The OpenACC®
Application Programming Interface

Draft Specification - Technical Report 24-1

OpenACC Technical Committee

November 2024

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 writing parallel programs in C, C++, and Fortran that run identified regions in parallel on multicore CPUs or attached accelerators. The method described provides a model for parallel programming that is portable across operating systems and various types of multicore 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 parallel multicore and 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 a host and accelerator or to 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 for parallel execution, and similar performance-related details.

1.1 Scope

This OpenACC API document covers only user-directed parallel and accelerator programming, where the user specifies the regions of a program to be targeted for parallel execution. The remainder of the program will be executed sequentially 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 executed in parallel on a multicore CPU or an accelerator.

This document does not describe automatic detection of parallel regions or automatic offloading of regions of code to an accelerator by a compiler or other tool. This document does not describe splitting loops or code regions across multiple accelerators attached to a single host. While future compilers may allow for automatic parallelization or automatic offloading, or parallelizing across multiple accelerators of the same type, or across 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 parallel accelerator, such as a GPU, or a multicore host with a host thread that initiates parallel execution on the multiple cores, thus treating the multicore CPU itself as a device. Much of a user application executes on a host thread. Compute intensive regions are offloaded to an accelerator or executed on the multiple host cores under control of a host thread. A device, either an attached accelerator or the multicore CPU, executes *parallel regions*, which typically contain work-sharing loops, *kernels regions,* which typically contain one or more loops that may be executed as kernels, or *serial regions,* which are blocks of sequential code. Even in accelerator-targeted regions, the host thread 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 accelerator 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 a device, one after the other.

Most current accelerators and many multicore CPUs support two or three levels of parallelism. Most accelerators and multicore CPUs 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 and some CPUs 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 and CPUs also support SIMD or vector operations within each execution unit. The execution model exposes these multiple levels of parallelism on a 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. The gangs are organized in a one-, two-, or three-dimensional grid, where dimension one corresponds to the inner level of gang parallelism; the default is to only use dimension one. Worker parallelism is fine-grain. Each gang will have one or more workers. Vector parallelism is for SIMD or vector operations within a worker. In this way, OpenACC provides six levels of parallelism, which are arranged from highest to lowest as follows: gang dimension three, gang dimension two, gang dimension one, worker, vector, and sequential, which corresponds to no parallelism.

When executing a compute region on a 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. Each gang dimension is associated with a *gang-redundant* mode dimension, denoted GR1, GR2, and GR3. When the program reaches a loop or loop nest marked for gang-level work-sharing at some dimension, the program starts to execute in *gang-partitioned* mode for that dimension, denoted GP1, GP2, or GP3 mode, where the iterations of the loop or loops are partitioned across the gangs in that dimension for truly parallel execution, but still with only one worker per gang and one vector lane per worker active. The program may be simultaneously in different gang modes for different dimensions. For instance, after entering a loop partitioned for gang-level work-sharing at dimension 3, the program will be in GP3, GR2, GR1 mode.

When only one worker is active, in any gang-level execution 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 in dimension *d* and worker-partitioning, then the iterations of the loop are spread across all the workers of all the gangs of dimension *d*. 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 program starts executing with a single initial host thread, identified by a program counter and its stack. The initial host thread may spawn additional host threads, using OpenACC or another mechanism, such as with the OpenMP API. On a device, a single vector lane of a single worker of a single gang is called a device thread. When executing on an accelerator, a parallel execution context is created on the accelerator and may contain many such threads.

Attempting to implement barrier synchronization, critical sections, or locks across any of gang, worker, or vector parallelism might result in deadlock or non-portable code. 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.

Some devices, such as a multicore CPU, may also create and launch additional compute regions, allowing for nested parallelism. In that case, the OpenACC directives may be executed by a host thread or a device 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. The specification uses the term *local device* to mean the device on which the *local thread* is executing.

Most accelerators can operate asynchronously with respect to the host thread. Such devices have 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 an accelerator may be discrete 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 accelerator 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, the accelerator may not be able to read or write host memory; though this is supported by some accelerators, it may incur significant performance penalty.

The concept of discrete 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 discrete memories for many reasons, including but not limited to:

- Memory bandwidth between host memory and accelerator memory determines the level of compute intensity required to effectively accelerate a given region of code.
- Discrete accelerator memory is usually significantly smaller than the host memory, possibly prohibiting the offloading of regions of code that operate on very large amounts of data.
- Data in host memory may only be accessible on the host; data in accelerator memory may only be accessible on that accelerator. Explicitly transferring pointer values between host and accelerator memory is not advised. Dereferencing pointers to host memory on an accelerator or dereferencing pointers to accelerator memory on the host is likely to result in a runtime error or incorrect results on such targets.

OpenACC exposes the discrete 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 a device shares memory with the local thread, its 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 a device has a discrete memory and shares no memory with the local thread, the implementation will allocate space in device memory and copy data between the local memory and device memory, as appropriate. The local thread may share some memory with a device and also have some memory that is not shared with that device. In that case, data in shared memory may be accessed by both the local thread and the device. Data not in shared memory will be copied to device memory as necessary.

Some accelerators 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 a compute region that produces inconsistent numerical results.

Similarly, some accelerators implement a weak memory model for memory shared between the host and the accelerator, or memory shared between multiple accelerators. Programmers need to be very careful that the program uses appropriate synchronization to ensure that an assignment or modification by a thread on any device to data in shared memory is complete and available before that data is used by another thread on the same or another device.

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 Language Interoperability

The specification supports programs written using OpenACC in two or more of Fortran, C, and C++ languages. The parts of the program in any one base language will interoperate with the parts written in the other base languages as described here. In particular:

- Data made present in one base language on a device will be seen as present by any base language.
- A region that starts and ends in a procedure written in one base language may directly or indirectly call procedures written in any base language. The execution of those procedures are part of the region.

1.5 Runtime Errors

Common runtime errors are noted in this document. When one of these runtime errors is issued, one or more error callback routines are called by the program. Error conditions are noted throughout Chapter 2 Directives and Chapter 3 Runtime Library along with the error code that gets set for the error callback.

A list of error codes appears in Section 5.2.2. Since device actions may occur asynchronously, some errors may occur asynchronously as well. In such cases, the error callback routines may not be called immediately when the error occurs, but at some point later when the error is detected during program execution. In situations when more than one error may occur or has occurred, any one of the errors may be issued and different implementations may issue different errors. An **`acc_error_system`** error may be issued at any time if the current device becomes unavailable due to underlying system issues.

The default error callback routine may print an error message and halt program execution. The application can register one or more additional error callback routines, to allow a failing application to release resources or to cleanly shut down a large parallel runtime with many threads and processes. See Chapter 5 Profiling and Error Callback Interface. The error callback mechanism is not intended for error recovery. There is no support for restarting or retrying an OpenACC program, construct, or API routine after an error condition has been detected and an error callback routine has been called.

1.6 Conventions used in this document

Some terms are used in this specification that conflict with their usage as defined in the base languages. When there is potential confusion, the term will appear in the Glossary.

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; an option that may be repeated more than once is followed by ellipses:

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

In this spec, a *var* (in italics) is one of the following:

- a variable name (a scalar, array, or composite variable name);
- a subarray specification with subscript ranges;
- an array element;
- a member of a composite variable;
- a common block name between slashes.

Not all options are allowed in all clauses; the allowable options are clarified for each use of the term *var*. Unnamed common blocks (blank commons) are not permitted and common blocks of the same name must be of the same size in all scoping units as required by the Fortran standard.

To simplify the specification and convey appropriate constraint information, a *pqr-list* is a comma-separated list of one or more *pqr* items. For example, an *int-expr-list* is a comma-separated list of one or more integer expressions, and a *var-list* is a comma-separated list of one or more *vars*. Elements of such a list must not be empty and must not be followed by a trailing comma. 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*

For C/C++, unless otherwise specified, each expression inside of the OpenACC clauses and directive arguments must be a valid *assignment-expression*. This avoids ambiguity between the comma operator and comma-separated list items.

In this spec, a *do loop* (in italics) is the **`do`** construct as defined by the Fortran standard. The *do-stmt* of the **`do`** construct must conform to one of the following forms:

do [label] do-var = lb, ub [, incr]

do concurrent [label] concurrent-header [concurrent-locality]

The *do-var* is a variable name and the *lb, ub, incr* are scalar integer expressions. A **`do concurrent`** is treated as if defining a loop for each index in the *concurrent-header*.

An italicized *true* is used for a condition that evaluates to nonzero in C or C++, or **`.true.`** in Fortran. An italicized *false* is used for a condition that evaluates to zero in C or C++, or **`.false.`** in Fortran.

Further details of OpenACC directive syntax are presented in Section 2.1.

1.7 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 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 runtime.

Chapter 5 Profiling and Error Callback Interface, describes the OpenACC interface for tools that can be used for profile and trace data collection.

Chapter 6 Glossary, defines common terms used in this document.

Appendix A Recommendations for Implementers, gives advice to implementers to support more portability across implementations and interoperability with other accelerator APIs.

1.8 References

Each language version inherits the limitations that remain in previous versions of the language in this list.

- *American National Standard Programming Language C*, ANSI X3.159-1989 (ANSI C).
- ISO/IEC 9899:1999, *Information Technology – Programming Languages – C*, (C99).
- ISO/IEC 9899:2011, *Information Technology – Programming Languages – C*, (C11).

  The use of the following C11 features may result in unspecified behavior.
  - Threads
  - Thread-local storage
  - Parallel memory model
  - Atomic
- ISO/IEC 9899:2018, *Information Technology – Programming Languages – C*, (C18).

  The use of the following C18 features may result in unspecified behavior.
  - Thread related features
- ISO/IEC 14882:1998, *Information Technology – Programming Languages – C++*.
- ISO/IEC 14882:2011, *Information Technology – Programming Languages – C++*, (C++11).

  The use of the following C++11 features may result in unspecified behavior.
  - Extern templates
  - copy and rethrow exceptions
  - memory model
  - atomics
  - move semantics
  - std::thread
  - thread-local storage
- ISO/IEC 14882:2014, *Information Technology – Programming Languages – C++*, (C++14).
- ISO/IEC 14882:2017, *Information Technology – Programming Languages – C++*, (C++17).

- ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran – Part 1: Base Language*, (Fortran 2003).
- ISO/IEC 1539-1:2010, *Information Technology – Programming Languages – Fortran – Part 1: Base Language*, (Fortran 2008).

  The use of the following Fortran 2008 features may result in unspecified behavior.

  - Coarrays
  - Simply contiguous arrays rank remapping to rank>1 target
  - Allocatable components of recursive type
  - Polymorphic assignment
- ISO/IEC 1539-1:2018, *Information Technology – Programming Languages – Fortran – Part 1: Base Language*, (Fortran 2018).

  The use of the following Fortran 2018 features may result in unspecified behavior.

  - Interoperability with C
    - C functions declared in ISO Fortran binding.h
    - Assumed rank
  - All additional parallel/coarray features
- *OpenMP Application Program Interface,* version 5.0, November 2018
- *NVIDIA CUDA™ C Programming Guide*, version 11.1.1, October 2020
- *The OpenCL Specification*, version 2.2, Khronos OpenCL Working Group, July 2019
- *INCITS INCLUSIVE TERMINOLOGY GUIDELINES*, version 2021.06.07, InterNational Committee for Information Technology Standards, June 2021
- *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119, IETF Network Working Group, March 1997

1.9 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 the **`async`** clause
- 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 appear 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
- clarified that no arguments are allowed on the **vector** clause in a parallel region

1.10 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

1.11 Changes from Version 2.0 to 2.5

- The **_OPENACC** value was updated to **201510**; see Section 2.2 Conditional Compilation.
- The **num_gangs**, **num_workers**, and **vector_length** clauses are now allowed on the **kernels** construct; see Section 2.5.3 Kernels Construct.
- Reduction on C++ class members, array elements, and struct elements are explicitly disallowed; see Section 2.5.15 reduction clause.
- Reference counting is now used to manage the correspondence and lifetime of device data; see Section 2.6.7 Reference Counters.
- The behavior of the **exit data** directive has changed to decrement the dynamic reference counter. A new optional **finalize** clause was added to set the dynamic reference counter to zero. See Section 2.6.6 Enter Data and Exit Data Directives.
- The **copy**, **copyin**, **copyout**, and **create** data clauses were changed to behave like **present_or_copy**, etc. The **present_or_copy**, **pcopy**, **present_or_copyin**, **pcopyin**, **present_or_copyout**, **pcopyout**, **present_or_create**, and **pcreate** data clauses are no longer needed, though will be accepted for compatibility; see Section 2.7 Data Clauses.
- Reductions on orphaned gang loops are explicitly disallowed; see Section 2.9 Loop Construct.
- The description of the **loop auto** clause has changed; see Section 2.9.7 auto clause.
- Text was added to the **private** clause on a **loop** construct to clarify that a copy is made for each gang or worker or vector lane, not each thread; see Section 2.9.10 private clause.
- The description of the **reduction** clause on a **loop** construct was corrected; see Section 2.9.11 reduction clause.
- A restriction was added to the **cache** clause that all references to that variable must lie within the region being cached; see Section 2.10 Cache Directive.
- Text was added to the **private** and **reduction** clauses on a combined construct to clarify that they act like **private** and **reduction** on the **loop**, not **private** and **reduction** on the **parallel** or **reduction** on the **kernels**; see Section 2.11 Combined Constructs.
- The **declare create** directive with a Fortran **allocatable** has new behavior; see Section 2.13.2 create clause.
- New **init**, **shutdown**, **set** directives were added; see Section 2.14.1 Init Directive, 2.14.2 Shutdown Directive, and 2.14.3 Set Directive.
- A new **if_present** clause was added to the **update** directive, which changes the behavior when data is not present from a runtime error to a no-op; see Section 2.14.4 Update Directive.

- The **routine bind** clause definition changed; see Section 2.15.1 Routine Directive.
- An **acc routine** without **gang/worker/vector/seq** is now defined as an error; see Section 2.15.1 Routine Directive.
- A new **default(present)** clause was added for compute constructs; see Section 2.5.16 default clause.
- The Fortran header file **openacc_lib.h** is no longer supported; see Section 3.1 Runtime Library Definitions.
- New API routines were added to get and set the default async queue value; see Section 3.2.13 acc_get_default_async and 3.2.14 acc_set_default_async.
- The **acc_copyin**, **acc_create**, **acc_copyout**, and **acc_delete** API routines were changed to behave like **acc_present_or_copyin**, etc. The **acc_present_or_** names are no longer needed, though will be supported for compatibility. See Sections 3.2.18 and following.
- Asynchronous versions of the data API routines were added; see Sections 3.2.18 and following.
- A new API routine added, **acc_memcpy_device**, to copy from one device address to another device address; see Section 3.2.26 acc_memcpy_to_device.
- A new OpenACC interface for profile and trace tools was added; see Chapter 5 Profiling and Error Callback Interface.

1.12 Changes from Version 2.5 to 2.6

- The **_OPENACC** value was updated to **201711**.
- A new **serial** compute construct was added. See Section 2.5.2 Serial Construct.
- A new runtime API query routine was added. **acc_get_property** may be called from the host and returns properties about any device. See Section 3.2.6.
- The text has clarified that if a variable is in a reduction which spans two or more nested loops, each **loop** directive on any of those loops must have a **reduction** clause that contains the variable; see Section 2.9.11 reduction clause.
- An optional **if** or **if_present** clause is now allowed on the **host_data** construct. See Section 2.8 Host_Data Construct.
- A new **no_create** data clause is now allowed on compute and **data** constructs. See Section 2.7.11 no_create clause.
- The behavior of Fortran optional arguments in data clauses and in routine calls has been specified; see Section 2.17.1 Optional Arguments.
- The descriptions of some of the Fortran versions of the runtime library routines were simplified; see Section 3.2 Runtime Library Routines.
- To allow for manual deep copy of data structures with pointers, new *attach* and *detach* behavior was added to the data clauses, new **attach** and **detach** clauses were added, and matching **acc_attach** and **acc_detach** runtime API routines were added; see Sections 2.6.4, 2.7.13-2.7.14 and 3.2.29.

- The Intel Coprocessor Offload Interface target and API routine sections were removed from the Section A Recommendations for Implementers, since Intel no longer produces this product.

1.13 Changes from Version 2.6 to 2.7

- The **_OPENACC** value was updated to **201811**.
- The specification allows for hosts that share some memory with the device but not all memory. The wording in the text now discusses whether local thread data is in shared memory (memory shared between the local thread and the device) or discrete memory (local thread memory that is not shared with the device), instead of shared-memory devices and non-shared memory devices. See Sections 1.3 Memory Model and 2.6 Data Environment.
- The text was clarified to allow an implementation that treats a multicore CPU as a device, either an additional device or the only device.
- The **readonly** modifier was added to the **copyin** data clause and **cache** directive. See Sections 2.7.8 and 2.10.
- The term *local device* was defined; see Section 1.2 Execution Model and the Glossary.
- The term *var* is used more consistently throughout the specification to mean a variable name, array name, subarray specification, array element, composite variable member, or Fortran common block name between slashes. Some uses of *var* allow only a subset of these options, and those limitations are given in those cases.
- The **self** clause was added to the compute constructs; see Section 2.5.7 self clause.
- The appearance of a **reduction** clause on a compute construct implies a **copy** clause for each reduction variable; see Sections 2.5.15 reduction clause and 2.11 Combined Constructs.
- The **default(none)** and **default(present)** clauses were added to the **data** construct; see Section 2.6.5 Data Construct.
- Data is defined to be *present* based on the values of the structured and dynamic reference counters; see Section 2.6.7 Reference Counters and the Glossary.
- The interaction of the **acc_map_data** and **acc_unmap_data** runtime API calls on the present counters is defined; see Section 2.7.2, 3.2.21, and 3.2.22.
- A restriction clarifying that a **host_data** construct must have at least one **use_device** clause was added.
- Arrays, subarrays and composite variables are now allowed in **reduction** clauses; see Sections 2.9.11 reduction clause and 2.5.15 reduction clause.
- Changed behavior of ICVs to support nested compute regions and host as a device semantics. See Section 2.3.

1.14 Changes from Version 2.7 to 3.0

- Updated **_OPENACC** value to **201911**.
- Updated the normative references to the most recent standards for all base languages. See Section 1.8.

- Changed the text to clarify uses and limitations of the **device_type** clause and added examples; see Section 2.4.
- Clarified the conflict between the implicit **copy** clause for variables in a **reduction** clause and the implicit **firstprivate** for scalar variables not in a data clause but used in a **parallel** or **serial** construct; see Sections 2.5.1 and 2.5.2.
- Required at least one data clause on a **data** construct, an **enter data** directive, or an **exit data** directive; see Sections 2.6.5 and 2.6.6.
- Added text describing how a C++ *lambda* invoked in a compute region and the variables captured by the *lambda* are handled; see Section 2.6.2.
- Added a **zero** modifier to **create** and **copyout** data clauses that zeros the device memory after it is allocated; see Sections 2.7.9 and 2.7.10.
- Added a new restriction on the **loop** directive allowing only one of the **seq**, **independent**, and **auto** clauses to appear; see Section 2.9.
- Added a new restriction on the **loop** directive disallowing a **gang**, **worker**, or **vector** clause to appear if a **seq** clause appears; see Section 2.9.
- Allowed variables to be modified in an atomic region in a loop where the iterations must otherwise be data independent, such as loops with a **loop independent** clause or a **loop** directive in a **parallel** construct; see Sections 2.9.2, 2.9.3, 2.9.4, and 2.9.6.
- Clarified the behavior of the **auto** and **independent** clauses on the **loop** directive; see Sections 2.9.7 and 2.9.6.
- Clarified that an orphaned **loop** construct, or a **loop** construct in a **parallel** construct with no **auto** or **seq** clauses is treated as if an **independent** clause appears; see Section 2.9.6.
- For a variable in a **reduction** clause, clarified when the update to the original variable is complete, and added examples; see Section 2.9.11.
- Clarified that a variable in an orphaned **reduction** clause must be private; see Section 2.9.11.
- Required at least one clause on a **declare** directive; see Section 2.13.
- Added an **if** clause to **init**, **shutdown**, **set**, and **wait** directives; see Sections 2.14.1, 2.14.2, 2.14.3, and 2.16.3.
- Required at least one clause on a **set** directive; see Section 2.14.3.
- Added a *devnum* modifier to the **wait** directive and clause to specify a device to which the wait operation applies; see Section 2.16.3.
- Allowed a **routine** directive to include a C++ lambda name or to appear before a C++ lambda definition, and defined implicit **routine** directive behavior when a C++ lambda is called in a compute region or an accelerator routine; see Section 2.15.
- Added runtime API routine **acc_memcpy_d2d** for copying data directly between two device arrays on the same or different devices; see Section 3.2.30.
- Defined the values for the **acc_construct_t** and **acc_device_api** enumerations for cross-implementation compatibility; see Sections 5.2.2 and 5.2.3.

- Changed the return type of `acc_set_cuda_stream` from `int` (values were not specified) to `void`; see Section A.2.1.
- Edited and expanded Section 1.19 Topics Deferred For a Future Revision.

1.15 Changes from Version 3.0 to 3.1

- Updated `_OPENACC` value to `202011`.
- Clarified that Fortran blank common blocks are not permitted and that same-named common blocks must have the same size. See Section 1.6.
- Clarified that a `parallel` construct's block is considered to start in gang-redundant mode even if there's just a single gang. See Section 2.5.1.
- Added support for the Fortran BLOCK construct. See Sections 2.5.1, 2.5.3, 2.6.1, 2.6.5, 2.8, 2.13, and 6.
- Defined the `serial` construct in terms of the `parallel` construct to improve readability. Instead of defining it in terms of clauses `num_gangs(1) num_workers(1) vector_length(1)`, defined the `serial` construct as executing with a single gang of a single worker with a vector length of one. See Section 2.5.2.
- Consolidated compute construct restrictions into a new section to improve readability. See Section 2.5.4.
- Clarified that a `default` clause may appear at most once on a compute construct. See Section 2.5.16.
- Consolidated discussions of implicit data attributes on compute and combined constructs into a separate section. Clarified the conditions under which each data attribute is implied. See Section 2.6.2.
- Added a restriction that certain loop reduction variables must have explicit data clauses on their parent compute constructs. This change addresses portability across existing OpenACC implementations. See Sections 2.6.2 and A.3.3.
- Restored the OpenACC 2.5 behavior of the `present`, `copy`, `copyin`, `copyout`, `create`, `no_create`, `delete` data clauses at exit from a region, or on an `exit data` directive, as applicable, and `create` clause at exit from an implicit data region where a `declare` directive appears, and `acc_copyout`, `acc_delete` routines, such that no action is taken if the appropriate reference counter is zero, instead of a runtime error being issued if data is not present. See Sections 2.7.6, 2.7.7, 2.7.8, 2.7.9, 2.7.10, 2.7.11, 2.7.12, 2.13.2, and 3.2.19.
- Clarified restrictions on loop forms that can be associated with `loop` constructs, including the case of C++ range-based `for` loops. See Section 2.9.
- Specified where `gang` clauses are implied on `loop` constructs. This change standardizes behavior of existing OpenACC implementations. See Section 2.9.2.
- Corrected C/C++ syntax for `atomic capture` with a structured block. See Section 2.12.
- Added the behavior of the Fortran *do concurrent* construct. See Section 2.17.2.

- Changed the Fortran run-time procedures: **acc_device_property** has been renamed to **acc_device_property_kind** and **acc_get_property** uses a different integer kind for the result. See Section 3.2.
- Added or changed argument names for the Runtime Library routines to be descriptive and consistent. This mostly impacts Fortran programs, which can pass arguments by name. See Section 3.2.
- Replaced composite variable by aggregate variable in **reduction**, **default**, and **private** clauses and in implicitly determined data attributes; the new wording also includes Fortran character and allocatable/pointer variables. See glossary in Section 6.

1.16 Changes from Version 3.1 to 3.2

- Updated **_OPENACC** value to **202111**.
- Modified specification to comply with INCITS standard for inclusive terminology.
- The text was changed to state that certain runtime errors, when detected, result in a call to the current runtime error callback routines. See Section 1.5.
- An ambiguity issue with the C/C++ comma operator was resolved. See Section 1.6.
- The terms *true* and *false* were defined and used throughout to shorten the descriptions. See Section 1.6.
- Implicitly determined data attributes on compute constructs were clarified. See Section 2.6.2.
- Clarified that the **default(none)** clause applies to scalar variables. See Section 2.6.2.
- The **async**, **wait**, and **device_type** clauses may be specified on **data** constructs. See Section 2.6.5.
- The behavior of data clauses and data API routines with a null pointer in the clause or as a routine argument is defined. See Sections 2.7.6-2.7.12, 2.8.1, and 3.2.16-3.2.30.
- Precision issues with the loop trip count calculation were clarified. See Section 2.9.
- Text in Section 2.16 was moved and reorganized to improve clarity and reduce redundancy.
- Some runtime routine descriptions were expanded and clarified. See Section 3.2.
- The **acc_init_device** and **acc_shutdown_device** routines were added to initialize and shut down individual devices. See Section 3.2.7 and Section 3.2.8.
- Some runtime routine sections were reorganized and combined into a single section to simplify maintenance and reduce redundant text:
  - The sections for four **acc_async_test** routines were combined into a single section. See Section 3.2.9.
  - The sections for four **acc_wait** routines were combined into a single section. See Section 3.2.10.
  - The sections for four **acc_wait_async** routines were combined into a single section. See Section 3.2.11.

  - The two sections for **acc_copyin** and **acc_create** were combined into a single section. See Section 3.2.18.
  - The two sections for **acc_copyout** and **acc_delete** were combined into a single section. See Section 3.2.19.
  - The two sections for **acc_update_self** and **acc_update_device** were combined into a single section. See Section 3.2.20.
  - The two sections for **acc_attach** and **acc_detach** were combined into a single section. See Section 3.2.29.
- Added runtime API routine **acc_wait_any**. See section 3.2.12.
- The descriptions of the **async** and **async_queue** fields of **acc_callback_info** were clarified. See Section 5.2.1.

1.17 Changes from Version 3.2 to 3.3

- Updated **_OPENACC** value to **202211**.
- Allowed three dimensions of gang parallelism:
  - Defined multiple levels of *gang-redundant* and *gang-partitioned* execution modes. See Section 1.2
  - Allowed multiple values in the **num_gangs** clauses on the **parallel** construct. See Section 2.5.10.
  - Allowed a **dim** argument to the **gang** clause on the **loop** construct. See Section 2.9.2.
  - Allowed a **dim** argument to the **gang** clause on the **routine** directive. See Section 2.15.1.
  - Changed the launch event information to include all three gang dimension sizes. See Section 5.2.2.
- Clarified user-visible behavior of evaluation of expressions in clause arguments. See Section 2.1.
- Added the **force** modifier to the **collapse** clause on loops to enable collapsing non-tightly nested loops. See Section 2.9.1.
- Generalized implicit **routine** directives for all procedures instead of just C++ lambdas. See Section 2.15.1.
- Revised Section 2.15.1 for clarity and conciseness, including:
  - Specified predetermined **routine** directives that the implementation may apply.
  - Clarified where **routine** directives must appear relative to definitions or uses of their associated procedures in C and C++. This clarification includes the case of forward references in C++ class member lists.
  - Clarified to which procedure a **routine** directive with a name applies in C and C++.
  - Clarified how a **nohost** clause affects a procedure's use within a compute region.

- Added a Fortran interface for the following runtime routines (See Chapter 3):
  - **acc_malloc**
  - **acc_free**
  - **acc_map_data**
  - **acc_unmap_data**
  - **acc_deviceptr**
  - **acc_hostptr**
  - The two **acc_memcpy_to_device** routines
  - The two **acc_memcpy_from_device** routines
  - The two **acc_memcpy_device** routines
  - The two **acc_attach** routines
  - The four **acc_detach** routines
- Added a new error condition for **acc_map_data** when the **bytes** argument is zero. See Section 3.2.21.
- Added recommendations for how a **routine** directive affects multicore host CPU compilation. See Section A.1.3.
- Recommended additional diagnostics promoting portable and readable OpenACC. See Section A.3.

1.18 Changes from Version 3.3 to TR 24-1

- Clarified that a *pqr-list* must have at least one item and is not permitted to have a trailing comma. See Section 1.6.
- Clarified that the **_Pragma** operator form is supported for OpenACC directives in C and C++. See Section 2.1.
- Clarified user-visible behavior of evaluation of expressions in directive arguments. See Section-2.1.
- Clarified the analysis of implicit data attributes and parallelism across the boundaries of procedures that can appear within other procedures (e.g., C++ lambdas, C++ class member functions, and Fortran internal procedures). See Sections 2.5, 2.6.2, 2.9, and 2.15.1.
- Restated data actions to improve data clause descriptions. See Section 2.7.2.
- Added the **capture** modifier for specifying that a particular variable requires a discrete copy in device-accessible memory, even when already in shared memory. See Section 2.7.4, Section 2.7.9 and Section 2.7.10.
- Added the **always**, **alwaysin**, and **alwaysout** modifiers to the **copy**, **copyin**, and **copyout** data clauses. See Section 2.7.7, Section 2.7.8, and Section 2.7.9.
- Clarified that intrinsic assignment of *declare create* variable in Fortran will result in memory allocation and/or deallocation on the device if memory is allocated and/or deallocated on the host. See Section 2.7.10

- Clarified that compatibility of nested levels of parallelism can be validated at compile time. See Sections 2.9 and 2.15.1.
- Added the **if** clause to the **atomic** construct to enable conditional atomic operations based on the parallelism strategy employed. See Section 2.12.
- Clarified that in Fortran any **declare** directive with a **create** or **device_resident** clause referencing a variable with the *allocatable* or *pointer* attributes must be visible when the variable is allocated or deallocated. See Section 2.13.
- Specified that **routine** directives are implicitly determined for C++ lambdas such that **gang**, **worker**, **vector**, **seq**, and **nohost** clauses are selected based on their definitions. See Section 2.15.1.
- Clarified that a C++ lambda has an implicit **routine** directive with a **nohost** clause if an enclosing accelerator routine has a **nohost** clause even if the lambda is unused. This case might affect compilation of OpenACC programs during development. See Section 2.15.1.

1.19 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 send email to feedback@openacc.org. No promises are made or implied that all these items will be available in a future revision.

