The OpenACC™ Application Programming Interface

Version 2.0

June, 2013

Corrected, August, 2013

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 programs written 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 explicitly manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown. Rather, 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 splitting loops or code regions to multiple accelerators attached to a single host. While future compilers may allow for automatic offloading, or offloading to multiple accelerators of the same type, or to multiple accelerators of different types, these possibilities are not addressed in this document.

1.2 Execution Model

The execution model targeted by OpenACC API-enabled implementations is host-directed execution with an attached accelerator device, such as a GPU. Much 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 contain one or more loops which are executed as kernels on the accelerator. Even in accelerator-targeted regions, the host may orchestrate the execution by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the compute 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 exposes these multiple levels of parallelism on the device 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.

OpenACC exposes these three levels of parallelism via *gang*, *worker* and *vector* parallelism. Gang parallelism is coarse-grain. A number of gangs will be launched on the accelerator. Worker parallelism is fine-grain. Each gang will have one or more workers. Vector parallelism is for SIMD or vector operations within a worker.

When executing a compute region on the device, one or more gangs are launched, each with one or more workers, where each worker may have vector execution capability with one or more vector lanes. The gangs start executing in *gang-redundant* mode (GR mode), meaning one vector lane of one worker in each gang executes the same code, redundantly. When the program reaches a loop or loop nest marked for gang-level work-sharing, the program starts to execute in *gang-partitioned* mode (GP mode), where the iterations of the loop or loops are partitioned across gangs for truly parallel execution, but still with only one vector lane per worker and one worker per gang active.

When only one worker is active, in either GR or GP mode, the program is in *worker-single* mode (WS mode). When only one vector lane is active, the program is in *vector-single* mode (VS mode). If a gang reaches a loop or loop nest marked for worker-level work-sharing, the gang transitions to *worker-partitioned* mode (WP mode), which activates all the workers of the gang. The iterations of the loop or loops are partitioned across the workers of this gang. If the same loop is marked for both gang-partitioning and worker-partitioning, then the iterations of the loop are spread across all the workers of all the gangs. If a worker reaches a loop or loop nest marked for vector-level work-sharing, the worker will transition to *vector-partitioned* mode (VP mode). Similar to WP mode, the transition to VP mode activates all the vector lanes of the worker. The iterations of the loop or loops will be partitioned across the vector lanes using vector or SIMD operations. Again, a single loop may be marked for one, two or all three of gang, worker and vector parallelism, and the iterations of that loop will be spread across the gangs, workers and vector lanes as appropriate.

The host program starts executing with a single thread, identified by a program counter and its stack. The thread may spawn additional threads, for instance using the OpenMP API. On the accelerator, a single vector lane of a single worker of a single gang is called a thread. When executing on the device, a parallel execution context is created and may contain many such threads.

The user should not attempt to implement barrier synchronization, critical sections or locks across any of gang, worker or vector parallelism. The execution model allows for an implementation that executes some gangs to completion before starting to execute other gangs. This means that trying to implement synchronization between gangs is likely to fail. In particular, a barrier across gangs cannot be implemented in a portable fashion, since all gangs may not ever be active at the same time. Similarly, the execution model allows for an

implementation that executes some workers within a gang or vector lanes within a worker to completion before starting other workers or vector lanes, or for some workers or vector lanes to be suspended until other workers or vector lanes complete. This means that trying to implement synchronization across workers or vector lanes is likely to fail. In particular, implementing a barrier or critical section across workers or vector lanes using atomic operations and a busy-wait loop may never succeed, since the scheduler may suspend the worker or vector lane that owns the lock, and the worker or vector lane waiting on the lock can never complete.

On some devices, the accelerator may also create and launch parallel kernels, allowing for nested parallelism. In that case, the OpenACC directives may be executed by a host thread or an accelerator thread. This specification uses the term *local thread* or *local memory* to mean the thread that executes the directive, or the memory associated with that thread, whether that thread executes on the host or on the accelerator.

Most accelerators can operate asynchronously with respect to the host thread. With such devices, the accelerator has one or more activity queues. The host thread will enqueue operations onto the device activity queues, such as data transfers and procedure execution. After enqueuing the operation, the host thread can continue execution while the device operates independently and asynchronously. The host thread may query the device activity queue(s) and wait for all the operations in a queue to complete. Operations on a single device activity queue will complete before starting the next operation on the same queue; operations on different activity queues may be active simultaneously and may complete in any order.

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 thread may not be able to read or write device memory directly because it is not mapped into the host thread's virtual memory space. All data movement between host memory and device memory must be performed by the host thread through system 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, often with significant performance penalty.

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 can be 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.
- Host addresses stored to pointers on the host may only be valid on the host; addresses stored to pointers on the device may only be valid on the device. Dereferencing host

pointers on the device or dereferencing device pointers on the host is likely to be invalid on such targets.

OpenACC exposes the separate memories through the use of a device data environment. Device data has an explicit lifetime, from when it is allocated or created until it is deleted. If the device shares physical and virtual memory with the local thread, the device data environment will be shared with the local thread. In that case, the implementation need not create new copies of the data for the device and no data movement need be done. If the device has a physically or virtually separate memory from the local thread, the implementation will allocate new data in the device memory and copy data from the local memory to the device environment.

Some accelerators (such as current GPUs) implement a weak memory model. In particular, they do not support memory coherence between operations executed by different threads; even on the same execution unit, memory coherence is only guaranteed when the memory operations are separated by an explicit memory fence. Otherwise, if one thread updates a memory location and another reads the same location, or two threads 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 Conventions used in this document

Keywords and punctuation that are part of the actual specification will appear in typewriter font:

`#pragma acc`

Italic font is used where a keyword or other name must be used:

`#pragma acc` *directive-name*

For C and C++, *new-line* means the newline character at the end of a line:

`#pragma acc` *directive-name new-line*

Optional syntax is enclosed in square brackets; where an option that may be repeated more than once is followed by ellipses:

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

To simplify the specification and convey appropriate constraint information, a *pqr-list* is a comma-separated list of *pqr* items. For example, an *int-expr-list* is a comma-separated list of one or more integer expressions. A *var-list* is a comma-separated list of one or more variable names or array names; in some clauses, a *var-list* may include subarrays with subscript ranges or may include common block names between slashes. The one exception is *clause-list*, which is a list of one or more clauses optionally separated by commas.

`#pragma acc` *directive-name [clause-list] new-line*

1.5 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, 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.

Chapter Appendix A. Recommendations for Target-Specific Implementations, gives advice to implementers to support more portability across implementations and interoperability with other accelerator APIs.

1.6 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 5.0, October 2012.
- *The OpenCL Specification*, version 1.2, Khronos OpenCL Working Group, November 2011.

1.7 Changes from Version 1.0 to 2.0

- **`_OPENACC`** value updated to **`201306`**
- **`default(none)`** clause on **`parallel`** and **`kernels`** directives
- the implicit data attribute for scalars in **`parallel`** constructs has changed
- the implicit data attribute for scalars in loops with **`loop`** directives with the independent attribute has been clarified
- **`acc_async_sync`** and **`acc_async_noval`** values for async clauses
- Clarified the behavior of the **`reduction`** clause on a **`gang`** loop
- Clarified allowable loop nesting (**`gang`** may not appear inside **`worker`**, which may not appear within **`vector`**)

- **`wait`** clause on **`parallel`**, **`kernels`** and **`update`** directives
- **`async`** clause on the **`wait`** directive
- **`enter data`** and **`exit data`** directives
- Fortran *common block* names may now be specified in many data clauses
- **`link`** clause for the **`declare`** directive
- the behavior of the **`declare`** directive for global data
- the behavior of a data clause with a C or C++ pointer variable has been clarified
- predefined data attributes
- support for multidimensional dynamic C/C++ arrays
- **`tile`** and **`auto`** loop clauses
- **`update self`** introduced as a preferred synonym for **`update host`**
- **`routine`** directive and support for separate compilation
- **`device_type`** clause and support for multiple device types
- nested parallelism using parallel or kernels region containing another parallel or kernels region
- **`atomic`** constructs
- new concepts: gang-redundant, gang-partitioned; worker-single, worker-partitioned; vector-single, vector-partitioned; thread
- new API routines:
  - **`acc_wait`**, **`acc_wait_all`** instead of **`acc_async_wait`** and **`acc_async_wait_all`**
  - **`acc_wait_async`**
  - **`acc_copyin`**, **`acc_present_or_copyin`**
  - **`acc_create`**, **`acc_present_or_create`**
  - **`acc_copyout`**, **`acc_delete`**
  - **`acc_map_data`**, **`acc_unmap_data`**
  - **`acc_deviceptr`**, **`acc_hostptr`**
  - **`acc_is_present`**
  - **`acc_memcpy_to_device`**, **`acc_memcpy_from_device`**
  - **`acc_update_device, acc_update_self`**
- defined behavior with multiple host threads, such as with OpenMP
- recommendations for specific implementations

1.8 Corrections in the August 2013 document

- corrected the **`atomic capture`** syntax for C/C++

- fixed the name of the **acc_wait** and **acc_wait_all** procedures
- fixed description of the **acc_hostptr** procedure

1.9 Topics Deferred For a Future Revision

The following topics are under discussion for a future revision. Some of these are known to be important, while others will depend on feedback from users. Readers who have feedback or want to participate may post a message at the forum at www.openacc.org, or may send email to feedback@openacc.org. No promises are made or implied that all these items will be available in the next revision.

- Full support for C and C++ structs and struct members, including pointer members.
- Full support for Fortran derived types and derived type members, including allocatable and pointer members.
- Defined support with multiple host threads.
- Optionally removing the synchronization or barrier at the end of vector and worker loops.
- Allowing an **if** clause after a **device_type** clause.
- A **default(none)** clause for the loop directive.
- A **shared** clause (or something similar) for the loop directive.
- A standard interface for a profiler or trace or other runtime data collection tool.
- Better support for multiple devices from a single thread, whether of the same type or of different types.

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-list] 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-list]*

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 white 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-list]*

c$acc *directive-name [clause-list]*

***$acc** *directive-name [clause-list]*

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, except that a combined directive name is considered a single *directive-name*. The order in which clauses appear is not significant unless otherwise specified. Clauses may be repeated unless otherwise specified. Some clauses have an argument that can contain a list.

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 201306.

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. Clauses on OpenACC constructs do not modify the ICV values. There is one copy of each ICV for each host thread. An ICV value for a device thread may not be modified.

