

1

# **The OpenACC<sup>®</sup> Application Programming Interface**

2

3

**Version 3.1**

4

OpenACC-Standard.org

5

November, 2020

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 © 2011-2020 OpenACC-Standard.org. All rights reserved.

# 11 Contents

12	<b>1. Introduction</b>	<b>9</b>
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. Conventions used in this document . . . . .	13
18	1.6. Organization of this document . . . . .	13
19	1.7. References . . . . .	14
20	1.8. Changes from Version 1.0 to 2.0 . . . . .	15
21	1.9. Corrections in the August 2013 document . . . . .	17
22	1.10. Changes from Version 2.0 to 2.5 . . . . .	17
23	1.11. Changes from Version 2.5 to 2.6 . . . . .	18
24	1.12. Changes from Version 2.6 to 2.7 . . . . .	19
25	1.13. Changes from Version 2.7 to 3.0 . . . . .	19
26	1.14. Changes from Version 3.0 to 3.1 . . . . .	21
27	1.15. Topics Deferred For a Future Revision . . . . .	22
28	<b>2. Directives</b>	<b>25</b>
29	2.1. Directive Format . . . . .	25
30	2.2. Conditional Compilation . . . . .	26
31	2.3. Internal Control Variables . . . . .	26
32	2.3.1. Modifying and Retrieving ICV Values . . . . .	26
33	2.4. Device-Specific Clauses . . . . .	27
34	2.5. Compute Constructs . . . . .	28
35	2.5.1. Parallel Construct . . . . .	28
36	2.5.2. Serial Construct . . . . .	29
37	2.5.3. Kernels Construct . . . . .	30
38	2.5.4. Compute Construct Restrictions . . . . .	31
39	2.5.5. if clause . . . . .	32
40	2.5.6. self clause . . . . .	32
41	2.5.7. async clause . . . . .	32
42	2.5.8. wait clause . . . . .	32
43	2.5.9. num_gangs clause . . . . .	32
44	2.5.10. num_workers clause . . . . .	32
45	2.5.11. vector_length clause . . . . .	32
46	2.5.12. private clause . . . . .	33
47	2.5.13. firstprivate clause . . . . .	33
48	2.5.14. reduction clause . . . . .	33
49	2.5.15. default clause . . . . .	34
50	2.6. Data Environment . . . . .	34
51	2.6.1. Variables with Predetermined Data Attributes . . . . .	35
52	2.6.2. Variables with Implicitly Determined Data Attributes . . . . .	35
53	2.6.3. Data Regions and Data Lifetimes . . . . .	36
54	2.6.4. Data Structures with Pointers . . . . .	37

55	2.6.5. Data Construct . . . . .	37
56	2.6.6. Enter Data and Exit Data Directives . . . . .	38
57	2.6.7. Reference Counters . . . . .	40
58	2.6.8. Attachment Counter . . . . .	41
59	2.7. Data Clauses . . . . .	41
60	2.7.1. Data Specification in Data Clauses . . . . .	42
61	2.7.2. Data Clause Actions . . . . .	43
62	2.7.3. Data Clause Restrictions . . . . .	46
63	2.7.4. deviceptr clause . . . . .	46
64	2.7.5. present clause . . . . .	46
65	2.7.6. copy clause . . . . .	47
66	2.7.7. copyin clause . . . . .	47
67	2.7.8. copyout clause . . . . .	48
68	2.7.9. create clause . . . . .	49
69	2.7.10. no_create clause . . . . .	49
70	2.7.11. delete clause . . . . .	50
71	2.7.12. attach clause . . . . .	50
72	2.7.13. detach clause . . . . .	51
73	2.8. Host_Data Construct . . . . .	51
74	2.8.1. use_device clause . . . . .	52
75	2.8.2. if clause . . . . .	52
76	2.8.3. if_present clause . . . . .	52
77	2.9. Loop Construct . . . . .	52
78	2.9.1. collapse clause . . . . .	54
79	2.9.2. gang clause . . . . .	54
80	2.9.3. worker clause . . . . .	55
81	2.9.4. vector clause . . . . .	56
82	2.9.5. seq clause . . . . .	56
83	2.9.6. independent clause . . . . .	56
84	2.9.7. auto clause . . . . .	57
85	2.9.8. tile clause . . . . .	57
86	2.9.9. device_type clause . . . . .	57
87	2.9.10. private clause . . . . .	57
88	2.9.11. reduction clause . . . . .	57
89	2.10. Cache Directive . . . . .	61
90	2.11. Combined Constructs . . . . .	62
91	2.12. Atomic Construct . . . . .	63
92	2.13. Declare Directive . . . . .	67
93	2.13.1. device_resident clause . . . . .	69
94	2.13.2. create clause . . . . .	69
95	2.13.3. link clause . . . . .	70
96	2.14. Executable Directives . . . . .	70
97	2.14.1. Init Directive . . . . .	70
98	2.14.2. Shutdown Directive . . . . .	71
99	2.14.3. Set Directive . . . . .	72
100	2.14.4. Update Directive . . . . .	74
101	2.14.5. Wait Directive . . . . .	76
102	2.14.6. Enter Data Directive . . . . .	76

103	2.14.7. Exit Data Directive . . . . .	76
104	2.15. Procedure Calls in Compute Regions . . . . .	76
105	2.15.1. Routine Directive . . . . .	76
106	2.15.2. Global Data Access . . . . .	79
107	2.16. Asynchronous Behavior . . . . .	79
108	2.16.1. async clause . . . . .	79
109	2.16.2. wait clause . . . . .	80
110	2.16.3. Wait Directive . . . . .	80
111	2.17. Fortran Specific Behavior . . . . .	81
112	2.17.1. Optional Arguments . . . . .	81
113	2.17.2. Do Concurrent Construct . . . . .	82
114	<b>3. Runtime Library</b>	<b>83</b>
115	3.1. Runtime Library Definitions . . . . .	83
116	3.2. Runtime Library Routines . . . . .	84
117	3.2.1. acc_get_num_devices . . . . .	84
118	3.2.2. acc_set_device_type . . . . .	84
119	3.2.3. acc_get_device_type . . . . .	85
120	3.2.4. acc_set_device_num . . . . .	85
121	3.2.5. acc_get_device_num . . . . .	86
122	3.2.6. acc_get_property . . . . .	86
123	3.2.7. acc_init . . . . .	88
124	3.2.8. acc_shutdown . . . . .	88
125	3.2.9. acc_async_test . . . . .	89
126	3.2.10. acc_async_test_device . . . . .	89
127	3.2.11. acc_async_test_all . . . . .	90
128	3.2.12. acc_async_test_all_device . . . . .	90
129	3.2.13. acc_wait . . . . .	91
130	3.2.14. acc_wait_device . . . . .	91
131	3.2.15. acc_wait_async . . . . .	92
132	3.2.16. acc_wait_device_async . . . . .	92
133	3.2.17. acc_wait_all . . . . .	93
134	3.2.18. acc_wait_all_device . . . . .	93
135	3.2.19. acc_wait_all_async . . . . .	94
136	3.2.20. acc_wait_all_device_async . . . . .	94
137	3.2.21. acc_get_default_async . . . . .	94
138	3.2.22. acc_set_default_async . . . . .	95
139	3.2.23. acc_on_device . . . . .	95
140	3.2.24. acc_malloc . . . . .	96
141	3.2.25. acc_free . . . . .	96
142	3.2.26. acc_copyin . . . . .	96
143	3.2.27. acc_create . . . . .	97
144	3.2.28. acc_copyout . . . . .	98
145	3.2.29. acc_delete . . . . .	99
146	3.2.30. acc_update_device . . . . .	100
147	3.2.31. acc_update_self . . . . .	101
148	3.2.32. acc_map_data . . . . .	102
149	3.2.33. acc_unmap_data . . . . .	102

150	3.2.34. acc_deviceptr . . . . .	103
151	3.2.35. acc_hostptr . . . . .	103
152	3.2.36. acc_is_present . . . . .	103
153	3.2.37. acc_memcpy_to_device . . . . .	104
154	3.2.38. acc_memcpy_from_device . . . . .	104
155	3.2.39. acc_memcpy_device . . . . .	105
156	3.2.40. acc_attach . . . . .	105
157	3.2.41. acc_detach . . . . .	106
158	3.2.42. acc_memcpy_d2d . . . . .	106
159	<b>4. Environment Variables</b>	<b>109</b>
160	4.1. ACC_DEVICE_TYPE . . . . .	109
161	4.2. ACC_DEVICE_NUM . . . . .	109
162	4.3. ACC_PROFLIB . . . . .	109
163	<b>5. Profiling Interface</b>	<b>111</b>
164	5.1. Events . . . . .	111
165	5.1.1. Runtime Initialization and Shutdown . . . . .	112
166	5.1.2. Device Initialization and Shutdown . . . . .	112
167	5.1.3. Enter Data and Exit Data . . . . .	113
168	5.1.4. Data Allocation . . . . .	113
169	5.1.5. Data Construct . . . . .	114
170	5.1.6. Update Directive . . . . .	114
171	5.1.7. Compute Construct . . . . .	114
172	5.1.8. Enqueue Kernel Launch . . . . .	114
173	5.1.9. Enqueue Data Update (Upload and Download) . . . . .	115
174	5.1.10. Wait . . . . .	115
175	5.2. Callbacks Signature . . . . .	116
176	5.2.1. First Argument: General Information . . . . .	116
177	5.2.2. Second Argument: Event-Specific Information . . . . .	118
178	5.2.3. Third Argument: API-Specific Information . . . . .	121
179	5.3. Loading the Library . . . . .	122
180	5.3.1. Library Registration . . . . .	123
181	5.3.2. Statically-Linked Library Initialization . . . . .	123
182	5.3.3. Runtime Dynamic Library Loading . . . . .	124
183	5.3.4. Preloading with LD_PRELOAD . . . . .	125
184	5.3.5. Application-Controlled Initialization . . . . .	125
185	5.4. Registering Event Callbacks . . . . .	126
186	5.4.1. Event Registration and Unregistration . . . . .	126
187	5.4.2. Disabling and Enabling Callbacks . . . . .	127
188	5.5. Advanced Topics . . . . .	128
189	5.5.1. Dynamic Behavior . . . . .	128
190	5.5.2. OpenACC Events During Event Processing . . . . .	129
191	5.5.3. Multiple Host Threads . . . . .	130
192	<b>6. Glossary</b>	<b>133</b>

193	<b>A. Recommendations for Implementers</b>	<b>137</b>
194	A.1. Target Devices . . . . .	137
195	A.1.1. NVIDIA GPU Targets . . . . .	137
196	A.1.2. AMD GPU Targets . . . . .	137
197	A.1.3. Multicore Host CPU Target . . . . .	138
198	A.2. API Routines for Target Platforms . . . . .	138
199	A.2.1. NVIDIA CUDA Platform . . . . .	138
200	A.2.2. OpenCL Target Platform . . . . .	139
201	A.3. Recommended Options . . . . .	140
202	A.3.1. C Pointer in Present clause . . . . .	140
203	A.3.2. Automatic Data Attributes . . . . .	140





## 1. Introduction

This document describes the compiler directives, library routines, and environment variables that collectively define the OpenACC<sup>™</sup> 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,

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

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

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

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

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

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

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

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

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

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

### 313 1.3 Memory Model

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

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

- 328 • Memory bandwidth between host memory and accelerator memory determines the level of  
329 compute intensity required to effectively accelerate a given region of code.
- 330 • The user should be aware that a discrete device memory is usually significantly smaller than  
331 the host memory, prohibiting offloading regions of code that operate on very large amounts  
332 of data.

333     • Host addresses stored to pointers on the host may only be valid on the host; addresses stored  
334     to pointers in accelerator memory may only be valid on that device. Explicitly transferring  
335     pointer values between host and accelerator memory is not advised. Dereferencing host point-  
336     ers on an accelerator or dereferencing accelerator pointers on the host is likely to be invalid  
337     on such targets.

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

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

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

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

## 365 **1.4 Language Interoperability**

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

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

## 1.5 Conventions used in this document

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

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

**#pragma acc**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 1.6 Organization of this document

The rest of this document is organized as follows:

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

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

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

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

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

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

## 420 1.7 References

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

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

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

- 427 – Threads
- 428 – Thread-local storage
- 429 – Parallel memory model
- 430 – Atomic

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

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

- 433 – Thread related features

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

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

- 437 – Extern templates
- 438 – copy and rethrow exceptions
- 439 – memory model
- 440 – atomics
- 441 – move semantics

- 442        – `std::thread`
- 443        – thread-local storage
- 444        • ISO/IEC 14882:2014, *Information Technology – Programming Languages – C++*, (C++14).
- 445        • ISO/IEC 14882:2017, *Information Technology – Programming Languages – C++*, (C++17).
- 446        • ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran – Part*
- 447        *1: Base Language*, (Fortran 2003).
- 448        • ISO/IEC 1539-1:2010, *Information Technology – Programming Languages – Fortran – Part*
- 449        *1: Base Language*, (Fortran 2008).

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

- 451        – Coarrays
- 452        – Simply contiguous arrays rank remapping to rank>1 target
- 453        – Allocatable components of recursive type
- 454        – Polymorphic assignment
- 455        • ISO/IEC 1539-1:2018, *Information Technology – Programming Languages – Fortran – Part*
- 456        *1: Base Language*, (Fortran 2018).

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

- 458        – Interoperability with C
  - 459            \* C functions declared in ISO Fortran binding.h
  - 460            \* Assumed rank
- 461        – All additional parallel/coarray features
- 462        • *OpenMP Application Program Interface*, version 5.0, November 2018
- 463        • *NVIDIA CUDA<sup>™</sup> C Programming Guide*, version 11.1.1, October 2020
- 464        • *The OpenCL Specification*, version 2.2, Khronos OpenCL Working Group, July 2019

## 465 1.8 Changes from Version 1.0 to 2.0

- 466        • `_OPENACC` value updated to **201306**
- 467        • **default (none)** clause on **parallel** and **kernels** directives
- 468        • the implicit data attribute for scalars in **parallel** constructs has changed
- 469        • the implicit data attribute for scalars in loops with **loop** directives with the independent
- 470        attribute has been clarified
- 471        • **acc\_async\_sync** and **acc\_async\_noval** values for the **async** clause
- 472        • Clarified the behavior of the **reduction** clause on a **gang** loop
- 473        • Clarified allowable loop nesting (**gang** may not appear inside **worker**, which may not ap-
- 474        pear within **vector**)

- 475 • **wait** clause on **parallel**, **kernels** and **update** directives
- 476 • **async** clause on the **wait** directive
- 477 • **enter data** and **exit data** directives
- 478 • Fortran *common block* names may now appear in many data clauses
- 479 • **link** clause for the **declare** directive
- 480 • the behavior of the **declare** directive for global data
- 481 • the behavior of a data clause with a C or C++ pointer variable has been clarified
- 482 • predefined data attributes
- 483 • support for multidimensional dynamic C/C++ arrays
- 484 • **tile** and **auto** loop clauses
- 485 • **update self** introduced as a preferred synonym for **update host**
- 486 • **routine** directive and support for separate compilation
- 487 • **device\_type** clause and support for multiple device types
- 488 • nested parallelism using **parallel** or **kernels** region containing another **parallel** or **kernels** re-
- 489 **gion**
- 490 • **atomic** constructs
- 491 • new concepts: gang-redundant, gang-partitioned; worker-single, worker-partitioned; vector-
- 492 **single**, vector-partitioned; thread
- 493 • new API routines:
  - 494 – **acc\_wait**, **acc\_wait\_all** instead of **acc\_async\_wait** and **acc\_async\_wait\_all**
  - 495 – **acc\_wait\_async**
  - 496 – **acc\_copyin**, **acc\_present\_or\_copyin**
  - 497 – **acc\_create**, **acc\_present\_or\_create**
  - 498 – **acc\_copyout**, **acc\_delete**
  - 499 – **acc\_map\_data**, **acc\_unmap\_data**
  - 500 – **acc\_deviceptr**, **acc\_hostptr**
  - 501 – **acc\_is\_present**
  - 502 – **acc\_memcpy\_to\_device**, **acc\_memcpy\_from\_device**
  - 503 – **acc\_update\_device**, **acc\_update\_self**
- 504 • defined behavior with multiple host threads, such as with OpenMP
- 505 • recommendations for specific implementations
- 506 • clarified that no arguments are allowed on the **vector** clause in a **parallel** region



## 1.9 Corrections in the August 2013 document

- corrected the **atomic capture** syntax for C/C++
- fixed the name of the **acc\_wait** and **acc\_wait\_all** procedures
- fixed description of the **acc\_hostptr** procedure

## 1.10 Changes from Version 2.0 to 2.5

- The **\_OPENACC** value was updated to **201510**; see Section 2.2 Conditional Compilation.
- The **num\_gangs**, **num\_workers**, and **vector\_length** clauses are now allowed on the **kernels** construct; see Section 2.5.3 Kernels Construct.
- Reduction on C++ class members, array elements, and struct elements are explicitly disallowed; see Section 2.5.14 reduction clause.
- Reference counting is now used to manage the correspondence and lifetime of device data; see Section 2.6.7 Reference Counters.
- The behavior of the **exit data** directive has changed to decrement the dynamic reference counter. A new optional **finalize** clause was added to set the dynamic reference counter to zero. See Section 2.6.6 Enter Data and Exit Data Directives.
- The **copy**, **copyin**, **copyout**, and **create** data clauses were changed to behave like **present\_or\_copy**, etc. The **present\_or\_copy**, **pcopy**, **present\_or\_copyin**, **pcopyin**, **present\_or\_copyout**, **pcopyout**, **present\_or\_create**, and **pcreate** data clauses are no longer needed, though will be accepted for compatibility; see Section 2.7 Data Clauses.
- Reductions on orphaned gang loops are explicitly disallowed; see Section 2.9 Loop Construct.
- The description of the **loop auto** clause has changed; see Section 2.9.7 auto clause.
- Text was added to the **private** clause on a **loop** construct to clarify that a copy is made for each gang or worker or vector lane, not each thread; see Section 2.9.10 private clause.
- The description of the **reduction** clause on a **loop** construct was corrected; see Section 2.9.11 reduction clause.
- A restriction was added to the **cache** clause that all references to that variable must lie within the region being cached; see Section 2.10 Cache Directive.
- Text was added to the **private** and **reduction** clauses on a combined construct to clarify that they act like **private** and **reduction** on the **loop**, not **private** and **reduction** on the **parallel** or **reduction** on the **kernels**; see Section 2.11 Combined Constructs.
- The **declare create** directive with a Fortran **allocatable** has new behavior; see Section 2.13.2 create clause.
- New **init**, **shutdown**, **set** directives were added; see Section 2.14.1 Init Directive, 2.14.2 Shutdown Directive, and 2.14.3 Set Directive.
- A new **if\_present** clause was added to the **update** directive, which changes the behavior when data is not present from a runtime error to a no-op; see Section 2.14.4 Update Directive.

- 544 • The **routine bind** clause definition changed; see Section 2.15.1 Routine Directive.
- 545 • An **acc routine** without **gang/worker/vector/seq** is now defined as an error; see
- 546 Section 2.15.1 Routine Directive.
- 547 • A new **default (present)** clause was added for compute constructs; see Section 2.5.15
- 548 default clause.
- 549 • The Fortran header file **openacc\_lib.h** is no longer supported; the Fortran module **openacc**
- 550 should be used instead; see Section 3.1 Runtime Library Definitions.
- 551 • New API routines were added to get and set the default async queue value; see Section 3.2.21
- 552 `acc_get_default_async` and 3.2.22 `acc_set_default_async`.
- 553 • The **acc\_copyin**, **acc\_create**, **acc\_copyout**, and **acc\_delete** API routines were
- 554 changed to behave like **acc\_present\_or\_copyin**, etc. The **acc\_present\_or\_** names
- 555 are no longer needed, though will be supported for compatibility. See Sections 3.2.26 and fol-
- 556 lowing.
- 557 • Asynchronous versions of the data API routines were added; see Sections 3.2.26 and follow-
- 558 ing.
- 559 • A new API routine added, **acc\_memcpy\_device**, to copy from one device address to
- 560 another device address; see Section 3.2.37 `acc_memcpy_to_device`.
- 561 • A new OpenACC interface for profile and trace tools was added; see Chapter 5 Profiling Interface.

## 562 1.11 Changes from Version 2.5 to 2.6

- 563 • The **\_OPENACC** value was updated to **201711**.
- 564 • A new **serial** compute construct was added. See Section 2.5.2 Serial Construct.
- 565 • A new runtime API query routine was added. **acc\_get\_property** may be called from
- 566 the host and returns properties about any device. See Section 3.2.6.
- 567 • The text has clarified that if a variable is in a reduction which spans two or more nested loops,
- 568 each **loop** directive on any of those loops must have a **reduction** clause that contains the
- 569 variable; see Section 2.9.11 reduction clause.
- 570 • An optional **if** or **if\_present** clause is now allowed on the **host\_data** construct. See
- 571 Section 2.8 Host\_Data Construct.
- 572 • A new **no\_create** data clause is now allowed on compute and **data** constructs. See Sec-
- 573 tion 2.7.10 `no_create` clause.
- 574 • The behavior of Fortran optional arguments in data clauses and in routine calls has been
- 575 specified; see Section 2.17.1 Optional Arguments.
- 576 • The descriptions of some of the Fortran versions of the runtime library routines were simpli-
- 577 fied; see Section 3.2 Runtime Library Routines.
- 578 • To allow for manual deep copy of data structures with pointers, new *attach* and *detach* be-
- 579 havior was added to the data clauses, new **attach** and **detach** clauses were added, and
- 580 matching **acc\_attach** and **acc\_detach** runtime API routines were added; see Sections
- 581 2.6.4, 2.7.12-2.7.13 and 3.2.40-3.2.41.

- 582     • The Intel Coprocessor Offload Interface target and API routine sections were removed from  
583     the Section A Recommendations for Implementers, since Intel no longer produces this prod-  
584     uct.

## 585 1.12 Changes from Version 2.6 to 2.7

- 586     • The `_OPENACC` value was updated to **201811**.
- 587     • The specification allows for hosts that share some memory with the device but not all memory.  
588     The wording in the text now discusses whether local thread data is in shared memory (memory  
589     shared between the local thread and the device) or discrete memory (local thread memory that  
590     is not shared with the device), instead of shared-memory devices and non-shared memory  
591     devices. See Sections 1.3 Memory Model and 2.6 Data Environment.
- 592     • The text was clarified to allow an implementation that treats a multicore CPU as a device,  
593     either an additional device or the only device.
- 594     • The `readonly` modifier was added to the `copyin` data clause and `cache` directive. See  
595     Sections 2.7.7 and 2.10.
- 596     • The term *local device* was defined; see Section 1.2 Execution Model and the Glossary.
- 597     • The term *var* is used more consistently throughout the specification to mean a variable name,  
598     array name, subarray specification, array element, composite variable member, or Fortran  
599     common block name between slashes. Some uses of *var* allow only a subset of these options,  
600     and those limitations are given in those cases.
- 601     • The `self` clause was added to the compute constructs; see Section 2.5.6 self clause.
- 602     • The appearance of a `reduction` clause on a compute construct implies a `copy` clause for  
603     each reduction variable; see Sections 2.5.14 reduction clause and 2.11 Combined Constructs.
- 604     • The `default (none)` and `default (present)` clauses were added to the `data` con-  
605     struct; see Section 2.6.5 Data Construct.
- 606     • Data is defined to be *present* based on the values of the structured and dynamic reference  
607     counters; see Section 2.6.7 Reference Counters and the Glossary.
- 608     • The interaction of the `acc_map_data` and `acc_unmap_data` runtime API calls on the  
609     present counters is defined; see Section 2.7.2, 3.2.32, and 3.2.33.
- 610     • A restriction clarifying that a `host_data` construct must have at least one `use_device`  
611     clause was added.
- 612     • Arrays, subarrays and composite variables are now allowed in `reduction` clauses; see  
613     Sections 2.9.11 reduction clause and 2.5.14 reduction clause.
- 614     • Changed behavior of ICVs to support nested compute regions and host as a device semantics.  
615     See Section 2.3.

## 616 1.13 Changes from Version 2.7 to 3.0

- 617     • Updated `_OPENACC` value to **201911**.
- 618     • Updated the normative references to the most recent standards for all base languages. See  
619     Section 1.7.