- Directives to define implicit *deep copy* behavior for pointer-based data structures.
- Defined behavior when data in data clauses on a directive are aliases of each other.
- Clarifying when data becomes *present* or *not present* on the device for **enter data** or **exit data** directives with an **async** clause.
- Clarifying the behavior of Fortran **pointer** variables in data clauses.
- Allowing Fortran **pointer** variables to appear in **deviceptr** clauses.
- Support for attaching C/C++ pointers that point to an address past the end of a memory region.
- Fully defined interaction 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 **shared** clause (or something similar) for the loop directive.
- Better support for multiple devices from a single thread, whether of the same type or of different types.
- An *auto* construct (by some name), to allow **kernels**-like auto-parallelization behavior inside **parallel** constructs or accelerator routines.
- A **begin declare** ... **end declare** construct that behaves like putting any global variables declared inside the construct in a **declare** clause.
- Defining the behavior of additional parallelism constructs in the base languages when used inside a compute construct or accelerator routine.

- Optimization directives or clauses, such as an *unroll* directive or clause.
- Extended reductions.
- Fortran bindings for all the API routines.
- A **`linear`** clause for the **`loop`** directive.
- Allowing two or more of **`gang`**, **`worker`**, **`vector`**, or **`seq`** clause on an **`acc routine`** directive.
- A single list of all devices of all types, including the host device.
- A memory allocation API for specific types of memory, including device memory, host pinned memory, and unified memory.
- Allowing non-contiguous Fortran array sections as arguments to some Runtime API routines, such as **`acc_update_device`**.
- Bindings to other languages.
- Allowing capture modifier on unstructured data lifetimes.

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.

2.1 Directive Format

In C and C++, an OpenACC directive is specified as either a **#pragma** directive:

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

or a **_Pragma** operator:

_Pragma("acc *directive-name* [*clause-list*]**")**

While any OpenACC directive can be specified equivalently in either form, the convention in this document is to show only the **#pragma** form. The first preprocessing token within either form is **acc**. The remainder of the directive follows the C and C++ conventions for pragmas. Whitespace may be used before and after the **#**; whitespace may be required to separate words in a directive. Preprocessing tokens following **acc** are subject to macro replacement. Directives are case-sensitive.

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 whitespace (spaces and tabs). The sentinel (**!$acc**) must appear as a single word, with no intervening whitespace. Line length, whitespace, and continuation rules apply to the directive line. Initial directive lines must have whitespace 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 whitespace) and may have an ampersand as the first non-whitespace 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, whitespace, 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 appear 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. A program must not depend on the order of evaluation of expressions in clause, construct, or directive arguments, or on any side effects of the evaluations. (See examples below.) Clauses may be repeated unless otherwise specified.

Further details of OpenACC directive syntax are presented in Section 1.6.

Examples

- In the following example, the order and number of evaluations of **++i** and calls to **foo()** and **bar()** are unspecified.

  ```
  #pragma acc parallel \
    num_gangs(foo(++i)) \
    num_workers(bar(++i)) \
    async(foo(++i))
  { ... }
  ```

  See Section 2.5.1 for the **parallel** construct.

- In the following example, if the implementation knows that **array** is not present in the current device memory, it may omit calling **size()**.

  ```
  #pragma acc update \
    device(array[0:size()])
    if_present
  ```

  See Section 2.14.4 for the **update** directive.

- In the following example, execution and order of the constructor and destructor of **S** and **U** is not guaranteed.

  ```
  #pragma acc wait(devnum:S{}.Value:queues:acc_async_sync) \
  if (U{}.Condition)
  ```

  See Section 2.16.3 for the **wait** directive.

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 202211.

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-current-device-type-var* - controls which type of device is used.
- *acc-current-device-num-var* - controls which device of the selected type is used.
- *acc-default-async-var* - controls which asynchronous queue is used when none appears in an async clause.

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-current-device-type-var	**`acc_set_device_type`** **`set device_type`** **`init device_type`** **`ACC_DEVICE_TYPE`**	**`acc_get_device_type`**
acc-current-device-num-var	**`acc_set_device_num`** **`set device_num`** **`init device_num`** **`ACC_DEVICE_NUM`**	**`acc_get_device_num`**
acc-default-async-var	**`acc_set_default_async`** **`set default_async`**	**`acc_get_default_async`**

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. There is one copy of each ICV for each host thread that is not generated by a compute construct. For threads that are generated by a compute construct the initial value for each ICV is inherited from the local thread. The behavior for each ICV is as if there is a copy for each thread. If an ICV is modified, then a unique copy of that ICV must be created for the modifying thread.

2.4 Device-Specific Clauses

OpenACC directives can specify different clauses or clause arguments for different devices using the **`device_type`** clause. Clauses that precede any **`device_type`** clause are *default clauses*. Clauses that follow a **`device_type`** clause up to the end of the directive or up to the next **`device_type`** clause are *device-specific clauses* for the device types specified in the **`device_type`** argument. For each directive, only certain clauses may be device-specific clauses. If a directive has at least one device-specific clause, it is *device-dependent*, and otherwise it is *device-independent*.

The argument to the **`device_type`** clause is a comma-separated list of one or more device architecture name identifiers, or an asterisk. An asterisk indicates all device types that are not named in any other **device_type** clause on that directive. A single directive may have one or several **device_type** clauses. The **device_type** clauses may appear in any order.

Except where otherwise noted, the rest of this document describes device-independent directives, on which all clauses apply when compiling for any device type. When compiling a device-dependent directive for a particular device type, the directive is treated as if the only clauses that appear are (a) the clauses specific to that device type and (b) all default clauses for which there are no like-named clauses specific to that device type. If, for any device type, the resulting directive is nonconforming, then the original directive is nonconforming.

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

A device architecture name may be generic, such as a vendor, or more specific, such as a particular generation of device; see Appendix A Recommendations for Implementers for recommended names. When compiling for a particular device, the implementation will use the clauses associated with the **device_type** clause that specifies the 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**.

Examples

- On the following directive, **worker** appears as a device-specific clause for devices of type **foo**, but **gang** appears as a default clause and so applies to all device types, including **foo**.

```
#pragma acc loop gang device_type(foo) worker
```

- The first directive below is identical to the previous directive except that **loop** is replaced with **routine**. Unlike **loop**, **routine** does not permit **gang** to appear with **worker**, but both apply for device type **foo**, so the directive is nonconforming. The second directive below is conforming because **gang** there applies to all device types except **foo**.

```
// nonconforming: gang and worker not permitted together
#pragma acc routine gang device_type(foo) worker

// conforming: gang and worker for different device types
#pragma acc routine device_type(foo) worker \
                    device_type(*)   gang
```

- On the directive below, the value of **num_gangs** is **4** for device type **foo**, but it is **2** for all other device types, including **bar**. That is, **foo** has a device-specific **num_gangs** clause, so the default **num_gangs** clause does not apply to **foo**.

```
!$acc parallel                      num_gangs(2)   &
!$acc          device_type(foo) num_gangs(4)   &
!$acc          device_type(bar) num_workers(8)
```

- The directive below is the same as the previous directive except that **num_gangs(2)** has moved after **device_type(*)** and so now does not apply to **foo** or **bar**.

```
!$acc parallel device_type(*)   num_gangs(2)   &
!$acc          device_type(foo) num_gangs(4)   &
!$acc          device_type(bar) num_workers(8)
```

▲ ▲

2.5 Compute Constructs

Compute constructs indicate code that is intended to be executed on the current device. It is implementation defined how users specify for which accelerators that code is compiled and whether it is also compiled for the host.

For any point in the program, the *parent procedure* is the nearest lexically enclosing procedure such that expressions at this point are not evaluated until the procedure is called. For example, the parent procedure within the capture specification of a C++ lambda is the procedure in which the lambda is defined, but the parent procedure within the lambda's body is the lambda itself.

For any point in the program, the *parent compute construct* is the nearest lexically enclosing compute construct that has the same parent procedure.

For any point in the program, the *parent compute scope* is the parent compute construct or, if none, the parent procedure.

2.5.1 Parallel Construct

Summary

This fundamental construct starts parallel execution on the current device.

Syntax

In C and C++, the syntax of the OpenACC **parallel** construct 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

or

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

where *clause* is one of the following:

```
async [ ( int-expr ) ]
wait [ ( int-expr-list ) ]
num_gangs( int-expr-list )
num_workers( int-expr )
vector_length( int-expr )
device_type( device-type-list )
if( condition )
self [ ( condition ) ]
reduction( operator : var-list )
copy( [ modifier-list : ] var-list )
copyin( [ modifier-list : ] var-list )
copyout( [ modifier-list : ] var-list )
create( [ modifier-list : ] var-list )
no_create( var-list )
present( var-list )
deviceptr( var-list )
attach( var-list )
private( var-list )
firstprivate( var-list )
default( none | present )
```

Description

When the program encounters an accelerator **parallel** construct, one or more gangs of workers are created to execute the accelerator parallel region. The number of gangs, and the number of workers in each 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 even if there is only a single gang. This means that code within the parallel region, but outside of a loop construct with gang-level worksharing, will be executed redundantly by all gangs.

One worker in each gang begins executing the code in the structured block of the construct. **Note:** Unless there is a **loop** construct within the parallel region, all gangs will execute all the code within the region redundantly.

If the **async** clause does not appear, 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.

The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach** data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate** clauses are described in Sections 2.5.13 and Sections 2.5.14. The **device_type** clause is described in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described in Section 2.6.2. Restrictions are described in Section 2.5.4.

2.5.2 Serial Construct

Summary

This construct defines a region of the program that is to be executed sequentially on the current device. The behavior of the **serial** construct is the same as that of the **parallel** construct except that it always executes with a single gang of a single worker with a vector length of one. **Note:** The **serial** construct may be used to execute sequential code on the current device, which removes the need for data movement when the required data is already present on the device.

Syntax

In C and C++, the syntax of the OpenACC **serial** construct is

```
#pragma acc serial [clause-list] new-line
    structured block
```

and in Fortran, the syntax is

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

or

```
!$acc serial [ clause-list ]
    block construct
[!$acc end serial]
```

where *clause* is as for the **parallel** construct except that the **num_gangs**, **num_workers**, and **vector_length** clauses are not permitted.

2.5.3 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 current device.

Syntax

In C and C++, the syntax of the OpenACC **kernels** construct 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
```

or

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

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* **)**
self [**(** *condition* **)**]
copy([*modifier-list* :] *var-list* **)**
copyin([*modifier-list* :] *var-list* **)**
copyout([*modifier-list* :] *var-list* **)**
create([*modifier-list* :] *var-list* **)**
no_create(*var-list* **)**
present(*var-list* **)**
deviceptr(*var-list* **)**
attach(*var-list* **)**
default(none | present)

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 does not appear, there is an implicit barrier at the end of the kernels region, and the local thread execution will not proceed until the entire sequence of kernels has completed execution.

The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach** data clauses are described in Section 2.7 Data Clauses. The **device_type** clause is described in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described in Section 2.6.2. Restrictions are described in Section 2.5.4.

2.5.4 Compute Construct Restrictions

The following restrictions apply to all compute constructs:

- A program may not branch into or out of a compute construct.
- 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.
- At most one **default** clause may appear, and it must have a value of either **none** or **present**.
- A **reduction** clause may not appear on a **parallel** construct with a **num_gangs** clause that has more than one argument.

2.5.5 Compute Construct Errors

- An `acc_error_wrong_device_type` error is issued if the compute construct was not compiled for the current device type. This includes the case when the current device is the host multicore.
- An `acc_error_device_type_unavailable` error is issued if no device of the current device type is available.
- An `acc_error_device_unavailable` error is issued if the current device is not available.
- An `acc_error_device_init` error is issued if the current device cannot be initialized.
- An `acc_error_execution` error is issued if the execution of the compute construct on the current device type fails and the failure can be detected.
- Explicit or implicitly determined data attributes can cause an error to be issued; see Section 2.7.3.
- An `async` or `wait` clause can cause an error to be issued; see Sections 2.16.1 and 2.16.2.

See Section 5.2.2.

2.5.6 if clause

The `if` clause is optional.

When the *condition* in the `if` clause evaluates to *true*., the region will execute on the current device. When the *condition* in the `if` clause evaluates to *false*, the local thread will execute the region.

2.5.7 self clause

The `self` clause is optional.

The `self` clause may have a single *condition-argument*. If the *condition-argument* is not present it is assumed to evaluate to *true*. When both an `if` clause and a `self` clause appear and the *condition* in the `if` clause evaluates to *false*, the `self` clause has no effect.

When the *condition* evaluates to *true*, the region will execute on the local device. When the *condition* in the `self` clause evaluates to *false*, the region will execute on the current device.

2.5.8 async clause

The `async` clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2.5.9 wait clause

The `wait` clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2.5.10 num_gangs clause

The `num_gangs` clause is allowed on the `parallel` and `kernels` constructs. On a `parallel` construct, it may have one, two, or three arguments. The values of the integer expressions define the number of parallel gangs along dimensions one, two, and three that will execute the parallel region. If it has fewer than three arguments, the missing values are treated as having the value 1. The total number of gangs must be at least 1 and is the product of the values of the arguments. On a **kernels** construct, the **num_gangs** clause must have a single argument, the value of which will define the number of parallel gangs that will execute each kernel created for the kernels region.

If the **num_gangs** clause does not appear, an implementation-defined default will be used which may depend on the code within the construct. The implementation may use a lower value than specified based on limitations imposed by the target architecture.

2.5.11 num_workers clause

The **num_workers** clause is allowed on the **parallel** and **kernels** constructs. 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 does not appear, an implementation-defined default will be used; the default value may be 1, and may be different for each **parallel** construct or for each kernel created for a **kernels** construct. The implementation may use a different value than specified based on limitations imposed by the target architecture.

2.5.12 vector_length clause

The **vector_length** clause is allowed on the **parallel** and **kernels** constructs. 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 does not appear, an implementation-defined default will be used. This vector length will be used for loop constructs annotated with the **vector** clause, as well as loops automatically vectorized by the compiler. The implementation may use a different value than specified based on limitations imposed by the target architecture.

2.5.13 private clause

The **private** clause is allowed on the **parallel** and **serial** constructs; it declares that a copy of each item on the list will be created for each gang in all dimensions.

Restrictions

- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private** clauses.

2.5.14 firstprivate clause

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

Restrictions

- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **firstprivate** clauses.

2.5.15 reduction clause

The **`reduction`** clause is allowed on the **`parallel`** and **`serial`** constructs. It specifies a reduction operator and one or more *vars*. It implies **`copy`** clauses as described in Section 2.6.2. For each reduction *var*, 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 *var* and stored in the original *var*. If the reduction *var* is an array or subarray, the array reduction operation is logically equivalent to applying that reduction operation to each element of the array or subarray individually. If the reduction *var* is a composite variable, the reduction operation is logically equivalent to applying that reduction operation to each member of the composite variable individually. 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 data type of the *var*. For **`max`** and **`min`** reductions, the initialization values are the least representable value and the largest representable value for that data type, respectively. At a minimum, the supported data types include Fortran **`logical`** as well as the numerical data types in C (e.g., **`_Bool`**, **`char`**, **`int`**, **`float`**, **`double`**, **`float _Complex`**, **`double _Complex`**), C++ (e.g., **`bool`**, **`char`**, **`wchar_t`**, **`int`**, **`float`**, **`double`**), and Fortran (e.g., **`integer`**, **`real`**, **`double precision`**, **`complex`**). However, for each reduction operator, the supported data types include only the types permitted as operands to the corresponding operator in the base language where (1) for max and min, the corresponding operator is less-than and (2) for other operators, the operands and the result are the same type.

<table>
<tr><th colspan="2">C and C++</th><th colspan="2">Fortran</th></tr>
<tr><th>operator</th><th>initialization value</th><th>operator</th><th>initialization value</th></tr>
<tr><td><code>+</code></td><td><code>0</code></td><td><code>+</code></td><td><code>0</code></td></tr>
<tr><td><code>*</code></td><td><code>1</code></td><td><code>*</code></td><td><code>1</code></td></tr>
<tr><td><code>max</code></td><td>least</td><td><code>max</code></td><td>least</td></tr>
<tr><td><code>min</code></td><td>largest</td><td><code>min</code></td><td>largest</td></tr>
<tr><td><code>&</code></td><td><code>~0</code></td><td><code>iand</code></td><td>all bits on</td></tr>
<tr><td><code>|</code></td><td><code>0</code></td><td><code>ior</code></td><td><code>0</code></td></tr>
<tr><td><code>^</code></td><td><code>0</code></td><td><code>ieor</code></td><td><code>0</code></td></tr>
<tr><td><code>&&</code></td><td><code>1</code></td><td><code>.and.</code></td><td><code>.true.</code></td></tr>
<tr><td><code>||</code></td><td><code>0</code></td><td><code>.or.</code></td><td><code>.false.</code></td></tr>
<tr><td></td><td></td><td><code>.eqv.</code></td><td><code>.true.</code></td></tr>
<tr><td></td><td></td><td><code>.neqv.</code></td><td><code>.false.</code></td></tr>
</table>

Restrictions

- A *var* in a **`reduction`** clause must be a scalar variable name, an aggregate variable name, an array element, or a subarray (refer to Section 2.7.1).
- If the reduction *var* is an array element or a subarray, accessing the elements of the array outside the specified index range results in unspecified behavior.
- The reduction *var* may not be a member of a composite variable.
- If the reduction *var* is a composite variable, each member of the composite variable must be a supported datatype for the reduction operation.

- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **`reduction`** clauses.

2.5.16 default clause

The **`default`** clause is optional. At most one **`default`** clause may appear. It adjusts what data attributes are implicitly determined for variables used in the compute construct as described in Section 2.6.2.

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. See Section A.3.3 for recommended diagnostics related to data attributes.

OpenACC supports systems with accelerators that have discrete memory from the host, systems with accelerators that share memory with the host, as well as systems where an accelerator shares some memory with the host but also has some discrete memory that is not shared with the host. In the first case, no data is in shared memory. In the second case, all data is in shared memory. In the third case, some data may be in shared memory and some data may be in discrete memory, although a single array or aggregate data structure must be allocated completely in shared or discrete memory. 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.

Memory is considered *shared memory* if data residing in that memory is accessible from both the host and the current device. Memory is considered *device memory* if it is physically connected to the current device. Memory is considered *device-accessible* if it is accessible from the current device, regardless of where the physical memory resides. A *captured variable* is a variable which the user has specific must have a *device-accessible* copy that is discrete from the original, even if the original is in *shared memory*.

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 compute construct are predetermined to be private to the thread that executes the loop.

Variables declared in a C block or Fortran block construct that is executed in *vector-partitioned* mode are private to the thread associated with each vector lane. Variables declared in a C block or Fortran block construct that is executed in *worker-partitioned vector-single* mode are private to the worker and shared across the threads associated with the vector lanes of that worker. Variables declared in a C block or Fortran block construct that is executed in *worker-single* mode are private to the gang and shared across the threads associated with the workers and vector lanes of that gang.

A procedure called from a compute construct will be annotated as **`seq`**, **`vector`**, **`worker`**, or

gang, as described Section 2.15 Procedure Calls in Compute Regions. Variables declared in **seq** routine are private to the thread that made the call. Variables declared in **vector** routine are private to the worker that made the call and shared across the threads associated with the vector lanes of that worker. Variables declared in **worker** or **gang** routine are private to the gang that made the call and shared across the threads associated with the workers and vector lanes of that gang.

2.6.2 Variables with Implicitly Determined Data Attributes

When implicitly determining data attributes on a compute construct, the following clauses are visible and variable accesses are exposed to the compute construct:

- *Visible* ***default*** *clause*: The nearest **default** clause appearing on the compute construct or on a lexically enclosing **data** construct that has the same parent compute scope.
- *Visible data clause*: Any data clause on the compute construct, on a lexically enclosing **data** construct that has the same parent compute scope, or on a visible **declare** directive.
- *Exposed variable access*: Any access to the data or address of a variable at a point within the compute construct where the variable is not private to a scope lexically enclosed within the compute construct.

  Note: In the argument of C's **sizeof** operator, the appearance of a variable is not an exposed access because neither its data nor its address is accessed. In the argument of a **reduction** clause on an enclosed **loop** construct, the appearance of a variable that is not otherwise privatized is an exposed access to the original variable.

On a compute or combined construct, if a variable appears in a **reduction** clause but no other data clause, it is treated as if it also appears in a **copy** clause. Otherwise, for any variable, the compiler will implicitly determine its data attribute on a compute construct if all of the following conditions are met:

- There is no **default(none)** clause visible at the compute construct.
- An access to the variable is exposed to the compute construct.
- The variable does not appear in a data clause visible at the compute construct.

An aggregate variable will be treated as if it appears either:

- In a **present** clause if there is a **default(present)** clause visible at the compute construct.
- In a **copy** clause otherwise.

A scalar variable will be treated as if it appears either:

- In a **copy** clause if the compute construct is a **kernels** construct.
- In a **firstprivate** clause otherwise.

Note: Any **default(none)** clause visible at the compute construct applies to both aggregate and scalar variables. However, any **default(present)** clause visible at the compute construct applies only to aggregate variables.

Restrictions

- If there is a **default(none)** clause visible at a compute construct, for any variable access exposed to the compute construct, the compiler requires the variable to appear either in an explicit data clause visible at the compute construct or in a **firstprivate**, **private**, or **reduction** clause on the compute construct.
- If a scalar variable appears in a **reduction** clause on a **loop** construct that has a parent **parallel** or **serial** construct, and if the reduction's access to the original variable is exposed to the parent compute construct, the variable must appear either in an explicit data clause visible at the compute construct or in a **firstprivate**, **private**, or **reduction** clause on the compute construct. **Note:** Implementations are encouraged to issue a compile-time diagnostic when this restriction is violated to assist users in writing portable OpenACC applications.

If a C++ *lambda* is called in a compute region and does not appear in a data clause, then it is treated as if it appears in a **copyin** clause on the current construct. A variable captured by a *lambda* is processed according to its data types: a pointer type variable is treated as if it appears in a **no_create** clause; a reference type variable is treated as if it appears in a **present** clause; for a struct or a class type variable, any pointer member is treated as if it appears in a **no_create** clause on the current construct. If the variable is defined as global or file or function static, it must appear in a **declare** directive.

2.6.3 Data Regions and Data Lifetimes

Data in shared memory is accessible from the current device as well as to the local thread. Such data is available to the accelerator for the lifetime of the variable. Data not in shared memory must be copied to and from device memory using data constructs, clauses, and API routines. A *data lifetime* is the duration from when the data is first made available to the accelerator until it becomes unavailable. For data in shared memory, the data lifetime begins when the data is allocated and ends when it is deallocated; for statically allocated data, the data lifetime begins when the program begins and does not end. For data not in shared memory, the data lifetime begins when it is made present and ends when it is no longer present.

There are four types of data regions. When the program encounters a **data** construct, it creates a 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 duration of the compute construct.

When the program enters a procedure, it creates an implicit data region that has a duration 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. There is also an implicit data region associated with the execution of the program itself. The implicit program data region has a duration of the execution of the program.

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, each *var* on the directive or the variable on the runtime API argument list will be made live on accelerator.

2.6.4 Data Structures with Pointers

This section describes the behavior of data structures that contain pointers. A pointer may be a C or C++ pointer (e.g., **float***), a Fortran pointer or array pointer (e.g., **real, pointer, dimension(:)**), or a Fortran allocatable (e.g., **real, allocatable, dimension(:)**).

When a data object is copied to device memory, the values are copied exactly. If the data is a data structure that includes a pointer, or is just a pointer, the pointer value copied to device memory will be the host pointer value. If the pointer target object is also allocated in or copied to device memory, the pointer itself needs to be updated with the device address of the target object before dereferencing the pointer in device memory.

An *attach* action updates the pointer in device memory to point to the device copy of the data that the host pointer targets; see Section 2.7.2. For Fortran array pointers and allocatable arrays, this includes copying any associated descriptor (dope vector) to the device copy of the pointer. When the device pointer target is deallocated, the pointer in device memory is restored to the host value, so it can be safely copied back to host memory. A *detach* action updates the pointer in device memory to have the same value as the corresponding pointer in local memory; see Section 2.7.2. The *attach* and *detach* actions are performed by the **copy**, **copyin**, **copyout**, **create**, **attach**, and **detach** data clauses (Sections 2.7.5-2.7.14), and the **acc_attach** and **acc_detach** runtime API routines (Section 3.2.29). The *attach* and *detach* actions use attachment counters to determine when the pointer in device memory needs to be updated; see Section 2.6.8.

2.6.5 Data Construct

Summary

The **data** construct defines *vars* are accessible to the current device for the duration of the region. It also defines the data actions that occur upon entry to and exit from the region.

Syntax

In C and C++, the syntax of the OpenACC **data** construct 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

or

    !$acc data [*clause-list*]
        block construct
    [**!$acc end data**]

where *clause* is one of the following:

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

copy ([*modifier-list* :] *var-list* **)**
copyin ([*modifier-list* :] *var-list* **)**
copyout ([*modifier-list* :] *var-list* **)**
create ([*modifier-list* :] *var-list* **)**
no_create (*var-list* **)**
present (*var-list* **)**
deviceptr (*var-list* **)**
attach (*var-list* **)**
default(none | present)

Description

Data will be allocated in the memory of the current device and copied from local memory to device memory, or copied back, as required. The data clauses are described in Section 2.7 Data Clauses. Structured reference counters are incremented for data when entering a data region, and decremented when leaving the region, as described in Section 2.6.7 Reference Counters. The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

Restrictions

- At least one **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, **attach**, or **default** clause must appear on a **data** construct.
- Only the **async** and **wait** clauses may follow a **device_type** clause.

if clause

The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate space in the current device memory and move data from and to the local memory as required. When an **if** clause appears, the program will conditionally allocate memory in and move data to and/or from device memory. When the *condition* in the **if** clause evaluates to *false*, no device memory will be allocated, and no data will be moved. When the *condition* evaluates to *true*, the data will be allocated and moved as specified. At most one **if** clause may appear.

async clause

The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

Note: The **async** clause only affects operations directly associated with this particular **data** construct, such as data transfers. Execution of the associated structured block or block construct remains synchronous to the local thread. Nested OpenACC constructs, directives, and calls to runtime library routines do not inherit the **async** clause from this construct, and the programmer must take care to not accidentally introduce race conditions related to asynchronous data transfers.

wait clause

The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

default clause

The **default** clause is optional. At most one **default** clause may appear. It adjusts what data attributes are implicitly determined for variables used in lexically contained compute constructs as described in Section 2.6.2.

Errors

- See Section 2.7.3 for errors due to data clauses.
- See Sections 2.16.1 and 2.16.2 for errors due to **async** or **wait** clauses.

2.6.6 Enter Data and Exit Data Directives

Summary

An **enter data** directive defines *vars* are accessible to the current device for the remaining duration of the program, or until an **exit data** directive makes the data no longer accessible. These directives also specify data actions which occur upon reaching the **enter data** or **exit data** directive. The dynamic data lifetime for data referred to by an **enter data** or **exit data** directive is defined by its dynamic reference counter, as defined in Section 2.6.7.

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 [(wait-argument)]
    copyin([modifier-list :] var-list)
    create([modifier-list :] var-list)
    attach(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 [(wait-argument)]
    copyout([modifier-list :] var-list)
    delete(var-list)
    detach(var-list)
    finalize

Description

At an **enter data** directive, data may be allocated in the current device memory and copied from local memory to device memory. This action enters a data lifetime for those *vars*, and will make the data available for **present** clauses on constructs within the data lifetime. Dynamic reference counters are incremented for this data, as described in Section 2.6.7 Reference Counters. Pointers in device memory may be *attached* to point to the corresponding device copy of the host pointer target.

At an **exit data** directive, data may be copied from device memory to local memory and deallocated from device memory. If no **finalize** clause appears, dynamic reference counters are decremented for this data. If a **finalize** clause appears, the dynamic reference counters are set to zero for this data. Pointers in device memory may be *detached* so as to have the same value as the original host pointer.

The data clauses are described in Section 2.7 Data Clauses. Reference counting behavior is described in Section 2.6.7 Reference Counters.

Restrictions

- At least one **copyin**, **create**, or **attach** clause must appear on an **enter data** directive.
- At least one **copyout**, **delete**, or **detach** clause must appear on an **exit data** directive.

if clause

The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or deallocate space in the current device memory and move data from and to local memory. When an **if** clause appears, the program will conditionally allocate or deallocate device memory and move data to and/or from device memory. When the *condition* in the **if** clause evaluates to *false*, no device memory will be allocated or deallocated, and no data will be moved. When the *condition* evaluates to *true*, the data will be allocated or deallocated and moved as specified.

async clause

The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

wait clause

The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

finalize clause

The **finalize** clause is allowed on the **exit data** directive and is optional. When no **finalize** clause appears, the **exit data** directive will decrement the dynamic reference counters for *vars* appearing in **copyout** and **delete** clauses, and will decrement the attachment counters for pointers appearing in **detach** clauses. If a **finalize** clause appears, the **exit data** directive will set the dynamic reference counters to zero for *vars* appearing in **copyout** and **delete** clauses, and will set the attachment counters to zero for pointers appearing in **detach** clauses.

Errors

- See Section 2.7.3 for errors due to data clauses.
- See Sections 2.16.1 and 2.16.2 for errors due to **async** or **wait** clauses.

2.6.7 Reference Counters

When device memory is allocated for data not in shared memory due to data clauses or OpenACC API routine calls, the OpenACC implementation keeps track of that section of device memory and its relationship to the corresponding data in host memory.

Each section of device memory is associated with two *reference counters* per device, a structured reference counter and a dynamic reference counter. The structured and dynamic reference counters are used to determine when to allocate or deallocate data in device memory. The structured reference counter for a section of memory keeps track of how many nested data regions have been entered for that data. The initial value of the structured reference counter for static data in device memory (in a global **`declare`** directive) is one; for all other data, the initial value is zero. The dynamic reference counter for a section of memory keeps track of how many dynamic data lifetimes are currently active in device memory for that section. The initial value of the dynamic reference counter is zero. Data is considered *present* if the sum of the structured and dynamic reference counters is greater than zero.

A structured reference counter is incremented when entering each data or compute region that contain an explicit data clause or implicitly-determined data attributes for that section of memory, and is decremented when exiting that region. A dynamic reference counter is incremented for each **`enter data copyin`** or **`create`** clause, or each **`acc_copyin`** or **`acc_create`** API routine call for that section of memory. The dynamic reference counter is decremented for each **`exit data copyout`** or **`delete`** clause when no **`finalize`** clause appears, or each **`acc_copyout`** or **`acc_delete`** API routine call for that section of memory. The dynamic reference counter will be set to zero with an **`exit data copyout`** or **`delete`** clause when a **`finalize`** clause appears, or each **`acc_copyout_finalize`** or **`acc_delete_finalize`** API routine call for the section of memory. The reference counters are modified synchronously with the local thread, even if the data directives include an **`async`** clause. When both structured and dynamic reference counters reach zero, the data lifetime in device memory for that data ends.

Memory mapped by **`acc_map_data`** may not have the associated dynamic reference count decremented to zero, except by a call to **`acc_unmap_data`**.

2.6.8 Attachment Counter

Since multiple pointers can target the same address, each pointer in device memory is associated with an *attachment counter* per device. The *attachment counter* for a pointer is initialized to zero when the pointer is allocated in device memory. The *attachment counter* for a pointer is set to one whenever the pointer is *attached* to new target address, and incremented whenever an *attach* action for that pointer is performed for the same target address. The *attachment counter* is decremented whenever a *detach* action occurs for the pointer, and the pointer is *detached* when the *attachment counter* reaches zero. This is described in more detail in Section 2.7.2 Data Clause Actions.

A pointer in device memory can be assigned a device address in two ways. The pointer can be attached to a device address due to data clauses or API routines, as described in Section 2.7.2 Data Clause Actions, or the pointer can be assigned in a compute region executed on that device. Unspecified behavior may result if both ways are used for the same pointer.

Pointer members of structs, classes, or derived types in device or host memory can be overwritten due to update directives or API routines. It is the user's responsibility to ensure that the pointers have the appropriate values before or after the data movement in either direction. The behavior of the program is undefined if any of the pointer members are attached when an update of a composite variable is performed.

2.7 Data Clauses

Data clauses may appear on the **parallel** construct, **serial** construct, **kernels** construct, **data** construct, the **enter data** and **exit data** directives, and **declare** directives. In the descriptions, the *region* is a compute region with a clause appearing on a **parallel**, **serial**, or **kernels** construct, a data region with a clause on a **data** construct, or an implicit data region with a clause on a **declare** directive. If the **declare** directive appears in a global context, the corresponding implicit data region has a duration of the program. The list argument to each data clause is a comma-separated collection of *vars*. On a **declare** directive, the list argument of a **copyin**, **create**, **device_resident**, or **link** clause may include a Fortran *common block* name enclosed within slashes. On any directive, for any clause 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 *var* in the memory of the current device, creating a visible device copy of that *var*, for data not in shared memory.

OpenACC supports accelerators with discrete 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 unless an explicit copy in device-accessible memory is specified. 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.
- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in data clauses.

2.7.1 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 elements **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 extent to preserve the contiguous data restriction, discussed below. For dynamically allocated dimensions, the implementation will allocate pointers in device memory corresponding to the pointers in local memory 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 extent, 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 appears in a data clause, the implementation may choose to allocate memory for only that subarray on the accelerator.
- In Fortran, array pointers may appear, but pointer association is not preserved in device memory.
- Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous section of memory, except for dynamic multidimensional C arrays.
- In C and C++, if a variable or array of composite type appears, all the data members of the struct or class are allocated and copied, as appropriate. If a composite member is a pointer type, the data addressed by that pointer are not implicitly copied.
- In Fortran, if a variable or array of composite type appears, 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.

2.7.2 Data Clause Actions

Data clauses perform one or more the following actions.

Increment Counter Action

An *increment counter* action is one of the actions that may be performed for a `present` (Section 2.7.6), `copy` (Section 2.7.7), `copyin` (Section 2.7.8), `copyout` (Section 2.7.9), `create` (Section 2.7.10), `no_create` (Section 2.7.11), or `attach` (Section 2.7.13) clause, or for a call to an `acc_copyin`, `acc_create`, or `acc_attach` API routine (Sections 3.2.18 and 3.2.29). See those sections for details.

An *increment counter* action for a *var* increments the structured or dynamic reference counter or the attachment counter for *var* by one.

Decrement Counter Action

A *decrement counter* action is one of the actions that may be performed for a `present` (Section 2.7.6), `copy` (Section 2.7.7), `copyin` (Section 2.7.8), `copyout` (Section 2.7.9), `create` (Section 2.7.10), `no_create` (Section 2.7.11), `delete` (Section 2.7.12), `attach` (Section 2.7.13), or `detach` clause, or for a call to an `acc_copyout`, `acc_delete`, or `acc_detach` API routine (Sections 3.2.19 and **??**). See those sections for details.

A *decrement counter* action for a *var* decrements the structured or dynamic reference counter or the attachment counter for *var* by one. If the reference counter is already zero, its value is left unchanged.

If the device memory associated with *var* was mapped to the device using `acc_map_data`, the dynamic reference count may not be decremented to zero, except by a call to `acc_unmap_data`.

Reset Counter Action

A *reset counter* action is one of the actions that may be performed for a `copyout` (Section 2.7.9), `delete` (Section 2.7.12), or `detach` (Section 2.7.14) clause, or for a call to an `acc_copyout`, `acc_delete`, or `acc_detach` API routine (Sections 3.2.19 and 3.2.29). See those sections for details.

A *reset counter* action for a *var* sets the structured or dynamic reference counter or attachment counter for *var* to zero.

Allocate Memory Action

An *allocate memory* action is one of the actions that may be performed for a `copy` (Section 2.7.7), `copyin` (Section 2.7.8), `copyout` (Section 2.7.9) or `create` (Section 2.7.10) clause, or for a call to an `acc_copyin` or `acc_create` API routine (Section 3.2.18). See those sections for details.

An *allocate memory* action for a *var* allocates device-accessible memory for *var*. If device memory is unavailable, shared memory is allocated. If shared memory is unavailable, device memory is allocated. When both shared and device memory are available, the choice of memory allocated is implementation-defined.

Deallocate Memory Action

A *deallocate memory* action is one of the actions that may be performed for a **copy** (Section 2.7.8), **copyin** (Section 2.7.8), **copyout** (Section 2.7.8), **create** (Section 2.7.10), **no_create** (Section 2.7.11), or **delete** (Section 2.7.12) clause, or for a call to an **acc_copyout** or **acc_delete** API routine (Section 3.2.19). See those sections for details.

A *deallocate memory* action for *var* deallocates device-accessible memory for *var*.

Transfer In Action

A *transfer in* action is one of the actions that may be performed for a **copy** (Section 2.7.7) or **copyin** (Section 2.7.8) clause, **update** (Section 2.14.4) directive, or for a call to an **acc_copyin** or **acc_update_device** API routine (Sections 3.2.18 and 3.2.20). See those sections for details.

A *transfer in* action for a *var* initiates a transfer of the data for *var* from the local thread memory to the corresponding device-accessible memory.

The data copy may occur asynchronously, depending on other clauses on the directive.

Transfer Out Action

A *transfer out* action is one of the actions that may be performed for a **copy** (Section 2.7.7) or **copyout** (Section 2.7.9) clause, **update** (Section 2.14.4) directive, or for a call to an **acc_copyout** or **acc_update_self** API routine (Sections 3.2.19 and 3.2.20). See those sections for details.

A *transfer out* action for a *var* initiates a transfer of the data for *var* from device-accesible memory to the corresponding local thread memory.

The data copy may occur asynchronously, depending on other clauses on the directive, in which case the memory is deallocated when the data copy is complete.

Attach Pointer Action

An *attach pointer* action is one of the actions that may be performed for a **present** (Section 2.7.6), **copy** (Section 2.7.7), **copyin** (Section 2.7.8), **copyout** (Section 2.7.9), **create** (Section 2.7.10), **no_create** (Section 2.7.11), or **attach** (Section 2.7.12) clause, or for a call to an **acc_attach** API routine (Section 3.2.29). See those sections for details.

An *attach pointer* action for a *var* occurs only when *var* is a pointer reference.

If the pointer *var* is in shared memory and it is not a captured variable or is not present in the current device-accessible memory, or if the address to which *var* points is not present in the current device-accessible memory, no action is taken. If the pointer is a null pointer, the pointer in device-accessible memory is updated to have the same value. Otherwise, the pointer in device-accessible memory is updated to point to the corresponding copy of the data. The update may occur asynchronously, depending on other clauses on the directive. The implementation schedules pointer updates after any data transfers due to *transfer in* actions that are performed for the same directive.

Detach Pointer Action

A *detach pointer* action is one of the actions that may be performed for a **present** (Section 2.7.6), **copy** (Section 2.7.7), **copyin** (Section 2.7.8), **copyout** (Section 2.7.9), **create** (Section 2.7.10), **no_create** (Section 2.7.11), **delete** (Section 2.7.12), or **attach** (Section 2.7.13), or **detach** (Section 2.7.12) clause, or for a call to an **acc_detach** API routine (Section 3.2.29). See those sections for details.

A *detach pointer* action for a *var* occurs only when *var* is a pointer reference.

If the pointer *var* is in shared memory and is not a captured variable or is not present in the current device-accessible memory, or if the *attachment counter* for *var* for the pointer is not zero, no action is taken. The *var* in device-accessible memory is updated to have the same value as the corresponding pointer in local memory. The update may occur asynchronously, depending on other clauses on the directive. The implementation schedules pointer updates before any data transfers due to *transfer out* actions that are performed for the same directive.

2.7.3 Data Clause Errors

An error is issued for a *var* that appears in a **copy**, **copyin**, **copyout**, **create**, and **delete** clause as follows:

- An **acc_error_partly_present** error is issued if part of *var* is present in device-accessible memory of the current device but all of *var* is not.
- An **acc_error_invalid_data_section** error is issued if *var* is a Fortran subarray with a stride that is not one.
- An **acc_error_out_of_memory** error is issued if the accelerator device does not have enough memory for *var*.

An error is issued for a *var* that appears in a **present** clause as follows:

- An **acc_error_not_present** error is issued if *var* is not present in the current device memory at entry to a data or compute construct.
- An **acc_error_partly_present** error is issued if part of *var* is present in device-accessible memory of the current device but all of *var* is not.

See Section 5.2.2.

2.7.4 Data Clause Modifiers

Some clauses allow an optional modifier list, with the following supported modifiers:

- **always** indicating that the data *transfer in* and *transfer out* actions must always occur even if the data is present in the device.
- **alwaysin** indicating that the data *transfer in* action must always occur even if the data is present in the device.
- **alwaysout** indicating that the data *transfer out* action must always occur even if the data is present in the device.

- **capture** indicating that the implementation must capture the variables in the clause with a distinct copy of such variables created in the device-accessible memory even if the original variable is already in accessible shared memory.
- **readonly** indicating that the data in the data region are only read and not written.
- **zero** indicating that the implementation must zero-initialise the variables in the clause.

2.7.5 deviceptr clause

The **deviceptr** clause may appear on structured **data** and compute constructs and **declare** directives.

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

In C and C++, the *vars* in *var-list* must be pointer variables.

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

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

2.7.6 present clause

The **present** clause may appear on structured **data** and compute constructs and **declare** directives. The **present** clause specifies that *vars* in *var-list* are in shared memory or are already present in the current device memory due to data regions or data lifetimes that contain the construct on which the **present** clause appears.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is taken; otherwise, the **present** clause behaves as follows:

- At entry to the region:
  1. If *var* is a pointer reference,
     a) If the attachment counter for *var* is zero, an *attach pointer* action is performed.
     b) An *increment counter* action is performed with the associated attachment counter.
  2. An *increment counter* action is performed with the associated structured reference counter.
- At exit from the region:
  1. If the structured reference counter for *var* is zero, no action is taken.
  2. Otherwise,
     a) If *var* is a pointer reference,
        i. A *decrement counter* action is performed with the associated attachment counter.
        ii. If the attachment counter for *var* is now zero, a *detach pointer* action is performed.

b) A *decrement counter* action is performed with the associate structured reference counter.

The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

2.7.7 copy clause

The **copy** clause may appear on structured **data** and compute constructs and on **declare** directives.

Only the following modifiers may appear in the optional *modifier-list*: *always*, *alwaysin*, *alwaysout* or *capture*.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable and has no **capture** modifier, no action is taken; otherwise, the **copy** clause behaves as follows:

- At entry to the region:
  1. If *var* is not present and is not a null pointer, an *allocate memory* action is performed. If a **zero** modifier appears, the memory is initialized to zero.
  2. If *var* is not present or if an **always** or **alwaysin** modifier appears, a *transfer in* action is performed.
  3. An *increment counter* action is performed with the associated structured reference counter.
  4. If *var* is a pointer reference, an *attach pointer* action is performed, followed by an *increment counter* action on the associated attachment counter.
- At exit from the region:
  - If the structured reference counter for *var* is zero, no action is taken.
  - Otherwise,
    1. If *var* is a pointer reference, a *decrement counter* action is performed with the associated attachment counter
    2. If the associated attachment counter is now zero, a *detach pointer* action is performed.
    3. A *decrement counter* action is performed with the structured associated reference counter.
    4. If both structured and dynamic reference counters are now zero or if an **always** or **alwaysout** modifier appears, a *transfer out* action is performed.
    5. If both structured and dynamic reference counters are now zero, a *deallocate memory* action is performed.

The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

For compatibility with OpenACC 2.0, **present_or_copy** and **pcopy** are alternate names for **copy**.

2.7.8 copyin clause

The **copyin** clause may appear on structured **data** and compute constructs, on **declare** directives, and on **enter data** directives.

Only the following modifiers may appear in the optional *modifier-list*: *always*, *alwaysin* or *readonly*. Additionally, on structured **data** and compute constructs *capture* modifier may appear.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable and has no **capture** modifier, no action is taken; otherwise, the **copyin** clause behaves as follows:

- At entry to a region, the structured reference counter is used. On an **enter data** directive, the dynamic reference counter is used.
    1. If *var* is not present and is not a null pointer, an *allocate memory* action is performed.
    2. If *var* is not present or if an **always** or **alwaysin** modifier appears, a *transfer in* action is performed.
    3. If *var* is a pointer reference, an *attach pointer* action is performed followed by an *increment counter* action with the associated attachment counter.
    4. An *increment counter* action is performed with the appropriate associated reference counter.
- At exit from the region:
    - If the structured reference counter for *var* is zero, no action is taken.
    - Otherwise,
        1. If *var* is a pointer reference, a *decrement counter* action is performed on the associated attachment counter.
        2. If *var* is a pointer reference and the associated attachment counter is now zero, a *detach pointer* action is performed.
        3. A *decrement counter* action is performed with the associated structured reference counter.
        4. If both structured and dynamic reference counters are now zero, a *deallocate memory* action is performed.

If the optional **readonly** modifier appears, then the implementation may assume that the data referenced by *var-list* is never written to within the applicable region.

The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

For compatibility with OpenACC 2.0, **present_or_copyin** and **pcopyin** are alternate names for **copyin**.

An **enter data** directive with a **copyin** clause is functionally equivalent to a call to the **acc_copyin** API routine, as described in Section 3.2.18.

2.7.9 copyout clause

The **copyout** clause may appear on structured **data** and compute constructs, on **declare** directives, and on **exit data** directives. The clause may optionally have a **zero** modifier if the **copyout** clause appears on a structured **data** or compute construct.

Only the following modifiers may appear in the optional *modifier-list*: *always*, *alwaysin* or *zero*. Additionally, on structured **data** and compute constructs *capture* modifier may appear.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable and has no **capture** modifier, no action is taken; otherwise, the **copyout** clause behaves as follows:

- At entry to a region:
    1. If *var* is not present and is not a null pointer, an *allocate memory* action is performed. If a **zero** modifier appears, the memory is initialized to zero.
    2. If *var* is a pointer reference, an *attach pointer* action is performed, followed by an *increment counter* action on the associated attachment counter.
    3. An *increment counter* action is performed with the associated structured reference counter.
- At exit from a region, the structured reference counter is used. On an **exit data** directive, the dynamic reference counter is used.
    - If the appropriate reference counter for *var* is zero, no action is taken.
    - Otherwise,
        1. If *var* is a pointer reference, a *decrement counter* action is performed on the associated attachment counter.
        2. If *var* is a pointer reference and the associated attachment counter is now zero, a *detach pointer* action is performed.
        3. The reference count is updated as follows:
            * On an **exit data** directive with a **finalize** clause, a *reset counter* action is performed to the dynamic reference.
            * Otherwise, a *decrement counter* action is performed with the appropriate associated reference counter.
        4. If both structured and dynamic reference counters are now zero or an **always** or **alwaysout** modifier appears, a *transfer out* action is performed.
        5. If both structured and dynamic reference counters are now zero, a *deallocate memory* action is performed.

The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

For compatibility with OpenACC 2.0, **present_or_copyout** and **pcopyout** are alternate names for **copyout**.

An **exit data** directive with a **copyout** clause and with or without a **finalize** clause is functionally equivalent to a call to the **acc_copyout_finalize** or **acc_copyout** API routine, respectively, as described in Section 3.2.19.

2.7.10 create clause

The **create** clause may appear on structured **data** and compute constructs, on **declare** directives, and on **enter data** directives.

Only the following modifiers may appear in the optional *modifier-list*: *zero*. Additionally, on structured **data** and compute constructs *capture* modifier may appear.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable and has no **capture** modifier, no action is taken; otherwise, the **create** clause behaves as follows:

- At entry to a region, the structured reference counter is used. On an **enter data** directive, the dynamic reference counter is used.
    1. If *var* is not present and is not a null pointer, an *allocate memory* action is performed. If a **zero** modifier appears, the memory is initialized to zero.
    2. If *var* is a pointer reference, an *attach pointer* action is performed, followed by an *increment counter* action on the associated attachment counter.
    3. An *increment counter* action is performed on the appropriate associated reference counter.
- At exit from the region:
    - If the structured reference counter for *var* is zero, no action is taken.
    - Otherwise,
        1. If *var* is a pointer reference, a *decrement counter* action is performed on the associated attachment counter.
        2. If *var* is a pointer reference and the associated attachment counter is now zero, a *detach pointer* action is performed.
        3. A *decrement counter* action is performed with the associated structured reference counter.
        4. If both structured and dynamic reference counters are zero, a *deallocate memory* action is performed.

The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

For compatibility with OpenACC 2.0, **present_or_create** and **pcreate** are alternate names for **create**.

An **enter data** directive with a **create** clause is functionally equivalent to a call to the **acc_create** API routine, as described in Section 3.2.18, except the directive may perform an *attach* action for a pointer reference.

2.7.11 no_create clause

The **no_create** clause may appear on structured **data** and compute constructs.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is taken; otherwise, the **no_create** clause behaves as follows:

- At entry to the region:

  - If *var* is present and is not a null pointer, an *increment counter* action is performed with the structured reference counter.
  - If *var* is present and is a pointer reference,
    1. an *increment counter* action is performed on the associated attachment counter,
    2. and if the associated attachment counter is now one, an *attach pointer* action is performed.
  - If *var* is not present, no action is performed, and any device code in this construct will use the local memory address for *var*.
- At exit from the region:
  - If the structured reference counter for *var* is zero or *var* is a null pointer, no action is taken.
  - Otherwise,
    1. If *var* is a pointer reference,
       a) a *decrement counter* action is performed on the associated attachment counter,
       b) and if the associated attachment counter is now zero, a *detach pointer* action is performed.
    2. A *decrement counter* action is performed with the structured reference counter.
    3. If both structured and dynamic reference counters are zero, a *deallocate memory* action is performed.

2.7.12 delete clause

The **`delete`** clause may appear on **`exit data`** directives.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is taken; otherwise, the **`delete`** clause behaves as follows:

- If the dynamic reference counter for *var* is zero, no action is taken.
- Otherwise,
  1. If *var* is a pointer reference,
     a) a *decrement counter* action is performed on the associated attachment counter,
     b) and if the associated attachment counter is now zero, a *detach pointer* action is performed.
  2. If *var* is not a null pointer, the dynamic reference counter is updated, as follows:
     - On an **`exit data`** directive with a **`finalize`** clause, a *reset counter* action is performed on the associated dynamic reference counter.
     - Otherwise, a *decrement counter* action is performed with the associated dynamic reference counter.

3. If both structured and dynamic reference counters are now zero, a *deallocate memory* action is performed.

An **`exit data`** directive with a **`delete`** clause and with or without a **`finalize`** clause is functionally equivalent to a call to the **`acc_delete_finalize`** or **`acc_delete`** API routine, respectively, as described in Section 3.2.19.

The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

2.7.13 attach clause

The **`attach`** clause may appear on structured **`data`** and compute constructs and on **`enter data`** directives. Each *var* argument to an **`attach`** clause must be a C or C++ pointer or a Fortran variable or array with the **`pointer`** or **`allocatable`** attribute.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is taken; otherwise, the **`attach`** clause behaves as follows:

- At entry to a region or at an **`enter data`** directive, an *attach pointer* action is performed followed by an *increment counter* action with the associated attachment counter.
- At exit from the region,
  1. a *decrement counter* action is performed with the associated attachment counter,
  2. and if the associated attachment counter is now zero, a *detach pointer* action is performed.

2.7.14 detach clause

The **`detach`** clause may appear on **`exit data`** directives. Each *var* argument to a **`detach`** clause must be a C or C++ pointer or a Fortran variable or array with the **`pointer`** or **`allocatable`** attribute.

For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is taken; otherwise, the **`detach`** clause behaves as follows:

- If there is a **`finalize`** clause on the **`exit data`** directive, a *reset counter* action with the attachment counter is performed. Otherwise, a *decrement counter* action is performed with the associated attachment counter.
- If the attachment counter is now zero, a *detach pointer* action is performed.

2.8 Host_Data Construct

Summary

The **`host_data`** construct makes the address of data in device-accessible memory available on the host.

Syntax

In C and C++, the syntax of the OpenACC **`host_data`** construct 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
```

