 構文、**data** 構文で現れてもよい。各データ節へのリスト引数は変数名・配列名・部分配列指定のカンマ区切りコレクションである。すべての場合で、コンパイラは変数や配列の可視デバイスコピーを作成し、デバイスメモリ内の変数や配列のコピーの確保と管理をする。

ホストから物理的または論理的に独立したメモリを持つアクセラレータをサポートする目的がある。 しかしながら、もしアクセラレータが直接ホストメモリにアクセスできるなら、実装はメモリ確保とデータ移動を避けて単純にホストメモリを使用してもよい。したがって、1 つのデータ領域内でホスト上のデータの使用や割り当てをし、アクセラレータ上の同じデータの使用や割り当てをするプログラムで、2 つのコピーの一貫性を管理するための update 指示文がないものは、異なるアクセラレータや実装上で異なる答えを得るかもしれない。

C,C++では、部分配列はブラケット内の開始位置と長さによる拡張配列範囲指定が後ろについた次のような配列名である。

arr[2:n]

もし下限がなければ、ゼロが使われる。もし長さがなく配列のサイズが分かっていれば、下限と宣言された配列のサイズの差が使われる。そうでなければ長さが必要である。部分配列 **arr[2:n]**は要素 **a[2], a[3], ..., a[2+n-1]**を意味する。

Fortran では、部分配列は丸括弧内の下限と上限の添字によるカンマ区切り範囲指定リストが後ろについた次のような配列名である。

arr(1:high, low:100)

もし下限か上限のどちらかがなければ、宣言または確保された配列の境界が分かれるなら使われる。

制約

- Fortran では、大きさ引継ぎ配列の最終次元のための上限が指定される必要がある。
- C,C++では、動的に確保された配列の長さが明示的に指定される必要がある。
- もし部分配列がデータ節で指定されるなら、コンパイラはアクセラレータ上の部分配列のみに対してメモリを確保してもよい。
- コンパイラはメモリアライメントやプログラム性能の改善のためにアクセラレータ上の配列の次元に詰め物をしてもよい。
- Fortran では、配列ポインタが指定されてもよいが、ポインタ結合はデバイスメモリでは保存されない。
- Fortran 配列ポインタを含むデータ節内の配列や部分配列は、メモリの隣接ブロックでなければならない。
- C,C++では、もし構造体やクラスの変数や配列が指定されているなら、構造体やクラスのすべてのデータメンバは適切に確保やコピーされる。もし構造体やクラスのメンバがポインタ型ならば、そのポインタがさすデータは暗黙にコピーされない。

- C,C++では、構造体やクラスのメンバがメンバの部分配列を含めて指定されてもよい。しかしながら、構造体変数自身はポインタを通してアクセスされない自動的・静的・大域でなければならない。構造体やクラス型の部分配列のメンバは指定されてはならない。
- Fortran では、もし派生型の変数や配列が指定されているならば、派生型のすべてのメンバは適切に確保やコピーされる。もしメンバが **allocatable** か **pointer** 属性ならば、そのメンバを通してアクセスされるデータはコピーされない。
- Fortran では、メンバの部分配列を含め、派生型の変数のメンバが指定されてもよい。しかしながら、派生型の変数は **allocatable** や **pointer** 属性を持ってはならない。派生型の部分配列のメンバは指定されなくてもよい。

2.7.1 deviceptr 節

deviceptr 節は *list* 内のポインタがデバイスポインタであることを宣言するために使われる。そのため、このポインタのためにデータが確保されたり、ホストとデバイス間で移動されたりする必要がない。 C,C++では、*list* 内の変数はポインタでなければならない。 Fortran では、*list* 内の変数は仮引数(配列やスカラ)でなければならず、Fortran の **pointer**, **allocatable**, **value** 属性を持たなくてもよい。

2.7.2 copy 節

copy 節は *list* 内の変数・配列・部分配列がデバイスメモリにコピーする必要のあるホストメモリ内の値を持ち、ホストにコピーバックする必要のある値がアクセラレータ上で割り当てられることを宣言するために使われる。もし部分配列が指定されるなら、配列の部分配列の所だけコピーする必要がある。データはその領域に入る前にデバイスメモリにコピーされ、領域が完了したときにホストメモリにコピーバックされる。

2.7.3 copyin 節

copyin 節は *list* 内の変数・配列・部分配列がデバイスメモリにコピーする必要のあるホストメモリ内の値を持つことを宣言するために使われる。もし部分配列が指定されるなら、配列の部分配列の所だけコピーする必要がある。もし変数・配列・部分配列が **copyin** で現れたら、その節はたとえ値がアクセラレータ上で変化しても、データをデバイスメモリからホストメモリへコピーバックする必要がないことを意味する。データはその領域に入る前にデバイスメモリにコピーされる。

2.7.4 copyout 節

copyout 節は *list* 内の変数・配列・部分配列がデバイスメモリで割り当てられるか含まれる値で、アクセラレータ領域の終わりにホストメモリにコピーバックする必要があることを宣言するために使われる。もし部分配列が指定されるなら、配列の部分配列の所だけコピーする必要がある。もし変数・配列・部分配列が **copyout** で現れたら、その節はたとえ値がアクセラレータ上で使われるとしても、データをホストメモリからデバイスメモリへコピーする必要がないことを意味する。データはその領域の出口でホストメモリにコピーバックされる。

2.7.5 create 節

create 節は *list* 内の変数・配列・部分配列がデバイスメモリで確保される（作られる）必要があるが、ホストメモリの値はアクセラレータ上で必要なく、アクセラレータ上で計算、割り振