- 620 • Changed the text to clarify uses and limitations of the **device\_type** clause and added  
621 examples; see Section 2.4.
- 622 • Clarified the conflict between the implicit **copy** clause for variables in a **reduction** clause  
623 and the implicit **firstprivate** for scalar variables not in a data clause but used in a  
624 **parallel** or **serial** construct; see Sections 2.5.1 and 2.5.2.
- 625 • Required at least one data clause on a **data** construct, an **enter data** directive, or an **exit**  
626 **data** directive; see Sections 2.6.5 and 2.6.6.
- 627 • Added text describing how a C++ *lambda* invoked in a compute region and the variables  
628 captured by the *lambda* are handled; see Section 2.6.2.
- 629 • Added a **zero** modifier to **create** and **copyout** data clauses that zeros the device memory  
630 after it is allocated; see Sections 2.7.8 and 2.7.9.
- 631 • Added a new restriction on the **loop** directive allowing only one of the **seq**, **independent**,  
632 and **auto** clauses to appear; see Section 2.9.
- 633 • Added a new restriction on the **loop** directive disallowing a **gang**, **worker**, or **vector**  
634 clause to appear if a **seq** clause appears; see Section 2.9.
- 635 • Allowed variables to be modified in an atomic region in a loop where the iterations must  
636 otherwise be data independent, such as loops with a **loop independent** clause or a **loop**  
637 directive in a **parallel** construct; see Sections 2.9.2, 2.9.3, 2.9.4, and 2.9.6.
- 638 • Clarified the behavior of the **auto** and **independent** clauses on the **loop** directive; see  
639 Sections 2.9.7 and 2.9.6.
- 640 • Clarified that an orphaned **loop** construct, or a **loop** construct in a **parallel** construct  
641 with no **auto** or **seq** clauses is treated as if an **independent** clause appears; see Sec-  
642 tion 2.9.6.
- 643 • For a variable in a **reduction** clause, clarified when the update to the original variable is  
644 complete, and added examples; see Section 2.9.11.
- 645 • Clarified that a variable in an orphaned **reduction** clause must be private; see Section 2.9.11.
- 646 • Required at least one clause on a **declare** directive; see Section 2.13.
- 647 • Added an **if** clause to **init**, **shutdown**, **set**, and **wait** directives; see Sections 2.14.1,  
648 2.14.2, 2.14.3, and 2.16.3.
- 649 • Required at least one clause on a **set** directive; see Section 2.14.3.
- 650 • Added a *devnum* modifier to the **wait** directive and clause to specify a device to which the  
651 wait operation applies; see Section 2.16.3.
- 652 • Allowed a **routine** directive to include a C++ *lambda* name or to appear before a C++  
653 *lambda* definition, and defined implicit **routine** directive behavior when a C++ *lambda* is  
654 called in a compute region or an *accelerator routine*; see Section 2.15.
- 655 • Added runtime API routine **acc\_memcpy\_d2d** for copying data directly between two de-  
656 vice arrays on the same or different devices; see Section 3.2.42.
- 657 • Defined the values for the **acc\_construct\_t** and **acc\_device\_api** enumerations for  
658 cross-implementation compatibility; see Sections 5.2.2 and 5.2.3.

- 659 • Changed the return type of `acc_set_cuda_stream` from `int` (values were not specified)  
660 to `void`; see Section A.2.1.
- 661 • Edited and expanded Section 1.15 Topics Deferred For a Future Revision.

## 662 1.14 Changes from Version 3.0 to 3.1

- 663 • Updated `_OPENACC` value to `202011`.
- 664 • Clarified that Fortran blank common blocks are not permitted and that same-named common  
665 blocks must have the same size. See Section 1.5.
- 666 • Clarified that a `parallel` construct's block is considered to start in gang-redundant mode  
667 even if there's just a single gang. See Section 2.5.1.
- 668 • Added support for the Fortran `BLOCK` construct. See Sections 2.5.1, 2.5.3, 2.6.1, 2.6.5, 2.8,  
669 2.13, and 6.
- 670 • Defined the `serial` construct in terms of the `parallel` construct to improve readability.  
671 Instead of defining it in terms of clauses `num_gangs (1) num_workers (1)`  
672 `vector_length (1)`, defined the `serial` construct as executing with a single gang of a  
673 single worker with a vector length of one. See Section 2.5.2.
- 674 • Consolidated compute construct restrictions into a new section to improve readability. See  
675 Section 2.5.4.
- 676 • Clarified that a `default` clause may appear at most once on a compute construct. See  
677 Section 2.5.15.
- 678 • Consolidated discussions of implicit data attributes on compute and combined constructs into  
679 a separate section. Clarified the conditions under which each data attribute is implied. See  
680 Section 2.6.2.
- 681 • Added a restriction that certain loop reduction variables must have explicit data clauses on  
682 their parent compute constructs. This change addresses portability across existing OpenACC  
683 implementations. See Sections 2.6.2 and A.3.2.
- 684 • Restored the OpenACC 2.5 behavior of the `present`, `copy`, `copyin`, `copyout`, `create`,  
685 `no_create`, `delete` data clauses at exit from a region, or on an `exit data` directive, as  
686 applicable, and `create` clause at exit from an implicit data region where a `declare` di-  
687 rective appears, and `acc_copyout`, `acc_delete` routines, such that no action is taken  
688 if the appropriate reference counter is zero, instead of a runtime error being issued if data is  
689 not 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, 3.2.28, and  
690 3.2.29.
- 691 • Clarified restrictions on loop forms that can be associated with `loop` constructs, including  
692 the case of C++ range-based `for` loops. See Section 2.9.
- 693 • Specified where `gang` clauses are implied on `loop` constructs. This change standardizes  
694 behavior of existing OpenACC implementations. See Section 2.9.2.
- 695 • Corrected C/C++ syntax for `atomic capture` with a structured block. See Section 2.12.
- 696 • Added the behavior of the Fortran `do concurrent` construct. See Section 2.17.2.

- 697 • Changed the Fortran run-time procedures: **acc\_device\_property** has been renamed to  
698 **acc\_device\_property\_kind** and **acc\_get\_property** uses a different integer kind  
699 for the result. See Section 3.2.
- 700 • Added or changed argument names for the Runtime Library routines to be descriptive and  
701 consistent. This mostly impacts Fortran programs, which can pass arguments by name. See  
702 Section 3.2.
- 703 • Replaced composite variable by aggregate variable in **reduction**, **default**, and **private**  
704 clauses and in implicitly determined data attributes; the new wording also includes Fortran  
705 character and allocatable/pointer variables. See glossary in Section 6.

## 706 1.15 Topics Deferred For a Future Revision

707 The following topics are under discussion for a future revision. Some of these are known to be  
708 important, while others will depend on feedback from users. Readers who have feedback or want  
709 to participate may send email to [feedback@openacc.org](mailto:feedback@openacc.org). No promises are made or implied that all  
710 these items will be available in a future revision.

- 711 • Directives to define implicit *deep copy* behavior for pointer-based data structures.
- 712 • Defined behavior when data in data clauses on a directive are aliases of each other.
- 713 • Clarifying when data becomes *present* or *not present* on the device for **enter data** or **exit**  
714 **data** directives with an **async** clause.
- 715 • Clarifying the behavior of Fortran **pointer** variables in data clauses.
- 716 • Allowing Fortran **pointer** variables to appear in **deviceptr** clauses.
- 717 • Defining the behavior of data clauses and runtime API routines for pointers that are **NULL**, or  
718 Fortran **pointer** variables that are not associated, or Fortran **allocatable** variables that  
719 are not allocated.
- 720 • Support for attaching C/C++ pointers that point to an address past the end of a memory region.
- 721 • Fully defined interaction with multiple host threads.
- 722 • Optionally removing the synchronization or barrier at the end of vector and worker loops.
- 723 • Allowing an **if** clause after a **device\_type** clause.
- 724 • A **shared** clause (or something similar) for the loop directive.
- 725 • Better support for multiple devices from a single thread, whether of the same type or of  
726 different types.
- 727 • An *auto* construct (by some name), to allow **kernels**-like auto-parallelization behavior  
728 inside **parallel** constructs or accelerator routines.
- 729 • A **begin declare ... end declare** construct that behaves like putting any global vari-  
730 ables declared inside the construct in a **declare** clause.
- 731 • Defining the behavior of additional parallelism constructs in the base languages when used  
732 inside a compute construct or accelerator routine.
- 733 • Optimization directives or clauses, such as an *unroll* directive or clause.

- 734 • Define runtime error behavior and allowing a user-defined error handlers.
- 735 • Extended reductions.
- 736 • Fortran bindings for all the API routines.
- 737 • A **linear** clause for the **loop** directive.
- 738 • Allowing two or more of **gang**, **worker**, **vector**, or **seq** clause on an **acc routine**  
739 directive.
- 740 • Requiring the implementation to imply an **acc routine** directive for procedures called  
741 within a compute construct or accelerator routine.
- 742 • A single list of all devices of all types, including the host device.
- 743 • A memory allocation API for specific types of memory, including device memory, host pinned  
744 memory, and unified memory.
- 745 • Bindings to other languages.





## 2. Directives

746

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

### 2.1 Directive Format

751

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

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

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

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

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

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

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

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

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

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

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

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

782 Only one *directive-name* can appear per directive, except that a combined directive name is consid-  
 783 ered a single *directive-name*. The order in which clauses appear is not significant unless otherwise

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

## 786 2.2 Conditional Compilation

787 The `_OPENACC` macro name is defined to have a value `yyyymm` where `yyyy` is the year and `mm` is  
788 the month designation of the version of the OpenACC directives supported by the implementation.  
789 This macro must be defined by a compiler only when OpenACC directives are enabled. The version  
790 described here is 202011.

## 791 2.3 Internal Control Variables

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

796 The ICVs are:

- 797 • `acc-current-device-type-var` - controls which type of device is used.
- 798 • `acc-current-device-num-var` - controls which device of the selected type is used.
- 799 • `acc-default-async-var` - controls which asynchronous queue is used when none appears in an  
800 `async` clause.

### 801 2.3.1 Modifying and Retrieving ICV Values

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

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

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

## 2.4 Device-Specific Clauses

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

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

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

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

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

### Syntax

The syntax of the **device\_type** clause is

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

The **device\_type** clause may be abbreviated to **dtype**.

---

### Examples

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

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

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

```
855     // non-conforming: gang and worker are not permitted together
856     #pragma acc routine gang device_type(foo) worker
857
858     // conforming: gang and worker apply to different device types
859     #pragma acc routine device_type(foo) worker \
860             device_type(*) gang
```

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

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

- 867 • The directive below is the same as the previous directive except that **num\_gangs(2)** has  
868 moved after **device\_type(\*)** and so now does not apply to **foo** or **bar**.

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

872 ▲  
873

## 874 2.5 Compute Constructs

### 875 2.5.1 Parallel Construct

#### 876 Summary

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

#### 878 Syntax

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

```
880     #pragma acc parallel [clause-list] new-line
881             structured block
882
```

883 and in Fortran, the syntax is

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

887 OR

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

```

890     [!$acc end parallel]
891 where clause is one of the following:
892     async [ ( int-expr ) ]
893     wait [ ( int-expr-list ) ]
894     num_gangs ( int-expr )
895     num_workers ( int-expr )
896     vector_length ( int-expr )
897     device_type ( device-type-list )
898     if ( condition )
899     self [ ( condition ) ]
900     reduction ( operator : var-list )
901     copy ( var-list )
902     copyin ( [ readonly: ] var-list )
903     copyout ( [ zero: ] var-list )
904     create ( [ zero: ] var-list )
905     no_create ( var-list )
906     present ( var-list )
907     deviceptr ( var-list )
908     attach ( var-list )
909     private ( var-list )
910     firstprivate ( var-list )
911     default ( none | present )

```

## 912 Description

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

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

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

925 The **copy**, **copyin**, **copyout**, **create**, **no\_create**, **present**, **deviceptr**, and **attach**  
926 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**  
927 clauses are described in Sections 2.5.12 and Sections 2.5.13. The **device\_type** clause is de-  
928 scribed in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described  
929 in Section 2.6.2. Restrictions are described in Section 2.5.4.

## 930 2.5.2 Serial Construct

931 **Summary**

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

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

937 **Syntax**

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

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

941

942 and in Fortran, the syntax is

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

946 OR

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

950 where *clause* is as for the **parallel** construct except that the **num\_gangs**, **num\_workers**, and  
 951 **vector\_length** clauses are not permitted.

952 **2.5.3 Kernels Construct**953 **Summary**

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

956 **Syntax**

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

```
958     #pragma acc kernels [ clause-list ] new-line
959         structured block
```

960

961 and in Fortran, the syntax is

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

965 OR

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

969 where *clause* is one of the following:

```

970   async [ ( int-expr ) ]
971   wait [ ( int-expr-list ) ]
972   num_gangs ( int-expr )
973   num_workers ( int-expr )
974   vector_length ( int-expr )
975   device_type ( device-type-list )
976   if ( condition )
977   self [ ( condition ) ]
978   copy ( var-list )
979   copyin ( [ readonly: ] var-list )
980   copyout ( [ zero: ] var-list )
981   create ( [ zero: ] var-list )
982   no_create ( var-list )
983   present ( var-list )
984   deviceptr ( var-list )
985   attach ( var-list )
986   default ( none | present )

```

#### 987 Description

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

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

995 The **copy**, **copyin**, **copyout**, **create**, **no\_create**, **present**, **deviceptr**, and **attach**  
 996 data clauses are described in Section 2.7 Data Clauses. The **device\_type** clause is described  
 997 in Section 2.4 Device-Specific Clauses. Implicitly determined data attributes are described in Sec-  
 998 tion 2.6.2. Restrictions are described in Section 2.5.4.

### 999 2.5.4 Compute Construct Restrictions

1000 The following restrictions apply to all compute constructs:

- 1001 • A program may not branch into or out of a compute construct.
- 1002 • A program must not depend on the order of evaluation of the clauses or on any side effects of  
 1003 the evaluations.
- 1004 • Only the **async**, **wait**, **num\_gangs**, **num\_workers**, and **vector\_length** clauses  
 1005 may follow a **device\_type** clause.
- 1006 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical  
 1007 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1008 • At most one **default** clause may appear, and it must have a value of either **none** or  
 1009 **present**.

### 1010 2.5.5 if clause

1011 The **if** clause is optional.

1012 When the *condition* in the **if** clause evaluates to nonzero in C or C++, or **.true.** in Fortran, the  
1013 region will execute on the current device. When the *condition* in the **if** clause evaluates to zero in  
1014 C or C++, or **.false.** in Fortran, the local thread will execute the region.

### 1015 2.5.6 self clause

1016 The **self** clause is optional.

1017 The **self** clause may have a single *condition-argument*. If the *condition-argument* is not present  
1018 it is assumed to be nonzero in C or C++, or **.true.** in Fortran. When both an **if** clause and a  
1019 **self** clause appear and the *condition* in the **if** clause evaluates to 0 in C or C++ or **.false.** in  
1020 Fortran, the **self** clause has no effect.

1021 When the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the region will execute  
1022 on the local device. When the *condition* in the **self** clause evaluates to zero in C or C++, or  
1023 **.false.** in Fortran, the region will execute on the current device.

### 1024 2.5.7 async clause

1025 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

### 1026 2.5.8 wait clause

1027 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

### 1028 2.5.9 num\_gangs clause

1029 The **num\_gangs** clause is allowed on the **parallel** and **kernels** constructs. The value of  
1030 the integer expression defines the number of parallel gangs that will execute the parallel region,  
1031 or that will execute each kernel created for the kernels region. If the clause does not appear, an  
1032 implementation-defined default will be used; the default may depend on the code within the con-  
1033 struct. The implementation may use a lower value than specified based on limitations imposed by  
1034 the target architecture.

### 1035 2.5.10 num\_workers clause

1036 The **num\_workers** clause is allowed on the **parallel** and **kernels** constructs. The value  
1037 of the integer expression defines the number of workers within each gang that will be active after  
1038 a gang transitions from worker-single mode to worker-partitioned mode. If the clause does not  
1039 appear, an implementation-defined default will be used; the default value may be 1, and may be  
1040 different for each **parallel** construct or for each kernel created for a **kernels** construct. The  
1041 implementation may use a different value than specified based on limitations imposed by the target  
1042 architecture.

### 1043 2.5.11 vector\_length clause

1044 The **vector\_length** clause is allowed on the **parallel** and **kernels** constructs. The value  
1045 of the integer expression defines the number of vector lanes that will be active after a worker transi-



1046 tions from vector-single mode to vector-partitioned mode. This clause determines the vector length  
1047 to use for vector or SIMD operations. If the clause does not appear, an implementation-defined  
1048 default will be used. This vector length will be used for loop constructs annotated with the **vector**  
1049 clause, as well as loops automatically vectorized by the compiler. The implementation may use a  
1050 different value than specified based on limitations imposed by the target architecture.

## 1051 2.5.12 private clause

1052 The **private** clause is allowed on the **parallel** and **serial** constructs; it declares that a copy  
1053 of each item on the list will be created for each gang.

### 1054 Restrictions

- 1055 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private**  
1056 clauses.

## 1057 2.5.13 firstprivate clause

1058 The **firstprivate** clause is allowed on the **parallel** and **serial** constructs; it declares that  
1059 a copy of each item on the list will be created for each gang, and that the copy will be initialized with  
1060 the value of that item on the local thread when a **parallel** or **serial** construct is encountered.

### 1061 Restrictions

- 1062 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **firstprivate**  
1063 clauses.

## 1064 2.5.14 reduction clause

1065 The **reduction** clause is allowed on the **parallel** and **serial** constructs. It specifies a  
1066 reduction operator and one or more *vars*. It implies **copy** clauses as described in Section 2.6.2. For  
1067 each reduction *var*, a private copy is created for each parallel gang and initialized for that operator.  
1068 At the end of the region, the values for each gang are combined using the reduction operator, and  
1069 the result combined with the value of the original *var* and stored in the original *var*. If the reduction  
1070 *var* is an array or subarray, the array reduction operation is logically equivalent to applying that  
1071 reduction operation to each element of the array or subarray individually. If the reduction *var*  
1072 is a composite variable, the reduction operation is logically equivalent to applying that reduction  
1073 operation to each member of the composite variable individually. The reduction result is available  
1074 after the region.

1075 The following table lists the operators that are valid and the initialization values; in each case, the  
1076 initialization value will be cast into the data type of the *var*. For **max** and **min** reductions, the  
1077 initialization values are the least representable value and the largest representable value for that data  
1078 type, respectively. At a minimum, the supported data types include Fortran **logical** as well as  
1079 the numerical data types in C (e.g., **\_Bool**, **char**, **int**, **float**, **double**, **float \_Complex**,  
1080 **double \_Complex**), C++ (e.g., **bool**, **char**, **wchar\_t**, **int**, **float**, **double**), and Fortran  
1081 (e.g., **integer**, **real**, **double precision**, **complex**). However, for each reduction operator,  
1082 the supported data types include only the types permitted as operands to the corresponding operator  
1083 in the base language where (1) for max and min, the corresponding operator is less-than and (2) for  
1084 other operators, the operands and the result are the same type.

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

1085

## 1086 Restrictions

- 1087 • A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name,  
1088 an array element, or a subarray (refer to Section 2.7.1).
- 1089 • If the reduction *var* is an array element or a subarray, accessing the elements of the array  
1090 outside the specified index range results in unspecified behavior.
- 1091 • The reduction *var* may not be a member of a composite variable.
- 1092 • If the reduction *var* is a composite variable, each member of the composite variable must be  
1093 a supported datatype for the reduction operation.
- 1094 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **reduction**  
1095 clauses.

## 1096 2.5.15 default clause

1097 The **default** clause is optional. At most one **default** clause may appear. It adjusts what  
1098 data attributes are implicitly determined for variables used in the compute construct as described in  
1099 Section 2.6.2.

## 1100 2.6 Data Environment

1101 This section describes the data attributes for variables. The data attributes for a variable may be  
1102 *predetermined*, *implicitly determined*, or *explicitly determined*. Variables with predetermined data  
1103 attributes may not appear in a data clause that conflicts with that data attribute. Variables with  
1104 implicitly determined data attributes may appear in a data clause that overrides the implicit attribute.  
1105 Variables with explicitly determined data attributes are those which appear in a data clause on a  
1106 **data** construct, a compute construct, or a **declare** directive.

1107 OpenACC supports systems with accelerators that have discrete memory from the host, systems  
1108 with accelerators that share memory with the host, as well as systems where an accelerator shares  
1109 some memory with the host but also has some discrete memory that is not shared with the host.  
1110 In the first case, no data is in shared memory. In the second case, all data is in shared memory.  
1111 In the third case, some data may be in shared memory and some data may be in discrete memory,  
1112 although a single array or aggregate data structure must be allocated completely in shared or discrete

1113 memory. When a nested OpenACC construct is executed on the device, the default target device for  
 1114 that construct is the same device on which the encountering accelerator thread is executing. In that  
 1115 case, the target device shares memory with the encountering thread.

### 1116 2.6.1 Variables with Predetermined Data Attributes

1117 The loop variable in a C **for** statement or Fortran **do** statement that is associated with a loop  
 1118 directive is predetermined to be private to each thread that will execute each iteration of the loop.  
 1119 Loop variables in Fortran **do** statements within a compute construct are predetermined to be private  
 1120 to the thread that executes the loop.

1121 Variables declared in a C block or Fortran block construct that is executed in *vector-partitioned*  
 1122 mode are private to the thread associated with each vector lane. Variables declared in a C block  
 1123 or Fortran block construct that is executed in *worker-partitioned vector-single* mode are private to  
 1124 the worker and shared across the threads associated with the vector lanes of that worker. Variables  
 1125 declared in a C block or Fortran block construct that is executed in *worker-single* mode are private  
 1126 to the gang and shared across the threads associated with the workers and vector lanes of that gang.

1127 A procedure called from a compute construct will be annotated as **seq**, **vector**, **worker**, or  
 1128 **gang**, as described Section 2.15 Procedure Calls in Compute Regions. Variables declared in **seq**  
 1129 routine are private to the thread that made the call. Variables declared in **vector** routine are private  
 1130 to the worker that made the call and shared across the threads associated with the vector lanes of  
 1131 that worker. Variables declared in **worker** or **gang** routine are private to the gang that made the  
 1132 call and shared across the threads associated with the workers and vector lanes of that gang.

### 1133 2.6.2 Variables with Implicitly Determined Data Attributes

1134 When implicitly determining data attributes on a compute construct, the following clauses are visi-  
 1135 ble at the compute construct:

- 1136 • The nearest **default** clause appearing on the compute construct or a lexically containing  
 1137 **data** construct.
- 1138 • All data clauses on the compute construct, a lexically containing **data** construct, or a visible  
 1139 **declare** directive.

1140 On a compute or combined construct, if a variable appears in a **reduction** clause but no other  
 1141 data clause, it is treated as if it also appears in a **copy** clause. Otherwise, for any variable, the  
 1142 compiler will implicitly determine its data attribute on a compute construct if all of the following  
 1143 conditions are met:

- 1144 • There is no **default (none)** clause visible at the compute construct.
- 1145 • The variable is referenced in the compute construct.
- 1146 • The variable does not have a predetermined data attribute.
- 1147 • The variable does not appear in a data clause visible at the compute construct.

1148 An aggregate variable will be treated as if it appears either:

- 1149 • In a **present** clause if there is a **default (present)** clause visible at the compute con-  
 1150 struct.

- In a **copy** clause otherwise.

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

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

### Restrictions

- If there is a **default (none)** clause visible at a compute construct, the compiler requires any variable referenced in the compute construct either to have a predetermined data attribute or to appear in a visible data clause. **Note:** A **copy** clause implied by a **reduction** clause suffices as such a data clause.
- If a scalar variable appears in a **reduction** clause on a **loop** construct that has a parent **parallel** or **serial** construct, it must have a predetermined data attribute or appear in an explicit data or **reduction** clause visible at the compute construct. **Note:** Implementations are encouraged to issue a compile-time diagnostic when this restriction is violated to assist users in writing portable OpenACC applications.

If a C++ *lambda* is called in a compute region and does not appear in a data clause, then it is treated as if it appears in a **copyin** clause on the current construct. A variable captured by a *lambda* is processed according to its data types: a pointer type variable is treated as if it appears in a **no\_create** clause; a reference type variable is treated as if it appears in a **present** clause; for a struct or a class type variable, any pointer member is treated as if it appears in a **no\_create** clause on the current construct. If the variable is defined as global or file or function static, it must appear in a **declare** directive.

### 2.6.3 Data Regions and Data Lifetimes

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

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

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

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

In addition to data regions, a program may create and delete data on the accelerator using **enter data** and **exit data** directives or using runtime API routines. When the program executes

1192 an **enter data** directive, or executes a call to a runtime API **acc\_copyin** or **acc\_create**  
 1193 routine, each *var* on the directive or the variable on the runtime API argument list will be made live  
 1194 on accelerator.

## 1195 2.6.4 Data Structures with Pointers

1196 This section describes the behavior of data structures that contain pointers. A pointer may be a  
 1197 C or C++ pointer (e.g., **float\***), a Fortran pointer or array pointer (e.g., **real, pointer,**  
 1198 **dimension(:)**), or a Fortran allocatable (e.g., **real, allocatable, dimension(:)**).