2.4 Device-Specific Clauses

OpenACC directives can specify different clauses or clause arguments for different accelerators using the **device_type** clause. The argument to the **device_type** clause is a comma-separated list of one or more accelerator architecture name identifiers, or an asterisk. A single directive may have one or several **device_type** clauses. Clauses on a directive with no **device_type** clause apply to all accelerator device types. Clauses that follow a **device_type** clause up to the end of the directive or up to the next **device_type** clause are associated with this **device_type** clause. Clauses associated with a **device_type** clause apply only when compiling for the named accelerator device type. Clauses associated with a **device_type** clause that has an asterisk argument apply to any accelerator device type that was not named in any **device_type** clause on that directive. The **device_type** clauses may appear in any order. For each directive, only certain clauses may follow a **device_type** clause.

Clauses that precede any **device_type** clause are default values. If the same clause is associated with a **device_type** clause, the specific value from the clause associated with the **device_type** is used for that device. If no **device_type** clause applies for a device, or a **device_type** clause applies but the same clause is not associated with this **device_type** clause, the default value is used.

The supported accelerator device types are implementation-defined. Depending on the implementation and the compiling environment, an implementation may support only a single accelerator device type, or may support multiple accelerator device types but only one at a time, or many support multiple accelerator device types in a single compilation.

An accelerator architecture name may be generic, such as a vendor, or more specific, such as a particular generation of device; see Appendix A.1 Target Devices for recommended names. When compiling for a particular device, the implementation will use the clauses associated with the **device_type** clause that specifies most specific architecture name that applies for this device; clauses associated with any other **device_type** clause are ignored. In this context, the asterisk is the least specific architecture name.

Syntax

The syntax of the **device_type** clause is

device_type(*)
device_type(*device-type-list* **)**

The **device_type** clause may be abbreviated to **dtype**.

2.5 Accelerator Compute Constructs

2.5.1 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-list] new-line*

structured block

and in Fortran, the syntax is

```
!$acc parallel [clause-list]
       structured block
!$acc end parallel
```

where *clause* is one of the following:

```
async [( int-expr )]
wait [( int-expr-list )]
num_gangs( int-expr )
num_workers( int-expr )
vector_length( int-expr )
device_type( device-type-list )
if( condition )
reduction( operator : var-list )
copy( var-list )
copyin( var-list )
copyout( var-list )
create( var-list )
present( var-list )
present_or_copy( var-list )
present_or_copyin( var-list )
present_or_copyout( var-list )
present_or_create( var-list )
deviceptr( var-list )
private( var-list )
firstprivate( var-list )
default( none )
```

Description

When the program encounters an accelerator **parallel** construct, one or more gangs are created to execute the accelerator parallel region. The number of gangs, the number of workers per gang and the number of vector lanes per worker remain constant for the duration of that parallel region. Each gang begins executing the code in the structured block in gang-redundant mode. This means that code within the parallel region, but outside of a loop with a **loop** directive and gang-level worksharing, will be executed redundantly by all gangs.

If the **async** clause is not present, there is an implicit barrier at the end of the accelerator parallel region, and the execution of the local thread will not proceed until all gangs have reached the end of the parallel region.

If there is no **default(none)** clause on the construct, the compiler will implicitly determine data attributes for variables that are referenced in the compute construct that do not appear in a data clause on the compute construct or a lexically containing data construct and do not have predetermined data attributes. 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 **firstprivate** clause.

Restrictions

- 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.
- Only the **async**, **wait**, **num_gangs**, **num_workers**, and **vector_length** clauses may follow a **device_type** clause.
- 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 2.6 Data Environment. The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

2.5.2 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-list] new-line
    structured block
```

and in Fortran, the syntax is

```
!$acc kernels [clause-list]
     structured block
!$acc end kernels
```

where *clause* is one of the following:

```
async [ ( int-expr ) ]
wait [ ( int-expr-list ) ]
device_type( device-type-list )
if( condition )
copy( var-list )
copyin( var-list )
copyout( var-list )
create( var-list )
present( var-list )
present_or_copy( var-list )
```

present_or_copyin(*var-list* **)**
present_or_copyout(*var-list* **)**
present_or_create(*var-list* **)**
deviceptr(*var-list* **)**
default(none)

Description

The compiler will split 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 local thread execution will not proceed until all kernels have completed execution.

If there is no **default(none)** clause on the construct, the compiler will implicitly determine data attributes for variables that are referenced in the compute construct that do not appear in a data clause on the compute construct or a lexically containing data construct and do not have predetermined data attributes. 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 variable 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 **copy** clause.

Restrictions

- 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.
- Only the **async** and **wait** clauses may follow a **device_type** clause.
- 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 2.6 Data Environment. The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

2.5.3 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 encountering local thread. When the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the accelerator copy will be executed. When the *condition* in the **if** clause evaluates to zero in C or C++, or **.false.** in Fortran, the encountering local thread will execute the construct.

2.5.4 async clause

The **async** clause is optional; see section 2.14 Asynchronous Behavior for more information.

2.5.5 wait clause

The **wait** clause is optional; see section 2.14 Asynchronous Behavior for more information.

2.5.6 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; the default may depend on the code within the construct.

2.5.7 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 be active after a gang transitions from worker-single mode to worker-partitioned mode. If the clause is not specified, an implementation-defined default will be used; the default value may be 1, and may be different for each **parallel** construct.

2.5.8 vector_length clause

The **vector_length** clause is allowed on the **parallel** construct. The value of the integer expression defines the number of vector lanes that will be active after a worker transitions from vector-single mode to vector-partitioned mode. This clause determines the vector length to use for vector or SIMD operations. 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, as well as loops automatically vectorized by the compiler. There may be implementation-defined limits on the allowed values for the vector length expression.

2.5.9 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.

2.5.10 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.

2.5.11 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.**

2.5.12 default(none) clause

The **default(none)** clause is optional. It tells the compiler not to implicitly determine a data attribute for any variable, but to require that all variables or arrays used in the compute region that do not have predetermined data attributes to explicitly appear in a data clause for the compute construct or for a data construct that lexically contains the **parallel** or **kernels** construct.

2.6 Data Environment

This section describes the data attributes for variables. The data attributes for a variable may be *predetermined*, *implicitly determined*, or *explicitly determined*. Variables with predetermined data attributes may not appear in a data clause that conflicts with that data attribute. Variables with implicitly determined data attributes may appear in a data clause that overrides the implicit attribute. Variables with explicitly determined data attributes are those which appear in a data clause on a data construct, a compute construct, or a declare directive.

OpenACC supports systems with accelerators that have distinct memory from the host, as well as systems with accelerators that share memory with the host. In the former case, the system has separate host memory and device memory. In the latter case, the system has one shared memory. The latter case is called a shared memory device as the accelerator shares

memory with the host thread; the former case is called a non-shared memory device. When a nested OpenACC construct is executed on the device, the default target device for that construct is the same device on which the encountering accelerator thread is executing. In that case, the target device shares memory with the encountering thread.

2.6.1 Variables with Predetermined Data Attributes

The loop variable in a C **`for`** statement or Fortran **`do`** statement that is associated with a loop directive is predetermined to be private to each thread that will execute each iteration of the loop. Loop variables in Fortran **`do`** statements within a parallel or kernels region are predetermined to be private to the thread that executes the loop.

Variables declared in a C block within a compute construct are predetermined to be private to the thread that executes the block. Variables declared in procedures called from a compute construct are predetermined to be private to the thread that executes the procedure call.

2.6.2 Data Regions and Data Lifetimes

There are four types of data regions. When the program encounters a data construct, it creates a data region. Data created on the accelerator for the data construct has a lifetime of the region associated with the construct; it remains live until the program exits the data region.

When the program encounters a compute construct with explicit data clauses or with implicit data allocation added by the compiler, it creates a data region that has a lifetime of the compute construct. Data created on the accelerator for the compute construct has a lifetime of the region associated with the construct, just as with a data construct.

When the program enters a procedure, it creates an implicit data region that has a lifetime of the procedure. That is, the implicit data region is created when the procedure is called, and exited when the program returns from that procedure invocation. Data created on the accelerator for an implicit data region has a lifetime of that invocation of the procedure.

There is also an implicit data region associated with the execution of the program itself. The implicit program data region has a lifetime of the execution of the program. Static or global data created on the accelerator has a lifetime of the execution of the program, or from the time the program attaches to and initializes the accelerator until it detaches and shuts the accelerator down.

In addition to data regions, a program may create and delete data on the accelerator using **`enter data`** and **`exit data`** directives or using runtime API routines. When the program executes an **`enter data`** directive, or executes a call to a runtime API **`acc_copyin`** or **`acc_create`** routine, the program enters a data lifetime for each variable, array or subarray on the directive or for the variable on the runtime API argument list. Such data created on the accelerator has a lifetime from when the directive is executed or the runtime API routine is called until an **`exit data`** directive is executed or a runtime API **`acc_copyout`** or **`acc_delete`** routine is called for that data; if no **`exit data`** directive or appropriate runtime API routine is executed, the data lifetime on the accelerator continues until the program exits.

2.6.3 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-list] new-line
  structured block
```

and in Fortran, the syntax is

```
!$acc data [clause-list]
      structured block
!$acc end data
```

where *clause* is one of the following:

```
if( condition )
copy( var-list )
copyin( var-list )
copyout( var-list )
create( var-list )
present( var-list )
present_or_copy( var-list )
present_or_copyin( var-list )
present_or_copyout( var-list )
present_or_create( var-list )
deviceptr( var-list )
```

Description

Data will be allocated in the device memory and copied from the host or local memory to the device, or copied back, as required. The data clauses are described in Sections 2.6.5 Data Clauses.

2.6.3.1 if clause

The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate memory on the accelerator device and move data from and to the local memory as required. 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. At most one **if** clause may appear.

2.6.4 Enter Data and Exit Data Directives

Summary

An **enter data** directive may be used to define scalars, arrays and subarrays to be allocated in the device memory for the remaining duration of the program, or until an **exit data** directive that deallocates the data. They also tell whether data should be copied from the host to the device memory at the **enter data** directive, and copied from the device to host memory at the **exit data** directive. The dynamic range of the program between the **enter data** directive and the matching **exit data** directive is the data lifetime for that data.

