

1 **The OpenACC[®]**
2 **Application Programming Interface**

3 **Version 3.3**

4 OpenACC-Standard.org

5 November 2022

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 © 2011-2022 OpenACC-Standard.org. All rights reserved.

Contents

11

12	1. Introduction	9
13	1.1. Scope	9
14	1.2. Execution Model	9
15	1.3. Memory Model	11
16	1.4. Language Interoperability	12
17	1.5. Runtime Errors	13
18	1.6. Conventions used in this document	13
19	1.7. Organization of this document	14
20	1.8. References	15
21	1.9. Changes from Version 1.0 to 2.0	16
22	1.10. Corrections in the August 2013 document	17
23	1.11. Changes from Version 2.0 to 2.5	17
24	1.12. Changes from Version 2.5 to 2.6	19
25	1.13. Changes from Version 2.6 to 2.7	19
26	1.14. Changes from Version 2.7 to 3.0	20
27	1.15. Changes from Version 3.0 to 3.1	21
28	1.16. Changes from Version 3.1 to 3.2	23
29	1.17. Changes from Version 3.2 to 3.3	24
30	1.18. Topics Deferred For a Future Revision	25
31	2. Directives	27
32	2.1. Directive Format	27
33	2.2. Conditional Compilation	28
34	2.3. Internal Control Variables	28
35	2.3.1. Modifying and Retrieving ICV Values	29
36	2.4. Device-Specific Clauses	29
37	2.5. Compute Constructs	31
38	2.5.1. Parallel Construct	31
39	2.5.2. Serial Construct	32
40	2.5.3. Kernels Construct	33
41	2.5.4. Compute Construct Restrictions	34
42	2.5.5. Compute Construct Errors	34
43	2.5.6. if clause	34
44	2.5.7. self clause	35
45	2.5.8. async clause	35
46	2.5.9. wait clause	35
47	2.5.10. num_gangs clause	35
48	2.5.11. num_workers clause	35
49	2.5.12. vector_length clause	35
50	2.5.13. private clause	36
51	2.5.14. firstprivate clause	36
52	2.5.15. reduction clause	36
53	2.5.16. default clause	37

54	2.6. Data Environment	37
55	2.6.1. Variables with Predetermined Data Attributes	38
56	2.6.2. Variables with Implicitly Determined Data Attributes	38
57	2.6.3. Data Regions and Data Lifetimes	39
58	2.6.4. Data Structures with Pointers	40
59	2.6.5. Data Construct	40
60	2.6.6. Enter Data and Exit Data Directives	42
61	2.6.7. Reference Counters	44
62	2.6.8. Attachment Counter	44
63	2.7. Data Clauses	45
64	2.7.1. Data Specification in Data Clauses	45
65	2.7.2. Data Clause Actions	47
66	2.7.3. Data Clause Errors	50
67	2.7.4. deviceptr clause	50
68	2.7.5. present clause	50
69	2.7.6. copy clause	51
70	2.7.7. copyin clause	51
71	2.7.8. copyout clause	52
72	2.7.9. create clause	53
73	2.7.10. no_create clause	53
74	2.7.11. delete clause	54
75	2.7.12. attach clause	54
76	2.7.13. detach clause	55
77	2.8. Host_Data Construct	55
78	2.8.1. use_device clause	56
79	2.8.2. if clause	56
80	2.8.3. if_present clause	56
81	2.9. Loop Construct	56
82	2.9.1. collapse clause	58
83	2.9.2. gang clause	59
84	2.9.3. worker clause	60
85	2.9.4. vector clause	60
86	2.9.5. seq clause	61
87	2.9.6. independent clause	61
88	2.9.7. auto clause	61
89	2.9.8. tile clause	61
90	2.9.9. device_type clause	62
91	2.9.10. private clause	62
92	2.9.11. reduction clause	63
93	2.10. Cache Directive	67
94	2.11. Combined Constructs	67
95	2.12. Atomic Construct	69
96	2.13. Declare Directive	73
97	2.13.1. device_resident clause	74
98	2.13.2. create clause	75
99	2.13.3. link clause	75
100	2.14. Executable Directives	76
101	2.14.1. Init Directive	76

102	2.14.2. Shutdown Directive	77
103	2.14.3. Set Directive	78
104	2.14.4. Update Directive	80
105	2.14.5. Wait Directive	82
106	2.14.6. Enter Data Directive	82
107	2.14.7. Exit Data Directive	82
108	2.15. Procedure Calls in Compute Regions	82
109	2.15.1. Routine Directive	82
110	2.15.2. Global Data Access	87
111	2.16. Asynchronous Behavior	87
112	2.16.1. async clause	89
113	2.16.2. wait clause	89
114	2.16.3. Wait Directive	89
115	2.17. Fortran Specific Behavior	90
116	2.17.1. Optional Arguments	90
117	2.17.2. Do Concurrent Construct	91
118	3. Runtime Library	93
119	3.1. Runtime Library Definitions	93
120	3.2. Runtime Library Routines	94
121	3.2.1. acc_get_num_devices	94
122	3.2.2. acc_set_device_type	94
123	3.2.3. acc_get_device_type	95
124	3.2.4. acc_set_device_num	96
125	3.2.5. acc_get_device_num	96
126	3.2.6. acc_get_property	97
127	3.2.7. acc_init	98
128	3.2.8. acc_shutdown	98
129	3.2.9. acc_async_test	99
130	3.2.10. acc_wait	100
131	3.2.11. acc_wait_async	101
132	3.2.12. acc_wait_any	103
133	3.2.13. acc_get_default_async	103
134	3.2.14. acc_set_default_async	104
135	3.2.15. acc_on_device	104
136	3.2.16. acc_malloc	105
137	3.2.17. acc_free	105
138	3.2.18. acc_copyin and acc_create	106
139	3.2.19. acc_copyout and acc_delete	107
140	3.2.20. acc_update_device and acc_update_self	109
141	3.2.21. acc_map_data	111
142	3.2.22. acc_unmap_data	112
143	3.2.23. acc_deviceptr	112
144	3.2.24. acc_hostptr	113
145	3.2.25. acc_is_present	113
146	3.2.26. acc_memcpy_to_device	114
147	3.2.27. acc_memcpy_from_device	115
148	3.2.28. acc_memcpy_device	116

149	3.2.29. acc_attach and acc_detach	118
150	3.2.30. acc_memcpy_d2d	119
151	4. Environment Variables	121
152	4.1. ACC_DEVICE_TYPE	121
153	4.2. ACC_DEVICE_NUM	121
154	4.3. ACC_PROFLIB	121
155	5. Profiling and Error Callback Interface	123
156	5.1. Events	123
157	5.1.1. Runtime Initialization and Shutdown	124
158	5.1.2. Device Initialization and Shutdown	124
159	5.1.3. Enter Data and Exit Data	125
160	5.1.4. Data Allocation	125
161	5.1.5. Data Construct	126
162	5.1.6. Update Directive	126
163	5.1.7. Compute Construct	126
164	5.1.8. Enqueue Kernel Launch	127
165	5.1.9. Enqueue Data Update (Upload and Download)	127
166	5.1.10. Wait	127
167	5.1.11. Error Event	128
168	5.2. Callbacks Signature	128
169	5.2.1. First Argument: General Information	129
170	5.2.2. Second Argument: Event-Specific Information	130
171	5.2.3. Third Argument: API-Specific Information	135
172	5.3. Loading the Library	136
173	5.3.1. Library Registration	137
174	5.3.2. Statically-Linked Library Initialization	138
175	5.3.3. Runtime Dynamic Library Loading	138
176	5.3.4. Preloading with LD_PRELOAD	139
177	5.3.5. Application-Controlled Initialization	140
178	5.4. Registering Event Callbacks	140
179	5.4.1. Event Registration and Unregistration	140
180	5.4.2. Disabling and Enabling Callbacks	142
181	5.5. Advanced Topics	143
182	5.5.1. Dynamic Behavior	143
183	5.5.2. OpenACC Events During Event Processing	144
184	5.5.3. Multiple Host Threads	144
185	6. Glossary	147
186	A. Recommendations for Implementers	153
187	A.1. Target Devices	153
188	A.1.1. NVIDIA GPU Targets	153
189	A.1.2. AMD GPU Targets	153
190	A.1.3. Multicore Host CPU Target	154
191	A.2. API Routines for Target Platforms	154
192	A.2.1. NVIDIA CUDA Platform	154

193	A.2.2. OpenCL Target Platform	155
194	A.3. Recommended Options and Diagnostics	156
195	A.3.1. C Pointer in Present clause	156
196	A.3.2. Nonconforming Applications and Implementations	157
197	A.3.3. Automatic Data Attributes	157
198	A.3.4. Routine Directive with a Name	157

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,

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

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

254 OpenACC exposes these three *levels of parallelism* via *gang*, *worker*, and *vector* parallelism. Gang
255 parallelism is coarse-grain. A number of gangs will be launched on the accelerator. The gangs are
256 organized in a one-, two-, or three-dimensional grid, where dimension one corresponds to the inner
257 level of gang parallelism; the default is to only use dimension one. Worker parallelism is fine-grain.
258 Each gang will have one or more workers. Vector parallelism is for SIMD or vector operations
259 within a worker.

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

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

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

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

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

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

316 1.3 Memory Model

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

325 The concept of discrete host and accelerator memories is very apparent in low-level accelerator
326 programming languages such as CUDA or OpenCL, in which data movement between the memories
327 can dominate user code. In the OpenACC model, data movement between the memories can be
328 implicit and managed by the compiler, based on directives from the programmer. However, the

329 programmer must be aware of the potentially discrete memories for many reasons, including but
330 not limited to:

- 331 • Memory bandwidth between host memory and accelerator memory determines the level of
332 compute intensity required to effectively accelerate a given region of code.
- 333 • The user should be aware that a discrete accelerator memory is usually significantly smaller
334 than the host memory, prohibiting offloading regions of code that operate on very large
335 amounts of data.
- 336 • Data in host memory may only be accessible on the host; data in accelerator memory may
337 only be accessible on that accelerator. Explicitly transferring pointer values between host and
338 accelerator memory is not advised. Dereferencing pointers to host memory on an accelerator
339 or dereferencing pointers to accelerator memory on the host is likely to result in a runtime
340 error or incorrect results on such targets.

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

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

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

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

368 **1.4 Language Interoperability**

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

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

377 1.5 Runtime Errors

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

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

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

395 1.6 Conventions used in this document

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

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

399 **#pragma acc**

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

401 **#pragma acc** *directive-name*

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

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

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

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

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

- 408 • a variable name (a scalar, array, or composite variable name);
- 409 • a subarray specification with subscript ranges;

- 410 • an array element;
- 411 • a member of a composite variable;
- 412 • a common block name between slashes.

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

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

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

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

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

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

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

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

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

433 1.7 Organization of this document

434 The rest of this document is organized as follows:

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

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

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

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

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

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

1.8 References

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

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

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

- Threads
- Thread-local storage
- Parallel memory model
- Atomic

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

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

- Thread related features

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

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

- Extern templates
- copy and rethrow exceptions
- memory model
- atomics
- move semantics
- `std::thread`
- thread-local storage

- ISO/IEC 14882:2014, *Information Technology – Programming Languages – C++*, (C++14).
- ISO/IEC 14882:2017, *Information Technology – Programming Languages – C++*, (C++17).
- ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran – Part 1: Base Language*, (Fortran 2003).
- ISO/IEC 1539-1:2010, *Information Technology – Programming Languages – Fortran – Part 1: Base Language*, (Fortran 2008).

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

- Coarrays
- Simply contiguous arrays rank remapping to rank>1 target

- 480 – Allocatable components of recursive type
- 481 – Polymorphic assignment
- 482 • ISO/IEC 1539-1:2018, *Information Technology – Programming Languages – Fortran – Part*
- 483 *1: Base Language*, (Fortran 2018).
- 484 The use of the following Fortran 2018 features may result in unspecified behavior.
- 485 – Interoperability with C
 - 486 * C functions declared in ISO Fortran binding.h
 - 487 * Assumed rank
- 488 – All additional parallel/coarray features
- 489 • *OpenMP Application Program Interface*, version 5.0, November 2018
- 490 • *NVIDIA CUDATM C Programming Guide*, version 11.1.1, October 2020
- 491 • *The OpenCL Specification*, version 2.2, Khronos OpenCL Working Group, July 2019
- 492 • *INCITS INCLUSIVE TERMINOLOGY GUIDELINES*, version 2021.06.07, InterNational Com-
- 493 mittee for Information Technology Standards, June 2021

494 1.9 Changes from Version 1.0 to 2.0

- 495 • `_OPENACC` value updated to `201306`
- 496 • `default (none)` clause on `parallel` and `kernels` directives
- 497 • the implicit data attribute for scalars in `parallel` constructs has changed
- 498 • the implicit data attribute for scalars in loops with `loop` directives with the independent
- 499 attribute has been clarified
- 500 • `acc_async_sync` and `acc_async_noval` values for the `async` clause
- 501 • Clarified the behavior of the `reduction` clause on a `gang` loop
- 502 • Clarified allowable loop nesting (`gang` may not appear inside `worker`, which may not ap-
- 503 pear within `vector`)
- 504 • `wait` clause on `parallel`, `kernels` and `update` directives
- 505 • `async` clause on the `wait` directive
- 506 • `enter data` and `exit data` directives
- 507 • Fortran *common block* names may now appear in many data clauses
- 508 • `link` clause for the `declare` directive
- 509 • the behavior of the `declare` directive for global data
- 510 • the behavior of a data clause with a C or C++ pointer variable has been clarified
- 511 • predefined data attributes
- 512 • support for multidimensional dynamic C/C++ arrays

- 513 • **tile** and **auto** loop clauses
- 514 • **update self** introduced as a preferred synonym for **update host**
- 515 • **routine** directive and support for separate compilation
- 516 • **device_type** clause and support for multiple device types
- 517 • nested parallelism using **parallel** or **kernels** region containing another **parallel** or **kernels** re-
- 518 **gion**
- 519 • **atomic** constructs
- 520 • new concepts: **gang-redundant**, **gang-partitioned**; **worker-single**, **worker-partitioned**; **vector-**
- 521 **single**, **vector-partitioned**; **thread**
- 522 • new API routines:
 - 523 – **acc_wait**, **acc_wait_all** instead of **acc_async_wait** and **acc_async_wait_all**
 - 524 – **acc_wait_async**
 - 525 – **acc_copyin**, **acc_present_or_copyin**
 - 526 – **acc_create**, **acc_present_or_create**
 - 527 – **acc_copyout**, **acc_delete**
 - 528 – **acc_map_data**, **acc_unmap_data**
 - 529 – **acc_deviceptr**, **acc_hostptr**
 - 530 – **acc_is_present**
 - 531 – **acc_memcpy_to_device**, **acc_memcpy_from_device**
 - 532 – **acc_update_device**, **acc_update_self**
- 533 • defined behavior with multiple host threads, such as with OpenMP
- 534 • recommendations for specific implementations
- 535 • clarified that no arguments are allowed on the **vector** clause in a parallel region

536 1.10 Corrections in the August 2013 document

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

540 1.11 Changes from Version 2.0 to 2.5

- 541 • The **_OPENACC** value was updated to **201510**; see Section 2.2 Conditional Compilation.
- 542 • The **num_gangs**, **num_workers**, and **vector_length** clauses are now allowed on the
- 543 **kernels** construct; see Section 2.5.3 Kernels Construct.
- 544 • Reduction on C++ class members, array elements, and struct elements are explicitly disal-
- 545 lowed; see Section 2.5.15 reduction clause.

- 546 • Reference counting is now used to manage the correspondence and lifetime of device data;
547 see Section 2.6.7 Reference Counters.
- 548 • The behavior of the **exit data** directive has changed to decrement the dynamic reference
549 counter. A new optional **finalize** clause was added to set the dynamic reference counter
550 to zero. See Section 2.6.6 Enter Data and Exit Data Directives.
- 551 • The **copy**, **copyin**, **copyout**, and **create** data clauses were changed to behave like
552 **present_or_copy**, etc. The **present_or_copy**, **pcopy**, **present_or_copyin**,
553 **pcopyin**, **present_or_copyout**, **pcopyout**, **present_or_create**, and **pcreate**
554 data clauses are no longer needed, though will be accepted for compatibility; see Section 2.7
555 Data Clauses.
- 556 • Reductions on orphaned gang loops are explicitly disallowed; see Section 2.9 Loop Construct.
- 557 • The description of the **loop auto** clause has changed; see Section 2.9.7 auto clause.
- 558 • Text was added to the **private** clause on a **loop** construct to clarify that a copy is made
559 for each gang or worker or vector lane, not each thread; see Section 2.9.10 private clause.
- 560 • The description of the **reduction** clause on a **loop** construct was corrected; see Sec-
561 tion 2.9.11 reduction clause.
- 562 • A restriction was added to the **cache** clause that all references to that variable must lie within
563 the region being cached; see Section 2.10 Cache Directive.
- 564 • Text was added to the **private** and **reduction** clauses on a combined construct to clarify
565 that they act like **private** and **reduction** on the **loop**, not **private** and **reduction**
566 on the **parallel** or **reduction** on the **kernels**; see Section 2.11 Combined Constructs.
- 567 • The **declare create** directive with a Fortran **allocatable** has new behavior; see Sec-
568 tion 2.13.2 create clause.
- 569 • New **init**, **shutdown**, **set** directives were added; see Section 2.14.1 Init Directive, 2.14.2
570 Shutdown Directive, and 2.14.3 Set Directive.
- 571 • A new **if_present** clause was added to the **update** directive, which changes the behavior
572 when data is not present from a runtime error to a no-op; see Section 2.14.4 Update Directive.
- 573 • The **routine bind** clause definition changed; see Section 2.15.1 Routine Directive.
- 574 • An **acc routine** without **gang/worker/vector/seq** is now defined as an error; see
575 Section 2.15.1 Routine Directive.
- 576 • A new **default (present)** clause was added for compute constructs; see Section 2.5.16
577 default clause.
- 578 • The Fortran header file **openacc_lib.h** is no longer supported; the Fortran module **openacc**
579 should be used instead; see Section 3.1 Runtime Library Definitions.
- 580 • New API routines were added to get and set the default async queue value; see Section 3.2.13
581 **acc_get_default_async** and 3.2.14 **acc_set_default_async**.
- 582 • The **acc_copyin**, **acc_create**, **acc_copyout**, and **acc_delete** API routines were
583 changed to behave like **acc_present_or_copyin**, etc. The **acc_present_or_** names

584 are no longer needed, though will be supported for compatibility. See Sections 3.2.18 and fol-
585 lowing.

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

592 1.12 Changes from Version 2.5 to 2.6

- 593 • The **_OPENACC** value was updated to **201711**.
- 594 • A new **serial** compute construct was added. See Section 2.5.2 Serial Construct.
- 595 • A new runtime API query routine was added. **acc_get_property** may be called from
596 the host and returns properties about any device. See Section 3.2.6.
- 597 • The text has clarified that if a variable is in a reduction which spans two or more nested loops,
598 each **loop** directive on any of those loops must have a **reduction** clause that contains the
599 variable; see Section 2.9.11 reduction clause.
- 600 • An optional **if** or **if_present** clause is now allowed on the **host_data** construct. See
601 Section 2.8 Host_Data Construct.
- 602 • A new **no_create** data clause is now allowed on compute and **data** constructs. See Sec-
603 tion 2.7.10 `no_create` clause.
- 604 • The behavior of Fortran optional arguments in data clauses and in routine calls has been
605 specified; see Section 2.17.1 Optional Arguments.
- 606 • The descriptions of some of the Fortran versions of the runtime library routines were simpli-
607 fied; see Section 3.2 Runtime Library Routines.
- 608 • To allow for manual deep copy of data structures with pointers, new *attach* and *detach* be-
609 havior was added to the data clauses, new **attach** and **detach** clauses were added, and
610 matching **acc_attach** and **acc_detach** runtime API routines were added; see Sections
611 2.6.4, 2.7.12-2.7.13 and 3.2.29.
- 612 • The Intel Coprocessor Offload Interface target and API routine sections were removed from
613 the Section A Recommendations for Implementers, since Intel no longer produces this prod-
614 uct.

615 1.13 Changes from Version 2.6 to 2.7

- 616 • The **_OPENACC** value was updated to **201811**.
- 617 • The specification allows for hosts that share some memory with the device but not all memory.
618 The wording in the text now discusses whether local thread data is in shared memory (memory
619 shared between the local thread and the device) or discrete memory (local thread memory that
620 is not shared with the device), instead of shared-memory devices and non-shared memory
621 devices. See Sections 1.3 Memory Model and 2.6 Data Environment.

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

646 1.14 Changes from Version 2.7 to 3.0

- 647 • Updated **_OPENACC** value to **201911**.
- 648 • Updated the normative references to the most recent standards for all base languages. See
649 Section 1.8.
- 650 • Changed the text to clarify uses and limitations of the **device_type** clause and added
651 examples; see Section 2.4.
- 652 • Clarified the conflict between the implicit **copy** clause for variables in a **reduction** clause
653 and the implicit **firstprivate** for scalar variables not in a data clause but used in a
654 **parallel** or **serial** construct; see Sections 2.5.1 and 2.5.2.
- 655 • Required at least one data clause on a **data** construct, an **enter data** directive, or an **exit**
656 **data** directive; see Sections 2.6.5 and 2.6.6.
- 657 • Added text describing how a C++ *lambda* invoked in a compute region and the variables
658 captured by the *lambda* are handled; see Section 2.6.2.

- 659 • Added a **zero** modifier to **create** and **copyout** data clauses that zeros the device memory
660 after it is allocated; see Sections 2.7.8 and 2.7.9.
- 661 • Added a new restriction on the **loop** directive allowing only one of the **seq**, **independent**,
662 and **auto** clauses to appear; see Section 2.9.
- 663 • Added a new restriction on the **loop** directive disallowing a **gang**, **worker**, or **vector**
664 clause to appear if a **seq** clause appears; see Section 2.9.
- 665 • Allowed variables to be modified in an atomic region in a loop where the iterations must
666 otherwise be data independent, such as loops with a **loop independent** clause or a **loop**
667 directive in a **parallel** construct; see Sections 2.9.2, 2.9.3, 2.9.4, and 2.9.6.
- 668 • Clarified the behavior of the **auto** and **independent** clauses on the **loop** directive; see
669 Sections 2.9.7 and 2.9.6.
- 670 • Clarified that an orphaned **loop** construct, or a **loop** construct in a **parallel** construct
671 with no **auto** or **seq** clauses is treated as if an **independent** clause appears; see Sec-
672 tion 2.9.6.
- 673 • For a variable in a **reduction** clause, clarified when the update to the original variable is
674 complete, and added examples; see Section 2.9.11.
- 675 • Clarified that a variable in an orphaned **reduction** clause must be private; see Section 2.9.11.
- 676 • Required at least one clause on a **declare** directive; see Section 2.13.
- 677 • Added an **if** clause to **init**, **shutdown**, **set**, and **wait** directives; see Sections 2.14.1,
678 2.14.2, 2.14.3, and 2.16.3.
- 679 • Required at least one clause on a **set** directive; see Section 2.14.3.
- 680 • Added a *devnum* modifier to the **wait** directive and clause to specify a device to which the
681 wait operation applies; see Section 2.16.3.
- 682 • Allowed a **routine** directive to include a C++ lambda name or to appear before a C++
683 lambda definition, and defined implicit **routine** directive behavior when a C++ lambda is
684 called in a compute region or an accelerator routine; see Section 2.15.
- 685 • Added runtime API routine **acc_memcpy_d2d** for copying data directly between two de-
686 vice arrays on the same or different devices; see Section 3.2.30.
- 687 • Defined the values for the **acc_construct_t** and **acc_device_api** enumerations for
688 cross-implementation compatibility; see Sections 5.2.2 and 5.2.3.
- 689 • Changed the return type of **acc_set_cuda_stream** from **int** (values were not specified)
690 to **void**; see Section A.2.1.
- 691 • Edited and expanded Section 1.18 Topics Deferred For a Future Revision.

692 1.15 Changes from Version 3.0 to 3.1

- 693 • Updated **_OPENACC** value to **202011**.
- 694 • Clarified that Fortran blank common blocks are not permitted and that same-named common
695 blocks must have the same size. See Section 1.6.

- 696 • Clarified that a **parallel** construct's block is considered to start in gang-redundant mode
697 even if there's just a single gang. See Section 2.5.1.
- 698 • Added support for the Fortran BLOCK construct. See Sections 2.5.1, 2.5.3, 2.6.1, 2.6.5, 2.8,
699 2.13, and 6.
- 700 • Defined the **serial** construct in terms of the **parallel** construct to improve readability.
701 Instead of defining it in terms of clauses **num_gangs(1) num_workers(1)**
702 **vector_length(1)**, defined the **serial** construct as executing with a single gang of a
703 single worker with a vector length of one. See Section 2.5.2.
- 704 • Consolidated compute construct restrictions into a new section to improve readability. See
705 Section 2.5.4.
- 706 • Clarified that a **default** clause may appear at most once on a compute construct. See
707 Section 2.5.16.
- 708 • Consolidated discussions of implicit data attributes on compute and combined constructs into
709 a separate section. Clarified the conditions under which each data attribute is implied. See
710 Section 2.6.2.
- 711 • Added a restriction that certain loop reduction variables must have explicit data clauses on
712 their parent compute constructs. This change addresses portability across existing OpenACC
713 implementations. See Sections 2.6.2 and A.3.3.
- 714 • Restored the OpenACC 2.5 behavior of the **present**, **copy**, **copyin**, **copyout**, **create**,
715 **no_create**, **delete** data clauses at exit from a region, or on an **exit data** directive, as
716 applicable, and **create** clause at exit from an implicit data region where a **declare** di-
717 rective appears, and **acc_copyout**, **acc_delete** routines, such that no action is taken if
718 the appropriate reference counter is zero, instead of a runtime error being issued if data is not
719 present. See Sections 2.7.5, 2.7.6, 2.7.7, 2.7.8, 2.7.9, 2.7.10, 2.7.11, 2.13.2, and 3.2.19.
- 720 • Clarified restrictions on loop forms that can be associated with **loop** constructs, including
721 the case of C++ range-based **for** loops. See Section 2.9.
- 722 • Specified where **gang** clauses are implied on **loop** constructs. This change standardizes
723 behavior of existing OpenACC implementations. See Section 2.9.2.
- 724 • Corrected C/C++ syntax for **atomic capture** with a structured block. See Section 2.12.
- 725 • Added the behavior of the Fortran *do concurrent* construct. See Section 2.17.2.
- 726 • Changed the Fortran run-time procedures: **acc_device_property** has been renamed to
727 **acc_device_property_kind** and **acc_get_property** uses a different integer kind
728 for the result. See Section 3.2.
- 729 • Added or changed argument names for the Runtime Library routines to be descriptive and
730 consistent. This mostly impacts Fortran programs, which can pass arguments by name. See
731 Section 3.2.
- 732 • Replaced composite variable by aggregate variable in **reduction**, **default**, and **private**
733 clauses and in implicitly determined data attributes; the new wording also includes Fortran
734 character and allocatable/pointer variables. See glossary in Section 6.

1.16 Changes from Version 3.1 to 3.2

735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770

- Updated `_OPENACC` value to `202111`.
- Modified specification to comply with INCITS standard for inclusive terminology.
- The text was changed to state that certain runtime errors, when detected, result in a call to the current runtime error callback routines. See Section 1.5.
- An ambiguity issue with the C/C++ comma operator was resolved. See Section 1.6.
- The terms *true* and *false* were defined and used throughout to shorten the descriptions. See Section 1.6.
- Implicitly determined data attributes on compute constructs were clarified. See Section 2.6.2.
- Clarified that the **default (none)** clause applies to scalar variables. See Section 2.6.2.
- The **async**, **wait**, and **device_type** clauses may be specified on **data** constructs. See Section 2.6.5.
- The behavior of data clauses and data API routines with a null pointer in the clause or as a routine argument is defined. See Sections 2.7.5-2.7.11, 2.8.1, and 3.2.16-3.2.30.
- Precision issues with the loop trip count calculation were clarified. See Section 2.9.
- Text in Section 2.16 was moved and reorganized to improve clarity and reduce redundancy.
- Some runtime routine descriptions were expanded and clarified. See Section 3.2.
- The **acc_init_device** and **acc_shutdown_device** routines were added to initialize and shut down individual devices. See Section 3.2.7 and Section 3.2.8.
- Some runtime routine sections were reorganized and combined into a single section to simplify maintenance and reduce redundant text:
 - The sections for four **acc_async_test** routines were combined into a single section. See Section 3.2.9.
 - The sections for four **acc_wait** routines were combined into a single section. See Section 3.2.10.
 - The sections for four **acc_wait_async** routines were combined into a single section. See Section 3.2.11.
 - The two sections for **acc_copyin** and **acc_create** were combined into a single section. See Section 3.2.18.
 - The two sections for **acc_copyout** and **acc_delete** were combined into a single section. See Section 3.2.19.
 - The two sections for **acc_update_self** and **acc_update_device** were combined into a single section. See Section 3.2.20.
 - The two sections for **acc_attach** and **acc_detach** were combined into a single section. See Section 3.2.29.
- Added runtime API routine **acc_wait_any**. See section 3.2.12.

- 771 • The descriptions of the **async** and **async_queue** fields of **acc_callback_info** were
772 clarified. See Section 5.2.1.

773 1.17 Changes from Version 3.2 to 3.3

- 774 • Updated **_OPENACC** value to **202211**.
- 775 • Allowed three dimensions of gang parallelism:
- 776 – Defined multiple levels of *gang-redundant* and *gang-partitioned* execution modes. See
777 Section 1.2
- 778 – Allowed multiple values in the **num_gangs** clauses on the **parallel** construct. See
779 Section 2.5.10.
- 780 – Allowed a **dim** argument to the **gang** clause on the **loop** construct. See Section 2.9.2.
- 781 – Allowed a **dim** argument to the **gang** clause on the **routine** directive. See Sec-
782 tion 2.15.1.
- 783 – Changed the launch event information to include all three gang dimension sizes. See
784 Section 5.2.2.
- 785 • Clarified user-visible behavior of evaluation of expressions in clause arguments. See Sec-
786 tion 2.1.
- 787 • Added the **force** modifier to the **collapse** clause on loops to enable collapsing non-
788 tightly nested loops. See Section 2.9.1.
- 789 • Generalized implicit **routine** directives for all procedures instead of just C++ lambdas. See
790 Section 2.15.1.
- 791 • Revised Section 2.15.1 for clarity and conciseness, including:
- 792 – Specified predetermined **routine** directives that the implementation may apply.
- 793 – Clarified where **routine** directives must appear relative to definitions or uses of their
794 associated procedures in C and C++. This clarification includes the case of forward
795 references in C++ class member lists.
- 796 – Clarified to which procedure a **routine** directive with a name applies in C and C++.
- 797 – Clarified how a **nohost** clause affects a procedure's use within a compute region.
- 798 • Added a Fortran interface for the following runtime routines (See Chapter 3):
- 799 – **acc_malloc**
- 800 – **acc_free**
- 801 – **acc_map_data**
- 802 – **acc_unmap_data**
- 803 – **acc_deviceptr**
- 804 – **acc_hostptr**
- 805 – The two **acc_memcpy_to_device** routines

- 806 – The two **acc_memcpy_from_device** routines
- 807 – The two **acc_memcpy_device** routines
- 808 – The two **acc_attach** routines
- 809 – The four **acc_detach** routines
- 810 • Added a new error condition for **acc_map_data** when the **bytes** argument is zero. See
- 811 Section 3.2.21.
- 812 • Added recommendations for how a **routine** directive should affect multicore host CPU
- 813 compilation. See Section A.1.3.
- 814 • Recommended additional diagnostics promoting portable and readable OpenACC. See Section A.3.

815 1.18 Topics Deferred For a Future Revision

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

- 820 • Directives to define implicit *deep copy* behavior for pointer-based data structures.
- 821 • Defined behavior when data in data clauses on a directive are aliases of each other.
- 822 • Clarifying when data becomes *present* or *not present* on the device for **enter data** or **exit**
- 823 **data** directives with an **async** clause.
- 824 • Clarifying the behavior of Fortran **pointer** variables in data clauses.
- 825 • Allowing Fortran **pointer** variables to appear in **deviceptr** clauses.
- 826 • Support for attaching C/C++ pointers that point to an address past the end of a memory region.
- 827 • Fully defined interaction with multiple host threads.
- 828 • Optionally removing the synchronization or barrier at the end of vector and worker loops.
- 829 • Allowing an **if** clause after a **device_type** clause.
- 830 • A **shared** clause (or something similar) for the loop directive.
- 831 • Better support for multiple devices from a single thread, whether of the same type or of
- 832 different types.
- 833 • An *auto* construct (by some name), to allow **kernels**-like auto-parallelization behavior
- 834 inside **parallel** constructs or accelerator routines.
- 835 • A **begin declare ... end declare** construct that behaves like putting any global vari-
- 836 ables declared inside the construct in a **declare** clause.
- 837 • Defining the behavior of additional parallelism constructs in the base languages when used
- 838 inside a compute construct or accelerator routine.
- 839 • Optimization directives or clauses, such as an *unroll* directive or clause.
- 840 • Extended reductions.

- 841 • Fortran bindings for all the API routines.
- 842 • A **linear** clause for the **loop** directive.
- 843 • Allowing two or more of **gang**, **worker**, **vector**, or **seq** clause on an **acc routine**
844 directive.
- 845 • A single list of all devices of all types, including the host device.
- 846 • A memory allocation API for specific types of memory, including device memory, host pinned
847 memory, and unified memory.
- 848 • Allowing non-contiguous Fortran array sections as arguments to some Runtime API routines,
849 such as **acc_update_device**.
- 850 • Bindings to other languages.

2. Directives

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

2.1 Directive Format

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

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

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

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

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

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

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

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

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

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

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

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

Only one *directive-name* can appear per directive, except that a combined directive name is considered a single *directive-name*.

889 The order in which clauses appear is not significant unless otherwise specified. A program must not
 890 depend on the order of evaluation of expressions in clause arguments or on any side effects of the
 891 evaluations. (See examples below.) Clauses may be repeated unless otherwise specified.

892 Examples

- 893
- 894
- 895 • In the following example, the order and number of evaluations of `++i` and calls to `foo()`
 896 and `bar()` are unspecified.

```
897     #pragma acc parallel \
898         num_gangs(foo(++i)) \
899         num_workers(bar(++i)) \
900         async(foo(++i))
901     { ... }
```

902 See Section 2.5.1 for the `parallel` construct.

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

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

908 See Section 2.14.4 for the `update` directive.

911 2.2 Conditional Compilation

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

916 2.3 Internal Control Variables

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

921 The ICVs are:

- 922 • `acc-current-device-type-var` - controls which type of device is used.
- 923 • `acc-current-device-num-var` - controls which device of the selected type is used.
- 924 • `acc-default-async-var` - controls which asynchronous queue is used when none appears in an
 925 `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>	acc_set_device_type set device_type init device_type ACC_DEVICE_TYPE	acc_get_device_type
<i>acc-current-device-num-var</i>	acc_set_device_num set device_num init device_num ACC_DEVICE_NUM	acc_get_device_num
<i>acc-default-async-var</i>	acc_set_default_async set default_async	acc_get_default_async

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

2.4 Device-Specific Clauses

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

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

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

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

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

964 Syntax

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

```
966     device_type( * )
967     device_type( device-type-list )
```

968

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

970 Examples

971

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

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

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

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

982

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

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

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

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

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

997

998

999 2.5 Compute Constructs

1000 Compute constructs indicate code that should be executed on the current device. It is implementa-
 1001 tion defined how users specify for which accelerators that code is compiled and whether it is also
 1002 compiled for the host.

1003 2.5.1 Parallel Construct

1004 Summary

1005 This fundamental construct starts parallel execution on the current device.

1006 Syntax

1007 In C and C++, the syntax of the OpenACC **parallel** construct is

```
1008     #pragma acc parallel [clause-list] new-line  
1009         structured block
```

1010

1011 and in Fortran, the syntax is

```
1012     !$acc parallel [clause-list ]  
1013         structured block  
1014     !$acc end parallel
```

1015 OR

```
1016     !$acc parallel [clause-list ]  
1017         block construct  
1018     [!$acc end parallel]
```

1019 where *clause* is one of the following:

```
1020     async [ ( int-expr ) ]  
1021     wait [ ( int-expr-list ) ]  
1022     num_gangs ( int-expr-list )  
1023     num_workers ( int-expr )  
1024     vector_length ( int-expr )  
1025     device_type ( device-type-list )  
1026     if ( condition )  
1027     self [ ( condition ) ]  
1028     reduction ( operator : var-list )  
1029     copy ( var-list )  
1030     copyin ( [ readonly: ] var-list )  
1031     copyout ( [ zero: ] var-list )  
1032     create ( [ zero: ] var-list )  
1033     no_create ( var-list )  
1034     present ( var-list )  
1035     deviceptr ( var-list )  
1036     attach ( var-list )  
1037     private ( var-list )  
1038     firstprivate ( var-list )  
1039     default ( none | present )
```

1040 **Description**

1041 When the program encounters an accelerator **parallel** construct, one or more gangs of workers
 1042 are created to execute the accelerator parallel region. The number of gangs, and the number of
 1043 workers in each gang and the number of vector lanes per worker remain constant for the duration of
 1044 that parallel region. Each gang begins executing the code in the structured block in gang-redundant
 1045 mode even if there is only a single gang. This means that code within the parallel region, but outside
 1046 of a loop construct with gang-level worksharing, will be executed redundantly by all gangs.