1199 When a data object is copied to device memory, the values are copied exactly. If the data is a data  
 1200 structure that includes a pointer, or is just a pointer, the pointer value copied to device memory  
 1201 will be the host pointer value. If the pointer target object is also allocated in or copied to device  
 1202 memory, the pointer itself needs to be updated with the device address of the target object before  
 1203 dereferencing the pointer in device memory.

1204 An *attach* action updates the pointer in device memory to point to the device copy of the data  
 1205 that the host pointer targets; see Section 2.7.2. For Fortran array pointers and allocatable arrays,  
 1206 this includes copying any associated descriptor (dope vector) to the device copy of the pointer.  
 1207 When the device pointer target is deallocated, the pointer in device memory should be restored  
 1208 to the host value, so it can be safely copied back to host memory. A *detach* action updates the  
 1209 pointer in device memory to have the same value as the corresponding pointer in local memory;  
 1210 see Section 2.7.2. The *attach* and *detach* actions are performed by the **copy, copyin, copyout,**  
 1211 **create, attach,** and **detach** data clauses (Sections 2.7.4-2.7.13), and the **acc\_attach** and  
 1212 **acc\_detach** runtime API routines (Sections 3.2.40 and 3.2.41). The *attach* and *detach* actions  
 1213 use attachment counters to determine when the pointer in device memory needs to be updated; see  
 1214 Section 2.6.8.

## 1215 2.6.5 Data Construct

### 1216 Summary

1217 The **data** construct defines *vars* to be allocated in the current device memory for the duration of  
 1218 the region, whether data should be copied from local memory to the current device memory upon  
 1219 region entry, and copied from device memory to local memory upon region exit.

### 1220 Syntax

1221 In C and C++, the syntax of the OpenACC **data** construct is

```
1222     #pragma acc data [clause-list] new-line  
1223         structured block
```

1224 and in Fortran, the syntax is

```
1225     !$acc data [clause-list]  
1226         structured block  
1227     !$acc end data
```

1228 OR

```
1229     !$acc data [clause-list]  
1230         block construct  
1231     !$acc end data
```

1232 where *clause* is one of the following:

```

1233     if ( condition )
1234     copy ( var-list )
1235     copyin ( [readonly:] var-list )
1236     copyout ( [zero:] var-list )
1237     create ( [zero:] var-list )
1238     no_create ( var-list )
1239     present ( var-list )
1240     deviceptr ( var-list )
1241     attach ( var-list )
1242     default ( none | present )

```

### 1243 Description

1244 Data will be allocated in the memory of the current device and copied from local memory to device  
 1245 memory, or copied back, as required. The data clauses are described in Section 2.7 Data Clauses.  
 1246 Structured reference counters are incremented for data when entering a data region, and decre-  
 1247 mented when leaving the region, as described in Section 2.6.7 Reference Counters.

### 1248 Restrictions

- 1249 • At least one **copy**, **copyin**, **copyout**, **create**, **no\_create**, **present**, **deviceptr**,  
 1250 **attach**, or **default** clause must appear on a **data** construct.

### 1251 **if** clause

1252 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate  
 1253 space in the current device memory and move data from and to the local memory as required.  
 1254 When an **if** clause appears, the program will conditionally allocate memory in and move data to  
 1255 and/or from device memory. When the *condition* in the **if** clause evaluates to zero in C or C++, or  
 1256 **.false.** in Fortran, no device memory will be allocated, and no data will be moved. When the  
 1257 *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the data will be allocated and  
 1258 moved as specified. At most one **if** clause may appear.

### 1259 **default** clause

1260 The **default** clause is optional. At most one **default** clause may appear. It adjusts what data  
 1261 attributes are implicitly determined for variables used in lexically contained compute constructs as  
 1262 described in Section 2.6.2.

## 1263 2.6.6 Enter Data and Exit Data Directives

### 1264 Summary

1265 An **enter data** directive may be used to define *vars* to be allocated in the current device memory  
 1266 for the remaining duration of the program, or until an **exit data** directive that deallocates the data.  
 1267 They also tell whether data should be copied from local memory to device memory at the **enter**  
 1268 **data** directive, and copied from device memory to local memory at the **exit data** directive. The  
 1269 dynamic range of the program between the **enter data** directive and the matching **exit data**  
 1270 directive is the data lifetime for that data.

1271 **Syntax**

1272 In C and C++, the syntax of the OpenACC **enter data** directive is

1273     **#pragma acc enter data** *clause-list new-line*

1274 and in Fortran, the syntax is

1275     **!\$acc enter data** *clause-list*

1276 where *clause* is one of the following:

1277     **if** ( *condition* )  
 1278     **async** [ ( *int-expr* ) ]  
 1279     **wait** [ ( *wait-argument* ) ]  
 1280     **copyin** ( *var-list* )  
 1281     **create** ( [ **zero** : ] *var-list* )  
 1282     **attach** ( *var-list* )

1283 In C and C++, the syntax of the OpenACC **exit data** directive is

1284     **#pragma acc exit data** *clause-list new-line*

1285 and in Fortran, the syntax is

1286     **!\$acc exit data** *clause-list*

1287 where *clause* is one of the following:

1288     **if** ( *condition* )  
 1289     **async** [ ( *int-expr* ) ]  
 1290     **wait** [ ( *wait-argument* ) ]  
 1291     **copyout** ( *var-list* )  
 1292     **delete** ( *var-list* )  
 1293     **detach** ( *var-list* )  
 1294     **finalize**

1295 **Description**

1296 At an **enter data** directive, data may be allocated in the current device memory and copied from  
 1297 local memory to device memory. This action enters a data lifetime for those *vars*, and will make  
 1298 the data available for **present** clauses on constructs within the data lifetime. Dynamic reference  
 1299 counters are incremented for this data, as described in Section 2.6.7 Reference Counters. Pointers  
 1300 in device memory may be *attached* to point to the corresponding device copy of the host pointer  
 1301 target.

1302 At an **exit data** directive, data may be copied from device memory to local memory and deal-  
 1303 located from device memory. If no **finalize** clause appears, dynamic reference counters are  
 1304 decremented for this data. If a **finalize** clause appears, the dynamic reference counters are set  
 1305 to zero for this data. Pointers in device memory may be *detached* so as to have the same value as  
 1306 the original host pointer.

1307 The data clauses are described in Section 2.7 Data Clauses. Reference counting behavior is de-  
 1308 scribed in Section 2.6.7 Reference Counters.

## 1309 Restrictions

- 1310 • At least one **copyin**, **create**, or **attach** clause must appear on an **enter data** direc-  
1311 tive.
- 1312 • At least one **copyout**, **delete**, or **detach** clause must appear on an **exit data** direc-  
1313 tive.

## 1314 if clause

1315 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or  
1316 deallocate space in the current device memory and move data from and to local memory. When an  
1317 **if** clause appears, the program will conditionally allocate or deallocate device memory and move  
1318 data to and/or from device memory. When the *condition* in the **if** clause evaluates to zero in C or  
1319 C++, or **.false.** in Fortran, no device memory will be allocated or deallocated, and no data will  
1320 be moved. When the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the data  
1321 will be allocated or deallocated and moved as specified.

## 1322 async clause

1323 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

## 1324 wait clause

1325 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

## 1326 finalize clause

1327 The **finalize** clause is allowed on the **exit data** directive and is optional. When no **finalize**  
1328 clause appears, the **exit data** directive will decrement the dynamic reference counters for *vars*  
1329 appearing in **copyout** and **delete** clauses, and will decrement the attachment counters for point-  
1330 ers appearing in **detach** clauses. If a **finalize** clause appears, the **exit data** directive will  
1331 set the dynamic reference counters to zero for *vars* appearing in **copyout** and **delete** clauses,  
1332 and will set the attachment counters to zero for pointers appearing in **detach** clauses.

## 1333 2.6.7 Reference Counters

1334 When device memory is allocated for data not in shared memory due to data clauses or OpenACC  
1335 API routine calls, the OpenACC implementation keeps track of that device memory and its relation-  
1336 ship to the corresponding data in host memory.

1337 Each section of device memory will be associated with two *reference counters* per device, a struc-  
1338 tured reference counter and a dynamic reference counter. The structured and dynamic reference  
1339 counters are used to determine when to allocate or deallocate data in device memory. The struc-  
1340 tured reference counter for a block of data keeps track of how many nested data regions have been  
1341 entered for that data. The initial value of the structured reference counter for static data in device  
1342 memory (in a global **declare** directive) is one; for all other data, the initial value is zero. The  
1343 dynamic reference counter for a block of data keeps track of how many dynamic data lifetimes are  
1344 currently active in device memory for that block. The initial value of the dynamic reference counter  
1345 is zero. Data is considered *present* if the sum of the structured and dynamic reference counters is  
1346 greater than zero.



1347 A structured reference counter is incremented when entering each data or compute region that con-  
1348 tain an explicit data clause or implicitly-determined data attributes for that block of memory, and  
1349 is decremented when exiting that region. A dynamic reference counter is incremented for each  
1350 **enter data copyin** or **create** clause, or each **acc\_copyin** or **acc\_create** API routine  
1351 call for that block of memory. The dynamic reference counter is decremented for each **exit data**  
1352 **copyout** or **delete** clause when no **finalize** clause appears, or each **acc\_copyout** or  
1353 **acc\_delete** API routine call for that block of memory. The dynamic reference counter will be  
1354 set to zero with an **exit data copyout** or **delete** clause when a **finalize** clause appears,  
1355 or each **acc\_copyout\_finalize** or **acc\_delete\_finalize** API routine call for the block  
1356 of memory. The reference counters are modified synchronously with the local thread, even if the  
1357 data directives include an **async** clause. When both structured and dynamic reference counters  
1358 reach zero, the data lifetime in device memory for that data ends.

### 1359 2.6.8 Attachment Counter

1360 Since multiple pointers can target the same address, each pointer in device memory is associated  
1361 with an *attachment counter* per device. The *attachment counter* for a pointer is initialized to zero  
1362 when the pointer is allocated in device memory. The *attachment counter* for a pointer is set to one  
1363 whenever the pointer is *attached* to new target address, and incremented whenever an *attach* action  
1364 for that pointer is performed for the same target address. The *attachment counter* is decremented  
1365 whenever a *detach* action occurs for the pointer, and the pointer is *detached* when the *attachment*  
1366 *counter* reaches zero. This is described in more detail in Section 2.7.2 Data Clause Actions.

1367 A pointer in device memory can be assigned a device address in two ways. The pointer can be  
1368 attached to a device address due to data clauses or API routines, as described in Section 2.7.2  
1369 Data Clause Actions, or the pointer can be assigned in a compute region executed on that device.  
1370 Unspecified behavior may result if both ways are used for the same pointer.

1371 Pointer members of structs, classes, or derived types in device or host memory can be overwritten  
1372 due to update directives or API routines. It is the user's responsibility to ensure that the pointers  
1373 have the appropriate values before or after the data movement in either direction. The behavior of  
1374 the program is undefined if any of the pointer members are attached when an update of a composite  
1375 variable is performed.

## 1376 2.7 Data Clauses

1377 These data clauses may appear on the **parallel** construct, **kernels** construct, **serial** con-  
1378 struct, **data** construct, the **enter data** and **exit data** directives, and **declare** directives.  
1379 In the descriptions, the *region* is a compute region with a clause appearing on a **parallel**,  
1380 **kernels**, or **serial** construct, a data region with a clause on a **data** construct, or an implicit  
1381 data region with a clause on a **declare** directive. If the **declare** directive appears in a global  
1382 context, the corresponding implicit data region has a duration of the program. The list argument to  
1383 each data clause is a comma-separated collection of *vars*. For all clauses except **deviceptr** and  
1384 **present**, the list argument may include a Fortran *common block* name enclosed within slashes,  
1385 if that *common block* name also appears in a **declare** directive **link** clause. In all cases, the  
1386 compiler will allocate and manage a copy of the *var* in the memory of the current device, creating a  
1387 visible device copy of that *var*, for data not in shared memory.

1388 OpenACC supports accelerators with discrete memories from the local thread. However, if the  
1389 accelerator can access the local memory directly, the implementation may avoid the memory allo-

1390 cation and data movement and simply share the data in local memory. Therefore, a program that  
 1391 uses and assigns data on the host and uses and assigns the same data on the accelerator within a  
 1392 data region without update directives to manage the coherence of the two copies may get different  
 1393 answers on different accelerators or implementations.

#### 1394 **Restrictions**

- 1395 • Data clauses may not follow a **device\_type** clause.
- 1396 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in data  
 1397 clauses.

### 1398 **2.7.1 Data Specification in Data Clauses**

1399 In C and C++, a subarray is an array name followed by an extended array range specification in  
 1400 brackets, with start and length, such as

1401 **AA[2:n]**

1402 If the lower bound is missing, zero is used. If the length is missing and the array has known size, the  
 1403 size of the array is used; otherwise the length is required. The subarray **AA[2:n]** means element  
 1404 **AA[2], AA[3], ..., AA[2+n-1]**.

1405 In C and C++, a two dimensional array may be declared in at least four ways:

- 1406 • Statically-sized array: **float AA[100][200];**
- 1407 • Pointer to statically sized rows: **typedef float row[200]; row\* BB;**
- 1408 • Statically-sized array of pointers: **float\* CC[200];**
- 1409 • Pointer to pointers: **float\*\* DD;**

1410 Each dimension may be statically sized, or a pointer to dynamically allocated memory. Each of  
 1411 these may be included in a data clause using subarray notation to specify a rectangular array:

- 1412 • **AA[2:n][0:200]**
- 1413 • **BB[2:n][0:m]**
- 1414 • **CC[2:n][0:m]**
- 1415 • **DD[2:n][0:m]**

1416 Multidimensional rectangular subarrays in C and C++ may be specified for any array with any com-  
 1417 bination of statically-sized or dynamically-allocated dimensions. For statically sized dimensions,  
 1418 all dimensions except the first must specify the whole extent, to preserve the contiguous data re-  
 1419 striction, discussed below. For dynamically allocated dimensions, the implementation will allocate  
 1420 pointers in device memory corresponding to the pointers in local memory, and will fill in those  
 1421 pointers as appropriate.

1422 In Fortran, a subarray is an array name followed by a comma-separated list of range specifications  
 1423 in parentheses, with lower and upper bound subscripts, such as

1424 **arr(1:high, low:100)**

1425 If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if  
1426 known, are used. All dimensions except the last must specify the whole extent, to preserve the  
1427 contiguous data restriction, discussed below.

### 1428 Restrictions

- 1429 • In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be  
1430 specified.
- 1431 • In C and C++, the length for dynamically allocated dimensions of an array must be explicitly  
1432 specified.
- 1433 • In C and C++, modifying pointers in pointer arrays during the data lifetime, either on the host  
1434 or on the device, may result in undefined behavior.
- 1435 • If a subarray appears in a data clause, the implementation may choose to allocate memory for  
1436 only that subarray on the accelerator.
- 1437 • In Fortran, array pointers may appear, but pointer association is not preserved in device mem-  
1438 ory.
- 1439 • Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous  
1440 block of memory, except for dynamic multidimensional C arrays.
- 1441 • In C and C++, if a variable or array of composite type appears, all the data members of the  
1442 struct or class are allocated and copied, as appropriate. If a composite member is a pointer  
1443 type, the data addressed by that pointer are not implicitly copied.
- 1444 • In Fortran, if a variable or array of composite type appears, all the members of that derived  
1445 type are allocated and copied, as appropriate. If any member has the **allocatable** or  
1446 **pointer** attribute, the data accessed through that member are not copied.
- 1447 • If an expression is used in a subscript or subarray expression in a clause on a **data** construct,  
1448 the same value is used when copying data at the end of the data region, even if the values of  
1449 variables in the expression change during the data region.

### 1450 2.7.2 Data Clause Actions

1451 Most of the data clauses perform one or more the following actions. The actions test or modify one  
1452 or both of the structured and dynamic reference counters, depending on the directive on which the  
1453 data clause appears.

#### 1454 Present Increment Action

1455 A *present increment* action is one of the actions that may be performed for a **present** (Section  
1456 2.7.5), **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Sec-  
1457 tion 2.7.9), or **no\_create** (Section 2.7.10) clause, or for a call to an **acc\_copyin** (Section  
1458 3.2.26) or **acc\_create** (Section 3.2.27) API routine. See those sections for details.

1459 A *present increment* action for a *var* occurs only when *var* is already present in device memory.

1460 A *present increment* action for a *var* increments the structured or dynamic reference counter for *var*.

### 1461 Present Decrement Action

1462 A *present decrement* action is one of the actions that may be performed for a **present** (Section  
1463 2.7.5), **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Sec-  
1464 tion 2.7.9), **no\_create** (Section 2.7.10), or **delete** (Section 2.7.11) clause, or for a call to an  
1465 **acc\_copyout** (Section 3.2.28) or **acc\_delete** (Section 3.2.29) API routine. See those sections  
1466 for details.

1467 A *present decrement* action for a *var* occurs only when *var* is already present in device memory.

1468 A *present decrement* action for a *var* decrements the structured or dynamic reference counter for  
1469 *var*, if its value is greater than zero. If the device memory associated with *var* was mapped to  
1470 the device using **acc\_map\_data**, the dynamic reference count may not be decremented to zero,  
1471 except by a call to **acc\_unmap\_data**. If the reference counter is already zero, its value is left  
1472 unchanged.

### 1473 Create Action

1474 A *create* action is one of the actions that may be performed for a **copyout** (Section 2.7.8) or  
1475 **create** (Section 2.7.9) clause, or for a call to an **acc\_create** API routine (Section 3.2.27). See  
1476 those sections for details.

1477 A *create* action for a *var* occurs only when *var* is not already present in device memory.

1478 A *create* action for a *var*:

- 1479 • allocates device memory for *var*; and
- 1480 • sets the structured or dynamic reference counter to one.

### 1481 Copyin Action

1482 A *copyin* action is one of the actions that may be performed for a **copy** (Section 2.7.6) or **copyin**  
1483 (Section 2.7.7) clause, or for a call to an **acc\_copyin** API routine (Section 3.2.26). See those  
1484 sections for details.

1485 A *copyin* action for a *var* occurs only when *var* is not already present in device memory.

1486 A *copyin* action for a *var*:

- 1487 • allocates device memory for *var*;
- 1488 • initiates a copy of the data for *var* from the local thread memory to the corresponding device  
1489 memory; and
- 1490 • sets the structured or dynamic reference counter to one.

1491 The data copy may complete asynchronously, depending on other clauses on the directive.

### 1492 Copyout Action

1493 A *copyout* action is one of the actions that may be performed for a **copy** (Section 2.7.6) or  
1494 **copyout** (Section 2.7.8) clause, or for a call to an **acc\_copyout** API routine (Section 3.2.28).  
1495 See those sections for details.

1496 A *copyout* action for a *var* occurs only when *var* is present in device memory.

1497 A *copyout* action for a *var*:

- 1498 • performs an *immediate detach* action for any pointer in *var*;
- 1499 • initiates a copy of the data for *var* from device memory to the corresponding local thread
- 1500 memory; and
- 1501 • deallocates device memory for *var*.

1502 The data copy may complete asynchronously, depending on other clauses on the directive, in which  
1503 case the memory is deallocated when the data copy is complete.

## 1504 Delete Action

1505 A *delete* action is one of the actions that may be performed for a **present** (Section 2.7.5),  
1506 **copyin** (Section 2.7.7), **create** (Section 2.7.9), **no\_create** (Section 2.7.10), or **delete** (Sec-  
1507 tion 2.7.11) clause, or for a call to an **acc\_delete** API routine (Section 3.2.29). See those sections  
1508 for details.

1509 A *delete* action for a *var* occurs only when *var* is present in device memory.

1510 A *delete* action for *var*:

- 1511 • performs an *immediate detach* action for any pointer in *var*; and
- 1512 • deallocates device memory for *var*.

## 1513 Attach Action

1514 An *attach* action is one of the actions that may be performed for a **present** (Section 2.7.5),  
1515 **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Section 2.7.9),  
1516 **no\_create** (Section 2.7.10), or **attach** (Section 2.7.11) clause, or for a call to an **acc\_attach**  
1517 API routine (Section 3.2.40). See those sections for details.

1518 An *attach* action for a *var* occurs only when *var* is a pointer reference.

1519 If the pointer *var* is in shared memory or is not present in the current device memory, or if the  
1520 address to which *var* points is not present in the current device memory, no action is taken. If the  
1521 *attachment counter* for *var* is nonzero and the pointer in device memory already points to the device  
1522 copy of the data in *var*, the *attachment counter* for the pointer *var* is incremented. Otherwise, the  
1523 pointer in device memory is *attached* to the device copy of the data by initiating an update for the  
1524 pointer in device memory to point to the device copy of the data and setting the *attachment counter*  
1525 for the pointer *var* to one. The update may complete asynchronously, depending on other clauses  
1526 on the directive. The pointer update must follow any data copies due to *copyin* actions that are  
1527 performed for the same directive.

## 1528 Detach Action

1529 A *detach* action is one of the actions that may be performed for a **present** (Section 2.7.5),  
1530 **copy** (Section 2.7.6), **copyin** (Section 2.7.7), **copyout** (Section 2.7.8), **create** (Section 2.7.9),  
1531 **no\_create** (Section 2.7.10), **delete** (Section 2.7.11), or **detach** (Section 2.7.11) clause, or  
1532 for a call to an **acc\_detach** API routine (Section 3.2.41). See those sections for details.

1533 A *detach* action for a *var* occurs only when *var* is a pointer reference.

1534 If the pointer *var* is in shared memory or is not present in the current device memory, or if the  
1535 *attachment counter* for *var* for the pointer is zero, no action is taken. Otherwise, the *attachment*  
1536 *counter* for the pointer *var* is decremented. If the *attachment counter* is decreased to zero, the  
1537 pointer is *detached* by initiating an update for the pointer *var* in device memory to have the same  
1538 value as the corresponding pointer in local memory. The update may complete asynchronously,  
1539 depending on other clauses on the directive. The pointer update must precede any data copies due  
1540 to *copyout* actions that are performed for the same directive.

### 1541 Immediate Detach Action

1542 An *immediate detach* action is one of the actions that may be performed for a **detach** (Section  
1543 2.7.11) clause, or for a call to an **acc\_detach\_finalize** API routine (Section 3.2.41). See  
1544 those sections for details.

1545 An *immediate detach* action for a *var* occurs only when *var* is a pointer reference and is present in  
1546 device memory.

1547 If the *attachment counter* for the pointer is zero, the *immediate detach* action has no effect. Other-  
1548 wise, the *attachment counter* for the pointer set to zero and the pointer is *detached* by initiating an  
1549 update for the pointer in device memory to have the same value as the corresponding pointer in local  
1550 memory. The update may complete asynchronously, depending on other clauses on the directive.  
1551 The pointer update must precede any data copies due to *copyout* actions that are performed for the  
1552 same directive.

### 1553 2.7.3 Data Clause Restrictions

1554 The following restriction applies to data that appear in a **present**, **copy**, **copyin**, **copyout**,  
1555 **create**, and **delete** clause:

- 1556 • If only a subarray of an array is present in the current device memory, it is a runtime error if  
1557 *var* includes array elements that are not part of the existing data lifetime.

### 1558 2.7.4 deviceptr clause

1559 The **deviceptr** clause may appear on structured **data** and compute constructs and **declare**  
1560 directives.

1561 The **deviceptr** clause is used to declare that the pointers in *var-list* are device pointers, so the  
1562 data need not be allocated or moved between the host and device for this pointer.

1563 In C and C++, the *vars* in *var-list* must be pointer variables.

1564 In Fortran, the *vars* in *var-list* must be dummy arguments (arrays or scalars), and may not have the  
1565 Fortran **pointer**, **allocatable**, or **value** attributes.

1566 For data in shared memory, host pointers are the same as device pointers, so this clause has no  
1567 effect.

### 1568 2.7.5 present clause

1569 The **present** clause may appear on structured **data** and compute constructs and **declare** di-  
1570 rectives. The **present** clause specifies that *vars* in *var-list* are in shared memory or are already

1571 present in the current device memory due to data regions or data lifetimes that contain the construct  
1572 on which the **present** clause appears.

1573 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,  
1574 the **present** clause behaves as follows:

- 1575 • At entry to the region:
  - 1576 – If *var* is not present in the current device memory, a runtime error is issued.
  - 1577 – Otherwise, a *present increment* action with the structured reference counter is performed.
  - 1578 If *var* is a pointer reference, an *attach* action is performed.