or

```
!$acc host_data clause-list
    block construct
[!$acc end host_data]
```

where *clause* is one of the following:

```
use_device( var-list )
if( condition )
if_present
```

Description

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

Restrictions

- A *var* in a **use_device** clause must be the name of a variable or array.
- At least one **use_device** clause must appear.
- 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.
- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **use_device** clauses.

2.8.1 use_device clause

The **use_device** clause tells the compiler to use device-accessible memory address of any *var* in *var-list* in code within the construct. In particular, this may be used to pass the device address of *var* to optimized procedures written in a lower-level API. If *var* is a null pointer, the same value is used for the device address. Otherwise, when there is no **if_present** clause, and either there is no **if** clause or the condition in the **if** clause evaluates to *true*, the *var* in *var-list* must be present in device-accessible memory due to data regions or data lifetimes that contain this construct. For data in shared memory which is not a captured variable, the device address is the same as the host address.

2.8.2 if clause

The **if** clause is optional. When an **if** clause appears and the condition evaluates to *false*, the compiler will not replace the addresses of any *var* in code within the construct. When there is no **if** clause, or when an **if** clause appears and the condition evaluates to *true*, the compiler will replace the addresses as described in the previous subsection.

2.8.3 if_present clause

When an **`if_present`** clause appears on the directive, the compiler will only replace the address of any *var* which appears in *var-list* that is present in device-accessible memory for the current device.

2.9 Loop Construct

Summary

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

Syntax

In C and C++, the syntax of the **`loop`** construct is

`#pragma acc loop` [*clause-list*] *new-line*
  for loop

In Fortran, the syntax of the **`loop`** construct is

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

where *clause* is one of the following:

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

where *gang-arg* is one of:

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

and *gang-arg-list* may have at most one **`num`**, one **`dim`**, and one **`static`** argument, and where *size-expr* is one of:

`*`
int-expr

Some clauses are only valid in the context of a **`kernels`** construct; see the descriptions below.

An *orphaned* **`loop`** construct is a **`loop`** construct that has no parent compute construct.

A **loop** construct is *data-independent* if it has an **independent** clause that is determined explicitly, implicitly, or from an **auto** clause. A **loop** construct is *sequential* if it has a **seq** clause that is determined explicitly or from an **auto** clause.

When *do-loop* is a **do concurrent**, the OpenACC **loop** construct applies to the loop for each index in the *concurrent-header*. The **loop** construct can describe what type of parallelism to use to execute all the loops, and declares all indices appearing in the *concurrent-header* to be implicitly private. If the **loop** construct that is associated with **do concurrent** is combined with a compute construct then *concurrent-locality* is processed as follows: variables appearing in a *local* are treated as appearing in a **private** clause; variables appearing in a *local_init* are treated as appearing in a **firstprivate** clause; variables appearing in a *shared* are treated as appearing in a **copy** clause; and a *default(none)* locality spec implies a **default(none)** clause on the compute construct. If the **loop** construct is not combined with a compute construct, the behavior is implementation-defined.

Restrictions

- Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **independent**, **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.
- A loop associated with a **loop** construct that does not have a **seq** clause must be written to meet all of the following conditions:
  - The loop variable must be of integer, C/C++ pointer, or C++ random-access iterator type.
  - The loop variable must monotonically increase or decrease in the direction of its termination condition.
  - The loop trip count must be computable in constant time when entering the **loop** construct.

  For a C++ range-based **for** loop, the loop variable identified by the above conditions is the internal iterator, such as a pointer, that the compiler generates to iterate the range. It is not the variable declared by the **for** loop.
- Only one of the **seq**, **independent**, and **auto** clauses may appear.
- A **gang**, **worker**, or **vector** clause may not appear if a **seq** clause appears.
- A **loop** construct with a **gang**, **worker**, or **vector** clause must not lexically enclose another **loop** construct with a **gang**, **worker**, or **vector** clause specifying an equal or higher level of parallelism unless the **loop** constructs have different parent compute scopes. For example, in a loop nest that contains no interleaved compute constructs or procedures, a **gang(dim:1)** loop must not enclose a **gang(dim:3)** loop or be enclosed by a **worker** loop, but a **seq** loop is permitted at any nesting level.
- At most one **gang** clause may appear on a **loop** construct.
- A **tile** and **collapse** clause may not appear on **loop** that is associated with **do concurrent**.

2.9.1 collapse clause

The **collapse** clause is used to specify how many 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 appears, only the immediately following loop is associated with the **loop** construct.

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. The particular integer type used to compute the trip count for the collapsed loops is implementation defined. However, the integer type used for the trip count has at least the precision of each loop variable of the associated loops.

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

The associated loops are the *n* nested loops that immediately follow the loop construct. If the **force** modifier does not appear, then the associated loops must be tightly nested. If the **force** modifier appears, then any intervening code may be executed multiple times as needed to perform the collapse.

Restrictions

- Each associated loop, except the innermost, must contain exactly one loop or loop nest.
- Intervening code must not contain other OpenACC directives or calls to API routines.

Examples

- In the code below, a compiler may choose to move the call to **tan** inside the inner loop in order to collapse the two loops, resulting in redundant execution of the intervening code.

```
#pragma acc parallel loop collapse(force:2)
{
  for ( int i = 0; i < 360; i++ )
  {
    // This operation may be executed additional times in order
    // to perform the forced collapse.
    tanI = tan(a[i]);
    for ( int j = 0; j < N; j++ )
    {
      // Do Something.
    }
  }
}
```

2.9.2 gang clause

When the parent compute construct is a **`parallel`** construct, or on an orphaned **`loop`** construct, the **`gang`** clause behaves as follows. It specifies that the iterations of the associated loop or loops are to be executed in parallel by distributing the iterations among the gangs along the associated dimension created by the compute construct. The associated dimension is the value of the **`dim`** argument, if it appears, or is dimension one. The **`dim`** argument must be a constant positive integer with value 1, 2, or 3. If the associated dimension is *d*, a **`loop`** construct with the **`gang`** clause transitions a compute region from gang-redundant mode to gang-partitioned mode on dimension *d* (GR*d* to GP*d*). The number of gangs in dimension *d* is controlled by the **`parallel`** construct; the **`num`** argument is not allowed. The loop iterations must be data independent, except for *vars* which appear in a **`reduction`** clause or which are modified in an atomic region.

When the parent compute construct is a **`kernels`** construct, the **`gang`** clause behaves as follows. It specifies that the iterations of the associated loop or loops are to be executed in parallel across the gangs. The **`dim`** argument is not allowed. An argument with no keyword or with the **`num`** keyword is allowed only when the **`num_gangs`** does not appear on the **`kernels`** construct. If an argument with no keyword or an argument after the **`num`** keyword appears, it specifies how many gangs to use to execute the iterations of this loop.

The scheduling of loop iterations to gangs is not specified unless the **`static`** modifier appears as an argument. If the **`static`** modifier appears with an integer expression, that expression is used as a *chunk* size. If the static modifier 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.

A **`gang(dim:1)`** clause is implied on a data-independent **`loop`** construct without an explicit **`gang`** clause if the following conditions hold while ignoring **`gang`**, **`worker`**, and **`vector`** clauses on any sequential **`loop`** constructs and while treating implicit **`routine`** directives as if they are explicit:

- This **`loop`** construct's parent compute construct, if any, is not a **`kernels`** construct.
- An explicit **`gang(dim:1)`** clause would be permitted on this **`loop`** construct. For example, it must not conflict with a nested **`loop`** construct or an enclosing procedure's **`routine`** directive, as specified in Sections 2.9 and 2.15.1.
- For every lexically enclosing data-independent **`loop`** construct, either an explicit **`gang(dim:1)`** clause would not be permitted on the enclosing **`loop`** construct, or the **`loop`** constructs have different parent compute scopes.

Note: An important consequence of the above specification is that, before implicitly determining **`gang`** clauses on **`loop`** constructs, the implementation must analyze any **`auto`** clauses to determine if **`loop`** constructs are sequential, and it must determine relevant implicit **`routine`** directives (see the implicit **`gang`** clause example in Section 2.15.1).

Note: As a performance optimization, the implementation might select different levels of parallelism for a **`loop`** construct than specified by explicitly or implicitly determined clauses as long as it can prove program semantics are preserved. In particular, the implementation must consider semantic differences between gang-redundant and gang-partitioned mode. For example, in a series of tightly nested, data-independent **loop** constructs, implementations often move gang-partitioning from one **loop** construct to another without affecting semantics.

Note: If the **auto** or **device_type** clause appears on a **loop** construct, it is the programmer's responsibility to ensure that program semantics are the same regardless of whether the **auto** clause is treated as **independent** or **seq** and regardless of the device type for which the program is compiled. In particular, the programmer must consider the effect on both explicitly and implicitly determined **gang** clauses and thus on gang-redundant and gang-partitioned mode. Examples in Sections 2.9.11 and 2.15.1 demonstrate how this issue for the **auto** clause might affect portability across OpenACC implementations.

2.9.3 worker clause

When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct, 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 *vars* which appear in a **reduction** clause or which are modified in an atomic region.

When the parent compute construct is a **kernels** construct, the **worker** clause specifies that the iterations of the associated loop or loops are to be executed in parallel across the workers within a single gang. An argument is allowed only when the **num_workers** does not appear on the **kernels** construct. The optional argument specifies how many workers per gang to use to execute the iterations of this loop.

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

2.9.4 vector clause

When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct, 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 loop iterations must be data independent, except for *vars* which appear in a **reduction** clause or which are modified in an atomic region.

When the parent compute construct is a **kernels** construct, the **vector** clause specifies that the iterations of the associated loop or loops are to be executed with vector or SIMD processing. An argument is allowed only when the **vector_length** does not appear on the **kernels** construct. If an argument appears, the iterations will be processed in vector strips of that length; if no argument appears, the implementation will choose an appropriate vector length.

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

2.9.5 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.

