

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 © 2011-2024 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	13
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	15
21	1.9. Changes from Version 1.0 to 2.0	16
22	1.10. Corrections in the August 2013 document	18
23	1.11. Changes from Version 2.0 to 2.5	18
24	1.12. Changes from Version 2.5 to 2.6	19
25	1.13. Changes from Version 2.6 to 2.7	20
26	1.14. Changes from Version 2.7 to 3.0	20
27	1.15. Changes from Version 3.0 to 3.1	22
28	1.16. Changes from Version 3.1 to 3.2	23
29	1.17. Changes from Version 3.2 to 3.3	24
30	1.18. Changes from Version 3.3 to TR 24-1	25
31	1.19. Topics Deferred For a Future Revision	26
32	2. Directives	29
33	2.1. Directive Format	29
34	2.2. Conditional Compilation	30
35	2.3. Internal Control Variables	31
36	2.3.1. Modifying and Retrieving ICV Values	31
37	2.4. Device-Specific Clauses	31
38	2.5. Compute Constructs	33
39	2.5.1. Parallel Construct	33
40	2.5.2. Serial Construct	34
41	2.5.3. Kernels Construct	35
42	2.5.4. Compute Construct Restrictions	36
43	2.5.5. Compute Construct Errors	37
44	2.5.6. if clause	37
45	2.5.7. self clause	37
46	2.5.8. async clause	37
47	2.5.9. wait clause	37
48	2.5.10. num_gangs clause	37
49	2.5.11. num_workers clause	38
50	2.5.12. vector_length clause	38
51	2.5.13. private clause	38
52	2.5.14. firstprivate clause	38
53	2.5.15. reduction clause	39
54	2.5.16. default clause	40

55	2.6. Data Environment	40
56	2.6.1. Variables with Predetermined Data Attributes	40
57	2.6.2. Variables with Implicitly Determined Data Attributes	41
58	2.6.3. Data Regions and Data Lifetimes	42
59	2.6.4. Data Structures with Pointers	43
60	2.6.5. Data Construct	43
61	2.6.6. Enter Data and Exit Data Directives	45
62	2.6.7. Reference Counters	47
63	2.6.8. Attachment Counter	47
64	2.7. Data Clauses	48
65	2.7.1. Data Specification in Data Clauses	48
66	2.7.2. Data Clause Actions	50
67	2.7.3. Data Clause Errors	52
68	2.7.4. Data Clause Modifiers	52
69	2.7.5. deviceptr clause	53
70	2.7.6. present clause	53
71	2.7.7. copy clause	54
72	2.7.8. copyin clause	55
73	2.7.9. copyout clause	56
74	2.7.10. create clause	57
75	2.7.11. no_create clause	57
76	2.7.12. delete clause	58
77	2.7.13. attach clause	59
78	2.7.14. detach clause	59
79	2.8. Host_Data Construct	59
80	2.8.1. use_device clause	60
81	2.8.2. if clause	60
82	2.8.3. if_present clause	61
83	2.9. Loop Construct	61
84	2.9.1. collapse clause	63
85	2.9.2. gang clause	64
86	2.9.3. worker clause	65
87	2.9.4. vector clause	65
88	2.9.5. seq clause	66
89	2.9.6. independent clause	66
90	2.9.7. auto clause	66
91	2.9.8. tile clause	66
92	2.9.9. device_type clause	67
93	2.9.10. private clause	67
94	2.9.11. reduction clause	68
95	2.10. Cache Directive	72
96	2.11. Combined Constructs	72
97	2.12. Atomic Construct	74
98	2.13. Declare Directive	78
99	2.13.1. device_resident clause	79
100	2.13.2. create clause	80
101	2.13.3. link clause	81

102	2.14. Executable Directives	81
103	2.14.1. Init Directive	81
104	2.14.2. Shutdown Directive	82
105	2.14.3. Set Directive	84
106	2.14.4. Update Directive	85
107	2.14.5. Wait Directive	87
108	2.14.6. Enter Data Directive	87
109	2.14.7. Exit Data Directive	88
110	2.15. Procedure Calls in Compute Regions	88
111	2.15.1. Routine Directive	88
112	2.15.2. Global Data Access	95
113	2.16. Asynchronous Behavior	95
114	2.16.1. async clause	96
115	2.16.2. wait clause	97
116	2.16.3. Wait Directive	97
117	2.17. Fortran Specific Behavior	98
118	2.17.1. Optional Arguments	98
119	2.17.2. Do Concurrent Construct	99
120	3. Runtime Library	101
121	3.1. Runtime Library Definitions	101
122	3.2. Runtime Library Routines	102
123	3.2.1. acc_get_num_devices	102
124	3.2.2. acc_set_device_type	102
125	3.2.3. acc_get_device_type	103
126	3.2.4. acc_set_device_num	104
127	3.2.5. acc_get_device_num	104
128	3.2.6. acc_get_property	105
129	3.2.7. acc_init	106
130	3.2.8. acc_shutdown	106
131	3.2.9. acc_async_test	107
132	3.2.10. acc_wait	108
133	3.2.11. acc_wait_async	109
134	3.2.12. acc_wait_any	111
135	3.2.13. acc_get_default_async	111
136	3.2.14. acc_set_default_async	112
137	3.2.15. acc_on_device	112
138	3.2.16. acc_malloc	113
139	3.2.17. acc_free	113
140	3.2.18. acc_copyin and acc_create	114
141	3.2.19. acc_copyout and acc_delete	116
142	3.2.20. acc_update_device and acc_update_self	118
143	3.2.21. acc_map_data	119
144	3.2.22. acc_unmap_data	120
145	3.2.23. acc_deviceptr	121
146	3.2.24. acc_hostptr	121
147	3.2.25. acc_is_present	122
148	3.2.26. acc_memcpy_to_device	122

149	3.2.27. acc_memcpy_from_device	123
150	3.2.28. acc_memcpy_device	125
151	3.2.29. acc_attach and acc_detach	126
152	3.2.30. acc_memcpy_d2d	127
153	4. Environment Variables	131
154	4.1. ACC_DEVICE_TYPE	131
155	4.2. ACC_DEVICE_NUM	131
156	4.3. ACC_PROFLIB	131
157	5. Profiling and Error Callback Interface	133
158	5.1. Events	133
159	5.1.1. Runtime Initialization and Shutdown	134
160	5.1.2. Device Initialization and Shutdown	134
161	5.1.3. Enter Data and Exit Data	135
162	5.1.4. Data Allocation	135
163	5.1.5. Data Construct	136
164	5.1.6. Update Directive	136
165	5.1.7. Compute Construct	136
166	5.1.8. Enqueue Kernel Launch	137
167	5.1.9. Enqueue Data Update (Upload and Download)	137
168	5.1.10. Wait	137
169	5.1.11. Error Event	138
170	5.2. Callbacks Signature	138
171	5.2.1. First Argument: General Information	139
172	5.2.2. Second Argument: Event-Specific Information	140
173	5.2.3. Third Argument: API-Specific Information	145
174	5.3. Loading the Library	146
175	5.3.1. Library Registration	147
176	5.3.2. Statically-Linked Library Initialization	148
177	5.3.3. Runtime Dynamic Library Loading	148
178	5.3.4. Preloading with LD.PRELOAD	149
179	5.3.5. Application-Controlled Initialization	150
180	5.4. Registering Event Callbacks	150
181	5.4.1. Event Registration and Unregistration	150
182	5.4.2. Disabling and Enabling Callbacks	152
183	5.5. Advanced Topics	153
184	5.5.1. Dynamic Behavior	153
185	5.5.2. OpenACC Events During Event Processing	154
186	5.5.3. Multiple Host Threads	155
187	6. Glossary	157
188	A. Recommendations for Implementers	163
189	A.1. Target Devices	163
190	A.1.1. NVIDIA GPU Targets	163
191	A.1.2. AMD GPU Targets	163
192	A.1.3. Multicore Host CPU Target	164

193	A.2. API Routines for Target Platforms	164
194	A.2.1. NVIDIA CUDA Platform	164
195	A.2.2. OpenCL Target Platform	165
196	A.3. Recommended Options and Diagnostics	166
197	A.3.1. C Pointer in Present clause	166
198	A.3.2. Nonconforming Applications and Implementations	167
199	A.3.3. Automatic Data Attributes	167
200	A.3.4. Routine Directive with a Name	167
201	Index	169

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,

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

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

257 OpenACC exposes these three *levels of parallelism* via *gang*, *worker*, and *vector* parallelism. Gang
258 parallelism is coarse-grain. A number of gangs will be launched on the accelerator. The gangs are
259 organized in a one-, two-, or three-dimensional grid, where dimension one corresponds to the inner
260 level of gang parallelism; the default is to only use dimension one. Worker parallelism is fine-grain.
261 Each gang will have one or more workers. Vector parallelism is for SIMD or vector operations
262 within a worker. In this way, OpenACC provides six levels of parallelism, which are arranged
263 from highest to lowest as follows: gang dimension three, gang dimension two, gang dimension one,
264 worker, vector, and sequential, which corresponds to no parallelism.

265 When executing a compute region on a device, one or more gangs are launched, each with one or
266 more workers, where each worker may have vector execution capability with one or more vector
267 lanes. The gangs start executing in *gang-redundant* mode (GR mode), meaning one vector lane of
268 one worker in each gang executes the same code, redundantly. Each gang dimension is associated
269 with a *gang-redundant* mode dimension, denoted GR1, GR2, and GR3. When the program reaches
270 a loop or loop nest marked for gang-level work-sharing at some dimension, the program starts to
271 execute in *gang-partitioned* mode for that dimension, denoted GP1, GP2, or GP3 mode, where the
272 iterations of the loop or loops are partitioned across the gangs in that dimension for truly parallel
273 execution, but still with only one worker per gang and one vector lane per worker active. The
274 program may be simultaneously in different gang modes for different dimensions. For instance,
275 after entering a loop partitioned for gang-level work-sharing at dimension 3, the program will be in
276 GP3, GR2, GR1 mode.

277 When only one worker is active, in any gang-level execution mode, the program is in *worker-single*
278 mode (WS mode). When only one vector lane is active, the program is in *vector-single* mode
279 (VS mode). If a gang reaches a loop or loop nest marked for worker-level work-sharing, the gang
280 transitions to *worker-partitioned* mode (WP mode), which activates all the workers of the gang. The
281 iterations of the loop or loops are partitioned across the workers of this gang. If the same loop is
282 marked for both gang-partitioning in dimension d and worker-partitioning, then the iterations of the
283 loop are spread across all the workers of all the gangs of dimension d . If a worker reaches a loop
284 or loop nest marked for vector-level work-sharing, the worker will transition to *vector-partitioned*
285 mode (VP mode). Similar to WP mode, the transition to VP mode activates all the vector lanes of
286 the worker. The iterations of the loop or loops will be partitioned across the vector lanes using vector
287 or SIMD operations. Again, a single loop may be marked for one, two, or all three of gang, worker,

288 and vector parallelism, and the iterations of that loop will be spread across the gangs, workers, and
289 vector lanes as appropriate.

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

295 Attempting to implement barrier synchronization, critical sections, or locks across any of gang,
296 worker, or vector parallelism might result in deadlock or non-portable code. The execution model
297 allows for an implementation that executes some gangs to completion before starting to execute
298 other gangs. This means that trying to implement synchronization between gangs is likely to fail. In
299 particular, a barrier across gangs cannot be implemented in a portable fashion, since all gangs may
300 not ever be active at the same time. Similarly, the execution model allows for an implementation
301 that executes some workers within a gang or vector lanes within a worker to completion before
302 starting other workers or vector lanes, or for some workers or vector lanes to be suspended until
303 other workers or vector lanes complete. This means that trying to implement synchronization across
304 workers or vector lanes is likely to fail. In particular, implementing a barrier or critical section across
305 workers or vector lanes using atomic operations and a busy-wait loop may never succeed, since the
306 scheduler may suspend the worker or vector lane that owns the lock, and the worker or vector lane
307 waiting on the lock can never complete.

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

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

322 1.3 Memory Model

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

331 The concept of discrete host and accelerator memories is very apparent in low-level accelerator

332 programming languages such as CUDA or OpenCL, in which data movement between the memories
333 can dominate user code. In the OpenACC model, data movement between the memories can be
334 implicit and managed by the compiler, based on directives from the programmer. However, the
335 programmer must be aware of the potentially discrete memories for many reasons, including but
336 not limited to:

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

346 OpenACC exposes the discrete memories through the use of a device data environment. Device data
347 has an explicit lifetime, from when it is allocated or created until it is deleted. If a device shares
348 memory with the local thread, its device data environment will be shared with the local thread. In
349 that case, the implementation need not create new copies of the data for the device and no data
350 movement need be done. If a device has a discrete memory and shares no memory with the local
351 thread, the implementation will allocate space in device memory and copy data between the local
352 memory and device memory, as appropriate. The local thread may share some memory with a
353 device and also have some memory that is not shared with that device. In that case, data in shared
354 memory may be accessed by both the local thread and the device. Data not in shared memory will
355 be copied to device memory as necessary.

356 Some accelerators implement a weak memory model. In particular, they do not support memory
357 coherence between operations executed by different threads; even on the same execution unit, mem-
358 ory coherence is only guaranteed when the memory operations are separated by an explicit memory
359 fence. Otherwise, if one thread updates a memory location and another reads the same location, or
360 two threads store a value to the same location, the hardware may not guarantee the same result for
361 each execution. While a compiler can detect some potential errors of this nature, it is nonetheless
362 possible to write a compute region that produces inconsistent numerical results.

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

368 Some current accelerators have a software-managed cache, some have hardware managed caches,
369 and most have hardware caches that can be used only in certain situations and are limited to read-
370 only data. In low-level programming models such as CUDA or OpenCL languages, it is up to the
371 programmer to manage these caches. In the OpenACC model, these caches are managed by the
372 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:

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

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

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

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

421 To simplify the specification and convey appropriate constraint information, a *pqr-list* is a comma-
422 separated list of one or more *pqr* items. For example, an *int-expr-list* is a comma-separated list
423 of one or more integer expressions, and a *var-list* is a comma-separated list of one or more *vars*.
424 Elements of such a list must not be empty and must not be followed by a trailing comma. The one
425 exception is *clause-list*, which is a list of one or more clauses optionally separated by commas.

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

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

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

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

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

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

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

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

440 1.7 Organization of this document

441 The rest of this document is organized as follows:

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

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

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

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

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

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

454 1.8 References

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

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

460 The use of the following C11 features may result in unspecified behavior.

- 461 – Threads
- 462 – Thread-local storage
- 463 – Parallel memory model
- 464 – Atomic

- 465 • ISO/IEC 9899:2018, *Information Technology – Programming Languages – C*, (C18).

466 The use of the following C18 features may result in unspecified behavior.

- 467 – Thread related features

- 468 • ISO/IEC 14882:1998, *Information Technology – Programming Languages – C++*.
- 469 • ISO/IEC 14882:2011, *Information Technology – Programming Languages – C++*, (C++11).

470 The use of the following C++11 features may result in unspecified behavior.

- 471 – Extern templates
- 472 – copy and rethrow exceptions
- 473 – memory model
- 474 – atomics
- 475 – move semantics
- 476 – `std::thread`
- 477 – thread-local storage

- 478 • ISO/IEC 14882:2014, *Information Technology – Programming Languages – C++*, (C++14).
- 479 • ISO/IEC 14882:2017, *Information Technology – Programming Languages – C++*, (C++17).

480 • ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran – Part*
481 *1: Base Language*, (Fortran 2003).

482 • ISO/IEC 1539-1:2010, *Information Technology – Programming Languages – Fortran – Part*
483 *1: Base Language*, (Fortran 2008).

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

- 485 – Coarrays
- 486 – Simply contiguous arrays rank remapping to rank>1 target
- 487 – Allocatable components of recursive type
- 488 – Polymorphic assignment

489 • ISO/IEC 1539-1:2018, *Information Technology – Programming Languages – Fortran – Part*
490 *1: Base Language*, (Fortran 2018).

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

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

503 1.9 Changes from Version 1.0 to 2.0

- 504 • `_OPENACC` value updated to `201306`
- 505 • `default (none)` clause on `parallel` and `kernels` directives
- 506 • the implicit data attribute for scalars in `parallel` constructs has changed
- 507 • the implicit data attribute for scalars in loops with `loop` directives with the independent
508 attribute has been clarified
- 509 • `acc_async_sync` and `acc_async_noval` values for the `async` clause
- 510 • Clarified the behavior of the `reduction` clause on a `gang` loop
- 511 • Clarified allowable loop nesting (`gang` may not appear inside `worker`, which may not ap-
512 pear within `vector`)
- 513 • `wait` clause on `parallel`, `kernels` and `update` directives

- 514 • **async** clause on the **wait** directive
- 515 • **enter data** and **exit data** directives
- 516 • Fortran *common block* names may now appear in many data clauses
- 517 • **link** clause for the **declare** directive
- 518 • the behavior of the **declare** directive for global data
- 519 • the behavior of a data clause with a C or C++ pointer variable has been clarified
- 520 • predefined data attributes
- 521 • support for multidimensional dynamic C/C++ arrays
- 522 • **tile** and **auto** loop clauses
- 523 • **update self** introduced as a preferred synonym for **update host**
- 524 • **routine** directive and support for separate compilation
- 525 • **device_type** clause and support for multiple device types
- 526 • nested parallelism using **parallel** or **kernels** region containing another **parallel** or **kernels** re-
- 527 **gion**
- 528 • **atomic** constructs
- 529 • new concepts: gang-redundant, gang-partitioned; worker-single, worker-partitioned; vector-
- 530 **single**, vector-partitioned; thread
- 531 • new API routines:
 - 532 – **acc_wait**, **acc_wait_all** instead of **acc_async_wait** and **acc_async_wait_all**
 - 533 – **acc_wait_async**
 - 534 – **acc_copyin**, **acc_present_or_copyin**
 - 535 – **acc_create**, **acc_present_or_create**
 - 536 – **acc_copyout**, **acc_delete**
 - 537 – **acc_map_data**, **acc_unmap_data**
 - 538 – **acc_deviceptr**, **acc_hostptr**
 - 539 – **acc_is_present**
 - 540 – **acc_memcpy_to_device**, **acc_memcpy_from_device**
 - 541 – **acc_update_device**, **acc_update_self**
- 542 • defined behavior with multiple host threads, such as with OpenMP
- 543 • recommendations for specific implementations
- 544 • clarified that no arguments are allowed on the **vector** clause in a **parallel** region

1.10 Corrections in the August 2013 document

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

1.11 Changes from Version 2.0 to 2.5

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

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

600 1.12 Changes from Version 2.5 to 2.6

- 601 • The **_OPENACC** value was updated to **201711**.
- 602 • A new **serial** compute construct was added. See Section 2.5.2 Serial Construct.
- 603 • A new runtime API query routine was added. **acc_get_property** may be called from
604 the host and returns properties about any device. See Section 3.2.6.
- 605 • The text has clarified that if a variable is in a reduction which spans two or more nested loops,
606 each **loop** directive on any of those loops must have a **reduction** clause that contains the
607 variable; see Section 2.9.11 reduction clause.
- 608 • An optional **if** or **if_present** clause is now allowed on the **host_data** construct. See
609 Section 2.8 Host_Data Construct.
- 610 • A new **no_create** data clause is now allowed on compute and **data** constructs. See Sec-
611 tion 2.7.11 no.create clause.
- 612 • The behavior of Fortran optional arguments in data clauses and in routine calls has been
613 specified; see Section 2.17.1 Optional Arguments.
- 614 • The descriptions of some of the Fortran versions of the runtime library routines were simpli-
615 fied; see Section 3.2 Runtime Library Routines.
- 616 • To allow for manual deep copy of data structures with pointers, new *attach* and *detach* be-
617 havior was added to the data clauses, new **attach** and **detach** clauses were added, and
618 matching **acc_attach** and **acc_detach** runtime API routines were added; see Sections
619 2.6.4, 2.7.13-2.7.14 and 3.2.29.

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

1.13 Changes from Version 2.6 to 2.7

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

1.14 Changes from Version 2.7 to 3.0

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

- 658 • Changed the text to clarify uses and limitations of the **device_type** clause and added
659 examples; see Section 2.4.
- 660 • Clarified the conflict between the implicit **copy** clause for variables in a **reduction** clause
661 and the implicit **firstprivate** for scalar variables not in a data clause but used in a
662 **parallel** or **serial** construct; see Sections 2.5.1 and 2.5.2.
- 663 • Required at least one data clause on a **data** construct, an **enter data** directive, or an **exit**
664 **data** directive; see Sections 2.6.5 and 2.6.6.
- 665 • Added text describing how a C++ *lambda* invoked in a compute region and the variables
666 captured by the *lambda* are handled; see Section 2.6.2.
- 667 • Added a **zero** modifier to **create** and **copyout** data clauses that zeros the device memory
668 after it is allocated; see Sections 2.7.9 and 2.7.10.
- 669 • Added a new restriction on the **loop** directive allowing only one of the **seq**, **independent**,
670 and **auto** clauses to appear; see Section 2.9.
- 671 • Added a new restriction on the **loop** directive disallowing a **gang**, **worker**, or **vector**
672 clause to appear if a **seq** clause appears; see Section 2.9.
- 673 • Allowed variables to be modified in an atomic region in a loop where the iterations must
674 otherwise be data independent, such as loops with a **loop independent** clause or a **loop**
675 directive in a **parallel** construct; see Sections 2.9.2, 2.9.3, 2.9.4, and 2.9.6.
- 676 • Clarified the behavior of the **auto** and **independent** clauses on the **loop** directive; see
677 Sections 2.9.7 and 2.9.6.
- 678 • Clarified that an orphaned **loop** construct, or a **loop** construct in a **parallel** construct
679 with no **auto** or **seq** clauses is treated as if an **independent** clause appears; see Sec-
680 tion 2.9.6.
- 681 • For a variable in a **reduction** clause, clarified when the update to the original variable is
682 complete, and added examples; see Section 2.9.11.
- 683 • Clarified that a variable in an orphaned **reduction** clause must be private; see Section 2.9.11.
- 684 • Required at least one clause on a **declare** directive; see Section 2.13.
- 685 • Added an **if** clause to **init**, **shutdown**, **set**, and **wait** directives; see Sections 2.14.1,
686 2.14.2, 2.14.3, and 2.16.3.
- 687 • Required at least one clause on a **set** directive; see Section 2.14.3.
- 688 • Added a *devnum* modifier to the **wait** directive and clause to specify a device to which the
689 wait operation applies; see Section 2.16.3.
- 690 • Allowed a **routine** directive to include a C++ lambda name or to appear before a C++
691 lambda definition, and defined implicit **routine** directive behavior when a C++ lambda is
692 called in a compute region or an accelerator routine; see Section 2.15.
- 693 • Added runtime API routine **acc_memcpy_d2d** for copying data directly between two de-
694 vice arrays on the same or different devices; see Section 3.2.30.
- 695 • Defined the values for the **acc_construct_t** and **acc_device_api** enumerations for
696 cross-implementation compatibility; see Sections 5.2.2 and 5.2.3.

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

700 1.15 Changes from Version 3.0 to 3.1

- 701 • Updated `_OPENACC` value to `202011`.
- 702 • Clarified that Fortran blank common blocks are not permitted and that same-named common
- 703 blocks must have the same size. See Section 1.6.
- 704 • Clarified that a `parallel` construct's block is considered to start in gang-redundant mode
- 705 even if there's just a single gang. See Section 2.5.1.
- 706 • Added support for the Fortran `BLOCK` construct. See Sections 2.5.1, 2.5.3, 2.6.1, 2.6.5, 2.8,
- 707 2.13, and 6.
- 708 • Defined the `serial` construct in terms of the `parallel` construct to improve readability.
- 709 Instead of defining it in terms of clauses `num_gangs (1) num_workers (1)`
- 710 `vector_length (1)`, defined the `serial` construct as executing with a single gang of a
- 711 single worker with a vector length of one. See Section 2.5.2.
- 712 • Consolidated compute construct restrictions into a new section to improve readability. See
- 713 Section 2.5.4.
- 714 • Clarified that a `default` clause may appear at most once on a compute construct. See
- 715 Section 2.5.16.
- 716 • Consolidated discussions of implicit data attributes on compute and combined constructs into
- 717 a separate section. Clarified the conditions under which each data attribute is implied. See
- 718 Section 2.6.2.
- 719 • Added a restriction that certain loop reduction variables must have explicit data clauses on
- 720 their parent compute constructs. This change addresses portability across existing OpenACC
- 721 implementations. See Sections 2.6.2 and A.3.3.
- 722 • Restored the OpenACC 2.5 behavior of the `present`, `copy`, `copyin`, `copyout`, `create`,
- 723 `no_create`, `delete` data clauses at exit from a region, or on an `exit data` directive, as
- 724 applicable, and `create` clause at exit from an implicit data region where a `declare` di-
- 725 rective appears, and `acc_copyout`, `acc_delete` routines, such that no action is taken if
- 726 the appropriate reference counter is zero, instead of a runtime error being issued if data is not
- 727 present. See Sections 2.7.6, 2.7.7, 2.7.8, 2.7.9, 2.7.10, 2.7.11, 2.7.12, 2.13.2, and 3.2.19.
- 728 • Clarified restrictions on loop forms that can be associated with `loop` constructs, including
- 729 the case of C++ range-based `for` loops. See Section 2.9.
- 730 • Specified where `gang` clauses are implied on `loop` constructs. This change standardizes
- 731 behavior of existing OpenACC implementations. See Section 2.9.2.
- 732 • Corrected C/C++ syntax for `atomic capture` with a structured block. See Section 2.12.
- 733 • Added the behavior of the Fortran `do concurrent` construct. See Section 2.17.2.

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

743 1.16 Changes from Version 3.1 to 3.2

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

- 770 – The two sections for **acc_copyin** and **acc_create** were combined into a single
771 section. See Section 3.2.18.
- 772 – The two sections for **acc_copyout** and **acc_delete** were combined into a single
773 section. See Section 3.2.19.
- 774 – The two sections for **acc_update_self** and **acc_update_device** were com-
775 bined into a single section. See Section 3.2.20.
- 776 – The two sections for **acc_attach** and **acc_detach** were combined into a single
777 section. See Section 3.2.29.
- 778 • Added runtime API routine **acc_wait_any**. See section 3.2.12.
- 779 • The descriptions of the **async** and **async_queue** fields of **acc_callback_info** were
780 clarified. See Section 5.2.1.

781 1.17 Changes from Version 3.2 to 3.3

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

- 806 • Added a Fortran interface for the following runtime routines (See Chapter 3):
 - 807 – **acc_malloc**
 - 808 – **acc_free**
 - 809 – **acc_map_data**
 - 810 – **acc_unmap_data**
 - 811 – **acc_deviceptr**
 - 812 – **acc_hostptr**
 - 813 – The two **acc_memcpy_to_device** routines
 - 814 – The two **acc_memcpy_from_device** routines
 - 815 – The two **acc_memcpy_device** routines
 - 816 – The two **acc_attach** routines
 - 817 – The four **acc_detach** routines
- 818 • Added a new error condition for **acc_map_data** when the **bytes** argument is zero. See
819 Section 3.2.21.
- 820 • Added recommendations for how a **routine** directive affects multicore host CPU compila-
821 tion. See Section A.1.3.
- 822 • Recommended additional diagnostics promoting portable and readable OpenACC. See Section A.3.

823 1.18 Changes from Version 3.3 to TR 24-1

- 824 • Clarified that a *pqr-list* must have at least one item and is not permitted to have a trailing
825 comma. See Section 1.6.
- 826 • Clarified that the **_Pragma** operator form is supported for OpenACC directives in C and
827 C++. See Section 2.1.
- 828 • Clarified user-visible behavior of evaluation of expressions in directive arguments. See Section-
829 2.1.
- 830 • Clarified the analysis of implicit data attributes and parallelism across the boundaries of pro-
831 cedures that can appear within other procedures (e.g., C++ lambdas, C++ class member func-
832 tions, and Fortran internal procedures). See Sections 2.5, 2.6.2, 2.9, and 2.15.1.
- 833 • Restated data actions to improve data clause descriptions. See Section 2.7.2.
- 834 • Added the **capture** modifier for specifying that a particular variable requires a discrete
835 copy in device-accessible memory, even when already in shared memory. See Section 2.7.4,
836 Section 2.7.9 and Section 2.7.10.
- 837 • Added the **always**, **alwaysin**, and **alwaysout** modifiers to the **copy**, **copyin**, and
838 **copyout** data clauses. See Section 2.7.7, Section 2.7.8, and Section 2.7.9.
- 839 • Clarified that intrinsic assignment of *declare create* variable in Fortran will result in memory
840 allocation and/or deallocation on the device if memory is allocated and/or deallocated on the
841 host. See Section 2.7.10

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

855 1.19 Topics Deferred For a Future Revision

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

- 860 • Directives to define implicit *deep copy* behavior for pointer-based data structures.
- 861 • Defined behavior when data in data clauses on a directive are aliases of each other.
- 862 • Clarifying when data becomes *present* or *not present* on the device for **enter data** or **exit**
863 **data** directives with an **async** clause.
- 864 • Clarifying the behavior of Fortran **pointer** variables in data clauses.
- 865 • Allowing Fortran **pointer** variables to appear in **deviceptr** clauses.
- 866 • Support for attaching C/C++ pointers that point to an address past the end of a memory region.
- 867 • Fully defined interaction with multiple host threads.
- 868 • Optionally removing the synchronization or barrier at the end of vector and worker loops.
- 869 • Allowing an **if** clause after a **device_type** clause.
- 870 • A **shared** clause (or something similar) for the loop directive.
- 871 • Better support for multiple devices from a single thread, whether of the same type or of
872 different types.
- 873 • An *auto* construct (by some name), to allow **kernels**-like auto-parallelization behavior
874 inside **parallel** constructs or accelerator routines.
- 875 • A **begin declare ... end declare** construct that behaves like putting any global vari-
876 ables declared inside the construct in a **declare** clause.
- 877 • Defining the behavior of additional parallelism constructs in the base languages when used
878 inside a compute construct or accelerator routine.

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

2. Directives

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

2.1 Directive Format

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

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

or a **_Pragma** operator:

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

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

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

```
!$acc directive-name [clause-list]
```

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

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

```
!$acc directive-name [clause-list]
```

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

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

The sentinel (**!\$acc**, **c\$acc**, or ***\$acc**) must occupy columns 1-5. Fixed form line length, whitespace, continuation, and column rules apply to the directive line. Initial directive lines must have a space or zero in column 6, and continuation directive lines must have a character other than a space or zero in column 6. Comments may appear on the same line as a directive, starting with an exclamation point on or after column 7 and continuing to the end of the line.

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

930 Only one *directive-name* can appear per directive, except that a combined directive name is consid-
 931 ered a single *directive-name*.

932 The order in which clauses appear is not significant unless otherwise specified. A program must
 933 not depend on the order of evaluation of expressions in clause, construct, or directive arguments,
 934 or on any side effects of the evaluations. (See examples below.) Clauses may be repeated unless
 935 otherwise specified.

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

937 Examples

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

```
942     #pragma acc parallel \
943         num_gangs(foo(++i)) \
944         num_workers(bar(++i)) \
945         async(foo(++i))
946     { ... }
```

947 See Section 2.5.1 for the `parallel` construct.

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

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

953 See Section 2.14.4 for the `update` directive.

- 954 • In the following example, execution and order of the constructor and destructor of `S` and `U` is
 955 not guaranteed.

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

958 See Section 2.16.3 for the `wait` directive.

961 2.2 Conditional Compilation

962 The `_OPENACC` macro name is defined to have a value `yyyymm` where `yyyy` is the year and `mm` is
 963 the month designation of the version of the OpenACC directives supported by the implementation.
 964 This macro must be defined by a compiler only when OpenACC directives are enabled. The version
 965 described here is 202211.

2.3 Internal Control Variables

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

The ICVs are:

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

2.3.1 Modifying and Retrieving ICV Values

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

ICV	Ways to modify values	Way to retrieve value
<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>
<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>

The initial values are implementation-defined. After initial values are assigned, but before any OpenACC construct or API routine is executed, the values of any environment variables that were set by the user are read and the associated ICVs are modified accordingly. There is one copy of each ICV for each host thread that is not generated by a compute construct. For threads that are generated by a compute construct the initial value for each ICV is inherited from the local thread. The behavior for each ICV is as if there is a copy for each thread. If an ICV is modified, then a unique copy of that ICV must be created for the modifying thread.

2.4 Device-Specific Clauses

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

The argument to the `device_type` clause is a comma-separated list of one or more device architecture name identifiers, or an asterisk. An asterisk indicates all device types that are not named

996 in any other **device_type** clause on that directive. A single directive may have one or several
 997 **device_type** clauses. The **device_type** clauses may appear in any order.

998 Except where otherwise noted, the rest of this document describes device-independent directives, on
 999 which all clauses apply when compiling for any device type. When compiling a device-dependent
 1000 directive for a particular device type, the directive is treated as if the only clauses that appear are (a)
 1001 the clauses specific to that device type and (b) all default clauses for which there are no like-named
 1002 clauses specific to that device type. If, for any device type, the resulting directive is nonconforming,
 1003 then the original directive is nonconforming.

1004 The supported device types are implementation-defined. Depending on the implementation and the
 1005 compiling environment, an implementation may support only a single device type, or may support
 1006 multiple device types but only one at a time, or may support multiple device types in a single
 1007 compilation.

1008 A device architecture name may be generic, such as a vendor, or more specific, such as a partic-
 1009 ular generation of device; see Appendix A Recommendations for Implementers for recommended
 1010 names. When compiling for a particular device, the implementation will use the clauses associated
 1011 with the **device_type** clause that specifies the most specific architecture name that applies for
 1012 this device; clauses associated with any other **device_type** clause are ignored. In this context,
 1013 the asterisk is the least specific architecture name.

1014 Syntax

1015 The syntax of the **device_type** clause is

```
1016     device_type( * )
1017     device_type( device-type-list )
```

1018

1019 The **device_type** clause may be abbreviated to **dtype**.

1020

1021 Examples

1022

- 1023 • On the following directive, **worker** appears as a device-specific clause for devices of type
 1024 **foo**, but **gang** appears as a default clause and so applies to all device types, including **foo**.

```
1025     #pragma acc loop gang device_type(foo) worker
```

- 1026 • The first directive below is identical to the previous directive except that **loop** is replaced
 1027 with **routine**. Unlike **loop**, **routine** does not permit **gang** to appear with **worker**,
 1028 but both apply for device type **foo**, so the directive is nonconforming. The second directive
 1029 below is conforming because **gang** there applies to all device types except **foo**.

```
1030     // nonconforming: gang and worker not permitted together
1031     #pragma acc routine gang device_type(foo) worker
1032
1033     // conforming: gang and worker for different device types
1034     #pragma acc routine device_type(foo) worker \
1035         device_type(*) gang
```



```

1074     !$acc parallel [ clause-list ]
1075         block construct
1076     [ !$acc end parallel ]

```

1077 where *clause* is one of the following:

```

1078     async [ ( int-expr ) ]
1079     wait [ ( int-expr-list ) ]
1080     num_gangs ( int-expr-list )
1081     num_workers ( int-expr )
1082     vector_length ( int-expr )
1083     device_type ( device-type-list )
1084     if ( condition )
1085     self [ ( condition ) ]
1086     reduction ( operator : var-list )
1087     copy ( [ modifier-list : ] var-list )
1088     copyin ( [ modifier-list : ] var-list )
1089     copyout ( [ modifier-list : ] var-list )
1090     create ( [ modifier-list : ] var-list )
1091     no_create ( var-list )
1092     present ( var-list )
1093     deviceptr ( var-list )
1094     attach ( var-list )
1095     private ( var-list )
1096     firstprivate ( var-list )
1097     default ( none | present )

```

1098 Description

1099 When the program encounters an accelerator **parallel** construct, one or more gangs of workers
1100 are created to execute the accelerator parallel region. The number of gangs, and the number of
1101 workers in each gang and the number of vector lanes per worker remain constant for the duration of
1102 that parallel region. Each gang begins executing the code in the structured block in gang-redundant
1103 mode even if there is only a single gang. This means that code within the parallel region, but outside
1104 of a loop construct with gang-level worksharing, will be executed redundantly by all gangs.

1105 One worker in each gang begins executing the code in the structured block of the construct. **Note:**
1106 Unless there is a **loop** construct within the parallel region, all gangs will execute all the code within
1107 the region redundantly.

1108 If the **async** clause does not appear, there is an implicit barrier at the end of the accelerator parallel
1109 region, and the execution of the local thread will not proceed until all gangs have reached the end
1110 of the parallel region.

1111 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
1112 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**
1113 clauses are described in Sections 2.5.13 and Sections 2.5.14. The **device_type** clause is de-
1114 scribed in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described
1115 in Section 2.6.2. Restrictions are described in Section 2.5.4.

1116 2.5.2 Serial Construct

1117 **Summary**

1118 This construct defines a region of the program that is to be executed sequentially on the current
 1119 device. The behavior of the **serial** construct is the same as that of the **parallel** construct
 1120 except that it always executes with a single gang of a single worker with a vector length of one.

1121 **Note:** The **serial** construct may be used to execute sequential code on the current device,
 1122 which removes the need for data movement when the required data is already present on the device.

1123 **Syntax**

1124 In C and C++, the syntax of the OpenACC **serial** construct is

```
1125     #pragma acc serial [clause-list] new-line
1126         structured block
```

1127

1128 and in Fortran, the syntax is

```
1129     !$acc serial [ clause-list ]
1130         structured block
1131     !$acc end serial
```

1132 OR

```
1133     !$acc serial [ clause-list ]
1134         block construct
1135     [!$acc end serial]
```

1136 where *clause* is as for the **parallel** construct except that the **num_gangs**, **num_workers**, and
 1137 **vector_length** clauses are not permitted.

1138 **2.5.3 Kernels Construct**1139 **Summary**

1140 This construct defines a region of the program that is to be compiled into a sequence of kernels for
 1141 execution on the current device.

1142 **Syntax**

1143 In C and C++, the syntax of the OpenACC **kernels** construct is

```
1144     #pragma acc kernels [ clause-list ] new-line
1145         structured block
```

1146

1147 and in Fortran, the syntax is

```
1148     !$acc kernels [ clause-list ]
1149         structured block
1150     !$acc end kernels
```

1151 OR

```
1152     !$acc kernels [ clause-list ]
1153         block construct
1154     [!$acc end kernels]
```

1155 where *clause* is one of the following:

```

1156   async [ ( int-expr ) ]
1157   wait [ ( int-expr-list ) ]
1158   num_gangs ( int-expr )
1159   num_workers ( int-expr )
1160   vector_length ( int-expr )
1161   device_type ( device-type-list )
1162   if ( condition )
1163   self [ ( condition ) ]
1164   copy ( [ modifier-list : ] var-list )
1165   copyin ( [ modifier-list : ] var-list )
1166   copyout ( [ modifier-list : ] var-list )
1167   create ( [ modifier-list : ] var-list )
1168   no_create ( var-list )
1169   present ( var-list )
1170   deviceptr ( var-list )
1171   attach ( var-list )
1172   default ( none | present )

```

1173 Description

1174 The compiler will split the code in the kernels region into a sequence of accelerator kernels. Typi-
 1175 cally, each loop nest will be a distinct kernel. When the program encounters a **kernels** construct,
 1176 it will launch the sequence of kernels in order on the device. The number and configuration of gangs
 1177 of workers and vector length may be different for each kernel.

1178 If the **async** clause does not appear, there is an implicit barrier at the end of the kernels region,
 1179 and the local thread execution will not proceed until the entire sequence of kernels has completed
 1180 execution.

1181 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
 1182 data clauses are described in Section 2.7 Data Clauses. The **device_type** clause is described
 1183 in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described in Sec-
 1184 tion 2.6.2. Restrictions are described in Section 2.5.4.

