The OpenACC™ Application Programming Interface

Version 1.0

November, 2011

Contents

1. Introduction

This document describes the compiler directives, library routines and environment variables that collectively define the OpenACC™ Application Programming Interface (OpenACC API) for offloading code in C, C++ and Fortran programs from a *host* CPU to an attached *accelerator* device. The method outlined provides a model for accelerator programming that is portable across operating systems and various types of host CPUs and accelerators. The directives extend the ISO/ANSI standard C, C++ and Fortran base languages in a way that allows a programmer to migrate applications incrementally to accelerator targets using standards-based C, C++ or Fortran.

The directives and programming model defined in this document allow programmers to create applications capable of using accelerators, without the need to manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown. Rather, all of these details are implicit in the programming model and are managed by the OpenACC API-enabled compilers and runtime environments. The programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on mapping of loops onto an accelerator, and similar performance-related details.

1.1 Scope

This OpenACC API document covers only user-directed accelerator programming, where the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The remainder of the program will be executed on the host. This document does not describe features or limitations of the host programming environment as a whole; it is limited to specification of loops and regions of code to be offloaded to an accelerator.

This document does not describe automatic detection and offloading of regions of code to an accelerator by a compiler or other tool. This document does not describe targeting loops or code regions to multiple accelerators attached to a single host. While future compilers may allow for automatic offloading, multiple accelerators of the same type, or multiple accelerators of different types, none of these features are addressed in this document.

1.2 Execution Model

The execution model targeted by OpenACC API-enabled compilers is host-directed execution with an attached accelerator device, such as a GPU. The bulk of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The device executes *parallel regions*, which typically contain work-sharing loops, or *kernels regions,* which typically contains one or more loops which are executed as kernels. Even in accelerator-targeted regions, the host must orchestrate the execution by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the parallel region, queuing the device code, waiting for completion, transferring results back to the host, and deallocating memory. In most cases, the host can queue a sequence of operations to be executed on the device, one after the other.

Most current accelerators support two or three levels of parallelism. Most accelerators support coarse-grain parallelism, which is fully parallel execution across execution units. There may be limited support for synchronization across coarse-grain parallel operations.

Many accelerators also support fine-grain parallelism, often implemented as multiple threads of execution within a single execution unit, which are typically rapidly switched on the execution unit to tolerate long latency memory operations. Finally, most accelerators also support SIMD or vector operations within each execution unit. The execution model on the device side exposes these multiple levels of parallelism and the programmer is required to understand the difference between, for example, a fully parallel loop and a loop that is vectorizable but requires synchronization between statements. A fully parallel loop can be programmed for coarse-grain parallel execution. Loops with dependences must either be split to allow coarse-grain parallel execution, or be programmed to execute on a single execution unit using fine-grain parallelism, vector parallelism, or sequentially.

1.3 Memory Model

The most significant difference between a host-only program and a host+accelerator program is that the memory on the accelerator may be completely separate from host memory. This is the case with most current GPUs, for example. In this case, the host may not be able to read or write device memory directly because it is not mapped into the host's virtual memory space. All data movement between host memory and device memory must be performed by the host through runtime library calls that explicitly move data between the separate memories, typically using direct memory access (DMA) transfers. Similarly, it is not valid to assume the accelerator can read or write host memory, though this is supported by some accelerator devices.

The concept of separate host and accelerator memories is very apparent in low-level accelerator programming languages such as CUDA or OpenCL, in which data movement between the memories can dominate user code. In the OpenACC model, data movement between the memories is implicit and managed by the compiler, based on directives from the programmer. However, the programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- Memory bandwidth between host memory and device memory determines the level of compute intensity required to effectively accelerate a given region of code, and
- The limited device memory size may prohibit offloading of regions of code that operate on very large amounts of data.

On the accelerator side, some accelerators (such as current GPUs) implement a weak memory model. In particular, they do not support memory coherence between operations executed by different execution units; even on the same execution unit, memory coherence is only guaranteed when the memory operations are separated by an explicit barrier. Otherwise, if one operation updates a memory location and another reads the same location, or two operations store a value to the same location, the hardware may not guarantee the same result for each execution. While a compiler can detect some potential errors of this nature, it is nonetheless possible to write an accelerator parallel or kernels region that produces inconsistent numerical results.

Some current accelerators have a software-managed cache, some have hardware managed caches, and most have hardware caches that can be used only in certain situations and are limited to read-only data. In low-level programming models such as CUDA or OpenCL languages, it is up to the programmer to manage these caches. In the OpenACC model, these caches are managed by the compiler with hints from the programmer in the form of directives.

1.4 Organization of this document

The rest of this document is organized as follows:

Chapter 2, *Directives*, describes the C, C++ and Fortran directives used to delineate accelerator regions and augment information available to the compiler for scheduling of loops and classification of data.

Chapter 3, *Runtime Library Routines*, defines user-callable functions and library routines to query the accelerator device features and control behavior of accelerator-enabled programs at runtime.

Chapter 4, *Environment Variables*, defines user-settable environment variables used to control behavior of accelerator-enabled programs at execution.

Chapter 5, *Glossary*, defines common terms used in this document.

1.5 References

- *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. Directives

This chapter describes the syntax and behavior of the OpenACC directives. In C and C++, OpenACC directives are specified using the **#pragma** mechanism provided by the language. In Fortran, OpenACC directives are specified using special comments that are identified by a unique sentinel. Compilers will typically ignore OpenACC directives if support is disabled or not provided.

Restrictions

- OpenACC directives may not appear in Fortran **PURE** or **ELEMENTAL** procedures.

2.1 Directive Format

In C and C++, OpenACC directives are specified with the **#pragma** mechanism. The syntax of an OpenACC directive is:

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

Each directive starts with **#pragma acc**. The remainder of the directive follows the C and C++ conventions for pragmas. White space may be used before and after the **#**; white space may be required to separate words in a directive. Preprocessing tokens following the **#pragma acc** are subject to macro replacement. Directives are case sensitive. An OpenACC directive applies to the immediately following statement, structured block or loop.

In Fortran, OpenACC directives are specified in free-form source files as

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

The comment prefix (**!**) may appear in any column, but may only be preceded by white space (spaces and tabs). The sentinel (**!$acc**) must appear as a single word, with no intervening white space. Line length, white space, and continuation rules apply to the directive line. Initial directive lines must have a space after the sentinel. Continued directive lines must have an ampersand (**&**) as the last nonblank character on the line, prior to any comment placed in the directive. Continuation directive lines must begin with the sentinel (possibly preceded by white space) and may have an ampersand as the first non-white space character after the sentinel. Comments may appear on the same line as a directive, starting with an exclamation point and extending to the end of the line. If the first nonblank character after the sentinel is an exclamation point, the line is ignored.