Syntax

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

#pragma acc enter data *clause-list new-line*

and in Fortran, the syntax is

!$acc enter data *clause-list*

where *clause* is one of the following:

if(*condition* **)**
async *[* **(** *int-expr* **)** *]*
wait *[* **(** *int-expr-list* **)** *]*
copyin(*var-list* **)**
create(*var-list* **)**
present_or_copyin(*var-list* **)**
present_or_create(*var-list* **)**

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

#pragma acc exit data *clause-list new-line*

and in Fortran, the syntax is

!$acc exit data *clause-list*

where *clause* is one of the following:

if(*condition* **)**
async *[* **(** *int-expr* **)** *]*
wait *[* **(** *int-expr-list* **)** *]*
copyout(*var-list* **)**
delete(*var-list* **)**

Description

At an **enter data** directive, data will be allocated in the device memory and optionally copied from the host or local memory to the device. This action enters a data lifetime for those variables, arrays or subarrays, and will make the data available for present clauses on constructs within the data lifetime.

At an **exit data** directive, data will be optionally copied from the device memory to the host or local memory and deallocated from device memory. This action exits the

corresponding data lifetime. An **exit data** directive may only be used to exit a data lifetime created by an **enter data** construct or a runtime API routine.

The data clauses are described in Sections 2.6.5 Data Clauses.

2.6.4.1 if clause

The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or deallocate memory on the accelerator device and move data from and to the local memory. When an **if** clause appears, the program will conditionally allocate or deallocate device memory 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 or deallocated, 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 or deallocated and moved as specified.

2.6.4.2 async clause

The **async** clause is optional; see section 2.14 Asynchronous Behavior for more information.

2.6.4.3 wait clause

The **wait** clause is optional; see section 2.14 Asynchronous Behavior for more information.

2.6.5 Data Clauses

These data clauses may appear on the **parallel** construct, **kernels** construct, the **data** construct, and the **enter data** and **exit data** directives. The list argument to each data clause is a comma-separated collection of variable names, array names, or subarray specifications. For all clauses except **deviceptr** and **present**, the list argument may include a Fortran *common block* name enclosed within slashes, if that *common block* name also appears in a **declare** directive **link** clause. 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 local thread. However, if the accelerator can access the local memory directly, the implementation may avoid the memory allocation and data movement and simply share the data in local 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 or implementations.

Restrictions

- Data clauses may not follow a **device_type** clause.

2.6.5.1 Data Specification in Data Clauses

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

```
AA[2:n]
```

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

In C and C++, a two dimensional array may be declared in at least four ways:

- Statically-sized array: **float AA[100][200];**

- Pointer to statically sized rows: **`typedef float row[200]; row* BB;`**
- Statically-sized array of pointers: **`float* CC[200];`**
- Pointer to pointers: **`float** DD;`**

Each dimension may be statically sized, or a pointer to dynamically allocated memory. Each of these may be included in a data clause using subarray notation to specify a rectangular array:

- **`AA[2:n][0:200]`**
- **`BB[2:n][0:m]`**
- **`CC[2:n][0:m]`**
- **`DD[2:n][0:m]`**

Multidimensional rectangular subarrays in C and C++ may be specified for any array with any combination of statically-sized or dynamically-allocated dimensions. For statically sized dimensions, all dimensions except the first must specify the whole dimension, to preserve the contiguous data restriction, discussed below. For dynamically allocated dimensions, the implementation will allocate pointers on the device corresponding to the pointers on the host, and will fill in those pointers as appropriate.

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. All dimensions except the last must specify the whole dimension, to preserve the contiguous data restriction, discussed below.

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 dynamically allocated dimensions of an array must be explicitly specified.
- In C and C++, modifying pointers in pointer arrays during the data lifetime, either on the host or on the device, may result in undefined behavior.
- If a subarray is specified in a data clause, the implementation may choose to allocate memory for only that subarray on the accelerator.
- 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, except for dynamic multidimensional C arrays.
- 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 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.

- If an expression is used in a subscript or subarray expression in a clause on a **data** construct, the same value is used when copying data at the end of the data region, even if the values of variables in the expression change during the data region.

2.6.5.2 deviceptr clause

The **deviceptr** clause is used to declare that the pointers in the *var-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 *var-list* must be pointer variables.

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

For a shared-memory device, host pointers are the same as device pointers, so this clause has no effect.

2.6.5.3 copy clause

The **copy** clause is used to declare that the variables, array, subarrays or common blocks in the *var-list* have values in the local memory that need to be copied to the device memory, for a non-shared memory accelerator, and are assigned values on the accelerator that need to be copied back to the local memory. If a subarray is specified, then only that subarray of the array needs to be copied. On a data construct or compute construct, the data is allocated and copied to the device memory upon entry to the region, and copied back to the local memory and deallocated upon exit from the region. If the device shares memory with the local thread, the data in the **copy** clause will be shared; no memory is allocated or copied.

2.6.5.4 copyin clause

The **copyin** clause is used to declare that the variables, arrays, subarrays or common blocks in the *var-list* have values in the local memory that need to be copied to the device memory, for a non-shared memory accelerator. 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 local memory, even if those values were changed on the accelerator. On a data construct or compute construct, the data is allocated and copied to the device memory upon entry to the region and deallocated upon exit from the region. On an **enter data** directive, the data is allocated and copied to the device memory. If the device shares memory with the local thread, the data in the **copyin** clause will be shared; no memory is allocated or copied.

2.6.5.5 copyout clause

The **copyout** clause is used to declare that the variables, arrays, subarrays or common blocks in the *var-list* are assigned or contain values in the device memory that need to be copied back to the local memory at the end of the accelerator region, for a non-shared memory accelerator. 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 local memory, even if those values are used on the accelerator. On a data construct or compute construct, the data is allocated upon entry to the region, and copied back to the local memory and deallocated upon exit from the region. On an **exit data** directive, the data is copied back to the local memory and deallocated. If the device shares memory with the local thread, the data in the **copyout** clause will be shared; no memory is allocated or copied.

2.6.5.6 create clause

The **create** clause is used to declare that the variables, arrays, subarrays or common blocks in the *var-list* need to be allocated (created) in the device memory, for a non-shared memory accelerator, but the values in the local memory are not needed on the accelerator, and any values computed and assigned on the accelerator are not needed back in local memory. On a data construct or compute construct, the data is allocated in device memory upon entry to the region, and deallocated upon exit from the region. On an **enter data** directive, the data is allocated in device memory. No data in this clause will be copied between the local and device memories. If the device shares memory with the local thread, the data in the **create** clause will be shared; no memory is allocated or copied.

2.6.5.7 delete clause

The **delete** clause is used on **exit data** directives to deallocate arrays, subarrays or common blocks without copying values back to local memory. The data is deallocated, on a non-shared memory device. No action is required or taken if the device shares memory with the local thread.

2.6.5.8 present clause

The **present** clause is used to tell the implementation on a non-shared memory device that the variables or arrays in the *var-list* are already present in device memory due to data regions or data lifetimes that contain this region, such as data constructs within procedures that call the procedure containing this construct, or an enter data directive or runtime API routine called before this routine. The implementation will find and use that existing accelerator data. If there is no active data lifetime that has placed any of the variables or arrays on the accelerator, the behavior is unspecified; in particular, the program may halt with a runtime error.

If a containing data lifetime 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 lifetime. 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 lifetime.

2.6.5.9 present_or_copy clause

The **present_or_copy** clause is used to tell the implementation on a non-shared memory accelerator to test whether each of the variables or arrays on the *var-list* is already present in the accelerator memory, as with the **present** clause.

If the data is already present, the program behaves as with the **present** clause. No new device memory will be allocated and no data will be moved to or from the device memory.

If the data is not present, the program behaves as with the **copy** clause. The data is allocated and copied to the device memory upon entry to the region, and copied back to the local memory and deallocated upon exit from the region.

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

2.6.5.10 present_or_copyin clause

The **present_or_copyin** clause is used to tell the implementation on a non-shared memory accelerator to test whether each of the variables or arrays on the *var-list* is already present in the accelerator memory, as with the **present** clause.

If the data is already present, the program behaves as with the **present** clause. No new device memory will be allocated and no data will be moved to or from the device memory.

If the data is not present, the program behaves as with the **copyin** clause. On a data construct or compute construct, the data is allocated and copied to the device memory upon entry to the region and deallocated upon exit from the region. On an **enter data** directive, the data is allocated and copied to the device memory.

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

2.6.5.11 present_or_copyout clause

The **present_or_copyout** clause is used to tell the implementation on a non-shared memory accelerator to test whether each of the variables or arrays on the *var-list* is already present in the accelerator memory, as with the **present** clause.

If the data is already present, the program behaves as with the **present** clause. No new device memory will be allocated and no data will be moved to or from the device memory.

If the data is not present, the program behaves as with the **copyout** clause. The data is allocated upon entry to the region, and copied back to the local memory and deallocated upon exit from the region.

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

2.6.5.12 present_or_create clause

The **present_or_create** clause is used to tell the implementation on a non-shared memory accelerator to test whether each of the variables or arrays on the *var-list* is already present in the accelerator memory, as with the **present** clause.

If the data is already present, the program behaves as with the **present** clause. No new device memory will be allocated.

If the data is not present, the program behaves as with the **create** clause. On a data construct or compute construct, the data is allocated in device memory upon entry to the region, and deallocated upon exit from the region. On an **enter data** directive, the data is allocated in device memory.

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

2.6.6 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-list new-line
   structured block
```

and in Fortran, the syntax is

```
!$acc host_data clause-list
        structured block
!$acc end host_data
```

where the only valid *clause* is:

```
use_device( var-list )
```

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

2.6.6.1 use_device clause

The **use_device** tells the compiler to use the device address of any variable or array in the *var-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 *var-list* must be present in the accelerator memory due to data regions or data lifetimes that contain this construct. On a shared memory accelerator, the device address may be the same as the host address.

2.7 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-list] new-line
  for loop
```

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

```
!$acc loop [clause-list]
  do loop
```

where *clause* is one of the following:

```
collapse( n )
gang [( gang-arg-list )]
worker [( [num:] int-expr )]
vector [( [length:] int-expr )]
seq
auto
tile( size-expr-list )
device_type( device-type-list )
independent
private( var-list )
reduction( operator : var-list )
```

where *gang-arg* is one of:

```
[num:] int-expr
static: size-expr
```

and *gang-arg-list* may have at most one **num** and one **static** argument,

and where *size-expr* is one of:

*****
int-expr

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.

Restrictions

- Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **auto** and **tile** clauses may follow a **device_type** clause.
- The *int-expr* argument to the **worker** and **vector** clauses must be invariant in the kernels region.

2.7.1 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.