2.9.6 independent clause

The **`independent`** clause tells the implementation that the loop iterations must be data independent, except for *vars* which appear in a **`reduction`** clause or which are modified in an atomic region. This allows the implementation to generate code to execute the iterations in parallel with no synchronization.

A **`loop`** construct with no **`auto`** or **`seq`** clause is treated as if it has the **`independent`** clause when it is an orphaned **`loop`** construct or its parent compute construct is a **`parallel`** construct.

Note

- It is likely a programming error to use the **`independent`** clause on a loop if any iteration writes to a variable or array element that any other iteration also writes or reads, except for *vars* which appear in a **`reduction`** clause or which are modified in an atomic region.
- The implementation may be restricted in the levels of parallelism it can apply by the presence of **`loop`** constructs with **`gang`**, **`worker`**, or **`vector`** clauses for outer or inner loops.

2.9.7 auto clause

The **`auto`** clause specifies that the implementation must analyze the loop and determine whether the loop iterations are data-independent. If it determines that the loop iterations are data-independent, the implementation must treat the **`auto`** clause as if it is an **`independent`** clause. If not, or if it is unable to make a determination, it must treat the **`auto`** clause as if it is a **`seq`** clause, and it must ignore any **`gang`**, **`worker`**, or **`vector`** clauses on the loop construct.

When the parent compute construct is a **`kernels`** construct, a **`loop`** construct with no **`independent`** or **`seq`** clause is treated as if it has the **`auto`** clause.

Note: Combining the **`auto`** and **`gang`** clauses might impact a program's portability across OpenACC implementations. See Section 2.9.2 for details.

2.9.8 tile clause

The **`tile`** clause specifies that the implementation will 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`** construct 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 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** construct, the **vector** clause is applied to the *element* loops. If the **gang** clause appears on the **loop** construct, the **gang** clause is applied to the *tile* loops. If the **worker** clause appears on the **loop** construct, the **worker** clause is applied to the *element* loops if no **vector** clause appears, and to the *tile* loops otherwise.

2.9.9 device_type clause

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

2.9.10 private clause

The **private** clause on a **loop** construct specifies that a copy of each item in *var-list* will be created. If the body of the loop is executed in *vector-partitioned* mode, a copy of the item is created for each thread associated with each vector lane. If the body of the loop is executed in *worker-partitioned vector-single* mode, a copy of the item is created for each worker and shared across the set of threads associated with all the vector lanes of that worker. Otherwise, a copy of the item is created for each gang in all dimensions and shared across the set of threads associated with all the vector lanes of all the workers of that gang.

Restrictions

- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private** clauses.

Examples

- In the example below, **tmp** is private to each worker of every gang but shared across all the vector lanes of a worker.

```
!$acc parallel
 !$acc loop gang
 do k = 1, n
  !$acc loop worker private(tmp)
  do j = 1, n
   !a single vector lane in each gang and worker assigns to tmp
   tmp = b(j,k) + c(j,k)
   !$acc loop vector
   do i = 1, n
    !all vector lanes use the result of the above update to tmp
    a(i,j,k) = a(i,j,k) + tmp/div
   enddo
  enddo
 enddo
!$acc end parallel
```

- In the example below, **tmp** is private to each gang in every dimension.

```
        !$acc parallel num_gangs(3,50,150)
         !$acc loop gang(dim:3)
         do k = 1, n
          !$acc loop gang(dim:2) private(tmp)
          do j = 1, n
           !all gangs along dimension 1 execute in gang redundant mode and
           !assign to tmp which is private to each gang in all dimensions
           tmp = b(j,k) + c(j,k)
           !$acc loop gang(dim:1)
           do i = 1, n
            a(i,j,k) = a(i,j,k) + tmp/div
           enddo
          enddo
         enddo
        !$acc end parallel
```

2.9.11 reduction clause

The **reduction** clause specifies a reduction operator and one or more *vars*. For each reduction *var*, a private copy is created in the same manner as for a **private** clause on the **loop** construct, and initialized for that operator; see the table in Section 2.5.15 reduction clause. After the loop, the values for each thread are combined using the specified reduction operator, and the result combined with the value of the original *var* and stored in the original *var*. If the original *var* is not private, this update occurs by the end of the compute region, and any access to the original *var* is undefined within the compute region. Otherwise, the update occurs at the end of the loop. If the reduction *var* is an array or subarray, the reduction operation is logically equivalent to applying that reduction operation to each array element of the array or subarray individually. If the reduction *var* is a composite variable, the reduction operation is logically equivalent to applying that reduction operation to each member of the composite variable individually.

If a variable is involved in a reduction that spans multiple nested loops where two or more of those loops have associated **loop** directives, a **reduction** clause containing that variable must appear on each of those **loop** directives.

Restrictions

- A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name, an array element, or a subarray (refer to Section 2.7.1).
- Reduction clauses on nested constructs for the same reduction *var* must have the same reduction operator.
- Every *var* in a **reduction** clause appearing on an orphaned **loop** construct must be private.
- The restrictions for a **reduction** clause on a compute construct listed in in Section 2.5.15 reduction clause also apply to a **reduction** clause on a **loop** construct.
- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **reduction** clauses.
- See Section 2.6.2 Variables with Implicitly Determined Data Attributes for a restriction requiring certain loop reduction variables to have explicit data clauses on their parent compute

constructs.

- A **reduction** clause may not appear on a **loop** directive that has a **gang** clause with a **dim:** argument whose value is greater than 1.
- A **reduction** clause may not appear on a **loop** directive that has a **gang** clause and is within a compute construct that has a **num_gangs** clause with more than one explicit argument.

Examples

- **x** is not private at the **loop** directive below, so its reduction normally updates **x** at the end of the parallel region, where gangs synchronize. When possible, the implementation might choose to partially update **x** at the loop exit instead, or fully if **num_gangs(1)** were added to the **parallel** directive. However, portable applications cannot rely on such early updates, so accesses to **x** are undefined within the parallel region outside the loop.

```
int x = 0;
#pragma acc parallel copy(x)
{
  // gang-shared x undefined
  #pragma acc loop gang worker vector reduction(+:x)
  for (int i = 0; i < I; ++i)
    x += 1; // vector-private x modified
  // gang-shared x undefined
} // gang-shared x updated for gang/worker/vector reduction
// x = I
```

- **x** is private at each of the innermost two **loop** directives below, so each of their reductions updates **x** at the loop's exit. However, **x** is not private at the outer **loop** directive, so its reduction updates **x** by the end of the parallel region instead.

```
int x = 0;
#pragma acc parallel copy(x)
{
  // gang-shared x undefined
  #pragma acc loop gang reduction(+:x)
  for (int i = 0; i < I; ++i) {
    #pragma acc loop worker reduction(+:x)
    for (int j = 0; j < J; ++j) {
      #pragma acc loop vector reduction(+:x)
      for (int k = 0; k < K; ++k) {
        x += 1; // vector-private x modified
      } // worker-private x updated for vector reduction
    } // gang-private x updated for worker reduction
  }
  // gang-shared x undefined
} // gang-shared x updated for gang reduction
// x = I * J * K
```

- At each **loop** directive below, **x** is private and **y** is not private due to the data clauses on the **parallel** directive. Thus, each reduction updates **x** at the loop exit, but each reduction updates **y** by the end of the parallel region instead.

```
int x = 0, y = 0;
#pragma acc parallel firstprivate(x) copy(y)
{
  // gang-private x = 0; gang-shared y undefined
  #pragma acc loop seq reduction(+:x,y)
  for (int i = 0; i < I; ++i) {
    x += 1; y += 2; // loop-private x and y modified
  } // gang-private x updated for trivial seq reduction
  // gang-private x = I; gang-shared y undefined
  #pragma acc loop worker reduction(+:x,y)
  for (int i = 0; i < I; ++i) {
    x += 1; y += 2; // worker-private x and y modified
  } // gang-private x updated for worker reduction
  // gang-private x = 2 * I; gang-shared y undefined
  #pragma acc loop vector reduction(+:x,y)
  for (int i = 0; i < I; ++i) {
    x += 1; y += 2; // vector-private x and y modified
  } // gang-private x updated for vector reduction
  // gang-private x = 3 * I; gang-shared y undefined
} // gang-shared y updated for gang/seq/worker/vector reductions
// x = 0; y = 3 * I * 2
```

- The examples below are equivalent. That is, the **reduction** clause on the combined construct applies to the **loop** construct but implies a **copy** clause on the parallel construct. Thus, **x** is not private at the **loop** directive, so the reduction updates **x** by the end of the parallel region.

```
int x = 0;
#pragma acc parallel loop worker reduction(+:x)
for (int i = 0; i < I; ++i) {
  x += 1; // worker-private x modified
} // gang-shared x updated for gang/worker reduction
// x = I

int x = 0;
#pragma acc parallel copy(x)
{
  // gang-shared x undefined
  #pragma acc loop worker reduction(+:x)
  for (int i = 0; i < I; ++i) {
    x += 1; // worker-private x modified
  }
  // gang-shared x undefined
} // gang-shared x updated for gang/worker reduction
// x = I
```

- If the implementation treats the **auto** clause below as **independent**, the loop executes in gang-partitioned mode and thus examines every element of **arr** once to compute **arr**'s maximum. However, if the implementation treats **auto** as **seq**, the gangs redundantly compute

arr's maximum, but the combined result is still **arr**'s maximum. Either way, because **x** is not private at the **loop** directive, the reduction updates **x** by the end of the parallel region.

```
int x = 0;
const int *arr = /*array of I values*/;
#pragma acc parallel copy(x)
{
  // gang-shared x undefined
  #pragma acc loop auto gang reduction(max:x)
  for (int i = 0; i < I; ++i) {
    // complex loop body
    x = x < arr[i] ? arr[i] : x; // gang- or loop-private
                                 // x modified
  }
  // gang-shared x undefined
} // gang-shared x updated for gang or gang/seq reduction
// x = arr maximum
```

- The following example is the same as the previous one except that the reduction operator is now **+**. While gang-partitioned mode sums the elements of **arr** once, gang-redundant mode sums them once per gang, producing a result many times **arr**'s sum. This example shows that, for some reduction operators, combining **auto**, **gang**, and **reduction** is typically non-portable.

```
int x = 0;
const int *arr = /*array of I values*/;
#pragma acc parallel copy(x)
{
  // gang-shared x undefined
  #pragma acc loop auto gang reduction(+:x)
  for (int i = 0; i < I; ++i) {
    // complex loop body
    x += arr[i]; // gang or loop-private x modified
  }
  // gang-shared x undefined
} // gang-shared x updated for gang or gang/seq reduction
// x = arr sum possibly times number of gangs
```

- At the following **loop** directive, **x** and **z** are private, so the loop reductions are not across gangs even though the loop is gang-partitioned. Nevertheless, the **reduction** clause on the **loop** directive is important as the loop is also vector-partitioned. These reductions are only partial reductions relative to the full set of values computed by the loop, so the **reduction** clause is needed on the **parallel** directive to reduce across gangs.

```
int x = 0, y = 0;
#pragma acc parallel copy(x) reduction(+:x,y)
{
  int z = 0;
  #pragma acc loop gang vector reduction(+:x,z)
  for (int i = 0; i < I; ++i) {
    x += 1; z += 2; // vector-private x and z modified
  } // gang-private x and z updated for vector reduction
  y += z; // gang-private y modified
} // gang-shared x and y updated for gang reduction
```

```
        // x = I; y = I * 2
```

▲ ▲

2.10 Cache Directive

Summary

The **cache** directive may appear at the top of (inside of) a loop. It suggests array elements or subarrays to 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([**readonly:**]*var-list* **)** *new-line*

In Fortran, the syntax of the **cache** directive is

!$acc cache([**readonly:**]*var-list* **)**

A *var* in a **cache** directive must be a single array element or a 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 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 variable plus or minus a constant or loop invariant; moreover the difference between the corresponding upper and lower bounds must be a constant.

If the optional **readonly** modifier appears, then the implementation may assume that the data referenced by any *var* in that directive is never written to within the applicable region.

Restrictions

- If an array element or subarray is listed in a **cache** directive, all references to that array during execution of that loop iteration must not refer to elements of the array outside the index range specified in the **cache** directive.
- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **cache** directives.

2.11 Combined Constructs

Summary

The combined OpenACC **parallel loop**, **serial loop**, and **kernels loop** constructs are shortcuts for specifying a **loop** construct nested immediately inside a **parallel**, **serial**, or

kernels construct. The meaning is identical to explicitly specifying a **parallel**, **serial**, or **kernels** construct containing a **loop** construct. Any clause that is allowed on a **parallel** or **loop** construct is allowed on the **parallel loop** construct; any clause allowed on a **serial** or **loop** construct is allowed on a **serial loop** construct; and any clause allowed on a **kernels** or **loop** construct is allowed on a **kernels loop** construct.

Syntax

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

```
#pragma acc parallel loop [clause-list] new-line
    for loop
```

In Fortran, the syntax of the **parallel loop** construct 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 **serial loop** construct is

```
#pragma acc serial loop [clause-list] new-line
    for loop
```

In Fortran, the syntax of the **serial loop** construct is

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

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

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

```
#pragma acc kernels loop [clause-list] new-line
    for loop
```

In Fortran, the syntax of the **kernels loop** construct 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.

A **private** or **reduction** clause on a combined construct is treated as if it appeared on the **loop** construct. In addition, a **reduction** clause on a combined construct implies a **copy** clause as described in Section 2.6.2.

Restrictions

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

2.12 Atomic Construct

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 is:

```
#pragma acc atomic [ atomic-clause ] [ if( condition ) ] new-line
    expression-stmt
```

or:

```
#pragma acc atomic capture [ if( condition ) ] 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 no clause appears:

```
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 is:

```
!$acc atomic read [ if( condition ) ]
    capture-statement
[!$acc end atomic]
```

or

```
!$acc atomic write [ if( condition ) ]
    write-statement
[!$acc end atomic]
```

or

```
!$acc atomic [update] [ if( condition ) ]
    update-statement
```

[**!$acc end atomic**]

or

!$acc atomic capture [**if (** *condition* **)**]
update-statement
capture-statement
!$acc end atomic

or

!$acc atomic capture [**if (** *condition* **)**]
capture-statement
update-statement
!$acc end atomic

or

!$acc atomic capture [**if (** *condition* **)**]
capture-statement
write-statement
!$acc end atomic

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

x = expr

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

v = x

and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**, or no clause appears):

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 appears, 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 with respect to the read or write of the location designated by **x**.

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.

The **if** clause specifies a condition where an atomic operation is required for correct parallel execution. If *condition* evaluates to *true* or no **if** clause appears, the atomic operation is required. If *condition* evaluates to *false*, the atomic directive can be safely ignored. **Note:** Conditional atomics are useful when different parallelism strategies are employed for different architectures; it is the programmer's responsibility to ensure that the atomic operation is safe to ignore if *condition* is *false*. Although not required, conditional atomics are recommended to be used with conditions that can be evaluated at compile-time, including the **acc_on_device** routine.

Restrictions

- 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.
- At most one **if** clause may appear.

2.13 Declare Directive

Summary

A **declare** directive is used in the declaration section of a Fortran subroutine, function, block construct, or module, or following a variable declaration in C or C++. It can specify that a *var* is to be allocated in 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 local memory to device memory upon entry to the implicit data region, and from device memory to local memory upon exit from the implicit data region. These directives create a visible device copy of the *var*.

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 ([**readonly:**]*var-list* **)**
copyout (*var-list* **)**
create (*var-list* **)**
present (*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, for a Fortran *common block*, or in a C or C++ global or namespace scope, the associated region is the implicit region for the whole program. The **copy**, **copyin**, **copyout**, **present**, and **deviceptr** data clauses are described in Section 2.7 Data Clauses.

Restrictions

- A **declare** directive must be in the same scope as the declaration of any *var* that appears in the clauses of the directive or any scope within a C or C++ function or Fortran function, subroutine, or program.
- At least one clause must appear on a **declare** directive.
- A *var* in a **declare** declare must be a variable or array name, or a Fortran *common block* name between slashes.
- A *var* may appear at most once in all the clauses of **declare** directives for a function, subroutine, program, or module.
- In Fortran, assumed-size dummy arrays may not appear in a **declare** directive.
- In Fortran, pointer arrays may appear, but pointer association is not preserved in device memory.
- In a Fortran *module* declaration section, only **create**, **copyin**, **device_resident**, and **link** clauses are allowed.
- In Fortran, any **create** or **device_resident** clause affecting a variable with the *allocatable* or *pointer* attribute must be visible at the allocation and deallocation of that variable.
- In C or C++ global or namespace 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.
- In C or C++, the **link** clause must appear at global or namespace 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.
- In C or C++, a **longjmp** call in the region must return to a **setjmp** call within the region.
- In C++, an exception thrown in the region must be handled within the region.
- See Section 2.17.1 Optional Arguments for discussion of Fortran optional dummy arguments in data clauses, including **device_resident** clauses.

2.13.1 device_resident clause

Summary

The **device_resident** clause specifies that the memory for the named variables is allocated in the current device memory and not in local memory. The host may not be able to access variables in a **device_resident** clause. The accelerator data lifetime of global variables or common blocks that appear 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 current device memory when the host thread executes an **allocate** or **deallocate** statement for that variable, if the current device is a non-shared memory device. If the variable has the Fortran *pointer* attribute, it may be allocated or deallocated by the host in the current 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 local memory. The *common block* will be available to accelerator routines; see Section 2.15 Procedure Calls in Compute Regions.

In a Fortran *module* declaration section, a *var* in a `device_resident` clause will be available to accelerator subprograms.

In C or C++ global scope, a *var* 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.

2.13.2 create clause

For data in shared memory, no action is taken.

For data not in shared memory, the `create` clause on a `declare` directive behaves as follows, for each *var* in *var-list*:

- At entry to an implicit data region where the `declare` directive appears:
  - If *var* is present, a *present increment* action with the structured reference counter is performed. If *var* is a pointer reference, an *attach* action is performed.
  - Otherwise, a *create* action with the structured reference counter is performed. If *var* is a pointer reference, an *attach* action is performed.
- At exit from an implicit data region where the `declare` directive appears:
  - If the structured reference counter for *var* is zero, no action is taken.
  - Otherwise, a *present decrement* action with the structured reference counter is performed. If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic reference counters are zero, a *delete* action is performed.

If the `declare` directive appears in a global context, then the data in *var-list* is statically allocated in device memory and the structured reference counter is set to one.

In Fortran, if a variable *var* in *var-list* has the Fortran *allocatable* or *pointer* attribute, then for a non-shared memory device:

- For an `allocate` statement for *var* or an intrinsic assignment statement of *var* that will allocate memory, memory will be allocated in both local memory as well as in the current device memory and the dynamic reference counter will be set to one.
- For a `deallocate` statement for *var* or an intrinsic assignment statement of *var* that will deallocate memory, memory will be deallocated from both local memory as well as the current device memory and the dynamic reference counter will be set to zero.
- In Fortran, an intrinsic assignment statement that reallocates *var* behaves the same as a deallocation followed by an allocation of *var*. **Note:** No update of device memory will occur as

the result of an intrinsic assignment statement on the host; if data coherency between the host and device is required, it is the user's responsibility.

- An **allocate**, **deallocate**, or intrinsic assignment statement on a device other than the host device will result in undefined behavior.
- If the structured reference counter is not zero, a runtime error is issued.

In Fortran, if a variable *var* in *var-list* has the Fortran *pointer* attribute, then it may appear on the left hand side of a pointer assignment statement, if the right hand side variable itself appears in a **create** clause.

Errors

- In Fortran, an **acc_error_present** error is issued at a deallocate statement if the structured reference counter is not zero.

See Section 5.2.2.

2.13.3 link clause

The **link** clause is used for large global host static data that is referenced within an accelerator routine and that has a dynamic data lifetime on the device. The **link** clause specifies that only a global link for the named variables is 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. 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 that appear 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.

2.14 Executable Directives

2.14.1 Init Directive

Summary

The **init** directive initializes the runtime for the given device or devices of the given device type. This can be used to isolate any initialization cost from the computational cost, when collecting performance statistics. If no device type appears all devices will be initialized. An **init** directive may be used in place of a call to the **acc_init** or **acc_init_device** runtime API routine, as described in Section 3.2.7.

Syntax

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

#pragma acc init [*clause-list*] *new-line*

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

!$acc init [*clause-list*]

where *clause* is one of the following:

device_type (*device-type-list* **)**
device_num (*int-expr* **)**
if(*condition* **)**

device_type clause

The **device_type** clause specifies the type of device that is to be initialized in the runtime. If the **device_type** clause appears, then the *acc-current-device-type-var* for the current thread is set to the argument value. If no **device_num** clause appears then all devices of this type are initialized.

device_num clause

The **device_num** clause specifies the device id to be initialized. If the **device_num** clause appears, then the *acc-current-device-num-var* for the current thread is set to the argument value. If no **device_type** clause appears, then the specified device id will be initialized for all available device types.

if clause

The **if** clause is optional; when there is no **if** clause, the implementation will generate code to perform the initialization unconditionally. When an **if** clause appears, the implementation will generate code to conditionally perform the initialization only when the *condition* evaluates to *true*.

Restrictions

- This directive may only appear in code executed on the host.
- If the directive is called more than once without an intervening **acc_shutdown** call or **shutdown** directive, 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, using this directive with a different device type may produce undefined behavior.

Errors

- An **acc_error_device_type_unavailable** error is issued if a **device_type** clause appears and no device of that device type is available, or if no **device_type** clause appears and no device of the current device type is available.
- An **acc_error_device_unavailable** error is issued if a **device_num** clause appears and the *int-expr* is not a valid device number or that device is not available, or if no **device_num** clause appears and the current device is not available.
- An **acc_error_device_init** error is issued if the device cannot be initialized.

See Section 5.2.2.

2.14.2 Shutdown Directive

Summary

The **shutdown** directive shuts down the connection to the given device or devices of the given device type, and frees any associated runtime resources. This ends all data lifetimes in device memory, which effectively sets structured and dynamic reference counters to zero. A **shutdown** directive may be used in place of a call to the **acc_shutdown** or **acc_shutdown_device** runtime API routine, as described in Section 3.2.8.

Syntax

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

> **#pragma acc shutdown** [*clause-list*] *new-line*

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

> **!$acc shutdown** [*clause-list*]

where *clause* is one of the following:

> **device_type (** *device-type-list* **)**
> **device_num (** *int-expr* **)**
> **if (** *condition* **)**

device_type clause

The **device_type** clause specifies the type of device that is to be disconnected from the runtime. If no **device_num** clause appears then all devices of this type are disconnected.

device_num clause

The **device_num** clause specifies the device id to be disconnected.

If no clauses appear then all available devices will be disconnected.

if clause

The **if** clause is optional; when there is no **if** clause, the implementation will generate code to perform the shutdown unconditionally. When an **if** clause appears, the implementation will generate code to conditionally perform the shutdown only when the *condition* evaluates to *true*.

Restrictions

- This directive may only appear in code executed on the host.

Errors

- An **acc_error_device_type_unavailable** error is issued if a **device_type** clause appears and no device of that device type is available,
- An **acc_error_device_unavailable** error is issued if a **device_num** clause appears and the *int-expr* is not a valid device number or that device is not available.
- An **acc_error_device_shutdown** error is issued if there is an error shutting down the device.

See Section 5.2.2.

2.14.3 Set Directive

Summary

The **set** directive provides a means to modify internal control variables using directives. Each form of the **set** directive is functionally equivalent to a matching runtime API routine.

Syntax

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

#pragma acc set [*clause-list*] *new-line*

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

!$acc set [*clause-list*]

where *clause* is one of the following

default_async (*int-expr* **)**
device_num (*int-expr* **)**
device_type (*device-type-list* **)**
if (*condition* **)**

default_async clause

The **default_async** clause specifies the asynchronous queue that is used if no queue appears and changes the value of *acc-default-async-var* for the current thread to the argument value. If the value is **acc_async_default**, the value of *acc-default-async-var* will revert to the initial value, which is implementation-defined. A **set default_async** directive is functionally equivalent to a call to the **acc_set_default_async** runtime API routine, as described in Section 3.2.14.

device_num clause

The **device_num** clause specifies the device number to set as the default device for accelerator regions and changes the value of *acc-current-device-num-var* for the current thread to the argument value. If the value of **device_num** argument is negative, the runtime will revert to the default behavior, which is implementation-defined. A **set device_num** directive is functionally equivalent to the **acc_set_device_num** runtime API routine, as described in Section 3.2.4.

device_type clause

The **device_type** clause specifies the device type to set as the default device type for accelerator regions and sets the value of *acc-current-device-type-var* for the current thread to the argument value. If the value of the **device_type** argument is zero or the clause does not appear, the selected device number will be used for all attached accelerator types. A **set device_type** directive is functionally equivalent to a call to the **acc_set_device_type** runtime API routine, as described in Section 3.2.2.

if clause

The **if** clause is optional; when there is no **if** clause, the implementation will generate code to perform the set operation unconditionally. When an **if** clause appears, the implementation will generate code to conditionally perform the set operation only when the *condition* evaluates to *true*.

Restrictions

- This directive may only appear in code executed on the host.
- Passing **default_async** the value of **acc_async_noval** has no effect.
- Passing **default_async** the value of **acc_async_sync** will cause all asynchronous directives in the default asynchronous queue to become synchronous.
- Passing **default_async** the value of **acc_async_default** will restore the default asynchronous queue to the initial value, which is implementation-defined.
- At least one **default_async**, **device_num**, or **device_type** clause must appear.
- Two instances of the same clause may not appear on the same directive.

Errors

- An **acc_error_device_type_unavailable** error is issued if a **device_type** clause appears, and no device of that device type is available.
- An **acc_error_device_unavailable** error is issued if a **device_num** clause appears, and the *int-expr* is not a valid device number.
- An **acc_error_invalid_async** error is issued if a **default_async** clause appears, and the *int-expr* is not a valid *async-argument*.

See Section 5.2.2.

2.14.4 Update Directive

Summary

The **update** directive is used during the lifetime of accelerator data to update *vars* in local memory with values from the corresponding data in device-accessible memory, or to update *vars* in device-accessible memory with values from the corresponding data in 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 [**(** *wait-argument* **)**]
device_type (*device-type-list* **)**
if (*condition* **)**
if_present
self (*var-list* **)**
host (*var-list* **)**
device (*var-list* **)**

Multiple subarrays of the same array may appear in a *var-list* of the same or different clauses on the same directive. For any *var* in *var-list* that is in shared memory and that is not a captured variable, no data action will occur. When a **`device`** clause appears, then for each *var* in the associated *var-list* an transfer in action is performed.

When a **`host`** or **`self`** clause appears, then for each *var* in the associated *var-list* an transfer out action is performed.

The transfer actions are performed in the order in which they appear on the directive, from left to right.

Restrictions

- At least one **`self`**, **`host`**, or **`device`** clause must appear on an **`update`** directive.

self clause

The **`self`** clause specifies that, for data not in shared memory or for captured variables, a *transfer out* action for the *vars* in *var-list* is performed. Otherwise, no action is taken.

An **`update`** directive with the **`self`** clause is equivalent to a call to the **`acc_update_self`** routine, described in Section 3.2.20.

host clause

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

device clause

The **`device`** clause specifies that a *transfer in* action for the *vars* in *var-list* is performed for data not in shared memory or for the captured variables. Otherwise, no action is taken.

An **`update`** directive with the **`device`** clause is equivalent to a call to the **`acc_update_device`** routine, described in Section 3.2.20.

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 *true*.

async clause

The **`async`** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

wait clause

The **`wait`** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

if_present clause

When an **`if_present`** clause appears on the directive, no action is taken for a *var* which appears in *var-list* that is not present in the device-accessible memory of the current device.

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.
- If no **if_present** clause appears on the directive, each *var* in *var-list* must be present in the device-accessible memory of the current device.
- 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 appear. It is implementation-specific whether noncontiguous regions are updated by using one transfer for each contiguous subregion, or whether the non-contiguous data is packed, transferred once, and unpacked, or whether one or more larger subarrays (no larger than the smallest contiguous region that contains the specified subarray) are updated.
- In C and C++, a member of a struct or class may appear, including a subarray of a member. Members of a subarray of struct or class type may not appear.
- 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 appear, including a subarray of a member. Members of subarrays of derived type may not appear.
- In Fortran, if array or subarray notation is used for a derived type member, array or subarray notation may not be used for a parent of that derived type member.
- See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **self**, **host**, and **device** clauses.

Errors

- An **acc_error_not_present** error is issued if no **if_present** clause appears and any *var* in a **device** or **self** clause is not present on the current device.
- An **acc_error_partly_present** error is issued if part of *var* is present in the current device memory but all of *var* is not.
- An **async** or **wait** clause can cause an error to be issued; see Sections 2.16.1 and 2.16.2.

See Section 5.2.2.

2.14.5 Wait Directive

See Section 2.16 Asynchronous Behavior for more information.

2.14.6 Enter Data Directive

See Section 2.6.6 Enter Data and Exit Data Directives for more information.

2.14.7 Exit Data Directive

See Section 2.6.6 Enter Data and Exit Data Directives for more information.

2.15 Procedure Calls in Compute Regions

This section describes how routines are compiled for an accelerator and how procedure calls are compiled in compute regions. See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in procedure calls inside compute regions.

2.15.1 Routine Directive

Summary

The **routine** directive is used to tell the compiler to compile the definition for a procedure, such as a function or C++ lambda, for an accelerator as well as for the host. The **routine** directive is also used to tell the compiler 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, a function prototype, or a C++ lambda and applies to the function or C++ lambda. The **routine** directive with a name may appear anywhere that a function prototype is allowed and applies to the function or the C++ lambda in scope with that name. See Section A.3.4 for recommended diagnostics for a **routine** directive with a name.

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.

The *clause* is one of the following:

gang [**(dim:** *int-expr* **)**]
worker
vector
seq
bind(*name* **)**
bind(*string* **)**
device_type(*device-type-list* **)**
nohost

A **gang**, **worker**, **vector**, or **seq** clause specifies the *level of parallelism* in the routine.

A procedure compiled with the **routine** directive for an accelerator is called an *accelerator routine*.

If no explicit **routine** directive applies to a procedure whose definition appears in the program unit being compiled, then the implementation applies an implicit **routine** directive to that procedure if any of the following conditions holds:

- The procedure is called or its address is accessed in a compute region.
- The procedure is a C++ lambda defined in an accelerator routine that has a **nohost** clause, which is considered relevant below.
- The procedure is a C++ lambda that is the parent compute scope of either:
  - A **loop** construct. If it is data-independent, then its explicit **gang**, **worker**, and **vector** clauses are considered relevant below.
  - A call to an accelerator routine whose **routine** directive has a **gang**, **worker**, **vector**, or **nohost** clause, each of which is considered relevant below.