- 1579 • At exit from the region:
  - 1580 – If the structured reference counter for *var* is zero, no action is taken.
  - 1581 – Otherwise, a *present decrement* action with the structured reference counter is per-  
1582 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured  
1583 and dynamic reference counters are zero, a *delete* action is performed.

1584 The restrictions in Section 2.7.3 Data Clause Restrictions apply to this clause.

## 1585 2.7.6 copy clause

1586 The **copy** clause may appear on structured **data** and compute constructs and on **declare** direc-  
1587 tives.

1588 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,  
1589 the **copy** clause behaves as follows:

- 1590 • At entry to the region:
  - 1591 – If *var* is present, a *present increment* action with the structured reference counter is  
1592 performed. If *var* is a pointer reference, an *attach* action is performed.
  - 1593 – Otherwise, a *copyin* action with the structured reference counter is performed. If *var* is  
1594 a pointer reference, an *attach* action is performed.
- 1595 • At exit from the region:
  - 1596 – If the structured reference counter for *var* is zero, no action is taken.
  - 1597 – Otherwise, a *present decrement* action with the structured reference counter is per-  
1598 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured  
1599 and dynamic reference counters are zero, a *copyout* action is performed.

1600 The restrictions in Section 2.7.3 Data Clause Restrictions apply to this clause.

1601 For compatibility with OpenACC 2.0, **present\_or\_copy** and **pcopy** are alternate names for  
1602 **copy**.

## 1603 2.7.7 copyin clause

1604 The **copyin** clause may appear on structured **data** and compute constructs, on **declare** direc-  
1605 tives, and on **enter data** directives.

1606 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,  
 1607 the **copyin** clause behaves as follows:

- 1608 • At entry to a region, the structured reference counter is used. On an **enter data** directive,  
 1609 the dynamic reference counter is used.
  - 1610 – If *var* is present, a *present increment* action with the appropriate reference counter is  
 1611 performed. If *var* is a pointer reference, an *attach* action is performed.
  - 1612 – Otherwise, a *copyin* action with the appropriate reference counter is performed. If *var*  
 1613 is a pointer reference, an *attach* action is performed.
- 1614 • At exit from the region:
  - 1615 – If the structured reference counter for *var* is zero, no action is taken.
  - 1616 – Otherwise, a *present decrement* action with the structured reference counter is per-  
 1617 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured  
 1618 and dynamic reference counters are zero, a *delete* action is performed.

1619 If the optional **readonly** modifier appears, then the implementation may assume that the data  
 1620 referenced by *var-list* is never written to within the applicable region.

1621 The restrictions in Section 2.7.3 Data Clause Restrictions apply to this clause.

1622 For compatibility with OpenACC 2.0, **present\_or\_copyin** and **pcopyin** are alternate names  
 1623 for **copyin**.

1624 An **enter data** directive with a **copyin** clause is functionally equivalent to a call to the **acc\_copyin**  
 1625 API routine, as described in Section 3.2.26.

## 1626 2.7.8 copyout clause

1627 The **copyout** clause may appear on structured **data** and compute constructs, on **declare** di-  
 1628 rectives, and on **exit data** directives. The clause may optionally have a **zero** modifier if the  
 1629 **copyout** clause appears on a structured **data** or compute construct.

1630 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,  
 1631 the **copyout** clause behaves as follows:

- 1632 • At entry to a region:
  - 1633 – If *var* is present, a *present increment* action with the structured reference counter is  
 1634 performed. If *var* is a pointer reference, an *attach* action is performed.
  - 1635 – Otherwise, a *create* action with the structured reference is performed. If *var* is a pointer  
 1636 reference, an *attach* action is performed. If a **zero** modifier appears, the memory is  
 1637 zeroed after the *create* action.
- 1638 • At exit from a region, the structured reference counter is used. On an **exit data** directive,  
 1639 the dynamic reference counter is used.
  - 1640 – If the appropriate reference counter for *var* is zero, no action is taken.
  - 1641 – Otherwise, the reference counter is updated:



1642 \* On an **exit data** directive with a **finalize** clause, the dynamic reference  
1643 counter is set to zero.

1644 \* Otherwise, a *present decrement* action with the appropriate reference counter is  
1645 performed.

1646 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
1647 reference counters are zero, a *copyout* action is performed.

1648 The restrictions in Section 2.7.3 Data Clause Restrictions apply to this clause.

1649 For compatibility with OpenACC 2.0, **present\_or\_copyout** and **pcopyout** are alternate  
1650 names for **copyout**.

1651 An **exit data** directive with a **copyout** clause and with or without a **finalize** clause is func-  
1652 tionally equivalent to a call to the **acc\_copyout\_finalize** or **acc\_copyout** API routine,  
1653 respectively, as described in Section 3.2.28.

## 1654 2.7.9 create clause

1655 The **create** clause may appear on structured **data** and compute constructs, on **declare** direc-  
1656 tives, and on **enter data** directives. The clause may optionally have a **zero** modifier.

1657 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,  
1658 the **create** clause behaves as follows:

1659 • At entry to a region, the structured reference counter is used. On an **enter data** directive,  
1660 the dynamic reference counter is used.

1661 – If *var* is present, a *present increment* action with the appropriate reference counter is  
1662 performed. If *var* is a pointer reference, an *attach* action is performed.

1663 – Otherwise, a *create* action with the appropriate reference counter is performed. If *var*  
1664 is a pointer reference, an *attach* action is performed. If a **zero** modifier appears, the  
1665 memory is zeroed after the *create* action.

1666 • At exit from the region:

1667 – If the structured reference counter for *var* is zero, no action is taken.

1668 – Otherwise, a *present decrement* action with the structured reference counter is per-  
1669 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured  
1670 and dynamic reference counters are zero, a *delete* action is performed.

1671 The restrictions in Section 2.7.3 Data Clause Restrictions apply to this clause.

1672 For compatibility with OpenACC 2.0, **present\_or\_create** and **pcreate** are alternate names  
1673 for **create**.

1674 An **enter data** directive with a **create** clause is functionally equivalent to a call to the **acc\_create**  
1675 API routine, as described in Section 3.2.27.

## 1676 2.7.10 no\_create clause

1677 The **no\_create** clause may appear on structured **data** and compute constructs.

1678 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,  
1679 the **no\_create** clause behaves as follows:

- 1680 • At entry to the region:
  - 1681 – If *var* is present, a *present increment* action with the structured reference counter is  
1682 performed. If *var* is a pointer reference, an *attach* action is performed.
  - 1683 – Otherwise, no action is performed, and any device code in this construct will use the  
1684 local memory address for *var*.
- 1685 • At exit from the region:
  - 1686 – If the structured reference counter for *var* is zero, no action is taken.
  - 1687 – Otherwise, a *present decrement* action with the structured reference counter is per-  
1688 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured  
1689 and dynamic reference counters are zero, a *delete* action is performed.

1690 The restrictions in Section 2.7.3 Data Clause Restrictions do not apply to this clause.

### 1691 2.7.11 delete clause

1692 The **delete** clause may appear on **exit data** directives.

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

- 1695 • If the dynamic reference counter for *var* is zero, no action is taken.
  - 1696 • Otherwise, the dynamic reference counter is updated:
    - 1697 – On an **exit data** directive with a **finalize** clause, the dynamic reference counter  
1698 is set to zero.
    - 1699 – Otherwise, a *present decrement* action with the dynamic reference counter is performed.
- 1700 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
1701 reference counters are zero, a *delete* action is performed.

1702 An **exit data** directive with a **delete** clause and with or without a **finalize** clause is func-  
1703 tionally equivalent to a call to the **acc\_delete\_finalize** or **acc\_delete** API routine, re-  
1704 spectively, as described in Section 3.2.29.

1705 The restrictions in Section 2.7.3 Data Clause Restrictions apply to this clause.

### 1706 2.7.12 attach clause

1707 The **attach** clause may appear on structured **data** and compute constructs and on **enter data**  
1708 directives. Each *var* argument to an **attach** clause must be a C or C++ pointer or a Fortran variable  
1709 or array with the **pointer** or **allocatable** attribute.

1710 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,  
1711 the **attach** clause behaves as follows:

- 1712 • At entry to a region or at an **enter data** directive, an *attach* action is performed.

- 1713
- At exit from the region, a *detach* action is performed.

### 1714 2.7.13 detach clause

1715 The **detach** clause may appear on **exit data** directives. Each *var* argument to a **detach** clause  
1716 must be a C or C++ pointer or a Fortran variable or array with the **pointer** or **allocatable**  
1717 attribute.

1718 For each *var* in *var-list*, if *var* is in shared memory, no action is taken; if *var* is not in shared memory,  
1719 the **detach** clause behaves as follows:

- 1720
- If there is a **finalize** clause on the **exit data** directive, an *immediate detach* action is  
1721 performed.
  - Otherwise, a *detach* action is performed.
- 1722

## 1723 2.8 Host\_Data Construct

### 1724 Summary

1725 The **host\_data** construct makes the address of data in device memory available on the host.

### 1726 Syntax

1727 In C and C++, the syntax of the OpenACC **host\_data** construct is

```
1728     #pragma acc host_data clause-list new-line
1729         structured block
```

1730 and in Fortran, the syntax is

```
1731     !$acc host_data clause-list
1732         structured block
1733     !$acc end host_data
```

1734 or

```
1735     !$acc host_data clause-list
1736         block construct
1737     [!$acc end host_data]
```

1738 where *clause* is one of the following:

```
1739     use_device ( var-list )
1740     if ( condition )
1741     if_present
```

### 1742 Description

1743 This construct is used to make the address of data in device memory available in host code.

### 1744 Restrictions

- 1745
- A *var* in a **use\_device** clause must be the name of a variable or array.
  - At least one **use\_device** clause must appear.
  - At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical  
1747 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1748

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

### 1751 2.8.1 use\_device clause

1752 The **use\_device** clause tells the compiler to use the current device address of any *var* in *var-list*  
 1753 in code within the construct. In particular, this may be used to pass the device address of *var* to  
 1754 optimized procedures written in a lower-level API. When there is no **if\_present** clause, and  
 1755 either there is no **if** clause or the condition in the **if** clause evaluates to nonzero (in C or C++)  
 1756 or **.true.** (in Fortran), the *var* in *var-list* must be present in the accelerator memory due to data  
 1757 regions or data lifetimes that contain this construct. For data in shared memory, the device address  
 1758 is the same as the host address.

### 1759 2.8.2 if clause

1760 The **if** clause is optional. When an **if** clause appears and the condition evaluates to zero in C  
 1761 or C++, or **.false.** in Fortran, the compiler will not replace the addresses of any *var* in code  
 1762 within the construct. When there is no **if** clause, or when an **if** clause appears and the condition  
 1763 evaluates to nonzero in C or C++, or **.true.** in Fortran, the compiler will replace the addresses as  
 1764 described in the previous subsection.

### 1765 2.8.3 if\_present clause

1766 When an **if\_present** clause appears on the directive, the compiler will only replace the address  
 1767 of any *var* which appears in *var-list* that is present in the current device memory.

## 1768 2.9 Loop Construct

### 1769 Summary

1770 The OpenACC **loop** construct applies to a loop which must immediately follow this directive. The  
 1771 **loop** construct can describe what type of parallelism to use to execute the loop and declare private  
 1772 *vars* and reduction operations.

### 1773 Syntax

1774 In C and C++, the syntax of the **loop** construct is

```
1775     #pragma acc loop [clause-list] new-line
1776           for loop
```

1777 In Fortran, the syntax of the **loop** construct is

```
1778     !$acc loop [clause-list]
1779           do loop
```

1780 where *clause* is one of the following:

```
1781     collapse ( n )
1782     gang [ ( gang-arg-list ) ]
1783     worker [ ( [num:] int-expr ) ]
1784     vector [ ( [length:] int-expr ) ]
1785     seq
1786     independent
```

```

1787     auto
1788     tile( size-expr-list )
1789     device_type( device-type-list )
1790     private( var-list )
1791     reduction( operator:var-list )

```

1792 where *gang-arg* is one of:

```

1793     [num:] int-expr
1794     static: size-expr

```

1795 and *gang-arg-list* may have at most one **num** and one **static** argument,

1796 and where *size-expr* is one of:

```

1797     *
1798     int-expr

```

1799

1800 Some clauses are only valid in the context of a **kernels** construct; see the descriptions below.

1801 An *orphaned loop* construct is a **loop** construct that is not lexically enclosed within a compute  
 1802 construct. The parent compute construct of a **loop** construct is the nearest compute construct that  
 1803 lexically contains the **loop** construct.

1804 A **loop** construct is *data-independent* if it has an **independent** clause that is determined explic-  
 1805 itly, implicitly, or from an **auto** clause. A **loop** construct is *sequential* if it has a **seq** clause that  
 1806 is determined explicitly or from an **auto** clause.

1807 When *do-loop* is a **do concurrent**, the OpenACC **loop** construct applies to the loop for each  
 1808 index in the *concurrent-header*. The **loop** construct can describe what type of parallelism to use  
 1809 to execute all the loops, and declares all indices appearing in the *concurrent-header* to be implicitly  
 1810 private. If the **loop** construct that is associated with **do concurrent** is combined with a compute  
 1811 construct then *concurrent-locality* is processed as follows: variables appearing in a *local* are treated  
 1812 as appearing in a **private** clause; variables appearing in a *local\_init* are treated as appearing in a  
 1813 **firstprivate** clause; variables appearing in a *shared* are treated as appearing in a **copy** clause;  
 1814 and a *default(none)* locality spec implies a **default (none)** clause on the compute construct. If  
 1815 the **loop** construct is not combined with a compute construct, the behavior is implementation-  
 1816 defined.

### 1817 Restrictions

- 1818 • Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **independent**, **auto**, and **tile**  
 1819 clauses may follow a **device\_type** clause.
- 1820 • The *int-expr* argument to the **worker** and **vector** clauses must be invariant in the kernels  
 1821 region.
- 1822 • A loop associated with a **loop** construct that does not have a **seq** clause must be written to  
 1823 meet all of the following conditions:
  - 1824 – The loop variable must be of integer, C/C++ pointer, or C++ random-access iterator  
 1825 type.
  - 1826 – The loop variable must monotonically increase or decrease in the direction of its termi-  
 1827 nation condition.

1828           – The loop iteration count must be computable in constant time when entering the **loop**  
1829           construct.

1830           For a C++ range-based **for** loop, the loop variable identified by the above conditions is the  
1831           internal iterator, such as a pointer, that the compiler generates to iterate the range. It is not the  
1832           variable declared by the **for** loop.

- 1833           • Only one of the **seq**, **independent**, and **auto** clauses may appear.
- 1834           • A **gang**, **worker**, or **vector** clause may not appear if a **seq** clause appears.
- 1835           • A **tile** and **collapse** clause may not appear on **loop** that is associated with **do concurrent**.

### 1836 **2.9.1 collapse clause**

1837           The **collapse** clause is used to specify how many tightly nested loops are associated with the  
1838           **loop** construct. The argument to the **collapse** clause must be a constant positive integer expres-  
1839           sion. If no **collapse** clause appears, only the immediately following loop is associated with the  
1840           **loop** construct.

1841           If more than one loop is associated with the **loop** construct, the iterations of all the associated loops  
1842           are all scheduled according to the rest of the clauses. The trip count for all loops associated with the  
1843           **collapse** clause must be computable and invariant in all the loops.

1844           It is implementation-defined whether a **gang**, **worker** or **vector** clause on the construct is ap-  
1845           plied to each loop, or to the linearized iteration space.

### 1846 **2.9.2 gang clause**

1847           When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,  
1848           the **gang** clause specifies that the iterations of the associated loop or loops are to be executed in  
1849           parallel by distributing the iterations among the gangs created by the **parallel** construct. A  
1850           **loop** construct with the **gang** clause transitions a compute region from gang-redundant mode to  
1851           gang-partitioned mode. The number of gangs is controlled by the **parallel** construct; only the  
1852           **static** argument is allowed. The loop iterations must be data independent, except for *vars* which  
1853           appear in a **reduction** clause or which are modified in an atomic region. The region of a loop  
1854           with the **gang** clause may not contain another loop with the **gang** clause unless within a nested  
1855           compute region.

1856           When the parent compute construct is a **kernels** construct, the **gang** clause specifies that the  
1857           iterations of the associated loop or loops are to be executed in parallel across the gangs. An argument  
1858           with no keyword or with the **num** keyword is allowed only when the **num\_gangs** does not appear  
1859           on the **kernels** construct. If an argument with no keyword or an argument after the **num** keyword  
1860           appears, it specifies how many gangs to use to execute the iterations of this loop. The region of a  
1861           loop with the **gang** clause may not contain another loop with a **gang** clause unless within a nested  
1862           compute region.

1863           The scheduling of loop iterations to gangs is not specified unless the **static** modifier appears as  
1864           an argument. If the **static** modifier appears with an integer expression, that expression is used  
1865           as a *chunk* size. If the static modifier appears with an asterisk, the implementation will select a  
1866           *chunk* size. The iterations are divided into chunks of the selected *chunk* size, and the chunks are  
1867           assigned to gangs starting with gang zero and continuing in round-robin fashion. Two **gang** loops

1868 in the same parallel region with the same number of iterations, and with **static** clauses with the  
1869 same argument, will assign the iterations to gangs in the same manner. Two **gang** loops in the  
1870 same kernels region with the same number of iterations, the same number of gangs to use, and with  
1871 **static** clauses with the same argument, will assign the iterations to gangs in the same manner.

1872 A **gang** clause without arguments is implied on a data-independent **loop** construct without an  
1873 explicit **gang** clause if the following conditions hold while ignoring **gang**, **worker**, and **vector**  
1874 clauses on any sequential **loop** constructs:

- 1875 • This **loop** construct's parent compute construct, if any, is not a **kernels** construct.
- 1876 • An explicit **gang** clause would be permitted on this **loop** construct.
- 1877 • For every lexically enclosing data-independent **loop** construct, either an explicit **gang** clause  
1878 would not be permitted on the enclosing **loop** construct, or the enclosing **loop** construct  
1879 lexically encloses a compute construct that lexically encloses this **loop** construct.

1880 **Note:** As a performance optimization, the implementation might select different levels of paral-  
1881 lelism for a **loop** construct than specified by explicitly or implicitly determined clauses as long  
1882 as it can prove program semantics are preserved. In particular, the implementation must consider  
1883 semantic differences between gang-redundant and gang-partitioned mode. For example, in a series  
1884 of tightly nested, data-independent **loop** constructs, implementations often move gang-partitioning  
1885 from one **loop** construct to another without affecting semantics.

1886 **Note:** If the **auto** or **device\_type** clause appears on a **loop** construct, it is the programmer's  
1887 responsibility to ensure that program semantics are the same regardless of whether the **auto** clause  
1888 is treated as **independent** or **seq** and regardless of the device type for which the program is  
1889 compiled. In particular, the programmer must consider the effect on both explicitly and implicitly  
1890 determined **gang** clauses and thus on gang-redundant and gang-partitioned mode. Examples in  
1891 Section 2.9.11 demonstrate this issue for the **auto** clause.

### 1892 2.9.3 worker clause

1893 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,  
1894 the **worker** clause specifies that the iterations of the associated loop or loops are to be executed  
1895 in parallel by distributing the iterations among the multiple workers within a single gang. A **loop**  
1896 construct with a **worker** clause causes a gang to transition from worker-single mode to worker-  
1897 partitioned mode. In contrast to the **gang** clause, the **worker** clause first activates additional  
1898 worker-level parallelism and then distributes the loop iterations across those workers. No argu-  
1899 ment is allowed. The loop iterations must be data independent, except for *vars* which appear in  
1900 a **reduction** clause or which are modified in an atomic region. The region of a loop with the  
1901 **worker** clause may not contain a loop with the **gang** or **worker** clause unless within a nested  
1902 compute region.

1903 When the parent compute construct is a **kernels** construct, the **worker** clause specifies that the  
1904 iterations of the associated loop or loops are to be executed in parallel across the workers within  
1905 a single gang. An argument is allowed only when the **num\_workers** does not appear on the  
1906 **kernels** construct. The optional argument specifies how many workers per gang to use to execute  
1907 the iterations of this loop. The region of a loop with the **worker** clause may not contain a loop  
1908 with a **gang** or **worker** clause unless within a nested compute region.

1909 All workers will complete execution of their assigned iterations before any worker proceeds beyond  
1910 the end of the loop.

#### 1911 **2.9.4 vector clause**

1912 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,  
1913 the **vector** clause specifies that the iterations of the associated loop or loops are to be executed  
1914 in vector or SIMD mode. A **loop** construct with a **vector** clause causes a worker to transition  
1915 from vector-single mode to vector-partitioned mode. Similar to the **worker** clause, the **vector**  
1916 clause first activates additional vector-level parallelism and then distributes the loop iterations across  
1917 those vector lanes. The operations will execute using vectors of the length specified or chosen for  
1918 the parallel region. The loop iterations must be data independent, except for *vars* which appear in  
1919 a **reduction** clause or which are modified in an atomic region. The region of a loop with the  
1920 **vector** clause may not contain a loop with the **gang**, **worker**, or **vector** clause unless within  
1921 a nested compute region.

1922 When the parent compute construct is a **kernels** construct, the **vector** clause specifies that the  
1923 iterations of the associated loop or loops are to be executed with vector or SIMD processing. An  
1924 argument is allowed only when the **vector\_length** does not appear on the **kernels** construct.  
1925 If an argument appears, the iterations will be processed in vector strips of that length; if no argument  
1926 appears, the implementation will choose an appropriate vector length. The region of a loop with the  
1927 **vector** clause may not contain a loop with a **gang**, **worker**, or **vector** clause unless within a  
1928 nested compute region.

1929 All vector lanes will complete execution of their assigned iterations before any vector lane proceeds  
1930 beyond the end of the loop.

#### 1931 **2.9.5 seq clause**

1932 The **seq** clause specifies that the associated loop or loops are to be executed sequentially by the  
1933 accelerator. This clause will override any automatic parallelization or vectorization.

#### 1934 **2.9.6 independent clause**

1935 The **independent** clause tells the implementation that the loop iterations must be data indepen-  
1936 dent, except for *vars* which appear in a **reduction** clause or which are modified in an atomic  
1937 region. This allows the implementation to generate code to execute the iterations in parallel with no  
1938 synchronization.

1939 A **loop** construct with no **auto** or **seq** clause is treated as if it has the **independent** clause  
1940 when it is an orphaned **loop** construct or its parent compute construct is a **parallel** construct.

#### 1941 **Note**

- 1942 • It is likely a programming error to use the **independent** clause on a loop if any iteration  
1943 writes to a variable or array element that any other iteration also writes or reads, except for  
1944 *vars* which appear in a **reduction** clause or which are modified in an atomic region.
- 1945 • The implementation may be restricted in the levels of parallelism it can apply by the presence  
1946 of **loop** constructs with **gang**, **worker**, or **vector** clauses for outer or inner loops.



### 1947 2.9.7 auto clause

1948 The **auto** clause specifies that the implementation must analyze the loop and determine whether the  
1949 loop iterations are data-independent. If it determines that the loop iterations are data-independent,  
1950 the implementation must treat the **auto** clause as if it is an **independent** clause. If not, or if it  
1951 is unable to make a determination, it must treat the **auto** clause as if it is a **seq** clause, and it must  
1952 ignore any **gang**, **worker**, or **vector** clauses on the loop construct.

1953 When the parent compute construct is a **kernels** construct, a **loop** construct with no **independent**  
1954 or **seq** clause is treated as if it has the **auto** clause.

### 1955 2.9.8 tile clause

1956 The **tile** clause specifies that the implementation should split each loop in the loop nest into two  
1957 loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the **tile**  
1958 clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression  
1959 or an asterisk. If there are  $n$  tile sizes in the list, the **loop** construct must be immediately followed  
1960 by  $n$  tightly-nested loops. The first argument in the *size-expr-list* corresponds to the innermost loop  
1961 of the  $n$  associated loops, and the last element corresponds to the outermost associated loop. If the  
1962 tile size is an asterisk, the implementation will choose an appropriate value. Each loop in the nest  
1963 will be split or *strip-mined* into two loops, an outer *tile* loop and an inner *element* loop. The trip  
1964 count of the element loop will be limited to the corresponding tile size from the *size-expr-list*. The  
1965 *tile* loops will be reordered to be outside all the *element* loops, and the *element* loops will all be  
1966 inside the *tile* loops.

1967 If the **vector** clause appears on the **loop** construct, the **vector** clause is applied to the *element*  
1968 loops. If the **gang** clause appears on the **loop** construct, the **gang** clause is applied to the *tile*  
1969 loops. If the **worker** clause appears on the **loop** construct, the **worker** clause is applied to the  
1970 *element* loops if no **vector** clause appears, and to the *tile* loops otherwise.