2.7.2 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. A loop construct with the **gang** clause transitions a compute region from gang-redundant mode to gang-partitioned mode. The number of gangs is controlled by the **parallel** construct; only the **static** argument is allowed. The loop iterations must be data independent, except for variables specified in a **reduction** clause. The region of a loop with the **gang** clause may not contain another loop with the **gang** clause unless within a nested **parallel** or **kernels** region

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 with no keyword or an argument after the **num** keyword is specified, it specifies how many gangs to use to execute the iterations of this loop. The region of a loop with the **gang** clause may not contain another loop with a **gang** clause unless within a nested **parallel** or **kernels** region.

The scheduling of loop iterations to gangs is not specified unless the **static** argument appears as an argument. If the **static** argument appears with an integer expression, that expression is used as a *chunk* size. If the static argument appears with an asterisk, the implementation will select a *chunk* size. The iterations are divided into chunks of the selected

chunk size, and the chunks are assigned to gangs starting with gang zero and continuing in round-robin fashion. Two **gang** loops in the same **parallel** region with the same number of iterations, and with **static** clauses with the same argument, will assign the iterations to gangs in the same manner. Two **gang** loops in the same **kernels** region with the same number of iterations, the same number of gangs to use, and with **static** clauses with the same argument, will assign the iterations to gangs in the same manner.

2.7.3 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. A loop construct with a **worker** clause causes a gang to transition from worker-single mode to worker-partitioned mode. In contrast to the **gang** clause, the **worker** clause first activates additional worker-level parallelism and then distributes the loop iterations across those workers. No argument is allowed. The loop iterations must be data independent, except for variables specified in a **reduction** clause. The region of a loop with the **worker** clause may not contain a loop with the **gang** or **worker** clause unless within a nested **parallel** or **kernels** region.

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 a gang created for any kernel contained within the loop or loops. If an argument is specified, it specifies how many workers per gang to use to execute the iterations of this loop. The region of a loop with the **worker** clause may not contain a loop with a **gang** or **worker** clause unless within a nested **parallel** or **kernels** region.

All workers will complete execution of their assigned iterations before any worker proceeds beyond the end of the loop.

2.7.4 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. A loop construct with a **vector** clause causes a worker to transition from vector-single mode to vector-partitioned mode. Similar to the **worker** clause, the **vector** clause first activates additional vector-level parallelism and then distributes the loop iterations across those vector lanes. The operations will execute using vectors of the length specified or chosen for the parallel region. The region of a loop with the **vector** clause may not contain a loop with the **gang**, **worker** or **vector** clause unless within a nested **parallel** or **kernels** region.

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 implementation will choose an appropriate vector length. The region of a loop with the **vector** clause may not contain a loop with a **gang**, **worker** or **vector** clause unless within a nested **parallel** or **kernels** region.

All vector lanes will complete execution of their assigned iterations before any vector lane proceeds beyond the end of the loop.

2.7.5 seq clause

The **seq** clause specifies that the associated loop or loops are to be executed sequentially by the accelerator. This clause will override any automatic parallelization or vectorization.

2.7.6 auto clause

The **auto** clause specifies that the implementation should select whether to apply gang, worker or vector parallelism to this loop. The implementation may be restricted to the types of parallelism it can apply by the presence of loop directives with **gang**, **worker** or **vector** clauses for outer or inner loops. This clause by itself does not tell the implementation that the loop iterations are data independent, and the implementation cannot apply any parallelism unless the loop has the **independent** clause, is implicitly independent because it is in a parallel construct, or the implementation can analyze the loop and determine that the loop iterations are data independent. In a kernels construct, a **loop** directive with no **gang**, **worker**, **vector** or **seq** clause is treated as if it has the **auto** clause.

2.7.7 tile clause

The **tile** clause specifies that the implementation should split each loop in the loop nest into two loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the **tile** clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression or an asterisk. If there are *n* tile sizes in the list, the loop directive must be immediately followed by *n* tightly-nested loops. The first argument in the *size-expr-list* corresponds to the innermost loop of the *n* associated loops, and the last element corresponds to the outermost associated loop. If the tile size is specified with an asterisk, the implementation will choose an appropriate value. Each loop in the nest will be split or *strip-mined* into two loops, an outer *tile* loop and an inner *element* loop. The trip count of the element loop will be limited to the corresponding tile size from the *size-expr-list*. The *tile* loops will be reordered to be outside all the *element* loops, and the *element* loops will all be inside the *tile* loops.

If the **vector** clause appears on the loop directive, the **vector** clause is applied to the *element* loops. If the **gang** clause appears on the loop directive, the **gang** clause is applied to the *tile* loops. If the **worker** clause appears on the loop directive, the **worker** clause is applied to the *element* loops if no **vector** clause appears, and to the *tile* loops otherwise.

2.7.8 device_type clause

The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

2.7.9 independent clause

In a **kernels** construct, the **independent** clause tells the implementation that the iterations of this loop are data-independent with respect to each other. This allows the implementation to generate code to execute the iterations in parallel with no synchronization. In a **parallel** construct, the **independent** clause is implied on all **loop** directives without a **seq** clause.

Restrictions

- It is a programming error to use the **independent** clause on a loop in a kernels construct 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.

2.7.10 private clause

The **private** clause on a **loop** directive specifies that a copy of each item on the *var-list* will be created for each thread that executes one or more iterations of the associated loop or loops. Variables referenced in the loop and not in a **private** clause or predetermined private are not privatized for a thread that execute the loop iterations.

2.7.11 reduction clause

The **reduction** clause specifies a reduction operator and one or more scalar variables. For each reduction variable, a private copy is created for each thread that executes iterations of the associated loop or loops and initialized for that operator; see the table in section 2.5.11 reduction clause. At the end of the loop, the values for each thread 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. If the scalar variable does not appear in a **private** clause on the **parallel** construct, or if the **reduction** clause is used on a loop with the **gang** clause, the value of the scalar will not be updated until the end of the parallel region.

2.8 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( var-list ) new-line
```

In Fortran, the syntax of the cache directive is

```
!$acc cache ( var-list )
```

The entries in *var-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.

2.9 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 is 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-list] new-line*
for loop

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

!$acc parallel loop *[clause-list]*
do loop
*[***!$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-list] new-line*
for loop

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

!$acc kernels loop *[clause-list]*
do loop
*[***!$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

- The restrictions for the **parallel**, **kernels** and **loop** constructs apply.

2.10 Atomic Directive

Summary

An atomic construct ensures that a specific storage location is accessed and/or updated atomically, preventing simultaneous reading and writing by gangs, workers and vector threads that could result in indeterminate values.

Syntax

In C and C++, the syntax of the **atomic** constructs are:

#pragma acc atomic [*atomic-clause* **]** new-line
expression-stmt

or:

#pragma acc atomic capture *new-line*
structured-block

Where *atomic-clause* is one of **read**, **write**, **update**, or **capture**. The *expression-stmt* is an expression statement with one of the following forms:

If the *atomic-clause* is **read**:

```
v = x;
```

If the *atomic-clause* is **write**:

```
x = expr;
```

If the *atomic-clause* is **update** or not present:

```
x++;
x--;
++x;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

If the *atomic-clause* is **capture**:

```
v = x++;
v = x--;
v = ++x;
v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;
```

The *structured-block* is a structured block with one of the following forms:

```
{v = x; x binop= expr;}
{x binop= expr; v = x;}
{v = x; x = x binop expr;}
{v = x; x = expr binop x;}
{x = x binop expr; v = x;}
{x = expr binop x; v = x;}
{v = x; x = expr;}
{v = x; x++;}
{v = x; ++x;}
{++x; v = x;}
{x++; v = x;}
{v = x; x--;}
{v = x; --x;}
```

```
{--x; v = x;}
{x--; v = x;}
```

In the preceding expressions:

- **x** and **v** (as applicable) are both l-value expressions with scalar type.
- During the execution of an atomic region, multiple syntactic occurrences of **x** must designate the same storage location.
- Neither of **v** and *expr* (as applicable) may access the storage location designated by **x**.
- Neither of **x** and *expr* (as applicable) may access the storage location designated by **v**.
- *expr* is an expression with scalar type.
- *binop* is one of **+**, *****, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.
- *binop*, *binop***=**, **++**, and **--** are not overloaded operators.
- The expression **x** *binop expr* must be mathematically equivalent to **x** *binop* **(***expr***)**. This requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr binop* **x** must be mathematically equivalent to **(***expr***)** *binop* **x**. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*, or by using parentheses around *expr* or subexpressions of *expr*.
- For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is unspecified.

In Fortran the syntax of the **atomic** constructs are:

```
!$acc atomic read
  capture-statement
[!$acc end atomic]
```

or

```
!$acc atomic write
  write-statement
[!$acc end atomic]
```

or

```
!$acc atomic [update]
  update-statement
[!$acc end atomic]
```

or

```
!$acc atomic capture
  update-statement
  capture-statement
!$acc end atomic
```

or

```
!$acc atomic capture
  capture-statement
  update-statement
!$acc end atomic
```

or

```
!$acc atomic capture
  capture-statement
  write-statement
!$acc end atomic
```

where *write-statement* has the following form (if clause is **write** or **capture**):

```
x = expr
```

where *capture-statement* has the following form (if clause is **capture** or **read**):

```
v = x
```

and where *update-statement* has one of the following forms (if clause is **update**, **capture**, or not present):

```
x = x operator expr
x = expr operator x
x = intrinsic_procedure_name(x, expr-list)
x = intrinsic_procedure_name(expr-list, x)
```

In the preceding statements:

- **x** and **v** (as applicable) are both scalar variables of intrinsic type.
- **x** must not be an allocatable variable.
- During the execution of an atomic region, multiple syntactic occurrences of **x** must designate the same storage location.
- None of **v**, *expr* and *expr-list* (as applicable) may access the same storage location as **x**.
- None of **x**, *expr* and *expr-list* (as applicable) may access the same storage location as **v**.
- *expr* is a scalar expression.
- *expr-list* is a comma-separated, non-empty list of scalar expressions. If *intrinsic_procedure_name* refers to **iand**, **ior**, or **ieor**, exactly one expression must appear in *expr-list*.
- *intrinsic_procedure_name* is one of **max**, **min**, **iand**, **ior**, or **ieor**. *operator* is one of **+**, *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.** .
- The expression **x** *operator expr* must be mathematically equivalent to **x** *operator* **(***expr***)** . This requirement is satisfied if the operators in *expr* have precedence greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr operator* **x** must be mathematically equivalent to **(***expr***)** *operator* **x**. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- *intrinsic_procedure_name* must refer to the intrinsic procedure name and not to other program entities.
- *operator* must refer to the intrinsic operator and not to a user-defined operator. All assignments must be intrinsic assignments.
- For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is unspecified.