From the set containing **seq** and all relevant clauses identified above, the implicit **routine** directive then copies any **nohost** clause and the highest level-of-parallelism clause.

The implementation may apply predetermined **routine** directives with a **seq** clause to any procedures that it provides for an accelerator, such as those of base language standard libraries.

Note: Important consequences of the above specification are:

- An implicit **routine** directive always has only a **seq** clause if the procedure is not a lambda.
- Before determining an implicit **routine** directive for a lambda, the implementation must analyze **auto** clauses to determine if the lambda's orphaned **loop** constructs are data-independent (see the **auto** clause example later in this section).
- When the implementation applies an implicit **routine** directive to a procedure, it must recursively apply implicit **routine** directives to other procedures for which the above rules specify relevant dependencies. Such dependencies can form a cycle, so the implementation must take care to avoid infinite recursion.

gang clause

The associated dimension is the value of the **dim** clause, if it appears, or is dimension one. The **dim** argument must be a constant positive integer with value 1, 2, or 3.

The **gang** clause with dimension *d* specifies that the procedure can be the parent compute scope of a loop or a call to a routine with a **gang** clause associated with dimension *d* or less, but it must not be the parent compute scope of a loop or a call to a routine with a **gang** clause with dimension greater than *d*.

worker clause

The **worker** clause specifies that the procedure can be the parent compute scope of a loop or a call to a routine with a **worker** clause, but it must not be the parent compute scope of a loop or a call to a routine with a **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.

vector clause

The **vector** clause specifies that the procedure can be the parent compute scope of a loop or a call to a routine with a **vector** clause, but it must not be the parent compute scope of a loop or a call to a routine with 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.

seq clause

The **seq** clause specifies that the procedure must not be the parent compute scope of a loop or a call to a routine 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.

bind clause

The **bind** clause specifies the name to use when calling the procedure on a device other than the host. If the name is specified as an identifier, it is 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. A **bind** clause on a procedure definition behaves as if it had appeared on a declaration by changing the name used to call the procedure on a device other than the host; however, the procedure is not compiled for the device with either the original name or the name in the **bind** clause.

If there is both a Fortran bind and an acc **bind** clause for a procedure definition then a call on the host will call the Fortran bound name and a call on another device will call the name in the **bind** clause.

device_type clause

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

nohost clause

The **nohost** clause tells the compiler not to compile a version of this procedure for the host.

Restrictions

- Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device_type** clause.
- Exactly one of the **gang**, **worker**, **vector**, or **seq** clauses must appear.
- 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.
- A call to a procedure with a **nohost** clause must not appear in a compute construct that is compiled for the host. See examples below.
- If a call to a procedure with a **nohost** clause appears in another procedure but outside any compute construct, that other procedure must also have a **nohost** clause.
- A call to a procedure with a **gang(dim:***d***)** clause must appear in code that is executed in *gang-redundant* mode in all dimensions d and lower. For instance, a procedure with a **gang(dim:2)** clause may not be called from within a loop that has a **gang(dim:1)** or a **gang(dim:2)** clause. The user needs to ensure that a call to a procedure with a **gang(dim:***d***)** clause, when present in a region executing in *GRe* or *GPe* mode with $e > d$ and called by a gang along dimension e, is executed by all of its corresponding gangs along dimension d.
- A **bind** clause may not bind to a routine name that has a visible **bind** clause.
- If a procedure has a **bind** clause on both the declaration and the definition then they both must bind to the same name.
- In C and C++, a definition or use of a procedure must appear within the scope of at least one explicit and applying **routine** directive if any appears in the same compilation unit. An explicit **routine** directive's scope is from the directive to the end of the compilation unit. If the **routine** directive appears in the member list of a C++ class, then its scope also extends in the same manner as any class member's scope (e.g., it includes the bodies of all other member functions).

Examples

- A function, such as **f** below, requires a **nohost** clause if it contains accelerator-specific code that cannot be compiled for the host. By default, some implementations compile all compute constructs for the host in addition to accelerators. In that case, a call to **f** must not appear in any compute construct or compilation will fail. However, **f** can appear in the **bind** clause of another function, such as **g** below, that does not have a **nohost** clause, and a call to **g** can appear in a compute construct. Thus, **g** is called when the compute construct is compiled for the host, and **f** is called when the compute construct is compiled for accelerators.

```
#pragma acc routine seq nohost
void f() { /*accelerator implementation*/ }

#pragma acc routine seq bind(f)
void g() { /*host implementation*/ }

void h() {
  #pragma acc parallel
  g();
}
```

- In C, the restriction that a function's definitions and uses must appear within any applying `routine` directive's scope has a simple interpretation: the `routine` directive must appear first. This interpretation seems intuitive for the common case in C where prototypes, definitions, and `routine` directives for a function, such as `f` below, appear at global scope.

```
void f();
void scopeA() {
  #pragma acc parallel
  f(); // nonconforming
}
// The routine directive's scope is not f's full scope.
// Instead, it starts at the routine directive.
#pragma acc routine(f) gang
void scopeB() {
  #pragma acc parallel
  f(); // conforming
}
void f() {} // conforming
```

- C++ classes permit forward references from member function bodies to other members declared later. For example, immediately within `class A` below, `g`'s scope does not start until after `f`'s definition. Nevertheless, within `f`'s body, `g` is in scope throughout. The same is true for `g`'s `routine` directive. Thus, `f`'s call to `g` is conforming.

```
class A {
  void f() {
    #pragma acc parallel
    g(); // conforming
  }
  #pragma acc routine gang
  void g();
};
```

- In some places, C++ classes do not permit forward references. For example, in the return type of a member function, a member typedef that is declared later is not in scope. Likewise, `g`'s definition below is not fully within the scope of `g`'s `routine` directive even though its body is, so its definition is nonconforming.

```
class A {
  #pragma acc routine(f) gang
  void f() {} // conforming
  void g() {} // nonconforming
  #pragma acc routine(g) gang
};
```

- The C++ scope resolution operator and `using` directive do not affect the scope of `routine` directives. For example, the `routine` directive below is specified for the name `f`, which resolves to `A::f`. Every reference to both `A::f` and `C::f` afterward is in the `routine` directive's scope, but the `routine` directive always applies to `A::f` and never `C::f` even when referenced as just `f`.

```
namespace A {
  void f();
  namespace B {
```

```
      #pragma acc routine(f) gang // applies to A::f
    }
  }
  void g() {
    #pragma acc parallel
    A::f(); // conforming
  }
  void h() {
    using A::f;
    #pragma acc parallel
    f(); // conforming
  }
  namespace C {
    void f();
    using namespace A::B;
    void i() {
      #pragma acc parallel
      f(); // nonconforming
    }
  }
```

- Based on the specification of implicit **gang** clauses in Section 2.9.2, the implementation must determine the implicit **routine** directive for a C++ lambda before it determines implicit **gang** clauses on its orphaned **loop** constructs. This behavior minimizes the implicit **routine** directive's level of parallelism and thus maximizes the number of places the lambda can be called. For example, the implicit **routine** directive for **f** below has only a **vector** clause so that **f** can be called within gang or worker loops. An orphaned **loop** construct has an implicit **gang** clause only if, as in **h** below, it does not have an explicit **gang** clause but gang parallelism appears elsewhere in the lambda, such as the call to **g**.

```
  // step 1: implicit #pragma acc routine vector
  auto f = []() {
    #pragma acc loop vector // step 2: no implicit gang clause
    for (int i = 0; i < I; ++i)
    ;
  };

  #pragma acc routine gang
  void g();

  // step 1: implicit #pragma acc routine gang
  auto h = []() {
    #pragma acc loop // step 2: implicit gang clause
    for (int i = 0; i < I; ++i)
    ;
    g();
  };
```

- As specified earlier in this section, before the implementation determines the implicit **routine** directive for a C++ lambda, it must analyze **auto** clauses on its orphaned **loop** constructs. This behavior can enable additional parallelism at the lambda's call sites when the implementation cannot find parallelism within the lambda. For example, within **f** below, if the implementation treats **auto** as **seq**, then **f**'s implicit **routine** directive has a **seq** clause, which permits the implementation to worker- or vector-partition **h**'s **loop** construct. If the implementation instead treats **f**'s **auto** as **independent**, then **f**'s implicit **routine** directive has a **worker** clause, so the implementation cannot worker- or vector-partition **h**'s **loop** construct.

```
// step 2: implicit #pragma acc routine with seq or worker
auto f = []() {
  // step 1: auto -> seq or independent
  #pragma acc loop auto worker vector
  for (int j = 0; j < J; ++j) {
    // complex loop body
  }
};

#pragma acc routine seq
void g();

void h() {
  #pragma acc parallel num_gangs(NG)
  // step 3: implicit gang, possibly worker or vector
  #pragma acc loop
  for (int i = 0; i < I; ++i) {
    f();
    g();
  }
}
```

When combining **auto** and **gang** on a **loop** construct within a lambda, the above behavior might expose portability issues across implementations. For example, if the user adds an explicit **gang** clause to **f**'s **loop** construct, then whether the implementation treats **f**'s **auto** as **seq** or **independent** determines whether **f**'s implicit **routine** directive has a **seq** or **gang** clause. That determines whether **h**'s **loop** construct has an implicit **gang** clause, which determines how many times **g** is called: **I** times in gang-partitioned mode, or **NG*I** times in gang-redundant mode.

- By specifying a contract between a procedure and its callers, implicit **routine** directives help to establish the semantics of OpenACC programs to facilitate both the user's understanding of the behavior and also the implementation's analysis and diagnostics. However, as usual, the implementation is free to perform optimizations that preserve program semantics. For example, the implicit **routine** directive for the C++ lambda **f** below has a **seq** clause because **f**'s definition provides no means to determine a higher parallelism level and because executing **f**'s **loop** constructs sequentially is compatible with any conceivable call site. Nevertheless, observing that both of **f**'s **loop** constructs are data-independent and that **g**'s call to **f** is in vector-single mode, the implementation might choose to inline a version of **f** such that both **loop** constructs are vector-partitioned.

```
// implicit #pragma acc routine seq
auto f = []() {
  #pragma acc loop auto // auto -> independent
  for (int i = 0; i < I; ++i)
    ;
  #pragma acc loop // implicit independent
  for (int i = 0; i < I; ++i)
```

```
            ;
        };
        void g() {
          #pragma acc parallel loop gang worker
          for (int i = 0; i < I; ++i)
            f(); // can inline with vector partitioning
        }
```

2.15.2 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.

2.16 Asynchronous Behavior

This section describes the **async** clause, the **wait** clause, the **wait** directive, and the behavior of programs that use asynchronous data movement, compute regions, and asynchronous API routines.

In this section and throughout the specification, the term *async-argument* means a nonnegative scalar integer expression (*int* for C or C++, *integer* for Fortran), or one of the special values **acc_async_noval** or **acc_async_sync**, as defined in the C header file and the Fortran **openacc** module. The special values are negative values, so as not to conflict with a user-specified nonnegative *async-argument*. An *async-argument* is used in **async** clauses, **wait** clauses, **wait** directives, and as an argument to various runtime routines.

The *async-value* of an *async-argument* is

- **acc_async_sync** if *async-argument* has a value equal to the special value **acc_async_sync**,
- the value of *acc-default-async-var* if *async-argument* has a value equal to the special value **acc_async_noval**,
- the value of the *async-argument*, if it is nonnegative,
- implementation-defined, otherwise.

The *async-value* is used to select the activity queue to which the clause or directive or API routine refers. The properties of the current 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 on the same device 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 or concurrently relative to each other. If there are two or more host threads executing and sharing the same device, asynchronous operations on any thread with the same *async-value* will be enqueued onto the same activity queue. If the threads are not synchronized with respect to each other, the operations may be enqueued in either order and therefore may execute on the device in either order.

Asynchronous operations enqueued to difference devices may execute in any order or may execute concurrently, regardless of the *async-value* used for each.

If a compute construct, data directive, or runtime API call has an *async-value* of **acc_async_sync**, the associated operations are executed on the activity queue associated with the *async-value* **acc_async_sync**, and the local thread will wait until the associated operations have completed before executing the code following the construct or directive. If a **data** construct has an *async-value* of **acc_async_sync**, the associated operations are executed on the activity queue associated with the *async-value* **acc_async_sync**, and the local thread will wait until the associated operations that occur upon entry of the construct have completed before executing the code of the construct's structured block or block construct, and after that, will wait until the associated operations that occur upon exit of the construct have completed before executing the code following the construct.

If a compute construct, data directive, or runtime API call has an *async-value* other than **acc_async_sync**, the associated operations are executed on the activity queue associated with that *async-value* and the associated operations may be processed asynchronously while the local thread continues executing the code following the construct or directive. If a **data** construct has an *async-value* other than **acc_async_sync**, the associated operations are executed on the activity queue associated with that *async-value*, and the associated operations that occur upon entry of the construct may be processed asynchronously while the local thread continues executing the code of the construct's structured block or block construct, and after that, the associated operations that occur upon exit of the construct may be processed asynchronously while the local thread continues executing the code following the construct.

In this section and throughout the specification, the term *wait-argument*, means:

[devnum : *int-expr* **:] [queues :]** *async-argument-list*

If a **devnum** modifier appears in the *wait-argument* then the associated device is the device with that device number of the current device type. If no **devnum** modifier appears then the associated device is the current device.

Each *async-argument* is associated with an *async-value*. The *async-values* select the associated activity queue or queues on the associated device. If there is no *async-argument-list*, the associated activity queues are all activity queues for the associated device.

The **queues** modifier within a *wait-argument* is optional to improve clarity of the expression list.

2.16.1 async clause

The **async** clause may appear on a **parallel**, **serial**, **kernels**, or **data** construct, or an **enter data**, **exit data**, **update**, or **wait** directive. In all cases, the **async** clause is optional. The **async** clause may have a single *async-argument*, as defined above. If the **async** clause does not appear, the behavior is as if the *async-argument* is **acc_async_sync**. If the **async** clause appears with no argument, the behavior is as if the *async-argument* is **acc_async_noval**. The *async-value* for a construct or directive is defined in Section 2.16.

Errors

- An **acc_error_invalid_async** error is issued if an **async** clause with an argument appears on any directive and the argument is not a valid *async-argument*.

See Section 5.2.2.

2.16.2 wait clause

The **`wait`** clause may appear on a **`parallel`**, **`serial`**, or **`kernels`**, or **`data`** 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 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 *wait-argument*, the associated device and activity queues are as specified in the *wait-argument*; see Section 2.16. If there is no argument to the **`wait`** clause, the associated device is the current device and associated activity queues are all activity queues. The associated operations may not be launched or executed until all operations already enqueued up to this point by this thread on the associated asynchronous device activity queues have completed. **Note:** One legal implementation is for the local thread to wait until the operations already enqueued on the associated asynchronous device activity queues have completed; another legal implementation is for the local thread to enqueue the associated operations in such a way that they will not start until the operations already enqueued on the associated asynchronous device activity queues have completed.

Errors

- An **`acc_error_device_unavailable`** error is issued if a **`wait`** clause appears on any directive with a **`devnum`** modifier and the associated *int-expr* is not a valid device number.
- An **`acc_error_invalid_async`** error is issued if a **`wait`** clause appears on any directive with a **`queues`** modifier or no modifier and any value in the associated list is not a valid *async-argument*.

See Section 5.2.2.

2.16.3 Wait Directive

Summary

The **`wait`** directive causes the local thread or operations enqueued onto a device activity queue on the current device to wait for completion of asynchronous operations.

Syntax

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

`#pragma acc wait` [**`(`** *wait-argument* **`)`**] [*clause-list*] *new-line*

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

`!$acc wait` [**`(`** *wait-argument* **`)`**] [*clause-list*]

where *clause* is:

`async` [**`(`** *async-argument* **`)`**]
`if(` *condition* **`)`**

If it appears, the *wait-argument* is as defined in Section 2.16, and the associated device and activity queues are as specified in the *wait-argument*. If there is no *wait-argument* clause, the associated device is the current device and associated activity queues are all activity queues.

If there is no **`async`** clause, the local thread will wait until all operations enqueued by this thread onto each of the associated device activity queues for the associated device have completed. There is no guarantee that all the asynchronous operations initiated by other threads onto those queues will have completed without additional synchronization with those threads.

If there is an **async** clause, no new operation may be launched or executed on the activity queue associated with the *async-argument* on the current device until all operations enqueued up to this point by this thread on the activity queues associated with the *wait-argument* have completed. **Note:** One legal implementation is for the local thread to wait for all the associated 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 activity queues have completed.

The **if** clause is optional; when there is no **if** clause, the implementation will generate code to perform the wait operation unconditionally. When an **if** clause appears, the implementation will generate code to conditionally perform the wait operation only when the *condition* evaluates to *true*.

A **wait** directive is functionally equivalent to a call to one of the **acc_wait**, **acc_wait_async**, **acc_wait_all**, or **acc_wait_all_async** runtime API routines, as described in Sections 3.2.10 and 3.2.11.

Errors

- An **acc_error_device_unavailable** error is issued if a **devnum** modifier appears and the *int-expr* is not a valid device number.
- An **acc_error_invalid_async** error is issued if a **queues** modifier or no modifier appears and any value in the associated list is not a valid *async-argument*.

See Section 5.2.2.

2.17 Fortran Specific Behavior

2.17.1 Optional Arguments

This section refers to the Fortran intrinsic function **PRESENT**. A call to the Fortran intrinsic function **PRESENT(arg)** returns **.true.**, if **arg** is an optional dummy argument and an actual argument for **arg** was present in the argument list of the call site. This is unrelated to the OpenACC **present** data clause.

The appearance of a Fortran optional argument **arg** as a *var* in any of the following clauses has no effect at runtime if **PRESENT(arg)** is **.false.**:

- in data clauses on compute and **data** constructs;
- in data clauses on **enter data** and **exit data** directives;
- in data and **device_resident** clauses on **declare** directives;
- in **use_device** clauses on **host_data** directives;
- in **self**, **host**, and **device** clauses on **update** directives.

The appearance of a Fortran optional argument **arg** in the following situations may result in undefined behavior if **PRESENT(arg)** is **.false.** when the associated construct is executed:

- as a *var* in **private**, **firstprivate**, and **reduction** clauses;
- as a *var* in **cache** directives;

- as part of an expression in any clause or directive.

A call to the Fortran intrinsic function **PRESENT** behaves the same way in a compute construct or an accelerator routine as on the host. The function call **PRESENT(arg)** must return the same value in a compute construct as **PRESENT(arg)** would outside of the compute construct. If a Fortran optional argument **arg** appears as an actual argument in a procedure call in a compute construct or an accelerator routine, and the associated dummy argument **subarg** also has the **optional** attribute, then **PRESENT(subarg)** returns the same value as **PRESENT(subarg)** would when executed on the host.

2.17.2 Do Concurrent Construct

This section refers to the Fortran **do concurrent** construct that is a form of **do** construct. When **do concurrent** appears without a **loop** construct in a **kernels** construct it is treated as if it is annotated with **loop auto**. If it appears in a **parallel** construct or an accelerator routine then it is treated as if it is annotated with **loop independent**.

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

There are four categories of runtime routines:

- Device management routines, to get the number of devices, set the current device, and so on.
- Asynchronous queue management, to synchronize until all activities on an async queue are complete, for instance.
- Device test routine, to test whether this statement is executing on the device or not.
- Data and memory management, to manage memory allocation or copy data between memories.

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 the supported device types.
- The values of **acc_async_noval**, **acc_async_sync**, and **acc_async_default**.

In Fortran, interface declarations are provided in a Fortran module named **openacc**. The **openacc** module defines:

- 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**.
- Interfaces for all routines in the chapter.
- Integer parameters to define integer kinds for arguments to and return values for those routines.
- Integer parameters to describe the supported device types.
- Integer parameters to define the values of **acc_async_noval**, **acc_async_sync**, and **acc_async_default**.

Many of the routines accept or return a value corresponding to the type of 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 defined in the C or C++ include file `openacc.h` and the Fortran module `openacc`. Five values are always supported: `acc_device_none`, `acc_device_default`, `acc_device_host`, `acc_device_not_host`, and `acc_device_current`. 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 memory address or device memory address, when these calls are executed on the host, as if the following definitions were included:

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

Many Fortran API bindings defined in this section rely on types defined in Fortran's `iso_c_binding` module. It is implied that the `iso_c_binding` module is used in these bindings, even if not explicitly stated in the format section for that routine.

Restrictions

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 available devices of the given type.

Format

C or C++:

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

Fortran:

```
integer function acc_get_num_devices(dev_type)
 integer(acc_device_kind) ::  dev_type
```

Description

The `acc_get_num_devices` routine returns the number of available devices of device type `dev_type`. If device type `dev_type` is not supported or no device of `dev_type` is available, this routine returns zero.

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 a compute region and sets the value of *acc-current-device-type-var*. This is useful when the implementation allows the program to be compiled to use more than one type of device.

Format

C or C++:

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

Fortran:

```
subroutine acc_set_device_type(dev_type)
 integer(acc_device_kind) :: dev_type
```

Description

A call to **acc_set_device_type** is functionally equivalent to a **set device_type(dev_type)** directive, as described in Section 2.14.3. This routine tells the runtime which type of device to use among those available and sets the value of *acc-current-device-type-var* for the current thread to **dev_type**.

Restrictions

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

Errors

- An **acc_error_device_type_unavailable** error is issued if device type **dev_type** is not supported or no device of **dev_type** is available.

See Section 5.2.2.

3.2.3 acc_get_device_type

Summary

The **acc_get_device_type** routine returns the value of *acc-current-device-type-var*, which is the device type of the current device. This is useful when the implementation allows the program to be compiled to use more than one type of device.

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 the value of *acc-current-device-type-var* for the current thread to tell the program what type of device will be used to run the next compute region, if one has been selected. The device type may have been selected by the program with a runtime API call or a directive, by an environment variable, or by the default behavior of the implementation; see the table in Section 2.3.1.

Restrictions

- 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 and sets the value of *acc-current-device-num-var*.

Format

C or C++:

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

Fortran:

```
subroutine acc_set_device_num(dev_num, dev_type)
 integer :: dev_num
 integer(acc_device_kind) :: dev_type
```

Description

A call to **acc_set_device_num** is functionally equivalent to a **set device_type(dev_type) device_num(dev_num)** directive, as described in Section 2.14.3. This routine tells the runtime which device to use among those available of the given type for compute or data regions in the current thread and sets the value of *acc-current-device-num-var* to **dev_num**. If the value of **dev_num** is negative, the runtime will revert to its default behavior, which is implementation-defined. If the value of the **dev_type** is zero, the selected device number will be used for all device types. Calling **acc_set_device_num** implies a call to **acc_set_device_type(dev_type)**.

Errors

- An **acc_error_device_type_unavailable** error is issued if device type **dev_type** is not supported or no device of **dev_type** is available.
- An **acc_error_device_unavailable** error is issued if the value of **dev_num** is not a valid device number.

See Section 5.2.2.

3.2.5 acc_get_device_num

Summary

The **acc_get_device_num** routine returns the value of *acc-current-device-num-var* for the current thread.

Format

C or C++:

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

Fortran:

```
integer function acc_get_device_num(dev_type)
 integer(acc_device_kind) :: dev_type
```

Description

The **acc_get_device_num** routine returns the value of *acc-current-device-num-var* for the current thread. If there are no devices of device type **dev_type** or if device type **dev_type** is not supported, this routine returns **-1**.

3.2.6 acc_get_property

Summary

The **acc_get_property** and **acc_get_property_string** routines return the value of a *device-property* for the specified device.

Format

C or C++:

```
size_t acc_get_property(int dev_num,
                        acc_device_t dev_type,
                        acc_device_property_t property);
const
char* acc_get_property_string(int dev_num,
                        acc_device_t dev_type,
                        acc_device_property_t property);
```

Fortran:

```
function acc_get_property(dev_num, dev_type, property)
subroutine acc_get_property_string(dev_num, dev_type,&
                        property, string)
 integer, value ::  dev_num
 integer(acc_device_kind), value ::  dev_type
 integer(acc_device_property_kind), value ::  property
 integer(c_size_t) ::  acc_get_property
 character*(*) ::  string
```

Description

The **acc_get_property** and **acc_get_property_string** routines return the value of the *property*. **dev_num** and **dev_type** specify the device being queried. If **dev_type** has the value **acc_device_current**, then **dev_num** is ignored and the value of the property for the current device is returned. **property** is an enumeration constant, defined in **openacc.h**, for C or C++, or an integer parameter, defined in the **openacc** module, for Fortran. Integer-valued properties are returned by **acc_get_property**, and string-valued properties are returned by **acc_get_property_string**. In Fortran, **acc_get_property_string** returns the result into the **string** argument.

The supported values of **property** are given in the following table.

property	*return type*	*return value*
acc_property_memory	*integer*	size of device memory in bytes
acc_property_free_memory	*integer*	free device memory in bytes
acc_property_shared_memory_support	*integer*	nonzero if the specified device supports sharing memory with the local thread
acc_property_name	*string*	device name
acc_property_vendor	*string*	device vendor
acc_property_driver	*string*	device driver version

An implementation may support additional properties for some devices.

Restrictions

- **acc_get_property** will return 0 and **acc_get_property_string** will return a null pointer (in C or C++) or a blank string (in Fortran) in the following cases:
  - If device type **dev_type** is not supported or no device of **dev_type** is available.
  - If the value of **dev_num** is not a valid device number for device type **dev_type**.
  - If the value of **property** is not one of the known values for that query routine, or that property has no value for the specified device.

3.2.7 acc_init

Summary

The **acc_init** and **acc_init_device** routines initialize the runtime for the specified device type and device number. This can be used to isolate any initialization cost from the computational cost, such as when collecting performance statistics.

Format

C or C++:

```
void acc_init(acc_device_t dev_type);
void acc_init_device(int dev_num, acc_device_t dev_type);
```

Fortran:

```
subroutine acc_init(dev_type)
subroutine acc_init_device(dev_num, dev_type)
 integer :: dev_num
 integer(acc_device_kind) :: dev_type
```

Description

A call to **acc_init** or **acc_init_device** is functionally equivalent to an **init** directive with matching **dev_type** and **dev_num** arguments, as described in Section 2.14.1. **dev_type** must be one of the defined accelerator types. **dev_num** must be a valid device number of the device type **dev_type**. These routines also implicitly call **acc_set_device_type(dev_type)**. In the case of **acc_init_device**, **acc_set_device_num(dev_num)** is also called.

If a program initializes one or more devices without an intervening **shutdown** directive or **acc_shutdown** call to shut down those same devices, no action is taken.

Errors

- An **acc_error_device_type_unavailable** error is issued if device type **dev_type** is not supported or no device of **dev_type** is available.
- An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device number.

See Section 5.2.2.

3.2.8 acc_shutdown

Summary

The **acc_shutdown** and **acc_shutdown_device** routines shut down the connection to specified devices and free up any related resources in the runtime. This ends all data lifetimes in device memory for the device or devices that are shut down, which effectively sets structured and dynamic reference counters to zero.

Format

C or C++:

```
void acc_shutdown(acc_device_t dev_type);
void acc_shutdown_device(int dev_num, acc_device_t dev_type);
```

Fortran:

```
subroutine acc_shutdown(dev_type)
subroutine acc_shutdown_device(dev_num, dev_type)
 integer :: dev_num
 integer(acc_device_kind) :: dev_type
```

Description

A call to **acc_shutdown** or **acc_shutdown_device** is functionally equivalent to a **shutdown** directive, with matching **dev_type** and **dev_num** arguments, as described in Section 2.14.2. **dev_type** must be one of the defined accelerator types. **dev_num** must be a valid device number of the device type **dev_type**. **acc_shutdown** routine disconnects the program from all devices of device type **dev_type**. The **acc_shutdown_device** routine disconnects the program from **dev_num** of type **dev_type**. Any data that is present in the memory of a device that is shut down is immediately deallocated.

Restrictions

- This routine may not be called while a compute region is executing on a device of type **dev_type**.
- If the program attempts to execute a compute region on a device or to access any data in the memory of a device that was shut down, the behavior is undefined.
- If the program attempts to shut down the **acc_device_host** device type, the behavior is undefined.

Errors

- An **acc_error_device_type_unavailable** error is issued if device type **dev_type** is not supported or no device of **dev_type** is available.
- An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device number.
- An **acc_error_device_shutdown** error is issued if there is an error shutting down the device.

See Section 5.2.2.

3.2.9 acc_async_test

Summary

The **acc_async_test** routines test for completion of all associated asynchronous operations for a single specified async queue or for all async queues on the current device or on a specified device.

Format

C or C++:

```
int acc_async_test(int wait_arg);
int acc_async_test_device(int wait_arg, int dev_num);
int acc_async_test_all(void);
int acc_async_test_all_device(int dev_num);
```

Fortran:

```
logical function acc_async_test(wait_arg)
logical function acc_async_test_device(wait_arg, dev_num)
logical function acc_async_test_all()
logical function acc_async_test_all_device(dev_num)
 integer(acc_handle_kind) ::  wait_arg
 integer ::  dev_num
```

Description

wait_arg must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior. **dev_num** must be a valid device number of the current device type.

The behavior of the **acc_async_test** routines is:

- If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- If any asynchronous operations initiated by this host thread on device **dev_num** either on async queue **wait_arg** (if there is a **wait_arg** argument), or on any async queue (if there is no **wait_arg** argument) have not completed, a call to the routine returns *false*.
- If all such asynchronous operations have completed, or there are no such asynchronous operations, a call to the routine returns *true*. A return value of *true* is no guarantee that asynchronous operations initiated by other host threads have completed.

Errors

- An **acc_error_invalid_async** error is issued if **wait_arg** is not a valid *async-argument* value.
- An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device number.

See Section 5.2.2.

3.2.10 acc_wait

Summary

The **acc_wait** routines wait for completion of all associated asynchronous operations on a single specified async queue or on all async queues on the current device or on a specified device.

Format

C or C++:

```
void acc_wait(int wait_arg);
void acc_wait_device(int wait_arg, int dev_num);
void acc_wait_all(void);
void acc_wait_all_device(int dev_num);
```

Fortran:

```
    subroutine acc_wait(wait_arg)
    subroutine acc_wait_device(wait_arg, dev_num)
    subroutine acc_wait_all()
    subroutine acc_wait_all_device(dev_num)
     integer(acc_handle_kind) ::  wait_arg
     integer ::  dev_num
```

Description

A call to an **acc_wait** routine is functionally equivalent to a **wait** directive as follows, see Section 2.16.3:

- **acc_wait** to a **wait(wait_arg)** directive.
- **acc_wait_device** to a **wait(devnum:dev_num, queues:wait_arg)** directive.
- **acc_wait_all** to a **wait** directive with no *wait-argument*.
- **acc_wait_all_device** to a **wait(devnum:dev_num)** directive.

wait_arg must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior. **dev_num** must be a valid device number of the current device type.