### 1971 2.9.9 device\_type clause

1972 The **device\_type** clause is described in Section 2.4 Device-Specific Clauses.

### 1973 2.9.10 private clause

1974 The **private** clause on a **loop** construct specifies that a copy of each item in *var-list* will be  
1975 created. If the body of the loop is executed in *vector-partitioned* mode, a copy of the item is created  
1976 for each thread associated with each vector lane. If the body of the loop is executed in *worker-*  
1977 *partitioned vector-single* mode, a copy of the item is created for and shared across the set of threads  
1978 associated with all the vector lanes of each worker. Otherwise, a copy of the item is created for and  
1979 shared across the set of threads associated with all the vector lanes of all the workers of each gang.

#### 1980 Restrictions

- 1981 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **private**  
1982 clauses.

### 1983 2.9.11 reduction clause

1984 The **reduction** clause specifies a reduction operator and one or more *vars*. For each reduction  
1985 *var*, a private copy is created in the same manner as for a **private** clause on the **loop** construct,

1986 and initialized for that operator; see the table in Section 2.5.14 reduction clause. After the loop, the  
 1987 values for each thread are combined using the specified reduction operator, and the result combined  
 1988 with the value of the original *var* and stored in the original *var*. If the original *var* is not private,  
 1989 this update occurs by the end of the compute region, and any access to the original *var* is undefined  
 1990 within the compute region. Otherwise, the update occurs at the end of the loop. If the reduction  
 1991 *var* is an array or subarray, the reduction operation is logically equivalent to applying that reduction  
 1992 operation to each array element of the array or subarray individually. If the reduction *var* is a com-  
 1993 posite variable, the reduction operation is logically equivalent to applying that reduction operation  
 1994 to each member of the composite variable individually.

1995 If a variable is involved in a reduction that spans multiple nested loops where two or more of those  
 1996 loops have associated **loop** directives, a **reduction** clause containing that variable must appear  
 1997 on each of those **loop** directives.

### 1998 Restrictions

- 1999 • A *var* in a **reduction** clause must be a scalar variable name, an aggregate variable name,  
 2000 an array element, or a subarray (refer to Section 2.7.1).
- 2001 • Reduction clauses on nested constructs for the same reduction *var* must have the same reduc-  
 2002 tion operator.
- 2003 • Every *var* in a **reduction** clause appearing on an orphaned **loop** construct must be private.
- 2004 • The restrictions for a **reduction** clause on a compute construct listed in in Section 2.5.14  
 2005 reduction clause also apply to a **reduction** clause on a **loop** construct.
- 2006 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **reduction**  
 2007 clauses.
- 2008 • See Section 2.6.2 Variables with Implicitly Determined Data Attributes for a restriction re-  
 2009 quiring certain loop reduction variables to have explicit data clauses on their parent compute  
 2010 constructs.

2011

### 2012 Examples

2013

- 2014 • **x** is not private at the **loop** directive below, so its reduction normally updates **x** at the end  
 2015 of the parallel region, where gangs synchronize. When possible, the implementation might  
 2016 choose to partially update **x** at the loop exit instead, or fully if **num\_gangs(1)** were added  
 2017 to the **parallel** directive. However, portable applications cannot rely on such early up-  
 2018 dates, so accesses to **x** are undefined within the parallel region outside the loop.

2019

2020

2021

2022

2023

2024

2025

2026

```

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

```

2027     } // gang-shared x updated for gang/worker/vector reduction
2028     // x = I

```

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

```

2032     int x = 0;
2033     #pragma acc parallel copy(x)
2034     {
2035         // gang-shared x undefined
2036         #pragma acc loop gang reduction(+:x)
2037         for (int i = 0; i < I; ++i) {
2038             #pragma acc loop worker reduction(+:x)
2039             for (int j = 0; j < J; ++j) {
2040                 #pragma acc loop vector reduction(+:x)
2041                 for (int k = 0; k < K; ++k) {
2042                     x += 1; // vector-private x modified
2043                 } // worker-private x updated for vector reduction
2044             } // gang-private x updated for worker reduction
2045         }
2046         // gang-shared x undefined
2047     } // gang-shared x updated for gang reduction
2048     // 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.

```

2052     int x = 0, y = 0;
2053     #pragma acc parallel firstprivate(x) copy(y)
2054     {
2055         // gang-private x = 0; gang-shared y undefined
2056         #pragma acc loop seq reduction(+:x,y)
2057         for (int i = 0; i < I; ++i) {
2058             x += 1; y += 2; // loop-private x and y modified
2059         } // gang-private x updated for seq reduction (trivial reduction)
2060         // gang-private x = I; gang-shared y undefined
2061         #pragma acc loop worker reduction(+:x,y)
2062         for (int i = 0; i < I; ++i) {
2063             x += 1; y += 2; // worker-private x and y modified
2064         } // gang-private x updated for worker reduction
2065         // gang-private x = 2 * I; gang-shared y undefined
2066         #pragma acc loop vector reduction(+:x,y)
2067         for (int i = 0; i < I; ++i) {
2068             x += 1; y += 2; // vector-private x and y modified
2069         } // gang-private x updated for vector reduction
2070         // gang-private x = 3 * I; gang-shared y undefined
2071     } // gang-shared y updated for gang/seq/worker/vector reductions
2072     // x = 0; y = 3 * I * 2

```

- 2073 • 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.

```
2077     int x = 0;
2078     #pragma acc parallel loop worker reduction(+:x)
2079     for (int i = 0; i < I; ++i) {
2080         x += 1; // worker-private x modified
2081     } // gang-shared x updated for gang/worker reduction
2082     // x = I
```

```
2083
2084     int x = 0;
2085     #pragma acc parallel copy(x)
2086     {
2087         // gang-shared x undefined
2088         #pragma acc loop worker reduction(+:x)
2089         for (int i = 0; i < I; ++i) {
2090             x += 1; // worker-private x modified
2091         }
2092         // gang-shared x undefined
2093     } // gang-shared x updated for gang/worker reduction
2094     // x = I
```

- 2095 • 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.

```
2100     int x = 0;
2101     const int *arr = /*array of I values*/;
2102     #pragma acc parallel copy(x)
2103     {
2104         // gang-shared x undefined
2105         #pragma acc loop auto gang reduction(max:x)
2106         for (int i = 0; i < I; ++i) {
2107             // complex loop body
2108             x = x < arr[i] ? arr[i] : x; // gang or loop-private x modified
2109         }
2110         // gang-shared x undefined
2111     } // gang-shared x updated for gang or gang/seq reduction
2112     // x = arr maximum
```

- 2113 • 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.



2160 with start and length, such as

2161       **arr** [*lower* : *length*]

2162 where the lower bound is a constant, loop invariant, or the **for** loop variable plus or minus a  
2163 constant or loop invariant, and the length is a constant.

2164 In Fortran, a simple subarray is an array name followed by a comma-separated list of range specifi-  
2165 cations in parentheses, with lower and upper bound subscripts, such as

2166       **arr** (*lower* : *upper*, *lower2* : *upper2*)

2167 The lower bounds must be constant, loop invariant, or the **do** loop variable plus or minus a constant  
2168 or loop invariant; moreover the difference between the corresponding upper and lower bounds must  
2169 be a constant.

2170 If the optional **readonly** modifier appears, then the implementation may assume that the data  
2171 referenced by any *var* in that directive is never written to within the applicable region.

## 2172 Restrictions

- 2173       • If an array element or subarray is listed in a **cache** directive, all references to that array  
2174       during execution of that loop iteration must not refer to elements of the array outside the  
2175       index range specified in the **cache** directive.
- 2176       • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **cache**  
2177       directives.

## 2178 2.11 Combined Constructs

### 2179 Summary

2180 The combined OpenACC **parallel loop**, **kernels loop**, and **serial loop** constructs are  
2181 shortcuts for specifying a **loop** construct nested immediately inside a **parallel**, **kernels**, or  
2182 **serial** construct. The meaning is identical to explicitly specifying a **parallel**, **kernels**, or  
2183 **serial** construct containing a **loop** construct. Any clause that is allowed on a **parallel** or  
2184 **loop** construct is allowed on the **parallel loop** construct; any clause allowed on a **kernels**  
2185 or **loop** construct is allowed on a **kernels loop** construct; and any clause allowed on a **serial**  
2186 or **loop** construct is allowed on a **serial loop** construct.

### 2187 Syntax

2188 In C and C++, the syntax of the **parallel loop** construct is

2189       **#pragma acc parallel loop** [*clause-list*] *new-line*  
2190        *for loop*

2191 In Fortran, the syntax of the **parallel loop** construct is

2192       **!\$acc parallel loop** [*clause-list*]  
2193        *do loop*  
2194       **[!\$acc end parallel loop]**

2195 The associated structured block is the loop which must immediately follow the directive. Any of  
2196 the **parallel** or **loop** clauses valid in a parallel region may appear.

2197 In C and C++, the syntax of the **kernels loop** construct is

2198       **#pragma acc kernels loop** [*clause-list*] *new-line*  
 2199            *for loop*

2200 In Fortran, the syntax of the **kernels loop** construct is

2201       **!\$acc kernels loop** [*clause-list*]  
 2202            *do loop*  
 2203       **[\$acc end kernels loop]**

2204 The associated structured block is the loop which must immediately follow the directive. Any of  
 2205 the **kernels** or **loop** clauses valid in a kernels region may appear.

2206 In C and C++, the syntax of the **serial loop** construct is

2207       **#pragma acc serial loop** [*clause-list*] *new-line*  
 2208            *for loop*

2209 In Fortran, the syntax of the **serial loop** construct is

2210       **!\$acc serial loop** [*clause-list*]  
 2211            *do loop*  
 2212       **[\$acc end serial loop]**

2213 The associated structured block is the loop which must immediately follow the directive. Any of  
 2214 the **serial** or **loop** clauses valid in a serial region may appear.

2215 A **private** or **reduction** clause on a combined construct is treated as if it appeared on the  
 2216 **loop** construct. In addition, a **reduction** clause on a combined construct implies **copy** clauses  
 2217 as described in Section 2.6.2.

## 2218 Restrictions

- 2219       • The restrictions for the **parallel**, **kernels**, **serial**, and **loop** constructs apply.

## 2220 2.12 Atomic Construct

### 2221 Summary

2222 An **atomic** construct ensures that a specific storage location is accessed and/or updated atomically,  
 2223 preventing simultaneous reading and writing by gangs, workers, and vector threads that could result  
 2224 in indeterminate values.

### 2225 Syntax

2226 In C and C++, the syntax of the **atomic** constructs is:

2227       **#pragma acc atomic** [ *atomic-clause* ] *new-line*  
 2228            *expression-stmt*

2229 OR:

2230       **#pragma acc atomic capture** *new-line*  
 2231            *structured block*

2232 Where *atomic-clause* is one of **read**, **write**, **update**, or **capture**. The *expression-stmt* is an  
 2233 expression statement with one of the following forms:

2234 If the *atomic-clause* is **read**:

2235        **v** = **x**;

2236    If the *atomic-clause* is **write**:

2237        **x** = *expr*;

2238    If the *atomic-clause* is **update** or no clause appears:

2239        **x**++;

2240        **x**--;

2241        ++**x**;

2242        --**x**;

2243        **x** *binop*= *expr*;

2244        **x** = **x** *binop* *expr*;

2245        **x** = *expr* *binop* **x**;

2246    If the *atomic-clause* is **capture**:

2247        **v** = **x**++;

2248        **v** = **x**--;

2249        **v** = ++**x**;

2250        **v** = --**x**;

2251        **v** = **x** *binop*= *expr*;

2252        **v** = **x** = **x** *binop* *expr*;

2253        **v** = **x** = *expr* *binop* **x**;

2254    The *structured-block* is a structured block with one of the following forms:

2255        { **v** = **x**; **x** *binop*= *expr*; }

2256        { **x** *binop*= *expr*; **v** = **x**; }

2257        { **v** = **x**; **x** = **x** *binop* *expr*; }

2258        { **v** = **x**; **x** = *expr* *binop* **x**; }

2259        { **x** = **x** *binop* *expr*; **v** = **x**; }

2260        { **x** = *expr* *binop* **x**; **v** = **x**; }

2261        { **v** = **x**; **x** = *expr*; }

2262        { **v** = **x**; **x**++; }

2263        { **v** = **x**; ++**x**; }

2264        { ++**x**; **v** = **x**; }

2265        { **x**++; **v** = **x**; }

2266        { **v** = **x**; **x**--; }

2267        { **v** = **x**; --**x**; }

2268        { --**x**; **v** = **x**; }

2269        { **x**--; **v** = **x**; }

2270    In the preceding expressions:

- 2271        • **x** and **v** (as applicable) are both l-value expressions with scalar type.
- 2272        • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate the same storage location.
- 2273
- 2274        • Neither of **v** and *expr* (as applicable) may access the storage location designated by **x**.
- 2275        • Neither of **x** and *expr* (as applicable) may access the storage location designated by **v**.



- 2276 • *expr* is an expression with scalar type.
- 2277 • *binop* is one of **+**, **\***, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.
- 2278 • *binop*, *binop=*, **++**, and **--** are not overloaded operators.
- 2279 • The expression **x binop expr** must be mathematically equivalent to **x binop (expr)**. This  
2280 requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by  
2281 using parentheses around *expr* or subexpressions of *expr*.
- 2282 • The expression *expr binop x* must be mathematically equivalent to **(expr) binop x**. This  
2283 requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*,  
2284 or by using parentheses around *expr* or subexpressions of *expr*.
- 2285 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is  
2286 unspecified.

2287 In Fortran the syntax of the **atomic** constructs is:

```
2288   !$acc atomic read
2289       capture-statement
2290   [$acc end atomic]
```

2291 OR

```
2292   !$acc atomic write
2293       write-statement
2294   [$acc end atomic]
```

2295 OR

```
2296   !$acc atomic [update]
2297       update-statement
2298   [$acc end atomic]
```

2299 OR

```
2300   !$acc atomic capture
2301       update-statement
2302       capture-statement
2303   !$acc end atomic
```

2304 OR

```
2305   !$acc atomic capture
2306       capture-statement
2307       update-statement
2308   !$acc end atomic
```

2309 OR

```
2310   !$acc atomic capture
2311       capture-statement
2312       write-statement
2313   !$acc end atomic
```

2314 where *write-statement* has the following form (if *atomic-clause* is **write** or **capture**):

2315        **x** = **expr**

2316 where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

2317        **v** = **x**

2318 and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**,  
2319 or no clause appears):

2320        **x** = **x** *operator* *expr*

2321        **x** = *expr* *operator* **x**

2322        **x** = *intrinsic\_procedure\_name* ( **x**, *expr-list* )

2323        **x** = *intrinsic\_procedure\_name* ( *expr-list*, **x** )

2324 In the preceding statements:

- 2325        • **x** and **v** (as applicable) are both scalar variables of intrinsic type.
- 2326        • **x** must not be an allocatable variable.
- 2327        • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate  
2328        the same storage location.
- 2329        • None of **v**, *expr*, and *expr-list* (as applicable) may access the same storage location as **x**.
- 2330        • None of **x**, *expr*, and *expr-list* (as applicable) may access the same storage location as **v**.
- 2331        • *expr* is a scalar expression.
- 2332        • *expr-list* is a comma-separated, non-empty list of scalar expressions. If *intrinsic\_procedure\_name*  
2333        refers to **iand**, **ior**, or **ieor**, exactly one expression must appear in *expr-list*.
- 2334        • *intrinsic\_procedure\_name* is one of **max**, **min**, **iand**, **ior**, or **ieor**. *operator* is one of **+**,  
2335        **\***, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**
- 2336        • The expression **x** *operator* *expr* must be mathematically equivalent to **x** *operator* (*expr*).  
2337        This requirement is satisfied if the operators in *expr* have precedence greater than *operator*,  
2338        or by using parentheses around *expr* or subexpressions of *expr*.
- 2339        • The expression *expr* *operator* **x** must be mathematically equivalent to (*expr*) *operator* **x**.  
2340        This requirement is satisfied if the operators in *expr* have precedence equal to or greater than  
2341        *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- 2342        • *intrinsic\_procedure\_name* must refer to the intrinsic procedure name and not to other program  
2343        entities.
- 2344        • *operator* must refer to the intrinsic operator and not to a user-defined operator. All assign-  
2345        ments must be intrinsic assignments.
- 2346        • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is  
2347        unspecified.

2348 An **atomic** construct with the **read** clause forces an atomic read of the location designated by **x**.  
2349 An **atomic** construct with the **write** clause forces an atomic write of the location designated by  
2350 **x**.

2351 An **atomic** construct with the **update** clause forces an atomic update of the location designated  
2352 by **x** using the designated operator or intrinsic. Note that when no clause appears, the semantics

2353 are equivalent to **atomic update**. Only the read and write of the location designated by **x** are  
 2354 performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect  
 2355 to the read or write of the location designated by **x**.

2356 An **atomic** construct with the **capture** clause forces an atomic update of the location designated  
 2357 by **x** using the designated operator or intrinsic while also capturing the original or final value of  
 2358 the location designated by **x** with respect to the atomic update. The original or final value of the  
 2359 location designated by **x** is written into the location designated by **v** depending on the form of the  
 2360 **atomic** construct structured block or statements following the usual language semantics. Only  
 2361 the read and write of the location designated by **x** are performed mutually atomically. Neither the  
 2362 evaluation of *expr* or *expr-list*, nor the write to the location designated by **v**, need to be atomic with  
 2363 respect to the read or write of the location designated by **x**.

2364 For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs  
 2365 enforces mutually exclusive access to the locations designated by **x**. To avoid race conditions, all  
 2366 accesses of the locations designated by **x** that could potentially occur in parallel must be protected  
 2367 with an **atomic** construct.

2368 Atomic regions do not guarantee exclusive access with respect to any accesses outside of atomic re-  
 2369 gions to the same storage location **x** even if those accesses occur during the execution of a reduction  
 2370 clause.

2371 If the storage location designated by **x** is not size-aligned (that is, if the byte alignment of **x** is not a  
 2372 multiple of the size of **x**), then the behavior of the atomic region is implementation-defined.

### 2373 Restrictions

- 2374 • All atomic accesses to the storage locations designated by **x** throughout the program are  
 2375 required to have the same type and type parameters.
- 2376 • Storage locations designated by **x** must be less than or equal in size to the largest available  
 2377 native atomic operator width.

## 2378 2.13 Declare Directive

### 2379 Summary

2380 A **declare** directive is used in the declaration section of a Fortran subroutine, function, block  
 2381 construct, or module, or following a variable declaration in C or C++. It can specify that a *var* is to  
 2382 be allocated in device memory for the duration of the implicit data region of a function, subroutine  
 2383 or program, and specify whether the data values are to be transferred from local memory to device  
 2384 memory upon entry to the implicit data region, and from device memory to local memory upon exit  
 2385 from the implicit data region. These directives create a visible device copy of the *var*.

### 2386 Syntax

2387 In C and C++, the syntax of the **declare** directive is:

```
2388 #pragma acc declare clause-list new-line
```

2389 In Fortran the syntax of the **declare** directive is:

```
2390 !$acc declare clause-list
```

2391 where *clause* is one of the following:

```

2392 copy ( var-list )
2393 copyin ( [readonly:] var-list )
2394 copyout ( var-list )
2395 create ( var-list )
2396 present ( var-list )
2397 deviceptr ( var-list )
2398 device_resident ( var-list )
2399 link ( var-list )

```

2400 The associated region is the implicit region associated with the function, subroutine, or program in  
 2401 which the directive appears. If the directive appears in the declaration section of a Fortran *module*  
 2402 subprogram, for a Fortran *common block*, or in a C or C++ global or namespace scope, the associated  
 2403 region is the implicit region for the whole program. The **copy**, **copyin**, **copyout**, **present**,  
 2404 and **deviceptr** data clauses are described in Section 2.7 Data Clauses.

### 2405 Restrictions

- 2406 • A **declare** directive must be in the same scope as the declaration of any *var* that appears  
 2407 in the clauses of the directive or any scope within a C or C++ function or Fortran function,  
 2408 subroutine, or program.
- 2409 • At least one clause must appear on a **declare** directive.
- 2410 • A *var* in a **declare** declare must be a variable or array name, or a Fortran *common block*  
 2411 name between slashes.
- 2412 • A *var* may appear at most once in all the clauses of **declare** directives for a function,  
 2413 subroutine, program, or module.
- 2414 • In Fortran, assumed-size dummy arrays may not appear in a **declare** directive.
- 2415 • In Fortran, pointer arrays may appear, but pointer association is not preserved in device mem-  
 2416 ory.
- 2417 • In a Fortran *module* declaration section, only **create**, **copyin**, **device\_resident**, and  
 2418 **link** clauses are allowed.
- 2419 • In C or C++ global or namespace scope, only **create**, **copyin**, **deviceptr**, **device\_resident**  
 2420 and **link** clauses are allowed.
- 2421 • C and C++ *extern* variables may only appear in **create**, **copyin**, **deviceptr**, **device\_resident**  
 2422 and **link** clauses on a **declare** directive.
- 2423 • In C or C++, the **link** clause must appear at global or namespace scope or the arguments  
 2424 must be *extern* variables. In Fortran, the **link** clause must appear in a *module* declaration  
 2425 section, or the arguments must be *common block* names enclosed in slashes.
- 2426 • In C or C++, a **longjmp** call in the region must return to a **setjmp** call within the region.
- 2427 • In C++, an exception thrown in the region must be handled within the region.
- 2428 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional dummy arguments  
 2429 in data clauses, including **device\_resident** clauses.

### 2430 2.13.1 device\_resident clause

#### 2431 Summary

2432 The **device\_resident** clause specifies that the memory for the named variables should be  
 2433 allocated in the current device memory and not in local memory. The host may not be able to access  
 2434 variables in a **device\_resident** clause. The accelerator data lifetime of global variables or  
 2435 common blocks that appear in a **device\_resident** clause is the entire execution of the program.

2436 In Fortran, if the variable has the Fortran *allocatable* attribute, the memory for the variable will  
 2437 be allocated in and deallocated from the current device memory when the host thread executes  
 2438 an **allocate** or **deallocate** statement for that variable, if the current device is a non-shared  
 2439 memory device. If the variable has the Fortran *pointer* attribute, it may be allocated or deallocated  
 2440 by the host in the current device memory, or may appear on the left hand side of a pointer assignment  
 2441 statement, if the right hand side variable itself appears in a **device\_resident** clause.

2442 In Fortran, the argument to a **device\_resident** clause may be a *common block* name enclosed  
 2443 in slashes; in this case, all declarations of the common block must have a matching **device\_resident**  
 2444 clause. In this case, the *common block* will be statically allocated in device memory, and not  
 2445 in local memory. The *common block* will be available to accelerator routines; see Section 2.15  
 2446 Procedure Calls in Compute Regions.

2447 In a Fortran *module* declaration section, a *var* in a **device\_resident** clause will be available to  
 2448 accelerator subprograms.

2449 In C or C++ global scope, a *var* in a **device\_resident** clause will be available to accelerator  
 2450 routines. A C or C++ *extern* variable may appear in a **device\_resident** clause only if the  
 2451 actual declaration and all *extern* declarations are also followed by **device\_resident** clauses.

### 2452 2.13.2 create clause

2453 For data in shared memory, no action is taken.

2454 For data not in shared memory, the **create** clause on a **declare** directive behaves as follows,  
 2455 for each *var* in *var-list*:

- 2456 • At entry to an implicit data region where the **declare** directive appears:
  - 2457 – If *var* is present, a *present increment* action with the structured reference counter is  
 2458 performed. If *var* is a pointer reference, an *attach* action is performed.
  - 2459 – Otherwise, a *create* action with the structured reference counter is performed. If *var* is  
 2460 a pointer reference, an *attach* action is performed.
- 2461 • At exit from an implicit data region where the **declare** directive appears:
  - 2462 – If the structured reference counter for *var* is zero, no action is taken.
  - 2463 – Otherwise, a *present decrement* action with the structured reference counter is per-  
 2464 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured  
 2465 and dynamic reference counters are zero, a *delete* action is performed.

2466 If the **declare** directive appears in a global context, then the data in *var-list* is statically allocated  
 2467 in device memory and the structured reference counter is set to one.

2468 In Fortran, if a variable *var* in *var-list* has the Fortran *allocatable* or *pointer* attribute, then:

2469 • An **allocate** statement for *var* will allocate memory in both local memory as well as in the  
 2470 current device memory, for a non-shared memory device, and the dynamic reference counter  
 2471 will be set to one.

2472 • A **deallocate** statement for *var* will deallocate memory from both local memory as well  
 2473 as the current device memory, for a non-shared memory device, and the dynamic reference  
 2474 counter will be set to zero. If the structured reference counter is not zero, a runtime error is  
 2475 issued.