1185 2.5.4 Compute Construct Restrictions

1186 The following restrictions apply to all compute constructs:

- 1187 • A program may not branch into or out of a compute construct.
- 1188 • Only the **async**, **wait**, **num_gangs**, **num_workers**, and **vector_length** clauses
 1189 may follow a **device_type** clause.
- 1190 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
 1191 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1192 • At most one **default** clause may appear, and it must have a value of either **none** or
 1193 **present**.
- 1194 • A **reduction** clause may not appear on a **parallel** construct with a **num_gangs** clause
 1195 that has more than one argument.

2.5.5 Compute Construct Errors

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

See Section 5.2.2.

2.5.6 if clause

The **if** clause is optional.

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

When the *condition* in the **if** clause evaluates to *false*., the local thread will execute the region.

2.5.7 self clause

The **self** clause is optional.

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

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

2.5.8 async clause

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

2.5.9 wait clause

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

2.5.10 num_gangs clause

The **num_gangs** clause is allowed on the **parallel** and **kernels** constructs. On a **parallel** construct, it may have one, two, or three arguments. The values of the integer expressions define

1229 the number of parallel gangs along dimensions one, two, and three that will execute the parallel
1230 region. If it has fewer than three arguments, the missing values are treated as having the value 1.
1231 The total number of gangs must be at least 1 and is the product of the values of the arguments. On a
1232 **kernels** construct, the **num_gangs** clause must have a single argument, the value of which will
1233 define the number of parallel gangs that will execute each kernel created for the kernels region.

1234 If the **num_gangs** clause does not appear, an implementation-defined default will be used which
1235 may depend on the code within the construct. The implementation may use a lower value than
1236 specified based on limitations imposed by the target architecture.

1237 **2.5.11 num_workers clause**

1238 The **num_workers** clause is allowed on the **parallel** and **kernels** constructs. The value
1239 of the integer expression defines the number of workers within each gang that will be active after
1240 a gang transitions from worker-single mode to worker-partitioned mode. If the clause does not
1241 appear, an implementation-defined default will be used; the default value may be 1, and may be
1242 different for each **parallel** construct or for each kernel created for a **kernels** construct. The
1243 implementation may use a different value than specified based on limitations imposed by the target
1244 architecture.

1245 **2.5.12 vector_length clause**

1246 The **vector_length** clause is allowed on the **parallel** and **kernels** constructs. The value
1247 of the integer expression defines the number of vector lanes that will be active after a worker transi-
1248 tions from vector-single mode to vector-partitioned mode. This clause determines the vector length
1249 to use for vector or SIMD operations. If the clause does not appear, an implementation-defined
1250 default will be used. This vector length will be used for loop constructs annotated with the **vector**
1251 clause, as well as loops automatically vectorized by the compiler. The implementation may use a
1252 different value than specified based on limitations imposed by the target architecture.

1253 **2.5.13 private clause**

1254 The **private** clause is allowed on the **parallel** and **serial** constructs; it declares that a copy
1255 of each item on the list will be created for each gang in all dimensions.

1256 **Restrictions**

- 1257 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private**
1258 clauses.

1259 **2.5.14 firstprivate clause**

1260 The **firstprivate** clause is allowed on the **parallel** and **serial** constructs; it declares that
1261 a copy of each item on the list will be created for each gang, and that the copy will be initialized with
1262 the value of that item on the local thread when a **parallel** or **serial** construct is encountered.

1263 **Restrictions**

- 1264 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
1265 **firstprivate** clauses.

1266 2.5.15 reduction clause

1267 The **reduction** clause is allowed on the **parallel** and **serial** constructs. It specifies a
 1268 reduction operator and one or more *vars*. It implies **copy** clauses as described in Section 2.6.2. For
 1269 each reduction *var*, a private copy is created for each parallel gang and initialized for that operator.
 1270 At the end of the region, the values for each gang are combined using the reduction operator, and
 1271 the result combined with the value of the original *var* and stored in the original *var*. If the reduction
 1272 *var* is an array or subarray, the array reduction operation is logically equivalent to applying that
 1273 reduction operation to each element of the array or subarray individually. If the reduction *var*
 1274 is a composite variable, the reduction operation is logically equivalent to applying that reduction
 1275 operation to each member of the composite variable individually. The reduction result is available
 1276 after the region.

1277 The following table lists the operators that are valid and the initialization values; in each case, the
 1278 initialization value will be cast into the data type of the *var*. For **max** and **min** reductions, the
 1279 initialization values are the least representable value and the largest representable value for that data
 1280 type, respectively. At a minimum, the supported data types include Fortran **logical** as well as
 1281 the numerical data types in C (e.g., **_Bool**, **char**, **int**, **float**, **double**, **float _Complex**,
 1282 **double _Complex**), C++ (e.g., **bool**, **char**, **wchar_t**, **int**, **float**, **double**), and Fortran
 1283 (e.g., **integer**, **real**, **double precision**, **complex**). However, for each reduction operator,
 1284 the supported data types include only the types permitted as operands to the corresponding operator
 1285 in the base language where (1) for max and min, the corresponding operator is less-than and (2) for
 1286 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.

1287 Restrictions

- 1288 • A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name,
 1289 an array element, or a subarray (refer to Section 2.7.1).
- 1291 • If the reduction *var* is an array element or a subarray, accessing the elements of the array
 1292 outside the specified index range results in unspecified behavior.
- 1293 • The reduction *var* may not be a member of a composite variable.
- 1294 • If the reduction *var* is a composite variable, each member of the composite variable must be
 1295 a supported datatype for the reduction operation.

- 1296 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
1297 **reduction** clauses.

1298 2.5.16 default clause

1299 The **default** clause is optional. At most one **default** clause may appear. It adjusts what
1300 data attributes are implicitly determined for variables used in the compute construct as described in
1301 Section 2.6.2.

1302 2.6 Data Environment

1303 This section describes the data attributes for variables. The data attributes for a variable may be
1304 *predetermined*, *implicitly determined*, or *explicitly determined*. Variables with predetermined data
1305 attributes may not appear in a data clause that conflicts with that data attribute. Variables with
1306 implicitly determined data attributes may appear in a data clause that overrides the implicit attribute.
1307 Variables with explicitly determined data attributes are those which appear in a data clause on a
1308 **data** construct, a compute construct, or a **declare** directive. See Section A.3.3 for recommended
1309 diagnostics related to data attributes.

1310 OpenACC supports systems with accelerators that have discrete memory from the host, systems
1311 with accelerators that share memory with the host, as well as systems where an accelerator shares
1312 some memory with the host but also has some discrete memory that is not shared with the host.
1313 In the first case, no data is in shared memory. In the second case, all data is in shared memory.
1314 In the third case, some data may be in shared memory and some data may be in discrete memory,
1315 although a single array or aggregate data structure must be allocated completely in shared or discrete
1316 memory. When a nested OpenACC construct is executed on the device, the default target device for
1317 that construct is the same device on which the encountering accelerator thread is executing. In that
1318 case, the target device shares memory with the encountering thread.

1319 Memory is considered *shared memory* if data residing in that memory is accessible from both the
1320 host and the current device. Memory is considered *device memory* if it is physically connected to the
1321 current device. Memory is considered *device-accessible* if it is accessible from the current device,
1322 regardless of where the physical memory resides. A *captured variable* is a variable which the user
1323 has specific must have a *device-accessible* copy that is discrete from the original, even if the original
1324 is in *shared memory*.

1325 2.6.1 Variables with Predetermined Data Attributes

1326 The loop variable in a C **for** statement or Fortran **do** statement that is associated with a loop
1327 directive is predetermined to be private to each thread that will execute each iteration of the loop.
1328 Loop variables in Fortran **do** statements within a compute construct are predetermined to be private
1329 to the thread that executes the loop.

1330 Variables declared in a C block or Fortran block construct that is executed in *vector-partitioned*
1331 mode are private to the thread associated with each vector lane. Variables declared in a C block
1332 or Fortran block construct that is executed in *worker-partitioned vector-single* mode are private to
1333 the worker and shared across the threads associated with the vector lanes of that worker. Variables
1334 declared in a C block or Fortran block construct that is executed in *worker-single* mode are private
1335 to the gang and shared across the threads associated with the workers and vector lanes of that gang.

1336 A procedure called from a compute construct will be annotated as **seq**, **vector**, **worker**, or

1337 **gang**, as described Section 2.15 Procedure Calls in Compute Regions. Variables declared in **seq**
 1338 routine are private to the thread that made the call. Variables declared in **vector** routine are private
 1339 to the worker that made the call and shared across the threads associated with the vector lanes of
 1340 that worker. Variables declared in **worker** or **gang** routine are private to the gang that made the
 1341 call and shared across the threads associated with the workers and vector lanes of that gang.

1342 2.6.2 Variables with Implicitly Determined Data Attributes

1343 When implicitly determining data attributes on a compute construct, the following clauses are visi-
 1344 ble and variable accesses are exposed to the compute construct:

- 1345 • *Visible **default** clause:* The nearest **default** clause appearing on the compute construct
 1346 or on a lexically enclosing **data** construct that has the same parent compute scope.
- 1347 • *Visible data clause:* Any data clause on the compute construct, on a lexically enclosing **data**
 1348 construct that has the same parent compute scope, or on a visible **declare** directive.
- 1349 • *Exposed variable access:* Any access to the data or address of a variable at a point within the
 1350 compute construct where the variable is not private to a scope lexically enclosed within the
 1351 compute construct.

1352 **Note:** In the argument of C's **sizeof** operator, the appearance of a variable is not an exposed
 1353 access because neither its data nor its address is accessed. In the argument of a **reduction**
 1354 clause on an enclosed **loop** construct, the appearance of a variable that is not otherwise
 1355 privatized is an exposed access to the original variable.

1356 On a compute or combined construct, if a variable appears in a **reduction** clause but no other
 1357 data clause, it is treated as if it also appears in a **copy** clause. Otherwise, for any variable, the
 1358 compiler will implicitly determine its data attribute on a compute construct if all of the following
 1359 conditions are met:

- 1360 • There is no **default (none)** clause visible at the compute construct.
- 1361 • An access to the variable is exposed to the compute construct.
- 1362 • The variable does not appear in a data clause visible at the compute construct.

1363 An aggregate variable will be treated as if it appears either:

- 1364 • In a **present** clause if there is a **default (present)** clause visible at the compute con-
 1365 struct.
- 1366 • In a **copy** clause otherwise.

1367 A scalar variable will be treated as if it appears either:

- 1368 • In a **copy** clause if the compute construct is a **kernels** construct.
- 1369 • In a **firstprivate** clause otherwise.

1370 **Note:** Any **default (none)** clause visible at the compute construct applies to both aggregate
 1371 and scalar variables. However, any **default (present)** clause visible at the compute construct
 1372 applies only to aggregate variables.

1373 Restrictions

- 1374 • If there is a **default (none)** clause visible at a compute construct, for any variable access
1375 exposed to the compute construct, the compiler requires the variable to appear either in an
1376 explicit data clause visible at the compute construct or in a **firstprivate**, **private**, or
1377 **reduction** clause on the compute construct.
- 1378 • If a scalar variable appears in a **reduction** clause on a **loop** construct that has a parent
1379 **parallel** or **serial** construct, and if the reduction's access to the original variable is
1380 exposed to the parent compute construct, the variable must appear either in an explicit data
1381 clause visible at the compute construct or in a **firstprivate**, **private**, or **reduction**
1382 clause on the compute construct. **Note:** Implementations are encouraged to issue a compile-
1383 time diagnostic when this restriction is violated to assist users in writing portable OpenACC
1384 applications.

1385 If a C++ *lambda* is called in a compute region and does not appear in a data clause, then it is
1386 treated as if it appears in a **copyin** clause on the current construct. A variable captured by a
1387 *lambda* is processed according to its data types: a pointer type variable is treated as if it appears
1388 in a **no_create** clause; a reference type variable is treated as if it appears in a **present** clause;
1389 for a struct or a class type variable, any pointer member is treated as if it appears in a **no_create**
1390 clause on the current construct. If the variable is defined as global or file or function static, it must
1391 appear in a **declare** directive.

1392 2.6.3 Data Regions and Data Lifetimes

1393 Data in shared memory is accessible from the current device as well as to the local thread. Such
1394 data is available to the accelerator for the lifetime of the variable. Data not in shared memory must
1395 be copied to and from device memory using data constructs, clauses, and API routines. A *data*
1396 *lifetime* is the duration from when the data is first made available to the accelerator until it becomes
1397 unavailable. For data in shared memory, the data lifetime begins when the data is allocated and
1398 ends when it is deallocated; for statically allocated data, the data lifetime begins when the program
1399 begins and does not end. For data not in shared memory, the data lifetime begins when it is made
1400 present and ends when it is no longer present.

1401 There are four types of data regions. When the program encounters a **data** construct, it creates a
1402 data region.

1403 When the program encounters a compute construct with explicit data clauses or with implicit data
1404 allocation added by the compiler, it creates a data region that has a duration of the compute construct.

1405 When the program enters a procedure, it creates an implicit data region that has a duration of the
1406 procedure. That is, the implicit data region is created when the procedure is called, and exited when
1407 the program returns from that procedure invocation. There is also an implicit data region associated
1408 with the execution of the program itself. The implicit program data region has a duration of the
1409 execution of the program.

1410 In addition to data regions, a program may create and delete data on the accelerator using **enter**
1411 **data** and **exit data** directives or using runtime API routines. When the program executes
1412 an **enter data** directive, or executes a call to a runtime API **acc_copyin** or **acc_create**
1413 routine, each *var* on the directive or the variable on the runtime API argument list will be made live
1414 on accelerator.

1415 2.6.4 Data Structures with Pointers

1416 This section describes the behavior of data structures that contain pointers. A pointer may be a
 1417 C or C++ pointer (e.g., **float***), a Fortran pointer or array pointer (e.g., **real, pointer,**
 1418 **dimension(:)**), or a Fortran allocatable (e.g., **real, allocatable, dimension(:)**).

1419 When a data object is copied to device memory, the values are copied exactly. If the data is a data
 1420 structure that includes a pointer, or is just a pointer, the pointer value copied to device memory
 1421 will be the host pointer value. If the pointer target object is also allocated in or copied to device
 1422 memory, the pointer itself needs to be updated with the device address of the target object before
 1423 dereferencing the pointer in device memory.

1424 An *attach* action updates the pointer in device memory to point to the device copy of the data that
 1425 the host pointer targets; see Section 2.7.2. For Fortran array pointers and allocatable arrays, this
 1426 includes copying any associated descriptor (dope vector) to the device copy of the pointer. When
 1427 the device pointer target is deallocated, the pointer in device memory is restored to the host value, so
 1428 it can be safely copied back to host memory. A *detach* action updates the pointer in device memory
 1429 to have the same value as the corresponding pointer in local memory; see Section 2.7.2. The *attach*
 1430 and *detach* actions are performed by the **copy, copyin, copyout, create, attach,** and
 1431 **detach** data clauses (Sections 2.7.5-2.7.14), and the **acc_attach** and **acc_detach** runtime
 1432 API routines (Section 3.2.29). The *attach* and *detach* actions use attachment counters to determine
 1433 when the pointer in device memory needs to be updated; see Section 2.6.8.

1434 2.6.5 Data Construct

1435 Summary

1436 The **data** construct defines *vars* are accessible to the current device for the duration of the region.
 1437 It also defines the data actions that occur upon entry to and exit from the region.

1438 Syntax

1439 In C and C++, the syntax of the OpenACC **data** construct is

```
1440     #pragma acc data [clause-list] new-line
1441         structured block
```

1442 and in Fortran, the syntax is

```
1443     !$acc data [clause-list]
1444         structured block
1445     !$acc end data
```

1446 OR

```
1447     !$acc data [clause-list]
1448         block construct
1449     [!$acc end data]
```

1450 where *clause* is one of the following:

```
1451     if ( condition )
1452     async [ ( int-expr ) ]
1453     wait [ ( wait-argument ) ]
1454     device_type ( device-type-list )
```

```

1455     copy ( [modifier-list : ] var-list )
1456     copyin ( [modifier-list : ] var-list )
1457     copyout ( [modifier-list : ] var-list )
1458     create ( [modifier-list : ] var-list )
1459     no_create ( var-list )
1460     present ( var-list )
1461     deviceptr ( var-list )
1462     attach ( var-list )
1463     default ( none | present )

```

1464 **Description**

1465 Data will be allocated in the memory of the current device and copied from local memory to device
 1466 memory, or copied back, as required. The data clauses are described in Section 2.7 Data Clauses.
 1467 Structured reference counters are incremented for data when entering a data region, and decre-
 1468 mented when leaving the region, as described in Section 2.6.7 Reference Counters. The **device_type**
 1469 clause is described in Section 2.4 Device-Specific Clauses.

1470 **Restrictions**

- 1471 • At least one **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**,
 1472 **attach**, or **default** clause must appear on a **data** construct.
- 1473 • Only the **async** and **wait** clauses may follow a **device_type** clause.

1474 **if clause**

1475 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate
 1476 space in the current device memory and move data from and to the local memory as required. When
 1477 an **if** clause appears, the program will conditionally allocate memory in and move data to and/or
 1478 from device memory. When the *condition* in the **if** clause evaluates to *false*, no device memory
 1479 will be allocated, and no data will be moved. When the *condition* evaluates to *true*, the data will be
 1480 allocated and moved as specified. At most one **if** clause may appear.

1481 **async clause**

1482 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1483 **Note:** The **async** clause only affects operations directly associated with this particular **data** con-
 1484 struct, such as data transfers. Execution of the associated structured block or block construct remains
 1485 synchronous to the local thread. Nested OpenACC constructs, directives, and calls to runtime li-
 1486 brary routines do not inherit the **async** clause from this construct, and the programmer must take
 1487 care to not accidentally introduce race conditions related to asynchronous data transfers.

1488 **wait clause**

1489 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1490 **default clause**

1491 The **default** clause is optional. At most one **default** clause may appear. It adjusts what data
 1492 attributes are implicitly determined for variables used in lexically contained compute constructs as
 1493 described in Section 2.6.2.

1494 **Errors**

- 1495 • See Section 2.7.3 for errors due to data clauses.
- 1496 • See Sections 2.16.1 and 2.16.2 for errors due to **async** or **wait** clauses.

1497 **2.6.6 Enter Data and Exit Data Directives**1498 **Summary**

1499 An **enter data** directive defines *vars* are accessible to the current device for the remaining dura-
 1500 tion of the program, or until an **exit data** directive makes the data no longer accessible. These
 1501 directives also specify data actions which occur upon reaching the **enter data** or **exit data** di-
 1502 rective. The dynamic data lifetime for data referred to by an **enter data** or **exit data** directive
 1503 is defined by its dynamic reference counter, as defined in Section 2.6.7.

1504 **Syntax**

1505 In C and C++, the syntax of the OpenACC **enter data** directive is

1506 **#pragma acc enter data** *clause-list new-line*

1507 and in Fortran, the syntax is

1508 **!\$acc enter data** *clause-list*

1509 where *clause* is one of the following:

```

1510 if ( condition )
1511 async [ ( int-expr ) ]
1512 wait [ ( wait-argument ) ]
1513 copyin ( [ modifier-list : ] var-list )
1514 create ( [ modifier-list : ] var-list )
1515 attach ( var-list )

```

1516 In C and C++, the syntax of the OpenACC **exit data** directive is

1517 **#pragma acc exit data** *clause-list new-line*

1518 and in Fortran, the syntax is

1519 **!\$acc exit data** *clause-list*

1520 where *clause* is one of the following:

```

1521 if ( condition )
1522 async [ ( int-expr ) ]
1523 wait [ ( wait-argument ) ]
1524 copyout ( [ modifier-list : ] var-list )
1525 delete ( var-list )
1526 detach ( var-list )
1527 finalize

```

1528 **Description**

1529 At an **enter data** directive, data may be allocated in the current device memory and copied from
 1530 local memory to device memory. This action enters a data lifetime for those *vars*, and will make
 1531 the data available for **present** clauses on constructs within the data lifetime. Dynamic reference

1532 counters are incremented for this data, as described in Section 2.6.7 Reference Counters. Pointers
1533 in device memory may be *attached* to point to the corresponding device copy of the host pointer
1534 target.

1535 At an **exit data** directive, data may be copied from device memory to local memory and deal-
1536 located from device memory. If no **finalize** clause appears, dynamic reference counters are
1537 decremented for this data. If a **finalize** clause appears, the dynamic reference counters are set
1538 to zero for this data. Pointers in device memory may be *detached* so as to have the same value as
1539 the original host pointer.

1540 The data clauses are described in Section 2.7 Data Clauses. Reference counting behavior is de-
1541 scribed in Section 2.6.7 Reference Counters.

1542 **Restrictions**

- 1543 • At least one **copyin**, **create**, or **attach** clause must appear on an **enter data** direc-
1544 tive.
- 1545 • At least one **copyout**, **delete**, or **detach** clause must appear on an **exit data** direc-
1546 tive.

1547 **if clause**

1548 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or
1549 deallocate space in the current device memory and move data from and to local memory. When an
1550 **if** clause appears, the program will conditionally allocate or deallocate device memory and move
1551 data to and/or from device memory. When the *condition* in the **if** clause evaluates to *false*, no
1552 device memory will be allocated or deallocated, and no data will be moved. When the *condition*
1553 evaluates to *true*, the data will be allocated or deallocated and moved as specified.

1554 **async clause**

1555 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1556 **wait clause**

1557 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1558 **finalize clause**

1559 The **finalize** clause is allowed on the **exit data** directive and is optional. When no **finalize**
1560 clause appears, the **exit data** directive will decrement the dynamic reference counters for *vars*
1561 appearing in **copyout** and **delete** clauses, and will decrement the attachment counters for point-
1562 ers appearing in **detach** clauses. If a **finalize** clause appears, the **exit data** directive will
1563 set the dynamic reference counters to zero for *vars* appearing in **copyout** and **delete** clauses,
1564 and will set the attachment counters to zero for pointers appearing in **detach** clauses.

1565 **Errors**

- 1566 • See Section 2.7.3 for errors due to data clauses.
- 1567 • See Sections 2.16.1 and 2.16.2 for errors due to **async** or **wait** clauses.

1568 2.6.7 Reference Counters

1569 When device memory is allocated for data not in shared memory due to data clauses or OpenACC
1570 API routine calls, the OpenACC implementation keeps track of that section of device memory and
1571 its relationship to the corresponding data in host memory.

1572 Each section of device memory is associated with two *reference counters* per device, a structured
1573 reference counter and a dynamic reference counter. The structured and dynamic reference counters
1574 are used to determine when to allocate or deallocate data in device memory. The structured reference
1575 counter for a section of memory keeps track of how many nested data regions have been entered for
1576 that data. The initial value of the structured reference counter for static data in device memory (in a
1577 global **declare** directive) is one; for all other data, the initial value is zero. The dynamic reference
1578 counter for a section of memory keeps track of how many dynamic data lifetimes are currently active
1579 in device memory for that section. The initial value of the dynamic reference counter is zero. Data
1580 is considered *present* if the sum of the structured and dynamic reference counters is greater than
1581 zero.

1582 A structured reference counter is incremented when entering each data or compute region that con-
1583 tain an explicit data clause or implicitly-determined data attributes for that section of memory, and
1584 is decremented when exiting that region. A dynamic reference counter is incremented for each
1585 **enter data copyin** or **create** clause, or each **acc_copyin** or **acc_create** API routine
1586 call for that section of memory. The dynamic reference counter is decremented for each **exit**
1587 **data copyout** or **delete** clause when no **finalize** clause appears, or each **acc_copyout**
1588 or **acc_delete** API routine call for that section of memory. The dynamic reference counter will
1589 be set to zero with an **exit data copyout** or **delete** clause when a **finalize** clause ap-
1590 pears, or each **acc_copyout_finalize** or **acc_delete_finalize** API routine call for
1591 the section of memory. The reference counters are modified synchronously with the local thread,
1592 even if the data directives include an **async** clause. When both structured and dynamic reference
1593 counters reach zero, the data lifetime in device memory for that data ends.

1594 Memory mapped by **acc_map_data** may not have the associated dynamic reference count decre-
1595 mented to zero, except by a call to **acc_unmap_data**.

1596 2.6.8 Attachment Counter

1597 Since multiple pointers can target the same address, each pointer in device memory is associated
1598 with an *attachment counter* per device. The *attachment counter* for a pointer is initialized to zero
1599 when the pointer is allocated in device memory. The *attachment counter* for a pointer is set to one
1600 whenever the pointer is *attached* to new target address, and incremented whenever an *attach* action
1601 for that pointer is performed for the same target address. The *attachment counter* is decremented
1602 whenever a *detach* action occurs for the pointer, and the pointer is *detached* when the *attachment*
1603 *counter* reaches zero. This is described in more detail in Section 2.7.2 Data Clause Actions.

1604 A pointer in device memory can be assigned a device address in two ways. The pointer can be
1605 attached to a device address due to data clauses or API routines, as described in Section 2.7.2
1606 Data Clause Actions, or the pointer can be assigned in a compute region executed on that device.
1607 Unspecified behavior may result if both ways are used for the same pointer.

1608 Pointer members of structs, classes, or derived types in device or host memory can be overwritten
1609 due to update directives or API routines. It is the user's responsibility to ensure that the pointers
1610 have the appropriate values before or after the data movement in either direction. The behavior of

1611 the program is undefined if any of the pointer members are attached when an update of a composite
1612 variable is performed.

1613 2.7 Data Clauses

1614 Data clauses may appear on the **parallel** construct, **serial** construct, **kernels** construct,
1615 **data** construct, the **enter data** and **exit data** directives, and **declare** directives. In the
1616 descriptions, the *region* is a compute region with a clause appearing on a **parallel**, **serial**, or
1617 **kernels** construct, a data region with a clause on a **data** construct, or an implicit data region
1618 with a clause on a **declare** directive. If the **declare** directive appears in a global context,
1619 the corresponding implicit data region has a duration of the program. The list argument to each
1620 data clause is a comma-separated collection of *vars*. On a **declare** directive, the list argument
1621 of a **copyin**, **create**, **device_resident**, or **link** clause may include a Fortran *common*
1622 *block* name enclosed within slashes. On any directive, for any clause except **deviceptr** and
1623 **present**, the list argument may include a Fortran *common block* name enclosed within slashes
1624 if that *common block* name also appears in a **declare** directive **link** clause. In all cases, the
1625 compiler will allocate and manage a copy of the *var* in the memory of the current device, creating a
1626 visible device copy of that *var*, for data not in shared memory.

1627 OpenACC supports accelerators with discrete memories from the local thread. However, if the
1628 accelerator can access the local memory directly, the implementation may avoid the memory allo-
1629 cation and data movement and simply share the data in local memory unless an explicit copy in
1630 device-accessible memory is specified. Therefore, a program that uses and assigns data on the host
1631 and uses and assigns the same data on the accelerator within a data region without update directives
1632 to manage the coherence of the two copies may get different answers on different accelerators or
1633 implementations.

1634 Restrictions

- 1635 • Data clauses may not follow a **device_type** clause.
- 1636 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in data
1637 clauses.

1638 2.7.1 Data Specification in Data Clauses

1639 In C and C++, a subarray is an array name followed by an extended array range specification in
1640 brackets, with start and length, such as

1641 **AA[2:n]**

1642 If the lower bound is missing, zero is used. If the length is missing and the array has known size, the
1643 size of the array is used; otherwise the length is required. The subarray **AA[2:n]** means elements
1644 **AA[2], AA[3], ..., AA[2+n-1]**.

1645 In C and C++, a two dimensional array may be declared in at least four ways:

- 1646 • Statically-sized array: **float AA[100][200];**
- 1647 • Pointer to statically sized rows: **typedef float row[200]; row* BB;**
- 1648 • Statically-sized array of pointers: **float* CC[200];**
- 1649 • Pointer to pointers: **float** DD;**

1650 Each dimension may be statically sized, or a pointer to dynamically allocated memory. Each of
1651 these may be included in a data clause using subarray notation to specify a rectangular array:

- 1652 • **AA**[2:n][0:200]
- 1653 • **BB**[2:n][0:m]
- 1654 • **CC**[2:n][0:m]
- 1655 • **DD**[2:n][0:m]

1656 Multidimensional rectangular subarrays in C and C++ may be specified for any array with any com-
1657 bination of statically-sized or dynamically-allocated dimensions. For statically sized dimensions, all
1658 dimensions except the first must specify the whole extent to preserve the contiguous data restriction,
1659 discussed below. For dynamically allocated dimensions, the implementation will allocate pointers
1660 in device memory corresponding to the pointers in local memory and will fill in those pointers as
1661 appropriate.

1662 In Fortran, a subarray is an array name followed by a comma-separated list of range specifications
1663 in parentheses, with lower and upper bound subscripts, such as

1664 **arr(1:high, low:100)**

1665 If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if
1666 known, are used. All dimensions except the last must specify the whole extent, to preserve the
1667 contiguous data restriction, discussed below.

1668 Restrictions

- 1669 • In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be
1670 specified.
- 1671 • In C and C++, the length for dynamically allocated dimensions of an array must be explicitly
1672 specified.
- 1673 • In C and C++, modifying pointers in pointer arrays during the data lifetime, either on the host
1674 or on the device, may result in undefined behavior.
- 1675 • If a subarray appears in a data clause, the implementation may choose to allocate memory for
1676 only that subarray on the accelerator.
- 1677 • In Fortran, array pointers may appear, but pointer association is not preserved in device mem-
1678 ory.
- 1679 • Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous
1680 section of memory, except for dynamic multidimensional C arrays.
- 1681 • In C and C++, if a variable or array of composite type appears, all the data members of the
1682 struct or class are allocated and copied, as appropriate. If a composite member is a pointer
1683 type, the data addressed by that pointer are not implicitly copied.
- 1684 • In Fortran, if a variable or array of composite type appears, all the members of that derived
1685 type are allocated and copied, as appropriate. If any member has the **allocatable** or
1686 **pointer** attribute, the data accessed through that member are not copied.

- 1687 • If an expression is used in a subscript or subarray expression in a clause on a **data** construct,
1688 the same value is used when copying data at the end of the data region, even if the values of
1689 variables in the expression change during the data region.

1690 2.7.2 Data Clause Actions

1691 Data clauses perform one or more the following actions.

1692 Increment Counter Action

1693 An *increment counter* action is one of the actions that may be performed for a **present** (Section
1694 2.7.6), **copy** (Section 2.7.7), **copyin** (Section 2.7.8), **copyout** (Section 2.7.9), **create** (Sec-
1695 tion 2.7.10), **no_create** (Section 2.7.11), or **attach** (Section 2.7.13) clause, or for a call to an
1696 **acc_copyin**, **acc_create**, or **acc_attach** API routine (Sections 3.2.18 and 3.2.29). See
1697 those sections for details.

1698 An *increment counter* action for a *var* increments the structured or dynamic reference counter or
1699 the attachment counter for *var* by one.

1700 Decrement Counter Action

1701 A *decrement counter* action is one of the actions that may be performed for a **present** (Section
1702 2.7.6), **copy** (Section 2.7.7), **copyin** (Section 2.7.8), **copyout** (Section 2.7.9), **create** (Sec-
1703 tion 2.7.10), **no_create** (Section 2.7.11), **delete** (Section 2.7.12), **attach** (Section 2.7.13), or
1704 **detach** clause, or for a call to an **acc_copyout**, **acc_delete**, or **acc_detach** API routine
1705 (Sections 3.2.19 and ??). See those sections for details.

1706 A *decrement counter* action for a *var* decrements the structured or dynamic reference counter or
1707 the attachment counter for *var* by one. If the reference counter is already zero, its value is left
1708 unchanged.

1709 If the device memory associated with *var* was mapped to the device using **acc_map_data**, the
1710 dynamic reference count may not be decremented to zero, except by a call to **acc_unmap_data**.

1711 Reset Counter Action

1712 A *reset counter* action is one of the actions that may be performed for a **copyout** (Section 2.7.9),
1713 **delete** (Section 2.7.12), or **detach** (Section 2.7.14) clause, or for a call to an **acc_copyout**,
1714 **acc_delete**, or **acc_detach** API routine (Sections 3.2.19 and 3.2.29). See those sections for
1715 details.

1716 A *reset counter* action for a *var* sets the structured or dynamic reference counter or attachment
1717 counter for *var* to zero.

1718 Allocate Memory Action

1719 An *allocate memory* action is one of the actions that may be performed for a **copy** (Section 2.7.7),
1720 **copyin** (Section 2.7.8), **copyout** (Section 2.7.9) or **create** (Section 2.7.10) clause, or for a call
1721 to an **acc_copyin** or **acc_create** API routine (Section 3.2.18). See those sections for details.

1722 An *allocate memory* action for a *var* allocates device-accessible memory for *var*. If device memory
1723 is unavailable, shared memory is allocated. If shared memory is unavailable, device memory is

1724 allocated. When both shared and device memory are available, the choice of memory allocated is
1725 implementation-defined.

1726 **Deallocate Memory Action**

1727 A *deallocate memory* action is one of the actions that may be performed for a **copy** (Section 2.7.8),
1728 **copyin** (Section 2.7.8), **copyout** (Section 2.7.8), **create** (Section 2.7.10), **no_create** (Sec-
1729 tion 2.7.11), or **delete** (Section 2.7.12) clause, or for a call to an **acc_copyout** or **acc_delete**
1730 API routine (Section 3.2.19). See those sections for details.

1731 A *deallocate memory* action for *var* deallocates device-accessible memory for *var*.

1732 **Transfer In Action**

1733 A *transfer in* action is one of the actions that may be performed for a **copy** (Section 2.7.7) or
1734 **copyin** (Section 2.7.8) clause, **update** (Section 2.14.4) directive, or for a call to an **acc_copyin**
1735 or **acc_update_device** API routine (Sections 3.2.18 and 3.2.20). See those sections for details.

1736 A *transfer in* action for a *var* initiates a transfer of the data for *var* from the local thread memory to
1737 the corresponding device-accessible memory.

1738 The data copy may occur asynchronously, depending on other clauses on the directive.

1739 **Transfer Out Action**

1740 A *transfer out* action is one of the actions that may be performed for a **copy** (Section 2.7.7) or
1741 **copyout** (Section 2.7.9) clause, **update** (Section 2.14.4) directive, or for a call to an **acc_copyout**
1742 or **acc_update_self** API routine (Sections 3.2.19 and 3.2.20). See those sections for details.

1743 A *transfer out* action for a *var* initiates a transfer of the data for *var* from device-accessible memory
1744 to the corresponding local thread memory.

1745 The data copy may occur asynchronously, depending on other clauses on the directive, in which
1746 case the memory is deallocated when the data copy is complete.

1747 **Attach Pointer Action**

1748 An *attach pointer* action is one of the actions that may be performed for a **present** (Section
1749 2.7.6), **copy** (Section 2.7.7), **copyin** (Section 2.7.8), **copyout** (Section 2.7.9), **create** (Sec-
1750 tion 2.7.10), **no_create** (Section 2.7.11), or **attach** (Section 2.7.12) clause, or for a call to an
1751 **acc_attach** API routine (Section 3.2.29). See those sections for details.

1752 An *attach pointer* action for a *var* occurs only when *var* is a pointer reference.

1753 If the pointer *var* is in shared memory and it is not a captured variable or is not present in the current
1754 device-accessible memory, or if the address to which *var* points is not present in the current device-
1755 accessible memory, no action is taken. If the pointer is a null pointer, the pointer in device-accessible
1756 memory is updated to have the same value. Otherwise, the pointer in device-accessible memory is
1757 updated to point to the corresponding copy of the data. The update may occur asynchronously,
1758 depending on other clauses on the directive. The implementation schedules pointer updates after
1759 any data transfers due to *transfer in* actions that are performed for the same directive.

1760 Detach Pointer Action

1761 A *detach pointer* action is one of the actions that may be performed for a **present** (Section
1762 2.7.6), **copy** (Section 2.7.7), **copyin** (Section 2.7.8), **copyout** (Section 2.7.9), **create** (Sec-
1763 tion 2.7.10), **no_create** (Section 2.7.11), **delete** (Section 2.7.12), or **attach** (Section 2.7.13),
1764 or **detach** (Section 2.7.12) clause, or for a call to an **acc_detach** API routine (Section 3.2.29).
1765 See those sections for details.

1766 A *detach pointer* action for a *var* occurs only when *var* is a pointer reference.

1767 If the pointer *var* is in shared memory and is not a captured variable or is not present in the current
1768 device-accessible memory, or if the *attachment counter* for *var* for the pointer is not zero, no action
1769 is taken. The *var* in device-accessible memory is updated to have the same value as the correspond-
1770 ing pointer in local memory. The update may occur asynchronously, depending on other clauses
1771 on the directive. The implementation schedules pointer updates before any data transfers due to
1772 *transfer out* actions that are performed for the same directive.

1773 2.7.3 Data Clause Errors

1774 An error is issued for a *var* that appears in a **copy**, **copyin**, **copyout**, **create**, and **delete**
1775 clause as follows:

- 1776 • An **acc_error_partly_present** error is issued if part of *var* is present in device-
1777 accessible memory of the current device but all of *var* is not.
- 1778 • An **acc_error_invalid_data_section** error is issued if *var* is a Fortran subarray
1779 with a stride that is not one.
- 1780 • An **acc_error_out_of_memory** error is issued if the accelerator device does not have
1781 enough memory for *var*.

1782 An error is issued for a *var* that appears in a **present** clause as follows:

- 1783 • An **acc_error_not_present** error is issued if *var* is not present in the current device
1784 memory at entry to a data or compute construct.
- 1785 • An **acc_error_partly_present** error is issued if part of *var* is present in device-
1786 accessible memory of the current device but all of *var* is not.

1787 See Section 5.2.2.

1788 2.7.4 Data Clause Modifiers

1789 Some clauses allow an optional modifier list, with the following supported modifiers:

- 1790 • **always** indicating that the data *transfer in* and *transfer out* actions must always occur even
1791 if the data is present in the device.
- 1792 • **alwaysin** indicating that the data *transfer in* action must always occur even if the data is
1793 present in the device.
- 1794 • **alwaysout** indicating that the data *transfer out* action must always occur even if the data is
1795 present in the device.

- 1796 • **capture** indicating that the implementation must capture the variables in the clause with a
1797 distinct copy of such variables created in the device-accessible memory even if the original
1798 variable is already in accessible shared memory.
- 1799 • **readonly** indicating that the data in the data region are only read and not written.
- 1800 • **zero** indicating that the implementation must zero-initialise the variables in the clause.

1801 2.7.5 deviceptr clause

1802 The **deviceptr** clause may appear on structured **data** and compute constructs and **declare**
1803 directives.

1804 The **deviceptr** clause is used to declare that the pointers in *var-list* are device-accessible pointers,
1805 so the data need not be allocated or moved between the host and device for this pointer.

1806 In C and C++, the *vars* in *var-list* must be pointer variables.

1807 In Fortran, the *vars* in *var-list* must be dummy arguments (arrays or scalars), and may not have the
1808 Fortran **pointer**, **allocatable**, or **value** attributes.

1809 For data in shared memory, host pointers are the same as device pointers, so this clause has no
1810 effect.

1811 2.7.6 present clause

1812 The **present** clause may appear on structured **data** and compute constructs and **declare** di-
1813 rectives. The **present** clause specifies that *vars* in *var-list* are in shared memory or are already
1814 present in the current device memory due to data regions or data lifetimes that contain the construct
1815 on which the **present** clause appears.