In Fortran fixed-form source files, OpenACC directives are specified as one of

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

c$acc *directive-name [clause [[,] clause]…]*

***$acc** *directive-name [clause [[,] clause]…]*

The sentinel (**!$acc**, **c$acc**, or ***$acc**) must occupy columns 1-5. Fixed form line length, white space, continuation, and column rules apply to the directive line. Initial directive lines

must have a space or zero in column 6, and continuation directive lines must have a character other than a space or zero in column 6. Comments may appear on the same line as a directive, starting with an exclamation point on or after column 7 and continuing to the end of the line.

In Fortran, directives are case-insensitive. Directives cannot be embedded within continued statements, and statements must not be embedded within continued directives. In this document, free form is used for all Fortran OpenACC directive examples.

Only one *directive-name* can be specified per directive. The order in which clauses appear is not significant, and clauses may be repeated unless otherwise specified. Some clauses have a *list* argument; a *list* is a comma-separated list of variable names, array names, or, in some cases, subarrays with subscript ranges.

2.2 Conditional Compilation

The **_OPENACC** macro name is defined to have a value *yyyymm* where *yyyy* is the year and *mm* is the month designation of the version of the OpenACC directives supported by the implementation. This macro must be defined by a compiler only when OpenACC directives are enabled. The version described here is 201111.

2.3 Internal Control Variables

An OpenACC implementation acts as if there are internal control variables (ICVs) that control the behavior of the program. These ICVs are initialized by the implementation, and may be given values through environment variables and through calls to OpenACC API routines. The program can retrieve values through calls to OpenACC API routines.

The ICVs are:

- *acc-device-type-var* - controls which type of accelerator device is used..
- *acc-device-num-var* - controls which accelerator device of the selected type is used.

2.3.1 Modifying and Retrieving ICV Values

The following table shows environment variables or procedures to modify the values of the internal control variables, and procedures to retrieve the values:

ICV	Ways to modify values	Way to retrieve value
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**

The initial values are implementation defined. After initial values are assigned, but before any OpenACC construct or API routine is executed, the values of any environment variables that were set by the user are read and the associated ICVs are modified accordingly. After this point, no changes to the environment variables, either by the program or externally, will affect the ICVs. Clauses on OpenACC constructs do not modify the ICV values.

2.4 Accelerator Compute Constructs

2.4.1 Parallel Construct

Summary

This fundamental construct starts parallel execution on the accelerator device.

Syntax

In C and C++, the syntax of the OpenACC parallel directive is

```
#pragma acc parallel [clause [[,] clause]…] new-line
  structured block
```

and in Fortran, the syntax is

```
!$acc parallel [clause [[,] clause]…]
      structured block
!$acc end parallel
```

where *clause* is one of the following:

```
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 )
```

Description

When the program encounters an accelerator **parallel** construct, gangs of workers are created to execute the accelerator parallel region. Once the gangs are created, the number of gangs and the number of workers in each gang remain constant for the duration of that parallel region. One worker in each gang begins executing the code in the structured block of the construct.

If the **async** clause is not present, there is an implicit barrier at the end of the accelerator parallel region, and the host program will wait until all gangs have completed execution.

An array or variable of aggregate data type referenced in the **parallel** construct that does not appear in a data clause for the construct or any enclosing **data** construct will be treated as if it appeared in a **present_or_copy** clause for the **parallel** construct. A scalar variable referenced in the **parallel** construct that does not appear in a data clause for the construct or any enclosing **data** construct will be treated as if it appeared in a **private** clause (if not live-in or live-out) or a **copy** clause for the **parallel** construct.

Restrictions

- OpenACC parallel regions may not contain other parallel regions or kernels regions.
- A program may not branch into or out of an OpenACC **parallel** construct.
- A program must not depend on the order of evaluation of the clauses, or on any side effects of the evaluations.
- At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical value; in C or C++, the condition must evaluate to a scalar integer value.

The **copy**, **copyin**, **copyout**, **create**, **present**, **present_or_copy**, **present_or_copyin**, **present_or_copyout**, **present_or_create**, **deviceptr**, **firstprivate**, and **private** data clauses are described in Section 2.7.

2.4.2 Kernels Construct

Summary

This construct defines a region of the program that is to be compiled into a sequence of kernels for execution on the accelerator device.

Syntax

In C and C++, the syntax of the OpenACC kernels directive is

```
#pragma acc kernels [clause [[,] clause]…] new-line
    structured block
```

and in Fortran, the syntax is

```
!$acc kernels [clause [[,] clause]…]
     structured block
!$acc end kernels
```

