

6 Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright,
7 no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form
8 or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express
9 written permission of the authors.

10 © 2011-2021 OpenACC-Standard.org. All rights reserved.

11 Contents

12	1. Introduction	9
13	1.1. Scope	9
14	1.2. Execution Model	9
15	1.3. Memory Model	11
16	1.4. Language Interoperability	12
17	1.5. Runtime Errors	13
18	1.6. Conventions used in this document	13
19	1.7. Organization of this document	14
20	1.8. References	14
21	1.9. Changes from Version 1.0 to 2.0	16
22	1.10. Corrections in the August 2013 document	17
23	1.11. Changes from Version 2.0 to 2.5	17
24	1.12. Changes from Version 2.5 to 2.6	19
25	1.13. Changes from Version 2.6 to 2.7	19
26	1.14. Changes from Version 2.7 to 3.0	20
27	1.15. Changes from Version 3.0 to 3.1	21
28	1.16. Changes from Version 3.1 to 3.2	22
29	1.17. Topics Deferred For a Future Revision	24
30	2. Directives	27
31	2.1. Directive Format	27
32	2.2. Conditional Compilation	28
33	2.3. Internal Control Variables	28
34	2.3.1. Modifying and Retrieving ICV Values	28
35	2.4. Device-Specific Clauses	29
36	2.5. Compute Constructs	30
37	2.5.1. Parallel Construct	30
38	2.5.2. Serial Construct	31
39	2.5.3. Kernels Construct	32
40	2.5.4. Compute Construct Restrictions	33
41	2.5.5. Compute Construct Errors	34
42	2.5.6. if clause	34
43	2.5.7. self clause	34
44	2.5.8. async clause	34
45	2.5.9. wait clause	34
46	2.5.10. num_gangs clause	34
47	2.5.11. num_workers clause	35
48	2.5.12. vector_length clause	35
49	2.5.13. private clause	35
50	2.5.14. firstprivate clause	35
51	2.5.15. reduction clause	35
52	2.5.16. default clause	36
53	2.6. Data Environment	37
54	2.6.1. Variables with Predetermined Data Attributes	37

55	2.6.2. Variables with Implicitly Determined Data Attributes	37
56	2.6.3. Data Regions and Data Lifetimes	39
57	2.6.4. Data Structures with Pointers	39
58	2.6.5. Data Construct	40
59	2.6.6. Enter Data and Exit Data Directives	41
60	2.6.7. Reference Counters	43
61	2.6.8. Attachment Counter	44
62	2.7. Data Clauses	44
63	2.7.1. Data Specification in Data Clauses	45
64	2.7.2. Data Clause Actions	46
65	2.7.3. Data Clause Errors	49
66	2.7.4. deviceptr clause	49
67	2.7.5. present clause	50
68	2.7.6. copy clause	50
69	2.7.7. copyin clause	51
70	2.7.8. copyout clause	51
71	2.7.9. create clause	52
72	2.7.10. no_create clause	53
73	2.7.11. delete clause	53
74	2.7.12. attach clause	54
75	2.7.13. detach clause	54
76	2.8. Host_Data Construct	54
77	2.8.1. use_device clause	55
78	2.8.2. if clause	55
79	2.8.3. if_present clause	55
80	2.9. Loop Construct	55
81	2.9.1. collapse clause	57
82	2.9.2. gang clause	57
83	2.9.3. worker clause	59
84	2.9.4. vector clause	59
85	2.9.5. seq clause	59
86	2.9.6. independent clause	60
87	2.9.7. auto clause	60
88	2.9.8. tile clause	60
89	2.9.9. device_type clause	60
90	2.9.10. private clause	61
91	2.9.11. reduction clause	61
92	2.10. Cache Directive	65
93	2.11. Combined Constructs	65
94	2.12. Atomic Construct	67
95	2.13. Declare Directive	71
96	2.13.1. device_resident clause	72
97	2.13.2. create clause	72
98	2.13.3. link clause	73
99	2.14. Executable Directives	74
100	2.14.1. Init Directive	74
101	2.14.2. Shutdown Directive	75
102	2.14.3. Set Directive	76

103	2.14.4. Update Directive	78
104	2.14.5. Wait Directive	80
105	2.14.6. Enter Data Directive	80
106	2.14.7. Exit Data Directive	80
107	2.15. Procedure Calls in Compute Regions	80
108	2.15.1. Routine Directive	80
109	2.15.2. Global Data Access	83
110	2.16. Asynchronous Behavior	83
111	2.16.1. async clause	84
112	2.16.2. wait clause	85
113	2.16.3. Wait Directive	85
114	2.17. Fortran Specific Behavior	86
115	2.17.1. Optional Arguments	86
116	2.17.2. Do Concurrent Construct	87
117	3. Runtime Library	89
118	3.1. Runtime Library Definitions	89
119	3.2. Runtime Library Routines	90
120	3.2.1. acc_get_num_devices	90
121	3.2.2. acc_set_device_type	90
122	3.2.3. acc_get_device_type	91
123	3.2.4. acc_set_device_num	92
124	3.2.5. acc_get_device_num	92
125	3.2.6. acc_get_property	93
126	3.2.7. acc_init	94
127	3.2.8. acc_shutdown	94
128	3.2.9. acc_async_test	95
129	3.2.10. acc_wait	96
130	3.2.11. acc_wait_async	97
131	3.2.12. acc_wait_any	99
132	3.2.13. acc_get_default_async	99
133	3.2.14. acc_set_default_async	100
134	3.2.15. acc_on_device	100
135	3.2.16. acc_malloc	101
136	3.2.17. acc_free	101
137	3.2.18. acc_copyin and acc_create	102
138	3.2.19. acc_copyout and acc_delete	103
139	3.2.20. acc_update_device and acc_update_self	105
140	3.2.21. acc_map_data	107
141	3.2.22. acc_unmap_data	107
142	3.2.23. acc_deviceptr	108
143	3.2.24. acc_hostptr	108
144	3.2.25. acc_is_present	109
145	3.2.26. acc_memcpy_to_device	110
146	3.2.27. acc_memcpy_from_device	111
147	3.2.28. acc_memcpy_device	112
148	3.2.29. acc_attach and acc_detach	112
149	3.2.30. acc_memcpy_d2d	113

150	4. Environment Variables	117
151	4.1. ACC_DEVICE_TYPE	117
152	4.2. ACC_DEVICE_NUM	117
153	4.3. ACC_PROFLIB	117
154	5. Profiling and Error Callback Interface	119
155	5.1. Events	119
156	5.1.1. Runtime Initialization and Shutdown	120
157	5.1.2. Device Initialization and Shutdown	120
158	5.1.3. Enter Data and Exit Data	121
159	5.1.4. Data Allocation	121
160	5.1.5. Data Construct	122
161	5.1.6. Update Directive	122
162	5.1.7. Compute Construct	122
163	5.1.8. Enqueue Kernel Launch	123
164	5.1.9. Enqueue Data Update (Upload and Download)	123
165	5.1.10. Wait	123
166	5.1.11. Error Event	124
167	5.2. Callbacks Signature	124
168	5.2.1. First Argument: General Information	125
169	5.2.2. Second Argument: Event-Specific Information	126
170	5.2.3. Third Argument: API-Specific Information	131
171	5.3. Loading the Library	132
172	5.3.1. Library Registration	133
173	5.3.2. Statically-Linked Library Initialization	134
174	5.3.3. Runtime Dynamic Library Loading	134
175	5.3.4. Preloading with LD_PRELOAD	135
176	5.3.5. Application-Controlled Initialization	136
177	5.4. Registering Event Callbacks	136
178	5.4.1. Event Registration and Unregistration	136
179	5.4.2. Disabling and Enabling Callbacks	138
180	5.5. Advanced Topics	139
181	5.5.1. Dynamic Behavior	139
182	5.5.2. OpenACC Events During Event Processing	140
183	5.5.3. Multiple Host Threads	140
184	6. Glossary	143
185	A. Recommendations for Implementers	149
186	A.1. Target Devices	149
187	A.1.1. NVIDIA GPU Targets	149
188	A.1.2. AMD GPU Targets	149
189	A.1.3. Multicore Host CPU Target	150
190	A.2. API Routines for Target Platforms	150
191	A.2.1. NVIDIA CUDA Platform	150
192	A.2.2. OpenCL Target Platform	151
193	A.3. Recommended Options	152
194	A.3.1. C Pointer in Present clause	152

A.3.2. Automatic Data Attributes 152

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,

236 and deallocating memory. In most cases, the host can queue a sequence of operations to be executed
237 on a device, one after the other.

238 Most current accelerators and many multicore CPUs support two or three levels of parallelism.
239 Most accelerators and multicore CPUs support coarse-grain parallelism, which is fully parallel execution
240 across execution units. There may be limited support for synchronization across coarse-grain
241 parallel operations. Many accelerators and some CPUs also support fine-grain parallelism, often
242 implemented as multiple threads of execution within a single execution unit, which are typically
243 rapidly switched on the execution unit to tolerate long latency memory operations. Finally, most
244 accelerators and CPUs also support SIMD or vector operations within each execution unit. The
245 execution model exposes these multiple levels of parallelism on a device and the programmer is
246 required to understand the difference between, for example, a fully parallel loop and a loop that
247 is vectorizable but requires synchronization between statements. A fully parallel loop can be programmed
248 for coarse-grain parallel execution. Loops with dependences must either be split to allow
249 coarse-grain parallel execution, or be programmed to execute on a single execution unit using fine-
250 grain parallelism, vector parallelism, or sequentially.

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

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

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

274 The program starts executing with a single initial host thread, identified by a program counter and
275 its stack. The initial host thread may spawn additional host threads, using OpenACC or another
276 mechanism, such as with the OpenMP API. On a device, a single vector lane of a single worker of a
277 single gang is called a device thread. When executing on an accelerator, a parallel execution context
278 is created on the accelerator and may contain many such threads.

279 The user should not attempt to implement barrier synchronization, critical sections, or locks across
280 any of gang, worker, or vector parallelism. The execution model allows for an implementation that

281 executes some gangs to completion before starting to execute other gangs. This means that trying
282 to implement synchronization between gangs is likely to fail. In particular, a barrier across gangs
283 cannot be implemented in a portable fashion, since all gangs may not ever be active at the same time.
284 Similarly, the execution model allows for an implementation that executes some workers within a
285 gang or vector lanes within a worker to completion before starting other workers or vector lanes,
286 or for some workers or vector lanes to be suspended until other workers or vector lanes complete.
287 This means that trying to implement synchronization across workers or vector lanes is likely to fail.
288 In particular, implementing a barrier or critical section across workers or vector lanes using atomic
289 operations and a busy-wait loop may never succeed, since the scheduler may suspend the worker or
290 vector lane that owns the lock, and the worker or vector lane waiting on the lock can never complete.

291 Some devices, such as a multicore CPU, may also create and launch additional compute regions,
292 allowing for nested parallelism. In that case, the OpenACC directives may be executed by a host
293 thread or a device thread. This specification uses the term *local thread* or *local memory* to mean the
294 thread that executes the directive, or the memory associated with that thread, whether that thread
295 executes on the host or on the accelerator. The specification uses the term *local device* to mean the
296 device on which the *local thread* is executing.

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

305 1.3 Memory Model

306 The most significant difference between a host-only program and a host+accelerator program is that
307 the memory on an accelerator may be discrete from host memory. This is the case with most current
308 GPUs, for example. In this case, the host thread may not be able to read or write device memory
309 directly because it is not mapped into the host thread's virtual memory space. All data movement
310 between host memory and accelerator memory must be performed by the host thread through system
311 calls that explicitly move data between the separate memories, typically using direct memory access
312 (DMA) transfers. Similarly, the accelerator may not be able to read or write host memory; though
313 this is supported by some accelerators, it may incur significant performance penalty.

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

- 320 • Memory bandwidth between host memory and accelerator memory determines the level of
321 compute intensity required to effectively accelerate a given region of code.
- 322 • The user should be aware that a discrete accelerator memory is usually significantly smaller
323 than the host memory, prohibiting offloading regions of code that operate on very large
324 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.

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

405 To simplify the specification and convey appropriate constraint information, a *pqr-list* is a comma-
 406 separated list of *pqr* items. For example, an *int-expr-list* is a comma-separated list of one or more
 407 integer expressions, and a *var-list* is a comma-separated list of one or more *vars*. The one exception
 408 is *clause-list*, which is a list of one or more clauses optionally separated by commas.

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

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

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

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

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

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

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

422 1.7 Organization of this document

423 The rest of this document is organized as follows:

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

427 Chapter 3 Runtime Library, defines user-callable functions and library routines to query the accel-
 428 erator features and control behavior of accelerator-enabled programs at runtime.

429 Chapter 4 Environment Variables, defines user-settable environment variables used to control be-
 430 havior of accelerator-enabled programs at runtime.

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

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

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

436 1.8 References

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

- 439 ● *American National Standard Programming Language C*, ANSI X3.159-1989 (ANSI C).
- 440 ● ISO/IEC 9899:1999, *Information Technology – Programming Languages – C*, (C99).
- 441 ● ISO/IEC 9899:2011, *Information Technology – Programming Languages – C*, (C11).
- 442 The use of the following C11 features may result in unspecified behavior.
- 443 – Threads
- 444 – Thread-local storage
- 445 – Parallel memory model
- 446 – Atomic
- 447 ● ISO/IEC 9899:2018, *Information Technology – Programming Languages – C*, (C18).
- 448 The use of the following C18 features may result in unspecified behavior.
- 449 – Thread related features
- 450 ● ISO/IEC 14882:1998, *Information Technology – Programming Languages – C++*.
- 451 ● ISO/IEC 14882:2011, *Information Technology – Programming Languages – C++*, (C++11).
- 452 The use of the following C++11 features may result in unspecified behavior.
- 453 – Extern templates
- 454 – copy and rethrow exceptions
- 455 – memory model
- 456 – atomics
- 457 – move semantics
- 458 – std::thread
- 459 – thread-local storage
- 460 ● ISO/IEC 14882:2014, *Information Technology – Programming Languages – C++*, (C++14).
- 461 ● ISO/IEC 14882:2017, *Information Technology – Programming Languages – C++*, (C++17).
- 462 ● ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran – Part*
- 463 *I: Base Language*, (Fortran 2003).
- 464 ● ISO/IEC 1539-1:2010, *Information Technology – Programming Languages – Fortran – Part*
- 465 *I: Base Language*, (Fortran 2008).
- 466 The use of the following Fortran 2008 features may result in unspecified behavior.
- 467 – Coarrays
- 468 – Simply contiguous arrays rank remapping to rank>1 target
- 469 – Allocatable components of recursive type
- 470 – Polymorphic assignment

- 471 • ISO/IEC 1539-1:2018, *Information Technology – Programming Languages – Fortran – Part*
- 472 *1: Base Language*, (Fortran 2018).

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

- 474 – Interoperability with C

- 475 * C functions declared in ISO Fortran binding.h

- 476 * Assumed rank

- 477 – All additional parallel/coarray features

- 478 • *OpenMP Application Program Interface*, version 5.0, November 2018
- 479 • *NVIDIA CUDA[™] C Programming Guide*, version 11.1.1, October 2020
- 480 • *The OpenCL Specification*, version 2.2, Khronos OpenCL Working Group, July 2019
- 481 • *INCITS INCLUSIVE TERMINOLOGY GUIDELINES*, version 2021.06.07, InterNational Com-
- 482 mittee for Information Technology Standards, June 2021

483 1.9 Changes from Version 1.0 to 2.0

- 484 • `_OPENACC` value updated to `201306`
- 485 • `default (none)` clause on `parallel` and `kernels` directives
- 486 • the implicit data attribute for scalars in `parallel` constructs has changed
- 487 • the implicit data attribute for scalars in loops with `loop` directives with the independent
- 488 attribute has been clarified
- 489 • `acc_async_sync` and `acc_async_noval` values for the `async` clause
- 490 • Clarified the behavior of the `reduction` clause on a `gang` loop
- 491 • Clarified allowable loop nesting (`gang` may not appear inside `worker`, which may not ap-
- 492 pear within `vector`)
- 493 • `wait` clause on `parallel`, `kernels` and `update` directives
- 494 • `async` clause on the `wait` directive
- 495 • `enter data` and `exit data` directives
- 496 • Fortran *common block* names may now appear in many data clauses
- 497 • `link` clause for the `declare` directive
- 498 • the behavior of the `declare` directive for global data
- 499 • the behavior of a data clause with a C or C++ pointer variable has been clarified
- 500 • predefined data attributes
- 501 • support for multidimensional dynamic C/C++ arrays
- 502 • `tile` and `auto` loop clauses
- 503 • `update self` introduced as a preferred synonym for `update host`

- 504 • **routine** directive and support for separate compilation
- 505 • **device_type** clause and support for multiple device types
- 506 • nested parallelism using **parallel** or **kernels** region containing another **parallel** or **kernels** re-
- 507 **gion**
- 508 • **atomic** constructs
- 509 • new concepts: **gang-redundant**, **gang-partitioned**; **worker-single**, **worker-partitioned**; **vector-**
- 510 **single**, **vector-partitioned**; **thread**
- 511 • new API routines:
 - 512 – **acc_wait**, **acc_wait_all** instead of **acc_async_wait** and **acc_async_wait_all**
 - 513 – **acc_wait_async**
 - 514 – **acc_copyin**, **acc_present_or_copyin**
 - 515 – **acc_create**, **acc_present_or_create**
 - 516 – **acc_copyout**, **acc_delete**
 - 517 – **acc_map_data**, **acc_unmap_data**
 - 518 – **acc_deviceptr**, **acc_hostptr**
 - 519 – **acc_is_present**
 - 520 – **acc_memcpy_to_device**, **acc_memcpy_from_device**
 - 521 – **acc_update_device**, **acc_update_self**
- 522 • defined behavior with multiple host threads, such as with OpenMP
- 523 • recommendations for specific implementations
- 524 • clarified that no arguments are allowed on the **vector** clause in a **parallel** region

525 1.10 Corrections in the August 2013 document

- 526 • corrected the **atomic capture** syntax for C/C++
- 527 • fixed the name of the **acc_wait** and **acc_wait_all** procedures
- 528 • fixed description of the **acc_hostptr** procedure

529 1.11 Changes from Version 2.0 to 2.5

- 530 • The **_OPENACC** value was updated to **201510**; see Section 2.2 Conditional Compilation.
- 531 • The **num_gangs**, **num_workers**, and **vector_length** clauses are now allowed on the
- 532 **kernels** construct; see Section 2.5.3 Kernels Construct.
- 533 • Reduction on C++ class members, array elements, and struct elements are explicitly disal-
- 534 lowed; see Section 2.5.15 reduction clause.
- 535 • Reference counting is now used to manage the correspondence and lifetime of device data;
- 536 see Section 2.6.7 Reference Counters.

- 537 • The behavior of the **exit data** directive has changed to decrement the dynamic reference
538 counter. A new optional **finalize** clause was added to set the dynamic reference counter
539 to zero. See Section 2.6.6 Enter Data and Exit Data Directives.
- 540 • The **copy**, **copyin**, **copyout**, and **create** data clauses were changed to behave like
541 **present_or_copy**, etc. The **present_or_copy**, **pcopy**, **present_or_copyin**,
542 **pcopyin**, **present_or_copyout**, **pcopyout**, **present_or_create**, and **pcreate**
543 data clauses are no longer needed, though will be accepted for compatibility; see Section 2.7
544 Data Clauses.
- 545 • Reductions on orphaned gang loops are explicitly disallowed; see Section 2.9 Loop Construct.
- 546 • The description of the **loop auto** clause has changed; see Section 2.9.7 auto clause.
- 547 • Text was added to the **private** clause on a **loop** construct to clarify that a copy is made
548 for each gang or worker or vector lane, not each thread; see Section 2.9.10 private clause.
- 549 • The description of the **reduction** clause on a **loop** construct was corrected; see Sec-
550 tion 2.9.11 reduction clause.
- 551 • A restriction was added to the **cache** clause that all references to that variable must lie within
552 the region being cached; see Section 2.10 Cache Directive.
- 553 • Text was added to the **private** and **reduction** clauses on a combined construct to clarify
554 that they act like **private** and **reduction** on the **loop**, not **private** and **reduction**
555 on the **parallel** or **reduction** on the **kernels**; see Section 2.11 Combined Constructs.
- 556 • The **declare create** directive with a Fortran **allocatable** has new behavior; see Sec-
557 tion 2.13.2 create clause.
- 558 • New **init**, **shutdown**, **set** directives were added; see Section 2.14.1 Init Directive, 2.14.2
559 Shutdown Directive, and 2.14.3 Set Directive.
- 560 • A new **if_present** clause was added to the **update** directive, which changes the behavior
561 when data is not present from a runtime error to a no-op; see Section 2.14.4 Update Directive.
- 562 • The **routine bind** clause definition changed; see Section 2.15.1 Routine Directive.
- 563 • An **acc routine** without **gang/worker/vector/seq** is now defined as an error; see
564 Section 2.15.1 Routine Directive.
- 565 • A new **default (present)** clause was added for compute constructs; see Section 2.5.16
566 default clause.
- 567 • The Fortran header file **openacc_lib.h** is no longer supported; the Fortran module **openacc**
568 should be used instead; see Section 3.1 Runtime Library Definitions.
- 569 • New API routines were added to get and set the default async queue value; see Section 3.2.13
570 **acc_get_default_async** and 3.2.14 **acc_set_default_async**.
- 571 • The **acc_copyin**, **acc_create**, **acc_copyout**, and **acc_delete** API routines were
572 changed to behave like **acc_present_or_copyin**, etc. The **acc_present_or_** names
573 are no longer needed, though will be supported for compatibility. See Sections 3.2.18 and fol-
574 lowing.

- 575 • Asynchronous versions of the data API routines were added; see Sections 3.2.18 and follow-
576 ing.
- 577 • A new API routine added, **acc_memcpy_device**, to copy from one device address to
578 another device address; see Section 3.2.26 `acc_memcpy_to_device`.
- 579 • A new OpenACC interface for profile and trace tools was added;
580 see Chapter 5 Profiling and Error Callback Interface.

581 1.12 Changes from Version 2.5 to 2.6

- 582 • The **_OPENACC** value was updated to **201711**.
- 583 • A new **serial** compute construct was added. See Section 2.5.2 Serial Construct.
- 584 • A new runtime API query routine was added. **acc_get_property** may be called from
585 the host and returns properties about any device. See Section 3.2.6.
- 586 • The text has clarified that if a variable is in a reduction which spans two or more nested loops,
587 each **loop** directive on any of those loops must have a **reduction** clause that contains the
588 variable; see Section 2.9.11 reduction clause.
- 589 • An optional **if** or **if_present** clause is now allowed on the **host_data** construct. See
590 Section 2.8 Host_Data Construct.
- 591 • A new **no_create** data clause is now allowed on compute and **data** constructs. See Sec-
592 tion 2.7.10 `no_create` clause.
- 593 • The behavior of Fortran optional arguments in data clauses and in routine calls has been
594 specified; see Section 2.17.1 Optional Arguments.
- 595 • The descriptions of some of the Fortran versions of the runtime library routines were simpli-
596 fied; see Section 3.2 Runtime Library Routines.
- 597 • To allow for manual deep copy of data structures with pointers, new *attach* and *detach* be-
598 havior was added to the data clauses, new **attach** and **detach** clauses were added, and
599 matching **acc_attach** and **acc_detach** runtime API routines were added; see Sections
600 2.6.4, 2.7.12-2.7.13 and 3.2.29.
- 601 • The Intel Coprocessor Offload Interface target and API routine sections were removed from
602 the Section A Recommendations for Implementers, since Intel no longer produces this prod-
603 uct.

604 1.13 Changes from Version 2.6 to 2.7

- 605 • The **_OPENACC** value was updated to **201811**.
- 606 • The specification allows for hosts that share some memory with the device but not all memory.
607 The wording in the text now discusses whether local thread data is in shared memory (memory
608 shared between the local thread and the device) or discrete memory (local thread memory that
609 is not shared with the device), instead of shared-memory devices and non-shared memory
610 devices. See Sections 1.3 Memory Model and 2.6 Data Environment.
- 611 • The text was clarified to allow an implementation that treats a multicore CPU as a device,
612 either an additional device or the only device.

- 613 • The **readonly** modifier was added to the **copyin** data clause and **cache** directive. See
614 Sections 2.7.7 and 2.10.
- 615 • The term *local device* was defined; see Section 1.2 Execution Model and the Glossary.
- 616 • The term *var* is used more consistently throughout the specification to mean a variable name,
617 array name, subarray specification, array element, composite variable member, or Fortran
618 common block name between slashes. Some uses of *var* allow only a subset of these options,
619 and those limitations are given in those cases.
- 620 • The **self** clause was added to the compute constructs; see Section 2.5.7 self clause.
- 621 • The appearance of a **reduction** clause on a compute construct implies a **copy** clause for
622 each reduction variable; see Sections 2.5.15 reduction clause and 2.11 Combined Constructs.
- 623 • The **default (none)** and **default (present)** clauses were added to the **data** con-
624 struct; see Section 2.6.5 Data Construct.
- 625 • Data is defined to be *present* based on the values of the structured and dynamic reference
626 counters; see Section 2.6.7 Reference Counters and the Glossary.
- 627 • The interaction of the **acc_map_data** and **acc_unmap_data** runtime API calls on the
628 present counters is defined; see Section 2.7.2, 3.2.21, and 3.2.22.
- 629 • A restriction clarifying that a **host_data** construct must have at least one **use_device**
630 clause was added.
- 631 • Arrays, subarrays and composite variables are now allowed in **reduction** clauses; see
632 Sections 2.9.11 reduction clause and 2.5.15 reduction clause.
- 633 • Changed behavior of ICVs to support nested compute regions and host as a device semantics.
634 See Section 2.3.

635 1.14 Changes from Version 2.7 to 3.0

- 636 • Updated **_OPENACC** value to **201911**.
- 637 • Updated the normative references to the most recent standards for all base languages. See
638 Section 1.8.
- 639 • Changed the text to clarify uses and limitations of the **device_type** clause and added
640 examples; see Section 2.4.
- 641 • Clarified the conflict between the implicit **copy** clause for variables in a **reduction** clause
642 and the implicit **firstprivate** for scalar variables not in a data clause but used in a
643 **parallel** or **serial** construct; see Sections 2.5.1 and 2.5.2.
- 644 • Required at least one data clause on a **data** construct, an **enter data** directive, or an **exit**
645 **data** directive; see Sections 2.6.5 and 2.6.6.
- 646 • Added text describing how a C++ *lambda* invoked in a compute region and the variables
647 captured by the *lambda* are handled; see Section 2.6.2.
- 648 • Added a **zero** modifier to **create** and **copyout** data clauses that zeros the device memory
649 after it is allocated; see Sections 2.7.8 and 2.7.9.

- 650 • Added a new restriction on the **loop** directive allowing only one of the **seq**, **independent**,
651 and **auto** clauses to appear; see Section 2.9.
- 652 • Added a new restriction on the **loop** directive disallowing a **gang**, **worker**, or **vector**
653 clause to appear if a **seq** clause appears; see Section 2.9.
- 654 • Allowed variables to be modified in an atomic region in a loop where the iterations must
655 otherwise be data independent, such as loops with a **loop independent** clause or a **loop**
656 directive in a **parallel** construct; see Sections 2.9.2, 2.9.3, 2.9.4, and 2.9.6.
- 657 • Clarified the behavior of the **auto** and **independent** clauses on the **loop** directive; see
658 Sections 2.9.7 and 2.9.6.
- 659 • Clarified that an orphaned **loop** construct, or a **loop** construct in a **parallel** construct
660 with no **auto** or **seq** clauses is treated as if an **independent** clause appears; see Sec-
661 tion 2.9.6.
- 662 • For a variable in a **reduction** clause, clarified when the update to the original variable is
663 complete, and added examples; see Section 2.9.11.
- 664 • Clarified that a variable in an orphaned **reduction** clause must be private; see Section 2.9.11.
- 665 • Required at least one clause on a **declare** directive; see Section 2.13.
- 666 • Added an **if** clause to **init**, **shutdown**, **set**, and **wait** directives; see Sections 2.14.1,
667 2.14.2, 2.14.3, and 2.16.3.
- 668 • Required at least one clause on a **set** directive; see Section 2.14.3.
- 669 • Added a *devnum* modifier to the **wait** directive and clause to specify a device to which the
670 wait operation applies; see Section 2.16.3.
- 671 • Allowed a **routine** directive to include a C++ *lambda* name or to appear before a C++
672 *lambda* definition, and defined implicit **routine** directive behavior when a C++ *lambda* is
673 called in a compute region or an *accelerator routine*; see Section 2.15.
- 674 • Added runtime API routine **acc_memcpy_d2d** for copying data directly between two de-
675 vice arrays on the same or different devices; see Section 3.2.30.
- 676 • Defined the values for the **acc_construct_t** and **acc_device_api** enumerations for
677 cross-implementation compatibility; see Sections 5.2.2 and 5.2.3.
- 678 • Changed the return type of **acc_set_cuda_stream** from **int** (values were not specified)
679 to **void**; see Section A.2.1.
- 680 • Edited and expanded Section 1.17 Topics Deferred For a Future Revision.

681 1.15 Changes from Version 3.0 to 3.1

- 682 • Updated **_OPENACC** value to **202011**.
- 683 • Clarified that Fortran blank common blocks are not permitted and that same-named common
684 blocks must have the same size. See Section 1.6.
- 685 • Clarified that a **parallel** construct's block is considered to start in gang-redundant mode
686 even if there's just a single gang. See Section 2.5.1.

- 687 • Added support for the Fortran BLOCK construct. See Sections 2.5.1, 2.5.3, 2.6.1, 2.6.5, 2.8,
688 2.13, and 6.
- 689 • Defined the **serial** construct in terms of the **parallel** construct to improve readability.
690 Instead of defining it in terms of clauses **num_gangs (1) num_workers (1)**
691 **vector_length (1)**, defined the **serial** construct as executing with a single gang of a
692 single worker with a vector length of one. See Section 2.5.2.
- 693 • Consolidated compute construct restrictions into a new section to improve readability. See
694 Section 2.5.4.
- 695 • Clarified that a **default** clause may appear at most once on a compute construct. See
696 Section 2.5.16.
- 697 • Consolidated discussions of implicit data attributes on compute and combined constructs into
698 a separate section. Clarified the conditions under which each data attribute is implied. See
699 Section 2.6.2.
- 700 • Added a restriction that certain loop reduction variables must have explicit data clauses on
701 their parent compute constructs. This change addresses portability across existing OpenACC
702 implementations. See Sections 2.6.2 and A.3.2.
- 703 • Restored the OpenACC 2.5 behavior of the **present**, **copy**, **copyin**, **copyout**, **create**,
704 **no_create**, **delete** data clauses at exit from a region, or on an **exit data** directive, as
705 applicable, and **create** clause at exit from an implicit data region where a **declare** di-
706 rective appears, and **acc_copyout**, **acc_delete** routines, such that no action is taken if
707 the appropriate reference counter is zero, instead of a runtime error being issued if data is not
708 present. See Sections 2.7.5, 2.7.6, 2.7.7, 2.7.8, 2.7.9, 2.7.10, 2.7.11, 2.13.2, and 3.2.19.
- 709 • Clarified restrictions on loop forms that can be associated with **loop** constructs, including
710 the case of C++ range-based **for** loops. See Section 2.9.
- 711 • Specified where **gang** clauses are implied on **loop** constructs. This change standardizes
712 behavior of existing OpenACC implementations. See Section 2.9.2.
- 713 • Corrected C/C++ syntax for **atomic capture** with a structured block. See Section 2.12.
- 714 • Added the behavior of the Fortran *do concurrent* construct. See Section 2.17.2.
- 715 • Changed the Fortran run-time procedures: **acc_device_property** has been renamed to
716 **acc_device_property_kind** and **acc_get_property** uses a different integer kind
717 for the result. See Section 3.2.
- 718 • Added or changed argument names for the Runtime Library routines to be descriptive and
719 consistent. This mostly impacts Fortran programs, which can pass arguments by name. See
720 Section 3.2.
- 721 • Replaced composite variable by aggregate variable in **reduction**, **default**, and **private**
722 clauses and in implicitly determined data attributes; the new wording also includes Fortran
723 character and allocatable/pointer variables. See glossary in Section 6.

724 1.16 Changes from Version 3.1 to 3.2

- 725 • Updated **_OPENACC** value to 202111.

- 726 • Modified specification to comply with INCITS standard for inclusive terminology.
- 727 • The text was changed to state that certain runtime errors, when detected, result in a call to the
728 current runtime error callback routines. See Section 1.5.
- 729 • An ambiguity issue with the C/C++ comma operator was resolved. See Section 1.6.
- 730 • The terms *true* and *false* were defined and used throughout to shorten the descriptions. See
731 Section 1.6.
- 732 • Implicitly determined data attributes on compute constructs were clarified. See Section 2.6.2.
- 733 • Clarified that the **default (none)** clause applies to scalar variables. See Section 2.6.2.
- 734 • The **async**, **wait**, and **device_type** clauses may be specified on **data** constructs. See
735 Section 2.6.5.
- 736 • The behavior of data clauses and data API routines with a null pointer in the clause or as a
737 routine argument is defined. See Sections 2.7.5-2.7.11, 2.8.1, and 3.2.16-3.2.30.
- 738 • Precision issues with the loop trip count calculation were clarified. See Section 2.9.
- 739 • Text in Section 2.16 was moved and reorganized to improve clarity and reduce redundancy.
- 740 • Some runtime routine descriptions were expanded and clarified. See Section 3.2.
- 741 • The **acc_init_device** and **acc_shutdown_device** routines were added to initialize
742 and shut down individual devices. See Section 3.2.7 and Section 3.2.8.
- 743 • Some runtime routine sections were reorganized and combined into a single section to sim-
744 plify maintenance and reduce redundant text:
 - 745 – The sections for four **acc_async_test** routines were combined into a single section.
746 See Section 3.2.9.
 - 747 – The sections for four **acc_wait** routines were combined into a single section. See
748 Section 3.2.10.
 - 749 – The sections for four **acc_wait_async** routines were combined into a single section.
750 See Section 3.2.11.
 - 751 – The two sections for **acc_copyin** and **acc_create** were combined into a single
752 section. See Section 3.2.18.
 - 753 – The two sections for **acc_copyout** and **acc_delete** were combined into a single
754 section. See Section 3.2.19.
 - 755 – The two sections for **acc_update_self** and **acc_update_device** were com-
756 bined into a single section. See Section 3.2.20.
 - 757 – The two sections for **acc_attach** and **acc_detach** were combined into a single
758 section. See Section 3.2.29.
- 759 • Added runtime API routine **acc_wait_any**. See section 3.2.12.
- 760 • The descriptions of the **async** and **async_queue** fields of **acc_callback_info** were
761 clarified. See Section 5.2.1.

1.17 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.
- Requiring the implementation to imply an **acc routine** directive for procedures called within a compute construct or accelerator routine.
- 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.