られた値はホストで必要ないことを宣言するために使われる。この節にないデータはホストとデバイスメモリ間でコピーされる。

2.7.6 present 節

present 節は *list* 内の変数や配列がアクセラレータメモリ上に既にあることを実装に伝えるために使われる。おそらくこの構文を含むプロシージャを呼び出すプロシージャからのこの領域を含むデータ領域によって。実装は存在するアクセラレータデータを見つけて使う。もしアクセラレータ上に置かれた変数や配列を含むデータ領域がなければ、プログラムはエラーとともに終了する。

もし含むデータ領域が部分配列を指定するなら、**present** 節はデータ領域で同じ部分配列か、部分配列の適切な部分集合である部分配列でなければならない。もし **present** 節内の部分配列がデータ領域で指定されている部分配列の一部である配列要素を含まないなら、ランタイムエラーとなる。

2.7.7 present_or_copy 節

present_or_copy 節は *list* 内の変数や配列がアクセラレータメモリにすでにあるかどうかテストするよう実装に伝えるために使われる。もしすでに存在するなら、アクセラレータのデータを使用する。もし存在しないなら、**copy** 節のようにアクセラレータメモリを確保し、領域の開始時にホストからアクセラレータへコピーし、領域の終了時にホストへコピーして返す。この節は **pcopy** と略してよい。**present** 節の部分配列に対するのと同じ制約をこの節にも適用する。

2.7.8 present_or_copyin 節

present_or_copyin 節は *list* 内の変数や配列がアクセラレータメモリにすでにあるかどうかテストするよう実装に伝えるために使われる。もしすでに存在するなら、アクセラレータのデータを使用する。もし存在しないなら、**copyin** 節のようにアクセラレータメモリを確保し、領域の開始時にホストからアクセラレータへコピーする。この節は **pcopyin** と略してよい。**present** 節の部分配列に対するのと同じ制約をこの節にも適用する。

2.7.9 present_or_copyout 節

present_or_copyout 節は *list* 内の変数や配列がアクセラレータメモリにすでにあるかどうかテストするよう実装に伝えるために使われる。もしすでに存在するなら、アクセラレータのデータを使用する。もし存在しないなら、**copyout** 節のようにアクセラレータメモリを確保し、領域の終了時にホストへコピーして返す。この節は **pcopyout** と略してよい。**present** 節の部分配列に対するのと同じ制約をこの節にも適用する。

2.7.10 present_or_create 節

present_or_create 節は *list* 内の変数や配列がアクセラレータメモリにすでにあるかどうかテストするよう実装に伝えるために使われる。もしすでに存在するなら、アクセラレータのデータを使用する。もし存在しないなら、**create** 節のようにアクセラレータメモリを確保する。この節は **pcreate** と略してよい。**present** 節の部分配列に対するのと同じ制約をこの節にも適用する。

2.8 Loop 構文

概要

OpenACC**loop** 指示文はこの指示文の直後のループに適用する。**loop** 指示文はループの実行に使う並列処理の種類を記述でき、ループのプライベート変数や配列やリダクション演算を宣言できる。

文法

C,C++での **loop** 指示文の文法は

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

for loop

Fortran での **loop** 指示文の文法は

!$acc loop *[clause [[,] clause] ...]*

do loop

clause の部分は以下の 1 つである。

- **collapse(** *n* **)**
- **gang** *[* **(** *scalar-integer-expression* **)** *]*
- **worker** *[* **(** *scalar-integer-expression* **)** *]*
- **vector** *[* **(** *scalar-integer-expression* **)** *]*
- **seq**
- **independent**
- **private(** *list* **)**
- **reduction(** *operator : list* **)**

いくつかの指示節は parallel 領域の文脈でのみ、いくつかは kernels 領域の文脈でのみ有効である。下の説明を見るように。 parallel 領域では、**gang**, **worker**, **vector** 節のない **loop** 指示文によって、gang または gang 内の worker でループを実行するか、ベクトル演算として実行するかを実装が自動的に選択できる。実装は古典的な自動ベクトル化を用いて **loop** 指示文のないループを実行するのにベクトル演算を使用することを選択してもよい。

2.8.1 collapse 節

collapse 節はどれだけのしっかりとネストしたループを **loop** 構文と関連付けるかを指定するために使う。**collapse** 節の引数は定数の正の整数式でなければならない。もし **collapse** 節が存在しないなら、直後のループだけをループ指示文と関連させる。

もし 1 つ以上のループが **loop** 構文に関連付けられるなら、すべての関連したループの反復はその残りの節に従ってスケジュールされる。**collapse** 節で関連付けられたすべてのループのトリップカウントはすべてのループで計算可能で、不変でなければならない。

指示文上の **gang**, **worker**, **vector** 節が各ループに適用されるか、線形化された反復空間に適用されるかは実装で定義される。

2.8.2 gang 節

アクセラレータ parallel 領域では、**gang** 節は parallel 構文によって生成された gang 間に反復を分散することで、対応する 1 つのループまたは複数のループの反復を並列に実行するよう指定する。引数は許されていない。ループ反復は **reduction** 節で指定される変数を除き、データ非依存でなければならない。

アクセラレータ kernels 領域では、**gang** 節は対応する 1 つのループまたは複数のループの反復を、その 1 つのループまたは複数のループに含まれた任意のカーネル用に生成された gang 間で並列に実行するよう指定する。もし引数が指定されるなら、それはこのループの反復を実行するのに使う gang 数を指定する。

2.8.3 worker 節

アクセラレータ parallel 領域では、**worker** 節は単一 gang 内の複数 worker 間で反復を分散することで、対応した 1 つのループまたは複数のループの反復を並列に実行するよう指定する。引数は許されていない。ループ反復は **reduction** 節で指定される変数を除き、データ非依存でなければならない。**worker** 節のあるループが **gang** 節を含むループを含めてよいかどうかは実装で定義される。