1816 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is
1817 taken; otherwise, the **present** clause behaves as follows:

- 1818 • At entry to the region:
 - 1819 1. If *var* is a pointer reference,
 - 1820 a) If the attachment counter for *var* is zero, an *attach pointer* action is performed.
 - 1821 b) An *increment counter* action is performed with the associated attachment counter.
 - 1822 2. An *increment counter* action is performed with the associated structured reference counter.
- 1823 • At exit from the region:
 - 1824 1. If the structured reference counter for *var* is zero, no action is taken.
 - 1825 2. Otherwise,
 - 1826 a) If *var* is a pointer reference,
 - 1827 i. A *decrement counter* action is performed with the associated attachment counter.
 - 1828 ii. If the attachment counter for *var* is now zero, a *detach pointer* action is per-
1829 formed.

1830 b) A *decrement counter* action is performed with the associate structured reference
1831 counter.

1832 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1833 **2.7.7 copy clause**

1834 The **copy** clause may appear on structured **data** and compute constructs and on **declare** direc-
1835 tives.

1836 Only the following modifiers may appear in the optional *modifier-list*: *always*, *alwaysin*, *alwaysout*
1837 or *capture*.

1838 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable and has no
1839 **capture** modifier, no action is taken; otherwise, the **copy** clause behaves as follows:

- 1840 • At entry to the region:
 - 1841 1. If *var* is not present and is not a null pointer, an *allocate memory* action is performed. If
1842 a **zero** modifier appears, the memory is initialized to zero.
 - 1843 2. If *var* is not present or if an **always** or **alwaysin** modifier appears, a *transfer in*
1844 action is performed.
 - 1845 3. An *increment counter* action is performed with the associated structured reference counter.
 - 1846 4. If *var* is a pointer reference, an *attach pointer* action is performed, followed by an
1847 *increment counter* action on the associated attachment counter.
- 1848 • At exit from the region:
 - 1849 – If the structured reference counter for *var* is zero, no action is taken.
 - 1850 – Otherwise,
 - 1851 1. If *var* is a pointer reference, a *decrement counter* action is performed with the as-
1852 sociated attachment counter
 - 1853 2. If the associated attachment counter is now zero, a *detach pointer* action is per-
1854 formed.
 - 1855 3. A *decrement counter* action is performed with the structured associated reference
1856 counter.
 - 1857 4. If both structured and dynamic reference counters are now zero or if an **always**
1858 or **alwaysout** modifier appears, a *transfer out* action is performed.
 - 1859 5. If both structured and dynamic reference counters are now zero, a *deallocate memory*
1860 action is performed.

1861 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1862 For compatibility with OpenACC 2.0, **present_or_copy** and **pcopy** are alternate names for
1863 **copy**.

1864 2.7.8 copyin clause

1865 The **copyin** clause may appear on structured **data** and compute constructs, on **declare** direc-
1866 tives, and on **enter data** directives.

1867 Only the following modifiers may appear in the optional *modifier-list*: *always*, *alwaysin* or *readonly*.
1868 Additionally, on structured **data** and compute constructs *capture* modifier may appear.

1869 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable and has no
1870 **capture** modifier, no action is taken; otherwise, the **copyin** clause behaves as follows:

- 1871 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
1872 the dynamic reference counter is used.

- 1873 1. If *var* is not present and is not a null pointer, an *allocate memory* action is performed.
- 1874 2. If *var* is not present or if an **always** or **alwaysin** modifier appears, a *transfer in*
1875 action is performed.
- 1876 3. If *var* is a pointer reference, an *attach pointer* action is performed followed by an
1877 *increment counter* action with the associated attachment counter.
- 1878 4. An *increment counter* action is performed with the appropriate associated reference
1879 counter.

- 1880 • At exit from the region:

- 1881 – If the structured reference counter for *var* is zero, no action is taken.

- 1882 – Otherwise,

- 1883 1. If *var* is a pointer reference, a *decrement counter* action is performed on the asso-
1884 ciated attachment counter.
- 1885 2. If *var* is a pointer reference and the associated attachment counter is now zero, a
1886 *detach pointer* action is performed.
- 1887 3. A *decrement counter* action is performed with the associated structured reference
1888 counter.
- 1889 4. If both structured and dynamic reference counters are now zero, a *deallocate memory*
1890 action is performed.

1891 If the optional **readonly** modifier appears, then the implementation may assume that the data
1892 referenced by *var-list* is never written to within the applicable region.

1893 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1894 For compatibility with OpenACC 2.0, **present_or_copyin** and **pcopyin** are alternate names
1895 for **copyin**.

1896 An **enter data** directive with a **copyin** clause is functionally equivalent to a call to the **acc_copyin**
1897 API routine, as described in Section 3.2.18.

1898 2.7.9 copyout clause

1899 The **copyout** clause may appear on structured **data** and compute constructs, on **declare** di-
 1900 rectives, and on **exit data** directives. The clause may optionally have a **zero** modifier if the
 1901 **copyout** clause appears on a structured **data** or compute construct.

1902 Only the following modifiers may appear in the optional *modifier-list*: *always*, *alwayssin* or *zero*.
 1903 Additionally, on structured **data** and compute constructs *capture* modifier may appear.

1904 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable and has no
 1905 **capture** modifier, no action is taken; otherwise, the **copyout** clause behaves as follows:

- 1906 • At entry to a region:
 - 1907 1. If *var* is not present and is not a null pointer, an *allocate memory* action is performed. If
 1908 a **zero** modifier appears, the memory is initialized to zero.
 - 1909 2. If *var* is a pointer reference, an *attach pointer* action is performed, followed by an
 1910 *increment counter* action on the associated attachment counter.
 - 1911 3. An *increment counter* action is performed with the associated structured reference counter.
- 1912 • At exit from a region, the structured reference counter is used. On an **exit data** directive,
 1913 the dynamic reference counter is used.
 - 1914 – If the appropriate reference counter for *var* is zero, no action is taken.
 - 1915 – Otherwise,
 - 1916 1. If *var* is a pointer reference, a *decrement counter* action is performed on the asso-
 1917 ciated attachment counter.
 - 1918 2. If *var* is a pointer reference and the associated attachment counter is now zero, a
 1919 *detach pointer* action is performed.
 - 1920 3. The reference count is updated as follows:
 - 1921 * On an **exit data** directive with a **finalize** clause, a *reset counter* action
 1922 is performed to the dynamic reference.
 - 1923 * Otherwise, a *decrement counter* action is performed with the appropriate asso-
 1924 ciated reference counter.
 - 1925 4. If both structured and dynamic reference counters are now zero or an **always** or
 1926 **alwayssout** modifier appears, a *transfer out* action is performed.
 - 1927 5. If both structured and dynamic reference counters are now zero, a *deallocate memory*
 1928 action is performed.

1929 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1930 For compatibility with OpenACC 2.0, **present_or_copyout** and **pcopyout** are alternate
 1931 names for **copyout**.

1932 An **exit data** directive with a **copyout** clause and with or without a **finalize** clause is func-
 1933 tionally equivalent to a call to the **acc_copyout_finalize** or **acc_copyout** API routine,
 1934 respectively, as described in Section 3.2.19.

1935 2.7.10 create clause

1936 The **create** clause may appear on structured **data** and compute constructs, on **declare** direc-
1937 tives, and on **enter data** directives.

1938 Only the following modifiers may appear in the optional *modifier-list*: *zero*. Additionally, on struc-
1939 tured **data** and compute constructs *capture* modifier may appear.

1940 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable and has no
1941 **capture** modifier, no action is taken; otherwise, the **create** clause behaves as follows:

- 1942 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
1943 the dynamic reference counter is used.
 - 1944 1. If *var* is not present and is not a null pointer, an *allocate memory* action is performed. If
1945 a **zero** modifier appears, the memory is initialized to zero.
 - 1946 2. If *var* is a pointer reference, an *attach pointer* action is performed, followed by an
1947 *increment counter* action on the associated attachment counter.
 - 1948 3. An *increment counter* action is performed on the appropriate associated reference counter.
- 1949 • At exit from the region:
 - 1950 – If the structured reference counter for *var* is zero, no action is taken.
 - 1951 – Otherwise,
 - 1952 1. If *var* is a pointer reference, a *decrement counter* action is performed on the asso-
1953 ciated attachment counter.
 - 1954 2. If *var* is a pointer reference and the associated attachment counter is now zero, a
1955 *detach pointer* action is performed.
 - 1956 3. A *decrement counter* action is performed with the associated structured reference
1957 counter.
 - 1958 4. If both structured and dynamic reference counters are zero, a *deallocate memory*
1959 action is performed.

1960 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1961 For compatibility with OpenACC 2.0, **present_or_create** and **pcreate** are alternate names
1962 for **create**.

1963 An **enter data** directive with a **create** clause is functionally equivalent to a call to the **acc_create**
1964 API routine, as described in Section 3.2.18, except the directive may perform an *attach* action for a
1965 pointer reference.

1966 2.7.11 no_create clause

1967 The **no_create** clause may appear on structured **data** and compute constructs.

1968 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is
1969 taken; otherwise, the **no_create** clause behaves as follows:

- 1970 • At entry to the region:

- 1971 – If *var* is present and is not a null pointer, an *increment counter* action is performed with
 1972 the structured reference counter.
- 1973 – If *var* is present and is a pointer reference,
- 1974 1. an *increment counter* action is performed on the associated attachment counter,
 1975 2. and if the associated attachment counter is now one, an *attach pointer* action is
 1976 performed.
- 1977 – If *var* is not present, no action is performed, and any device code in this construct will
 1978 use the local memory address for *var*.
- 1979 • At exit from the region:
- 1980 – If the structured reference counter for *var* is zero or *var* is a null pointer, no action is
 1981 taken.
- 1982 – Otherwise,
- 1983 1. If *var* is a pointer reference,
- 1984 a) a *decrement counter* action is performed on the associated attachment counter,
 1985 b) and if the associated attachment counter is now zero, a *detach pointer* action is
 1986 performed.
- 1987 2. A *decrement counter* action is performed with the structured reference counter.
- 1988 3. If both structured and dynamic reference counters are zero, a *deallocate memory*
 1989 action is performed.

1990 2.7.12 delete clause

1991 The **delete** clause may appear on **exit data** directives.

1992 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is
 1993 taken; otherwise, the **delete** clause behaves as follows:

- 1994 • If the dynamic reference counter for *var* is zero, no action is taken.
- 1995 • Otherwise,
- 1996 1. If *var* is a pointer reference,
- 1997 a) a *decrement counter* action is performed on the associated attachment counter,
 1998 b) and if the associated attachment counter is now zero, a *detach pointer* action is
 1999 performed.
- 2000 2. If *var* is not a null pointer, the dynamic reference counter is updated, as follows:
- 2001 – On an **exit data** directive with a **finalize** clause, a *reset counter* action is
 2002 performed on the associated dynamic reference counter.
- 2003 – Otherwise, a *decrement counter* action is performed with the associated dynamic
 2004 reference counter.

2005 3. If both structured and dynamic reference counters are now zero, a *deallocate memory*
2006 action is performed.

2007 An **exit data** directive with a **delete** clause and with or without a **finalize** clause is func-
2008 tionally equivalent to a call to the **acc_delete_finalize** or **acc_delete** API routine, re-
2009 spectively, as described in Section 3.2.19.

2010 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

2011 **2.7.13 attach clause**

2012 The **attach** clause may appear on structured **data** and compute constructs and on **enter data**
2013 directives. Each *var* argument to an **attach** clause must be a C or C++ pointer or a Fortran variable
2014 or array with the **pointer** or **allocatable** attribute.

2015 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is
2016 taken; otherwise, the **attach** clause behaves as follows:

- 2017 • At entry to a region or at an **enter data** directive, an *attach pointer* action is performed
2018 followed by an *increment counter* action with the associated attachment counter.
- 2019 • At exit from the region,
 - 2020 1. a *decrement counter* action is performed with the associated attachment counter,
 - 2021 2. and if the associated attachment counter is now zero, a *detach pointer* action is per-
2022 formed.

2023 **2.7.14 detach clause**

2024 The **detach** clause may appear on **exit data** directives. Each *var* argument to a **detach** clause
2025 must be a C or C++ pointer or a Fortran variable or array with the **pointer** or **allocatable**
2026 attribute.

2027 For each *var* in *var-list*, if *var* is in shared memory and it is not a captured variable, no action is
2028 taken; otherwise, the **detach** clause behaves as follows:

- 2029 • If there is a **finalize** clause on the **exit data** directive, a *reset counter* action with the
2030 attachment counter is performed. Otherwise, a *decrement counter* action is performed with
2031 the associated attachment counter.
- 2032 • If the attachment counter is now zero, a *detach pointer* action is performed.

2033 **2.8 Host_Data Construct**

2034 **Summary**

2035 The **host_data** construct makes the address of data in device-accessible memory available on the
2036 host.

2037 **Syntax**

2038 In C and C++, the syntax of the OpenACC **host_data** construct is

```
2039       #pragma acc host_data clause-list new-line  
2040           structured block
```

2041 and in Fortran, the syntax is

```
2042     !$acc host_data clause-list
2043         structured block
2044     !$acc end host_data
```

2045 OR

```
2046     !$acc host_data clause-list
2047         block construct
2048     [!$acc end host_data]
```

2049 where *clause* is one of the following:

```
2050     use_device ( var-list )
2051     if ( condition )
2052     if_present
```

2053 Description

2054 This construct is used to make the address of data in device-accessible memory available in host
2055 code.

2056 Restrictions

- 2057 • A *var* in a **use_device** clause must be the name of a variable or array.
- 2058 • At least one **use_device** clause must appear.
- 2059 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
2060 value; in C or C++, the condition must evaluate to a scalar integer value.
- 2061 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
2062 **use_device** clauses.

2063 2.8.1 use_device clause

2064 The **use_device** clause tells the compiler to use device-accessible memory address of any *var* in
2065 *var-list* in code within the construct. In particular, this may be used to pass the device address of
2066 *var* to optimized procedures written in a lower-level API. If *var* is a null pointer, the same value is
2067 used for the device address. Otherwise, when there is no **if_present** clause, and either there is
2068 no **if** clause or the condition in the **if** clause evaluates to *true*, the *var* in *var-list* must be present
2069 in device-accessible memory due to data regions or data lifetimes that contain this construct. For
2070 data in shared memory which is not a captured variable, the device address is the same as the host
2071 address.

2072 2.8.2 if clause

2073 The **if** clause is optional. When an **if** clause appears and the condition evaluates to *false*, the
2074 compiler will not replace the addresses of any *var* in code within the construct. When there is no **if**
2075 clause, or when an **if** clause appears and the condition evaluates to *true*, the compiler will replace
2076 the addresses as described in the previous subsection.

2077 2.8.3 if_present clause

2078 When an **if_present** clause appears on the directive, the compiler will only replace the address
 2079 of any *var* which appears in *var-list* that is present in device-accessible memory for the current
 2080 device.

2081 2.9 Loop Construct

2082 Summary

2083 The OpenACC **loop** construct applies to a loop which must immediately follow this directive. The
 2084 **loop** construct can describe what type of parallelism to use to execute the loop and declare private
 2085 *vars* and reduction operations.

2086 Syntax

2087 In C and C++, the syntax of the **loop** construct is

```
2088     #pragma acc loop [clause-list] new-line
2089         for loop
```

2090 In Fortran, the syntax of the **loop** construct is

```
2091     !$acc loop [clause-list]
2092         do loop
```

2093 where *clause* is one of the following:

```
2094     collapse ( [force:] n )
2095     gang [ ( gang-arg-list ) ]
2096     worker [ ( [num:]int-expr ) ]
2097     vector [ ( [length:]int-expr ) ]
2098     seq
2099     independent
2100     auto
2101     tile ( size-expr-list )
2102     device_type ( device-type-list )
2103     private ( var-list )
2104     reduction ( operator:var-list )
```

2105 where *gang-arg* is one of:

```
2106     [num:]int-expr
2107     dim:int-expr
2108     static:size-expr
```

2109 and *gang-arg-list* may have at most one **num**, one **dim**, and one **static** argument, and where
 2110 *size-expr* is one of:

```
2111     *
2112     int-expr
```

2114 Some clauses are only valid in the context of a **kernels** construct; see the descriptions below.

2115 An *orphaned loop* construct is a **loop** construct that has no parent compute construct.

2116 A **loop** construct is *data-independent* if it has an **independent** clause that is determined explic-
 2117 itly, implicitly, or from an **auto** clause. A **loop** construct is *sequential* if it has a **seq** clause that
 2118 is determined explicitly or from an **auto** clause.

2119 When *do-loop* is a **do concurrent**, the OpenACC **loop** construct applies to the loop for each
 2120 index in the *concurrent-header*. The **loop** construct can describe what type of parallelism to use
 2121 to execute all the loops, and declares all indices appearing in the *concurrent-header* to be implicitly
 2122 private. If the **loop** construct that is associated with **do concurrent** is combined with a compute
 2123 construct then *concurrent-locality* is processed as follows: variables appearing in a *local* are treated
 2124 as appearing in a **private** clause; variables appearing in a *local_init* are treated as appearing in a
 2125 **firstprivate** clause; variables appearing in a *shared* are treated as appearing in a **copy** clause;
 2126 and a *default(none)* locality spec implies a **default (none)** clause on the compute construct. If
 2127 the **loop** construct is not combined with a compute construct, the behavior is implementation-
 2128 defined.

2129 Restrictions

- 2130 • Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **independent**, **auto**, and **tile**
 2131 clauses may follow a **device_type** clause.
- 2132 • The *int-expr* argument to the **worker** and **vector** clauses must be invariant in the kernels
 2133 region.
- 2134 • A loop associated with a **loop** construct that does not have a **seq** clause must be written to
 2135 meet all of the following conditions:
 - 2136 – The loop variable must be of integer, C/C++ pointer, or C++ random-access iterator
 2137 type.
 - 2138 – The loop variable must monotonically increase or decrease in the direction of its termi-
 2139 nation condition.
 - 2140 – The loop trip count must be computable in constant time when entering the **loop** con-
 2141 struct.

2142 For a C++ range-based **for** loop, the loop variable identified by the above conditions is the
 2143 internal iterator, such as a pointer, that the compiler generates to iterate the range. It is not the
 2144 variable declared by the **for** loop.

- 2145 • Only one of the **seq**, **independent**, and **auto** clauses may appear.
- 2146 • A **gang**, **worker**, or **vector** clause may not appear if a **seq** clause appears.
- 2147 • A **loop** construct with a **gang**, **worker**, or **vector** clause must not lexically enclose
 2148 another **loop** construct with a **gang**, **worker**, or **vector** clause specifying an equal or
 2149 higher level of parallelism unless the **loop** constructs have different parent compute scopes.
 2150 For example, in a loop nest that contains no interleaved compute constructs or procedures, a
 2151 **gang (dim: 1)** loop must not enclose a **gang (dim: 3)** loop or be enclosed by a **worker**
 2152 loop, but a **seq** loop is permitted at any nesting level.
- 2153 • At most one **gang** clause may appear on a **loop** construct.
- 2154 • A **tile** and **collapse** clause may not appear on **loop** that is associated with **do concurrent**.

2155 2.9.1 collapse clause

2156 The **collapse** clause is used to specify how many nested loops are associated with the **loop**
 2157 construct. The argument to the **collapse** clause must be a constant positive integer expression.
 2158 If no **collapse** clause appears, only the immediately following loop is associated with the **loop**
 2159 construct.

2160 If more than one loop is associated with the **loop** construct, the iterations of all the associated loops
 2161 are all scheduled according to the rest of the clauses. The trip count for all loops associated with
 2162 the **collapse** clause must be computable and invariant in all the loops. The particular integer
 2163 type used to compute the trip count for the collapsed loops is implementation defined. However, the
 2164 integer type used for the trip count has at least the precision of each loop variable of the associated
 2165 loops.

2166 It is implementation-defined whether a **gang**, **worker** or **vector** clause on the construct is ap-
 2167 plied to each loop, or to the linearized iteration space.

2168 The associated loops are the n nested loops that immediately follow the loop construct. If the
 2169 **force** modifier does not appear, then the associated loops must be tightly nested. If the **force**
 2170 modifier appears, then any intervening code may be executed multiple times as needed to perform
 2171 the collapse.

2172 Restrictions

- 2173 • Each associated loop, except the innermost, must contain exactly one loop or loop nest.
- 2174 • Intervening code must not contain other OpenACC directives or calls to API routines.

2175 2176 Examples

- 2177
- 2178 • In the code below, a compiler may choose to move the call to **tan** inside the inner loop in
 2179 order to collapse the two loops, resulting in redundant execution of the intervening code.

```

2180     #pragma acc parallel loop collapse(force:2)
2181     {
2182         for ( int i = 0; i < 360; i++ )
2183         {
2184             // This operation may be executed additional times in order
2185             // to perform the forced collapse.
2186             tanI = tan(a[i]);
2187             for ( int j = 0; j < N; j++ )
2188             {
2189                 // Do Something.
2190             }
2191         }
2192     }
  
```

2193

2194 2.9.2 gang clause

2195 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
 2196 the **gang** clause behaves as follows. It specifies that the iterations of the associated loop or loops
 2197 are to be executed in parallel by distributing the iterations among the gangs along the associated
 2198 dimension created by the compute construct. The associated dimension is the value of the **dim**
 2199 argument, if it appears, or is dimension one. The **dim** argument must be a constant positive integer
 2200 with value 1, 2, or 3. If the associated dimension is d , a **loop** construct with the **gang** clause
 2201 transitions a compute region from gang-redundant mode to gang-partitioned mode on dimension d
 2202 (GRd to GPd). The number of gangs in dimension d is controlled by the **parallel** construct; the
 2203 **num** argument is not allowed. The loop iterations must be data independent, except for *vars* which
 2204 appear in a **reduction** clause or which are modified in an atomic region.

2205 When the parent compute construct is a **kernels** construct, the **gang** clause behaves as follows.
 2206 It specifies that the iterations of the associated loop or loops are to be executed in parallel across the
 2207 gangs. The **dim** argument is not allowed. An argument with no keyword or with the **num** keyword
 2208 is allowed only when the **num_gangs** does not appear on the **kernels** construct. If an argument
 2209 with no keyword or an argument after the **num** keyword appears, it specifies how many gangs to use
 2210 to execute the iterations of this loop.

2211 The scheduling of loop iterations to gangs is not specified unless the **static** modifier appears as
 2212 an argument. If the **static** modifier appears with an integer expression, that expression is used
 2213 as a *chunk* size. If the **static** modifier appears with an asterisk, the implementation will select a
 2214 *chunk* size. The iterations are divided into chunks of the selected *chunk* size, and the chunks are
 2215 assigned to gangs starting with gang zero and continuing in round-robin fashion. Two **gang** loops
 2216 in the same parallel region with the same number of iterations, and with **static** clauses with the
 2217 same argument, will assign the iterations to gangs in the same manner. Two **gang** loops in the
 2218 same kernels region with the same number of iterations, the same number of gangs to use, and with
 2219 **static** clauses with the same argument, will assign the iterations to gangs in the same manner.

2220 A **gang(dim:1)** clause is implied on a data-independent **loop** construct without an explicit
 2221 **gang** clause if the following conditions hold while ignoring **gang**, **worker**, and **vector** clauses
 2222 on any sequential **loop** constructs and while treating implicit **routine** directives as if they are
 2223 explicit:

- 2224 • This **loop** construct's parent compute construct, if any, is not a **kernels** construct.
- 2225 • An explicit **gang(dim:1)** clause would be permitted on this **loop** construct. For example,
 2226 it must not conflict with a nested **loop** construct or an enclosing procedure's **routine**
 2227 directive, as specified in Sections 2.9 and 2.15.1.
- 2228 • For every lexically enclosing data-independent **loop** construct, either an explicit **gang(dim:1)**
 2229 clause would not be permitted on the enclosing **loop** construct, or the **loop** constructs have
 2230 different parent compute scopes.

2231 **Note:** An important consequence of the above specification is that, before implicitly determining
 2232 **gang** clauses on **loop** constructs, the implementation must analyze any **auto** clauses to determine
 2233 if **loop** constructs are sequential, and it must determine relevant implicit **routine** directives (see
 2234 the implicit **gang** clause example in Section 2.15.1).

2235 **Note:** As a performance optimization, the implementation might select different levels of paral-
 2236 lelism for a **loop** construct than specified by explicitly or implicitly determined clauses as long

2237 as it can prove program semantics are preserved. In particular, the implementation must consider
2238 semantic differences between gang-redundant and gang-partitioned mode. For example, in a series
2239 of tightly nested, data-independent **loop** constructs, implementations often move gang-partitioning
2240 from one **loop** construct to another without affecting semantics.

2241 **Note:** If the **auto** or **device_type** clause appears on a **loop** construct, it is the programmer's
2242 responsibility to ensure that program semantics are the same regardless of whether the **auto** clause
2243 is treated as **independent** or **seq** and regardless of the device type for which the program is
2244 compiled. In particular, the programmer must consider the effect on both explicitly and implicitly
2245 determined **gang** clauses and thus on gang-redundant and gang-partitioned mode. Examples in
2246 Sections 2.9.11 and 2.15.1 demonstrate how this issue for the **auto** clause might affect portability
2247 across OpenACC implementations.

2248 2.9.3 worker clause

2249 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
2250 the **worker** clause specifies that the iterations of the associated loop or loops are to be executed
2251 in parallel by distributing the iterations among the multiple workers within a single gang. A **loop**
2252 construct with a **worker** clause causes a gang to transition from worker-single mode to worker-
2253 partitioned mode. In contrast to the **gang** clause, the **worker** clause first activates additional
2254 worker-level parallelism and then distributes the loop iterations across those workers. No argu-
2255 ment is allowed. The loop iterations must be data independent, except for *vars* which appear in a
2256 **reduction** clause or which are modified in an atomic region.

2257 When the parent compute construct is a **kernels** construct, the **worker** clause specifies that the
2258 iterations of the associated loop or loops are to be executed in parallel across the workers within
2259 a single gang. An argument is allowed only when the **num_workers** does not appear on the
2260 **kernels** construct. The optional argument specifies how many workers per gang to use to execute
2261 the iterations of this loop.

2262 All workers will complete execution of their assigned iterations before any worker proceeds beyond
2263 the end of the loop.

2264 2.9.4 vector clause

2265 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
2266 the **vector** clause specifies that the iterations of the associated loop or loops are to be executed in
2267 vector or SIMD mode. A **loop** construct with a **vector** clause causes a worker to transition from
2268 vector-single mode to vector-partitioned mode. Similar to the **worker** clause, the **vector** clause
2269 first activates additional vector-level parallelism and then distributes the loop iterations across those
2270 vector lanes. The operations will execute using vectors of the length specified or chosen for the
2271 parallel region. The loop iterations must be data independent, except for *vars* which appear in a
2272 **reduction** clause or which are modified in an atomic region.

2273 When the parent compute construct is a **kernels** construct, the **vector** clause specifies that the
2274 iterations of the associated loop or loops are to be executed with vector or SIMD processing. An
2275 argument is allowed only when the **vector_length** does not appear on the **kernels** construct.
2276 If an argument appears, the iterations will be processed in vector strips of that length; if no argument
2277 appears, the implementation will choose an appropriate vector length.

2278 All vector lanes will complete execution of their assigned iterations before any vector lane proceeds
2279 beyond the end of the loop.

2280 2.9.5 seq clause

2281 The **seq** clause specifies that the associated loop or loops are to be executed sequentially by the
2282 accelerator. This clause will override any automatic parallelization or vectorization.

2283 2.9.6 independent clause

2284 The **independent** clause tells the implementation that the loop iterations must be data indepen-
2285 dent, except for *vars* which appear in a **reduction** clause or which are modified in an atomic
2286 region. This allows the implementation to generate code to execute the iterations in parallel with no
2287 synchronization.

2288 A **loop** construct with no **auto** or **seq** clause is treated as if it has the **independent** clause
2289 when it is an orphaned **loop** construct or its parent compute construct is a **parallel** construct.

2290 Note

- 2291 • It is likely a programming error to use the **independent** clause on a loop if any iteration
2292 writes to a variable or array element that any other iteration also writes or reads, except for
2293 *vars* which appear in a **reduction** clause or which are modified in an atomic region.
- 2294 • The implementation may be restricted in the levels of parallelism it can apply by the presence
2295 of **loop** constructs with **gang**, **worker**, or **vector** clauses for outer or inner loops.

2296 2.9.7 auto clause

2297 The **auto** clause specifies that the implementation must analyze the loop and determine whether the
2298 loop iterations are data-independent. If it determines that the loop iterations are data-independent,
2299 the implementation must treat the **auto** clause as if it is an **independent** clause. If not, or if it
2300 is unable to make a determination, it must treat the **auto** clause as if it is a **seq** clause, and it must
2301 ignore any **gang**, **worker**, or **vector** clauses on the loop construct.

2302 When the parent compute construct is a **kernels** construct, a **loop** construct with no **independent**
2303 or **seq** clause is treated as if it has the **auto** clause.

2304 **Note:** Combining the **auto** and **gang** clauses might impact a program's portability across Open-
2305 ACC implementations. See Section 2.9.2 for details.

2306 2.9.8 tile clause

2307 The **tile** clause specifies that the implementation will split each loop in the loop nest into two
2308 loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the **tile**
2309 clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression
2310 or an asterisk. If there are n tile sizes in the list, the **loop** construct must be immediately followed
2311 by n tightly-nested loops. The first argument in the *size-expr-list* corresponds to the innermost loop
2312 of the n associated loops, and the last element corresponds to the outermost associated loop. If the
2313 tile size is an asterisk, the implementation will choose an appropriate value. Each loop in the nest
2314 will be split, or *strip-mined*, into two loops, an outer *tile* loop and an inner *element* loop. The trip
2315 count of the element loop will be limited to the corresponding tile size from the *size-expr-list*. The

2316 *tile* loops will be reordered to be outside all the *element* loops, and the *element* loops will all be
2317 inside the *tile* loops.

2318 If the **vector** clause appears on the **loop** construct, the **vector** clause is applied to the *element*
2319 loops. If the **gang** clause appears on the **loop** construct, the **gang** clause is applied to the *tile*
2320 loops. If the **worker** clause appears on the **loop** construct, the **worker** clause is applied to the
2321 *element* loops if no **vector** clause appears, and to the *tile* loops otherwise.

2322 2.9.9 device_type clause

2323 The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

2324 2.9.10 private clause

2325 The **private** clause on a **loop** construct specifies that a copy of each item in *var-list* will be
2326 created. If the body of the loop is executed in *vector-partitioned* mode, a copy of the item is created
2327 for each thread associated with each vector lane. If the body of the loop is executed in *worker-*
2328 *partitioned vector-single* mode, a copy of the item is created for each worker and shared across the
2329 set of threads associated with all the vector lanes of that worker. Otherwise, a copy of the item is
2330 created for each gang in all dimensions and shared across the set of threads associated with all the
2331 vector lanes of all the workers of that gang.

2332 Restrictions

- 2333 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private**
2334 clauses.

2335 2336 Examples

- 2337
2338 • In the example below, **tmp** is private to each worker of every gang but shared across all the
2339 vector lanes of a worker.

```
2340     !$acc parallel
2341     !$acc loop gang
2342     do k = 1, n
2343     !$acc loop worker private(tmp)
2344     do j = 1, n
2345     !a single vector lane in each gang and worker assigns to tmp
2346     tmp = b(j,k) + c(j,k)
2347     !$acc loop vector
2348     do i = 1, n
2349     !all vector lanes use the result of the above update to tmp
2350     a(i,j,k) = a(i,j,k) + tmp/div
2351     enddo
2352     enddo
2353     enddo
2354     !$acc end parallel
```

- 2355 • In the example below, **tmp** is private to each gang in every dimension.

```

2356     !$acc parallel num_gangs(3, 50, 150)
2357     !$acc loop gang(dim:3)
2358     do k = 1, n
2359         !$acc loop gang(dim:2) private(tmp)
2360         do j = 1, n
2361             !all gangs along dimension 1 execute in gang redundant mode and
2362             !assign to tmp which is private to each gang in all dimensions
2363             tmp = b(j,k) + c(j,k)
2364             !$acc loop gang(dim:1)
2365             do i = 1, n
2366                 a(i,j,k) = a(i,j,k) + tmp/div
2367             enddo
2368         enddo
2369     enddo
2370 !$acc end parallel

```

2371



2372 2.9.11 reduction clause

2373 The **reduction** clause specifies a reduction operator and one or more *vars*. For each reduction
 2374 *var*, a private copy is created in the same manner as for a **private** clause on the **loop** construct,
 2375 and initialized for that operator; see the table in Section 2.5.15 reduction clause. After the loop, the
 2376 values for each thread are combined using the specified reduction operator, and the result combined
 2377 with the value of the original *var* and stored in the original *var*. If the original *var* is not private,
 2378 this update occurs by the end of the compute region, and any access to the original *var* is undefined
 2379 within the compute region. Otherwise, the update occurs at the end of the loop. If the reduction
 2380 *var* is an array or subarray, the reduction operation is logically equivalent to applying that reduction
 2381 operation to each array element of the array or subarray individually. If the reduction *var* is a com-
 2382 posite variable, the reduction operation is logically equivalent to applying that reduction operation
 2383 to each member of the composite variable individually.

2384 If a variable is involved in a reduction that spans multiple nested loops where two or more of those
 2385 loops have associated **loop** directives, a **reduction** clause containing that variable must appear
 2386 on each of those **loop** directives.

2387 Restrictions

- 2388 • A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name,
 2389 an array element, or a subarray (refer to Section 2.7.1).
- 2390 • Reduction clauses on nested constructs for the same reduction *var* must have the same reduc-
 2391 tion operator.
- 2392 • Every *var* in a **reduction** clause appearing on an orphaned **loop** construct must be private.
- 2393 • The restrictions for a **reduction** clause on a compute construct listed in in Section 2.5.15
 2394 reduction clause also apply to a **reduction** clause on a **loop** construct.
- 2395 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
 2396 **reduction** clauses.
- 2397 • See Section 2.6.2 Variables with Implicitly Determined Data Attributes for a restriction re-
 2398 quiring certain loop reduction variables to have explicit data clauses on their parent compute

2399 constructs.

- 2400 • A **reduction** clause may not appear on a **loop** directive that has a **gang** clause with a
- 2401 **dim:** argument whose value is greater than 1.
- 2402 • A **reduction** clause may not appear on a **loop** directive that has a **gang** clause and
- 2403 is within a compute construct that has a **num_gangs** clause with more than one explicit
- 2404 argument.

▼

2405 Examples

2407

- 2408 • **x** is not private at the **loop** directive below, so its reduction normally updates **x** at the end
- 2409 of the parallel region, where gangs synchronize. When possible, the implementation might
- 2410 choose to partially update **x** at the loop exit instead, or fully if **num_gangs (1)** were added
- 2411 to the **parallel** directive. However, portable applications cannot rely on such early up-
- 2412 dates, so accesses to **x** are undefined within the parallel region outside the loop.

```

2413     int x = 0;
2414     #pragma acc parallel copy(x)
2415     {
2416         // gang-shared x undefined
2417         #pragma acc loop gang worker vector reduction(+:x)
2418         for (int i = 0; i < I; ++i)
2419             x += 1; // vector-private x modified
2420         // gang-shared x undefined
2421     } // gang-shared x updated for gang/worker/vector reduction
2422     // x = I

```

- 2423 • **x** is private at each of the innermost two **loop** directives below, so each of their reductions
- 2424 updates **x** at the loop's exit. However, **x** is not private at the outer **loop** directive, so its
- 2425 reduction updates **x** by the end of the parallel region instead.

```

2426     int x = 0;
2427     #pragma acc parallel copy(x)
2428     {
2429         // gang-shared x undefined
2430         #pragma acc loop gang reduction(+:x)
2431         for (int i = 0; i < I; ++i) {
2432             #pragma acc loop worker reduction(+:x)
2433             for (int j = 0; j < J; ++j) {
2434                 #pragma acc loop vector reduction(+:x)
2435                 for (int k = 0; k < K; ++k) {
2436                     x += 1; // vector-private x modified
2437                 } // worker-private x updated for vector reduction
2438             } // gang-private x updated for worker reduction
2439         }
2440         // gang-shared x undefined
2441     } // gang-shared x updated for gang reduction
2442     // x = I * J * K

```

- 2443 • At each **loop** directive below, **x** is private and **y** is not private due to the data clauses on
 2444 the **parallel** directive. Thus, each reduction updates **x** at the loop exit, but each reduction
 2445 updates **y** by the end of the parallel region instead.

```

2446     int x = 0, y = 0;
2447     #pragma acc parallel firstprivate(x) copy(y)
2448     {
2449         // gang-private x = 0; gang-shared y undefined
2450         #pragma acc loop seq reduction(+:x,y)
2451         for (int i = 0; i < I; ++i) {
2452             x += 1; y += 2; // loop-private x and y modified
2453         } // gang-private x updated for trivial seq reduction
2454         // gang-private x = I; gang-shared y undefined
2455         #pragma acc loop worker reduction(+:x,y)
2456         for (int i = 0; i < I; ++i) {
2457             x += 1; y += 2; // worker-private x and y modified
2458         } // gang-private x updated for worker reduction
2459         // gang-private x = 2 * I; gang-shared y undefined
2460         #pragma acc loop vector reduction(+:x,y)
2461         for (int i = 0; i < I; ++i) {
2462             x += 1; y += 2; // vector-private x and y modified
2463         } // gang-private x updated for vector reduction
2464         // gang-private x = 3 * I; gang-shared y undefined
2465     } // gang-shared y updated for gang/seq/worker/vector reductions
2466     // x = 0; y = 3 * I * 2
  
```

- 2467 • The examples below are equivalent. That is, the **reduction** clause on the combined con-
 2468 struct applies to the **loop** construct but implies a **copy** clause on the parallel construct. Thus,
 2469 **x** is not private at the **loop** directive, so the reduction updates **x** by the end of the parallel
 2470 region.

```

2471     int x = 0;
2472     #pragma acc parallel loop worker reduction(+:x)
2473     for (int i = 0; i < I; ++i) {
2474         x += 1; // worker-private x modified
2475     } // gang-shared x updated for gang/worker reduction
2476     // x = I
2477
2478     int x = 0;
2479     #pragma acc parallel copy(x)
2480     {
2481         // gang-shared x undefined
2482         #pragma acc loop worker reduction(+:x)
2483         for (int i = 0; i < I; ++i) {
2484             x += 1; // worker-private x modified
2485         }
2486         // gang-shared x undefined
2487     } // gang-shared x updated for gang/worker reduction
2488     // x = I
  
```

- 2489 • If the implementation treats the **auto** clause below as **independent**, the loop executes in
 2490 gang-partitioned mode and thus examines every element of **arr** once to compute **arr**'s max-
 2491 imum. However, if the implementation treats **auto** as **seq**, the gangs redundantly compute

2492 **arr**'s maximum, but the combined result is still **arr**'s maximum. Either way, because **x** is
 2493 not private at the **loop** directive, the reduction updates **x** by the end of the parallel region.

```

2494     int x = 0;
2495     const int *arr = /*array of I values*/;
2496     #pragma acc parallel copy(x)
2497     {
2498         // gang-shared x undefined
2499         #pragma acc loop auto gang reduction(max:x)
2500         for (int i = 0; i < I; ++i) {
2501             // complex loop body
2502             x = x < arr[i] ? arr[i] : x; // gang- or loop-private
2503                                     // x modified
2504         }
2505         // gang-shared x undefined
2506     } // gang-shared x updated for gang or gang/seq reduction
2507     // x = arr maximum
  
```

2508 • The following example is the same as the previous one except that the reduction operator is
 2509 now **+**. While gang-partitioned mode sums the elements of **arr** once, gang-redundant mode
 2510 sums them once per gang, producing a result many times **arr**'s sum. This example shows
 2511 that, for some reduction operators, combining **auto**, **gang**, and **reduction** is typically
 2512 non-portable.

```

2513     int x = 0;
2514     const int *arr = /*array of I values*/;
2515     #pragma acc parallel copy(x)
2516     {
2517         // gang-shared x undefined
2518         #pragma acc loop auto gang reduction(+:x)
2519         for (int i = 0; i < I; ++i) {
2520             // complex loop body
2521             x += arr[i]; // gang or loop-private x modified
2522         }
2523         // gang-shared x undefined
2524     } // gang-shared x updated for gang or gang/seq reduction
2525     // x = arr sum possibly times number of gangs
  
```

2526 • At the following **loop** directive, **x** and **z** are private, so the loop reductions are not across
 2527 gangs even though the loop is gang-partitioned. Nevertheless, the **reduction** clause on the
 2528 **loop** directive is important as the loop is also vector-partitioned. These reductions are only
 2529 partial reductions relative to the full set of values computed by the loop, so the **reduction**
 2530 clause is needed on the **parallel** directive to reduce across gangs.

```

2531     int x = 0, y = 0;
2532     #pragma acc parallel copy(x) reduction(+:x,y)
2533     {
2534         int z = 0;
2535         #pragma acc loop gang vector reduction(+:x,z)
2536         for (int i = 0; i < I; ++i) {
2537             x += 1; z += 2; // vector-private x and z modified
2538         } // gang-private x and z updated for vector reduction
2539         y += z; // gang-private y modified
2540     } // gang-shared x and y updated for gang reduction
  
```

2541 // x = I; y = I * 2

2542

2543

2544 **2.10 Cache Directive**

2545 **Summary**

2546 The **cache** directive may appear at the top of (inside of) a loop. It suggests array elements or
2547 subarrays to be fetched into the highest level of the cache for the body of the loop.