1047 One worker in each gang begins executing the code in the structured block of the construct. **Note:**
 1048 Unless there is a **loop** construct within the parallel region, all gangs will execute all the code within
 1049 the region redundantly.

1050 If the **async** clause does not appear, there is an implicit barrier at the end of the accelerator parallel
 1051 region, and the execution of the local thread will not proceed until all gangs have reached the end
 1052 of the parallel region.

1053 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
 1054 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**
 1055 clauses are described in Sections 2.5.13 and Sections 2.5.14. The **device_type** clause is de-
 1056 scribed in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described
 1057 in Section 2.6.2. Restrictions are described in Section 2.5.4.

1058 **2.5.2 Serial Construct**1059 **Summary**

1060 This construct defines a region of the program that is to be executed sequentially on the current
 1061 device. The behavior of the **serial** construct is the same as that of the **parallel** construct
 1062 except that it always executes with a single gang of a single worker with a vector length of one.
 1063 **Note:** The **serial** construct may be used to execute sequential code on the current device,
 1064 which removes the need for data movement when the required data is already present on the device.

1065 **Syntax**

1066 In C and C++, the syntax of the OpenACC **serial** construct is

```
1067     #pragma acc serial [clause-list] new-line  
1068         structured block
```

1069

1070 and in Fortran, the syntax is

```
1071     !$acc serial [ clause-list ]  
1072         structured block  
1073     !$acc end serial
```

1074 OR

```
1075     !$acc serial [ clause-list ]  
1076         block construct  
1077     [!$acc end serial]
```

1078 where *clause* is as for the **parallel** construct except that the **num_gangs**, **num_workers**, and
 1079 **vector_length** clauses are not permitted.

1080 2.5.3 Kernels Construct

1081 Summary

1082 This construct defines a region of the program that is to be compiled into a sequence of kernels for
1083 execution on the current device.

1084 Syntax

1085 In C and C++, the syntax of the OpenACC **kernels** construct is

```
1086     #pragma acc kernels [ clause-list ] new-line
1087         structured block
1088
```

1089 and in Fortran, the syntax is

```
1090     !$acc kernels [ clause-list ]
1091         structured block
1092     !$acc end kernels
```

1093 OR

```
1094     !$acc kernels [ clause-list ]
1095         block construct
1096     [!$acc end kernels]
```

1097 where *clause* is one of the following:

```
1098     async [ ( int-expr ) ]
1099     wait [ ( int-expr-list ) ]
1100     num_gangs ( int-expr )
1101     num_workers ( int-expr )
1102     vector_length ( int-expr )
1103     device_type ( device-type-list )
1104     if ( condition )
1105     self [ ( condition ) ]
1106     copy ( var-list )
1107     copyin ( [ readonly: ] var-list )
1108     copyout ( [ zero: ] var-list )
1109     create ( [ zero: ] var-list )
1110     no_create ( var-list )
1111     present ( var-list )
1112     deviceptr ( var-list )
1113     attach ( var-list )
1114     default ( none | present )
```

1115 Description

1116 The compiler will split the code in the kernels region into a sequence of accelerator kernels. Typi-
1117 cally, each loop nest will be a distinct kernel. When the program encounters a **kernels** construct,
1118 it will launch the sequence of kernels in order on the device. The number and configuration of gangs
1119 of workers and vector length may be different for each kernel.

1120 If the **async** clause does not appear, there is an implicit barrier at the end of the kernels region,
1121 and the local thread execution will not proceed until the entire sequence of kernels has completed
1122 execution.

1123 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
1124 data clauses are described in Section 2.7 Data Clauses. The **device_type** clause is described
1125 in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described in Sec-
1126 tion 2.6.2. Restrictions are described in Section 2.5.4.

1127 2.5.4 Compute Construct Restrictions

1128 The following restrictions apply to all compute constructs:

- 1129 • A program may not branch into or out of a compute construct.
- 1130 • Only the **async**, **wait**, **num_gangs**, **num_workers**, and **vector_length** clauses
1131 may follow a **device_type** clause.
- 1132 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
1133 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1134 • At most one **default** clause may appear, and it must have a value of either **none** or
1135 **present**.
- 1136 • A **reduction** clause may not appear on a **parallel** construct with a **num_gangs** clause
1137 that has more than one argument.

1138 2.5.5 Compute Construct Errors

- 1139 • An **acc_error_wrong_device_type** error is issued if the compute construct was not
1140 compiled for the current device type. This includes the case when the current device is the
1141 host multicore.
- 1142 • An **acc_error_device_type_unavailable** error is issued if no device of the cur-
1143 rent device type is available.
- 1144 • An **acc_error_device_unavailable** error is issued if the current device is not avail-
1145 able.
- 1146 • An **acc_error_device_init** error is issued if the current device cannot be initialized.
- 1147 • An **acc_error_execution** error is issued if the execution of the compute construct on
1148 the current device type fails and the failure can be detected.
- 1149 • Explicit or implicitly determined data attributes can cause an error to be issued; see Sec-
1150 tion 2.7.3.
- 1151 • An **async** or **wait** clause can cause an error to be issued; see Sections 2.16.1 and 2.16.2.

1152 See Section 5.2.2.

1153 2.5.6 if clause

1154 The **if** clause is optional.

1155 When the *condition* in the **if** clause evaluates to *true*., the region will execute on the current device.
1156 When the *condition* in the **if** clause evaluates to *false*, the local thread will execute the region.

1157 **2.5.7 self clause**

1158 The **self** clause is optional.

1159 The **self** clause may have a single *condition-argument*. If the *condition-argument* is not present it
1160 is assumed to evaluate to *true*. When both an **if** clause and a **self** clause appear and the *condition*
1161 in the **if** clause evaluates to *false*, the **self** clause has no effect.

1162 When the *condition* evaluates to *true*, the region will execute on the local device. When the *condition*
1163 in the **self** clause evaluates to *false*, the region will execute on the current device.

1164 **2.5.8 async clause**

1165 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1166 **2.5.9 wait clause**

1167 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1168 **2.5.10 num_gangs clause**

1169 The **num_gangs** clause is allowed on the **parallel** and **kernels** constructs. On a **parallel**
1170 construct, it may have one, two, or three arguments. The values of the integer expressions define
1171 the number of parallel gangs along dimensions one, two, and three that will execute the parallel
1172 region. If it has fewer than three arguments, the missing values are treated as having the value 1.
1173 The total number of gangs must be at least 1 and is the product of the values of the arguments. On a
1174 **kernels** construct, the **num_gangs** clause must have a single argument, the value of which will
1175 define the number of parallel gangs that will execute each kernel created for the kernels region.

1176 If the **num_gangs** clause does not appear, an implementation-defined default will be used which
1177 may depend on the code within the construct. The implementation may use a lower value than
1178 specified based on limitations imposed by the target architecture.

1179 **2.5.11 num_workers clause**

1180 The **num_workers** clause is allowed on the **parallel** and **kernels** constructs. The value
1181 of the integer expression defines the number of workers within each gang that will be active after
1182 a gang transitions from worker-single mode to worker-partitioned mode. If the clause does not
1183 appear, an implementation-defined default will be used; the default value may be 1, and may be
1184 different for each **parallel** construct or for each kernel created for a **kernels** construct. The
1185 implementation may use a different value than specified based on limitations imposed by the target
1186 architecture.

1187 **2.5.12 vector_length clause**

1188 The **vector_length** clause is allowed on the **parallel** and **kernels** constructs. The value
1189 of the integer expression defines the number of vector lanes that will be active after a worker transi-
1190 tions from vector-single mode to vector-partitioned mode. This clause determines the vector length
1191 to use for vector or SIMD operations. If the clause does not appear, an implementation-defined

1192 default will be used. This vector length will be used for loop constructs annotated with the **vector**
 1193 clause, as well as loops automatically vectorized by the compiler. The implementation may use a
 1194 different value than specified based on limitations imposed by the target architecture.

1195 **2.5.13 private clause**

1196 The **private** clause is allowed on the **parallel** and **serial** constructs; it declares that a copy
 1197 of each item on the list will be created for each gang in all dimensions.

1198 **Restrictions**

- 1199 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private**
 1200 clauses.

1201 **2.5.14 firstprivate clause**

1202 The **firstprivate** clause is allowed on the **parallel** and **serial** constructs; it declares that
 1203 a copy of each item on the list will be created for each gang, and that the copy will be initialized with
 1204 the value of that item on the local thread when a **parallel** or **serial** construct is encountered.

1205 **Restrictions**

- 1206 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
 1207 **firstprivate** clauses.

1208 **2.5.15 reduction clause**

1209 The **reduction** clause is allowed on the **parallel** and **serial** constructs. It specifies a
 1210 reduction operator and one or more *vars*. It implies **copy** clauses as described in Section 2.6.2. For
 1211 each reduction *var*, a private copy is created for each parallel gang and initialized for that operator.
 1212 At the end of the region, the values for each gang are combined using the reduction operator, and
 1213 the result combined with the value of the original *var* and stored in the original *var*. If the reduction
 1214 *var* is an array or subarray, the array reduction operation is logically equivalent to applying that
 1215 reduction operation to each element of the array or subarray individually. If the reduction *var*
 1216 is a composite variable, the reduction operation is logically equivalent to applying that reduction
 1217 operation to each member of the composite variable individually. The reduction result is available
 1218 after the region.

1219 The following table lists the operators that are valid and the initialization values; in each case, the
 1220 initialization value will be cast into the data type of the *var*. For **max** and **min** reductions, the
 1221 initialization values are the least representable value and the largest representable value for that data
 1222 type, respectively. At a minimum, the supported data types include Fortran **logical** as well as
 1223 the numerical data types in C (e.g., **_Bool**, **char**, **int**, **float**, **double**, **float _Complex**,
 1224 **double _Complex**), C++ (e.g., **bool**, **char**, **wchar_t**, **int**, **float**, **double**), and Fortran
 1225 (e.g., **integer**, **real**, **double precision**, **complex**). However, for each reduction operator,
 1226 the supported data types include only the types permitted as operands to the corresponding operator
 1227 in the base language where (1) for max and min, the corresponding operator is less-than and (2) for
 1228 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.

1229

1230 Restrictions

- 1231 • A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name,
1232 an array element, or a subarray (refer to Section 2.7.1).
- 1233 • If the reduction *var* is an array element or a subarray, accessing the elements of the array
1234 outside the specified index range results in unspecified behavior.
- 1235 • The reduction *var* may not be a member of a composite variable.
- 1236 • If the reduction *var* is a composite variable, each member of the composite variable must be
1237 a supported datatype for the reduction operation.
- 1238 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
1239 **reduction** clauses.

1240 2.5.16 default clause

1241 The **default** clause is optional. At most one **default** clause may appear. It adjusts what
1242 data attributes are implicitly determined for variables used in the compute construct as described in
1243 Section 2.6.2.

1244 2.6 Data Environment

1245 This section describes the data attributes for variables. The data attributes for a variable may be
1246 *predetermined*, *implicitly determined*, or *explicitly determined*. Variables with predetermined data
1247 attributes may not appear in a data clause that conflicts with that data attribute. Variables with
1248 implicitly determined data attributes may appear in a data clause that overrides the implicit attribute.
1249 Variables with explicitly determined data attributes are those which appear in a data clause on a
1250 **data** construct, a compute construct, or a **declare** directive. See Section A.3.3 for recommended
1251 diagnostics related to data attributes.

1252 OpenACC supports systems with accelerators that have discrete memory from the host, systems
1253 with accelerators that share memory with the host, as well as systems where an accelerator shares
1254 some memory with the host but also has some discrete memory that is not shared with the host.
1255 In the first case, no data is in shared memory. In the second case, all data is in shared memory.
1256 In the third case, some data may be in shared memory and some data may be in discrete memory,

1257 although a single array or aggregate data structure must be allocated completely in shared or discrete
 1258 memory. When a nested OpenACC construct is executed on the device, the default target device for
 1259 that construct is the same device on which the encountering accelerator thread is executing. In that
 1260 case, the target device shares memory with the encountering thread.

1261 2.6.1 Variables with Predetermined Data Attributes

1262 The loop variable in a C **for** statement or Fortran **do** statement that is associated with a loop
 1263 directive is predetermined to be private to each thread that will execute each iteration of the loop.
 1264 Loop variables in Fortran **do** statements within a compute construct are predetermined to be private
 1265 to the thread that executes the loop.

1266 Variables declared in a C block or Fortran block construct that is executed in *vector-partitioned*
 1267 mode are private to the thread associated with each vector lane. Variables declared in a C block
 1268 or Fortran block construct that is executed in *worker-partitioned vector-single* mode are private to
 1269 the worker and shared across the threads associated with the vector lanes of that worker. Variables
 1270 declared in a C block or Fortran block construct that is executed in *worker-single* mode are private
 1271 to the gang and shared across the threads associated with the workers and vector lanes of that gang.

1272 A procedure called from a compute construct will be annotated as **seq**, **vector**, **worker**, or
 1273 **gang**, as described Section 2.15 Procedure Calls in Compute Regions. Variables declared in **seq**
 1274 routine are private to the thread that made the call. Variables declared in **vector** routine are private
 1275 to the worker that made the call and shared across the threads associated with the vector lanes of
 1276 that worker. Variables declared in **worker** or **gang** routine are private to the gang that made the
 1277 call and shared across the threads associated with the workers and vector lanes of that gang.

1278 2.6.2 Variables with Implicitly Determined Data Attributes

1279 When implicitly determining data attributes on a compute construct, the following clauses are visi-
 1280 ble and variable accesses are exposed to the compute construct:

- 1281 • *Visible **default** clause*: The nearest **default** clause appearing on the compute construct
 1282 or a lexically containing **data** construct.
- 1283 • *Visible data clause*: Any data clause on the compute construct, a lexically containing **data**
 1284 construct, or a visible **declare** directive.
- 1285 • *Exposed variable access*: Any access to the data or address of a variable at a point within the
 1286 compute construct where the variable is not private to a scope lexically enclosed within the
 1287 compute construct.

1288 **Note:** In the argument of C's **sizeof** operator, the appearance of a variable is not an exposed
 1289 access because neither its data nor its address is accessed. In the argument of a **reduction**
 1290 clause on an enclosed **loop** construct, the appearance of a variable that is not otherwise
 1291 privatized is an exposed access to the original variable.

1292 On a compute or combined construct, if a variable appears in a **reduction** clause but no other
 1293 data clause, it is treated as if it also appears in a **copy** clause. Otherwise, for any variable, the
 1294 compiler will implicitly determine its data attribute on a compute construct if all of the following
 1295 conditions are met:

- 1296 • There is no **default (none)** clause visible at the compute construct.

- 1297 • An access to the variable is exposed to the compute construct.
- 1298 • The variable does not appear in a data clause visible at the compute construct.

1299 An aggregate variable will be treated as if it appears either:

- 1300 • In a **present** clause if there is a **default (present)** clause visible at the compute con-
- 1301 struct.
- 1302 • In a **copy** clause otherwise.

1303 A scalar variable will be treated as if it appears either:

- 1304 • In a **copy** clause if the compute construct is a **kernels** construct.
- 1305 • In a **firstprivate** clause otherwise.

1306 **Note:** Any **default (none)** clause visible at the compute construct applies to both aggregate
 1307 and scalar variables. However, any **default (present)** clause visible at the compute construct
 1308 applies only to aggregate variables.

1309 Restrictions

- 1310 • If there is a **default (none)** clause visible at a compute construct, for any variable access
 1311 exposed to the compute construct, the compiler requires the variable to appear either in an
 1312 explicit data clause visible at the compute construct or in a **firstprivate**, **private**, or
 1313 **reduction** clause on the compute construct.
- 1314 • If a scalar variable appears in a **reduction** clause on a **loop** construct that has a parent
 1315 **parallel** or **serial** construct, and if the reduction's access to the original variable is
 1316 exposed to the parent compute construct, the variable must appear either in an explicit data
 1317 clause visible at the compute construct or in a **firstprivate**, **private**, or **reduction**
 1318 clause on the compute construct. **Note:** Implementations are encouraged to issue a compile-
 1319 time diagnostic when this restriction is violated to assist users in writing portable OpenACC
 1320 applications.

1321 If a C++ *lambda* is called in a compute region and does not appear in a data clause, then it is
 1322 treated as if it appears in a **copyin** clause on the current construct. A variable captured by a
 1323 *lambda* is processed according to its data types: a pointer type variable is treated as if it appears
 1324 in a **no_create** clause; a reference type variable is treated as if it appears in a **present** clause;
 1325 for a struct or a class type variable, any pointer member is treated as if it appears in a **no_create**
 1326 clause on the current construct. If the variable is defined as global or file or function static, it must
 1327 appear in a **declare** directive.

1328 2.6.3 Data Regions and Data Lifetimes

1329 Data in shared memory is accessible from the current device as well as to the local thread. Such
 1330 data is available to the accelerator for the lifetime of the variable. Data not in shared memory must
 1331 be copied to and from device memory using data constructs, clauses, and API routines. A *data*
 1332 *lifetime* is the duration from when the data is first made available to the accelerator until it becomes
 1333 unavailable. For data in shared memory, the data lifetime begins when the data is allocated and
 1334 ends when it is deallocated; for statically allocated data, the data lifetime begins when the program
 1335 begins and does not end. For data not in shared memory, the data lifetime begins when it is made
 1336 present and ends when it is no longer present.

1337 There are four types of data regions. When the program encounters a **data** construct, it creates a
1338 data region.

1339 When the program encounters a compute construct with explicit data clauses or with implicit data
1340 allocation added by the compiler, it creates a data region that has a duration of the compute construct.

1341 When the program enters a procedure, it creates an implicit data region that has a duration of the
1342 procedure. That is, the implicit data region is created when the procedure is called, and exited when
1343 the program returns from that procedure invocation. There is also an implicit data region associated
1344 with the execution of the program itself. The implicit program data region has a duration of the
1345 execution of the program.

1346 In addition to data regions, a program may create and delete data on the accelerator using **enter**
1347 **data** and **exit data** directives or using runtime API routines. When the program executes
1348 an **enter data** directive, or executes a call to a runtime API **acc_copyin** or **acc_create**
1349 routine, each *var* on the directive or the variable on the runtime API argument list will be made live
1350 on accelerator.

1351 2.6.4 Data Structures with Pointers

1352 This section describes the behavior of data structures that contain pointers. A pointer may be a
1353 C or C++ pointer (e.g., **float***), a Fortran pointer or array pointer (e.g., **real, pointer,**
1354 **dimension(:)**), or a Fortran allocatable (e.g., **real, allocatable, dimension(:)**).

1355 When a data object is copied to device memory, the values are copied exactly. If the data is a data
1356 structure that includes a pointer, or is just a pointer, the pointer value copied to device memory
1357 will be the host pointer value. If the pointer target object is also allocated in or copied to device
1358 memory, the pointer itself needs to be updated with the device address of the target object before
1359 dereferencing the pointer in device memory.

1360 An *attach* action updates the pointer in device memory to point to the device copy of the data
1361 that the host pointer targets; see Section 2.7.2. For Fortran array pointers and allocatable arrays,
1362 this includes copying any associated descriptor (dope vector) to the device copy of the pointer.
1363 When the device pointer target is deallocated, the pointer in device memory should be restored
1364 to the host value, so it can be safely copied back to host memory. A *detach* action updates the
1365 pointer in device memory to have the same value as the corresponding pointer in local memory;
1366 see Section 2.7.2. The *attach* and *detach* actions are performed by the **copy, copyin, copyout,**
1367 **create, attach,** and **detach** data clauses (Sections 2.7.4-2.7.13), and the **acc_attach** and
1368 **acc_detach** runtime API routines (Section 3.2.29). The *attach* and *detach* actions use attachment
1369 counters to determine when the pointer in device memory needs to be updated; see Section 2.6.8.

1370 2.6.5 Data Construct

1371 Summary

1372 The **data** construct defines *vars* to be allocated in the current device memory for the duration of
1373 the region, whether data should be copied from local memory to the current device memory upon
1374 region entry, and copied from device memory to local memory upon region exit.

1375 Syntax

1376 In C and C++, the syntax of the OpenACC **data** construct is


```
1377     #pragma acc data [clause-list] new-line
1378         structured block
```

1379 and in Fortran, the syntax is

```
1380     !$acc data [clause-list]
1381         structured block
1382     !$acc end data
```

1383 or

```
1384     !$acc data [clause-list]
1385         block construct
1386     [!$acc end data]
```

1387 where *clause* is one of the following:

```
1388     if ( condition )
1389     async [ ( int-expr ) ]
1390     wait [ ( wait-argument ) ]
1391     device_type ( device-type-list )
1392     copy ( var-list )
1393     copyin ( [readonly:] var-list )
1394     copyout ( [zero:] var-list )
1395     create ( [zero:] var-list )
1396     no_create ( var-list )
1397     present ( var-list )
1398     deviceptr ( var-list )
1399     attach ( var-list )
1400     default ( none | present )
```

1401 Description

1402 Data will be allocated in the memory of the current device and copied from local memory to device
 1403 memory, or copied back, as required. The data clauses are described in Section 2.7 Data Clauses.
 1404 Structured reference counters are incremented for data when entering a data region, and decre-
 1405 mented when leaving the region, as described in Section 2.6.7 Reference Counters. The **device_type**
 1406 clause is described in Section 2.4 Device-Specific Clauses.

1407 Restrictions

- 1408 • At least one **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**,
 1409 **attach**, or **default** clause must appear on a **data** construct.
- 1410 • Only the **async** and **wait** clauses may follow a **device_type** clause.

1411 if clause

1412 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate
 1413 space in the current device memory and move data from and to the local memory as required. When
 1414 an **if** clause appears, the program will conditionally allocate memory in and move data to and/or
 1415 from device memory. When the *condition* in the **if** clause evaluates to *false*, no device memory
 1416 will be allocated, and no data will be moved. When the *condition* evaluates to *true*, the data will be
 1417 allocated and moved as specified. At most one **if** clause may appear.

1418 **async clause**1419 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1420 **Note:** The **async** clause only affects operations directly associated with this particular **data** construct, such as data transfers. Execution of the associated structured block or block construct remains synchronous to the local thread. Nested OpenACC constructs, directives, and calls to runtime library routines do not inherit the **async** clause from this construct, and the programmer must take care to not accidentally introduce race conditions related to asynchronous data transfers.

1425 **wait clause**1426 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.1427 **default clause**

1428 The **default** clause is optional. At most one **default** clause may appear. It adjusts what data attributes are implicitly determined for variables used in lexically contained compute constructs as described in Section 2.6.2.

1431 **Errors**

- 1432 • See Section 2.7.3 for errors due to data clauses.
- 1433 • See Sections 2.16.1 and 2.16.2 for errors due to **async** or **wait** clauses.

1434 **2.6.6 Enter Data and Exit Data Directives**1435 **Summary**

1436 An **enter data** directive may be used to define *vars* to be allocated in the current device memory for the remaining duration of the program, or until an **exit data** directive that deallocates the data. They also tell whether data should be copied from local memory to device memory at the **enter data** directive, and copied from device memory to local memory at the **exit data** directive. The dynamic range of the program between the **enter data** directive and the matching **exit data** directive is the data lifetime for that data.

1442 **Syntax**1443 In C and C++, the syntax of the OpenACC **enter data** directive is1444 **#pragma acc enter data** *clause-list new-line*

1445 and in Fortran, the syntax is

1446 **!\$acc enter data** *clause-list*1447 where *clause* is one of the following:

```

1448 if ( condition )
1449 async [ ( int-expr ) ]
1450 wait [ ( wait-argument ) ]
1451 copyin ( var-list )
1452 create ( [zero:] var-list )
1453 attach ( var-list )

```

1454 In C and C++, the syntax of the OpenACC **exit data** directive is

1455 **#pragma acc exit data** *clause-list new-line*

1456 and in Fortran, the syntax is

1457 **!\$acc exit data** *clause-list*

1458 where *clause* is one of the following:

```

1459       if ( condition )
1460       async [ ( int-expr ) ]
1461       wait [ ( wait-argument ) ]
1462       copyout ( var-list )
1463       delete ( var-list )
1464       detach ( var-list )
1465       finalize

```

1466 **Description**

1467 At an **enter data** directive, data may be allocated in the current device memory and copied from
 1468 local memory to device memory. This action enters a data lifetime for those *vars*, and will make
 1469 the data available for **present** clauses on constructs within the data lifetime. Dynamic reference
 1470 counters are incremented for this data, as described in Section 2.6.7 Reference Counters. Pointers
 1471 in device memory may be *attached* to point to the corresponding device copy of the host pointer
 1472 target.

1473 At an **exit data** directive, data may be copied from device memory to local memory and deal-
 1474 located from device memory. If no **finalize** clause appears, dynamic reference counters are
 1475 decremented for this data. If a **finalize** clause appears, the dynamic reference counters are set
 1476 to zero for this data. Pointers in device memory may be *detached* so as to have the same value as
 1477 the original host pointer.

1478 The data clauses are described in Section 2.7 Data Clauses. Reference counting behavior is de-
 1479 scribed in Section 2.6.7 Reference Counters.

1480 **Restrictions**

- 1481 • At least one **copyin**, **create**, or **attach** clause must appear on an **enter data** direc-
 1482 tive.
- 1483 • At least one **copyout**, **delete**, or **detach** clause must appear on an **exit data** direc-
 1484 tive.

1485 **if clause**

1486 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or
 1487 deallocate space in the current device memory and move data from and to local memory. When an
 1488 **if** clause appears, the program will conditionally allocate or deallocate device memory and move
 1489 data to and/or from device memory. When the *condition* in the **if** clause evaluates to *false*, no
 1490 device memory will be allocated or deallocated, and no data will be moved. When the *condition*
 1491 evaluates to *true*, the data will be allocated or deallocated and moved as specified.

1492 **async clause**

1493 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1494 **wait clause**

1495 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1496 **finalize clause**

1497 The **finalize** clause is allowed on the **exit data** directive and is optional. When no **finalize**
1498 clause appears, the **exit data** directive will decrement the dynamic reference counters for *vars*
1499 appearing in **copyout** and **delete** clauses, and will decrement the attachment counters for point-
1500 ers appearing in **detach** clauses. If a **finalize** clause appears, the **exit data** directive will
1501 set the dynamic reference counters to zero for *vars* appearing in **copyout** and **delete** clauses,
1502 and will set the attachment counters to zero for pointers appearing in **detach** clauses.

1503 **Errors**

- 1504 • See Section 2.7.3 for errors due to data clauses.
- 1505 • See Sections 2.16.1 and 2.16.2 for errors due to **async** or **wait** clauses.

1506 **2.6.7 Reference Counters**

1507 When device memory is allocated for data not in shared memory due to data clauses or OpenACC
1508 API routine calls, the OpenACC implementation keeps track of that section of device memory and
1509 its relationship to the corresponding data in host memory.

1510 Each section of device memory is associated with two *reference counters* per device, a structured
1511 reference counter and a dynamic reference counter. The structured and dynamic reference counters
1512 are used to determine when to allocate or deallocate data in device memory. The structured reference
1513 counter for a section of memory keeps track of how many nested data regions have been entered for
1514 that data. The initial value of the structured reference counter for static data in device memory (in a
1515 global **declare** directive) is one; for all other data, the initial value is zero. The dynamic reference
1516 counter for a section of memory keeps track of how many dynamic data lifetimes are currently active
1517 in device memory for that section. The initial value of the dynamic reference counter is zero. Data
1518 is considered *present* if the sum of the structured and dynamic reference counters is greater than
1519 zero.

1520 A structured reference counter is incremented when entering each data or compute region that con-
1521 tain an explicit data clause or implicitly-determined data attributes for that section of memory, and
1522 is decremented when exiting that region. A dynamic reference counter is incremented for each
1523 **enter data copyin** or **create** clause, or each **acc_copyin** or **acc_create** API routine
1524 call for that section of memory. The dynamic reference counter is decremented for each **exit**
1525 **data copyout** or **delete** clause when no **finalize** clause appears, or each **acc_copyout**
1526 or **acc_delete** API routine call for that section of memory. The dynamic reference counter will
1527 be set to zero with an **exit data copyout** or **delete** clause when a **finalize** clause ap-
1528 pears, or each **acc_copyout_finalize** or **acc_delete_finalize** API routine call for
1529 the section of memory. The reference counters are modified synchronously with the local thread,
1530 even if the data directives include an **async** clause. When both structured and dynamic reference
1531 counters reach zero, the data lifetime in device memory for that data ends.

1532 **2.6.8 Attachment Counter**

1533 Since multiple pointers can target the same address, each pointer in device memory is associated
1534 with an *attachment counter* per device. The *attachment counter* for a pointer is initialized to zero

1535 when the pointer is allocated in device memory. The *attachment counter* for a pointer is set to one
1536 whenever the pointer is *attached* to new target address, and incremented whenever an *attach* action
1537 for that pointer is performed for the same target address. The *attachment counter* is decremented
1538 whenever a *detach* action occurs for the pointer, and the pointer is *detached* when the *attachment*
1539 *counter* reaches zero. This is described in more detail in Section 2.7.2 Data Clause Actions.

1540 A pointer in device memory can be assigned a device address in two ways. The pointer can be
1541 attached to a device address due to data clauses or API routines, as described in Section 2.7.2
1542 Data Clause Actions, or the pointer can be assigned in a compute region executed on that device.
1543 Unspecified behavior may result if both ways are used for the same pointer.

1544 Pointer members of structs, classes, or derived types in device or host memory can be overwritten
1545 due to update directives or API routines. It is the user's responsibility to ensure that the pointers
1546 have the appropriate values before or after the data movement in either direction. The behavior of
1547 the program is undefined if any of the pointer members are attached when an update of a composite
1548 variable is performed.

1549 2.7 Data Clauses

1550 Data clauses may appear on the **parallel** construct, **serial** construct, **kernels** construct,
1551 **data** construct, the **enter data** and **exit data** directives, and **declare** directives. In the
1552 descriptions, the *region* is a compute region with a clause appearing on a **parallel**, **serial**, or
1553 **kernels** construct, a data region with a clause on a **data** construct, or an implicit data region
1554 with a clause on a **declare** directive. If the **declare** directive appears in a global context,
1555 the corresponding implicit data region has a duration of the program. The list argument to each
1556 data clause is a comma-separated collection of *vars*. On a **declare** directive, the list argument
1557 of a **copyin**, **create**, **device_resident**, or **link** clause may include a Fortran *common*
1558 *block* name enclosed within slashes. On any directive, for any clause except **deviceptr** and
1559 **present**, the list argument may include a Fortran *common block* name enclosed within slashes
1560 if that *common block* name also appears in a **declare** directive **link** clause. In all cases, the
1561 compiler will allocate and manage a copy of the *var* in the memory of the current device, creating a
1562 visible device copy of that *var*, for data not in shared memory.

1563 OpenACC supports accelerators with discrete memories from the local thread. However, if the
1564 accelerator can access the local memory directly, the implementation may avoid the memory allo-
1565 cation and data movement and simply share the data in local memory. Therefore, a program that
1566 uses and assigns data on the host and uses and assigns the same data on the accelerator within a
1567 data region without update directives to manage the coherence of the two copies may get different
1568 answers on different accelerators or implementations.

1569 Restrictions

- 1570 • Data clauses may not follow a **device_type** clause.
- 1571 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in data
1572 clauses.

1573 2.7.1 Data Specification in Data Clauses

1574 In C and C++, a subarray is an array name followed by an extended array range specification in
1575 brackets, with start and length, such as

1576 **AA[2:n]**

1577 If the lower bound is missing, zero is used. If the length is missing and the array has known size, the
 1578 size of the array is used; otherwise the length is required. The subarray **AA[2:n]** means elements
 1579 **AA[2], AA[3], ..., AA[2+n-1]**.

1580 In C and C++, a two dimensional array may be declared in at least four ways:

- 1581 • Statically-sized array: **float AA[100][200];**
- 1582 • Pointer to statically sized rows: **typedef float row[200]; row* BB;**
- 1583 • Statically-sized array of pointers: **float* CC[200];**
- 1584 • Pointer to pointers: **float** DD;**

1585 Each dimension may be statically sized, or a pointer to dynamically allocated memory. Each of
 1586 these may be included in a data clause using subarray notation to specify a rectangular array:

- 1587 • **AA[2:n][0:200]**
- 1588 • **BB[2:n][0:m]**
- 1589 • **CC[2:n][0:m]**
- 1590 • **DD[2:n][0:m]**

1591 Multidimensional rectangular subarrays in C and C++ may be specified for any array with any com-
 1592 bination of statically-sized or dynamically-allocated dimensions. For statically sized dimensions, all
 1593 dimensions except the first must specify the whole extent to preserve the contiguous data restriction,
 1594 discussed below. For dynamically allocated dimensions, the implementation will allocate pointers
 1595 in device memory corresponding to the pointers in local memory and will fill in those pointers as
 1596 appropriate.

1597 In Fortran, a subarray is an array name followed by a comma-separated list of range specifications
 1598 in parentheses, with lower and upper bound subscripts, such as

1599 **arr(1:high, low:100)**

1600 If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if
 1601 known, are used. All dimensions except the last must specify the whole extent, to preserve the
 1602 contiguous data restriction, discussed below.

1603 Restrictions

- 1604 • In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be
 1605 specified.
- 1606 • In C and C++, the length for dynamically allocated dimensions of an array must be explicitly
 1607 specified.
- 1608 • In C and C++, modifying pointers in pointer arrays during the data lifetime, either on the host
 1609 or on the device, may result in undefined behavior.
- 1610 • If a subarray appears in a data clause, the implementation may choose to allocate memory for
 1611 only that subarray on the accelerator.
- 1612 • In Fortran, array pointers may appear, but pointer association is not preserved in device mem-
 1613 ory.

- 1614 • Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous
1615 section of memory, except for dynamic multidimensional C arrays.
- 1616 • In C and C++, if a variable or array of composite type appears, all the data members of the
1617 struct or class are allocated and copied, as appropriate. If a composite member is a pointer
1618 type, the data addressed by that pointer are not implicitly copied.
- 1619 • In Fortran, if a variable or array of composite type appears, all the members of that derived
1620 type are allocated and copied, as appropriate. If any member has the **allocatable** or
1621 **pointer** attribute, the data accessed through that member are not copied.
- 1622 • If an expression is used in a subscript or subarray expression in a clause on a **data** construct,
1623 the same value is used when copying data at the end of the data region, even if the values of
1624 variables in the expression change during the data region.

1625 2.7.2 Data Clause Actions

1626 Most of the data clauses perform one or more the following actions. The actions test or modify one
1627 or both of the structured and dynamic reference counters, depending on the directive on which the
1628 data clause appears.

1629 Present Increment Action

1630 A *present increment* action is one of the actions that may be performed for a **present** (Sec-
1631 tion 2.7.5), **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create**
1632 (Section 2.7.9), or **no_create** (Section 2.7.10) clause, or for a call to an **acc_copyin** or
1633 **acc_create** (Section 3.2.18) API routine. See those sections for details.

1634 A *present increment* action for a *var* occurs only when *var* is already present in device memory.

1635 A *present increment* action for a *var* increments the structured or dynamic reference counter for *var*.

1636 Present Decrement Action

1637 A *present decrement* action is one of the actions that may be performed for a **present** (Section
1638 2.7.5), **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Sec-
1639 tion 2.7.9), **no_create** (Section 2.7.10), or **delete** (Section 2.7.11) clause, or for a call to an
1640 **acc_copyout** or **acc_delete** (Section 3.2.19) API routine. See those sections for details.

1641 A *present decrement* action for a *var* occurs only when *var* is already present in device memory.

1642 A *present decrement* action for a *var* decrements the structured or dynamic reference counter for
1643 *var*, if its value is greater than zero. If the device memory associated with *var* was mapped to
1644 the device using **acc_map_data**, the dynamic reference count may not be decremented to zero,
1645 except by a call to **acc_unmap_data**. If the reference counter is already zero, its value is left
1646 unchanged.

1647 Create Action

1648 A *create* action is one of the actions that may be performed for a **copyout** (Section 2.7.8) or
1649 **create** (Section 2.7.9) clause, or for a call to an **acc_create** API routine (Section 3.2.18). See
1650 those sections for details.

1651 A *create* action for a *var* occurs only when *var* is not already present in device memory.

1652 A *create* action for a *var*:

- 1653 • allocates device memory for *var*; and
- 1654 • sets the structured or dynamic reference counter to one.

1655 **Copyin Action**

1656 A *copyin* action is one of the actions that may be performed for a **copy** (Section 2.7.6) or **copyin**
1657 (Section 2.7.7) clause, or for a call to an **acc_copyin** API routine (Section 3.2.18). See those
1658 sections for details.

1659 A *copyin* action for a *var* occurs only when *var* is not already present in device memory.

1660 A *copyin* action for a *var*:

- 1661 • allocates device memory for *var*;
- 1662 • initiates a copy of the data for *var* from the local thread memory to the corresponding device
1663 memory; and
- 1664 • sets the structured or dynamic reference counter to one.

1665 The data copy may complete asynchronously, depending on other clauses on the directive.

1666 **Copyout Action**

1667 A *copyout* action is one of the actions that may be performed for a **copy** (Section 2.7.6) or
1668 **copyout** (Section 2.7.8) clause, or for a call to an **acc_copyout** API routine (Section 3.2.19).
1669 See those sections for details.

1670 A *copyout* action for a *var* occurs only when *var* is present in device memory.

1671 A *copyout* action for a *var*:

- 1672 • performs an *immediate detach* action for any pointer in *var*;
- 1673 • initiates a copy of the data for *var* from device memory to the corresponding local thread
1674 memory; and
- 1675 • deallocates device memory for *var*.

1676 The data copy may complete asynchronously, depending on other clauses on the directive, in which
1677 case the memory is deallocated when the data copy is complete.