アクセラレータ kernels 領域では、**worker** 節は対応した 1 つのループまたは複数のループの反復を、その 1 つのループまたは複数のループに含まれた任意のカーネル用に生成された複数の gang 内の worker 間で並列に実行するよう指定する。もし引数が指定されるなら、それはこのループの反復を実行するのに使う worker 数を指定する。

2.8.4 seq 節

seq 節は対応した 1 つのループまたは複数のループをアクセラレータにより逐次的に実行するよう指定する。これはアクセラレータ **parallel** 領域でデフォルトである。この節はいかなる自動コンパイラ並列化やベクトル化も上書きする。

2.8.5 vector 節

アクセラレータ parallel 領域では、**vector** 節は対応する 1 つのループまたは複数のループをベクトルまたは SIMD モードで実行するよう指定する。演算はその parallel 領域で指定または選択された長さのベクトルを使って実行する。**vector** 節のあるループが **gang** または **worker** 節を含むループを含めてよいかどうかは実装で定義される。

アクセラレータ kernels 領域では、**vector** 節は対応する 1 つのループまたは複数のループをベクトルか SIMD 処理で実行するよう指定する。もし引数が指定されるなら、反復はその長さのベクトルストリップで処理される。もし引数が指定されないなら、コンパイラが適切なベクトル長を選ぶ。

2.8.6 independent 節

independent 節は kernels 領域の **loop** 指示文で許されており、このループの反復は互いに関してデータ非依存であることをコンパイラに伝える。これによりコンパイラは非同期で反復を並列に実行するコードを生成できる。

制約

- もしある反復がリダクション節内の変数を除き、他の反復も同様に読み書きする変数や配列要素に書き込むならば、**independent** 節を使うのはプログラミングエラーである。

2.8.7 private 節

ループ指示文上の **private** 節はリスト内の各項目のコピーが、対応する 1 つのループまたは複数のループの各反復のために生成されるよう指定する。

2.8.8 reduction 節

reduction 節は **gang**, **worker**,**vector** 節のある loop 構文で許される。リダクション演算子と 1 つ以上のスカラー変数を指定する。各リダクション変数のために、対応した 1 つのループまたは複数のループの各反復にプライベートコピーが生成され、その演算子用に初期化される(セクション 2.4.10 の表を参照)。ループの最後で、指定されたリダクション演算子で各反復の値を結合し、その結果は parallel または kernels 領域の最後に元の変数に格納される。

parallel 領域では、もし **reduction** 節が **vector** または **worker** 節のある(**gang** 節のない)ループで使われ、スカラー変数が同様にその **parallel** 構文上の **private** 節で現れるなら、そのスカラーのプライベートコピーの値はループの出口で更新される。そうでなければ、parallel 領域内のループ上の **reduction** 節で現れる変数は領域の終わりまで更新されない。

2.9 キャッシュ指示文

概要

キャッシュ指示文はループの一番上(内側の）で現れてよい。そのループの本体のための最高レベルのキャッシュにフェッチするべき配列要素や部分配列を指定する。

文法

C,C++での cache 指示文の文法は

#pragma acc cache(*list* **)** *new-line*

Fortran での cache 指示文の文法は

!$acc cache (*list* **)**

list の項目は単配列要素か単部分配列でなければならない。C,C++では、単部分配列はブラケット内の開始と長さによる拡張配列範囲指定が後ろに付いた配列名で、t 次のようなものである

arr[*lower* **:** *length* **]**

下限は定数、ループ不変の変数、定数またはループ不変の変数を加算または減算する for ループ添字変数であり、長さは定数である。

Fortran では、単部分配列は丸括弧内の上下限添字による範囲指定のカンマ区切りリストが後ろに付いた配列名で、次のようなものである

arr(*lower*:*upper*, *lower2*:*upper2* **)**

下限は定数、ループ不変の変数、定数またはループ不変の変数を加算または減算する do ループ添字変数であり、さらに対応する上限と下限の差は一定でなければならない。

2.10 結合指示文

概要

結合した OpenACC の **parallel loop** や **kernels loop** 指示文は **parallel** や **kernels** 構文のすぐ内側にネストした **loop** 指示文を指定するためのショートカットである。意味は **loop** 指示文を含む **parallel** や **kernels** 指示文を明確に指定することに等しい。**parallel** または **loop** 指示文で使える節は **parallel loop** 指示文でも使え、**kernels** または **loop** 指示文で使える節は **kernels loop** 指示文でも使える。

文法

C,C++での **parallel loop** 指示文の文法は

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

for loop

Fortran での **parallel loop** 指示文の文法は

!$acc parallel loop *[clause [[,] clause] ...]*

do loop

[**!$acc end parallel loop** *]*

対応する構造化ブロックはこの指示文の直後のループである。parallel 領域で有効な **parallel** または **loop** 節が現れてもよい。

C,C++での **kernels loop** 指示文の文法は

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

for loop

Fortran での **kernels loop** 指示文の文法は

!$acc kernels loop *[clause [[,] clause] ...]*

do loop

[**!$acc end kernels loop** *]*

対応する構造化ブロックはこの指示文の直後のループである。kernels 領域で有効な **kernels** または **loop** 節が現れてもよい。

制約

- この結合指示文は他のアクセラレータの parallel または kernels 領域の本体内で現れてはならない。
- **parallel**, **kernels**, **loop** 構文に対する制約を適用する。