2476 In Fortran, if a variable *var* in *var-list* has the Fortran *pointer* attribute, then it may appear on the  
 2477 left hand side of a pointer assignment statement, if the right hand side variable itself appears in a  
 2478 **create** clause.

### 2479 2.13.3 link clause

2480 The **link** clause is used for large global host static data that is referenced within an accelerator  
 2481 routine and that should have a dynamic data lifetime on the device. The **link** clause specifies that  
 2482 only a global link for the named variables should be statically created in accelerator memory. The  
 2483 host data structure remains statically allocated and globally available. The device data memory will  
 2484 be allocated only when the global variable appears on a data clause for a **data** construct, compute  
 2485 construct, or **enter data** directive. The arguments to the **link** clause must be global data. A  
 2486 **declare link** clause must be visible everywhere the global variables or common block variables  
 2487 are explicitly or implicitly used in a data clause, compute construct, or accelerator routine. The  
 2488 global variable or *common block* variables may be used in accelerator routines. The accelerator  
 2489 data lifetime of variables or common blocks that appear in a **link** clause is the data region that  
 2490 allocates the variable or common block with a data clause, or from the execution of the **enter**  
 2491 **data** directive that allocates the data until an **exit data** directive deallocates it or until the end  
 2492 of the program.

## 2493 2.14 Executable Directives

### 2494 2.14.1 Init Directive

#### 2495 Summary

2496 The **init** directive tells the runtime to initialize the runtime for that device type. This can be used  
 2497 to isolate any initialization cost from the computational cost, when collecting performance statistics.  
 2498 If no device type appears all devices will be initialized. An **init** directive may be used in place of  
 2499 a call to the **acc\_init** runtime API routine, as described in Section 3.2.7.

#### 2500 Syntax

2501 In C and C++, the syntax of the **init** directive is:

```
2502 #pragma acc init [clause-list] new-line
```

2503 In Fortran the syntax of the **init** directive is:

```
2504 !$acc init [clause-list]
```

2505 where *clause* is one of the following:

```
2506 device_type ( device-type-list )
```

```
2507 device_num ( int-expr )
```

2508       **if** ( *condition* )

2509

### 2510 **device\_type** clause

2511 The **device\_type** clause specifies the type of device that is to be initialized in the runtime. If the  
2512 **device\_type** clause appears, then the *acc-current-device-type-var* for the current thread is set to  
2513 the argument value. If no **device\_num** clause appears then all devices of this type are initialized.

### 2514 **device\_num** clause

2515 The **device\_num** clause specifies the device id to be initialized. If the **device\_num** clause  
2516 appears, then the *acc-current-device-num-var* for the current thread is set to the argument value. If  
2517 no **device\_type** clause appears, then the specified device id will be initialized for all available  
2518 device types.

### 2519 **if** clause

2520 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to  
2521 perform the initialization unconditionally. When an **if** clause appears, the implementation will gen-  
2522 erate code to conditionally perform the initialization only when the *condition* evaluates to nonzero  
2523 in C or C++, or **.true.** in Fortran.

### 2524 **Restrictions**

- 2525       • This directive may not be called within a compute region.
- 2526       • If the device type specified is not available, the behavior is implementation-defined; in partic-  
2527       ular, the program may abort.
- 2528       • If the directive is called more than once without an intervening **acc\_shutdown** call or  
2529       **shutdown** directive, with a different value for the device type argument, the behavior is  
2530       implementation-defined.
- 2531       • If some accelerator regions are compiled to only use one device type, using this directive with  
2532       a different device type may produce undefined behavior.

## 2533 **2.14.2 Shutdown Directive**

### 2534 **Summary**

2535 The **shutdown** directive tells the runtime to shut down the connection to the given accelerator, and  
2536 free any runtime resources. This ends all data lifetimes in device memory, which effectively sets  
2537 structured and dynamic reference counters to zero. A **shutdown** directive may be used in place of  
2538 a call to the **acc\_shutdown** runtime API routine, as described in Section 3.2.8.

### 2539 **Syntax**

2540 In C and C++, the syntax of the **shutdown** directive is:

2541       **#pragma acc shutdown** [*clause-list*] *new-line*

2542 In Fortran the syntax of the **shutdown** directive is:

2543       **!\$acc shutdown** [*clause-list*]

2544 where *clause* is one of the following:

```
2545     device_type ( device-type-list )
2546     device_num ( int-expr )
2547     if ( condition )
2548
```

### 2549 **device\_type clause**

2550 The **device\_type** clause specifies the type of device that is to be disconnected from the runtime.  
2551 If no **device\_num** clause appears then all devices of this type are disconnected.

### 2552 **device\_num clause**

2553 The **device\_num** clause specifies the device id to be disconnected.  
2554 If no clauses appear then all available devices will be disconnected.

### 2555 **if clause**

2556 The **if** clause is optional; when there is no **if** clause, the implementation will generate code  
2557 to perform the shutdown unconditionally. When an **if** clause appears, the implementation will  
2558 generate code to conditionally perform the shutdown only when the *condition* evaluates to nonzero  
2559 in C or C++, or **.true.** in Fortran.

### 2560 **Restrictions**

- 2561 • This directive may not be used during the execution of a compute region.

## 2562 **2.14.3 Set Directive**

### 2563 **Summary**

2564 The **set** directive provides a means to modify internal control variables using directives. Each form  
2565 of the **set** directive is functionally equivalent to a matching runtime API routine.

### 2566 **Syntax**

2567 In C and C++, the syntax of the **set** directive is:

```
2568     #pragma acc set [clause-list] new-line
```

2569 In Fortran the syntax of the **set** directive is:

```
2570     !$acc set [clause-list]
```

2571 where *clause* is one of the following

```
2572     default_async ( int-expr )
2573     device_num ( int-expr )
2574     device_type ( device-type-list )
2575     if ( condition )
```



### 2576 **default\_async clause**

2577 The **default\_async** clause specifies the asynchronous queue that should be used if no queue ap-  
2578 pears and changes the value of *acc-default-async-var* for the current thread to the argument value.  
2579 If the value is **acc\_async\_default**, the value of *acc-default-async-var* will revert to the ini-  
2580 tial value, which is implementation-defined. A **set default\_async** directive is functionally  
2581 equivalent to a call to the **acc\_set\_default\_async** runtime API routine, as described in Sec-  
2582 tion 3.2.22.

### 2583 **device\_num clause**

2584 The **device\_num** clause specifies the device number to set as the default device for accelerator  
2585 regions and changes the value of *acc-current-device-num-var* for the current thread to the argument  
2586 value. If the value of **device\_num** argument is negative, the runtime will revert to the default be-  
2587 havior, which is implementation-defined. A **set device\_num** directive is functionally equivalent  
2588 to the **acc\_set\_device\_num** runtime API routine, as described in Section 3.2.4.

### 2589 **device\_type clause**

2590 The **device\_type** clause specifies the device type to set as the default device type for accelerator  
2591 regions and sets the value of *acc-current-device-type-var* for the current thread to the argument  
2592 value. If the value of the **device\_type** argument is zero or the clause does not appear, the  
2593 selected device number will be used for all attached accelerator types. A **set device\_type**  
2594 directive is functionally equivalent to a call to the **acc\_set\_device\_type** runtime API routine,  
2595 as described in Section 3.2.2.

### 2596 **if clause**

2597 The **if** clause is optional; when there is no **if** clause, the implementation will generate code  
2598 to perform the set operation unconditionally. When an **if** clause appears, the implementation  
2599 will generate code to conditionally perform the set operation only when the *condition* evaluates to  
2600 nonzero in C or C++, or **.true.** in Fortran.

### 2601 **Restrictions**

- 2602 • This directive may not be used within a compute region.
- 2603 • Passing **default\_async** the value of **acc\_async\_noval** has no effect.
- 2604 • Passing **default\_async** the value of **acc\_async\_sync** will cause all asynchronous  
2605 directives in the default asynchronous queue to become synchronous.
- 2606 • Passing **default\_async** the value of **acc\_async\_default** will restore the default  
2607 asynchronous queue to the initial value, which is implementation-defined.
- 2608 • If the value of **device\_num** is larger than the maximum supported value for the given type,  
2609 the behavior is implementation-defined.
- 2610 • At least one **default\_async**, **device\_num**, or **device\_type** clause must appear.
- 2611 • Two instances of the same clause may not appear on the same directive.

## 2.14.4 Update Directive

### Summary

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

### Syntax

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

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

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

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

where *clause* is one of the following:

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

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

### Restrictions

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

### self clause

The **self** clause specifies that the *vars* in *var-list* are to be copied from the current device memory to local memory for data not in shared memory. For data in shared memory, no action is taken. An **update** directive with the **self** clause is equivalent to a call to the **acc\_update\_self** routine, described in Section 3.2.31.

### host clause

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

### device clause

The **device** clause specifies that the *vars* in *var-list* are to be copied from local memory to the current device memory, for data not in shared memory. For data in shared memory, no action is taken. An **update** directive with the **device** clause is equivalent to a call to the **acc\_update\_device** routine, described in Section 3.2.30.

2649 **if clause**

2650 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to  
 2651 perform the updates unconditionally. When an **if** clause appears, the implementation will generate  
 2652 code to conditionally perform the updates only when the *condition* evaluates to nonzero in C or  
 2653 C++, or **.true.** in Fortran.

2654 **async clause**

2655 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2656 **wait clause**

2657 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

2658 **if\_present clause**

2659 When an **if\_present** clause appears on the directive, no action is taken for a *var* which appears  
 2660 in *var-list* that is not present in the current device memory. When no **if\_present** clause ap-  
 2661 pears, all *vars* in a **device** or **self** clause must be present in the current device memory, and an  
 2662 implementation may halt the program with an error message if some data is not present.

2663 **Restrictions**

- 2664 • The **update** directive is executable. It must not appear in place of the statement following  
 2665 an *if*, *while*, *do*, *switch*, or *label* in C or C++, or in place of the statement following a logical  
 2666 *if* in Fortran.
- 2667 • If no **if\_present** clause appears on the directive, each *var* in *var-list* must be present in  
 2668 the current device memory.
- 2669 • Only the **async** and **wait** clauses may follow a **device\_type** clause.
- 2670 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical  
 2671 value; in C or C++, the condition must evaluate to a scalar integer value.
- 2672 • Noncontiguous subarrays may appear. It is implementation-specific whether noncontiguous  
 2673 regions are updated by using one transfer for each contiguous subregion, or whether the non-  
 2674 contiguous data is packed, transferred once, and unpacked, or whether one or more larger  
 2675 subarrays (no larger than the smallest contiguous region that contains the specified subarray)  
 2676 are updated.
- 2677 • In C and C++, a member of a struct or class may appear, including a subarray of a member.  
 2678 Members of a subarray of struct or class type may not appear.
- 2679 • In C and C++, if a subarray notation is used for a struct member, subarray notation may not  
 2680 be used for any parent of that struct member.
- 2681 • In Fortran, members of variables of derived type may appear, including a subarray of a mem-  
 2682 ber. Members of subarrays of derived type may not appear.
- 2683 • In Fortran, if array or subarray notation is used for a derived type member, array or subarray  
 2684 notation may not be used for a parent of that derived type member.
- 2685 • See Section 2.17.1 Optional Arguments for discussion of Fortran optional arguments in **self**,  
 2686 **host**, and **device** clauses.

### 2687 2.14.5 Wait Directive

2688 See Section 2.16 Asynchronous Behavior for more information.

### 2689 2.14.6 Enter Data Directive

2690 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

### 2691 2.14.7 Exit Data Directive

2692 See Section 2.6.6 Enter Data and Exit Data Directives for more information.

## 2693 2.15 Procedure Calls in Compute Regions

2694 This section describes how routines are compiled for an accelerator and how procedure calls are  
 2695 compiled in compute regions. See Section 2.17.1 Optional Arguments for discussion of Fortran  
 2696 optional arguments in procedure calls inside compute regions.

### 2697 2.15.1 Routine Directive

#### 2698 Summary

2699 The **routine** directive is used to tell the compiler to compile a given procedure or a C++ *lambda*  
 2700 for an accelerator as well as for the host. In a file or routine with a procedure call, the **routine**  
 2701 directive tells the implementation the attributes of the procedure when called on the accelerator.

#### 2702 Syntax

2703 In C and C++, the syntax of the **routine** directive is:

```
2704     #pragma acc routine clause-list new-line
2705     #pragma acc routine( name ) clause-list new-line
```

2706 In C and C++, the **routine** directive without a name may appear immediately before a function  
 2707 definition, a C++ *lambda*, or just before a function prototype and applies to that immediately fol-  
 2708 lowing function or prototype. The **routine** directive with a name may appear anywhere that a  
 2709 function prototype is allowed and applies to the function or the C++ *lambda* in that scope with that  
 2710 name, but must appear before any definition or use of that function.

2711 In Fortran the syntax of the **routine** directive is:

```
2712     !$acc routine clause-list
2713     !$acc routine( name ) clause-list
```

2714 In Fortran, the **routine** directive without a name may appear within the specification part of a  
 2715 subroutine or function definition, or within an interface body for a subroutine or function in an  
 2716 interface block, and applies to the containing subroutine or function. The **routine** directive with  
 2717 a name may appear in the specification part of a subroutine, function or module, and applies to the  
 2718 named subroutine or function.

2719 A C or C++ function or Fortran subprogram compiled with the **routine** directive for an accelera-  
 2720 tor is called an *accelerator routine*.

2721 If an *accelerator routine* is a C++ *lambda*, the associated function will be compiled for both the  
 2722 accelerator and the host.

2723 If a *lambda* is called in a compute region and it is not an *accelerator routine*, then the *lambda* is  
2724 treated as if its name appears in the name list of a **routine** directive with **seq** clause. If *lambda*  
2725 is defined in an *accelerator routine* that has a **nohost** clause then the *lambda* is treated as if its  
2726 name appears in the name list of a **routine** directive with a **nohost** clause.

2727 The *clause* is one of the following:

```
2728     gang  
2729     worker  
2730     vector  
2731     seq  
2732     bind( name )  
2733     bind( string )  
2734     device_type( device-type-list )  
2735     nohost
```

2736 A **gang**, **worker**, **vector**, or **seq** clause specifies the *level of parallelism* in the routine.

### 2737 **gang clause**

2738 The **gang** clause specifies that the procedure contains, may contain, or may call another procedure  
2739 that contains a loop with a **gang** clause. A call to this procedure must appear in code that is  
2740 executed in *gang-redundant* mode, and all gangs must execute the call. For instance, a procedure  
2741 with a **routine gang** directive may not be called from within a loop that has a **gang** clause.  
2742 Only one of the **gang**, **worker**, **vector** and **seq** clauses may appear for each device type.

### 2743 **worker clause**

2744 The **worker** clause specifies that the procedure contains, may contain, or may call another pro-  
2745 cedure that contains a loop with a **worker** clause, but does not contain nor does it call another  
2746 procedure that contains a loop with the **gang** clause. A loop in this procedure with an **auto** clause  
2747 may be selected by the compiler to execute in **worker** or **vector** mode. A call to this procedure  
2748 must appear in code that is executed in *worker-single* mode, though it may be in *gang-redundant*  
2749 or *gang-partitioned* mode. For instance, a procedure with a **routine worker** directive may be  
2750 called from within a loop that has the **gang** clause, but not from within a loop that has the **worker**  
2751 clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may appear for each device  
2752 type.

### 2753 **vector clause**

2754 The **vector** clause specifies that the procedure contains, may contain, or may call another pro-  
2755 cedure that contains a loop with the **vector** clause, but does not contain nor does it call another  
2756 procedure that contains a loop with either a **gang** or **worker** clause. A loop in this procedure with  
2757 an **auto** clause may be selected by the compiler to execute in **vector** mode, but not **worker**  
2758 mode. A call to this procedure must appear in code that is executed in *vector-single* mode, though  
2759 it may be in *gang-redundant* or *gang-partitioned* mode, and in *worker-single* or *worker-partitioned*  
2760 mode. For instance, a procedure with a **routine vector** directive may be called from within  
2761 a loop that has the **gang** clause or the **worker** clause, but not from within a loop that has the  
2762 **vector** clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may appear for each  
2763 device type.

### 2764 **seq clause**

2765 The **seq** clause specifies that the procedure does not contain nor does it call another procedure that  
2766 contains a loop with a **gang**, **worker**, or **vector** clause. A loop in this procedure with an **auto**  
2767 clause will be executed in **seq** mode. A call to this procedure may appear in any mode. Only one  
2768 of the **gang**, **worker**, **vector** and **seq** clauses may appear for each device type.

### 2769 **bind clause**

2770 The **bind** clause specifies the name to use when calling the procedure on a device other than the  
2771 host. If the name is specified as an identifier, it is called as if that name were specified in the  
2772 language being compiled. If the name is specified as a string, the string is used for the procedure  
2773 name unmodified. A **bind** clause on a procedure definition behaves as if it had appeared on a  
2774 declaration by changing the name used to call the function on a device other than the host; however,  
2775 the procedure is not compiled for the device with either the original name or the name in the **bind**  
2776 clause.

2777 If there is both a Fortran bind and an acc **bind** clause for a procedure definition then a call on the  
2778 host will call the Fortran bound name and a call on another device will call the name in the **bind**  
2779 clause.

### 2780 **device\_type clause**

2781 The **device\_type** clause is described in Section 2.4 Device-Specific Clauses.

### 2782 **nohost clause**

2783 The **nohost** tells the compiler not to compile a version of this procedure for the host. All calls  
2784 to this procedure must appear within compute regions. If this procedure is called from other pro-  
2785 cedures, those other procedures must also have a matching **routine** directive with the **nohost**  
2786 clause.

### 2787 **Restrictions**

- 2788 • Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device\_type**  
2789 clause.
- 2790 • At least one of the (**gang**, **worker**, **vector**, or **seq**) clauses must appear on the construct.  
2791 If the **device\_type** clause appears on the **routine** directive, a default level of parallelism  
2792 clause must appear before the **device\_type** clause, or a level of parallelism clause must  
2793 appear following each **device\_type** clause on the directive.
- 2794 • In C and C++, function static variables are not supported in functions to which a **routine**  
2795 directive applies.
- 2796 • In Fortran, variables with the *save* attribute, either explicitly or implicitly, are not supported  
2797 in subprograms to which a **routine** directive applies.
- 2798 • A **bind** clause may not bind to a routine name that has a visible **bind** clause.
- 2799 • If a function or subroutine has a **bind** clause on both the declaration and the definition then  
2800 they both must bind to the same name.

## 2.15.2 Global Data Access

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

## 2.16 Asynchronous Behavior

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

### 2.16.1 async clause

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

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

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

The *async-value* of any operation is the value of the *async-argument*, if it appears, or the value of *acc-default-async-var* if it is **acc\_async\_noval** or if the **async** clause had no value, or **acc\_async\_sync** if no **async** clause appeared. If the current device supports asynchronous operation with one or more device activity queues, the *async-value* is used to select the queue on the current device onto which to enqueue an operation. The properties of the current device and the implementation will determine how many actual activity queues are supported, and how the *async-value* is mapped onto the actual activity queues. Two asynchronous operations with the same current device and the same *async-value* will be enqueued onto the same activity queue, and therefore will be executed on the device in the order they are encountered by the local thread. Two asynchronous operations with different *async-values* may be enqueued onto different activity queues, and therefore may be executed on the device in either order relative to each other. If there are two or more host threads executing and sharing the same device, two asynchronous operations with the same *async-*

2843 *value* will be enqueued on the same activity queue. If the threads are not synchronized with respect  
 2844 to each other, the operations may be enqueued in either order and therefore may execute on the  
 2845 device in either order. Asynchronous operations enqueued to different devices may execute in any  
 2846 order, regardless of the *async-value* used for each.

## 2847 2.16.2 wait clause

2848 The **wait** clause may appear on a **parallel**, **kernels**, or **serial** construct, or an **enter**  
 2849 **data**, **exit data**, or **update** directive. In all cases, the **wait** clause is optional. When there  
 2850 is no **wait** clause, the associated compute or update operations may be enqueued or launched or  
 2851 executed immediately on the device. If there is an argument to the **wait** clause, it must be a *wait-*  
 2852 *argument* (See 2.16.3). The compute, data, or update operation may not be launched or executed  
 2853 until all operations enqueued up to this point by this thread on the associated asynchronous device  
 2854 activity queues have completed. One legal implementation is for the local thread to wait for all  
 2855 the associated asynchronous device activity queues. Another legal implementation is for the local  
 2856 thread to enqueue the compute, data, or update operation in such a way that the operation will  
 2857 not start until the operations enqueued on the associated asynchronous device activity queues have  
 2858 completed.

## 2859 2.16.3 Wait Directive

### 2860 Summary

2861 The **wait** directive causes the local thread or a device activity queue on the current device to wait  
 2862 for completion of asynchronous operations, such as an accelerator **parallel**, **kernels**, or **serial** region  
 2863 or an **update** directive.

### 2864 Syntax

2865 In C and C++, the syntax of the **wait** directive is:

```
2866 #pragma acc wait [ ( wait-argument ) ] [ clause-list ] new-line
```

2867 In Fortran the syntax of the **wait** directive is:

```
2868 !$acc wait [ ( wait-argument ) ] [ clause-list ]
```

2869 where *clause* is:

```
2870 async [ ( int-expr ) ]
```

```
2871 if ( condition )
```

2872 The wait argument, if it appears, must be a *wait-argument* where *wait-argument* is:

```
2873 [ devnum : int-expr : ] [ queues : ] int-expr-list
```

2874 If there is no wait argument and no **async** clause, the local thread will wait until all operations  
 2875 enqueued by this thread on any activity queue on the current device have completed.

2876 If there are one or more *int-expr* expressions and no **async** clause, the local thread will wait  
 2877 until all operations enqueued by this thread on each of the associated device activity queues have  
 2878 completed. If a **devnum** modifier exists in the *wait-argument* then the device activity queues in the  
 2879 *int-expr* expressions apply to the queues on that device number of the current device type. If no  
 2880 **devnum** modifier exists then the expressions apply to the current device. It is an error to specify a



2881 device number that is not between 0 and the number of available devices of the current device type  
2882 minus 1.

2883 The **queues** modifier within a *wait-argument* is optional to improve clarity of the expression list.

2884 If there are two or more threads executing and sharing the same device, a **wait** directive with no  
2885 **async** clause will cause the local thread to wait until all of the appropriate asynchronous opera-  
2886 tions previously enqueued by that thread have completed. To guarantee that operations have been  
2887 enqueued by other threads requires additional synchronization between those threads. There is no  
2888 guarantee that all the similar asynchronous operations initiated by other threads will have completed.

2889 If there is an **async** clause, no new operation may be launched or executed on the **async** activ-  
2890 ity queue on the current device until all operations enqueued up to this point by this thread on the  
2891 asynchronous activity queues associated with the wait argument have completed. One legal imple-  
2892 mentation is for the local thread to wait for all the associated asynchronous device activity queues.  
2893 Another legal implementation is for the thread to enqueue a synchronization operation in such a  
2894 way that no new operation will start until the operations enqueued on the associated asynchronous  
2895 device activity queues have completed.

2896 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to  
2897 perform the wait operation unconditionally. When an **if** clause appears, the implementation will  
2898 generate code to conditionally perform the wait operation only when the *condition* evaluates to  
2899 nonzero in C or C++, or **.true.** in Fortran.

2900 A **wait** directive is functionally equivalent to a call to one of the **acc\_wait**, **acc\_wait\_async**,  
2901 **acc\_wait\_all** or **acc\_wait\_all\_async** runtime API routines, as described in Sections 3.2.13,  
2902 3.2.15, 3.2.17 and 3.2.19.

### 2903 Restrictions

- 2904 • The *int-expr* that appears in a **devnum** modifier must be a legal device number of the current  
2905 device type.

## 2906 2.17 Fortran Specific Behavior

### 2907 2.17.1 Optional Arguments

2908 This section refers to the Fortran intrinsic function **PRESENT**. A call to the Fortran intrinsic function  
2909 **PRESENT(arg)** returns **.true.**, if **arg** is an optional dummy argument and an actual argument  
2910 for **arg** was present in the argument list of the call site. This should not be confused with the  
2911 OpenACC **present** data clause.

2912 The appearance of a Fortran optional argument **arg** as a *var* in any of the following clauses has no  
2913 effect at runtime if **PRESENT(arg)** is **.false.:**

- 2914 • in data clauses on compute and **data** constructs;
- 2915 • in data clauses on **enter data** and **exit data** directives;
- 2916 • in data and **device\_resident** clauses on **declare** directives;
- 2917 • in **use\_device** clauses on **host\_data** directives;
- 2918 • in **self**, **host**, and **device** clauses on **update** directives.