1678 **Delete Action**

1679 A *delete* action is one of the actions that may be performed for a **present** (Section 2.7.5),
1680 **copyin** (Section 2.7.7), **create** (Section 2.7.9), **no_create** (Section 2.7.10), or **delete** (Sec-
1681 tion 2.7.11) clause, or for a call to an **acc_delete** API routine (Section 3.2.19). See those sections
1682 for details.

1683 A *delete* action for a *var* occurs only when *var* is present in device memory.

1684 A *delete* action for *var*:

- 1685 • performs an *immediate detach* action for any pointer in *var*; and
- 1686 • deallocates device memory for *var*.

1687 **Attach Action**

1688 An *attach* action is one of the actions that may be performed for a **present** (Section 2.7.5),
1689 **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Section 2.7.9),
1690 **no_create** (Section 2.7.10), or **attach** (Section 2.7.11) clause, or for a call to an **acc_attach**
1691 API routine (Section 3.2.29). See those sections for details.

1692 An *attach* action for a *var* occurs only when *var* is a pointer reference.

1693 If the pointer *var* is in shared memory or is not present in the current device memory, or if the
1694 address to which *var* points is not present in the current device memory, no action is taken. If the
1695 *attachment counter* for *var* is nonzero and the pointer in device memory already points to the device
1696 copy of the data in *var*, the *attachment counter* for the pointer *var* is incremented. Otherwise, the
1697 pointer in device memory is *attached* to the device copy of the data by initiating an update for the
1698 pointer in device memory to point to the device copy of the data and setting the *attachment counter*
1699 for the pointer *var* to one. If the pointer is a null pointer, the pointer in device memory is updated to
1700 have the same value. The update may complete asynchronously, depending on other clauses on the
1701 directive. The implementation schedules pointer updates after any data copies due to *copyin* actions
1702 that are performed for the same directive.

1703 **Detach Action**

1704 A *detach* action is one of the actions that may be performed for a **present** (Section 2.7.5),
1705 **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Section 2.7.9),
1706 **no_create** (Section 2.7.10), **delete** (Section 2.7.11), or **detach** (Section 2.7.11) clause, or
1707 for a call to an **acc_detach** API routine (Section 3.2.29). See those sections for details.

1708 A *detach* action for a *var* occurs only when *var* is a pointer reference.

1709 If the pointer *var* is in shared memory or is not present in the current device memory, or if the
1710 *attachment counter* for *var* for the pointer is zero, no action is taken. Otherwise, the *attachment*
1711 *counter* for the pointer *var* is decremented. If the *attachment counter* is decreased to zero, the
1712 pointer is *detached* by initiating an update for the pointer *var* in device memory to have the same
1713 value as the corresponding pointer in local memory. The update may complete asynchronously,
1714 depending on other clauses on the directive. The implementation schedules pointer updates before
1715 any data copies due to *copyout* actions that are performed for the same directive.

1716 **Immediate Detach Action**

1717 An *immediate detach* action is one of the actions that may be performed for a **detach** (Section
1718 2.7.11) clause, or for a call to an **acc_detach_finalize** API routine (Section 3.2.29). See
1719 those sections for details.

1720 An *immediate detach* action for a *var* occurs only when *var* is a pointer reference and is present in
1721 device memory.

1722 If the *attachment counter* for the pointer is zero, the *immediate detach* action has no effect. Other-
1723 wise, the *attachment counter* for the pointer is set to zero and the pointer is *detached* by initiating an
1724 update for the pointer in device memory to have the same value as the corresponding pointer in local

1725 memory. The update may complete asynchronously, depending on other clauses on the directive.
1726 The implementation schedules pointer updates before any data copies due to *copyout* actions that
1727 are performed for the same directive.

1728 2.7.3 Data Clause Errors

1729 An error is issued for a *var* that appears in a **copy**, **copyin**, **copyout**, **create**, and **delete**
1730 clause as follows:

- 1731 • An **acc_error_partly_present** error is issued if part of *var* is present in the current
1732 device memory but all of *var* is not.
- 1733 • An **acc_error_invalid_data_section** error is issued if *var* is a Fortran subarray
1734 with a stride that is not one.
- 1735 • An **acc_error_out_of_memory** error is issued if the accelerator device does not have
1736 enough memory for *var*.

1737 An error is issued for a *var* that appears in a **present** clause as follows:

- 1738 • An **acc_error_not_present** error is issued if *var* is not present in the current device
1739 memory at entry to a data or compute construct.
- 1740 • An **acc_error_partly_present** error is issued if part of *var* is present in the current
1741 device memory but all of *var* is not.

1742 See Section 5.2.2.

1743 2.7.4 deviceptr clause

1744 The **deviceptr** clause may appear on structured **data** and compute constructs and **declare**
1745 directives.

1746 The **deviceptr** clause is used to declare that the pointers in *var-list* are device pointers, so the
1747 data need not be allocated or moved between the host and device for this pointer.

1748 In C and C++, the *vars* in *var-list* must be pointer variables.

1749 In Fortran, the *vars* in *var-list* must be dummy arguments (arrays or scalars), and may not have the
1750 Fortran **pointer**, **allocatable**, or **value** attributes.

1751 For data in shared memory, host pointers are the same as device pointers, so this clause has no
1752 effect.

1753 2.7.5 present clause

1754 The **present** clause may appear on structured **data** and compute constructs and **declare** di-
1755 rectives. The **present** clause specifies that *vars* in *var-list* are in shared memory or are already
1756 present in the current device memory due to data regions or data lifetimes that contain the construct
1757 on which the **present** clause appears.

1758 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1759 the **present** clause behaves as follows:

- 1760 • At entry to the region:

1761 – An *attach* action is performed if *var* is a pointer reference, and a *present increment*
 1762 action with the structured reference counter is performed if *var* is not a null pointer.

1763 • At exit from the region:

1764 – If the structured reference counter for *var* is zero, no action is taken.

1765 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
 1766 action with the structured reference counter is performed if *var* is not a null pointer. If
 1767 both structured and dynamic reference counters are zero, a *delete* action is performed.

1768 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1769 2.7.6 copy clause

1770 The **copy** clause may appear on structured **data** and compute constructs and on **declare** direc-
 1771 tives.

1772 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1773 the **copy** clause behaves as follows:

1774 • At entry to the region:

1775 – If *var* is present and is not a null pointer, a *present increment* action with the structured
 1776 reference counter is performed.

1777 – If *var* is not present, a *copyin* action with the structured reference counter is performed.

1778 – If *var* is a pointer reference, an *attach* action is performed.

1779 • At exit from the region:

1780 – If the structured reference counter for *var* is zero, no action is taken.

1781 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
 1782 action with the structured reference counter is performed if *var* is not a null pointer. If
 1783 both structured and dynamic reference counters are zero, a *copyout* action is performed.

1784 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1785 For compatibility with OpenACC 2.0, **present_or_copy** and **pcopy** are alternate names for
 1786 **copy**.

1787 2.7.7 copyin clause

1788 The **copyin** clause may appear on structured **data** and compute constructs, on **declare** direc-
 1789 tives, and on **enter data** directives.

1790 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1791 the **copyin** clause behaves as follows:

1792 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
 1793 the dynamic reference counter is used.

1794 – If *var* is present and is not a null pointer, a *present increment* action with the appropriate
 1795 reference counter is performed.

- 1796 – If *var* is not present, a *copyin* action with the appropriate reference counter is performed.
- 1797 – If *var* is a pointer reference, an *attach* action is performed.

- 1798 • At exit from the region:

- 1799 – If the structured reference counter for *var* is zero, no action is taken.
- 1800 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
- 1801 action with the structured reference counter is performed if *var* is not a null pointer. If
- 1802 both structured and dynamic reference counters are zero, a *delete* action is performed.

1803 If the optional **readonly** modifier appears, then the implementation may assume that the data
1804 referenced by *var-list* is never written to within the applicable region.

1805 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1806 For compatibility with OpenACC 2.0, **present_or_copyin** and **pcopyin** are alternate names
1807 for **copyin**.

1808 An **enter data** directive with a **copyin** clause is functionally equivalent to a call to the **acc_copyin**
1809 API routine, as described in Section 3.2.18.

1810 2.7.8 copyout clause

1811 The **copyout** clause may appear on structured **data** and compute constructs, on **declare** di-
1812 rectives, and on **exit data** directives. The clause may optionally have a **zero** modifier if the
1813 **copyout** clause appears on a structured **data** or compute construct.

1814 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1815 the **copyout** clause behaves as follows:

- 1816 • At entry to a region:

- 1817 – If *var* is present and is not a null pointer, a *present increment* action with the structured
- 1818 reference counter is performed.
- 1819 – If *var* is not present, a *create* action with the structured reference counter is performed.
- 1820 If a **zero** modifier appears, the memory is zeroed after the *create* action.
- 1821 – If *var* is a pointer reference, an *attach* action is performed.

- 1822 • At exit from a region, the structured reference counter is used. On an **exit data** directive,
1823 the dynamic reference counter is used.

- 1824 – If the appropriate reference counter for *var* is zero, no action is taken.
- 1825 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and the reference
- 1826 counter is updated if **var** is not a null pointer:
 - 1827 * On an **exit data** directive with a **finalize** clause, the dynamic reference
 - 1828 counter is set to zero.
 - 1829 * Otherwise, a *present decrement* action with the appropriate reference counter is
 - 1830 performed.

1831 If both structured and dynamic reference counters are zero, a *copyout* action is per-
1832 formed.

1833 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1834 For compatibility with OpenACC 2.0, **present_or_copyout** and **pcopyout** are alternate
1835 names for **copyout**.

1836 An **exit data** directive with a **copyout** clause and with or without a **finalize** clause is func-
1837 tionally equivalent to a call to the **acc_copyout_finalize** or **acc_copyout** API routine,
1838 respectively, as described in Section 3.2.19.

1839 2.7.9 create clause

1840 The **create** clause may appear on structured **data** and compute constructs, on **declare** direc-
1841 tives, and on **enter data** directives. The clause may optionally have a **zero** modifier.

1842 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1843 the **create** clause behaves as follows:

- 1844 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
1845 the dynamic reference counter is used.
 - 1846 – If *var* is present and is not a null pointer, a *present increment* action with the appropriate
1847 reference counter is performed.
 - 1848 – If *var* is not present and is not a null pointer, a *create* action with the appropriate refer-
1849 ence counter is performed. If a **zero** modifier appears, the memory is zeroed after the
1850 *create* action.
 - 1851 – If *var* is a pointer reference, an *attach* action is performed.
- 1852 • At exit from the region:
 - 1853 – If the structured reference counter for *var* is zero, no action is taken.
 - 1854 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
1855 action with the structured reference counter is performed if *var* is not a null pointer. If
1856 both structured and dynamic reference counters are zero, a *delete* action is performed.

1857 The errors in Section 2.7.3 Data Clause Errors may be issued for this clause.

1858 For compatibility with OpenACC 2.0, **present_or_create** and **pcreate** are alternate names
1859 for **create**.

1860 An **enter data** directive with a **create** clause is functionally equivalent to a call to the **acc_create**
1861 API routine, as described in Section 3.2.18, except the directive may perform an *attach* action for a
1862 pointer reference.

1863 2.7.10 no_create clause

1864 The **no_create** clause may appear on structured **data** and compute constructs.

1865 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1866 the **no_create** clause behaves as follows:

- 1867 • At entry to the region:
 - 1868 – If *var* is present and is not a null pointer, a *present increment* action with the structured
 - 1869 reference counter is performed. If *var* is present and is a pointer reference, an *attach*
 - 1870 action is performed.
 - 1871 – If *var* is not present, no action is performed, and any device code in this construct will
 - 1872 use the local memory address for *var*.
- 1873 • At exit from the region:
 - 1874 – If the structured reference counter for *var* is zero, no action is taken.
 - 1875 – Otherwise, a *detach* action is performed if *var* is a pointer reference, and a *present decrement*
 - 1876 action with the structured reference counter is performed if *var* is not a null pointer. If
 - 1877 both structured and dynamic reference counters are zero, a *delete* action is performed.

1878 2.7.11 delete clause

1879 The **delete** clause may appear on **exit data** directives.

1880 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1881 the **delete** clause behaves as follows:

- 1882 • If the dynamic reference counter for *var* is zero, no action is taken.
 - 1883 • Otherwise, a *detach* action is performed if *var* is a pointer reference, and the dynamic refer-
1884 ence counter is updated if *var* is not a null pointer:
 - 1885 – On an **exit data** directive with a **finalize** clause, the dynamic reference counter
1886 is set to zero.
 - 1887 – Otherwise, a *present decrement* action with the dynamic reference counter is performed.
- 1888 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic
1889 reference counters are zero, a *delete* action is performed.

1890 An **exit data** directive with a **delete** clause and with or without a **finalize** clause is func-
1891 tionally equivalent to a call to the **acc_delete_finalize** or **acc_delete** API routine, re-
1892 spectively, as described in Section 3.2.19.

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

1894 2.7.12 attach clause

1895 The **attach** clause may appear on structured **data** and compute constructs and on **enter data**
1896 directives. Each *var* argument to an **attach** clause must be a C or C++ pointer or a Fortran variable
1897 or array with the **pointer** or **allocatable** attribute.

1898 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
1899 the **attach** clause behaves as follows:

- 1900 • At entry to a region or at an **enter data** directive, an *attach* action is performed.
- 1901 • At exit from the region, a *detach* action is performed.

1902 2.7.13 detach clause

1903 The **detach** clause may appear on **exit data** directives. Each *var* argument to a **detach** clause
 1904 must be a C or C++ pointer or a Fortran variable or array with the **pointer** or **allocatable**
 1905 attribute.

1906 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,
 1907 the **detach** clause behaves as follows:

- 1908 • If there is a **finalize** clause on the **exit data** directive, an *immediate detach* action is
 1909 performed.
- 1910 • Otherwise, a *detach* action is performed.

1911 2.8 Host_Data Construct

1912 Summary

1913 The **host_data** construct makes the address of data in device memory available on the host.

1914 Syntax

1915 In C and C++, the syntax of the OpenACC **host_data** construct is

```
1916 #pragma acc host_data clause-list new-line
1917     structured block
```

1918 and in Fortran, the syntax is

```
1919 !$acc host_data clause-list
1920     structured block
1921 !$acc end host_data
```

1922 or

```
1923 !$acc host_data clause-list
1924     block construct
1925 [!$acc end host_data]
```

1926 where *clause* is one of the following:

```
1927 use_device ( var-list )
1928 if ( condition )
1929 if_present
```

1930 Description

1931 This construct is used to make the address of data in device memory available in host code.

1932 Restrictions

- 1933 • A *var* in a **use_device** clause must be the name of a variable or array.
- 1934 • At least one **use_device** clause must appear.
- 1935 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
 1936 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1937 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
 1938 **use_device** clauses.

1939 2.8.1 use_device clause

1940 The **use_device** clause tells the compiler to use the current device address of any *var* in *var-list*
 1941 in code within the construct. In particular, this may be used to pass the device address of *var* to
 1942 optimized procedures written in a lower-level API. If *var* is a null pointer, the same value is used
 1943 for the device address. Otherwise, when there is no **if_present** clause, and either there is no
 1944 **if** clause or the condition in the **if** clause evaluates to *true*, the *var* in *var-list* must be present in
 1945 the accelerator memory due to data regions or data lifetimes that contain this construct. For data in
 1946 shared memory, the device address is the same as the host address.

1947 2.8.2 if clause

1948 The **if** clause is optional. When an **if** clause appears and the condition evaluates to *false*, the
 1949 compiler will not replace the addresses of any *var* in code within the construct. When there is no **if**
 1950 clause, or when an **if** clause appears and the condition evaluates to *true*, the compiler will replace
 1951 the addresses as described in the previous subsection.

1952 2.8.3 if_present clause

1953 When an **if_present** clause appears on the directive, the compiler will only replace the address
 1954 of any *var* which appears in *var-list* that is present in the current device memory.

1955 2.9 Loop Construct

1956 Summary

1957 The OpenACC **loop** construct applies to a loop which must immediately follow this directive. The
 1958 **loop** construct can describe what type of parallelism to use to execute the loop and declare private
 1959 *vars* and reduction operations.

1960 Syntax

1961 In C and C++, the syntax of the **loop** construct is

```
1962     #pragma acc loop [clause-list] new-line
1963         for loop
```

1964 In Fortran, the syntax of the **loop** construct is

```
1965     !$acc loop [clause-list]
1966         do loop
```

1967 where *clause* is one of the following:

```
1968     collapse ( [force:] n )
1969     gang [ ( gang-arg-list ) ]
1970     worker [ ( [num:]int-expr ) ]
1971     vector [ ( [length:]int-expr ) ]
1972     seq
1973     independent
1974     auto
1975     tile ( size-expr-list )
1976     device_type ( device-type-list )
```


1977 **private** (*var-list*)
 1978 **reduction** (*operator* : *var-list*)

1979 where *gang-arg* is one of:

1980 [**num** :] *int-expr*
 1981 **dim** : *int-expr*
 1982 **static** : *size-expr*

1983 and *gang-arg-list* may have at most one **num**, one **dim**, and one **static** argument, and where
 1984 *size-expr* is one of:

1985 *
 1986 *int-expr*

1988 Some clauses are only valid in the context of a **kernels** construct; see the descriptions below.

1989 An *orphaned loop* construct is a **loop** construct that is not lexically enclosed within a compute
 1990 construct. The parent compute construct of a **loop** construct is the nearest compute construct that
 1991 lexically contains the **loop** construct.

1992 A **loop** construct is *data-independent* if it has an **independent** clause that is determined explic-
 1993 itly, implicitly, or from an **auto** clause. A **loop** construct is *sequential* if it has a **seq** clause that
 1994 is determined explicitly or from an **auto** clause.

1995 When *do-loop* is a **do concurrent**, the OpenACC **loop** construct applies to the loop for each
 1996 index in the *concurrent-header*. The **loop** construct can describe what type of parallelism to use
 1997 to execute all the loops, and declares all indices appearing in the *concurrent-header* to be implicitly
 1998 private. If the **loop** construct that is associated with **do concurrent** is combined with a compute
 1999 construct then *concurrent-locality* is processed as follows: variables appearing in a *local* are treated
 2000 as appearing in a **private** clause; variables appearing in a *local_init* are treated as appearing in a
 2001 **firstprivate** clause; variables appearing in a *shared* are treated as appearing in a **copy** clause;
 2002 and a *default(none)* locality spec implies a **default (none)** clause on the compute construct. If
 2003 the **loop** construct is not combined with a compute construct, the behavior is implementation-
 2004 defined.

2005 Restrictions

- 2006 • Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **independent**, **auto**, and **tile**
 2007 clauses may follow a **device_type** clause.
- 2008 • The *int-expr* argument to the **worker** and **vector** clauses must be invariant in the kernels
 2009 region.
- 2010 • A loop associated with a **loop** construct that does not have a **seq** clause must be written to
 2011 meet all of the following conditions:
 - 2012 – The loop variable must be of integer, C/C++ pointer, or C++ random-access iterator
 2013 type.
 - 2014 – The loop variable must monotonically increase or decrease in the direction of its termi-
 2015 nation condition.
 - 2016 – The loop trip count must be computable in constant time when entering the **loop** con-
 2017 struct.

2018 For a C++ range-based **for** loop, the loop variable identified by the above conditions is the
 2019 internal iterator, such as a pointer, that the compiler generates to iterate the range. It is not the
 2020 variable declared by the **for** loop.

- 2021 • Only one of the **seq**, **independent**, and **auto** clauses may appear.
- 2022 • A **gang**, **worker**, or **vector** clause may not appear if a **seq** clause appears.
- 2023 • A **tile** and **collapse** clause may not appear on **loop** that is associated with **do concurrent**.

2024 2.9.1 collapse clause

2025 The **collapse** clause is used to specify how many nested loops are associated with the **loop**
 2026 construct. The argument to the **collapse** clause must be a constant positive integer expression.
 2027 If no **collapse** clause appears, only the immediately following loop is associated with the **loop**
 2028 construct.

2029 If more than one loop is associated with the **loop** construct, the iterations of all the associated loops
 2030 are all scheduled according to the rest of the clauses. The trip count for all loops associated with
 2031 the **collapse** clause must be computable and invariant in all the loops. The particular integer
 2032 type used to compute the trip count for the collapsed loops is implementation defined. However, the
 2033 integer type used for the trip count has at least the precision of each loop variable of the associated
 2034 loops.

2035 It is implementation-defined whether a **gang**, **worker** or **vector** clause on the construct is ap-
 2036 plied to each loop, or to the linearized iteration space.

2037 The associated loops are the n nested loops that immediately follow the loop construct. If the
 2038 **force** modifier does not appear, then the associated loops must be tightly nested. If the **force**
 2039 modifier appears, then any intervening code may be executed multiple times as needed to perform
 2040 the collapse.

2041 Restrictions

- 2042 • Each associated loop, except the innermost, must contain exactly one loop or loop nest.
- 2043 • Intervening code must not contain other OpenACC directives or calls to API routines.

2044 2045 Examples

- 2047 • In the code below, a compiler may choose to move the call to **tan** inside the inner loop in
 2048 order to collapse the two loops, resulting in redundant execution of the intervening code.

```

2049     #pragma acc parallel loop collapse(force:2)
2050     {
2051         for ( int i = 0; i < 360; i++ )
2052         {
2053             // This operation may be executed additional times in order
2054             // to perform the forced collapse.
2055             tanI = tan(a[i]);
2056             for ( int j = 0; j < N; j++ )
2057             {
  
```

```

2058         // Do Something.
2059     }
2060 }
2061 }

```

2062



2063 2.9.2 gang clause

2064 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
 2065 the **gang** clause behaves as follows. It specifies that the iterations of the associated loop or loops
 2066 are to be executed in parallel by distributing the iterations among the gangs along the associated
 2067 dimension created by the compute construct. The associated dimension is the value of the **dim**
 2068 argument, if it appears, or is dimension one. The **dim** argument must be a constant positive integer
 2069 with value 1, 2, or 3. If the associated dimension is d , a **loop** construct with the **gang** clause
 2070 transitions a compute region from gang-redundant mode to gang-partitioned mode on dimension d
 2071 (GRd to GPd). The number of gangs in dimension d is controlled by the **parallel** construct; the
 2072 **num** argument is not allowed. The loop iterations must be data independent, except for *vars* which
 2073 appear in a **reduction** clause or which are modified in an atomic region.

2074 When the parent compute construct is a **kernels** construct, the **gang** clause behaves as follows.
 2075 It specifies that the iterations of the associated loop or loops are to be executed in parallel across the
 2076 gangs. The **dim** argument is not allowed. An argument with no keyword or with the **num** keyword
 2077 is allowed only when the **num_gangs** does not appear on the **kernels** construct. If an argument
 2078 with no keyword or an argument after the **num** keyword appears, it specifies how many gangs to use
 2079 to execute the iterations of this loop. The region of a loop with the **gang** clause may not contain
 2080 another loop with a **gang** clause unless within a nested compute region.

2081 The scheduling of loop iterations to gangs is not specified unless the **static** modifier appears as
 2082 an argument. If the **static** modifier appears with an integer expression, that expression is used
 2083 as a *chunk* size. If the static modifier appears with an asterisk, the implementation will select a
 2084 *chunk* size. The iterations are divided into chunks of the selected *chunk* size, and the chunks are
 2085 assigned to gangs starting with gang zero and continuing in round-robin fashion. Two **gang** loops
 2086 in the same parallel region with the same number of iterations, and with **static** clauses with the
 2087 same argument, will assign the iterations to gangs in the same manner. Two **gang** loops in the
 2088 same kernels region with the same number of iterations, the same number of gangs to use, and with
 2089 **static** clauses with the same argument, will assign the iterations to gangs in the same manner.

2090 A **gang(dim:1)** clause is implied on a data-independent **loop** construct without an explicit
 2091 **gang** clause if the following conditions hold while ignoring **gang**, **worker**, and **vector** clauses
 2092 on any sequential **loop** constructs:

- 2093 • This **loop** construct's parent compute construct, if any, is not a **kernels** construct.
- 2094 • An explicit **gang(dim:1)** clause would be permitted on this **loop** construct.
- 2095 • For every lexically enclosing data-independent **loop** construct, either an explicit **gang(dim:1)**
 2096 clause would not be permitted on the enclosing **loop** construct, or the enclosing **loop** con-
 2097 struct lexically encloses a compute construct that lexically encloses this **loop** construct.

2098 **Note:** As a performance optimization, the implementation might select different levels of paral-
 2099 lelism for a **loop** construct than specified by explicitly or implicitly determined clauses as long

2100 as it can prove program semantics are preserved. In particular, the implementation must consider
 2101 semantic differences between gang-redundant and gang-partitioned mode. For example, in a series
 2102 of tightly nested, data-independent **loop** constructs, implementations often move gang-partitioning
 2103 from one **loop** construct to another without affecting semantics.

2104 **Note:** If the **auto** or **device_type** clause appears on a **loop** construct, it is the programmer's
 2105 responsibility to ensure that program semantics are the same regardless of whether the **auto** clause
 2106 is treated as **independent** or **seq** and regardless of the device type for which the program is
 2107 compiled. In particular, the programmer must consider the effect on both explicitly and implicitly
 2108 determined **gang** clauses and thus on gang-redundant and gang-partitioned mode. Examples in
 2109 Section 2.9.11 demonstrate this issue for the **auto** clause.

2110 Restrictions

- 2111 • At most one **gang** clause may appear on a loop directive.
- 2112 • The region of a loop with a **gang(dim:d)** clause may not contain a loop construct with a
 2113 **gang(dim:e)** clause where $e \geq d$ unless it appears within a nested compute region.

2114 2.9.3 worker clause

2115 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
 2116 the **worker** clause specifies that the iterations of the associated loop or loops are to be executed
 2117 in parallel by distributing the iterations among the multiple workers within a single gang. A **loop**
 2118 construct with a **worker** clause causes a gang to transition from worker-single mode to worker-
 2119 partitioned mode. In contrast to the **gang** clause, the **worker** clause first activates additional
 2120 worker-level parallelism and then distributes the loop iterations across those workers. No argu-
 2121 ment is allowed. The loop iterations must be data independent, except for *vars* which appear in
 2122 a **reduction** clause or which are modified in an atomic region. The region of a loop with the
 2123 **worker** clause may not contain a loop with the **gang** or **worker** clause unless within a nested
 2124 compute region.

2125 When the parent compute construct is a **kernels** construct, the **worker** clause specifies that the
 2126 iterations of the associated loop or loops are to be executed in parallel across the workers within
 2127 a single gang. An argument is allowed only when the **num_workers** does not appear on the
 2128 **kernels** construct. The optional argument specifies how many workers per gang to use to execute
 2129 the iterations of this loop. The region of a loop with the **worker** clause may not contain a loop
 2130 with a **gang** or **worker** clause unless within a nested compute region.

2131 All workers will complete execution of their assigned iterations before any worker proceeds beyond
 2132 the end of the loop.

2133 2.9.4 vector clause

2134 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
 2135 the **vector** clause specifies that the iterations of the associated loop or loops are to be executed
 2136 in vector or SIMD mode. A **loop** construct with a **vector** clause causes a worker to transition
 2137 from vector-single mode to vector-partitioned mode. Similar to the **worker** clause, the **vector**
 2138 clause first activates additional vector-level parallelism and then distributes the loop iterations across
 2139 those vector lanes. The operations will execute using vectors of the length specified or chosen for
 2140 the parallel region. The loop iterations must be data independent, except for *vars* which appear in
 2141 a **reduction** clause or which are modified in an atomic region. The region of a loop with the

2142 **vector** clause may not contain a loop with the **gang**, **worker**, or **vector** clause unless within
2143 a nested compute region.

2144 When the parent compute construct is a **kernels** construct, the **vector** clause specifies that the
2145 iterations of the associated loop or loops are to be executed with vector or SIMD processing. An
2146 argument is allowed only when the **vector_length** does not appear on the **kernels** construct.
2147 If an argument appears, the iterations will be processed in vector strips of that length; if no argument
2148 appears, the implementation will choose an appropriate vector length. The region of a loop with the
2149 **vector** clause may not contain a loop with a **gang**, **worker**, or **vector** clause unless within a
2150 nested compute region.

2151 All vector lanes will complete execution of their assigned iterations before any vector lane proceeds
2152 beyond the end of the loop.

2153 2.9.5 seq clause

2154 The **seq** clause specifies that the associated loop or loops are to be executed sequentially by the
2155 accelerator. This clause will override any automatic parallelization or vectorization.

2156 2.9.6 independent clause

2157 The **independent** clause tells the implementation that the loop iterations must be data indepen-
2158 dent, except for *vars* which appear in a **reduction** clause or which are modified in an atomic
2159 region. This allows the implementation to generate code to execute the iterations in parallel with no
2160 synchronization.

2161 A **loop** construct with no **auto** or **seq** clause is treated as if it has the **independent** clause
2162 when it is an orphaned **loop** construct or its parent compute construct is a **parallel** construct.

2163 Note

- 2164 • It is likely a programming error to use the **independent** clause on a loop if any iteration
2165 writes to a variable or array element that any other iteration also writes or reads, except for
2166 *vars* which appear in a **reduction** clause or which are modified in an atomic region.
- 2167 • The implementation may be restricted in the levels of parallelism it can apply by the presence
2168 of **loop** constructs with **gang**, **worker**, or **vector** clauses for outer or inner loops.

2169 2.9.7 auto clause

2170 The **auto** clause specifies that the implementation must analyze the loop and determine whether the
2171 loop iterations are data-independent. If it determines that the loop iterations are data-independent,
2172 the implementation must treat the **auto** clause as if it is an **independent** clause. If not, or if it
2173 is unable to make a determination, it must treat the **auto** clause as if it is a **seq** clause, and it must
2174 ignore any **gang**, **worker**, or **vector** clauses on the loop construct.

2175 When the parent compute construct is a **kernels** construct, a **loop** construct with no **independent**
2176 or **seq** clause is treated as if it has the **auto** clause.

2177 2.9.8 tile clause

2178 The **tile** clause specifies that the implementation should split each loop in the loop nest into two
2179 loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the **tile**

2180 clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression
 2181 or an asterisk. If there are n tile sizes in the list, the **loop** construct must be immediately followed
 2182 by n tightly-nested loops. The first argument in the *size-expr-list* corresponds to the innermost loop
 2183 of the n associated loops, and the last element corresponds to the outermost associated loop. If the
 2184 tile size is an asterisk, the implementation will choose an appropriate value. Each loop in the nest
 2185 will be split or *strip-mined* into two loops, an outer *tile* loop and an inner *element* loop. The trip
 2186 count of the element loop will be limited to the corresponding tile size from the *size-expr-list*. The
 2187 *tile* loops will be reordered to be outside all the *element* loops, and the *element* loops will all be
 2188 inside the *tile* loops.

2189 If the **vector** clause appears on the **loop** construct, the **vector** clause is applied to the *element*
 2190 loops. If the **gang** clause appears on the **loop** construct, the **gang** clause is applied to the *tile*
 2191 loops. If the **worker** clause appears on the **loop** construct, the **worker** clause is applied to the
 2192 *element* loops if no **vector** clause appears, and to the *tile* loops otherwise.

2193 2.9.9 device_type clause

2194 The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

2195 2.9.10 private clause

2196 The **private** clause on a **loop** construct specifies that a copy of each item in *var-list* will be
 2197 created. If the body of the loop is executed in *vector-partitioned* mode, a copy of the item is created
 2198 for each thread associated with each vector lane. If the body of the loop is executed in *worker-*
 2199 *partitioned vector-single* mode, a copy of the item is created for each worker and shared across the
 2200 set of threads associated with all the vector lanes of that worker. Otherwise, a copy of the item is
 2201 created for each gang in all dimensions and shared across the set of threads associated with all the
 2202 vector lanes of all the workers of that gang.

2203 Restrictions

- 2204 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private**
 2205 clauses.

2206 2207 Examples

2208

- 2209 • In the example below, **tmp** is private to each worker of every gang but shared across all the
 2210 vector lanes of a worker.

```

2211     !$acc parallel
2212     !$acc loop gang
2213     do k = 1, n
2214         !$acc loop worker private(tmp)
2215         do j = 1, n
2216             !a single vector lane in each gang and worker assigns to tmp
2217             tmp = b(j,k) + c(j,k)
2218             !$acc loop vector
2219             do i = 1, n
2220                 !all vector lanes use the result of the above update to tmp
2221                 a(i,j,k) = a(i,j,k) + tmp/div
  
```

```

2222         enddo
2223     enddo
2224 enddo
2225 !$acc end parallel

```

2226 • In the example below, **tmp** is private to each gang in every dimension.

```

2227 !$acc parallel num_gangs(3,50,150)
2228 !$acc loop gang(dim:3)
2229 do k = 1, n
2230     !$acc loop gang(dim:2) private(tmp)
2231     do j = 1, n
2232         !all gangs along dimension 1 execute in gang redundant mode and
2233         !assign to tmp which is private to each gang in all dimensions
2234         tmp = b(j,k) + c(j,k)
2235         !$acc loop gang(dim:1)
2236         do i = 1, n
2237             a(i,j,k) = a(i,j,k) + tmp/div
2238         enddo
2239     enddo
2240 enddo
2241 !$acc end parallel

```

2242 ▲

2.9.11 reduction clause

2244 The **reduction** clause specifies a reduction operator and one or more *vars*. For each reduction
2245 *var*, a private copy is created in the same manner as for a **private** clause on the **loop** construct,
2246 and initialized for that operator; see the table in Section 2.5.15 reduction clause. After the loop, the
2247 values for each thread are combined using the specified reduction operator, and the result combined
2248 with the value of the original *var* and stored in the original *var*. If the original *var* is not private,
2249 this update occurs by the end of the compute region, and any access to the original *var* is undefined
2250 within the compute region. Otherwise, the update occurs at the end of the loop. If the reduction
2251 *var* is an array or subarray, the reduction operation is logically equivalent to applying that reduction
2252 operation to each array element of the array or subarray individually. If the reduction *var* is a com-
2253 posite variable, the reduction operation is logically equivalent to applying that reduction operation
2254 to each member of the composite variable individually.

2255 If a variable is involved in a reduction that spans multiple nested loops where two or more of those
2256 loops have associated **loop** directives, a **reduction** clause containing that variable must appear
2257 on each of those **loop** directives.

Restrictions

- 2258 • A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name,
2259 an array element, or a subarray (refer to Section 2.7.1).
- 2260 • Reduction clauses on nested constructs for the same reduction *var* must have the same reduc-
2261 tion operator.
- 2262 • Every *var* in a **reduction** clause appearing on an orphaned **loop** construct must be private.
- 2263 • The restrictions for a **reduction** clause on a compute construct listed in in Section 2.5.15
2264 reduction clause also apply to a **reduction** clause on a **loop** construct.

- 2266 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in
2267 **reduction** clauses.
- 2268 • See Section 2.6.2 Variables with Implicitly Determined Data Attributes for a restriction re-
2269 quiring certain loop reduction variables to have explicit data clauses on their parent compute
2270 constructs.
- 2271 • A **reduction** clause may not appear on a **loop** directive that has a **gang** clause with a
2272 **dim:** argument whose value is greater than 1.
- 2273 • A **reduction** clause may not appear on a **loop** directive that has a **gang** clause and
2274 is within a compute construct that has a **num_gangs** clause with more than one explicit
2275 argument.