2.11 宣言指示文

declare 指示文は Fortran のサブルーチン・関数・モジュールの宣言部、または C,C++の以下の変数宣言で使う。 関数・サブルーチン・プログラムの暗黙的なデータ領域間のためにデバイスメモリで確保する変数や配列を指定したり、暗黙のデータ領域に入るときにホストからデバイスメモリに、出るときにデバイスからホストメモリにデータ値を転送するかどうかを指定したりできる。これらの指示文は変数や配列の可視デバイスコピーを作る。

文法

C,C++での **declare** 指示文の文法は

#pragma acc declare *declclause [[,] declclause]... new-line*

Fortran での **declare** 指示文の文法は

!$acc declare *declclause [[,] declclause]...*

declclause の部分は以下の 1 つである。

- **copy(** *list* **)**
- **copyin(** *list* **)**
- **copyout(** *list* **)**
- **create(** *list* **)**
- **present(** *list* **)**
- **present_or_copy(** *list* **)**
- **present_or_copyin(** *list* **)**
- **present_or_copyout(** *list* **)**
- **present_or_create(** *list* **)**
- **deviceptr(** *list* **)**
- **device_resident(** *list* **)**

対応する領域は指示文が現れるところにある関数・サブルーチン・プログラムに関連する暗黙の領域である。もし指示文が Fortran の MODULE サブプログラムで現れるなら、対応する領域はプログラム全体のための暗黙の領域である。そうでなければ、節はそれらの節があるプロシージャ本体を囲む明確な data 構文と全く同じ動作をする。 データ節はセクション 2.7 で説明する。

制約

- １つの変数や配列は関数・サブルーチン・プログラム・モジュールの **declare** 指示文のすべての節で高々1 回現れてよい。
- 部分配列は **declare** 指示文で使えない。
- もし変数や配列が **declare** 指示文で現れたなら、同じ変数や配列は変数宣言が見えるいかなる構文の data 節で現れてはならない。
- Fortran では、擬寸法仮配列は **declare** 指示文で現れてはならない。
- コンパイラはメモリアライメントやプログラム性能を改善するためにアクセラレータ上の配列の次元に詰めものをしてもよい。
- Fortran では、ポインタ配列を指定してもよいが、ポインタ結合はデバイスメモリで保存されない。

2.11.1 device_resident 節

device_resident はホストメモリでなくアクセラレータデバイスメモリ内に確保されるべき名前付き変数用のメモリを指定する。 C,C++では、これはホストがこれらの変数にアクセスできないことを意味する。**list** 内の変数は関数への静的ファイルかローカルでなければならない。
Fortran では、もし変数が Fortran の **allocatable** 属性を持つなら、ホストプログラムがその変数に **allocate** や **deallocate** 文を実行するときにその変数のメモリをアクセラレータデバイスメモリに確保したり、解放したりする。もし変数が Fortran の **pointer** 属性を持つなら、アクセラレータデバイスメモリに確保か解放するか、もし右辺の変数自身が **device_resident** 節に現れるならポインタ割当文の左辺に現れてもよい。もし変数が allocatable や pointer 属性のどちらも持たないなら、サブプログラムのローカルでなければならない。

2.12 実行可能指示文

2.12.1 update 指示文

概要

update 指示文は明示的、暗黙的 data 領域内のホストメモリ配列のすべてまたは一部をデバイスメモリ内の対応する配列からの値で更新するため、またはデバイスメモリ配列のすべてまたは一部をホストメモリ内の対応する配列からの値で更新するために使う。

文法 C,C++での **update** 指示文の文法は

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

Fortran での **update** データ指示文の文法は

!$acc update *clause [[,] clause]...*

clause の部分は以下の 1 つである。

- **host(** *list* **)**
- **device(** *list* **)**
- **if(** *condition* **)**
- **async** [**(** *scalar-integer-expression* **)**]

update 節のリスト引数は変数名・配列名・部分配列指定のカンマ区切りコレクションである。同じ配列の複数の部分配列がリストに現れてもよい。**update** 節の作用は、**update host** でアクセラレータデバイスメモリからホストメモリへ、**update device** でホストメモリからアクセラレータデバイスメモリへデータをコピーすることだ。更新は指示文で現れる順に行われる。**host** や **device** 節で現れる変数や配列の可視デバイスコピーがなければならない。少なくとも 1 つの **host** か **device** 節が現れなければならない。

2.12.1.1 host 節

host 節はリスト内の変数・配列・部分配列をアクセラレータデバイスメモリからホストメモリへコピーするよう指定する。

2.12.1.2 device 節

device 節はリスト内の変数・配列・部分配列をホストメモリからアクセラレータデバイスメモリへコピーするよう指定する。

2.12.1.3 if 節

if 節は任意である。つまり **if** 節がなければ、コンパイラは無条件に更新を実行するコードを生成する。**if** 節が現れるときは、コンパイラは *condition* が C,C++で非ゼロ、Fortran で**.true.**を評価する時のみ更新を実行する条件付きのコードを生成する。

2.12.1.4 async 節

async 節は任意である。つまり **async** 節がなければ、ホストプロセスは更新が完了するまで **update** 指示文の後のコードを実行しない。**async** 節があるときは、更新を非同期的に実行し、その間ホストプロセスは指示文の後のコードの実行を続ける。

もし **async** 節に引数があれば、引数は整数変数(C,C++なら int, Fortran なら integer)の名前でなければならない。その変数を **wait** 指示文や様々なランタイムルーチンでホストプロセスが更新の完了をテストしたり待ったりするために使ってもよい。**async** 節は引数なしで使ってもよく、その場合は実装がプログラム内の明示的な **async** 節の引数すべてと異なる値を使う。