- 797 • Allowing non-contiguous Fortran array sections as arguments to some Runtime API routines,
798 such as **acc_update_device**.
- 799 • Bindings to other languages.

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

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

Each directive starts with **#pragma acc**. The remainder of the directive follows the C and C++ conventions for pragmas. Whitespace may be used before and after the **#**; whitespace may be required to separate words in a directive. Preprocessing tokens following the **#pragma 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

838 specified. Clauses may be repeated unless otherwise specified. Some clauses have an argument that
839 can contain a list.

840 2.2 Conditional Compilation

841 The `_OPENACC` macro name is defined to have a value `yyyymm` where `yyyy` is the year and `mm` is
842 the month designation of the version of the OpenACC directives supported by the implementation.
843 This macro must be defined by a compiler only when OpenACC directives are enabled. The version
844 described here is 202111.

845 2.3 Internal Control Variables

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

850 The ICVs are:

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

855 2.3.1 Modifying and Retrieving ICV Values

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

ICV	Ways to modify values	Way to retrieve value
<i>acc-current-device-type-var</i>	<code>acc_set_device_type</code> <code>set device_type</code> <code>init device_type</code> <code>ACC_DEVICE_TYPE</code>	<code>acc_get_device_type</code>
858 <i>acc-current-device-num-var</i>	<code>acc_set_device_num</code> <code>set device_num</code> <code>init device_num</code> <code>ACC_DEVICE_NUM</code>	<code>acc_get_device_num</code>
<i>acc-default-async-var</i>	<code>acc_set_default_async</code> <code>set default_async</code>	<code>acc_get_default_async</code>

859 The initial values are implementation-defined. After initial values are assigned, but before any
860 OpenACC construct or API routine is executed, the values of any environment variables that were
861 set by the user are read and the associated ICVs are modified accordingly. There is one copy of
862 each ICV for each host thread that is not generated by a compute construct. For threads that are
863 generated by a compute construct the initial value for each ICV is inherited from the local thread.
864 The behavior for each ICV is as if there is a copy for each thread. If an ICV is modified, then a
865 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 non-conforming, then the original directive is non-conforming.

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
```

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

```
909 // non-conforming: gang and worker are not permitted together
910 #pragma acc routine gang device_type(foo) worker
```

```
911 // conforming: gang and worker apply to different device types
912 #pragma acc routine device_type(foo) worker \
913 device_type(*) gang
```

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

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

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

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

928 2.5 Compute Constructs

929 2.5.1 Parallel Construct

930 Summary

931 This fundamental construct starts parallel execution on the current device.

932 Syntax

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

```
934 #pragma acc parallel [clause-list] new-line
935 structured block
```

937 and in Fortran, the syntax is

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

941 OR

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

```

944     [!$acc end parallel]
945 where clause is one of the following:
946     async [ ( int-expr ) ]
947     wait [ ( int-expr-list ) ]
948     num_gangs ( int-expr )
949     num_workers ( int-expr )
950     vector_length ( int-expr )
951     device_type ( device-type-list )
952     if ( condition )
953     self [ ( condition ) ]
954     reduction ( operator : var-list )
955     copy ( var-list )
956     copyin ( [ readonly: ] var-list )
957     copyout ( [ zero: ] var-list )
958     create ( [ zero: ] var-list )
959     no_create ( var-list )
960     present ( var-list )
961     deviceptr ( var-list )
962     attach ( var-list )
963     private ( var-list )
964     firstprivate ( var-list )
965     default ( none | present )

```

966 Description

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

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

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

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

984 2.5.2 Serial Construct

985 **Summary**

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

991 **Syntax**

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

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

995

996 and in Fortran, the syntax is

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

1000 OR

```
1001     !$acc serial [ clause-list ]  
1002         block construct  
1003     [!$acc end serial]
```

1004 where *clause* is as for the **parallel** construct except that the **num_gangs**, **num_workers**, and
 1005 **vector_length** clauses are not permitted.

1006 **2.5.3 Kernels Construct**1007 **Summary**

1008 This construct defines a region of the program that is to be compiled into a sequence of kernels for
 1009 execution on the current device.

1010 **Syntax**

1011 In C and C++, the syntax of the OpenACC **kernels** construct is

```
1012     #pragma acc kernels [ clause-list ] new-line  
1013         structured block
```

1014

1015 and in Fortran, the syntax is

```
1016     !$acc kernels [ clause-list ]  
1017         structured block  
1018     !$acc end kernels
```

1019 OR

```
1020     !$acc kernels [ clause-list ]  
1021         block construct  
1022     [!$acc end kernels]
```


1023 where *clause* is one of the following:

```

1024   async [ ( int-expr ) ]
1025   wait [ ( int-expr-list ) ]
1026   num_gangs ( int-expr )
1027   num_workers ( int-expr )
1028   vector_length ( int-expr )
1029   device_type ( device-type-list )
1030   if ( condition )
1031   self [ ( condition ) ]
1032   copy ( var-list )
1033   copyin ( [ readonly: ] var-list )
1034   copyout ( [ zero: ] var-list )
1035   create ( [ zero: ] var-list )
1036   no_create ( var-list )
1037   present ( var-list )
1038   deviceptr ( var-list )
1039   attach ( var-list )
1040   default ( none | present )

```

1041 Description

1042 The compiler will split the code in the kernels region into a sequence of accelerator kernels. Typi-
 1043 cally, each loop nest will be a distinct kernel. When the program encounters a **kernels** construct,
 1044 it will launch the sequence of kernels in order on the device. The number and configuration of gangs
 1045 of workers and vector length may be different for each kernel.

1046 If the **async** clause does not appear, there is an implicit barrier at the end of the kernels region,
 1047 and the local thread execution will not proceed until the entire sequence of kernels has completed
 1048 execution.

1049 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
 1050 data clauses are described in Section 2.7 Data Clauses. The **device_type** clause is described
 1051 in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described in Sec-
 1052 tion 2.6.2. Restrictions are described in Section 2.5.4.

1053 2.5.4 Compute Construct Restrictions

1054 The following restrictions apply to all compute constructs:

- 1055 • A program may not branch into or out of a compute construct.
- 1056 • A program must not depend on the order of evaluation of the clauses or on any side effects of
 1057 the evaluations.
- 1058 • Only the **async**, **wait**, **num_gangs**, **num_workers**, and **vector_length** clauses
 1059 may follow a **device_type** clause.
- 1060 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
 1061 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1062 • At most one **default** clause may appear, and it must have a value of either **none** or
 1063 **present**.

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. The value of the integer expression defines the number of parallel gangs that will execute the parallel region, or that will execute each kernel created for the kernels region. If the clause does not appear, an

1098 implementation-defined default will be used; the default may depend on the code within the con-
1099 struct. The implementation may use a lower value than specified based on limitations imposed by
1100 the target architecture.

1101 **2.5.11 num_workers clause**

1102 The **num_workers** clause is allowed on the **parallel** and **kernels** constructs. The value
1103 of the integer expression defines the number of workers within each gang that will be active after
1104 a gang transitions from worker-single mode to worker-partitioned mode. If the clause does not
1105 appear, an implementation-defined default will be used; the default value may be 1, and may be
1106 different for each **parallel** construct or for each kernel created for a **kernels** construct. The
1107 implementation may use a different value than specified based on limitations imposed by the target
1108 architecture.

1109 **2.5.12 vector_length clause**

1110 The **vector_length** clause is allowed on the **parallel** and **kernels** constructs. The value
1111 of the integer expression defines the number of vector lanes that will be active after a worker transi-
1112 tions from vector-single mode to vector-partitioned mode. This clause determines the vector length
1113 to use for vector or SIMD operations. If the clause does not appear, an implementation-defined
1114 default will be used. This vector length will be used for loop constructs annotated with the **vector**
1115 clause, as well as loops automatically vectorized by the compiler. The implementation may use a
1116 different value than specified based on limitations imposed by the target architecture.

1117 **2.5.13 private clause**

1118 The **private** clause is allowed on the **parallel** and **serial** constructs; it declares that a copy
1119 of each item on the list will be created for each gang.

1120 **Restrictions**

- 1121 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private**
1122 clauses.

1123 **2.5.14 firstprivate clause**

1124 The **firstprivate** clause is allowed on the **parallel** and **serial** constructs; it declares that
1125 a copy of each item on the list will be created for each gang, and that the copy will be initialized with
1126 the value of that item on the local thread when a **parallel** or **serial** construct is encountered.

1127 **Restrictions**

- 1128 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
1129 **firstprivate** clauses.

1130 **2.5.15 reduction clause**

1131 The **reduction** clause is allowed on the **parallel** and **serial** constructs. It specifies a
1132 reduction operator and one or more *vars*. It implies **copy** clauses as described in Section 2.6.2. For
1133 each reduction *var*, a private copy is created for each parallel gang and initialized for that operator.
1134 At the end of the region, the values for each gang are combined using the reduction operator, and
1135 the result combined with the value of the original *var* and stored in the original *var*. If the reduction

1136 *var* is an array or subarray, the array reduction operation is logically equivalent to applying that
 1137 reduction operation to each element of the array or subarray individually. If the reduction *var*
 1138 is a composite variable, the reduction operation is logically equivalent to applying that reduction
 1139 operation to each member of the composite variable individually. The reduction result is available
 1140 after the region.

1141 The following table lists the operators that are valid and the initialization values; in each case, the
 1142 initialization value will be cast into the data type of the *var*. For **max** and **min** reductions, the
 1143 initialization values are the least representable value and the largest representable value for that data
 1144 type, respectively. At a minimum, the supported data types include Fortran **logical** as well as
 1145 the numerical data types in C (e.g., **_Bool**, **char**, **int**, **float**, **double**, **float _Complex**,
 1146 **double _Complex**), C++ (e.g., **bool**, **char**, **wchar_t**, **int**, **float**, **double**), and Fortran
 1147 (e.g., **integer**, **real**, **double precision**, **complex**). However, for each reduction operator,
 1148 the supported data types include only the types permitted as operands to the corresponding operator
 1149 in the base language where (1) for max and min, the corresponding operator is less-than and (2) for
 1150 other operators, the operands and the result are the same type.

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

1152 Restrictions

- 1153 • A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name,
 1154 an array element, or a subarray (refer to Section 2.7.1).
- 1155 • If the reduction *var* is an array element or a subarray, accessing the elements of the array
 1156 outside the specified index range results in unspecified behavior.
- 1157 • The reduction *var* may not be a member of a composite variable.
- 1158 • If the reduction *var* is a composite variable, each member of the composite variable must be
 1159 a supported datatype for the reduction operation.
- 1160 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
 1161 **reduction** clauses.

1162 2.5.16 default clause

1163 The **default** clause is optional. At most one **default** clause may appear. It adjusts what
 1164 data attributes are implicitly determined for variables used in the compute construct as described in

1165 Section 2.6.2.

1166 2.6 Data Environment

1167 This section describes the data attributes for variables. The data attributes for a variable may be
1168 *predetermined*, *implicitly determined*, or *explicitly determined*. Variables with predetermined data
1169 attributes may not appear in a data clause that conflicts with that data attribute. Variables with
1170 implicitly determined data attributes may appear in a data clause that overrides the implicit attribute.
1171 Variables with explicitly determined data attributes are those which appear in a data clause on a
1172 **data** construct, a compute construct, or a **declare** directive.

1173 OpenACC supports systems with accelerators that have discrete memory from the host, systems
1174 with accelerators that share memory with the host, as well as systems where an accelerator shares
1175 some memory with the host but also has some discrete memory that is not shared with the host.
1176 In the first case, no data is in shared memory. In the second case, all data is in shared memory.
1177 In the third case, some data may be in shared memory and some data may be in discrete memory,
1178 although a single array or aggregate data structure must be allocated completely in shared or discrete
1179 memory. When a nested OpenACC construct is executed on the device, the default target device for
1180 that construct is the same device on which the encountering accelerator thread is executing. In that
1181 case, the target device shares memory with the encountering thread.

1182 2.6.1 Variables with Predetermined Data Attributes

1183 The loop variable in a C **for** statement or Fortran **do** statement that is associated with a loop
1184 directive is predetermined to be private to each thread that will execute each iteration of the loop.
1185 Loop variables in Fortran **do** statements within a compute construct are predetermined to be private
1186 to the thread that executes the loop.

1187 Variables declared in a C block or Fortran block construct that is executed in *vector-partitioned*
1188 mode are private to the thread associated with each vector lane. Variables declared in a C block
1189 or Fortran block construct that is executed in *worker-partitioned vector-single* mode are private to
1190 the worker and shared across the threads associated with the vector lanes of that worker. Variables
1191 declared in a C block or Fortran block construct that is executed in *worker-single* mode are private
1192 to the gang and shared across the threads associated with the workers and vector lanes of that gang.

1193 A procedure called from a compute construct will be annotated as **seq**, **vector**, **worker**, or
1194 **gang**, as described Section 2.15 Procedure Calls in Compute Regions. Variables declared in **seq**
1195 routine are private to the thread that made the call. Variables declared in **vector** routine are private
1196 to the worker that made the call and shared across the threads associated with the vector lanes of
1197 that worker. Variables declared in **worker** or **gang** routine are private to the gang that made the
1198 call and shared across the threads associated with the workers and vector lanes of that gang.

1199 2.6.2 Variables with Implicitly Determined Data Attributes

1200 When implicitly determining data attributes on a compute construct, the following clauses are visi-
1201 ble and variable accesses are exposed to the compute construct:

- 1202 • *Visible default clause*: The nearest **default** clause appearing on the compute construct
1203 or a lexically containing **data** construct.
- 1204 • *Visible data clause*: Any data clause on the compute construct, a lexically containing **data**

1205 construct, or a visible **declare** directive.

1206 • *Exposed variable access*: Any access to the data or address of a variable at a point within the
1207 compute construct where the variable is not private to a scope lexically enclosed within the
1208 compute construct.

1209 **Note**: In the argument of C's **sizeof** operator, the appearance of a variable is not an exposed
1210 access because neither its data nor its address is accessed. In the argument of a **reduction**
1211 clause on an enclosed **loop** construct, the appearance of a variable that is not otherwise
1212 privatized is an exposed access to the original variable.

1213 On a compute or combined construct, if a variable appears in a **reduction** clause but no other
1214 data clause, it is treated as if it also appears in a **copy** clause. Otherwise, for any variable, the
1215 compiler will implicitly determine its data attribute on a compute construct if all of the following
1216 conditions are met:

- 1217 • There is no **default (none)** clause visible at the compute construct.
- 1218 • An access to the variable is exposed to the compute construct.
- 1219 • The variable does not appear in a data clause visible at the compute construct.

1220 An aggregate variable will be treated as if it appears either:

- 1221 • In a **present** clause if there is a **default (present)** clause visible at the compute con-
1222 struct.
- 1223 • In a **copy** clause otherwise.

1224 A scalar variable will be treated as if it appears either:

- 1225 • In a **copy** clause if the compute construct is a **kernels** construct.
- 1226 • In a **firstprivate** clause otherwise.

1227 **Note**: Any **default (none)** clause visible at the compute construct applies to both aggregate
1228 and scalar variables. However, any **default (present)** clause visible at the compute construct
1229 applies only to aggregate variables.

1230 Restrictions

- 1231 • If there is a **default (none)** clause visible at a compute construct, for any variable access
1232 exposed to the compute construct, the compiler requires the variable to appear either in an
1233 explicit data clause visible at the compute construct or in a **firstprivate**, **private**, or
1234 **reduction** clause on the compute construct.
- 1235 • If a scalar variable appears in a **reduction** clause on a **loop** construct that has a parent
1236 **parallel** or **serial** construct, and if the reduction's access to the original variable is
1237 exposed to the parent compute construct, the variable must appear either in an explicit data
1238 clause visible at the compute construct or in a **firstprivate**, **private**, or **reduction**
1239 clause on the compute construct. **Note**: Implementations are encouraged to issue a compile-
1240 time diagnostic when this restriction is violated to assist users in writing portable OpenACC
1241 applications.

1242 If a C++ *lambda* is called in a compute region and does not appear in a data clause, then it is
1243 treated as if it appears in a **copyin** clause on the current construct. A variable captured by a

1244 *lambda* is processed according to its data types: a pointer type variable is treated as if it appears
1245 in a **no_create** clause; a reference type variable is treated as if it appears in a **present** clause;
1246 for a struct or a class type variable, any pointer member is treated as if it appears in a **no_create**
1247 clause on the current construct. If the variable is defined as global or file or function static, it must
1248 appear in a **declare** directive.

1249 2.6.3 Data Regions and Data Lifetimes

1250 Data in shared memory is accessible from the current device as well as to the local thread. Such
1251 data is available to the accelerator for the lifetime of the variable. Data not in shared memory must
1252 be copied to and from device memory using data constructs, clauses, and API routines. A *data*
1253 *lifetime* is the duration from when the data is first made available to the accelerator until it becomes
1254 unavailable. For data in shared memory, the data lifetime begins when the data is allocated and
1255 ends when it is deallocated; for statically allocated data, the data lifetime begins when the program
1256 begins and does not end. For data not in shared memory, the data lifetime begins when it is made
1257 present and ends when it is no longer present.

1258 There are four types of data regions. When the program encounters a **data** construct, it creates a
1259 data region.

1260 When the program encounters a compute construct with explicit data clauses or with implicit data
1261 allocation added by the compiler, it creates a data region that has a duration of the compute construct.

1262 When the program enters a procedure, it creates an implicit data region that has a duration of the
1263 procedure. That is, the implicit data region is created when the procedure is called, and exited when
1264 the program returns from that procedure invocation. There is also an implicit data region associated
1265 with the execution of the program itself. The implicit program data region has a duration of the
1266 execution of the program.

1267 In addition to data regions, a program may create and delete data on the accelerator using **enter**
1268 **data** and **exit data** directives or using runtime API routines. When the program executes
1269 an **enter data** directive, or executes a call to a runtime API **acc_copyin** or **acc_create**
1270 routine, each *var* on the directive or the variable on the runtime API argument list will be made live
1271 on accelerator.

1272 2.6.4 Data Structures with Pointers

1273 This section describes the behavior of data structures that contain pointers. A pointer may be a
1274 C or C++ pointer (e.g., **float***), a Fortran pointer or array pointer (e.g., **real, pointer,**
1275 **dimension(:)**), or a Fortran allocatable (e.g., **real, allocatable, dimension(:)**).

1276 When a data object is copied to device memory, the values are copied exactly. If the data is a data
1277 structure that includes a pointer, or is just a pointer, the pointer value copied to device memory
1278 will be the host pointer value. If the pointer target object is also allocated in or copied to device
1279 memory, the pointer itself needs to be updated with the device address of the target object before
1280 dereferencing the pointer in device memory.

1281 An *attach* action updates the pointer in device memory to point to the device copy of the data
1282 that the host pointer targets; see Section 2.7.2. For Fortran array pointers and allocatable arrays,
1283 this includes copying any associated descriptor (dope vector) to the device copy of the pointer.
1284 When the device pointer target is deallocated, the pointer in device memory should be restored

1285 to the host value, so it can be safely copied back to host memory. A *detach* action updates the
 1286 pointer in device memory to have the same value as the corresponding pointer in local memory;
 1287 see Section 2.7.2. The *attach* and *detach* actions are performed by the **copy**, **copyin**, **copyout**,
 1288 **create**, **attach**, and **detach** data clauses (Sections 2.7.4-2.7.13), and the **acc_attach** and
 1289 **acc_detach** runtime API routines (Section 3.2.29). The *attach* and *detach* actions use attachment
 1290 counters to determine when the pointer in device memory needs to be updated; see Section 2.6.8.

1291 2.6.5 Data Construct

1292 Summary

1293 The **data** construct defines *vars* to be allocated in the current device memory for the duration of
 1294 the region, whether data should be copied from local memory to the current device memory upon
 1295 region entry, and copied from device memory to local memory upon region exit.

1296 Syntax

1297 In C and C++, the syntax of the OpenACC **data** construct is

```
1298     #pragma acc data [clause-list] new-line
1299         structured block
```

1300 and in Fortran, the syntax is

```
1301     !$acc data [clause-list]
1302         structured block
1303     !$acc end data
```

1304 OR

```
1305     !$acc data [clause-list]
1306         block construct
1307     [!$acc end data]
```

1308 where *clause* is one of the following:

```
1309     if ( condition )
1310     async [ ( int-expr ) ]
1311     wait [ ( wait-argument ) ]
1312     device_type ( device-type-list )
1313     copy ( var-list )
1314     copyin ( [readonly:]var-list )
1315     copyout ( [zero:]var-list )
1316     create ( [zero:]var-list )
1317     no_create ( var-list )
1318     present ( var-list )
1319     deviceptr ( var-list )
1320     attach ( var-list )
1321     default ( none | present )
```

1322 Description

1323 Data will be allocated in the memory of the current device and copied from local memory to device
 1324 memory, or copied back, as required. The data clauses are described in Section 2.7 Data Clauses.

1325 Structured reference counters are incremented for data when entering a data region, and decre-
1326 mented when leaving the region, as described in Section 2.6.7 Reference Counters. The **device_type**
1327 clause is described in Section 2.4 Device-Specific Clauses.

1328 Restrictions

- 1329 • At least one **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**,
1330 **attach**, or **default** clause must appear on a **data** construct.
- 1331 • Only the **async** and **wait** clauses may follow a **device_type** clause.

1332 if clause

1333 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate
1334 space in the current device memory and move data from and to the local memory as required. When
1335 an **if** clause appears, the program will conditionally allocate memory in and move data to and/or
1336 from device memory. When the *condition* in the **if** clause evaluates to *false*, no device memory
1337 will be allocated, and no data will be moved. When the *condition* evaluates to *true*, the data will be
1338 allocated and moved as specified. At most one **if** clause may appear.

1339 async clause

1340 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1341 **Note:** The **async** clause only affects operations directly associated with this particular **data** con-
1342 struct, such as data transfers. Execution of the associated structured block or block construct remains
1343 synchronous to the local thread. Nested OpenACC constructs, directives, and calls to runtime li-
1344 brary routines do not inherit the **async** clause from this construct, and the programmer must take
1345 care to not accidentally introduce race conditions related to asynchronous data transfers.

1346 wait clause

1347 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1348 default clause

1349 The **default** clause is optional. At most one **default** clause may appear. It adjusts what data
1350 attributes are implicitly determined for variables used in lexically contained compute constructs as
1351 described in Section 2.6.2.

1352 Errors

- 1353 • See Section 2.7.3 for errors due to data clauses.
- 1354 • See Sections 2.16.1 and 2.16.2 for errors due to **async** or **wait** clauses.

1355 2.6.6 Enter Data and Exit Data Directives

1356 Summary

1357 An **enter data** directive may be used to define *vars* to be allocated in the current device memory
1358 for the remaining duration of the program, or until an **exit data** directive that deallocates the data.
1359 They also tell whether data should be copied from local memory to device memory at the **enter**
1360 **data** directive, and copied from device memory to local memory at the **exit data** directive. The
1361 dynamic range of the program between the **enter data** directive and the matching **exit data**
1362 directive is the data lifetime for that data.

1363 **Syntax**

1364 In C and C++, the syntax of the OpenACC **enter data** directive is

```
1365     #pragma acc enter data clause-list new-line
```

1366 and in Fortran, the syntax is

```
1367     !$acc enter data clause-list
```

1368 where *clause* is one of the following:

```
1369     if ( condition )
1370     async [ ( int-expr ) ]
1371     wait [ ( wait-argument ) ]
1372     copyin ( var-list )
1373     create ( [ zero: ] var-list )
1374     attach ( var-list )
```

1375 In C and C++, the syntax of the OpenACC **exit data** directive is

```
1376     #pragma acc exit data clause-list new-line
```

1377 and in Fortran, the syntax is

```
1378     !$acc exit data clause-list
```

1379 where *clause* is one of the following:

```
1380     if ( condition )
1381     async [ ( int-expr ) ]
1382     wait [ ( wait-argument ) ]
1383     copyout ( var-list )
1384     delete ( var-list )
1385     detach ( var-list )
1386     finalize
```

1387 **Description**

1388 At an **enter data** directive, data may be allocated in the current device memory and copied from
 1389 local memory to device memory. This action enters a data lifetime for those *vars*, and will make
 1390 the data available for **present** clauses on constructs within the data lifetime. Dynamic reference
 1391 counters are incremented for this data, as described in Section 2.6.7 Reference Counters. Pointers
 1392 in device memory may be *attached* to point to the corresponding device copy of the host pointer
 1393 target.

1394 At an **exit data** directive, data may be copied from device memory to local memory and deal-
 1395 located from device memory. If no **finalize** clause appears, dynamic reference counters are
 1396 decremented for this data. If a **finalize** clause appears, the dynamic reference counters are set
 1397 to zero for this data. Pointers in device memory may be *detached* so as to have the same value as
 1398 the original host pointer.

1399 The data clauses are described in Section 2.7 Data Clauses. Reference counting behavior is de-
 1400 scribed in Section 2.6.7 Reference Counters.

1401 Restrictions

- 1402 • At least one **copyin**, **create**, or **attach** clause must appear on an **enter data** direc-
1403 tive.
- 1404 • At least one **copyout**, **delete**, or **detach** clause must appear on an **exit data** direc-
1405 tive.

1406 if clause

1407 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or
1408 deallocate space in the current device memory and move data from and to local memory. When an
1409 **if** clause appears, the program will conditionally allocate or deallocate device memory and move
1410 data to and/or from device memory. When the *condition* in the **if** clause evaluates to *false*, no
1411 device memory will be allocated or deallocated, and no data will be moved. When the *condition*
1412 evaluates to *true*, the data will be allocated or deallocated and moved as specified.

1413 async clause

1414 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1415 wait clause

1416 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1417 finalize clause

1418 The **finalize** clause is allowed on the **exit data** directive and is optional. When no **finalize**
1419 clause appears, the **exit data** directive will decrement the dynamic reference counters for *vars*
1420 appearing in **copyout** and **delete** clauses, and will decrement the attachment counters for point-
1421 ers appearing in **detach** clauses. If a **finalize** clause appears, the **exit data** directive will
1422 set the dynamic reference counters to zero for *vars* appearing in **copyout** and **delete** clauses,
1423 and will set the attachment counters to zero for pointers appearing in **detach** clauses.

1424 Errors

- 1425 • See Section 2.7.3 for errors due to data clauses.
- 1426 • See Sections 2.16.1 and 2.16.2 for errors due to **async** or **wait** clauses.

1427 2.6.7 Reference Counters

1428 When device memory is allocated for data not in shared memory due to data clauses or OpenACC
1429 API routine calls, the OpenACC implementation keeps track of that section of device memory and
1430 its relationship to the corresponding data in host memory.

1431 Each section of device memory is associated with two *reference counters* per device, a structured
1432 reference counter and a dynamic reference counter. The structured and dynamic reference counters
1433 are used to determine when to allocate or deallocate data in device memory. The structured reference
1434 counter for a section of memory keeps track of how many nested data regions have been entered for
1435 that data. The initial value of the structured reference counter for static data in device memory (in a
1436 global **declare** directive) is one; for all other data, the initial value is zero. The dynamic reference
1437 counter for a section of memory keeps track of how many dynamic data lifetimes are currently active

1438 in device memory for that section. The initial value of the dynamic reference counter is zero. Data
1439 is considered *present* if the sum of the structured and dynamic reference counters is greater than
1440 zero.

1441 A structured reference counter is incremented when entering each data or compute region that con-
1442 tain an explicit data clause or implicitly-determined data attributes for that section of memory, and
1443 is decremented when exiting that region. A dynamic reference counter is incremented for each
1444 **enter data copyin** or **create** clause, or each **acc_copyin** or **acc_create** API routine
1445 call for that section of memory. The dynamic reference counter is decremented for each **exit**
1446 **data copyout** or **delete** clause when no **finalize** clause appears, or each **acc_copyout**
1447 or **acc_delete** API routine call for that section of memory. The dynamic reference counter will
1448 be set to zero with an **exit data copyout** or **delete** clause when a **finalize** clause ap-
1449 pears, or each **acc_copyout_finalize** or **acc_delete_finalize** API routine call for
1450 the section of memory. The reference counters are modified synchronously with the local thread,
1451 even if the data directives include an **async** clause. When both structured and dynamic reference
1452 counters reach zero, the data lifetime in device memory for that data ends.

1453 2.6.8 Attachment Counter

1454 Since multiple pointers can target the same address, each pointer in device memory is associated
1455 with an *attachment counter* per device. The *attachment counter* for a pointer is initialized to zero
1456 when the pointer is allocated in device memory. The *attachment counter* for a pointer is set to one
1457 whenever the pointer is *attached* to new target address, and incremented whenever an *attach* action
1458 for that pointer is performed for the same target address. The *attachment counter* is decremented
1459 whenever a *detach* action occurs for the pointer, and the pointer is *detached* when the *attachment*
1460 *counter* reaches zero. This is described in more detail in Section 2.7.2 Data Clause Actions.

1461 A pointer in device memory can be assigned a device address in two ways. The pointer can be
1462 attached to a device address due to data clauses or API routines, as described in Section 2.7.2
1463 Data Clause Actions, or the pointer can be assigned in a compute region executed on that device.
1464 Unspecified behavior may result if both ways are used for the same pointer.

1465 Pointer members of structs, classes, or derived types in device or host memory can be overwritten
1466 due to update directives or API routines. It is the user's responsibility to ensure that the pointers
1467 have the appropriate values before or after the data movement in either direction. The behavior of
1468 the program is undefined if any of the pointer members are attached when an update of a composite
1469 variable is performed.

1470 2.7 Data Clauses

1471 Data clauses may appear on the **parallel** construct, **serial** construct, **kernels** construct,
1472 **data** construct, the **enter data** and **exit data** directives, and **declare** directives. In the
1473 descriptions, the *region* is a compute region with a clause appearing on a **parallel**, **serial**, or
1474 **kernels** construct, a data region with a clause on a **data** construct, or an implicit data region
1475 with a clause on a **declare** directive. If the **declare** directive appears in a global context,
1476 the corresponding implicit data region has a duration of the program. The list argument to each
1477 data clause is a comma-separated collection of *vars*. On a **declare** directive, the list argument
1478 of a **copyin**, **create**, **device_resident**, or **link** clause may include a Fortran *common*
1479 *block* name enclosed within slashes. On any directive, for any clause except **deviceptr** and
1480 **present**, the list argument may include a Fortran *common block* name enclosed within slashes

1481 if that *common block* name also appears in a **declare** directive **link** clause. In all cases, the
 1482 compiler will allocate and manage a copy of the *var* in the memory of the current device, creating a
 1483 visible device copy of that *var*, for data not in shared memory.

1484 OpenACC supports accelerators with discrete memories from the local thread. However, if the
 1485 accelerator can access the local memory directly, the implementation may avoid the memory allo-
 1486 cation and data movement and simply share the data in local memory. Therefore, a program that
 1487 uses and assigns data on the host and uses and assigns the same data on the accelerator within a
 1488 data region without update directives to manage the coherence of the two copies may get different
 1489 answers on different accelerators or implementations.

1490 Restrictions

- 1491 • Data clauses may not follow a **device_type** clause.
- 1492 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in data
 1493 clauses.

1494 2.7.1 Data Specification in Data Clauses

1495 In C and C++, a subarray is an array name followed by an extended array range specification in
 1496 brackets, with start and length, such as

1497 **AA[2:n]**

1498 If the lower bound is missing, zero is used. If the length is missing and the array has known size, the
 1499 size of the array is used; otherwise the length is required. The subarray **AA[2:n]** means elements
 1500 **AA[2], AA[3], ..., AA[2+n-1]**.

1501 In C and C++, a two dimensional array may be declared in at least four ways:

- 1502 • Statically-sized array: **float AA[100][200];**
- 1503 • Pointer to statically sized rows: **typedef float row[200]; row* BB;**
- 1504 • Statically-sized array of pointers: **float* CC[200];**
- 1505 • Pointer to pointers: **float** DD;**

1506 Each dimension may be statically sized, or a pointer to dynamically allocated memory. Each of
 1507 these may be included in a data clause using subarray notation to specify a rectangular array:

- 1508 • **AA[2:n][0:200]**
- 1509 • **BB[2:n][0:m]**
- 1510 • **CC[2:n][0:m]**
- 1511 • **DD[2:n][0:m]**

1512 Multidimensional rectangular subarrays in C and C++ may be specified for any array with any com-
 1513 bination of statically-sized or dynamically-allocated dimensions. For statically sized dimensions, all
 1514 dimensions except the first must specify the whole extent to preserve the contiguous data restriction,
 1515 discussed below. For dynamically allocated dimensions, the implementation will allocate pointers
 1516 in device memory corresponding to the pointers in local memory and will fill in those pointers as
 1517 appropriate.

1518 In Fortran, a subarray is an array name followed by a comma-separated list of range specifications
1519 in parentheses, with lower and upper bound subscripts, such as

1520 **arr(1:high, low:100)**

1521 If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if
1522 known, are used. All dimensions except the last must specify the whole extent, to preserve the
1523 contiguous data restriction, discussed below.

1524 Restrictions

- 1525 • In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be
1526 specified.
- 1527 • In C and C++, the length for dynamically allocated dimensions of an array must be explicitly
1528 specified.
- 1529 • In C and C++, modifying pointers in pointer arrays during the data lifetime, either on the host
1530 or on the device, may result in undefined behavior.
- 1531 • If a subarray appears in a data clause, the implementation may choose to allocate memory for
1532 only that subarray on the accelerator.
- 1533 • In Fortran, array pointers may appear, but pointer association is not preserved in device mem-
1534 ory.
- 1535 • Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous
1536 section of memory, except for dynamic multidimensional C arrays.
- 1537 • In C and C++, if a variable or array of composite type appears, all the data members of the
1538 struct or class are allocated and copied, as appropriate. If a composite member is a pointer
1539 type, the data addressed by that pointer are not implicitly copied.
- 1540 • In Fortran, if a variable or array of composite type appears, all the members of that derived
1541 type are allocated and copied, as appropriate. If any member has the **allocatable** or
1542 **pointer** attribute, the data accessed through that member are not copied.
- 1543 • If an expression is used in a subscript or subarray expression in a clause on a **data** construct,
1544 the same value is used when copying data at the end of the data region, even if the values of
1545 variables in the expression change during the data region.

1546 2.7.2 Data Clause Actions

1547 Most of the data clauses perform one or more the following actions. The actions test or modify one
1548 or both of the structured and dynamic reference counters, depending on the directive on which the
1549 data clause appears.

1550 Present Increment Action

1551 A *present increment* action is one of the actions that may be performed for a **present** (Sec-
1552 tion 2.7.5), **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create**
1553 (Section 2.7.9), or **no_create** (Section 2.7.10) clause, or for a call to an **acc_copyin** or
1554 **acc_create** (Section 3.2.18) API routine. See those sections for details.

1555 A *present increment* action for a *var* occurs only when *var* is already present in device memory.

1556 A *present increment* action for a *var* increments the structured or dynamic reference counter for *var*.

1557 Present Decrement Action

1558 A *present decrement* action is one of the actions that may be performed for a **present** (Section
1559 2.7.5), **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Sec-
1560 tion 2.7.9), **no_create** (Section 2.7.10), or **delete** (Section 2.7.11) clause, or for a call to an
1561 **acc_copyout** or **acc_delete** (Section 3.2.19) API routine. See those sections for details.

1562 A *present decrement* action for a *var* occurs only when *var* is already present in device memory.

1563 A *present decrement* action for a *var* decrements the structured or dynamic reference counter for
1564 *var*, if its value is greater than zero. If the device memory associated with *var* was mapped to
1565 the device using **acc_map_data**, the dynamic reference count may not be decremented to zero,
1566 except by a call to **acc_unmap_data**. If the reference counter is already zero, its value is left
1567 unchanged.

1568 Create Action

1569 A *create* action is one of the actions that may be performed for a **copyout** (Section 2.7.8) or
1570 **create** (Section 2.7.9) clause, or for a call to an **acc_create** API routine (Section 3.2.18). See
1571 those sections for details.

1572 A *create* action for a *var* occurs only when *var* is not already present in device memory.

1573 A *create* action for a *var*:

- 1574 • allocates device memory for *var*; and
- 1575 • sets the structured or dynamic reference counter to one.

1576 Copyin Action

1577 A *copyin* action is one of the actions that may be performed for a **copy** (Section 2.7.6) or **copyin**
1578 (Section 2.7.7) clause, or for a call to an **acc_copyin** API routine (Section 3.2.18). See those
1579 sections for details.

1580 A *copyin* action for a *var* occurs only when *var* is not already present in device memory.

1581 A *copyin* action for a *var*:

- 1582 • allocates device memory for *var*;
- 1583 • initiates a copy of the data for *var* from the local thread memory to the corresponding device
1584 memory; and
- 1585 • sets the structured or dynamic reference counter to one.

1586 The data copy may complete asynchronously, depending on other clauses on the directive.

1587 Copyout Action

1588 A *copyout* action is one of the actions that may be performed for a **copy** (Section 2.7.6) or
1589 **copyout** (Section 2.7.8) clause, or for a call to an **acc_copyout** API routine (Section 3.2.19).
1590 See those sections for details.

1591 A *copyout* action for a *var* occurs only when *var* is present in device memory.

1592 A *copyout* action for a *var*:

- 1593 • performs an *immediate detach* action for any pointer in *var*;
- 1594 • initiates a copy of the data for *var* from device memory to the corresponding local thread
1595 memory; and
- 1596 • deallocates device memory for *var*.

1597 The data copy may complete asynchronously, depending on other clauses on the directive, in which
1598 case the memory is deallocated when the data copy is complete.

1599 Delete Action

1600 A *delete* action is one of the actions that may be performed for a **present** (Section 2.7.5),
1601 **copyin** (Section 2.7.7), **create** (Section 2.7.9), **no_create** (Section 2.7.10), or **delete** (Sec-
1602 tion 2.7.11) clause, or for a call to an **acc_delete** API routine (Section 3.2.19). See those sections
1603 for details.

1604 A *delete* action for a *var* occurs only when *var* is present in device memory.

1605 A *delete* action for *var*:

- 1606 • performs an *immediate detach* action for any pointer in *var*; and
- 1607 • deallocates device memory for *var*.

1608 Attach Action

1609 An *attach* action is one of the actions that may be performed for a **present** (Section 2.7.5),
1610 **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Section 2.7.9),
1611 **no_create** (Section 2.7.10), or **attach** (Section 2.7.11) clause, or for a call to an **acc_attach**
1612 API routine (Section 3.2.29). See those sections for details.

1613 An *attach* action for a *var* occurs only when *var* is a pointer reference.

1614 If the pointer *var* is in shared memory or is not present in the current device memory, or if the
1615 address to which *var* points is not present in the current device memory, no action is taken. If the
1616 *attachment counter* for *var* is nonzero and the pointer in device memory already points to the device
1617 copy of the data in *var*, the *attachment counter* for the pointer *var* is incremented. Otherwise, the
1618 pointer in device memory is *attached* to the device copy of the data by initiating an update for the
1619 pointer in device memory to point to the device copy of the data and setting the *attachment counter*
1620 for the pointer *var* to one. If the pointer is a null pointer, the pointer in device memory is updated to
1621 have the same value. The update may complete asynchronously, depending on other clauses on the
1622 directive. The implementation schedules pointer updates after any data copies due to *copyin* actions
1623 that are performed for the same directive.

1624 Detach Action

1625 A *detach* action is one of the actions that may be performed for a **present** (Section 2.7.5),
1626 **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Section 2.7.9),
1627 **no_create** (Section 2.7.10), **delete** (Section 2.7.11), or **detach** (Section 2.7.11) clause, or
1628 for a call to an **acc_detach** API routine (Section 3.2.29). See those sections for details.

1629 A *detach* action for a *var* occurs only when *var* is a pointer reference.

1630 If the pointer *var* is in shared memory or is not present in the current device memory, or if the
 1631 *attachment counter* for *var* for the pointer is zero, no action is taken. Otherwise, the *attachment*
 1632 *counter* for the pointer *var* is decremented. If the *attachment counter* is decreased to zero, the
 1633 pointer is *detached* by initiating an update for the pointer *var* in device memory to have the same
 1634 value as the corresponding pointer in local memory. The update may complete asynchronously,
 1635 depending on other clauses on the directive. The implementation schedules pointer updates before
 1636 any data copies due to *copyout* actions that are performed for the same directive.

1637 Immediate Detach Action

1638 An *immediate detach* action is one of the actions that may be performed for a **detach** (Section
 1639 2.7.11) clause, or for a call to an **acc_detach_finalize** API routine (Section 3.2.29). See
 1640 those sections for details.

1641 An *immediate detach* action for a *var* occurs only when *var* is a pointer reference and is present in
 1642 device memory.

1643 If the *attachment counter* for the pointer is zero, the *immediate detach* action has no effect. Other-
 1644 wise, the *attachment counter* for the pointer set to zero and the pointer is *detached* by initiating an
 1645 update for the pointer in device memory to have the same value as the corresponding pointer in local
 1646 memory. The update may complete asynchronously, depending on other clauses on the directive.
 1647 The implementation schedules pointer updates before any data copies due to *copyout* actions that
 1648 are performed for the same directive.

1649 2.7.3 Data Clause Errors

1650 An error is issued for a *var* that appears in a **copy**, **copyin**, **copyout**, **create**, and **delete**
 1651 clause as follows:

- 1652 • An **acc_error_partly_present** error is issued if part of *var* is present in the current
 1653 device memory but all of *var* is not.
- 1654 • An **acc_error_invalid_data_section** error is issued if *var* is a Fortran subarray
 1655 with a stride that is not one.
- 1656 • An **acc_error_out_of_memory** error is issued if the accelerator device does not have
 1657 enough memory for *var*.

1658 An error is issued for a *var* that appears in a **present** clause as follows:

- 1659 • An **acc_error_not_present** error is issued if *var* is not present in the current device
 1660 memory at entry to a data or compute construct.
- 1661 • An **acc_error_partly_present** error is issued if part of *var* is present in the current
 1662 device memory but all of *var* is not.

1663 See Section 5.2.2.

1664 2.7.4 deviceptr clause

1665 The **deviceptr** clause may appear on structured **data** and compute constructs and **declare**
 1666 directives.

1667 The **deviceptr** clause is used to declare that the pointers in *var-list* are device pointers, so the
 1668 data need not be allocated or moved between the host and device for this pointer.

1669 In C and C++, the *vars* in *var-list* must be pointer variables.

1670 In Fortran, the *vars* in *var-list* must be dummy arguments (arrays or scalars), and may not have the
 1671 Fortran **pointer**, **allocatable**, or **value** attributes.

1672 For data in shared memory, host pointers are the same as device pointers, so this clause has no
 1673 effect.

1674 2.7.5 present clause

1675 The **present** clause may appear on structured **data** and compute constructs and **declare** di-
 1676 rectives. The **present** clause specifies that *vars* in *var-list* are in shared memory or are already
 1677 present in the current device memory due to data regions or data lifetimes that contain the construct
 1678 on which the **present** clause appears.

1679 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1680 the **present** clause behaves as follows:

- 1681 • At entry to the region:
 - 1682 – An *attach* action is performed if *var* is a pointer reference, and a *present increment*
 - 1683 action with the structured reference counter is performed if *var* is not a null pointer.
- 1684 • At exit from the region:
 - 1685 – If the structured reference counter for *var* is zero, no action is taken.
 - 1686 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
 - 1687 action with the structured reference counter is performed if *var* is not a null pointer. If
 - 1688 both structured and dynamic reference counters are zero, a *delete* action is performed.

1689 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1690 2.7.6 copy clause

1691 The **copy** clause may appear on structured **data** and compute constructs and on **declare** direc-
 1692 tives.

1693 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1694 the **copy** clause behaves as follows:

- 1695 • At entry to the region:
 - 1696 – If *var* is present and is not a null pointer, a *present increment* action with the structured
 - 1697 reference counter is performed.
 - 1698 – If *var* is not present, a *copyin* action with the structured reference counter is performed.
 - 1699 – If *var* is a pointer reference, an *attach* action is performed.
- 1700 • At exit from the region:
 - 1701 – If the structured reference counter for *var* is zero, no action is taken.

1702 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
 1703 action with the structured reference counter is performed if *var* is not a null pointer. If
 1704 both structured and dynamic reference counters are zero, a *copyout* action is performed.

1705 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1706 For compatibility with OpenACC 2.0, **present_or_copy** and **pcopy** are alternate names for
 1707 **copy**.

1708 2.7.7 copyin clause

1709 The **copyin** clause may appear on structured **data** and compute constructs, on **declare** direc-
 1710 tives, and on **enter data** directives.

1711 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1712 the **copyin** clause behaves as follows:

- 1713 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
 1714 the dynamic reference counter is used.
 - 1715 – If *var* is present and is not a null pointer, a *present increment* action with the appropriate
 1716 reference counter is performed.
 - 1717 – If *var* is not present, a *copyin* action with the appropriate reference counter is performed.
 - 1718 – If *var* is a pointer reference, an *attach* action is performed.
- 1719 • At exit from the region:
 - 1720 – If the structured reference counter for *var* is zero, no action is taken.
 - 1721 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
 1722 action with the structured reference counter is performed if *var* is not a null pointer. If
 1723 both structured and dynamic reference counters are zero, a *delete* action is performed.

1724 If the optional **readonly** modifier appears, then the implementation may assume that the data
 1725 referenced by *var-list* is never written to within the applicable region.

1726 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1727 For compatibility with OpenACC 2.0, **present_or_copyin** and **pcopyin** are alternate names
 1728 for **copyin**.

1729 An **enter data** directive with a **copyin** clause is functionally equivalent to a call to the **acc_copyin**
 1730 API routine, as described in Section 3.2.18.

1731 2.7.8 copyout clause

1732 The **copyout** clause may appear on structured **data** and compute constructs, on **declare** di-
 1733 rectives, and on **exit data** directives. The clause may optionally have a **zero** modifier if the
 1734 **copyout** clause appears on a structured **data** or compute construct.

1735 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1736 the **copyout** clause behaves as follows:

- 1737 • At entry to a region:

- 1738 – If *var* is present and is not a null pointer, a *present increment* action with the structured
1739 reference counter is performed.
- 1740 – If *var* is not present, a *create* action with the structured reference counter is performed.
1741 If a **zero** modifier appears, the memory is zeroed after the *create* action.
- 1742 – If *var* is a pointer reference, an *attach* action is performed.
- 1743 • At exit from a region, the structured reference counter is used. On an **exit data** directive,
1744 the dynamic reference counter is used.
- 1745 – If the appropriate reference counter for *var* is zero, no action is taken.
- 1746 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and the reference
1747 counter is updated if **var** is not a null pointer:
 - 1748 * On an **exit data** directive with a **finalize** clause, the dynamic reference
1749 counter is set to zero.
 - 1750 * Otherwise, a *present decrement* action with the appropriate reference counter is
1751 performed.
- 1752 If both structured and dynamic reference counters are zero, a *copyout* action is per-
1753 formed.

1754 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1755 For compatibility with OpenACC 2.0, **present_or_copyout** and **pcopyout** are alternate
1756 names for **copyout**.

1757 An **exit data** directive with a **copyout** clause and with or without a **finalize** clause is func-
1758 tionally equivalent to a call to the **acc_copyout_finalize** or **acc_copyout** API routine,
1759 respectively, as described in Section 3.2.19.

1760 2.7.9 create clause

1761 The **create** clause may appear on structured **data** and compute constructs, on **declare** direc-
1762 tives, and on **enter data** directives. The clause may optionally have a **zero** modifier.

1763 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1764 the **create** clause behaves as follows:

- 1765 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
1766 the dynamic reference counter is used.
 - 1767 – If *var* is present and is not a null pointer, a *present increment* action with the appropriate
1768 reference counter is performed.
 - 1769 – If *var* is not present and is not a null pointer, a *create* action with the appropriate refer-
1770 ence counter is performed. If a **zero** modifier appears, the memory is zeroed after the
1771 *create* action.
 - 1772 – If *var* is a pointer reference, an *attach* action is performed.
- 1773 • At exit from the region:
 - 1774 – If the structured reference counter for *var* is zero, no action is taken.

1775 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
 1776 action with the structured reference counter is performed if *var* is not a null pointer. If
 1777 both structured and dynamic reference counters are zero, a *delete* action is performed.

1778 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1779 For compatibility with OpenACC 2.0, **present_or_create** and **pcreate** are alternate names
 1780 for **create**.

1781 An **enter data** directive with a **create** clause is functionally equivalent to a call to the **acc_create**
 1782 API routine, as described in Section 3.2.18, except the directive may perform an *attach* action for a
 1783 pointer reference.

1784 2.7.10 no_create clause

1785 The **no_create** clause may appear on structured **data** and compute constructs.

1786 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1787 the **no_create** clause behaves as follows:

- 1788 • At entry to the region:
 - 1789 – If *var* is present and is not a null pointer, a *present increment* action with the structured
 1790 reference counter is performed. If *var* is present and is a pointer reference, an *attach*
 1791 action is performed.
 - 1792 – If *var* is not present, no action is performed, and any device code in this construct will
 1793 use the local memory address for *var*.
- 1794 • At exit from the region:
 - 1795 – If the structured reference counter for *var* is zero, no action is taken.
 - 1796 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
 1797 action with the structured reference counter is performed if *var* is not a null pointer. If
 1798 both structured and dynamic reference counters are zero, a *delete* action is performed.

1799 2.7.11 delete clause

1800 The **delete** clause may appear on **exit data** directives.

1801 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1802 the **delete** clause behaves as follows:

- 1803 • If the dynamic reference counter for *var* is zero, no action is taken.
 - 1804 • Otherwise, a *detach* action is performed if *var* is a pointer reference, and the dynamic refer-
 1805 ence counter is updated if *var* is not a null pointer:
 - 1806 – On an **exit data** directive with a **finalize** clause, the dynamic reference counter
 1807 is set to zero.
 - 1808 – Otherwise, a *present decrement* action with the dynamic reference counter is performed.
- 1809 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic
 1810 reference counters are zero, a *delete* action is performed.

1811 An **exit data** directive with a **delete** clause and with or without a **finalize** clause is func-
 1812 tionally equivalent to a call to the **acc_delete_finalize** or **acc_delete** API routine, re-
 1813 spectively, as described in Section 3.2.19.

1814 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1815 2.7.12 attach clause

1816 The **attach** clause may appear on structured **data** and compute constructs and on **enter data**
 1817 directives. Each *var* argument to an **attach** clause must be a C or C++ pointer or a Fortran variable
 1818 or array with the **pointer** or **allocatable** attribute.

1819 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1820 the **attach** clause behaves as follows:

- 1821 • At entry to a region or at an **enter data** directive, an *attach* action is performed.
- 1822 • At exit from the region, a *detach* action is performed.

1823 2.7.13 detach clause

1824 The **detach** clause may appear on **exit data** directives. Each *var* argument to a **detach** clause
 1825 must be a C or C++ pointer or a Fortran variable or array with the **pointer** or **allocatable**
 1826 attribute.

1827 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1828 the **detach** clause behaves as follows:

- 1829 • If there is a **finalize** clause on the **exit data** directive, an *immediate detach* action is
 1830 performed.
- 1831 • Otherwise, a *detach* action is performed.

1832 2.8 Host_Data Construct

1833 Summary

1834 The **host_data** construct makes the address of data in device memory available on the host.

1835 Syntax

1836 In C and C++, the syntax of the OpenACC **host_data** construct is

```
1837 #pragma acc host_data clause-list new-line
1838     structured block
```

1839 and in Fortran, the syntax is

```
1840 !$acc host_data clause-list
1841     structured block
1842 !$acc end host_data
```

1843 OR

```
1844 !$acc host_data clause-list
1845     block construct
1846 [!$acc end host_data]
```

1847 where *clause* is one of the following:

```
1848     use_device ( var-list )
1849     if ( condition )
1850     if_present
```

1851 Description

1852 This construct is used to make the address of data in device memory available in host code.

1853 Restrictions

- 1854 • A *var* in a **use_device** clause must be the name of a variable or array.
- 1855 • At least one **use_device** clause must appear.
- 1856 • 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.
- 1858 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **use_device** clauses.

1860 2.8.1 use_device clause

1861 The **use_device** clause tells the compiler to use the current device address of any *var* in *var-list*
1862 in code within the construct. In particular, this may be used to pass the device address of *var* to
1863 optimized procedures written in a lower-level API. If *var* is a null pointer, the same value is used
1864 for the device address. Otherwise, when there is no **if_present** clause, and either there is no
1865 **if** clause or the condition in the **if** clause evaluates to *true*, the *var* in *var-list* must be present in
1866 the accelerator memory due to data regions or data lifetimes that contain this construct. For data in
1867 shared memory, the device address is the same as the host address.

1868 2.8.2 if clause

1869 The **if** clause is optional. When an **if** clause appears and the condition evaluates to *false*, the
1870 compiler will not replace the addresses of any *var* in code within the construct. When there is no **if**
1871 clause, or when an **if** clause appears and the condition evaluates to *true*, the compiler will replace
1872 the addresses as described in the previous subsection.

1873 2.8.3 if_present clause

1874 When an **if_present** clause appears on the directive, the compiler will only replace the address
1875 of any *var* which appears in *var-list* that is present in the current device memory.

1876 2.9 Loop Construct

1877 Summary

1878 The OpenACC **loop** construct applies to a loop which must immediately follow this directive. The
1879 **loop** construct can describe what type of parallelism to use to execute the loop and declare private
1880 *vars* and reduction operations.

1881 **Syntax**1882 In C and C++, the syntax of the **loop** construct is1883 **#pragma acc loop** [*clause-list*] *new-line*
1884 *for loop*1885 In Fortran, the syntax of the **loop** construct is1886 **!\$acc loop** [*clause-list*]
1887 *do loop*1888 where *clause* is one of the following:1889 **collapse** (*n*)
1890 **gang** [(*gang-arg-list*)]
1891 **worker** [([**num** :] *int-expr*)]
1892 **vector** [([**length** :] *int-expr*)]
1893 **seq**
1894 **independent**
1895 **auto**
1896 **tile** (*size-expr-list*)
1897 **device_type** (*device-type-list*)
1898 **private** (*var-list*)
1899 **reduction** (*operator* : *var-list*)1900 where *gang-arg* is one of:1901 [**num** :] *int-expr*
1902 **static** : *size-expr*1903 and *gang-arg-list* may have at most one **num** and one **static** argument,1904 and where *size-expr* is one of:1905 *****
1906 *int-expr*

1907

1908 Some clauses are only valid in the context of a **kernels** construct; see the descriptions below.1909 An *orphaned loop* construct is a **loop** construct that is not lexically enclosed within a compute
1910 construct. The parent compute construct of a **loop** construct is the nearest compute construct that
1911 lexically contains the **loop** construct.1912 A **loop** construct is *data-independent* if it has an **independent** clause that is determined explic-
1913 itly, implicitly, or from an **auto** clause. A **loop** construct is *sequential* if it has a **seq** clause that
1914 is determined explicitly or from an **auto** clause.1915 When *do-loop* is a **do concurrent**, the OpenACC **loop** construct applies to the loop for each
1916 index in the *concurrent-header*. The **loop** construct can describe what type of parallelism to use
1917 to execute all the loops, and declares all indices appearing in the *concurrent-header* to be implicitly
1918 private. If the **loop** construct that is associated with **do concurrent** is combined with a compute
1919 construct then *concurrent-locality* is processed as follows: variables appearing in a *local* are treated
1920 as appearing in a **private** clause; variables appearing in a *local_init* are treated as appearing in a

1921 **firstprivate** clause; variables appearing in a *shared* are treated as appearing in a **copy** clause;
 1922 and a *default(none)* locality spec implies a **default (none)** clause on the compute construct. If
 1923 the **loop** construct is not combined with a compute construct, the behavior is implementation-
 1924 defined.

1925 Restrictions

1926 • Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **independent**, **auto**, and **tile**
 1927 clauses may follow a **device_type** clause.

1928 • The *int-expr* argument to the **worker** and **vector** clauses must be invariant in the kernels
 1929 region.

1930 • A loop associated with a **loop** construct that does not have a **seq** clause must be written to
 1931 meet all of the following conditions:

1932 – The loop variable must be of integer, C/C++ pointer, or C++ random-access iterator
 1933 type.

1934 – The loop variable must monotonically increase or decrease in the direction of its termi-
 1935 nation condition.

1936 – The loop trip count must be computable in constant time when entering the **loop** con-
 1937 struct.

1938 For a C++ range-based **for** loop, the loop variable identified by the above conditions is the
 1939 internal iterator, such as a pointer, that the compiler generates to iterate the range. It is not the
 1940 variable declared by the **for** loop.

1941 • Only one of the **seq**, **independent**, and **auto** clauses may appear.

1942 • A **gang**, **worker**, or **vector** clause may not appear if a **seq** clause appears.

1943 • A **tile** and **collapse** clause may not appear on **loop** that is associated with **do concurrent**.

1944 2.9.1 collapse clause

1945 The **collapse** clause is used to specify how many tightly nested loops are associated with the
 1946 **loop** construct. The argument to the **collapse** clause must be a constant positive integer expres-
 1947 sion. If no **collapse** clause appears, only the immediately following loop is associated with the
 1948 **loop** construct.

1949 If more than one loop is associated with the **loop** construct, the iterations of all the associated loops
 1950 are all scheduled according to the rest of the clauses. The trip count for all loops associated with
 1951 the **collapse** clause must be computable and invariant in all the loops. The particular integer
 1952 type used to compute the trip count for the collapsed loops is implementation defined. However, the
 1953 integer type used for the trip count has at least the precision of each loop variable of the associated
 1954 loops.

1955 It is implementation-defined whether a **gang**, **worker** or **vector** clause on the construct is ap-
 1956 plied to each loop, or to the linearized iteration space.

1957 2.9.2 gang clause

1958 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
 1959 the **gang** clause specifies that the iterations of the associated loop or loops are to be executed in

1960 parallel by distributing the iterations among the gangs created by the **parallel** construct. A
1961 **loop** construct with the **gang** clause transitions a compute region from gang-redundant mode to
1962 gang-partitioned mode. The number of gangs is controlled by the **parallel** construct; only the
1963 **static** argument is allowed. The loop iterations must be data independent, except for *vars* which
1964 appear in a **reduction** clause or which are modified in an atomic region. The region of a loop
1965 with the **gang** clause may not contain another loop with the **gang** clause unless within a nested
1966 compute region.

1967 When the parent compute construct is a **kernels** construct, the **gang** clause specifies that the
1968 iterations of the associated loop or loops are to be executed in parallel across the gangs. An argument
1969 with no keyword or with the **num** keyword is allowed only when the **num_gangs** does not appear
1970 on the **kernels** construct. If an argument with no keyword or an argument after the **num** keyword
1971 appears, it specifies how many gangs to use to execute the iterations of this loop. The region of a
1972 loop with the **gang** clause may not contain another loop with a **gang** clause unless within a nested
1973 compute region.

1974 The scheduling of loop iterations to gangs is not specified unless the **static** modifier appears as
1975 an argument. If the **static** modifier appears with an integer expression, that expression is used
1976 as a *chunk* size. If the static modifier appears with an asterisk, the implementation will select a
1977 *chunk* size. The iterations are divided into chunks of the selected *chunk* size, and the chunks are
1978 assigned to gangs starting with gang zero and continuing in round-robin fashion. Two **gang** loops
1979 in the same parallel region with the same number of iterations, and with **static** clauses with the
1980 same argument, will assign the iterations to gangs in the same manner. Two **gang** loops in the
1981 same kernels region with the same number of iterations, the same number of gangs to use, and with
1982 **static** clauses with the same argument, will assign the iterations to gangs in the same manner.

1983 A **gang** clause without arguments is implied on a data-independent **loop** construct without an
1984 explicit **gang** clause if the following conditions hold while ignoring **gang**, **worker**, and **vector**
1985 clauses on any sequential **loop** constructs:

- 1986 • This **loop** construct's parent compute construct, if any, is not a **kernels** construct.
- 1987 • An explicit **gang** clause would be permitted on this **loop** construct.
- 1988 • For every lexically enclosing data-independent **loop** construct, either an explicit **gang** clause
1989 would not be permitted on the enclosing **loop** construct, or the enclosing **loop** construct
1990 lexically encloses a compute construct that lexically encloses this **loop** construct.

1991 **Note:** As a performance optimization, the implementation might select different levels of paral-
1992 lelism for a **loop** construct than specified by explicitly or implicitly determined clauses as long
1993 as it can prove program semantics are preserved. In particular, the implementation must consider
1994 semantic differences between gang-redundant and gang-partitioned mode. For example, in a series
1995 of tightly nested, data-independent **loop** constructs, implementations often move gang-partitioning
1996 from one **loop** construct to another without affecting semantics.

1997 **Note:** If the **auto** or **device_type** clause appears on a **loop** construct, it is the programmer's
1998 responsibility to ensure that program semantics are the same regardless of whether the **auto** clause
1999 is treated as **independent** or **seq** and regardless of the device type for which the program is
2000 compiled. In particular, the programmer must consider the effect on both explicitly and implicitly
2001 determined **gang** clauses and thus on gang-redundant and gang-partitioned mode. Examples in
2002 Section 2.9.11 demonstrate this issue for the **auto** clause.

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. The region of a loop with the **worker** clause may not contain a loop with the **gang** or **worker** clause unless within a nested compute 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. The region of a loop with the **worker** clause may not contain a loop with a **gang** or **worker** clause unless within a nested compute region.

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. The region of a loop with the **vector** clause may not contain a loop with the **gang**, **worker**, or **vector** clause unless within a nested compute 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. The region of a loop with the **vector** clause may not contain a loop with a **gang**, **worker**, or **vector** clause unless within a nested compute region.

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.

2.9.8 tile clause

The **tile** clause specifies that the implementation should split each loop in the loop nest into two loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the **tile** clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression or an asterisk. If there are n tile sizes in the list, the **loop** 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.

2084 2.9.10 private clause

2085 The **private** clause on a **loop** construct specifies that a copy of each item in *var-list* will be
2086 created. If the body of the loop is executed in *vector-partitioned* mode, a copy of the item is created
2087 for each thread associated with each vector lane. If the body of the loop is executed in *worker-*
2088 *partitioned vector-single* mode, a copy of the item is created for and shared across the set of threads
2089 associated with all the vector lanes of each worker. Otherwise, a copy of the item is created for and
2090 shared across the set of threads associated with all the vector lanes of all the workers of each gang.

2091 Restrictions

- 2092 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private**
2093 clauses.

2094 2.9.11 reduction clause

2095 The **reduction** clause specifies a reduction operator and one or more *vars*. For each reduction
2096 *var*, a private copy is created in the same manner as for a **private** clause on the **loop** construct,
2097 and initialized for that operator; see the table in Section 2.5.15 reduction clause. After the loop, the
2098 values for each thread are combined using the specified reduction operator, and the result combined
2099 with the value of the original *var* and stored in the original *var*. If the original *var* is not private,
2100 this update occurs by the end of the compute region, and any access to the original *var* is undefined
2101 within the compute region. Otherwise, the update occurs at the end of the loop. If the reduction
2102 *var* is an array or subarray, the reduction operation is logically equivalent to applying that reduction
2103 operation to each array element of the array or subarray individually. If the reduction *var* is a com-
2104 posite variable, the reduction operation is logically equivalent to applying that reduction operation
2105 to each member of the composite variable individually.

2106 If a variable is involved in a reduction that spans multiple nested loops where two or more of those
2107 loops have associated **loop** directives, a **reduction** clause containing that variable must appear
2108 on each of those **loop** directives.

2109 Restrictions

- 2110 • A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name,
2111 an array element, or a subarray (refer to Section 2.7.1).
- 2112 • Reduction clauses on nested constructs for the same reduction *var* must have the same reduc-
2113 tion operator.
- 2114 • Every *var* in a **reduction** clause appearing on an orphaned **loop** construct must be private.
- 2115 • The restrictions for a **reduction** clause on a compute construct listed in in Section 2.5.15
2116 reduction clause also apply to a **reduction** clause on a **loop** construct.
- 2117 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
2118 **reduction** clauses.
- 2119 • See Section 2.6.2 Variables with Implicitly Determined Data Attributes for a restriction re-
2120 quiring certain loop reduction variables to have explicit data clauses on their parent compute
2121 constructs.

2122

2123 **Examples**

2124

- 2125 • **x** is not private at the **loop** directive below, so its reduction normally updates **x** at the end
 2126 of the parallel region, where gangs synchronize. When possible, the implementation might
 2127 choose to partially update **x** at the loop exit instead, or fully if **num_gangs(1)** were added
 2128 to the **parallel** directive. However, portable applications cannot rely on such early up-
 2129 dates, so accesses to **x** are undefined within the parallel region outside the loop.

```
2130     int x = 0;
2131     #pragma acc parallel copy(x)
2132     {
2133         // gang-shared x undefined
2134         #pragma acc loop gang worker vector reduction(+:x)
2135         for (int i = 0; i < I; ++i)
2136             x += 1; // vector-private x modified
2137         // gang-shared x undefined
2138     } // gang-shared x updated for gang/worker/vector reduction
2139     // x = I
```

- 2140 • **x** is private at each of the innermost two **loop** directives below, so each of their reductions
 2141 updates **x** at the loop's exit. However, **x** is not private at the outer **loop** directive, so its
 2142 reduction updates **x** by the end of the parallel region instead.

```
2143     int x = 0;
2144     #pragma acc parallel copy(x)
2145     {
2146         // gang-shared x undefined
2147         #pragma acc loop gang reduction(+:x)
2148         for (int i = 0; i < I; ++i) {
2149             #pragma acc loop worker reduction(+:x)
2150             for (int j = 0; j < J; ++j) {
2151                 #pragma acc loop vector reduction(+:x)
2152                 for (int k = 0; k < K; ++k) {
2153                     x += 1; // vector-private x modified
2154                 } // worker-private x updated for vector reduction
2155             } // gang-private x updated for worker reduction
2156         }
2157         // gang-shared x undefined
2158     } // gang-shared x updated for gang reduction
2159     // x = I * J * K
```

- 2160 • At each **loop** directive below, **x** is private and **y** is not private due to the data clauses on
 2161 the **parallel** directive. Thus, each reduction updates **x** at the loop exit, but each reduction
 2162 updates **y** by the end of the parallel region instead.

```
2163     int x = 0, y = 0;
2164     #pragma acc parallel firstprivate(x) copy(y)
2165     {
2166         // gang-private x = 0; gang-shared y undefined
```

```

2167     #pragma acc loop seq reduction(+:x,y)
2168     for (int i = 0; i < I; ++i) {
2169         x += 1; y += 2; // loop-private x and y modified
2170     } // gang-private x updated for seq reduction (trivial reduction)
2171     // gang-private x = I; gang-shared y undefined
2172     #pragma acc loop worker reduction(+:x,y)
2173     for (int i = 0; i < I; ++i) {
2174         x += 1; y += 2; // worker-private x and y modified
2175     } // gang-private x updated for worker reduction
2176     // gang-private x = 2 * I; gang-shared y undefined
2177     #pragma acc loop vector reduction(+:x,y)
2178     for (int i = 0; i < I; ++i) {
2179         x += 1; y += 2; // vector-private x and y modified
2180     } // gang-private x updated for vector reduction
2181     // gang-private x = 3 * I; gang-shared y undefined
2182     } // gang-shared y updated for gang/seq/worker/vector reductions
2183     // 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.

```

2188     int x = 0;
2189     #pragma acc parallel loop worker reduction(+:x)
2190     for (int i = 0; i < I; ++i) {
2191         x += 1; // worker-private x modified
2192     } // gang-shared x updated for gang/worker reduction
2193     // x = I
2194
2195     int x = 0;
2196     #pragma acc parallel copy(x)
2197     {
2198         // gang-shared x undefined
2199         #pragma acc loop worker reduction(+:x)
2200         for (int i = 0; i < I; ++i) {
2201             x += 1; // worker-private x modified
2202         }
2203         // gang-shared x undefined
2204     } // gang-shared x updated for gang/worker reduction
2205     // 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.

```

2211     int x = 0;
2212     const int *arr = /*array of I values*/;

```

```

2213     #pragma acc parallel copy(x)
2214     {
2215         // gang-shared x undefined
2216         #pragma acc loop auto gang reduction(max:x)
2217         for (int i = 0; i < I; ++i) {
2218             // complex loop body
2219             x = x < arr[i] ? arr[i] : x; // gang or loop-private x modified
2220         }
2221         // gang-shared x undefined
2222     } // gang-shared x updated for gang or gang/seq reduction
2223     // 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.

```

2229     int x = 0;
2230     const int *arr = /*array of I values*/;
2231     #pragma acc parallel copy(x)
2232     {
2233         // gang-shared x undefined
2234         #pragma acc loop auto gang reduction(+:x)
2235         for (int i = 0; i < I; ++i) {
2236             // complex loop body
2237             x += arr[i]; // gang or loop-private x modified
2238         }
2239         // gang-shared x undefined
2240     } // gang-shared x updated for gang or gang/seq reduction
2241     // 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.