2276 Examples

2277

- 2279 • **x** is not private at the **loop** directive below, so its reduction normally updates **x** at the end
2280 of the parallel region, where gangs synchronize. When possible, the implementation might
2281 choose to partially update **x** at the loop exit instead, or fully if **num_gangs (1)** were added
2282 to the **parallel** directive. However, portable applications cannot rely on such early up-
2283 dates, so accesses to **x** are undefined within the parallel region outside the loop.

```
2284     int x = 0;
2285     #pragma acc parallel copy(x)
2286     {
2287         // gang-shared x undefined
2288         #pragma acc loop gang worker vector reduction(+:x)
2289         for (int i = 0; i < I; ++i)
2290             x += 1; // vector-private x modified
2291         // gang-shared x undefined
2292     } // gang-shared x updated for gang/worker/vector reduction
2293     // x = I
```

- 2294 • **x** is private at each of the innermost two **loop** directives below, so each of their reductions
2295 updates **x** at the loop's exit. However, **x** is not private at the outer **loop** directive, so its
2296 reduction updates **x** by the end of the parallel region instead.

```
2297     int x = 0;
2298     #pragma acc parallel copy(x)
2299     {
2300         // gang-shared x undefined
2301         #pragma acc loop gang reduction(+:x)
2302         for (int i = 0; i < I; ++i) {
2303             #pragma acc loop worker reduction(+:x)
2304             for (int j = 0; j < J; ++j) {
2305                 #pragma acc loop vector reduction(+:x)
2306                 for (int k = 0; k < K; ++k) {
2307                     x += 1; // vector-private x modified
2308                 } // worker-private x updated for vector reduction
2309             } // gang-private x updated for worker reduction
2310         }

```



```

2311         // gang-shared x undefined
2312     } // gang-shared x updated for gang reduction
2313     // x = I * J * K

```

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

```

2317     int x = 0, y = 0;
2318     #pragma acc parallel firstprivate(x) copy(y)
2319     {
2320         // gang-private x = 0; gang-shared y undefined
2321         #pragma acc loop seq reduction(+:x,y)
2322         for (int i = 0; i < I; ++i) {
2323             x += 1; y += 2; // loop-private x and y modified
2324         } // gang-private x updated for trivial seq reduction
2325         // gang-private x = I; gang-shared y undefined
2326         #pragma acc loop worker reduction(+:x,y)
2327         for (int i = 0; i < I; ++i) {
2328             x += 1; y += 2; // worker-private x and y modified
2329         } // gang-private x updated for worker reduction
2330         // gang-private x = 2 * I; gang-shared y undefined
2331         #pragma acc loop vector reduction(+:x,y)
2332         for (int i = 0; i < I; ++i) {
2333             x += 1; y += 2; // vector-private x and y modified
2334         } // gang-private x updated for vector reduction
2335         // gang-private x = 3 * I; gang-shared y undefined
2336     } // gang-shared y updated for gang/seq/worker/vector reductions
2337     // x = 0; y = 3 * I * 2

```

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

```

2342     int x = 0;
2343     #pragma acc parallel loop worker reduction(+:x)
2344     for (int i = 0; i < I; ++i) {
2345         x += 1; // worker-private x modified
2346     } // gang-shared x updated for gang/worker reduction
2347     // x = I
2348
2349     int x = 0;
2350     #pragma acc parallel copy(x)
2351     {
2352         // gang-shared x undefined
2353         #pragma acc loop worker reduction(+:x)
2354         for (int i = 0; i < I; ++i) {
2355             x += 1; // worker-private x modified
2356         }
2357         // gang-shared x undefined
2358     } // gang-shared x updated for gang/worker reduction
2359     // x = I

```

- If the implementation treats the **auto** clause below as **independent**, the loop executes in gang-partitioned mode and thus examines every element of **arr** once to compute **arr**'s maximum. However, if the implementation treats **auto** as **seq**, the gangs redundantly compute **arr**'s maximum, but the combined result is still **arr**'s maximum. Either way, because **x** is not private at the **loop** directive, the reduction updates **x** by the end of the parallel region.

```

2365     int x = 0;
2366     const int *arr = /*array of I values*/;
2367     #pragma acc parallel copy(x)
2368     {
2369         // gang-shared x undefined
2370         #pragma acc loop auto gang reduction(max:x)
2371         for (int i = 0; i < I; ++i) {
2372             // complex loop body
2373             x = x < arr[i] ? arr[i] : x; // gang- or loop-private
2374                                     // x modified
2375         }
2376         // gang-shared x undefined
2377     } // gang-shared x updated for gang or gang/seq reduction
2378     // x = arr maximum

```

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

```

2384     int x = 0;
2385     const int *arr = /*array of I values*/;
2386     #pragma acc parallel copy(x)
2387     {
2388         // gang-shared x undefined
2389         #pragma acc loop auto gang reduction(+:x)
2390         for (int i = 0; i < I; ++i) {
2391             // complex loop body
2392             x += arr[i]; // gang or loop-private x modified
2393         }
2394         // gang-shared x undefined
2395     } // gang-shared x updated for gang or gang/seq reduction
2396     // x = arr sum possibly times number of gangs

```

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

```

2402     int x = 0, y = 0;
2403     #pragma acc parallel copy(x) reduction(+:x,y)
2404     {
2405         int z = 0;
2406         #pragma acc loop gang vector reduction(+:x,z)
2407         for (int i = 0; i < I; ++i) {
2408             x += 1; z += 2; // vector-private x and z modified

```

```

2409         } // gang-private x and z updated for vector reduction
2410         y += z; // gang-private y modified
2411     } // gang-shared x and y updated for gang reduction
2412     // x = I; y = I * 2

```

2413

2414

2.10 Cache Directive

2415

Summary

2416

2417 The **cache** directive may appear at the top of (inside of) a loop. It specifies array elements or
 2418 subarrays that should be fetched into the highest level of the cache for the body of the loop.

Syntax

2419

2420 In C and C++, the syntax of the **cache** directive is

```

2421     #pragma acc cache ( [readonly:]var-list ) new-line

```

2422 In Fortran, the syntax of the **cache** directive is

```

2423     !$acc cache ( [readonly:]var-list )

```

2424 A *var* in a **cache** directive must be a single array element or a simple subarray. In C and C++,
 2425 a simple subarray is an array name followed by an extended array range specification in brackets,
 2426 with start and length, such as

```

2427     arr [lower:length]

```

2428 where the lower bound is a constant, loop invariant, or the **for** loop variable plus or minus a
 2429 constant or loop invariant, and the length is a constant.

2430 In Fortran, a simple subarray is an array name followed by a comma-separated list of range specifi-
 2431 cations in parentheses, with lower and upper bound subscripts, such as

```

2432     arr (lower:upper, lower2:upper2)

```

2433 The lower bounds must be constant, loop invariant, or the **do** loop variable plus or minus a constant
 2434 or loop invariant; moreover the difference between the corresponding upper and lower bounds must
 2435 be a constant.

2436 If the optional **readonly** modifier appears, then the implementation may assume that the data
 2437 referenced by any *var* in that directive is never written to within the applicable region.

Restrictions

2438

- 2439 • If an array element or subarray is listed in a **cache** directive, all references to that array
 2440 during execution of that loop iteration must not refer to elements of the array outside the
 2441 index range specified in the **cache** directive.
- 2442 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **cache**
 2443 directives.

2.11 Combined Constructs

2444

2445 **Summary**

2446 The combined OpenACC **parallel loop**, **serial loop**, and **kernels loop** constructs are
 2447 shortcuts for specifying a **loop** construct nested immediately inside a **parallel**, **serial**, or
 2448 **kernels** construct. The meaning is identical to explicitly specifying a **parallel**, **serial**, or
 2449 **kernels** construct containing a **loop** construct. Any clause that is allowed on a **parallel** or
 2450 **loop** construct is allowed on the **parallel loop** construct; any clause allowed on a **serial** or
 2451 **loop** construct is allowed on a **serial loop** construct; and any clause allowed on a **kernels**
 2452 or **loop** construct is allowed on a **kernels loop** construct.

2453 **Syntax**

2454 In C and C++, the syntax of the **parallel loop** construct is

```
2455     #pragma acc parallel loop [clause-list] new-line  
2456         for loop
```

2457 In Fortran, the syntax of the **parallel loop** construct is

```
2458     !$acc parallel loop [clause-list]  
2459         do loop  
2460     [!$acc end parallel loop]
```

2461 The associated structured block is the loop which must immediately follow the directive. Any of
 2462 the **parallel** or **loop** clauses valid in a parallel region may appear.

2463 In C and C++, the syntax of the **serial loop** construct is

```
2464     #pragma acc serial loop [clause-list] new-line  
2465         for loop
```

2466 In Fortran, the syntax of the **serial loop** construct is

```
2467     !$acc serial loop [clause-list]  
2468         do loop  
2469     [!$acc end serial loop]
```

2470 The associated structured block is the loop which must immediately follow the directive. Any of
 2471 the **serial** or **loop** clauses valid in a serial region may appear.

2472 In C and C++, the syntax of the **kernels loop** construct is

```
2473     #pragma acc kernels loop [clause-list] new-line  
2474         for loop
```

2475 In Fortran, the syntax of the **kernels loop** construct is

```
2476     !$acc kernels loop [clause-list]  
2477         do loop  
2478     [!$acc end kernels loop]
```

2479 The associated structured block is the loop which must immediately follow the directive. Any of
 2480 the **kernels** or **loop** clauses valid in a kernels region may appear.

2481 A **private** or **reduction** clause on a combined construct is treated as if it appeared on the
 2482 **loop** construct. In addition, a **reduction** clause on a combined construct implies a **copy** clause
 2483 as described in Section 2.6.2.

2484 **Restrictions**

- 2485 • The restrictions for the **parallel**, **serial**, **kernels**, and **loop** constructs apply.

2486 **2.12 Atomic Construct**2487 **Summary**

2488 An **atomic** construct ensures that a specific storage location is accessed and/or updated atomically,
 2489 preventing simultaneous reading and writing by gangs, workers, and vector threads that could result
 2490 in indeterminate values.

2491 **Syntax**

2492 In C and C++, the syntax of the **atomic** constructs is:

```
2493     #pragma acc atomic [ atomic-clause ] new-line  
2494         expression-stmt
```

2495 OR:

```
2496     #pragma acc atomic capture new-line  
2497         structured block
```

2498 Where *atomic-clause* is one of **read**, **write**, **update**, or **capture**. The *expression-stmt* is an
 2499 expression statement with one of the following forms:

2500 If the *atomic-clause* is **read**:

```
2501     v = x;
```

2502 If the *atomic-clause* is **write**:

```
2503     x = expr;
```

2504 If the *atomic-clause* is **update** or no clause appears:

```
2505     x++;
```

```
2506     x--;
```

```
2507     ++x;
```

```
2508     --x;
```

```
2509     x binop= expr;
```

```
2510     x = x binop expr;
```

```
2511     x = expr binop x;
```

2512 If the *atomic-clause* is **capture**:

```
2513     v = x++;
```

```
2514     v = x--;
```

```
2515     v = ++x;
```

```
2516     v = --x;
```

```
2517     v = x binop= expr;
```

```
2518     v = x = x binop expr;
```

```
2519     v = x = expr binop x;
```

2520 The *structured-block* is a structured block with one of the following forms:

```

2521   { v = x; x binop= expr; }
2522   { x binop= expr; v = x; }
2523   { v = x; x = x binop expr; }
2524   { v = x; x = expr binop x; }
2525   { x = x binop expr; v = x; }
2526   { x = expr binop x; v = x; }
2527   { v = x; x = expr; }
2528   { v = x; x++; }
2529   { v = x; ++x; }
2530   { ++x; v = x; }
2531   { x++; v = x; }
2532   { v = x; x--; }
2533   { v = x; --x; }
2534   { --x; v = x; }
2535   { x--; v = x; }

```

2536 In the preceding expressions:

- 2537 • **x** and **v** (as applicable) are both l-value expressions with scalar type.
- 2538 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
2539 the same storage location.
- 2540 • Neither of **v** and *expr* (as applicable) may access the storage location designated by **x**.
- 2541 • Neither of **x** and *expr* (as applicable) may access the storage location designated by **v**.
- 2542 • *expr* is an expression with scalar type.
- 2543 • *binop* is one of +, *, -, /, &, ^, |, <<, or >>.
- 2544 • *binop*, *binop*=, ++, and -- are not overloaded operators.
- 2545 • The expression **x** *binop* *expr* must be mathematically equivalent to **x** *binop* (*expr*). This
2546 requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by
2547 using parentheses around *expr* or subexpressions of *expr*.
- 2548 • The expression *expr* *binop* **x** must be mathematically equivalent to (*expr*) *binop* **x**. This
2549 requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*,
2550 or by using parentheses around *expr* or subexpressions of *expr*.
- 2551 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is
2552 unspecified.

2553 In Fortran the syntax of the **atomic** constructs is:

```

2554   !$acc atomic read
2555       capture-statement
2556   [!$acc end atomic]

```

2557 or

```

2558   !$acc atomic write
2559       write-statement
2560   [!$acc end atomic]

```

2561 OR

```

2562     !$acc atomic [update]
2563         update-statement
2564     [!$acc end atomic]

```

2565 OR

```

2566     !$acc atomic capture
2567         update-statement
2568         capture-statement
2569     !$acc end atomic

```

2570 OR

```

2571     !$acc atomic capture
2572         capture-statement
2573         update-statement
2574     !$acc end atomic

```

2575 OR

```

2576     !$acc atomic capture
2577         capture-statement
2578         write-statement
2579     !$acc end atomic

```

2580 where *write-statement* has the following form (if *atomic-clause* is **write** or **capture**):2581 **x = expr**2582 where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):2583 **v = x**2584 and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**,
2585 or no clause appears):2586 **x = x operator expr**2587 **x = expr operator x**2588 **x = intrinsic_procedure_name (x, expr-list)**2589 **x = intrinsic_procedure_name (expr-list, x)**

2590 In the preceding statements:

- 2591 ● **x** and **v** (as applicable) are both scalar variables of intrinsic type.
- 2592 ● **x** must not be an allocatable variable.
- 2593 ● During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
2594 the same storage location.
- 2595 ● None of **v**, *expr*, and *expr-list* (as applicable) may access the same storage location as **x**.
- 2596 ● None of **x**, *expr*, and *expr-list* (as applicable) may access the same storage location as **v**.
- 2597 ● *expr* is a scalar expression.

- 2598 • *expr-list* is a comma-separated, non-empty list of scalar expressions. If *intrinsic_procedure_name*
2599 refers to **iand**, **ior**, or **ieor**, exactly one expression must appear in *expr-list*.
 - 2600 • *intrinsic_procedure_name* is one of **max**, **min**, **iand**, **ior**, or **ieor**. *operator* is one of **+**,
2601 *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**
 - 2602 • The expression **x operator expr** must be mathematically equivalent to **(expr) operator x**.
2603 This requirement is satisfied if the operators in *expr* have precedence greater than *operator*,
2604 or by using parentheses around *expr* or subexpressions of *expr*.
 - 2605 • The expression *expr operator x* must be mathematically equivalent to **(expr) operator x**.
2606 This requirement is satisfied if the operators in *expr* have precedence equal to or greater than
2607 *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
 - 2608 • *intrinsic_procedure_name* must refer to the intrinsic procedure name and not to other program
2609 entities.
 - 2610 • *operator* must refer to the intrinsic operator and not to a user-defined operator. All assign-
2611 ments must be intrinsic assignments.
 - 2612 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is
2613 unspecified.
- 2614 An **atomic** construct with the **read** clause forces an atomic read of the location designated by **x**.
2615 An **atomic** construct with the **write** clause forces an atomic write of the location designated by
2616 **x**.
- 2617 An **atomic** construct with the **update** clause forces an atomic update of the location designated
2618 by **x** using the designated operator or intrinsic. Note that when no clause appears, the semantics
2619 are equivalent to **atomic update**. Only the read and write of the location designated by **x** are
2620 performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect
2621 to the read or write of the location designated by **x**.
- 2622 An **atomic** construct with the **capture** clause forces an atomic update of the location designated
2623 by **x** using the designated operator or intrinsic while also capturing the original or final value of
2624 the location designated by **x** with respect to the atomic update. The original or final value of the
2625 location designated by **x** is written into the location designated by **v** depending on the form of the
2626 **atomic** construct structured block or statements following the usual language semantics. Only
2627 the read and write of the location designated by **x** are performed mutually atomically. Neither the
2628 evaluation of *expr* or *expr-list*, nor the write to the location designated by **v**, need to be atomic with
2629 respect to the read or write of the location designated by **x**.
- 2630 For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs
2631 enforces mutually exclusive access to the locations designated by **x**. To avoid race conditions, all
2632 accesses of the locations designated by **x** that could potentially occur in parallel must be protected
2633 with an **atomic** construct.
- 2634 Atomic regions do not guarantee exclusive access with respect to any accesses outside of atomic re-
2635 gions to the same storage location **x** even if those accesses occur during the execution of a reduction
2636 clause.
- 2637 If the storage location designated by **x** is not size-aligned (that is, if the byte alignment of **x** is not a
2638 multiple of the size of **x**), then the behavior of the atomic region is implementation-defined.

2639 **Restrictions**

- 2640 • All atomic accesses to the storage locations designated by **x** throughout the program are
- 2641 required to have the same type and type parameters.
- 2642 • Storage locations designated by **x** must be less than or equal in size to the largest available
- 2643 native atomic operator width.

2644 **2.13 Declare Directive**2645 **Summary**

2646 A **declare** directive is used in the declaration section of a Fortran subroutine, function, block
 2647 construct, or module, or following a variable declaration in C or C++. It can specify that a *var* is to
 2648 be allocated in device memory for the duration of the implicit data region of a function, subroutine
 2649 or program, and specify whether the data values are to be transferred from local memory to device
 2650 memory upon entry to the implicit data region, and from device memory to local memory upon exit
 2651 from the implicit data region. These directives create a visible device copy of the *var*.

2652 **Syntax**

2653 In C and C++, the syntax of the **declare** directive is:

2654 **#pragma acc declare** *clause-list new-line*

2655 In Fortran the syntax of the **declare** directive is:

2656 **!\$acc declare** *clause-list*

2657 where *clause* is one of the following:

2658 **copy** (*var-list*)
 2659 **copyin** ([**readonly**:] *var-list*)
 2660 **copyout** (*var-list*)
 2661 **create** (*var-list*)
 2662 **present** (*var-list*)
 2663 **deviceptr** (*var-list*)
 2664 **device_resident** (*var-list*)
 2665 **link** (*var-list*)

2666 The associated region is the implicit region associated with the function, subroutine, or program in
 2667 which the directive appears. If the directive appears in the declaration section of a Fortran *module*
 2668 subprogram, for a Fortran *common block*, or in a C or C++ global or namespace scope, the associated
 2669 region is the implicit region for the whole program. The **copy**, **copyin**, **copyout**, **present**,
 2670 and **deviceptr** data clauses are described in Section 2.7 Data Clauses.

2671 **Restrictions**

- 2672 • A **declare** directive must be in the same scope as the declaration of any *var* that appears
- 2673 in the clauses of the directive or any scope within a C or C++ function or Fortran function,
- 2674 subroutine, or program.
- 2675 • At least one clause must appear on a **declare** directive.
- 2676 • A *var* in a **declare** declare must be a variable or array name, or a Fortran *common block*
- 2677 name between slashes.

- 2678 • A *var* may appear at most once in all the clauses of **declare** directives for a function,
2679 subroutine, program, or module.
- 2680 • In Fortran, assumed-size dummy arrays may not appear in a **declare** directive.
- 2681 • In Fortran, pointer arrays may appear, but pointer association is not preserved in device mem-
2682 mory.
- 2683 • In a Fortran *module* declaration section, only **create**, **copyin**, **device_resident**, and
2684 **link** clauses are allowed.
- 2685 • In C or C++ global or namespace scope, only **create**, **copyin**, **deviceptr**,
2686 **device_resident** and **link** clauses are allowed.
- 2687 • C and C++ *extern* variables may only appear in **create**, **copyin**, **deviceptr**,
2688 **device_resident** and **link** clauses on a **declare** directive.
- 2689 • In C or C++, the **link** clause must appear at global or namespace scope or the arguments
2690 must be *extern* variables. In Fortran, the **link** clause must appear in a *module* declaration
2691 section, or the arguments must be *common block* names enclosed in slashes.
- 2692 • In C or C++, a **longjmp** call in the region must return to a **setjmp** call within the region.
- 2693 • In C++, an exception thrown in the region must be handled within the region.
- 2694 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional dummy arguments
2695 in data clauses, including **device_resident** clauses.

2696 2.13.1 device_resident clause

2697 Summary

2698 The **device_resident** clause specifies that the memory for the named variables should be
2699 allocated in the current device memory and not in local memory. The host may not be able to access
2700 variables in a **device_resident** clause. The accelerator data lifetime of global variables or
2701 common blocks that appear in a **device_resident** clause is the entire execution of the program.

2702 In Fortran, if the variable has the Fortran *allocatable* attribute, the memory for the variable will
2703 be allocated in and deallocated from the current device memory when the host thread executes
2704 an **allocate** or **deallocate** statement for that variable, if the current device is a non-shared
2705 memory device. If the variable has the Fortran *pointer* attribute, it may be allocated or deallocated
2706 by the host in the current device memory, or may appear on the left hand side of a pointer assignment
2707 statement, if the right hand side variable itself appears in a **device_resident** clause.

2708 In Fortran, the argument to a **device_resident** clause may be a *common block* name enclosed
2709 in slashes; in this case, all declarations of the common block must have a matching
2710 **device_resident** clause. In this case, the *common block* will be statically allocated in de-
2711 vice memory, and not in local memory. The *common block* will be available to accelerator routines;
2712 see Section 2.15 Procedure Calls in Compute Regions.

2713 In a Fortran *module* declaration section, a *var* in a **device_resident** clause will be available to
2714 accelerator subprograms.

2715 In C or C++ global scope, a *var* in a **device_resident** clause will be available to accelerator
2716 routines. A C or C++ *extern* variable may appear in a **device_resident** clause only if the

2717 actual declaration and all *extern* declarations are also followed by **device_resident** clauses.

2718 2.13.2 create clause

2719 For data in shared memory, no action is taken.

2720 For data not in shared memory, the **create** clause on a **declare** directive behaves as follows,
2721 for each *var* in *var-list*:

- 2722 • At entry to an implicit data region where the **declare** directive appears:
 - 2723 – If *var* is present, a *present increment* action with the structured reference counter is
 - 2724 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 2725 – Otherwise, a *create* action with the structured reference counter is performed. If *var* is
 - 2726 a pointer reference, an *attach* action is performed.
- 2727 • At exit from an implicit data region where the **declare** directive appears:
 - 2728 – If the structured reference counter for *var* is zero, no action is taken.
 - 2729 – Otherwise, a *present decrement* action with the structured reference counter is per-
 - 2730 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
 - 2731 and dynamic reference counters are zero, a *delete* action is performed.

2732 If the **declare** directive appears in a global context, then the data in *var-list* is statically allocated
2733 in device memory and the structured reference counter is set to one.

2734 In Fortran, if a variable *var* in *var-list* has the Fortran *allocatable* or *pointer* attribute, then:

- 2735 • An **allocate** statement for *var* will allocate memory in both local memory as well as in the
- 2736 current device memory, for a non-shared memory device, and the dynamic reference counter
- 2737 will be set to one.
- 2738 • A **deallocate** statement for *var* will deallocate memory from both local memory as well
- 2739 as the current device memory, for a non-shared memory device, and the dynamic reference
- 2740 counter will be set to zero. If the structured reference counter is not zero, a runtime error is
- 2741 issued.

2742 In Fortran, if a variable *var* in *var-list* has the Fortran *pointer* attribute, then it may appear on the
2743 left hand side of a pointer assignment statement, if the right hand side variable itself appears in a
2744 **create** clause.

2745 Errors

- 2746 • In Fortran, an **acc_error_present** error is issued at a deallocate statement if the struc-
- 2747 tured reference counter is not zero.

2748 See Section 5.2.2.

2749 2.13.3 link clause

2750 The **link** clause is used for large global host static data that is referenced within an accelerator
2751 routine and that should have a dynamic data lifetime on the device. The **link** clause specifies that
2752 only a global link for the named variables should be statically created in accelerator memory. The
2753 host data structure remains statically allocated and globally available. The device data memory will

2754 be allocated only when the global variable appears on a data clause for a **data** construct, compute
 2755 construct, or **enter data** directive. The arguments to the **link** clause must be global data. A
 2756 **declare link** clause must be visible everywhere the global variables or common block variables
 2757 are explicitly or implicitly used in a data clause, compute construct, or accelerator routine. The
 2758 global variable or *common block* variables may be used in accelerator routines. The accelerator
 2759 data lifetime of variables or common blocks that appear in a **link** clause is the data region that
 2760 allocates the variable or common block with a data clause, or from the execution of the **enter**
 2761 **data** directive that allocates the data until an **exit data** directive deallocates it or until the end
 2762 of the program.

2763 2.14 Executable Directives

2764 2.14.1 Init Directive

2765 Summary

2766 The **init** directive initializes the runtime for the given device or devices of the given device type.
 2767 This can be used to isolate any initialization cost from the computational cost, when collecting
 2768 performance statistics. If no device type appears all devices will be initialized. An **init** directive
 2769 may be used in place of a call to the **acc_init** or **acc_init_device** runtime API routine, as
 2770 described in Section 3.2.7.

2771 Syntax

2772 In C and C++, the syntax of the **init** directive is:

```
2773     #pragma acc init [clause-list] new-line
```

2774 In Fortran the syntax of the **init** directive is:

```
2775     !$acc init [clause-list]
```

2776 where *clause* is one of the following:

```
2777     device_type ( device-type-list )
```

```
2778     device_num ( int-expr )
```

```
2779     if ( condition )
```

2780

2781 device_type clause

2782 The **device_type** clause specifies the type of device that is to be initialized in the runtime. If the
 2783 **device_type** clause appears, then the *acc-current-device-type-var* for the current thread is set to
 2784 the argument value. If no **device_num** clause appears then all devices of this type are initialized.

2785 device_num clause

2786 The **device_num** clause specifies the device id to be initialized. If the **device_num** clause
 2787 appears, then the *acc-current-device-num-var* for the current thread is set to the argument value. If
 2788 no **device_type** clause appears, then the specified device id will be initialized for all available
 2789 device types.

2790 **if clause**

2791 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
 2792 perform the initialization unconditionally. When an **if** clause appears, the implementation will
 2793 generate code to conditionally perform the initialization only when the *condition* evaluates to *true*.

2794 **Restrictions**

- 2795 • This directive may only appear in code executed on the host.
- 2796 • If the directive is called more than once without an intervening **acc_shutdown** call or
 2797 **shutdown** directive, with a different value for the device type argument, the behavior is
 2798 implementation-defined.
- 2799 • If some accelerator regions are compiled to only use one device type, using this directive with
 2800 a different device type may produce undefined behavior.

2801 **Errors**

- 2802 • An **acc_error_device_type_unavailable** error is issued if a **device_type** clause
 2803 appears and no device of that device type is available, or if no **device_type** clause appears
 2804 and no device of the current device type is available.
- 2805 • An **acc_error_device_unavailable** error is issued if a **device_num** clause ap-
 2806 pears and the *int-expr* is not a valid device number or that device is not available, or if no
 2807 **device_num** clause appears and the current device is not available.
- 2808 • An **acc_error_device_init** error is issued if the device cannot be initialized.

2809 See Section 5.2.2.

2810 **2.14.2 Shutdown Directive**2811 **Summary**

2812 The **shutdown** directive shuts down the connection to the given device or devices of the given
 2813 device type, and frees any associated runtime resources. This ends all data lifetimes in device
 2814 memory, which effectively sets structured and dynamic reference counters to zero. A **shutdown**
 2815 directive may be used in place of a call to the **acc_shutdown** or **acc_shutdown_device**
 2816 runtime API routine, as described in Section 3.2.8.

2817 **Syntax**

2818 In C and C++, the syntax of the **shutdown** directive is:

```
2819 #pragma acc shutdown [clause-list] new-line
```

2820 In Fortran the syntax of the **shutdown** directive is:

```
2821 !$acc shutdown [clause-list]
```

2822 where *clause* is one of the following:

```
2823 device_type ( device-type-list )
```

```
2824 device_num ( int-expr )
```

```
2825 if ( condition )
```

2826

2827 **device_type clause**

2828 The **device_type** clause specifies the type of device that is to be disconnected from the runtime.
 2829 If no **device_num** clause appears then all devices of this type are disconnected.

2830 **device_num clause**

2831 The **device_num** clause specifies the device id to be disconnected.
 2832 If no clauses appear then all available devices will be disconnected.

2833 **if clause**

2834 The **if** clause is optional; when there is no **if** clause, the implementation will generate code
 2835 to perform the shutdown unconditionally. When an **if** clause appears, the implementation will
 2836 generate code to conditionally perform the shutdown only when the *condition* evaluates to *true*.

2837 **Restrictions**

- 2838 • This directive may only appear in code executed on the host.

2839 **Errors**

- 2840 • An **acc_error_device_type_unavailable** error is issued if a **device_type** clause
 2841 appears and no device of that device type is available,
- 2842 • An **acc_error_device_unavailable** error is issued if a **device_num** clause ap-
 2843 pears and the *int-expr* is not a valid device number or that device is not available.
- 2844 • An **acc_error_device_shutdown** error is issued if there is an error shutting down the
 2845 device.

2846 See Section 5.2.2.

2847 **2.14.3 Set Directive**2848 **Summary**

2849 The **set** directive provides a means to modify internal control variables using directives. Each form
 2850 of the **set** directive is functionally equivalent to a matching runtime API routine.

2851 **Syntax**

2852 In C and C++, the syntax of the **set** directive is:

```
2853 #pragma acc set [clause-list] new-line
```

2854 In Fortran the syntax of the **set** directive is:

```
2855 !$acc set [clause-list]
```

2856 where *clause* is one of the following

```
2857 default_async ( int-expr )
2858 device_num ( int-expr )
2859 device_type ( device-type-list )
2860 if ( condition )
```

2861 **default_async clause**

2862 The **default_async** clause specifies the asynchronous queue that should be used if no queue ap-
 2863 pears and changes the value of *acc-default-async-var* for the current thread to the argument value.
 2864 If the value is **acc_async_default**, the value of *acc-default-async-var* will revert to the ini-
 2865 tial value, which is implementation-defined. A **set default_async** directive is functionally
 2866 equivalent to a call to the **acc_set_default_async** runtime API routine, as described in Sec-
 2867 tion 3.2.14.

2868 **device_num clause**

2869 The **device_num** clause specifies the device number to set as the default device for accelerator
 2870 regions and changes the value of *acc-current-device-num-var* for the current thread to the argument
 2871 value. If the value of **device_num** argument is negative, the runtime will revert to the default be-
 2872 havior, which is implementation-defined. A **set device_num** directive is functionally equivalent
 2873 to the **acc_set_device_num** runtime API routine, as described in Section 3.2.4.

2874 **device_type clause**

2875 The **device_type** clause specifies the device type to set as the default device type for accelerator
 2876 regions and sets the value of *acc-current-device-type-var* for the current thread to the argument
 2877 value. If the value of the **device_type** argument is zero or the clause does not appear, the
 2878 selected device number will be used for all attached accelerator types. A **set device_type**
 2879 directive is functionally equivalent to a call to the **acc_set_device_type** runtime API routine,
 2880 as described in Section 3.2.2.

2881 **if clause**

2882 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
 2883 perform the set operation unconditionally. When an **if** clause appears, the implementation will
 2884 generate code to conditionally perform the set operation only when the *condition* evaluates to *true*.

2885 **Restrictions**

- 2886 • This directive may only appear in code executed on the host.
- 2887 • Passing **default_async** the value of **acc_async_noval** has no effect.
- 2888 • Passing **default_async** the value of **acc_async_sync** will cause all asynchronous
 2889 directives in the default asynchronous queue to become synchronous.
- 2890 • Passing **default_async** the value of **acc_async_default** will restore the default
 2891 asynchronous queue to the initial value, which is implementation-defined.
- 2892 • At least one **default_async**, **device_num**, or **device_type** clause must appear.
- 2893 • Two instances of the same clause may not appear on the same directive.

2894 **Errors**

- 2895 • An **acc_error_device_type_unavailable** error is issued if a **device_type** clause
 2896 appears, and no device of that device type is available.
- 2897 • An **acc_error_device_unavailable** error is issued if a **device_num** clause ap-
 2898 pears, and the *int-expr* is not a valid device number.

- 2899 • An **acc_error_invalid_async** error is issued if a **default_async** clause appears,
2900 and the *int-expr* is not a valid *async-argument*.

2901 See Section 5.2.2.

2902 2.14.4 Update Directive

2903 Summary

2904 The **update** directive is used during the lifetime of accelerator data to update *vars* in local memory
2905 with values from the corresponding data in device memory, or to update *vars* in device memory with
2906 values from the corresponding data in local memory.

2907 Syntax

2908 In C and C++, the syntax of the **update** directive is:

2909 **#pragma acc update** *clause-list new-line*

2910 In Fortran the syntax of the **update** data directive is:

2911 **!\$acc update** *clause-list*

2912 where *clause* is one of the following:

```
2913         async [ ( int-expr ) ]
2914         wait [ ( wait-argument ) ]
2915         device_type ( device-type-list )
2916         if ( condition )
2917         if_present
2918         self ( var-list )
2919         host ( var-list )
2920         device ( var-list )
```

2921 Multiple subarrays of the same array may appear in a *var-list* of the same or different clauses on
2922 the same directive. The effect of an **update** clause is to copy data from device memory to local
2923 memory for **update self**, and from local memory to device memory for **update device**. The
2924 updates are done in the order in which they appear on the directive.

2925 Restrictions

- 2926 • At least one **self**, **host**, or **device** clause must appear on an **update** directive.

2927 self clause

2928 The **self** clause specifies that the *vars* in *var-list* are to be copied from the current device memory
2929 to local memory for data not in shared memory. For data in shared memory, no action is taken. An
2930 **update** directive with the **self** clause is equivalent to a call to the **acc_update_self** routine,
2931 described in Section 3.2.20.

2932 host clause

2933 The **host** clause is a synonym for the **self** clause.

2934 device clause

2935 The **device** clause specifies that the *vars* in *var-list* are to be copied from local memory to the cur-
2936 rent device memory, for data not in shared memory. For data in shared memory, no action is taken.
2937 An **update** directive with the **device** clause is equivalent to a call to the **acc_update_device**
2938 routine, described in Section 3.2.20.

2939 if clause

2940 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
2941 perform the updates unconditionally. When an **if** clause appears, the implementation will generate
2942 code to conditionally perform the updates only when the *condition* evaluates to *true*.

2943 async clause

2944 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2945 wait clause

2946 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2947 if_present clause

2948 When an **if_present** clause appears on the directive, no action is taken for a *var* which appears
2949 in *var-list* that is not present in the current device memory.

2950 Restrictions

- 2951 • The **update** directive is executable. It must not appear in place of the statement following
2952 an *if*, *while*, *do*, *switch*, or *label* in C or C++, or in place of the statement following a logical
2953 *if* in Fortran.
- 2954 • If no **if_present** clause appears on the directive, each *var* in *var-list* must be present in
2955 the current device memory.
- 2956 • Only the **async** and **wait** clauses may follow a **device_type** clause.
- 2957 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
2958 value; in C or C++, the condition must evaluate to a scalar integer value.
- 2959 • Noncontiguous subarrays may appear. It is implementation-specific whether noncontiguous
2960 regions are updated by using one transfer for each contiguous subregion, or whether the non-
2961 contiguous data is packed, transferred once, and unpacked, or whether one or more larger
2962 subarrays (no larger than the smallest contiguous region that contains the specified subarray)
2963 are updated.
- 2964 • In C and C++, a member of a struct or class may appear, including a subarray of a member.
2965 Members of a subarray of struct or class type may not appear.
- 2966 • In C and C++, if a subarray notation is used for a struct member, subarray notation may not
2967 be used for any parent of that struct member.
- 2968 • In Fortran, members of variables of derived type may appear, including a subarray of a mem-
2969 ber. Members of subarrays of derived type may not appear.

- 2970 • In Fortran, if array or subarray notation is used for a derived type member, array or subarray
2971 notation may not be used for a parent of that derived type member.
- 2972 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **self**,
2973 **host**, and **device** clauses.

2974 Errors

- 2975 • An **acc_error_not_present** error is issued if no **if_present** clause appears and
2976 any *var* in a **device** or **self** clause is not present on the current device.
- 2977 • An **acc_error_partly_present** error is issued if part of *var* is present in the current
2978 device memory but all of *var* is not.
- 2979 • An **async** or **wait** clause can cause an error to be issued; see Sections 2.16.1 and 2.16.2.

2980 See Section 5.2.2.

2981 2.14.5 Wait Directive

2982 See Section 2.16 Asynchronous Behavior for more information.

2983 2.14.6 Enter Data Directive

2984 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

2985 2.14.7 Exit Data Directive

2986 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

2987 2.15 Procedure Calls in Compute Regions

2988 This section describes how routines are compiled for an accelerator and how procedure calls are
2989 compiled in compute regions. See Section 2.17.1 Optional Arguments for discussion of Fortran
2990 optional arguments in procedure calls inside compute regions.

2991 2.15.1 Routine Directive

2992 Summary

2993 The **routine** directive is used to tell the compiler to compile the definition for a procedure, such
2994 as a function or C++ lambda, for an accelerator as well as for the host. The **routine** directive is
2995 also used to tell the compiler the attributes of the procedure when called on the accelerator.

2996 Syntax

2997 In C and C++, the syntax of the **routine** directive is:

```
2998 #pragma acc routine clause-list new-line
2999 #pragma acc routine( name ) clause-list new-line
```

3000 In C and C++, the **routine** directive without a name may appear immediately before a function
3001 definition, a function prototype, or a C++ lambda and applies to the function or C++ lambda. The
3002 **routine** directive with a name may appear anywhere that a function prototype is allowed and
3003 applies to the function or the C++ lambda in scope with that name. See Section A.3.4 for recom-
3004 mended diagnostics for a **routine** directive with a name.

3005 In Fortran the syntax of the **routine** directive is:

```
3006     !$acc routine clause-list
3007     !$acc routine ( name ) clause-list
```

3008 In Fortran, the **routine** directive without a name may appear within the specification part of a
 3009 subroutine or function definition, or within an interface body for a subroutine or function in an
 3010 interface block, and applies to the containing subroutine or function. The **routine** directive with
 3011 a name may appear in the specification part of a subroutine, function or module, and applies to the
 3012 named subroutine or function.

3013 The *clause* is one of the following:

```
3014     gang [ ( dim: int-expr ) ]
3015     worker
3016     vector
3017     seq
3018     bind( name )
3019     bind( string )
3020     device_type( device-type-list )
3021     nohost
```

3022 A **gang**, **worker**, **vector**, or **seq** clause specifies the *level of parallelism* in the routine.

3023 A procedure compiled with the **routine** directive for an accelerator is called an *accelerator rou-*
 3024 *tine*.

3025 If no explicit **routine** directive applies to a procedure that is called or whose address is accessed
 3026 in a compute region, and the procedure's definition appears in the program unit being compiled, then
 3027 the implementation applies an implicit **routine** directive with a **seq** clause to that procedure. A
 3028 C++ lambda's implicit **routine** directive also has a **nohost** clause if the lambda is defined in an
 3029 accelerator routine that has a **nohost** clause.

3030 When the implementation applies an implicit **routine** directive, it must recursively handle proce-
 3031 dure references in that accelerator routine.

3032 The implementation may apply predetermined **routine** directives with a **seq** clause to any pro-
 3033 cedures that it provides for an accelerator, such as those of base language standard libraries.

3034 **gang clause**

3035 The associated dimension is the value of the **dim** clause, if it appears, or is dimension one. The **dim**
 3036 argument must be a constant positive integer with value 1, 2, or 3. The **gang** clause with dimension
 3037 *d* specifies that the procedure contains, may contain, or may call another procedure that contains a
 3038 loop with a **gang** clause associated with dimension *d* or less.