2 つの同じ引数値をもつ非同期アクティビティはホストプロセスで出会った順にデバイスで実行される。2 つの異なるハンドル値を持つ非同期アクティビティは互いに相対的に任意の順でデバイスで実行される。もし 2 つ以上の実行中で同じアクセラレータデバイスを共有するホストスレッドがあるなら、2 つの同じ引数値をもつ非同期アクティビティは相対的な順序が決まっていないにも関わらずデバイスで順番に実行される。

制約

- **update** 指示文は実行可能である。C,C++では *if, while, do, switch, label* に続く文の場所、Fortran では論理 *if* に続く文の場所に現れてはならない。
- **update** 指示文のリストに現れる変数や配列は可視デバイスコピーを持たなければならない。

2.12.2 wait 指示文

概要

wait 指示文はアクセラレータ parallel または kernels 領域や **update** 指示文のような非同期アクティビティの完了をプログラムが待つようにする。

文法

C,C++での **wait** 指示文の文法は

#pragma acc wait *[* **(** *scalar-integer-expression* **)** *] new-line*

Fortran での **wait** 指示文の文法は

!$acc wait *[* **(** *scalar-integer-expression* **)** *]*

引数が指定されるなら、引数は整数式(C,C++なら int, Fortran なら integer)でなければならない。ホストスレッドは同じ値の引数の **async** 節をもつすべての非同期アクティビティが完了するまで待つ。

引数が指定されないなら、ホストプロセスはすべての非同期アクティビティが完了するまで待つ。

もし 2 つ以上の実行中で同じアクセラレータデバイスを共有するホストスレッドがあるなら、wait 指示文はホストスレッドが少なくともそのホストスレッドが開始した非同期アクティビティのすべてが完了するまで待つようにする。他のホストスレッドが開始した同様の非同期アクティビティのすべてが完了している保証はない。

3. ランタイムライブラリルーチン

この章はプログラマが使用できる OpenACC ランタイムライブラリルーチンを記述する。これらのルーチンの使用により OpenACC API をサポートしていないシステムへの移植が制限される。**_OPENACC** プリプロセッサ変数を用いた条件付きコンパイルにより移植性を保てるだろう。 この章は 2 つのセクションがある。

- ランタイムライブラリ定義
- ランタイムライブラリルーチン

制約

- Fortran では、**PURE** や **ELEMENTAL** プロシージャから呼ばれる OpenACC ランタイムライブラリルーチンはないだろう。

3.1 ランタイムライブラリ定義

C,C++では、この章で記述するランタイムライブラリルーチンのプロトタイプは **openacc.h** という名のヘッダファイルで与えられる。すべてのライブラリルーチンは"C"リンケージの **extern** 関数である。このファイルが定義するのは

- この章のすべてのルーチンのプロトタイプ
- アクセラレータのタイプを記述するための列挙体を含めた、それらのプロトタイプで使われるデータ型

Fortran では、インターフェース定義は **openacc_lib.h** という名の Fortran インクルードファイルと **openacc** という名の Fortran モジュールで与えられる。それらのファイルが定義するのは

- この章のすべてのルーチンのインターフェース
- アクセラレータプログラミングモデルがサポートするバージョンの年を yyyy、月を mm として yyyymm という値をもつ整数パラメータ **openacc_version**。この値はプリプロセッサ変数 **_OPENACC** の値と一致する。
- それらのルーチンの引数のための整数の種類を定義する整数パラメータ
- アクセラレータのタイプを記述する整数パラメータ

多くのルーチンが受け入れる、または返す値はアクセラレータデバイスのタイプに対応する。C,C++ではデバイスタイプ値に使われるデータ型は **acc_device_t** で、Fortran では対応するデータ型は **integer(kind=acc_device_kind)**である。デバイス型で可能な値は実装定義で、C,C++インクルードファイル **openacc.h** や Fortran インクルードファイル **openacc_lib.h** や Fortran モジュール **openacc_lib** でリストアップされている。次の 4 つの値はいつもサポートされる。**acc_device_none, acc_device_default, acc_device_host, acc_device_not_host**。他の値は実装によりインクルードされる適切なファイルから探すか、実装のドキュメントを読むように。**acc_device_default** はどの関数からも決して返されない。つまりそれを引数として使うとランタイムライブラリにその実装のデフォルトデバイス型を使うよう伝える。

3.2 ランタイムライブラリルーチン

3.2.1 acc_get_num_devices

概要

acc_get_num_devices ルーチンはホストに接続されている与えられた型のアクセラレータデバイスの数を返す。

形式

C,C++:

```
int acc_get_num_devices( acc_device_t );
```

Fortran:

```
integer function acc_get_num_devices( devicetype )
integer(acc_device_kind) devicetype
```

説明

acc_get_num_devices ルーチンはホストに接続されている与えられた型のアクセラレータデバイスの数を返す。引数はどの種類のデバイスを数えるかを伝える。

3.2.2 acc_set_device_type

概要

acc_set_device_type ルーチンはアクセラレータ parallel または kernels 領域を実行するときにどのデバイスの型を使うかランタイムに伝える。これは実装によりプログラムが 1 つより多くの型のアクセラレータを使うようコンパイルできるときに便利である。

形式

C,C++:

```
void acc_set_device_tuype ( acc_device_t );
```

Fortran:

```
subroutine acc_set_device_type ( devicetype )
integer(acc_device_kind) devicetype
```

説明

acc_set_device_type ルーチンはそれらが利用可能な間にどのデバイスの型を使うかランタイムに伝える。効果的であるために、このルーチンはアクセラレータ data, parallel, kernels 領域に入る前か **acc_shutdown** を呼んだ後に呼び出すべきである。