```

2247     int x = 0, y = 0;
2248     #pragma acc parallel copy(x) reduction(+:x,y)
2249     {
2250         int z = 0;
2251         #pragma acc loop gang vector reduction(+:x,z)
2252         for (int i = 0; i < I; ++i) {
2253             x += 1; z += 2; // vector-private x and z modified
2254         } // gang-private x and z updated for vector reduction (trivial 1-gang reduction)
2255         y += z; // gang-private y modified
2256     } // gang-shared x and y updated for gang reduction
2257     // x = I; y = I * 2

```


2258

2259

2260 2.10 Cache Directive

2261 Summary

2262 The **cache** directive may appear at the top of (inside of) a loop. It specifies array elements or
2263 subarrays that should be fetched into the highest level of the cache for the body of the loop.

2264 Syntax

2265 In C and C++, the syntax of the **cache** directive is

```
2266 #pragma acc cache( [readonly:]var-list ) new-line
```

2267 In Fortran, the syntax of the **cache** directive is

```
2268 !$acc cache( [readonly:]var-list )
```

2269 A *var* in a **cache** directive must be a single array element or a simple subarray. In C and C++,
2270 a simple subarray is an array name followed by an extended array range specification in brackets,
2271 with start and length, such as

```
2272 arr[lower:length]
```

2273 where the lower bound is a constant, loop invariant, or the **for** loop variable plus or minus a
2274 constant or loop invariant, and the length is a constant.

2275 In Fortran, a simple subarray is an array name followed by a comma-separated list of range specifi-
2276 cations in parentheses, with lower and upper bound subscripts, such as

```
2277 arr(lower:upper, lower2:upper2)
```

2278 The lower bounds must be constant, loop invariant, or the **do** loop variable plus or minus a constant
2279 or loop invariant; moreover the difference between the corresponding upper and lower bounds must
2280 be a constant.

2281 If the optional **readonly** modifier appears, then the implementation may assume that the data
2282 referenced by any *var* in that directive is never written to within the applicable region.

2283 Restrictions

- 2284 • If an array element or subarray is listed in a **cache** directive, all references to that array
2285 during execution of that loop iteration must not refer to elements of the array outside the
2286 index range specified in the **cache** directive.
- 2287 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **cache**
2288 directives.

2289 2.11 Combined Constructs

2290 Summary

2291 The combined OpenACC **parallel loop**, **serial loop**, and **kernels loop** constructs are
2292 shortcuts for specifying a **loop** construct nested immediately inside a **parallel**, **serial**, or
2293 **kernels** construct. The meaning is identical to explicitly specifying a **parallel**, **serial**, or
2294 **kernels** construct containing a **loop** construct. Any clause that is allowed on a **parallel** or

2295 **loop** construct is allowed on the **parallel loop** construct; any clause allowed on a **serial** or
 2296 **loop** construct is allowed on a **serial loop** construct; and any clause allowed on a **kernels**
 2297 or **loop** construct is allowed on a **kernels loop** construct.

2298 Syntax

2299 In C and C++, the syntax of the **parallel loop** construct is

```
2300     #pragma acc parallel loop [clause-list] new-line  
2301         for loop
```

2302 In Fortran, the syntax of the **parallel loop** construct is

```
2303     !$acc parallel loop [clause-list]  
2304         do loop  
2305     [!$acc end parallel loop]
```

2306 The associated structured block is the loop which must immediately follow the directive. Any of
 2307 the **parallel** or **loop** clauses valid in a parallel region may appear.

2308 In C and C++, the syntax of the **serial loop** construct is

```
2309     #pragma acc serial loop [clause-list] new-line  
2310         for loop
```

2311 In Fortran, the syntax of the **serial loop** construct is

```
2312     !$acc serial loop [clause-list]  
2313         do loop  
2314     [!$acc end serial loop]
```

2315 The associated structured block is the loop which must immediately follow the directive. Any of
 2316 the **serial** or **loop** clauses valid in a serial region may appear.

2317 In C and C++, the syntax of the **kernels loop** construct is

```
2318     #pragma acc kernels loop [clause-list] new-line  
2319         for loop
```

2320 In Fortran, the syntax of the **kernels loop** construct is

```
2321     !$acc kernels loop [clause-list]  
2322         do loop  
2323     [!$acc end kernels loop]
```

2324 The associated structured block is the loop which must immediately follow the directive. Any of
 2325 the **kernels** or **loop** clauses valid in a kernels region may appear.

2326 A **private** or **reduction** clause on a combined construct is treated as if it appeared on the
 2327 **loop** construct. In addition, a **reduction** clause on a combined construct implies a **copy** clause
 2328 as described in Section 2.6.2.

2329 Restrictions

- 2330 • The restrictions for the **parallel**, **serial**, **kernels**, and **loop** constructs apply.

2331 2.12 Atomic Construct

2332 Summary

2333 An **atomic** construct ensures that a specific storage location is accessed and/or updated atomically,
 2334 preventing simultaneous reading and writing by gangs, workers, and vector threads that could result
 2335 in indeterminate values.

2336 Syntax

2337 In C and C++, the syntax of the **atomic** constructs is:

```
2338     #pragma acc atomic [ atomic-clause ] new-line  
2339         expression-stmt
```

2340 OR:

```
2341     #pragma acc atomic capture new-line  
2342         structured block
```

2343 Where *atomic-clause* is one of **read**, **write**, **update**, or **capture**. The *expression-stmt* is an
 2344 expression statement with one of the following forms:

2345 If the *atomic-clause* is **read**:

```
2346     v = x;
```

2347 If the *atomic-clause* is **write**:

```
2348     x = expr;
```

2349 If the *atomic-clause* is **update** or no clause appears:

```
2350     x++;  
2351     x--;  
2352     ++x;  
2353     --x;  
2354     x binop= expr;  
2355     x = x binop expr;  
2356     x = expr binop x;
```

2357 If the *atomic-clause* is **capture**:

```
2358     v = x++;  
2359     v = x--;  
2360     v = ++x;  
2361     v = --x;  
2362     v = x binop= expr;  
2363     v = x = x binop expr;  
2364     v = x = expr binop x;
```

2365 The *structured-block* is a structured block with one of the following forms:

```
2366     { v = x; x binop= expr; }  
2367     { x binop= expr; v = x; }  
2368     { v = x; x = x binop expr; }  
2369     { v = x; x = expr binop x; }
```

```

2370 { x = x binop expr; v = x; }
2371 { x = expr binop x; v = x; }
2372 { v = x; x = expr; }
2373 { v = x; x++; }
2374 { v = x; ++x; }
2375 { ++x; v = x; }
2376 { x++; v = x; }
2377 { v = x; x--; }
2378 { v = x; --x; }
2379 { --x; v = x; }
2380 { x--; v = x; }

```

2381 In the preceding expressions:

- 2382 • **x** and **v** (as applicable) are both l-value expressions with scalar type.
- 2383 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
2384 the same storage location.
- 2385 • Neither of **v** and *expr* (as applicable) may access the storage location designated by **x**.
- 2386 • Neither of **x** and *expr* (as applicable) may access the storage location designated by **v**.
- 2387 • *expr* is an expression with scalar type.
- 2388 • *binop* is one of +, *, -, /, &, ^, |, <<, or >>.
- 2389 • *binop*, *binop*=, ++, and -- are not overloaded operators.
- 2390 • The expression **x** *binop* *expr* must be mathematically equivalent to **x** *binop* (*expr*). This
2391 requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by
2392 using parentheses around *expr* or subexpressions of *expr*.
- 2393 • The expression *expr* *binop* **x** must be mathematically equivalent to (*expr*) *binop* **x**. This
2394 requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*,
2395 or by using parentheses around *expr* or subexpressions of *expr*.
- 2396 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is
2397 unspecified.

2398 In Fortran the syntax of the **atomic** constructs is:

```

2399 !$acc atomic read
2400 capture-statement
2401 [$acc end atomic]

```

2402 OR

```

2403 !$acc atomic write
2404 write-statement
2405 [$acc end atomic]

```

2406 OR

```

2407 !$acc atomic [update]
2408 update-statement

```

2409 [!\$acc end atomic]

2410 OR

2411 !\$acc atomic capture

2412 *update-statement*

2413 *capture-statement*

2414 !\$acc end atomic

2415 OR

2416 !\$acc atomic capture

2417 *capture-statement*

2418 *update-statement*

2419 !\$acc end atomic

2420 OR

2421 !\$acc atomic capture

2422 *capture-statement*

2423 *write-statement*

2424 !\$acc end atomic

2425 where *write-statement* has the following form (if *atomic-clause* is **write** or **capture**):

2426 **x** = **expr**

2427 where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

2428 **v** = **x**

2429 and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**,
2430 or no clause appears):

2431 **x** = **x** *operator* *expr*

2432 **x** = *expr* *operator* **x**

2433 **x** = *intrinsic_procedure_name* (**x**, *expr-list*)

2434 **x** = *intrinsic_procedure_name* (*expr-list*, **x**)

2435 In the preceding statements:

- 2436 • **x** and **v** (as applicable) are both scalar variables of intrinsic type.
- 2437 • **x** must not be an allocatable variable.
- 2438 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
2439 the same storage location.
- 2440 • None of **v**, *expr*, and *expr-list* (as applicable) may access the same storage location as **x**.
- 2441 • None of **x**, *expr*, and *expr-list* (as applicable) may access the same storage location as **v**.
- 2442 • *expr* is a scalar expression.
- 2443 • *expr-list* is a comma-separated, non-empty list of scalar expressions. If *intrinsic_procedure_name*
2444 refers to **iand**, **ior**, or **ieor**, exactly one expression must appear in *expr-list*.

- 2445 • *intrinsic_procedure_name* is one of **max**, **min**, **iand**, **ior**, or **ieor**. *operator* is one of **+**,
- 2446 *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**
- 2447 • The expression **x operator expr** must be mathematically equivalent to **x operator (expr)**.
- 2448 This requirement is satisfied if the operators in *expr* have precedence greater than *operator*,
- 2449 or by using parentheses around *expr* or subexpressions of *expr*.
- 2450 • The expression *expr operator x* must be mathematically equivalent to **(expr) operator x**.
- 2451 This requirement is satisfied if the operators in *expr* have precedence equal to or greater than
- 2452 *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- 2453 • *intrinsic_procedure_name* must refer to the intrinsic procedure name and not to other program
- 2454 entities.
- 2455 • *operator* must refer to the intrinsic operator and not to a user-defined operator. All assign-
- 2456 ments must be intrinsic assignments.
- 2457 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is
- 2458 unspecified.

2459 An **atomic** construct with the **read** clause forces an atomic read of the location designated by **x**.
 2460 An **atomic** construct with the **write** clause forces an atomic write of the location designated by
 2461 **x**.

2462 An **atomic** construct with the **update** clause forces an atomic update of the location designated
 2463 by **x** using the designated operator or intrinsic. Note that when no clause appears, the semantics
 2464 are equivalent to **atomic update**. Only the read and write of the location designated by **x** are
 2465 performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect
 2466 to the read or write of the location designated by **x**.

2467 An **atomic** construct with the **capture** clause forces an atomic update of the location designated
 2468 by **x** using the designated operator or intrinsic while also capturing the original or final value of
 2469 the location designated by **x** with respect to the atomic update. The original or final value of the
 2470 location designated by **x** is written into the location designated by **v** depending on the form of the
 2471 **atomic** construct structured block or statements following the usual language semantics. Only
 2472 the read and write of the location designated by **x** are performed mutually atomically. Neither the
 2473 evaluation of *expr* or *expr-list*, nor the write to the location designated by **v**, need to be atomic with
 2474 respect to the read or write of the location designated by **x**.

2475 For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs
 2476 enforces mutually exclusive access to the locations designated by **x**. To avoid race conditions, all
 2477 accesses of the locations designated by **x** that could potentially occur in parallel must be protected
 2478 with an **atomic** construct.

2479 Atomic regions do not guarantee exclusive access with respect to any accesses outside of atomic re-
 2480 gions to the same storage location **x** even if those accesses occur during the execution of a reduction
 2481 clause.

2482 If the storage location designated by **x** is not size-aligned (that is, if the byte alignment of **x** is not a
 2483 multiple of the size of **x**), then the behavior of the atomic region is implementation-defined.

2484 Restrictions

- 2485 • All atomic accesses to the storage locations designated by **x** throughout the program are
- 2486 required to have the same type and type parameters.

- Storage locations designated by **x** must be less than or equal in size to the largest available native atomic operator width.

2.13 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.

- 2526 • In Fortran, pointer arrays may appear, but pointer association is not preserved in device mem-
2527 ory.
- 2528 • In a Fortran *module* declaration section, only **create**, **copyin**, **device_resident**, and
2529 **link** clauses are allowed.
- 2530 • In C or C++ global or namespace scope, only **create**, **copyin**, **deviceptr**,
2531 **device_resident** and **link** clauses are allowed.
- 2532 • C and C++ *extern* variables may only appear in **create**, **copyin**, **deviceptr**,
2533 **device_resident** and **link** clauses on a **declare** directive.
- 2534 • In C or C++, the **link** clause must appear at global or namespace scope or the arguments
2535 must be *extern* variables. In Fortran, the **link** clause must appear in a *module* declaration
2536 section, or the arguments must be *common block* names enclosed in slashes.
- 2537 • In C or C++, a **longjmp** call in the region must return to a **setjmp** call within the region.
- 2538 • In C++, an exception thrown in the region must be handled within the region.
- 2539 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional dummy arguments
2540 in data clauses, including **device_resident** clauses.

2541 2.13.1 device_resident clause

2542 Summary

2543 The **device_resident** clause specifies that the memory for the named variables should be
2544 allocated in the current device memory and not in local memory. The host may not be able to access
2545 variables in a **device_resident** clause. The accelerator data lifetime of global variables or
2546 common blocks that appear in a **device_resident** clause is the entire execution of the program.

2547 In Fortran, if the variable has the Fortran *allocatable* attribute, the memory for the variable will
2548 be allocated in and deallocated from the current device memory when the host thread executes
2549 an **allocate** or **deallocate** statement for that variable, if the current device is a non-shared
2550 memory device. If the variable has the Fortran *pointer* attribute, it may be allocated or deallocated
2551 by the host in the current device memory, or may appear on the left hand side of a pointer assignment
2552 statement, if the right hand side variable itself appears in a **device_resident** clause.

2553 In Fortran, the argument to a **device_resident** clause may be a *common block* name enclosed
2554 in slashes; in this case, all declarations of the common block must have a matching
2555 **device_resident** clause. In this case, the *common block* will be statically allocated in de-
2556 vice memory, and not in local memory. The *common block* will be available to accelerator routines;
2557 see Section 2.15 Procedure Calls in Compute Regions.

2558 In a Fortran *module* declaration section, a *var* in a **device_resident** clause will be available to
2559 accelerator subprograms.

2560 In C or C++ global scope, a *var* in a **device_resident** clause will be available to accelerator
2561 routines. A C or C++ *extern* variable may appear in a **device_resident** clause only if the
2562 actual declaration and all *extern* declarations are also followed by **device_resident** clauses.

2563 2.13.2 create clause

2564 For data in shared memory, no action is taken.

2565 For data not in shared memory, the **create** clause on a **declare** directive behaves as follows,
 2566 for each *var* in *var-list*:

- 2567 • At entry to an implicit data region where the **declare** directive appears:
 - 2568 – If *var* is present, a *present increment* action with the structured reference counter is
 - 2569 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 2570 – Otherwise, a *create* action with the structured reference counter is performed. If *var* is
 - 2571 a pointer reference, an *attach* action is performed.
- 2572 • At exit from an implicit data region where the **declare** directive appears:
 - 2573 – If the structured reference counter for *var* is zero, no action is taken.
 - 2574 – Otherwise, a *present decrement* action with the structured reference counter is per-
 - 2575 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
 - 2576 and dynamic reference counters are zero, a *delete* action is performed.

2577 If the **declare** directive appears in a global context, then the data in *var-list* is statically allocated
 2578 in device memory and the structured reference counter is set to one.

2579 In Fortran, if a variable *var* in *var-list* has the Fortran *allocatable* or *pointer* attribute, then:

- 2580 • An **allocate** statement for *var* will allocate memory in both local memory as well as in the
 2581 current device memory, for a non-shared memory device, and the dynamic reference counter
 2582 will be set to one.
- 2583 • A **deallocate** statement for *var* will deallocate memory from both local memory as well
 2584 as the current device memory, for a non-shared memory device, and the dynamic reference
 2585 counter will be set to zero. If the structured reference counter is not zero, a runtime error is
 2586 issued.

2587 In Fortran, if a variable *var* in *var-list* has the Fortran *pointer* attribute, then it may appear on the
 2588 left hand side of a pointer assignment statement, if the right hand side variable itself appears in a
 2589 **create** clause.

2590 Errors

- 2591 • In Fortran, an **acc_error_present** error is issued at a deallocate statement if the struc-
 2592 tured reference counter is not zero.

2593 See Section 5.2.2.

2594 2.13.3 link clause

2595 The **link** clause is used for large global host static data that is referenced within an accelerator
 2596 routine and that should have a dynamic data lifetime on the device. The **link** clause specifies that
 2597 only a global link for the named variables should be statically created in accelerator memory. The
 2598 host data structure remains statically allocated and globally available. The device data memory will
 2599 be allocated only when the global variable appears on a data clause for a **data** construct, compute
 2600 construct, or **enter data** directive. The arguments to the **link** clause must be global data. A
 2601 **declare link** clause must be visible everywhere the global variables or common block variables
 2602 are explicitly or implicitly used in a data clause, compute construct, or accelerator routine. The
 2603 global variable or *common block* variables may be used in accelerator routines. The accelerator

2604 data lifetime of variables or common blocks that appear in a **link** clause is the data region that
 2605 allocates the variable or common block with a data clause, or from the execution of the **enter**
 2606 **data** directive that allocates the data until an **exit data** directive deallocates it or until the end
 2607 of the program.

2608 **2.14 Executable Directives**

2609 **2.14.1 Init Directive**

2610 **Summary**

2611 The **init** directive initializes the runtime for the given device or devices of the given device type.
 2612 This can be used to isolate any initialization cost from the computational cost, when collecting
 2613 performance statistics. If no device type appears all devices will be initialized. An **init** directive
 2614 may be used in place of a call to the **acc_init** or **acc_init_device** runtime API routine, as
 2615 described in Section 3.2.7.

2616 **Syntax**

2617 In C and C++, the syntax of the **init** directive is:

```
2618     #pragma acc init [clause-list] new-line
```

2619 In Fortran the syntax of the **init** directive is:

```
2620     !$acc init [clause-list]
```

2621 where *clause* is one of the following:

```
2622     device_type ( device-type-list )
```

```
2623     device_num ( int-expr )
```

```
2624     if ( condition )
```

2625

2626 **device_type clause**

2627 The **device_type** clause specifies the type of device that is to be initialized in the runtime. If the
 2628 **device_type** clause appears, then the *acc-current-device-type-var* for the current thread is set to
 2629 the argument value. If no **device_num** clause appears then all devices of this type are initialized.

2630 **device_num clause**

2631 The **device_num** clause specifies the device id to be initialized. If the **device_num** clause
 2632 appears, then the *acc-current-device-num-var* for the current thread is set to the argument value. If
 2633 no **device_type** clause appears, then the specified device id will be initialized for all available
 2634 device types.

2635 **if clause**

2636 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
 2637 perform the initialization unconditionally. When an **if** clause appears, the implementation will
 2638 generate code to conditionally perform the initialization only when the *condition* evaluates to *true*.

2639 **Restrictions**

- 2640 • This directive may only appear in code executed on the host.
- 2641 • If the directive is called more than once without an intervening **acc_shutdown** call or
- 2642 **shutdown** directive, with a different value for the device type argument, the behavior is
- 2643 implementation-defined.
- 2644 • If some accelerator regions are compiled to only use one device type, using this directive with
- 2645 a different device type may produce undefined behavior.

2646 **Errors**

- 2647 • An **acc_error_device_type_unavailable** error is issued if a **device_type** clause
- 2648 appears and no device of that device type is available, or if no **device_type** clause appears
- 2649 and no device of the current device type is available.
- 2650 • An **acc_error_device_unavailable** error is issued if a **device_num** clause ap-
- 2651 pears and the *int-expr* is not a valid device number or that device is not available, or if no
- 2652 **device_num** clause appears and the current device is not available.
- 2653 • An **acc_error_device_init** error is issued if the device cannot be initialized.

2654 See Section 5.2.2.

2655 **2.14.2 Shutdown Directive**2656 **Summary**

2657 The **shutdown** directive shuts down the connection to the given device or devices of the given
 2658 device type, and frees any associated runtime resources. This ends all data lifetimes in device
 2659 memory, which effectively sets structured and dynamic reference counters to zero. A **shutdown**
 2660 directive may be used in place of a call to the **acc_shutdown** or **acc_shutdown_device**
 2661 runtime API routine, as described in Section 3.2.8.

2662 **Syntax**2663 In C and C++, the syntax of the **shutdown** directive is:2664 **#pragma acc shutdown** [*clause-list*] *new-line*2665 In Fortran the syntax of the **shutdown** directive is:2666 **!\$acc shutdown** [*clause-list*]2667 where *clause* is one of the following:2668 **device_type** (*device-type-list*)2669 **device_num** (*int-expr*)2670 **if** (*condition*)

2671

2672 **device_type clause**

2673 The **device_type** clause specifies the type of device that is to be disconnected from the runtime.
 2674 If no **device_num** clause appears then all devices of this type are disconnected.

2675 **device_num clause**2676 The **device_num** clause specifies the device id to be disconnected.

2677 If no clauses appear then all available devices will be disconnected.

2678 **if clause**2679 The **if** clause is optional; when there is no **if** clause, the implementation will generate code
2680 to perform the shutdown unconditionally. When an **if** clause appears, the implementation will
2681 generate code to conditionally perform the shutdown only when the *condition* evaluates to *true*.2682 **Restrictions**

- 2683
- This directive may only appear in code executed on the host.

2684 **Errors**

- 2685
- An **acc_error_device_type_unavailable** error is issued if a **device_type** clause
2686 appears and no device of that device type is available,
 - An **acc_error_device_unavailable** error is issued if a **device_num** clause ap-
2687 pears 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
2689 device.
2690

2691 See Section 5.2.2.

2692 **2.14.3 Set Directive**2693 **Summary**2694 The **set** directive provides a means to modify internal control variables using directives. Each form
2695 of the **set** directive is functionally equivalent to a matching runtime API routine.2696 **Syntax**2697 In C and C++, the syntax of the **set** directive is:2698

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

2699 In Fortran the syntax of the **set** directive is:2700

```
!$acc set [clause-list]
```

2701 where *clause* is one of the following2702

```
default_async ( int-expr )  
2703 device_num ( int-expr )  
2704 device_type ( device-type-list )  
2705 if ( condition )
```

2706 **default_async clause**2707 The **default_async** clause specifies the asynchronous queue that should be used if no queue ap-
2708 pears and changes the value of *acc-default-async-var* for the current thread to the argument value.
2709 If the value is **acc_async_default**, the value of *acc-default-async-var* will revert to the ini-
2710 tial value, which is implementation-defined. A **set default_async** directive is functionally

2711 equivalent to a call to the **acc_set_default_async** runtime API routine, as described in Sec-
 2712 tion 3.2.14.

2713 **device_num clause**

2714 The **device_num** clause specifies the device number to set as the default device for accelerator
 2715 regions and changes the value of *acc-current-device-num-var* for the current thread to the argument
 2716 value. If the value of **device_num** argument is negative, the runtime will revert to the default be-
 2717 havior, which is implementation-defined. A **set device_num** directive is functionally equivalent
 2718 to the **acc_set_device_num** runtime API routine, as described in Section 3.2.4.

2719 **device_type clause**

2720 The **device_type** clause specifies the device type to set as the default device type for accelerator
 2721 regions and sets the value of *acc-current-device-type-var* for the current thread to the argument
 2722 value. If the value of the **device_type** argument is zero or the clause does not appear, the
 2723 selected device number will be used for all attached accelerator types. A **set device_type**
 2724 directive is functionally equivalent to a call to the **acc_set_device_type** runtime API routine,
 2725 as described in Section 3.2.2.

2726 **if clause**

2727 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
 2728 perform the set operation unconditionally. When an **if** clause appears, the implementation will
 2729 generate code to conditionally perform the set operation only when the *condition* evaluates to *true*.

2730 **Restrictions**

- 2731 • This directive may only appear in code executed on the host.
- 2732 • Passing **default_async** the value of **acc_async_noval** has no effect.
- 2733 • Passing **default_async** the value of **acc_async_sync** will cause all asynchronous
 2734 directives in the default asynchronous queue to become synchronous.
- 2735 • Passing **default_async** the value of **acc_async_default** will restore the default
 2736 asynchronous queue to the initial value, which is implementation-defined.
- 2737 • At least one **default_async**, **device_num**, or **device_type** clause must appear.
- 2738 • Two instances of the same clause may not appear on the same directive.

2739 **Errors**

- 2740 • An **acc_error_device_type_unavailable** error is issued if a **device_type** clause
 2741 appears, and no device of that device type is available.
- 2742 • An **acc_error_device_unavailable** error is issued if a **device_num** clause ap-
 2743 pears, and the *int-expr* is not a valid device number.
- 2744 • An **acc_error_invalid_async** error is issued if a **default_async** clause appears,
 2745 and the *int-expr* is not a valid *async-argument*.

2746 See Section 5.2.2.

2747 2.14.4 Update Directive

2748 Summary

2749 The **update** directive is used during the lifetime of accelerator data to update *vars* in local memory
2750 with values from the corresponding data in device memory, or to update *vars* in device memory with
2751 values from the corresponding data in local memory.

2752 Syntax

2753 In C and C++, the syntax of the **update** directive is:

```
2754 #pragma acc update clause-list new-line
```

2755 In Fortran the syntax of the **update** data directive is:

```
2756 !$acc update clause-list
```

2757 where *clause* is one of the following:

```
2758 async [ ( int-expr ) ]
2759 wait [ ( wait-argument ) ]
2760 device_type ( device-type-list )
2761 if ( condition )
2762 if_present
2763 self ( var-list )
2764 host ( var-list )
2765 device ( var-list )
```

2766 Multiple subarrays of the same array may appear in a *var-list* of the same or different clauses on
2767 the same directive. The effect of an **update** clause is to copy data from device memory to local
2768 memory for **update self**, and from local memory to device memory for **update device**. The
2769 updates are done in the order in which they appear on the directive.

2770 Restrictions

- 2771 • At least one **self**, **host**, or **device** clause must appear on an **update** directive.

2772 self clause

2773 The **self** clause specifies that the *vars* in *var-list* are to be copied from the current device memory
2774 to local memory for data not in shared memory. For data in shared memory, no action is taken. An
2775 **update** directive with the **self** clause is equivalent to a call to the **acc_update_self** routine,
2776 described in Section 3.2.20.

2777 host clause

2778 The **host** clause is a synonym for the **self** clause.

2779 device clause

2780 The **device** clause specifies that the *vars* in *var-list* are to be copied from local memory to the cur-
2781 rent device memory, for data not in shared memory. For data in shared memory, no action is taken.
2782 An **update** directive with the **device** clause is equivalent to a call to the **acc_update_device**
2783 routine, described in Section 3.2.20.

2784 **if clause**

2785 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
2786 perform the updates unconditionally. When an **if** clause appears, the implementation will generate
2787 code to conditionally perform the updates only when the *condition* evaluates to *true*.

2788 **async clause**

2789 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2790 **wait clause**

2791 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2792 **if_present clause**

2793 When an **if_present** clause appears on the directive, no action is taken for a *var* which appears
2794 in *var-list* that is not present in the current device memory.

2795 **Restrictions**

- 2796 • The **update** directive is executable. It must not appear in place of the statement following
2797 an *if*, *while*, *do*, *switch*, or *label* in C or C++, or in place of the statement following a logical
2798 *if* in Fortran.
- 2799 • If no **if_present** clause appears on the directive, each *var* in *var-list* must be present in
2800 the current device memory.
- 2801 • Only the **async** and **wait** clauses may follow a **device_type** clause.
- 2802 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
2803 value; in C or C++, the condition must evaluate to a scalar integer value.
- 2804 • Noncontiguous subarrays may appear. It is implementation-specific whether noncontiguous
2805 regions are updated by using one transfer for each contiguous subregion, or whether the non-
2806 contiguous data is packed, transferred once, and unpacked, or whether one or more larger
2807 subarrays (no larger than the smallest contiguous region that contains the specified subarray)
2808 are updated.
- 2809 • In C and C++, a member of a struct or class may appear, including a subarray of a member.
2810 Members of a subarray of struct or class type may not appear.
- 2811 • In C and C++, if a subarray notation is used for a struct member, subarray notation may not
2812 be used for any parent of that struct member.
- 2813 • In Fortran, members of variables of derived type may appear, including a subarray of a mem-
2814 ber. Members of subarrays of derived type may not appear.
- 2815 • In Fortran, if array or subarray notation is used for a derived type member, array or subarray
2816 notation may not be used for a parent of that derived type member.
- 2817 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **self**,
2818 **host**, and **device** clauses.

2819 **Errors**

- 2820 • An **acc_error_not_present** error is issued if no **if_present** clause appears and
2821 any *var* in a **device** or **self** clause is not present on the current device.
- 2822 • An **acc_error_partly_present** error is issued if part of *var* is present in the current
2823 device memory but all of *var* is not.
- 2824 • An **async** or **wait** clause can cause an error to be issued; see Sections 2.16.1 and 2.16.2.
2825 See Section 5.2.2.

2826 **2.14.5 Wait Directive**

2827 See Section 2.16 Asynchronous Behavior for more information.

2828 **2.14.6 Enter Data Directive**

2829 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

2830 **2.14.7 Exit Data Directive**

2831 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

2832 **2.15 Procedure Calls in Compute Regions**

2833 This section describes how routines are compiled for an accelerator and how procedure calls are
2834 compiled in compute regions. See Section 2.17.1 Optional Arguments for discussion of Fortran
2835 optional arguments in procedure calls inside compute regions.

2836 **2.15.1 Routine Directive**2837 **Summary**

2838 The **routine** directive is used to tell the compiler to compile a given procedure or a C++ *lambda*
2839 for an accelerator as well as for the host. In a file or routine with a procedure call, the **routine**
2840 directive tells the implementation the attributes of the procedure when called on the accelerator.

2841 **Syntax**

2842 In C and C++, the syntax of the **routine** directive is:

```
2843 #pragma acc routine clause-list new-line
2844 #pragma acc routine( name ) clause-list new-line
```

2845 In C and C++, the **routine** directive without a name may appear immediately before a function
2846 definition, a C++ *lambda*, or just before a function prototype and applies to that immediately fol-
2847 lowing function or prototype. The **routine** directive with a name may appear anywhere that a
2848 function prototype is allowed and applies to the function or the C++ *lambda* in that scope with that
2849 name, but must appear before any definition or use of that function.

2850 In Fortran the syntax of the **routine** directive is:

```
2851 !$acc routine clause-list
2852 !$acc routine( name ) clause-list
```


2853 In Fortran, the **routine** directive without a name may appear within the specification part of a
 2854 subroutine or function definition, or within an interface body for a subroutine or function in an
 2855 interface block, and applies to the containing subroutine or function. The **routine** directive with
 2856 a name may appear in the specification part of a subroutine, function or module, and applies to the
 2857 named subroutine or function.

2858 A C or C++ function or Fortran subprogram compiled with the **routine** directive for an accelera-
 2859 tor is called an *accelerator routine*.

2860 If an *accelerator routine* is a C++ *lambda*, the associated function will be compiled for both the
 2861 accelerator and the host.

2862 If a *lambda* is called in a compute region and it is not an *accelerator routine*, then the *lambda* is
 2863 treated as if its name appears in the name list of a **routine** directive with **seq** clause. If *lambda*
 2864 is defined in an *accelerator routine* that has a **nohost** clause then the *lambda* is treated as if its
 2865 name appears in the name list of a **routine** directive with a **nohost** clause.

2866 The *clause* is one of the following:

```
2867     gang
2868     worker
2869     vector
2870     seq
2871     bind( name )
2872     bind( string )
2873     device_type( device-type-list )
2874     nohost
```

2875 A **gang**, **worker**, **vector**, or **seq** clause specifies the *level of parallelism* in the routine.

2876 **gang clause**

2877 The **gang** clause specifies that the procedure contains, may contain, or may call another procedure
 2878 that contains a loop with a **gang** clause. A call to this procedure must appear in code that is
 2879 executed in *gang-redundant* mode, and all gangs must execute the call. For instance, a procedure
 2880 with a **routine gang** directive may not be called from within a loop that has a **gang** clause.
 2881 Only one of the **gang**, **worker**, **vector** and **seq** clauses may appear for each device type.

2882 **worker clause**

2883 The **worker** clause specifies that the procedure contains, may contain, or may call another pro-
 2884 cedure that contains a loop with a **worker** clause, but does not contain nor does it call another
 2885 procedure that contains a loop with the **gang** clause. A loop in this procedure with an **auto** clause
 2886 may be selected by the compiler to execute in **worker** or **vector** mode. A call to this procedure
 2887 must appear in code that is executed in *worker-single* mode, though it may be in *gang-redundant*
 2888 or *gang-partitioned* mode. For instance, a procedure with a **routine worker** directive may be
 2889 called from within a loop that has the **gang** clause, but not from within a loop that has the **worker**
 2890 clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may appear for each device
 2891 type.

2892 **vector clause**

2893 The **vector** clause specifies that the procedure contains, may contain, or may call another pro-
2894 cedure that contains a loop with the **vector** clause, but does not contain nor does it call another
2895 procedure that contains a loop with either a **gang** or **worker** clause. A loop in this procedure with
2896 an **auto** clause may be selected by the compiler to execute in **vector** mode, but not **worker**
2897 mode. A call to this procedure must appear in code that is executed in *vector-single* mode, though
2898 it may be in *gang-redundant* or *gang-partitioned* mode, and in *worker-single* or *worker-partitioned*
2899 mode. For instance, a procedure with a **routine vector** directive may be called from within
2900 a loop that has the **gang** clause or the **worker** clause, but not from within a loop that has the
2901 **vector** clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may appear for each
2902 device type.

2903 **seq clause**

2904 The **seq** clause specifies that the procedure does not contain nor does it call another procedure that
2905 contains a loop with a **gang**, **worker**, or **vector** clause. A loop in this procedure with an **auto**
2906 clause will be executed in **seq** mode. A call to this procedure may appear in any mode. Only one
2907 of the **gang**, **worker**, **vector** and **seq** clauses may appear for each device type.

2908 **bind clause**

2909 The **bind** clause specifies the name to use when calling the procedure on a device other than the
2910 host. If the name is specified as an identifier, it is called as if that name were specified in the
2911 language being compiled. If the name is specified as a string, the string is used for the procedure
2912 name unmodified. A **bind** clause on a procedure definition behaves as if it had appeared on a
2913 declaration by changing the name used to call the function on a device other than the host; however,
2914 the procedure is not compiled for the device with either the original name or the name in the **bind**
2915 clause.

2916 If there is both a Fortran **bind** and an acc **bind** clause for a procedure definition then a call on the
2917 host will call the Fortran bound name and a call on another device will call the name in the **bind**
2918 clause.

2919 **device_type clause**

2920 The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

2921 **nohost clause**

2922 The **nohost** tells the compiler not to compile a version of this procedure for the host. All calls
2923 to this procedure must appear within compute regions. If this procedure is called from other pro-
2924 cedures, those other procedures must also have a matching **routine** directive with the **nohost**
2925 clause.

2926 **Restrictions**

- 2927 • Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device_type**
2928 clause.
- 2929 • At least one of the (**gang**, **worker**, **vector**, or **seq**) clauses must appear on the construct.
2930 If the **device_type** clause appears on the **routine** directive, a default level of parallelism

2931 clause must appear before the **device_type** clause, or a level of parallelism clause must
 2932 appear following each **device_type** clause on the directive.

- 2933 • In C and C++, function static variables are not supported in functions to which a **routine**
 2934 directive applies.
- 2935 • In Fortran, variables with the *save* attribute, either explicitly or implicitly, are not supported
 2936 in subprograms to which a **routine** directive applies.
- 2937 • A **bind** clause may not bind to a routine name that has a visible **bind** clause.
- 2938 • If a function or subroutine has a **bind** clause on both the declaration and the definition then
 2939 they both must bind to the same name.

2940 2.15.2 Global Data Access

2941 C or C++ global, file static, or *extern* variables or array, and Fortran *module* or *common block* vari-
 2942 ables or arrays, that are used in accelerator routines must appear in a declare directive in a **create**,
 2943 **copyin**, **device_resident** or **link** clause. If the data appears in a **device_resident**
 2944 clause, the **routine** directive for the procedure must include the **nohost** clause. If the data ap-
 2945 pears in a **link** clause, that data must have an active accelerator data lifetime by virtue of appearing
 2946 in a data clause for a **data** construct, compute construct, or **enter data** directive.

2947 2.16 Asynchronous Behavior

2948 This section describes the **async** clause, the **wait** clause, the **wait** directive, and the behavior of
 2949 programs that use asynchronous data movement, compute regions, and asynchronous API routines.

2950 In this section and throughout the specification, the term *async-argument* means a nonnegative
 2951 scalar integer expression (*int* for C or C++, *integer* for Fortran), or one of the special values
 2952 **acc_async_noval** or **acc_async_sync**, as defined in the C header file and the Fortran
 2953 **openacc** module. The special values are negative values, so as not to conflict with a user-specified
 2954 nonnegative *async-argument*. An *async-argument* is used in **async** clauses, **wait** clauses, **wait**
 2955 directives, and as an argument to various runtime routines.

2956 The *async-value* of an *async-argument* is

- 2957 • **acc_async_sync** if *async-argument* has a value equal to the special value **acc_async_sync**,
- 2958 • the value of *acc-default-async-var* if *async-argument* has a value equal to the special value
 2959 **acc_async_noval**,
- 2960 • the value of the *async-argument*, if it is nonnegative,
- 2961 • implementation-defined, otherwise.

2962 The *async-value* is used to select the activity queue to which the clause or directive or API routine
 2963 refers. The properties of the current device and the implementation will determine how many actual
 2964 activity queues are supported, and how the *async-value* is mapped onto the actual activity queues.
 2965 Two asynchronous operations on the same device with the same *async-value* will be enqueued
 2966 onto the same activity queue, and therefore will be executed on the device in the order they are
 2967 encountered by the local thread. Two asynchronous operations with different *async-values* may be
 2968 enqueued onto different activity queues, and therefore may be executed on the device in either order
 2969 or concurrently relative to each other. If there are two or more host threads executing and sharing the

2970 same device, asynchronous operations on any thread with the same *async-value* will be enqueued
 2971 onto the same activity queue. If the threads are not synchronized with respect to each other, the
 2972 operations may be enqueued in either order and therefore may execute on the device in either order.
 2973 Asynchronous operations enqueued to different devices may execute in any order or may execute
 2974 concurrently, regardless of the *async-value* used for each.

2975 If a compute construct, data directive, or runtime API call has an *async-value* of **acc_async_sync**,
 2976 the associated operations are executed on the activity queue associated with the *async-value*
 2977 **acc_async_sync**, and the local thread will wait until the associated operations have completed
 2978 before executing the code following the construct or directive. If a **data** construct has an *async-*
 2979 *value* of **acc_async_sync**, the associated operations are executed on the activity queue associ-
 2980 ated with the *async-value* **acc_async_sync**, and the local thread will wait until the associated
 2981 operations that occur upon entry of the construct have completed before executing the code of the
 2982 construct's structured block or block construct, and after that, will wait until the associated opera-
 2983 tions that occur upon exit of the construct have completed before executing the code following the
 2984 construct.

2985 If a compute construct, data directive, or runtime API call has an *async-value* other than
 2986 **acc_async_sync**, the associated operations are executed on the activity queue associated with
 2987 that *async-value* and the associated operations may be processed asynchronously while the local
 2988 thread continues executing the code following the construct or directive. If a **data** construct has an
 2989 *async-value* other than **acc_async_sync**, the associated operations are executed on the activity
 2990 queue associated with that *async-value*, and the associated operations that occur upon entry of the
 2991 construct may be processed asynchronously while the local thread continues executing the code
 2992 of the construct's structured block or block construct, and after that, the associated operations that
 2993 occur upon exit of the construct may be processed asynchronously while the local thread continues
 2994 executing the code following the construct.

2995 In this section and throughout the specification, the term *wait-argument*, means:

2996 [**devnum** : *int-expr* :] [**queues** :] *async-argument-list*

2997 If a **devnum** modifier appears in the *wait-argument* then the associated device is the device with
 2998 that device number of the current device type. If no **devnum** modifier appears then the associated
 2999 device is the current device.

3000 Each *async-argument* is associated with an *async-value*. The *async-values* select the associated
 3001 activity queue or queues on the associated device. If there is no *async-argument-list*, the associated
 3002 activity queues are all activity queues for the associated device.

3003 The **queues** modifier within a *wait-argument* is optional to improve clarity of the expression list.

3004 2.16.1 **async clause**

3005 The **async** clause may appear on a **parallel**, **serial**, **kernels**, or **data** construct, or an
 3006 **enter data**, **exit data**, **update**, or **wait** directive. In all cases, the **async** clause is optional.
 3007 The **async** clause may have a single *async-argument*, as defined above. If the **async** clause does
 3008 not appear, the behavior is as if the *async-argument* is **acc_async_sync**. If the **async** clause
 3009 appears with no argument, the behavior is as if the *async-argument* is **acc_async_noval**. The
 3010 *async-value* for a construct or directive is defined in Section 2.16.

3011 **Errors**

- 3012 • An **acc_error_invalid_async** error is issued if an **async** clause with an argument
- 3013 appears on any directive and the argument is not a valid *async-argument*.

3014 See Section 5.2.2.

3015 **2.16.2 wait clause**

3016 The **wait** clause may appear on a **parallel**, **serial**, or **kernels**, or **data** construct, or
 3017 an **enter data**, **exit data**, or **update** directive. In all cases, the **wait** clause is optional.
 3018 When there is no **wait** clause, the associated operations may be enqueued or launched or executed
 3019 immediately on the device.

3020 If there is an argument to the **wait** clause, it must be a *wait-argument*, the associated device and
 3021 activity queues are as specified in the *wait-argument*; see Section 2.16. If there is no argument to
 3022 the **wait** clause, the associated device is the current device and associated activity queues are all
 3023 activity queues. The associated operations may not be launched or executed until all operations
 3024 already enqueued up to this point by this thread on the associated asynchronous device activity
 3025 queues have completed. **Note:** One legal implementation is for the local thread to wait until the
 3026 operations already enqueued on the associated asynchronous device activity queues have completed;
 3027 another legal implementation is for the local thread to enqueue the associated operations in such a
 3028 way that they will not start until the operations already enqueued on the associated asynchronous
 3029 device activity queues have completed.

3030 **Errors**

- 3031 • An **acc_error_device_unavailable** error is issued if a **wait** clause appears on any
- 3032 directive with a **devnum** modifier and the associated *int-expr* is not a valid device number.
- 3033 • An **acc_error_invalid_async** error is issued if a **wait** clause appears on any direc-
- 3034 tive with a **queues** modifier or no modifier and any value in the associated list is not a valid
- 3035 *async-argument*.

3036 See Section 5.2.2.

3037 **2.16.3 Wait Directive**3038 **Summary**

3039 The **wait** directive causes the local thread or operations enqueued onto a device activity queue on
 3040 the current device to wait for completion of asynchronous operations.

3041 **Syntax**

3042 In C and C++, the syntax of the **wait** directive is:

```
3043 #pragma acc wait [ ( wait-argument ) ] [ clause-list ] new-line
```

3044 In Fortran the syntax of the **wait** directive is:

```
3045 !$acc wait [ ( wait-argument ) ] [ clause-list ]
```

3046 where *clause* is:

```
3047 async [ ( async-argument ) ]
```

```
3048 if ( condition )
```

3049 If it appears, the *wait-argument* is as defined in Section 2.16, and the associated device and activity
 3050 queues are as specified in the *wait-argument*. If there is no *wait-argument* clause, the associated
 3051 device is the current device and associated activity queues are all activity queues.

3052 If there is no **async** clause, the local thread will wait until all operations enqueued by this thread
 3053 onto each of the associated device activity queues for the associated device have completed. There
 3054 is no guarantee that all the asynchronous operations initiated by other threads onto those queues will
 3055 have completed without additional synchronization with those threads.

3056 If there is an **async** clause, no new operation may be launched or executed on the activity queue
 3057 associated with the *async-argument* on the current device until all operations enqueued up to this
 3058 point by this thread on the activity queues associated with the *wait-argument* have completed. **Note:**
 3059 One legal implementation is for the local thread to wait for all the associated activity queues; another
 3060 legal implementation is for the thread to enqueue a synchronization operation in such a way that
 3061 no new operation will start until the operations enqueued on the associated activity queues have
 3062 completed.

3063 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
 3064 perform the wait operation unconditionally. When an **if** clause appears, the implementation will
 3065 generate code to conditionally perform the wait operation only when the *condition* evaluates to *true*.

3066 A **wait** directive is functionally equivalent to a call to one of the **acc_wait**, **acc_wait_async**,
 3067 **acc_wait_all**, or **acc_wait_all_async** runtime API routines, as described in Sections 3.2.10
 3068 and 3.2.11.

3069 Errors

- 3070 • An **acc_error_device_unavailable** error is issued if a **devnum** modifier appears
 3071 and the *int-expr* is not a valid device number.
- 3072 • An **acc_error_invalid_async** error is issued if a **queues** modifier or no modifier
 3073 appears and any value in the associated list is not a valid *async-argument*.

3074 See Section 5.2.2.

3075 2.17 Fortran Specific Behavior

3076 2.17.1 Optional Arguments

3077 This section refers to the Fortran intrinsic function **PRESENT**. A call to the Fortran intrinsic function
 3078 **PRESENT(arg)** returns **.true.**, if **arg** is an optional dummy argument and an actual argument
 3079 for **arg** was present in the argument list of the call site. This should not be confused with the
 3080 OpenACC **present** data clause.

3081 The appearance of a Fortran optional argument **arg** as a *var* in any of the following clauses has no
 3082 effect at runtime if **PRESENT(arg)** is **.false.**:

- 3083 • in data clauses on compute and **data** constructs;
- 3084 • in data clauses on **enter data** and **exit data** directives;
- 3085 • in data and **device_resident** clauses on **declare** directives;
- 3086 • in **use_device** clauses on **host_data** directives;
- 3087 • in **self**, **host**, and **device** clauses on **update** directives.

3088 The appearance of a Fortran optional argument **arg** in the following situations may result in unde-
3089 fined behavior if **PRESENT (arg)** is **.false.** when the associated construct is executed:

- 3090 • as a *var* in **private**, **firstprivate**, and **reduction** clauses;
- 3091 • as a *var* in **cache** directives;
- 3092 • as part of an expression in any clause or directive.

3093 A call to the Fortran intrinsic function **PRESENT** behaves the same way in a compute construct or
3094 an accelerator routine as on the host. The function call **PRESENT (arg)** must return the same value
3095 in a compute construct as **PRESENT (arg)** would outside of the compute construct. If a Fortran
3096 optional argument **arg** appears as an actual argument in a procedure call in a compute construct
3097 or an accelerator routine, and the associated dummy argument **subarg** also has the **optional**
3098 attribute, then **PRESENT (subarg)** returns the same value as **PRESENT (subarg)** would when
3099 executed on the host.

3100 2.17.2 Do Concurrent Construct

3101 This section refers to the Fortran **do concurrent** construct that is a form of **do** construct. When
3102 **do concurrent** appears without a **loop** construct in a **kernels** construct it is treated as if it is
3103 annotated with **loop auto**. If it appears in a **parallel** construct or an accelerator routine then
3104 it is treated as if it is annotated with **loop independent**.

3. Runtime Library

3105

3106 This chapter describes the OpenACC runtime library routines that are available for use by program-
3107 mers. Use of these routines may limit portability to systems that do not support the OpenACC API.
3108 Conditional compilation using the `_OPENACC` preprocessor variable may preserve portability.

3109 This chapter has two sections:

- 3110 • Runtime library definitions
- 3111 • Runtime library routines

3112 There are four categories of runtime routines:

- 3113 • Device management routines, to get the number of devices, set the current device, and so on.
- 3114 • Asynchronous queue management, to synchronize until all activities on an async queue are
3115 complete, for instance.
- 3116 • Device test routine, to test whether this statement is executing on the device or not.
- 3117 • Data and memory management, to manage memory allocation or copy data between memo-
3118 ries.

3.1 Runtime Library Definitions

3119

3120 In C and C++, prototypes for the runtime library routines described in this chapter are provided in
3121 a header file named `openacc.h`. All the library routines are *extern* functions with “C” linkage.
3122 This file defines:

- 3123 • The prototypes of all routines in the chapter.
- 3124 • Any datatypes used in those prototypes, including an enumeration type to describe the sup-
3125 ported device types.
- 3126 • The values of `acc_async_noval`, `acc_async_sync`, and `acc_async_default`.

3127 In Fortran, interface declarations are provided in a Fortran module named `openacc`. The `openacc`
3128 module defines:

- 3129 • The integer parameter `openacc_version` with a value `yyyymm` where `yyyy` and `mm` are the
3130 year and month designations of the version of the Accelerator programming model supported.
3131 This value matches the value of the preprocessor variable `_OPENACC`.
- 3132 • Interfaces for all routines in the chapter.
- 3133 • Integer parameters to define integer kinds for arguments to and return values for those rou-
3134 tines.
- 3135 • Integer parameters to describe the supported device types.
- 3136 • Integer parameters to define the values of `acc_async_noval`, `acc_async_sync`, and
3137 `acc_async_default`.

3138 Many of the routines accept or return a value corresponding to the type of device. In C and C++, the
 3139 datatype used for device type values is `acc_device_t`; in Fortran, the corresponding datatype
 3140 is `integer(kind=acc_device_kind)`. The possible values for device type are implemen-
 3141 tation specific, and are defined in the C or C++ include file `openacc.h` and the Fortran module
 3142 `openacc`. Five values are always supported: `acc_device_none`, `acc_device_default`,
 3143 `acc_device_host`, `acc_device_not_host`, and `acc_device_current`. For other val-
 3144 ues, look at the appropriate files included with the implementation, or read the documentation for
 3145 the implementation. The value `acc_device_default` will never be returned by any function;
 3146 its use as an argument will tell the runtime library to use the default device type for that implemen-
 3147 tation.

3148 3.2 Runtime Library Routines

3149 In this section, for the C and C++ prototypes, pointers are typed `h_void*` or `d_void*` to design-
 3150 ate a host memory address or device memory address, when these calls are executed on the host,
 3151 as if the following definitions were included:

```
3152     #define h_void void
3153     #define d_void void
```

3154 Restrictions

3155 Except for `acc_on_device`, these routines are only available on the host.

3156 3.2.1 acc_get_num_devices

3157 Summary

3158 The `acc_get_num_devices` routine returns the number of available devices of the given type.

3159 Format

3160 C or C++:

```
3161     int acc_get_num_devices(acc_device_t dev_type);
```

3162 Fortran:

```
3163     integer function acc_get_num_devices(dev_type)
3164     integer(acc_device_kind) :: dev_type
```

3165 Description

3166 The `acc_get_num_devices` routine returns the number of available devices of device type
 3167 `dev_type`. If device type `dev_type` is not supported or no device of `dev_type` is available,
 3168 this routine returns zero.

3169 3.2.2 acc_set_device_type

3170 Summary

3171 The `acc_set_device_type` routine tells the runtime which type of device to use when exe-
 3172 cuting a compute region and sets the value of `acc-current-device-type-var`. This is useful when the
 3173 implementation allows the program to be compiled to use more than one type of device.

3174 **Format**

3175 C or C++:

3176 `void acc_set_device_type(acc_device_t dev_type);`

3177 Fortran:

3178 `subroutine acc_set_device_type(dev_type)`3179 `integer(acc_device_kind) :: dev_type`3180 **Description**

3181 A call to `acc_set_device_type` is functionally equivalent to a `set device_type(dev_type)`
 3182 directive, as described in Section 2.14.3. This routine tells the runtime which type of device to use
 3183 among those available and sets the value of *acc-current-device-type-var* for the current thread to
 3184 `dev_type`.

3185 **Restrictions**

- 3186 • If some compute regions are compiled to only use one device type, the result of calling this
 3187 routine with a different device type may produce undefined behavior.

3188 **Errors**

- 3189 • An `acc_error_device_type_unavailable` error is issued if device type `dev_type`
 3190 is not supported or no device of `dev_type` is available.

3191 See Section 5.2.2.

3192 **3.2.3 acc_get_device_type**3193 **Summary**

3194 The `acc_get_device_type` routine returns the value of *acc-current-device-type-var*, which is
 3195 the device type of the current device. This is useful when the implementation allows the program to
 3196 be compiled to use more than one type of device.

3197 **Format**

3198 C or C++:

3199 `acc_device_t acc_get_device_type(void);`

3200 Fortran:

3201 `function acc_get_device_type()`3202 `integer(acc_device_kind) :: acc_get_device_type`3203 **Description**

3204 The `acc_get_device_type` routine returns the value of *acc-current-device-type-var* for the
 3205 current thread to tell the program what type of device will be used to run the next compute region, if
 3206 one has been selected. The device type may have been selected by the program with a runtime API
 3207 call or a directive, by an environment variable, or by the default behavior of the implementation; see
 3208 the table in Section 2.3.1.

3209 **Restrictions**

- 3210 • If the device type has not yet been selected, the value `acc_device_none` may be returned.

3211 3.2.4 `acc_set_device_num`

3212 Summary

3213 The `acc_set_device_num` routine tells the runtime which device to use and sets the value of
3214 `acc-current-device-num-var`.

3215 Format

3216 C or C++:

```
3217     void acc_set_device_num(int dev_num, acc_device_t dev_type);
```

3218 Fortran:

```
3219     subroutine acc_set_device_num(dev_num, dev_type)
3220         integer :: dev_num
3221         integer(acc_device_kind) :: dev_type
```

3222 Description

3223 A call to `acc_set_device_num` is functionally equivalent to a `set device_type (dev_type)`
3224 `device_num (dev_num)` directive, as described in Section 2.14.3. This routine tells the runtime
3225 which device to use among those available of the given type for compute or data regions in the cur-
3226 rent thread and sets the value of `acc-current-device-num-var` to `dev_num`. If the value of `dev_num`
3227 is negative, the runtime will revert to its default behavior, which is implementation-defined. If the
3228 value of the `dev_type` is zero, the selected device number will be used for all device types. Calling
3229 `acc_set_device_num` implies a call to `acc_set_device_type (dev_type)`.

3230 Errors

- 3231 • An `acc_error_device_type_unavailable` error is issued if device type `dev_type`
3232 is not supported or no device of `dev_type` is available.
- 3233 • An `acc_error_device_unavailable` error is issued if the value of `dev_num` is not
3234 a valid device number.

3235 See Section 5.2.2.

3236 3.2.5 `acc_get_device_num`

3237 Summary

3238 The `acc_get_device_num` routine returns the value of `acc-current-device-num-var` for the cur-
3239 rent thread.

3240 Format

3241 C or C++:

```
3242     int acc_get_device_num(acc_device_t dev_type);
```

3243 Fortran:

```
3244     integer function acc_get_device_num(dev_type)
3245         integer(acc_device_kind) :: dev_type
```

3246 Description

3247 The `acc_get_device_num` routine returns the value of `acc-current-device-num-var` for the cur-
3248 rent thread. If there are no devices of device type `dev_type` or if device type `dev_type` is not
3249 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)
    use iso_c_binding, only: c_size_t
    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.

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

An implementation may support additional properties for some devices.

3275 **Restrictions**

- 3276 • **acc_get_property** will return 0 and **acc_get_property_string** will return a null
 3277 pointer (in C or C++) or a blank string (in Fortran) in the following cases:
- 3278 – If device type **dev_type** is not supported or no device of **dev_type** is available.
 - 3279 – If the value of **dev_num** is not a valid device number for device type **dev_type**.
 - 3280 – If the value of **property** is not one of the known values for that query routine, or that
 3281 property has no value for the specified device.

3282 **3.2.7 acc_init**3283 **Summary**

3284 The **acc_init** and **acc_init_device** routines initialize the runtime for the specified device
 3285 type and device number. This can be used to isolate any initialization cost from the computational
 3286 cost, such as when collecting performance statistics.

3287 **Format**

3288 C or C++:

```
3289 void acc_init(acc_device_t dev_type);
3290 void acc_init_device(int dev_num, acc_device_t dev_type);
```

3291 Fortran:

```
3292 subroutine acc_init(dev_type)
3293 subroutine acc_init_device(dev_num, dev_type)
3294 integer :: dev_num
3295 integer(acc_device_kind) :: dev_type
```

3296 **Description**

3297 A call to **acc_init** or **acc_init_device** is functionally equivalent to an **init** directive with
 3298 matching **dev_type** and **dev_num** arguments, as described in Section 2.14.1. **dev_type** must
 3299 be one of the defined accelerator types. **dev_num** must be a valid device number of the device type
 3300 **dev_type**. These routines also implicitly call **acc_set_device_type(dev_type)**. In the
 3301 case of **acc_init_device**, **acc_set_device_num(dev_num)** is also called.

3302 If a program initializes one or more devices without an intervening **shutdown** directive or
 3303 **acc_shutdown** call to shut down those same devices, no action is taken.

3304 **Errors**

- 3305 • An **acc_error_device_type_unavailable** error is issued if device type **dev_type**
 3306 is not supported or no device of **dev_type** is available.
- 3307 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3308 number.

3309 See Section 5.2.2.

3310 **3.2.8 acc_shutdown**

3311 **Summary**

3312 The **acc_shutdown** and **acc_shutdown_device** routines shut down the connection to spec-
 3313 ified devices and free up any related resources in the runtime. This ends all data lifetimes in device
 3314 memory for the device or devices that are shut down, which effectively sets structured and dynamic
 3315 reference counters to zero.

3316 **Format**

3317 C or C++:

```
3318     void acc_shutdown(acc_device_t dev_type);
3319     void acc_shutdown_device(int dev_num, acc_device_t dev_type);
```

3320 Fortran:

```
3321     subroutine acc_shutdown(dev_type)
3322     subroutine acc_shutdown_device(dev_num, dev_type)
3323         integer :: dev_num
3324         integer(acc_device_kind) :: dev_type
```

3325 **Description**

3326 A call to **acc_shutdown** or **acc_shutdown_device** is functionally equivalent to a **shutdown**
 3327 directive, with matching **dev_type** and **dev_num** arguments, as described in Section 2.14.2.
 3328 **dev_type** must be one of the defined accelerator types. **dev_num** must be a valid device number
 3329 of the device type **dev_type**. **acc_shutdown** routine disconnects the program from all devices
 3330 of device type **dev_type**. The **acc_shutdown_device** routine disconnects the program from
 3331 **dev_num** of type **dev_type**. Any data that is present in the memory of a device that is shut down
 3332 is immediately deallocated.

3333 **Restrictions**

- 3334 • This routine may not be called while a compute region is executing on a device of type
 3335 **dev_type**.
- 3336 • If the program attempts to execute a compute region on a device or to access any data in the
 3337 memory of a device that was shut down, the behavior is undefined.
- 3338 • If the program attempts to shut down the **acc_device_host** device type, the behavior is
 3339 undefined.

3340 **Errors**

- 3341 • An **acc_error_device_type_unavailable** error is issued if device type **dev_type**
 3342 is not supported or no device of **dev_type** is available.
- 3343 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3344 number.
- 3345 • An **acc_error_device_shutdown** error is issued if there is an error shutting down the
 3346 device.

3347 See Section 5.2.2.

3348 **3.2.9 acc_async_test**3349 **Summary**

3350 The **acc_async_test** routines test for completion of all associated asynchronous operations for
 3351 a single specified async queue or for all async queues on the current device or on a specified device.

3352 **Format**

3353 C or C++:

```

3354     int acc_async_test(int wait_arg);
3355     int acc_async_test_device(int wait_arg, int dev_num);
3356     int acc_async_test_all(void);
3357     int acc_async_test_all_device(int dev_num);

```

3358 Fortran:

```

3359     logical function acc_async_test(wait_arg)
3360     logical function acc_async_test_device(wait_arg, dev_num)
3361     logical function acc_async_test_all()
3362     logical function acc_async_test_all_device(dev_num)
3363     integer(acc_handle_kind) :: wait_arg
3364     integer :: dev_num

```

3365 **Description**

3366 **wait_arg** must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior. **dev_num**
 3367 must be a valid device number of the current device type.

3368 The behavior of the **acc_async_test** routines is:

- 3369 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3370 • If any asynchronous operations initiated by this host thread on device **dev_num** either on
 3371 async queue **wait_arg** (if there is a **wait_arg** argument), or on any async queue (if there
 3372 is no **wait_arg** argument) have not completed, a call to the routine returns *false*.
- 3373 • If all such asynchronous operations have completed, or there are no such asynchronous op-
 3374 erations, a call to the routine returns *true*. A return value of *true* is no guarantee that asyn-
 3375 chronous operations initiated by other host threads have completed.

3376 **Errors**

- 3377 • An **acc_error_invalid_async** error is issued if **wait_arg** is not a valid *async-*
 3378 *argument* value.
- 3379 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3380 number.

3381 See Section 5.2.2.

3382 **3.2.10 acc_wait**3383 **Summary**

3384 The **acc_wait** routines wait for completion of all associated asynchronous operations on a single
 3385 specified async queue or on all async queues on the current device or on a specified device.

3386 **Format**

3387 C or C++:

```

3388     void acc_wait(int wait_arg);
3389     void acc_wait_device(int wait_arg, int dev_num);
3390     void acc_wait_all(void);
3391     void acc_wait_all_device(int dev_num);

```


3392 Fortran:

```
3393     subroutine acc_wait(wait_arg)
3394     subroutine acc_wait_device(wait_arg, dev_num)
3395     subroutine acc_wait_all()
3396     subroutine acc_wait_all_device(dev_num)
3397     integer(acc_handle_kind) :: wait_arg
3398     integer :: dev_num
```

3399 Description

3400 A call to an **acc_wait** routine is functionally equivalent to a **wait** directive as follows, see Section 2.16.3:

- 3402 • **acc_wait** to a **wait(wait_arg)** directive.
- 3403 • **acc_wait_device** to a **wait(devnum:dev_num, queues:wait_arg)** directive.
- 3404 • **acc_wait_all** to a **wait** directive with no *wait-argument*.
- 3405 • **acc_wait_all_device** to a **wait(devnum:dev_num)** directive.

3406 **wait_arg** must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior. **dev_num**
3407 must be a valid device number of the current device type.

3408 The behavior of the **acc_wait** routines is:

- 3409 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3410 • The routine will not return until all asynchronous operations initiated by this host thread on
3411 device **dev_num** either on async queue **wait_arg** (if there is a **wait_arg** argument) or
3412 on all async queues (if there is no **wait_arg** argument) have completed.
- 3413 • If two or more threads share the same accelerator, there is no guarantee that matching asyn-
3414 chronous operations initiated by other threads have completed.

3415 For compatibility with OpenACC version 1.0, **acc_wait** may also be spelled **acc_async_wait**,
3416 and **acc_wait_all** may also be spelled **acc_async_wait_all**.

3417 Errors

- 3418 • An **acc_error_invalid_async** error is issued if **wait_arg** is not a valid *async-*
3419 *argument* value.
- 3420 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
3421 number.

3422 See Section 5.2.2.

3423 3.2.11 acc_wait_async

3424 Summary

3425 The **acc_wait_async** routines enqueue a wait operation on one async queue of the current
3426 device or a specified device for the operations previously enqueued on a single specified async
3427 queue or on all other async queues.

3428 **Format**

C or C++:

```

3429     void acc_wait_async(int wait_arg, int async_arg);
3430     void acc_wait_device_async(int wait_arg, int async_arg,
3431                               int dev_num);
3432     void acc_wait_all_async(int async_arg);
3433     void acc_wait_all_device_async(int async_arg, int dev_num);

```

Fortran:

```

3434     subroutine acc_wait_async(wait_arg, async_arg)
3435     subroutine acc_wait_device_async(wait_arg, async_arg, dev_num)
3436     subroutine acc_wait_all_async(async_arg)
3437     subroutine acc_wait_all_device_async(async_arg, dev_num)
3438     integer(acc_handle_kind) :: wait_arg, async_arg
3439     integer :: dev_num

```

3439 **Description**

3440 A call to an **acc_wait_async** routine is functionally equivalent to a **wait async (async_arg)**
 3441 directive as follows, see Section 2.16.3:

- 3442 • A call to **acc_wait_async** is functionally equivalent to a **wait (wait_arg)**
 3443 **async (async_arg)** directive.
- 3444 • A call to **acc_wait_device_async** is functionally equivalent to a **wait (devnum:**
 3445 **dev_num, queues:wait_arg) async (async_arg)** directive.
- 3446 • A call to **acc_wait_all_async** is functionally equivalent to a **wait async (async_arg)**
 3447 directive with no *wait-argument*.
- 3448 • A call to **acc_wait_all_device_async** is functionally equivalent to a
 3449 **wait (devnum:dev_num) async (async_arg)** directive.

3450 **async_arg** and **wait_arg** must be *async-arguments*, as defined in
 3451 Section 2.16 Asynchronous Behavior. **dev_num** must be a valid device number of the current
 3452 device type.

3453 The behavior of the **acc_wait_async** routines is:

- 3454 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3455 • The routine will enqueue a wait operation on the async queue associated with **async_arg**
 3456 for the current device which will wait for operations initiated on the async queue **wait_arg**
 3457 of device **dev_num** (if there is a **wait_arg** argument), or for each async queue of device
 3458 **dev_num** (if there is no **wait_arg** argument).

3459 See Section 2.16 Asynchronous Behavior for more information.

3460 **Errors**

- 3461 • An **acc_error_invalid_async** error is issued if either **async_arg** or **wait_arg** is
 3462 not a valid *async-argument* value.
- 3463 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3464 number.

3465 See Section 5.2.2.

3466 **3.2.12 acc_wait_any**3467 **Summary**

3468 The `acc_wait_any` and `acc_wait_any_device` routines wait for any of the specified asyn-
 3469 chronous queues to complete all pending operations on the current device or the specified device
 3470 number, respectively. Both routines return the queue's index in the provided array of asynchronous
 3471 queues.

3472 **Format**

3473 C or C++:

```
3474     int acc_wait_any(int count, int wait_arg[]);
3475     int acc_wait_any_device(int count, int wait_arg[], int dev_num);
```

3476 Fortran:

```
3477     integer function acc_wait_any(count, wait_arg)
3478     integer function acc_wait_any_device(count, wait_arg, dev_num)
3479     integer :: count, dev_num
3480     integer(acc_handle_kind) :: wait_arg(count)
```

3481 **Description**

3482 `wait_arg` is an array of *async-arguments* as defined in Section 2.16 and `count` is a nonneg-
 3483 ative integer indicating the array length. If there is no `dev_num` argument, it is treated as if
 3484 `dev_num` is the current device number. Otherwise, `dev_num` must be a valid device number
 3485 of the current device type. A call to any of these routines returns an index `i` associated with
 3486 a `wait_arg[i]` that is not `acc_async_sync` and meets the conditions that would evalu-
 3487 ate `acc_async_test_device(wait_arg[i], dev_num)` to *true*. If all the elements in
 3488 `wait_arg` are equal to `acc_async_sync` or `count` is equal to 0, these routines return -1.
 3489 Otherwise, the return value is an integer in the range of $0 \leq i < \text{count}$ in C or C++ and
 3490 $1 \leq i \leq \text{count}$ in Fortran.

3491 **Errors**

- 3492 • An `acc_error_invalid_argument` error is issued if `count` is a negative number.
- 3493 • An `acc_error_invalid_async` error is issued if any element encountered in `wait_arg`
 3494 is not a valid *async-argument* value.
- 3495 • An `acc_error_device_unavailable` error is issued if `dev_num` is not a valid device
 3496 number.

3497 See Section 5.2.2.

3498 **3.2.13 acc_get_default_async**3499 **Summary**

3500 The `acc_get_default_async` routine returns the value of *acc-default-async-var* for the cur-
 3501 rent thread.

3502 **Format**

3503 C or C++:

```
3504     int acc_get_default_async(void);
```

3505 Fortran:

```
3506     function acc_get_default_async()
3507         integer(acc_handle_kind) :: acc_get_default_async
```

3508 Description

3509 The **acc_get_default_async** routine returns the value of *acc-default-async-var* for the cur-
3510 rent thread, which is the asynchronous queue used when an **async** clause appears without an
3511 *async-argument* or with the value **acc_async_noval**.

3512 3.2.14 acc_set_default_async

3513 Summary

3514 The **acc_set_default_async** routine tells the runtime which asynchronous queue to use
3515 when an **async** clause appears with no queue argument.

3516 Format

3517 C or C++:

```
3518     void acc_set_default_async(int async_arg);
```

3519 Fortran:

```
3520     subroutine acc_set_default_async(async_arg)
3521         integer(acc_handle_kind) :: async_arg
```

3522 Description

3523 A call to **acc_set_default_async** is functionally equivalent to a **set default_async(async_arg)**
3524 directive, as described in Section 2.14.3. This **acc_set_default_async** routine tells the
3525 runtime to place any directives with an **async** clause that does not have an *async-argument* or
3526 with the special **acc_async_noval** value into the asynchronous activity queue associated with
3527 **async_arg** instead of the default asynchronous activity queue for that device by setting the value
3528 of *acc-default-async-var* for the current thread. The special argument **acc_async_default** will
3529 reset the default asynchronous activity queue to the initial value, which is implementation-defined.

3530 Errors

- 3531 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
3532 *argument* value.

3533 See Section 5.2.2.

3534 3.2.15 acc_on_device

3535 Summary

3536 The **acc_on_device** routine tells the program whether it is executing on a particular device.

3537 Format

3538 C or C++:

```
3539     int acc_on_device(acc_device_t dev_type);
```

3540 Fortran:

```
3541     logical function acc_on_device(dev_type)
3542         integer(acc_device_kind) :: dev_type
```

3543 Description

3544 The `acc_on_device` routine may be used to execute different paths depending on whether the
3545 code is running on the host or on some accelerator. If the `acc_on_device` routine has a compile-
3546 time constant argument, the call evaluates at compile time to a constant. `dev_type` must be one
3547 of the defined accelerator types.

3548 The behavior of the `acc_on_device` routine is:

- 3549 • If `dev_type` is `acc_device_host`, then outside of a compute region or accelerator rou-
3550 tine, or in a compute region or accelerator routine that is executed on the host CPU, a call to
3551 this routine will evaluate to *true*; otherwise, it will evaluate to *false*.
- 3552 • If `dev_type` is `acc_device_not_host`, the result is the negation of the result with
3553 argument `acc_device_host`.
- 3554 • If `dev_type` is an accelerator device type, then in a compute region or routine that is ex-
3555 ecuted on a device of that type, a call to this routine will evaluate to *true*; otherwise, it will
3556 evaluate to *false*.
- 3557 • The result with argument `acc_device_default` is undefined.

3558 3.2.16 `acc_malloc`

3559 Summary

3560 The `acc_malloc` routine allocates space in the current device memory.

3561 Format

3562 C or C++:

```
3563     d_void* acc_malloc(size_t bytes);
```

3564 Description

3565 The `acc_malloc` routine may be used to allocate space in the current device memory. Pointers
3566 assigned from this routine may be used in `deviceptr` clauses to tell the compiler that the pointer
3567 target is resident on the device. In case of an allocation error or if `bytes` has the value zero,
3568 `acc_malloc` returns a null pointer.

3569 3.2.17 `acc_free`

3570 Summary

3571 The `acc_free` routine frees memory on the current device.

3572 Format

3573 C or C++:

```
3574     void acc_free(d_void* data_dev);
```

3575 Description

3576 The `acc_free` routine will free previously allocated space in the current device memory; `data_dev`
3577 should be a pointer value that was returned by a call to `acc_malloc`. If `data_dev` is a null
3578 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 the current device memory; if not, they allocate space in the current device memory to correspond to the specified local memory, and the `acc_copyin` routines copy the data to that device 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 2.7.7 and 2.7.9, except that no `attach` 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, 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 the current device memory, the routines perform a *present increment* action with the dynamic reference counter. The C/C++ synchronous `acc_copyin` and

3623 **acc_create** routines return a pointer to the existing device memory.

3624 • Otherwise:

3625 – The **acc_copyin** routines perform a *copyin* action with the dynamic reference counter.

3626 – The **acc_create** routines perform a *create* action with the dynamic reference counter.

3627 The C/C++ synchronous **acc_copyin** and **acc_create** routines return a pointer to the
3628 newly allocated device memory.

3629 This data may be accessed using the **present** data clause. Pointers assigned from the C/C++
3630 synchronous **acc_copyin** and **acc_create** routines may be used in **deviceptr** clauses to
3631 tell the compiler that the pointer target is resident on the device.

3632 The synchronous versions will not return until the memory has been allocated and any data transfers
3633 are complete.

3634 The **_async** versions of these routines will perform any data transfers asynchronously on the *async*
3635 queue associated with **async_arg**. The routine may return before the data has been transferred;
3636 see Section 2.16 Asynchronous Behavior for more details. The data will be treated as present in
3637 the current device memory even if the data has not been allocated or transferred before the routine
3638 returns.

3639 For compatibility with OpenACC 2.0, **acc_present_or_copyin** and **acc_pcopyin** are al-
3640 ternate names for **acc_copyin**, and **acc_present_or_create** and **acc_pcreate** are al-
3641 ternate names for **acc_create**.

3642 **Errors**

3643 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
3644 **bytes** is nonzero.

3645 • An **acc_error_partly_present** error is issued if part of the *data section* is already
3646 present in the current device memory but all of the *data section* is not.

3647 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
3648 tion that is not contiguous (in Fortran).

3649 • An **acc_error_out_of_memory** error is issued if the accelerator device does not have
3650 enough memory for the data.

3651 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
3652 *argument* value.

3653 See Section 5.2.2.

3654 **3.2.19 acc_copyout and acc_delete**

3655 **Summary**

3656 The **acc_copyout** and **acc_delete** routines test to see if the argument is in shared memory;
3657 if not, the argument must be present in the current device memory. The **acc_copyout** routines
3658 copy data from device memory to the corresponding local memory, and both **acc_copyout** and
3659 **acc_delete** routines deallocate that space from the device memory.

3660 **Format**

3661 C or C++:

```

3662     void acc_copyout(h_void* data_arg, size_t bytes);
3663     void acc_delete (h_void* data_arg, size_t bytes);
3664
3665     void acc_copyout_finalize(h_void* data_arg, size_t bytes);
3666     void acc_delete_finalize (h_void* data_arg, size_t bytes);
3667
3668     void acc_copyout_async(h_void* data_arg, size_t bytes,
3669                           int async_arg);
3670     void acc_delete_async (h_void* data_arg, size_t bytes,
3671                           int async_arg);
3672
3673     void acc_copyout_finalize_async(h_void* data_arg, size_t bytes,
3674                                    int async_arg);
3675     void acc_delete_finalize_async (h_void* data_arg, size_t bytes,
3676                                    int async_arg);
3677

```

3678 Fortran:

```

3679     subroutine acc_copyout (data_arg [, bytes])
3680     subroutine acc_delete (data_arg [, bytes])
3681
3682     subroutine acc_copyout_finalize (data_arg [, bytes])
3683     subroutine acc_delete_finalize (data_arg [, bytes])
3684
3685     subroutine acc_copyout_async (data_arg [, bytes], async_arg)
3686     subroutine acc_delete_async (data_arg [, bytes], async_arg)
3687
3688     subroutine acc_copyout_finalize_async (data_arg [, bytes], &
3689                                           async_arg)
3690     subroutine acc_delete_finalize_async (data_arg [, bytes], &
3691                                           async_arg)
3692
3693     type(*), dimension(..) :: data_arg
3694     integer :: bytes
3695     integer(acc_handle_kind) :: async_arg

```

3696 **Description**

3697 A call to an **acc_copyout** or **acc_delete** routine is similar to an **exit data** directive
3698 with a **copyout** or **delete** clause, respectively, and a call to an **acc_copyout_finalize**
3699 or **acc_delete_finalize** routine is similar to an **exit data finalize** directive with a
3700 **copyout** or **delete** clause, respectively, as described in Section 2.7.8 and 2.7.11, except that no
3701 *detach* action is performed for a pointer reference. The arguments and the associated *data section*
3702 are as for **acc_copyin**.

3703 The behavior of these routines for the associated *data section* is:

- 3704 • If the *data section* is in shared memory, no action is taken.

- 3705 • If the dynamic reference counter for the *data section* is zero, no action is taken.
- 3706 • Otherwise, the dynamic reference counter is updated:
 - 3707 – The **acc_copyout** and **acc_delete** routines perform a *present decrement* action
 - 3708 with the dynamic reference counter.
 - 3709 – The **acc_copyout_finalize** or **acc_delete_finalize** routines set the dy-
 - 3710 namic reference counter to zero.
- 3711 If both reference counters are then zero:
 - 3712 – The **acc_copyout** routines perform a *copyout* action.
 - 3713 – The **acc_delete** routines perform a *delete* action.

3714 The synchronous versions will not return until the data has been completely transferred and the
 3715 memory has been deallocated.

3716 The **_async** versions of these routines will perform any associated data transfers asynchronously
 3717 on the async queue associated with **async_arg**. The routine may return before the data has been
 3718 transferred or deallocated; see Section 2.16 Asynchronous Behavior for more details. Even if the
 3719 data has not been transferred or deallocated before the routine returns, the data will be treated as not
 3720 present in the current device memory if both reference counters are zero.

3721 Errors

- 3722 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
 3723 **bytes** is nonzero.
- 3724 • An **acc_error_not_present** error is issued if the *data section* is not in shared memory
 3725 and is not present in the current device memory.
- 3726 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
 3727 tion that is not contiguous (in Fortran).
- 3728 • An **acc_error_partly_present** error is issued if part of the *data section* is already
 3729 present in the current device memory but all of the *data section* is not.
- 3730 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
 3731 *argument* value.

3732 See Section 5.2.2.

3733 3.2.20 **acc_update_device** and **acc_update_self**

3734 Summary

3735 The **acc_update_device** and **acc_update_self** routines test to see if the argument is in
 3736 shared memory; if not, the argument must be present in the current device memory, and the routines
 3737 update the data in device memory from the corresponding local memory (**acc_update_device**)
 3738 or update the data in local memory from the corresponding device memory (**acc_update_self**).

3739 Format

3740 C or C++:

```
3741 void acc_update_device(h_void* data_arg, size_t bytes);
3742 void acc_update_self (h_void* data_arg, size_t bytes);
```

```

3743
3744 void acc_update_device_async(h_void* data_arg, size_t bytes,
3745                             int async_arg);
3746 void acc_update_self_async (h_void* data_arg, size_t bytes,
3747                             int async_arg);
3748
3749 Fortran:
3750 subroutine acc_update_device(data_arg [, bytes])
3751 subroutine acc_update_self (data_arg [, bytes])
3752
3753 subroutine acc_update_device_async(data_arg [, bytes], async_arg)
3754 subroutine acc_update_self_async (data_arg [, bytes], async_arg)
3755
3756 type(*), dimension(..) :: data_arg
3757 integer :: bytes
3758 integer(acc_handle_kind) :: async_arg

```

3759 Description

3760 A call to an **acc_update_device** routine is functionally equivalent to an **update device**
3761 directive. A call to an **acc_update_self** routine is functionally equivalent to an **update self**
3762 directive. See Section 2.14.4. The arguments and the *data section* are as for **acc_copyin**.

3763 The behavior of these routines for the associated *data section* is:

- 3764 • If the *data section* is in shared memory or **bytes** is zero, no action is taken.
- 3765 • Otherwise:
 - 3766 – A call to an **acc_update_device** routine copies the data in the local memory to the
3767 corresponding device memory.
 - 3768 – A call to an **acc_update_self** routine copies the data in the corresponding device
3769 memory to the local memory.

3770 The **_async** versions of these routines will perform the data transfers asynchronously on the async
3771 queue associated with **async_arg**. The routine may return before the data has been transferred;
3772 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
3773 until the data has been completely transferred.

3774 Errors

- 3775 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
3776 **bytes** is nonzero.
- 3777 • An **acc_error_not_present** error is issued if the *data section* is not in shared memory
3778 and is not present in the current device memory.
- 3779 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
3780 tion that is not contiguous (in Fortran).
- 3781 • An **acc_error_partly_present** error is issued if part of the *data section* is already
3782 present in the current device memory but all of the *data section* is not.

- 3783 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
3784 *argument* value.

3785 See Section 5.2.2.

3786 3.2.21 **acc_map_data**

3787 Summary

3788 The **acc_map_data** routine maps previously allocated space in the current device memory to the
3789 specified host data.

3790 Format

C or C++:

```
3791     void acc_map_data(h_void* data_arg, d_void* data_dev,  
                      size_t bytes);
```

3792 Description

3793 A call to the **acc_map_data** routine is similar to a call to **acc_create**, except that instead of
3794 allocating new device memory to start a data lifetime, the device address to use for the data lifetime
3795 is specified as an argument. **data_arg** is a host address, **data_dev** is the corresponding device
3796 address, and **bytes** is the length in bytes. **data_dev** may be the result of a call to **acc_malloc**,
3797 or may come from some other device-specific API routine.

3798 The behavior of the **acc_map_data** routine is:

- 3799 • If the data referred to by **data_arg** is in shared memory, the behavior is undefined.
- 3800 • If any of the data referred to by **data_dev** is already mapped to any host memory address,
3801 the behavior is undefined.
- 3802 • Otherwise, after this call, when **data_arg** appears in a data clause, the **data_dev** address
3803 will be used. The dynamic reference count for the data referred to by **data_arg** is set to
3804 one, but no data movement will occur.

3805 Memory mapped by **acc_map_data** may not have the associated dynamic reference count decre-
3806 mented to zero, except by a call to **acc_unmap_data**. See Section 2.6.7 Reference Counters.

3807 Errors

- 3808 • An **acc_invalid_null_pointer** error is issued if either **data_arg** or **data_dev** is
3809 a null pointer.
- 3810 • An **acc_error_present** error is issued if any part of the data is already present in the
3811 current device memory.

3812 See Section 5.2.2.

3813 3.2.22 **acc_unmap_data**

3814 Summary

3815 The **acc_unmap_data** routine unmaps device data from the specified host data.

3816 Format

C or C++:

```
3818     void acc_unmap_data(h_void* data_arg);
```

3819 Description

3820 A call to the `acc_unmap_data` routine is similar to a call to `acc_delete`, except the device
3821 memory is not deallocated. `data_arg` is a host address.

3822 The behavior of the `acc_unmap_data` routine is:

- 3823 • If `data_arg` was not previously mapped to some device address via a call to `acc_map_data`,
3824 the behavior is undefined.
- 3825 • Otherwise, the data lifetime for `data_arg` is ended. The dynamic reference count for
3826 `data_arg` is set to zero, but no data movement will occur and the corresponding device
3827 memory is not deallocated. See Section 2.6.7 Reference Counters.

3828 Errors

- 3829 • An `acc_invalid_null_pointer` error is issued if `data_arg` is a null pointer.
- 3830 • An `acc_error_present` error is issued if the structured reference count for the any part
3831 of the data is not zero.

3832 See Section 5.2.2.

3833 3.2.23 `acc_deviceptr`

3834 Summary

3835 The `acc_deviceptr` routine returns the device pointer associated with a specific host address.

3836 Format

3837 C or C++:

```
3838     d_void* acc_deviceptr(h_void* data_arg);
```

3839 Description

3840 The `acc_deviceptr` routine returns the device pointer associated with a host address. `data_arg`
3841 is the address of a host variable or array that may have an active lifetime on the current device.

3842 The behavior of the `acc_deviceptr` routine for the data referred to by `data_arg` is:

- 3843 • If the data is in shared memory or `data_arg` is a null pointer, `acc_deviceptr` returns
3844 the incoming address.
- 3845 • If the data is not present in the current device memory, `acc_deviceptr` returns a null
3846 pointer.
- 3847 • Otherwise, `acc_deviceptr` returns the address in the current device memory that corre-
3848 sponds to the address `data_arg`.

3849 3.2.24 `acc_hostptr`

3850 Summary

3851 The `acc_hostptr` routine returns the host pointer associated with a specific device address.

3852 Format

3853 C or C++:

```
3854     h_void* acc_hostptr(d_void* data_dev);
```

3855 **Description**

3856 The `acc_hostptr` routine returns the host pointer associated with a device address. `data_dev`
 3857 is the address of a device variable or array, such as that returned from `acc_deviceptr`, `acc_create`
 3858 or `acc_copyin`.

3859 The behavior of the `acc_hostptr` routine for the data referred to by `data_dev` is:

- 3860 • If the data is in shared memory or `data_dev` is a null pointer, `acc_hostptr` returns the
 3861 incoming address.
- 3862 • If the data corresponds to a host address which is present in the current device memory,
 3863 `acc_hostptr` returns the host address.
- 3864 • Otherwise, `acc_hostptr` returns a null pointer.

3865 **3.2.25 acc_is_present**3866 **Summary**

3867 The `acc_is_present` routine tests whether a variable or array region is accessible from the
 3868 current device.

3869 **Format**

3870 C or C++:

```
3871     int acc_is_present(h_void* data_arg, size_t bytes);
```

3872 Fortran:

```
3873     logical function acc_is_present(data_arg)
3874     logical function acc_is_present(data_arg, bytes)
3875     type(*), dimension(..) :: data_arg
3876     integer :: bytes
```

3877 **Description**

3878 The `acc_is_present` routine tests whether the specified host data is accessible from the current
 3879 device. In C/C++, `data_arg` is a pointer to the data, and `bytes` specifies the data size in bytes. In
 3880 Fortran, two forms are supported. In the first, `data_arg` is a variable or contiguous array section.
 3881 In the second, `data_arg` is a variable or array element and `bytes` is the length in bytes. A
 3882 `bytes` value of zero is treated as a value of one if `data_arg` is not a null pointer.

3883 The behavior of the `acc_is_present` routines for the data referred to by `data_arg` is:

- 3884 • If the data is in shared memory, a call to `acc_is_present` will evaluate to *true*.
- 3885 • If the data is present in the current device memory, a call to `acc_is_present` will evaluate
 3886 to *true*.
- 3887 • Otherwise, a call to `acc_is_present` will evaluate to *false*.

3888 **Errors**

- 3889 • An `acc_error_invalid_argument` error is issued if `bytes` is negative (in Fortran).
- 3890 • An `acc_error_invalid_data_section` error is issued if `data_arg` is an array sec-
 3891 tion that is not contiguous (in Fortran).

3892 See Section 5.2.2.

3893 **3.2.26 acc_memcpy_to_device**3894 **Summary**3895 The **acc_memcpy_to_device** routine copies data from local memory to device memory.3896 **Format**

C or C++:

```

3897     void acc_memcpy_to_device(d_void* data_dev_dest,
                               h_void* data_host_src, size_t bytes);
3898     void acc_memcpy_to_device_async(d_void* data_dev_dest,
                                     h_void* data_host_src, size_t bytes,
                                     int async_arg);

```

3898 **Description**

3899 The **acc_memcpy_to_device** routine copies **bytes** bytes of data from the local address in
3900 **data_host_src** to the device address in **data_dev_dest**. **data_dev_dest** must be an
3901 address accessible from the current device, such as an address returned from **acc_malloc** or
3902 **acc_deviceptr**, or an address in shared memory.

3903 The behavior of the **acc_memcpy_to_device** routines is:

- 3904 • If **bytes** is zero, no action is taken.
- 3905 • If **data_dev_dest** and **data_host_src** both refer to shared memory and have the same
3906 value, no action is taken.
- 3907 • If **data_dev_dest** and **data_host_src** both refer to shared memory and the memory
3908 regions overlap, the behavior is undefined.
- 3909 • If the data referred to by **data_dev_dest** is not accessible by the current device, the be-
3910 havior is undefined.
- 3911 • If the data referred to by **data_host_src** is not accessible by the local thread, the behavior
3912 is undefined.
- 3913 • Otherwise, **bytes** bytes of data at **data_host_src** in local memory are copied to
3914 **data_dev_dest** in the current device memory.

3915 The **_async** version of this routine will perform the data transfers asynchronously on the async
3916 queue associated with **async_arg**. The routine may return before the data has been transferred;
3917 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
3918 until the data has been completely transferred.

3919 **Errors**

- 3920 • An **acc_error_invalid_null_pointer** error is issued if **data_dev_dest** or
3921 **data_host_src** is a null pointer and **bytes** is nonzero.
- 3922 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
3923 *argument* value.

3924 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);
```

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);
```

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 memory to point to the corresponding device copy of the host pointer target. The `acc_detach` routines restore a pointer in device memory to point to the host pointer target.

3993 **Format**

3994 C or C++:

```

3995     void acc_attach(h_void** ptr_addr);
3996     void acc_attach_async(h_void** ptr_addr, int async_arg);
3997
3998     void acc_detach(h_void** ptr_addr);
3999     void acc_detach_async(h_void** ptr_addr, int async_arg);
4000     void acc_detach_finalize(h_void** ptr_addr);
4001     void acc_detach_finalize_async(h_void** ptr_addr,
4002                                     int async_arg);

```

4003 **Description**

4004 A call to an **acc_attach** routine is functionally equivalent to an **enter data attach** direc-
4005 tive, as described in Section 2.7.12. A call to an **acc_detach** routine is functionally equivalent to
4006 an **exit data detach** directive, and a call to an **acc_detach_finalize** routine is function-
4007 ally equivalent to an **exit data finalize detach** directive, as described in Section 2.7.13.
4008 **ptr_addr** must be the address of a host pointer. **async_arg** must be an *async-argument* as
4009 defined in Section 2.16.

4010 The behavior of these routines is:

- 4011 • If **ptr_addr** refers to shared memory, no action is taken.
- 4012 • If the pointer referred to by **ptr_addr** is not present in the current device memory, no action
4013 is taken.
- 4014 • Otherwise:
 - 4015 – The **acc_attach** routines perform an *attach* action on the pointer referred to by
4016 **ptr_addr**; see Section 2.7.2.
 - 4017 – The **acc_detach** routines perform a *detach* action on the pointer referred to by **ptr_addr**;
4018 See Section 2.7.2.
 - 4019 – The **acc_detach_finalize** routines perform an *immediate detach* action on the
4020 pointer referred to by **ptr_addr**; see Section 2.7.2.

4021 These routines may issue a data transfer from local memory to device memory. The **_async** ver-
4022 sions of these routines will perform the data transfers asynchronously on the async queue associated
4023 with **async_arg**. These routines may return before the data has been transferred; see Section 2.16
4024 for more details. The synchronous versions will not return until the data has been completely trans-
4025 ferred.

4026 **Errors**

- 4027 • An **acc_error_invalid_null_pointer** error is issued if **ptr_addr** is a null pointer.
- 4028 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4029 *argument* value.

4030 See Section 5.2.2.

4031 **3.2.30 acc_memcpy_d2d**

4032 **Summary**

4033 The `acc_memcpy_d2d` routines copy the contents of an array on one device to an array on the
 4034 same or a different device without updating the value on the host.

4035 **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);

```

4036

4037

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)

```

4038
 4039 `type(*)`, `dimension(..)` :: `data_arg_dest`
 4040 `type(*)`, `dimension(..)` :: `data_arg_src`
 4041 `integer` :: `bytes`
 4042 `integer` :: `dev_num_dest`
 4043 `integer` :: `dev_num_src`
 4044 `integer` :: `async_arg_src`

4038

4039

4040

4041

4042

4043

4044

4045

4046 **Description**

4047 The `acc_memcpy_d2d` routines are passed the address of destination and source host data as well
 4048 as integer device numbers for the destination and source devices, which must both be of the current
 4049 device type.

4050 The behavior of the `acc_memcpy_d2d` routines is:

- 4051 • If `bytes` is zero, no action is taken.
- 4052 • If both pointers have the same value and either the two device numbers are the same or the
 4053 addresses are in shared memory, then no action is taken.
- 4054 • Otherwise, `bytes` bytes of data at the device address corresponding to `data_arg_src` on
 4055 device `dev_num_src` are copied to the device address corresponding to `data_arg_dest`
 4056 on device `dev_num_dest`.

4057 For `acc_memcpy_d2d_async` the value of `async_arg_src` is the number of an async queue
 4058 on the source device. This routine will perform the data transfers asynchronously on the async queue
 4059 associated with `async_arg_src` for device `dev_num_src`; see Section 2.16 Asynchronous Behavior
 4060 for more details.

4061 **Errors**

- 4062 • An **acc_error_device_unavailable** error is issued if **dev_num_dest** or **dev_num_src**
4063 is not a valid device number.
- 4064 • An **acc_error_invalid_null_pointer** error is issued if either **data_arg_dest**
4065 or **data_arg_src** is a null pointer and **bytes** is nonzero.
- 4066 • An **acc_error_not_present** error is issued if the data at either address is not in shared
4067 memory and is not present in the respective device memory.
- 4068 • An **acc_error_partly_present** error is issued if part of the data is already present in
4069 the current device memory but all of the data is not.
- 4070 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4071 *argument* value.

4072 See Section 5.2.2.

4. Environment Variables

4073

4074 This chapter describes the environment variables that modify the behavior of accelerator regions.
4075 The names of the environment variables must be upper case. The values assigned environment
4076 variables are case-insensitive and may have leading and trailing whitespace. If the values of the
4077 environment variables change after the program has started, even if the program itself modifies the
4078 values, the behavior is implementation-defined.

4.1 ACC_DEVICE_TYPE

4079

4080 The **ACC_DEVICE_TYPE** environment variable controls the default device type to use when ex-
4081 ecuting parallel, serial, and kernels regions, if the program has been compiled to use more than
4082 one different type of device. The allowed values of this environment variable are implementation-
4083 defined. See the release notes for currently-supported values of this environment variable.

4084 Example:

```
4085     setenv ACC_DEVICE_TYPE NVIDIA  
4086     export ACC_DEVICE_TYPE=NVIDIA
```

4.2 ACC_DEVICE_NUM

4087

4088 The **ACC_DEVICE_NUM** environment variable controls the default device number to use when
4089 executing accelerator regions. The value of this environment variable must be a nonnegative integer
4090 between zero and the number of devices of the desired type attached to the host. If the value is
4091 greater than or equal to the number of devices attached, the behavior is implementation-defined.

4092 Example:

```
4093     setenv ACC_DEVICE_NUM 1  
4094     export ACC_DEVICE_NUM=1
```

4.3 ACC_PROFLIB

4095

4096 The **ACC_PROFLIB** environment variable specifies the profiling library. More details about the
4097 evaluation at runtime is given in section 5.3.3 Runtime Dynamic Library Loading.

4098 Example:

```
4099     setenv ACC_PROFLIB /path/to/proflib/libaccprof.so  
4100     export ACC_PROFLIB=/path/to/proflib/libaccprof.so
```


5. Profiling and Error Callback Interface

4101

4102 This chapter describes the OpenACC interface for runtime callback routines. These routines may be
4103 provided by the programmer or by a tool or library developer. Calls to these routines are triggered
4104 during the application execution at specific OpenACC events. There are two classes of events,
4105 profiling events and error events. Profiling events can be used by tools for profile or trace data
4106 collection. Currently, this interface does not support tools that employ asynchronous sampling.
4107 Error events can be used to release resources or cleanly shut down a large parallel application when
4108 the OpenACC runtime detects an error condition from which it cannot recover. This is specifically
4109 for error handling, not for error recovery. There is no support provided for restarting or retrying
4110 an OpenACC program, construct, or API routine after an error condition has been detected and an
4111 error callback routine has been called.

4112 In this chapter, the term *runtime* refers to the OpenACC runtime library. The term *library* refers to
4113 the routines invoked at specified events by the OpenACC runtime.

4114 There are three steps for interfacing a *library* to the *runtime*. The first step is to write the library
4115 callback routines. Section 5.1 Events describes the supported runtime events and the order in which
4116 callbacks to the callback routines will occur. Section 5.2 Callbacks Signature describes the signature
4117 of the callback routines for all events.

4118 The second step is to load the *library* at runtime. The *library* may be statically linked to the appli-
4119 cation or dynamically loaded by the application, a library, or a tool. This is described in Section 5.3
4120 Loading the Library.

4121 The third step is to register the desired callbacks with the events. This may be done explicitly by the
4122 application, if the library is statically linked with the application, implicitly by including a call to a
4123 registration routine in a `.init` section, or by including an initialization routine in the library if it is
4124 dynamically loaded by the *runtime*. This is described in Section 5.4 Registering Event Callbacks.

5.1 Events

4125

4126 This section describes the events that are recognized by the runtime. Most profiling events have a
4127 start and end callback routine, that is, a routine that is called just before the runtime code to handle
4128 the event starts and another routine that is called just after the event is handled. The event names
4129 and routine prototypes are available in the header file `acc_callback.h`, which is delivered with
4130 the OpenACC implementation. For backward compatibility with previous versions of OpenACC,
4131 the implementation also delivers the same information in `acc_prof.h`. Event names are prefixed
4132 with `acc_ev_`.

4133 The ordering of events must reflect the order in which the OpenACC runtime actually executes them,
4134 i.e. if a runtime moves the enqueueing of data transfers or kernel launches outside the originating
4135 clauses/constructs, it needs to issue the corresponding launch callbacks when they really occur. A
4136 callback for a start event must always precede the matching end callback. No callbacks will be
4137 issued after a runtime shutdown event.

4138 The events that the runtime supports can be registered with a callback and are defined in the enu-
4139 meration type `acc_event_t`.

```
4140     typedef enum acc_event_t{
4141         acc_ev_none = 0,
4142         acc_ev_device_init_start = 1,
4143         acc_ev_device_init_end = 2,
4144         acc_ev_device_shutdown_start = 3,
4145         acc_ev_device_shutdown_end = 4,
4146         acc_ev_runtime_shutdown = 5,
4147         acc_ev_create = 6,
4148         acc_ev_delete = 7,
4149         acc_ev_alloc = 8,
4150         acc_ev_free = 9,
4151         acc_ev_enter_data_start = 10,
4152         acc_ev_enter_data_end = 11,
4153         acc_ev_exit_data_start = 12,
4154         acc_ev_exit_data_end = 13,
4155         acc_ev_update_start = 14,
4156         acc_ev_update_end = 15,
4157         acc_ev_compute_construct_start = 16,
4158         acc_ev_compute_construct_end = 17,
4159         acc_ev_enqueue_launch_start = 18,
4160         acc_ev_enqueue_launch_end = 19,
4161         acc_ev_enqueue_upload_start = 20,
4162         acc_ev_enqueue_upload_end = 21,
4163         acc_ev_enqueue_download_start = 22,
4164         acc_ev_enqueue_download_end = 23,
4165         acc_ev_wait_start = 24,
4166         acc_ev_wait_end = 25,
4167         acc_ev_error = 100,
4168         acc_ev_last = 101
4169     }acc_event_t;
```

4170 The value of `acc_ev_last` will change if new events are added to the enumeration, so a library
4171 should not depend on that value.

4172 5.1.1 Runtime Initialization and Shutdown

4173 No callbacks can be registered for the runtime initialization. Instead the initialization of the tool is
4174 handled as described in Section 5.3 Loading the Library.

4175 The *runtime shutdown* profiling event name is

```
4176     acc_ev_runtime_shutdown
```

4177 This event is triggered before the OpenACC runtime shuts down, either because all devices have
4178 been shutdown by calls to the `acc_shutdown` API routine, or at the end of the program.

4179 5.1.2 Device Initialization and Shutdown

4180 The *device initialization* profiling event names are

4181 **acc_ev_device_init_start**
4182 **acc_ev_device_init_end**

4183 These events are triggered when a device is being initialized by the OpenACC runtime. This may be
4184 when the program starts, or may be later during execution when the program reaches an **acc_init**
4185 call or an OpenACC construct. The **acc_ev_device_init_start** is triggered before device
4186 initialization starts and **acc_ev_device_init_end** after initialization is complete.

4187 The *device shutdown* profiling event names are

4188 **acc_ev_device_shutdown_start**
4189 **acc_ev_device_shutdown_end**

4190 These events are triggered when a device is shut down, most likely by a call to the OpenACC
4191 **acc_shutdown** API routine. The **acc_ev_device_shutdown_start** is triggered before
4192 the device shutdown process starts and **acc_ev_device_shutdown_end** after the device shut-
4193 down is complete.

4194 **5.1.3 Enter Data and Exit Data**

4195 The *enter data* profiling event names are

4196 **acc_ev_enter_data_start**
4197 **acc_ev_enter_data_end**

4198 These events are triggered at **enter data** directives, entry to data constructs, and entry to implicit
4199 data regions such as those generated by compute constructs. The **acc_ev_enter_data_start**
4200 event is triggered before any *data allocation*, *data update*, or *wait* events that are associated with
4201 that directive or region entry, and the **acc_ev_enter_data_end** is triggered after those events.

4202 The *exit data* profiling event names are

4203 **acc_ev_exit_data_start**
4204 **acc_ev_exit_data_end**

4205 These events are triggered at **exit data** directives, exit from **data** constructs, and exit from
4206 implicit data regions. The **acc_ev_exit_data_start** event is triggered before any *data*
4207 *deallocation*, *data update*, or *wait* events associated with that directive or region exit, and the
4208 **acc_ev_exit_data_end** event is triggered after those events.

4209 When the construct that triggers an *enter data* or *exit data* event was generated implicitly by the
4210 compiler the **implicit** field in the event structure will be set to **1**. When the construct that
4211 triggers these events was specified explicitly by the application code the **implicit** field in the
4212 event structure will be set to **0**.

4213 **5.1.4 Data Allocation**

4214 The *data allocation* profiling event names are

4215 **acc_ev_create**
4216 **acc_ev_delete**
4217 **acc_ev_alloc**
4218 **acc_ev_free**

4219 An **acc_ev_alloc** event is triggered when the OpenACC runtime allocates memory from the de-
 4220 vice memory pool, and an **acc_ev_free** event is triggered when the runtime frees that memory.
 4221 An **acc_ev_create** event is triggered when the OpenACC runtime associates device memory
 4222 with local memory, such as for a data clause (**create**, **copyin**, **copy**, **copyout**) at entry to
 4223 a data construct, compute construct, at an **enter data** directive, or in a call to a data API rou-
 4224 tine (**acc_copyin**, **acc_create**, ...). An **acc_ev_create** event may be preceded by an
 4225 **acc_ev_alloc** event, if newly allocated memory is used for this device data, or it may not, if
 4226 the runtime manages its own memory pool. An **acc_ev_delete** event is triggered when the
 4227 OpenACC runtime disassociates device memory from local memory, such as for a data clause at
 4228 exit from a data construct, compute construct, at an **exit data** directive, or in a call to a data API
 4229 routine (**acc_copyout**, **acc_delete**, ...). An **acc_ev_delete** event may be followed by
 4230 an **acc_ev_free** event, if the disassociated device memory is freed, or it may not, if the runtime
 4231 manages its own memory pool.

4232 When the action that generates a *data allocation* event was generated explicitly by the application
 4233 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event
 4234 is triggered because of a variable or array with implicitly-determined data attributes or otherwise
 4235 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

4236 5.1.5 Data Construct

4237 The profiling events for entering and leaving *data constructs* are mapped to *enter data* and *exit data*
 4238 events as described in Section 5.1.3 Enter Data and Exit Data.

4239 5.1.6 Update Directive

4240 The *update directive* profiling event names are

```
4241     acc_ev_update_start
4242     acc_ev_update_end
```

4243 The **acc_ev_update_start** event will be triggered at an **update** directive, before any *data*
 4244 *update* or *wait* events that are associated with the update directive are carried out, and the corre-
 4245 sponding **acc_ev_update_end** event will be triggered after any of the associated events.

4246 5.1.7 Compute Construct

4247 The *compute construct* profiling event names are