The behavior of the **acc_wait** routines is:

- If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- The routine will not return until all asynchronous operations initiated by this host thread on device **dev_num** either on async queue **wait_arg** (if there is a **wait_arg** argument) or on all async queues (if there is no **wait_arg** argument) have completed.
- If two or more threads share the same accelerator, there is no guarantee that matching asynchronous operations initiated by other threads have completed.

For compatibility with OpenACC version 1.0, **acc_wait** may also be spelled **acc_async_wait**, and **acc_wait_all** may also be spelled **acc_async_wait_all**.

Errors

- An **acc_error_invalid_async** error is issued if **wait_arg** is not a valid *async-argument* value.
- An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device number.

See Section 5.2.2.

3.2.11 acc_wait_async

Summary

The **acc_wait_async** routines enqueue a wait operation on one async queue of the current device or a specified device for the operations previously enqueued on a single specified async queue or on all other async queues.

Format

C or C++:

```
void acc_wait_async(int wait_arg, int async_arg);
void acc_wait_device_async(int wait_arg, int async_arg,
                           int dev_num);
void acc_wait_all_async(int async_arg);
void acc_wait_all_device_async(int async_arg, int dev_num);
```

Fortran:

```
subroutine acc_wait_async(wait_arg, async_arg)
subroutine acc_wait_device_async(wait_arg, async_arg, dev_num)
subroutine acc_wait_all_async(async_arg)
subroutine acc_wait_all_device_async(async_arg, dev_num)
 integer(acc_handle_kind) ::  wait_arg, async_arg
 integer ::  dev_num
```

Description

A call to an **acc_wait_async** routine is functionally equivalent to a **wait async(async_arg)** directive as follows, see Section 2.16.3:

- A call to **acc_wait_async** is functionally equivalent to a **wait(wait_arg) async(async_arg)** directive.
- A call to **acc_wait_device_async** is functionally equivalent to a **wait(devnum: dev_num, queues:wait_arg) async(async_arg)** directive.
- A call to **acc_wait_all_async** is functionally equivalent to a **wait async(async_arg)** directive with no *wait-argument*.
- A call to **acc_wait_all_device_async** is functionally equivalent to a **wait(devnum:dev_num) async(async_arg)** directive.

async_arg and **wait_arg** must must be *async-arguments*, as defined in Section 2.16 Asynchronous Behavior. **dev_num** must be a valid device number of the current device type.

The behavior of the **acc_wait_async** routines is:

- If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- The routine will enqueue a wait operation on the async queue associated with **async_arg** for the current device which will wait for operations initiated on the async queue **wait_arg** of device **dev_num** (if there is a **wait_arg** argument), or for each async queue of device **dev_num** (if there is no **wait_arg** argument).

See Section 2.16 Asynchronous Behavior for more information.

Errors

- An **acc_error_invalid_async** error is issued if either **async_arg** or **wait_arg** is not a valid *async-argument* value.
- An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device number.

See Section 5.2.2.

3.2.12 acc_wait_any

Summary

The **acc_wait_any** and **acc_wait_any_device** routines wait for any of the specified asynchronous queues to complete all pending operations on the current device or the specified device number, respectively. Both routines return the queue's index in the provided array of asynchronous queues.

Format

C or C++:

```
int acc_wait_any(int count, int wait_arg[]);
int acc_wait_any_device(int count, int wait_arg[], int dev_num);
```

Fortran:

```
integer function acc_wait_any(count, wait_arg)
integer function acc_wait_any_device(count, wait_arg, dev_num)
 integer ::  count, dev_num
 integer(acc_handle_kind) ::  wait_arg(count)
```

Description

wait_arg is an array of *async-arguments* as defined in Section 2.16 and **count** is a nonnegative integer indicating the array length. If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number. Otherwise, **dev_num** must be a valid device number of the current device type. A call to any of these routines returns an index **i** associated with a **wait_arg[i]** that is not **acc_async_sync** and meets the conditions that would evaluate **acc_async_test_device(wait_arg[i], dev_num)** to *true*. If all the elements in **wait_arg** are equal to **acc_async_sync** or **count** is equal to **0**, these routines return **-1**. Otherwise, the return value is an integer in the range of **0** $\leq$ **i** $<$ **count** in C or C++ and **1** $\leq$ **i** $\leq$ **count** in Fortran.

Errors

- An **acc_error_invalid_argument** error is issued if **count** is a negative number.
- An **acc_error_invalid_async** error is issued if any element encountered in **wait_arg** is not a valid *async-argument* value.
- An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device number.

See Section 5.2.2.

3.2.13 acc_get_default_async

Summary

The **acc_get_default_async** routine returns the value of *acc-default-async-var* for the current thread.

Format

C or C++:

```
int acc_get_default_async(void);
```

Fortran:

```
function acc_get_default_async()
 integer(acc_handle_kind) :: acc_get_default_async
```

Description

The **acc_get_default_async** routine returns the value of *acc-default-async-var* for the current thread, which is the asynchronous queue used when an **async** clause appears without an *async-argument* or with the value **acc_async_noval**.

3.2.14 acc_set_default_async

Summary

The **acc_set_default_async** routine tells the runtime which asynchronous queue to use when an **async** clause appears with no queue argument.

Format

C or C++:

```
void acc_set_default_async(int async_arg);
```

Fortran:

```
subroutine acc_set_default_async(async_arg)
 integer(acc_handle_kind) :: async_arg
```

Description

A call to **acc_set_default_async** is functionally equivalent to a **set default_async(async_arg)** directive, as described in Section 2.14.3. This **acc_set_default_async** routine tells the runtime to place any directives with an **async** clause that does not have an *async-argument* or with the special **acc_async_noval** value into the asynchronous activity queue associated with **async_arg** instead of the default asynchronous activity queue for that device by setting the value of *acc-default-async-var* for the current thread. The special argument **acc_async_default** will reset the default asynchronous activity queue to the initial value, which is implementation-defined.

Errors

- An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-argument* value.

See Section 5.2.2.

3.2.15 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 dev_type);
```

Fortran:

```
logical function acc_on_device(dev_type)
 integer(acc_device_kind) :: dev_type
```

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, the call evaluates at compile time to a constant. **dev_type** must be one of the defined accelerator types.

The behavior of the **acc_on_device** routine is:

- If **dev_type** is **acc_device_host**, then outside of a compute region or accelerator routine, or in a compute region or accelerator routine that is executed on the host CPU, a call to this routine will evaluate to *true*; otherwise, it will evaluate to *false*.
- If **dev_type** is **acc_device_not_host**, the result is the negation of the result with argument **acc_device_host**.
- If **dev_type** is an accelerator device type, then in a compute region or routine that is executed on a device of that type, a call to this routine will evaluate to *true*; otherwise, it will evaluate to *false*.
- The result with argument **acc_device_default** is undefined.

3.2.16 acc_malloc

Summary

The **acc_malloc** routine allocates space in the current device memory.

Format

C or C++:

```
d_void* acc_malloc(size_t bytes);
```

Fortran:

```
type(c_ptr) function acc_malloc(bytes)
 integer(c_size_t), value :: bytes
```

Description

The **acc_malloc** routine may be used to allocate space in the current device memory. Pointers assigned from this routine may be used in **deviceptr** clauses to tell the compiler that the pointer target is resident on the device. In case of an allocation error or if **bytes** has the value zero, **acc_malloc** returns a null pointer.

3.2.17 acc_free

Summary

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

Format

C or C++:

```
void acc_free(d_void* data_dev);
```

Fortran:

```
subroutine acc_free(data_dev)
 type(c_ptr), value :: data_dev
```

Description

The **acc_free** routine will free previously allocated space in the current device memory; **data_dev** must be a pointer value that was returned by a call to **acc_malloc** or a null pointer. If **data_dev** is a null pointer, no operation is performed.

3.2.18 acc_copyin and acc_create

Summary

The **acc_copyin** and **acc_create** routines test to see if the argument is in shared memory or already present in device-accessible memory of the current device; if not, they allocate space in device-accessible memory of the current device to correspond to the specified local memory, and the **acc_copyin** routines copy the data to that device-accessible memory.

Format

C or C++:

```
d_void* acc_copyin(h_void* data_arg, size_t bytes);
d_void* acc_create(h_void* data_arg, size_t bytes);

void acc_copyin_async(h_void* data_arg, size_t bytes,
                      int async_arg);
void acc_create_async(h_void* data_arg, size_t bytes,
                      int async_arg);
```

Fortran:

```
subroutine acc_copyin(data_arg [, bytes])
subroutine acc_create(data_arg [, bytes])

subroutine acc_copyin_async(data_arg [, bytes], async_arg)
subroutine acc_create_async(data_arg [, bytes], async_arg)

 type(*), dimension(..) :: data_arg
 integer :: bytes
 integer(acc_handle_kind) :: async_arg
```

Description

A call to an **acc_copyin** or **acc_create** routine is similar to an **enter data** directive with a **copyin** or **create** clause, respectively, as described in Sections 2.7.8 and 2.7.10, except that no *attach pointer* action is performed for a pointer reference. In C/C++, **data_arg** is a pointer to the data, and **bytes** specifies the data size in bytes; the associated *data section* starts at the address in **data_arg** and continues for **bytes** bytes. The synchronous routines return a pointer to the allocated device memory, as with **acc_malloc**. In Fortran, two forms are supported. In the first, **data_arg** is a variable or a contiguous array section; the associated *data section* starts at the address of, and continues to the end of the variable or array section. In the second, **data_arg** is a variable or array element and **bytes** is the length in bytes; the associated *data section* starts at the address of the variable or array element and continues for **bytes** bytes. For the **_async** versions of these routines, **async_arg** must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

The behavior of these routines for the associated *data section* is:

- If the *data section* is in shared memory and does not refers to a captured variable, no action is taken. The C/C++ synchronous **acc_copyin** and **acc_create** routines return the incoming pointer.
- If the *data section* is present in device-accessible memory of the current device, the routines perform a *increment counter* action with the dynamic reference counter. The C/C++ synchronous **acc_copyin** and **acc_create** routines return a pointer to the existing device-accessible memory.
- Otherwise:
  - The **acc_copyin** routines behave as follows:
    1. An *allocate memory* action is performed.
    2. A *transfer in* action is performed.
    3. A *increment counter* action with the dynamic reference counter is performed.
  - The **acc_create** routines behave as follows:
    1. An *allocate memory* action is performed.
    2. A *increment counter* action with the dynamic reference counter is performed.

  The C/C++ synchronous **acc_copyin** and **acc_create** routines return a pointer to the newly allocated device memory.

This data may be accessed using the **present** data clause. Pointers assigned from the C/C++ synchronous **acc_copyin** and **acc_create** routines may be used in **deviceptr** clauses to tell the compiler that the pointer target is resident on the device.

The synchronous versions will not return until the memory has been allocated and any data transfers are complete.

The **_async** versions of these routines will perform any data transfers asynchronously on the async queue associated with **async_arg**. The routine may return before the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The data will be treated as present in device-accessible memory of the current device even if the data has not been allocated or transferred before the routine returns.

For compatibility with OpenACC 2.0, **acc_present_or_copyin** and **acc_pcopyin** are alternate names for **acc_copyin**, and **acc_present_or_create** and **acc_pcreate** are alternate names for **acc_create**.

Errors

- An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and **bytes** is nonzero.
- An **acc_error_partly_present** error is issued if part of the *data section* is already present in device-accessible memory of the current device but all of the *data section* is not.
- An **acc_error_invalid_data_section** error is issued if **data_arg** is an array section that is not contiguous (in Fortran).
- An **acc_error_out_of_memory** error is issued if the accelerator device does not have enough memory for the data.

- An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-argument* value.

See Section 5.2.2.

3.2.19 acc_copyout and acc_delete

Summary

The **acc_copyout** and **acc_delete** routines test to see if the argument is in shared memory and does not refer to a captured variable; if not, the argument must be present in device-accessible memory of the current device. The **acc_copyout** routines copy data from device-accessible memory to the corresponding local memory, and both **acc_copyout** and **acc_delete** routines deallocate that space from the device-accessible memory.

Format

C or C++:

```
void acc_copyout(h_void* data_arg, size_t bytes);
void acc_delete (h_void* data_arg, size_t bytes);

void acc_copyout_finalize(h_void* data_arg, size_t bytes);
void acc_delete_finalize (h_void* data_arg, size_t bytes);

void acc_copyout_async(h_void* data_arg, size_t bytes,
                       int async_arg);
void acc_delete_async (h_void* data_arg, size_t bytes,
                       int async_arg);

void acc_copyout_finalize_async(h_void* data_arg, size_t bytes,
                                int async_arg);
void acc_delete_finalize_async (h_void* data_arg, size_t bytes,
                                int async_arg);
```

Fortran:

```
subroutine acc_copyout(data_arg [, bytes])
subroutine acc_delete (data_arg [, bytes])

subroutine acc_copyout_finalize(data_arg [, bytes])
subroutine acc_delete_finalize (data_arg [, bytes])

subroutine acc_copyout_async(data_arg [, bytes], async_arg)
subroutine acc_delete_async (data_arg [, bytes], async_arg)

subroutine acc_copyout_finalize_async(data_arg [, bytes], &
                                      async_arg)
subroutine acc_delete_finalize_async (data_arg [, bytes], &
                                      async_arg)

 type(*), dimension(..)  ::  data_arg
```

```
    integer :: bytes
    integer(acc_handle_kind) :: async_arg
```

Description

A call to an **acc_copyout** or **acc_delete** routine is similar to an **exit data** directive with a **copyout** or **delete** clause, respectively, and a call to an **acc_copyout_finalize** or **acc_delete_finalize** routine is similar to an **exit data finalize** directive with a **copyout** or **delete** clause, respectively, as described in Section 2.7.9 and 2.7.12, except that no *detach pointer* action is performed for a pointer reference. The arguments and the associated *data section* are as for **acc_copyin**.

The behavior of these routines for the associated *data section* is:

- If the *data section* is in shared memory and does not refer to a captured variable, no action is taken.
- If the dynamic reference counter for the *data section* is zero, no action is taken.
- Otherwise, the dynamic reference counter is updated:
  - The **acc_copyout** and **acc_delete**) routines perform a *decrement counter* action with the dynamic reference counter.
  - The **acc_copyout_finalize** or **acc_delete_finalize** routines perform a reset counter action with the dynamic reference counter.

  If both reference counters are then zero:

  - The **acc_copyout** routines perform a *transfer out* action followed by a *deallocate memory* action.
  - The **acc_delete** routines perform a *deallocate memory* action.

The synchronous versions will not return until the data has been completely transferred and the memory has been deallocated.

The **_async** versions of these routines will perform any associated data transfers asynchronously on the async queue associated with **async_arg**. The routine may return before the data has been transferred or deallocated; see Section 2.16 Asynchronous Behavior for more details. Even if the data has not been transferred or deallocated before the routine returns, the data will be treated as not present in device-accessible memory of the current device if both reference counters are zero.

Errors

- An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and **bytes** is nonzero.
- An **acc_error_not_present** error is issued if the *data section* is not in shared memory and is not present in the current device memory.
- An **acc_error_invalid_data_section** error is issued if **data_arg** is an array section that is not contiguous (in Fortran).
- An **acc_error_partly_present** error is issued if part of the *data section* is already present in device-accessible memory of the current device but all of the *data section* is not.

- An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-argument* value.

See Section 5.2.2.

3.2.20 acc_update_device and acc_update_self

Summary

The **acc_update_device** and **acc_update_self** routines test to see if the argument is in shared memory and it is not a captured variable; if not, the argument must be present in the device-accessible memory of the current device, and the routines update the data in device memory from the corresponding local memory (**acc_update_device**) or update the data in local memory from the corresponding device-accessible memory (**acc_update_self**).

Format

C or C++:

```
void acc_update_device(h_void* data_arg, size_t bytes);
void acc_update_self  (h_void* data_arg, size_t bytes);

void acc_update_device_async(h_void* data_arg, size_t bytes,
                             int async_arg);
void acc_update_self_async  (h_void* data_arg, size_t bytes,
                             int async_arg);
```

Fortran:

```
subroutine acc_update_device(data_arg [, bytes])
subroutine acc_update_self  (data_arg [, bytes])

subroutine acc_update_device_async(data_arg [, bytes], async_arg)
subroutine acc_update_self_async  (data_arg [, bytes], async_arg)

 type(*), dimension(..)  ::  data_arg
 integer ::  bytes
 integer(acc_handle_kind) ::  async_arg
```

Description

A call to an **acc_update_device** routine is functionally equivalent to an **update device** directive. A call to an **acc_update_self** routine is functionally equivalent to an **update self** directive. See Section 2.14.4. The arguments and the *data section* are as for **acc_copyin**.

The behavior of these routines for the associated *data section* is:

- If the *data section* is in shared memory and does not refer to a captured variable or **bytes** is zero, no action is taken.
- Otherwise:
  - A call to an **acc_update_device** routine performs a *transfer in* action with the corresponding memory.

  – A call to an **acc_update_self** routine performs a *transfer out* action with the corresponding memory.

The **_async** versions of these routines will perform the data transfers asynchronously on the async queue associated with **async_arg**. The routine may return before the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return until the data has been completely transferred.

Errors

- An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and **bytes** is nonzero.
- An **acc_error_not_present** error is issued if the *data section* is not in shared memory and is not present in the current device memory.
- An **acc_error_invalid_data_section** error is issued if **data_arg** is an array section that is not contiguous (in Fortran).
- An **acc_error_partly_present** error is issued if part of the *data section* is already present in device-accessible memory of the current device but all of the *data section* is not.
- An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-argument* value.

See Section 5.2.2.

3.2.21 acc_map_data

Summary

The **acc_map_data** routine maps previously allocated space in the current device memory to the specified host data.

Format

C or C++:

```
void acc_map_data(h_void* data_arg, d_void* data_dev,
                       size_t bytes);
```

Fortran:

```
subroutine acc_map_data(data_arg, data_dev, bytes)
 type(*),dimension(*) ::  data_arg
 type(c_ptr), value ::  data_dev
 integer(c_size_t), value ::  bytes
```

Description

A call to the **acc_map_data** routine is similar to a call to **acc_create**, except that 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. **data_arg** is a host address, **data_dev** is the corresponding device address, and **bytes** is the length in bytes. **data_dev** may be the result of a call to **acc_malloc**, or may come from some other device-specific API routine. The associated *data section* is as for **acc_copyin**.

The behavior of the **acc_map_data** routine is:

- If the *data section* is in shared memory, the behavior is undefined.
- If any of the data referred to by **data_dev** is already mapped to any host memory address, the behavior is undefined.
- Otherwise, after this call, when **data_arg** appears in a data clause, the **data_dev** address will be used. The dynamic reference count for the data referred to by **data_arg** is set to one, but no data movement will occur.

Memory mapped by **acc_map_data** may not have the associated dynamic reference count decremented to zero, except by a call to **acc_unmap_data**. See Section 2.6.7 Reference Counters.

Errors

- An **acc_invalid_null_pointer** error is issued if either **data_arg** or **data_dev** is a null pointer.
- An **acc_invalid_argument** error is issued if **bytes** is zero.
- An **acc_error_present** error is issued if any part of the *data section* is already present in the current device memory.

See Section 5.2.2.

3.2.22 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* data_arg);
```

Fortran:

```
subroutine acc_unmap_data(data_arg)
 type(*),dimension(*) :: data_arg
```

Description

A call to the **acc_unmap_data** routine is similar to a call to **acc_delete**, except the device memory is not deallocated. **data_arg** is a host address.

The behavior of the **acc_unmap_data** routine is:

- If **data_arg** was not previously mapped to some device address via a call to **acc_map_data**, the behavior is undefined.
- Otherwise, the data lifetime for **data_arg** is ended. The dynamic reference count for **data_arg** is set to zero, but no data movement will occur and the corresponding device memory is not deallocated. See Section 2.6.7 Reference Counters.

Errors

- An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer.
- An **acc_error_present** error is issued if the structured reference count for the any part of the data is not zero.

See Section 5.2.2.

3.2.23 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* data_arg);
```

Fortran:

```
type(c_ptr) function acc_deviceptr(data_arg)
 type(*),dimension(*) :: data_arg
```

Description

The **acc_deviceptr** routine returns the device pointer associated with a host address. **data_arg** is the address of a host variable or array that may have an active lifetime on the current device.

The behavior of the **acc_deviceptr** routine for the data referred to by **data_arg** is:

- If the data is in shared memory or **data_arg** is a null pointer, **acc_deviceptr** returns the incoming address.
- If the data is not present in the current device memory, **acc_deviceptr** returns a null pointer.
- Otherwise, **acc_deviceptr** returns the address in the current device memory that corresponds to the address **data_arg**.

3.2.24 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* data_dev);
```

Fortran:

```
type(c_ptr) function acc_hostptr(data_dev)
 type(c_ptr), value :: data_dev
```

Description

The **acc_hostptr** routine returns the host pointer associated with a device address. **data_dev** is the address of a device variable or array, such as that returned from **acc_deviceptr**, **acc_create** or **acc_copyin**.

The behavior of the **acc_hostptr** routine for the data referred to by **data_dev** is:

- If the data is in shared memory or **data_dev** is a null pointer, **acc_hostptr** returns the incoming address.
- If the data corresponds to a host address which is present in the current device memory, **acc_hostptr** returns the host address.
- Otherwise, **acc_hostptr** returns a null pointer.

3.2.25 acc_is_present

Summary

The **acc_is_present** routine tests whether a variable or array region is accessible from the current device.

Format

C or C++:

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

Fortran:

```
logical function acc_is_present(data_arg)
logical function acc_is_present(data_arg, bytes)
 type(*), dimension(..)  ::  data_arg
 integer ::  bytes
```

Description

The **acc_is_present** routine tests whether the specified host data is accessible from the current device. In C/C++, **data_arg** is a pointer to the data, and **bytes** specifies the data size in bytes. In Fortran, two forms are supported. In the first, **data_arg** is a variable or contiguous array section. In the second, **data_arg** is a variable or array element and **bytes** is the length in bytes. A **bytes** value of zero is treated as a value of one if **data_arg** is not a null pointer.

The behavior of the **acc_is_present** routines for the data referred to by **data_arg** is:

- If the data is in shared memory, a call to **acc_is_present** will evaluate to *true*.
- If the data is present in the current device memory, a call to **acc_is_present** will evaluate to *true*.
- Otherwise, a call to **acc_is_present** will evaluate to *false*.

Errors

- An **acc_error_invalid_argument** error is issued if **bytes** is negative (in Fortran).
- An **acc_error_invalid_data_section** error is issued if **data_arg** is an array section that is not contiguous (in Fortran).

See Section 5.2.2.

3.2.26 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* data_dev_dest,
                          h_void* data_host_src, size_t bytes);
void acc_memcpy_to_device_async(d_void* data_dev_dest,
                          h_void* data_host_src, size_t bytes,
                          int async_arg);
```

Fortran:

```
subroutine acc_memcpy_to_device(data_dev_dest,
                                data_host_src, bytes)
subroutine acc_memcpy_to_device_async(data_dev_dest,
                                data_host_src, bytes, async_arg)
 type(c_ptr), value ::  data_dev_dest
 type(*),dimension(*)       ::  data_host_src
 integer(c_size_t), value ::  bytes
 integer(acc_handle_kind), value ::  async_arg
```

Description

The **acc_memcpy_to_device** routine copies **bytes** bytes of data from the local address in **data_host_src** to the device address in **data_dev_dest**. **data_dev_dest** must be an address accessible from the current device, such as an address returned from **acc_malloc** or **acc_deviceptr**, or an address in shared memory.

The behavior of the **acc_memcpy_to_device** routines is:

- If **bytes** is zero, no action is taken.
- If **data_dev_dest** and **data_host_src** both refer to shared memory and have the same value, no action is taken.
- If **data_dev_dest** and **data_host_src** both refer to shared memory and the memory regions overlap, the behavior is undefined.
- If the data referred to by **data_dev_dest** is not accessible by the current device, the behavior is undefined.
- If the data referred to by **data_host_src** is not accessible by the local thread, the behavior is undefined.
- Otherwise, **bytes** bytes of data at **data_host_src** in local memory are copied to **data_dev_dest** in the current device memory.

The **_async** version of this routine will perform the data transfers asynchronously on the async queue associated with **async_arg**. The routine may return before the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return until the data has been completely transferred.

Errors

- An **acc_error_invalid_null_pointer** error is issued if **data_dev_dest** or **data_host_src** is a null pointer and **bytes** is nonzero.
- An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-argument* value.

See Section 5.2.2.

3.2.27 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* data_host_dest,
                            d_void* data_dev_src, size_t bytes);
void acc_memcpy_from_device_async(h_void* data_host_dest,
                            d_void* data_dev_src, size_t bytes,
                            int async_arg);
```

Fortran:

```
subroutine acc_memcpy_from_device(data_host_dest,
                            data_dev_src, bytes)
subroutine acc_memcpy_from_device_async(data_host_dest,
                            data_dev_src, bytes, async_arg)
 type(*),dimension(*) :: data_host_dest
 type(c_ptr), value :: data_dev_src
 integer(c_size_t), value :: bytes
 integer(acc_handle_kind), value :: async_arg
```

Description

The **acc_memcpy_from_device** routine copies **bytes** bytes of data from the device address in **data_dev_src** to the local address in **data_host_dest**. **data_dev_src** must be an address accessible from the current device, such as an address returned from **acc_malloc** or **acc_deviceptr**, or an address in shared memory.

The behavior of the **acc_memcpy_from_device** routines is:

- If **bytes** is zero, no action is taken.
- If **data_host_dest** and **data_dev_src** both refer to shared memory and have the same value, no action is taken.
- If **data_host_dest** and **data_dev_src** both refer to shared memory and the memory regions overlap, the behavior is undefined.
- If the data referred to by **data_dev_src** is not accessible by the current device, the behavior is undefined.
- If the data referred to by **data_host_dest** is not accessible by the local thread, the behavior is undefined.
- Otherwise, **bytes** bytes of data at **data_dev_src** in the current device memory are copied to **data_host_dest** in local memory.

The **_async** version of this routine will perform the data transfers asynchronously on the async queue associated with **async_arg**. The routine may return before the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return until the data has been completely transferred.

Errors

- An **acc_error_invalid_null_pointer** error is issued if **data_host_dest** or **data_dev_src** is a null pointer and **bytes** is nonzero.

- An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-argument* value.

See Section 5.2.2.

3.2.28 acc_memcpy_device

Summary

The **acc_memcpy_device** routine copies data from one memory location to another memory location on the current device.

Format

C or C++:

```
void acc_memcpy_device(d_void* data_dev_dest,
                       d_void* data_dev_src, size_t bytes);
void acc_memcpy_device_async(d_void* data_dev_dest,
                       d_void* data_dev_src, size_t bytes,
                       int async_arg);
```

Fortran:

```
subroutine acc_memcpy_device(data_dev_dest,
                       data_dev_src, bytes);
subroutine acc_memcpy_device_async(data_dev_dest,
                       data_dev_src, bytes,
                       async_arg);
 type(c_ptr), value :: data_dev_dest
 type(c_ptr), value :: data_dev_src
 integer(c_size_t), value :: bytes
 integer(acc_handle_kind), value :: async_arg
```

Description

The **acc_memcpy_device** routine copies **bytes** bytes of data from the device address in **data_dev_src** to the device address in **data_dev_dest**. Both addresses must be addresses in the current device memory, such as would be returned from **acc_malloc** or **acc_deviceptr**.

The behavior of the **acc_memcpy_device** routines is:

- If **bytes** is zero, no action is taken.
- If **data_dev_dest** and **data_dev_src** have the same value, no action is taken.
- If the memory regions referred to by **data_dev_dest** and **data_dev_src** overlap, the behavior is undefined.
- If the data referred to by **data_dev_src** or **data_dev_dest** is not accessible by the current device, the behavior is undefined.
- Otherwise, **bytes** bytes of data at **data_dev_src** in the current device memory are copied to **data_dev_dest** in the current device memory.

The **_async** version of this routine will perform the data transfers asynchronously on the async queue associated with **async_arg**. The routine may return before the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return until the data has been completely transferred.

Errors

- An **acc_error_invalid_null_pointer** error is issued if **data_dev_dest** or **data_dev_src** is a null pointer and **bytes** is nonzero.
- An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-argument* value.

See Section 5.2.2.

3.2.29 acc_attach and acc_detach

Summary

The **acc_attach** routines update a pointer in device-accessible memory to point to the corresponding copy of the host pointer target. The **acc_detach** routines restore a pointer in device-accessible memory to point to the host pointer target.

Format

C or C++:

```
void acc_attach(h_void** ptr_addr);
void acc_attach_async(h_void** ptr_addr, int async_arg);

void acc_detach(h_void** ptr_addr);
void acc_detach_async(h_void** ptr_addr, int async_arg);
void acc_detach_finalize(h_void** ptr_addr);
void acc_detach_finalize_async(h_void** ptr_addr,
                               int async_arg);
```

Fortran:

```
subroutine acc_attach(ptr_addr)
subroutine acc_attach_async(ptr_addr, async_arg)
 type(*),dimension(..)           ::  ptr_addr
 integer(acc_handle_kind),value ::  async_arg

subroutine acc_detach(ptr_addr)
subroutine acc_detach_async(ptr_addr, async_arg)
subroutine acc_detach_finalize(ptr_addr)
subroutine acc_detach_finalize_async(ptr_addr,
                                      async_arg)
 type(*),dimension(..)           ::  ptr_addr
 integer(acc_handle_kind),value ::  async_arg
```

Description

A call to an **acc_attach** routine is functionally equivalent to an **enter data attach** directive, as described in Section 2.7.13. A call to an **acc_detach** routine is functionally equivalent to an **exit data detach** directive, and a call to an **acc_detach_finalize** routine is functionally equivalent to an **exit data finalize detach** directive, as described in Section 2.7.14. **ptr_addr** must be the address of a host pointer. **async_arg** must be an *async-argument* as defined in Section 2.16.

The behavior of these routines is:

- If **ptr_addr** refers to shared memory and does not refer to a captured variable, no action is taken.
- If the pointer referred to by **ptr_addr** is not present in device-accessible memory of the current device, no action is taken.
- Otherwise:
  - The **acc_attach** routines behave as follows,
    1. an *increment counter* action is performed on the associated attachment counter,
    2. if the associated attachment counter is now one, an *attach pointer* action is performed on the pointer referred to by **ptr_addr**; see Section 2.7.2.
  - The **acc_detach** routines behave as follows
    1. an *decrement counter* action is performed on the associated attachment counter,
    2. if the associated attachment counter is now zero, an *detach pointer* action is performed on the pointer referred to by **ptr_addr**; see Section 2.7.2.