3039 **worker clause**

3040 The **worker** clause specifies that the procedure contains, may contain, or may call another pro-
 3041 cedure that contains a loop with a **worker** clause, but does not contain nor does it call another
 3042 procedure that contains a loop with the **gang** clause. A loop in this procedure with an **auto** clause
 3043 may be selected by the compiler to execute in **worker** or **vector** mode. A call to this procedure
 3044 must appear in code that is executed in *worker-single* mode, though it may be in *gang-redundant*

3045 or *gang-partitioned* mode. For instance, a procedure with a **routine worker** directive may be
3046 called from within a loop that has the **gang** clause, but not from within a loop that has the **worker**
3047 clause.

3048 **vector clause**

3049 The **vector** clause specifies that the procedure contains, may contain, or may call another pro-
3050 cedure that contains a loop with the **vector** clause, but does not contain nor does it call another
3051 procedure that contains a loop with either a **gang** or **worker** clause. A loop in this procedure with
3052 an **auto** clause may be selected by the compiler to execute in **vector** mode, but not **worker**
3053 mode. A call to this procedure must appear in code that is executed in *vector-single* mode, though
3054 it may be in *gang-redundant* or *gang-partitioned* mode, and in *worker-single* or *worker-partitioned*
3055 mode. For instance, a procedure with a **routine vector** directive may be called from within
3056 a loop that has the **gang** clause or the **worker** clause, but not from within a loop that has the
3057 **vector** clause.

3058 **seq clause**

3059 The **seq** clause specifies that the procedure does not contain nor does it call another procedure that
3060 contains a loop with a **gang**, **worker**, or **vector** clause. A loop in this procedure with an **auto**
3061 clause will be executed in **seq** mode. A call to this procedure may appear in any mode.

3062 **bind clause**

3063 The **bind** clause specifies the name to use when calling the procedure on a device other than the
3064 host. If the name is specified as an identifier, it is called as if that name were specified in the
3065 language being compiled. If the name is specified as a string, the string is used for the procedure
3066 name unmodified. A **bind** clause on a procedure definition behaves as if it had appeared on a
3067 declaration by changing the name used to call the procedure on a device other than the host; however,
3068 the procedure is not compiled for the device with either the original name or the name in the **bind**
3069 clause.

3070 If there is both a Fortran **bind** and an acc **bind** clause for a procedure definition then a call on the
3071 host will call the Fortran bound name and a call on another device will call the name in the **bind**
3072 clause.

3073 **device_type clause**

3074 The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

3075 **nohost clause**

3076 The **nohost** clause tells the compiler not to compile a version of this procedure for the host.

3077 **Restrictions**

- 3078 • Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device_type**
3079 clause.
- 3080 • Exactly one of the **gang**, **worker**, **vector**, or **seq** clauses must appear.
- 3081 • In C and C++, function static variables are not supported in functions to which a **routine**
3082 directive applies.

- 3083 • In Fortran, variables with the *save* attribute, either explicitly or implicitly, are not supported
3084 in subprograms to which a **routine** directive applies.
- 3085 • A call to a procedure with a **nohost** clause must not appear in a compute construct that is
3086 compiled for the host. See examples below.
- 3087 • If a call to a procedure with a **nohost** clause appears in another procedure but outside any
3088 compute construct, that other procedure must also have a **nohost** clause.
- 3089 • A call to a procedure with a **gang(dim:d)** clause must appear in code that is executed
3090 in *gang-redundant* mode in all dimensions *d* and lower. For instance, a procedure with a
3091 **gang(dim:2)** clause may not be called from within a loop that has a **gang(dim:1)**
3092 or a **gang(dim:2)** clause. The user needs to ensure that a call to a procedure with a
3093 **gang(dim:d)** clause, when present in a region executing in *GR_e* or *GPe* mode with $e > d$
3094 and called by a gang along dimension *e*, is executed by all of its corresponding gangs along
3095 dimension *d*.
- 3096 • A **bind** clause may not bind to a routine name that has a visible **bind** clause.
- 3097 • If a procedure has a **bind** clause on both the declaration and the definition then they both
3098 must bind to the same name.
- 3099 • In C and C++, a definition or use of a procedure must appear within the scope of at least
3100 one explicit and applying **routine** directive if any appears in the same compilation unit.
3101 An explicit **routine** directive's scope is from the directive to the end of the compilation
3102 unit. If the **routine** directive appears in the member list of a C++ class, then its scope also
3103 extends in the same manner as any class member's scope (e.g., it includes the bodies of all
3104 other member functions).

Examples

- 3108 • A function, such as **f** below, requires a **nohost** clause if it contains accelerator-specific code
3109 that cannot be compiled for the host. By default, some implementations compile all compute
3110 constructs for the host in addition to accelerators. In that case, a call to **f** must not appear in
3111 any compute construct or compilation will fail. However, **f** can appear in the **bind** clause of
3112 another function, such as **g** below, that does not have a **nohost** clause, and a call to **g** can
3113 appear in a compute construct. Thus, **g** is called when the compute construct is compiled for
3114 the host, and **f** is called when the compute construct is compiled for accelerators.

```

3115     #pragma acc routine seq nohost
3116     void f() { /*accelerator implementation*/ }
3117
3118     #pragma acc routine seq bind(f)
3119     void g() { /*host implementation*/ }
3120
3121     void h() {
3122         #pragma acc parallel
3123         g();
3124     }

```

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

```

3129     void f();
3130     void scopeA() {
3131         #pragma acc parallel
3132         f(); // nonconforming
3133     }
3134     // The routine directive's scope is not f's full scope.
3135     // Instead, it starts at the routine directive.
3136     #pragma acc routine(f) gang
3137     void scopeB() {
3138         #pragma acc parallel
3139         f(); // conforming
3140     }
3141     void f() {} // conforming

```

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

```

3146     class A {
3147         void f() {
3148             #pragma acc parallel
3149             g(); // conforming
3150         }
3151         #pragma acc routine gang
3152         void g();
3153     };

```

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

```

3158     class A {
3159         #pragma acc routine(f) gang
3160         void f() {} // conforming
3161         void g() {} // nonconforming
3162         #pragma acc routine(g) gang
3163     };

```

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

```

3169     namespace A {
3170         void f();
3171         namespace B {

```

```

3172     #pragma acc routine(f) gang // applies to A::f
3173   }
3174 }
3175 void g() {
3176   #pragma acc parallel
3177   A::f(); // conforming
3178 }
3179 void h() {
3180   using A::f;
3181   #pragma acc parallel
3182   f(); // conforming
3183 }
3184 namespace C {
3185   void f();
3186   using namespace A::B;
3187   void i() {
3188     #pragma acc parallel
3189     f(); // nonconforming
3190   }
3191 }

```

3192



3193 2.15.2 Global Data Access

3194 C or C++ global, file static, or *extern* variables or array, and Fortran *module* or *common block* vari-
3195 ables or arrays, that are used in accelerator routines must appear in a declare directive in a **create**,
3196 **copyin**, **device_resident** or **link** clause. If the data appears in a **device_resident**
3197 clause, the **routine** directive for the procedure must include the **nohost** clause. If the data ap-
3198 pears in a **link** clause, that data must have an active accelerator data lifetime by virtue of appearing
3199 in a data clause for a **data** construct, compute construct, or **enter data** directive.

3200 2.16 Asynchronous Behavior

3201 This section describes the **async** clause, the **wait** clause, the **wait** directive, and the behavior of
3202 programs that use asynchronous data movement, compute regions, and asynchronous API routines.

3203 In this section and throughout the specification, the term *async-argument* means a nonnegative
3204 scalar integer expression (*int* for C or C++, *integer* for Fortran), or one of the special values
3205 **acc_async_noval** or **acc_async_sync**, as defined in the C header file and the Fortran
3206 **openacc** module. The special values are negative values, so as not to conflict with a user-specified
3207 nonnegative *async-argument*. An *async-argument* is used in **async** clauses, **wait** clauses, **wait**
3208 directives, and as an argument to various runtime routines.

3209 The *async-value* of an *async-argument* is

- 3210 • **acc_async_sync** if *async-argument* has a value equal to the special value **acc_async_sync**,
- 3211 • the value of *acc-default-async-var* if *async-argument* has a value equal to the special value
- 3212 **acc_async_noval**,
- 3213 • the value of the *async-argument*, if it is nonnegative,
- 3214 • implementation-defined, otherwise.

3215 The *async-value* is used to select the activity queue to which the clause or directive or API routine
 3216 refers. The properties of the current device and the implementation will determine how many actual
 3217 activity queues are supported, and how the *async-value* is mapped onto the actual activity queues.
 3218 Two asynchronous operations on the same device with the same *async-value* will be enqueued
 3219 onto the same activity queue, and therefore will be executed on the device in the order they are
 3220 encountered by the local thread. Two asynchronous operations with different *async-values* may be
 3221 enqueued onto different activity queues, and therefore may be executed on the device in either order
 3222 or concurrently relative to each other. If there are two or more host threads executing and sharing the
 3223 same device, asynchronous operations on any thread with the same *async-value* will be enqueued
 3224 onto the same activity queue. If the threads are not synchronized with respect to each other, the
 3225 operations may be enqueued in either order and therefore may execute on the device in either order.
 3226 Asynchronous operations enqueued to different devices may execute in any order or may execute
 3227 concurrently, regardless of the *async-value* used for each.

3228 If a compute construct, data directive, or runtime API call has an *async-value* of **acc_async_sync**,
 3229 the associated operations are executed on the activity queue associated with the *async-value*
 3230 **acc_async_sync**, and the local thread will wait until the associated operations have completed
 3231 before executing the code following the construct or directive. If a **data** construct has an *async-*
 3232 *value* of **acc_async_sync**, the associated operations are executed on the activity queue associ-
 3233 ated with the *async-value* **acc_async_sync**, and the local thread will wait until the associated
 3234 operations that occur upon entry of the construct have completed before executing the code of the
 3235 construct's structured block or block construct, and after that, will wait until the associated opera-
 3236 tions that occur upon exit of the construct have completed before executing the code following the
 3237 construct.

3238 If a compute construct, data directive, or runtime API call has an *async-value* other than
 3239 **acc_async_sync**, the associated operations are executed on the activity queue associated with
 3240 that *async-value* and the associated operations may be processed asynchronously while the local
 3241 thread continues executing the code following the construct or directive. If a **data** construct has an
 3242 *async-value* other than **acc_async_sync**, the associated operations are executed on the activity
 3243 queue associated with that *async-value*, and the associated operations that occur upon entry of the
 3244 construct may be processed asynchronously while the local thread continues executing the code
 3245 of the construct's structured block or block construct, and after that, the associated operations that
 3246 occur upon exit of the construct may be processed asynchronously while the local thread continues
 3247 executing the code following the construct.

3248 In this section and throughout the specification, the term *wait-argument*, means:

3249 [**devnum** : *int-expr* :] [**queues** :] *async-argument-list*

3250 If a **devnum** modifier appears in the *wait-argument* then the associated device is the device with
 3251 that device number of the current device type. If no **devnum** modifier appears then the associated
 3252 device is the current device.

3253 Each *async-argument* is associated with an *async-value*. The *async-values* select the associated
 3254 activity queue or queues on the associated device. If there is no *async-argument-list*, the associated
 3255 activity queues are all activity queues for the associated device.

3256 The **queues** modifier within a *wait-argument* is optional to improve clarity of the expression list.

2.16.1 `async` clause

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

Errors

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

See Section 5.2.2.

2.16.2 `wait` clause

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

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

Errors

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

See Section 5.2.2.

2.16.3 `Wait Directive`

Summary

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

Syntax

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

3296 **#pragma acc wait** [(*wait-argument*)] [*clause-list*] *new-line*

3297 In Fortran the syntax of the **wait** directive is:

3298 **!\$acc wait** [(*wait-argument*)] [*clause-list*]

3299 where *clause* is:

3300 **async** [(*async-argument*)]

3301 **if** (*condition*)

3302 If it appears, the *wait-argument* is as defined in Section 2.16, and the associated device and activity
3303 queues are as specified in the *wait-argument*. If there is no *wait-argument* clause, the associated
3304 device is the current device and associated activity queues are all activity queues.

3305 If there is no **async** clause, the local thread will wait until all operations enqueued by this thread
3306 onto each of the associated device activity queues for the associated device have completed. There
3307 is no guarantee that all the asynchronous operations initiated by other threads onto those queues will
3308 have completed without additional synchronization with those threads.

3309 If there is an **async** clause, no new operation may be launched or executed on the activity queue
3310 associated with the *async-argument* on the current device until all operations enqueued up to this
3311 point by this thread on the activity queues associated with the *wait-argument* have completed. **Note:**
3312 One legal implementation is for the local thread to wait for all the associated activity queues; another
3313 legal implementation is for the thread to enqueue a synchronization operation in such a way that
3314 no new operation will start until the operations enqueued on the associated activity queues have
3315 completed.

3316 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
3317 perform the wait operation unconditionally. When an **if** clause appears, the implementation will
3318 generate code to conditionally perform the wait operation only when the *condition* evaluates to *true*.

3319 A **wait** directive is functionally equivalent to a call to one of the **acc_wait**, **acc_wait_async**,
3320 **acc_wait_all**, or **acc_wait_all_async** runtime API routines, as described in Sections 3.2.10
3321 and 3.2.11.

3322 Errors

- 3323 • An **acc_error_device_unavailable** error is issued if a **devnum** modifier appears
3324 and the *int-expr* is not a valid device number.
- 3325 • An **acc_error_invalid_async** error is issued if a **queues** modifier or no modifier
3326 appears and any value in the associated list is not a valid *async-argument*.

3327 See Section 5.2.2.

3328 2.17 Fortran Specific Behavior

3329 2.17.1 Optional Arguments

3330 This section refers to the Fortran intrinsic function **PRESENT**. A call to the Fortran intrinsic function
3331 **PRESENT(arg)** returns **.true.**, if **arg** is an optional dummy argument and an actual argument
3332 for **arg** was present in the argument list of the call site. This should not be confused with the
3333 OpenACC **present** data clause.

3334 The appearance of a Fortran optional argument **arg** as a *var* in any of the following clauses has no
3335 effect at runtime if **PRESENT (arg)** is **.false.**:

- 3336 • in data clauses on **compute** and **data** constructs;
- 3337 • in data clauses on **enter data** and **exit data** directives;
- 3338 • in data and **device_resident** clauses on **declare** directives;
- 3339 • in **use_device** clauses on **host_data** directives;
- 3340 • in **self**, **host**, and **device** clauses on **update** directives.

3341 The appearance of a Fortran optional argument **arg** in the following situations may result in unde-
3342 fined behavior if **PRESENT (arg)** is **.false.** when the associated construct is executed:

- 3343 • as a *var* in **private**, **firstprivate**, and **reduction** clauses;
- 3344 • as a *var* in **cache** directives;
- 3345 • as part of an expression in any clause or directive.

3346 A call to the Fortran intrinsic function **PRESENT** behaves the same way in a **compute** construct or
3347 an accelerator routine as on the host. The function call **PRESENT (arg)** must return the same value
3348 in a **compute** construct as **PRESENT (arg)** would outside of the **compute** construct. If a Fortran
3349 optional argument **arg** appears as an actual argument in a procedure call in a **compute** construct
3350 or an accelerator routine, and the associated dummy argument **subarg** also has the **optional**
3351 attribute, then **PRESENT (subarg)** returns the same value as **PRESENT (subarg)** would when
3352 executed on the host.

3353 2.17.2 Do Concurrent Construct

3354 This section refers to the Fortran **do concurrent** construct that is a form of **do** construct. When
3355 **do concurrent** appears without a **loop** construct in a **kernels** construct it is treated as if it is
3356 annotated with **loop auto**. If it appears in a **parallel** construct or an accelerator routine then
3357 it is treated as if it is annotated with **loop independent**.

3. Runtime Library

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

This chapter has two sections:

- Runtime library definitions
- Runtime library routines

There are four categories of runtime routines:

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

3.1 Runtime Library Definitions

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

- The prototypes of all routines in the chapter.
- Any datatypes used in those prototypes, including an enumeration type to describe the supported device types.
- The values of `acc_async_noval`, `acc_async_sync`, and `acc_async_default`.

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

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

3391 Many of the routines accept or return a value corresponding to the type of device. In C and C++, the
 3392 datatype used for device type values is `acc_device_t`; in Fortran, the corresponding datatype
 3393 is `integer(kind=acc_device_kind)`. The possible values for device type are implemen-
 3394 tation specific, and are defined in the C or C++ include file `openacc.h` and the Fortran module
 3395 `openacc`. Five values are always supported: `acc_device_none`, `acc_device_default`,
 3396 `acc_device_host`, `acc_device_not_host`, and `acc_device_current`. For other val-
 3397 ues, look at the appropriate files included with the implementation, or read the documentation for
 3398 the implementation. The value `acc_device_default` will never be returned by any function;
 3399 its use as an argument will tell the runtime library to use the default device type for that implemen-
 3400 tation.

3401 3.2 Runtime Library Routines

3402 In this section, for the C and C++ prototypes, pointers are typed `h_void*` or `d_void*` to desig-
 3403 nate a host memory address or device memory address, when these calls are executed on the host,
 3404 as if the following definitions were included:

```
3405     #define h_void void
3406     #define d_void void
```

3407 Many Fortran API bindings defined in this section rely on types defined in Fortran's `iso_c_binding`
 3408 module. It is implied that the `iso_c_binding` module is used in these bindings, even if not ex-
 3409 plicitly stated in the format section for that routine.

3410 Restrictions

3411 Except for `acc_on_device`, these routines are only available on the host.

3412 3.2.1 acc_get_num_devices

3413 Summary

3414 The `acc_get_num_devices` routine returns the number of available devices of the given type.

3415 Format

3416 C or C++:

```
3417     int acc_get_num_devices(acc_device_t dev_type);
```

3418 Fortran:

```
3419     integer function acc_get_num_devices(dev_type)
3420     integer(acc_device_kind) :: dev_type
```

3421 Description

3422 The `acc_get_num_devices` routine returns the number of available devices of device type
 3423 `dev_type`. If device type `dev_type` is not supported or no device of `dev_type` is available,
 3424 this routine returns zero.

3425 3.2.2 acc_set_device_type

3426 Summary

3427 The `acc_set_device_type` routine tells the runtime which type of device to use when exe-
 3428 cuting a compute region and sets the value of `acc-current-device-type-var`. This is useful when the
 3429 implementation allows the program to be compiled to use more than one type of device.

3430 **Format**

3431 C or C++:

3432 `void acc_set_device_type(acc_device_t dev_type);`

3433 Fortran:

3434 `subroutine acc_set_device_type(dev_type)`3435 `integer(acc_device_kind) :: dev_type`3436 **Description**

3437 A call to `acc_set_device_type` is functionally equivalent to a `set device_type (dev_type)`
 3438 directive, as described in Section 2.14.3. This routine tells the runtime which type of device to use
 3439 among those available and sets the value of *acc-current-device-type-var* for the current thread to
 3440 `dev_type`.

3441 **Restrictions**

- 3442 • If some compute regions are compiled to only use one device type, the result of calling this
 3443 routine with a different device type may produce undefined behavior.

3444 **Errors**

- 3445 • An `acc_error_device_type_unavailable` error is issued if device type `dev_type`
 3446 is not supported or no device of `dev_type` is available.

3447 See Section 5.2.2.

3448 **3.2.3 acc_get_device_type**3449 **Summary**

3450 The `acc_get_device_type` routine returns the value of *acc-current-device-type-var*, which is
 3451 the device type of the current device. This is useful when the implementation allows the program to
 3452 be compiled to use more than one type of device.

3453 **Format**

3454 C or C++:

3455 `acc_device_t acc_get_device_type(void);`

3456 Fortran:

3457 `function acc_get_device_type()`3458 `integer(acc_device_kind) :: acc_get_device_type`3459 **Description**

3460 The `acc_get_device_type` routine returns the value of *acc-current-device-type-var* for the
 3461 current thread to tell the program what type of device will be used to run the next compute region, if
 3462 one has been selected. The device type may have been selected by the program with a runtime API
 3463 call or a directive, by an environment variable, or by the default behavior of the implementation; see
 3464 the table in Section 2.3.1.

3465 **Restrictions**

- 3466 • If the device type has not yet been selected, the value `acc_device_none` may be returned.

3467 **3.2.4 acc_set_device_num**

3468 **Summary**

3469 The `acc_set_device_num` routine tells the runtime which device to use and sets the value of
3470 `acc-current-device-num-var`.

3471 **Format**

3472 C or C++:

```
3473     void acc_set_device_num(int dev_num, acc_device_t dev_type);
```

3474 Fortran:

```
3475     subroutine acc_set_device_num(dev_num, dev_type)
3476         integer :: dev_num
3477         integer(acc_device_kind) :: dev_type
```

3478 **Description**

3479 A call to `acc_set_device_num` is functionally equivalent to a `set device_type (dev_type)`
3480 `device_num (dev_num)` directive, as described in Section 2.14.3. This routine tells the runtime
3481 which device to use among those available of the given type for compute or data regions in the cur-
3482 rent thread and sets the value of `acc-current-device-num-var` to `dev_num`. If the value of `dev_num`
3483 is negative, the runtime will revert to its default behavior, which is implementation-defined. If the
3484 value of the `dev_type` is zero, the selected device number will be used for all device types. Calling
3485 `acc_set_device_num` implies a call to `acc_set_device_type (dev_type)`.

3486 **Errors**

- 3487 • An `acc_error_device_type_unavailable` error is issued if device type `dev_type`
3488 is not supported or no device of `dev_type` is available.
- 3489 • An `acc_error_device_unavailable` error is issued if the value of `dev_num` is not
3490 a valid device number.

3491 See Section 5.2.2.

3492 **3.2.5 acc_get_device_num**

3493 **Summary**

3494 The `acc_get_device_num` routine returns the value of `acc-current-device-num-var` for the cur-
3495 rent thread.

3496 **Format**

3497 C or C++:

```
3498     int acc_get_device_num(acc_device_t dev_type);
```

3499 Fortran:

```
3500     integer function acc_get_device_num(dev_type)
3501         integer(acc_device_kind) :: dev_type
```

3502 **Description**

3503 The `acc_get_device_num` routine returns the value of `acc-current-device-num-var` for the cur-
3504 rent thread. If there are no devices of device type `dev_type` or if device type `dev_type` is not
3505 supported, this routine returns `-1`.

3.2.6 `acc_get_property`

Summary

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

Format

C or C++:

```

size_t acc_get_property(int dev_num,
                        acc_device_t dev_type,
                        acc_device_property_t property);

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

```

Fortran:

```

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

```

Description

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

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

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

An implementation may support additional properties for some devices.

3530 **Restrictions**

- 3531 • **acc_get_property** will return 0 and **acc_get_property_string** will return a null
 3532 pointer (in C or C++) or a blank string (in Fortran) in the following cases:
- 3533 – If device type **dev_type** is not supported or no device of **dev_type** is available.
 - 3534 – If the value of **dev_num** is not a valid device number for device type **dev_type**.
 - 3535 – If the value of **property** is not one of the known values for that query routine, or that
 3536 property has no value for the specified device.

3537 **3.2.7 acc_init**3538 **Summary**

3539 The **acc_init** and **acc_init_device** routines initialize the runtime for the specified device
 3540 type and device number. This can be used to isolate any initialization cost from the computational
 3541 cost, such as when collecting performance statistics.

3542 **Format**

3543 C or C++:

```
3544 void acc_init(acc_device_t dev_type);
3545 void acc_init_device(int dev_num, acc_device_t dev_type);
```

3546 Fortran:

```
3547 subroutine acc_init(dev_type)
3548 subroutine acc_init_device(dev_num, dev_type)
3549 integer :: dev_num
3550 integer(acc_device_kind) :: dev_type
```

3551 **Description**

3552 A call to **acc_init** or **acc_init_device** is functionally equivalent to an **init** directive with
 3553 matching **dev_type** and **dev_num** arguments, as described in Section 2.14.1. **dev_type** must
 3554 be one of the defined accelerator types. **dev_num** must be a valid device number of the device type
 3555 **dev_type**. These routines also implicitly call **acc_set_device_type(dev_type)**. In the
 3556 case of **acc_init_device**, **acc_set_device_num(dev_num)** is also called.

3557 If a program initializes one or more devices without an intervening **shutdown** directive or
 3558 **acc_shutdown** call to shut down those same devices, no action is taken.

3559 **Errors**

- 3560 • An **acc_error_device_type_unavailable** error is issued if device type **dev_type**
 3561 is not supported or no device of **dev_type** is available.
- 3562 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3563 number.

3564 See Section 5.2.2.

3565 **3.2.8 acc_shutdown**

3566 **Summary**

3567 The **acc_shutdown** and **acc_shutdown_device** routines shut down the connection to spec-
 3568 ified devices and free up any related resources in the runtime. This ends all data lifetimes in device
 3569 memory for the device or devices that are shut down, which effectively sets structured and dynamic
 3570 reference counters to zero.

3571 **Format**

3572 C or C++:

```
3573     void acc_shutdown(acc_device_t dev_type);
3574     void acc_shutdown_device(int dev_num, acc_device_t dev_type);
```

3575 Fortran:

```
3576     subroutine acc_shutdown(dev_type)
3577     subroutine acc_shutdown_device(dev_num, dev_type)
3578         integer :: dev_num
3579         integer(acc_device_kind) :: dev_type
```

3580 **Description**

3581 A call to **acc_shutdown** or **acc_shutdown_device** is functionally equivalent to a **shutdown**
 3582 directive, with matching **dev_type** and **dev_num** arguments, as described in Section 2.14.2.
 3583 **dev_type** must be one of the defined accelerator types. **dev_num** must be a valid device number
 3584 of the device type **dev_type**. **acc_shutdown** routine disconnects the program from all devices
 3585 of device type **dev_type**. The **acc_shutdown_device** routine disconnects the program from
 3586 **dev_num** of type **dev_type**. Any data that is present in the memory of a device that is shut down
 3587 is immediately deallocated.

3588 **Restrictions**

- 3589 • This routine may not be called while a compute region is executing on a device of type
 3590 **dev_type**.
- 3591 • If the program attempts to execute a compute region on a device or to access any data in the
 3592 memory of a device that was shut down, the behavior is undefined.
- 3593 • If the program attempts to shut down the **acc_device_host** device type, the behavior is
 3594 undefined.

3595 **Errors**

- 3596 • An **acc_error_device_type_unavailable** error is issued if device type **dev_type**
 3597 is not supported or no device of **dev_type** is available.
- 3598 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3599 number.
- 3600 • An **acc_error_device_shutdown** error is issued if there is an error shutting down the
 3601 device.

3602 See Section 5.2.2.

3603 **3.2.9 acc_async_test**3604 **Summary**

3605 The **acc_async_test** routines test for completion of all associated asynchronous operations for
 3606 a single specified async queue or for all async queues on the current device or on a specified device.

3607 **Format**

3608 C or C++:

```

3609     int acc_async_test(int wait_arg);
3610     int acc_async_test_device(int wait_arg, int dev_num);
3611     int acc_async_test_all(void);
3612     int acc_async_test_all_device(int dev_num);

```

3613 Fortran:

```

3614     logical function acc_async_test(wait_arg)
3615     logical function acc_async_test_device(wait_arg, dev_num)
3616     logical function acc_async_test_all()
3617     logical function acc_async_test_all_device(dev_num)
3618     integer(acc_handle_kind) :: wait_arg
3619     integer :: dev_num

```

3620 **Description**

3621 **wait_arg** must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior. **dev_num**
 3622 must be a valid device number of the current device type.

3623 The behavior of the **acc_async_test** routines is:

- 3624 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3625 • If any asynchronous operations initiated by this host thread on device **dev_num** either on
 3626 async queue **wait_arg** (if there is a **wait_arg** argument), or on any async queue (if there
 3627 is no **wait_arg** argument) have not completed, a call to the routine returns *false*.
- 3628 • If all such asynchronous operations have completed, or there are no such asynchronous op-
 3629 erations, a call to the routine returns *true*. A return value of *true* is no guarantee that asyn-
 3630 chronous operations initiated by other host threads have completed.

3631 **Errors**

- 3632 • An **acc_error_invalid_async** error is issued if **wait_arg** is not a valid *async-*
 3633 *argument* value.
- 3634 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3635 number.

3636 See Section 5.2.2.

3637 **3.2.10 acc_wait**3638 **Summary**

3639 The **acc_wait** routines wait for completion of all associated asynchronous operations on a single
 3640 specified async queue or on all async queues on the current device or on a specified device.

3641 **Format**

3642 C or C++:

```

3643     void acc_wait(int wait_arg);
3644     void acc_wait_device(int wait_arg, int dev_num);
3645     void acc_wait_all(void);
3646     void acc_wait_all_device(int dev_num);

```

3647 Fortran:

```
3648     subroutine acc_wait(wait_arg)
3649     subroutine acc_wait_device(wait_arg, dev_num)
3650     subroutine acc_wait_all()
3651     subroutine acc_wait_all_device(dev_num)
3652     integer(acc_handle_kind) :: wait_arg
3653     integer :: dev_num
```

3654 Description

3655 A call to an **acc_wait** routine is functionally equivalent to a **wait** directive as follows, see Sec-
3656 tion 2.16.3:

- 3657 • **acc_wait** to a **wait(wait_arg)** directive.
- 3658 • **acc_wait_device** to a **wait(devnum:dev_num, queues:wait_arg)** directive.
- 3659 • **acc_wait_all** to a **wait** directive with no *wait-argument*.
- 3660 • **acc_wait_all_device** to a **wait(devnum:dev_num)** directive.

3661 **wait_arg** must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior. **dev_num**
3662 must be a valid device number of the current device type.

3663 The behavior of the **acc_wait** routines is:

- 3664 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3665 • The routine will not return until all asynchronous operations initiated by this host thread on
3666 device **dev_num** either on async queue **wait_arg** (if there is a **wait_arg** argument) or
3667 on all async queues (if there is no **wait_arg** argument) have completed.
- 3668 • If two or more threads share the same accelerator, there is no guarantee that matching asyn-
3669 chronous operations initiated by other threads have completed.

3670 For compatibility with OpenACC version 1.0, **acc_wait** may also be spelled **acc_async_wait**,
3671 and **acc_wait_all** may also be spelled **acc_async_wait_all**.

3672 Errors

- 3673 • An **acc_error_invalid_async** error is issued if **wait_arg** is not a valid *async-*
3674 *argument* value.
- 3675 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
3676 number.

3677 See Section 5.2.2.

3678 3.2.11 acc_wait_async

3679 Summary

3680 The **acc_wait_async** routines enqueue a wait operation on one async queue of the current
3681 device or a specified device for the operations previously enqueued on a single specified async
3682 queue or on all other async queues.

3683 **Format**

C or C++:

```

3683 void acc_wait_async(int wait_arg, int async_arg);
3684 void acc_wait_device_async(int wait_arg, int async_arg,
3685                             int dev_num);
3686 void acc_wait_all_async(int async_arg);
3686 void acc_wait_all_device_async(int async_arg, int dev_num);

```

3687 Fortran:

```

3688 subroutine acc_wait_async(wait_arg, async_arg)
3689 subroutine acc_wait_device_async(wait_arg, async_arg, dev_num)
3690 subroutine acc_wait_all_async(async_arg)
3691 subroutine acc_wait_all_device_async(async_arg, dev_num)
3692 integer(acc_handle_kind) :: wait_arg, async_arg
3693 integer :: dev_num

```

3694 **Description**

3695 A call to an **acc_wait_async** routine is functionally equivalent to a **wait async (async_arg)**
 3696 directive as follows, see Section 2.16.3:

- 3697 • A call to **acc_wait_async** is functionally equivalent to a **wait (wait_arg)**
 3698 **async (async_arg)** directive.
- 3699 • A call to **acc_wait_device_async** is functionally equivalent to a **wait (devnum:**
 3700 **dev_num, queues:wait_arg) async (async_arg)** directive.
- 3701 • A call to **acc_wait_all_async** is functionally equivalent to a **wait async (async_arg)**
 3702 directive with no *wait-argument*.
- 3703 • A call to **acc_wait_all_device_async** is functionally equivalent to a
 3704 **wait (devnum:dev_num) async (async_arg)** directive.

3705 **async_arg** and **wait_arg** must be *async-arguments*, as defined in
 3706 Section 2.16 Asynchronous Behavior. **dev_num** must be a valid device number of the current
 3707 device type.

3708 The behavior of the **acc_wait_async** routines is:

- 3709 • If there is no **dev_num** argument, it is treated as if **dev_num** is the current device number.
- 3710 • The routine will enqueue a wait operation on the async queue associated with **async_arg**
 3711 for the current device which will wait for operations initiated on the async queue **wait_arg**
 3712 of device **dev_num** (if there is a **wait_arg** argument), or for each async queue of device
 3713 **dev_num** (if there is no **wait_arg** argument).

3714 See Section 2.16 Asynchronous Behavior for more information.

3715 **Errors**

- 3716 • An **acc_error_invalid_async** error is issued if either **async_arg** or **wait_arg** is
 3717 not a valid *async-argument* value.
- 3718 • An **acc_error_device_unavailable** error is issued if **dev_num** is not a valid device
 3719 number.

3720 See Section 5.2.2.

3721 **3.2.12 acc_wait_any**3722 **Summary**

3723 The `acc_wait_any` and `acc_wait_any_device` routines wait for any of the specified asyn-
 3724 chronous queues to complete all pending operations on the current device or the specified device
 3725 number, respectively. Both routines return the queue's index in the provided array of asynchronous
 3726 queues.

3727 **Format**

3728 C or C++:

```
3729     int acc_wait_any(int count, int wait_arg[]);
3730     int acc_wait_any_device(int count, int wait_arg[], int dev_num);
```

3731 Fortran:

```
3732     integer function acc_wait_any(count, wait_arg)
3733     integer function acc_wait_any_device(count, wait_arg, dev_num)
3734     integer :: count, dev_num
3735     integer(acc_handle_kind) :: wait_arg(count)
```

3736 **Description**

3737 `wait_arg` is an array of *async-arguments* as defined in Section 2.16 and `count` is a nonneg-
 3738 ative integer indicating the array length. If there is no `dev_num` argument, it is treated as if
 3739 `dev_num` is the current device number. Otherwise, `dev_num` must be a valid device number
 3740 of the current device type. A call to any of these routines returns an index `i` associated with
 3741 a `wait_arg[i]` that is not `acc_async_sync` and meets the conditions that would evalu-
 3742 ate `acc_async_test_device(wait_arg[i], dev_num)` to *true*. If all the elements in
 3743 `wait_arg` are equal to `acc_async_sync` or `count` is equal to 0, these routines return -1.
 3744 Otherwise, the return value is an integer in the range of $0 \leq i < \text{count}$ in C or C++ and
 3745 $1 \leq i \leq \text{count}$ in Fortran.

3746 **Errors**

- 3747 • An `acc_error_invalid_argument` error is issued if `count` is a negative number.
- 3748 • An `acc_error_invalid_async` error is issued if any element encountered in `wait_arg`
 3749 is not a valid *async-argument* value.
- 3750 • An `acc_error_device_unavailable` error is issued if `dev_num` is not a valid device
 3751 number.

3752 See Section 5.2.2.

3753 **3.2.13 acc_get_default_async**3754 **Summary**

3755 The `acc_get_default_async` routine returns the value of *acc-default-async-var* for the cur-
 3756 rent thread.

3757 **Format**

3758 C or C++:

```
3759     int acc_get_default_async(void);
```

3760 Fortran:

```
3761     function acc_get_default_async()
3762     integer(acc_handle_kind) :: acc_get_default_async
```

3763 Description

3764 The **acc_get_default_async** routine returns the value of *acc-default-async-var* for the cur-
3765 rent thread, which is the asynchronous queue used when an **async** clause appears without an
3766 *async-argument* or with the value **acc_async_noval**.

3767 3.2.14 acc_set_default_async

3768 Summary

3769 The **acc_set_default_async** routine tells the runtime which asynchronous queue to use
3770 when an **async** clause appears with no queue argument.

3771 Format

3772 C or C++:

```
3773     void acc_set_default_async(int async_arg);
```

3774 Fortran:

```
3775     subroutine acc_set_default_async(async_arg)
3776     integer(acc_handle_kind) :: async_arg
```

3777 Description

3778 A call to **acc_set_default_async** is functionally equivalent to a **set default_async(async_arg)**
3779 directive, as described in Section 2.14.3. This **acc_set_default_async** routine tells the
3780 runtime to place any directives with an **async** clause that does not have an *async-argument* or
3781 with the special **acc_async_noval** value into the asynchronous activity queue associated with
3782 **async_arg** instead of the default asynchronous activity queue for that device by setting the value
3783 of *acc-default-async-var* for the current thread. The special argument **acc_async_default** will
3784 reset the default asynchronous activity queue to the initial value, which is implementation-defined.

3785 Errors

- 3786 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
3787 *argument* value.

3788 See Section 5.2.2.

3789 3.2.15 acc_on_device

3790 Summary

3791 The **acc_on_device** routine tells the program whether it is executing on a particular device.

3792 Format

3793 C or C++:

```
3794     int acc_on_device(acc_device_t dev_type);
```

3795 Fortran:

```
3796     logical function acc_on_device(dev_type)
3797     integer(acc_device_kind) :: dev_type
```


3798 **Description**

3799 The `acc_on_device` routine may be used to execute different paths depending on whether the
 3800 code is running on the host or on some accelerator. If the `acc_on_device` routine has a compile-
 3801 time constant argument, the call evaluates at compile time to a constant. `dev_type` must be one
 3802 of the defined accelerator types.