制約

- このルーチンはアクセラレータ parallel, kernels, data 領域を実行中に呼び出してはならない。

- もしデバイス型の指定が利用できないなら、動作は実装で定義される。特に、プログラムを中止するかもしれない。
- もしアクセラレータ領域が 1 つのデバイス型のみを使うようコンパイルされるなら、異なるデバイス型でこのルーチンを呼び出すと定義されていない動作が発生するかもしれない。

3.2.3 acc_get_device_type

概要

acc_get_device_type ルーチンはもし次のアクセラレータ領域を実行するのに使うデバイスの型が選ばれているならそれをプログラムに伝える。これは実装によりプログラムが 1 つ以上の型のアクセラレータを使うようコンパイルできるときに便利である。

形式

C,C++:

```
acc_device_t acc_get_device_type ( void );
```

Fortran:

```
function acc_get_device_type ()
integer(acc_device_kind) acc_get_device
```

説明

acc_get_device_type ルーチンは値を返す。もし次のアクセラレータ parallel または kernels 領域を実行するのに使うデバイスの型が選ばれているならプログラムにそれを伝えるための値を返す。デバイス型は **acc_set_device_type** 呼び出しや環境変数を用いてプログラムにより選択されているか、プログラムのデフォルト動作により選択されているだろう。 これは 1 つ以上の型のアクセラレータデバイスで実行するようコンパイルされたアクセラレータ領域のみに効果がある。

制約

- このルーチンはアクセラレータ parallel または kernels 領域の実行中に呼び出してはならない。
- もしデバイス型が選択されていなかったら、**acc_device_none** 値が返される。

3.2.4 acc_set_device_num

概要

acc_set_device_num ルーチンはどのデバイスを使うかをランタイムに伝える。

形式

C,C++:

```
void acc_set_device_num( int, acc_device_t );
```

Fortran:

```
subroutine acc_set_device_num( devicenum, devicetype )
integer devicenum
integer(acc_device_kind) devicetype
```

説明

acc_set_device_num ルーチンは与えられた型の接続されているどのデバイスを使うかをランタイムに伝える。もし **devicenum** がゼロなら、ランタイムは実装で定義されるデフォルト動作に戻る。もし第 2 引数の値がゼロなら、選択されたデバイス番号はすべての接続されたアクセラレータ型で使われる。

制約

- このルーチンをアクセラレータ parallel, kernels, data 領域の実行中に呼び出してはならない。
- **devicenum** の値がそのデバイス型に対する **acc_get_num_devices** が返す値より大きいときの動作は実装で定義される。
- **acc_set_device_num** の呼び出しはそのデバイス型を引数とする **acc_set_device_type** の呼び出しを伴う。

3.2.5 acc_get_device_num

概要

acc_get_device_num ルーチンは次のアクセラレータ parallel または kernels 領域を実行する指定されたデバイス型のデバイス番号を返す。

形式

C,C++:

```
int acc_get_device_num( acc_device_t );
```

Fortran:

```
integer function acc_get_device_num( devicetype )
integer(acc_device_kind) devicetype
```

説明

acc_get_device_num ルーチンは次のアクセラレータ parallel または kernels 領域を実行する指定されたデバイス型のデバイス番号に対応する整数を返す。

制約

- このルーチンをアクセラレータ parallel または kernels 領域の実行中に呼び出してはならない。

3.2.6 acc_async_test

概要

acc_async_test ルーチンはすべての関連した非同期アクティビティの完了をテストする。

形式

C,C++:

```
int acc_async_test( int );
```

Fortran:

```
logical function acc_async_test( arg )
integer(acc_handle_kind) arg
```

説明

引数は整数式でなければならない。もしその値が 1 つ以上の **async** 節で現れ、すべてのその非同期アクティビティが完了しているなら、**acc_async_test** ルーチンは非ゼロ値か**.true.**を返す。もしその非同期アクティビティのいくつかが完了していないなら、**acc_async_test** ルーチンはゼロ値か**.false.**を返す。もし同じアクセラレータを共有するホストスレッドが 2 つ以上あるなら、このホストスレッドが開始した該当するすべての非同期アクティビティが完了している場合のみ **acc_async_test** ルーチンはゼロ値か**.false.**を返す。つまり他のホストスレッドが開始したすべての該当する非同期アクティビティが完了している保証はない。

制約

- このルーチンをアクセラレータ parallel または kernels 領域が呼び出してはならない。

3.2.7 acc_async_test_all

概要

acc_async_test_all ルーチンはすべての非同期アクティビティの完了を待機する。*1

形式

C,C++:

```
int acc_async_test_all();
```

Fortran:

```
logical function acc_async_test_all()
```

説明

もしすべての未完了の非同期アクティビティが完了しているなら、**acc_async_test_all** ルーチンは非ゼロ値か**.true.**を返す。もしいくつかの非同期アクティビティが完了していないなら、**acc_async_test_all** ルーチンはゼロ値か**.false.**を返す。もし同じアクセラレータを共有するホス

トスレッドが 2 つ以上あるなら、このホストスレッドが開始した未完了の非同期アクティビティすべてが完了している場合のみ **acc_async_test_all** ルーチンはゼロ値か**.false.**を返す。つまり他のホストスレッドが開始した非同期アクティビティすべてが完了している保証はない。

制約

- このルーチンをアクセラレータ parallel または kernels 領域が呼び出してはならない。

3.2.8 acc_async_wait

概要

acc_async_wait ルーチンはすべての関連する非同期アクティビティの完了を待機する。

形式

C,C++:

```
void acc_async_wait( int );
```

Fortran:

```
subroutine acc_async_wait( arg )
integer(acc_handle_kind) arg
```

説明

引数は整数式でなければならない。もしその値が 1 つ以上の **async** 節で現れるなら、**acc_async_wait** ルーチンは最後の非同期アクティビティが完了するまで戻らない。もし同じアクセラレータを共有するホストスレッドが 2 つ以上あるなら、このホストスレッドが開始したすべての該当する非同期アクティビティが完了している場合のみ **acc_async_wait** ルーチンは戻る。つまり他のホストスレッドが開始したすべての該当する非同期アクティビティが完了している保証はない。