    See Section 2.7.2.
  - The **acc_detach_finalize** routines behave as follows, perform a *detach pointer* action on the pointer referred to by **ptr_addr** followed by a *reset counter* action on the associated attachment counter; see Section 2.7.2.

These routines may issue a data transfer from local memory to device-accessible memory. The **_async** versions of these routines will perform the data transfers asynchronously on the async queue associated with **async_arg**. These routines may return before the data has been transferred; see Section 2.16 for more details. The synchronous versions will not return until the data has been completely transferred.

Errors

- An **acc_error_invalid_null_pointer** error is issued if **ptr_addr** is a null pointer.
- An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-argument* value.

See Section 5.2.2.

3.2.30 acc_memcpy_d2d

Summary

The **acc_memcpy_d2d** routines copy the contents of an array on one device to an array on the same or a different device without updating the value on the host.

Format

C or C++:

```
void acc_memcpy_d2d(h_void* data_arg_dest,
                    h_void* data_arg_src, size_t bytes,
                    int dev_num_dest, int dev_num_src);
void acc_memcpy_d2d_async(h_void* data_arg_dest,
```

```
                            h_void* data_arg_src, size_t bytes,
                            int dev_num_dest, int dev_num_src,
                            int async_arg_src);
```

Fortran:

```
    subroutine acc_memcpy_d2d(data_arg_dest, data_arg_src,&
                            bytes, dev_num_dest, dev_num_src)
    subroutine acc_memcpy_d2d_async(data_arg_dest, data_arg_src,&
                            bytes, dev_num_dest, dev_num_src,&
                            async_arg_src)
     type(*), dimension(..)  ::  data_arg_dest
     type(*), dimension(..)  ::  data_arg_src
     integer ::  bytes
     integer ::  dev_num_dest
     integer ::  dev_num_src
     integer ::  async_arg_src
```

Description

The **acc_memcpy_d2d** routines are passed the address of destination and source host data as well as integer device numbers for the destination and source devices, which must both be of the current device type.

The behavior of the **acc_memcpy_d2d** routines is:

- If **bytes** is zero, no action is taken.
- If both pointers have the same value and either the two device numbers are the same or the addresses are in shared memory, then no action is taken.
- Otherwise, **bytes** bytes of data at the device address corresponding to **data_arg_src** on device **dev_num_src** are copied to the device address corresponding to **data_arg_dest** on device **dev_num_dest**.

For **acc_memcpy_d2d_async** the value of **async_arg_src** is the number of an async queue on the source device. This routine will perform the data transfers asynchronously on the async queue associated with **async_arg_src** for device **dev_num_src**; see Section 2.16 Asynchronous Behavior for more details.

Errors

- An **acc_error_device_unavailable** error is issued if **dev_num_dest** or **dev_num_src** is not a valid device number.
- An **acc_error_invalid_null_pointer** error is issued if either **data_arg_dest** or **data_arg_src** is a null pointer and **bytes** is nonzero.
- An **acc_error_not_present** error is issued if the data at either address is not in shared memory and is not present in the respective device memory.
- An **acc_error_partly_present** error is issued if part of the data is already present in the current device memory but all of the data is not.

- An **`acc_error_invalid_async`** error is issued if **`async_arg`** is not a valid *async-argument* value.

See Section 5.2.2.

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 whitespace. 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 parallel, serial, 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
```

4.3 ACC_PROFLIB

The **ACC_PROFLIB** environment variable specifies the profiling library. More details about the evaluation at runtime is given in section 5.3.3 Runtime Dynamic Library Loading.

Example:

```
setenv ACC_PROFLIB /path/to/proflib/libaccprof.so
export ACC_PROFLIB=/path/to/proflib/libaccprof.so
```

5. Profiling and Error Callback Interface

This chapter describes the OpenACC interface for runtime callback routines. These routines may be provided by the programmer or by a tool or library developer. Calls to these routines are triggered during the application execution at specific OpenACC events. There are two classes of events, profiling events and error events. Profiling events can be used by tools for profile or trace data collection. Currently, this interface does not support tools that employ asynchronous sampling. Error events can be used to release resources or cleanly shut down a large parallel application when the OpenACC runtime detects an error condition from which it cannot recover. This is specifically for error handling, not for error recovery. There is no support provided for restarting or retrying an OpenACC program, construct, or API routine after an error condition has been detected and an error callback routine has been called.

In this chapter, the term *runtime* refers to the OpenACC runtime library. The term *library* refers to the routines invoked at specified events by the OpenACC runtime.

There are three steps for interfacing a *library* to the *runtime*. The first step is to write the library callback routines. Section 5.1 Events describes the supported runtime events and the order in which callbacks to the callback routines will occur. Section 5.2 Callbacks Signature describes the signature of the callback routines for all events.

The second step is to load the *library* at runtime. The *library* may be statically linked to the application or dynamically loaded by the application, a library, or a tool. This is described in Section 5.3 Loading the Library.

The third step is to register the desired callbacks with the events. This may be done explicitly by the application, if the library is statically linked with the application, implicitly by including a call to a registration routine in a **`.init`** section, or by including an initialization routine in the library if it is dynamically loaded by the *runtime*. This is described in Section 5.4 Registering Event Callbacks.

5.1 Events

This section describes the events that are recognized by the runtime. Most profiling events have a start and end callback routine, that is, a routine that is called just before the runtime code to handle the event starts and another routine that is called just after the event is handled. The event names and routine prototypes are available in the header file **`acc_callback.h`**, which is delivered with the OpenACC implementation. For backward compatibility with previous versions of OpenACC, the implementation also delivers the same information in **`acc_prof.h`**. Event names are prefixed with **`acc_ev_`**.

The ordering of events must reflect the order in which the OpenACC runtime actually executes them, i.e. if a runtime moves the enqueuing of data transfers or kernel launches outside the originating clauses/constructs, it needs to issue the corresponding launch callbacks when they really occur. A callback for a start event must always precede the matching end callback. No callbacks will be issued after a runtime shutdown event.

The events that the runtime supports can be registered with a callback and are defined in the enumeration type **`acc_event_t`**.

```
typedef enum acc_event_t{
    acc_ev_none = 0,
    acc_ev_device_init_start = 1,
    acc_ev_device_init_end = 2,
    acc_ev_device_shutdown_start = 3,
    acc_ev_device_shutdown_end = 4,
    acc_ev_runtime_shutdown = 5,
    acc_ev_create = 6,
    acc_ev_delete = 7,
    acc_ev_alloc = 8,
    acc_ev_free = 9,
    acc_ev_enter_data_start = 10,
    acc_ev_enter_data_end = 11,
    acc_ev_exit_data_start = 12,
    acc_ev_exit_data_end = 13,
    acc_ev_update_start = 14,
    acc_ev_update_end = 15,
    acc_ev_compute_construct_start = 16,
    acc_ev_compute_construct_end = 17,
    acc_ev_enqueue_launch_start = 18,
    acc_ev_enqueue_launch_end = 19,
    acc_ev_enqueue_upload_start = 20,
    acc_ev_enqueue_upload_end = 21,
    acc_ev_enqueue_download_start = 22,
    acc_ev_enqueue_download_end = 23,
    acc_ev_wait_start = 24,
    acc_ev_wait_end = 25,
    acc_ev_error = 100,
    acc_ev_last = 101
}acc_event_t;
```

The value of **acc_ev_last** will change if new events are added to the enumeration, so a library must not depend on that value.

5.1.1 Runtime Initialization and Shutdown

No callbacks can be registered for the runtime initialization. Instead the initialization of the tool is handled as described in Section 5.3 Loading the Library.

The *runtime shutdown* profiling event name is

```
acc_ev_runtime_shutdown
```

This event is triggered before the OpenACC runtime shuts down, either because all devices have been shutdown by calls to the **acc_shutdown** API routine, or at the end of the program.

5.1.2 Device Initialization and Shutdown

The *device initialization* profiling event names are

acc_ev_device_init_start
acc_ev_device_init_end

These events are triggered when a device is being initialized by the OpenACC runtime. This may be when the program starts, or may be later during execution when the program reaches an **acc_init** call or an OpenACC construct. The **acc_ev_device_init_start** is triggered before device initialization starts and **acc_ev_device_init_end** after initialization is complete.

The *device shutdown* profiling event names are

acc_ev_device_shutdown_start
acc_ev_device_shutdown_end

These events are triggered when a device is shut down, most likely by a call to the OpenACC **acc_shutdown** API routine. The **acc_ev_device_shutdown_start** is triggered before the device shutdown process starts and **acc_ev_device_shutdown_end** after the device shutdown is complete.

5.1.3 Enter Data and Exit Data

The *enter data* profiling event names are

acc_ev_enter_data_start
acc_ev_enter_data_end

These events are triggered at **enter data** directives, entry to data constructs, and entry to implicit data regions such as those generated by compute constructs. The **acc_ev_enter_data_start** event is triggered before any *data allocation*, *data update*, or *wait* events that are associated with that directive or region entry, and the **acc_ev_enter_data_end** is triggered after those events.

The *exit data* profiling event names are

acc_ev_exit_data_start
acc_ev_exit_data_end

These events are triggered at **exit data** directives, exit from **data** constructs, and exit from implicit data regions. The **acc_ev_exit_data_start** event is triggered before any *data deallocation*, *data update*, or *wait* events associated with that directive or region exit, and the **acc_ev_exit_data_end** event is triggered after those events.

When the construct that triggers an *enter data* or *exit data* event was generated implicitly by the compiler the **implicit** field in the event structure will be set to **1**. When the construct that triggers these events was specified explicitly by the application code the **implicit** field in the event structure will be set to **0**.

5.1.4 Data Allocation

The *data allocation* profiling event names are

acc_ev_create
acc_ev_delete
acc_ev_alloc
acc_ev_free

An **`acc_ev_alloc`** event is triggered when the OpenACC runtime allocates memory from the device memory pool, and an **`acc_ev_free`** event is triggered when the runtime frees that memory. An **`acc_ev_create`** event is triggered when the OpenACC runtime associates device memory with local memory, such as for a data clause (**`create`**, **`copyin`**, **`copy`**, **`copyout`**) at entry to a data construct, compute construct, at an **`enter data`** directive, or in a call to a data API routine (**`acc_copyin`**, **`acc_create`**, ...). An **`acc_ev_create`** event may be preceded by an **`acc_ev_alloc`** event, if newly allocated memory is used for this device data, or it may not, if the runtime manages its own memory pool. An **`acc_ev_delete`** event is triggered when the OpenACC runtime disassociates device memory from local memory, such as for a data clause at exit from a data construct, compute construct, at an **`exit data`** directive, or in a call to a data API routine (**`acc_copyout`**, **`acc_delete`**, ...). An **`acc_ev_delete`** event may be followed by an **`acc_ev_free`** event, if the disassociated device memory is freed, or it may not, if the runtime manages its own memory pool.

When the action that generates a *data allocation* event was generated explicitly by the application code the **`implicit`** field in the event structure will be set to **`0`**. When the *data allocation* event is triggered because of a variable or array with implicitly-determined data attributes or otherwise implicitly by the compiler the **`implicit`** field in the event structure will be set to **`1`**.

5.1.5 Data Construct

The profiling events for entering and leaving *data constructs* are mapped to *enter data* and *exit data* events as described in Section 5.1.3 Enter Data and Exit Data.

5.1.6 Update Directive

The *update directive* profiling event names are

```
acc_ev_update_start
acc_ev_update_end
```

The **`acc_ev_update_start`** event will be triggered at an **`update`** directive, before any *data update* or *wait* events that are associated with the update directive are carried out, and the corresponding **`acc_ev_update_end`** event will be triggered after any of the associated events.

5.1.7 Compute Construct

The *compute construct* profiling event names are

```
acc_ev_compute_construct_start
acc_ev_compute_construct_end
```

The **`acc_ev_compute_construct_start`** event is triggered at entry to a compute construct, before any *launch* events that are associated with entry to the compute construct. The **`acc_ev_compute_construct_end`** event is triggered at the exit of the compute construct, after any *launch* events associated with exit from the compute construct. If there are data clauses on the compute construct, those data clauses may be treated as part of the compute construct, or as part of a data construct containing the compute construct. The callbacks for data clauses must use the same line numbers as for the compute construct events.

5.1.8 Enqueue Kernel Launch

The *launch* profiling event names are

acc_ev_enqueue_launch_start
acc_ev_enqueue_launch_end

The **acc_ev_enqueue_launch_start** event is triggered just before an accelerator computation is enqueued for execution on a device, and **acc_ev_enqueue_launch_end** is triggered just after the computation is enqueued. Note that these events are synchronous with the local thread enqueueing the computation to a device, not with the device executing the computation. The **acc_ev_enqueue_launch_start** event callback routine is invoked just before the computation is enqueued, not just before the computation starts execution. More importantly, the **acc_ev_enqueue_launch_end** event callback routine is invoked after the computation is enqueued, not after the computation finished executing.

Note: Measuring the time between the start and end launch callbacks is often unlikely to be useful, since it will only measure the time to manage the launch queue, not the time to execute the code on the device.

5.1.9 Enqueue Data Update (Upload and Download)

The *data update* profiling event names are

acc_ev_enqueue_upload_start
acc_ev_enqueue_upload_end
acc_ev_enqueue_download_start
acc_ev_enqueue_download_end

The **_start** events are triggered just before each upload (data copy from local memory to device memory) operation is or download (data copy from device memory to local memory) operation is enqueued for execution on a device. The corresponding **_end** events are triggered just after each upload or download operation is enqueued.

Note: Measuring the time between the start and end update callbacks is often unlikely to be useful, since it will only measure the time to manage the enqueue operation, not the time to perform the actual upload or download.

When the action that generates a *data update* event was generated explicitly by the application code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event is triggered because of a variable or array with implicitly-determined data attributes or otherwise implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

5.1.10 Wait

The *wait* profiling event names are

acc_ev_wait_start
acc_ev_wait_end

An **acc_ev_wait_start** event will be triggered for each relevant queue before the local thread waits for that queue to be empty. A **acc_ev_wait_end** event will be triggered for each relevant queue after the local thread has determined that the queue is empty.

Wait events occur when the local thread and a device synchronize, either due to a **wait** directive or by a *wait* clause on a synchronous data construct, compute construct, or **enter data**, **exit data**, or **update** directive. For *wait* events triggered by an explicit synchronous **wait** directive or *wait* clause, the **implicit** field in the event structure will be **0**. For all other wait events, the **implicit** field in the event structure will be **1**.

The OpenACC runtime need not trigger *wait* events for queues that have not been used in the program, and need not trigger *wait* events for queues that have not been used by this thread since the last *wait* operation. For instance, an **acc wait** directive with no arguments is defined to wait on all queues. If the program only uses the default (synchronous) queue and the queue associated with **async(1)** and **async(2)** then an **acc wait** directive may trigger *wait* events only for those three queues. If the implementation knows that no activities have been enqueued on the **async(2)** queue since the last *wait* operation, then the **acc wait** directive may trigger *wait* events only for the default queue and the **async(1)** queue.

5.1.11 Error Event

The only error event is

acc_ev_error

An **acc_ev_error** event is triggered when the OpenACC program detects a runtime error condition. The default runtime error callback routine may print an error message and halt program execution. An application can register additional error event callback routines, to allow a failing application to release resources or to cleanly shut down a large parallel runtime with many threads and processes, for instance.

The application can register multiple alternate error callbacks. As described in Section 5.4.1 Multiple Callbacks, the callbacks will be invoked in the order in which they are registered. If all the error callbacks return, the default error callback will be invoked. The error callback routine must not execute any OpenACC compute or data constructs. The only OpenACC API routines that can be safely invoked from an error callback routine are **acc_get_property**, **acc_get_property_string**, and **acc_shutdown**.

5.2 Callbacks Signature

This section describes the signature of event callbacks. All event callbacks have the same signature. The routine prototypes are available in the header file **acc_callback.h**, which is delivered with the OpenACC implementation.

All callback routines have three arguments. The first argument is a pointer to a struct containing general information; the same struct type is used for all callback events. The second argument is a pointer to a struct containing information specific to that callback event; there is one struct type containing information for data events, another struct type containing information for kernel launch events, and a third struct type for other events, containing essentially no information. The third argument is a pointer to a struct containing information about the application programming interface (API) being used for the specific device. For NVIDIA CUDA devices, this contains CUDA-specific information; for OpenCL devices, this contains OpenCL-specific information. Other interfaces can be supported as they are added by implementations. The prototype for a callback routine is:

```
typedef void (*acc_callback)
    (acc_callback_info*, acc_event_info*, acc_api_info*);
typedef acc_callback acc_prof_callback;
```

In the descriptions, the datatype **ssize_t** means a signed 32-bit integer for a 32-bit binary and a 64-bit integer for a 64-bit binary, the datatype **size_t** means an unsigned 32-bit integer for a 32-bit binary and a 64-bit integer for a 64-bit binary, and the datatype **int** means a 32-bit integer for both 32-bit and 64-bit binaries.

5.2.1 First Argument: General Information

The first argument is a pointer to the **acc_callback_info** struct type:

```
typedef struct acc_prof_info{
    acc_event_t event_type;
    int valid_bytes;
    int version;
    acc_device_t device_type;
    int device_number;
    int thread_id;
    ssize_t async;
    ssize_t async_queue;
    const char* src_file;
    const char* func_name;
    int line_no, end_line_no;
    int func_line_no, func_end_line_no;
}acc_callback_info;
typedef struct acc_prof_info acc_prof_info;
```

The name **acc_prof_info** is preserved for backward compatibility with previous versions of OpenACC. The fields are described below.

- **acc_event_t event_type** - The event type that triggered this callback. The datatype is the enumeration type **acc_event_t**, described in the previous section. This allows the same callback routine to be used for different events.

- **int valid_bytes** - The number of valid bytes in this struct. This allows a library to interface with newer runtimes that may add new fields to the struct at the end while retaining compatibility with older runtimes. A runtime must fill in the **event_type** and **valid_bytes** fields, and must fill in values for all fields with offset less than **valid_bytes**. The value of **valid_bytes** for a struct is recursively defined as:

  ```
  valid_bytes(struct) = offset(lastfield) + valid_bytes(lastfield)
  valid_bytes(type[n]) = (n-1)*sizeof(type) + valid_bytes(type)
  valid_bytes(basictype) = sizeof(basictype)
  ```

- **int version** - A version number; the value of **_OPENACC**.

- **acc_device_t device_type** - The device type corresponding to this event. The datatype is **acc_device_t**, an enumeration type of all the supported device types, defined in **openacc.h**.

- **int device_number** - The device number. Each device is numbered, typically starting at device zero. For applications that use more than one device type, the device numbers may be unique across all devices or may be unique only across all devices of the same device type.

- **int thread_id** - The host thread ID making the callback. Host threads are given unique thread ID numbers typically starting at zero. This is not necessarily the same as the OpenMP thread number.
- **ssize_t async** - The *async-value* used for operations associated with this event; see Section 2.16 Asynchronous Behavior.
- **ssize_t async_queue** - The actual activity queue onto which the **async** field gets mapped; see Section 2.16 Asynchronous Behavior.
- **const char* src_file** - A pointer to null-terminated string containing the name of or path to the source file, if known, or a null pointer if not. If the library wants to save the source file name, it must allocate memory and copy the string.
- **const char* func_name** - A pointer to a null-terminated string containing the name of the function in which the event occurred, if known, or a null pointer if not. If the library wants to save the function name, it must allocate memory and copy the string.
- **int line_no** - The line number of the directive or program construct or the starting line number of the OpenACC construct corresponding to the event. A negative or zero value means the line number is not known.
- **int end_line_no** - For an OpenACC construct, this contains the line number of the end of the construct. A negative or zero value means the line number is not known.
- **int func_line_no** - The line number of the first line of the function named in **func_name**. A negative or zero value means the line number is not known.
- **int func_end_line_no** - The last line number of the function named in **func_name**. A negative or zero value means the line number is not known.

5.2.2 Second Argument: Event-Specific Information

The second argument is a pointer to the **acc_event_info** union type.

```
typedef union acc_event_info{
    acc_event_t event_type;
    acc_data_event_info data_event;
    acc_launch_event_info launch_event;
    acc_other_event_info other_event;
}acc_event_info;
```

The **event_type** field selects which union member to use. The first five members of each union member are identical. The second through fifth members of each union member (**valid_bytes**, **parent_construct**, **implicit**, and **tool_info**) have the same semantics for all event types:

- **int valid_bytes** - The number of valid bytes in the respective struct. (This field is similar used as discussed in Section 5.2.1 First Argument: General Information.)

- **acc_construct_t parent_construct** - This field describes the type of construct that caused the event to be emitted. The possible values for this field are defined by the **acc_construct_t** enum, described at the end of this section.
- **int implicit** - This field is set to 1 for any implicit event, such as an implicit wait at a synchronous data construct or synchronous enter data, exit data or update directive. This field is set to zero when the event is triggered by an explicit directive or call to a runtime API routine.
- **void* tool_info** - This field is used to pass tool-specific information from a **_start** event to the matching **_end** event. For a **_start** event callback, this field will be initialized to a null pointer. The value of this field for a **_end** event will be the value returned by the library in this field from the matching **_start** event callback, if there was one, or a null pointer otherwise. For events that are neither **_start** or **_end** events, this field will be a null pointer.

Data Events

For a data event, as noted in the event descriptions, the second argument will be a pointer to the **acc_data_event_info** struct.

```
typedef struct acc_data_event_info{
    acc_event_t event_type;
    int valid_bytes;
    acc_construct_t parent_construct;
    int implicit;
    void* tool_info;
    const char* var_name;
    size_t bytes;
    const void* host_ptr;
    const void* device_ptr;
}acc_data_event_info;
```

The fields specific for a data event are:

- **acc_event_t event_type** - The event type that triggered this callback. The events that use the **acc_data_event_info** struct are:

```
acc_ev_enqueue_upload_start
acc_ev_enqueue_upload_end
acc_ev_enqueue_download_start
acc_ev_enqueue_download_end
acc_ev_create
acc_ev_delete
acc_ev_alloc
acc_ev_free
```

- **const char* var_name** - A pointer to null-terminated string containing the name of the variable for which this event is triggered, if known, or a null pointer if not. If the library wants to save the variable name, it must allocate memory and copy the string.
- **size_t bytes** - The number of bytes for the data event.

- **const void* host_ptr** - If available and appropriate for this event, this is a pointer to the host data.
- **const void* device_ptr** - If available and appropriate for this event, this is a pointer to the corresponding device data.

Launch Events

For a launch event, as noted in the event descriptions, the second argument will be a pointer to the **acc_launch_event_info** struct.

```
typedef struct acc_launch_event_info{
    acc_event_t event_type;
    int valid_bytes;
    acc_construct_t parent_construct;
    int implicit;
    void* tool_info;
    const char* kernel_name;
    size_t num_gangs, num_workers, vector_length;
    size_t* num_gangs_per_dim;
}acc_launch_event_info;
```

The fields specific for a launch event are:

- **acc_event_t event_type** - The event type that triggered this callback. The events that use the **acc_launch_event_info** struct are:

  ```
  acc_ev_enqueue_launch_start
  acc_ev_enqueue_launch_end
  ```

- **const char* kernel_name** - A pointer to null-terminated string containing the name of the kernel being launched, if known, or a null pointer if not. If the library wants to save the kernel name, it must allocate memory and copy the string.
- **size_t num_gangs, num_workers, vector_length** - The number of gangs, workers, and vector lanes created for this kernel launch.
- **size_t* num_gangs_per_dim** - An array of **size_t** whose first element indicates the number of dimensions of gang parallelism and each subsequent element gives the number of gangs along each dimension starting with dimension 1. The product of the values of elements 1 through **num_gangs_per_dim[0]** is **num_gangs**.

Error Events

For an error event, as noted in the event descriptions, the second argument will be a pointer to the **acc_error_event_info** struct.

```
typedef struct acc_error_event_info{
    acc_event_t event_type;
    int valid_bytes;
    acc_construct_t parent_construct;
    int implicit;
    void* tool_info;
```

```
        acc_error_t error_code;
        const char* error_message;
        size_t runtime_info;
    }acc_error_event_info;
```

The enumeration type for the error code is

```
    typedef enum acc_error_t{
        acc_error_none = 0,
        acc_error_other = 1,
        acc_error_system = 2,
        acc_error_execution = 3,
        acc_error_device_init = 4,
        acc_error_device_shutdown = 5,
        acc_error_device_unavailable = 6,
        acc_error_device_type_unavailable = 7,
        acc_error_wrong_device_type = 8,
        acc_error_out_of_memory = 9,
        acc_error_not_present = 10,
        acc_error_partly_present = 11,
        acc_error_present = 12,
        acc_error_invalid_argument = 13,
        acc_error_invalid_async = 14,
        acc_error_invalid_null_pointer = 15,
        acc_error_invalid_data_section = 16,
        acc_error_implementation_defined = 100
    }acc_error_t;
```

The fields specific for an error event are:

- **acc_event_t event_type** - The event type that triggered this callback. The only event that uses the **acc_error_event_info** struct is:

  acc_ev_error

- **int implicit** - This will be set to 1.
- **acc_error_t error_code** - The error codes used are:
  - **acc_error_other** is used for error conditions other than those described below.
  - **acc_error_system** is used when there is a system error condition.
  - **acc_error_execution** is used when there is an error condition issued from code executing on the device.
  - **acc_error_device_init** is used for any error initializing a device.
  - **acc_error_device_shutdown** is used for any error shutting down a device.
  - **acc_error_device_unavailable** is used when there is an error where the selected device is unavailable.
  - **acc_error_device_type_unavailable** is used when there is an error where no device of the selected device type is available or is supported.

  - **acc_error_wrong_device_type** is used when there is an error related to the device type, such as a mismatch between the device type for which a compute construct was compiled and the device available at runtime.
  - **acc_error_out_of_memory** is used when the program tries to allocate more memory on the device than is available.
  - **acc_error_not_present** is used for an error related to data not being present at runtime.
  - **acc_error_partly_present** is used for an error related to part of the data being present but not being completely present at runtime.
  - **acc_error_present** is used for an error related to data being unexpectedly present at runtime.
  - **acc_error_invalid_argument** is used when an API routine is called with a invalid argument value, other than those described above.
  - **acc_error_invalid_async** is used when an API routine is called with an invalid *async-argument*, or when a directive is used with an invalid *async-argument*.
  - **acc_error_invalid_null_pointer** is used when an API routine is called with a null pointer argument where it is invalid, or when a directive is used with a null pointer in a context where it is invalid.
  - **acc_error_invalid_data_section** is used when an invalid array section appears in a directive data clause, or an invalid array section appears as a runtime API call argument.
  - **acc_error_implementation_defined**: any value greater or equal to this value may be used for an implementation-defined error code.

- **const char* error_message** - A pointer to null-terminated string containing an error message from the OpenACC runtime describing the error, or a null pointer.
- **size_t runtime_info** - A value, such as an error code, from the underlying device runtime or driver, if one is available and appropriate.

Other Events

For any event that does not use the **acc_data_event_info**, **acc_launch_event_info**, or **acc_error_event_info** struct, the second argument to the callback routine will be a pointer to **acc_other_event_info** struct.

```
typedef struct acc_other_event_info{
    acc_event_t event_type;
    int valid_bytes;
    acc_construct_t parent_construct;
    int implicit;
    void* tool_info;
}acc_other_event_info;
```

Parent Construct Enumeration

All event structures contain a **parent_construct** member that describes the type of construct that caused the event to be emitted. The purpose of this field is to provide a means to identify the type of construct emitting the event in the cases where an event may be emitted by multiple contruct types, such as is the case with data and wait events. The possible values for the **parent_construct** field are defined in the enumeration type **acc_construct_t**. In the case of combined directives, the outermost construct of the combined construct is specified as the **parent_construct**. If the event was emitted as the result of the application making a call to the runtime api, the value will be **acc_construct_runtime_api**.

```
typedef enum acc_construct_t{
    acc_construct_parallel = 0,
    acc_construct_serial = 16
    acc_construct_kernels = 1,
    acc_construct_loop = 2,
    acc_construct_data = 3,
    acc_construct_enter_data = 4,
    acc_construct_exit_data = 5,
    acc_construct_host_data = 6,
    acc_construct_atomic = 7,
    acc_construct_declare = 8,
    acc_construct_init = 9,
    acc_construct_shutdown = 10,
    acc_construct_set = 11,
    acc_construct_update = 12,
    acc_construct_routine = 13,
    acc_construct_wait = 14,
    acc_construct_runtime_api = 15,
}acc_construct_t;
```

5.2.3 Third Argument: API-Specific Information

The third argument is a pointer to the **acc_api_info** struct type, shown here.

```
typedef struct acc_api_info{
    acc_device_api device_api;
    int valid_bytes;
    acc_device_t device_type;
    int vendor;
    const void* device_handle;
    const void* context_handle;
    const void* async_handle;
}acc_api_info;
```

The fields are described below:

- **acc_device_api device_api** - The API in use for this device. The data type is the enumeration **acc_device_api**, which is described later in this section.
- **int valid_bytes** - The number of valid bytes in this struct. See the discussion above in

Section 5.2.1 First Argument: General Information.

- **acc_device_t device_type** - The device type; the datatype is **acc_device_t**, defined in **openacc.h**.
- **int vendor** - An identifier to identify the OpenACC vendor; contact your vendor to determine the value used by that vendor's runtime.
- **const void* device_handle** - If applicable, this will be a pointer to the API-specific device information.
- **const void* context_handle** - If applicable, this will be a pointer to the API-specific context information.
- **const void* async_handle** - If applicable, this will be a pointer to the API-specific async queue information.

According to the value of **device_api** a library can cast the pointers of the fields **device_handle**, **context_handle** and **async_handle** to the respective device API type. The following device APIs are defined in the interface below. Any implementation-defined device API type must have a value greater than **acc_device_api_implementation_defined**.