2548 **Syntax**

2549 In C and C++, the syntax of the **cache** directive is

2550 **#pragma acc cache** ([**readonly**:] *var-list*) *new-line*

2551 In Fortran, the syntax of the **cache** directive is

2552 **!\$acc cache** ([**readonly**:] *var-list*)

2553 A *var* in a **cache** directive must be a single array element or a simple subarray. In C and C++,
2554 a simple subarray is an array name followed by an extended array range specification in brackets,
2555 with start and length, such as

2556 **arr** [*lower*:*length*]

2557 where the lower bound is a constant, loop invariant, or the **for** loop variable plus or minus a
2558 constant or loop invariant, and the length is a constant.

2559 In Fortran, a simple subarray is an array name followed by a comma-separated list of range specifi-
2560 cations in parentheses, with lower and upper bound subscripts, such as

2561 **arr** (*lower*:*upper*, *lower2*:*upper2*)

2562 The lower bounds must be constant, loop invariant, or the **do** loop variable plus or minus a constant
2563 or loop invariant; moreover the difference between the corresponding upper and lower bounds must
2564 be a constant.

2565 If the optional **readonly** modifier appears, then the implementation may assume that the data
2566 referenced by any *var* in that directive is never written to within the applicable region.

2567 **Restrictions**

- 2568 • If an array element or subarray is listed in a **cache** directive, all references to that array
2569 during execution of that loop iteration must not refer to elements of the array outside the
2570 index range specified in the **cache** directive.
- 2571 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **cache**
2572 directives.

2573 **2.11 Combined Constructs**

2574 **Summary**

2575 The combined OpenACC **parallel loop**, **serial loop**, and **kernels loop** constructs are
2576 shortcuts for specifying a **loop** construct nested immediately inside a **parallel**, **serial**, or

2577 **kernels** construct. The meaning is identical to explicitly specifying a **parallel**, **serial**, or
 2578 **kernels** construct containing a **loop** construct. Any clause that is allowed on a **parallel** or
 2579 **loop** construct is allowed on the **parallel loop** construct; any clause allowed on a **serial** or
 2580 **loop** construct is allowed on a **serial loop** construct; and any clause allowed on a **kernels**
 2581 or **loop** construct is allowed on a **kernels loop** construct.

2582 Syntax

2583 In C and C++, the syntax of the **parallel loop** construct is

```
2584     #pragma acc parallel loop [clause-list] new-line
2585         for loop
```

2586 In Fortran, the syntax of the **parallel loop** construct is

```
2587     !$acc parallel loop [clause-list]
2588         do loop
2589     [!$acc end parallel loop]
```

2590 The associated structured block is the loop which must immediately follow the directive. Any of
 2591 the **parallel** or **loop** clauses valid in a parallel region may appear.

2592 In C and C++, the syntax of the **serial loop** construct is

```
2593     #pragma acc serial loop [clause-list] new-line
2594         for loop
```

2595 In Fortran, the syntax of the **serial loop** construct is

```
2596     !$acc serial loop [clause-list]
2597         do loop
2598     [!$acc end serial loop]
```

2599 The associated structured block is the loop which must immediately follow the directive. Any of
 2600 the **serial** or **loop** clauses valid in a serial region may appear.

2601 In C and C++, the syntax of the **kernels loop** construct is

```
2602     #pragma acc kernels loop [clause-list] new-line
2603         for loop
```

2604 In Fortran, the syntax of the **kernels loop** construct is

```
2605     !$acc kernels loop [clause-list]
2606         do loop
2607     [!$acc end kernels loop]
```

2608 The associated structured block is the loop which must immediately follow the directive. Any of
 2609 the **kernels** or **loop** clauses valid in a kernels region may appear.

2610 A **private** or **reduction** clause on a combined construct is treated as if it appeared on the
 2611 **loop** construct. In addition, a **reduction** clause on a combined construct implies a **copy** clause
 2612 as described in Section 2.6.2.

2613 Restrictions

- 2614 • The restrictions for the **parallel**, **serial**, **kernels**, and **loop** constructs apply.

2.12 Atomic Construct

Summary

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

Syntax

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

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

OR:

```
#pragma acc atomic capture [ if( condition ) ] new-line
    structured block
```

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

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

```
v = x;
```

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

```
x = expr;
```

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

```
x++;
```

```
x--;
```

```
++x;
```

```
--x;
```

```
x binop= expr;
```

```
x = x binop expr;
```

```
x = expr binop x;
```

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

```
v = x++;
```

```
v = x--;
```

```
v = ++x;
```

```
v = --x;
```

```
v = x binop= expr;
```

```
v = x = x binop expr;
```

```
v = x = expr binop x;
```

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

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

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

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

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

```

2654   {x = x binop expr; v = x; }
2655   {x = expr binop x; v = x; }
2656   {v = x; x = expr; }
2657   {v = x; x++; }
2658   {v = x; ++x; }
2659   {++x; v = x; }
2660   {x++; v = x; }
2661   {v = x; x--; }
2662   {v = x; --x; }
2663   {--x; v = x; }
2664   {x--; v = x; }

```

2665 In the preceding expressions:

- 2666 • **x** and **v** (as applicable) are both l-value expressions with scalar type.
- 2667 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
2668 the same storage location.
- 2669 • Neither of **v** and *expr* (as applicable) may access the storage location designated by **x**.
- 2670 • Neither of **x** and *expr* (as applicable) may access the storage location designated by **v**.
- 2671 • *expr* is an expression with scalar type.
- 2672 • *binop* is one of +, *, -, /, &, ^, |, <<, or >>.
- 2673 • *binop*, *binop*=, ++, and -- are not overloaded operators.
- 2674 • The expression **x** *binop* *expr* must be mathematically equivalent to **x** *binop* (*expr*). This
2675 requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by
2676 using parentheses around *expr* or subexpressions of *expr*.
- 2677 • The expression *expr* *binop* **x** must be mathematically equivalent to (*expr*) *binop* **x**. This
2678 requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*,
2679 or by using parentheses around *expr* or subexpressions of *expr*.
- 2680 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is
2681 unspecified.

2682 In Fortran the syntax of the **atomic** constructs is:

```

2683   !$acc atomic read [ if ( condition ) ]
2684       capture-statement
2685   [ !$acc end atomic ]

```

2686 OR

```

2687   !$acc atomic write [ if ( condition ) ]
2688       write-statement
2689   [ !$acc end atomic ]

```

2690 OR

```

2691   !$acc atomic [update] [ if ( condition ) ]
2692       update-statement

```

```

2693     [!$acc end atomic]
2694 OR
2695     !$acc atomic capture [ if( condition ) ]
2696         update-statement
2697         capture-statement
2698     !$acc end atomic
2699 OR
2700     !$acc atomic capture [ if( condition ) ]
2701         capture-statement
2702         update-statement
2703     !$acc end atomic
2704 OR
2705     !$acc atomic capture [ if( condition ) ]
2706         capture-statement
2707         write-statement
2708     !$acc end atomic

```

2709 where *write-statement* has the following form (if *atomic-clause* is **write** or **capture**):

```
2710     x = expr
```

2711 where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

```
2712     v = x
```

2713 and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**,
2714 or no clause appears):

```

2715     x = x operator expr
2716     x = expr operator x
2717     x = intrinsic_procedure_name ( x, expr-list )
2718     x = intrinsic_procedure_name ( expr-list, x )

```

2719 In the preceding statements:

- 2720 • **x** and **v** (as applicable) are both scalar variables of intrinsic type.
- 2721 • **x** must not be an allocatable variable.
- 2722 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
2723 the same storage location.
- 2724 • None of **v**, *expr*, and *expr-list* (as applicable) may access the same storage location as **x**.
- 2725 • None of **x**, *expr*, and *expr-list* (as applicable) may access the same storage location as **v**.
- 2726 • *expr* is a scalar expression.
- 2727 • *expr-list* is a comma-separated, non-empty list of scalar expressions. If *intrinsic_procedure_name*
2728 refers to **iand**, **ior**, or **ieor**, exactly one expression must appear in *expr-list*.

- 2729 • *intrinsic_procedure_name* is one of **max**, **min**, **iand**, **ior**, or **ieor**. *operator* is one of **+**,
2730 *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**
 - 2731 • The expression **x operator expr** must be mathematically equivalent to **x operator (expr)**.
2732 This requirement is satisfied if the operators in *expr* have precedence greater than *operator*,
2733 or by using parentheses around *expr* or subexpressions of *expr*.
 - 2734 • The expression *expr operator x* must be mathematically equivalent to **(expr) operator x**.
2735 This requirement is satisfied if the operators in *expr* have precedence equal to or greater than
2736 *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
 - 2737 • *intrinsic_procedure_name* must refer to the intrinsic procedure name and not to other program
2738 entities.
 - 2739 • *operator* must refer to the intrinsic operator and not to a user-defined operator. All assign-
2740 ments must be intrinsic assignments.
 - 2741 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is
2742 unspecified.
- 2743 An **atomic** construct with the **read** clause forces an atomic read of the location designated by **x**.
2744 An **atomic** construct with the **write** clause forces an atomic write of the location designated by
2745 **x**.
- 2746 An **atomic** construct with the **update** clause forces an atomic update of the location designated
2747 by **x** using the designated operator or intrinsic. Note that when no clause appears, the semantics
2748 are equivalent to **atomic update**. Only the read and write of the location designated by **x** are
2749 performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect
2750 to the read or write of the location designated by **x**.
- 2751 An **atomic** construct with the **capture** clause forces an atomic update of the location designated
2752 by **x** using the designated operator or intrinsic while also capturing the original or final value of
2753 the location designated by **x** with respect to the atomic update. The original or final value of the
2754 location designated by **x** is written into the location designated by **v** depending on the form of the
2755 **atomic** construct structured block or statements following the usual language semantics. Only
2756 the read and write of the location designated by **x** are performed mutually atomically. Neither the
2757 evaluation of *expr* or *expr-list*, nor the write to the location designated by **v**, need to be atomic with
2758 respect to the read or write of the location designated by **x**.
- 2759 For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs
2760 enforces mutually exclusive access to the locations designated by **x**. To avoid race conditions, all
2761 accesses of the locations designated by **x** that could potentially occur in parallel must be protected
2762 with an **atomic** construct.
- 2763 Atomic regions do not guarantee exclusive access with respect to any accesses outside of atomic re-
2764 gions to the same storage location **x** even if those accesses occur during the execution of a reduction
2765 clause.
- 2766 If the storage location designated by **x** is not size-aligned (that is, if the byte alignment of **x** is not a
2767 multiple of the size of **x**), then the behavior of the atomic region is implementation-defined.
- 2768 The **if** clause specifies a condition where an atomic operation is required for correct parallel exe-
2769 cution. If *condition* evaluates to *true* or no **if** clause appears, the atomic operation is required. If

2770 *condition* evaluates to *false*, the atomic directive can be safely ignored. **Note:** Conditional atom-
 2771 ics are useful when different parallelism strategies are employed for different architectures; it is the
 2772 programmer's responsibility to ensure that the atomic operation is safe to ignore if *condition* is *false*.
 2773 Although not required, conditional atomics are recommended to be used with conditions that can
 2774 be evaluated at compile-time, including the **acc_on_device** routine.

2775 Restrictions

- 2776 • All atomic accesses to the storage locations designated by **x** throughout the program are
 2777 required to have the same type and type parameters.
- 2778 • Storage locations designated by **x** must be less than or equal in size to the largest available
 2779 native atomic operator width.
- 2780 • At most one **if** clause may appear.

2781 2.13 Declare Directive

2782 Summary

2783 A **declare** directive is used in the declaration section of a Fortran subroutine, function, block
 2784 construct, or module, or following a variable declaration in C or C++. It can specify that a *var* is to
 2785 be allocated in device memory for the duration of the implicit data region of a function, subroutine
 2786 or program, and specify whether the data values are to be transferred from local memory to device
 2787 memory upon entry to the implicit data region, and from device memory to local memory upon exit
 2788 from the implicit data region. These directives create a visible device copy of the *var*.

2789 Syntax

2790 In C and C++, the syntax of the **declare** directive is:

```
2791 #pragma acc declare clause-list new-line
```

2792 In Fortran the syntax of the **declare** directive is:

```
2793 !$acc declare clause-list
```

2794 where *clause* is one of the following:

```
2795 copy ( var-list )
2796 copyin ( [readonly:]var-list )
2797 copyout ( var-list )
2798 create ( var-list )
2799 present ( var-list )
2800 deviceptr ( var-list )
2801 device_resident ( var-list )
2802 link ( var-list )
```

2803 The associated region is the implicit region associated with the function, subroutine, or program in
 2804 which the directive appears. If the directive appears in the declaration section of a Fortran *module*
 2805 subprogram, for a Fortran *common block*, or in a C or C++ global or namespace scope, the associated
 2806 region is the implicit region for the whole program. The **copy**, **copyin**, **copyout**, **present**,
 2807 and **deviceptr** data clauses are described in Section 2.7 Data Clauses.

2808 **Restrictions**

- 2809 • A **declare** directive must be in the same scope as the declaration of any *var* that appears
- 2810 in the clauses of the directive or any scope within a C or C++ function or Fortran function,
- 2811 subroutine, or program.
- 2812 • At least one clause must appear on a **declare** directive.
- 2813 • A *var* in a **declare** declare must be a variable or array name, or a Fortran *common block*
- 2814 name between slashes.
- 2815 • A *var* may appear at most once in all the clauses of **declare** directives for a function,
- 2816 subroutine, program, or module.
- 2817 • In Fortran, assumed-size dummy arrays may not appear in a **declare** directive.
- 2818 • In Fortran, pointer arrays may appear, but pointer association is not preserved in device mem-
- 2819 ory.
- 2820 • In a Fortran *module* declaration section, only **create**, **copyin**, **device_resident**, and
- 2821 **link** clauses are allowed.
- 2822 • In Fortran, any **create** or **device_resident** clause affecting a variable with the *allo-*
- 2823 *catable* or *pointer* attribute must be visible at the allocation and deallocation of that variable.
- 2824 • In C or C++ global or namespace scope, only **create**, **copyin**, **deviceptr**,
- 2825 **device_resident** and **link** clauses are allowed.
- 2826 • C and C++ *extern* variables may only appear in **create**, **copyin**, **deviceptr**,
- 2827 **device_resident** and **link** clauses on a **declare** directive.
- 2828 • In C or C++, the **link** clause must appear at global or namespace scope or the arguments
- 2829 must be *extern* variables. In Fortran, the **link** clause must appear in a *module* declaration
- 2830 section, or the arguments must be *common block* names enclosed in slashes.
- 2831 • In C or C++, a **longjmp** call in the region must return to a **set jmp** call within the region.
- 2832 • In C++, an exception thrown in the region must be handled within the region.
- 2833 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional dummy arguments
- 2834 in data clauses, including **device_resident** clauses.

2835 **2.13.1 device_resident clause**2836 **Summary**

2837 The **device_resident** clause specifies that the memory for the named variables is allocated in
 2838 the current device memory and not in local memory. The host may not be able to access variables in
 2839 a **device_resident** clause. The accelerator data lifetime of global variables or common blocks
 2840 that appear in a **device_resident** clause is the entire execution of the program.

2841 In Fortran, if the variable has the Fortran *allocatable* attribute, the memory for the variable will
 2842 be allocated in and deallocated from the current device memory when the host thread executes
 2843 an **allocate** or **deallocate** statement for that variable, if the current device is a non-shared
 2844 memory device. If the variable has the Fortran *pointer* attribute, it may be allocated or deallocated

2845 by the host in the current device memory, or may appear on the left hand side of a pointer assignment
 2846 statement, if the right hand side variable itself appears in a **device_resident** clause.

2847 In Fortran, the argument to a **device_resident** clause may be a *common block* name enclosed
 2848 in slashes; in this case, all declarations of the common block must have a matching
 2849 **device_resident** clause. In this case, the *common block* will be statically allocated in de-
 2850 vice memory, and not in local memory. The *common block* will be available to accelerator routines;
 2851 see Section 2.15 Procedure Calls in Compute Regions.

2852 In a Fortran *module* declaration section, a *var* in a **device_resident** clause will be available to
 2853 accelerator subprograms.

2854 In C or C++ global scope, a *var* in a **device_resident** clause will be available to accelerator
 2855 routines. A C or C++ *extern* variable may appear in a **device_resident** clause only if the
 2856 actual declaration and all *extern* declarations are also followed by **device_resident** clauses.

2857 2.13.2 create clause

2858 For data in shared memory, no action is taken.

2859 For data not in shared memory, the **create** clause on a **declare** directive behaves as follows,
 2860 for each *var* in *var-list*:

- 2861 • At entry to an implicit data region where the **declare** directive appears:
 - 2862 – If *var* is present, a *present increment* action with the structured reference counter is
 2863 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 2864 – Otherwise, a *create* action with the structured reference counter is performed. If *var* is
 2865 a pointer reference, an *attach* action is performed.
- 2866 • At exit from an implicit data region where the **declare** directive appears:
 - 2867 – If the structured reference counter for *var* is zero, no action is taken.
 - 2868 – Otherwise, a *present decrement* action with the structured reference counter is per-
 2869 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
 2870 and dynamic reference counters are zero, a *delete* action is performed.

2871 If the **declare** directive appears in a global context, then the data in *var-list* is statically allocated
 2872 in device memory and the structured reference counter is set to one.

2873 In Fortran, if a variable *var* in *var-list* has the Fortran *allocatable* or *pointer* attribute, then for a
 2874 non-shared memory device:

- 2875 • For an **allocate** statement for *var* or an intrinsic assignment statement of *var* that will
 2876 allocate memory, memory will be allocated in both local memory as well as in the current
 2877 device memory and the dynamic reference counter will be set to one.
- 2878 • For a **deallocate** statement for *var* or an intrinsic assignment statement of *var* that will
 2879 deallocate memory, memory will be deallocated from both local memory as well as the current
 2880 device memory and the dynamic reference counter will be set to zero.
- 2881 • In Fortran, an intrinsic assignment statement that reallocates *var* behaves the same as a deal-
 2882 location followed by an allocation of *var*. **Note:** No update of device memory will occur as

2883 the result of an intrinsic assignment statement on the host; if data coherency between the host
2884 and device is required, it is the user's responsibility.

- 2885 • An **allocate**, **deallocate**, or intrinsic assignment statement on a device other than the
2886 host device will result in undefined behavior.
- 2887 • If the structured reference counter is not zero, a runtime error is issued.

2888 In Fortran, if a variable *var* in *var-list* has the Fortran *pointer* attribute, then it may appear on the
2889 left hand side of a pointer assignment statement, if the right hand side variable itself appears in a
2890 **create** clause.

2891 Errors

- 2892 • In Fortran, an **acc_error_present** error is issued at a deallocate statement if the struc-
2893 tured reference counter is not zero.

2894 See Section 5.2.2.

2895 2.13.3 link clause

2896 The **link** clause is used for large global host static data that is referenced within an accelerator
2897 routine and that has a dynamic data lifetime on the device. The **link** clause specifies that only a
2898 global link for the named variables is statically created in accelerator memory. The host data struc-
2899 ture remains statically allocated and globally available. The device data memory will be allocated
2900 only when the global variable appears on a data clause for a **data** construct, compute construct, or
2901 **enter data** directive. The arguments to the **link** clause must be global data. A **declare link**
2902 clause must be visible everywhere the global variables or common block variables are explicitly or
2903 implicitly used in a data clause, compute construct, or accelerator routine. The global variable or
2904 *common block* variables may be used in accelerator routines. The accelerator data lifetime of vari-
2905 ables or common blocks that appear in a **link** clause is the data region that allocates the variable or
2906 common block with a data clause, or from the execution of the **enter data** directive that allocates
2907 the data until an **exit data** directive deallocates it or until the end of the program.

2908 2.14 Executable Directives

2909 2.14.1 Init Directive

2910 Summary

2911 The **init** directive initializes the runtime for the given device or devices of the given device type.
2912 This can be used to isolate any initialization cost from the computational cost, when collecting
2913 performance statistics. If no device type appears all devices will be initialized. An **init** directive
2914 may be used in place of a call to the **acc_init** or **acc_init_device** runtime API routine, as
2915 described in Section 3.2.7.

2916 Syntax

2917 In C and C++, the syntax of the **init** directive is:

```
2918 #pragma acc init [clause-list] new-line
```

2919 In Fortran the syntax of the **init** directive is:

```
2920 !$acc init [clause-list]
```

2921 where *clause* is one of the following:

```
2922     device_type ( device-type-list )
2923     device_num ( int-expr )
2924     if ( condition )
```

2925

2926 **device_type clause**

2927 The **device_type** clause specifies the type of device that is to be initialized in the runtime. If the
 2928 **device_type** clause appears, then the *acc-current-device-type-var* for the current thread is set to
 2929 the argument value. If no **device_num** clause appears then all devices of this type are initialized.

2930 **device_num clause**

2931 The **device_num** clause specifies the device id to be initialized. If the **device_num** clause
 2932 appears, then the *acc-current-device-num-var* for the current thread is set to the argument value. If
 2933 no **device_type** clause appears, then the specified device id will be initialized for all available
 2934 device types.

2935 **if clause**

2936 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
 2937 perform the initialization unconditionally. When an **if** clause appears, the implementation will
 2938 generate code to conditionally perform the initialization only when the *condition* evaluates to *true*.

2939 **Restrictions**

- 2940 • This directive may only appear in code executed on the host.
- 2941 • If the directive is called more than once without an intervening **acc_shutdown** call or
 2942 **shutdown** directive, with a different value for the device type argument, the behavior is
 2943 implementation-defined.
- 2944 • If some accelerator regions are compiled to only use one device type, using this directive with
 2945 a different device type may produce undefined behavior.

2946 **Errors**

- 2947 • An **acc_error_device_type_unavailable** error is issued if a **device_type** clause
 2948 appears and no device of that device type is available, or if no **device_type** clause appears
 2949 and no device of the current device type is available.
- 2950 • An **acc_error_device_unavailable** error is issued if a **device_num** clause ap-
 2951 pears and the *int-expr* is not a valid device number or that device is not available, or if no
 2952 **device_num** clause appears and the current device is not available.
- 2953 • An **acc_error_device_init** error is issued if the device cannot be initialized.

2954 See Section 5.2.2.

2955 **2.14.2 Shutdown Directive**

2956 **Summary**

2957 The **shutdown** directive shuts down the connection to the given device or devices of the given
 2958 device type, and frees any associated runtime resources. This ends all data lifetimes in device
 2959 memory, which effectively sets structured and dynamic reference counters to zero. A **shutdown**
 2960 directive may be used in place of a call to the **acc_shutdown** or **acc_shutdown_device**
 2961 runtime API routine, as described in Section 3.2.8.

2962 **Syntax**

2963 In C and C++, the syntax of the **shutdown** directive is:

```
2964     #pragma acc shutdown [clause-list] new-line
```

2965 In Fortran the syntax of the **shutdown** directive is:

```
2966     !$acc shutdown [clause-list]
```

2967 where *clause* is one of the following:

```
2968     device_type ( device-type-list )
```

```
2969     device_num ( int-expr )
```

```
2970     if ( condition )
```

2971

2972 **device_type clause**

2973 The **device_type** clause specifies the type of device that is to be disconnected from the runtime.

2974 If no **device_num** clause appears then all devices of this type are disconnected.

2975 **device_num clause**

2976 The **device_num** clause specifies the device id to be disconnected.

2977 If no clauses appear then all available devices will be disconnected.

2978 **if clause**

2979 The **if** clause is optional; when there is no **if** clause, the implementation will generate code
 2980 to perform the shutdown unconditionally. When an **if** clause appears, the implementation will
 2981 generate code to conditionally perform the shutdown only when the *condition* evaluates to *true*.

2982 **Restrictions**

- 2983 • This directive may only appear in code executed on the host.

2984 **Errors**

- 2985 • An **acc_error_device_type_unavailable** error is issued if a **device_type** clause
 2986 appears and no device of that device type is available,
- 2987 • An **acc_error_device_unavailable** error is issued if a **device_num** clause ap-
 2988 pears and the *int-expr* is not a valid device number or that device is not available.
- 2989 • An **acc_error_device_shutdown** error is issued if there is an error shutting down the
 2990 device.

2991 See Section 5.2.2.

2.14.3 Set Directive

Summary

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

Syntax

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

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

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

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

where *clause* is one of the following

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

default_async clause

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

device_num clause

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

device_type clause

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

if clause

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

3029 **Restrictions**

- 3030 • This directive may only appear in code executed on the host.
- 3031 • Passing **default_async** the value of **acc_async_noval** has no effect.
- 3032 • Passing **default_async** the value of **acc_async_sync** will cause all asynchronous directives in the default asynchronous queue to become synchronous.
- 3033
- 3034 • Passing **default_async** the value of **acc_async_default** will restore the default asynchronous queue to the initial value, which is implementation-defined.
- 3035
- 3036 • At least one **default_async**, **device_num**, or **device_type** clause must appear.
- 3037 • Two instances of the same clause may not appear on the same directive.

3038 **Errors**

- 3039 • An **acc_error_device_type_unavailable** error is issued if a **device_type** clause appears, and no device of that device type is available.
- 3040
- 3041 • An **acc_error_device_unavailable** error is issued if a **device_num** clause appears, and the *int-expr* is not a valid device number.
- 3042
- 3043 • An **acc_error_invalid_async** error is issued if a **default_async** clause appears, and the *int-expr* is not a valid *async-argument*.
- 3044

3045 See Section 5.2.2.

3046 **2.14.4 Update Directive**3047 **Summary**

3048 The **update** directive is used during the lifetime of accelerator data to update *vars* in local memory with values from the corresponding data in device-accessible memory, or to update *vars* in device-accessible memory with values from the corresponding data in local memory.

3051 **Syntax**

3052 In C and C++, the syntax of the **update** directive is:

```
3053 #pragma acc update clause-list new-line
```

3054 In Fortran the syntax of the **update** data directive is:

```
3055 !$acc update clause-list
```

3056 where *clause* is one of the following:

```
3057 async [ ( int-expr ) ]
3058 wait [ ( wait-argument ) ]
3059 device_type ( device-type-list )
3060 if ( condition )
3061 if_present
3062 self ( var-list )
3063 host ( var-list )
3064 device ( var-list )
```

3065 Multiple subarrays of the same array may appear in a *var-list* of the same or different clauses on the
3066 same directive. For any *var* in *var-list* that is in shared memory and that is not a captured variable,
3067 no data action will occur. When a **device** clause appears, then for each *var* in the associated
3068 *var-list* an transfer in action is performed.

3069 When a **host** or **self** clause appears, then for each *var* in the associated *var-list* an transfer out
3070 action is performed.

3071 The transfer actions are performed in the order in which they appear on the directive, from left to
3072 right.

3073 **Restrictions**

- 3074 • At least one **self**, **host**, or **device** clause must appear on an **update** directive.

3075 **self clause**

3076 The **self** clause specifies that, for data not in shared memory or for captured variables, a *transfer out*
3077 action for the *vars* in *var-list* is performed. Otherwise, no action is taken.

3078 An **update** directive with the **self** clause is equivalent to a call to the **acc_update_self**
3079 routine, described in Section 3.2.20.

3080 **host clause**

3081 The **host** clause is a synonym for the **self** clause.

3082 **device clause**

3083 The **device** clause specifies that a *transfer in* action for the *vars* in *var-list* is performed for data
3084 not in shared memory or for the captured variables. Otherwise, no action is taken.

3085 An **update** directive with the **device** clause is equivalent to a call to the **acc_update_device**
3086 routine, described in Section 3.2.20.

3087 **if clause**

3088 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
3089 perform the updates unconditionally. When an **if** clause appears, the implementation will generate
3090 code to conditionally perform the updates only when the *condition* evaluates to *true*.

3091 **async clause**

3092 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

3093 **wait clause**

3094 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

3095 **if_present clause**

3096 When an **if_present** clause appears on the directive, no action is taken for a *var* which appears
3097 in *var-list* that is not present in the device-accessible memory of the current device.

3098 Restrictions

- 3099 • The **update** directive is executable. It must not appear in place of the statement following
3100 an *if*, *while*, *do*, *switch*, or *label* in C or C++, or in place of the statement following a logical
3101 *if* in Fortran.
- 3102 • If no **if_present** clause appears on the directive, each *var* in *var-list* must be present in
3103 the device-accessible memory of the current device.
- 3104 • Only the **async** and **wait** clauses may follow a **device_type** clause.
- 3105 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
3106 value; in C or C++, the condition must evaluate to a scalar integer value.
- 3107 • Noncontiguous subarrays may appear. It is implementation-specific whether noncontiguous
3108 regions are updated by using one transfer for each contiguous subregion, or whether the non-
3109 contiguous data is packed, transferred once, and unpacked, or whether one or more larger
3110 subarrays (no larger than the smallest contiguous region that contains the specified subarray)
3111 are updated.
- 3112 • In C and C++, a member of a struct or class may appear, including a subarray of a member.
3113 Members of a subarray of struct or class type may not appear.
- 3114 • In C and C++, if a subarray notation is used for a struct member, subarray notation may not
3115 be used for any parent of that struct member.
- 3116 • In Fortran, members of variables of derived type may appear, including a subarray of a mem-
3117 ber. Members of subarrays of derived type may not appear.
- 3118 • In Fortran, if array or subarray notation is used for a derived type member, array or subarray
3119 notation may not be used for a parent of that derived type member.
- 3120 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **self**,
3121 **host**, and **device** clauses.

3122 Errors

- 3123 • An **acc_error_not_present** error is issued if no **if_present** clause appears and
3124 any *var* in a **device** or **self** clause is not present on the current device.
- 3125 • An **acc_error_partly_present** error is issued if part of *var* is present in the current
3126 device memory but all of *var* is not.
- 3127 • An **async** or **wait** clause can cause an error to be issued; see Sections 2.16.1 and 2.16.2.

3128 See Section 5.2.2.

3129 2.14.5 Wait Directive

3130 See Section 2.16 Asynchronous Behavior for more information.

3131 2.14.6 Enter Data Directive

3132 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

3133 2.14.7 Exit Data Directive

3134 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

3135 2.15 Procedure Calls in Compute Regions

3136 This section describes how routines are compiled for an accelerator and how procedure calls are
 3137 compiled in compute regions. See Section 2.17.1 Optional Arguments for discussion of Fortran
 3138 optional arguments in procedure calls inside compute regions.

3139 2.15.1 Routine Directive

3140 Summary

3141 The **routine** directive is used to tell the compiler to compile the definition for a procedure, such
 3142 as a function or C++ lambda, for an accelerator as well as for the host. The **routine** directive is
 3143 also used to tell the compiler the attributes of the procedure when called on the accelerator.

3144 Syntax

3145 In C and C++, the syntax of the **routine** directive is:

```
3146     #pragma acc routine clause-list new-line
3147     #pragma acc routine( name ) clause-list new-line
```

3148 In C and C++, the **routine** directive without a name may appear immediately before a function
 3149 definition, a function prototype, or a C++ lambda and applies to the function or C++ lambda. The
 3150 **routine** directive with a name may appear anywhere that a function prototype is allowed and
 3151 applies to the function or the C++ lambda in scope with that name. See Section A.3.4 for recom-
 3152 mended diagnostics for a **routine** directive with a name.

3153 In Fortran the syntax of the **routine** directive is:

```
3154     !$acc routine clause-list
3155     !$acc routine( name ) clause-list
```

3156 In Fortran, the **routine** directive without a name may appear within the specification part of a
 3157 subroutine or function definition, or within an interface body for a subroutine or function in an
 3158 interface block, and applies to the containing subroutine or function. The **routine** directive with
 3159 a name may appear in the specification part of a subroutine, function or module, and applies to the
 3160 named subroutine or function.

3161 The *clause* is one of the following:

```
3162     gang [ ( dim:int-expr ) ]
3163     worker
3164     vector
3165     seq
3166     bind( name )
3167     bind( string )
3168     device_type( device-type-list )
3169     nohost
```

3170 A **gang**, **worker**, **vector**, or **seq** clause specifies the *level of parallelism* in the routine.

3171 A procedure compiled with the **routine** directive for an accelerator is called an *accelerator rou-*
3172 *tine*.

3173 If no explicit **routine** directive applies to a procedure whose definition appears in the program unit
3174 being compiled, then the implementation applies an implicit **routine** directive to that procedure
3175 if any of the following conditions holds:

- 3176 • The procedure is called or its address is accessed in a compute region.
- 3177 • The procedure is a C++ lambda defined in an accelerator routine that has a **nohost** clause,
3178 which is considered relevant below.
- 3179 • The procedure is a C++ lambda that is the parent compute scope of either:
 - 3180 – A **loop** construct. If it is data-independent, then its explicit **gang**, **worker**, and
3181 **vector** clauses are considered relevant below.
 - 3182 – A call to an accelerator routine whose **routine** directive has a **gang**, **worker**,
3183 **vector**, or **nohost** clause, each of which is considered relevant below.

3184 From the set containing **seq** and all relevant clauses identified above, the implicit **routine** direc-
3185 tive then copies any **nohost** clause and the highest level-of-parallelism clause.

3186 The implementation may apply predetermined **routine** directives with a **seq** clause to any pro-
3187 cedures that it provides for an accelerator, such as those of base language standard libraries.

3188 **Note:** Important consequences of the above specification are:

- 3189 • An implicit **routine** directive always has only a **seq** clause if the procedure is not a lambda.
- 3190 • Before determining an implicit **routine** directive for a lambda, the implementation must
3191 analyze **auto** clauses to determine if the lambda's orphaned **loop** constructs are data-
3192 independent (see the **auto** clause example later in this section).
- 3193 • When the implementation applies an implicit **routine** directive to a procedure, it must
3194 recursively apply implicit **routine** directives to other procedures for which the above rules
3195 specify relevant dependencies. Such dependencies can form a cycle, so the implementation
3196 must take care to avoid infinite recursion.

3197 **gang clause**

3198 The associated dimension is the value of the **dim** clause, if it appears, or is dimension one. The
3199 **dim** argument must be a constant positive integer with value 1, 2, or 3.

3200 The **gang** clause with dimension d specifies that the procedure can be the parent compute scope
3201 of a loop or a call to a routine with a **gang** clause associated with dimension d or less, but it must
3202 not be the parent compute scope of a loop or a call to a routine with a **gang** clause with dimension
3203 greater than d .

3204 **worker clause**

3205 The **worker** clause specifies that the procedure can be the parent compute scope of a loop or a call
3206 to a routine with a **worker** clause, but it must not be the parent compute scope of a loop or a call
3207 to a routine with a **gang** clause. A loop in this procedure with an **auto** clause may be selected by
3208 the compiler to execute in **worker** or **vector** mode. A call to this procedure must appear in code

3209 that is executed in *worker-single* mode, though it may be in *gang-redundant* or *gang-partitioned*
3210 mode. For instance, a procedure with a **routine worker** directive may be called from within a
3211 loop that has the **gang** clause, but not from within a loop that has the **worker** clause.

3212 **vector clause**

3213 The **vector** clause specifies that the procedure can be the parent compute scope of a loop or a
3214 call to a routine with a **vector** clause, but it must not be the parent compute scope of a loop or
3215 a call to a routine with a **gang** or **worker** clause. A loop in this procedure with an **auto** clause
3216 may be selected by the compiler to execute in **vector** mode, but not **worker** mode. A call to
3217 this procedure must appear in code that is executed in *vector-single* mode, though it may be in
3218 *gang-redundant* or *gang-partitioned* mode, and in *worker-single* or *worker-partitioned* mode. For
3219 instance, a procedure with a **routine vector** directive may be called from within a loop that has
3220 the **gang** clause or the **worker** clause, but not from within a loop that has the **vector** clause.

3221 **seq clause**

3222 The **seq** clause specifies that the procedure must not be the parent compute scope of a loop or a
3223 call to a routine with a **gang**, **worker**, or **vector** clause. A loop in this procedure with an **auto**
3224 clause will be executed in **seq** mode. A call to this procedure may appear in any mode.

3225 **bind clause**

3226 The **bind** clause specifies the name to use when calling the procedure on a device other than the
3227 host. If the name is specified as an identifier, it is called as if that name were specified in the
3228 language being compiled. If the name is specified as a string, the string is used for the procedure
3229 name unmodified. A **bind** clause on a procedure definition behaves as if it had appeared on a
3230 declaration by changing the name used to call the procedure on a device other than the host; however,
3231 the procedure is not compiled for the device with either the original name or the name in the **bind**
3232 clause.

3233 If there is both a Fortran bind and an acc **bind** clause for a procedure definition then a call on the
3234 host will call the Fortran bound name and a call on another device will call the name in the **bind**
3235 clause.

3236 **device_type clause**

3237 The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

3238 **nohost clause**

3239 The **nohost** clause tells the compiler not to compile a version of this procedure for the host.

3240 **Restrictions**

- 3241 • Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device_type**
3242 clause.
- 3243 • Exactly one of the **gang**, **worker**, **vector**, or **seq** clauses must appear.
- 3244 • In C and C++, function static variables are not supported in functions to which a **routine**
3245 directive applies.

- 3246 • In Fortran, variables with the *save* attribute, either explicitly or implicitly, are not supported
3247 in subprograms to which a **routine** directive applies.
- 3248 • A call to a procedure with a **nohost** clause must not appear in a compute construct that is
3249 compiled for the host. See examples below.
- 3250 • If a call to a procedure with a **nohost** clause appears in another procedure but outside any
3251 compute construct, that other procedure must also have a **nohost** clause.
- 3252 • A call to a procedure with a **gang(dim:d)** clause must appear in code that is executed
3253 in *gang-redundant* mode in all dimensions *d* and lower. For instance, a procedure with a
3254 **gang(dim:2)** clause may not be called from within a loop that has a **gang(dim:1)**
3255 or a **gang(dim:2)** clause. The user needs to ensure that a call to a procedure with a
3256 **gang(dim:d)** clause, when present in a region executing in *GR_e* or *GPe* mode with $e > d$
3257 and called by a gang along dimension *e*, is executed by all of its corresponding gangs along
3258 dimension *d*.
- 3259 • A **bind** clause may not bind to a routine name that has a visible **bind** clause.
- 3260 • If a procedure has a **bind** clause on both the declaration and the definition then they both
3261 must bind to the same name.
- 3262 • In C and C++, a definition or use of a procedure must appear within the scope of at least
3263 one explicit and applying **routine** directive if any appears in the same compilation unit.
3264 An explicit **routine** directive's scope is from the directive to the end of the compilation
3265 unit. If the **routine** directive appears in the member list of a C++ class, then its scope also
3266 extends in the same manner as any class member's scope (e.g., it includes the bodies of all
3267 other member functions).