制約

- このルーチンをアクセラレータ parallel または kernels 領域が呼び出してはならない。

3.2.9 acc_async_wait_all

概要

acc_async_wait_all ルーチンはすべての非同期アクティビティの完了を待機する。

形式

C,C++:

```
void acc_async_wait_all( );
```

Fortran:

subroutine acc_async_wait_all( )

説明

acc_async_wait_all ルーチンはすべての非同期アクティビティが完了するまで戻らない。もし同じアクセラレータを共有するホストスレッドが 2 つ以上あるなら、このホストスレッドが開始したすべての非同期アクティビティが完了している場合のみ **acc_async_wait_all** ルーチンは戻る。つまり他のホストスレッドが開始したすべての非同期アクティビティが完了している保証はない。

制約

- このルーチンをアクセラレータ parallel または kernels 領域が呼び出してはならない。

3.2.10 acc_init

概要

acc_init ルーチンはそのデバイス型にランタイムを初期化するようランタイムに伝える。性能統計を収集するときに計算コストから初期化コストを分けるために使う。

形式

C,C++:

void acc_init (acc_device_t);

Fortran:

subroutine acc_init (devicetype)
integer(acc_device_kind) devicetype

説明

acc_init ルーチンは **acc_set_device** も呼び出す。効果的であるために、このルーチンはアクセラレータ領域に入る前か、**acc_shutdown** 呼び出しの後に呼び出すべきである。

制約

- このルーチンをアクセラレータ parallel または kernels 領域が呼び出してはならない。
- 指定されたデバイス型が利用できないときの動作は実装で定義される。特に、プログラムを中止するかもしれない。
- もし間に **acc_shutdown** を呼び出すことなく、異なる値のデバイス型引数でルーチンが 2 回以上呼び出されるなら、実装で定義された動作をする。
- もしいくつかのアクセラレータ領域がただ 1 つのデバイス型を使うようコンパイルされているなら、異なるデバイス型でこのルーチンを呼び出すと定義されていない動作が発生するかもしれない。

3.2.11 acc_shutdown

概要

acc_shutdown ルーチンは与えられたアクセラレータデバイスとの接続を終了し、ランタイム資源を空けるようランタイムに伝える。これはプログラムに異なるデバイス型上で実行する方法が組み込まれているときに、異なるデバイスに接続するために使う。

形式

C,C++:

```
void acc_shutdown ( acc_devie_t );
```

Fortran:

```
subroutine acc_shutdown ( devicetype )
integer(acc_device_kind) devicetype
```

説明

acc_shutdown ルーチンはアクセラレータデバイスからプログラムを切り離す。

制約

- このルーチンをアクセラレータ領域の実行中に呼び出してはならない。

3.2.12 acc_on_device

概要

acc_on_device ルーチンはプログラムに特定のデバイス上で実行しているかどうかを伝える。

形式

C,C++:

```
int acc_on_device ( acc_device_t );
```

Fortran:

```
logical function acc_on_device ( devicetype )
integer(acc_device_kind) devicetype
```

説明

acc_on_device ルーチンはコードがホストで実行しているか、アクセラレータで実行しているかに依存して異なる経路を実行するために使う。もし **acc_on_device** ルーチンがコンパイル時に不変の引数をもつなら、コンパイル時に定数に評価される。引数は定義されたアクセラレータ

型のひとつでなければならない。もし引数が **acc_device_host** なら、アクセラレータ parallel または kernels 領域の外側か、ホストプロセッサで実行しているアクセラレータ parallel または kernels 領域の内側で、このルーチンは C,C++で非ゼロ、Fortran で**.true.**を評価する。そうでなければ C,C++でゼロ、Fortran で**.false.**を評価する。

3.2.13 acc_malloc

概要

acc_malloc ルーチンはアクセラレータデバイス上のメモリを確保する。

形式

C,C++:

```
void* acc_malloc ( size_t );
```

説明

acc_malloc ルーチンはアクセラレータデバイス上のメモリを確保するのに使う。この関数で割り当てられたポインタは **deviceptr** 節でコンパイラにポインタの対象がアクセラレータにあることを伝えるのに使う。

3.2.14 acc_free

概要

acc_free はアクセラレータデバイス上のメモリを解放する。

形式

C,C++:

```
void acc_free ( void* );
```

説明

acc_free ルーチンは以前に確保したアクセラレータデバイス上のメモリを解放する。つまり引数は **acc_malloc** 呼び出しで返されるポインタ値であるべきである。

4. 環境変数

この章はアクセラレータ領域の動作を変更する環境変数について記述する。環境変数名は大文字でなければならない。環境変数の値は大文字と小文字を区別せず、先頭や末尾に空白があってもよい。もしプログラムが開始した後に環境変数の値が変化したなら、プログラム自身がその値を変更するとしても、動作は実装で定義される。

4.1 ACC_DEVICE_TYPE

プログラムが二つ以上の異なるタイプのデバイスを使用するようコンパイルされているならば、**ACC_DEVICE_TYPE** 環境変数はアクセラレータ並列・カーネル領域を実行するときに使うデフォルトデバイスタイプを制御する。この環境変数で許可される値は実装で定義される。現在サポートされている値はリリースノートを見るように。