where *clause* is one of the following:

```
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* **)**

Description

The compiler will break the code in the kernels region into a sequence of accelerator kernels. Typically, each loop nest will be a distinct kernel. When the program encounters a **kernels** construct, it will launch the sequence of kernels in order on the device. The number and configuration of gangs of workers and vector length may be different for each kernel.

If the **async** clause is not present, there is an implicit barrier at the end of the kernels region, and the host program will wait until all kernels have completed execution.

An array or variable of aggregate data type referenced in the **kernels** construct that does not appear in a data clause for the construct or any enclosing **data** construct will be treated as if it appeared in a **present_or_copy** clause for the **kernels** construct. A scalar referenced in the **kernels** construct that does not appear in a data clause for the construct or any enclosing **data** construct will be treated as if it appeared in a **private** clause (if not live-in or live-out) or a **copy** clause for the **kernels** construct.

Restrictions

- OpenACC kernels regions may not contain other parallel regions or kernels regions.
- A program may not branch into or out of an OpenACC **kernels** construct.
- A program must not depend on the order of evaluation of the clauses, or on any side effects of the evaluations.
- At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical value; in C or C++, the condition must evaluate to a scalar integer value.

The **copy**, **copyin**, **copyout**, **create**, **present**, **present_or_copy**, **present_or_copyin**, **present_or_copyout**, **present_or_create**, and **deviceptr** data clauses are described in Section 2.7.

2.4.3 if clause

The **if** clause is optional on the **parallel** and **kernels** constructs; when there is no **if** clause, the compiler will generate code to execute the region on the accelerator device.

When an **if** clause appears, the compiler will generate two copies of the construct, one copy to execute on the accelerator and one copy to execute on the host. When the *condition* in the **if** clause evaluates to zero in C or C++, or **.false.** in Fortran, the host copy will be executed. When the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the accelerator copy will be executed.

2.4.4 async clause

The **async** clause is optional on the **parallel** and **kernels** constructs; when there is no **async** clause, the host process will wait until the parallel or kernels region is complete before executing any of the code that follows the construct. When there is an **async** clause,

the parallel or kernels region will be executed by the accelerator device asynchronously while the host process continues with the code following the region.

If present, the argument to the **async** must be an integer expression (**int** for C or C++, **integer** for Fortran). The same integer expression value may be used in a **wait** directive or various runtime routines to have the host process test for or wait for completion of the region. An **async** clause may also be used with no argument, in which case the implementation will use a value distinct from all explicit **async** arguments in the program.

Two asynchronous activities with the same argument value will be executed on the device in the order they are encountered by the host process. Two asynchronous activities with different handle values may be executed on the device in any order relative to each other. If there are two or more host threads executing and sharing the same accelerator device, two asynchronous activities with the same argument value will execute on the device one after the other, though the relative order is not determined.

2.4.5 num_gangs clause

The **num_gangs** clause is allowed on the **parallel** construct. The value of the integer expression defines the number of parallel gangs that will execute the region. If the clause is not specified, an implementation-defined default will be used.

2.4.6 num_workers clause

The **num_workers** clause is allowed on the **parallel** construct. The value of the integer expression defines the number of workers within each gang that will execute the region. If the clause is not specified, an implementation-defined default will be used; the default value may be 1.

2.4.7 vector_length clause

The **vector_length** clause is allowed on the **parallel** construct. The value of the integer expression defines the vector length to use for vector or SIMD operations within each worker of the gang. If the clause is not specified, an implementation-defined default will be used. This vector length will be used for loops annotated with the **vector** clause on a **loop** directive, and for loop automatically vectorized by the compiler. There may be implementation-defined limits on the allowed values for the vector length expression.

2.4.8 private clause

The **private** clause is allowed on the **parallel** construct; it declares that a copy of each item on the list will be created for each parallel gang.

2.4.9 firstprivate clause

The **firstprivate** clause is allowed on the **parallel** construct; it declares that a copy of each item on the list will be created for each parallel gang, and that the copy will be initialized with the value of that item on the host when the **parallel** construct is encountered.

2.4.10 reduction clause

The **reduction** clause is allowed on the **parallel** construct. It specifies a reduction operator and one or more scalar variables. For each variable, a private copy is created for each parallel gang and initialized for that operator. At the end of the region, the values for each gang are combined using the reduction operator, and the result combined with the value of the original variable and stored in the original variable. The reduction result is available after the region.

The following table lists the operators that are valid and the initialization values; in each case, the initialization value will be cast into the variable type. For **max** and **min** reductions, the initialization values are the least representable value and the largest representable value for the variable's data type, respectively. Supported data types are the numerical data types in C and C++ (int, float, double, complex) and Fortran (integer, real, double precision, complex).

C and C++		Fortran	
operator	initialization value	operator	initialization value
+	**0**	**+**	**0**
*****	**1**	*****	**1**
max	least	**max**	least
min	largest	**min**	largest
&	**~0**	**iand**	all bits on
|	**0**	**ior**	**0**
^	**0**	**ieor**	**0**
&&	**1**	**.and.**	**.true.**
||	**0**	**.or.**	**.false.**
		.eqv.	**.true.**
		.neqv.	**.false.**

2.5 Data Construct

Summary

The **data** construct defines scalars, arrays and subarrays to be allocated in the device memory for the duration of the region, whether data should be copied from the host to the device memory upon region entry, and copied from the device to host memory upon region exit.

Syntax

In C and C++, the syntax of the OpenACC data directive is

#pragma acc data *[clause [[,] clause]…] new-line*
structured block

and in Fortran, the syntax is

```
!$acc data [clause [[,] clause]…]
      structured block
!$acc end data
```

where *clause* is one of the following:

```
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 )
```

Description

Data will be allocated in the device memory and copied from the host memory to the device, or copied back, as required. The data clauses are described in Sections 2.7.

2.5.1 if clause

The **if** clause is optional; when there is no **if** clause, the compiler will generate code to to allocate memory on the accelerator device and move data from and to the host.
When an **if** clause appears, the program will conditionally allocate memory on, and move data to and/or from the device. When the *condition* in the **if** clause evaluates to zero in C or C++, or **.false.** in Fortran, no device memory will be allocated, and no data will be moved. When the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the data will be allocated and moved as specified.

2.6 Host_Data Construct

Summary

The **host_data** construct makes the address of device data available on the host.

Syntax

In C and C++, the syntax of the OpenACC data directive is

```
#pragma acc host_data [clause [[,] clause]…] new-line
  structured block
```

and in Fortran, the syntax is

```
!$acc host_data [clause [[,] clause]…]
      structured block
!$acc end host_data
```

where the only valid *clause* is:

use_device(*list* **)**

Description

This construct is used to make the device address of data available in host code.

2.6.1 use_device clause

The **use_device** tells the compiler to use the device address of any variable or array in the *list* in code within the construct. In particular, this may be used to pass the device address of variables or arrays to optimized procedures written in a lower-level API. The variables or arrays in *list* must be present in the accelerator memory due to data regions that contain this construct.

2.7 Data Clauses

These data clauses may appear on the **parallel** construct, **kernels** construct or the **data** construct. The list argument to each data clause is a comma-separated collection of variable names, array names, or subarray specifications. In all cases, the compiler will allocate and manage a copy of the variable or array in device memory, creating a visible device copy of that variable or array.

The intent is to support accelerators with physically and logically separate memories from the host. However, if the accelerator can access the host memory directly, the implementation may avoid the memory allocation and data movement and simply use the host memory. Therefore, a program that uses and assigns data on the host and uses and assigns the same data on the accelerator within a data region without update directives to manage the coherence of the two copies may get different answers on different accelerators and implementations.

In C and C++, a subarray is an array name followed by an extended array range specification in brackets, with start and length, such as

```
arr[2:n]
```

If the lower bound is missing, zero is used. If the length is missing and the array has known size, the difference between the lower bound and the declared size of the array is used; otherwise the length is required. The subarray **arr[2:n]** means element **a[2]**, **a[3]**, …, **a[2+n-1]**.

In Fortran, a subarray is an array name followed by a comma-separated list of range specifications in parentheses, with lower and upper bound subscripts, such as