```
typedef enum acc_device_api{
    acc_device_api_none = 0,                       /* no device API    */
    acc_device_api_cuda = 1,                       /* CUDA driver API */
    acc_device_api_opencl = 2,                     /* OpenCL API      */
    acc_device_api_other = 4,                      /* other device API */
    acc_device_api_implementation_defined = 1000 /* other device API */
}acc_device_api;
```

5.3 Loading the Library

This section describes how a tools library is loaded when the program is run. Four methods are described.

- A tools library may be linked with the program, as any other library is linked, either as a static library or a dynamic library, and the runtime will call a predefined library initialization routine that will register the event callbacks.
- The OpenACC runtime implementation may support a dynamic tools library, such as a shared object for Linux or OS/X, or a DLL for Windows, which is then dynamically loaded at runtime under control of the environment variable **ACC_PROFLIB**.
- Some implementations where the OpenACC runtime is itself implemented as a dynamic library may support adding a tools library using the **LD_PRELOAD** feature in Linux.
- A tools library may be linked with the program, as in the first option, and the application itself may directly register event callback routines, or may invoke a library initialization routine that will register the event callbacks.

Callbacks are registered with the runtime by calling **acc_callback_register** for each event as described in Section 5.4 Registering Event Callbacks. The prototype for **acc_callback_register** is:

```
extern void acc_callback_register
        (acc_event_t event_type, acc_callback cb,
         acc_register_t info);
```

The first argument to **acc_callback_register** is the event for which a callback is being registered (compare Section 5.1 Events). The second argument is a pointer to the callback routine:

```
typedef void (*acc_callback)
        (acc_callback_info*,acc_event_info*,acc_api_info*);
```

The third argument is an enum type:

```
typedef enum acc_register_t{
    acc_reg = 0,
    acc_toggle = 1,
    acc_toggle_per_thread = 2
}acc_register_t;
```

This is usually **acc_reg**, but see Section 5.4.2 Disabling and Enabling Callbacks for cases where different values are used.

An example of registering callbacks for launch, upload, and download events is:

```
acc_callback_register(acc_ev_enqueue_launch_start,
        prof_launch, acc_reg);
acc_callback_register(acc_ev_enqueue_upload_start,
        prof_data, acc_reg);
acc_callback_register(acc_ev_enqueue_download_start,
        prof_data, acc_reg);
```

As shown in this example, the same routine (**prof_data**) can be registered for multiple events. The routine can use the **event_type** field in the **acc_callback_info** structure to determine for what event it was invoked.

The names **acc_prof_register** and **acc_prof_unregister** are preserved for backward compatibility with previous versions of OpenACC.

5.3.1 Library Registration

The OpenACC runtime will invoke **acc_register_library**, passing the addresses of the registration routines **acc_callback_register** and **acc_callback_unregister**, in case that routine comes from a dynamic library. In the third argument it passes the address of the lookup routine **acc_prof_lookup** to obtain the addresses of inquiry functions. No inquiry functions are defined in this profiling interface, but we preserve this argument for future support of sampling-based tools.

Typically, the OpenACC runtime will include a *weak* definition of **acc_register_library**, which does nothing and which will be called when there is no tools library. In this case, the library can save the addresses of these routines and/or make registration calls to register any appropriate callbacks. The prototype for **acc_register_library** is:

```
extern void acc_register_library
    (acc_prof_reg reg, acc_prof_reg unreg,
```

```
        acc_prof_lookup_func lookup);
```

The first two arguments of this routine are of type:

```
    typedef void (*acc_prof_reg)
        (acc_event_t event_type, acc_callback cb,
        acc_register_t info);
```

The third argument passes the address to the lookup function **acc_prof_lookup** to obtain the address of interface functions. It is of type:

```
    typedef void (*acc_query_fn)();
    typedef acc_query_fn (*acc_prof_lookup_func)
        (const char* acc_query_fn_name);
```

The argument of the lookup function is a string with the name of the inquiry function. There are no inquiry functions defined for this interface.

5.3.2 Statically-Linked Library Initialization

A tools library can be compiled and linked directly into the application. If the library provides an external routine **acc_register_library** as specified in Section 5.3.1Library Registration, the runtime will invoke that routine to initialize the library.

The sequence of events is:

1. The runtime invokes the **acc_register_library** routine from the library.
2. The **acc_register_library** routine calls **acc_callback_register** for each event to be monitored.
3. **acc_callback_register** records the callback routines.
4. The program runs, and your callback routines are invoked at the appropriate events.

In this mode, only one tool library is supported.

5.3.3 Runtime Dynamic Library Loading

A common case is to build the tools library as a dynamic library (shared object for Linux or OS/X, DLL for Windows). In that case, you can have the OpenACC runtime load the library during initialization. This allows you to enable runtime profiling without rebuilding or even relinking your application. The dynamic library must implement a registration routine **acc_register_library** as specified in Section 5.3.1 Library Registration.

The user may set the environment variable **ACC_PROFLIB** to the path to the library will tell the OpenACC runtime to load your dynamic library at initialization time:

Bash:

```
    export ACC_PROFLIB=/home/user/lib/myprof.so
    ./myapp
```

or

```
    ACC_PROFLIB=/home/user/lib/myprof.so ./myapp
```

C-shell:

```
setenv ACC_PROFLIB /home/user/lib/myprof.so
./myapp
```

When the OpenACC runtime initializes, it will read the **ACC_PROFLIB** environment variable (with **getenv**). The runtime will open the dynamic library (using **dlopen** or **LoadLibraryA**); if the library cannot be opened, the runtime may cause the program to halt execution and return an error status, or may continue execution with or without an error message. If the library is successfully opened, the runtime will get the address of the **acc_register_library** routine (using **dlsym** or **GetProcAddress**). If this routine is resolved in the library, it will be invoked passing in the addresses of the registration routine **acc_callback_register**, the deregistration routine **acc_callback_unregister**, and the lookup routine **acc_prof_lookup**. The registration routine in your library, **acc_register_library**, registers the callbacks by calling the **register** argument, and must save the addresses of the arguments (**register**, **unregister**, and **lookup**) for later use, if needed.

The sequence of events is:

1. Initialization of the OpenACC runtime.
2. OpenACC runtime reads **ACC_PROFLIB**.
3. OpenACC runtime loads the library.
4. OpenACC runtime calls the **acc_register_library** routine in that library.
5. Your **acc_register_library** routine calls **acc_callback_register** for each event to be monitored.
6. **acc_callback_register** records the callback routines.
7. The program runs, and your callback routines are invoked at the appropriate events.

If supported, paths to multiple dynamic libraries may be specified in the **ACC_PROFLIB** environment variable, separated by semicolons (**;**). The OpenACC runtime will open these libraries and invoke the **acc_register_library** routine for each, in the order they appear in **ACC_PROFLIB**.

5.3.4 Preloading with LD_PRELOAD

The implementation may also support dynamic loading of a tools library using the **LD_PRELOAD** feature available in some systems. In such an implementation, you need only specify your tools library path in the **LD_PRELOAD** environment variable before executing your program. The OpenACC runtime will invoke the **acc_register_library** routine in your tools library at initialization time. This requires that the OpenACC runtime include a dynamic library with a default (empty) implementation of **acc_register_library** that will be invoked in the normal case where there is no **LD_PRELOAD** setting. If an implementation only supports static linking, or if the application is linked without dynamic library support, this feature will not be available.

Bash:

```
export LD_PRELOAD=/home/user/lib/myprof.so
./myapp
```

or

```
LD_PRELOAD=/home/user/lib/myprof.so ./myapp
```

C-shell:

```
setenv LD_PRELOAD /home/user/lib/myprof.so
./myapp
```

The sequence of events is:

1. The operating system loader loads the library specified in **LD_PRELOAD**.
2. The call to **acc_register_library** in the OpenACC runtime is resolved to the routine in the loaded tools library.
3. OpenACC runtime calls the **acc_register_library** routine in that library.
4. Your **acc_register_library** routine calls **acc_callback_register** for each event to be monitored.
5. **acc_callback_register** records the callback routines.
6. The program runs, and your callback routines are invoked at the appropriate events.

In this mode, only a single tools library is supported, since only one **acc_register_library** initialization routine will get resolved by the dynamic loader.

5.3.5 Application-Controlled Initialization

An alternative to default initialization is to have the application itself call the library initialization routine, which then calls **acc_callback_register** for each appropriate event. The library may be statically linked to the application or your application may dynamically load the library.

The sequence of events is:

1. Your application calls the library initialization routine.
2. The library initialization routine calls **acc_callback_register** for each event to be monitored.
3. **acc_callback_register** records the callback routines.
4. The program runs, and your callback routines are invoked at the appropriate events.

In this mode, multiple tools libraries can be supported, with each library initialization routine invoked by the application.

5.4 Registering Event Callbacks

This section describes how to register and unregister callbacks, temporarily disabling and enabling callbacks, the behavior of dynamic registration and unregistration, and requirements on an OpenACC implementation to correctly support the interface.

5.4.1 Event Registration and Unregistration

The library must call the registration routine **acc_callback_register** to register each callback with the runtime. A simple example:

```
extern void prof_data(acc_callback_info* profinfo,
        acc_event_info* eventinfo, acc_api_info* apiinfo);
```

```
extern void prof_launch(acc_callback_info* profinfo,
      acc_event_info* eventinfo, acc_api_info* apiinfo);
...
void acc_register_library(acc_prof_reg reg,
      acc_prof_reg unreg, acc_prof_lookup_func lookup){
   reg(acc_ev_enqueue_upload_start, prof_data, acc_reg);
   reg(acc_ev_enqueue_download_start, prof_data, acc_reg);
   reg(acc_ev_enqueue_launch_start, prof_launch, acc_reg);
}
```

In this example the **prof_data** routine will be invoked for each data upload and download event, and the **prof_launch** routine will be invoked for each launch event. The **prof_data** routine might start out with:

```
void prof_data(acc_callback_info* profinfo,
      acc_event_info* eventinfo, acc_api_info* apiinfo){
   acc_data_event_info* datainfo;
   datainfo = (acc_data_event_info*)eventinfo;
   switch( datainfo->event_type ){
      case acc_ev_enqueue_upload_start :
      ...
   }
}
```

Multiple Callbacks

Multiple callback routines can be registered on the same event:

```
acc_callback_register(acc_ev_enqueue_upload_start,
      prof_data, acc_reg);
acc_callback_register(acc_ev_enqueue_upload_start,
      prof_up, acc_reg);
```

For most events, the callbacks will be invoked in the order in which they are registered. However, *end* events, named **acc_ev_..._end**, invoke callbacks in the reverse order. Essentially, each event has an ordered list of callback routines. A new callback routine is appended to the tail of the list for that event. For most events, that list is traversed from the head to the tail, but for *end* events, the list is traversed from the tail to the head.

If a callback is registered, then later unregistered, then later still registered again, the second registration is considered to be a new callback, and the callback routine will then be appended to the tail of the callback list for that event.

Unregistering

A matching call to **acc_callback_unregister** will remove that routine from the list of callback routines for that event.

```
acc_callback_register(acc_ev_enqueue_upload_start,
      prof_data, acc_reg);
// prof_data is on the callback list for acc_ev_enqueue_upload_start
```

```
...
acc_callback_unregister(acc_ev_enqueue_upload_start,
        prof_data, acc_reg);
// prof_data is removed from the callback list
//  for acc_ev_enqueue_upload_start
```

Each entry on the callback list must also have a *ref* count. This keeps track of how many times this routine was added to this event's callback list. If a routine is registered *n* times, it must be unregistered *n* times before it is removed from the list. Note that if a routine is registered multiple times for the same event, its *ref* count will be incremented with each registration, but it will only be invoked once for each event instance.

5.4.2 Disabling and Enabling Callbacks

A callback routine may be temporarily disabled on the callback list for an event, then later re-enabled. The behavior is slightly different than unregistering and later re-registering that event. When a routine is disabled and later re-enabled, the routine's position on the callback list for that event is preserved. When a routine is unregistered and later re-registered, the routine's position on the callback list for that event will move to the tail of the list. Also, unregistering a callback must be done *n* times if the callback routine was registered *n* times. In contrast, disabling, and enabling an event sets a toggle. Disabling a callback will immediately reset the toggle and disable calls to that routine for that event, even if it was enabled multiple times. Enabling a callback will immediately set the toggle and enable calls to that routine for that event, even if it was disabled multiple times. Registering a new callback initially sets the toggle.

A call to **acc_callback_unregister** with a value of **acc_toggle** as the third argument will disable callbacks to the given routine. A call to **acc_callback_register** with a value of **acc_toggle** as the third argument will enable those callbacks.

```
acc_callback_unregister(acc_ev_enqueue_upload_start,
        prof_data, acc_toggle);
// prof_data is disabled
...
acc_callback_register(acc_ev_enqueue_upload_start,
        prof_data, acc_toggle);
// prof_data is re-enabled
```

A call to either **acc_callback_unregister** or **acc_callback_register** to disable or enable a callback when that callback is not currently registered for that event will be ignored with no error.

All callbacks for an event may be disabled (and re-enabled) by passing **NULL** to the second argument and **acc_toggle** to the third argument of **acc_callback_unregister** (and **acc_callback_register**). This sets a toggle for that event, which is distinct from the toggle for each callback for that event. While the event is disabled, no callbacks for that event will be invoked. Callbacks for that event can be registered, unregistered, enabled, and disabled while that event is disabled, but no callbacks will be invoked for that event until the event itself is enabled. Initially, all events are enabled.

```
acc_callback_unregister(acc_ev_enqueue_upload_start,
        prof_data, acc_toggle);
```

```
    // prof_data is disabled
    ...
    acc_callback_unregister(acc_ev_enqueue_upload_start,
            NULL, acc_toggle);
    // acc_ev_enqueue_upload_start callbacks are disabled
    ...
    acc_callback_register(acc_ev_enqueue_upload_start,
            prof_data, acc_toggle);
    // prof_data is re-enabled, but
    // acc_ev_enqueue_upload_start callbacks still disabled
    ...
    acc_callback_register(acc_ev_enqueue_upload_start,
            prof_up, acc_reg);
    // prof_up is registered and initially enabled, but
    // acc_ev_enqueue_upload_start callbacks still disabled
    ...
    acc_callback_register(acc_ev_enqueue_upload_start,
            NULL, acc_toggle);
    // acc_ev_enqueue_upload_start callbacks are enabled
```

Finally, all callbacks can be disabled (and enabled) by passing the argument list **(acc_ev_none, NULL, acc_toggle)** to **acc_callback_unregister** (and **acc_callback_register**). This sets a global toggle disabling all callbacks, which is distinct from the toggle enabling callbacks for each event and the toggle enabling each callback routine.

The behavior of passing **acc_ev_none** as the first argument and a non-**NULL** value as the second argument to **acc_callback_unregister** or **acc_callback_register** is not defined, and may be ignored by the runtime without error.

All callbacks can be disabled (or enabled) for just the current thread by passing the argument list **(acc_ev_none, NULL, acc_toggle_per_thread)** to **acc_callback_unregister** (and **acc_callback_register**). This is the only thread-specific interface to **acc_callback_register** and **acc_callback_unregister**, all other calls to register, unregister, enable, or disable callbacks affect all threads in the application.

5.5 Advanced Topics

This section describes advanced topics such as dynamic registration and changes of the execution state for callback routines as well as the runtime and tool behavior for multiple host threads.

5.5.1 Dynamic Behavior

Callback routines may be registered or unregistered, enabled or disabled at any point in the execution of the program. Calls may appear in the library itself, during the processing of an event. The OpenACC runtime must allow for this case, where the callback list for an event is modified while that event is being processed.

Dynamic Registration and Unregistration

Calls to **acc_register** and **acc_unregister** may occur at any point in the application. A callback routine can be registered or unregistered from a callback routine, either the same routine or another routine, for a different event or the same event for which the callback was invoked. If a callback routine is registered for an event while that event is being processed, then the new callback routine will be added to the tail of the list of callback routines for this event. Some events (the **_end**) events process the callback routines in reverse order, from the tail to the head. For those events, adding a new callback routine will not cause the new routine to be invoked for this instance of the event. The other events process the callback routines in registration order, from the head to the tail. Adding a new callback routine for such an event will cause the runtime to invoke that newly registered callback routine for this instance of the event. Both the runtime and the library must implement and expect this behavior.

If an existing callback routine is unregistered for an event while that event is being processed, that callback routine is removed from the list of callbacks for this event. For any event, if that callback routine had not yet been invoked for this instance of the event, it will not be invoked.

Registering and unregistering a callback routine is a global operation and affects all threads, in a multithreaded application. See Section 5.4.1 Multiple Callbacks.

Dynamic Enabling and Disabling

Calls to **acc_register** and **acc_unregister** to enable and disable a specific callback for an event, enable or disable all callbacks for an event, or enable or disable all callbacks may occur at any point in the application. A callback routine can be enabled or disabled from a callback routine, either the same routine or another routine, for a different event or the same event for which the callback was invoked. If a callback routine is enabled for an event while that event is being processed, then the new callback routine will be immediately enabled. If it appears on the list of callback routines closer to the head (for **_end** events) or closer to the tail (for other events), that newly-enabled callback routine will be invoked for this instance of this event, unless it is disabled or unregistered before that callback is reached.

If a callback routine is disabled for an event while that event is being processed, that callback routine is immediately disabled. For any event, if that callback routine had not yet been invoked for this instance of the event, it will not be invoked, unless it is enabled before that callback routine is reached in the list of callbacks for this event. If all callbacks for an event are disabled while that event is being processed, or all callbacks are disabled for all events while an event is being processed, then when this callback routine returns, no more callbacks will be invoked for this instance of the event.

Registering and unregistering a callback routine is a global operation and affects all threads, in a multithreaded application. See Section 5.4.1 Multiple Callbacks.

5.5.2 OpenACC Events During Event Processing

OpenACC events may occur during event processing. This may be because of OpenACC API routine calls or OpenACC constructs being reached during event processing, or because of multiple host threads executing asynchronously. Both the OpenACC runtime and the tool library must implement the proper behavior.

5.5.3 Multiple Host Threads

Many programs that use OpenACC also use multiple host threads, such as programs using the OpenMP API. The appearance of multiple host threads affects both the OpenACC runtime and the tools library.

Runtime Support for Multiple Threads

The OpenACC runtime must be thread-safe, and the OpenACC runtime implementation of this tools interface must also be thread-safe. All threads use the same set of callbacks for all events, so registering a callback from one thread will cause all threads to execute that callback. This means that managing the callback lists for each event must be protected from multiple simultaneous updates. This includes adding a callback to the tail of the callback list for an event, removing a callback from the list for an event, and incrementing or decrementing the *ref* count for a callback routine for an event.

In addition, one thread may register, unregister, enable, or disable a callback for an event while another thread is processing the callback list for that event asynchronously. The exact behavior may be dependent on the implementation, but some behaviors are expected and others are disallowed. In the following examples, there are three callbacks, A, B, and C, registered for event E in that order, where callbacks A and B are enabled and callback C is temporarily disabled. Thread T1 is dynamically modifying the callbacks for event E while thread T2 is processing an instance of event E.

- Suppose thread T1 unregisters or disables callback A for event E. Thread T2 may or may not invoke callback A for this event instance, but it must invoke callback B; if it invokes callback A, that must precede the invocation of callback B.
- Suppose thread T1 unregisters or disables callback B for event E. Thread T2 may or may not invoke callback B for this event instance, but it must invoke callback A; if it invokes callback B, that must follow the invocation of callback A.
- Suppose thread T1 unregisters or disables callback A and then unregisters or disables callback B for event E. Thread T2 may or may not invoke callback A and may or may not invoke callback B for this event instance, but if it invokes both callbacks, it must invoke callback A before it invokes callback B.
- Suppose thread T1 unregisters or disables callback B and then unregisters or disables callback A for event E. Thread T2 may or may not invoke callback A and may or may not invoke callback B for this event instance, but if it invokes callback B, it must have invoked callback A for this event instance.
- Suppose thread T1 is registering a new callback D for event E. Thread T2 may or may not invoke callback D for this event instance, but it must invoke both callbacks A and B. If it invokes callback D, that must follow the invocations of A and B.
- Suppose thread T1 is enabling callback C for event E. Thread T2 may or may not invoke callback C for this event instance, but it must invoke both callbacks A and B. If it invokes callback C, that must follow the invocations of A and B.

The **`acc_callback_info`** struct has a **`thread_id`** field, which the runtime must set to a unique value for each host thread, though it need not be the same as the OpenMP threadnum value.

Library Support for Multiple Threads

The tool library must also be thread-safe. The callback routine will be invoked in the context of the thread that reaches the event. The library may receive a callback from a thread T2 while it's still processing a callback, from the same event type or from a different event type, from another thread T1. The **acc_callback_info** struct has a **thread_id** field, which the runtime must set to a unique value for each host thread.

If the tool library uses dynamic callback registration and unregistration, or callback disabling and enabling, recall that unregistering or disabling an event callback from one thread will unregister or disable that callback for all threads, and registering or enabling an event callback from any thread will register or enable it for all threads. If two or more threads register the same callback for the same event, the behavior is the same as if one thread registered that callback multiple times; see Section 5.4.1 Multiple Callbacks. The **acc_unregister** routine must be called as many times as **acc_register** for that callback/event pair in order to totally unregister it. If two threads register two different callback routines for the same event, unless the order of the registration calls is guaranteed by some sychronization method, the order in which the runtime sees the registration may differ for multiple runs, meaning the order in which the callbacks occur will differ as well.

6. 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. In particular, some terms used in this specification conflict with their usage in the base language specifications. When there is potential confusion, the term will appear here.

Accelerator – a device attached to a CPU and to which the CPU can offload data and compute kernels to perform compute-intensive calculations.

Accelerator routine – a procedure 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.

Aggregate datatype – any non-scalar datatype such as array and composite datatypes. In Fortran, aggregate datatypes include arrays, derived types, character types. In C, aggregate datatypes include arrays, targets of pointers, structs, and unions. In C++, aggregate datatypes include arrays, targets of pointers, classes, structs, and unions.

Aggregate variables – a variable of any non-scalar datatype, including array or composite variables. In Fortran, this includes any variable with allocatable or pointer attribute and character variables.

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

Block construct – a *block-construct*, as specified by the Fortran language.

Captured variable – a variable for which a distinct copy from its original variable exists in the device-accessible memory. Such variable is only captured from the time its copy is created and until such a copy is deleted.

Composite datatype – a derived type in Fortran, or a **`struct`** or **`union`** type in C, or a **`class`**, **`struct`**, or **`union`** type in C++. (This is different from the use of the term *composite data type* in the C and C++ languages.)

Composite variable – a variable of composite datatype. In Fortran, a composite variable must not have allocatable or pointer attributes.

Compute construct – a *parallel construct*, *serial construct*, or *kernels construct*.

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.

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

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

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

Current device – the device represented by the *acc-current-device-type-var* and *acc-current-device-num-var* ICVs

Current device type – the device type represented by the *acc-current-device-type-var* ICV

Data lifetime – the lifetime of a data object in device memory, which may 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 a **data** construct, or an implicit data region for a function or subroutine containing OpenACC directives. Data constructs typically allocate device memory and copy data from host to device memory upon entry, and copy data from device to local memory and deallocate device memory upon exit. Data regions may contain other data regions and compute regions.

Default asynchronous queue – the asynchronous activity queue represented in the *acc-default-async-var* ICV

Device – a general reference to an accelerator or a multicore CPU.

Device-accessible memory – any memory which can be accessed from the device.

Device memory – memory attached to a device, logically and physically separate from the host memory.

Device thread – a thread of execution that executes on any device.

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.

Discrete memory – memory accessible from the local thread that is not accessible from the current device, or memory accessible from the current device that is not accessible from the local thread.

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.

Exposed variable access – with respect to a compute construct, any access to the data or address of a variable at a point within the compute construct where the variable is not private to a scope lexically enclosed within the compute construct. See Section 2.6.2.

false – a condition that evaluates to zero in C or C++, or **.false.** in Fortran.

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

GPGPU – General Purpose computation on Graphics Processing Units.

Host – the main CPU that in this context may have one or more attached accelerators. The host CPU controls the program regions and data loaded into and executed on one or more devices.

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 a `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 space in device memory to be allocated and data to be copied from local memory to device memory upon region entry, and data to be copied from device memory to local memory and space in device memory to be deallocated upon exit.

Level of parallelism – one of the following, which are arranged from the highest to the lowest level: gang dimension three, gang dimension two, gang dimension one, worker, vector, or sequential. One or more of gang, worker, and vector parallelism may appear on a loop construct. Sequential execution corresponds to no parallelism. The `gang`, `worker`, `vector`, and `seq` clauses specify the level of parallelism for a loop.

Local device – the device where the *local thread* executes.

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.

null pointer – a C or C++ pointer variable with the value zero, `NULL`, or (in C++) `nullptr`, or a Fortran `pointer` variable that is not associated, or a Fortran `allocatable` variable that is not allocated.

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.

Orphaned loop construct – a `loop` construct that has no parent compute construct.

Parallel region – a *region* defined by a `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 space in device memory to be allocated and data to be copied from local memory to device memory upon region entry, and data to be copied from device memory to local memory and space in device memory to be deallocated upon exit.

Parent compute construct – for any point in the program, the nearest lexically enclosing compute construct that has the same parent procedure.

Parent compute scope – for any point in the program, the parent compute construct or, if none, the parent procedure.

Parent procedure – for any point in the program, the nearest lexically enclosing procedure such that expressions at this point are not evaluated until the procedure is called.

Partly present data – a section of data for which some of the data is present in a single device memory section, but part of the data is either not present or is present in a different device memory section. For instance, if a subarray of an array is present, the array is partly present.

Present data – data for which the sum of the structured and dynamic reference counters is greater than zero in a single device memory section; see Section 2.6.7. A null pointer is defined as always present with a length of zero bytes.

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 or C++ lambda; 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*, *serial region*, *kernels region*, *data region*, or *implicit data region*.

Scalar – a variable of scalar datatype. In Fortran, scalars must not have allocatable or pointer attributes.

Scalar datatype – an intrinsic or built-in datatype that is not an array or aggregate datatype. In Fortran, scalar datatypes are integer, real, double precision, complex, or logical. In C, scalar datatypes are char (signed or unsigned), int (signed or unsigned, with optional short, long or long long attribute), enum, float, double, long double, _Complex (with optional float or long attribute), or any pointer datatype. In C++, scalar datatypes are char (signed or unsigned), wchar_t, int (signed or unsigned, with optional short, long or long long attribute), enum, bool, float, double, long double, or any pointer datatype. Not all implementations or targets will support all of these datatypes.

Serial region – a *region* defined by a `serial` construct. A serial region is a structured block which is compiled for the accelerator. A serial region contains code that is executed by a single gang of a single worker with a vector length of one. A serial region may require space in device memory to be allocated and data to be copied from local memory to device memory upon region entry, and data to be copied from device memory to local memory and space in device memory to be deallocated upon exit.

Shared memory – memory that is accessible from both the local thread and the current device.

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 – a host CPU thread or an accelerator thread. On a host CPU, 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.

true – a condition that evaluates to nonzero in C or C++, or `.true.` in Fortran.

var – the name of a variable (scalar, array, or composite variable), or a subarray specification, or an array element, or a composite variable member, or the name of a Fortran common block between slashes.

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

Visible data clause – with respect to a compute construct, any data clause on the compute construct, on a lexically enclosing **`data`** construct that has the same parent procedure, or on a visible **`declare`** directive. See Section 2.6.2.

Visible `default` clause – with respect to a compute construct, the nearest **`default`** clause appearing on the compute construct or on a lexically enclosing **`data`** construct that has the same parent procedure. See Section 2.6.2.

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

A. Recommendations for Implementers

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, and recommended options that programmers find useful. 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 devices, such as NVIDIA GPUs. 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.

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.

ACC_DEVICE_TYPE

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

device_type clause argument

An implementation should use the case-insensitive name **`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.

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.

ACC_DEVICE_TYPE

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

device_type clause argument

An implementation should use the case-insensitive name **`radeon`** as the argument to the **`device_type`** clause.

A.1.3 Multicore Host CPU Target

This section gives recommendations for implementations that target the multicore host CPU.

Accelerator Device Type

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

ACC_DEVICE_TYPE

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

device_type clause argument

An implementation should use the case-insensitive name **host** as the argument to the **device_type** clause.

routine directive

Given a **routine** directive for a procedure, an implementation should:

- Suppress the procedure's compilation for the multicore host CPU if a **nohost** clause appears.
- Ignore any **bind** clause when compiling the procedure for the multicore host CPU.
- Disallow a **bind** clause to appear after a **device_type(host)** 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.

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 ();
```

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 ();
```

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 asynchronous activity queue associated with the **async** argument. This argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

Format

C or C++:

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

acc_set_cuda_stream

Summary

The **acc_set_cuda_stream** routine sets the NVIDIA CUDA stream handle the current device for the asynchronous activity queue associated with the **async** argument. This argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

Format

C or C++:

```
void 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.

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 ();
```

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 ();
```

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 asynchronous activity queue associated with the **async** argument. This argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

Format

C or C++:

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

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 asynchronous activity queue associated with the **async** argument. This argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

Format

C or C++:

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

A.3 Recommended Options and Diagnostics

This section recommends options and diagnostics for implementations. Possible ways to implement the options include 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 in the current device memory. Implementations before this revision commonly implemented this by testing whether the pointer target **p[0]** was present in the current device memory, 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 Nonconforming Applications and Implementations

Where feasible, implementations should diagnose OpenACC applications that do not conform with this specification's syntactic or semantic restrictions. Many but not all of these restrictions appear in lists entitled "Restrictions."

While compile-time diagnostics are preferable (e.g., invalid clauses on a directive), some cases of nonconformity are more feasible to diagnose at run time (e.g., see Section 1.5). Where implementations are not able to diagnose nonconformity reliably (e.g., an **independent** clause on a loop with data-dependent loop iterations), they might offer no diagnostics, or they might diagnose only subcases.

In order to support OpenACC extensions, some implementations intentionally accept nonconforming OpenACC applications without issuing diagnostics by default, and some implementations accept conforming OpenACC applications but interpret their semantics differently than as detailed in this specification. To promote program portability across implementations, implementations should provide an option to disable or report uses of these extensions. Some such extensions and diagnostics are described in detail in the remainder of this section.

A.3.3 Automatic Data Attributes

Some implementations provide autoscoping or other analysis to automatically determine a variable's data attributes, including the addition of reduction, private, and firstprivate clauses. To promote program portability across implementations, it would be helpful to provide an option to disable the automatic determination of data attributes or report which variables' data attributes are not as defined in Section 2.6.

A.3.4 Routine Directive with a Name

In C and C++, if a **routine** directive with a name appears immediately before a procedure declaration or definition with that name, it does not necessarily apply to that procedure according to Section 2.15.1 and C and C++ name resolution. Implementations should issue diagnostics in the following two cases:

1. When no procedure with that name is already in scope, the directive is nonconforming, so implementations should issue a compile-time error diagnostic regardless of the following procedure. For example:

   ```
   #pragma acc routine(f) seq // compile-time error
   void f();
   ```

2. When a procedure with that name is in scope and it is not the same procedure as the immediately following procedure declaration or definition, the resolution of the name can be confusing. Implementations should then issue a compile-time warning diagnostic even though the application is conforming. For example:

   ```
   void g(); // routine directive applies
   namespace NS {
     #pragma acc routine(g) seq // compile-time warning
     void g(); // routine directive does not apply
   }
   ```

The diagnostic in this case should suggest the programmer either (1) relocate the **`routine`** directive so that it more clearly applies to the procedure that is in scope or (2) remove the name from the **`routine`** directive so that it applies to the following procedure.

Index