3803 The behavior of the `acc_on_device` routine is:

- 3804 • If `dev_type` is `acc_device_host`, then outside of a compute region or accelerator rou-
 3805 tine, or in a compute region or accelerator routine that is executed on the host CPU, a call to
 3806 this routine will evaluate to *true*; otherwise, it will evaluate to *false*.
- 3807 • If `dev_type` is `acc_device_not_host`, the result is the negation of the result with
 3808 argument `acc_device_host`.
- 3809 • If `dev_type` is an accelerator device type, then in a compute region or routine that is ex-
 3810 ecuted on a device of that type, a call to this routine will evaluate to *true*; otherwise, it will
 3811 evaluate to *false*.
- 3812 • The result with argument `acc_device_default` is undefined.

3813 **3.2.16 acc_malloc**3814 **Summary**

3815 The `acc_malloc` routine allocates space in the current device memory.

3816 **Format**

3817 C or C++:

```
3818     d_void* acc_malloc(size_t bytes);
```

3819 Fortran:

```
3820     type(c_ptr) function acc_malloc(bytes)
3821     integer(c_size_t), value :: bytes
```

3822 **Description**

3823 The `acc_malloc` routine may be used to allocate space in the current device memory. Pointers
 3824 assigned from this routine may be used in `deviceptr` clauses to tell the compiler that the pointer
 3825 target is resident on the device. In case of an allocation error or if `bytes` has the value zero,
 3826 `acc_malloc` returns a null pointer.

3827 **3.2.17 acc_free**3828 **Summary**

3829 The `acc_free` routine frees memory on the current device.

3830 **Format**

3831 C or C++:

```
3832     void acc_free(d_void* data_dev);
```

3833 Fortran:

```
3834     subroutine acc_free(data_dev)
3835     type(c_ptr), value :: data_dev
```

3836 **Description**

3837 The **acc_free** routine will free previously allocated space in the current device memory; **data_dev**
 3838 should be a pointer value that was returned by a call to **acc_malloc**. If **data_dev** is a null
 3839 pointer, no operation is performed.

3840 **3.2.18 acc_copyin and acc_create**3841 **Summary**

3842 The **acc_copyin** and **acc_create** routines test to see if the argument is in shared memory
 3843 or already present in the current device memory; if not, they allocate space in the current device
 3844 memory to correspond to the specified local memory, and the **acc_copyin** routines copy the data
 3845 to that device memory.

3846 **Format**

3847 C or C++:

```
3848     d_void* acc_copyin(h_void* data_arg, size_t bytes);
3849     d_void* acc_create(h_void* data_arg, size_t bytes);
3850
3851     void acc_copyin_async(h_void* data_arg, size_t bytes,
3852                          int async_arg);
3853     void acc_create_async(h_void* data_arg, size_t bytes,
3854                          int async_arg);
3855
```

3856 Fortran:

```
3857     subroutine acc_copyin(data_arg [, bytes])
3858     subroutine acc_create(data_arg [, bytes])
3859
3860     subroutine acc_copyin_async(data_arg [, bytes], async_arg)
3861     subroutine acc_create_async(data_arg [, bytes], async_arg)
3862
3863     type(*), dimension(..) :: data_arg
3864     integer :: bytes
3865     integer(acc_handle_kind) :: async_arg
3866
```

3866 **Description**

3867 A call to an **acc_copyin** or **acc_create** routine is similar to an **enter data** directive with
 3868 a **copyin** or **create** clause, respectively, as described in Sections 2.7.7 and 2.7.9, except that
 3869 no *attach* action is performed for a pointer reference. In C/C++, **data_arg** is a pointer to the
 3870 data, and **bytes** specifies the data size in bytes; the associated *data section* starts at the address
 3871 in **data_arg** and continues for **bytes** bytes. The synchronous routines return a pointer to the
 3872 allocated device memory, as with **acc_malloc**. In Fortran, two forms are supported. In the first,
 3873 **data_arg** is a variable or a contiguous array section; the associated *data section* starts at the
 3874 address of, and continues to the end of the variable or array section. In the second, **data_arg**
 3875 is a variable or array element and **bytes** is the length in bytes; the associated *data section* starts
 3876 at the address of the variable or array element and continues for **bytes** bytes. For the **_async**
 3877 versions of these routines, **async_arg** must be an *async-argument* as defined in Section 2.16
 3878 Asynchronous Behavior.

3879 The behavior of these routines for the associated *data section* is:

- 3880 • If the *data section* is in shared memory, no action is taken. The C/C++ synchronous **acc_copyin**
3881 and **acc_create** routines return the incoming pointer.
- 3882 • If the *data section* is present in the current device memory, the routines perform a *present increment*
3883 action with the dynamic reference counter. The C/C++ synchronous **acc_copyin** and
3884 **acc_create** routines return a pointer to the existing device memory.
- 3885 • Otherwise:
- 3886 – The **acc_copyin** routines perform a *copyin* action with the dynamic reference counter.
3887 – The **acc_create** routines perform a *create* action with the dynamic reference counter.
- 3888 The C/C++ synchronous **acc_copyin** and **acc_create** routines return a pointer to the
3889 newly allocated device memory.

3890 This data may be accessed using the **present** data clause. Pointers assigned from the C/C++
3891 synchronous **acc_copyin** and **acc_create** routines may be used in **deviceptr** clauses to
3892 tell the compiler that the pointer target is resident on the device.

3893 The synchronous versions will not return until the memory has been allocated and any data transfers
3894 are complete.

3895 The **_async** versions of these routines will perform any data transfers asynchronously on the async
3896 queue associated with **async_arg**. The routine may return before the data has been transferred;
3897 see Section 2.16 Asynchronous Behavior for more details. The data will be treated as present in
3898 the current device memory even if the data has not been allocated or transferred before the routine
3899 returns.

3900 For compatibility with OpenACC 2.0, **acc_present_or_copyin** and **acc_pcopyin** are al-
3901 ternate names for **acc_copyin**, and **acc_present_or_create** and **acc_pcreate** are al-
3902 ternate names for **acc_create**.

3903 Errors

- 3904 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
3905 **bytes** is nonzero.
- 3906 • An **acc_error_partly_present** error is issued if part of the *data section* is already
3907 present in the current device memory but all of the *data section* is not.
- 3908 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
3909 tion that is not contiguous (in Fortran).
- 3910 • An **acc_error_out_of_memory** error is issued if the accelerator device does not have
3911 enough memory for the data.
- 3912 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
3913 *argument* value.

3914 See Section 5.2.2.

3915 3.2.19 acc_copyout and acc_delete

3916 **Summary**

3917 The **acc_copyout** and **acc_delete** routines test to see if the argument is in shared memory;
 3918 if not, the argument must be present in the current device memory. The **acc_copyout** routines
 3919 copy data from device memory to the corresponding local memory, and both **acc_copyout** and
 3920 **acc_delete** routines deallocate that space from the device memory.

3921 **Format**

3922 C or C++:

```

3923     void acc_copyout(h_void* data_arg, size_t bytes);
3924     void acc_delete (h_void* data_arg, size_t bytes);
3925
3926     void acc_copyout_finalize(h_void* data_arg, size_t bytes);
3927     void acc_delete_finalize (h_void* data_arg, size_t bytes);
3928
3929     void acc_copyout_async(h_void* data_arg, size_t bytes,
3930                           int async_arg);
3931     void acc_delete_async (h_void* data_arg, size_t bytes,
3932                           int async_arg);
3933
3934     void acc_copyout_finalize_async(h_void* data_arg, size_t bytes,
3935                                    int async_arg);
3936     void acc_delete_finalize_async (h_void* data_arg, size_t bytes,
3937                                    int async_arg);
3938

```

3939 Fortran:

```

3940     subroutine acc_copyout (data_arg [, bytes])
3941     subroutine acc_delete (data_arg [, bytes])
3942
3943     subroutine acc_copyout_finalize (data_arg [, bytes])
3944     subroutine acc_delete_finalize (data_arg [, bytes])
3945
3946     subroutine acc_copyout_async (data_arg [, bytes], async_arg)
3947     subroutine acc_delete_async (data_arg [, bytes], async_arg)
3948
3949     subroutine acc_copyout_finalize_async (data_arg [, bytes], &
3950                                           async_arg)
3951     subroutine acc_delete_finalize_async (data_arg [, bytes], &
3952                                           async_arg)
3953
3954     type(*), dimension(..) :: data_arg
3955     integer :: bytes
3956     integer(acc_handle_kind) :: async_arg

```

3957 **Description**

3958 A call to an **acc_copyout** or **acc_delete** routine is similar to an **exit data** directive
 3959 with a **copyout** or **delete** clause, respectively, and a call to an **acc_copyout_finalize**
 3960 or **acc_delete_finalize** routine is similar to an **exit data finalize** directive with a
 3961 **copyout** or **delete** clause, respectively, as described in Section 2.7.8 and 2.7.11, except that no

3962 *detach* action is performed for a pointer reference. The arguments and the associated *data section*
 3963 are as for **acc_copyin**.

3964 The behavior of these routines for the associated *data section* is:

- 3965 • If the *data section* is in shared memory, no action is taken.
 - 3966 • If the dynamic reference counter for the *data section* is zero, no action is taken.
 - 3967 • Otherwise, the dynamic reference counter is updated:
 - 3968 – The **acc_copyout** and **acc_delete**) routines perform a *present decrement* action
 - 3969 with the dynamic reference counter.
 - 3970 – The **acc_copyout_finalize** or **acc_delete_finalize** routines set the dy-
 - 3971 namic reference counter to zero.
- 3972 If both reference counters are then zero:
- 3973 – The **acc_copyout** routines perform a *copyout* action.
 - 3974 – The **acc_delete** routines perform a *delete* action.

3975 The synchronous versions will not return until the data has been completely transferred and the
 3976 memory has been deallocated.

3977 The **_async** versions of these routines will perform any associated data transfers asynchronously
 3978 on the async queue associated with **async_arg**. The routine may return before the data has been
 3979 transferred or deallocated; see Section 2.16 Asynchronous Behavior for more details. Even if the
 3980 data has not been transferred or deallocated before the routine returns, the data will be treated as not
 3981 present in the current device memory if both reference counters are zero.

3982 Errors

- 3983 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
 3984 **bytes** is nonzero.
- 3985 • An **acc_error_not_present** error is issued if the *data section* is not in shared memory
 3986 and is not present in the current device memory.
- 3987 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
 3988 tion that is not contiguous (in Fortran).
- 3989 • An **acc_error_partly_present** error is issued if part of the *data section* is already
 3990 present in the current device memory but all of the *data section* is not.
- 3991 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
 3992 *argument* value.

3993 See Section 5.2.2.

3994 3.2.20 acc_update_device and acc_update_self

3995 Summary

3996 The **acc_update_device** and **acc_update_self** routines test to see if the argument is in
 3997 shared memory; if not, the argument must be present in the current device memory, and the routines

3998 update the data in device memory from the corresponding local memory (**acc_update_device**)
 3999 or update the data in local memory from the corresponding device memory (**acc_update_self**).

4000 **Format**

4001 C or C++:

```
4002     void acc_update_device(h_void* data_arg, size_t bytes);
4003     void acc_update_self  (h_void* data_arg, size_t bytes);
4004
4005     void acc_update_device_async(h_void* data_arg, size_t bytes,
4006                                 int async_arg);
4007     void acc_update_self_async (h_void* data_arg, size_t bytes,
4008                                 int async_arg);
4009
```

4010 Fortran:

```
4011     subroutine acc_update_device(data_arg [, bytes])
4012     subroutine acc_update_self  (data_arg [, bytes])
4013
4014     subroutine acc_update_device_async(data_arg [, bytes], async_arg)
4015     subroutine acc_update_self_async (data_arg [, bytes], async_arg)
4016
4017     type(*), dimension(..) :: data_arg
4018     integer :: bytes
4019     integer(acc_handle_kind) :: async_arg
```

4020 **Description**

4021 A call to an **acc_update_device** routine is functionally equivalent to an **update device**
 4022 directive. A call to an **acc_update_self** routine is functionally equivalent to an **update self**
 4023 directive. See Section 2.14.4. The arguments and the *data section* are as for **acc_copyin**.

4024 The behavior of these routines for the associated *data section* is:

- 4025 • If the *data section* is in shared memory or **bytes** is zero, no action is taken.
- 4026 • Otherwise:
 - 4027 – A call to an **acc_update_device** routine copies the data in the local memory to the
 4028 corresponding device memory.
 - 4029 – A call to an **acc_update_self** routine copies the data in the corresponding device
 4030 memory to the local memory.

4031 The **_async** versions of these routines will perform the data transfers asynchronously on the **async**
 4032 queue associated with **async_arg**. The routine may return before the data has been transferred;
 4033 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
 4034 until the data has been completely transferred.

4035 **Errors**

- 4036 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer and
 4037 **bytes** is nonzero.
- 4038 • An **acc_error_not_present** error is issued if the *data section* is not in shared memory
 4039 and is not present in the current device memory.

- 4040 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
4041 tion that is not contiguous (in Fortran).
- 4042 • An **acc_error_partly_present** error is issued if part of the *data section* is already
4043 present in the current device memory but all of the *data section* is not.
- 4044 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4045 *argument* value.

4046 See Section 5.2.2.

4047 3.2.21 acc_map_data

4048 Summary

4049 The **acc_map_data** routine maps previously allocated space in the current device memory to the
4050 specified host data.

4051 Format

C or C++:

```
4052     void acc_map_data(h_void* data_arg, d_void* data_dev,  
                     size_t bytes);
```

4053 Fortran:

```
4054     subroutine acc_map_data(data_arg, data_dev, bytes)  
4055         type(*), dimension(*) :: data_arg  
4056         type(c_ptr), value :: data_dev  
4057         integer(c_size_t), value :: bytes
```

4058 Description

4059 A call to the **acc_map_data** routine is similar to a call to **acc_create**, except that instead of
4060 allocating new device memory to start a data lifetime, the device address to use for the data lifetime
4061 is specified as an argument. **data_arg** is a host address, **data_dev** is the corresponding device
4062 address, and **bytes** is the length in bytes. **data_dev** may be the result of a call to **acc_malloc**,
4063 or may come from some other device-specific API routine. The associated *data section* is as for
4064 **acc_copyin**.

4065 The behavior of the **acc_map_data** routine is:

- 4066 • If the *data section* is in shared memory, the behavior is undefined.
- 4067 • If any of the data referred to by **data_dev** is already mapped to any host memory address,
4068 the behavior is undefined.
- 4069 • Otherwise, after this call, when **data_arg** appears in a data clause, the **data_dev** address
4070 will be used. The dynamic reference count for the data referred to by **data_arg** is set to
4071 one, but no data movement will occur.

4072 Memory mapped by **acc_map_data** may not have the associated dynamic reference count decre-
4073 mented to zero, except by a call to **acc_unmap_data**. See Section 2.6.7 Reference Counters.

4074 Errors

- 4075 • An **acc_invalid_null_pointer** error is issued if either **data_arg** or **data_dev** is
4076 a null pointer.

- 4077 • An **acc_invalid_argument** error is issued if **bytes** is zero.
- 4078 • An **acc_error_present** error is issued if any part of the *data section* is already present
- 4079 in the current device memory.

4080 See Section 5.2.2.

4081 **3.2.22 acc_unmap_data**

4082 **Summary**

4083 The **acc_unmap_data** routine unmaps device data from the specified host data.

4084 **Format**

4085 C or C++:

```
4086     void acc_unmap_data(h_void* data_arg);
```

4087 Fortran:

```
4088     subroutine acc_unmap_data(data_arg)
4089         type(*), dimension(*) :: data_arg
```

4090 **Description**

4091 A call to the **acc_unmap_data** routine is similar to a call to **acc_delete**, except the device

4092 memory is not deallocated. **data_arg** is a host address.

4093 The behavior of the **acc_unmap_data** routine is:

- 4094 • If **data_arg** was not previously mapped to some device address via a call to **acc_map_data**,
- 4095 the behavior is undefined.
- 4096 • Otherwise, the data lifetime for **data_arg** is ended. The dynamic reference count for
- 4097 **data_arg** is set to zero, but no data movement will occur and the corresponding device
- 4098 memory is not deallocated. See Section 2.6.7 Reference Counters.

4099 **Errors**

- 4100 • An **acc_invalid_null_pointer** error is issued if **data_arg** is a null pointer.
- 4101 • An **acc_error_present** error is issued if the structured reference count for the any part
- 4102 of the data is not zero.

4103 See Section 5.2.2.

4104 **3.2.23 acc_deviceptr**

4105 **Summary**

4106 The **acc_deviceptr** routine returns the device pointer associated with a specific host address.

4107 **Format**

4108 C or C++:

```
4109     d_void* acc_deviceptr(h_void* data_arg);
```

4110 Fortran:

```
4111     type(c_ptr) function acc_deviceptr(data_arg)
4112         type(*), dimension(*) :: data_arg
```


4113 **Description**

4114 The `acc_deviceptr` routine returns the device pointer associated with a host address. `data_arg`
 4115 is the address of a host variable or array that may have an active lifetime on the current device.

4116 The behavior of the `acc_deviceptr` routine for the data referred to by `data_arg` is:

- 4117 • If the data is in shared memory or `data_arg` is a null pointer, `acc_deviceptr` returns
 4118 the incoming address.
- 4119 • If the data is not present in the current device memory, `acc_deviceptr` returns a null
 4120 pointer.
- 4121 • Otherwise, `acc_deviceptr` returns the address in the current device memory that corre-
 4122 sponds to the address `data_arg`.

4123 **3.2.24 acc_hostptr**4124 **Summary**

4125 The `acc_hostptr` routine returns the host pointer associated with a specific device address.

4126 **Format**

4127 C or C++:

```
4128     h_void* acc_hostptr(d_void* data_dev);
```

4129 Fortran:

```
4130     type(c_ptr) function acc_hostptr(data_dev)
4131     type(c_ptr), value :: data_dev
```

4132 **Description**

4133 The `acc_hostptr` routine returns the host pointer associated with a device address. `data_dev`
 4134 is the address of a device variable or array, such as that returned from `acc_deviceptr`, `acc_create`
 4135 or `acc_copyin`.

4136 The behavior of the `acc_hostptr` routine for the data referred to by `data_dev` is:

- 4137 • If the data is in shared memory or `data_dev` is a null pointer, `acc_hostptr` returns the
 4138 incoming address.
- 4139 • If the data corresponds to a host address which is present in the current device memory,
 4140 `acc_hostptr` returns the host address.
- 4141 • Otherwise, `acc_hostptr` returns a null pointer.

4142 **3.2.25 acc_is_present**4143 **Summary**

4144 The `acc_is_present` routine tests whether a variable or array region is accessible from the
 4145 current device.

4146 **Format**

4147 C or C++:

```
4148     int acc_is_present(h_void* data_arg, size_t bytes);
```

4149 Fortran:

```
4150     logical function acc_is_present (data_arg)
4151     logical function acc_is_present (data_arg, bytes)
4152     type(*), dimension(..) :: data_arg
4153     integer :: bytes
```

4154 Description

4155 The **acc_is_present** routine tests whether the specified host data is accessible from the current
 4156 device. In C/C++, **data_arg** is a pointer to the data, and **bytes** specifies the data size in bytes. In
 4157 Fortran, two forms are supported. In the first, **data_arg** is a variable or contiguous array section.
 4158 In the second, **data_arg** is a variable or array element and **bytes** is the length in bytes. A
 4159 **bytes** value of zero is treated as a value of one if **data_arg** is not a null pointer.

4160 The behavior of the **acc_is_present** routines for the data referred to by **data_arg** is:

- 4161 • If the data is in shared memory, a call to **acc_is_present** will evaluate to *true*.
- 4162 • If the data is present in the current device memory, a call to **acc_is_present** will evaluate
 4163 to *true*.
- 4164 • Otherwise, a call to **acc_is_present** will evaluate to *false*.

4165 Errors

- 4166 • An **acc_error_invalid_argument** error is issued if **bytes** is negative (in Fortran).
- 4167 • An **acc_error_invalid_data_section** error is issued if **data_arg** is an array sec-
 4168 tion that is not contiguous (in Fortran).

4169 See Section 5.2.2.

4170 3.2.26 acc_memcpy_to_device

4171 Summary

4172 The **acc_memcpy_to_device** routine copies data from local memory to device memory.

4173 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,
4174 int async_arg);
```

Fortran:

```
subroutine acc_memcpy_to_device(data_dev_dest,
                                data_host_src, bytes)
subroutine acc_memcpy_to_device_async(data_dev_dest,
4175 data_host_src, bytes, async_arg)
4176 type(c_ptr), value :: data_dev_dest
4177 type(*), dimension(*) :: data_host_src
4178 integer(c_size_t), value :: bytes
4179 integer(acc_handle_kind), value :: async_arg
```

4180 **Description**

4181 The `acc_memcpy_to_device` routine copies `bytes` bytes of data from the local address in
 4182 `data_host_src` to the device address in `data_dev_dest`. `data_dev_dest` must be an
 4183 address accessible from the current device, such as an address returned from `acc_malloc` or
 4184 `acc_deviceptr`, or an address in shared memory.

4185 The behavior of the `acc_memcpy_to_device` routines is:

- 4186 • If `bytes` is zero, no action is taken.
- 4187 • If `data_dev_dest` and `data_host_src` both refer to shared memory and have the same
 4188 value, no action is taken.
- 4189 • If `data_dev_dest` and `data_host_src` both refer to shared memory and the memory
 4190 regions overlap, the behavior is undefined.
- 4191 • If the data referred to by `data_dev_dest` is not accessible by the current device, the be-
 4192 havior is undefined.
- 4193 • If the data referred to by `data_host_src` is not accessible by the local thread, the behavior
 4194 is undefined.
- 4195 • Otherwise, `bytes` bytes of data at `data_host_src` in local memory are copied to
 4196 `data_dev_dest` in the current device memory.

4197 The `_async` version of this routine will perform the data transfers asynchronously on the `async`
 4198 queue associated with `async_arg`. The routine may return before the data has been transferred;
 4199 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
 4200 until the data has been completely transferred.

4201 **Errors**

- 4202 • An `acc_error_invalid_null_pointer` error is issued if `data_dev_dest` or
 4203 `data_host_src` is a null pointer and `bytes` is nonzero.
- 4204 • An `acc_error_invalid_async` error is issued if `async_arg` is not a valid *async-*
 4205 *argument* value.

4206 See Section 5.2.2.

4207 **3.2.27 acc_memcpy_from_device**4208 **Summary**

4209 The `acc_memcpy_from_device` routine copies data from device memory to local memory.

4210 **Format**

C or C++:

```

4211     void acc_memcpy_from_device(h_void* data_host_dest,
4212                               d_void* data_dev_src, size_t bytes);
4211     void acc_memcpy_from_device_async(h_void* data_host_dest,
4212                                     d_void* data_dev_src, size_t bytes,
4212                                     int async_arg);
  
```

Fortran:

```

    subroutine acc_memcpy_from_device(data_host_dest,
  
```

```

                                data_dev_src, bytes)
subroutine acc_memcpy_from_device_async(data_host_dest,
4213                                data_dev_src, bytes, async_arg)
4214    type(*), dimension(*) :: data_host_dest
4215    type(c_ptr), value :: data_dev_src
4216    integer(c_size_t), value :: bytes
4217    integer(acc_handle_kind), value :: async_arg

```

4218 Description

4219 The `acc_memcpy_from_device` routine copies `bytes` bytes of data from the device address
 4220 in `data_dev_src` to the local address in `data_host_dest`. `data_dev_src` must be an
 4221 address accessible from the current device, such as an address returned from `acc_malloc` or
 4222 `acc_deviceptr`, or an address in shared memory.

4223 The behavior of the `acc_memcpy_from_device` routines is:

- 4224 • If `bytes` is zero, no action is taken.
- 4225 • If `data_host_dest` and `data_dev_src` both refer to shared memory and have the same
 4226 value, no action is taken.
- 4227 • If `data_host_dest` and `data_dev_src` both refer to shared memory and the memory
 4228 regions overlap, the behavior is undefined.
- 4229 • If the data referred to by `data_dev_src` is not accessible by the current device, the behav-
 4230 ior is undefined.
- 4231 • If the data referred to by `data_host_dest` is not accessible by the local thread, the behav-
 4232 ior is undefined.
- 4233 • Otherwise, `bytes` bytes of data at `data_dev_src` in the current device memory are copied
 4234 to `data_host_dest` in local memory.

4235 The `_async` version of this routine will perform the data transfers asynchronously on the `async`
 4236 queue associated with `async_arg`. The routine may return before the data has been transferred;
 4237 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
 4238 until the data has been completely transferred.

4239 Errors

- 4240 • An `acc_error_invalid_null_pointer` error is issued if `data_host_dest` or
 4241 `data_dev_src` is a null pointer and `bytes` is nonzero.
- 4242 • An `acc_error_invalid_async` error is issued if `async_arg` is not a valid *async-*
 4243 *argument* value.

4244 See Section 5.2.2.

4245 3.2.28 acc_memcpy_device

4246 Summary

4247 The `acc_memcpy_device` routine copies data from one memory location to another memory
 4248 location on the current device.

4249 **Format**

C or C++:

```

4249     void acc_memcpy_device(d_void* data_dev_dest,
4250                           d_void* data_dev_src, size_t bytes);
4251     void acc_memcpy_device_async(d_void* data_dev_dest,
4252                                 d_void* data_dev_src, size_t bytes,
4253                                 int async_arg);

```

Fortran:

```

4252     subroutine acc_memcpy_device(data_dev_dest,
4253                                 data_dev_src, bytes);
4254     subroutine acc_memcpy_device_async(data_dev_dest,
4255                                       data_dev_src, bytes,
4256                                       async_arg);
4257     type(c_ptr), value :: data_dev_dest
4258     type(c_ptr), value :: data_dev_src
4259     integer(c_size_t), value :: bytes
4260     integer(acc_handle_kind), value :: async_arg

```

4257 **Description**

4258 The `acc_memcpy_device` routine copies `bytes` bytes of data from the device address in
4259 `data_dev_src` to the device address in `data_dev_dest`. Both addresses must be addresses in
4260 the current device memory, such as would be returned from `acc_malloc` or `acc_deviceptr`.

4261 The behavior of the `acc_memcpy_device` routines is:

- 4262 • If `bytes` is zero, no action is taken.
- 4263 • If `data_dev_dest` and `data_dev_src` have the same value, no action is taken.
- 4264 • If the memory regions referred to by `data_dev_dest` and `data_dev_src` overlap, the
4265 behavior is undefined.
- 4266 • If the data referred to by `data_dev_src` or `data_dev_dest` is not accessible by the
4267 current device, the behavior is undefined.
- 4268 • Otherwise, `bytes` bytes of data at `data_dev_src` in the current device memory are copied
4269 to `data_dev_dest` in the current device memory.

4270 The `_async` version of this routine will perform the data transfers asynchronously on the `async`
4271 queue associated with `async_arg`. The routine may return before the data has been transferred;
4272 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return
4273 until the data has been completely transferred.

4274 **Errors**

- 4275 • An `acc_error_invalid_null_pointer` error is issued if `data_dev_dest` or
4276 `data_dev_src` is a null pointer and `bytes` is nonzero.
- 4277 • An `acc_error_invalid_async` error is issued if `async_arg` is not a valid *async-*
4278 *argument* value.

4279 See Section 5.2.2.

4280 **3.2.29 acc_attach and acc_detach**4281 **Summary**

4282 The **acc_attach** routines update a pointer in device memory to point to the corresponding device
 4283 copy of the host pointer target. The **acc_detach** routines restore a pointer in device memory to
 4284 point to the host pointer target.

4285 **Format**

4286 C or C++:

```
4287     void acc_attach(h_void** ptr_addr);
4288     void acc_attach_async(h_void** ptr_addr, int async_arg);
4289
4290     void acc_detach(h_void** ptr_addr);
4291     void acc_detach_async(h_void** ptr_addr, int async_arg);
4292     void acc_detach_finalize(h_void** ptr_addr);
4293     void acc_detach_finalize_async(h_void** ptr_addr,
4294                                   int async_arg);
```

4295 Fortran:

```
4296     subroutine acc_attach(ptr_addr)
4297     subroutine acc_attach_async(ptr_addr, async_arg)
4298         type(*), dimension(..)          :: ptr_addr
4299         integer(acc_handle_kind), value :: async_arg
4300
4301     subroutine acc_detach(ptr_addr)
4302     subroutine acc_detach_async(ptr_addr, async_arg)
4303     subroutine acc_detach_finalize(ptr_addr)
4304     subroutine acc_detach_finalize_async(ptr_addr,
4305                                         async_arg)
4306         type(*), dimension(..)          :: ptr_addr
4307         integer(acc_handle_kind), value :: async_arg
```

4308 **Description**

4309 A call to an **acc_attach** routine is functionally equivalent to an **enter data attach** direc-
 4310 tive, as described in Section 2.7.12. A call to an **acc_detach** routine is functionally equivalent to
 4311 an **exit data detach** directive, and a call to an **acc_detach_finalize** routine is function-
 4312 ally equivalent to an **exit data finalize detach** directive, as described in Section 2.7.13.
 4313 **ptr_addr** must be the address of a host pointer. **async_arg** must be an *async-argument* as
 4314 defined in Section 2.16.

4315 The behavior of these routines is:

- 4316 • If **ptr_addr** refers to shared memory, no action is taken.
- 4317 • If the pointer referred to by **ptr_addr** is not present in the current device memory, no action
 4318 is taken.
- 4319 • Otherwise:
 - 4320 – The **acc_attach** routines perform an *attach* action on the pointer referred to by
 4321 **ptr_addr**; see Section 2.7.2.

- 4322 – The **acc_detach** routines perform a *detach* action on the pointer referred to by **ptr_addr**;
4323 See Section 2.7.2.
- 4324 – The **acc_detach_finalize** routines perform an *immediate detach* action on the
4325 pointer referred to by **ptr_addr**; see Section 2.7.2.

4326 These routines may issue a data transfer from local memory to device memory. The **_async** ver-
4327 sions of these routines will perform the data transfers asynchronously on the async queue associated
4328 with **async_arg**. These routines may return before the data has been transferred; see Section 2.16
4329 for more details. The synchronous versions will not return until the data has been completely trans-
4330 ferred.

4331 Errors

- 4332 • An **acc_error_invalid_null_pointer** error is issued if **ptr_addr** is a null pointer.
- 4333 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4334 *argument* value.

4335 See Section 5.2.2.

4336 3.2.30 acc_memcpy_d2d

4337 Summary

4338 The **acc_memcpy_d2d** routines copy the contents of an array on one device to an array on the
4339 same or a different device without updating the value on the host.

4340 Format

C or C++:

```

4341     void acc_memcpy_d2d(h_void* data_arg_dest,
4342                       h_void* data_arg_src, size_t bytes,
4343                       int dev_num_dest, int dev_num_src);
4344     void acc_memcpy_d2d_async(h_void* data_arg_dest,
4345                              h_void* data_arg_src, size_t bytes,
4346                              int dev_num_dest, int dev_num_src,
4347                              int async_arg_src);

```

Fortran:

```

4343     subroutine acc_memcpy_d2d(data_arg_dest, data_arg_src, &
4344                              bytes, dev_num_dest, dev_num_src)
4345     subroutine acc_memcpy_d2d_async(data_arg_dest, data_arg_src, &
4346                                    bytes, dev_num_dest, dev_num_src, &
4347                                    async_arg_src)
4348     type(*), dimension(..) :: data_arg_dest
4349     type(*), dimension(..) :: data_arg_src
4350     integer :: bytes
4351     integer :: dev_num_dest
4352     integer :: dev_num_src
4353     integer :: async_arg_src

```

4351 Description

4352 The **acc_memcpy_d2d** routines are passed the address of destination and source host data as well
4353 as integer device numbers for the destination and source devices, which must both be of the current
4354 device type.

4355 The behavior of the **acc_memcpy_d2d** routines is:

- 4356 • If **bytes** is zero, no action is taken.
- 4357 • If both pointers have the same value and either the two device numbers are the same or the
4358 addresses are in shared memory, then no action is taken.
- 4359 • Otherwise, **bytes** bytes of data at the device address corresponding to **data_arg_src** on
4360 device **dev_num_src** are copied to the device address corresponding to **data_arg_dest**
4361 on device **dev_num_dest**.

4362 For **acc_memcpy_d2d_async** the value of **async_arg_src** is the number of an async queue
4363 on the source device. This routine will perform the data transfers asynchronously on the async queue
4364 associated with **async_arg_src** for device **dev_num_src**; see Section 2.16 Asynchronous Behavior
4365 for more details.

4366 Errors

- 4367 • An **acc_error_device_unavailable** error is issued if **dev_num_dest** or **dev_num_src**
4368 is not a valid device number.
- 4369 • An **acc_error_invalid_null_pointer** error is issued if either **data_arg_dest**
4370 or **data_arg_src** is a null pointer and **bytes** is nonzero.
- 4371 • An **acc_error_not_present** error is issued if the data at either address is not in shared
4372 memory and is not present in the respective device memory.
- 4373 • An **acc_error_partly_present** error is issued if part of the data is already present in
4374 the current device memory but all of the data is not.
- 4375 • An **acc_error_invalid_async** error is issued if **async_arg** is not a valid *async-*
4376 *argument* value.

4377 See Section 5.2.2.

4. Environment Variables

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

4.1 ACC_DEVICE_TYPE

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

Example:

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

4.2 ACC_DEVICE_NUM

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

Example:

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

4.3 ACC_PROFLIB

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

Example:

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


5. Profiling and Error Callback Interface

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

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

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

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

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

5.1 Events

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

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

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

```
4445     typedef enum acc_event_t{
4446         acc_ev_none = 0,
4447         acc_ev_device_init_start = 1,
4448         acc_ev_device_init_end = 2,
4449         acc_ev_device_shutdown_start = 3,
4450         acc_ev_device_shutdown_end = 4,
4451         acc_ev_runtime_shutdown = 5,
4452         acc_ev_create = 6,
4453         acc_ev_delete = 7,
4454         acc_ev_alloc = 8,
4455         acc_ev_free = 9,
4456         acc_ev_enter_data_start = 10,
4457         acc_ev_enter_data_end = 11,
4458         acc_ev_exit_data_start = 12,
4459         acc_ev_exit_data_end = 13,
4460         acc_ev_update_start = 14,
4461         acc_ev_update_end = 15,
4462         acc_ev_compute_construct_start = 16,
4463         acc_ev_compute_construct_end = 17,
4464         acc_ev_enqueue_launch_start = 18,
4465         acc_ev_enqueue_launch_end = 19,
4466         acc_ev_enqueue_upload_start = 20,
4467         acc_ev_enqueue_upload_end = 21,
4468         acc_ev_enqueue_download_start = 22,
4469         acc_ev_enqueue_download_end = 23,
4470         acc_ev_wait_start = 24,
4471         acc_ev_wait_end = 25,
4472         acc_ev_error = 100,
4473         acc_ev_last = 101
4474     }acc_event_t;
```

4475 The value of `acc_ev_last` will change if new events are added to the enumeration, so a library
4476 should not depend on that value.

4477 5.1.1 Runtime Initialization and Shutdown

4478 No callbacks can be registered for the runtime initialization. Instead the initialization of the tool is
4479 handled as described in Section 5.3 Loading the Library.

4480 The *runtime shutdown* profiling event name is