3268 Examples

- 3271 • A function, such as **f** below, requires a **nohost** clause if it contains accelerator-specific code
3272 that cannot be compiled for the host. By default, some implementations compile all compute
3273 constructs for the host in addition to accelerators. In that case, a call to **f** must not appear in
3274 any compute construct or compilation will fail. However, **f** can appear in the **bind** clause of
3275 another function, such as **g** below, that does not have a **nohost** clause, and a call to **g** can
3276 appear in a compute construct. Thus, **g** is called when the compute construct is compiled for
3277 the host, and **f** is called when the compute construct is compiled for accelerators.

```

3278     #pragma acc routine seq nohost
3279     void f() { /*accelerator implementation*/ }
3280
3281     #pragma acc routine seq bind(f)
3282     void g() { /*host implementation*/ }
3283
3284     void h() {
3285         #pragma acc parallel
3286         g();
3287     }

```

- 3288 • In C, the restriction that a function's definitions and uses must appear within any applying
 3289 **routine** directive's scope has a simple interpretation: the **routine** directive must appear
 3290 first. This interpretation seems intuitive for the common case in C where prototypes, defini-
 3291 tions, and **routine** directives for a function, such as **f** below, appear at global scope.

```

3292     void f();
3293     void scopeA() {
3294         #pragma acc parallel
3295         f(); // nonconforming
3296     }
3297     // The routine directive's scope is not f's full scope.
3298     // Instead, it starts at the routine directive.
3299     #pragma acc routine(f) gang
3300     void scopeB() {
3301         #pragma acc parallel
3302         f(); // conforming
3303     }
3304     void f() {} // conforming
  
```

- 3305 • C++ classes permit forward references from member function bodies to other members de-
 3306 clared later. For example, immediately within **class A** below, **g**'s scope does not start until
 3307 after **f**'s definition. Nevertheless, within **f**'s body, **g** is in scope throughout. The same is true
 3308 for **g**'s **routine** directive. Thus, **f**'s call to **g** is conforming.

```

3309     class A {
3310         void f() {
3311             #pragma acc parallel
3312             g(); // conforming
3313         }
3314         #pragma acc routine gang
3315         void g();
3316     };
  
```

- 3317 • In some places, C++ classes do not permit forward references. For example, in the return type
 3318 of a member function, a member typedef that is declared later is not in scope. Likewise, **g**'s
 3319 definition below is not fully within the scope of **g**'s **routine** directive even though its body
 3320 is, so its definition is nonconforming.

```

3321     class A {
3322         #pragma acc routine(f) gang
3323         void f() {} // conforming
3324         void g() {} // nonconforming
3325         #pragma acc routine(g) gang
3326     };
  
```

- 3327 • The C++ scope resolution operator and **using** directive do not affect the scope of **routine**
 3328 directives. For example, the **routine** directive below is specified for the name **f**, which
 3329 resolves to **A::f**. Every reference to both **A::f** and **C::f** afterward is in the **routine**
 3330 directive's scope, but the **routine** directive always applies to **A::f** and never **C::f** even
 3331 when referenced as just **f**.

```

3332     namespace A {
3333         void f();
3334         namespace B {
  
```

```

3335         #pragma acc routine(f) gang // applies to A::f
3336     }
3337 }
3338 void g() {
3339     #pragma acc parallel
3340     A::f(); // conforming
3341 }
3342 void h() {
3343     using A::f;
3344     #pragma acc parallel
3345     f(); // conforming
3346 }
3347 namespace C {
3348     void f();
3349     using namespace A::B;
3350     void i() {
3351         #pragma acc parallel
3352         f(); // nonconforming
3353     }
3354 }

```

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

```

3363     // step 1: implicit #pragma acc routine vector
3364     auto f = []() {
3365         #pragma acc loop vector // step 2: no implicit gang clause
3366         for (int i = 0; i < I; ++i)
3367             ;
3368     };
3369
3370     #pragma acc routine gang
3371     void g();
3372
3373     // step 1: implicit #pragma acc routine gang
3374     auto h = []() {
3375         #pragma acc loop // step 2: implicit gang clause
3376         for (int i = 0; i < I; ++i)
3377             ;
3378         g();
3379     };

```

- As specified earlier in this section, before the implementation determines the implicit **routine** directive for a C++ lambda, it must analyze **auto** clauses on its orphaned **loop** constructs. This behavior can enable additional parallelism at the lambda’s call sites when the implementation cannot find parallelism within the lambda. For example, within **f** below, if the implementation treats **auto** as **seq**, then **f**’s implicit **routine** directive has a **seq** clause,

3385 which permits the implementation to worker- or vector-partition **h**'s **loop** construct. If the
 3386 implementation instead treats **f**'s **auto** as **independent**, then **f**'s implicit **routine** di-
 3387 rective has a **worker** clause, so the implementation cannot worker- or vector-partition **h**'s
 3388 **loop** construct.

```

3389 // step 2: implicit #pragma acc routine with seq or worker
3390 auto f = []() {
3391 // step 1: auto -> seq or independent
3392 #pragma acc loop auto worker vector
3393 for (int j = 0; j < J; ++j) {
3394 // complex loop body
3395 }
3396 };
3397
3398 #pragma acc routine seq
3399 void g();
3400
3401 void h() {
3402 #pragma acc parallel num_gangs(NG)
3403 // step 3: implicit gang, possibly worker or vector
3404 #pragma acc loop
3405 for (int i = 0; i < I; ++i) {
3406 f();
3407 g();
3408 }
3409 }

```

3410 When combining **auto** and **gang** on a **loop** construct within a lambda, the above behavior
 3411 might expose portability issues across implementations. For example, if the user adds an
 3412 explicit **gang** clause to **f**'s **loop** construct, then whether the implementation treats **f**'s **auto**
 3413 as **seq** or **independent** determines whether **f**'s implicit **routine** directive has a **seq**
 3414 or **gang** clause. That determines whether **h**'s **loop** construct has an implicit **gang** clause,
 3415 which determines how many times **g** is called: **I** times in gang-partitioned mode, or **NG*I**
 3416 times in gang-redundant mode.

- 3417 • By specifying a contract between a procedure and its callers, implicit **routine** directives
 3418 help to establish the semantics of OpenACC programs to facilitate both the user's under-
 3419 standing of the behavior and also the implementation's analysis and diagnostics. However,
 3420 as usual, the implementation is free to perform optimizations that preserve program seman-
 3421 tics. For example, the implicit **routine** directive for the C++ lambda **f** below has a **seq**
 3422 clause because **f**'s definition provides no means to determine a higher parallelism level and
 3423 because executing **f**'s **loop** constructs sequentially is compatible with any conceivable call
 3424 site. Nevertheless, observing that both of **f**'s **loop** constructs are data-independent and that
 3425 **g**'s call to **f** is in vector-single mode, the implementation might choose to inline a version of
 3426 **f** such that both **loop** constructs are vector-partitioned.

```

3427 // implicit #pragma acc routine seq
3428 auto f = []() {
3429 #pragma acc loop auto // auto -> independent
3430 for (int i = 0; i < I; ++i)
3431 ;
3432 #pragma acc loop // implicit independent
3433 for (int i = 0; i < I; ++i)

```

```

3434         ;
3435     };
3436     void g() {
3437         #pragma acc parallel loop gang worker
3438         for (int i = 0; i < I; ++i)
3439             f(); // can inline with vector partitioning
3440     }

```

3441



3442 2.15.2 Global Data Access

3443 C or C++ global, file static, or *extern* variables or array, and Fortran *module* or *common block* vari-
 3444 ables or arrays, that are used in accelerator routines must appear in a declare directive in a **create**,
 3445 **copyin**, **device_resident** or **link** clause. If the data appears in a **device_resident**
 3446 clause, the **routine** directive for the procedure must include the **nohost** clause. If the data ap-
 3447 pears in a **link** clause, that data must have an active accelerator data lifetime by virtue of appearing
 3448 in a data clause for a **data** construct, compute construct, or **enter data** directive.

3449 2.16 Asynchronous Behavior

3450 This section describes the **async** clause, the **wait** clause, the **wait** directive, and the behavior of
 3451 programs that use asynchronous data movement, compute regions, and asynchronous API routines.

3452 In this section and throughout the specification, the term *async-argument* means a nonnegative
 3453 scalar integer expression (*int* for C or C++, *integer* for Fortran), or one of the special values
 3454 **acc_async_noval** or **acc_async_sync**, as defined in the C header file and the Fortran
 3455 **openacc** module. The special values are negative values, so as not to conflict with a user-specified
 3456 nonnegative *async-argument*. An *async-argument* is used in **async** clauses, **wait** clauses, **wait**
 3457 directives, and as an argument to various runtime routines.

3458 The *async-value* of an *async-argument* is

- 3459 • **acc_async_sync** if *async-argument* has a value equal to the special value **acc_async_sync**,
- 3460 • the value of *acc-default-async-var* if *async-argument* has a value equal to the special value
3461 **acc_async_noval**,
- 3462 • the value of the *async-argument*, if it is nonnegative,
- 3463 • implementation-defined, otherwise.

3464 The *async-value* is used to select the activity queue to which the clause or directive or API routine
 3465 refers. The properties of the current device and the implementation will determine how many actual
 3466 activity queues are supported, and how the *async-value* is mapped onto the actual activity queues.
 3467 Two asynchronous operations on the same device with the same *async-value* will be enqueued
 3468 onto the same activity queue, and therefore will be executed on the device in the order they are
 3469 encountered by the local thread. Two asynchronous operations with different *async-values* may be
 3470 enqueued onto different activity queues, and therefore may be executed on the device in either order
 3471 or concurrently relative to each other. If there are two or more host threads executing and sharing the
 3472 same device, asynchronous operations on any thread with the same *async-value* will be enqueued
 3473 onto the same activity queue. If the threads are not synchronized with respect to each other, the
 3474 operations may be enqueued in either order and therefore may execute on the device in either order.

3475 Asynchronous operations enqueued to different devices may execute in any order or may execute
3476 concurrently, regardless of the *async-value* used for each.

3477 If a compute construct, data directive, or runtime API call has an *async-value* of **acc_async_sync**,
3478 the associated operations are executed on the activity queue associated with the *async-value*
3479 **acc_async_sync**, and the local thread will wait until the associated operations have completed
3480 before executing the code following the construct or directive. If a **data** construct has an *async-*
3481 *value* of **acc_async_sync**, the associated operations are executed on the activity queue associ-
3482 ated with the *async-value* **acc_async_sync**, and the local thread will wait until the associated
3483 operations that occur upon entry of the construct have completed before executing the code of the
3484 construct's structured block or block construct, and after that, will wait until the associated opera-
3485 tions that occur upon exit of the construct have completed before executing the code following the
3486 construct.

3487 If a compute construct, data directive, or runtime API call has an *async-value* other than
3488 **acc_async_sync**, the associated operations are executed on the activity queue associated with
3489 that *async-value* and the associated operations may be processed asynchronously while the local
3490 thread continues executing the code following the construct or directive. If a **data** construct has an
3491 *async-value* other than **acc_async_sync**, the associated operations are executed on the activity
3492 queue associated with that *async-value*, and the associated operations that occur upon entry of the
3493 construct may be processed asynchronously while the local thread continues executing the code
3494 of the construct's structured block or block construct, and after that, the associated operations that
3495 occur upon exit of the construct may be processed asynchronously while the local thread continues
3496 executing the code following the construct.

3497 In this section and throughout the specification, the term *wait-argument*, means:

3498 [**devnum** : *int-expr* :] [**queues** :] *async-argument-list*

3499 If a **devnum** modifier appears in the *wait-argument* then the associated device is the device with
3500 that device number of the current device type. If no **devnum** modifier appears then the associated
3501 device is the current device.

3502 Each *async-argument* is associated with an *async-value*. The *async-values* select the associated
3503 activity queue or queues on the associated device. If there is no *async-argument-list*, the associated
3504 activity queues are all activity queues for the associated device.

3505 The **queues** modifier within a *wait-argument* is optional to improve clarity of the expression list.

3506 2.16.1 **async clause**

3507 The **async** clause may appear on a **parallel**, **serial**, **kernels**, or **data** construct, or an
3508 **enter data**, **exit data**, **update**, or **wait** directive. In all cases, the **async** clause is optional.
3509 The **async** clause may have a single *async-argument*, as defined above. If the **async** clause does
3510 not appear, the behavior is as if the *async-argument* is **acc_async_sync**. If the **async** clause
3511 appears with no argument, the behavior is as if the *async-argument* is **acc_async_noval**. The
3512 *async-value* for a construct or directive is defined in Section 2.16.

3513 **Errors**

- 3514 • An **acc_error_invalid_async** error is issued if an **async** clause with an argument
3515 appears on any directive and the argument is not a valid *async-argument*.

3516 See Section 5.2.2.

2.16.2 wait clause

3517

3518 The **wait** clause may appear on a **parallel**, **serial**, or **kernels**, or **data** construct, or
 3519 an **enter data**, **exit data**, or **update** directive. In all cases, the **wait** clause is optional.
 3520 When there is no **wait** clause, the associated operations may be enqueued or launched or executed
 3521 immediately on the device.

3522 If there is an argument to the **wait** clause, it must be a *wait-argument*, the associated device and
 3523 activity queues are as specified in the *wait-argument*; see Section 2.16. If there is no argument to
 3524 the **wait** clause, the associated device is the current device and associated activity queues are all
 3525 activity queues. The associated operations may not be launched or executed until all operations
 3526 already enqueued up to this point by this thread on the associated asynchronous device activity
 3527 queues have completed. **Note:** One legal implementation is for the local thread to wait until the
 3528 operations already enqueued on the associated asynchronous device activity queues have completed;
 3529 another legal implementation is for the local thread to enqueue the associated operations in such a
 3530 way that they will not start until the operations already enqueued on the associated asynchronous
 3531 device activity queues have completed.

Errors

3532

- 3533 • An **acc_error_device_unavailable** error is issued if a **wait** clause appears on any
 3534 directive with a **devnum** modifier and the associated *int-expr* is not a valid device number.
- 3535 • An **acc_error_invalid_async** error is issued if a **wait** clause appears on any direc-
 3536 tive with a **queues** modifier or no modifier and any value in the associated list is not a valid
 3537 *async-argument*.

3538 See Section 5.2.2.

2.16.3 Wait Directive

3539

Summary

3540

3541 The **wait** directive causes the local thread or operations enqueued onto a device activity queue on
 3542 the current device to wait for completion of asynchronous operations.

Syntax

3543

3544 In C and C++, the syntax of the **wait** directive is:

```
3545 #pragma acc wait [ ( wait-argument ) ] [ clause-list ] new-line
```

3546 In Fortran the syntax of the **wait** directive is:

```
3547 !$acc wait [ ( wait-argument ) ] [ clause-list ]
```

3548 where *clause* is:

```
3549 async [ ( async-argument ) ]
3550 if ( condition )
```

3551 If it appears, the *wait-argument* is as defined in Section 2.16, and the associated device and activity
 3552 queues are as specified in the *wait-argument*. If there is no *wait-argument* clause, the associated
 3553 device is the current device and associated activity queues are all activity queues.

3554 If there is no **async** clause, the local thread will wait until all operations enqueued by this thread
 3555 onto each of the associated device activity queues for the associated device have completed. There

3556 is no guarantee that all the asynchronous operations initiated by other threads onto those queues will
3557 have completed without additional synchronization with those threads.

3558 If there is an **async** clause, no new operation may be launched or executed on the activity queue
3559 associated with the *async-argument* on the current device until all operations enqueued up to this
3560 point by this thread on the activity queues associated with the *wait-argument* have completed. **Note:**
3561 One legal implementation is for the local thread to wait for all the associated activity queues; another
3562 legal implementation is for the thread to enqueue a synchronization operation in such a way that
3563 no new operation will start until the operations enqueued on the associated activity queues have
3564 completed.

3565 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
3566 perform the wait operation unconditionally. When an **if** clause appears, the implementation will
3567 generate code to conditionally perform the wait operation only when the *condition* evaluates to *true*.

3568 A **wait** directive is functionally equivalent to a call to one of the **acc_wait**, **acc_wait_async**,
3569 **acc_wait_all**, or **acc_wait_all_async** runtime API routines, as described in Sections 3.2.10
3570 and 3.2.11.

3571 Errors

3572 • An **acc_error_device_unavailable** error is issued if a **devnum** modifier appears
3573 and the *int-expr* is not a valid device number.

3574 • An **acc_error_invalid_async** error is issued if a **queues** modifier or no modifier
3575 appears and any value in the associated list is not a valid *async-argument*.

3576 See Section 5.2.2.

3577 2.17 Fortran Specific Behavior

3578 2.17.1 Optional Arguments

3579 This section refers to the Fortran intrinsic function **PRESENT**. A call to the Fortran intrinsic function
3580 **PRESENT(arg)** returns **.true.**, if **arg** is an optional dummy argument and an actual argument
3581 for **arg** was present in the argument list of the call site. This is unrelated to the OpenACC **present**
3582 data clause.

3583 The appearance of a Fortran optional argument **arg** as a *var* in any of the following clauses has no
3584 effect at runtime if **PRESENT(arg)** is **.false.**:

- 3585 • in data clauses on compute and **data** constructs;
- 3586 • in data clauses on **enter data** and **exit data** directives;
- 3587 • in data and **device_resident** clauses on **declare** directives;
- 3588 • in **use_device** clauses on **host_data** directives;
- 3589 • in **self**, **host**, and **device** clauses on **update** directives.

3590 The appearance of a Fortran optional argument **arg** in the following situations may result in unde-
3591 fined behavior if **PRESENT(arg)** is **.false.** when the associated construct is executed:

- 3592 • as a *var* in **private**, **firstprivate**, and **reduction** clauses;
- 3593 • as a *var* in **cache** directives;

3594 • as part of an expression in any clause or directive.

3595 A call to the Fortran intrinsic function **PRESENT** behaves the same way in a compute construct or
3596 an accelerator routine as on the host. The function call **PRESENT (arg)** must return the same value
3597 in a compute construct as **PRESENT (arg)** would outside of the compute construct. If a Fortran
3598 optional argument **arg** appears as an actual argument in a procedure call in a compute construct
3599 or an accelerator routine, and the associated dummy argument **subarg** also has the **optional**
3600 attribute, then **PRESENT (subarg)** returns the same value as **PRESENT (subarg)** would when
3601 executed on the host.

3602 2.17.2 Do Concurrent Construct

3603 This section refers to the Fortran **do concurrent** construct that is a form of **do** construct. When
3604 **do concurrent** appears without a **loop** construct in a **kernels** construct it is treated as if it is
3605 annotated with **loop auto**. If it appears in a **parallel** construct or an accelerator routine then
3606 it is treated as if it is annotated with **loop independent**.

3. Runtime Library

3607

3608 This chapter describes the OpenACC runtime library routines that are available for use by program-
3609 mers. Use of these routines may limit portability to systems that do not support the OpenACC API.
3610 Conditional compilation using the `_OPENACC` preprocessor variable may preserve portability.

3611 This chapter has two sections:

- 3612 • Runtime library definitions
- 3613 • Runtime library routines

3614 There are four categories of runtime routines:

- 3615 • Device management routines, to get the number of devices, set the current device, and so on.
- 3616 • Asynchronous queue management, to synchronize until all activities on an async queue are
3617 complete, for instance.
- 3618 • Device test routine, to test whether this statement is executing on the device or not.
- 3619 • Data and memory management, to manage memory allocation or copy data between memo-
3620 ries.

3.1 Runtime Library Definitions

3621

3622 In C and C++, prototypes for the runtime library routines described in this chapter are provided in
3623 a header file named `openacc.h`. All the library routines are *extern* functions with “C” linkage.
3624 This file defines:

- 3625 • The prototypes of all routines in the chapter.
- 3626 • Any datatypes used in those prototypes, including an enumeration type to describe the sup-
3627 ported device types.
- 3628 • The values of `acc_async_noval`, `acc_async_sync`, and `acc_async_default`.

3629 In Fortran, interface declarations are provided in a Fortran module named `openacc`. The `openacc`
3630 module defines:

- 3631 • The integer parameter `openacc_version` with a value `yyyymm` where `yyyy` and `mm` are the
3632 year and month designations of the version of the Accelerator programming model supported.
3633 This value matches the value of the preprocessor variable `_OPENACC`.
- 3634 • Interfaces for all routines in the chapter.
- 3635 • Integer parameters to define integer kinds for arguments to and return values for those rou-
3636 tines.
- 3637 • Integer parameters to describe the supported device types.
- 3638 • Integer parameters to define the values of `acc_async_noval`, `acc_async_sync`, and
3639 `acc_async_default`.

3640 Many of the routines accept or return a value corresponding to the type of device. In C and C++, the
 3641 datatype used for device type values is `acc_device_t`; in Fortran, the corresponding datatype
 3642 is `integer(kind=acc_device_kind)`. The possible values for device type are implemen-
 3643 tation specific, and are defined in the C or C++ include file `openacc.h` and the Fortran module
 3644 `openacc`. Five values are always supported: `acc_device_none`, `acc_device_default`,
 3645 `acc_device_host`, `acc_device_not_host`, and `acc_device_current`. For other val-
 3646 ues, look at the appropriate files included with the implementation, or read the documentation for
 3647 the implementation. The value `acc_device_default` will never be returned by any function;
 3648 its use as an argument will tell the runtime library to use the default device type for that implemen-
 3649 tation.

3650 3.2 Runtime Library Routines

3651 In this section, for the C and C++ prototypes, pointers are typed `h_void*` or `d_void*` to desig-
 3652 nate a host memory address or device memory address, when these calls are executed on the host,
 3653 as if the following definitions were included:

```
3654     #define h_void void
3655     #define d_void void
```

3656 Many Fortran API bindings defined in this section rely on types defined in Fortran's `iso_c_binding`
 3657 module. It is implied that the `iso_c_binding` module is used in these bindings, even if not ex-
 3658 plicitly stated in the format section for that routine.

3659 Restrictions

3660 Except for `acc_on_device`, these routines are only available on the host.

3661 3.2.1 acc_get_num_devices

3662 Summary

3663 The `acc_get_num_devices` routine returns the number of available devices of the given type.

3664 Format

3665 C or C++:

```
3666     int acc_get_num_devices(acc_device_t dev_type);
```

3667 Fortran:

```
3668     integer function acc_get_num_devices(dev_type)
3669     integer(acc_device_kind) :: dev_type
```

3670 Description

3671 The `acc_get_num_devices` routine returns the number of available devices of device type
 3672 `dev_type`. If device type `dev_type` is not supported or no device of `dev_type` is available,
 3673 this routine returns zero.

3674 3.2.2 acc_set_device_type

3675 Summary

3676 The `acc_set_device_type` routine tells the runtime which type of device to use when exe-
 3677 cuting a compute region and sets the value of `acc-current-device-type-var`. This is useful when the
 3678 implementation allows the program to be compiled to use more than one type of device.

3679 Format

3680 C or C++:

```
3681     void acc_set_device_type(acc_device_t dev_type);
```

3682 Fortran:

```
3683     subroutine acc_set_device_type(dev_type)
3684         integer(acc_device_kind) :: dev_type
```

3685 Description

3686 A call to **acc_set_device_type** is functionally equivalent to a **set device_type (dev_type)**
3687 directive, as described in Section 2.14.3. This routine tells the runtime which type of device to use
3688 among those available and sets the value of *acc-current-device-type-var* for the current thread to
3689 **dev_type**.

3690 Restrictions

- 3691 • If some compute regions are compiled to only use one device type, the result of calling this
3692 routine with a different device type may produce undefined behavior.

3693 Errors

- 3694 • An **acc_error_device_type_unavailable** error is issued if device type **dev_type**
3695 is not supported or no device of **dev_type** is available.

3696 See Section 5.2.2.

3697 3.2.3 acc_get_device_type**3698 Summary**

3699 The **acc_get_device_type** routine returns the value of *acc-current-device-type-var*, which is
3700 the device type of the current device. This is useful when the implementation allows the program to
3701 be compiled to use more than one type of device.

3702 Format

3703 C or C++:

```
3704     acc_device_t acc_get_device_type(void);
```

3705 Fortran:

```
3706     function acc_get_device_type()
3707         integer(acc_device_kind) :: acc_get_device_type
```

3708 Description

3709 The **acc_get_device_type** routine returns the value of *acc-current-device-type-var* for the
3710 current thread to tell the program what type of device will be used to run the next compute region, if
3711 one has been selected. The device type may have been selected by the program with a runtime API
3712 call or a directive, by an environment variable, or by the default behavior of the implementation; see
3713 the table in Section 2.3.1.

3714 Restrictions

- 3715 • If the device type has not yet been selected, the value **acc_device_none** may be returned.

3716 3.2.4 `acc_set_device_num`

3717 Summary

3718 The `acc_set_device_num` routine tells the runtime which device to use and sets the value of
3719 `acc-current-device-num-var`.

3720 Format

3721 C or C++:

```
3722     void acc_set_device_num(int dev_num, acc_device_t dev_type);
```

3723 Fortran:

```
3724     subroutine acc_set_device_num(dev_num, dev_type)
```

```
3725         integer :: dev_num
```

```
3726         integer(acc_device_kind) :: dev_type
```

3727 Description

3728 A call to `acc_set_device_num` is functionally equivalent to a `set device_type(dev_type)`
3729 `device_num(dev_num)` directive, as described in Section 2.14.3. This routine tells the runtime
3730 which device to use among those available of the given type for compute or data regions in the cur-
3731 rent thread and sets the value of `acc-current-device-num-var` to `dev_num`. If the value of `dev_num`
3732 is negative, the runtime will revert to its default behavior, which is implementation-defined. If the
3733 value of the `dev_type` is zero, the selected device number will be used for all device types. Calling
3734 `acc_set_device_num` implies a call to `acc_set_device_type(dev_type)`.

3735 Errors

- 3736 • An `acc_error_device_type_unavailable` error is issued if device type `dev_type`
3737 is not supported or no device of `dev_type` is available.
- 3738 • An `acc_error_device_unavailable` error is issued if the value of `dev_num` is not
3739 a valid device number.

3740 See Section 5.2.2.

3741 3.2.5 `acc_get_device_num`

3742 Summary

3743 The `acc_get_device_num` routine returns the value of `acc-current-device-num-var` for the cur-
3744 rent thread.

3745 Format

3746 C or C++:

```
3747     int acc_get_device_num(acc_device_t dev_type);
```

3748 Fortran:

```
3749     integer function acc_get_device_num(dev_type)
```

```
3750         integer(acc_device_kind) :: dev_type
```

3751 Description

3752 The `acc_get_device_num` routine returns the value of `acc-current-device-num-var` for the cur-
3753 rent thread. If there are no devices of device type `dev_type` or if device type `dev_type` is not
3754 supported, this routine returns `-1`.

3755 **3.2.6 acc_get_property**3756 **Summary**

3757 The `acc_get_property` and `acc_get_property_string` routines return the value of a
 3758 *device-property* for the specified device.

3759 **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)
  
```

3761

```

    integer, value :: dev_num
  
```

3762

```

    integer(acc_device_kind), value :: dev_type
  
```

3763

```

    integer(acc_device_property_kind), value :: property
  
```

3764

```

    integer(c_size_t) :: acc_get_property
  
```

3765

```

    character*(*) :: string
  
```

3766

3767 **Description**

3768 The `acc_get_property` and `acc_get_property_string` routines return the value of the
 3769 *property*. `dev_num` and `dev_type` specify the device being queried. If `dev_type` has the
 3770 value `acc_device_current`, then `dev_num` is ignored and the value of the property for the
 3771 current device is returned. `property` is an enumeration constant, defined in `openacc.h`, for
 3772 C or C++, or an integer parameter, defined in the `openacc` module, for Fortran. Integer-valued
 3773 properties are returned by `acc_get_property`, and string-valued properties are returned by
 3774 `acc_get_property_string`. In Fortran, `acc_get_property_string` returns the result
 3775 into the `string` argument.

3776 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

3778 An implementation may support additional properties for some devices.

3779 **Restrictions**

- 3780 • **acc_get_property** will return 0 and **acc_get_property_string** will return a null
 3781 pointer (in C or C++) or a blank string (in Fortran) in the following cases:
- 3782 – If device type **dev_type** is not supported or no device of **dev_type** is available.
 - 3783 – If the value of **dev_num** is not a valid device number for device type **dev_type**.
 - 3784 – If the value of **property** is not one of the known values for that query routine, or that
 3785 property has no value for the specified device.

3786 **3.2.7 acc_init**3787 **Summary**

3788 The **acc_init** and **acc_init_device** routines initialize the runtime for the specified device
 3789 type and device number. This can be used to isolate any initialization cost from the computational
 3790 cost, such as when collecting performance statistics.

3791 **Format**

3792 C or C++:

```
3793 void acc_init(acc_device_t dev_type);
3794 void acc_init_device(int dev_num, acc_device_t dev_type);
```

3795 Fortran:

```
3796 subroutine acc_init(dev_type)
3797 subroutine acc_init_device(dev_num, dev_type)
3798 integer :: dev_num
3799 integer(acc_device_kind) :: dev_type
```

3800 **Description**

3801 A call to **acc_init** or **acc_init_device** is functionally equivalent to an **init** directive with
 3802 matching **dev_type** and **dev_num** arguments, as described in Section 2.14.1. **dev_type** must
 3803 be one of the defined accelerator types. **dev_num** must be a valid device number of the device type
 3804 **dev_type**. These routines also implicitly call **acc_set_device_type(dev_type)**. In the
 3805 case of **acc_init_device**, **acc_set_device_num(dev_num)** is also called.

3806 If a program initializes one or more devices without an intervening **shutdown** directive or
 3807 **acc_shutdown** call to shut down those same devices, no action is taken.

3808 **Errors**

- 3809 • An **acc_error_device_type_unavailable** error is issued if device type **dev_type**
 3810 is not supported or no device of **dev_type** is available.
- 3811 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3812 number.

3813 See Section 5.2.2.

3814 **3.2.8 acc_shutdown**

3815 Summary

3816 The **acc_shutdown** and **acc_shutdown_device** routines shut down the connection to spec-
3817 ified devices and free up any related resources in the runtime. This ends all data lifetimes in device
3818 memory for the device or devices that are shut down, which effectively sets structured and dynamic
3819 reference counters to zero.

3820 Format

3821 C or C++:

```
3822     void acc_shutdown(acc_device_t dev_type);  
3823     void acc_shutdown_device(int dev_num, acc_device_t dev_type);
```

3824 Fortran:

```
3825     subroutine acc_shutdown(dev_type)  
3826     subroutine acc_shutdown_device(dev_num, dev_type)  
3827         integer :: dev_num  
3828         integer(acc_device_kind) :: dev_type
```

3829 Description

3830 A call to **acc_shutdown** or **acc_shutdown_device** is functionally equivalent to a **shutdown**
3831 directive, with matching **dev_type** and **dev_num** arguments, as described in Section 2.14.2.
3832 **dev_type** must be one of the defined accelerator types. **dev_num** must be a valid device number
3833 of the device type **dev_type**. **acc_shutdown** routine disconnects the program from all devices
3834 of device type **dev_type**. The **acc_shutdown_device** routine disconnects the program from
3835 **dev_num** of type **dev_type**. Any data that is present in the memory of a device that is shut down
3836 is immediately deallocated.

3837 Restrictions

- 3838 • This routine may not be called while a compute region is executing on a device of type
3839 **dev_type**.
- 3840 • If the program attempts to execute a compute region on a device or to access any data in the
3841 memory of a device that was shut down, the behavior is undefined.
- 3842 • If the program attempts to shut down the **acc_device_host** device type, the behavior is
3843 undefined.

3844 Errors

- 3845 • An **acc_error_device_type_unavailable** error is issued if device type **dev_type**
3846 is not supported or no device of **dev_type** is available.
- 3847 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
3848 number.
- 3849 • An **acc_error_device_shutdown** error is issued if there is an error shutting down the
3850 device.

3851 See Section 5.2.2.

3852 3.2.9 acc_async_test

3853 Summary

3854 The **acc_async_test** routines test for completion of all associated asynchronous operations for
3855 a single specified async queue or for all async queues on the current device or on a specified device.

3856 **Format**

3857 C or C++:

```

3858     int acc_async_test(int wait_arg);
3859     int acc_async_test_device(int wait_arg, int dev_num);
3860     int acc_async_test_all(void);
3861     int acc_async_test_all_device(int dev_num);

```

3862 Fortran:

```

3863     logical function acc_async_test(wait_arg)
3864     logical function acc_async_test_device(wait_arg, dev_num)
3865     logical function acc_async_test_all()
3866     logical function acc_async_test_all_device(dev_num)
3867     integer(acc_handle_kind) :: wait_arg
3868     integer :: dev_num

```

3869 **Description**

3870 **wait_arg** must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior. **dev_num**
 3871 must be a valid device number of the current device type.

3872 The behavior of the **acc_async_test** routines is:

- 3873 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3874 • If any asynchronous operations initiated by this host thread on device **dev_num** either on
 3875 async queue **wait_arg** (if there is a **wait_arg** argument), or on any async queue (if there
 3876 is no **wait_arg** argument) have not completed, a call to the routine returns *false*.
- 3877 • If all such asynchronous operations have completed, or there are no such asynchronous op-
 3878 erations, a call to the routine returns *true*. A return value of *true* is no guarantee that asyn-
 3879 chronous operations initiated by other host threads have completed.

3880 **Errors**

- 3881 • An **acc_error_invalid_async** error is issued if **wait_arg** is not a valid *async-*
 3882 *argument* value.
- 3883 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3884 number.

3885 See Section 5.2.2.

3886 **3.2.10 acc_wait**3887 **Summary**

3888 The **acc_wait** routines wait for completion of all associated asynchronous operations on a single
 3889 specified async queue or on all async queues on the current device or on a specified device.

3890 **Format**

3891 C or C++:

```

3892     void acc_wait(int wait_arg);
3893     void acc_wait_device(int wait_arg, int dev_num);
3894     void acc_wait_all(void);
3895     void acc_wait_all_device(int dev_num);

```

3896 Fortran:

```
3897     subroutine acc_wait(wait_arg)
3898     subroutine acc_wait_device(wait_arg, dev_num)
3899     subroutine acc_wait_all()
3900     subroutine acc_wait_all_device(dev_num)
3901     integer(acc_handle_kind) :: wait_arg
3902     integer :: dev_num
```

3903 Description

3904 A call to an **acc_wait** routine is functionally equivalent to a **wait** directive as follows, see Section 2.16.3:

- 3906 • **acc_wait** to a **wait(wait_arg)** directive.
- 3907 • **acc_wait_device** to a **wait(devnum:dev_num, queues:wait_arg)** directive.
- 3908 • **acc_wait_all** to a **wait** directive with no *wait-argument*.
- 3909 • **acc_wait_all_device** to a **wait(devnum:dev_num)** directive.

3910 **wait_arg** must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior. **dev_num**

3911 must be a valid device number of the current device type.

3912 The behavior of the **acc_wait** routines is:

- 3913 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3914 • The routine will not return until all asynchronous operations initiated by this host thread on device **dev_num** either on async queue **wait_arg** (if there is a **wait_arg** argument) or on all async queues (if there is no **wait_arg** argument) have completed.
- 3917 • If two or more threads share the same accelerator, there is no guarantee that matching asynchronous operations initiated by other threads have completed.

3919 For compatibility with OpenACC version 1.0, **acc_wait** may also be spelled **acc_async_wait**,

3920 and **acc_wait_all** may also be spelled **acc_async_wait_all**.

3921 Errors

- 3922 • An **acc_error_invalid_async** error is issued if **wait_arg** is not a valid *async-*
- 3923 *argument* value.
- 3924 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
- 3925 number.

3926 See Section 5.2.2.

3927 3.2.11 acc_wait_async

3928 Summary

3929 The **acc_wait_async** routines enqueue a wait operation on one async queue of the current

3930 device or a specified device for the operations previously enqueued on a single specified async

3931 queue or on all other async queues.

3932 **Format**

C or C++:

```

3932 void acc_wait_async(int wait_arg, int async_arg);
3933 void acc_wait_device_async(int wait_arg, int async_arg,
3934                             int dev_num);
3935 void acc_wait_all_async(int async_arg);
3936 void acc_wait_all_device_async(int async_arg, int dev_num);

```

3936 Fortran:

```

3937 subroutine acc_wait_async(wait_arg, async_arg)
3938 subroutine acc_wait_device_async(wait_arg, async_arg, dev_num)
3939 subroutine acc_wait_all_async(async_arg)
3940 subroutine acc_wait_all_device_async(async_arg, dev_num)
3941 integer(acc_handle_kind) :: wait_arg, async_arg
3942 integer :: dev_num

```

3943 **Description**

3944 A call to an **acc_wait_async** routine is functionally equivalent to a **wait async (async_arg)**
 3945 directive as follows, see Section 2.16.3:

- 3946 • A call to **acc_wait_async** is functionally equivalent to a **wait (wait_arg)**
 3947 **async (async_arg)** directive.
- 3948 • A call to **acc_wait_device_async** is functionally equivalent to a **wait (devnum:**
 3949 **dev_num, queues:wait_arg) async (async_arg)** directive.
- 3950 • A call to **acc_wait_all_async** is functionally equivalent to a **wait async (async_arg)**
 3951 directive with no *wait-argument*.
- 3952 • A call to **acc_wait_all_device_async** is functionally equivalent to a
 3953 **wait (devnum:dev_num) async (async_arg)** directive.

3954 **async_arg** and **wait_arg** must be *async-arguments*, as defined in
 3955 Section 2.16 Asynchronous Behavior. **dev_num** must be a valid device number of the current
 3956 device type.

3957 The behavior of the **acc_wait_async** routines is:

- 3958 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3959 • The routine will enqueue a wait operation on the async queue associated with **async_arg**
 3960 for the current device which will wait for operations initiated on the async queue **wait_arg**
 3961 of device **dev_num** (if there is a **wait_arg** argument), or for each async queue of device
 3962 **dev_num** (if there is no **wait_arg** argument).

3963 See Section 2.16 Asynchronous Behavior for more information.

3964 **Errors**

- 3965 • An **acc_error_invalid_async** error is issued if either **async_arg** or **wait_arg** is
 3966 not a valid *async-argument* value.
- 3967 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3968 number.

3969 See Section 5.2.2.

3970 **3.2.12 acc_wait_any**3971 **Summary**

3972 The **acc_wait_any** and **acc_wait_any_device** routines wait for any of the specified asyn-
 3973 chronous queues to complete all pending operations on the current device or the specified device
 3974 number, respectively. Both routines return the queue's index in the provided array of asynchronous
 3975 queues.

3976 **Format**

3977 C or C++:

```
3978     int acc_wait_any(int count, int wait_arg[]);
3979     int acc_wait_any_device(int count, int wait_arg[], int dev_num);
```

3980 Fortran:

```
3981     integer function acc_wait_any(count, wait_arg)
3982     integer function acc_wait_any_device(count, wait_arg, dev_num)
3983     integer :: count, dev_num
3984     integer(acc_handle_kind) :: wait_arg(count)
```

3985 **Description**

3986 **wait_arg** is an array of *async-arguments* as defined in Section 2.16 and **count** is a nonneg-
 3987 ative integer indicating the array length. If there is no **dev_num** argument, it is treated as if
 3988 **dev_num** is the current device number. Otherwise, **dev_num** must be a valid device number
 3989 of the current device type. A call to any of these routines returns an index **i** associated with
 3990 a **wait_arg[i]** that is not **acc_async_sync** and meets the conditions that would evalu-
 3991 ate **acc_async_test_device(wait_arg[i], dev_num)** to *true*. If all the elements in
 3992 **wait_arg** are equal to **acc_async_sync** or **count** is equal to 0, these routines return -1.
 3993 Otherwise, the return value is an integer in the range of $0 \leq i < \mathbf{count}$ in C or C++ and
 3994 $1 \leq i \leq \mathbf{count}$ in Fortran.

3995 **Errors**

- 3996 • An **acc_error_invalid_argument** error is issued if **count** is a negative number.
- 3997 • An **acc_error_invalid_async** error is issued if any element encountered in **wait_arg**
 3998 is not a valid *async-argument* value.
- 3999 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 4000 number.

4001 See Section 5.2.2.

4002 **3.2.13 acc_get_default_async**4003 **Summary**

4004 The **acc_get_default_async** routine returns the value of *acc-default-async-var* for the cur-
 4005 rent thread.

4006 **Format**

4007 C or C++:

```
4008     int acc_get_default_async(void);
```

4009 Fortran:

```
4010     function acc_get_default_async()
4011         integer(acc_handle_kind) :: acc_get_default_async
```

4012 **Description**

4013 The **acc_get_default_async** routine returns the value of *acc-default-async-var* for the cur-
4014 rent thread, which is the asynchronous queue used when an **async** clause appears without an
4015 *async-argument* or with the value **acc_async_noval**.

4016 **3.2.14 acc_set_default_async**

4017 **Summary**

4018 The **acc_set_default_async** routine tells the runtime which asynchronous queue to use
4019 when an **async** clause appears with no queue argument.

4020 **Format**

4021 C or C++:

```
4022     void acc_set_default_async(int async_arg);
```

4023 Fortran:

```
4024     subroutine acc_set_default_async(async_arg)
4025         integer(acc_handle_kind) :: async_arg
```

4026 **Description**

4027 A call to **acc_set_default_async** is functionally equivalent to a **set default_async(async_arg)**
4028 directive, as described in Section 2.14.3. This **acc_set_default_async** routine tells the
4029 runtime to place any directives with an **async** clause that does not have an *async-argument* or
4030 with the special **acc_async_noval** value into the asynchronous activity queue associated with
4031 **async_arg** instead of the default asynchronous activity queue for that device by setting the value
4032 of *acc-default-async-var* for the current thread. The special argument **acc_async_default** will
4033 reset the default asynchronous activity queue to the initial value, which is implementation-defined.

4034 **Errors**

- 4035 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4036 *argument* value.

4037 See Section 5.2.2.

4038 **3.2.15 acc_on_device**

4039 **Summary**

4040 The **acc_on_device** routine tells the program whether it is executing on a particular device.

4041 **Format**

4042 C or C++:

```
4043     int acc_on_device(acc_device_t dev_type);
```

4044 Fortran:

```
4045     logical function acc_on_device(dev_type)
4046         integer(acc_device_kind) :: dev_type
```


4047 **Description**

4048 The `acc_on_device` routine may be used to execute different paths depending on whether the
 4049 code is running on the host or on some accelerator. If the `acc_on_device` routine has a compile-
 4050 time constant argument, the call evaluates at compile time to a constant. `dev_type` must be one
 4051 of the defined accelerator types.

4052 The behavior of the `acc_on_device` routine is:

- 4053 • If `dev_type` is `acc_device_host`, then outside of a compute region or accelerator rou-
 4054 tine, or in a compute region or accelerator routine that is executed on the host CPU, a call to
 4055 this routine will evaluate to *true*; otherwise, it will evaluate to *false*.
- 4056 • If `dev_type` is `acc_device_not_host`, the result is the negation of the result with
 4057 argument `acc_device_host`.
- 4058 • If `dev_type` is an accelerator device type, then in a compute region or routine that is ex-
 4059 ecuted on a device of that type, a call to this routine will evaluate to *true*; otherwise, it will
 4060 evaluate to *false*.
- 4061 • The result with argument `acc_device_default` is undefined.

4062 **3.2.16 acc_malloc**4063 **Summary**

4064 The `acc_malloc` routine allocates space in the current device memory.

4065 **Format**

4066 C or C++:

```
4067     d_void* acc_malloc(size_t bytes);
```

4068 Fortran:

```
4069     type(c_ptr) function acc_malloc(bytes)  

  4070     integer(c_size_t), value :: bytes
```

4071 **Description**

4072 The `acc_malloc` routine may be used to allocate space in the current device memory. Pointers
 4073 assigned from this routine may be used in `deviceptr` clauses to tell the compiler that the pointer
 4074 target is resident on the device. In case of an allocation error or if `bytes` has the value zero,
 4075 `acc_malloc` returns a null pointer.

4076 **3.2.17 acc_free**4077 **Summary**

4078 The `acc_free` routine frees memory on the current device.

4079 **Format**

4080 C or C++:

```
4081     void acc_free(d_void* data_dev);
```

4082 Fortran:

```
4083     subroutine acc_free(data_dev)  

  4084     type(c_ptr), value :: data_dev
```

4085 **Description**

4086 The **acc_free** routine will free previously allocated space in the current device memory; **data_dev**
 4087 must be a pointer value that was returned by a call to **acc_malloc** or a null pointer. If **data_dev**
 4088 is a null pointer, no operation is performed.

4089 **3.2.18 acc_copyin and acc_create**4090 **Summary**

4091 The **acc_copyin** and **acc_create** routines test to see if the argument is in shared memory or
 4092 already present in device-accessible memory of the current device; if not, they allocate space in
 4093 device-accessible memory of the current device to correspond to the specified local memory, and
 4094 the **acc_copyin** routines copy the data to that device-accessible memory.

4095 **Format**

4096 C or C++:

```
4097     d_void* acc_copyin(h_void* data_arg, size_t bytes);
4098     d_void* acc_create(h_void* data_arg, size_t bytes);
4099
4100     void acc_copyin_async(h_void* data_arg, size_t bytes,
4101                          int async_arg);
4102     void acc_create_async(h_void* data_arg, size_t bytes,
4103                          int async_arg);
4104
```

4105 Fortran:

```
4106     subroutine acc_copyin(data_arg [, bytes])
4107     subroutine acc_create(data_arg [, bytes])
4108
4109     subroutine acc_copyin_async(data_arg [, bytes], async_arg)
4110     subroutine acc_create_async(data_arg [, bytes], async_arg)
4111
4112     type(*), dimension(..) :: data_arg
4113     integer :: bytes
4114     integer(acc_handle_kind) :: async_arg
```

4115 **Description**

4116 A call to an **acc_copyin** or **acc_create** routine is similar to an **enter data** directive with
 4117 a **copyin** or **create** clause, respectively, as described in Sections 2.7.8 and 2.7.10, except that
 4118 no *attach pointer* action is performed for a pointer reference. In C/C++, **data_arg** is a pointer
 4119 to the data, and **bytes** specifies the data size in bytes; the associated *data section* starts at the
 4120 address in **data_arg** and continues for **bytes** bytes. The synchronous routines return a pointer
 4121 to the allocated device memory, as with **acc_malloc**. In Fortran, two forms are supported. In
 4122 the first, **data_arg** is a variable or a contiguous array section; the associated *data section* starts at
 4123 the address of, and continues to the end of the variable or array section. In the second, **data_arg**
 4124 is a variable or array element and **bytes** is the length in bytes; the associated *data section* starts
 4125 at the address of the variable or array element and continues for **bytes** bytes. For the **_async**
 4126 versions of these routines, **async_arg** must be an *async-argument* as defined in Section 2.16
 4127 Asynchronous Behavior.

4128 The behavior of these routines for the associated *data section* is:

- 4129 • If the *data section* is in shared memory and does not refer to a captured variable, no ac-
4130 tion is taken. The C/C++ synchronous **acc_copyin** and **acc_create** routines return the
4131 incoming pointer.
- 4132 • If the *data section* is present in device-accessible memory of the current device, the routines
4133 perform a *increment counter* action with the dynamic reference counter. The C/C++ syn-
4134 chronous **acc_copyin** and **acc_create** routines return a pointer to the existing device-
4135 accessible memory.
- 4136 • Otherwise:
 - 4137 – The **acc_copyin** routines behave as follows:
 - 4138 1. An *allocate memory* action is performed.
 - 4139 2. A *transfer in* action is performed.
 - 4140 3. A *increment counter* action with the dynamic reference counter is performed.
 - 4141 – The **acc_create** routines behave as follows:
 - 4142 1. An *allocate memory* action is performed.
 - 4143 2. A *increment counter* action with the dynamic reference counter is performed.

4144 The C/C++ synchronous **acc_copyin** and **acc_create** routines return a pointer to the
4145 newly allocated device memory.

4146 This data may be accessed using the **present** data clause. Pointers assigned from the C/C++
4147 synchronous **acc_copyin** and **acc_create** routines may be used in **deviceptr** clauses to
4148 tell the compiler that the pointer target is resident on the device.

4149 The synchronous versions will not return until the memory has been allocated and any data transfers
4150 are complete.

4151 The **_async** versions of these routines will perform any data transfers asynchronously on the async
4152 queue associated with **async_arg**. The routine may return before the data has been transferred;
4153 see Section 2.16 Asynchronous Behavior for more details. The data will be treated as present in
4154 device-accessible memory of the current device even if the data has not been allocated or transferred
4155 before the routine returns.

4156 For compatibility with OpenACC 2.0, **acc_present_or_copyin** and **acc_pcopyin** are al-
4157 ternate names for **acc_copyin**, and **acc_present_or_create** and **acc_pcreate** are al-
4158 ternate names for **acc_create**.

4159 Errors

- 4160 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
4161 **bytes** is nonzero.
- 4162 • An **acc_error_partly_present** error is issued if part of the *data section* is already
4163 present in device-accessible memory of the current device but all of the *data section* is not.
- 4164 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
4165 tion that is not contiguous (in Fortran).
- 4166 • An **acc_error_out_of_memory** error is issued if the accelerator device does not have
4167 enough memory for the data.

- 4168 • An `acc_error_invalid_async` error is issued if `async_arg` is not a valid *async-*
4169 *argument* value.

4170 See Section 5.2.2.

4171 3.2.19 `acc_copyout` and `acc_delete`

4172 Summary

4173 The `acc_copyout` and `acc_delete` routines test to see if the argument is in shared memory
4174 and does not refer to a captured variable; if not, the argument must be present in device-accessible
4175 memory of the current device. The `acc_copyout` routines copy data from device-accessible
4176 memory to the corresponding local memory, and both `acc_copyout` and `acc_delete` routines
4177 deallocate that space from the device-accessible memory.

4178 Format

4179 C or C++:

```
4180 void acc_copyout(h_void* data_arg, size_t bytes);
4181 void acc_delete (h_void* data_arg, size_t bytes);
4182
4183 void acc_copyout_finalize(h_void* data_arg, size_t bytes);
4184 void acc_delete_finalize (h_void* data_arg, size_t bytes);
4185
4186 void acc_copyout_async(h_void* data_arg, size_t bytes,
4187                       int async_arg);
4188 void acc_delete_async (h_void* data_arg, size_t bytes,
4189                      int async_arg);
4190
4191 void acc_copyout_finalize_async(h_void* data_arg, size_t bytes,
4192                               int async_arg);
4193 void acc_delete_finalize_async (h_void* data_arg, size_t bytes,
4194                               int async_arg);
4195
```

4196 Fortran:

```
4197 subroutine acc_copyout (data_arg [, bytes])
4198 subroutine acc_delete (data_arg [, bytes])
4199
4200 subroutine acc_copyout_finalize (data_arg [, bytes])
4201 subroutine acc_delete_finalize (data_arg [, bytes])
4202
4203 subroutine acc_copyout_async (data_arg [, bytes], async_arg)
4204 subroutine acc_delete_async (data_arg [, bytes], async_arg)
4205
4206 subroutine acc_copyout_finalize_async (data_arg [, bytes], &
4207                                       async_arg)
4208 subroutine acc_delete_finalize_async (data_arg [, bytes], &
4209                                       async_arg)
4210
4211 type(*), dimension(..) :: data_arg
```

```

4212     integer :: bytes
4213     integer(acc_handle_kind) :: async_arg

```

4214 Description

4215 A call to an **acc_copyout** or **acc_delete** routine is similar to an **exit data** directive
 4216 with a **copyout** or **delete** clause, respectively, and a call to an **acc_copyout_finalize**
 4217 or **acc_delete_finalize** routine is similar to an **exit data finalize** directive with a
 4218 **copyout** or **delete** clause, respectively, as described in Section 2.7.9 and 2.7.12, except that no
 4219 *detach pointer* action is performed for a pointer reference. The arguments and the associated *data*
 4220 *section* are as for **acc_copyin**.

4221 The behavior of these routines for the associated *data section* is:

- 4222 • If the *data section* is in shared memory and does not refer to a captured variable, no action is
 4223 taken.
- 4224 • If the dynamic reference counter for the *data section* is zero, no action is taken.
- 4225 • Otherwise, the dynamic reference counter is updated:
 - 4226 – The **acc_copyout** and **acc_delete**) routines perform a *decrement counter* action
 4227 with the dynamic reference counter.
 - 4228 – The **acc_copyout_finalize** or **acc_delete_finalize** routines perform a
 4229 reset counter action with the dynamic reference counter.

4230 If both reference counters are then zero:

- 4231 – The **acc_copyout** routines perform a *transfer out* action followed by a *deallocate memory*
 4232 action.
- 4233 – The **acc_delete** routines perform a *deallocate memory* action.

4234 The synchronous versions will not return until the data has been completely transferred and the
 4235 memory has been deallocated.

4236 The **_async** versions of these routines will perform any associated data transfers asynchronously
 4237 on the async queue associated with **async_arg**. The routine may return before the data has been
 4238 transferred or deallocated; see Section 2.16 Asynchronous Behavior for more details. Even if the
 4239 data has not been transferred or deallocated before the routine returns, the data will be treated as not
 4240 present in device-accessible memory of the current device if both reference counters are zero.

4241 Errors

- 4242 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
 4243 **bytes** is nonzero.
- 4244 • An **acc_error_not_present** error is issued if the *data section* is not in shared memory
 4245 and is not present in the current device memory.
- 4246 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
 4247 tion that is not contiguous (in Fortran).
- 4248 • An **acc_error_partly_present** error is issued if part of the *data section* is already
 4249 present in device-accessible memory of the current device but all of the *data section* is not.

- 4250 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4251 *argument* value.

4252 See Section 5.2.2.

4253 3.2.20 acc_update_device and acc_update_self

4254 Summary

4255 The **acc_update_device** and **acc_update_self** routines test to see if the argument is in
4256 shared memory and it is not a captured variable; if not, the argument must be present in the device-
4257 accessible memory of the current device, and the routines update the data in device memory from
4258 the corresponding local memory (**acc_update_device**) or update the data in local memory
4259 from the corresponding device-accessible memory (**acc_update_self**).

4260 Format

4261 C or C++:

```
4262 void acc_update_device(h_void* data_arg, size_t bytes);
4263 void acc_update_self (h_void* data_arg, size_t bytes);
4264
4265 void acc_update_device_async(h_void* data_arg, size_t bytes,
4266                             int async_arg);
4267 void acc_update_self_async (h_void* data_arg, size_t bytes,
4268                             int async_arg);
4269
```

4270 Fortran:

```
4271 subroutine acc_update_device(data_arg [, bytes])
4272 subroutine acc_update_self (data_arg [, bytes])
4273
4274 subroutine acc_update_device_async(data_arg [, bytes], async_arg)
4275 subroutine acc_update_self_async (data_arg [, bytes], async_arg)
4276
4277 type(*), dimension(..) :: data_arg
4278 integer :: bytes
4279 integer(acc_handle_kind) :: async_arg
```

4280 Description

4281 A call to an **acc_update_device** routine is functionally equivalent to an **update device**
4282 directive. A call to an **acc_update_self** routine is functionally equivalent to an **update self**
4283 directive. See Section 2.14.4. The arguments and the *data section* are as for **acc_copyin**.

4284 The behavior of these routines for the associated *data section* is:

- 4285 • If the *data section* is in shared memory and does not refer to a captured variable or **bytes** is
4286 zero, no action is taken.
- 4287 • Otherwise:
 - 4288 – A call to an **acc_update_device** routine performs a *transfer in* action with the
4289 corresponding memory.

4290 – A call to an **acc_update_self** routine performs a *transfer out* action with the cor-
4291 responding memory.

4292 The **_async** versions of these routines will perform the data transfers asynchronously on the async
4293 queue associated with **async_arg**. The routine may return before the data has been transferred;
4294 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
4295 until the data has been completely transferred.

4296 Errors

- 4297 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
4298 **bytes** is nonzero.
- 4299 • An **acc_error_not_present** error is issued if the *data section* is not in shared memory
4300 and is not present in the current device memory.
- 4301 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
4302 tion that is not contiguous (in Fortran).
- 4303 • An **acc_error_partly_present** error is issued if part of the *data section* is already
4304 present in device-accessible memory of the current device but all of the *data section* is not.
- 4305 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4306 *argument* value.

4307 See Section 5.2.2.

4308 3.2.21 acc_map_data

4309 Summary

4310 The **acc_map_data** routine maps previously allocated space in the current device memory to the
4311 specified host data.

4312 Format

C or C++:

```
4313     void acc_map_data(h_void* data_arg, d_void* data_dev,  
                      size_t bytes);
```

4314 Fortran:

```
4315     subroutine acc_map_data(data_arg, data_dev, bytes)  
4316         type(*), dimension(*) :: data_arg  
4317         type(c_ptr), value :: data_dev  
4318         integer(c_size_t), value :: bytes
```

4319 Description

4320 A call to the **acc_map_data** routine is similar to a call to **acc_create**, except that instead of
4321 allocating new device memory to start a data lifetime, the device address to use for the data lifetime
4322 is specified as an argument. **data_arg** is a host address, **data_dev** is the corresponding device
4323 address, and **bytes** is the length in bytes. **data_dev** may be the result of a call to **acc_malloc**,
4324 or may come from some other device-specific API routine. The associated *data section* is as for
4325 **acc_copyin**.

4326 The behavior of the **acc_map_data** routine is:

- 4327 • If the *data section* is in shared memory, the behavior is undefined.
- 4328 • If any of the data referred to by **data_dev** is already mapped to any host memory address,
4329 the behavior is undefined.
- 4330 • Otherwise, after this call, when **data_arg** appears in a data clause, the **data_dev** address
4331 will be used. The dynamic reference count for the data referred to by **data_arg** is set to
4332 one, but no data movement will occur.

4333 Memory mapped by **acc_map_data** may not have the associated dynamic reference count decre-
4334 mented to zero, except by a call to **acc_unmap_data**. See Section 2.6.7 Reference Counters.

4335 Errors

- 4336 • An **acc_invalid_null_pointer** error is issued if either **data_arg** or **data_dev** is
4337 a null pointer.
- 4338 • An **acc_invalid_argument** error is issued if **bytes** is zero.
- 4339 • An **acc_error_present** error is issued if any part of the *data section* is already present
4340 in the current device memory.

4341 See Section 5.2.2.

4342 3.2.22 acc_unmap_data

4343 Summary

4344 The **acc_unmap_data** routine unmaps device data from the specified host data.

4345 Format

4346 C or C++:

```
4347 void acc_unmap_data(h_void* data_arg);
```

4348 Fortran:

```
4349 subroutine acc_unmap_data(data_arg)
```

```
4350 type(*), dimension(*) :: data_arg
```

4351 Description

4352 A call to the **acc_unmap_data** routine is similar to a call to **acc_delete**, except the device
4353 memory is not deallocated. **data_arg** is a host address.

4354 The behavior of the **acc_unmap_data** routine is:

- 4355 • If **data_arg** was not previously mapped to some device address via a call to **acc_map_data**,
4356 the behavior is undefined.
- 4357 • Otherwise, the data lifetime for **data_arg** is ended. The dynamic reference count for
4358 **data_arg** is set to zero, but no data movement will occur and the corresponding device
4359 memory is not deallocated. See Section 2.6.7 Reference Counters.

4360 Errors

- 4361 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer.
- 4362 • An **acc_error_present** error is issued if the structured reference count for the any part
4363 of the data is not zero.

4364 See Section 5.2.2.

4365 **3.2.23 acc_deviceptr**4366 **Summary**4367 The **acc_deviceptr** routine returns the device pointer associated with a specific host address.4368 **Format**

4369 C or C++:

4370 `d_void* acc_deviceptr(h_void* data_arg);`

4371 Fortran:

4372 `type(c_ptr) function acc_deviceptr(data_arg)`4373 `type(*), dimension(*) :: data_arg`4374 **Description**4375 The **acc_deviceptr** routine returns the device pointer associated with a host address. **data_arg**
4376 is the address of a host variable or array that may have an active lifetime on the current device.4377 The behavior of the **acc_deviceptr** routine for the data referred to by **data_arg** is:

- 4378 • If the data is in shared memory or **data_arg** is a null pointer, **acc_deviceptr** returns
4379 the incoming address.
- 4380 • If the data is not present in the current device memory, **acc_deviceptr** returns a null
4381 pointer.
- 4382 • Otherwise, **acc_deviceptr** returns the address in the current device memory that corre-
4383 sponds to the address **data_arg**.

4384 **3.2.24 acc_hostptr**4385 **Summary**4386 The **acc_hostptr** routine returns the host pointer associated with a specific device address.4387 **Format**

4388 C or C++:

4389 `h_void* acc_hostptr(d_void* data_dev);`

4390 Fortran:

4391 `type(c_ptr) function acc_hostptr(data_dev)`4392 `type(c_ptr), value :: data_dev`4393 **Description**4394 The **acc_hostptr** routine returns the host pointer associated with a device address. **data_dev**
4395 is the address of a device variable or array, such as that returned from **acc_deviceptr**, **acc_create**
4396 or **acc_copyin**.4397 The behavior of the **acc_hostptr** routine for the data referred to by **data_dev** is:

- 4398 • If the data is in shared memory or **data_dev** is a null pointer, **acc_hostptr** returns the
4399 incoming address.
- 4400 • If the data corresponds to a host address which is present in the current device memory,
4401 **acc_hostptr** returns the host address.
- 4402 • Otherwise, **acc_hostptr** returns a null pointer.

4403 **3.2.25 acc_is_present**4404 **Summary**

4405 The **acc_is_present** routine tests whether a variable or array region is accessible from the
4406 current device.

4407 **Format**

4408 C or C++:

```
4409     int acc_is_present(h_void* data_arg, size_t bytes);
```

4410 Fortran:

```
4411     logical function acc_is_present(data_arg)
4412     logical function acc_is_present(data_arg, bytes)
4413     type(*), dimension(..) :: data_arg
4414     integer :: bytes
```

4415 **Description**

4416 The **acc_is_present** routine tests whether the specified host data is accessible from the current
4417 device. In C/C++, **data_arg** is a pointer to the data, and **bytes** specifies the data size in bytes. In
4418 Fortran, two forms are supported. In the first, **data_arg** is a variable or contiguous array section.
4419 In the second, **data_arg** is a variable or array element and **bytes** is the length in bytes. A
4420 **bytes** value of zero is treated as a value of one if **data_arg** is not a null pointer.

4421 The behavior of the **acc_is_present** routines for the data referred to by **data_arg** is:

- 4422 • If the data is in shared memory, a call to **acc_is_present** will evaluate to *true*.
- 4423 • If the data is present in the current device memory, a call to **acc_is_present** will evaluate
4424 to *true*.
- 4425 • Otherwise, a call to **acc_is_present** will evaluate to *false*.

4426 **Errors**

- 4427 • An **acc_error_invalid_argument** error is issued if **bytes** is negative (in Fortran).
- 4428 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
4429 tion that is not contiguous (in Fortran).

4430 See Section 5.2.2.

4431 **3.2.26 acc_memcpy_to_device**4432 **Summary**

4433 The **acc_memcpy_to_device** routine copies data from local memory to device memory.

4434 **Format**

C or C++:

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

4435

Fortran:

```

subroutine acc_memcpy_to_device(data_dev_dest,
                               data_host_src, bytes)
subroutine acc_memcpy_to_device_async(data_dev_dest,
4436                               data_host_src, bytes, async_arg)
4437   type(c_ptr), value :: data_dev_dest
4438   type(*), dimension(*) :: data_host_src
4439   integer(c_size_t), value :: bytes
4440   integer(acc_handle_kind), value :: async_arg

```

4441 Description

4442 The `acc_memcpy_to_device` routine copies `bytes` bytes of data from the local address in
4443 `data_host_src` to the device address in `data_dev_dest`. `data_dev_dest` must be an
4444 address accessible from the current device, such as an address returned from `acc_malloc` or
4445 `acc_deviceptr`, or an address in shared memory.

4446 The behavior of the `acc_memcpy_to_device` routines is:

- 4447 • If `bytes` is zero, no action is taken.
- 4448 • If `data_dev_dest` and `data_host_src` both refer to shared memory and have the same
4449 value, no action is taken.
- 4450 • If `data_dev_dest` and `data_host_src` both refer to shared memory and the memory
4451 regions overlap, the behavior is undefined.
- 4452 • If the data referred to by `data_dev_dest` is not accessible by the current device, the be-
4453 havior is undefined.
- 4454 • If the data referred to by `data_host_src` is not accessible by the local thread, the behavior
4455 is undefined.
- 4456 • Otherwise, `bytes` bytes of data at `data_host_src` in local memory are copied to
4457 `data_dev_dest` in the current device memory.

4458 The `_async` version of this routine will perform the data transfers asynchronously on the `async`
4459 queue associated with `async_arg`. The routine may return before the data has been transferred;
4460 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
4461 until the data has been completely transferred.

4462 Errors

- 4463 • An `acc_error_invalid_null_pointer` error is issued if `data_dev_dest` or
4464 `data_host_src` is a null pointer and `bytes` is nonzero.
- 4465 • An `acc_error_invalid_async` error is issued if `async_arg` is not a valid *async-*
4466 *argument* value.

4467 See Section 5.2.2.

4468 3.2.27 acc_memcpy_from_device

4469 Summary

4470 The `acc_memcpy_from_device` routine copies data from device memory to local memory.

4471 **Format**

C or C++:

```

4471 void acc_memcpy_from_device(h_void* data_host_dest,
                               d_void* data_dev_src, size_t bytes);
4472 void acc_memcpy_from_device_async(h_void* data_host_dest,
4473                                   d_void* data_dev_src, size_t bytes,
                                        int async_arg);

```

Fortran:

```

4474 subroutine acc_memcpy_from_device(data_host_dest,
4475                                   data_dev_src, bytes)
4476 subroutine acc_memcpy_from_device_async(data_host_dest,
4477                                         data_dev_src, bytes, async_arg)
4478 type(*), dimension(*) :: data_host_dest
4479 type(c_ptr), value :: data_dev_src
4480 integer(c_size_t), value :: bytes
4481 integer(acc_handle_kind), value :: async_arg

```

4479 **Description**

4480 The `acc_memcpy_from_device` routine copies `bytes` bytes of data from the device address
 4481 in `data_dev_src` to the local address in `data_host_dest`. `data_dev_src` must be an
 4482 address accessible from the current device, such as an address returned from `acc_malloc` or
 4483 `acc_deviceptr`, or an address in shared memory.

4484 The behavior of the `acc_memcpy_from_device` routines is:

- 4485 • If `bytes` is zero, no action is taken.
- 4486 • If `data_host_dest` and `data_dev_src` both refer to shared memory and have the same
 4487 value, no action is taken.
- 4488 • If `data_host_dest` and `data_dev_src` both refer to shared memory and the memory
 4489 regions overlap, the behavior is undefined.
- 4490 • If the data referred to by `data_dev_src` is not accessible by the current device, the behav-
 4491 ior is undefined.
- 4492 • If the data referred to by `data_host_dest` is not accessible by the local thread, the behav-
 4493 ior is undefined.
- 4494 • Otherwise, `bytes` bytes of data at `data_dev_src` in the current device memory are copied
 4495 to `data_host_dest` in local memory.

4496 The `_async` version of this routine will perform the data transfers asynchronously on the `async`
 4497 queue associated with `async_arg`. The routine may return before the data has been transferred;
 4498 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
 4499 until the data has been completely transferred.

4500 **Errors**

- 4501 • An `acc_error_invalid_null_pointer` error is issued if `data_host_dest` or
 4502 `data_dev_src` is a null pointer and `bytes` is nonzero.

- 4503 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4504 *argument* value.

4505 See Section 5.2.2.

4506 3.2.28 acc_memcpy_device

4507 Summary

4508 The **acc_memcpy_device** routine copies data from one memory location to another memory
4509 location on the current device.

4510 Format

C or C++:

```
4511 void acc_memcpy_device(d_void* data_dev_dest,
4512                       d_void* data_dev_src, size_t bytes);
4511 void acc_memcpy_device_async(d_void* data_dev_dest,
4512                              d_void* data_dev_src, size_t bytes,
4513                              int async_arg);
```

Fortran:

```
4513 subroutine acc_memcpy_device(data_dev_dest,
4514                             data_dev_src, bytes);
4515 subroutine acc_memcpy_device_async(data_dev_dest,
4516                                   data_dev_src, bytes,
4517                                   async_arg);
4514 type(c_ptr), value :: data_dev_dest
4515 type(c_ptr), value :: data_dev_src
4516 integer(c_size_t), value :: bytes
4517 integer(acc_handle_kind), value :: async_arg
```

4518 Description

4519 The **acc_memcpy_device** routine copies **bytes** bytes of data from the device address in
4520 **data_dev_src** to the device address in **data_dev_dest**. Both addresses must be addresses in
4521 the current device memory, such as would be returned from **acc_malloc** or **acc_deviceptr**.

4522 The behavior of the **acc_memcpy_device** routines is:

- 4523 • If **bytes** is zero, no action is taken.
- 4524 • If **data_dev_dest** and **data_dev_src** have the same value, no action is taken.
- 4525 • If the memory regions referred to by **data_dev_dest** and **data_dev_src** overlap, the
4526 behavior is undefined.
- 4527 • If the data referred to by **data_dev_src** or **data_dev_dest** is not accessible by the
4528 current device, the behavior is undefined.
- 4529 • Otherwise, **bytes** bytes of data at **data_dev_src** in the current device memory are copied
4530 to **data_dev_dest** in the current device memory.

4531 The **_async** version of this routine will perform the data transfers asynchronously on the **async**
4532 queue associated with **async_arg**. The routine may return before the data has been transferred;
4533 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
4534 until the data has been completely transferred.

4535 **Errors**

- 4536 • An **acc_error_invalid_null_pointer** error is issued if **data_dev_dest** or
4537 **data_dev_src** is a null pointer and **bytes** is nonzero.
- 4538 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4539 *argument* value.

4540 See Section 5.2.2.

4541 **3.2.29 acc_attach and acc_detach**4542 **Summary**

4543 The **acc_attach** routines update a pointer in device-accessible memory to point to the corre-
4544 sponding copy of the host pointer target. The **acc_detach** routines restore a pointer in device-
4545 accessible memory to point to the host pointer target.

4546 **Format**

4547 C or C++:

```
4548 void acc_attach(h_void** ptr_addr);
4549 void acc_attach_async(h_void** ptr_addr, int async_arg);
4550
4551 void acc_detach(h_void** ptr_addr);
4552 void acc_detach_async(h_void** ptr_addr, int async_arg);
4553 void acc_detach_finalize(h_void** ptr_addr);
4554 void acc_detach_finalize_async(h_void** ptr_addr,
4555                               int async_arg);
```

4556 Fortran:

```
4557 subroutine acc_attach(ptr_addr)
4558 subroutine acc_attach_async(ptr_addr, async_arg)
4559   type(*), dimension(..)      :: ptr_addr
4560   integer(acc_handle_kind), value :: async_arg
4561
4562 subroutine acc_detach(ptr_addr)
4563 subroutine acc_detach_async(ptr_addr, async_arg)
4564 subroutine acc_detach_finalize(ptr_addr)
4565 subroutine acc_detach_finalize_async(ptr_addr,
4566                                     async_arg)
4567   type(*), dimension(..)      :: ptr_addr
4568   integer(acc_handle_kind), value :: async_arg
```

4569 **Description**

4570 A call to an **acc_attach** routine is functionally equivalent to an **enter data attach** direc-
4571 tive, as described in Section 2.7.13. A call to an **acc_detach** routine is functionally equivalent to
4572 an **exit data detach** directive, and a call to an **acc_detach_finalize** routine is function-
4573 ally equivalent to an **exit data finalize detach** directive, as described in Section 2.7.14.
4574 **ptr_addr** must be the address of a host pointer. **async_arg** must be an *async-argument* as
4575 defined in Section 2.16.

4576 The behavior of these routines is:

- 4577 • If **ptr_addr** refers to shared memory and does not refer to a captured variable, no action is
4578 taken.
- 4579 • If the pointer referred to by **ptr_addr** is not present in device-accessible memory of the
4580 current device, no action is taken.
- 4581 • Otherwise:
- 4582 – The **acc_attach** routines behave as follows,
- 4583 1. an *increment counter* action is performed on the associated attachment counter,
4584 2. if the associated attachment counter is now one, an *attach pointer* action is per-
4585 formed on the pointer referred to by **ptr_addr**; see Section 2.7.2.
- 4586 – The **acc_detach** routines behave as follows
- 4587 1. an *decrement counter* action is performed on the associated attachment counter,
4588 2. if the associated attachment counter is now zero, an *detach pointer* action is per-
4589 formed on the pointer referred to by **ptr_addr**; see Section 2.7.2.
- 4590 See Section 2.7.2.
- 4591 – The **acc_detach_finalize** routines behave as follows, perform a *detach pointer*
4592 action on the pointer referred to by **ptr_addr** followed by a *reset counter* action on
4593 the associated attachment counter; see Section 2.7.2.

4594 These routines may issue a data transfer from local memory to device-accessible memory. The
4595 **_async** versions of these routines will perform the data transfers asynchronously on the **async**
4596 queue associated with **async_arg**. These routines may return before the data has been transferred;
4597 see Section 2.16 for more details. The synchronous versions will not return until the data has been
4598 completely transferred.

4599 Errors

- 4600 • An **acc_error_invalid_null_pointer** error is issued if **ptr_addr** is a null pointer.
- 4601 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4602 *argument* value.

4603 See Section 5.2.2.

4604 3.2.30 acc_memcpy_d2d

4605 Summary

4606 The **acc_memcpy_d2d** routines copy the contents of an array on one device to an array on the
4607 same or a different device without updating the value on the host.

4608 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,
```

```

4609         h_void* data_arg_src, size_t bytes,
4610         int dev_num_dest, int dev_num_src,
         int async_arg_src);

```

Fortran:

```

         subroutine acc_memcpy_d2d(data_arg_dest, data_arg_src, &
             bytes, dev_num_dest, dev_num_src)
         subroutine acc_memcpy_d2d_async(data_arg_dest, data_arg_src, &
             bytes, dev_num_dest, dev_num_src, &
4611             async_arg_src)
4612         type(*), dimension(..) :: data_arg_dest
4613         type(*), dimension(..) :: data_arg_src
4614         integer :: bytes
4615         integer :: dev_num_dest
4616         integer :: dev_num_src
4617         integer :: async_arg_src
4618

```

4619 Description

4620 The **acc_memcpy_d2d** routines are passed the address of destination and source host data as well
4621 as integer device numbers for the destination and source devices, which must both be of the current
4622 device type.

4623 The behavior of the **acc_memcpy_d2d** routines is:

- 4624 • If **bytes** is zero, no action is taken.
- 4625 • If both pointers have the same value and either the two device numbers are the same or the
4626 addresses are in shared memory, then no action is taken.
- 4627 • Otherwise, **bytes** bytes of data at the device address corresponding to **data_arg_src** on
4628 device **dev_num_src** are copied to the device address corresponding to **data_arg_dest**
4629 on device **dev_num_dest**.

4630 For **acc_memcpy_d2d_async** the value of **async_arg_src** is the number of an async queue
4631 on the source device. This routine will perform the data transfers asynchronously on the async queue
4632 associated with **async_arg_src** for device **dev_num_src**; see Section 2.16 Asynchronous Behavior
4633 for more details.

4634 Errors

- 4635 • An **acc_error_device_unavailable** error is issued if **dev_num_dest** or **dev_num_src**
4636 is not a valid device number.
- 4637 • An **acc_error_invalid_null_pointer** error is issued if either **data_arg_dest**
4638 or **data_arg_src** is a null pointer and **bytes** is nonzero.
- 4639 • An **acc_error_not_present** error is issued if the data at either address is not in shared
4640 memory and is not present in the respective device memory.
- 4641 • An **acc_error_partly_present** error is issued if part of the data is already present in
4642 the current device memory but all of the data is not.

4643 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4644 *argument* value.

4645 See Section 5.2.2.

4. Environment Variables

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

4.1 ACC_DEVICE_TYPE

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

Example:

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

4.2 ACC_DEVICE_NUM

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

Example:

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

4.3 ACC_PROFLIB

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

Example:

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


5. Profiling and Error Callback Interface

4674

4675 This chapter describes the OpenACC interface for runtime callback routines. These routines may be
4676 provided by the programmer or by a tool or library developer. Calls to these routines are triggered
4677 during the application execution at specific OpenACC events. There are two classes of events,
4678 profiling events and error events. Profiling events can be used by tools for profile or trace data
4679 collection. Currently, this interface does not support tools that employ asynchronous sampling.
4680 Error events can be used to release resources or cleanly shut down a large parallel application when
4681 the OpenACC runtime detects an error condition from which it cannot recover. This is specifically
4682 for error handling, not for error recovery. There is no support provided for restarting or retrying
4683 an OpenACC program, construct, or API routine after an error condition has been detected and an
4684 error callback routine has been called.

4685 In this chapter, the term *runtime* refers to the OpenACC runtime library. The term *library* refers to
4686 the routines invoked at specified events by the OpenACC runtime.

4687 There are three steps for interfacing a *library* to the *runtime*. The first step is to write the library
4688 callback routines. Section 5.1 Events describes the supported runtime events and the order in which
4689 callbacks to the callback routines will occur. Section 5.2 Callbacks Signature describes the signature
4690 of the callback routines for all events.

4691 The second step is to load the *library* at runtime. The *library* may be statically linked to the appli-
4692 cation or dynamically loaded by the application, a library, or a tool. This is described in Section 5.3
4693 Loading the Library.

4694 The third step is to register the desired callbacks with the events. This may be done explicitly by the
4695 application, if the library is statically linked with the application, implicitly by including a call to a
4696 registration routine in a `.init` section, or by including an initialization routine in the library if it is
4697 dynamically loaded by the *runtime*. This is described in Section 5.4 Registering Event Callbacks.

5.1 Events

4698

4699 This section describes the events that are recognized by the runtime. Most profiling events have a
4700 start and end callback routine, that is, a routine that is called just before the runtime code to handle
4701 the event starts and another routine that is called just after the event is handled. The event names
4702 and routine prototypes are available in the header file `acc_callback.h`, which is delivered with
4703 the OpenACC implementation. For backward compatibility with previous versions of OpenACC,
4704 the implementation also delivers the same information in `acc_prof.h`. Event names are prefixed
4705 with `acc_ev_`.

4706 The ordering of events must reflect the order in which the OpenACC runtime actually executes them,
4707 i.e. if a runtime moves the enqueueing of data transfers or kernel launches outside the originating
4708 clauses/constructs, it needs to issue the corresponding launch callbacks when they really occur. A
4709 callback for a start event must always precede the matching end callback. No callbacks will be
4710 issued after a runtime shutdown event.