An atomic construct with the **read** clause forces an atomic read of the location designated by **x**. An atomic construct with the **write** clause forces an atomic write of the location designated by **x**.

An atomic construct with the **update** clause forces an atomic update of the location designated by **x** using the designated operator or intrinsic. Note that when no clause is present, the semantics are equivalent to **atomic update**. Only the read and write of the location designated by **x** are performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect to the read or write of the location designated by **x**.

An atomic construct with the **capture** clause forces an atomic update of the location designated by **x** using the designated operator or intrinsic while also capturing the original or final value of the location designated by **x** with respect to the atomic update. The original or final value of the location designated by **x** is written into the location designated by **v** depending on the form of the atomic construct structured block or statements following the usual language semantics. Only the read and write of the location designated by **x** are performed mutually atomically. Neither the evaluation of *expr* or *expr-list*, nor the write to the location designated by **v,** need to be atomic.

For all forms of the atomic construct, any combination of two or more of these atomic constructs enforces mutually exclusive access to the locations designated by **x**. To avoid race conditions, all accesses of the locations designated by **x** that could potentially occur in parallel must be protected with an atomic construct.

Atomic regions do not guarantee exclusive access with respect to any accesses outside of atomic regions to the same storage location **x** even if those accesses occur during the execution of a reduction clause.

If the storage location designated by **x** is not size-aligned (that is, if the byte alignment of **x** is not a multiple of the size of **x**), then the behavior of the atomic region is implementation-defined.

Restrictions

The following restriction applies to the atomic construct:

- All atomic accesses to the storage locations designated by **x** throughout the program are required to have the same type and type parameters.
- Storage locations designated by **x** must be less than or equal in size to the largest available native atomic operator width.

2.11 Declare Directive

Summary

A **declare** directive is used in the declaration section of a Fortran subroutine, function, or module, or following a 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 *clause-list new-line*

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

!$acc declare *clause-list*

where *clause* is one of the following:

copy(*var-list* **)**
copyin(*var-list* **)**
copyout(*var-list* **)**
create(*var-list* **)**
present(*var-list* **)**
present_or_copy(*var-list* **)**
present_or_copyin(*var-list* **)**
present_or_copyout(*var-list* **)**
present_or_create(*var-list* **)**
deviceptr(*var-list* **)**
device_resident(*var-list* **)**
link(*var-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 the declaration section of a Fortran *module* subprogram or in a C or C++ global scope, 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 2.6.5 Data Clauses.

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.
- In Fortran, assumed-size dummy arrays may not appear in a **declare** directive.
- In Fortran, pointer arrays may be specified, but pointer association is not preserved in the device memory.
- In a Fortran *module* declaration section, only **create**, **copyin**, **device_resident** and **link** clauses are allowed.
- In C or C++ global scope, only **create**, **copyin**, **deviceptr**, **device_resident** and **link** clauses are allowed.
- C and C++ *extern* variables may only appear in **create**, **copyin**, **deviceptr**, **device_resident** and **link** clauses on a **declare** directive.

2.11.1 device_resident clause

Summary

The **device_resident** clause specifies that the memory for the named variables should be allocated in the accelerator device memory and not in the host memory. The names in the argument list may be variable or array names, or Fortran *common block* names enclosed between slashes; subarrays are not allowed. The host may not be able to access variables in a

device_resident clause. The accelerator data lifetime of global variables or common blocks specified in a **device_resident** clause is the entire execution of the program.

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 by the host 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.

In Fortran, the argument to a **device_resident** clause may be a *common block* name enclosed in slashes; in this case, all declarations of the common block must have a matching **device_resident** clause. In this case, the *common block* will be statically allocated in device memory, and not in host memory. The *common block* will be available to accelerator routines; see 2.13 Procedure Calls in Compute Regions.

In a Fortran *module* declaration section, a variable or array in a **device_resident** clause will be available to accelerator routines**.**

In C or C++ global scope, a variable or array in a **device_resident** clause will be available to accelerator routines . A C or C++ *extern* variable may appear in a **device_resident** clause only if the actual declaration and all *extern* declarations are also followed by **device_resident** clauses.

2.11.2 link clause

The **link** clause is used for large global host static data that is referenced within an accelerator routine and that should have a dynamic data lifetime on the device. The **link** clause specifies that only a global link for the named variables should be statically created in accelerator memory. The host data structure remains statically allocated and globally available. The device data memory will be allocated only when the global variable appears on a data clause for a data construct, compute construct or **enter data** directive. The arguments to the link clause must be global data. In C or C++, the **link** clause must appear on global scope, or the arguments must be *extern* variables. In Fortran, the **link** clause must appear in a *module* declaration section, or the arguments must be *common block* names enclosed in slashes. A **declare link** clause must be visible everywhere the global variables or common block variables are explicitly or implicitly used in a data clause, compute construct, or accelerator routine**.** The global variable or *common block* variables may be used in accelerator routines. The accelerator data lifetime of variables or common blocks specified in a **link** clause is the data region that allocates the variable or common block with a data clause, or from the execution of the **enter data** directive that allocates the data until an **exit data** directive deallocates it or until the end of the program.

2.12 Executable Directives

2.12.1 Update Directive

Summary

The **update** directive is used during the lifetime of accelerator data to update all or part of local variables or arrays with values from the corresponding memory in device memory, or to

update all or part of device variables or arrays with values from the corresponding local memory.

Syntax

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

#pragma acc update *clause-list new-line*

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

!$acc update *clause-list*

where *clause* is one of the following:

async *[* **(** *int-expr* **)** *]*
wait *[* **(** *int-expr-list* **)** *]*
device_type(*device-type-list* **)**
if(*condition* **)**
self(*var-list* **)**
host(*var-list* **)**
device(*var-list* **)**

The *var-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 *var-list* of the same or different clauses on the same directive. The effect of an **update** clause is to copy data from the accelerator device memory to the local memory for **update self**, and from local 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 device copy of the variables or arrays that appear in the **self** or **device** clauses. At least one **self**, **host** or **device** clause must appear.

2.12.1.1 self clause

The **self** clause specifies that the variables, arrays or subarrays in the *var-list* are to be copied from the accelerator device memory to the local memory for a non-shared memory accelerator. If the accelerator shares the same memory with the encountering thread, no action is taken.

2.12.1.2 host clause

The **host** clause is a synonym for the **self** clause.

2.12.1.3 device clause

The **device** clause specifies that the variables, arrays or subarrays in the *var-list* are to be copied from the local memory to the accelerator device memory, for a non-shared memory accelerator. If the accelerator shares the same memory with the encountering thread, no action is taken.

2.12.1.4 if clause

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

2.12.1.5 async clause

The **async** clause is optional; see section 2.14 Asynchronous Behavior for more information.

2.12.1.6 wait clause

The **`wait`** clause is optional; see section 2.14 Asynchronous Behavior for more information.

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 *var-list* of an **`update`** directive must have a device copy.
- Only the **`async`** and **`wait`** clauses may follow a **`device_type`** clause.
- 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.
- Noncontiguous subarrays may be specified. It is implementation-specific whether noncontiguous regions are updated by using one transfer for each contiguous subregion, or whether the noncontiguous data is packed, transferred once, and unpacked.
- In C and C++, a member of a struct or class may be specified, including a subarray of a member. Members of a subarray of struct or class type may not be specified.
- In C and C++, if a subarray notation is used for a struct member, subarray notation may not be used for any parent of that struct member.
- In Fortran, members of variables of derived type may be specified, including a subarray of a member. Members of subarrays of derived type may not be specified.
- In Fortran, if array or subarray notation is used for a derived type member, array or subarray notation may not be used for an parent of that derived type member.

2.12.2 Wait Directive

See section 2.14 Asynchronous Behavior for more information.

2.12.3 Enter Data Directive

See section 2.6.4 Enter Data and Exit Data Directives for more information.

2.12.4 Exit Data Directive

See section 2.6.4 Enter Data and Exit Data Directives for more information.

2.13 Procedure Calls in Compute Regions

This section describes how routines are compiled for an accelerator and how procedure calls are compiled in compute regions.

2.13.1 Routine Directive

Summary

The **routine** directive is used to tell the compiler to compile a given procedure for an accelerator as well as the host. In a file or routine with a procedure call, the **routine** directive tells the implementation the attributes of the procedure when called on the accelerator.

Syntax

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

```
#pragma acc routine clause-list new-line
#pragma acc routine( name ) clause-list new-line
```

In C and C++, the **routine** directive without a name may appear immediately before a function definition or just before a function prototype and applies to that immediately following function or prototype. The **routine** directive with a name may appear anywhere that a function prototype is allowed and applies to the function in that scope with that name, but must appear before any definition or use of that function.

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

```
!$acc routine clause-list
!$acc routine( name ) clause-list
```

In Fortran, the **routine** directive without a name may appear within the specification part of a subroutine or function definition, or within an interface body for a subroutine or function in an interface block, and applies to the containing subroutine or function. The **routine** directive with a name may appear in the specification part of a subroutine, function or module, and applies to the named subroutine or function.

A C or C++ function or Fortran subprogram compiled with the **routine** directive for an accelerator is called an *accelerator routine*.

The *clause* is one of the following:

```
gang
worker
vector
seq
bind( name )
bind( string )
device_type( device-type-list )
nohost
```

Restrictions

- Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device_type** clause.
- In C and C++, function static variables are not supported in functions to which a **routine** directive applies.
- In Fortran, variables with the *save* attribute, either explicitly or implicitly, are not supported in subprograms to which a **routine** directive applies.

2.13.1.1 gang clause

The **gang** clause specifies that the procedure contains, may contain, or may call another procedure that contains a loop with a **gang** clause. A call to this procedure must appear in code that is executed in *gang-redundant* mode, and all gangs must execute the call. For instance, a procedure with a **routine gang** directive may not be called from within a loop that has a **gang** clause. Only one of **gang**, **worker**, **vector** and **seq** may be specified for each device type.

2.13.1.2 worker clause

The **worker** clause specifies that the procedure contains, may contain, or may call another procedure that contains a loop with a **worker** clause, but does not contain nor does it call another procedure that contains a loop with the **gang** clause. A loop in this procedure with an **auto** clause may be selected by the compiler to execute in **worker** or **vector** mode. A call to this procedure must appear in code that is executed in *worker-single* mode, though it may be in *gang-redundant* or *gang-partitioned* mode. For instance, a procedure with a **routine worker** directive may be called from within a loop that has the **gang** clause, but not from within a loop that has the **worker** clause. Only one of **gang**, **worker**, **vector** and **seq** may be specified for each device type.