```
4248     acc_ev_compute_construct_start
4249     acc_ev_compute_construct_end
```

4250 The **acc_ev_compute_construct_start** event is triggered at entry to a compute construct,
 4251 before any *launch* events that are associated with entry to the compute construct. The
 4252 **acc_ev_compute_construct_end** event is triggered at the exit of the compute construct,
 4253 after any *launch* events associated with exit from the compute construct. If there are data clauses
 4254 on the compute construct, those data clauses may be treated as part of the compute construct, or as
 4255 part of a data construct containing the compute construct. The callbacks for data clauses must use
 4256 the same line numbers as for the compute construct events.

5.1.8 Enqueue Kernel Launch

4257

The *launch* profiling event names are

4258

acc_ev_enqueue_launch_start

4259

acc_ev_enqueue_launch_end

4260

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.

4261

4262

4263

4264

4265

4266

4267

4268

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.

4269

4270

4271

5.1.9 Enqueue Data Update (Upload and Download)

4272

The *data update* profiling event names are

4273

acc_ev_enqueue_upload_start

4274

acc_ev_enqueue_upload_end

4275

acc_ev_enqueue_download_start

4276

acc_ev_enqueue_download_end

4277

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.

4278

4279

4280

4281

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.

4282

4283

4284

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**.

4285

4286

4287

4288

5.1.10 Wait

4289

The *wait* profiling event names are

4290

acc_ev_wait_start

4291

acc_ev_wait_end

4292

4293

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

4294

4295

4296 queue after the local thread has determined that the queue is empty.

4297 Wait events occur when the local thread and a device synchronize, either due to a **wait** directive
4298 or by a *wait* clause on a synchronous data construct, compute construct, or **enter data**, **exit**
4299 **data**, or **update** directive. For *wait* events triggered by an explicit synchronous **wait** directive
4300 or *wait* clause, the **implicit** field in the event structure will be **0**. For all other wait events, the
4301 **implicit** field in the event structure will be **1**.

4302 The OpenACC runtime need not trigger *wait* events for queues that have not been used in the
4303 program, and need not trigger *wait* events for queues that have not been used by this thread since
4304 the last *wait* operation. For instance, an **acc wait** directive with no arguments is defined to wait on
4305 all queues. If the program only uses the default (synchronous) queue and the queue associated with
4306 **async (1)** and **async (2)** then an **acc wait** directive may trigger *wait* events only for those
4307 three queues. If the implementation knows that no activities have been enqueued on the **async (2)**
4308 queue since the last *wait* operation, then the **acc wait** directive may trigger *wait* events only for
4309 the default queue and the **async (1)** queue.

4310 5.1.11 Error Event

4311 The only error event is

4312 **acc_ev_error**

4313 An **acc_ev_error** event is triggered when the OpenACC program detects a runtime error con-
4314 dition. The default runtime error callback routine may print an error message and halt program
4315 execution. An application can register additional error event callback routines, to allow a failing
4316 application to release resources or to cleanly shut down a large parallel runtime with many threads
4317 and processes, for instance.

4318 The application can register multiple alternate error callbacks. As described in Section
4319 5.4.1 Multiple Callbacks, the callbacks will be invoked in the order in which they are registered.
4320 If all the error callbacks return, the default error callback will be invoked. The error callback
4321 routine must not execute any OpenACC compute or data constructs. The only OpenACC API
4322 routines that can be safely invoked from an error callback routine are **acc_get_property**,
4323 **acc_get_property_string**, and **acc_shutdown**.

4324 5.2 Callbacks Signature

4325 This section describes the signature of event callbacks. All event callbacks have the same signature.
4326 The routine prototypes are available in the header file **acc_callback.h**, which is delivered with
4327 the OpenACC implementation.

4328 All callback routines have three arguments. The first argument is a pointer to a struct containing
4329 general information; the same struct type is used for all callback events. The second argument is
4330 a pointer to a struct containing information specific to that callback event; there is one struct type
4331 containing information for data events, another struct type containing information for kernel launch
4332 events, and a third struct type for other events, containing essentially no information. The third
4333 argument is a pointer to a struct containing information about the application programming interface
4334 (API) being used for the specific device. For NVIDIA CUDA devices, this contains CUDA-specific
4335 information; for OpenCL devices, this contains OpenCL-specific information. Other interfaces can
4336 be supported as they are added by implementations. The prototype for a callback routine is:

```

4337     typedef void (*acc_callback)
4338         (acc_callback_info*, acc_event_info*, acc_api_info*);
4339     typedef acc_callback acc_prof_callback;

```

4340 In the descriptions, the datatype `ssize_t` means a signed 32-bit integer for a 32-bit binary and
 4341 a 64-bit integer for a 64-bit binary, the datatype `size_t` means an unsigned 32-bit integer for a
 4342 32-bit binary and a 64-bit integer for a 64-bit binary, and the datatype `int` means a 32-bit integer
 4343 for both 32-bit and 64-bit binaries.

4344 5.2.1 First Argument: General Information

4345 The first argument is a pointer to the `acc_callback_info` struct type:

```

4346     typedef struct acc_prof_info{
4347         acc_event_t event_type;
4348         int valid_bytes;
4349         int version;
4350         acc_device_t device_type;
4351         int device_number;
4352         int thread_id;
4353         ssize_t async;
4354         ssize_t async_queue;
4355         const char* src_file;
4356         const char* func_name;
4357         int line_no, end_line_no;
4358         int func_line_no, func_end_line_no;
4359     }acc_callback_info;
4360     typedef struct acc_prof_info acc_prof_info;

```

4361 The name `acc_prof_info` is preserved for backward compatibility with previous versions of
 4362 OpenACC. The fields are described below.

4363 • **acc_event_t event_type** - The event type that triggered this callback. The datatype
 4364 is the enumeration type `acc_event_t`, described in the previous section. This allows the
 4365 same callback routine to be used for different events.

4366 • **int valid_bytes** - The number of valid bytes in this struct. This allows a library to inter-
 4367 face with newer runtimes that may add new fields to the struct at the end while retaining com-
 4368 patibility with older runtimes. A runtime must fill in the `event_type` and `valid_bytes`
 4369 fields, and must fill in values for all fields with offset less than `valid_bytes`. The value of
 4370 `valid_bytes` for a struct is recursively defined as:

```

4371     valid_bytes(struct) = offset(lastfield) + valid_bytes(lastfield)
4372     valid_bytes(type[n]) = (n-1)*sizeof(type) + valid_bytes(type)
4373     valid_bytes(basictype) = sizeof(basictype)

```

4374 • **int version** - A version number; the value of `_OPENACC`.

4375 • **acc_device_t device_type** - The device type corresponding to this event. The datatype
 4376 is `acc_device_t`, an enumeration type of all the supported device types, defined in `openacc.h`.

4377 • **int device_number** - The device number. Each device is numbered, typically starting at

- 4378 device zero. For applications that use more than one device type, the device numbers may be
4379 unique across all devices or may be unique only across all devices of the same device type.
- 4380 • **int thread_id** - The host thread ID making the callback. Host threads are given unique
4381 thread ID numbers typically starting at zero. This is not necessarily the same as the OpenMP
4382 thread number.
 - 4383 • **ssize_t async** - The *async-value* used for operations associated with this event; see Sec-
4384 tion 2.16 Asynchronous Behavior.
 - 4385 • **ssize_t async_queue** - The actual activity queue onto which the **async** field gets
4386 mapped; see Section 2.16 Asynchronous Behavior.
 - 4387 • **const char* src_file** - A pointer to null-terminated string containing the name of or
4388 path to the source file, if known, or a null pointer if not. If the library wants to save the source
4389 file name, it should allocate memory and copy the string.
 - 4390 • **const char* func_name** - A pointer to a null-terminated string containing the name of
4391 the function in which the event occurred, if known, or a null pointer if not. If the library wants
4392 to save the function name, it should allocate memory and copy the string.
 - 4393 • **int line_no** - The line number of the directive or program construct or the starting line
4394 number of the OpenACC construct corresponding to the event. A negative or zero value
4395 means the line number is not known.
 - 4396 • **int end_line_no** - For an OpenACC construct, this contains the line number of the end
4397 of the construct. A negative or zero value means the line number is not known.
 - 4398 • **int func_line_no** - The line number of the first line of the function named in **func_name**.
4399 A negative or zero value means the line number is not known.
 - 4400 • **int func_end_line_no** - The last line number of the function named in **func_name**.
4401 A negative or zero value means the line number is not known.

4402 5.2.2 Second Argument: Event-Specific Information

4403 The second argument is a pointer to the **acc_event_info** union type.

```
4404     typedef union acc_event_info{
4405         acc_event_t event_type;
4406         acc_data_event_info data_event;
4407         acc_launch_event_info launch_event;
4408         acc_other_event_info other_event;
4409     }acc_event_info;
```

4410 The **event_type** field selects which union member to use. The first five members of each union
4411 member are identical. The second through fifth members of each union member (**valid_bytes**,
4412 **parent_construct**, **implicit**, and **tool_info**) have the same semantics for all event
4413 types:

- 4414 • **int valid_bytes** - The number of valid bytes in the respective struct. (This field is similar
4415 used as discussed in Section 5.2.1 First Argument: General Information.)

- 4416 • **acc_construct_t parent_construct** - This field describes the type of construct
4417 that caused the event to be emitted. The possible values for this field are defined by the
4418 **acc_construct_t** enum, described at the end of this section.
- 4419 • **int implicit** - This field is set to 1 for any implicit event, such as an implicit wait at
4420 a synchronous data construct or synchronous enter data, exit data or update directive. This
4421 field is set to zero when the event is triggered by an explicit directive or call to a runtime API
4422 routine.
- 4423 • **void* tool_info** - This field is used to pass tool-specific information from a **_start**
4424 event to the matching **_end** event. For a **_start** event callback, this field will be initialized
4425 to a null pointer. The value of this field for a **_end** event will be the value returned by the
4426 library in this field from the matching **_start** event callback, if there was one, or a null
4427 pointer otherwise. For events that are neither **_start** or **_end** events, this field will be a
4428 null pointer.

4429 Data Events

4430 For a data event, as noted in the event descriptions, the second argument will be a pointer to the
4431 **acc_data_event_info** struct.

```
4432 typedef struct acc_data_event_info{
4433     acc_event_t event_type;
4434     int valid_bytes;
4435     acc_construct_t parent_construct;
4436     int implicit;
4437     void* tool_info;
4438     const char* var_name;
4439     size_t bytes;
4440     const void* host_ptr;
4441     const void* device_ptr;
4442 }acc_data_event_info;
```

4443 The fields specific for a data event are:

- 4444 • **acc_event_t event_type** - The event type that triggered this callback. The events that
4445 use the **acc_data_event_info** struct are:
 - 4446 **acc_ev_enqueue_upload_start**
 - 4447 **acc_ev_enqueue_upload_end**
 - 4448 **acc_ev_enqueue_download_start**
 - 4449 **acc_ev_enqueue_download_end**
 - 4450 **acc_ev_create**
 - 4451 **acc_ev_delete**
 - 4452 **acc_ev_alloc**
 - 4453 **acc_ev_free**
- 4454 • **const char* var_name** - A pointer to null-terminated string containing the name of the
4455 variable for which this event is triggered, if known, or a null pointer if not. If the library wants
4456 to save the variable name, it should allocate memory and copy the string.
- 4457 • **size_t bytes** - The number of bytes for the data event.

- 4458 • **const void* host_ptr** - If available and appropriate for this event, this is a pointer to
4459 the host data.
- 4460 • **const void* device_ptr** - If available and appropriate for this event, this is a pointer
4461 to the corresponding device data.

4462 Launch Events

4463 For a launch event, as noted in the event descriptions, the second argument will be a pointer to the
4464 **acc_launch_event_info** struct.

```
4465     typedef struct acc_launch_event_info{
4466         acc_event_t event_type;
4467         int valid_bytes;
4468         acc_construct_t parent_construct;
4469         int implicit;
4470         void* tool_info;
4471         const char* kernel_name;
4472         size_t num_gangs, num_workers, vector_length;
4473     }acc_launch_event_info;
```

4474 The fields specific for a launch event are:

- 4475 • **acc_event_t event_type** - The event type that triggered this callback. The events that
4476 use the **acc_launch_event_info** struct are:

```
4477         acc_ev_enqueue_launch_start
4478         acc_ev_enqueue_launch_end
```

- 4479 • **const char* kernel_name** - A pointer to null-terminated string containing the name of
4480 the kernel being launched, if known, or a null pointer if not. If the library wants to save the
4481 kernel name, it should allocate memory and copy the string.
- 4482 • **size_t num_gangs, num_workers, vector_length** - The number of gangs, work-
4483 ers and vector lanes created for this kernel launch.

4484 Error Events

4485 For an error event, as noted in the event descriptions, the second argument will be a pointer to the
4486 **acc_error_event_info** struct.

```
4487     typedef struct acc_error_event_info{
4488         acc_event_t event_type;
4489         int valid_bytes;
4490         acc_construct_t parent_construct;
4491         int implicit;
4492         void* tool_info;
4493         acc_error_t error_code;
4494         const char* error_message;
4495         size_t runtime_info;
4496     }acc_error_event_info;
```

4497 The enumeration type for the error code is


```

4498     typedef enum acc_error_t{
4499         acc_error_none = 0,
4500         acc_error_other = 1,
4501         acc_error_system = 2,
4502         acc_error_execution = 3,
4503         acc_error_device_init = 4,
4504         acc_error_device_shutdown = 5,
4505         acc_error_device_unavailable = 6,
4506         acc_error_device_type_unavailable = 7,
4507         acc_error_wrong_device_type = 8,
4508         acc_error_out_of_memory = 9,
4509         acc_error_not_present = 10,
4510         acc_error_partly_present = 11,
4511         acc_error_present = 12,
4512         acc_error_invalid_argument = 13,
4513         acc_error_invalid_async = 14,
4514         acc_error_invalid_null_pointer = 15,
4515         acc_error_invalid_data_section = 16,
4516         acc_error_implementation_defined = 100
4517     }acc_error_t;

```

4518 The fields specific for an error event are:

- 4519 • **acc_event_t event_type** - The event type that triggered this callback. The only event
- 4520 that uses the **acc_error_event_info** struct is:

```
4521     acc_ev_error
```

- 4522 • **int implicit** - This will be set to 1.
- 4523 • **acc_error_t error_code** - The error codes used are:
 - 4524 – **acc_error_other** is used for error conditions other than those described below.
 - 4525 – **acc_error_system** is used when there is a system error condition.
 - 4526 – **acc_error_execution** is used when there is an error condition issued from code
 - 4527 executing on the device.
 - 4528 – **acc_error_device_init** is used for any error initializing a device.
 - 4529 – **acc_error_device_shutdown** is used for any error shutting down a device.
 - 4530 – **acc_error_device_unavailable** is used when there is an error where the se-
 - 4531 lected device is unavailable.
 - 4532 – **acc_error_device_type_unavailable** is used when there is an error where
 - 4533 no device of the selected device type is available or is supported.
 - 4534 – **acc_error_wrong_device_type** is used when there is an error related to the
 - 4535 device type, such as a mismatch between the device type for which a compute construct
 - 4536 was compiled and the device available at runtime.
 - 4537 – **acc_error_out_of_memory** is used when the program tries to allocate more mem-
 - 4538 ory on the device than is available.

- 4539 – **acc_error_not_present** is used for an error related to data not being present at
4540 runtime.
- 4541 – **acc_error_partly_present** is used for an error related to part of the data being
4542 present but not being completely present at runtime.
- 4543 – **acc_error_present** is used for an error related to data being unexpectedly present
4544 at runtime.
- 4545 – **acc_error_invalid_argument** is used when an API routine is called with a
4546 invalid argument value, other than those described above.
- 4547 – **acc_error_invalid_async** is used when an API routine is called with an invalid
4548 *async-argument*, or when a directive is used with an invalid *async-argument*.
- 4549 – **acc_error_invalid_null_pointer** is used when an API routine is called with
4550 a null pointer argument where it is invalid, or when a directive is used with a null pointer
4551 in a context where it is invalid.
- 4552 – **acc_error_invalid_data_section** is used when an invalid array section ap-
4553 pears in a directive data clause, or an invalid array section appears as a runtime API call
4554 argument.
- 4555 – **acc_error_implementation_defined**: any value greater or equal to this value
4556 may be used for an implementation-defined error code.
- 4557 • **const char* error_message** - A pointer to null-terminated string containing an error
4558 message from the OpenACC runtime describing the error, or a null pointer.
- 4559 • **size_t runtime_info** - A value, such as an error code, from the underlying device
4560 runtime or driver, if one is available and appropriate.

4561 Other Events

4562 For any event that does not use the **acc_data_event_info**, **acc_launch_event_info**, or
4563 **acc_error_event_info** struct, the second argument to the callback routine will be a pointer
4564 to **acc_other_event_info** struct.

```
4565     typedef struct acc_other_event_info{
4566         acc_event_t event_type;
4567         int valid_bytes;
4568         acc_construct_t parent_construct;
4569         int implicit;
4570         void* tool_info;
4571     }acc_other_event_info;
```

4572 Parent Construct Enumeration

4573 All event structures contain a **parent_construct** member that describes the type of construct
4574 that caused the event to be emitted. The purpose of this field is to provide a means to identify
4575 the type of construct emitting the event in the cases where an event may be emitted by multi-
4576 ple construct types, such as is the case with data and wait events. The possible values for the
4577 **parent_construct** field are defined in the enumeration type **acc_construct_t**. In the
4578 case of combined directives, the outermost construct of the combined construct should be specified

4579 as the **parent_construct**. If the event was emitted as the result of the application making a
 4580 call to the runtime api, the value will be **acc_construct_runtime_api**.

```

4581     typedef enum acc_construct_t{
4582         acc_construct_parallel = 0,
4583         acc_construct_serial = 16
4584         acc_construct_kernels = 1,
4585         acc_construct_loop = 2,
4586         acc_construct_data = 3,
4587         acc_construct_enter_data = 4,
4588         acc_construct_exit_data = 5,
4589         acc_construct_host_data = 6,
4590         acc_construct_atomic = 7,
4591         acc_construct_declare = 8,
4592         acc_construct_init = 9,
4593         acc_construct_shutdown = 10,
4594         acc_construct_set = 11,
4595         acc_construct_update = 12,
4596         acc_construct_routine = 13,
4597         acc_construct_wait = 14,
4598         acc_construct_runtime_api = 15,
4599     }acc_construct_t;
  
```

4600 5.2.3 Third Argument: API-Specific Information

4601 The third argument is a pointer to the **acc_api_info** struct type, shown here.

```

4602     typedef struct acc_api_info{
4603         acc_device_api device_api;
4604         int valid_bytes;
4605         acc_device_t device_type;
4606         int vendor;
4607         const void* device_handle;
4608         const void* context_handle;
4609         const void* async_handle;
4610     }acc_api_info;
  
```

4611 The fields are described below:

- 4612 • **acc_device_api device_api** - The API in use for this device. The data type is the
 4613 enumeration **acc_device_api**, which is described later in this section.
- 4614 • **int valid_bytes** - The number of valid bytes in this struct. See the discussion above in
 4615 Section 5.2.1 First Argument: General Information.
- 4616 • **acc_device_t device_type** - The device type; the datatype is **acc_device_t**, de-
 4617 fined in **openacc.h**.
- 4618 • **int vendor** - An identifier to identify the OpenACC vendor; contact your vendor to deter-
 4619 mine the value used by that vendor's runtime.

- 4620 • **const void* device_handle** - If applicable, this will be a pointer to the API-specific
4621 device information.
- 4622 • **const void* context_handle** - If applicable, this will be a pointer to the API-specific
4623 context information.
- 4624 • **const void* async_handle** - If applicable, this will be a pointer to the API-specific
4625 async queue information.

4626 According to the value of **device_api** a library can cast the pointers of the fields **device_handle**,
4627 **context_handle** and **async_handle** to the respective device API type. The following device
4628 APIs are defined in the interface below. Any implementation-defined device API type must have a
4629 value greater than **acc_device_api_implementation_defined**.

```

4630     typedef enum acc_device_api{
4631         acc_device_api_none = 0,           /* no device API */
4632         acc_device_api_cuda = 1,         /* CUDA driver API */
4633         acc_device_api_opencl = 2,       /* OpenCL API */
4634         acc_device_api_other = 4,        /* other device API */
4635         acc_device_api_implementation_defined = 1000 /* other device API */
4636     }acc_device_api;

```

4632 5.3 Loading the Library

4633 This section describes how a tools library is loaded when the program is run. Four methods are
4634 described.

- 4635 • A tools library may be linked with the program, as any other library is linked, either as a
4636 static library or a dynamic library, and the runtime will call a predefined library initialization
4637 routine that will register the event callbacks.
- 4638 • The OpenACC runtime implementation may support a dynamic tools library, such as a shared
4639 object for Linux or OS/X, or a DLL for Windows, which is then dynamically loaded at runtime
4640 under control of the environment variable **ACC_PROFLIB**.
- 4641 • Some implementations where the OpenACC runtime is itself implemented as a dynamic li-
4642 brary may support adding a tools library using the **LD_PRELOAD** feature in Linux.
- 4643 • A tools library may be linked with the program, as in the first option, and the application itself
4644 may directly register event callback routines, or may invoke a library initialization routine that
4645 will register the event callbacks.

4646 Callbacks are registered with the runtime by calling **acc_callback_register** for each event
4647 as described in Section 5.4 Registering Event Callbacks. The prototype for **acc_callback_register**
4648 is:

```

4649     extern void acc_callback_register
4650         (acc_event_t event_type, acc_callback cb,
4651          acc_register_t info);

```

4652 The first argument to **acc_callback_register** is the event for which a callback is being
4653 registered (compare Section 5.1 Events). The second argument is a pointer to the callback routine:

```

4654     typedef void (*acc_callback)
4655             (acc_callback_info*, acc_event_info*, acc_api_info*);

```

4656 The third argument is an enum type:

```

4657     typedef enum acc_register_t{
4658         acc_reg = 0,
4659         acc_toggle = 1,
4660         acc_toggle_per_thread = 2
4661     }acc_register_t;

```

4662 This is usually `acc_reg`, but see Section 5.4.2 Disabling and Enabling Callbacks for cases where
4663 different values are used.

4664 An example of registering callbacks for launch, upload, and download events is:

```

4665     acc_callback_register(acc_ev_enqueue_launch_start,
4666                         prof_launch, acc_reg);
4667     acc_callback_register(acc_ev_enqueue_upload_start,
4668                         prof_data, acc_reg);
4669     acc_callback_register(acc_ev_enqueue_download_start,
4670                         prof_data, acc_reg);

```

4671 As shown in this example, the same routine (`prof_data`) can be registered for multiple events.
4672 The routine can use the `event_type` field in the `acc_callback_info` structure to determine
4673 for what event it was invoked.

4674 The names `acc_prof_register` and `acc_prof_unregister` are preserved for backward
4675 compatibility with previous versions of OpenACC.

4676 5.3.1 Library Registration

4677 The OpenACC runtime will invoke `acc_register_library`, passing the addresses of the reg-
4678 istration routines `acc_callback_register` and `acc_callback_unregister`, in case
4679 that routine comes from a dynamic library. In the third argument it passes the address of the lookup
4680 routine `acc_prof_lookup` to obtain the addresses of inquiry functions. No inquiry functions
4681 are defined in this profiling interface, but we preserve this argument for future support of sampling-
4682 based tools.

4683 Typically, the OpenACC runtime will include a *weak* definition of `acc_register_library`,
4684 which does nothing and which will be called when there is no tools library. In this case, the library
4685 can save the addresses of these routines and/or make registration calls to register any appropriate
4686 callbacks. The prototype for `acc_register_library` is:

```

4687     extern void acc_register_library
4688             (acc_prof_reg reg, acc_prof_reg unreg,
4689             acc_prof_lookup_func lookup);

```

4690 The first two arguments of this routine are of type:

```

4691     typedef void (*acc_prof_reg)
4692             (acc_event_t event_type, acc_callback cb,
4693             acc_register_t info);

```

4694 The third argument passes the address to the lookup function `acc_prof_lookup` to obtain the
4695 address of interface functions. It is of type:

```
4696     typedef void (*acc_query_fn) ();
4697     typedef acc_query_fn (*acc_prof_lookup_func)
4698         (const char* acc_query_fn_name);
```

4699 The argument of the lookup function is a string with the name of the inquiry function. There are no
4700 inquiry functions defined for this interface.

4701 5.3.2 Statically-Linked Library Initialization

4702 A tools library can be compiled and linked directly into the application. If the library provides an
4703 external routine `acc_register_library` as specified in Section 5.3.1 Library Registration, the
4704 runtime will invoke that routine to initialize the library.

4705 The sequence of events is:

- 4706 1. The runtime invokes the `acc_register_library` routine from the library.
- 4707 2. The `acc_register_library` routine calls `acc_callback_register` for each event
4708 to be monitored.
- 4709 3. `acc_callback_register` records the callback routines.
- 4710 4. The program runs, and your callback routines are invoked at the appropriate events.

4711 In this mode, only one tool library is supported.

4712 5.3.3 Runtime Dynamic Library Loading

4713 A common case is to build the tools library as a dynamic library (shared object for Linux or OS/X,
4714 DLL for Windows). In that case, you can have the OpenACC runtime load the library during initial-
4715 ization. This allows you to enable runtime profiling without rebuilding or even relinking your ap-
4716 plication. The dynamic library must implement a registration routine `acc_register_library`
4717 as specified in Section 5.3.1 Library Registration.

4718 The user may set the environment variable `ACC_PROFLIB` to the path to the library will tell the
4719 OpenACC runtime to load your dynamic library at initialization time:

```
4720     Bash:
4721         export ACC_PROFLIB=/home/user/lib/myprof.so
4722         ./myapp
4723     or
4724         ACC_PROFLIB=/home/user/lib/myprof.so ./myapp
4725     C-shell:
4726         setenv ACC_PROFLIB /home/user/lib/myprof.so
4727         ./myapp
```

4728 When the OpenACC runtime initializes, it will read the `ACC_PROFLIB` environment variable (with
4729 `getenv`). The runtime will open the dynamic library (using `dlopen` or `LoadLibraryA`); if
4730 the library cannot be opened, the runtime may cause the program to halt execution and return an

4731 error status, or may continue execution with or without an error message. If the library is suc-
4732 cessfully opened, the runtime will get the address of the `acc_register_library` routine (us-
4733 ing `dlsym` or `GetProcAddress`). If this routine is resolved in the library, it will be invoked
4734 passing in the addresses of the registration routine `acc_callback_register`, the deregistra-
4735 tion routine `acc_callback_unregister`, and the lookup routine `acc_prof_lookup`. The
4736 registration routine in your library, `acc_register_library`, should register the callbacks by
4737 calling the `register` argument, and should save the addresses of the arguments (`register`,
4738 `unregister`, and `lookup`) for later use, if needed.

4739 The sequence of events is:

- 4740 1. Initialization of the OpenACC runtime.
- 4741 2. OpenACC runtime reads `ACC_PROFLIB`.
- 4742 3. OpenACC runtime loads the library.
- 4743 4. OpenACC runtime calls the `acc_register_library` routine in that library.
- 4744 5. Your `acc_register_library` routine calls `acc_callback_register` for each event
4745 to be monitored.
- 4746 6. `acc_callback_register` records the callback routines.
- 4747 7. The program runs, and your callback routines are invoked at the appropriate events.

4748 If supported, paths to multiple dynamic libraries may be specified in the `ACC_PROFLIB` environ-
4749 ment variable, separated by semicolons (;). The OpenACC runtime will open these libraries and in-
4750 voke the `acc_register_library` routine for each, in the order they appear in `ACC_PROFLIB`.

4751 5.3.4 Preloading with LD_PRELOAD

4752 The implementation may also support dynamic loading of a tools library using the `LD_PRELOAD`
4753 feature available in some systems. In such an implementation, you need only specify your tools
4754 library path in the `LD_PRELOAD` environment variable before executing your program. The Open-
4755 ACC runtime will invoke the `acc_register_library` routine in your tools library at initial-
4756 ization time. This requires that the OpenACC runtime include a dynamic library with a default
4757 (empty) implementation of `acc_register_library` that will be invoked in the normal case
4758 where there is no `LD_PRELOAD` setting. If an implementation only supports static linking, or if the
4759 application is linked without dynamic library support, this feature will not be available.

4760 Bash:

```
4761     export LD_PRELOAD=/home/user/lib/myprof.so  
4762     ./myapp
```

4763 or

```
4764     LD_PRELOAD=/home/user/lib/myprof.so ./myapp
```

4765 C-shell:

```
4766     setenv LD_PRELOAD /home/user/lib/myprof.so  
4767     ./myapp
```

4768 The sequence of events is:

- 4769 1. The operating system loader loads the library specified in `LD_PRELOAD`.

- 4770 2. The call to **acc_register_library** in the OpenACC runtime is resolved to the routine
4771 in the loaded tools library.
- 4772 3. OpenACC runtime calls the **acc_register_library** routine in that library.
- 4773 4. Your **acc_register_library** routine calls **acc_callback_register** for each event
4774 to be monitored.
- 4775 5. **acc_callback_register** records the callback routines.
- 4776 6. The program runs, and your callback routines are invoked at the appropriate events.

4777 In this mode, only a single tools library is supported, since only one **acc_register_library**
4778 initialization routine will get resolved by the dynamic loader.

4779 5.3.5 Application-Controlled Initialization

4780 An alternative to default initialization is to have the application itself call the library initialization
4781 routine, which then calls **acc_callback_register** for each appropriate event. The library
4782 may be statically linked to the application or your application may dynamically load the library.

4783 The sequence of events is:

- 4784 1. Your application calls the library initialization routine.
- 4785 2. The library initialization routine calls **acc_callback_register** for each event to be
4786 monitored.
- 4787 3. **acc_callback_register** records the callback routines.
- 4788 4. The program runs, and your callback routines are invoked at the appropriate events.

4789 In this mode, multiple tools libraries can be supported, with each library initialization routine in-
4790 voked by the application.

4791 5.4 Registering Event Callbacks

4792 This section describes how to register and unregister callbacks, temporarily disabling and enabling
4793 callbacks, the behavior of dynamic registration and unregistration, and requirements on an Open-
4794 ACC implementation to correctly support the interface.

4795 5.4.1 Event Registration and Unregistration

4796 The library must call the registration routine **acc_callback_register** to register each call-
4797 back with the runtime. A simple example:

```
4798 extern void prof_data(acc_callback_info* profinfo,
4799                     acc_event_info* eventinfo, acc_api_info* apiinfo);
4800 extern void prof_launch(acc_callback_info* profinfo,
4801                       acc_event_info* eventinfo, acc_api_info* apiinfo);
4802 ...
4803 void acc_register_library(acc_prof_reg reg,
4804                         acc_prof_reg unreg, acc_prof_lookup_func lookup) {
4805     reg(acc_ev_enqueue_upload_start, prof_data, acc_reg);
4806     reg(acc_ev_enqueue_download_start, prof_data, acc_reg);
```



```

4807     reg(acc_ev_enqueue_launch_start, prof_launch, acc_reg);
4808 }

```

4809 In this example the **prof_data** routine will be invoked for each data upload and download event,
 4810 and the **prof_launch** routine will be invoked for each launch event. The **prof_data** routine
 4811 might start out with:

```

4812 void prof_data(acc_callback_info* profinfo,
4813               acc_event_info* eventinfo, acc_api_info* apiinfo){
4814     acc_data_event_info* datainfo;
4815     datainfo = (acc_data_event_info*)eventinfo;
4816     switch( datainfo->event_type ){
4817         case acc_ev_enqueue_upload_start :
4818             ...
4819     }
4820 }

```

4821 Multiple Callbacks

4822 Multiple callback routines can be registered on the same event:

```

4823     acc_callback_register(acc_ev_enqueue_upload_start,
4824                          prof_data, acc_reg);
4825     acc_callback_register(acc_ev_enqueue_upload_start,
4826                          prof_up, acc_reg);

```

4827 For most events, the callbacks will be invoked in the order in which they are registered. However,
 4828 *end* events, named **acc_ev_..._end**, invoke callbacks in the reverse order. Essentially, each
 4829 event has an ordered list of callback routines. A new callback routine is appended to the tail of the
 4830 list for that event. For most events, that list is traversed from the head to the tail, but for *end* events,
 4831 the list is traversed from the tail to the head.

4832 If a callback is registered, then later unregistered, then later still registered again, the second regis-
 4833 tration is considered to be a new callback, and the callback routine will then be appended to the tail
 4834 of the callback list for that event.

4835 Unregistering

4836 A matching call to **acc_callback_unregister** will remove that routine from the list of call-
 4837 back routines for that event.

```

4838     acc_callback_register(acc_ev_enqueue_upload_start,
4839                          prof_data, acc_reg);
4840     // prof_data is on the callback list for acc_ev_enqueue_upload_start
4841     ...
4842     acc_callback_unregister(acc_ev_enqueue_upload_start,
4843                            prof_data, acc_reg);
4844     // prof_data is removed from the callback list
4845     // for acc_ev_enqueue_upload_start

```

4846 Each entry on the callback list must also have a *ref* count. This keeps track of how many times
 4847 this routine was added to this event's callback list. If a routine is registered *n* times, it must be

4848 unregistered n times before it is removed from the list. Note that if a routine is registered multiple
 4849 times for the same event, its *ref* count will be incremented with each registration, but it will only be
 4850 invoked once for each event instance.

4851 5.4.2 Disabling and Enabling Callbacks

4852 A callback routine may be temporarily disabled on the callback list for an event, then later re-
 4853 enabled. The behavior is slightly different than unregistering and later re-registering that event.
 4854 When a routine is disabled and later re-enabled, the routine's position on the callback list for that
 4855 event is preserved. When a routine is unregistered and later re-registered, the routine's position on
 4856 the callback list for that event will move to the tail of the list. Also, unregistering a callback must be
 4857 done n times if the callback routine was registered n times. In contrast, disabling, and enabling an
 4858 event sets a toggle. Disabling a callback will immediately reset the toggle and disable calls to that
 4859 routine for that event, even if it was enabled multiple times. Enabling a callback will immediately
 4860 set the toggle and enable calls to that routine for that event, even if it was disabled multiple times.
 4861 Registering a new callback initially sets the toggle.

4862 A call to `acc_callback_unregister` with a value of `acc_toggle` as the third argument
 4863 will disable callbacks to the given routine. A call to `acc_callback_register` with a value of
 4864 `acc_toggle` as the third argument will enable those callbacks.

```
4865     acc_callback_unregister(acc_ev_enqueue_upload_start,  
4866                           prof_data, acc_toggle);  
4867     // prof_data is disabled  
4868     ...  
4869     acc_callback_register(acc_ev_enqueue_upload_start,  
4870                          prof_data, acc_toggle);  
4871     // prof_data is re-enabled
```

4872 A call to either `acc_callback_unregister` or `acc_callback_register` to disable or
 4873 enable a callback when that callback is not currently registered for that event will be ignored with
 4874 no error.

4875 All callbacks for an event may be disabled (and re-enabled) by passing `NULL` to the second argument
 4876 and `acc_toggle` to the third argument of `acc_callback_unregister` (and
 4877 `acc_callback_register`). This sets a toggle for that event, which is distinct from the toggle
 4878 for each callback for that event. While the event is disabled, no callbacks for that event will be
 4879 invoked. Callbacks for that event can be registered, unregistered, enabled, and disabled while that
 4880 event is disabled, but no callbacks will be invoked for that event until the event itself is enabled.
 4881 Initially, all events are enabled.