```
arr(1:high,low:100)
```

If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if known, are used.

Restrictions

- In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be specified.
- In C and C++, the length for a dynamically allocated array must be explicitly specified.

- If a subarray is specified in a data clause, the compiler may choose to allocate memory for only that subarray on the accelerator.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.
- In Fortran, array pointers may be specified, but pointer association is not preserved in the device memory.
- Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous block of memory.
- In C and C++, if a variable or array of struct or class type is specified, all the data members of the struct or class are allocated and copied, as appropriate. If a struct or class member is a pointer type, the data addressed by that pointer are not implicitly copied.
- In C and C++, a member of a struct or class may be specified, including a subarray of a member. However, the struct variable itself must be automatic, static or global, that is, not accessed via a pointer. Members of a subarray of struct or class type may not be specified.
- In Fortran, if a variable or array with derived type is specified, all the members of that derived type are allocated and copied, as appropriate. If any member has the **`allocatable`** or **`pointer`** attribute, the data accessed through that member are not copied.
- In Fortran, members of variables of derived type may be specified, including a subarray of a member. However, the derived type variable must not have the **`allocatable`** or **`pointer`** attribute. Members of subarrays of derived type may not be specified.

2.7.1 deviceptr clause

The **`deviceptr`** clause is used to declare that the pointers in the *list* are device pointers, so the data need not be allocated or moved between the host and device for this pointer.

In C and C++, the variables in *list* must be pointers.

In Fortran, the variable in *list* must be dummy arguments (arrays or scalars), and may not have the Fortran **`pointer`**, **`allocatable`** or **`value`** attributes.

2.7.2 copy clause

The **`copy`** clause is used to declare that the variables, arrays or subarrays in the *list* have values in the host memory that need to be copied to the device memory, and are assigned values on the accelerator that need to be copied back to the host. If a subarray is specified, then only that subarray of the array needs to be copied. The data is copied to the device memory before entry to the region, and data copied back to the host memory when the region is complete.

2.7.3 copyin clause

The **`copyin`** clause is used to declare that the variables, arrays or subarrays in the *list* have values in the host memory that need to be copied to the device memory. If a subarray is

specified, then only that subarray of the array needs to be copied. If a variable, array or subarray appears in a **copyin,** the clause implies that the data need not be copied back from the device memory to the host memory, even if those values were changed on the accelerator. The data is copied to the device memory upon entry to the region.

2.7.4 copyout clause

The **copyout** clause is used to declare that the variables, arrays or subarrays in the *list* are assigned or contain values in the device memory that need to be copied back to the host memory at the end of the accelerator region. If a subarray is specified, then only that subarray of the array needs to be copied. If a variable, array or subarray appears in a **copyout**, the clause implies that the data need not be copied to the device memory from the host memory, even if those values are used on the accelerator. The data is copied back to the host memory upon exit from the region.

2.7.5 create clause

The **create** clause is used to declare that the variables, arrays or subarrays in the *list* need to be allocated (created) in the device memory, but the values in the host memory are not needed on the accelerator, and any values computed and assigned on the accelerator are not needed on the host. No data in this clause will be copied between the host and device memories.

2.7.6 present clause

The **present** clause is used to tell the implementation that the variables or arrays in the *list* are already present in the accelerator memory due to data regions that contain this region, perhaps from procedures that call the procedure containing this construct. The implementation will find and use that existing accelerator data. If there is no containing data region that has placed any of the variables or arrays on the accelerator, the program will halt with an error.

If the containing data region specifies a subarray, the **present** clause must specify the same subarray, or a subarray that is a proper subset of the subarray in the data region. It is a runtime error if the subarray in the **present** clause includes array elements that are not part of the subarray specified in the data region.

2.7.7 present_or_copy clause

The **present_or_copy** clause is used to tell the implementation to test whether each of the variables or arrays on the *list* is already present in the accelerator memory. If it is already present, that accelerator data is used. If it is not present, the data is allocated in the accelerator memory and copied from the host to the accelerator at region entry and back to the host at region exit, as with the **copy** clause. This clause may be shortened to **pcopy**. The same restrictions regarding subarrays in the **present** clause apply to this clause.

2.7.8 present_or_copyin clause

The **present_or_copyin** clause is used to tell the implementation to test whether each of the variables or arrays on the *list* is already present in the accelerator memory. If it is already present, that accelerator data is used. If it is not present, the data is allocated in the accelerator memory and copied from the host to the accelerator at region entry, as with the **copyin**

clause. This clause may be shortened to **pcopyin**. The same restrictions regarding subarrays in the **present** clause apply to this clause.

2.7.9 present_or_copyout clause

The **present_or_copyout** clause is used to tell the implementation to test whether each of the variables or arrays on the *list* is already present in the accelerator memory. If it is already present, that accelerator data is used. If it is not present, the data is allocated in the accelerator memory and copied from the accelerator back to the host at region exit, as with the **copyout** clause. This clause may be shortened to **pcopyout**. The same restrictions regarding subarrays in the **present** clause apply to this clause.

2.7.10 present_or_create clause

The **present_or_create** clause is used to tell the implementation to test whether each of the variables or arrays on the *list* is already present in the accelerator memory. If it is already present, that accelerator data is used. If it is not present, the data is allocated in the accelerator memory, as with the **create** clause. This clause may be shortened to **pcreate**. The same restrictions about subarrays in the **present** clause applies to this clause.

2.8 Loop Construct

Summary

The OpenACC **loop** directive applies to a loop which must immediately follow this directive. The **loop** directive can describe what type of parallelism to use to execute the loop and declare loop-private variables and arrays and reduction operations.

Syntax

In C and C++, the syntax of the **loop** directive is

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

In Fortran, the syntax of the l**oop** directive is

!$acc loop *[clause [[,] clause]…]*
do loop

where *clause* is one of the following:

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