2.13.1.3 vector clause

The **vector** clause specifies that the procedure contains, may contain, or may call another procedure that contains a loop with the **vector** clause, but does not contain nor does it call another procedure that contains a loop with either a **gang** or **worker** clause. A loop in this procedure with an **auto** clause may be selected by the compiler to execute in **vector** mode, but not **worker** mode. A call to this procedure must appear in code that is executed in *vector-single* mode, though it may be in *gang-redundant* or *gang-partitioned* mode, and in *worker-single* or *worker-partitioned* mode. For instance, a procedure with a **routine vector** directive may be called from within a loop that has the **gang** clause or the **worker** clause, but not from within a loop that has the **vector** clause. Only one of **gang**, **worker**, **vector** and **seq** may be specified for each device type.

2.13.1.4 seq clause

The **seq** clause specifies that the procedure does not contain nor does it call another procedure that contains a loop with a **gang**, **worker** or **vector** clause. A loop in this procedure with an **auto** clause will be executed in **seq** mode. A call to this procedure may appear in any mode. Only one of **gang**, **worker**, **vector** and **seq** may be specified for each device type.

2.13.1.5 bind clause

The **bind** clause specifies the name to use when compiling or calling the procedure. If the name is specified as an identifier, it is compiled or called as if that name were specified in the language being compiled. If the name is specified as a string, the string is used for the procedure name unmodified.

2.13.1.6 device_type clause

The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

2.13.1.7 nohost clause

The **nohost** tells the compiler not to compile a version of this procedure for the host. All calls to this procedure must appear within accelerator compute regions. If this procedure is

called from other procedures, those other procedures must also have a matching **routine** directive with the **nohost** clause.

2.13.2 Global Data Access

C or C++ global, file static or *extern* variables or array, and Fortran *module* or *common block* variables or arrays, that are used in accelerator routines must appear in a declare directive in a **create**, **copyin**, **device_resident** or **link** clause. If the data appears in a **device_resident** clause, the **routine** directive for the procedure must include the **nohost** clause. If the data appears in a **link** clause, that data must have an active accelerator data lifetime by virtue of appearing in a data clause for a data construct, compute construct or **enter data** directive.

2.14 Asynchronous Behavior

This section describes the **async** clause and the behavior of programs that use asynchronous data movement and compute constructs.

2.14.1 async clause

The async clause may appear on a **parallel** or **kernels** construct, or an **enter data**, **exit data**, **update** or **wait** directive. In all cases, the **async** clause is optional; when there is no **async** clause, the local thread will wait until the compute construct or data operations are complete before executing any of the code that follows, or, on the **wait** directive, until all operations on the appropriate asynchronous activity queues are complete. When there is an **async** clause, the parallel or kernels region or data operations may be processed asynchronously while the local thread continues with the code following the construct or directive.

The **async** clause may have a single *async-argument*, where an *async-argument* is a nonnegative scalar integer expression (*int* for C or C++, *integer* for Fortran), or one of the special async values defined below. The behavior with a negative *async-argument*, except the special async values defined below, is implementation-defined. The value of the *async-argument* may be used in a **wait** directive, **wait** clause, or various runtime routines to test or wait for completion of the operation.

Two special **async** values are defined in the C and Fortran header files and the Fortran **openacc** module. These are negative values, so as not to conflict with a user-specified nonnegative *async-argument*. An **async** clause with the *async-argument* **acc_async_noval** will behave the same as if the **async** clause had no argument. An **async** clause with the *async-argument* **acc_async_sync** will behave the same as if no **async** clause appeared.

The *async-value* of any operation is the value of the *async-argument*, if present, or **acc_async_noval** if the **async** clause had no value, or **acc_async_sync** if no **async** clause appeared. If the device supports asynchronous operation with one or more device activity queues, the *async-value* is used to select the queue onto which to enqueue an operation. The properties of the device and the implementation will determine how many actual activity queues are supported, and how the *async-value* is mapped onto the actual activity queues. Two asynchronous operations with the same *async-value* will be enqueued

onto the same activity queue, and therefore will be executed on the device in the order they are encountered by the local thread. Two asynchronous operations with different *async-values* may be enqueued onto different activity queues, and therefore may be executed on the device in either order relative to each other. If there are two or more threads executing and sharing the same accelerator device, two asynchronous operations with the same *async-value* will be enqueued on the same activity queue, but unless the threads are synchronized with respect to each other, the operations may be enqueued in either order and therefore may execute on the device in either order.

2.14.2 wait clause

The **wait** clause may appear on a **parallel** or **kernels** construct, or an **enter data**, **exit data**, or **update** directive. In all cases, the **wait** clause is optional. When there is no **wait** clause, the associated compute or update operations may be enqueued or launched or executed immediately on the device. If there is an argument to the **wait** clause, it must be a list of one or more *async-arguments*. The compute, data or update operation may not be launched or executed until all operations enqueued up to this point by this thread on the associated asynchronous device activity queues have completed. One legal implementation is for the local thread to wait for all the associated asynchronous device activity queues. Another legal implementation is for the local thread to enqueue the compute or update operation in such a way that the operation will not start until the operations enqueued on the associated asynchronous device activity queues have completed.

2.14.3 Wait Directive

Summary

The **wait** directive causes the local thread to wait for completion of asynchronous operations, such as an accelerator parallel or kernels region or an **update** directive, or causes one device activity queue to synchronize with one or more other activity queues.

Syntax

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

#pragma acc wait [(*int-expr-list* **)]** *clause-list new-line*

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

!$acc wait [(*int-expr-list* **)]** *clause-list*

where *clause* is:

async *[* **(** *int-expr* **)** *]*

The wait argument, if present, must be one or more *async-arguments*.

If there is no wait argument and no **async** clause, the local thread will wait until all operations enqueued by this thread on any device activity queue have completed.

If there are one or more *int-expr* expressions and no **async** clause, the local thread will wait until all operations enqueued by this thread on each of the associated device activity queues have completed.

If there are two or more threads executing and sharing the same accelerator device, a **wait** directive with no **async** clause will cause the local thread to wait until all of the appropriate asynchronous operations previously enqueued have completed. To guarantee that operations

have been enqueued by other threads requires additional synchronization between those threads.

If there is an **`async`** clause, no new operation may be launched or executed on the **`async`** device activity queue until all operations enqueued up to this point by this thread on the asynchronous activity queues associated with the wait argument have completed. One legal implementation is for the local thread to wait for all the associated asynchronous device activity queues. Another legal implementation is for the thread to enqueue a synchronization operation in such a way that no new operation will start until the operations enqueued on the associated asynchronous device activity queues have completed.

3. Runtime Library

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.
- The values of **acc_async_noval** and **acc_async_sync**.

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.
- The values of **acc_async_noval** and **acc_async_sync.**

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**. 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

In this section, for the C and C++ prototypes, pointers are typed **h_void*** or **d_void*** to designate a host address or device address, as if the following definitions were included:

```
#define h_void void
#define d_void void
```

Except for **acc_on_device**, these routines are only available on the host.

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.

Restrictions

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

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.

Restrictions

- This routine may not be called within 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 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_type
```

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.

Restrictions

- This routine may not be called within an accelerator parallel or kernels region.
- If the device type has not yet been selected, the value **acc_device_none** may 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 negative, 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 within an accelerator parallel, kernels or data region.
- If the value of **devicenum** is greater than or equal to 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 within 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 operations.

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 *async-argument* as defined in Section 2.14.1 async clause. If that value appeared in one or more **async** clauses, and all such asynchronous operations have completed, the **acc_async_test** routine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some such asynchronous operations have not completed, the

acc_async_test routine will return with a zero value in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the **acc_async_test** routine will return with a nonzero value or **.true.** only if all matching asynchronous operations initiated by this thread have completed; there is no guarantee that all matching asynchronous operations initiated by other threads have completed.

3.2.7 acc_async_test_all

Summary

The **acc_async_test_all** routine tests for completion of all asynchronous operations.

Format

C or C++:

```
int acc_async_test_all( );
```

Fortran:

```
logical function acc_async_test_all( )
```

Description

If all outstanding asynchronous operations have completed, the **acc_async_test_all** routine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some asynchronous operations have not completed, the **acc_async_test_all** routine will return with a zero value in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the **acc_async_test_all** routine will return with a nonzero value or **.true.** only if all outstanding asynchronous operations initiated by this thread have completed; there is no guarantee that all asynchronous operations initiated by other threads have completed.

3.2.8 acc_wait

Summary

The **acc_wait** routine waits for completion of all associated asynchronous operations.

Format

C or C++:

```
void acc_wait( int );
```

Fortran:

```
subroutine acc_wait( arg )
integer(acc_handle_kind) arg
```

Description

The argument must be an *async-argument* as defined in Section 2.14.1 async clause. If that value appeared in one or more **async** clauses, the **acc_wait** routine will not return until the latest such asynchronous operation has completed. If two or more threads share the same accelerator, the **acc_wait** routine will return only if all matching asynchronous operations initiated by this thread have completed; there is no guarantee that all matching asynchronous operations initiated by other threads have completed. For compatibility with version 1.0, this routine may also be spelled **acc_async_wait**.

3.2.9 acc_wait_async

Summary

The **acc_wait_async** routine enqueues a wait operation on one async queue for the operations previously enqueued on another async queue.

Format

C or C++:

```
void acc_wait_async( int, int );
```

Fortran:

```
subroutine acc_wait_async( arg, async )
integer(acc_handle_kind) arg, async
```

Description

The **acc_wait_async** routine is equivalent to the **wait** directive with an **async** clause. The arguments must be *async-arguments*, as defined in Section 2.14.1 async clause. The routine will enqueue a wait operation on the appropriate device queue associated with the second argument, which will wait for operations enqueued on the device queue associated with the first argument. See section 2.14 Asynchronous Behavior for more information.

3.2.10 acc_wait_all

Summary

The **acc_wait_all** routine waits for completion of all asynchronous operations.

Format

C or C++:

```
void acc_wait_all( );
```

Fortran:

```
subroutine acc_wait_all( )
```

Description

The **acc_wait_all** routine will not return until the all asynchronous operations have completed. If two or more threads share the same accelerator, the **acc_wait_all** routine will return only if all asynchronous operations initiated by this thread have completed; there is no guarantee that all asynchronous operations initiated by other threads have completed. For compatibility with version 1.0, this routine may also be spelled **acc_async_wait_all**.

3.2.11 acc_wait_all _async

Summary

The **acc_wait_all_async** routine enqueues wait operations on one async queue for the operations previously enqueued on all other async queues.

Format

C or C++:

```
void acc_wait_all_async( int );
```

Fortran:

```
subroutine acc_wait_all_async( async )
integer(acc_handle_kind) async
```

Description

The **acc_wait_all_async** routine is equivalent to the **wait** directive with an **async** clause containing values for all other asynchronous activity queues. The argument must be an *async-argument* as defined in Section 2.14.1 async clause. The routine will enqueue a wait operation on the appropriate device queue for each other device queue; see section 2.14 Asynchronous Behavior for more information.

3.2.12 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 implicitly calls **acc_set_device_type**.

Restrictions

- This routine may not be called within 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.

3.2.13 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.

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.

3.2.14 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 compute region or accelerator routine, or in an accelerator compute region or accelerator routine 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. If the argument is **acc_device_not_host**, the result is the negation of the result with argument **acc_device_host**. If the argument is any accelerator device type, then in an accelerator compute region or accelerator routine that is executed on an accelerator of that device type, 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. The result with argument **acc_device_default** is undefined.

3.2.15 acc_malloc

Summary

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

Format

C or C++:

```
d_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.