4711 The events that the runtime supports can be registered with a callback and are defined in the enu-
4712 meration type `acc_event_t`.

```
4713     typedef enum acc_event_t{
4714         acc_ev_none = 0,
4715         acc_ev_device_init_start = 1,
4716         acc_ev_device_init_end = 2,
4717         acc_ev_device_shutdown_start = 3,
4718         acc_ev_device_shutdown_end = 4,
4719         acc_ev_runtime_shutdown = 5,
4720         acc_ev_create = 6,
4721         acc_ev_delete = 7,
4722         acc_ev_alloc = 8,
4723         acc_ev_free = 9,
4724         acc_ev_enter_data_start = 10,
4725         acc_ev_enter_data_end = 11,
4726         acc_ev_exit_data_start = 12,
4727         acc_ev_exit_data_end = 13,
4728         acc_ev_update_start = 14,
4729         acc_ev_update_end = 15,
4730         acc_ev_compute_construct_start = 16,
4731         acc_ev_compute_construct_end = 17,
4732         acc_ev_enqueue_launch_start = 18,
4733         acc_ev_enqueue_launch_end = 19,
4734         acc_ev_enqueue_upload_start = 20,
4735         acc_ev_enqueue_upload_end = 21,
4736         acc_ev_enqueue_download_start = 22,
4737         acc_ev_enqueue_download_end = 23,
4738         acc_ev_wait_start = 24,
4739         acc_ev_wait_end = 25,
4740         acc_ev_error = 100,
4741         acc_ev_last = 101
4742     }acc_event_t;
```

4743 The value of `acc_ev_last` will change if new events are added to the enumeration, so a library
4744 must not depend on that value.

4745 5.1.1 Runtime Initialization and Shutdown

4746 No callbacks can be registered for the runtime initialization. Instead the initialization of the tool is
4747 handled as described in Section 5.3 Loading the Library.

4748 The *runtime shutdown* profiling event name is

```
4749     acc_ev_runtime_shutdown
```

4750 This event is triggered before the OpenACC runtime shuts down, either because all devices have
4751 been shutdown by calls to the `acc_shutdown` API routine, or at the end of the program.

4752 5.1.2 Device Initialization and Shutdown

4753 The *device initialization* profiling event names are

4754 **acc_ev_device_init_start**
4755 **acc_ev_device_init_end**

4756 These events are triggered when a device is being initialized by the OpenACC runtime. This may be
4757 when the program starts, or may be later during execution when the program reaches an **acc_init**
4758 call or an OpenACC construct. The **acc_ev_device_init_start** is triggered before device
4759 initialization starts and **acc_ev_device_init_end** after initialization is complete.

4760 The *device shutdown* profiling event names are

4761 **acc_ev_device_shutdown_start**
4762 **acc_ev_device_shutdown_end**

4763 These events are triggered when a device is shut down, most likely by a call to the OpenACC
4764 **acc_shutdown** API routine. The **acc_ev_device_shutdown_start** is triggered before
4765 the device shutdown process starts and **acc_ev_device_shutdown_end** after the device shut-
4766 down is complete.

4767 **5.1.3 Enter Data and Exit Data**

4768 The *enter data* profiling event names are

4769 **acc_ev_enter_data_start**
4770 **acc_ev_enter_data_end**

4771 These events are triggered at **enter data** directives, entry to data constructs, and entry to implicit
4772 data regions such as those generated by compute constructs. The **acc_ev_enter_data_start**
4773 event is triggered before any *data allocation*, *data update*, or *wait* events that are associated with
4774 that directive or region entry, and the **acc_ev_enter_data_end** is triggered after those events.

4775 The *exit data* profiling event names are

4776 **acc_ev_exit_data_start**
4777 **acc_ev_exit_data_end**

4778 These events are triggered at **exit data** directives, exit from **data** constructs, and exit from
4779 implicit data regions. The **acc_ev_exit_data_start** event is triggered before any *data*
4780 *deallocation*, *data update*, or *wait* events associated with that directive or region exit, and the
4781 **acc_ev_exit_data_end** event is triggered after those events.

4782 When the construct that triggers an *enter data* or *exit data* event was generated implicitly by the
4783 compiler the **implicit** field in the event structure will be set to **1**. When the construct that
4784 triggers these events was specified explicitly by the application code the **implicit** field in the
4785 event structure will be set to **0**.

4786 **5.1.4 Data Allocation**

4787 The *data allocation* profiling event names are

4788 **acc_ev_create**
4789 **acc_ev_delete**
4790 **acc_ev_alloc**
4791 **acc_ev_free**

4792 An **acc_ev_alloc** event is triggered when the OpenACC runtime allocates memory from the de-
4793 vice memory pool, and an **acc_ev_free** event is triggered when the runtime frees that memory.
4794 An **acc_ev_create** event is triggered when the OpenACC runtime associates device memory
4795 with local memory, such as for a data clause (**create**, **copyin**, **copy**, **copyout**) at entry to
4796 a data construct, compute construct, at an **enter data** directive, or in a call to a data API rou-
4797 tine (**acc_copyin**, **acc_create**, ...). An **acc_ev_create** event may be preceded by an
4798 **acc_ev_alloc** event, if newly allocated memory is used for this device data, or it may not, if
4799 the runtime manages its own memory pool. An **acc_ev_delete** event is triggered when the
4800 OpenACC runtime disassociates device memory from local memory, such as for a data clause at
4801 exit from a data construct, compute construct, at an **exit data** directive, or in a call to a data API
4802 routine (**acc_copyout**, **acc_delete**, ...). An **acc_ev_delete** event may be followed by
4803 an **acc_ev_free** event, if the disassociated device memory is freed, or it may not, if the runtime
4804 manages its own memory pool.

4805 When the action that generates a *data allocation* event was generated explicitly by the application
4806 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event
4807 is triggered because of a variable or array with implicitly-determined data attributes or otherwise
4808 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

4809 5.1.5 Data Construct

4810 The profiling events for entering and leaving *data constructs* are mapped to *enter data* and *exit data*
4811 events as described in Section 5.1.3 Enter Data and Exit Data.

4812 5.1.6 Update Directive

4813 The *update directive* profiling event names are

4814 **acc_ev_update_start**
4815 **acc_ev_update_end**

4816 The **acc_ev_update_start** event will be triggered at an **update** directive, before any *data*
4817 *update* or *wait* events that are associated with the update directive are carried out, and the corre-
4818 sponding **acc_ev_update_end** event will be triggered after any of the associated events.

4819 5.1.7 Compute Construct

4820 The *compute construct* profiling event names are

4821 **acc_ev_compute_construct_start**
4822 **acc_ev_compute_construct_end**

4823 The **acc_ev_compute_construct_start** event is triggered at entry to a compute construct,
4824 before any *launch* events that are associated with entry to the compute construct. The
4825 **acc_ev_compute_construct_end** event is triggered at the exit of the compute construct,
4826 after any *launch* events associated with exit from the compute construct. If there are data clauses
4827 on the compute construct, those data clauses may be treated as part of the compute construct, or as
4828 part of a data construct containing the compute construct. The callbacks for data clauses must use
4829 the same line numbers as for the compute construct events.

5.1.8 Enqueue Kernel Launch

4830

The *launch* profiling event names are

4831

acc_ev_enqueue_launch_start

4832

acc_ev_enqueue_launch_end

4833

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

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

5.1.9 Enqueue Data Update (Upload and Download)

4845

The *data update* profiling event names are

4846

acc_ev_enqueue_upload_start

4847

acc_ev_enqueue_upload_end

4848

acc_ev_enqueue_download_start

4849

acc_ev_enqueue_download_end

4850

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

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

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

5.1.10 Wait

4862

The *wait* profiling event names are

4863

acc_ev_wait_start

4864

acc_ev_wait_end

4865

4866

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

4867

4868

4869 queue after the local thread has determined that the queue is empty.

4870 Wait events occur when the local thread and a device synchronize, either due to a **wait** directive
4871 or by a *wait* clause on a synchronous data construct, compute construct, or **enter data**, **exit**
4872 **data**, or **update** directive. For *wait* events triggered by an explicit synchronous **wait** directive
4873 or *wait* clause, the **implicit** field in the event structure will be **0**. For all other wait events, the
4874 **implicit** field in the event structure will be **1**.

4875 The OpenACC runtime need not trigger *wait* events for queues that have not been used in the
4876 program, and need not trigger *wait* events for queues that have not been used by this thread since
4877 the last *wait* operation. For instance, an **acc wait** directive with no arguments is defined to wait on
4878 all queues. If the program only uses the default (synchronous) queue and the queue associated with
4879 **async (1)** and **async (2)** then an **acc wait** directive may trigger *wait* events only for those
4880 three queues. If the implementation knows that no activities have been enqueued on the **async (2)**
4881 queue since the last *wait* operation, then the **acc wait** directive may trigger *wait* events only for
4882 the default queue and the **async (1)** queue.

4883 5.1.11 Error Event

4884 The only error event is

4885 **acc_ev_error**

4886 An **acc_ev_error** event is triggered when the OpenACC program detects a runtime error con-
4887 dition. The default runtime error callback routine may print an error message and halt program
4888 execution. An application can register additional error event callback routines, to allow a failing
4889 application to release resources or to cleanly shut down a large parallel runtime with many threads
4890 and processes, for instance.

4891 The application can register multiple alternate error callbacks. As described in Section
4892 5.4.1 Multiple Callbacks, the callbacks will be invoked in the order in which they are registered.
4893 If all the error callbacks return, the default error callback will be invoked. The error callback
4894 routine must not execute any OpenACC compute or data constructs. The only OpenACC API
4895 routines that can be safely invoked from an error callback routine are **acc_get_property**,
4896 **acc_get_property_string**, and **acc_shutdown**.

4897 5.2 Callbacks Signature

4898 This section describes the signature of event callbacks. All event callbacks have the same signature.
4899 The routine prototypes are available in the header file **acc_callback.h**, which is delivered with
4900 the OpenACC implementation.

4901 All callback routines have three arguments. The first argument is a pointer to a struct containing
4902 general information; the same struct type is used for all callback events. The second argument is
4903 a pointer to a struct containing information specific to that callback event; there is one struct type
4904 containing information for data events, another struct type containing information for kernel launch
4905 events, and a third struct type for other events, containing essentially no information. The third
4906 argument is a pointer to a struct containing information about the application programming interface
4907 (API) being used for the specific device. For NVIDIA CUDA devices, this contains CUDA-specific
4908 information; for OpenCL devices, this contains OpenCL-specific information. Other interfaces can
4909 be supported as they are added by implementations. The prototype for a callback routine is:

```

4910     typedef void (*acc_callback)
4911         (acc_callback_info*, acc_event_info*, acc_api_info*);
4912     typedef acc_callback acc_prof_callback;

```

4913 In the descriptions, the datatype `ssize_t` means a signed 32-bit integer for a 32-bit binary and
 4914 a 64-bit integer for a 64-bit binary, the datatype `size_t` means an unsigned 32-bit integer for a
 4915 32-bit binary and a 64-bit integer for a 64-bit binary, and the datatype `int` means a 32-bit integer
 4916 for both 32-bit and 64-bit binaries.

4917 5.2.1 First Argument: General Information

4918 The first argument is a pointer to the `acc_callback_info` struct type:

```

4919     typedef struct acc_prof_info{
4920         acc_event_t event_type;
4921         int valid_bytes;
4922         int version;
4923         acc_device_t device_type;
4924         int device_number;
4925         int thread_id;
4926         ssize_t async;
4927         ssize_t async_queue;
4928         const char* src_file;
4929         const char* func_name;
4930         int line_no, end_line_no;
4931         int func_line_no, func_end_line_no;
4932     }acc_callback_info;
4933     typedef struct acc_prof_info acc_prof_info;

```

4934 The name `acc_prof_info` is preserved for backward compatibility with previous versions of
 4935 OpenACC. The fields are described below.

4936 • **acc_event_t event_type** - The event type that triggered this callback. The datatype
 4937 is the enumeration type `acc_event_t`, described in the previous section. This allows the
 4938 same callback routine to be used for different events.

4939 • **int valid_bytes** - The number of valid bytes in this struct. This allows a library to inter-
 4940 face with newer runtimes that may add new fields to the struct at the end while retaining com-
 4941 patibility with older runtimes. A runtime must fill in the `event_type` and `valid_bytes`
 4942 fields, and must fill in values for all fields with offset less than `valid_bytes`. The value of
 4943 `valid_bytes` for a struct is recursively defined as:

```

4944     valid_bytes(struct) = offset(lastfield) + valid_bytes(lastfield)
4945     valid_bytes(type[n]) = (n-1)*sizeof(type) + valid_bytes(type)
4946     valid_bytes(basictype) = sizeof(basictype)

```

4947 • **int version** - A version number; the value of `_OPENACC`.

4948 • **acc_device_t device_type** - The device type corresponding to this event. The datatype
 4949 is `acc_device_t`, an enumeration type of all the supported device types, defined in `openacc.h`.

4950 • **int device_number** - The device number. Each device is numbered, typically starting at

- 4951 device zero. For applications that use more than one device type, the device numbers may be
4952 unique across all devices or may be unique only across all devices of the same device type.
- 4953 • **int thread_id** - The host thread ID making the callback. Host threads are given unique
4954 thread ID numbers typically starting at zero. This is not necessarily the same as the OpenMP
4955 thread number.
 - 4956 • **ssize_t async** - The *async-value* used for operations associated with this event; see Sec-
4957 tion 2.16 Asynchronous Behavior.
 - 4958 • **ssize_t async_queue** - The actual activity queue onto which the **async** field gets
4959 mapped; see Section 2.16 Asynchronous Behavior.
 - 4960 • **const char* src_file** - A pointer to null-terminated string containing the name of or
4961 path to the source file, if known, or a null pointer if not. If the library wants to save the source
4962 file name, it must allocate memory and copy the string.
 - 4963 • **const char* func_name** - A pointer to a null-terminated string containing the name of
4964 the function in which the event occurred, if known, or a null pointer if not. If the library wants
4965 to save the function name, it must allocate memory and copy the string.
 - 4966 • **int line_no** - The line number of the directive or program construct or the starting line
4967 number of the OpenACC construct corresponding to the event. A negative or zero value
4968 means the line number is not known.
 - 4969 • **int end_line_no** - For an OpenACC construct, this contains the line number of the end
4970 of the construct. A negative or zero value means the line number is not known.
 - 4971 • **int func_line_no** - The line number of the first line of the function named in **func_name**.
4972 A negative or zero value means the line number is not known.
 - 4973 • **int func_end_line_no** - The last line number of the function named in **func_name**.
4974 A negative or zero value means the line number is not known.

4975 5.2.2 Second Argument: Event-Specific Information

4976 The second argument is a pointer to the **acc_event_info** union type.

```
4977 typedef union acc_event_info{
4978     acc_event_t event_type;
4979     acc_data_event_info data_event;
4980     acc_launch_event_info launch_event;
4981     acc_other_event_info other_event;
4982 }acc_event_info;
```

4983 The **event_type** field selects which union member to use. The first five members of each union
4984 member are identical. The second through fifth members of each union member (**valid_bytes**,
4985 **parent_construct**, **implicit**, and **tool_info**) have the same semantics for all event
4986 types:

- 4987 • **int valid_bytes** - The number of valid bytes in the respective struct. (This field is similar
4988 used as discussed in Section 5.2.1 First Argument: General Information.)

- 4989 • **acc_construct_t parent_construct** - This field describes the type of construct
4990 that caused the event to be emitted. The possible values for this field are defined by the
4991 **acc_construct_t** enum, described at the end of this section.
- 4992 • **int implicit** - This field is set to 1 for any implicit event, such as an implicit wait at
4993 a synchronous data construct or synchronous enter data, exit data or update directive. This
4994 field is set to zero when the event is triggered by an explicit directive or call to a runtime API
4995 routine.
- 4996 • **void* tool_info** - This field is used to pass tool-specific information from a **_start**
4997 event to the matching **_end** event. For a **_start** event callback, this field will be initialized
4998 to a null pointer. The value of this field for a **_end** event will be the value returned by the
4999 library in this field from the matching **_start** event callback, if there was one, or a null
5000 pointer otherwise. For events that are neither **_start** or **_end** events, this field will be a
5001 null pointer.

5002 Data Events

5003 For a data event, as noted in the event descriptions, the second argument will be a pointer to the
5004 **acc_data_event_info** struct.

```
5005     typedef struct acc_data_event_info{
5006         acc_event_t event_type;
5007         int valid_bytes;
5008         acc_construct_t parent_construct;
5009         int implicit;
5010         void* tool_info;
5011         const char* var_name;
5012         size_t bytes;
5013         const void* host_ptr;
5014         const void* device_ptr;
5015     }acc_data_event_info;
```

5016 The fields specific for a data event are:

- 5017 • **acc_event_t event_type** - The event type that triggered this callback. The events that
5018 use the **acc_data_event_info** struct are:
 - 5019 **acc_ev_enqueue_upload_start**
 - 5020 **acc_ev_enqueue_upload_end**
 - 5021 **acc_ev_enqueue_download_start**
 - 5022 **acc_ev_enqueue_download_end**
 - 5023 **acc_ev_create**
 - 5024 **acc_ev_delete**
 - 5025 **acc_ev_alloc**
 - 5026 **acc_ev_free**
- 5027 • **const char* var_name** - A pointer to null-terminated string containing the name of the
5028 variable for which this event is triggered, if known, or a null pointer if not. If the library wants
5029 to save the variable name, it must allocate memory and copy the string.
- 5030 • **size_t bytes** - The number of bytes for the data event.

- 5031 • **const void* host_ptr** - If available and appropriate for this event, this is a pointer to
5032 the host data.
- 5033 • **const void* device_ptr** - If available and appropriate for this event, this is a pointer
5034 to the corresponding device data.

5035 Launch Events

5036 For a launch event, as noted in the event descriptions, the second argument will be a pointer to the
5037 **acc_launch_event_info** struct.

```
5038     typedef struct acc_launch_event_info{
5039         acc_event_t event_type;
5040         int valid_bytes;
5041         acc_construct_t parent_construct;
5042         int implicit;
5043         void* tool_info;
5044         const char* kernel_name;
5045         size_t num_gangs, num_workers, vector_length;
5046         size_t* num_gangs_per_dim;
5047     }acc_launch_event_info;
```

5048 The fields specific for a launch event are:

- 5049 • **acc_event_t event_type** - The event type that triggered this callback. The events that
5050 use the **acc_launch_event_info** struct are:

```
5051         acc_ev_enqueue_launch_start
5052         acc_ev_enqueue_launch_end
```

- 5053 • **const char* kernel_name** - A pointer to null-terminated string containing the name of
5054 the kernel being launched, if known, or a null pointer if not. If the library wants to save the
5055 kernel name, it must allocate memory and copy the string.
- 5056 • **size_t num_gangs, num_workers, vector_length** - The number of gangs, work-
5057 ers, and vector lanes created for this kernel launch.
- 5058 • **size_t* num_gangs_per_dim** - An array of **size_t** whose first element indicates the
5059 number of dimensions of gang parallelism and each subsequent element gives the number of
5060 gangs along each dimension starting with dimension 1. The product of the values of elements
5061 1 through **num_gangs_per_dim[0]** is **num_gangs**.

5062 Error Events

5063 For an error event, as noted in the event descriptions, the second argument will be a pointer to the
5064 **acc_error_event_info** struct.

```
5065     typedef struct acc_error_event_info{
5066         acc_event_t event_type;
5067         int valid_bytes;
5068         acc_construct_t parent_construct;
5069         int implicit;
5070         void* tool_info;
```

```

5071     acc_error_t error_code;
5072     const char* error_message;
5073     size_t runtime_info;
5074 }acc_error_event_info;

```

5075 The enumeration type for the error code is

```

5076 typedef enum acc_error_t{
5077     acc_error_none = 0,
5078     acc_error_other = 1,
5079     acc_error_system = 2,
5080     acc_error_execution = 3,
5081     acc_error_device_init = 4,
5082     acc_error_device_shutdown = 5,
5083     acc_error_device_unavailable = 6,
5084     acc_error_device_type_unavailable = 7,
5085     acc_error_wrong_device_type = 8,
5086     acc_error_out_of_memory = 9,
5087     acc_error_not_present = 10,
5088     acc_error_partly_present = 11,
5089     acc_error_present = 12,
5090     acc_error_invalid_argument = 13,
5091     acc_error_invalid_async = 14,
5092     acc_error_invalid_null_pointer = 15,
5093     acc_error_invalid_data_section = 16,
5094     acc_error_implementation_defined = 100
5095 }acc_error_t;

```

5096 The fields specific for an error event are:

5097 • **acc_event_t event_type** - The event type that triggered this callback. The only event
5098 that uses the **acc_error_event_info** struct is:

```
5099     acc_ev_error
```

5100 • **int implicit** - This will be set to 1.

5101 • **acc_error_t error_code** - The error codes used are:

- 5102 - **acc_error_other** is used for error conditions other than those described below.
- 5103 - **acc_error_system** is used when there is a system error condition.
- 5104 - **acc_error_execution** is used when there is an error condition issued from code
5105 executing on the device.
- 5106 - **acc_error_device_init** is used for any error initializing a device.
- 5107 - **acc_error_device_shutdown** is used for any error shutting down a device.
- 5108 - **acc_error_device_unavailable** is used when there is an error where the se-
5109 lected device is unavailable.
- 5110 - **acc_error_device_type_unavailable** is used when there is an error where
5111 no device of the selected device type is available or is supported.

- 5112 – **acc_error_wrong_device_type** is used when there is an error related to the
5113 device type, such as a mismatch between the device type for which a compute construct
5114 was compiled and the device available at runtime.
- 5115 – **acc_error_out_of_memory** is used when the program tries to allocate more mem-
5116 ory on the device than is available.
- 5117 – **acc_error_not_present** is used for an error related to data not being present at
5118 runtime.
- 5119 – **acc_error_partly_present** is used for an error related to part of the data being
5120 present but not being completely present at runtime.
- 5121 – **acc_error_present** is used for an error related to data being unexpectedly present
5122 at runtime.
- 5123 – **acc_error_invalid_argument** is used when an API routine is called with a
5124 invalid argument value, other than those described above.
- 5125 – **acc_error_invalid_async** is used when an API routine is called with an invalid
5126 *async-argument*, or when a directive is used with an invalid *async-argument*.
- 5127 – **acc_error_invalid_null_pointer** is used when an API routine is called with
5128 a null pointer argument where it is invalid, or when a directive is used with a null pointer
5129 in a context where it is invalid.
- 5130 – **acc_error_invalid_data_section** is used when an invalid array section ap-
5131 pears in a directive data clause, or an invalid array section appears as a runtime API call
5132 argument.
- 5133 – **acc_error_implementation_defined**: any value greater or equal to this value
5134 may be used for an implementation-defined error code.
- 5135 • **const char* error_message** - A pointer to null-terminated string containing an error
5136 message from the OpenACC runtime describing the error, or a null pointer.
- 5137 • **size_t runtime_info** - A value, such as an error code, from the underlying device
5138 runtime or driver, if one is available and appropriate.

5139 Other Events

5140 For any event that does not use the **acc_data_event_info**, **acc_launch_event_info**, or
5141 **acc_error_event_info** struct, the second argument to the callback routine will be a pointer
5142 to **acc_other_event_info** struct.

```
5143     typedef struct acc_other_event_info{
5144         acc_event_t event_type;
5145         int valid_bytes;
5146         acc_construct_t parent_construct;
5147         int implicit;
5148         void* tool_info;
5149     }acc_other_event_info;
```


5150 Parent Construct Enumeration

5151 All event structures contain a **parent_construct** member that describes the type of construct
 5152 that caused the event to be emitted. The purpose of this field is to provide a means to identify
 5153 the type of construct emitting the event in the cases where an event may be emitted by multi-
 5154 ple construct types, such as is the case with data and wait events. The possible values for the
 5155 **parent_construct** field are defined in the enumeration type **acc_construct_t**. In the
 5156 case of combined directives, the outermost construct of the combined construct is specified as the
 5157 **parent_construct**. If the event was emitted as the result of the application making a call to
 5158 the runtime api, the value will be **acc_construct_runtime_api**.

```
5159     typedef enum acc_construct_t{
5160         acc_construct_parallel = 0,
5161         acc_construct_serial = 16
5162         acc_construct_kernels = 1,
5163         acc_construct_loop = 2,
5164         acc_construct_data = 3,
5165         acc_construct_enter_data = 4,
5166         acc_construct_exit_data = 5,
5167         acc_construct_host_data = 6,
5168         acc_construct_atomic = 7,
5169         acc_construct_declare = 8,
5170         acc_construct_init = 9,
5171         acc_construct_shutdown = 10,
5172         acc_construct_set = 11,
5173         acc_construct_update = 12,
5174         acc_construct_routine = 13,
5175         acc_construct_wait = 14,
5176         acc_construct_runtime_api = 15,
5177     }acc_construct_t;
```

5178 5.2.3 Third Argument: API-Specific Information

5179 The third argument is a pointer to the **acc_api_info** struct type, shown here.

```
5180     typedef struct acc_api_info{
5181         acc_device_api device_api;
5182         int valid_bytes;
5183         acc_device_t device_type;
5184         int vendor;
5185         const void* device_handle;
5186         const void* context_handle;
5187         const void* async_handle;
5188     }acc_api_info;
```

5189 The fields are described below:

- 5190 • **acc_device_api device_api** - The API in use for this device. The data type is the
 5191 enumeration **acc_device_api**, which is described later in this section.
- 5192 • **int valid_bytes** - The number of valid bytes in this struct. See the discussion above in

5193 Section 5.2.1 First Argument: General Information.

- 5194 • **acc_device_t device_type** - The device type; the datatype is **acc_device_t**, de-
5195 fined in **openacc.h**.
- 5196 • **int vendor** - An identifier to identify the OpenACC vendor; contact your vendor to deter-
5197 mine the value used by that vendor's runtime.
- 5198 • **const void* device_handle** - If applicable, this will be a pointer to the API-specific
5199 device information.
- 5200 • **const void* context_handle** - If applicable, this will be a pointer to the API-specific
5201 context information.
- 5202 • **const void* async_handle** - If applicable, this will be a pointer to the API-specific
5203 async queue information.

5204 According to the value of **device_api** a library can cast the pointers of the fields **device_handle**,
5205 **context_handle** and **async_handle** to the respective device API type. The following device
5206 APIs are defined in the interface below. Any implementation-defined device API type must have a
5207 value greater than **acc_device_api_implementation_defined**.

```

5208     typedef enum acc_device_api{
5209         acc_device_api_none = 0,                /* no device API */
5210         acc_device_api_cuda = 1,              /* CUDA driver API */
5211         acc_device_api_opencl = 2,           /* OpenCL API */
5212         acc_device_api_other = 4,           /* other device API */
5213         acc_device_api_implementation_defined = 1000 /* other device API */
5214     }acc_device_api;
```

5210 5.3 Loading the Library

5211 This section describes how a tools library is loaded when the program is run. Four methods are
5212 described.

- 5213 • A tools library may be linked with the program, as any other library is linked, either as a
5214 static library or a dynamic library, and the runtime will call a predefined library initialization
5215 routine that will register the event callbacks.
- 5216 • The OpenACC runtime implementation may support a dynamic tools library, such as a shared
5217 object for Linux or OS/X, or a DLL for Windows, which is then dynamically loaded at runtime
5218 under control of the environment variable **ACC_PROFLIB**.
- 5219 • Some implementations where the OpenACC runtime is itself implemented as a dynamic li-
5220 brary may support adding a tools library using the **LD_PRELOAD** feature in Linux.
- 5221 • A tools library may be linked with the program, as in the first option, and the application itself
5222 may directly register event callback routines, or may invoke a library initialization routine that
5223 will register the event callbacks.

5224 Callbacks are registered with the runtime by calling **acc_callback_register** for each event
5225 as described in Section 5.4 Registering Event Callbacks. The prototype for **acc_callback_register**
5226 is:

```

5227     extern void acc_callback_register
5228         (acc_event_t event_type, acc_callback cb,
5229          acc_register_t info);

```

5230 The first argument to **acc_callback_register** is the event for which a callback is being
5231 registered (compare Section 5.1 Events). The second argument is a pointer to the callback routine:

```

5232     typedef void (*acc_callback)
5233         (acc_callback_info*, acc_event_info*, acc_api_info*);

```

5234 The third argument is an enum type:

```

5235     typedef enum acc_register_t{
5236         acc_reg = 0,
5237         acc_toggle = 1,
5238         acc_toggle_per_thread = 2
5239     }acc_register_t;

```

5240 This is usually **acc_reg**, but see Section 5.4.2 Disabling and Enabling Callbacks for cases where
5241 different values are used.

5242 An example of registering callbacks for launch, upload, and download events is:

```

5243     acc_callback_register(acc_ev_enqueue_launch_start,
5244                        prof_launch, acc_reg);
5245     acc_callback_register(acc_ev_enqueue_upload_start,
5246                        prof_data, acc_reg);
5247     acc_callback_register(acc_ev_enqueue_download_start,
5248                        prof_data, acc_reg);

```

5249 As shown in this example, the same routine (**prof_data**) can be registered for multiple events.
5250 The routine can use the **event_type** field in the **acc_callback_info** structure to determine
5251 for what event it was invoked.

5252 The names **acc_prof_register** and **acc_prof_unregister** are preserved for backward
5253 compatibility with previous versions of OpenACC.

5254 5.3.1 Library Registration

5255 The OpenACC runtime will invoke **acc_register_library**, passing the addresses of the reg-
5256 istration routines **acc_callback_register** and **acc_callback_unregister**, in case
5257 that routine comes from a dynamic library. In the third argument it passes the address of the lookup
5258 routine **acc_prof_lookup** to obtain the addresses of inquiry functions. No inquiry functions
5259 are defined in this profiling interface, but we preserve this argument for future support of sampling-
5260 based tools.

5261 Typically, the OpenACC runtime will include a *weak* definition of **acc_register_library**,
5262 which does nothing and which will be called when there is no tools library. In this case, the library
5263 can save the addresses of these routines and/or make registration calls to register any appropriate
5264 callbacks. The prototype for **acc_register_library** is:

```

5265     extern void acc_register_library
5266         (acc_prof_reg reg, acc_prof_reg unreg,

```

5267 **acc_prof_lookup_func lookup);**

5268 The first two arguments of this routine are of type:

```
5269       typedef void (*acc_prof_reg)
5270           (acc_event_t event_type, acc_callback cb,
5271           acc_register_t info);
```

5272 The third argument passes the address to the lookup function **acc_prof_lookup** to obtain the
5273 address of interface functions. It is of type:

```
5274       typedef void (*acc_query_fn) ();
5275       typedef acc_query_fn (*acc_prof_lookup_func)
5276           (const char* acc_query_fn_name);
```

5277 The argument of the lookup function is a string with the name of the inquiry function. There are no
5278 inquiry functions defined for this interface.

5279 **5.3.2 Statically-Linked Library Initialization**

5280 A tools library can be compiled and linked directly into the application. If the library provides an
5281 external routine **acc_register_library** as specified in Section 5.3.1 Library Registration, the
5282 runtime will invoke that routine to initialize the library.

5283 The sequence of events is:

- 5284 1. The runtime invokes the **acc_register_library** routine from the library.
- 5285 2. The **acc_register_library** routine calls **acc_callback_register** for each event
5286 to be monitored.
- 5287 3. **acc_callback_register** records the callback routines.
- 5288 4. The program runs, and your callback routines are invoked at the appropriate events.

5289 In this mode, only one tool library is supported.

5290 **5.3.3 Runtime Dynamic Library Loading**

5291 A common case is to build the tools library as a dynamic library (shared object for Linux or OS/X,
5292 DLL for Windows). In that case, you can have the OpenACC runtime load the library during initial-
5293 ization. This allows you to enable runtime profiling without rebuilding or even relinking your ap-
5294 plication. The dynamic library must implement a registration routine **acc_register_library**
5295 as specified in Section 5.3.1 Library Registration.

5296 The user may set the environment variable **ACC_PROFLIB** to the path to the library will tell the
5297 OpenACC runtime to load your dynamic library at initialization time:

```
5298       Bash:
5299           export ACC_PROFLIB=/home/user/lib/myprof.so
5300           ./myapp
5301       or
5302           ACC_PROFLIB=/home/user/lib/myprof.so ./myapp
```

5303 C-shell:

```
5304     setenv ACC_PROFLIB /home/user/lib/myprof.so
5305     ./myapp
```

5306 When the OpenACC runtime initializes, it will read the **ACC_PROFLIB** environment variable (with
5307 **getenv**). The runtime will open the dynamic library (using **dlopen** or **LoadLibraryA**); if
5308 the library cannot be opened, the runtime may cause the program to halt execution and return an
5309 error status, or may continue execution with or without an error message. If the library is success-
5310 fully opened, the runtime will get the address of the **acc_register_library** routine (using
5311 **dlsym** or **GetProcAddress**). If this routine is resolved in the library, it will be invoked pass-
5312 ing in the addresses of the registration routine **acc_callback_register**, the deregistration
5313 routine **acc_callback_unregister**, and the lookup routine **acc_prof_lookup**. The reg-
5314 istration routine in your library, **acc_register_library**, registers the callbacks by calling the
5315 **register** argument, and must save the addresses of the arguments (**register**, **unregister**,
5316 and **lookup**) for later use, if needed.

5317 The sequence of events is:

- 5318 1. Initialization of the OpenACC runtime.
- 5319 2. OpenACC runtime reads **ACC_PROFLIB**.
- 5320 3. OpenACC runtime loads the library.
- 5321 4. OpenACC runtime calls the **acc_register_library** routine in that library.
- 5322 5. Your **acc_register_library** routine calls **acc_callback_register** for each event
5323 to be monitored.
- 5324 6. **acc_callback_register** records the callback routines.
- 5325 7. The program runs, and your callback routines are invoked at the appropriate events.

5326 If supported, paths to multiple dynamic libraries may be specified in the **ACC_PROFLIB** environ-
5327 ment variable, separated by semicolons (;). The OpenACC runtime will open these libraries and in-
5328 voke the **acc_register_library** routine for each, in the order they appear in **ACC_PROFLIB**.

5329 5.3.4 Preloading with LD_PRELOAD

5330 The implementation may also support dynamic loading of a tools library using the **LD_PRELOAD**
5331 feature available in some systems. In such an implementation, you need only specify your tools
5332 library path in the **LD_PRELOAD** environment variable before executing your program. The Open-
5333 ACC runtime will invoke the **acc_register_library** routine in your tools library at initial-
5334 ization time. This requires that the OpenACC runtime include a dynamic library with a default
5335 (empty) implementation of **acc_register_library** that will be invoked in the normal case
5336 where there is no **LD_PRELOAD** setting. If an implementation only supports static linking, or if the
5337 application is linked without dynamic library support, this feature will not be available.

5338 Bash:

```
5339     export LD_PRELOAD=/home/user/lib/myprof.so
5340     ./myapp
5341 or
5342     LD_PRELOAD=/home/user/lib/myprof.so ./myapp
```

```
5343 C-shell:
5344     setenv LD_PRELOAD /home/user/lib/myprof.so
5345     ./myapp
```

5346 The sequence of events is:

- 5347 1. The operating system loader loads the library specified in **LD_PRELOAD**.
- 5348 2. The call to **acc_register_library** in the OpenACC runtime is resolved to the routine
5349 in the loaded tools library.
- 5350 3. OpenACC runtime calls the **acc_register_library** routine in that library.
- 5351 4. Your **acc_register_library** routine calls **acc_callback_register** for each event
5352 to be monitored.
- 5353 5. **acc_callback_register** records the callback routines.
- 5354 6. The program runs, and your callback routines are invoked at the appropriate events.

5355 In this mode, only a single tools library is supported, since only one **acc_register_library**
5356 initialization routine will get resolved by the dynamic loader.

5357 5.3.5 Application-Controlled Initialization

5358 An alternative to default initialization is to have the application itself call the library initialization
5359 routine, which then calls **acc_callback_register** for each appropriate event. The library
5360 may be statically linked to the application or your application may dynamically load the library.

5361 The sequence of events is:

- 5362 1. Your application calls the library initialization routine.
- 5363 2. The library initialization routine calls **acc_callback_register** for each event to be
5364 monitored.
- 5365 3. **acc_callback_register** records the callback routines.
- 5366 4. The program runs, and your callback routines are invoked at the appropriate events.

5367 In this mode, multiple tools libraries can be supported, with each library initialization routine in-
5368 voked by the application.

5369 5.4 Registering Event Callbacks

5370 This section describes how to register and unregister callbacks, temporarily disabling and enabling
5371 callbacks, the behavior of dynamic registration and unregistration, and requirements on an Open-
5372 ACC implementation to correctly support the interface.

5373 5.4.1 Event Registration and Unregistration

5374 The library must call the registration routine **acc_callback_register** to register each call-
5375 back with the runtime. A simple example:

```
5376     extern void prof_data(acc_callback_info* profinfo,  
5377                        acc_event_info* eventinfo, acc_api_info* apiinfo);
```

```

5378     extern void prof_launch(acc_callback_info* profinfo,
5379                           acc_event_info* eventinfo, acc_api_info* apiinfo);
5380     ...
5381     void acc_register_library(acc_prof_reg reg,
5382                             acc_prof_reg unreg, acc_prof_lookup_func lookup){
5383         reg(acc_ev_enqueue_upload_start, prof_data, acc_reg);
5384         reg(acc_ev_enqueue_download_start, prof_data, acc_reg);
5385         reg(acc_ev_enqueue_launch_start, prof_launch, acc_reg);
5386     }

```

5387 In this example the `prof_data` routine will be invoked for each data upload and download event,
5388 and the `prof_launch` routine will be invoked for each launch event. The `prof_data` routine
5389 might start out with:

```

5390     void prof_data(acc_callback_info* profinfo,
5391                  acc_event_info* eventinfo, acc_api_info* apiinfo){
5392         acc_data_event_info* datainfo;
5393         datainfo = (acc_data_event_info*)eventinfo;
5394         switch( datainfo->event_type ){
5395             case acc_ev_enqueue_upload_start :
5396                 ...
5397         }
5398     }

```

5399 Multiple Callbacks

5400 Multiple callback routines can be registered on the same event:

```

5401     acc_callback_register(acc_ev_enqueue_upload_start,
5402                          prof_data, acc_reg);
5403     acc_callback_register(acc_ev_enqueue_upload_start,
5404                          prof_up, acc_reg);

```

5405 For most events, the callbacks will be invoked in the order in which they are registered. However,
5406 *end* events, named `acc_ev_..._end`, invoke callbacks in the reverse order. Essentially, each
5407 event has an ordered list of callback routines. A new callback routine is appended to the tail of the
5408 list for that event. For most events, that list is traversed from the head to the tail, but for *end*
5409 events, the list is traversed from the tail to the head.

5410 If a callback is registered, then later unregistered, then later still registered again, the second regis-
5411 tration is considered to be a new callback, and the callback routine will then be appended to the tail
5412 of the callback list for that event.

5413 Unregistering

5414 A matching call to `acc_callback_unregister` will remove that routine from the list of call-
5415 back routines for that event.

```

5416     acc_callback_register(acc_ev_enqueue_upload_start,
5417                          prof_data, acc_reg);
5418     // prof_data is on the callback list for acc_ev_enqueue_upload_start

```

```

5419     ...
5420     acc_callback_unregister(acc_ev_enqueue_upload_start,
5421                             prof_data, acc_reg);
5422     // prof_data is removed from the callback list
5423     // for acc_ev_enqueue_upload_start

```

Each entry on the callback list must also have a *ref* count. This keeps track of how many times this routine was added to this event's callback list. If a routine is registered *n* times, it must be unregistered *n* times before it is removed from the list. Note that if a routine is registered multiple times for the same event, its *ref* count will be incremented with each registration, but it will only be invoked once for each event instance.

5.4.2 Disabling and Enabling Callbacks

A callback routine may be temporarily disabled on the callback list for an event, then later re-enabled. The behavior is slightly different than unregistering and later re-registering that event. When a routine is disabled and later re-enabled, the routine's position on the callback list for that event is preserved. When a routine is unregistered and later re-registered, the routine's position on the callback list for that event will move to the tail of the list. Also, unregistering a callback must be done *n* times if the callback routine was registered *n* times. In contrast, disabling, and enabling an event sets a toggle. Disabling a callback will immediately reset the toggle and disable calls to that routine for that event, even if it was enabled multiple times. Enabling a callback will immediately set the toggle and enable calls to that routine for that event, even if it was disabled multiple times. Registering a new callback initially sets the toggle.