Some clauses are only valid in the context of a parallel region, and some only in the context of a kernels region; see the descriptions below.

In a parallel region, a **loop** directive with no **gang**, **worker** or **vector** clauses allows the implementation to automatically select whether to execute the loop across gangs, workers within a gang, or whether to execute as vector operations. The implementation may also choose to use vector operations to execute any loop with no **loop** directive, using classical automatic vectorization.

2.8.1 collapse clause

The **collapse** clause is used to specify how many tightly nested loops are associated with the **loop** construct. The argument to the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present, only the immediately following loop is associated with the loop directive.

If more than one loop is associated with the **loop** construct, the iterations of all the associated loops are all scheduled according to the rest of the clauses. The trip count for all loops associated with the **collapse** clause must be computable and invariant in all the loops.

It is implementation-defined whether a **gang**, **worker** or **vector** clause on the directive is applied to each loop, or to the linearized iteration space.

2.8.2 gang clause

In an accelerator parallel region, the **gang** clause specifies that the iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the gangs created by the **parallel** construct. No argument is allowed. The loop iterations must be data independent, except for variables specified in a **reduction** clause.

In an accelerator kernels region, the **gang** clause specifies that the iterations of the associated loop or loops are to be executed in parallel across the gangs created for any kernel contained within the loop or loops. If an argument is specified, it specifies how many gangs to use to execute the iterations of this loop.

2.8.3 worker clause

In an accelerator parallel region, the **worker** clause specifies that the iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the multiple workers within a single gang. No argument is allowed. The loop iterations must be data independent, except for variables specified in a **reduction** clause. It is implementation-defined whether a loop with the **worker** clause may contain a loop containing the **gang** clause.

In an accelerator kernels region, the **worker** clause specifies that the iterations of the associated loop or loops are to be executed in parallel across the workers within the gangs created for any kernel contained within the loop or loops. If an argument is specified, it specifies how many workers to use to execute the iterations of this loop.

2.8.4 seq clause

The **seq** clause specifies that the associated loop or loops are to be executed sequentially by the accelerator; this is the default in an accelerator **parallel** region. This clause will override any automatic compiler parallelization or vectorization.

2.8.5 vector clause

In an accelerator parallel region, the **vector** clause specifies that the iterations of the associated loop or loops are to be executed in vector or SIMD mode. The operations will execute using vectors of the length specified or chosen for the parallel region. It is implementation-defined whether a loop with the **vector** clause may contain a loop containing the **gang** or **worker** clause.

In an accelerator kernels region, the vector clause specifies that the iterations of the associated loop or loops are to be executed with vector or SIMD processing. If an argument is specified, the iterations will be processed in vector strips of that length; if no argument is specified, the compiler will choose an appropriate vector length.

2.8.6 independent clause

The **independent** clause is allowed on **loop** directives in kernels regions, and tells the compiler that the iterations of this loop are data-independent with respect to each other. This allows the compiler to generate code to execute the iterations in parallel with no synchronization.

Restrictions

- It is a programming error to use the **independent** clause if any iteration writes to a variable or array element that any other iteration also writes or reads, except for variables in a reduction clause.

2.8.7 private clause

The **private** clause on a loop directive specifies that a copy of each item on the list will be created for each iteration of the associated loop or loops.

2.8.8 reduction clause

The **reduction** clause is allowed on a loop construct with the **gang**, **worker** or **vector** clauses. It specifies a reduction operator and one or more scalar variables. For each reduction variable, a private copy is created for each iteration of the associated loop or loops and initialized for that operator; see the table in section 2.4.10. At the end of the loop, the values for each iteration are combined using the specified reduction operator, and the result stored in the original variable at the end of the parallel or kernels region.

In a parallel region, if the **reduction** clause is used on a loop with the **vector** or **worker** clauses (and no **gang** clause), and the scalar variable also appears in a **private** clause on the **parallel** construct, the value of the private copy of the scalar will be updated at the exit of the loop. Otherwise, variables that appear in a **reduction** clause on a loop in a parallel region will not be updated until the end of the region.

2.9 Cache Directive

Summary

The cache directive may appear at the top of (inside of) a loop. It specifies array elements or subarrays that should be fetched into the highest level of the cache for the body of the loop.

Syntax

In C and C++, the syntax of the cache directive is

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

In Fortran, the syntax of the cache directive is

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

The entries in *list* must be single array elements or simple subarray. In C and C++, a simple subarray is an array name followed by an extended array range specification in brackets, with start and length, such as

arr[_lower_**:**_length_**]**

where the lower bound is a constant, loop invariant, or the for loop index variable plus or minus a constant or loop invariant, and the length is a constant.

In Fortran, a simple subarray is an array name followed by a comma-separated list of range specifications in parentheses, with lower and upper bound subscripts, such as

arr(_lower_**:**_upper_**,**_lower2_**:**_upper2_**)**

The lower bounds must be constant, loop invariant, or the do loop index variable plus or minus a constant or loop invariant; moreover the difference between the corresponding upper and lower bounds must be a constant.

2.10 Combined Directives

Summary

The combined OpenACC **parallel loop** and **kernels loop** directives are shortcuts for specifying a **loop** directive nested immediately inside a **parallel** or **kernels** construct. The meaning is identical to explicitly specifying a **parallel** or **kernels** directive containing a **loop** directive. Any clause that is allowed on a **parallel** or **loop** directive are allowed on the **parallel loop** directive, and any clause allowed on a **kernels** or **loop** directive are allowed on a **kernels loop** directive.

Syntax

In C and C++, the syntax of the **parallel loop** directive is

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

In Fortran, the syntax of the **parallel loop** directive is

!$acc parallel loop *[clause [[,] clause]…]*
do loop
*[***!$acc end parallel loop***]*

The associated structured block is the loop which must immediately follow the directive. Any of the **parallel** or **loop** clauses valid in a parallel region may appear.

In C and C++, the syntax of the **kernels loop** directive is

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

In Fortran, the syntax of the **kernels loop** directive is