```
4882     acc_callback_unregister(acc_ev_enqueue_upload_start,  
4883                           prof_data, acc_toggle);  
4884     // prof_data is disabled  
4885     ...  
4886     acc_callback_unregister(acc_ev_enqueue_upload_start,  
4887                            NULL, acc_toggle);  
4888     // acc_ev_enqueue_upload_start callbacks are disabled  
4889     ...  
4890     acc_callback_register(acc_ev_enqueue_upload_start,
```

```

4891         prof_data, acc_toggle);
4892 // prof_data is re-enabled, but
4893 // acc_ev_enqueue_upload_start callbacks still disabled
4894 ...
4895 acc_callback_register(acc_ev_enqueue_upload_start,
4896                     prof_up, acc_reg);
4897 // prof_up is registered and initially enabled, but
4898 // acc_ev_enqueue_upload_start callbacks still disabled
4899 ...
4900 acc_callback_register(acc_ev_enqueue_upload_start,
4901                     NULL, acc_toggle);
4902 // acc_ev_enqueue_upload_start callbacks are enabled
4903

```

4904 Finally, all callbacks can be disabled (and enabled) by passing the argument list (**acc_ev_none**,
4905 **NULL**, **acc_toggle**) to **acc_callback_unregister** (and **acc_callback_register**).
4906 This sets a global toggle disabling all callbacks, which is distinct from the toggle enabling callbacks
4907 for each event and the toggle enabling each callback routine.

4908 The behavior of passing **acc_ev_none** as the first argument and a non-**NULL** value as the second
4909 argument to **acc_callback_unregister** or **acc_callback_register** is not defined,
4910 and may be ignored by the runtime without error.

4911 All callbacks can be disabled (or enabled) for just the current thread by passing the argument list
4912 (**acc_ev_none**, **NULL**, **acc_toggle_per_thread**) to **acc_callback_unregister**
4913 (and **acc_callback_register**). This is the only thread-specific interface to
4914 **acc_callback_register** and **acc_callback_unregister**, all other calls to register,
4915 unregister, enable, or disable callbacks affect all threads in the application.

4916 5.5 Advanced Topics

4917 This section describes advanced topics such as dynamic registration and changes of the execution
4918 state for callback routines as well as the runtime and tool behavior for multiple host threads.

4919 5.5.1 Dynamic Behavior

4920 Callback routines may be registered or unregistered, enabled or disabled at any point in the execution
4921 of the program. Calls may appear in the library itself, during the processing of an event. The
4922 OpenACC runtime must allow for this case, where the callback list for an event is modified while
4923 that event is being processed.

4924 Dynamic Registration and Unregistration

4925 Calls to **acc_register** and **acc_unregister** may occur at any point in the application. A
4926 callback routine can be registered or unregistered from a callback routine, either the same routine
4927 or another routine, for a different event or the same event for which the callback was invoked. If a
4928 callback routine is registered for an event while that event is being processed, then the new callback
4929 routine will be added to the tail of the list of callback routines for this event. Some events (the
4930 **_end**) events process the callback routines in reverse order, from the tail to the head. For those
4931 events, adding a new callback routine will not cause the new routine to be invoked for this instance

4932 of the event. The other events process the callback routines in registration order, from the head
4933 to the tail. Adding a new callback routine for such an event will cause the runtime to invoke that
4934 newly registered callback routine for this instance of the event. Both the runtime and the library
4935 must implement and expect this behavior.

4936 If an existing callback routine is unregistered for an event while that event is being processed, that
4937 callback routine is removed from the list of callbacks for this event. For any event, if that callback
4938 routine had not yet been invoked for this instance of the event, it will not be invoked.

4939 Registering and unregistering a callback routine is a global operation and affects all threads, in a
4940 multithreaded application. See Section 5.4.1 Multiple Callbacks.

4941 **Dynamic Enabling and Disabling**

4942 Calls to **acc_register** and **acc_unregister** to enable and disable a specific callback for
4943 an event, enable or disable all callbacks for an event, or enable or disable all callbacks may occur
4944 at any point in the application. A callback routine can be enabled or disabled from a callback
4945 routine, either the same routine or another routine, for a different event or the same event for which
4946 the callback was invoked. If a callback routine is enabled for an event while that event is being
4947 processed, then the new callback routine will be immediately enabled. If it appears on the list of
4948 callback routines closer to the head (for **_end** events) or closer to the tail (for other events), that
4949 newly-enabled callback routine will be invoked for this instance of this event, unless it is disabled
4950 or unregistered before that callback is reached.

4951 If a callback routine is disabled for an event while that event is being processed, that callback routine
4952 is immediately disabled. For any event, if that callback routine had not yet been invoked for this in-
4953 stance of the event, it will not be invoked, unless it is enabled before that callback routine is reached
4954 in the list of callbacks for this event. If all callbacks for an event are disabled while that event is
4955 being processed, or all callbacks are disabled for all events while an event is being processed, then
4956 when this callback routine returns, no more callbacks will be invoked for this instance of the event.

4957 Registering and unregistering a callback routine is a global operation and affects all threads, in a
4958 multithreaded application. See Section 5.4.1 Multiple Callbacks.

4959 **5.5.2 OpenACC Events During Event Processing**

4960 OpenACC events may occur during event processing. This may be because of OpenACC API rou-
4961 tine calls or OpenACC constructs being reached during event processing, or because of multiple host
4962 threads executing asynchronously. Both the OpenACC runtime and the tool library must implement
4963 the proper behavior.

4964 **5.5.3 Multiple Host Threads**

4965 Many programs that use OpenACC also use multiple host threads, such as programs using the
4966 OpenMP API. The appearance of multiple host threads affects both the OpenACC runtime and the
4967 tools library.

4968 **Runtime Support for Multiple Threads**

4969 The OpenACC runtime must be thread-safe, and the OpenACC runtime implementation of this
4970 tools interface must also be thread-safe. All threads use the same set of callbacks for all events, so

4971 registering a callback from one thread will cause all threads to execute that callback. This means that
4972 managing the callback lists for each event must be protected from multiple simultaneous updates.
4973 This includes adding a callback to the tail of the callback list for an event, removing a callback from
4974 the list for an event, and incrementing or decrementing the *ref* count for a callback routine for an
4975 event.

4976 In addition, one thread may register, unregister, enable, or disable a callback for an event while
4977 another thread is processing the callback list for that event asynchronously. The exact behavior may
4978 be dependent on the implementation, but some behaviors are expected and others are disallowed.
4979 In the following examples, there are three callbacks, A, B, and C, registered for event E in that
4980 order, where callbacks A and B are enabled and callback C is temporarily disabled. Thread T1 is
4981 dynamically modifying the callbacks for event E while thread T2 is processing an instance of event
4982 E.

- 4983 • Suppose thread T1 unregisters or disables callback A for event E. Thread T2 may or may not
4984 invoke callback A for this event instance, but it must invoke callback B; if it invokes callback
4985 A, that must precede the invocation of callback B.
- 4986 • Suppose thread T1 unregisters or disables callback B for event E. Thread T2 may or may not
4987 invoke callback B for this event instance, but it must invoke callback A; if it invokes callback
4988 B, that must follow the invocation of callback A.
- 4989 • Suppose thread T1 unregisters or disables callback A and then unregisters or disables callback
4990 B for event E. Thread T2 may or may not invoke callback A and may or may not invoke
4991 callback B for this event instance, but if it invokes both callbacks, it must invoke callback A
4992 before it invokes callback B.
- 4993 • Suppose thread T1 unregisters or disables callback B and then unregisters or disables callback
4994 A for event E. Thread T2 may or may not invoke callback A and may or may not invoke
4995 callback B for this event instance, but if it invokes callback B, it must have invoked callback
4996 A for this event instance.
- 4997 • Suppose thread T1 is registering a new callback D for event E. Thread T2 may or may not
4998 invoke callback D for this event instance, but it must invoke both callbacks A and B. If it
4999 invokes callback D, that must follow the invocations of A and B.
- 5000 • Suppose thread T1 is enabling callback C for event E. Thread T2 may or may not invoke
5001 callback C for this event instance, but it must invoke both callbacks A and B. If it invokes
5002 callback C, that must follow the invocations of A and B.

5003 The `acc_callback_info` struct has a `thread_id` field, which the runtime must set to a
5004 unique value for each host thread, though it need not be the same as the OpenMP threadnum value.

5005 Library Support for Multiple Threads

5006 The tool library must also be thread-safe. The callback routine will be invoked in the context of the
5007 thread that reaches the event. The library may receive a callback from a thread T2 while it's still
5008 processing a callback, from the same event type or from a different event type, from another thread
5009 T1. The `acc_callback_info` struct has a `thread_id` field, which the runtime must set to a
5010 unique value for each host thread.

5011 If the tool library uses dynamic callback registration and unregistration, or callback disabling and
5012 enabling, recall that unregistering or disabling an event callback from one thread will unregister or

5013 disable that callback for all threads, and registering or enabling an event callback from any thread
5014 will register or enable it for all threads. If two or more threads register the same callback for the
5015 same event, the behavior is the same as if one thread registered that callback multiple times; see
5016 Section 5.4.1 Multiple Callbacks. The **acc_unregister** routine must be called as many times
5017 as **acc_register** for that callback/event pair in order to totally unregister it. If two threads
5018 register two different callback routines for the same event, unless the order of the registration calls
5019 is guaranteed by some synchronization method, the order in which the runtime sees the registration
5020 may differ for multiple runs, meaning the order in which the callbacks occur will differ as well.

6. Glossary

5021

5022 Clear and consistent terminology is important in describing any programming model. We define
5023 here the terms you must understand in order to make effective use of this document and the asso-
5024 ciated programming model. In particular, some terms used in this specification conflict with their
5025 usage in the base language specifications. When there is potential confusion, the term will appear
5026 here.

5027 **Accelerator** – a device attached to a CPU and to which the CPU can offload data and compute
5028 kernels to perform compute-intensive calculations.

5029 **Accelerator routine** – a C or C++ function or Fortran subprogram compiled for the accelerator
5030 with the **routine** directive.

5031 **Accelerator thread** – a thread of execution that executes on the accelerator; a single vector lane of
5032 a single worker of a single gang.

5033 **Aggregate datatype** – any non-scalar datatype such as array and composite datatypes. In Fortran,
5034 aggregate datatypes include arrays, derived types, character types. In C, aggregate datatypes include
5035 arrays, targets of pointers, structs, and unions. In C++, aggregate datatypes include arrays, targets
5036 of pointers, classes, structs, and unions.

5037 **Aggregate variables** – a variable of any non-scalar datatype, including array or composite variables.
5038 In Fortran, this includes any variable with allocatable or pointer attribute and character variables.

5039 **Async-argument** – an *async-argument* is a nonnegative scalar integer expression (*int* for C or C++,
5040 *integer* for Fortran), or one of the special values **acc_async_noval** or **acc_async_sync**.

5041 **Barrier** – a type of synchronization where all parallel execution units or threads must reach the
5042 barrier before any execution unit or thread is allowed to proceed beyond the barrier; modeled after
5043 the starting barrier on a horse race track.

5044 **Block construct** – a *block-construct*, as specified by the Fortran language.

5045 **Composite datatype** – a derived type in Fortran, or a **struct** or **union** type in C, or a **class**,
5046 **struct**, or **union** type in C++. (This is different from the use of the term *composite data type* in
5047 the C and C++ languages.)

5048 **Composite variable** – a variable of composite datatype. In Fortran, a composite variable must not
5049 have allocatable or pointer attributes.

5050 **Compute construct** – a *parallel construct*, *serial construct*, or *kernels construct*.

5051 **Compute intensity** – for a given loop, region, or program unit, the ratio of the number of arithmetic
5052 operations performed on computed data divided by the number of memory transfers required to
5053 move that data between two levels of a memory hierarchy.

5054 **Compute region** – a *parallel region*, *serial region*, or *kernels region*.

5055 **Construct** – a directive and the associated statement, loop, or structured block, if any.

5056 **CUDA** – the CUDA environment from NVIDIA, a C-like programming environment used to ex-
5057 plicitly control and program an NVIDIA GPU.

- 5058 **Current device** – the device represented by the *acc-current-device-type-var* and *acc-current-device-*
5059 *num-var* ICVs
- 5060 **Current device type** – the device type represented by the *acc-current-device-type-var* ICV
- 5061 **Data lifetime** – the lifetime of a data object in device memory, which may begin at the entry to
5062 a data region, or at an **enter data** directive, or at a data API call such as **acc_copyin** or
5063 **acc_create**, and which may end at the exit from a data region, or at an **exit data** directive,
5064 or at a data API call such as **acc_delete**, **acc_copyout**, or **acc_shutdown**, or at the end of
5065 the program execution.
- 5066 **Data region** – a *region* defined by a **data** construct, or an implicit data region for a function or
5067 subroutine containing OpenACC directives. Data constructs typically allocate device memory and
5068 copy data from host to device memory upon entry, and copy data from device to local memory and
5069 deallocate device memory upon exit. Data regions may contain other data regions and compute
5070 regions.
- 5071 **Default asynchronous queue** – the asynchronous activity queue represented in the *acc-default-*
5072 *async-var* ICV
- 5073 **Device** – a general reference to an accelerator or a multicore CPU.
- 5074 **Device memory** – memory attached to a device, logically and physically separate from the host
5075 memory.
- 5076 **Device thread** – a thread of execution that executes on any device.
- 5077 **Directive** – in C or C++, a **#pragma**, or in Fortran, a specially formatted comment statement, that
5078 is interpreted by a compiler to augment information about or specify the behavior of the program.
- 5079 **Discrete memory** – memory accessible from the local thread that is not accessible from the current
5080 device, or memory accessible from the current device that is not accessible from the local thread.
- 5081 **DMA** – Direct Memory Access, a method to move data between physically separate memories;
5082 this is typically performed by a DMA engine, separate from the host CPU, that can access the host
5083 physical memory as well as an IO device or other physical memory.
- 5084 **Exposed variable access** – with respect to a compute construct, any access to the data or address
5085 of a variable at a point within the compute construct where the variable is not private to a scope
5086 lexically enclosed within the compute construct. See Section 2.6.2.
- 5087 **false** – a condition that evaluates to zero in C or C++, or **.false.** in Fortran.
- 5088 **GPU** – a Graphics Processing Unit; one type of accelerator.
- 5089 **GPGPU** – General Purpose computation on Graphics Processing Units.
- 5090 **Host** – the main CPU that in this context may have one or more attached accelerators. The host
5091 CPU controls the program regions and data loaded into and executed on one or more devices.
- 5092 **Host thread** – a thread of execution that executes on the host.
- 5093 **Implicit data region** – the data region that is implicitly defined for a Fortran subprogram or C
5094 function. A call to a subprogram or function enters the implicit data region, and a return from the
5095 subprogram or function exits the implicit data region.

- 5096 **Kernel** – a nested loop executed in parallel by the accelerator. Typically the loops are divided into
5097 a parallel domain, and the body of the loop becomes the body of the kernel.
- 5098 **Kernels region** – a *region* defined by a **kernels** construct. A kernels region is a structured block
5099 which is compiled for the accelerator. The code in the kernels region will be divided by the compiler
5100 into a sequence of kernels; typically each loop nest will become a single kernel. A kernels region
5101 may require space in device memory to be allocated and data to be copied from local memory to
5102 device memory upon region entry, and data to be copied from device memory to local memory and
5103 space in device memory to be deallocated upon exit.
- 5104 **Level of parallelism** – a possible level of parallelism, which in OpenACC is gang, worker, vector,
5105 or sequential. One or more of gang, worker, and vector parallelism may appear on a loop con-
5106 struct. Sequential execution corresponds to no parallelism. The **gang, worker, vector,** and
5107 **seq** clauses specify the level of parallelism for a loop.
- 5108 **Local device** – the device where the *local thread* executes.
- 5109 **Local memory** – the memory associated with the *local thread*.
- 5110 **Local thread** – the host thread or the accelerator thread that executes an OpenACC directive or
5111 construct.
- 5112 **Loop trip count** – the number of times a particular loop executes.
- 5113 **MIMD** – a method of parallel execution (Multiple Instruction, Multiple Data) where different exe-
5114 cution units or threads execute different instruction streams asynchronously with each other.
- 5115 **null pointer** – a C or C++ pointer variable with the value zero, **NULL**, or (in C++) **nullptr**, or a
5116 Fortran **pointer** variable that is not associated, or a Fortran **allocatable** variable that is not
5117 allocated.
- 5118 **OpenCL** – short for Open Compute Language, a developing, portable standard C-like programming
5119 environment that enables low-level general-purpose programming on GPUs and other accelerators.
- 5120 **Orphaned loop construct** - a **loop** construct that is not lexically contained in any compute con-
5121 struct, that is, that has no parent compute construct.
- 5122 **Parallel region** – a *region* defined by a **parallel** construct. A parallel region is a structured block
5123 which is compiled for the accelerator. A parallel region typically contains one or more work-sharing
5124 loops. A parallel region may require space in device memory to be allocated and data to be copied
5125 from local memory to device memory upon region entry, and data to be copied from device memory
5126 to local memory and space in device memory to be deallocated upon exit.
- 5127 **Parent compute construct** – for a **loop** construct, the **parallel, serial,** or **kernels** con-
5128 struct that lexically contains the **loop** construct and is the innermost compute construct that con-
5129 tains that **loop** construct, if any.
- 5130 **Partly present data** – a section of data for which some of the data is present in a single device
5131 memory section, but part of the data is either not present or is present in a different device memory
5132 section. For instance, if a subarray of an array is present, the array is partly present.
- 5133 **Present data** – data for which the sum of the structured and dynamic reference counters is greater
5134 than zero in a single device memory section; see Section 2.6.7. A null pointer is defined as always
5135 present with a length of zero bytes.

- 5136 **Private data** – with respect to an iterative loop, data which is used only during a particular loop
5137 iteration. With respect to a more general region of code, data which is used within the region but is
5138 not initialized prior to the region and is re-initialized prior to any use after the region.
- 5139 **Procedure** – in C or C++, a function in the program; in Fortran, a subroutine or function.
- 5140 **Region** – all the code encountered during an instance of execution of a construct. A region includes
5141 any code in called routines, and may be thought of as the dynamic extent of a construct. This may
5142 be a *parallel region*, *serial region*, *kernels region*, *data region*, or *implicit data region*.
- 5143 **Scalar** – a variable of scalar datatype. In Fortran, scalars must not have allocatable or pointer
5144 attributes.
- 5145 **Scalar datatype** – an intrinsic or built-in datatype that is not an array or aggregate datatype. In For-
5146 tran, scalar datatypes are integer, real, double precision, complex, or logical. In C, scalar datatypes
5147 are char (signed or unsigned), int (signed or unsigned, with optional short, long or long long at-
5148 tribute), enum, float, double, long double, `_Complex` (with optional float or long attribute), or any
5149 pointer datatype. In C++, scalar datatypes are char (signed or unsigned), `wchar_t`, int (signed or
5150 unsigned, with optional short, long or long long attribute), enum, bool, float, double, long double,
5151 or any pointer datatype. Not all implementations or targets will support all of these datatypes.
- 5152 **Serial region** – a *region* defined by a **serial** construct. A serial region is a structured block which
5153 is compiled for the accelerator. A serial region contains code that is executed by a single gang of a
5154 single worker with a vector length of one. A serial region may require space in device memory to be
5155 allocated and data to be copied from local memory to device memory upon region entry, and data
5156 to be copied from device memory to local memory and space in device memory to be deallocated
5157 upon exit.
- 5158 **Shared memory** – memory that is accessible from both the local thread and the current device.
- 5159 **SIMD** – a method of parallel execution (single-instruction, multiple-data) where the same instruc-
5160 tion is applied to multiple data elements simultaneously.
- 5161 **SIMD operation** – a *vector operation* implemented with SIMD instructions.
- 5162 **Structured block** – in C or C++, an executable statement, possibly compound, with a single entry
5163 at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single
5164 entry at the top and a single exit at the bottom.
- 5165 **Thread** – a host CPU thread or an accelerator thread. On a host CPU, a thread is defined by a
5166 program counter and stack location; several host threads may comprise a process and share host
5167 memory. On an accelerator, a thread is any one vector lane of one worker of one gang.
- 5168 **true** – a condition that evaluates to nonzero in C or C++, or `.true.` in Fortran.
- 5169 **var** – the name of a variable (scalar, array, or composite variable), or a subarray specification, or an
5170 array element, or a composite variable member, or the name of a Fortran common block between
5171 slashes.
- 5172 **Vector operation** – a single operation or sequence of operations applied uniformly to each element
5173 of an array.
- 5174 **Visible data clause** – with respect to a compute construct, any data clause on the compute construct,
5175 a lexically containing **data** construct, or a visible **declare** directive. See Section 2.6.2.

- 5176 **Visible default clause** – with respect to a compute construct, the nearest **default** clause ap-
5177 pearing on the compute construct or a lexically containing **data** construct. See Section 2.6.2.
- 5178 **Visible device copy** – a copy of a variable, array, or subarray allocated in device memory that is
5179 visible to the program unit being compiled.

5180 **A. Recommendations for Implementers**

5181 This section gives recommendations for standard names and extensions to use for implementations
5182 for specific targets and target platforms, to promote portability across such implementations, and
5183 recommended options that programmers find useful. While this appendix is not part of the Open-
5184 ACC specification, implementations that provide the functionality specified herein are strongly rec-
5185 ommended to use the names in this section. The first subsection describes devices, such as NVIDIA
5186 GPUs. The second subsection describes additional API routines for target platforms, such as CUDA
5187 and OpenCL. The third subsection lists several recommended options for implementations.

5188 **A.1 Target Devices**

5189 **A.1.1 NVIDIA GPU Targets**

5190 This section gives recommendations for implementations that target NVIDIA GPU devices.

5191 **Accelerator Device Type**

5192 These implementations should use the name **acc_device_nvidia** for the **acc_device_t**
5193 type or return values from OpenACC Runtime API routines.

5194 **ACC_DEVICE_TYPE**

5195 An implementation should use the case-insensitive name **nvidia** for the environment variable
5196 **ACC_DEVICE_TYPE**.

5197 **device_type clause argument**

5198 An implementation should use the case-insensitive name **nvidia** as the argument to the **device_type**
5199 clause.

5200 **A.1.2 AMD GPU Targets**

5201 This section gives recommendations for implementations that target AMD GPUs.

5202 **Accelerator Device Type**

5203 These implementations should use the name **acc_device_radeon** for the **acc_device_t**
5204 type or return values from OpenACC Runtime API routines.

5205 **ACC_DEVICE_TYPE**

5206 These implementations should use the case-insensitive name **radeon** for the environment variable
5207 **ACC_DEVICE_TYPE**.

5208 **device_type clause argument**

5209 An implementation should use the case-insensitive name **radeon** as the argument to the **device_type**
5210 clause.

5211 **A.1.3 Multicore Host CPU Target**

5212 This section gives recommendations for implementations that target the multicore host CPU.

5213 **Accelerator Device Type**

5214 These implementations should use the name **acc_device_host** for the **acc_device_t** type
5215 or return values from OpenACC Runtime API routines.

5216 **ACC_DEVICE_TYPE**

5217 These implementations should use the case-insensitive name **host** for the environment variable
5218 **ACC_DEVICE_TYPE**.

5219 **device_type clause argument**

5220 An implementation should use the case-insensitive name **host** as the argument to the **device_type**
5221 clause.

5222 **A.2 API Routines for Target Platforms**

5223 These runtime routines allow access to the interface between the OpenACC runtime API and the
5224 underlying target platform. An implementation may not implement all these routines, but if it
5225 provides this functionality, it should use these function names.

5226 **A.2.1 NVIDIA CUDA Platform**

5227 This section gives runtime API routines for implementations that target the NVIDIA CUDA Run-
5228 time or Driver API.

5229 **acc_get_current_cuda_device**

5230 **Summary**

5231 The **acc_get_current_cuda_device** routine returns the NVIDIA CUDA device handle for
5232 the current device.

5233 **Format**

5234 C or C++:

```
5235     void* acc_get_current_cuda_device ();
```

5236 **acc_get_current_cuda_context**

5237 **Summary**

5238 The **acc_get_current_cuda_context** routine returns the NVIDIA CUDA context handle
5239 in use for the current device.

5240 **Format**

5241 C or C++:

```
5242     void* acc_get_current_cuda_context ();
```

5243 acc_get_cuda_stream**5244 Summary**

5245 The **acc_get_cuda_stream** routine returns the NVIDIA CUDA stream handle in use for the
5246 current device for the asynchronous activity queue associated with the **async** argument. This
5247 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5248 Format

5249 C or C++:

```
5250     void* acc_get_cuda_stream ( int async );
```

5251 acc_set_cuda_stream**5252 Summary**

5253 The **acc_set_cuda_stream** routine sets the NVIDIA CUDA stream handle the current device
5254 for the asynchronous activity queue associated with the **async** argument. This argument must be
5255 an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5256 Format

5257 C or C++:

```
5258     void acc_set_cuda_stream ( int async, void* stream );
```

5259 A.2.2 OpenCL Target Platform

5260 This section gives runtime API routines for implementations that target the OpenCL API on any
5261 device.

5262 acc_get_current_opengl_device**5263 Summary**

5264 The **acc_get_current_opengl_device** routine returns the OpenCL device handle for the
5265 current device.

5266 Format

5267 C or C++:

```
5268     void* acc_get_current_opengl_device ();
```

5269 acc_get_current_opengl_context**5270 Summary**

5271 The **acc_get_current_opengl_context** routine returns the OpenCL context handle in use
5272 for the current device.

5273 Format

5274 C or C++:

```
5275     void* acc_get_current_opengl_context ();
```

5276 acc_get_opengl_queue**5277 Summary**

5278 The **acc_get_opengl_queue** routine returns the OpenCL command queue handle in use for
5279 the current device for the asynchronous activity queue associated with the **async** argument. This
5280 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5281 **Format**

5282 C or C++:

5283 `cl_command_queue acc_get_opengl_queue (int async);`5284 **acc_set_opengl_queue**5285 **Summary**

5286 The `acc_set_opengl_queue` routine returns the OpenCL command queue handle in use for
 5287 the current device for the asynchronous activity queue associated with the `async` argument. This
 5288 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5289 **Format**

5290 C or C++:

5291 `void acc_set_opengl_queue (int async, cl_command_queue cmdqueue`
 5292 `);`

5293 **A.3 Recommended Options**

5294 The following options are recommended for implementations; for instance, these may be imple-
 5295 mented as command-line options to a compiler or settings in an IDE.

5296 **A.3.1 C Pointer in Present clause**

5297 This revision of OpenACC clarifies the construct:

```
5298 void test(int n ){
5299 float* p;
5300 ...
5301 #pragma acc data present(p)
5302 {
5303     // code here...
5304 }
```

5305 This example tests whether the pointer `p` itself is present in the current device memory. Implemen-
 5306 tations before this revision commonly implemented this by testing whether the pointer target `p[0]`
 5307 was present in the current device memory, and this appears in many programs assuming such. Until
 5308 such programs are modified to comply with this revision, an option to implement `present(p)` as
 5309 `present(p[0])` for C pointers may be helpful to users.

5310 **A.3.2 Automatic Data Attributes**

5311 Some implementations provide autoscoping or other analysis to automatically determine a variable's
 5312 data attributes, including the addition of reduction, private, and firstprivate clauses. To promote
 5313 program portability across implementations, it would be helpful to provide an option to disable
 5314 the automatic determination of data attributes or report which variables' data attributes are not as
 5315 defined in Section 2.6.

Index

- 5316 `_OPENACC`, 28, 125
- 5317 `acc-current-device-num-var`, 28
- 5318 `acc-current-device-type-var`, 28
- 5319 `acc-default-async-var`, 28, 83
- 5320 `acc_async_noval`, 83
- 5321 `acc_async_sync`, 83
- 5322 `acc_device_host`, 150
- 5323 `ACC_DEVICE_NUM`, 28, 117
- 5324 `acc_device_nvidia`, 149
- 5325 `acc_device_radeon`, 149
- 5326 `ACC_DEVICE_TYPE`, 28, 117, 149, 150
- 5327 `ACC_PROFLIB`, 117
- 5328 action
 - 5329 `attach`, 44, 48
 - 5330 `copyin`, 47
 - 5331 `copyout`, 48
 - 5332 `create`, 47
 - 5333 `delete`, 48
 - 5334 `detach`, 44, 49
 - 5335 `immediate`, 49
 - 5336 `present decrement`, 47
 - 5337 `present increment`, 46
- 5338 AMD GPU target, 149
- 5339 `async` clause, 41, 43, 79, 84
- 5340 `async queue`, 11
- 5341 `async-argument`, 84
- 5342 asynchronous execution, 11, 83
- 5343 `atomic` construct, 67
- 5344 `attach` action, 44, 48
- 5345 `attach` clause, 54
- 5346 attachment counter, 44
- 5347 `auto` clause, 60
- 5348 `portability`, 58
- 5349 `autoscopying`, 152

- 5350 barrier synchronization, 10, 31, 33, 143
- 5351 `bind` clause, 82
- 5352 `block` construct, 143

- 5353 `cache` directive, 65
- 5354 `capture` clause, 70
- 5355 `collapse` clause, 57
- 5356 common block, 44, 71, 72, 83
- 5357 `compute` construct, 143
- 5358 `compute region`, 143

- 5359 `construct`, 143
- 5360 `atomic`, 67
- 5361 `compute`, 143
- 5362 `data`, 40, 44
- 5363 `host_data`, 54
- 5364 `kernels`, 32, 44
- 5365 `kernels loop`, 65
- 5366 `parallel`, 30, 44
- 5367 `parallel loop`, 65
- 5368 `serial`, 32, 44
- 5369 `serial loop`, 65
- 5370 `copy` clause, 38, 50
- 5371 `copyin` action, 47
- 5372 `copyin` clause, 51
- 5373 `copyout` action, 48
- 5374 `copyout` clause, 51
- 5375 `create` action, 47
- 5376 `create` clause, 52, 72
- 5377 CUDA, 11, 12, 143, 149, 150

- 5378 data attribute
 - 5379 `explicitly determined`, 37
 - 5380 `implicitly determined`, 37
 - 5381 `predetermined`, 37
- 5382 data clause, 44
- 5383 `visible`, 37, 146
- 5384 `data` construct, 40, 44
- 5385 data lifetime, 144
- 5386 data region, 39, 144
- 5387 `implicit`, 39
- 5388 data-independent `loop` construct, 56
- 5389 `declare` directive, 71
- 5390 `default` clause, 36, 41
- 5391 `visible`, 37, 147
- 5392 `default (none)` clause, 38
- 5393 `default(present)`, 38
- 5394 `delete` action, 48
- 5395 `delete` clause, 53
- 5396 `detach` action, 44, 49
- 5397 `immediate`, 49
- 5398 `detach` clause, 54
- 5399 `device` clause, 78
- 5400 `device_resident` clause, 72
- 5401 `device_type` clause, 29, 45, 149, 150
- 5402 `deviceptr` clause, 44, 50
- 5403 direct memory access, 11, 144

- 5404 DMA, 11, 144
- 5405 **enter data** directive, 41, 44
- 5406 environment variable
- 5407 **_OPENACC**, 28
- 5408 **ACC_DEVICE_NUM**, 28, 117
- 5409 **ACC_DEVICE_TYPE**, 28, 117, 149, 150
- 5410 **ACC_PROFLIB**, 117
- 5411 **exit data** directive, 41, 44
- 5412 explicitly determined data attribute, 37
- 5413 exposed variable access, 38, 144
- 5414 **firstprivate** clause, 35, 38
- 5415 gang, 31
- 5416 **gang** clause, 57, 81
- 5417 implicit, 58
- 5418 gang parallelism, 10
- 5419 *gang-arg*, 56
- 5420 gang-partitioned mode, 10
- 5421 optimizations, 58
- 5422 gang-redundant mode, 10, 31
- 5423 GP mode, 10
- 5424 GR mode, 10
- 5425 **host**, 150
- 5426 **host** clause, 78
- 5427 **host_data** construct, 54
- 5428 ICV, 28
- 5429 **if** clause, 41, 43, 74, 76, 77, 79, 86
- 5430 immediate detach action, 49
- 5431 implicit data region, 39
- 5432 implicit **gang** clause, 58
- 5433 implicitly determined data attribute, 37
- 5434 **independent** clause, 60
- 5435 **init** directive, 74
- 5436 internal control variable, 28
- 5437 **kernels** construct, 32, 44
- 5438 **kernels loop** construct, 65
- 5439 level of parallelism, 10, 145
- 5440 **link** clause, 45, 73
- 5441 local device, 11
- 5442 local memory, 11
- 5443 local thread, 11
- 5444 **loop** construct, 55
- 5445 data-independent, 56
- 5446 orphaned, 56
- 5447 sequential, 56
- 5448 **no_create** clause, 53
- 5449 **nohost** clause, 82
- 5450 **num_gangs** clause, 34
- 5451 **num_workers** clause, 35
- 5452 **nvidia**, 149
- 5453 NVIDIA GPU target, 149
- 5454 OpenCL, 11, 12, 145, 149, 151
- 5455 optimizations
- 5456 gang-partitioned mode, 58
- 5457 orphaned **loop** construct, 56
- 5458 **parallel** construct, 30, 44
- 5459 **parallel loop** construct, 65
- 5460 parallelism
- 5461 level, 10, 145
- 5462 parent compute construct, 56
- 5463 portability
- 5464 **auto** clause, 58
- 5465 predetermined data attribute, 37
- 5466 **present** clause, 38, 44, 50
- 5467 present decrement action, 47
- 5468 present increment action, 46
- 5469 **private** clause, 35, 61
- 5470 **radeon**, 149
- 5471 **read** clause, 70
- 5472 **reduction** clause, 35, 61
- 5473 reference counter, 43
- 5474 region
- 5475 compute, 143
- 5476 data, 39, 144
- 5477 implicit data, 39
- 5478 **routine** directive, 80
- 5479 **self** clause, 78
- 5480 sentinel, 27
- 5481 **seq** clause, 59, 82
- 5482 sequential **loop** construct, 56
- 5483 **serial** construct, 32, 44
- 5484 **serial loop** construct, 65
- 5485 **shutdown** directive, 75
- 5486 *size-expr*, 56
- 5487 structured-block, 146
- 5488 thread, 146
- 5489 **tile** clause, 60

- 5490 **update** clause, 70
- 5491 **update** directive, 78
- 5492 **use_device** clause, 55

- 5493 **vector** clause, 59, 82
- 5494 vector lane, 31
- 5495 vector parallelism, 10
- 5496 vector-partitioned mode, 10
- 5497 vector-single mode, 10
- 5498 **vector_length** clause, 35
- 5499 visible data clause, 37, 146
- 5500 visible **default** clause, 37, 147
- 5501 visible device copy, 147
- 5502 VP mode, 10
- 5503 VS mode, 10

- 5504 **wait** clause, 41, 43, 79, 85
- 5505 **wait** directive, 85
- 5506 worker, 31
- 5507 **worker** clause, 59, 81
- 5508 worker parallelism, 10
- 5509 worker-partitioned mode, 10
- 5510 worker-single mode, 10
- 5511 WP mode, 10
- 5512 WS mode, 10