A call to **acc_callback_unregister** with a value of **acc_toggle** as the third argument will disable callbacks to the given routine. A call to **acc_callback_register** with a value of **acc_toggle** as the third argument will enable those callbacks.

```

5443     acc_callback_unregister(acc_ev_enqueue_upload_start,
5444                             prof_data, acc_toggle);
5445     // prof_data is disabled
5446     ...
5447     acc_callback_register(acc_ev_enqueue_upload_start,
5448                             prof_data, acc_toggle);
5449     // prof_data is re-enabled

```

A call to either **acc_callback_unregister** or **acc_callback_register** to disable or enable a callback when that callback is not currently registered for that event will be ignored with no error.

All callbacks for an event may be disabled (and re-enabled) by passing **NULL** to the second argument and **acc_toggle** to the third argument of **acc_callback_unregister** (and **acc_callback_register**). This sets a toggle for that event, which is distinct from the toggle for each callback for that event. While the event is disabled, no callbacks for that event will be invoked. Callbacks for that event can be registered, unregistered, enabled, and disabled while that event is disabled, but no callbacks will be invoked for that event until the event itself is enabled. Initially, all events are enabled.

```

5460     acc_callback_unregister(acc_ev_enqueue_upload_start,
5461                             prof_data, acc_toggle);

```



```

5462 // prof_data is disabled
5463 ...
5464 acc_callback_unregister(acc_ev_enqueue_upload_start,
5465                        NULL, acc_toggle);
5466 // acc_ev_enqueue_upload_start callbacks are disabled
5467 ...
5468 acc_callback_register(acc_ev_enqueue_upload_start,
5469                      prof_data, acc_toggle);
5470 // prof_data is re-enabled, but
5471 // acc_ev_enqueue_upload_start callbacks still disabled
5472 ...
5473 acc_callback_register(acc_ev_enqueue_upload_start,
5474                      prof_up, acc_reg);
5475 // prof_up is registered and initially enabled, but
5476 // acc_ev_enqueue_upload_start callbacks still disabled
5477 ...
5478 acc_callback_register(acc_ev_enqueue_upload_start,
5479                      NULL, acc_toggle);
5480 // acc_ev_enqueue_upload_start callbacks are enabled
5481

```

5482 Finally, all callbacks can be disabled (and enabled) by passing the argument list (**acc_ev_none**,
5483 **NULL**, **acc_toggle**) to **acc_callback_unregister** (and **acc_callback_register**).
5484 This sets a global toggle disabling all callbacks, which is distinct from the toggle enabling callbacks
5485 for each event and the toggle enabling each callback routine.

5486 The behavior of passing **acc_ev_none** as the first argument and a non-**NULL** value as the second
5487 argument to **acc_callback_unregister** or **acc_callback_register** is not defined,
5488 and may be ignored by the runtime without error.

5489 All callbacks can be disabled (or enabled) for just the current thread by passing the argument list
5490 (**acc_ev_none**, **NULL**, **acc_toggle_per_thread**) to **acc_callback_unregister**
5491 (and **acc_callback_register**). This is the only thread-specific interface to
5492 **acc_callback_register** and **acc_callback_unregister**, all other calls to register,
5493 unregister, enable, or disable callbacks affect all threads in the application.

5494 5.5 Advanced Topics

5495 This section describes advanced topics such as dynamic registration and changes of the execution
5496 state for callback routines as well as the runtime and tool behavior for multiple host threads.

5497 5.5.1 Dynamic Behavior

5498 Callback routines may be registered or unregistered, enabled or disabled at any point in the execution
5499 of the program. Calls may appear in the library itself, during the processing of an event. The
5500 OpenACC runtime must allow for this case, where the callback list for an event is modified while
5501 that event is being processed.

5502 **Dynamic Registration and Unregistration**

5503 Calls to **acc_register** and **acc_unregister** may occur at any point in the application. A
5504 callback routine can be registered or unregistered from a callback routine, either the same routine
5505 or another routine, for a different event or the same event for which the callback was invoked. If a
5506 callback routine is registered for an event while that event is being processed, then the new callback
5507 routine will be added to the tail of the list of callback routines for this event. Some events (the
5508 **_end**) events process the callback routines in reverse order, from the tail to the head. For those
5509 events, adding a new callback routine will not cause the new routine to be invoked for this instance
5510 of the event. The other events process the callback routines in registration order, from the head
5511 to the tail. Adding a new callback routine for such an event will cause the runtime to invoke that
5512 newly registered callback routine for this instance of the event. Both the runtime and the library
5513 must implement and expect this behavior.

5514 If an existing callback routine is unregistered for an event while that event is being processed, that
5515 callback routine is removed from the list of callbacks for this event. For any event, if that callback
5516 routine had not yet been invoked for this instance of the event, it will not be invoked.

5517 Registering and unregistering a callback routine is a global operation and affects all threads, in a
5518 multithreaded application. See Section 5.4.1 Multiple Callbacks.

5519 **Dynamic Enabling and Disabling**

5520 Calls to **acc_register** and **acc_unregister** to enable and disable a specific callback for
5521 an event, enable or disable all callbacks for an event, or enable or disable all callbacks may occur
5522 at any point in the application. A callback routine can be enabled or disabled from a callback
5523 routine, either the same routine or another routine, for a different event or the same event for which
5524 the callback was invoked. If a callback routine is enabled for an event while that event is being
5525 processed, then the new callback routine will be immediately enabled. If it appears on the list of
5526 callback routines closer to the head (for **_end** events) or closer to the tail (for other events), that
5527 newly-enabled callback routine will be invoked for this instance of this event, unless it is disabled
5528 or unregistered before that callback is reached.

5529 If a callback routine is disabled for an event while that event is being processed, that callback routine
5530 is immediately disabled. For any event, if that callback routine had not yet been invoked for this in-
5531 stance of the event, it will not be invoked, unless it is enabled before that callback routine is reached
5532 in the list of callbacks for this event. If all callbacks for an event are disabled while that event is
5533 being processed, or all callbacks are disabled for all events while an event is being processed, then
5534 when this callback routine returns, no more callbacks will be invoked for this instance of the event.

5535 Registering and unregistering a callback routine is a global operation and affects all threads, in a
5536 multithreaded application. See Section 5.4.1 Multiple Callbacks.

5537 **5.5.2 OpenACC Events During Event Processing**

5538 OpenACC events may occur during event processing. This may be because of OpenACC API rou-
5539 tine calls or OpenACC constructs being reached during event processing, or because of multiple host
5540 threads executing asynchronously. Both the OpenACC runtime and the tool library must implement
5541 the proper behavior.

5542 5.5.3 Multiple Host Threads

5543 Many programs that use OpenACC also use multiple host threads, such as programs using the
5544 OpenMP API. The appearance of multiple host threads affects both the OpenACC runtime and the
5545 tools library.

5546 Runtime Support for Multiple Threads

5547 The OpenACC runtime must be thread-safe, and the OpenACC runtime implementation of this
5548 tools interface must also be thread-safe. All threads use the same set of callbacks for all events, so
5549 registering a callback from one thread will cause all threads to execute that callback. This means that
5550 managing the callback lists for each event must be protected from multiple simultaneous updates.
5551 This includes adding a callback to the tail of the callback list for an event, removing a callback from
5552 the list for an event, and incrementing or decrementing the *ref* count for a callback routine for an
5553 event.

5554 In addition, one thread may register, unregister, enable, or disable a callback for an event while
5555 another thread is processing the callback list for that event asynchronously. The exact behavior may
5556 be dependent on the implementation, but some behaviors are expected and others are disallowed.
5557 In the following examples, there are three callbacks, A, B, and C, registered for event E in that
5558 order, where callbacks A and B are enabled and callback C is temporarily disabled. Thread T1 is
5559 dynamically modifying the callbacks for event E while thread T2 is processing an instance of event
5560 E.

- 5561 • Suppose thread T1 unregisters or disables callback A for event E. Thread T2 may or may not
5562 invoke callback A for this event instance, but it must invoke callback B; if it invokes callback
5563 A, that must precede the invocation of callback B.
- 5564 • Suppose thread T1 unregisters or disables callback B for event E. Thread T2 may or may not
5565 invoke callback B for this event instance, but it must invoke callback A; if it invokes callback
5566 B, that must follow the invocation of callback A.
- 5567 • Suppose thread T1 unregisters or disables callback A and then unregisters or disables callback
5568 B for event E. Thread T2 may or may not invoke callback A and may or may not invoke
5569 callback B for this event instance, but if it invokes both callbacks, it must invoke callback A
5570 before it invokes callback B.
- 5571 • Suppose thread T1 unregisters or disables callback B and then unregisters or disables callback
5572 A for event E. Thread T2 may or may not invoke callback A and may or may not invoke
5573 callback B for this event instance, but if it invokes callback B, it must have invoked callback
5574 A for this event instance.
- 5575 • Suppose thread T1 is registering a new callback D for event E. Thread T2 may or may not
5576 invoke callback D for this event instance, but it must invoke both callbacks A and B. If it
5577 invokes callback D, that must follow the invocations of A and B.
- 5578 • Suppose thread T1 is enabling callback C for event E. Thread T2 may or may not invoke
5579 callback C for this event instance, but it must invoke both callbacks A and B. If it invokes
5580 callback C, that must follow the invocations of A and B.

5581 The `acc_callback_info` struct has a `thread_id` field, which the runtime must set to a
5582 unique value for each host thread, though it need not be the same as the OpenMP threadnum value.

5583 **Library Support for Multiple Threads**

5584 The tool library must also be thread-safe. The callback routine will be invoked in the context of the
5585 thread that reaches the event. The library may receive a callback from a thread T2 while it's still
5586 processing a callback, from the same event type or from a different event type, from another thread
5587 T1. The **acc_callback_info** struct has a **thread_id** field, which the runtime must set to a
5588 unique value for each host thread.

5589 If the tool library uses dynamic callback registration and unregistration, or callback disabling and
5590 enabling, recall that unregistering or disabling an event callback from one thread will unregister or
5591 disable that callback for all threads, and registering or enabling an event callback from any thread
5592 will register or enable it for all threads. If two or more threads register the same callback for the
5593 same event, the behavior is the same as if one thread registered that callback multiple times; see
5594 Section 5.4.1 Multiple Callbacks. The **acc_unregister** routine must be called as many times
5595 as **acc_register** for that callback/event pair in order to totally unregister it. If two threads
5596 register two different callback routines for the same event, unless the order of the registration calls
5597 is guaranteed by some synchronization method, the order in which the runtime sees the registration
5598 may differ for multiple runs, meaning the order in which the callbacks occur will differ as well.

6. Glossary

5599

5600 Clear and consistent terminology is important in describing any programming model. We define
5601 here the terms you must understand in order to make effective use of this document and the asso-
5602 ciated programming model. In particular, some terms used in this specification conflict with their
5603 usage in the base language specifications. When there is potential confusion, the term will appear
5604 here.

5605 **Accelerator** – a device attached to a CPU and to which the CPU can offload data and compute
5606 kernels to perform compute-intensive calculations.

5607 **Accelerator routine** – a procedure compiled for the accelerator with the **routine** directive.

5608 **Accelerator thread** – a thread of execution that executes on the accelerator; a single vector lane of
5609 a single worker of a single gang.

5610 **Aggregate datatype** – any non-scalar datatype such as array and composite datatypes. In Fortran,
5611 aggregate datatypes include arrays, derived types, character types. In C, aggregate datatypes include
5612 arrays, targets of pointers, structs, and unions. In C++, aggregate datatypes include arrays, targets
5613 of pointers, classes, structs, and unions.

5614 **Aggregate variables** – a variable of any non-scalar datatype, including array or composite variables.
5615 In Fortran, this includes any variable with allocatable or pointer attribute and character variables.

5616 **Async-argument** – an *async-argument* is a nonnegative scalar integer expression (*int* for C or C++,
5617 *integer* for Fortran), or one of the special values **acc_async_noval** or **acc_async_sync**.

5618 **Barrier** – a type of synchronization where all parallel execution units or threads must reach the
5619 barrier before any execution unit or thread is allowed to proceed beyond the barrier; modeled after
5620 the starting barrier on a horse race track.

5621 **Block construct** – a *block-construct*, as specified by the Fortran language.

5622 **Captured variable** – a variable for which a distinct copy from its original variable exists in the
5623 device-accessible memory. Such variable is only captured from the time its copy is created and
5624 until such a copy is deleted.

5625 **Composite datatype** – a derived type in Fortran, or a **struct** or **union** type in C, or a **class**,
5626 **struct**, or **union** type in C++. (This is different from the use of the term *composite data type* in
5627 the C and C++ languages.)

5628 **Composite variable** – a variable of composite datatype. In Fortran, a composite variable must not
5629 have allocatable or pointer attributes.

5630 **Compute construct** – a *parallel construct*, *serial construct*, or *kernels construct*.

5631 **Compute intensity** – for a given loop, region, or program unit, the ratio of the number of arithmetic
5632 operations performed on computed data divided by the number of memory transfers required to
5633 move that data between two levels of a memory hierarchy.

5634 **Compute region** – a *parallel region*, *serial region*, or *kernels region*.

5635 **Construct** – a directive and the associated statement, loop, or structured block, if any.

- 5636 **CUDA** – the CUDA environment from NVIDIA, a C-like programming environment used to ex-
5637 plicitly control and program an NVIDIA GPU.
- 5638 **Current device** – the device represented by the *acc-current-device-type-var* and *acc-current-device-*
5639 *num-var* ICVs
- 5640 **Current device type** – the device type represented by the *acc-current-device-type-var* ICV
- 5641 **Data lifetime** – the lifetime of a data object in device memory, which may begin at the entry to
5642 a data region, or at an **enter data** directive, or at a data API call such as **acc_copyin** or
5643 **acc_create**, and which may end at the exit from a data region, or at an **exit data** directive,
5644 or at a data API call such as **acc_delete**, **acc_copyout**, or **acc_shutdown**, or at the end of
5645 the program execution.
- 5646 **Data region** – a *region* defined by a **data** construct, or an implicit data region for a function or
5647 subroutine containing OpenACC directives. Data constructs typically allocate device memory and
5648 copy data from host to device memory upon entry, and copy data from device to local memory and
5649 deallocate device memory upon exit. Data regions may contain other data regions and compute
5650 regions.
- 5651 **Default asynchronous queue** – the asynchronous activity queue represented in the *acc-default-*
5652 *async-var* ICV
- 5653 **Device** – a general reference to an accelerator or a multicore CPU.
- 5654 **Device-accessible memory** – any memory which can be accessed from the device.
- 5655 **Device memory** – memory attached to a device, logically and physically separate from the host
5656 memory.
- 5657 **Device thread** – a thread of execution that executes on any device.
- 5658 **Directive** – in C or C++, a **#pragma**, or in Fortran, a specially formatted comment statement, that
5659 is interpreted by a compiler to augment information about or specify the behavior of the program.
- 5660 **Discrete memory** – memory accessible from the local thread that is not accessible from the current
5661 device, or memory accessible from the current device that is not accessible from the local thread.
- 5662 **DMA** – Direct Memory Access, a method to move data between physically separate memories;
5663 this is typically performed by a DMA engine, separate from the host CPU, that can access the host
5664 physical memory as well as an IO device or other physical memory.
- 5665 **Exposed variable access** – with respect to a compute construct, any access to the data or address
5666 of a variable at a point within the compute construct where the variable is not private to a scope
5667 lexically enclosed within the compute construct. See Section 2.6.2.
- 5668 **false** – a condition that evaluates to zero in C or C++, or **.false.** in Fortran.
- 5669 **GPU** – a Graphics Processing Unit; one type of accelerator.
- 5670 **GPGPU** – General Purpose computation on Graphics Processing Units.
- 5671 **Host** – the main CPU that in this context may have one or more attached accelerators. The host
5672 CPU controls the program regions and data loaded into and executed on one or more devices.
- 5673 **Host thread** – a thread of execution that executes on the host.

- 5674 **Implicit data region** – the data region that is implicitly defined for a Fortran subprogram or C
5675 function. A call to a subprogram or function enters the implicit data region, and a return from the
5676 subprogram or function exits the implicit data region.
- 5677 **Kernel** – a nested loop executed in parallel by the accelerator. Typically the loops are divided into
5678 a parallel domain, and the body of the loop becomes the body of the kernel.
- 5679 **Kernels region** – a *region* defined by a **kernels** construct. A kernels region is a structured block
5680 which is compiled for the accelerator. The code in the kernels region will be divided by the compiler
5681 into a sequence of kernels; typically each loop nest will become a single kernel. A kernels region
5682 may require space in device memory to be allocated and data to be copied from local memory to
5683 device memory upon region entry, and data to be copied from device memory to local memory and
5684 space in device memory to be deallocated upon exit.
- 5685 **Level of parallelism** – one of the following, which are arranged from the highest to the lowest level:
5686 gang dimension three, gang dimension two, gang dimension one, worker, vector, or sequential.
5687 One or more of gang, worker, and vector parallelism may appear on a loop construct. Sequential
5688 execution corresponds to no parallelism. The **gang**, **worker**, **vector**, and **seq** clauses specify
5689 the level of parallelism for a loop.
- 5690 **Local device** – the device where the *local thread* executes.
- 5691 **Local memory** – the memory associated with the *local thread*.
- 5692 **Local thread** – the host thread or the accelerator thread that executes an OpenACC directive or
5693 construct.
- 5694 **Loop trip count** – the number of times a particular loop executes.
- 5695 **MIMD** – a method of parallel execution (Multiple Instruction, Multiple Data) where different execution
5696 units or threads execute different instruction streams asynchronously with each other.
- 5697 **null pointer** – a C or C++ pointer variable with the value zero, **NULL**, or (in C++) **nullptr**, or a
5698 Fortran **pointer** variable that is not associated, or a Fortran **allocatable** variable that is not
5699 allocated.
- 5700 **OpenCL** – short for Open Compute Language, a developing, portable standard C-like programming
5701 environment that enables low-level general-purpose programming on GPUs and other accelerators.
- 5702 **Orphaned loop construct** – a **loop** construct that has no parent compute construct.
- 5703 **Parallel region** – a *region* defined by a **parallel** construct. A parallel region is a structured block
5704 which is compiled for the accelerator. A parallel region typically contains one or more work-sharing
5705 loops. A parallel region may require space in device memory to be allocated and data to be copied
5706 from local memory to device memory upon region entry, and data to be copied from device memory
5707 to local memory and space in device memory to be deallocated upon exit.
- 5708 **Parent compute construct** – for any point in the program, the nearest lexically enclosing compute
5709 construct that has the same parent procedure.
- 5710 **Parent compute scope** – for any point in the program, the parent compute construct or, if none, the
5711 parent procedure.
- 5712 **Parent procedure** – for any point in the program, the nearest lexically enclosing procedure such
5713 that expressions at this point are not evaluated until the procedure is called.

- 5714 **Partly present data** – a section of data for which some of the data is present in a single device
5715 memory section, but part of the data is either not present or is present in a different device memory
5716 section. For instance, if a subarray of an array is present, the array is partly present.
- 5717 **Present data** – data for which the sum of the structured and dynamic reference counters is greater
5718 than zero in a single device memory section; see Section 2.6.7. A null pointer is defined as always
5719 present with a length of zero bytes.
- 5720 **Private data** – with respect to an iterative loop, data which is used only during a particular loop
5721 iteration. With respect to a more general region of code, data which is used within the region but is
5722 not initialized prior to the region and is re-initialized prior to any use after the region.
- 5723 **Procedure** – in C or C++, a function or C++ lambda; in Fortran, a subroutine or function.
- 5724 **Region** – all the code encountered during an instance of execution of a construct. A region includes
5725 any code in called routines, and may be thought of as the dynamic extent of a construct. This may
5726 be a *parallel region*, *serial region*, *kernels region*, *data region*, or *implicit data region*.
- 5727 **Scalar** – a variable of scalar datatype. In Fortran, scalars must not have allocatable or pointer
5728 attributes.
- 5729 **Scalar datatype** – an intrinsic or built-in datatype that is not an array or aggregate datatype. In For-
5730 tran, scalar datatypes are integer, real, double precision, complex, or logical. In C, scalar datatypes
5731 are char (signed or unsigned), int (signed or unsigned, with optional short, long or long long at-
5732 tribute), enum, float, double, long double, `_Complex` (with optional float or long attribute), or any
5733 pointer datatype. In C++, scalar datatypes are char (signed or unsigned), `wchar_t`, int (signed or
5734 unsigned, with optional short, long or long long attribute), enum, bool, float, double, long double,
5735 or any pointer datatype. Not all implementations or targets will support all of these datatypes.
- 5736 **Serial region** – a *region* defined by a **serial** construct. A serial region is a structured block which
5737 is compiled for the accelerator. A serial region contains code that is executed by a single gang of a
5738 single worker with a vector length of one. A serial region may require space in device memory to be
5739 allocated and data to be copied from local memory to device memory upon region entry, and data
5740 to be copied from device memory to local memory and space in device memory to be deallocated
5741 upon exit.
- 5742 **Shared memory** – memory that is accessible from both the local thread and the current device.
- 5743 **SIMD** – a method of parallel execution (single-instruction, multiple-data) where the same instruc-
5744 tion is applied to multiple data elements simultaneously.
- 5745 **SIMD operation** – a *vector operation* implemented with SIMD instructions.
- 5746 **Structured block** – in C or C++, an executable statement, possibly compound, with a single entry
5747 at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single
5748 entry at the top and a single exit at the bottom.
- 5749 **Thread** – a host CPU thread or an accelerator thread. On a host CPU, a thread is defined by a
5750 program counter and stack location; several host threads may comprise a process and share host
5751 memory. On an accelerator, a thread is any one vector lane of one worker of one gang.
- 5752 **true** – a condition that evaluates to nonzero in C or C++, or `.true.` in Fortran.
- 5753 **var** – the name of a variable (scalar, array, or composite variable), or a subarray specification, or an
5754 array element, or a composite variable member, or the name of a Fortran common block between

5755 slashes.

5756 **Vector operation** – a single operation or sequence of operations applied uniformly to each element
5757 of an array.

5758 **Visible data clause** – with respect to a compute construct, any data clause on the compute con-
5759 struct, on a lexically enclosing **data** construct that has the same parent procedure, or on a visible
5760 **declare** directive. See Section 2.6.2.

5761 **Visible default clause** – with respect to a compute construct, the nearest **default** clause ap-
5762 pearing on the compute construct or on a lexically enclosing **data** construct that has the same
5763 parent procedure. See Section 2.6.2.

5764 **Visible device copy** – a copy of a variable, array, or subarray allocated in device memory that is
5765 visible to the program unit being compiled.

5766 **A. Recommendations for Implementers**

5767 This section gives recommendations for standard names and extensions to use for implementations
5768 for specific targets and target platforms, to promote portability across such implementations, and
5769 recommended options that programmers find useful. While this appendix is not part of the Open-
5770 ACC specification, implementations that provide the functionality specified herein are strongly rec-
5771 ommended to use the names in this section. The first subsection describes devices, such as NVIDIA
5772 GPUs. The second subsection describes additional API routines for target platforms, such as CUDA
5773 and OpenCL. The third subsection lists several recommended options for implementations.

5774 **A.1 Target Devices**

5775 **A.1.1 NVIDIA GPU Targets**

5776 This section gives recommendations for implementations that target NVIDIA GPU devices.

5777 **Accelerator Device Type**

5778 These implementations should use the name `acc_device_nvidia` for the `acc_device_t`
5779 type or return values from OpenACC Runtime API routines.

5780 **ACC_DEVICE_TYPE**

5781 An implementation should use the case-insensitive name `nvidia` for the environment variable
5782 `ACC_DEVICE_TYPE`.

5783 **device_type clause argument**

5784 An implementation should use the case-insensitive name `nvidia` as the argument to the `device_type`
5785 clause.

5786 **A.1.2 AMD GPU Targets**

5787 This section gives recommendations for implementations that target AMD GPUs.

5788 **Accelerator Device Type**

5789 These implementations should use the name `acc_device_radeon` for the `acc_device_t`
5790 type or return values from OpenACC Runtime API routines.

5791 **ACC_DEVICE_TYPE**

5792 These implementations should use the case-insensitive name `radeon` for the environment variable
5793 `ACC_DEVICE_TYPE`.

5794 **device_type clause argument**

5795 An implementation should use the case-insensitive name `radeon` as the argument to the `device_type`
5796 clause.

5797 **A.1.3 Multicore Host CPU Target**

5798 This section gives recommendations for implementations that target the multicore host CPU.

5799 **Accelerator Device Type**

5800 These implementations should use the name **acc_device_host** for the **acc_device_t** type
5801 or return values from OpenACC Runtime API routines.

5802 **ACC_DEVICE_TYPE**

5803 These implementations should use the case-insensitive name **host** for the environment variable
5804 **ACC_DEVICE_TYPE**.

5805 **device_type clause argument**

5806 An implementation should use the case-insensitive name **host** as the argument to the **device_type**
5807 clause.

5808 **routine directive**

5809 Given a **routine** directive for a procedure, an implementation should:

- 5810 • Suppress the procedure's compilation for the multicore host CPU if a **nohost** clause appears.
- 5811 • Ignore any **bind** clause when compiling the procedure for the multicore host CPU.
- 5812 • Disallow a **bind** clause to appear after a **device_type(host)** clause.

5813 **A.2 API Routines for Target Platforms**

5814 These runtime routines allow access to the interface between the OpenACC runtime API and the
5815 underlying target platform. An implementation may not implement all these routines, but if it
5816 provides this functionality, it should use these function names.

5817 **A.2.1 NVIDIA CUDA Platform**

5818 This section gives runtime API routines for implementations that target the NVIDIA CUDA Run-
5819 time or Driver API.

5820 **acc_get_current_cuda_device**

5821 **Summary**

5822 The **acc_get_current_cuda_device** routine returns the NVIDIA CUDA device handle for
5823 the current device.

5824 **Format**

5825 C or C++:

```
5826     void* acc_get_current_cuda_device ();
```

5827 acc_get_current_cuda_context**5828 Summary**

5829 The **acc_get_current_cuda_context** routine returns the NVIDIA CUDA context handle
5830 in use for the current device.

5831 Format

5832 C or C++:

```
5833     void* acc_get_current_cuda_context ();
```

5834 acc_get_cuda_stream**5835 Summary**

5836 The **acc_get_cuda_stream** routine returns the NVIDIA CUDA stream handle in use for the
5837 current device for the asynchronous activity queue associated with the **async** argument. This
5838 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5839 Format

5840 C or C++:

```
5841     void* acc_get_cuda_stream ( int async );
```

5842 acc_set_cuda_stream**5843 Summary**

5844 The **acc_set_cuda_stream** routine sets the NVIDIA CUDA stream handle the current device
5845 for the asynchronous activity queue associated with the **async** argument. This argument must be
5846 an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5847 Format

5848 C or C++:

```
5849     void acc_set_cuda_stream ( int async, void* stream );
```

5850 A.2.2 OpenCL Target Platform

5851 This section gives runtime API routines for implementations that target the OpenCL API on any
5852 device.

5853 acc_get_current_opengl_device**5854 Summary**

5855 The **acc_get_current_opengl_device** routine returns the OpenCL device handle for the
5856 current device.

5857 Format

5858 C or C++:

```
5859     void* acc_get_current_opengl_device ();
```

5860 acc_get_current_opengl_context**5861 Summary**

5862 The **acc_get_current_opengl_context** routine returns the OpenCL context handle in use
5863 for the current device.

5864 **Format**

5865 C or C++:

```
5866     void* acc_get_current_openc1_context ( );
```

5867 **acc_get_openc1_queue**

5868 **Summary**

5869 The **acc_get_openc1_queue** routine returns the OpenCL command queue handle in use for
5870 the current device for the asynchronous activity queue associated with the **async** argument. This
5871 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5872 **Format**

5873 C or C++:

```
5874     cl_command_queue acc_get_openc1_queue ( int async );
```

5875 **acc_set_openc1_queue**

5876 **Summary**

5877 The **acc_set_openc1_queue** routine returns the OpenCL command queue handle in use for
5878 the current device for the asynchronous activity queue associated with the **async** argument. This
5879 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5880 **Format**

5881 C or C++:

```
5882     void acc_set_openc1_queue ( int async, cl_command_queue cmdqueue  
5883 );
```

5884 **A.3 Recommended Options and Diagnostics**

5885 This section recommends options and diagnostics for implementations. Possible ways to implement
5886 the options include command-line options to a compiler or settings in an IDE.

5887 **A.3.1 C Pointer in Present clause**

5888 This revision of OpenACC clarifies the construct:

```
5889     void test(int n ){  
5890     float* p;  
5891     ...  
5892     #pragma acc data present(p)  
5893     {  
5894         // code here...  
5895     }
```

5896 This example tests whether the pointer **p** itself is present in the current device memory. Implemen-
5897 tations before this revision commonly implemented this by testing whether the pointer target **p[0]**
5898 was present in the current device memory, and this appears in many programs assuming such. Until
5899 such programs are modified to comply with this revision, an option to implement **present (p)** as
5900 **present (p[0])** for C pointers may be helpful to users.

5901 **A.3.2 Nonconforming Applications and Implementations**

5902 Where feasible, implementations should diagnose OpenACC applications that do not conform with
5903 this specification’s syntactic or semantic restrictions. Many but not all of these restrictions appear
5904 in lists entitled “Restrictions.”

5905 While compile-time diagnostics are preferable (e.g., invalid clauses on a directive), some cases of
5906 nonconformity are more feasible to diagnose at run time (e.g., see Section 1.5). Where implemen-
5907 tations are not able to diagnose nonconformity reliably (e.g., an **independent** clause on a loop
5908 with data-dependent loop iterations), they might offer no diagnostics, or they might diagnose only
5909 subcases.

5910 In order to support OpenACC extensions, some implementations intentionally accept nonconform-
5911 ing OpenACC applications without issuing diagnostics by default, and some implementations accept
5912 conforming OpenACC applications but interpret their semantics differently than as detailed in this
5913 specification. To promote program portability across implementations, implementations should pro-
5914 vide an option to disable or report uses of these extensions. Some such extensions and diagnostics
5915 are described in detail in the remainder of this section.

5916 **A.3.3 Automatic Data Attributes**

5917 Some implementations provide autoscoping or other analysis to automatically determine a variable’s
5918 data attributes, including the addition of reduction, private, and firstprivate clauses. To promote
5919 program portability across implementations, it would be helpful to provide an option to disable
5920 the automatic determination of data attributes or report which variables’ data attributes are not as
5921 defined in Section 2.6.

5922 **A.3.4 Routine Directive with a Name**

5923 In C and C++, if a **routine** directive with a name appears immediately before a procedure dec-
5924 laration or definition with that name, it does not necessarily apply to that procedure according to
5925 Section 2.15.1 and C and C++ name resolution. Implementations should issue diagnostics in the
5926 following two cases:

- 5927 1. When no procedure with that name is already in scope, the directive is nonconforming, so
5928 implementations should issue a compile-time error diagnostic regardless of the following
5929 procedure. For example:

```
5930     #pragma acc routine(f) seq // compile-time error  
5931     void f();
```

- 5932 2. When a procedure with that name is in scope and it is not the same procedure as the immedi-
5933 ately following procedure declaration or definition, the resolution of the name can be confus-
5934 ing. Implementations should then issue a compile-time warning diagnostic even though the
5935 application is conforming. For example:

```
5936     void g(); // routine directive applies  
5937     namespace NS {  
5938         #pragma acc routine(g) seq // compile-time warning  
5939         void g(); // routine directive does not apply  
5940     }
```

5941 The diagnostic in this case should suggest the programmer either (1) relocate the **routine**
5942 directive so that it more clearly applies to the procedure that is in scope or (2) remove the
5943 name from the **routine** directive so that it applies to the following procedure.

Index

- 5944 **_OPENACC**, 30, 139
- 5945 **acc-current-device-num-var**, 31
- 5946 **acc-current-device-type-var**, 31
- 5947 **acc-default-async-var**, 31, 95
- 5948 **acc_async_noval**, 95
- 5949 **acc_async_sync**, 95
- 5950 **acc_device_host**, 164
- 5951 **ACC_DEVICE_NUM**, 31, 131
- 5952 **acc_device_nvidia**, 163
- 5953 **acc_device_radeon**, 163
- 5954 **ACC_DEVICE_TYPE**, 31, 131, 163, 164
- 5955 **ACC_PROFLIB**, 131
- 5956 accelerator routine, 88
- 5957 action
 - 5958 allocate memory, 50
 - 5959 attach, 47
 - 5960 attach pointer, 51
 - 5961 detach, 47
 - 5962 detach pointer, 52
- 5963 allocate memory action, 50
- 5964 AMD GPU target, 163
- 5965 **async** clause, 44, 46, 86, 96
- 5966 **async** queue, 11
- 5967 *async-argument*, 96
- 5968 asynchronous execution, 11, 95
- 5969 **atomic** construct, 74
- 5970 attach action, 47
- 5971 **attach** clause, 59
- 5972 attach pointer action, 51
- 5973 attachment counter, 47
- 5974 **auto** clause, 64, 66, 89, 93
 - 5975 portability, 65
- 5976 autoscopying, 167
- 5977 barrier synchronization, 11, 34, 36, 157
- 5978 **bind** clause, 90
- 5979 block construct, 157
- 5980 **cache** directive, 72
- 5981 **capture** clause, 77
- 5982 **collapse** clause, 63
- 5983 common block, 48, 79, 95
- 5984 compiler options, 166
- 5985 compute construct, 157
 - 5986 parent, 33
 - 5987 compute region, 157
- 5988 construct, 157
 - 5989 **atomic**, 74
 - 5990 compute, 157
 - 5991 **data**, 43, 48
 - 5992 **host_data**, 59
 - 5993 **kernels**, 35, 48
 - 5994 **kernels loop**, 72
 - 5995 **parallel**, 33, 48
 - 5996 **parallel loop**, 72
 - 5997 **serial**, 35, 48
 - 5998 **serial loop**, 72
 - 5999 **copy** clause, 41, 54
 - 6000 **copyin** clause, 55
 - 6001 **copyout** clause, 56
 - 6002 **create** clause, 57, 80
 - 6003 CUDA, 12, 158, 163, 164
 - 6004 data attribute
 - 6005 explicitly determined, 40
 - 6006 implicitly determined, 40
 - 6007 predetermined, 40
 - 6008 data clause, 48
 - 6009 visible, 41, 161
 - 6010 **data** construct, 43, 48
 - 6011 data lifetime, 158
 - 6012 data region, 42, 158
 - 6013 implicit, 42
 - 6014 data-independent **loop** construct, 62
 - 6015 **declare** directive, 78
 - 6016 **default** clause, 40, 44
 - 6017 visible, 41, 161
 - 6018 **default (none)** clause, 41
 - 6019 **default(present)**, 41
 - 6020 **delete** clause, 58
 - 6021 detach action, 47
 - 6022 **detach** clause, 59
 - 6023 detach pointer action, 52
 - 6024 **device** clause, 86
 - 6025 **device_resident** clause, 79
 - 6026 **device_type** clause, 31, 48, 163, 164
 - 6027 **deviceptr** clause, 48, 53
 - 6028 diagnostics, 166
 - 6029 direct memory access, 11, 158
 - 6030 DMA, 11, 158

- 6031 **enter data** directive, 45, 48
6032 environment variable
6033 **_OPENACC**, 30
6034 **ACC_DEVICE_NUM**, 31, 131
6035 **ACC_DEVICE_TYPE**, 31, 131, 163, 164
6036 **ACC_PROFLIB**, 131
6037 **exit data** directive, 45, 48
6038 explicitly determined data attribute, 40
6039 exposed variable access, 41, 158
6040 extensions, 167
- 6041 **firstprivate** clause, 38, 41
- 6042 **gang**, 34
6043 **gang** clause, 64, 89
6044 implicit, 64, 93
6045 portability, 65
6046 **gang** parallelism, 10
6047 *gang-arg*, 61
6048 **gang-partitioned** mode, 10
6049 optimizations, 65
6050 **gang-redundant** mode, 10, 34
6051 **GR** mode, 10
- 6052 **host**, 164
6053 **host** clause, 86
6054 **host_data** construct, 59
- 6055 **ICV**, 31
6056 **if** clause
6057 compute construct, 37
6058 **data** construct, 44
6059 **enter data** directive, 46
6060 **exit data** directive, 46
6061 **host_data** construct, 60
6062 **init** directive, 82
6063 **set** directive, 84
6064 **shutdown** directive, 83
6065 **update** directive, 86
6066 **wait** directive, 98
6067 implicit data region, 42
6068 implicit **gang** clause, 64, 93
6069 implicit **routine** directive, 64, 89
6070 implicitly determined data attribute, 40
6071 **independent** clause, 66
6072 **init** directive, 81
6073 internal control variable, 31
- 6074 **kernels** construct, 35, 48
6075 **kernels loop** construct, 72
- 6076 level of parallelism, 10, 159
6077 **link** clause, 48, 81
6078 local device, 11
6079 local memory, 11
6080 local thread, 11
6081 **loop** construct, 61
6082 data-independent, 62
6083 orphaned, 61
6084 sequential, 62
- 6085 **no_create** clause, 57
6086 **nohost** clause, 90
6087 nonconformity, 167
6088 **num_gangs** clause, 37
6089 **num_workers** clause, 38
6090 **nvidia**, 163
6091 **NVIDIA GPU** target, 163
- 6092 **OpenCL**, 12, 159, 163, 165
6093 optimizations
6094 **gang-partitioned** mode, 65
6095 **routine** directive, 94
6096 orphaned **loop** construct, 61
- 6097 **parallel** construct, 33, 48
6098 **parallel loop** construct, 72
6099 parallelism
6100 level, 10, 159
6101 parent compute construct, 33
6102 parent compute scope, 33
6103 parent procedure, 33
6104 pointer in **present** clause, 166
6105 portability
6106 **auto** and **gang** clauses, 65
6107 predetermined data attribute, 40
6108 **present** clause, 41, 48, 53
6109 pointer, 166
6110 **private** clause, 38, 67
6111 procedure
6112 parent, 33
- 6113 **radeon**, 163
6114 **read** clause, 77
6115 **reduction** clause, 39, 68
6116 reference counter, 47
6117 region
6118 compute, 157

- 6119 data, 42, 158
- 6120 implicit data, 42
- 6121 **routine** directive, 88, 167
- 6122 implicit, 64, 89
- 6123 optimizations, 94

- 6124 **self** clause, 86
- 6125 compute construct, 37
- 6126 **update** directive, 86
- 6127 sentinel, 29
- 6128 **seq** clause, 66, 90
- 6129 sequential **loop** construct, 62
- 6130 **serial** construct, 35, 48
- 6131 **serial loop** construct, 72
- 6132 **shutdown** directive, 83
- 6133 *size-expr*, 61
- 6134 structured-block, 160

- 6135 thread, 160
- 6136 **tile** clause, 66

- 6137 **update** clause, 77
- 6138 **update** directive, 85
- 6139 **use_device** clause, 60

- 6140 **vector** clause, 65, 90
- 6141 vector lane, 34
- 6142 vector parallelism, 10
- 6143 vector-partitioned mode, 10
- 6144 vector-single mode, 10
- 6145 **vector_length** clause, 38
- 6146 visible data clause, 41, 161
- 6147 visible **default** clause, 41, 161
- 6148 visible device copy, 161
- 6149 VP mode, 10
- 6150 VS mode, 10

- 6151 **wait** clause, 44, 46, 86, 97
- 6152 **wait** directive, 97
- 6153 worker, 34
- 6154 **worker** clause, 65, 89
- 6155 worker parallelism, 10
- 6156 worker-partitioned mode, 10
- 6157 worker-single mode, 10
- 6158 WP mode, 10
- 6159 WS mode, 10