```
!$acc kernels loop [clause [[,] clause]…]
  do loop
[!$acc end kernels loop]
```

The associated structured block is the loop which must immediately follow the directive. Any of the **kernels** or **loop** clauses valid in a kernels region may appear.

Restrictions

- This combined directive may not appear within the body of another accelerator parallel or kernels region.
- The restrictions for the **parallel**, **kernels** and **loop** constructs apply.

2.11 Declare Directive

Summary

A **declare** directive is used in the declaration section of a Fortran subroutine, function, or module, or following an variable declaration in C or C++. It can specify that a variable or array is to be allocated in the device memory for the duration of the implicit data region of a function, subroutine or program, and specify whether the data values are to be transferred from the host to the device memory upon entry to the implicit data region, and from the device to the host memory upon exit from the implicit data region. These directives create a visible device copy of the variable or array.

Syntax

In C and C++, the syntax of the **declare** directive is:

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

In Fortran the syntax of the **declare** directive is:

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

where *declclause* is one of the following:

```
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 )
```

The associated region is the implicit region associated with the function, subroutine, or program in which the directive appears. If the directive appears in a Fortran MODULE subprogram, the associated region is the implicit region for the whole program. Otherwise, the clauses have exactly the same behavior as having an explicit data construct surrounding the body of the procedure with these clauses. The data clauses are described in section 2.7.

Restrictions

- A variable or array may appear at most once in all the clauses of **`declare`** directives for a function, subroutine, program, or module.
- Subarrays are not allowed in **`declare`** directives.
- If a variable or array appears in a **`declare`** directive, the same variable or array may not appear in a data clause for any construct where the declaration of the variable is visible.
- In Fortran, assumed-size dummy arrays may not appear in a **`declare`** directive.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.
- In Fortran, pointer arrays may be specified, but pointer association is not preserved in the device memory.

2.11.1 device_resident clause

Summary

The **`device_resident`** specifies that the memory for the named variables should be allocated in the accelerator device memory, not in the host memory.

In C and C++, this means the host may not be able to access these variables. Variables in *list* must be file static or local to a function.

In Fortran, if the variable has the Fortran **`allocatable`** attribute, the memory for the variable will be allocated in and deallocated from the accelerator device memory when the host program executes an **`allocate`** or **`deallocate`** statement for that variable. If the variable has the Fortran **`pointer`** attribute, it may be allocated or deallocated in the accelerator device memory, or may appear on the left hand side of a pointer assignment statement, if the right hand side variable itself appears in a **`device_resident`** clause. If the variable has neither allocatable nor pointer attributes, it must be local to a subprogram.

2.12 Executable Directives

2.12.1 update directive

Summary

The **`update`** directive is used within an explicit or implicit data region to update all or part of a host memory array with values from the corresponding array in device memory, or to update all or part of a device memory array with values from the corresponding array in host memory.

Syntax

In C and C++, the syntax of the **`update`** directive is:

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

In Fortran the syntax of the **`update`** data directive is:

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

where *clause* is one of the following:

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

The list argument to an **update** clause is a comma-separated collection of variable names, array names, or subarray specifications. Multiple subarrays of the same array may appear in a list. The effect of an **update** clause is to copy data from the accelerator device memory to the host memory for **update host**, and from host memory to accelerator device memory for **update device**. The updates are done in the order in which they appear on the directive. There must be a visible device copy of the variables or arrays that appear in the **host** or **device** clauses. At least one **host** or **device** clause must appear.

2.12.1.1 host clause

The **host** clause specifies that the variables, arrays or subarrays in the list are to be copied from the accelerator device memory to the host memory.

2.12.1.2 device clause

The **device** clause specifies that the variables, arrays or subarrays in the list are to be copied from the accelerator host memory to the accelerator device memory.

2.12.1.3 if clause

The **if** clause is optional; when there is no **if** clause, the compiler will generate code to perform the updates unconditionally. When an **if** clause appears, the compiler will generate code to conditionally perform the updates only when the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran.

2.12.1.4 async clause

The **async** clause is optional; when there is no **async** clause, the host process will wait until the updates are complete before executing any of the code that follows the **update** directive. When there is an **async** clause, the updates will be processed asynchronously while the host process continues with the code following the directive.

If the **async** clause has an argument, that argument must be the name of an integer variable (int for C or C++, integer for Fortran). The variable may be used in a **wait** directive or various runtime routines to make the host process test or wait for completion of the update. An **async** clause may also be used with no argument, in which case the implementation will use a value distinct from all explicit **async** arguments in the program.

Two asynchronous activities with the same argument value will be executed on the device in the order they are encountered by the host process. Two asynchronous activities with different handle values may be executed on the device in any order relative to each other. If there are two or more host threads executing and sharing the same accelerator device, two asynchronous activities with the same argument value will execute on the device one after the other, though the relative order is not determined.

Restrictions

- The **update** directive is executable. It must not appear in place of the statement following an *if*, *while*, *do*, *switch*, or *label* in C or C++, or in place of the statement following a logical *if* in Fortran.
- A variable or array which appears in the list of an **update** directive must have a visible device copy.

2.12.2 wait directive

Summary

The **wait** directive causes the program to wait for completion of an asynchronous activity, such as an accelerator parallel or kernels region or **update** directive.

Syntax

In C and C++, the syntax of the **wait** directive is:

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

In Fortran the syntax of the **wait** directive is:

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

The argument, if specified, must be an integer expression (int for C or C++, integer for Fortran). The host thread will wait until all asynchronous activities that had an **async** clause with an argument with the same value have completed.

If no argument is specified, the host process will wait until all asynchronous activities have completed.

If there are two or more host threads executing and sharing the same accelerator device, a wait directive will cause the host thread to wait until at least all of the appropriate asynchronous activities initiated by that host thread have completed. There is no guarantee that all the similar asynchronous activities initiated by some other host thread will have completed.

3. Runtime Library Routines

This chapter describes the OpenACC runtime library routines that are available for use by programmers. Use of these routines may limit portability to systems that do not support the OpenACC API. Conditional compilation using the **_OPENACC** preprocessor variable may preserve portability.

This chapter has two sections:

- Runtime library definitions
- Runtime library routines

Restrictions

- In Fortran, none of the OpenACC runtime library routines may be called from a **PURE** or **ELEMENTAL** procedure.

3.1 Runtime Library Definitions

In C and C++, prototypes for the runtime library routines described in this chapter are provided in a header file named **openacc.h**. All the library routines are **extern** functions with "C" linkage. This file defines:

- The prototypes of all routines in the chapter.
- Any datatypes used in those prototypes, including an enumeration type to describe types of accelerators.

In Fortran, interface declarations are provided in a Fortran include file named **openacc_lib.h** and in a Fortran module named **openacc**. These files define:

- Interfaces for all routines in the chapter.
- The integer parameter **openacc_version** with a value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the Accelerator programming model supported. This value matches the value of the preprocessor variable **_OPENACC**.
- Integer parameters to define integer kinds for arguments to those routines.
- Integer parameters to describe types of accelerators.

Many of the routines accept or return a value corresponding to the type of accelerator device. In C and C++, the datatype used for device type values is **acc_device_t**; in Fortran, the corresponding datatype is **integer(kind=acc_device_kind)**. The possible values for device type are implementation specific, and are listed in the C or C++ include file **openacc.h**, the Fortran include file **openacc_lib.h** and the Fortran module **openacc_lib**. Four values are always supported: **acc_device_none**, **acc_device_default**, **acc_device_host** and **acc_device_not_host**. For other values, look at the appropriate files included with the implementation, or read the documentation for the implementation. The value **acc_device_default** will never be returned by any function; its use as an argument will tell the runtime library to use the default device type for that implementation.

3.2 Runtime Library Routines

3.2.1 acc_get_num_devices

Summary

The **acc_get_num_devices** routine returns the number of accelerator devices of the given type attached to the host.

Format

C or C++:

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

Fortran:

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

Description

The **acc_get_num_devices** routine returns the number of accelerator devices of the given type attached to the host. The argument tells what kind of device to count.

3.2.2 acc_set_device_type

Summary

The **acc_set_device_type** routine tells the runtime which type of device to use when executing an accelerator parallel or kernels region. This is useful when the implementation allows the program to be compiled to use more than one type of accelerator.

Format

C or C++:

```
void acc_set_device_type ( acc_device_t );
```

Fortran:

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

Description

The **acc_set_device_type** routine tells the runtime which type of device to use among those available. To be effective, this routine should be called before any accelerator data, parallel or kernels regions have been entered, or after an **acc_shutdown** call.

Restrictions

- This routine may not be called during execution of an accelerator parallel, kernels or data region.
- If the device type specified is not available, the behavior is implementation-defined; in particular, the program may abort.
- If the routine is called more than once without an intervening **acc_shutdown** call, with a different value for the device type argument, the behavior is implementation-defined.

- If some accelerator regions are compiled to only use one device type, calling this routine with a different device type may produce undefined behavior.

3.2.3 acc_get_device_type

Summary

The **acc_get_device_type** routine tells the program what type of device will be used to run the next accelerator region, if one has been selected. This is useful when the implementation allows the program to be compiled to use more than one type of accelerator.

Format

C or C++:

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

Fortran:

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

Description

The **acc_get_device_type** routine returns a value to tell the program what type of device will be used to run the next accelerator parallel or kernels region, if one has been selected. The device type may have been selected by the program with an **acc_set_device_type** call, with an environment variable, or by the default behavior of the program. This is only effective for accelerator regions that were compiled to run on more than one type of accelerator device.

Restrictions

- This routine may not be called during execution of an accelerator parallel or kernels region.
- If the device type has not yet been selected, the value **acc_device_none** will be returned.

3.2.4 acc_set_device_num

Summary

The **acc_set_device_num** routine tells the runtime which device to use.

Format

C or 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
```

Description

The **acc_set_device_num** routine tells the runtime which device to use among those attached of the given type. If the value of **devicenum** is zero, the runtime will revert to its default behavior, which is implementation-defined. If the value of the second argument is zero, the selected device number will be used for all attached accelerator types.

Restrictions

- This routine may not be called during execution of an accelerator parallel, kernels or data region.
- If the value of **devicenum** is greater than the value returned by **acc_get_num_devices** for that device type, the behavior is implementation-defined.
- Calling **acc_set_device_num** implies a call to **acc_set_device_type** with that device type argument.

3.2.5 acc_get_device_num

Summary

The **acc_get_device_num** routine returns the device number of the specified device type that will be used to run the next accelerator parallel or kernels region.

Format

C or C++:

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

Fortran:

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

Description

The **acc_get_device_num** routine returns an integer corresponding to the device number of the specified device type that will be used to execute the next accelerator parallel or kernels region.

Restrictions

- This routine may not be called during execution of an accelerator parallel or kernels region.

3.2.6 acc_async_test

Summary

The **acc_async_test** routine tests for completion of all associated asynchronous activities.

Format

C or C++:

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

Fortran:

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

Description

The argument must be an integer expression. If that value appeared in one or more **async** clauses, and all such asynchronous activities have completed, the **acc_async_test** routine will return with a nonzero value or **.true.** If some such asynchronous activities have not completed, the **acc_async_test** routine will return with a zero value or **.false.**. If there are two or more host threads sharing the same accelerator, the **acc_async_test** routine will return with a zero value or **.false.** only if all matching asynchronous activities initiated by this host thread have completed; there is no guarantee that all matching asynchronous activities initiated by other host threads have completed.

Restrictions

- This routine may not be called by an accelerator parallel or kernels region.

3.2.7 acc_async_test_all

Summary

The **acc_async_test_all** routine waits for completion of all asynchronous activities.

Format

C or C++:

```
int acc_async_test_all( );
```

Fortran:

```
logical function acc_async_test_all( )
```

Description

If all outstanding asynchronous activities have completed, the **acc_async_test_all** routine will return with a nonzero value or **.true.** If some asynchronous activities have not completed, the **acc_async_test_all** routine will return with a zero value or **.false.**. If there are two or more host threads sharing the same accelerator, the **acc_async_test_all** routine will return with a zero value or **.false.** only if all outstanding asynchronous activities initiated by this host thread have completed; there is no guarantee that all asynchronous activities initiated by other host threads have completed.

Restrictions

- This routine may not be called by an accelerator parallel or kernels region.

3.2.8 acc_async_wait

Summary

The **acc_async_wait** routine waits for completion of all associated asynchronous activities.

Format

C or C++:

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

Fortran:

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

Description

The argument must be an integer expression. If that value appeared in one or more **async** clauses, the **acc_async_wait** routine will not return until the latest such asynchronous activity has completed. If there are two or more host threads sharing the same accelerator, the **acc_async_wait** routine will return only if all matching asynchronous activities initiated by this host thread have completed; there is no guarantee that all matching asynchronous activities initiated by other host threads have completed.

.**Restrictions**

- This routine may not be called by an accelerator parallel or kernels region.

3.2.9 acc_async_wait_all

Summary

The **acc_async_wait_all** routine waits for completion of all asynchronous activities.

Format

C or C++:

```
void acc_async_wait_all( );
```

Fortran:

```
subroutine acc_async_wait_all( )
```

Description

The **acc_async_wait_all** routine will not return until the all asynchronous activitie have completed. If there are two or more host threads sharing the same accelerator, the **acc_async_wait_all** routine will return only if all asynchronous activities initiated by this host thread have completed; there is no guarantee that all asynchronous activities initiated by other host threads have completed.

Restrictions

- This routine may not be called by an accelerator parallel or kernels region.

3.2.10 acc_init

Summary

The **acc_init** routine tells the runtime to initialize the runtime for that device type. This can be used to isolate any initialization cost from the computational cost, when collecting performance statistics.

Format

C or C++:

```
void acc_init ( acc_device_t );
```

Fortran:

```
subroutine acc_init ( devicetype )
integer(acc_device_kind) devicetype
```

Description

The **acc_init** routine also calls **acc_set_device**. To be effective, this routine should be called before any accelerator regions have been entered, or after an **acc_shutdown** call.

Restrictions

- This routine may not be called by an accelerator parallel or kernels region.
- If the device type specified is not available, the behavior is implementation-defined; in particular, the program may abort.
- If the routine is called more than once without an intervening **acc_shutdown** call, with a different value for the device type argument, the behavior is implementation-defined.
- If some accelerator regions are compiled to only use one device type, calling this routine with a different device type may produce undefined behavior.

3.2.11 acc_shutdown

Summary

The **acc_shutdown** routine tells the runtime to shut down the connection to the given accelerator device, and free up any runtime resources. This may be used to connect to a different device, if the program was built in a way to run on different device types.