```
4481     acc_ev_runtime_shutdown
```

4482 This event is triggered before the OpenACC runtime shuts down, either because all devices have
4483 been shutdown by calls to the `acc_shutdown` API routine, or at the end of the program.

4484 5.1.2 Device Initialization and Shutdown

4485 The *device initialization* profiling event names are

4486 **acc_ev_device_init_start**
4487 **acc_ev_device_init_end**

4488 These events are triggered when a device is being initialized by the OpenACC runtime. This may be
4489 when the program starts, or may be later during execution when the program reaches an **acc_init**
4490 call or an OpenACC construct. The **acc_ev_device_init_start** is triggered before device
4491 initialization starts and **acc_ev_device_init_end** after initialization is complete.

4492 The *device shutdown* profiling event names are

4493 **acc_ev_device_shutdown_start**
4494 **acc_ev_device_shutdown_end**

4495 These events are triggered when a device is shut down, most likely by a call to the OpenACC
4496 **acc_shutdown** API routine. The **acc_ev_device_shutdown_start** is triggered before
4497 the device shutdown process starts and **acc_ev_device_shutdown_end** after the device shut-
4498 down is complete.

4499 **5.1.3 Enter Data and Exit Data**

4500 The *enter data* profiling event names are

4501 **acc_ev_enter_data_start**
4502 **acc_ev_enter_data_end**

4503 These events are triggered at **enter data** directives, entry to data constructs, and entry to implicit
4504 data regions such as those generated by compute constructs. The **acc_ev_enter_data_start**
4505 event is triggered before any *data allocation*, *data update*, or *wait* events that are associated with
4506 that directive or region entry, and the **acc_ev_enter_data_end** is triggered after those events.

4507 The *exit data* profiling event names are

4508 **acc_ev_exit_data_start**
4509 **acc_ev_exit_data_end**

4510 These events are triggered at **exit data** directives, exit from **data** constructs, and exit from
4511 implicit data regions. The **acc_ev_exit_data_start** event is triggered before any *data*
4512 *deallocation*, *data update*, or *wait* events associated with that directive or region exit, and the
4513 **acc_ev_exit_data_end** event is triggered after those events.

4514 When the construct that triggers an *enter data* or *exit data* event was generated implicitly by the
4515 compiler the **implicit** field in the event structure will be set to **1**. When the construct that
4516 triggers these events was specified explicitly by the application code the **implicit** field in the
4517 event structure will be set to **0**.

4518 **5.1.4 Data Allocation**

4519 The *data allocation* profiling event names are

4520 **acc_ev_create**
4521 **acc_ev_delete**
4522 **acc_ev_alloc**
4523 **acc_ev_free**

4524 An **acc_ev_alloc** event is triggered when the OpenACC runtime allocates memory from the de-
4525 vice memory pool, and an **acc_ev_free** event is triggered when the runtime frees that memory.
4526 An **acc_ev_create** event is triggered when the OpenACC runtime associates device memory
4527 with local memory, such as for a data clause (**create**, **copyin**, **copy**, **copyout**) at entry to
4528 a data construct, compute construct, at an **enter data** directive, or in a call to a data API rou-
4529 tine (**acc_copyin**, **acc_create**, ...). An **acc_ev_create** event may be preceded by an
4530 **acc_ev_alloc** event, if newly allocated memory is used for this device data, or it may not, if
4531 the runtime manages its own memory pool. An **acc_ev_delete** event is triggered when the
4532 OpenACC runtime disassociates device memory from local memory, such as for a data clause at
4533 exit from a data construct, compute construct, at an **exit data** directive, or in a call to a data API
4534 routine (**acc_copyout**, **acc_delete**, ...). An **acc_ev_delete** event may be followed by
4535 an **acc_ev_free** event, if the disassociated device memory is freed, or it may not, if the runtime
4536 manages its own memory pool.

4537 When the action that generates a *data allocation* event was generated explicitly by the application
4538 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event
4539 is triggered because of a variable or array with implicitly-determined data attributes or otherwise
4540 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

4541 5.1.5 Data Construct

4542 The profiling events for entering and leaving *data constructs* are mapped to *enter data* and *exit data*
4543 events as described in Section 5.1.3 Enter Data and Exit Data.

4544 5.1.6 Update Directive

4545 The *update directive* profiling event names are

4546 **acc_ev_update_start**
4547 **acc_ev_update_end**

4548 The **acc_ev_update_start** event will be triggered at an **update** directive, before any *data*
4549 *update* or *wait* events that are associated with the update directive are carried out, and the corre-
4550 sponding **acc_ev_update_end** event will be triggered after any of the associated events.

4551 5.1.7 Compute Construct

4552 The *compute construct* profiling event names are

4553 **acc_ev_compute_construct_start**
4554 **acc_ev_compute_construct_end**

4555 The **acc_ev_compute_construct_start** event is triggered at entry to a compute construct,
4556 before any *launch* events that are associated with entry to the compute construct. The
4557 **acc_ev_compute_construct_end** event is triggered at the exit of the compute construct,
4558 after any *launch* events associated with exit from the compute construct. If there are data clauses
4559 on the compute construct, those data clauses may be treated as part of the compute construct, or as
4560 part of a data construct containing the compute construct. The callbacks for data clauses must use
4561 the same line numbers as for the compute construct events.

5.1.8 Enqueue Kernel Launch

The *launch* profiling event names are

```
acc_ev_enqueue_launch_start
acc_ev_enqueue_launch_end
```

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

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

5.1.9 Enqueue Data Update (Upload and Download)

The *data update* profiling event names are

```
acc_ev_enqueue_upload_start
acc_ev_enqueue_upload_end
acc_ev_enqueue_download_start
acc_ev_enqueue_download_end
```

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

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

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

5.1.10 Wait

The *wait* profiling event names are

```
acc_ev_wait_start
acc_ev_wait_end
```

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

4601 queue after the local thread has determined that the queue is empty.

4602 Wait events occur when the local thread and a device synchronize, either due to a **wait** directive
4603 or by a *wait* clause on a synchronous data construct, compute construct, or **enter data**, **exit**
4604 **data**, or **update** directive. For *wait* events triggered by an explicit synchronous **wait** directive
4605 or *wait* clause, the **implicit** field in the event structure will be **0**. For all other wait events, the
4606 **implicit** field in the event structure will be **1**.

4607 The OpenACC runtime need not trigger *wait* events for queues that have not been used in the
4608 program, and need not trigger *wait* events for queues that have not been used by this thread since
4609 the last *wait* operation. For instance, an **acc wait** directive with no arguments is defined to wait on
4610 all queues. If the program only uses the default (synchronous) queue and the queue associated with
4611 **async(1)** and **async(2)** then an **acc wait** directive may trigger *wait* events only for those
4612 three queues. If the implementation knows that no activities have been enqueued on the **async(2)**
4613 queue since the last *wait* operation, then the **acc wait** directive may trigger *wait* events only for
4614 the default queue and the **async(1)** queue.

4615 5.1.11 Error Event

4616 The only error event is

4617 **acc_ev_error**

4618 An **acc_ev_error** event is triggered when the OpenACC program detects a runtime error con-
4619 dition. The default runtime error callback routine may print an error message and halt program
4620 execution. An application can register additional error event callback routines, to allow a failing
4621 application to release resources or to cleanly shut down a large parallel runtime with many threads
4622 and processes, for instance.

4623 The application can register multiple alternate error callbacks. As described in Section
4624 5.4.1 Multiple Callbacks, the callbacks will be invoked in the order in which they are registered.
4625 If all the error callbacks return, the default error callback will be invoked. The error callback
4626 routine must not execute any OpenACC compute or data constructs. The only OpenACC API
4627 routines that can be safely invoked from an error callback routine are **acc_get_property**,
4628 **acc_get_property_string**, and **acc_shutdown**.

4629 5.2 Callbacks Signature

4630 This section describes the signature of event callbacks. All event callbacks have the same signature.
4631 The routine prototypes are available in the header file **acc_callback.h**, which is delivered with
4632 the OpenACC implementation.

4633 All callback routines have three arguments. The first argument is a pointer to a struct containing
4634 general information; the same struct type is used for all callback events. The second argument is
4635 a pointer to a struct containing information specific to that callback event; there is one struct type
4636 containing information for data events, another struct type containing information for kernel launch
4637 events, and a third struct type for other events, containing essentially no information. The third
4638 argument is a pointer to a struct containing information about the application programming interface
4639 (API) being used for the specific device. For NVIDIA CUDA devices, this contains CUDA-specific
4640 information; for OpenCL devices, this contains OpenCL-specific information. Other interfaces can
4641 be supported as they are added by implementations. The prototype for a callback routine is:


```

4642     typedef void (*acc_callback)
4643         (acc_callback_info*, acc_event_info*, acc_api_info*);
4644     typedef acc_callback acc_prof_callback;

```

4645 In the descriptions, the datatype `ssize_t` means a signed 32-bit integer for a 32-bit binary and
 4646 a 64-bit integer for a 64-bit binary, the datatype `size_t` means an unsigned 32-bit integer for a
 4647 32-bit binary and a 64-bit integer for a 64-bit binary, and the datatype `int` means a 32-bit integer
 4648 for both 32-bit and 64-bit binaries.

4649 5.2.1 First Argument: General Information

4650 The first argument is a pointer to the `acc_callback_info` struct type:

```

4651     typedef struct acc_prof_info{
4652         acc_event_t event_type;
4653         int valid_bytes;
4654         int version;
4655         acc_device_t device_type;
4656         int device_number;
4657         int thread_id;
4658         ssize_t async;
4659         ssize_t async_queue;
4660         const char* src_file;
4661         const char* func_name;
4662         int line_no, end_line_no;
4663         int func_line_no, func_end_line_no;
4664     }acc_callback_info;
4665     typedef struct acc_prof_info acc_prof_info;

```

4666 The name `acc_prof_info` is preserved for backward compatibility with previous versions of
 4667 OpenACC. The fields are described below.

4668 • **acc_event_t event_type** - The event type that triggered this callback. The datatype
 4669 is the enumeration type `acc_event_t`, described in the previous section. This allows the
 4670 same callback routine to be used for different events.

4671 • **int valid_bytes** - The number of valid bytes in this struct. This allows a library to inter-
 4672 face with newer runtimes that may add new fields to the struct at the end while retaining com-
 4673 patibility with older runtimes. A runtime must fill in the `event_type` and `valid_bytes`
 4674 fields, and must fill in values for all fields with offset less than `valid_bytes`. The value of
 4675 `valid_bytes` for a struct is recursively defined as:

```

4676     valid_bytes(struct) = offset(lastfield) + valid_bytes(lastfield)
4677     valid_bytes(type[n]) = (n-1)*sizeof(type) + valid_bytes(type)
4678     valid_bytes(basictype) = sizeof(basictype)

```

4679 • **int version** - A version number; the value of `_OPENACC`.

4680 • **acc_device_t device_type** - The device type corresponding to this event. The datatype
 4681 is `acc_device_t`, an enumeration type of all the supported device types, defined in `openacc.h`.

4682 • **int device_number** - The device number. Each device is numbered, typically starting at

- 4683 device zero. For applications that use more than one device type, the device numbers may be
4684 unique across all devices or may be unique only across all devices of the same device type.
- 4685 • **int thread_id** - The host thread ID making the callback. Host threads are given unique
4686 thread ID numbers typically starting at zero. This is not necessarily the same as the OpenMP
4687 thread number.
 - 4688 • **ssize_t async** - The *async-value* used for operations associated with this event; see Sec-
4689 tion 2.16 Asynchronous Behavior.
 - 4690 • **ssize_t async_queue** - The actual activity queue onto which the **async** field gets
4691 mapped; see Section 2.16 Asynchronous Behavior.
 - 4692 • **const char* src_file** - A pointer to null-terminated string containing the name of or
4693 path to the source file, if known, or a null pointer if not. If the library wants to save the source
4694 file name, it should allocate memory and copy the string.
 - 4695 • **const char* func_name** - A pointer to a null-terminated string containing the name of
4696 the function in which the event occurred, if known, or a null pointer if not. If the library wants
4697 to save the function name, it should allocate memory and copy the string.
 - 4698 • **int line_no** - The line number of the directive or program construct or the starting line
4699 number of the OpenACC construct corresponding to the event. A negative or zero value
4700 means the line number is not known.
 - 4701 • **int end_line_no** - For an OpenACC construct, this contains the line number of the end
4702 of the construct. A negative or zero value means the line number is not known.
 - 4703 • **int func_line_no** - The line number of the first line of the function named in **func_name**.
4704 A negative or zero value means the line number is not known.
 - 4705 • **int func_end_line_no** - The last line number of the function named in **func_name**.
4706 A negative or zero value means the line number is not known.

4707 5.2.2 Second Argument: Event-Specific Information

4708 The second argument is a pointer to the **acc_event_info** union type.

```
4709     typedef union acc_event_info{
4710         acc_event_t event_type;
4711         acc_data_event_info data_event;
4712         acc_launch_event_info launch_event;
4713         acc_other_event_info other_event;
4714     }acc_event_info;
```

4715 The **event_type** field selects which union member to use. The first five members of each union
4716 member are identical. The second through fifth members of each union member (**valid_bytes**,
4717 **parent_construct**, **implicit**, and **tool_info**) have the same semantics for all event
4718 types:

- 4719 • **int valid_bytes** - The number of valid bytes in the respective struct. (This field is similar
4720 used as discussed in Section 5.2.1 First Argument: General Information.)

- 4721 • **acc_construct_t parent_construct** - This field describes the type of construct
4722 that caused the event to be emitted. The possible values for this field are defined by the
4723 **acc_construct_t** enum, described at the end of this section.
- 4724 • **int implicit** - This field is set to 1 for any implicit event, such as an implicit wait at
4725 a synchronous data construct or synchronous enter data, exit data or update directive. This
4726 field is set to zero when the event is triggered by an explicit directive or call to a runtime API
4727 routine.
- 4728 • **void* tool_info** - This field is used to pass tool-specific information from a **_start**
4729 event to the matching **_end** event. For a **_start** event callback, this field will be initialized
4730 to a null pointer. The value of this field for a **_end** event will be the value returned by the
4731 library in this field from the matching **_start** event callback, if there was one, or a null
4732 pointer otherwise. For events that are neither **_start** or **_end** events, this field will be a
4733 null pointer.

4734 Data Events

4735 For a data event, as noted in the event descriptions, the second argument will be a pointer to the
4736 **acc_data_event_info** struct.

```
4737 typedef struct acc_data_event_info{
4738     acc_event_t event_type;
4739     int valid_bytes;
4740     acc_construct_t parent_construct;
4741     int implicit;
4742     void* tool_info;
4743     const char* var_name;
4744     size_t bytes;
4745     const void* host_ptr;
4746     const void* device_ptr;
4747 }acc_data_event_info;
```

4748 The fields specific for a data event are:

- 4749 • **acc_event_t event_type** - The event type that triggered this callback. The events that
4750 use the **acc_data_event_info** struct are:
 - 4751 **acc_ev_enqueue_upload_start**
 - 4752 **acc_ev_enqueue_upload_end**
 - 4753 **acc_ev_enqueue_download_start**
 - 4754 **acc_ev_enqueue_download_end**
 - 4755 **acc_ev_create**
 - 4756 **acc_ev_delete**
 - 4757 **acc_ev_alloc**
 - 4758 **acc_ev_free**
- 4759 • **const char* var_name** - A pointer to null-terminated string containing the name of the
4760 variable for which this event is triggered, if known, or a null pointer if not. If the library wants
4761 to save the variable name, it should allocate memory and copy the string.
- 4762 • **size_t bytes** - The number of bytes for the data event.

- 4763 • **const void* host_ptr** - If available and appropriate for this event, this is a pointer to
4764 the host data.
- 4765 • **const void* device_ptr** - If available and appropriate for this event, this is a pointer
4766 to the corresponding device data.

4767 Launch Events

4768 For a launch event, as noted in the event descriptions, the second argument will be a pointer to the
4769 **acc_launch_event_info** struct.

```
4770     typedef struct acc_launch_event_info{
4771         acc_event_t event_type;
4772         int valid_bytes;
4773         acc_construct_t parent_construct;
4774         int implicit;
4775         void* tool_info;
4776         const char* kernel_name;
4777         size_t num_gangs, num_workers, vector_length;
4778         size_t* num_gangs_per_dim;
4779     }acc_launch_event_info;
```

4780 The fields specific for a launch event are:

- 4781 • **acc_event_t event_type** - The event type that triggered this callback. The events that
4782 use the **acc_launch_event_info** struct are:
- 4783 **acc_ev_enqueue_launch_start**
4784 **acc_ev_enqueue_launch_end**
- 4785 • **const char* kernel_name** - A pointer to null-terminated string containing the name of
4786 the kernel being launched, if known, or a null pointer if not. If the library wants to save the
4787 kernel name, it should allocate memory and copy the string.
- 4788 • **size_t num_gangs, num_workers, vector_length** - The number of gangs, work-
4789 ers, and vector lanes created for this kernel launch.
- 4790 • **size_t* num_gangs_per_dim** - An array of **size_t** whose first element indicates the
4791 number of dimensions of gang parallelism and each subsequent element gives the number of
4792 gangs along each dimension starting with dimension 1. The product of the values of elements
4793 1 through **num_gangs_per_dim[0]** is **num_gangs**.

4794 Error Events

4795 For an error event, as noted in the event descriptions, the second argument will be a pointer to the
4796 **acc_error_event_info** struct.

```
4797     typedef struct acc_error_event_info{
4798         acc_event_t event_type;
4799         int valid_bytes;
4800         acc_construct_t parent_construct;
4801         int implicit;
4802         void* tool_info;
```

```

4803     acc_error_t error_code;
4804     const char* error_message;
4805     size_t runtime_info;
4806 }acc_error_event_info;

```

4807 The enumeration type for the error code is

```

4808     typedef enum acc_error_t{
4809         acc_error_none = 0,
4810         acc_error_other = 1,
4811         acc_error_system = 2,
4812         acc_error_execution = 3,
4813         acc_error_device_init = 4,
4814         acc_error_device_shutdown = 5,
4815         acc_error_device_unavailable = 6,
4816         acc_error_device_type_unavailable = 7,
4817         acc_error_wrong_device_type = 8,
4818         acc_error_out_of_memory = 9,
4819         acc_error_not_present = 10,
4820         acc_error_partly_present = 11,
4821         acc_error_present = 12,
4822         acc_error_invalid_argument = 13,
4823         acc_error_invalid_async = 14,
4824         acc_error_invalid_null_pointer = 15,
4825         acc_error_invalid_data_section = 16,
4826         acc_error_implementation_defined = 100
4827     }acc_error_t;

```

4828 The fields specific for an error event are:

4829 • **acc_event_t event_type** - The event type that triggered this callback. The only event
4830 that uses the **acc_error_event_info** struct is:

```
4831     acc_ev_error
```

4832 • **int implicit** - This will be set to 1.

4833 • **acc_error_t error_code** - The error codes used are:

- 4834 - **acc_error_other** is used for error conditions other than those described below.
- 4835 - **acc_error_system** is used when there is a system error condition.
- 4836 - **acc_error_execution** is used when there is an error condition issued from code
4837 executing on the device.
- 4838 - **acc_error_device_init** is used for any error initializing a device.
- 4839 - **acc_error_device_shutdown** is used for any error shutting down a device.
- 4840 - **acc_error_device_unavailable** is used when there is an error where the se-
4841 lected device is unavailable.
- 4842 - **acc_error_device_type_unavailable** is used when there is an error where
4843 no device of the selected device type is available or is supported.

- 4844 – **acc_error_wrong_device_type** is used when there is an error related to the
 4845 device type, such as a mismatch between the device type for which a compute construct
 4846 was compiled and the device available at runtime.
- 4847 – **acc_error_out_of_memory** is used when the program tries to allocate more mem-
 4848 ory on the device than is available.
- 4849 – **acc_error_not_present** is used for an error related to data not being present at
 4850 runtime.
- 4851 – **acc_error_partly_present** is used for an error related to part of the data being
 4852 present but not being completely present at runtime.
- 4853 – **acc_error_present** is used for an error related to data being unexpectedly present
 4854 at runtime.
- 4855 – **acc_error_invalid_argument** is used when an API routine is called with a
 4856 invalid argument value, other than those described above.
- 4857 – **acc_error_invalid_async** is used when an API routine is called with an invalid
 4858 *async-argument*, or when a directive is used with an invalid *async-argument*.
- 4859 – **acc_error_invalid_null_pointer** is used when an API routine is called with
 4860 a null pointer argument where it is invalid, or when a directive is used with a null pointer
 4861 in a context where it is invalid.
- 4862 – **acc_error_invalid_data_section** is used when an invalid array section ap-
 4863 pears in a directive data clause, or an invalid array section appears as a runtime API call
 4864 argument.
- 4865 – **acc_error_implementation_defined**: any value greater or equal to this value
 4866 may be used for an implementation-defined error code.
- 4867 • **const char* error_message** - A pointer to null-terminated string containing an error
 4868 message from the OpenACC runtime describing the error, or a null pointer.
- 4869 • **size_t runtime_info** - A value, such as an error code, from the underlying device
 4870 runtime or driver, if one is available and appropriate.

4871 Other Events

4872 For any event that does not use the **acc_data_event_info**, **acc_launch_event_info**, or
 4873 **acc_error_event_info** struct, the second argument to the callback routine will be a pointer
 4874 to **acc_other_event_info** struct.

```

4875     typedef struct acc_other_event_info{
4876         acc_event_t event_type;
4877         int valid_bytes;
4878         acc_construct_t parent_construct;
4879         int implicit;
4880         void* tool_info;
4881     }acc_other_event_info;
```

4882 Parent Construct Enumeration

4883 All event structures contain a **parent_construct** member that describes the type of construct
 4884 that caused the event to be emitted. The purpose of this field is to provide a means to identify
 4885 the type of construct emitting the event in the cases where an event may be emitted by multi-
 4886 ple construct types, such as is the case with data and wait events. The possible values for the
 4887 **parent_construct** field are defined in the enumeration type **acc_construct_t**. In the
 4888 case of combined directives, the outermost construct of the combined construct should be specified
 4889 as the **parent_construct**. If the event was emitted as the result of the application making a
 4890 call to the runtime api, the value will be **acc_construct_runtime_api**.

```

4891     typedef enum acc_construct_t{
4892         acc_construct_parallel = 0,
4893         acc_construct_serial = 16
4894         acc_construct_kernels = 1,
4895         acc_construct_loop = 2,
4896         acc_construct_data = 3,
4897         acc_construct_enter_data = 4,
4898         acc_construct_exit_data = 5,
4899         acc_construct_host_data = 6,
4900         acc_construct_atomic = 7,
4901         acc_construct_declare = 8,
4902         acc_construct_init = 9,
4903         acc_construct_shutdown = 10,
4904         acc_construct_set = 11,
4905         acc_construct_update = 12,
4906         acc_construct_routine = 13,
4907         acc_construct_wait = 14,
4908         acc_construct_runtime_api = 15,
4909     }acc_construct_t;
  
```

4910 5.2.3 Third Argument: API-Specific Information

4911 The third argument is a pointer to the **acc_api_info** struct type, shown here.

```

4912     typedef struct acc_api_info{
4913         acc_device_api device_api;
4914         int valid_bytes;
4915         acc_device_t device_type;
4916         int vendor;
4917         const void* device_handle;
4918         const void* context_handle;
4919         const void* async_handle;
4920     }acc_api_info;
  
```

4921 The fields are described below:

- 4922 • **acc_device_api device_api** - The API in use for this device. The data type is the
 4923 enumeration **acc_device_api**, which is described later in this section.
- 4924 • **int valid_bytes** - The number of valid bytes in this struct. See the discussion above in

4925 Section 5.2.1 First Argument: General Information.

- 4926 • **acc_device_t device_type** - The device type; the datatype is **acc_device_t**, de-
4927 fined in **openacc.h**.
- 4928 • **int vendor** - An identifier to identify the OpenACC vendor; contact your vendor to deter-
4929 mine the value used by that vendor's runtime.
- 4930 • **const void* device_handle** - If applicable, this will be a pointer to the API-specific
4931 device information.
- 4932 • **const void* context_handle** - If applicable, this will be a pointer to the API-specific
4933 context information.
- 4934 • **const void* async_handle** - If applicable, this will be a pointer to the API-specific
4935 async queue information.

4936 According to the value of **device_api** a library can cast the pointers of the fields **device_handle**,
4937 **context_handle** and **async_handle** to the respective device API type. The following device
4938 APIs are defined in the interface below. Any implementation-defined device API type must have a
4939 value greater than **acc_device_api_implementation_defined**.

```

4940     typedef enum acc_device_api{
4941         acc_device_api_none = 0,           /* no device API */
4942         acc_device_api_cuda = 1,         /* CUDA driver API */
4943         acc_device_api_opencl = 2,      /* OpenCL API */
4944         acc_device_api_other = 4,       /* other device API */
4945         acc_device_api_implementation_defined = 1000 /* other device API */
4946     }acc_device_api;

```

4942 5.3 Loading the Library

4943 This section describes how a tools library is loaded when the program is run. Four methods are
4944 described.

- 4945 • A tools library may be linked with the program, as any other library is linked, either as a
4946 static library or a dynamic library, and the runtime will call a predefined library initialization
4947 routine that will register the event callbacks.
- 4948 • The OpenACC runtime implementation may support a dynamic tools library, such as a shared
4949 object for Linux or OS/X, or a DLL for Windows, which is then dynamically loaded at runtime
4950 under control of the environment variable **ACC_PROFLIB**.
- 4951 • Some implementations where the OpenACC runtime is itself implemented as a dynamic li-
4952 brary may support adding a tools library using the **LD_PRELOAD** feature in Linux.
- 4953 • A tools library may be linked with the program, as in the first option, and the application itself
4954 may directly register event callback routines, or may invoke a library initialization routine that
4955 will register the event callbacks.

4956 Callbacks are registered with the runtime by calling **acc_callback_register** for each event
4957 as described in Section 5.4 Registering Event Callbacks. The prototype for **acc_callback_register**
4958 is:


```

4959     extern void acc_callback_register
4960           (acc_event_t event_type, acc_callback cb,
4961            acc_register_t info);

```

4962 The first argument to **acc_callback_register** is the event for which a callback is being
4963 registered (compare Section 5.1 Events). The second argument is a pointer to the callback routine:

```

4964     typedef void (*acc_callback)
4965           (acc_callback_info*, acc_event_info*, acc_api_info*);

```

4966 The third argument is an enum type:

```

4967     typedef enum acc_register_t{
4968         acc_reg = 0,
4969         acc_toggle = 1,
4970         acc_toggle_per_thread = 2
4971     }acc_register_t;

```

4972 This is usually **acc_reg**, but see Section 5.4.2 Disabling and Enabling Callbacks for cases where
4973 different values are used.

4974 An example of registering callbacks for launch, upload, and download events is:

```

4975     acc_callback_register(acc_ev_enqueue_launch_start,
4976                         prof_launch, acc_reg);
4977     acc_callback_register(acc_ev_enqueue_upload_start,
4978                         prof_data, acc_reg);
4979     acc_callback_register(acc_ev_enqueue_download_start,
4980                         prof_data, acc_reg);

```

4981 As shown in this example, the same routine (**prof_data**) can be registered for multiple events.
4982 The routine can use the **event_type** field in the **acc_callback_info** structure to determine
4983 for what event it was invoked.

4984 The names **acc_prof_register** and **acc_prof_unregister** are preserved for backward
4985 compatibility with previous versions of OpenACC.

4986 5.3.1 Library Registration

4987 The OpenACC runtime will invoke **acc_register_library**, passing the addresses of the reg-
4988 istration routines **acc_callback_register** and **acc_callback_unregister**, in case
4989 that routine comes from a dynamic library. In the third argument it passes the address of the lookup
4990 routine **acc_prof_lookup** to obtain the addresses of inquiry functions. No inquiry functions
4991 are defined in this profiling interface, but we preserve this argument for future support of sampling-
4992 based tools.

4993 Typically, the OpenACC runtime will include a *weak* definition of **acc_register_library**,
4994 which does nothing and which will be called when there is no tools library. In this case, the library
4995 can save the addresses of these routines and/or make registration calls to register any appropriate
4996 callbacks. The prototype for **acc_register_library** is:

```

4997     extern void acc_register_library
4998           (acc_prof_reg reg, acc_prof_reg unreg,

```

```
4999     acc_prof_lookup_func lookup);
```

5000 The first two arguments of this routine are of type:

```
5001     typedef void (*acc_prof_reg)
5002         (acc_event_t event_type, acc_callback cb,
5003         acc_register_t info);
```

5004 The third argument passes the address to the lookup function `acc_prof_lookup` to obtain the
5005 address of interface functions. It is of type:

```
5006     typedef void (*acc_query_fn) ();
5007     typedef acc_query_fn (*acc_prof_lookup_func)
5008         (const char* acc_query_fn_name);
```

5009 The argument of the lookup function is a string with the name of the inquiry function. There are no
5010 inquiry functions defined for this interface.

5011 5.3.2 Statically-Linked Library Initialization

5012 A tools library can be compiled and linked directly into the application. If the library provides an
5013 external routine `acc_register_library` as specified in Section 5.3.1 Library Registration, the
5014 runtime will invoke that routine to initialize the library.

5015 The sequence of events is:

- 5016 1. The runtime invokes the `acc_register_library` routine from the library.
- 5017 2. The `acc_register_library` routine calls `acc_callback_register` for each event
5018 to be monitored.
- 5019 3. `acc_callback_register` records the callback routines.
- 5020 4. The program runs, and your callback routines are invoked at the appropriate events.

5021 In this mode, only one tool library is supported.

5022 5.3.3 Runtime Dynamic Library Loading

5023 A common case is to build the tools library as a dynamic library (shared object for Linux or OS/X,
5024 DLL for Windows). In that case, you can have the OpenACC runtime load the library during initial-
5025 ization. This allows you to enable runtime profiling without rebuilding or even relinking your ap-
5026 plication. The dynamic library must implement a registration routine `acc_register_library`
5027 as specified in Section 5.3.1 Library Registration.

5028 The user may set the environment variable `ACC_PROFLIB` to the path to the library will tell the
5029 OpenACC runtime to load your dynamic library at initialization time:

```
5030     Bash:
5031         export ACC_PROFLIB=/home/user/lib/myprof.so
5032         ./myapp
5033     or
5034         ACC_PROFLIB=/home/user/lib/myprof.so ./myapp
```

```

5035 C-shell:
5036     setenv ACC_PROFLIB /home/user/lib/myprof.so
5037     ./myapp

```

When the OpenACC runtime initializes, it will read the **ACC_PROFLIB** environment variable (with **getenv**). The runtime will open the dynamic library (using **dlopen** or **LoadLibraryA**); if the library cannot be opened, the runtime may cause the program to halt execution and return an error status, or may continue execution with or without an error message. If the library is successfully opened, the runtime will get the address of the **acc_register_library** routine (using **dlsym** or **GetProcAddress**). If this routine is resolved in the library, it will be invoked passing in the addresses of the registration routine **acc_callback_register**, the deregistration routine **acc_callback_unregister**, and the lookup routine **acc_prof_lookup**. The registration routine in your library, **acc_register_library**, should register the callbacks by calling the **register** argument, and should save the addresses of the arguments (**register**, **unregister**, and **lookup**) for later use, if needed.

The sequence of events is:

- 5050 1. Initialization of the OpenACC runtime.
- 5051 2. OpenACC runtime reads **ACC_PROFLIB**.
- 5052 3. OpenACC runtime loads the library.
- 5053 4. OpenACC runtime calls the **acc_register_library** routine in that library.
- 5054 5. Your **acc_register_library** routine calls **acc_callback_register** for each event to be monitored.
- 5055 6. **acc_callback_register** records the callback routines.
- 5056 7. The program runs, and your callback routines are invoked at the appropriate events.

If supported, paths to multiple dynamic libraries may be specified in the **ACC_PROFLIB** environment variable, separated by semicolons (;). The OpenACC runtime will open these libraries and invoke the **acc_register_library** routine for each, in the order they appear in **ACC_PROFLIB**.

5061 5.3.4 Preloading with LD_PRELOAD

The implementation may also support dynamic loading of a tools library using the **LD_PRELOAD** feature available in some systems. In such an implementation, you need only specify your tools library path in the **LD_PRELOAD** environment variable before executing your program. The OpenACC runtime will invoke the **acc_register_library** routine in your tools library at initialization time. This requires that the OpenACC runtime include a dynamic library with a default (empty) implementation of **acc_register_library** that will be invoked in the normal case where there is no **LD_PRELOAD** setting. If an implementation only supports static linking, or if the application is linked without dynamic library support, this feature will not be available.

```

5070 Bash:
5071     export LD_PRELOAD=/home/user/lib/myprof.so
5072     ./myapp
5073 or
5074     LD_PRELOAD=/home/user/lib/myprof.so ./myapp

```

```

5075 C-shell:
5076     setenv LD_PRELOAD /home/user/lib/myprof.so
5077     ./myapp

```

5078 The sequence of events is:

- 5079 1. The operating system loader loads the library specified in **LD_PRELOAD**.
- 5080 2. The call to **acc_register_library** in the OpenACC runtime is resolved to the routine
5081 in the loaded tools library.
- 5082 3. OpenACC runtime calls the **acc_register_library** routine in that library.
- 5083 4. Your **acc_register_library** routine calls **acc_callback_register** for each event
5084 to be monitored.
- 5085 5. **acc_callback_register** records the callback routines.
- 5086 6. The program runs, and your callback routines are invoked at the appropriate events.

5087 In this mode, only a single tools library is supported, since only one **acc_register_library**
5088 initialization routine will get resolved by the dynamic loader.

5089 5.3.5 Application-Controlled Initialization

5090 An alternative to default initialization is to have the application itself call the library initialization
5091 routine, which then calls **acc_callback_register** for each appropriate event. The library
5092 may be statically linked to the application or your application may dynamically load the library.

5093 The sequence of events is:

- 5094 1. Your application calls the library initialization routine.
- 5095 2. The library initialization routine calls **acc_callback_register** for each event to be
5096 monitored.
- 5097 3. **acc_callback_register** records the callback routines.
- 5098 4. The program runs, and your callback routines are invoked at the appropriate events.

5099 In this mode, multiple tools libraries can be supported, with each library initialization routine in-
5100 voked by the application.

5101 5.4 Registering Event Callbacks

5102 This section describes how to register and unregister callbacks, temporarily disabling and enabling
5103 callbacks, the behavior of dynamic registration and unregistration, and requirements on an Open-
5104 ACC implementation to correctly support the interface.

5105 5.4.1 Event Registration and Unregistration

5106 The library must call the registration routine **acc_callback_register** to register each call-
5107 back with the runtime. A simple example:

```

5108     extern void prof_data(acc_callback_info* profinfo,
5109                         acc_event_info* eventinfo, acc_api_info* apiinfo);

```

```

5110     extern void prof_launch(acc_callback_info* profinfo,
5111                          acc_event_info* eventinfo, acc_api_info* apiinfo);
5112     ...
5113     void acc_register_library(acc_prof_reg reg,
5114                          acc_prof_reg unreg, acc_prof_lookup_func lookup){
5115         reg(acc_ev_enqueue_upload_start, prof_data, acc_reg);
5116         reg(acc_ev_enqueue_download_start, prof_data, acc_reg);
5117         reg(acc_ev_enqueue_launch_start, prof_launch, acc_reg);
5118     }

```

5119 In this example the `prof_data` routine will be invoked for each data upload and download event,
5120 and the `prof_launch` routine will be invoked for each launch event. The `prof_data` routine
5121 might start out with:

```

5122     void prof_data(acc_callback_info* profinfo,
5123                  acc_event_info* eventinfo, acc_api_info* apiinfo){
5124         acc_data_event_info* datainfo;
5125         datainfo = (acc_data_event_info*)eventinfo;
5126         switch( datainfo->event_type ){
5127             case acc_ev_enqueue_upload_start :
5128                 ...
5129         }
5130     }

```

5131 Multiple Callbacks

5132 Multiple callback routines can be registered on the same event:

```

5133     acc_callback_register(acc_ev_enqueue_upload_start,
5134                          prof_data, acc_reg);
5135     acc_callback_register(acc_ev_enqueue_upload_start,
5136                          prof_up, acc_reg);

```

5137 For most events, the callbacks will be invoked in the order in which they are registered. However,
5138 *end* events, named `acc_ev_..._end`, invoke callbacks in the reverse order. Essentially, each
5139 event has an ordered list of callback routines. A new callback routine is appended to the tail of the
5140 list for that event. For most events, that list is traversed from the head to the tail, but for *end*
5141 events, the list is traversed from the tail to the head.

5142 If a callback is registered, then later unregistered, then later still registered again, the second regis-
5143 tration is considered to be a new callback, and the callback routine will then be appended to the tail
5144 of the callback list for that event.

5145 Unregistering

5146 A matching call to `acc_callback_unregister` will remove that routine from the list of call-
5147 back routines for that event.

```

5148     acc_callback_unregister(acc_ev_enqueue_upload_start,
5149                          prof_data, acc_reg);
5150     // prof_data is on the callback list for acc_ev_enqueue_upload_start
5151     ...

```

```

5152     acc_callback_unregister(acc_ev_enqueue_upload_start,
5153                           prof_data, acc_reg);
5154     // prof_data is removed from the callback list
5155     // for acc_ev_enqueue_upload_start

```

5156 Each entry on the callback list must also have a *ref* count. This keeps track of how many times
5157 this routine was added to this event's callback list. If a routine is registered *n* times, it must be
5158 unregistered *n* times before it is removed from the list. Note that if a routine is registered multiple
5159 times for the same event, its *ref* count will be incremented with each registration, but it will only be
5160 invoked once for each event instance.

5161 5.4.2 Disabling and Enabling Callbacks

5162 A callback routine may be temporarily disabled on the callback list for an event, then later re-
5163 enabled. The behavior is slightly different than unregistering and later re-registering that event.
5164 When a routine is disabled and later re-enabled, the routine's position on the callback list for that
5165 event is preserved. When a routine is unregistered and later re-registered, the routine's position on
5166 the callback list for that event will move to the tail of the list. Also, unregistering a callback must be
5167 done *n* times if the callback routine was registered *n* times. In contrast, disabling, and enabling an
5168 event sets a toggle. Disabling a callback will immediately reset the toggle and disable calls to that
5169 routine for that event, even if it was enabled multiple times. Enabling a callback will immediately
5170 set the toggle and enable calls to that routine for that event, even if it was disabled multiple times.
5171 Registering a new callback initially sets the toggle.

5172 A call to **acc_callback_unregister** with a value of **acc_toggle** as the third argument
5173 will disable callbacks to the given routine. A call to **acc_callback_register** with a value of
5174 **acc_toggle** as the third argument will enable those callbacks.

```

5175     acc_callback_unregister(acc_ev_enqueue_upload_start,
5176                           prof_data, acc_toggle);
5177     // prof_data is disabled
5178     ...
5179     acc_callback_register(acc_ev_enqueue_upload_start,
5180                          prof_data, acc_toggle);
5181     // prof_data is re-enabled

```

5182 A call to either **acc_callback_unregister** or **acc_callback_register** to disable or
5183 enable a callback when that callback is not currently registered for that event will be ignored with
5184 no error.

5185 All callbacks for an event may be disabled (and re-enabled) by passing **NULL** to the second argument
5186 and **acc_toggle** to the third argument of **acc_callback_unregister** (and
5187 **acc_callback_register**). This sets a toggle for that event, which is distinct from the toggle
5188 for each callback for that event. While the event is disabled, no callbacks for that event will be
5189 invoked. Callbacks for that event can be registered, unregistered, enabled, and disabled while that
5190 event is disabled, but no callbacks will be invoked for that event until the event itself is enabled.
5191 Initially, all events are enabled.