3.2.16 acc_free

Summary

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

Format

C or C++:

```
void acc_free( d_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**.

3.2.17 acc_copyin

Summary

The **acc_copyin** routine allocates memory on the accelerator device to correspond to the specified host memory, and copies the data to that device memory on a non-shared memory accelerator.

Format

C or C++:

```
void* acc_copyin( h_void*, size_t );
```

Fortran:

```
subroutine acc_copyin( a )
 type, dimension(:[,:]…) :: a
subroutine acc_copyin( a, len )
 type :: a
 integer :: len
```

Description

The **acc_copyin** routine is equivalent to the **enter data** directive with a **copyin** clause. In C, the arguments are a pointer to the data and length in bytes; the function returns a pointer to the allocated space, as with **acc_malloc**. Pointers assigned from this function may be used in **deviceptr** clauses to tell the compiler that the pointer target is resident on the accelerator. In Fortran, two forms are supported. In the first, the argument is a contiguous array section of intrinsic type. In the second, the first argument is a variable or array element and the second is the length in bytes. Memory is allocated on the device, and the data is copied from the host memory to the newly allocated device memory. A call to this routine starts a data lifetime for the specified data. This data may be accessed in using the **present** data clause. It is a runtime error to call this routine if the data is already present on the device.

3.2.18 acc_present_or_copyin

Summary

The **acc_present_or_copyin** routine tests to see if the data is already present on the device; if not, it allocates memory on the accelerator device to correspond to the specified host memory, and copies the data to that device memory, on a non-shared memory device.

Format
C or C++:

```
void* acc_present_or_copyin( h_void*, size_t );
void* acc_pcopyin( h_void*, size_t );
```

Fortran:

```
subroutine acc_present_or_copyin( a )
 type, dimension(:[,:]…) :: a
subroutine acc_present_or_copyin( a, len )
 type :: a
 integer :: len
subroutine acc_pcopyin( a )
 type, dimension(:[,:]…) :: a
subroutine acc_pcopyin( a, len )
 type :: a
 integer :: len
```

Description
The **acc_present_or_copyin** routine is equivalent to the **enter data** directive with a **present_or_copyin** clause. The arguments are as for the **acc_copyin** routine. If the data is already present on the device, or if the device shares memory with the caller, no action is taken. On a non-shared memory device where the data is not present, memory is allocated on the device, and the data is copied to the newly allocated device memory. In the latter case, a call to this routine starts a data lifetime for the specified data. This data may be accessed in using the **present** data clause.

3.2.19 acc_create

Summary
The **acc_create** routine allocates memory on the accelerator device to correspond to the specified host memory on a non-shared memory accelerator.

Format
C or C++:

```
void* acc_create( h_void*, size_t );
```

Fortran:

```
subroutine acc_create( a )
 type, dimension(:[,:]…) :: a
subroutine acc_create( a, len )
 type :: a
 integer :: len
```

Description
The **acc_create** routine is equivalent to the **enter data** directive with a **create** clause. In C, the arguments are a pointer to the data and length in bytes; the function returns a pointer to the allocated space, as with **acc_malloc**. Pointers assigned from this function may be used in **deviceptr** clauses to tell the compiler that the pointer target is resident on the accelerator. In Fortran, two forms are supported. In the first, the argument is a contiguous array section of intrinsic type. In the second, the first argument is a variable or array element and the second is the length in bytes. On a non-shared memory device, memory is allocated

on the device. A call to this routine starts a data lifetime for the specified data. This data may be accessed in using the **present** data clause. It is a runtime error to call this routine if the data is already present on the device.

3.2.20 acc_present_or_create

Summary

The **acc_present_or_create** routine tests to see if the data is already present on the device; if not, it allocates memory on the accelerator device to correspond to the specified host memory, on a non-shared memory device.

Format

C or C++:

```
void* acc_present_or_create( h_void*, size_t );
void* acc_pcreate( h_void*, size_t );
```

Fortran:

```
subroutine acc_present_or_create( a )
 type, dimension(:[,:]…) :: a
subroutine acc_present_or_create( a, len )
 type :: a
 integer :: len
subroutine acc_pcreate( a )
 type, dimension(:[,:]…) :: a
subroutine acc_pcreate( a, len )
 type :: a
 integer :: len
```

Description

The **acc_present_or_create** routine is equivalent to the **enter data** directive with a **present_or_create** clause. The arguments are as for the **acc_create** routine. If the data is already present on the device, or if the device shares memory with the caller, no action is taken. On a non-shared memory device where the data is not present, memory is allocated on the device. In the latter case, a call to this routine starts a data lifetime for the specified data. This data may be accessed in using the **present** data clause.

3.2.21 acc_copyout

Summary

The **acc_copyout** routine copies data from device memory to the corresponding local memory, then deallocates that memory from the accelerator device, on a non-shared memory accelerator.

Format

C or C++:

```
void acc_copyout( h_void*, size_t );
```

Fortran:

```
subroutine acc_copyout( a )
 type, dimension(:[,:]…) :: a
subroutine acc_copyout( a, len )
 type :: a
 integer :: len
```

Description

The **acc_copyout** routine is equivalent to the **exit data** directive with a **copyout** clause. In C, the arguments are a pointer to the data and length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array section of intrinsic type. In the second, the first argument is a variable or array element and the second is the length in bytes. A call to this routine copies the data from the accelerator device to the local memory, then deallocates the accelerator memory. A call to this routine ends a data lifetime for the specified data. It is a runtime error to call this routine if the data is not present on the device or within a data region for the specified data.

3.2.22 acc_delete

Summary

The **acc_delete** routine deallocates the memory from the accelerator device corresponding to the specified local memory, on a non-shared memory accelerator.

Format

C or C++:

```
void acc_delete( h_void*, size_t );
```

Fortran:

```
subroutine acc_delete( a )
 type, dimension(:[,:]…) :: a
subroutine acc_delete( a, len )
 type :: a
 integer :: len
```

Description

The **acc_delete** routine is equivalent to the **exit data** directive with a **delete** clause. The arguments are as for **acc_copyout**. A call to this routine deallocates the accelerator memory corresponding to the specified local memory. A call to this routine ends a data lifetime for the specified data. It is a runtime error to call this routine if the data is not present on the device or within a data region for the specified data.

3.2.23 acc_update_device

Summary

The **acc_update_device** routine updates the device copy of data from the corresponding local memory on a non-shared memory accelerator.

Format

C or C++:

```
void acc_update_device( h_void*, size_t );
```

Fortran:

```
subroutine acc_update_device( a )
 type, dimension(:[,:]…) :: a
subroutine acc_update_device( a, len )
 type :: a
 integer :: len
```

Description

The **acc_update_device** routine is equivalent to the **update** directive with a **device** clause. In C, the arguments are a pointer to the data and length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array section of intrinsic type. In the second, the first argument is a variable or array element and the second is the length in bytes. On a non-shared memory device, the data in the local memory is copied to the corresponding device memory. It is a runtime error to call this routine if the data is not present on the device.

3.2.24 acc_update_self

Summary

The **acc_update_self** routine updates the device copy of data to the corresponding local memory on a non-shared memory accelerator.

Format

C or C++:

```
void acc_update_self( h_void*, size_t );
```

Fortran:

```
subroutine acc_update_self( a )
 type, dimension(:[,:]…) :: a
subroutine acc_update_self( a, len )
 type :: a
 integer :: len
```

Description

The **acc_update_self** routine is equivalent to the **update** directive with a **self** clause. In C, the arguments are a pointer to the data and length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array section of intrinsic type. In the second, the first argument is a variable or array element and the second is the length in bytes. On a non-shared memory device, the data in the local memory is copied to the corresponding device memory. It is a runtime error to call this routine if the data is not present on the device.

3.2.25 acc_map_data

Summary

The **acc_map_data** routine maps previously allocated device data to the specified host data.

Format

C or C++:

```
void acc_map_data( h_void*, d_void*, size_t );
```

Description

The **acc_map_data** routine is similar to an **enter data** directive with a **create** clause, except instead of allocating new device memory to start a data lifetime, the device address to use for the data lifetime is specified as an argument. The first argument is a host address, followed by the corresponding device address and the data length in bytes. After this call, when the host data appears in a data clause, the specified device memory will be used. It is an error to call **acc_map_data** for host data that is already present on the device. It is undefined to call **acc_map_data** with a device address that is already mapped to host data. The device address may be the result of a call to **acc_malloc**, or may come from some other device-specific API routine.

3.2.26 acc_unmap_data

Summary

The **acc_unmap_data** routine unmaps device data from the specified host data.

Format

C or C++:

```
void acc_unmap_data( h_void* );
```

Description

The **acc_unmap_data** routine is similar to an **exit data** directive with a **delete** clause, except the device memory is not deallocated. The argument is pointer to the host data. A call to this routine ends the data lifetime for the specified host data. The device memory is not deallocated. It is undefined behavior to call **acc_unmap_data** with a host address unless that host address was mapped to device memory using **acc_map_data**.

3.2.27 acc_deviceptr

Summary

The **acc_deviceptr** routine returns the device pointer associated with a specific host address.

Format

C or C++:

```
d_void* acc_deviceptr( h_void* );
```

Description

The **acc_deviceptr** routine returns the device pointer associated with a host address. The argument is the address of a host variable or array that has an active lifetime on the current device. If the data is not present on the device, the routine returns a NULL value.

3.2.28 acc_hostptr

Summary

The **acc_hostptr** routine returns the host pointer associated with a specific device address.

Format

C or C++:

```
h_void* acc_hostptr( d_void* );
```

Description

The **acc_hostptr** routine returns the host pointer associated with a device address. The argument is the address of a device variable or array, such as that returned from **acc_deviceptr**, **acc_create** or **acc_copyin**. If the device address is NULL, or does not correspond to any host address, the routine returns a NULL value.

3.2.29 acc_is_present

Summary

The **acc_is_present** routine tests whether a host variable or array region is present on the device.

Format

C or C++:

```
int acc_is_present( h_void*, size_t );
```

Fortran:

```
logical function acc_is_present( a )
 type, dimension(:[,:]…) :: a
logical function acc_is_present( a, len )
 type :: a
 integer :: len
```

Description

The **acc_is_present** routine tests whether the specified host data is present on the device. In C, the arguments are a pointer to the data and length in bytes; the function returns nonzero if the specified data is fully present, and zero otherwise. In Fortran, two forms are supported. In the first, the argument is a contiguous array section of intrinsic type. In the second, the first argument is a variable or array element and the second is the length in bytes. The function returns **.true.** if the specified data is fully present, and **.false.** otherwise. If the byte length is zero, the function returns nonzero in C or **.true.** in Fortran if the given address is present at all on the device.

3.2.30 acc_memcpy_to_device

Summary

The **acc_memcpy_to_device** routine copies data from local memory to device memory.

Format

C or C++:

```
void acc_memcpy_to_device( d_void* dest, h_void* src,
size_t bytes );
```

Description

The **acc_memcpy_to_device** routine copies **bytes** data from the local address in **src** to the device address in **dest**. The destination address must be a device address, such as would be returned from **acc_malloc** or **acc_deviceptr**.

3.2.31 acc_memcpy_from_device

Summary

The **acc_memcpy_from_device** routine copies data from device memory to local memory.

Format

C or C++:

```
    void acc_memcpy_from_device( h_void* dest, d_void* src,
size_t bytes );
```

Description

The **acc_memcpy_from_device** routine copies **bytes** data from the device address in **src** to the local address in **dest**. The source address must be a device address, such as would be returned from **acc_malloc** or **acc_deviceptr**.

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 greater than or equal to 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.

Accelerator routine – a C or C++ function or Fortran subprogram compiled for the accelerator with the **`routine`** directive.

Accelerator thread – a thread of execution that executes on the accelerator; a single vector lane of a single worker of a single gang.

Async-argument –An *async-argument* is a nonnegative scalar integer expression (*int* for C or C++, *integer* for Fortran), or one of the special async values **`acc_async_noval`** or **`acc_async_sync`**.

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 lifetime – the lifetime of a data object on the device, which may be begin at the entry to a data region, or at an **`enter data`** directive, or at a data API call such as **`acc_copyin`** or **`acc_create`**, and which may end at the exit from a data region, or at an **`exit data`** directive, or at a data API call such as **`acc_delete`**, **`acc_copyout`** or **`acc_shutdown`**, or at the end of the program execution.

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 allocate device memory and copy data from host to device memory upon entry, and copy data from device to host memory and deallocate device memory 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.

Host thread – a thread of execution that executes on the host.

Implicit data region – the data region that is implicitly defined for a Fortran subprogram or C function. A call to a subprogram or function enters the implicit data region, and a return from the subprogram or function exits the implicit data region.

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.

Local memory – the memory associated with the local thread.

Local thread – the host thread or the accelerator thread that executes an OpenACC directive or construct.

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.

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.

Procedure – in C or C++, a function in the program; in Fortran, a subroutine or function.

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.

Thread – On a host processor, a thread is defined by a program counter and stack location; several host threads may comprise a process and share host memory. On an accelerator, a thread is any one vector lane of one worker of one gang on the device.

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.

Appendix A. Recommendations for Target-Specific Implementations

This section gives recommendations for standard names and extensions to use for implementations for specific targets and target platforms, to promote portability across such implementations. While this appendix is not part of the OpenACC specification, implementations that provide the functionality specified herein are strongly recommended to use the names in this section. The first subsection describes target devices, such as NVIDIA GPUs and Intel Xeon Phi Coprocessor. The second subsection describes additional API routines for target platforms, such as CUDA and OpenCL. The third subsection lists several recommended options for implementations. .

A.1 Target Devices

A.1.1 NVIDIA GPU Targets

This section gives recommendations for implementations that target NVIDIA GPU devices.

A.1.1.1 Accelerator Device Type

These implementations should use the name **`acc_device_nvidia`** for the **`acc_device_t`** type or return values from OpenACC Runtime API routines.

A.1.1.2 ACC_DEVICE_TYPE

An implementation should use the case-insensitive name **`NVIDIA`** for the environment variable **`ACC_DEVICE_TYPE`**.

A.1.1.3 device_type clause argument

An implementation should use the name **`nvidia`** or **`NVIDIA`** as the argument to the **`device_type`** clause.

A.1.2 AMD GPU Targets

This section gives recommendations for implementations that target AMD GPUs.

A.1.2.1 Accelerator Device Type

These implementations should use the name **`acc_device_radeon`** for the **`acc_device_t`** type or return values from OpenACC Runtime API routines.

A.1.2.2 ACC_DEVICE_TYPE

These implementations should use the case-insensitive name **`RADEON`** for the environment variable **`ACC_DEVICE_TYPE`**.

A.1.2.3 device_type clause argument

An implementation should use the name **`radeon`** or **`RADEON`** as the argument to the **`device_type`** clause.

A.1.3 Intel Xeon Phi Coprocessor Targets

This section gives recommendations for implementations that target Intel Xeon Phi Coprocessors.

A.1.3.1 Accelerator Device Type

These implementations should use the name **acc_device_xeonphi** for the **acc_device_t** type or return values from OpenACC Runtime API routines.

A.1.3.2 ACC_DEVICE_TYPE

These implementations should use the case-insensitive name **XEONPHI** for the environment variable **ACC_DEVICE_TYPE**.

A.1.3.3 device_type clause argument

An implementation should use the name **xeonphi** or **XEONPHI** as the argument to the **device_type** clause.

A.2 API Routines for Target Platforms

These runtime routines allow access to the interface between the OpenACC runtime API and the underlying target platform. An implementation may not implement all these routines, but if it provides this functionality, it should use these function names.

A.2.1 NVIDIA CUDA Platform

This section gives runtime API routines for implementations that target the NVIDIA CUDA Runtime or Driver API.

A.2.1.1 acc_get_current_cuda_device

Summary

The **acc_get_current_cuda_device** routine returns the NVIDIA CUDA device handle for the current device.

Format

C or C++:

```
void* acc_get_current_cuda_device();
```

A.2.1.2 acc_get_current_cuda_context

Summary

The **acc_get_current_cuda_context** routine returns the NVIDIA CUDA context handle in use for the current device.

Format

C or C++:

```
void* acc_get_current_cuda_context();
```

A.2.1.3 acc_get_cuda_stream

Summary

The **acc_get_cuda_stream** routine returns the NVIDIA CUDA stream handle in use for the current device for the specified async value.

Format

C or C++:

```
void* acc_get_cuda_stream( int async );
```

A.2.1.4 acc_set_cuda_stream

Summary

The **acc_set_cuda_stream** routine sets the NVIDIA CUDA stream handle the current device for the specified async value.

Format

C or C++:

```
int acc_set_cuda_stream( int async, void* stream );
```

A.2.2 OpenCL Target Platform

This section gives runtime API routines for implementations that target the OpenCL API on any device.

A.2.2.1 acc_get_current_opencl_device

Summary

The **acc_get_current_opencl_device** routine returns the OpenCL device handle for the current device.

Format

C or C++:

```
void* acc_get_current_opencl_device();
```

A.2.2.2 acc_get_current_opencl_context

Summary

The **acc_get_current_opencl_context** routine returns the OpenCL context handle in use for the current device.

Format

C or C++:

```
void* acc_get_current_opencl_context();
```

A.2.2.3 acc_get_opencl_queue

Summary

The **acc_get_opencl_queue** routine returns the OpenCL command queue handle in use for the current device for the specified async value.

Format

C or C++:

```
cl_command_queue acc_get_opencl_queue( int async );
```

A.2.2.4 acc_set_opencl_queue

Summary

The **acc_set_opencl_queue** routine returns the OpenCL command queue handle in use for the current device for the specified async value.

Format

C or C++:

```
void acc_set_opencl_queue( int async, cl_command_queue
cmdqueue);
```

A.2.3 Intel Coprocessor Offload Infrastructure (COI) API

These runtime routines allow access to the interface between the OpenACC runtime API and the underlying Intel COI API.

A.2.3.1 acc_get_current_coi_device

Summary

The **acc_get_current_coi_device** routine returns the COI device handle for the current device.

Format

C or C++:

```
void* acc_get_current_coi_device();
```

A.2.3.2 acc_get_current_coi_context

Summary

The **acc_get_current_coi_context** routine returns the COI context handle in use for the current device.

Format

C or C++:

```
void* acc_get_current_coi_context();
```

A.2.3.3 acc_get_coi_pipeline

Summary

The **acc_get_coi_pipeline** routine returns the COI pipeline handle in use for the current device for the specified async value.

Format

C or C++:

```
void* acc_get_coi_pipeline( int async );
```

A.2.3.4 acc_set_coi_pipeline

Summary

The **acc_set_coi_pipeline** routine returns the COI pipeline handle in use for the current device for the specified async value.

Format

C or C++:

```
void acc_set_coi_pipeline( int async, void* pipeline );
```

A.3 Recommended Options

The following options are recommended for implementations; for instance, these may be implemented as command-line options to a compiler or settings in an IDE.

A.3.1 C Pointer in Present clause

This revision of OpenACC clarifies the construct:

```
void test(int n ){
   float* p;
   …
   #pragma acc data present(p)
   { // code here…
```

This example tests whether the pointer **p** itself is present on the device. Implementations before this revision commonly implemented this by testing whether the pointer target **p[0]** was present on the device, and this appears in many programs assuming such. Until such programs are modified to comply with this revision, an option to implement **present(p)** as **present(p[0])** for C pointers may be helpful to users.

A.3.2 Autoscoping

If an implementation implements autoscoping to automatically determine variables that are private to a compute region or to a loop, or to recognize reductions in a compute region or a loop, an option to print a message telling what variables were affected by the analysis would be helpful to users. An option to disable the autoscoping analysis would be helpful to promote program portability across implementations.