Format

C or C++:

```
void acc_shutdown ( acc_device_t );
```

Fortran:

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

Description

The **acc_shutdown** routine disconnects the program from the accelerator device.

Restrictions

- This routine may not be called during execution of an accelerator region.

3.2.12 acc_on_device

Summary

The **acc_on_device** routine tells the program whether it is executing on a particular device.

Format

C or C++:

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

Fortran:

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

Description

The **acc_on_device** routine may be used to execute different paths depending on whether the code is running on the host or on some accelerator. If the **acc_on_device** routine has a compile-time constant argument, it evaluates at compile time to a constant. The argument must be one of the defined accelerator types. If the argument is **acc_device_host**, then outside of an accelerator parallel or kernels region, or in an accelerator parallel or kernels region that is executed on the host processor, this routine will evaluate to nonzero for C or C++, and **.true.** for Fortran; otherwise, it will evaluate to zero for C or C++, and **.false.** for Fortran.

3.2.13 acc_malloc

Summary

The **acc_malloc** routine allocates memory on the accelerator device.

Format

C or C++:

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

Description

The **acc_malloc** routine may be used to allocate memory on the accelerator device. Pointers assigned from this function may be used in **deviceptr** clauses to tell the compiler that the pointer target is resident on the accelerator.

3.2.14 acc_free

Summary

The **acc_free** routine frees memory on the accelerator device.

Format

C or C++:

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

Description

The **acc_free** routine will free previously allocated memory on the accelerator device; the argument should be a pointer value that was returned by a call to **acc_malloc**.

4. Environment Variables

This chapter describes the environment variables that modify the behavior of accelerator regions. The names of the environment variables must be upper case. The values assigned environment variables are case insensitive and may have leading and trailing white space. If the values of the environment variables change after the program has started, even if the program itself modifies the values, the behavior is implementation defined.

4.1 ACC_DEVICE_TYPE

The **ACC_DEVICE_TYPE** environment variable controls the default device type to use when executing accelerator parallel and kernels regions, if the program has been compiled to use more than one different type of device. The allowed values of this environment variable are implementation-defined. See the release notes for currently-supported values of this environment variable.

Example:

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

4.2 ACC_DEVICE_NUM

The **ACC_DEVICE_NUM** environment variable controls the default device number to use when executing accelerator regions. The value of this environment variable must be a nonnegative integer between zero and the number of devices of the desired type attached to the host. If the value is zero, the implementation-defined default is used. If the value is greater than the number of devices attached, the behavior is implementation-defined.

Example:

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

5. Glossary

Clear and consistent terminology is important in describing any programming model. We define here the terms you must understand in order to make effective use of this document and the associated programming model.

Accelerator – a special-purpose co-processor attached to a CPU and to which the CPU can offload data and compute kernels to perform compute-intensive calculations.

Barrier – a type of synchronization where all parallel execution units or threads must reach the barrier before any execution unit or thread is allowed to proceed beyond the barrier; modeled after the starting barrier on a horse race track.

Compute intensity – for a given loop, region, or program unit, the ratio of the number of arithmetic operations performed on computed data divided by the number of memory transfers required to move that data between two levels of a memory hierarchy.

Construct – a directive and the associated statement, loop or structured block, if any.

Compute region – a *parallel region* or a *kernels region*.

CUDA –the CUDA environment from NVIDIA is a C-like programming environment used to explicitly control and program an NVIDIA GPU.

Data region – a *region* defined by an Accelerator **`data`** construct, or an implicit data region for a function or subroutine containing Accelerator directives. Data constructs typically require device memory to be allocated and data to be copied from host to device memory upon entry, and data to be copied from device to host memory and device memory deallocated upon exit. Data regions may contain other data regions and compute regions.

Device – a general reference to any type of accelerator.

Device memory – memory attached to an accelerator, logically and physically separate from the host memory.

Directive – in C or C++, a **`#pragma,`** or in Fortran, a specially formatted comment statement, that is interpreted by a compiler to augment information about or specify the behavior of the program.

DMA – Direct Memory Access, a method to move data between physically separate memories; this is typically performed by a DMA engine, separate from the host CPU, that can access the host physical memory as well as an IO device or other physical memory.

GPU – a Graphics Processing Unit; one type of accelerator device.

GPGPU – General Purpose computation on Graphics Processing Units.

Host – the main CPU that in this context has an attached accelerator device. The host CPU controls the program regions and data loaded into and executed on the device.

Kernel – a nested loop executed in parallel by the accelerator. Typically the loops are divided into a parallel domain, and the body of the loop becomes the body of the kernel.

Kernels region – a *region* defined by an Accelerator **`kernels`** construct. A kernels region is a structured block which is compiled for the accelerator. The code in the kernels region will be divided by the compiler into a sequence of kernels; typically each loop nest will become a single kernel. A kernels region may require device memory to be allocated and data

to be copied from host to device upon region entry, and data to be copied from device to host memory and device memory deallocated upon exit. Kernels regions may not contain other compute regions in this version of the standard.

Loop trip count – the number of times a particular loop executes.

MIMD – a method of parallel execution (Multiple Instruction, Multiple Data) where different execution units or threads execute different instruction streams asynchronously with each other.

OpenCL – short for Open Compute Language, a developing, portable standard C-like programming environment that enables low-level general-purpose programming on GPUs and other accelerators.

Parallel region – a *region* defined by an Accelerator **`parallel`** construct. A parallel region is a structured block which is compiled for the accelerator. A parallel region typically contains one or more work-sharing loops. A parallel region may require device memory to be allocated and data to be copied from host to device upon region entry, and data to be copied from device to host memory and device memory deallocated upon exit. Parallel regions may not contain other compute regions in this version of the standard.

Private data – with respect to an iterative loop, data which is used only during a particular loop iteration. With respect to a more general region of code, data which is used within the region but is not initialized prior to the region and is re-initialized prior to any use after the region.

Region – all the code encountered during an instance of execution of a construct. A region includes any code in called routines, and may be thought of as the dynamic extent of a construct. This may be a *parallel region*, *kernels region*, *data region* or *implicit data region*.

SIMD – A method of parallel execution (single-instruction, multiple-data) where the same instruction is applied to multiple data elements simultaneously.

SIMD operation – a *vector operation* implemented with SIMD instructions.

Structured block – in C or C++, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.

Vector operation – a single operation or sequence of operations applied uniformly to each element of an array.

Visible device copy – a copy of a variable, array, or subarray allocated in device memory, that is visible to the program unit being compiled.

This is a preliminary document and may be changed substantially prior to any release of the software implementing this standard.