例:

```
setenv ACC_DEVICE_TYPE NVIDIA
export ACC_DEVICE_TYPE=NVIDIA
```

4.2 ACC_DEVICE_NUM

ACC_DEVICE_NUM 環境変数はアクセラレータ領域を実行する際に使用するデフォルトデバイス番号を制御する。この環境変数の値は非負でゼロからホストに接続された希望したタイプのデバイス数の間の数でなければならない。もし値がゼロなら、実装で定義されたデフォルトが使われる。もし値が接続されたデバイス数より大きいなら、動作は実装で定義される。

例:

```
setenv ACC_DEVICE_NUM 1
export ACC_DEVICE_NUM=1
```

5. 用語解説

明確で、一致した用語はプログラミングモデルを説明する中で重要である。ここで定義するのはこのドキュメントと関連したプログラミングモデルを効果的に使うために理解しなければならない用語である。

アクセラレータ
: CPU に接続された特定目的のコプロセッサで、計算負荷の高い計算を実行するために CPU がデータと計算カーネルをオフロードできるもの。

バリア
: 同期の 1 つのタイプですべての並列実行ユニットとスレッドがバリアに到達しなければ、どの並列実行ユニットやスレッドもバリアの先へ進めない。

計算密度
: 与えられたループ、領域、プログラムユニットに対して、計算されたデータ上で行われた算術演算の数をメモリ階層の 2 つのレベル間でそのデータを移動するのに必要なメモリ転送数で割った比。

構文
: 指示文と、もしあるなら関連する文、ループ、構造化されたブロック。

計算領域
: *parallel 領域*または *kernels 領域*

CUDA
: NVIDIA からの CUDA 環境は NVIDIA GPU を明示的に制御、プログラムするための C ライクなプログラミング環境である。

データ領域
: アクセラレータ **data** 構文により定義された *region* か、アクセラレータディレクティブを含む関数やサブルーチンのための暗黙のデータ領域。データコンストラクトは典型的に入り口で確保されるデバイスメモリとホストからデバイスメモリへコピーされるデータや、出口でデバイスからホストメモリへコピーされるデータと解放されるデバイスメモリを必要とする。データ領域は他のデータ領域や計算領域を含んでもよい。

デバイス
: いかなる型のアクセラレータへの一般的な参照

DMA
: ダイレクトメモリアクセス、物理的に隔てられたメモリ間でデータを移動する方法。一般的に DMA エンジンにより実行される。DMA エンジンはホスト CPU から分けられており、IO デバイスや他の物理メモリと同じようにホスト物理メモリにアクセスできる。

GPU
: Graphics Processing Unit, アクセラレータデバイスの型の 1 つ

GPGPU
: General Purpose computation on Graphics Processing Units

ホスト
: この文脈でアクセラレータデバイスが接続された主 CPU。ホスト CPU はデバイスにロードされ実行されるプログラム領域やデータを制御する。

カーネル
: アクセラレータにより並列に実行される入れ子のループ。一般的にループは並列ドメインに分割され、ループの本体はカーネルの本体になる。

kernels 領域
: アクセラレータ **kernels** 構文で定義される *領域*。kernels 領域はアクセラレータ向けにコンパイルされる構造化ブロックである。kernels 領域内のコードはコンパイラにより一連のカーネルに分けられる。典型的に各ループネストは単一カーネルになる。kernels

領域は入り口でデバイスメモリを確保してデータをホストからデバイスへコピーし、出口でデータをデバイスからホストメモリへコピーしてデバイスメモリを解放する必要がある。標準のこのバージョンでは、kernels 領域は他の計算領域を含んではならない。

ループトリップカウント
: 特定のループ実行の回数。

MIMD
: 並列実行の手法で(Multiple Instruction, Multiple Data)、異なる実行ユニットやスレッドがそれぞれ非同期的に異なる命令列を実行する。

OpenCL
: Open Compute Language の略。開発中の移植標準 C ライクプログラミング環境で GPU や他のアクセラレータ上での低レベル汎用プログラミングを可能にする。

parallel 領域
: アクセラレータ **parallel** 構文で定義される*領域*。parallel 領域はアクセラレータ向けにコンパイルされた構造化ブロックである。parallel 領域は典型的に 1 つ以上のワークシェアリングループを含む。parallel 領域は入口でデバイスメモリを確保してデータをホストからデバイスへコピーし、出口でデータをデバイスからホストメモリへコピーしてデバイスメモリを解放する必要がある。標準のこのバージョンでは parallel 領域は他の計算領域を含んではならない。

プライベートデータ
: 反復ループに関して、特定のループ反復の間のみで使われるデータ。より一般的なコード領域に関して、領域内で使われ、領域の前に初期化されず、領域の後で使われる前に再初期化されるデータ。

領域
: 構文の実行インスタンス中で出会うすべてのコード。領域はルーチンで呼び出されるコードを含み、構文の動的な範囲と考えてよい。これは *parallel 領域, kernels 領域, data 領域, 暗黙的 data 領域*である。

SIMD
: 並列実行の手法(single-instruction, multiple-data)で、複数のデータ要素に対して同じ命令を同時に実行する。

SIMD 演算
: SIMD 命令で実装されたベクトル演算

構造化ブロック
: C,C++で、先頭に 1 つの入口、末尾に 1 つの出口がある可能な限り複合した実行可能なステートメント。Fortran で、先頭に 1 つの入口、末尾に 1 つの出口がある実行可能なステートメントのブロック。

ベクトル演算
: 配列の各要素に一律に実行される単一演算や一連の演算。

可視デバイスコピー
: デバイスメモリで確保された変数、配列、部分配列のコピーで、コンパイルされているプログラムユニットから見ることができる。