```

5192     acc_callback_unregister(acc_ev_enqueue_upload_start,
5193                           prof_data, acc_toggle);
5194     // prof_data is disabled

```

```

5195     ...
5196     acc_callback_unregister(acc_ev_enqueue_upload_start,
5197                           NULL, acc_toggle);
5198     // acc_ev_enqueue_upload_start callbacks are disabled
5199     ...
5200     acc_callback_register(acc_ev_enqueue_upload_start,
5201                           prof_data, acc_toggle);
5202     // prof_data is re-enabled, but
5203     // acc_ev_enqueue_upload_start callbacks still disabled
5204     ...
5205     acc_callback_register(acc_ev_enqueue_upload_start,
5206                           prof_up, acc_reg);
5207     // prof_up is registered and initially enabled, but
5208     // acc_ev_enqueue_upload_start callbacks still disabled
5209     ...
5210     acc_callback_register(acc_ev_enqueue_upload_start,
5211                           NULL, acc_toggle);
5212     // acc_ev_enqueue_upload_start callbacks are enabled
5213 
```

5214 Finally, all callbacks can be disabled (and enabled) by passing the argument list (**acc_ev_none**,
5215 **NULL**, **acc_toggle**) to **acc_callback_unregister** (and **acc_callback_register**).
5216 This sets a global toggle disabling all callbacks, which is distinct from the toggle enabling callbacks
5217 for each event and the toggle enabling each callback routine.

5218 The behavior of passing **acc_ev_none** as the first argument and a non-**NULL** value as the second
5219 argument to **acc_callback_unregister** or **acc_callback_register** is not defined,
5220 and may be ignored by the runtime without error.

5221 All callbacks can be disabled (or enabled) for just the current thread by passing the argument list
5222 (**acc_ev_none**, **NULL**, **acc_toggle_per_thread**) to **acc_callback_unregister**
5223 (and **acc_callback_register**). This is the only thread-specific interface to
5224 **acc_callback_register** and **acc_callback_unregister**, all other calls to register,
5225 unregister, enable, or disable callbacks affect all threads in the application.

5226 5.5 Advanced Topics

5227 This section describes advanced topics such as dynamic registration and changes of the execution
5228 state for callback routines as well as the runtime and tool behavior for multiple host threads.

5229 5.5.1 Dynamic Behavior

5230 Callback routines may be registered or unregistered, enabled or disabled at any point in the execution
5231 of the program. Calls may appear in the library itself, during the processing of an event. The
5232 OpenACC runtime must allow for this case, where the callback list for an event is modified while
5233 that event is being processed.

5234 Dynamic Registration and Unregistration

5235 Calls to **acc_register** and **acc_unregister** may occur at any point in the application. A
5236 callback routine can be registered or unregistered from a callback routine, either the same routine

5237 or another routine, for a different event or the same event for which the callback was invoked. If a
5238 callback routine is registered for an event while that event is being processed, then the new callback
5239 routine will be added to the tail of the list of callback routines for this event. Some events (the
5240 **_end**) events process the callback routines in reverse order, from the tail to the head. For those
5241 events, adding a new callback routine will not cause the new routine to be invoked for this instance
5242 of the event. The other events process the callback routines in registration order, from the head
5243 to the tail. Adding a new callback routine for such an event will cause the runtime to invoke that
5244 newly registered callback routine for this instance of the event. Both the runtime and the library
5245 must implement and expect this behavior.

5246 If an existing callback routine is unregistered for an event while that event is being processed, that
5247 callback routine is removed from the list of callbacks for this event. For any event, if that callback
5248 routine had not yet been invoked for this instance of the event, it will not be invoked.

5249 Registering and unregistering a callback routine is a global operation and affects all threads, in a
5250 multithreaded application. See Section 5.4.1 Multiple Callbacks.

5251 **Dynamic Enabling and Disabling**

5252 Calls to **acc_register** and **acc_unregister** to enable and disable a specific callback for
5253 an event, enable or disable all callbacks for an event, or enable or disable all callbacks may occur
5254 at any point in the application. A callback routine can be enabled or disabled from a callback
5255 routine, either the same routine or another routine, for a different event or the same event for which
5256 the callback was invoked. If a callback routine is enabled for an event while that event is being
5257 processed, then the new callback routine will be immediately enabled. If it appears on the list of
5258 callback routines closer to the head (for **_end** events) or closer to the tail (for other events), that
5259 newly-enabled callback routine will be invoked for this instance of this event, unless it is disabled
5260 or unregistered before that callback is reached.

5261 If a callback routine is disabled for an event while that event is being processed, that callback routine
5262 is immediately disabled. For any event, if that callback routine had not yet been invoked for this in-
5263 stance of the event, it will not be invoked, unless it is enabled before that callback routine is reached
5264 in the list of callbacks for this event. If all callbacks for an event are disabled while that event is
5265 being processed, or all callbacks are disabled for all events while an event is being processed, then
5266 when this callback routine returns, no more callbacks will be invoked for this instance of the event.

5267 Registering and unregistering a callback routine is a global operation and affects all threads, in a
5268 multithreaded application. See Section 5.4.1 Multiple Callbacks.

5269 **5.5.2 OpenACC Events During Event Processing**

5270 OpenACC events may occur during event processing. This may be because of OpenACC API rou-
5271 tine calls or OpenACC constructs being reached during event processing, or because of multiple host
5272 threads executing asynchronously. Both the OpenACC runtime and the tool library must implement
5273 the proper behavior.

5274 **5.5.3 Multiple Host Threads**

5275 Many programs that use OpenACC also use multiple host threads, such as programs using the
5276 OpenMP API. The appearance of multiple host threads affects both the OpenACC runtime and the
5277 tools library.

5278 Runtime Support for Multiple Threads

5279 The OpenACC runtime must be thread-safe, and the OpenACC runtime implementation of this
5280 tools interface must also be thread-safe. All threads use the same set of callbacks for all events, so
5281 registering a callback from one thread will cause all threads to execute that callback. This means that
5282 managing the callback lists for each event must be protected from multiple simultaneous updates.
5283 This includes adding a callback to the tail of the callback list for an event, removing a callback from
5284 the list for an event, and incrementing or decrementing the *ref* count for a callback routine for an
5285 event.

5286 In addition, one thread may register, unregister, enable, or disable a callback for an event while
5287 another thread is processing the callback list for that event asynchronously. The exact behavior may
5288 be dependent on the implementation, but some behaviors are expected and others are disallowed.
5289 In the following examples, there are three callbacks, A, B, and C, registered for event E in that
5290 order, where callbacks A and B are enabled and callback C is temporarily disabled. Thread T1 is
5291 dynamically modifying the callbacks for event E while thread T2 is processing an instance of event
5292 E.

- 5293 • Suppose thread T1 unregisters or disables callback A for event E. Thread T2 may or may not
5294 invoke callback A for this event instance, but it must invoke callback B; if it invokes callback
5295 A, that must precede the invocation of callback B.
- 5296 • Suppose thread T1 unregisters or disables callback B for event E. Thread T2 may or may not
5297 invoke callback B for this event instance, but it must invoke callback A; if it invokes callback
5298 B, that must follow the invocation of callback A.
- 5299 • Suppose thread T1 unregisters or disables callback A and then unregisters or disables callback
5300 B for event E. Thread T2 may or may not invoke callback A and may or may not invoke
5301 callback B for this event instance, but if it invokes both callbacks, it must invoke callback A
5302 before it invokes callback B.
- 5303 • Suppose thread T1 unregisters or disables callback B and then unregisters or disables callback
5304 A for event E. Thread T2 may or may not invoke callback A and may or may not invoke
5305 callback B for this event instance, but if it invokes callback B, it must have invoked callback
5306 A for this event instance.
- 5307 • Suppose thread T1 is registering a new callback D for event E. Thread T2 may or may not
5308 invoke callback D for this event instance, but it must invoke both callbacks A and B. If it
5309 invokes callback D, that must follow the invocations of A and B.
- 5310 • Suppose thread T1 is enabling callback C for event E. Thread T2 may or may not invoke
5311 callback C for this event instance, but it must invoke both callbacks A and B. If it invokes
5312 callback C, that must follow the invocations of A and B.

5313 The `acc_callback_info` struct has a `thread_id` field, which the runtime must set to a
5314 unique value for each host thread, though it need not be the same as the OpenMP threadnum value.

5315 Library Support for Multiple Threads

5316 The tool library must also be thread-safe. The callback routine will be invoked in the context of the
5317 thread that reaches the event. The library may receive a callback from a thread T2 while it's still
5318 processing a callback, from the same event type or from a different event type, from another thread

5319 T1. The **acc_callback_info** struct has a **thread_id** field, which the runtime must set to a
5320 unique value for each host thread.

5321 If the tool library uses dynamic callback registration and unregistration, or callback disabling and
5322 enabling, recall that unregistering or disabling an event callback from one thread will unregister or
5323 disable that callback for all threads, and registering or enabling an event callback from any thread
5324 will register or enable it for all threads. If two or more threads register the same callback for the
5325 same event, the behavior is the same as if one thread registered that callback multiple times; see
5326 Section 5.4.1 Multiple Callbacks. The **acc_unregister** routine must be called as many times
5327 as **acc_register** for that callback/event pair in order to totally unregister it. If two threads
5328 register two different callback routines for the same event, unless the order of the registration calls
5329 is guaranteed by some synchronization method, the order in which the runtime sees the registration
5330 may differ for multiple runs, meaning the order in which the callbacks occur will differ as well.

6. Glossary

5331

5332 Clear and consistent terminology is important in describing any programming model. We define
5333 here the terms you must understand in order to make effective use of this document and the asso-
5334 ciated programming model. In particular, some terms used in this specification conflict with their
5335 usage in the base language specifications. When there is potential confusion, the term will appear
5336 here.

5337 **Accelerator** – a device attached to a CPU and to which the CPU can offload data and compute
5338 kernels to perform compute-intensive calculations.

5339 **Accelerator routine** – a procedure compiled for the accelerator with the **routine** directive.

5340 **Accelerator thread** – a thread of execution that executes on the accelerator; a single vector lane of
5341 a single worker of a single gang.

5342 **Aggregate datatype** – any non-scalar datatype such as array and composite datatypes. In Fortran,
5343 aggregate datatypes include arrays, derived types, character types. In C, aggregate datatypes include
5344 arrays, targets of pointers, structs, and unions. In C++, aggregate datatypes include arrays, targets
5345 of pointers, classes, structs, and unions.

5346 **Aggregate variables** – a variable of any non-scalar datatype, including array or composite variables.
5347 In Fortran, this includes any variable with allocatable or pointer attribute and character variables.

5348 **Async-argument** – an *async-argument* is a nonnegative scalar integer expression (*int* for C or C++,
5349 *integer* for Fortran), or one of the special values **acc_async_noval** or **acc_async_sync**.

5350 **Barrier** – a type of synchronization where all parallel execution units or threads must reach the
5351 barrier before any execution unit or thread is allowed to proceed beyond the barrier; modeled after
5352 the starting barrier on a horse race track.

5353 **Block construct** – a *block-construct*, as specified by the Fortran language.

5354 **Composite datatype** – a derived type in Fortran, or a **struct** or **union** type in C, or a **class**,
5355 **struct**, or **union** type in C++. (This is different from the use of the term *composite data type* in
5356 the C and C++ languages.)

5357 **Composite variable** – a variable of composite datatype. In Fortran, a composite variable must not
5358 have allocatable or pointer attributes.

5359 **Compute construct** – a *parallel construct*, *serial construct*, or *kernels construct*.

5360 **Compute intensity** – for a given loop, region, or program unit, the ratio of the number of arithmetic
5361 operations performed on computed data divided by the number of memory transfers required to
5362 move that data between two levels of a memory hierarchy.

5363 **Compute region** – a *parallel region*, *serial region*, or *kernels region*.

5364 **Construct** – a directive and the associated statement, loop, or structured block, if any.

5365 **CUDA** – the CUDA environment from NVIDIA, a C-like programming environment used to ex-
5366 plicitly control and program an NVIDIA GPU.

- 5367 **Current device** – the device represented by the *acc-current-device-type-var* and *acc-current-device-*
5368 *num-var* ICVs
- 5369 **Current device type** – the device type represented by the *acc-current-device-type-var* ICV
- 5370 **Data lifetime** – the lifetime of a data object in device memory, which may begin at the entry to
5371 a data region, or at an **enter data** directive, or at a data API call such as **acc_copyin** or
5372 **acc_create**, and which may end at the exit from a data region, or at an **exit data** directive,
5373 or at a data API call such as **acc_delete**, **acc_copyout**, or **acc_shutdown**, or at the end of
5374 the program execution.
- 5375 **Data region** – a *region* defined by a **data** construct, or an implicit data region for a function or
5376 subroutine containing OpenACC directives. Data constructs typically allocate device memory and
5377 copy data from host to device memory upon entry, and copy data from device to local memory and
5378 deallocate device memory upon exit. Data regions may contain other data regions and compute
5379 regions.
- 5380 **Default asynchronous queue** – the asynchronous activity queue represented in the *acc-default-*
5381 *async-var* ICV
- 5382 **Device** – a general reference to an accelerator or a multicore CPU.
- 5383 **Device memory** – memory attached to a device, logically and physically separate from the host
5384 memory.
- 5385 **Device thread** – a thread of execution that executes on any device.
- 5386 **Directive** – in C or C++, a **#pragma**, or in Fortran, a specially formatted comment statement, that
5387 is interpreted by a compiler to augment information about or specify the behavior of the program.
- 5388 **Discrete memory** – memory accessible from the local thread that is not accessible from the current
5389 device, or memory accessible from the current device that is not accessible from the local thread.
- 5390 **DMA** – Direct Memory Access, a method to move data between physically separate memories;
5391 this is typically performed by a DMA engine, separate from the host CPU, that can access the host
5392 physical memory as well as an IO device or other physical memory.
- 5393 **Exposed variable access** – with respect to a compute construct, any access to the data or address
5394 of a variable at a point within the compute construct where the variable is not private to a scope
5395 lexically enclosed within the compute construct. See Section 2.6.2.
- 5396 **false** – a condition that evaluates to zero in C or C++, or **.false.** in Fortran.
- 5397 **GPU** – a Graphics Processing Unit; one type of accelerator.
- 5398 **GPGPU** – General Purpose computation on Graphics Processing Units.
- 5399 **Host** – the main CPU that in this context may have one or more attached accelerators. The host
5400 CPU controls the program regions and data loaded into and executed on one or more devices.
- 5401 **Host thread** – a thread of execution that executes on the host.
- 5402 **Implicit data region** – the data region that is implicitly defined for a Fortran subprogram or C
5403 function. A call to a subprogram or function enters the implicit data region, and a return from the
5404 subprogram or function exits the implicit data region.

- 5405 **Kernel** – a nested loop executed in parallel by the accelerator. Typically the loops are divided into
5406 a parallel domain, and the body of the loop becomes the body of the kernel.
- 5407 **Kernels region** – a *region* defined by a **kernels** construct. A kernels region is a structured block
5408 which is compiled for the accelerator. The code in the kernels region will be divided by the compiler
5409 into a sequence of kernels; typically each loop nest will become a single kernel. A kernels region
5410 may require space in device memory to be allocated and data to be copied from local memory to
5411 device memory upon region entry, and data to be copied from device memory to local memory and
5412 space in device memory to be deallocated upon exit.
- 5413 **Level of parallelism** – a possible level of parallelism, which in OpenACC is gang, worker, vector,
5414 or sequential. One or more of gang, worker, and vector parallelism may appear on a loop con-
5415 struct. Sequential execution corresponds to no parallelism. The **gang, worker, vector,** and
5416 **seq** clauses specify the level of parallelism for a loop.
- 5417 **Local device** – the device where the *local thread* executes.
- 5418 **Local memory** – the memory associated with the *local thread*.
- 5419 **Local thread** – the host thread or the accelerator thread that executes an OpenACC directive or
5420 construct.
- 5421 **Loop trip count** – the number of times a particular loop executes.
- 5422 **MIMD** – a method of parallel execution (Multiple Instruction, Multiple Data) where different ex-
5423 ecution units or threads execute different instruction streams asynchronously with each other.
- 5424 **null pointer** – a C or C++ pointer variable with the value zero, **NULL**, or (in C++) **nullptr**, or a
5425 Fortran **pointer** variable that is not associated, or a Fortran **allocatable** variable that is not
5426 allocated.
- 5427 **OpenCL** – short for Open Compute Language, a developing, portable standard C-like programming
5428 environment that enables low-level general-purpose programming on GPUs and other accelerators.
- 5429 **Orphaned loop construct** - a **loop** construct that is not lexically contained in any compute con-
5430 struct, that is, that has no parent compute construct.
- 5431 **Parallel region** – a *region* defined by a **parallel** construct. A parallel region is a structured block
5432 which is compiled for the accelerator. A parallel region typically contains one or more work-sharing
5433 loops. A parallel region may require space in device memory to be allocated and data to be copied
5434 from local memory to device memory upon region entry, and data to be copied from device memory
5435 to local memory and space in device memory to be deallocated upon exit.
- 5436 **Parent compute construct** – for a **loop** construct, the **parallel, serial,** or **kernels** con-
5437 struct that lexically contains the **loop** construct and is the innermost compute construct that con-
5438 tains that **loop** construct, if any.
- 5439 **Partly present data** – a section of data for which some of the data is present in a single device
5440 memory section, but part of the data is either not present or is present in a different device memory
5441 section. For instance, if a subarray of an array is present, the array is partly present.
- 5442 **Present data** – data for which the sum of the structured and dynamic reference counters is greater
5443 than zero in a single device memory section; see Section 2.6.7. A null pointer is defined as always
5444 present with a length of zero bytes.

- 5445 **Private data** – with respect to an iterative loop, data which is used only during a particular loop
5446 iteration. With respect to a more general region of code, data which is used within the region but is
5447 not initialized prior to the region and is re-initialized prior to any use after the region.
- 5448 **Procedure** – in C or C++, a function or C++ lambda; in Fortran, a subroutine or function.
- 5449 **Region** – all the code encountered during an instance of execution of a construct. A region includes
5450 any code in called routines, and may be thought of as the dynamic extent of a construct. This may
5451 be a *parallel region*, *serial region*, *kernels region*, *data region*, or *implicit data region*.
- 5452 **Scalar** – a variable of scalar datatype. In Fortran, scalars must not have allocatable or pointer
5453 attributes.
- 5454 **Scalar datatype** – an intrinsic or built-in datatype that is not an array or aggregate datatype. In For-
5455 tran, scalar datatypes are integer, real, double precision, complex, or logical. In C, scalar datatypes
5456 are char (signed or unsigned), int (signed or unsigned, with optional short, long or long long at-
5457 tribute), enum, float, double, long double, `_Complex` (with optional float or long attribute), or any
5458 pointer datatype. In C++, scalar datatypes are char (signed or unsigned), `wchar_t`, int (signed or
5459 unsigned, with optional short, long or long long attribute), enum, bool, float, double, long double,
5460 or any pointer datatype. Not all implementations or targets will support all of these datatypes.
- 5461 **Serial region** – a *region* defined by a **serial** construct. A serial region is a structured block which
5462 is compiled for the accelerator. A serial region contains code that is executed by a single gang of a
5463 single worker with a vector length of one. A serial region may require space in device memory to be
5464 allocated and data to be copied from local memory to device memory upon region entry, and data
5465 to be copied from device memory to local memory and space in device memory to be deallocated
5466 upon exit.
- 5467 **Shared memory** – memory that is accessible from both the local thread and the current device.
- 5468 **SIMD** – a method of parallel execution (single-instruction, multiple-data) where the same instruc-
5469 tion is applied to multiple data elements simultaneously.
- 5470 **SIMD operation** – a *vector operation* implemented with SIMD instructions.
- 5471 **Structured block** – in C or C++, an executable statement, possibly compound, with a single entry
5472 at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single
5473 entry at the top and a single exit at the bottom.
- 5474 **Thread** – a host CPU thread or an accelerator thread. On a host CPU, a thread is defined by a
5475 program counter and stack location; several host threads may comprise a process and share host
5476 memory. On an accelerator, a thread is any one vector lane of one worker of one gang.
- 5477 **true** – a condition that evaluates to nonzero in C or C++, or `.true.` in Fortran.
- 5478 **var** – the name of a variable (scalar, array, or composite variable), or a subarray specification, or an
5479 array element, or a composite variable member, or the name of a Fortran common block between
5480 slashes.
- 5481 **Vector operation** – a single operation or sequence of operations applied uniformly to each element
5482 of an array.
- 5483 **Visible data clause** – with respect to a compute construct, any data clause on the compute construct,
5484 a lexically containing **data** construct, or a visible **declare** directive. See Section 2.6.2.

- 5485 **Visible default clause** – with respect to a compute construct, the nearest **default** clause ap-
5486 pearing on the compute construct or a lexically containing **data** construct. See Section 2.6.2.
- 5487 **Visible device copy** – a copy of a variable, array, or subarray allocated in device memory that is
5488 visible to the program unit being compiled.

5489 **A. Recommendations for Implementers**

5490 This section gives recommendations for standard names and extensions to use for implementations
5491 for specific targets and target platforms, to promote portability across such implementations, and
5492 recommended options that programmers find useful. While this appendix is not part of the Open-
5493 ACC specification, implementations that provide the functionality specified herein are strongly rec-
5494 ommended to use the names in this section. The first subsection describes devices, such as NVIDIA
5495 GPUs. The second subsection describes additional API routines for target platforms, such as CUDA
5496 and OpenCL. The third subsection lists several recommended options for implementations.

5497 **A.1 Target Devices**

5498 **A.1.1 NVIDIA GPU Targets**

5499 This section gives recommendations for implementations that target NVIDIA GPU devices.

5500 **Accelerator Device Type**

5501 These implementations should use the name **acc_device_nvidia** for the **acc_device_t**
5502 type or return values from OpenACC Runtime API routines.

5503 **ACC_DEVICE_TYPE**

5504 An implementation should use the case-insensitive name **nvidia** for the environment variable
5505 **ACC_DEVICE_TYPE**.

5506 **device_type clause argument**

5507 An implementation should use the case-insensitive name **nvidia** as the argument to the **device_type**
5508 clause.

5509 **A.1.2 AMD GPU Targets**

5510 This section gives recommendations for implementations that target AMD GPUs.

5511 **Accelerator Device Type**

5512 These implementations should use the name **acc_device_radeon** for the **acc_device_t**
5513 type or return values from OpenACC Runtime API routines.

5514 **ACC_DEVICE_TYPE**

5515 These implementations should use the case-insensitive name **radeon** for the environment variable
5516 **ACC_DEVICE_TYPE**.

5517 **device_type clause argument**

5518 An implementation should use the case-insensitive name **radeon** as the argument to the **device_type**
5519 clause.

5520 **A.1.3 Multicore Host CPU Target**

5521 This section gives recommendations for implementations that target the multicore host CPU.

5522 **Accelerator Device Type**

5523 These implementations should use the name **acc_device_host** for the **acc_device_t** type
5524 or return values from OpenACC Runtime API routines.

5525 **ACC_DEVICE_TYPE**

5526 These implementations should use the case-insensitive name **host** for the environment variable
5527 **ACC_DEVICE_TYPE**.

5528 **device_type clause argument**

5529 An implementation should use the case-insensitive name **host** as the argument to the **device_type**
5530 clause.

5531 **routine directive**

5532 Given a **routine** directive for a procedure, an implementation should:

- 5533 • Suppress the procedure's compilation for the multicore host CPU if a **nohost** clause appears.
- 5534 • Ignore any **bind** clause when compiling the procedure for the multicore host CPU.
- 5535 • Disallow a **bind** clause to appear after a **device_type (host)** clause.

5536 **A.2 API Routines for Target Platforms**

5537 These runtime routines allow access to the interface between the OpenACC runtime API and the
5538 underlying target platform. An implementation may not implement all these routines, but if it
5539 provides this functionality, it should use these function names.

5540 **A.2.1 NVIDIA CUDA Platform**

5541 This section gives runtime API routines for implementations that target the NVIDIA CUDA Run-
5542 time or Driver API.

5543 **acc_get_current_cuda_device**

5544 **Summary**

5545 The **acc_get_current_cuda_device** routine returns the NVIDIA CUDA device handle for
5546 the current device.

5547 **Format**

5548 C or C++:

```
5549 void* acc_get_current_cuda_device ();
```

5550 acc_get_current_cuda_context**5551 Summary**

5552 The **acc_get_current_cuda_context** routine returns the NVIDIA CUDA context handle
5553 in use for the current device.

5554 Format

5555 C or C++:

```
5556     void* acc_get_current_cuda_context ();
```

5557 acc_get_cuda_stream**5558 Summary**

5559 The **acc_get_cuda_stream** routine returns the NVIDIA CUDA stream handle in use for the
5560 current device for the asynchronous activity queue associated with the **async** argument. This
5561 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5562 Format

5563 C or C++:

```
5564     void* acc_get_cuda_stream ( int async );
```

5565 acc_set_cuda_stream**5566 Summary**

5567 The **acc_set_cuda_stream** routine sets the NVIDIA CUDA stream handle the current device
5568 for the asynchronous activity queue associated with the **async** argument. This argument must be
5569 an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5570 Format

5571 C or C++:

```
5572     void acc_set_cuda_stream ( int async, void* stream );
```

5573 A.2.2 OpenCL Target Platform

5574 This section gives runtime API routines for implementations that target the OpenCL API on any
5575 device.

5576 acc_get_current_opengl_device**5577 Summary**

5578 The **acc_get_current_opengl_device** routine returns the OpenCL device handle for the
5579 current device.

5580 Format

5581 C or C++:

```
5582     void* acc_get_current_opengl_device ();
```

5583 acc_get_current_opengl_context**5584 Summary**

5585 The **acc_get_current_opengl_context** routine returns the OpenCL context handle in use
5586 for the current device.

5587 **Format**

5588 C or C++:

5589 `void* acc_get_current_opengl_context ();`5590 **acc_get_opengl_queue**5591 **Summary**

5592 The `acc_get_opengl_queue` routine returns the OpenCL command queue handle in use for
 5593 the current device for the asynchronous activity queue associated with the `async` argument. This
 5594 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5595 **Format**

5596 C or C++:

5597 `cl_command_queue acc_get_opengl_queue (int async);`5598 **acc_set_opengl_queue**5599 **Summary**

5600 The `acc_set_opengl_queue` routine returns the OpenCL command queue handle in use for
 5601 the current device for the asynchronous activity queue associated with the `async` argument. This
 5602 argument must be an *async-argument* as defined in Section 2.16 Asynchronous Behavior.

5603 **Format**

5604 C or C++:

5605 `void acc_set_opengl_queue (int async, cl_command_queue cmdqueue`
 5606 `);`

5607 **A.3 Recommended Options and Diagnostics**

5608 This section recommends options and diagnostics for implementations. Possible ways to implement
 5609 the options include command-line options to a compiler or settings in an IDE.

5610 **A.3.1 C Pointer in Present clause**

5611 This revision of OpenACC clarifies the construct:

```
5612 void test(int n ){
5613     float* p;
5614     ...
5615     #pragma acc data present(p)
5616     {
5617         // code here...
5618     }
```

5619 This example tests whether the pointer `p` itself is present in the current device memory. Implemen-
 5620 tations before this revision commonly implemented this by testing whether the pointer target `p[0]`
 5621 was present in the current device memory, and this appears in many programs assuming such. Until
 5622 such programs are modified to comply with this revision, an option to implement `present(p)` as
 5623 `present(p[0])` for C pointers may be helpful to users.

5624 **A.3.2 Nonconforming Applications and Implementations**

5625 Where feasible, implementations should diagnose OpenACC applications that do not conform with
5626 this specification's syntactic or semantic restrictions. Many but not all of these restrictions appear
5627 in lists entitled "Restrictions."

5628 While compile-time diagnostics are preferable (e.g., invalid clauses on a directive), some cases of
5629 nonconformity are more feasible to diagnose at run time (e.g., see Section 1.5). Where implemen-
5630 tations are not able to diagnose nonconformity reliably (e.g., an **independent** clause on a loop
5631 with data-dependent loop iterations), they might offer no diagnostics, or they might diagnose only
5632 subcases.

5633 In order to support OpenACC extensions, some implementations intentionally accept nonconform-
5634 ing OpenACC applications without issuing diagnostics by default, and some implementations accept
5635 conforming OpenACC applications but interpret their semantics differently than as detailed in this
5636 specification. To promote program portability across implementations, implementations should pro-
5637 vide an option to disable or report uses of these extensions. Some such extensions and diagnostics
5638 are described in detail in the remainder of this section.

5639 **A.3.3 Automatic Data Attributes**

5640 Some implementations provide autoscoping or other analysis to automatically determine a variable's
5641 data attributes, including the addition of reduction, private, and firstprivate clauses. To promote
5642 program portability across implementations, it would be helpful to provide an option to disable
5643 the automatic determination of data attributes or report which variables' data attributes are not as
5644 defined in Section 2.6.

5645 **A.3.4 Routine Directive with a Name**

5646 In C and C++, if a **routine** directive with a name appears immediately before a procedure dec-
5647 laration or definition with that name, it does not necessarily apply to that procedure according to
5648 Section 2.15.1 and C and C++ name resolution. Implementations should issue diagnostics in the
5649 following two cases:

- 5650 1. When no procedure with that name is already in scope, the directive is nonconforming, so
5651 implementations should issue a compile-time error diagnostic regardless of the following
5652 procedure. For example:

```
5653     #pragma acc routine(f) seq // compile-time error
5654     void f();
```

- 5655 2. When a procedure with that name is in scope and it is not the same procedure as the immedi-
5656 ately following procedure declaration or definition, the resolution of the name can be confus-
5657 ing. Implementations should then issue a compile-time warning diagnostic even though the
5658 application is conforming. For example:

```
5659     void g(); // routine directive applies
5660     namespace NS {
5661         #pragma acc routine(g) seq // compile-time warning
5662         void g(); // routine directive does not apply
5663     }
```

5664 The diagnostic in this case should suggest the programmer either (1) relocate the **routine**
5665 directive so that it more clearly applies to the procedure that is in scope or (2) remove the
5666 name from the **routine** directive so that it applies to the following procedure.

Index

- 5667 **_OPENACC**, 28, 129

- 5668 **acc-current-device-num-var**, 28
- 5669 **acc-current-device-type-var**, 28
- 5670 **acc-default-async-var**, 28, 87
- 5671 **acc_async_noval**, 87
- 5672 **acc_async_sync**, 87
- 5673 **acc_device_host**, 154
- 5674 **ACC_DEVICE_NUM**, 29, 121
- 5675 **acc_device_nvidia**, 153
- 5676 **acc_device_radeon**, 153
- 5677 **ACC_DEVICE_TYPE**, 29, 121, 153, 154
- 5678 **ACC_PROFLIB**, 121
- 5679 accelerator routine, 83
- 5680 action
 - 5681 attach, 45, 49
 - 5682 copyin, 48
 - 5683 copyout, 48
 - 5684 create, 48
 - 5685 delete, 48
 - 5686 detach, 45, 49
 - 5687 immediate, 49
 - 5688 present decrement, 47
 - 5689 present increment, 47
- 5690 AMD GPU target, 153
- 5691 **async** clause, 42, 43, 81, 89
- 5692 **async** queue, 11
- 5693 *async-argument*, 88, 89
- 5694 asynchronous execution, 11, 87
- 5695 **atomic** construct, 69
- 5696 attach action, 45, 49
- 5697 **attach** clause, 54
- 5698 attachment counter, 44
- 5699 **auto** clause, 61
- 5700 portability, 60
- 5701 autoscoping, 157

- 5702 barrier synchronization, 11, 32, 34, 147
- 5703 **bind** clause, 84
- 5704 block construct, 147

- 5705 **cache** directive, 67
- 5706 **capture** clause, 72
- 5707 **collapse** clause, 58
- 5708 common block, 45, 73, 74, 87
- 5709 compiler options, 156

- 5710 compute construct, 147
- 5711 compute region, 147
- 5712 construct, 147
 - 5713 **atomic**, 69
 - 5714 compute, 147
 - 5715 **data**, 40, 45
 - 5716 **host_data**, 55
 - 5717 **kernels**, 33, 45
 - 5718 **kernels loop**, 68
 - 5719 **parallel**, 31, 45
 - 5720 **parallel loop**, 68
 - 5721 **serial**, 32, 45
 - 5722 **serial loop**, 68
- 5723 **copy** clause, 39, 51
- 5724 copyin action, 48
- 5725 **copyin** clause, 51
- 5726 copyout action, 48
- 5727 **copyout** clause, 52
- 5728 create action, 48
- 5729 **create** clause, 53, 75
- 5730 CUDA, 11, 12, 147, 153, 154

- 5731 data attribute
 - 5732 explicitly determined, 37
 - 5733 implicitly determined, 73
 - 5734 predetermined, 37, 38
- 5735 data clause, 45
 - 5736 visible, 38, 150
- 5737 **data** construct, 40, 45
- 5738 data lifetime, 148
- 5739 data region, 40, 148
 - 5740 implicit, 40
- 5741 data-independent **loop** construct, 57
- 5742 **declare** directive, 73
- 5743 **default** clause, 37, 42
 - 5744 visible, 38, 151
- 5745 **default (none)** clause, 38
- 5746 default(present), 39
- 5747 delete action, 48
- 5748 **delete** clause, 54
- 5749 detach action, 45, 49
 - 5750 immediate, 49
- 5751 **detach** clause, 55
- 5752 **device** clause, 81
- 5753 **device_resident** clause, 74
- 5754 **device_type** clause, 29, 45, 153, 154

- 5755 **deviceptr** clause, 45, 50
- 5756 diagnostics, 156
- 5757 direct memory access, 11, 148
- 5758 DMA, 11, 148
- 5759 **enter data** directive, 42, 45
- 5760 environment variable
- 5761 **_OPENACC**, 28
- 5762 **ACC_DEVICE_NUM**, 29, 121
- 5763 **ACC_DEVICE_TYPE**, 29, 121, 153, 154
- 5764 **ACC_PROFLIB**, 121
- 5765 **exit data** directive, 42, 45
- 5766 explicitly determined data attribute, 37
- 5767 exposed variable access, 38, 148
- 5768 extensions, 157
- 5769 **firstprivate** clause, 36, 39
- 5770 gang, 32
- 5771 **gang** clause, 59, 83
- 5772 implicit, 59
- 5773 gang parallelism, 10
- 5774 *gang-arg*, 57
- 5775 gang-partitioned mode, 10
- 5776 optimizations, 60
- 5777 gang-redundant mode, 10, 32
- 5778 GR mode, 10
- 5779 **host**, 154
- 5780 **host** clause, 80
- 5781 **host_data** construct, 55
- 5782 ICV, 28
- 5783 **if** clause, 41, 43, 77–79, 81, 90
- 5784 immediate detach action, 49
- 5785 implicit data region, 40
- 5786 implicit **gang** clause, 59
- 5787 implicitly determined data attribute, 37
- 5788 **independent** clause, 61
- 5789 **init** directive, 76
- 5790 internal control variable, 28
- 5791 **kernels** construct, 33, 45
- 5792 **kernels loop** construct, 68
- 5793 level of parallelism, 10, 149
- 5794 **link** clause, 45, 75
- 5795 local device, 11
- 5796 local memory, 11
- 5797 local thread, 11
- 5798 **loop** construct, 56
- 5799 data-independent, 57
- 5800 orphaned, 57
- 5801 sequential, 57
- 5802 **no_create** clause, 53
- 5803 **nohost** clause, 84
- 5804 nonconformity, 157
- 5805 **num_gangs** clause, 35
- 5806 **num_workers** clause, 35
- 5807 **nvidia**, 153
- 5808 NVIDIA GPU target, 153
- 5809 OpenCL, 11, 12, 149, 153, 155
- 5810 optimizations
- 5811 gang-partitioned mode, 60
- 5812 orphaned **loop** construct, 57
- 5813 **parallel** construct, 31, 45
- 5814 **parallel loop** construct, 68
- 5815 parallelism
- 5816 level, 10, 149
- 5817 parent compute construct, 57
- 5818 pointer in **present** clause, 156
- 5819 portability
- 5820 **auto** clause, 60
- 5821 predetermined data attribute, 37, 38
- 5822 **present** clause, 39, 45, 50
- 5823 pointer, 156
- 5824 present decrement action, 47
- 5825 present increment action, 47
- 5826 **private** clause, 36, 62
- 5827 **radeon**, 153
- 5828 **read** clause, 72
- 5829 **reduction** clause, 36, 63
- 5830 reference counter, 44
- 5831 region
- 5832 compute, 147
- 5833 data, 40, 148
- 5834 implicit data, 40
- 5835 **routine** directive, 82, 157
- 5836 **self** clause, 80
- 5837 sentinel, 27
- 5838 **seq** clause, 61, 84
- 5839 sequential **loop** construct, 57
- 5840 **serial** construct, 32, 45
- 5841 **serial loop** construct, 68

- 5842 **shutdown** directive, 77
- 5843 *size-expr*, 57
- 5844 structured-block, 150

- 5845 thread, 150
- 5846 **tile** clause, 61

- 5847 **update** clause, 72
- 5848 **update** directive, 80
- 5849 **use_device** clause, 56

- 5850 **vector** clause, 60, 84
- 5851 vector lane, 32
- 5852 vector parallelism, 10
- 5853 vector-partitioned mode, 10
- 5854 vector-single mode, 10
- 5855 **vector_length** clause, 35
- 5856 visible data clause, 38, 150
- 5857 visible **default** clause, 38, 151
- 5858 visible device copy, 151
- 5859 VP mode, 10
- 5860 VS mode, 10

- 5861 **wait** clause, 42, 44, 81, 89
- 5862 **wait** directive, 89
- 5863 worker, 32
- 5864 **worker** clause, 60, 83
- 5865 worker parallelism, 10
- 5866 worker-partitioned mode, 10
- 5867 worker-single mode, 10
- 5868 WP mode, 10
- 5869 WS mode, 10