2919 The appearance of a Fortran optional argument **arg** in the following situations may result in unde-  
2920 fined behavior if **PRESENT (arg)** is **.false.** when the associated construct is executed:

- 2921 • as a *var* in **private**, **firstprivate**, and **reduction** clauses;
- 2922 • as a *var* in **cache** directives;
- 2923 • as part of an expression in any clause or directive.

2924 A call to the Fortran intrinsic function **PRESENT** behaves the same way in a compute construct or  
2925 an accelerator routine as on the host. The function call **PRESENT (arg)** must return the same value  
2926 in a compute construct as **PRESENT (arg)** would outside of the compute construct. If a Fortran  
2927 optional argument **arg** appears as an actual argument in a procedure call in a compute construct  
2928 or an accelerator routine, and the associated dummy argument **subarg** also has the **optional**  
2929 attribute, then **PRESENT (subarg)** returns the same value as **PRESENT (subarg)** would when  
2930 executed on the host.

### 2931 2.17.2 Do Concurrent Construct

2932 This section refers to the Fortran **do concurrent** construct that is a form of **do** construct. When  
2933 **do concurrent** appears without a **loop** construct in a **kernels** construct it is treated as if it is  
2934 annotated with **loop auto**. If it appears in a **parallel** construct or an accelerator routine then  
2935 it is treated as if it is annotated with **loop independent**.

## 3. Runtime Library

2936

2937 This chapter describes the OpenACC runtime library routines that are available for use by program-  
2938 mers. Use of these routines may limit portability to systems that do not support the OpenACC API.  
2939 Conditional compilation using the `_OPENACC` preprocessor variable may preserve portability.

2940 This chapter has two sections:

- 2941 • Runtime library definitions
- 2942 • Runtime library routines

2943 There are four categories of runtime routines:

- 2944 • Device management routines, to get the number of devices, set the current device, and so on.
- 2945 • Asynchronous queue management, to synchronize until all activities on an async queue are  
2946 complete, for instance.
- 2947 • Device test routine, to test whether this statement is executing on the device or not.
- 2948 • Data and memory management, to manage memory allocation or copy data between memo-  
2949 ries.

### 3.1 Runtime Library Definitions

2950

2951 In C and C++, prototypes for the runtime library routines described in this chapter are provided in  
2952 a header file named `openacc.h`. All the library routines are *extern* functions with “C” linkage.  
2953 This file defines:

- 2954 • The prototypes of all routines in the chapter.
- 2955 • Any datatypes used in those prototypes, including an enumeration type to describe the sup-  
2956 ported device types.
- 2957 • The values of `acc_async_noval`, `acc_async_sync`, and `acc_async_default`.

2958 In Fortran, interface declarations are provided in a Fortran module named `openacc`. The `openacc`  
2959 module defines:

- 2960 • The integer parameter `openacc_version` with a value `yyyymm` where `yyyy` and `mm` are the  
2961 year and month designations of the version of the Accelerator programming model supported.  
2962 This value matches the value of the preprocessor variable `_OPENACC`.
- 2963 • Interfaces for all routines in the chapter.
- 2964 • Integer parameters to define integer kinds for arguments to and return values for those rou-  
2965 tines.
- 2966 • Integer parameters to describe the supported device types.
- 2967 • Integer parameters to define the values of `acc_async_noval`, `acc_async_sync`, and  
2968 `acc_async_default`.

2969 Many of the routines accept or return a value corresponding to the type of device. In C and C++, the  
 2970 datatype used for device type values is `acc_device_t`; in Fortran, the corresponding datatype  
 2971 is `integer(kind=acc_device_kind)`. The possible values for device type are implemen-  
 2972 tation specific, and are defined in the C or C++ include file `openacc.h` and the Fortran module  
 2973 `openacc`. Five values are always supported: `acc_device_none`, `acc_device_default`,  
 2974 `acc_device_host`, `acc_device_not_host`, and `acc_device_current`. For other val-  
 2975 ues, look at the appropriate files included with the implementation, or read the documentation for  
 2976 the implementation. The value `acc_device_default` will never be returned by any function;  
 2977 its use as an argument will tell the runtime library to use the default device type for that implemen-  
 2978 tation.

## 2979 3.2 Runtime Library Routines

2980 In this section, for the C and C++ prototypes, pointers are typed `h_void*` or `d_void*` to design-  
 2981 ate a host memory address or device memory address, when these calls are executed on the host,  
 2982 as if the following definitions were included:

```
2983     #define h_void void
2984     #define d_void void
```

2985 Except for `acc_on_device`, these routines are only available on the host.

### 2986 3.2.1 acc\_get\_num\_devices

#### 2987 Summary

2988 The `acc_get_num_devices` routine returns the number of available devices of the given type.

#### 2989 Format

2990 C or C++:

```
2991     int acc_get_num_devices(acc_device_t dev_type);
```

2992 Fortran:

```
2993     integer function acc_get_num_devices(dev_type)
2994     integer(acc_device_kind) :: dev_type
```

#### 2995 Description

2996 The `acc_get_num_devices` routine returns the number of available devices of device type  
 2997 `dev_type`.

#### 2998 Restrictions

- 2999 • This routine may not be called within a compute region.

### 3000 3.2.2 acc\_set\_device\_type

#### 3001 Summary

3002 The `acc_set_device_type` routine tells the runtime which type of device to use when exe-  
 3003 cuting a compute region and sets the value of `acc-current-device-type-var`. This is useful when the  
 3004 implementation allows the program to be compiled to use more than one type of device.

3005 **Format**

3006 C or C++:

3007 `void acc_set_device_type(acc_device_t dev_type);`

3008 Fortran:

3009 `subroutine acc_set_device_type(dev_type)`3010 `integer(acc_device_kind) :: dev_type`3011 **Description**

3012 The `acc_set_device_type` routine tells the runtime which type of device to use among those  
 3013 available and sets the value of *acc-current-device-type-var* for the current thread to `dev_type`. A  
 3014 call to `acc_set_device_type` is functionally equivalent to a `set device_type (dev_type)`  
 3015 directive, as described in Section 2.14.3.

3016 **Restrictions**

- 3017 • If the device type `dev_type` is not available, the behavior is implementation-defined; in  
 3018 particular, the program may abort.
- 3019 • If some compute regions are compiled to only use one device type, calling this routine with a  
 3020 different device type may produce undefined behavior.

3021 **3.2.3 acc\_get\_device\_type**3022 **Summary**

3023 The `acc_get_device_type` routine returns the value of *acc-current-device-type-var*, which is  
 3024 the device type of the current device. This is useful when the implementation allows the program to  
 3025 be compiled to use more than one type of device.

3026 **Format**

3027 C or C++:

3028 `acc_device_t acc_get_device_type(void);`

3029 Fortran:

3030 `function acc_get_device_type()`3031 `integer(acc_device_kind) :: acc_get_device_type`3032 **Description**

3033 The `acc_get_device_type` routine returns the value of *acc-current-device-type-var* for the  
 3034 current thread to tell the program what type of device will be used to run the next compute re-  
 3035 gion, if one has been selected. The device type may have been selected by the program with an  
 3036 `acc_set_device_type` call, with an environment variable, or by the default behavior of the  
 3037 program.

3038 **Restrictions**

- 3039 • If the device type has not yet been selected, the value `acc_device_none` may be returned.

3040 **3.2.4 acc\_set\_device\_num**3041 **Summary**

3042 The `acc_set_device_num` routine tells the runtime which device to use and sets the value of  
 3043 *acc-current-device-num-var*.

3044 **Format**

3045 C or C++:

3046 `void acc_set_device_num(int dev_num, acc_device_t dev_type);`

3047 Fortran:

3048 `subroutine acc_set_device_num(dev_num, dev_type)`3049 `integer :: dev_num`3050 `integer(acc_device_kind) :: dev_type`3051 **Description**

3052 The `acc_set_device_num` routine tells the runtime which device to use among those available  
 3053 of the given type for compute or data regions in the current thread and sets the value of *acc-current-*  
 3054 *device-num-var* to `dev_num`. If the value of `dev_num` is negative, the runtime will revert to its  
 3055 default behavior, which is implementation-defined. If the value of the `dev_type` is zero, the se-  
 3056 lected device number will be used for all device types. Calling `acc_set_device_num` implies  
 3057 a call to `acc_set_device_type(dev_type)`. A call to `acc_set_device_num` is func-  
 3058 tionally equivalent to a `set device_type(dev_type) device_num(dev_num)` directive,  
 3059 as described in Section 2.14.3.

3060 **Restrictions**

- 3061 • If the value of `dev_num` is greater than or equal to the value returned by `acc_get_num_devices`  
 3062 for that device type, the behavior is implementation-defined.

3063 **3.2.5 acc\_get\_device\_num**3064 **Summary**

3065 The `acc_get_device_num` routine returns the value of *acc-current-device-num-var* for the cur-  
 3066 rent thread.

3067 **Format**

3068 C or C++:

3069 `int acc_get_device_num(acc_device_t dev_type);`

3070 Fortran:

3071 `integer function acc_get_device_num(dev_type)`3072 `integer(acc_device_kind) :: dev_type`3073 **Description**

3074 The `acc_get_device_num` routine returns the value of *acc-current-device-num-var* for the cur-  
 3075 rent thread.

3076 **3.2.6 acc\_get\_property**3077 **Summary**

3078 The `acc_get_property` and `acc_get_property_string` routines return the value of a  
 3079 *device-property* for the specified device.

3080 **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,
3081                                  acc_device_property_t property);

```

Fortran:

```

    function acc_get_property(dev_num, dev_type, property)
    subroutine acc_get_property_string(dev_num, dev_type, &
3082                                     property, string)
3083     use iso_c_binding, only: c_size_t
3084     integer, value :: dev_num
3085     integer(acc_device_kind), value :: dev_type
3086     integer(acc_device_property_kind), value :: property
3087     integer(c_size_t) :: acc_get_property
3088     character(*) :: string

```

3089 **Description**

3090 The `acc_get_property` and `acc_get_property_string` routines return the value of the  
 3091 *property*. `dev_num` and `dev_type` specify the device being queried. If `dev_type` has the  
 3092 value `acc_device_current`, then `dev_num` is ignored and the value of the property for the  
 3093 current device is returned. `property` is an enumeration constant, defined in `openacc.h`, for  
 3094 C or C++, or an integer parameter, defined in the `openacc` module, for Fortran. Integer-valued  
 3095 properties are returned by `acc_get_property`, and string-valued properties are returned by  
 3096 `acc_get_property_string`. In Fortran, `acc_get_property_string` returns the result  
 3097 into the `string` argument.

3098 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 sup- 3099 ports 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

3100 An implementation may support additional properties for some devices.

3101 **Restrictions**

- 3102 • These routines may not be called within a compute region.
- 3103 • If the value of `property` is not one of the known values for that query routine, or that  
 3104 property has no value for the specified device, `acc_get_property` will return 0 and

3105 `acc_get_property_string` will return NULL (in C or C++) or an blank string (in  
3106 Fortran).

### 3107 **3.2.7 acc\_init**

#### 3108 **Summary**

3109 The `acc_init` routine tells the runtime to initialize the runtime for that device type. This can  
3110 be used to isolate any initialization cost from the computational cost, when collecting performance  
3111 statistics.

#### 3112 **Format**

3113 C or C++:

```
3114 void acc_init(acc_device_t dev_type);
```

3115 Fortran:

```
3116 subroutine acc_init(dev_type)  
3117 integer(acc_device_kind) :: dev_type
```

#### 3118 **Description**

3119 The `acc_init` routine also implicitly calls `acc_set_device_type(dev_type)`. A call  
3120 to `acc_init` is functionally equivalent to a `init device_type(dev_type)` directive, as  
3121 described in Section 2.14.1.

#### 3122 **Restrictions**

- 3123 • This routine may not be called within a compute region.
- 3124 • If the device type `dev_type` is not available, the behavior is implementation-defined; in  
3125 particular, the program may abort.
- 3126 • If the routine is called more than once without an intervening `acc_shutdown` call, with a  
3127 different value for the device type argument, the behavior is implementation-defined.
- 3128 • If some accelerator regions are compiled to only use one device type, calling this routine with  
3129 a different device type may produce undefined behavior.

### 3130 **3.2.8 acc\_shutdown**

#### 3131 **Summary**

3132 The `acc_shutdown` routine tells the runtime to shut down any connection to devices of the given  
3133 device type, and free up any runtime resources. This ends all data lifetimes in device memory, which  
3134 effectively sets structured and dynamic reference counters to zero.

#### 3135 **Format**

3136 C or C++:

```
3137 void acc_shutdown(acc_device_t dev_type);
```

3138 Fortran:

```
3139 subroutine acc_shutdown(dev_type)  
3140 integer(acc_device_kind) :: dev_type
```



3141 **Description**

3142 The **acc\_shutdown** routine disconnects the program from any device of device type **dev\_type**.  
 3143 Any data that is present in the memory of any such device is immediately deallocated. A call to  
 3144 **acc\_shutdown** is functionally equivalent to a **shutdown device\_type (dev\_type)** direc-  
 3145 tive, as described in Section 2.14.2.

3146 **Restrictions**

- 3147 • This routine may not be called during execution of a compute region.
- 3148 • If the program attempts to execute a compute region on a device or to access any data in  
 3149 the memory of a device after a call to **acc\_shutdown** for that device type, the behavior is  
 3150 undefined.
- 3151 • If the program attempts to shut down the **acc\_device\_host** device type, the behavior is  
 3152 undefined.

3153 **3.2.9 acc\_async\_test**3154 **Summary**

3155 The **acc\_async\_test** routine tests for completion of all associated asynchronous operations on  
 3156 the current device.

3157 **Format**

3158 C or C++:

```
3159     int acc_async_test(int wait_arg);
```

3160 Fortran:

```
3161     logical function acc_async_test(wait_arg)
3162     integer(acc_handle_kind) :: wait_arg
```

3163 **Description**

3164 **wait\_arg** must be an *async-argument* as defined in Section 2.16.1 *async* clause. If that value  
 3165 did not appear in any **async** clauses, or if it did appear in one or more **async** clauses and all  
 3166 such asynchronous operations have completed on the current device, the **acc\_async\_test** rou-  
 3167 tine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some such asyn-  
 3168 chronous operations have not completed, the **acc\_async\_test** routine will return with a zero  
 3169 value in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the  
 3170 **acc\_async\_test** routine will return with a nonzero value or **.true.** only if all matching asyn-  
 3171 chronous operations initiated by this thread have completed; there is no guarantee that all matching  
 3172 asynchronous operations initiated by other threads have completed.

3173 **3.2.10 acc\_async\_test\_device**3174 **Summary**

3175 The **acc\_async\_test\_device** routine tests for completion of all associated asynchronous op-  
 3176 erations on a device.

3177 **Format**

3178 C or C++:

```
3179     int acc_async_test_device(int wait_arg, int dev_num);
```

3180 Fortran:

```
3181     logical function acc_async_test_device(wait_arg, dev_num)
3182     integer(acc_handle_kind) :: wait_arg
3183     integer :: dev_num
```

### 3184 Description

3185 **wait\_arg** must be an *async-argument* as defined in Section 2.16.1 *async* clause. **dev\_num** must  
3186 be a valid device number of the current device type.

3187 If **wait\_arg** did not appear in any **async** clauses, or if it did appear in one or more **async** clauses  
3188 and all such asynchronous operations have completed on the device **dev\_num**, the **acc\_async\_test\_device**  
3189 routine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some such asyn-  
3190 chronous operations have not completed, the **acc\_async\_test\_device** routine will return  
3191 with a zero value in C and C++, or **.false.** in Fortran. If two or more threads share the same ac-  
3192 celerator, the **acc\_async\_test\_device** routine will return with a nonzero value or **.true.**  
3193 only if all matching asynchronous operations initiated by this thread have completed; there is no  
3194 guarantee that all matching asynchronous operations initiated by other threads have completed.

## 3195 3.2.11 acc\_async\_test\_all

### 3196 Summary

3197 The **acc\_async\_test\_all** routine tests for completion of all asynchronous operations.

### 3198 Format

3199 C or C++:

```
3200     int acc_async_test_all(void);
```

3201 Fortran:

```
3202     logical function acc_async_test_all()
```

### 3203 Description

3204 If all outstanding asynchronous operations have completed, the **acc\_async\_test\_all** routine  
3205 will return with a nonzero value in C and C++, or **.true.** in Fortran. If some asynchronous op-  
3206 erations have not completed, the **acc\_async\_test\_all** routine will return with a zero value  
3207 in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the  
3208 **acc\_async\_test\_all** routine will return with a nonzero value or **.true.** only if all outstand-  
3209 ing asynchronous operations initiated by this thread have completed; there is no guarantee that all  
3210 asynchronous operations initiated by other threads have completed.

## 3211 3.2.12 acc\_async\_test\_all\_device

### 3212 Summary

3213 The **acc\_async\_test\_all\_device** routine tests for completion of all asynchronous opera-  
3214 tions.

### 3215 Format

3216 C or C++:

```
3217     int acc_async_test_all_device(int dev_num);
```

3218 Fortran:

```
3219     logical function acc_async_test_all_device(dev_num)
3220     integer :: dev_num
```

3221 **Description**

3222 **dev\_num** must be a valid device number of the current device type. If all outstanding asynchronous  
 3223 operations have completed on device **dev\_num**, the **acc\_async\_test\_all\_device** routine  
 3224 will return with a nonzero value in C and C++, or **.true.** in Fortran. If some asynchronous oper-  
 3225 ations have not completed, the **acc\_async\_test\_all\_device** routine will return with a zero  
 3226 value in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the  
 3227 **acc\_async\_test\_all\_device** routine will return with a nonzero value or **.true.** only if all  
 3228 outstanding asynchronous operations initiated by this thread have completed; there is no guarantee  
 3229 that all asynchronous operations initiated by other threads have completed.

3230 **3.2.13 acc\_wait**3231 **Summary**

3232 The **acc\_wait** routine waits for completion of all associated asynchronous operations on the cur-  
 3233 rent device.

3234 **Format**

3235 C or C++:

```
3236     void acc_wait(int wait_arg);
```

3237 Fortran:

```
3238     subroutine acc_wait(wait_arg)
3239         integer(acc_handle_kind) :: wait_arg
```

3240 **Description**

3241 **wait\_arg** must be an *async-argument* as defined in Section 2.16.1 *async* clause. If **wait\_arg**  
 3242 appeared in one or more **async** clauses, the **acc\_wait** routine will not return until the latest  
 3243 such asynchronous operation has completed on the current device. If two or more threads share  
 3244 the same accelerator, the **acc\_wait** routine will return only if all matching asynchronous opera-  
 3245 tions initiated by this thread have completed; there is no guarantee that all matching asynchronous  
 3246 operations initiated by other threads have completed. For compatibility with version 1.0, this rou-  
 3247 tine may also be spelled **acc\_async\_wait**. A call to **acc\_wait** is functionally equivalent to a  
 3248 **wait(wait\_arg)** directive with no **async** clause, as described in Section 2.16.3.

3249 **3.2.14 acc\_wait\_device**3250 **Summary**

3251 The **acc\_wait\_device** routine waits for completion of all associated asynchronous operations  
 3252 on a device.

3253 **Format**

3254 C or C++:

```
3255     void acc_wait_device(int wait_arg, int dev_num);
```

3256 Fortran:

```
3257     subroutine acc_wait_device(wait_arg, dev_num)
3258         integer(acc_handle_kind) :: wait_arg
3259         integer :: dev_num
```

3260 **Description**

3261 **wait\_arg** must be an *async-argument* as defined in Section 2.16.1 *async* clause. **dev\_num** must  
3262 be a valid device number of the current device type.

3263 If **wait\_arg** appeared in one or more **async** clauses, the **acc\_wait** routine will not return  
3264 until the latest such asynchronous operation has completed on device **dev\_num**. If two or more  
3265 threads share the same accelerator, the **acc\_wait** routine will return only if all matching asyn-  
3266 chronous operations initiated by this thread have completed; there is no guarantee that all matching  
3267 asynchronous operations initiated by other threads have completed.

3268 **3.2.15 acc\_wait\_async**3269 **Summary**

3270 The **acc\_wait\_async** routine enqueues a wait operation on one *async* queue of the current  
3271 device for the operations previously enqueued on another *async* queue.

3272 **Format**

3273 C or C++:

```
3274     void acc_wait_async(int wait_arg, int async_arg);
```

3275 Fortran:

```
3276     subroutine acc_wait_async(wait_arg, async_arg)
3277         integer(acc_handle_kind) :: wait_arg, async_arg
```

3278 **Description**

3279 The arguments must be *async-arguments*, as defined in Section 2.16.1 *async* clause. The routine  
3280 will enqueue a wait operation on the *async* queue associated with **async\_arg**, which will wait  
3281 for operations enqueued on the *async* queue associated with the **wait\_arg**. See Section 2.16  
3282 Asynchronous Behavior for more information. A call to **acc\_wait\_async** is functionally equiv-  
3283 alent to a **wait(wait\_arg)** directive with an **async(async\_arg)** clause, as described in  
3284 Section 2.16.3.

3285 **3.2.16 acc\_wait\_device\_async**3286 **Summary**

3287 The **acc\_wait\_device\_async** routine enqueues a wait operation on one *async* queue of a  
3288 device for the operations previously enqueued on another *async* queue.

3289 **Format**

C or C++:

```
3290     void acc_wait_device_async(int wait_arg, int async_arg,
3291                               int dev_num);
```

3291 Fortran:

```
3292     subroutine acc_wait_device_async(wait_arg, async_arg, dev_num)
3293         integer(acc_handle_kind) :: wait_arg, async_arg
3294         integer :: dev_num
```

3295 **Description**

3296 The first two arguments must be *async-arguments*, as defined in Section 2.16.1 *async* clause. **dev\_num**  
3297 must be a valid device number of the current device type.

3298 The routine will enqueue a wait operation on the *async* queue associated with **async\_arg** on  
3299 the current device, which will wait for operations enqueued on the *async* queue associated with  
3300 **wait\_arg** on device **dev\_num**.

3301 See Section 2.16 Asynchronous Behavior for more information. A call to **acc\_wait\_device\_async**  
3302 is functionally equivalent to a **wait (devnum:dev\_num, queues:wait\_arg)** directive with  
3303 an **async (async\_arg)** clause, as described in Section 2.16.3.

3304 **3.2.17 acc\_wait\_all**3305 **Summary**

3306 The **acc\_wait\_all** routine waits for completion of all asynchronous operations.

3307 **Format**

3308 C or C++:

```
3309     void acc_wait_all(void);
```

3310 Fortran:

```
3311     subroutine acc_wait_all()
```

3312 **Description**

3313 The **acc\_wait\_all** routine will not return until the all asynchronous operations have completed.  
3314 If two or more threads share the same accelerator, the **acc\_wait\_all** routine will return only if  
3315 all asynchronous operations initiated by this thread have completed; there is no guarantee that all  
3316 asynchronous operations initiated by other threads have completed. For compatibility with version  
3317 1.0, this routine may also be spelled **acc\_async\_wait\_all**. A call to **acc\_wait\_all** is  
3318 functionally equivalent to a **wait** directive with no argument and no **async** clause, as described  
3319 in Section 2.16.3.

3320 **3.2.18 acc\_wait\_all\_device**3321 **Summary**

3322 The **acc\_wait\_all\_device** routine waits for completion of all asynchronous operations the  
3323 specified device.

3324 **Format**

3325 C or C++:

```
3326     void acc_wait_all_device(int dev_num);
```

3327 Fortran:

```
3328     subroutine acc_wait_all_device(dev_num)
```

```
3329     integer :: dev_num
```

3330 **Description**

3331 **dev\_num** must be a valid device number of the current device type. The **acc\_wait\_all\_device**  
3332 routine will not return until the all asynchronous operations have completed on device **dev\_num**. If  
3333 two or more threads share the same accelerator, the **acc\_wait\_all\_device** routine will return  
3334 only if all asynchronous operations initiated by this thread have completed; there is no guarantee  
3335 that all asynchronous operations initiated by other threads have completed.

### 3336 3.2.19 `acc_wait_all_async`

#### 3337 Summary

3338 The `acc_wait_all_async` routine enqueues wait operations on one async queue for the oper-  
3339 ations previously enqueued on all other async queues.

#### 3340 Format

3341 C or C++:

```
3342     void acc_wait_all_async(int async_arg);
```

3343 Fortran:

```
3344     subroutine acc_wait_all_async(async_arg)
3345         integer(acc_handle_kind) :: async_arg
```

#### 3346 Description

3347 `async_arg` must be an *async-argument* as defined in Section 2.16.1 `async` clause. The rou-  
3348 tine will enqueue a wait operation on the async queue associated with `async_arg` for each  
3349 other async queue. See Section 2.16 Asynchronous Behavior for more information. A call to  
3350 `acc_wait_all_async` is functionally equivalent to a `wait` directive with no argument and  
3351 an `async(async_arg)` clause, as described in Section 2.16.3.

### 3352 3.2.20 `acc_wait_all_device_async`

#### 3353 Summary

3354 The `acc_wait_all_device_async` routine enqueues wait operations on one async queue for  
3355 the operations previously enqueued on all other async queues on the specified device.

#### 3356 Format

3357 C or C++:

```
3358     void acc_wait_all_device_async(int async_arg, int dev_num);
```

3359 Fortran:

```
3360     subroutine acc_wait_all_device_async(async_arg, dev_num)
3361         integer(acc_handle_kind) :: async_arg
3362         integer :: dev_num
```

#### 3363 Description

3364 `async_arg` must be an *async-argument* as defined in Section 2.16.1 `async` clause. `dev_num`  
3365 must be a valid device number of the current device type.

3366 The routine will enqueue a wait operation on the async queue associated with `async_arg` on the  
3367 current device for each async queue of device `dev_num`. See Section 2.16 Asynchronous Behavior  
3368 for more information. A call to `acc_wait_all_device_async` is functionally equivalent to  
3369 a `wait(devnum:dev_num)` directive with an `async(async_arg)` clause, as described in  
3370 Section 2.16.3.

### 3371 3.2.21 `acc_get_default_async`

#### 3372 Summary

3373 The `acc_get_default_async` routine returns the value of *acc-default-async-var* for the cur-  
3374 rent thread.

3375 **Format**

3376 C or C++:

3377 `int acc_get_default_async(void);`

3378 Fortran:

3379 `function acc_get_default_async()`3380 `integer(acc_handle_kind) :: acc_get_default_async`3381 **Description**

3382 The `acc_get_default_async` routine returns the value of *acc-default-async-var* for the current thread, which is the asynchronous queue used when an **async** clause appears without an *async-argument* or with the value `acc_async_noval`.

3385 **3.2.22 acc\_set\_default\_async**3386 **Summary**

3387 The `acc_set_default_async` routine tells the runtime which asynchronous queue to use when an **async** clause appears with no queue argument.

3389 **Format**

3390 C or C++:

3391 `void acc_set_default_async(int async_arg);`

3392 Fortran:

3393 `subroutine acc_set_default_async(async_arg)`3394 `integer(acc_handle_kind) :: async_arg`3395 **Description**

3396 The `acc_set_default_async` routine tells the runtime to place any directives with an **async** clause that does not have an *async-argument* or with the special `acc_async_noval` value into the asynchronous activity queue associated with `async_arg` instead of the default asynchronous activity queue for that device by setting the value of *acc-default-async-var* for the current thread.

3400 The special argument `acc_async_default` will reset the default asynchronous activity queue to the initial value, which is implementation-defined. A call to `acc_set_default_async` is functionally equivalent to a `set default_async(async_arg)` directive, as described in Section 2.14.3.

3404 **3.2.23 acc\_on\_device**3405 **Summary**

3406 The `acc_on_device` routine tells the program whether it is executing on a particular device.

3407 **Format**

3408 C or C++:

3409 `int acc_on_device(acc_device_t dev_type);`

3410 Fortran:

3411 `logical function acc_on_device(dev_type)`3412 `integer(acc_device_kind) :: dev_type`

### 3413 **Description**

3414 The `acc_on_device` routine may be used to execute different paths depending on whether  
3415 the code is running on the host or on some accelerator. If the `acc_on_device` routine has  
3416 a compile-time constant argument, it evaluates at compile time to a constant. `dev_type` must  
3417 be one of the defined accelerator types. If the argument is `acc_device_host`, then outside  
3418 of a compute region or accelerator routine, or in a compute region or accelerator routine that  
3419 is executed on the host CPU, this routine will evaluate to nonzero for C or C++, and `.true.`  
3420 for Fortran; otherwise, it will evaluate to zero for C or C++, and `.false.` for Fortran. If the  
3421 argument is `acc_device_not_host`, the result is the negation of the result with argument  
3422 `acc_device_host`. If the argument is an accelerator device type, then in a compute region  
3423 or routine that is executed on a device of that type, this routine will evaluate to nonzero for C or  
3424 C++, and `.true.` for Fortran; otherwise, it will evaluate to zero for C or C++, and `.false.` for  
3425 Fortran. The result with argument `acc_device_default` is undefined.

### 3426 **3.2.24 acc\_malloc**

#### 3427 **Summary**

3428 The `acc_malloc` routine allocates space in the current device memory.

#### 3429 **Format**

3430 C or C++:

```
3431     d_void* acc_malloc(size_t bytes);
```

#### 3432 **Description**

3433 The `acc_malloc` routine may be used to allocate space in the current device memory. Pointers  
3434 assigned from this routine may be used in `deviceptr` clauses to tell the compiler that the pointer  
3435 target is resident on the device. In case of an error, `acc_malloc` returns a NULL pointer.

### 3436 **3.2.25 acc\_free**

#### 3437 **Summary**

3438 The `acc_free` routine frees memory on the current device.

#### 3439 **Format**

3440 C or C++:

```
3441     void acc_free(d_void* data_dev);
```

#### 3442 **Description**

3443 The `acc_free` routine will free previously allocated space in the current device memory; `data_dev`  
3444 should be a pointer value that was returned by a call to `acc_malloc`. If the argument is a NULL  
3445 pointer, no operation is performed.

### 3446 **3.2.26 acc\_copyin**

#### 3447 **Summary**

3448 The `acc_copyin` routines test to see if the argument is in shared memory or already present in  
3449 the current device memory; if not, they allocate space in the current device memory to correspond  
3450 to the specified local memory, and copy the data to that device memory.



3451 **Format**

C or C++:

```

3451     d_void* acc_copyin(h_void* data_arg, size_t bytes);
3452     void acc_copyin_async(h_void* data_arg, size_t bytes,
                             int async_arg);

```

3453 Fortran:

```

3454     subroutine acc_copyin(data_arg)
3455     subroutine acc_copyin(data_arg, bytes)
3456     subroutine acc_copyin_async(data_arg, async_arg)
3457     subroutine acc_copyin_async(data_arg, bytes, async_arg)
3458     type(*) , dimension(..) :: data_arg
3459     integer :: bytes
3460     integer(acc_handle_kind) :: async_arg

```

3461 **Description**

3462 The **acc\_copyin** routines are equivalent to an **enter data** directive with a **copyin** clause, as  
3463 described in Section 2.7.7. In C/C++, **data\_arg** is a pointer to the data, and **bytes** specifies the  
3464 data size in bytes. The synchronous routine returns a pointer to the allocated device memory, as  
3465 with **acc\_malloc**. In Fortran, two forms are supported. In the first, **data\_arg** is a variable or  
3466 a contiguous array section. In the second, **data\_arg** is a variable or array element and **bytes**  
3467 is the length in bytes. For the **\_async** versions of these routines, **async\_arg** must be an *async-*  
3468 *argument* as defined in Section 2.16.1 *async* clause.

3469 The behavior of the **acc\_copyin** routines for the data referred to by **data\_arg** is:

- 3470 • If the data is in shared memory, no action is taken. The C/C++ **acc\_copyin** routine returns  
3471 the incoming pointer.
- 3472 • If the data is present in the current device memory, a *present increment* action with the dy-  
3473 namic reference counter is performed. The C/C++ **acc\_copyin** routine returns a pointer to  
3474 the existing device memory.
- 3475 • Otherwise, a *copyin* action with the dynamic reference counter is performed. The C/C++  
3476 **acc\_copyin** routine returns the device address of the newly allocated memory.

3477 This data may be accessed using the **present** data clause. Pointers assigned from the C/C++  
3478 **acc\_copyin** routine may be used in **deviceptr** clauses to tell the compiler that the pointer  
3479 target is resident on the device.

3480 The **\_async** versions of these routines will perform any data transfers asynchronously on the *async*  
3481 queue associated with **async\_arg**. The routine may return before the data has been transferred;  
3482 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return  
3483 until the data has been completely transferred.

3484 For compatibility with OpenACC 2.0, **acc\_present\_or\_copyin** and **acc\_pcopyin** are al-  
3485 ternate names for **acc\_copyin**.

3486 **3.2.27 acc\_create**

3487 **Summary**

3488 The **acc\_create** routines test to see if the argument is in shared memory or already present in  
 3489 the current device memory; if not, they allocate space in the current device memory to correspond  
 3490 to the specified local memory.

3491 **Format**

C or C++:

```

d_void* acc_create(h_void* data_arg, size_t bytes);
void acc_create_async(h_void* data_arg, size_t bytes,
3492 int async_arg);

```

3493 Fortran:

```

3494 subroutine acc_create(data_arg)
3495 subroutine acc_create(data_arg, bytes)
3496 subroutine acc_create_async(data_arg, async_arg)
3497 subroutine acc_create_async(data_arg, bytes, async_arg)
3498 type(*), dimension(..) :: data_arg
3499 integer :: bytes
3500 integer(acc_handle_kind) :: async_arg

```

3501 **Description**

3502 The **acc\_create** routines are equivalent to an **enter data** directive with a **create** clause, as  
 3503 described in Section 2.7.9. The arguments are as for **acc\_copyin**.

3504 The behavior of the **acc\_create** routines for the data referred to by **data\_arg** is:

- 3505 • If the data is in shared memory, no action is taken. The C/C++ **acc\_create** routine returns  
 3506 the incoming pointer.
- 3507 • If the data is present in the current device memory, a *present increment* action with the dy-  
 3508 namic reference counter is performed. The C/C++ **acc\_create** routine returns a pointer to  
 3509 the existing device memory.
- 3510 • Otherwise, a *create* action with the dynamic reference counter is performed. The C/C++  
 3511 **acc\_create** routine returns the device address of the newly allocated memory.

3512 This data may be accessed using the **present** data clause. Pointers assigned from the C/C++  
 3513 **acc\_create** routine may be used in **deviceptr** clauses to tell the compiler that the pointer  
 3514 target is resident on the device.

3515 The **\_async** versions of these routines may perform the data allocation asynchronously on the  
 3516 async queue associated with **async\_arg**. The synchronous versions will not return until the data  
 3517 has been allocated.

3518 For compatibility with OpenACC 2.0, **acc\_present\_or\_create** and **acc\_pcreate** are al-  
 3519 ternate names for **acc\_create**.

3520 **3.2.28 acc\_copyout**3521 **Summary**

3522 The **acc\_copyout** routines test to see if the argument is in shared memory; if not, the argument  
 3523 must be present in the current device memory, and the routines copy data from device memory to  
 3524 the corresponding local memory, then deallocate that space from the device memory.

3525 **Format**

C or C++:

```

void acc_copyout(h_void* data_arg, size_t bytes);
void acc_copyout_async(h_void* data_arg, size_t bytes,
                      int async_arg);
void acc_copyout_finalize(h_void* data_arg, size_t bytes);
void acc_copyout_finalize_async(h_void* data_arg, size_t bytes,
                                int async_arg);

```

3526

Fortran:

```

subroutine acc_copyout(data_arg)
subroutine acc_copyout(data_arg, bytes)
subroutine acc_copyout_async(data_arg, async_arg)
subroutine acc_copyout_async(data_arg, bytes, async_arg)
subroutine acc_copyout_finalize(data_arg)
subroutine acc_copyout_finalize(data_arg, bytes)
subroutine acc_copyout_finalize_async(data_arg, async_arg)
subroutine acc_copyout_finalize_async(data_arg, bytes, &
                                     async_arg)
type(*), dimension(..) :: data_arg
integer :: bytes
integer(acc_handle_kind) :: async_arg

```

3527

3528

3529

3530

3531 **Description**

3532 The **acc\_copyout** routines are equivalent to an **exit data** directive with a **copyout** clause,  
 3533 and the **acc\_copyout\_finalize** routines are equivalent to an **exit data** directive with  
 3534 both **copyout** and **finalize** clauses, as described in Section 2.7.8. The arguments are as for  
 3535 **acc\_copyin**.

3536 The behavior of the **acc\_copyout** routines for the data referred to by **data\_arg** is:

3537

- If the data is in shared memory, no action is taken.

3538

- Otherwise, if the dynamic reference counter for the data is zero, no action is taken.

3539

- Otherwise, a *present decrement* action with the dynamic reference counter is performed (**acc\_copyout**),  
 3540 or the dynamic reference counter is set to zero (**acc\_copyout\_finalize**). If both refer-  
 3541 ence counters are then zero, a *copyout* action is performed.

3542

3543

3544

3545

3546

3547

The **\_async** versions of these routines will perform any associated data transfers asynchronously  
 on the async queue associated with **async\_arg**. The routine may return before the data has  
 been transferred or deallocated; see Section 2.16 Asynchronous Behavior for more details. The  
 synchronous versions will not return until the data has been completely transferred. Even if the data  
 has not been transferred or deallocated before the routine returns, the data will be treated as not  
 present in the current device memory.

3548 **3.2.29 acc\_delete**3549 **Summary**

3550

The **acc\_delete** routines test to see if the argument is in shared memory; if not, the argument  
 3551 must be present in the current device memory, and the routines deallocate that space from the device

3552 memory.

### 3553 **Format**

C or C++:

```

void acc_delete(h_void* data_arg, size_t bytes);
void acc_delete_async(h_void* data_arg, size_t bytes,
                     int async_arg);
void acc_delete_finalize(h_void* data_arg, size_t bytes);
void acc_delete_finalize_async(h_void* data_arg,
                               size_t bytes, int async_arg);

```

3554

Fortran:

```

subroutine acc_delete(data_arg)
subroutine acc_delete(data_arg, bytes)
subroutine acc_delete_async(data_arg, async_arg)
subroutine acc_delete_async(data_arg, bytes, async_arg)
subroutine acc_delete_finalize(data_arg)
subroutine acc_delete_finalize(data_arg, bytes)
subroutine acc_delete_finalize_async(data_arg, async_arg)
subroutine acc_delete_finalize_async(data_arg, bytes, &
                                     async_arg)
type(*), dimension(..) :: data_arg
integer :: bytes
integer(acc_handle_kind) :: async_arg

```

3555

3556

3557

3558

### 3559 **Description**

3560 The **acc\_delete** routines are equivalent to an **exit data** directive with a **delete** clause,  
 3561 and the **acc\_delete\_finalize** routines are equivalent to an **exit data** directive with both  
 3562 **delete** clause and **finalize** clauses, as described in Section 2.7.11. The arguments are as for  
 3563 **acc\_copyin**.

3564 The behavior of the **acc\_delete** routines for the data referred to by **data\_arg** is:

- 3565 • If the data is in shared memory, no action is taken.
- 3566 • Otherwise, if the dynamic reference counter for the data is zero, no action is taken.
- 3567 • Otherwise, a *present decrement* action with the dynamic reference counter is performed (**acc\_delete**),  
 3568 or the dynamic reference counter is set to zero (**acc\_delete\_finalize**). If both refer-  
 3569 ence counters are then zero, a *delete* action is performed.

3570 The **\_async** versions of these routines may perform the data deallocation asynchronously on the  
 3571 **async** queue associated with **async\_arg**. Even if the data has not been deallocated before the rou-  
 3572 tine returns, the data will be treated as not present in the current device memory. The synchronous  
 3573 versions will not return until the data has been deallocated.

## 3574 **3.2.30 acc\_update\_device**

### 3575 **Summary**

3576 The **acc\_update\_device** routines test to see if the argument is in shared memory; if not, the  
 3577 argument must be present in the current device memory, and the routines update the data in device  
 3578 memory from the corresponding local memory.

3579 **Format**

C or C++:

```

3579 void acc_update_device(h_void* data_arg, size_t bytes);
3580 void acc_update_device_async(h_void* data_arg, size_t bytes,
                               int async_arg);

```

3581 Fortran:

```

3582 subroutine acc_update_device(data_arg)
3583 subroutine acc_update_device(data_arg, bytes)
3584 subroutine acc_update_device_async(data_arg, async_arg)
3585 subroutine acc_update_device_async(data_arg, bytes, async_arg)
3586 type(*), dimension(..) :: data_arg
3587 integer :: bytes
3588 integer(acc_handle_kind) :: async_arg

```

3589 **Description**

3590 The `acc_update_device` routine is equivalent to an `update` directive with a `device` clause,  
 3591 as described in Section 2.14.4. The arguments are as for `acc_copyin`. For the data referred  
 3592 to by `data_arg`, if data is not in shared memory, the data in the local memory is copied to the  
 3593 corresponding device memory. It is a runtime error to call this routine if the data is not present in  
 3594 the current device memory.

3595 The `_async` versions of these routines will perform the data transfers asynchronously on the async  
 3596 queue associated with `async_arg`. The routine may return before the data has been transferred;  
 3597 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return  
 3598 until the data has been completely transferred.

3599 **3.2.31 acc\_update\_self**3600 **Summary**

3601 The `acc_update_self` routines test to see if the argument is in shared memory; if not, the  
 3602 argument must be present in the current device memory, and the routines update the data in local  
 3603 memory from the corresponding device memory.

3604 **Format**

C or C++:

```

3605 void acc_update_self(h_void* data_arg, size_t bytes);
3606 void acc_update_self_async(h_void* data_arg, size_t bytes,
                             int async_arg);

```

3606 Fortran:

```

3607 subroutine acc_update_self(data_arg)
3608 subroutine acc_update_self(data_arg, bytes)
3609 subroutine acc_update_self_async(data_arg, async_arg)
3610 subroutine acc_update_self_async(data_arg, bytes, async_arg)
3611 type(*), dimension(..) :: data_arg
3612 integer :: bytes
3613 integer(acc_handle_kind) :: async_arg

```

### 3614 **Description**

3615 The `acc_update_self` routine is equivalent to an `update` directive with a `self` clause, as  
3616 described in Section 2.14.4. The arguments are as for `acc_copyin`. For the data referred to by  
3617 `data_arg`, if the data is not in shared memory, the data in the local memory is copied to the  
3618 corresponding device memory. There must be a device copy of the data on the device when calling  
3619 this routine. It is a runtime error to call this routine if the data is not present in the current device  
3620 memory.

3621 The `_async` versions of these routines will perform the data transfers asynchronously on the async  
3622 queue associated with `async_arg`. The routine may return before the data has been transferred;  
3623 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return  
3624 until the data has been completely transferred.

### 3625 **3.2.32 acc\_map\_data**

#### 3626 **Summary**

3627 The `acc_map_data` routine maps previously allocated space in the current device memory to the  
3628 specified host data.

#### 3629 **Format**

C or C++:

```
3630     void acc_map_data(h_void* data_arg, d_void* data_dev,  
                      size_t bytes);
```

#### 3631 **Description**

3632 The `acc_map_data` routine is similar to an `enter data` directive with a `create` clause, except  
3633 that instead of allocating new device memory to start a data lifetime, the device address to use for  
3634 the data lifetime is specified as an argument. `data_arg` is a host address, `data_dev` is the  
3635 corresponding device address, and `bytes` is the length in bytes. `data_dev` may be the result of  
3636 a call to `acc_malloc`, or may come from some other device-specific API routine. After this call,  
3637 when the host data appears in a data clause, the specified device memory will be used. It is an error  
3638 to call `acc_map_data` for host data that is already present in the current device memory. It is  
3639 undefined to call `acc_map_data` with a device address that is already mapped to host data. After  
3640 mapping the device memory, the dynamic reference count for the host data is set to one, but no data  
3641 movement will occur. Memory mapped by `acc_map_data` may not have the associated dynamic  
3642 reference count decremented to zero, except by a call to `acc_unmap_data`. See Section 2.6.7  
3643 Reference Counters.

### 3644 **3.2.33 acc\_unmap\_data**

#### 3645 **Summary**

3646 The `acc_unmap_data` routine unmaps device data from the specified host data.

#### 3647 **Format**

3648 C or C++:

```
3649     void acc_unmap_data(h_void* data_arg);
```

### 3650 Description

3651 The `acc_unmap_data` routine is similar to an `exit data` directive with a `delete` clause, ex-  
3652 cept the device memory is not deallocated. `data_arg` is a host address. A call to this routine ends  
3653 the data lifetime for the specified host data. The device memory is not deallocated. It is undefined  
3654 behavior to call `acc_unmap_data` with a host address unless that host address was mapped to  
3655 device memory using `acc_map_data`. After unmapping memory the dynamic reference count for  
3656 the pointer is set to zero, but no data movement will occur. It is an error to call `acc_unmap_data`  
3657 if the structured reference count for the pointer is not zero. See Section 2.6.7 Reference Counters.

### 3658 3.2.34 `acc_deviceptr`

#### 3659 Summary

3660 The `acc_deviceptr` routine returns the device pointer associated with a specific host address.

#### 3661 Format

3662 C or C++:

```
3663     d_void* acc_deviceptr(h_void* data_arg);
```

#### 3664 Description

3665 The `acc_deviceptr` routine returns the device pointer associated with a host address. `data_arg`  
3666 is the address of a host variable or array that has an active lifetime on the current device. If the data  
3667 is not present in the current device memory, the routine returns a NULL value.

### 3668 3.2.35 `acc_hostptr`

#### 3669 Summary

3670 The `acc_hostptr` routine returns the host pointer associated with a specific device address.

#### 3671 Format

3672 C or C++:

```
3673     h_void* acc_hostptr(d_void* data_dev);
```

#### 3674 Description

3675 The `acc_hostptr` routine returns the host pointer associated with a device address. `data_dev`  
3676 is the address of a device variable or array, such as that returned from `acc_deviceptr`, `acc_create`  
3677 or `acc_copyin`. If the device address is NULL, or does not correspond to any host address, the  
3678 routine returns a NULL value.

### 3679 3.2.36 `acc_is_present`

#### 3680 Summary

3681 The `acc_is_present` routine tests whether a variable or array region is accessible from the  
3682 current device.

#### 3683 Format

3684 C or C++:

```
3685     int acc_is_present(h_void* data_arg, size_t bytes);
```

3686 Fortran:

```
3687     logical function acc_is_present(data_arg)
```

```

3688     logical function acc_is_present(data_arg, bytes)
3689         type(*), dimension(..) :: data_arg
3690         integer :: bytes

```

### 3691 Description

3692 The `acc_is_present` routine tests whether the specified host data is accessible from the current  
3693 device. In C/C++, `data_arg` is a pointer to the data, and `bytes` specifies the data size in bytes;  
3694 the routine returns nonzero if the specified data is fully present, and zero otherwise. In Fortran, two  
3695 forms are supported. In the first, `data_arg` is a variable or contiguous array section. In the second,  
3696 `data_arg` is a variable or array element and `bytes` is the length in bytes. The routine returns  
3697 `.true.` if the specified data is in shared memory or is fully present, and `.false.` otherwise. If  
3698 the byte length is zero, the routine returns nonzero in C/C++ or `.true.` in Fortran if the given  
3699 address is in shared memory or is present at all in the current device memory.

## 3700 3.2.37 `acc_memcpy_to_device`

### 3701 Summary

3702 The `acc_memcpy_to_device` routine copies data from local memory to device memory.

### 3703 Format

C or C++:

```

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

```

### 3705 Description

3706 The `acc_memcpy_to_device` routine copies `bytes` bytes of data from the local address in  
3707 `data_host_src` to the device address in `data_dev_dest`. `data_dev_dest` must be an  
3708 address accessible from the current device, such as an address returned from `acc_malloc` or  
3709 `acc_deviceptr`, or an address in shared memory.

3710 The `_async` version of this routine will perform the data transfers asynchronously on the `async`  
3711 queue associated with `async_arg`. The routine may return before the data has been transferred;  
3712 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return  
3713 until the data has been completely transferred.

## 3714 3.2.38 `acc_memcpy_from_device`

### 3715 Summary

3716 The `acc_memcpy_from_device` routine copies data from device memory to local memory.

### 3717 Format

C or C++:

```

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

```



3720 **Description**

3721 The `acc_memcpy_from_device` routine copies `bytes` bytes of data from the device address  
 3722 in `data_dev_src` to the local address in `data_host_dest`. `data_dev_src` must be an  
 3723 address accessible from the current device, such as an address returned from `acc_malloc` or  
 3724 `acc_deviceptr`, or an address in shared memory.

3725 The `_async` version of this routine will perform the data transfers asynchronously on the async  
 3726 queue associated with `async_arg`. The routine may return before the data has been transferred;  
 3727 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return  
 3728 until the data has been completely transferred.

3729 **3.2.39 acc\_memcpy\_device**3730 **Summary**

3731 The `acc_memcpy_device` routine copies data from one memory location to another memory  
 3732 location on the current device.

3733 **Format**

C or C++:

```

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

```

3736 **Description**

3737 The `acc_memcpy_device` routine copies `bytes` bytes of data from the device address in `data_dev_src`  
 3738 to the device address in `data_dev_dest`. Both addresses must be addresses in the current device  
 3739 memory, such as would be returned from `acc_malloc` or `acc_deviceptr`. If `data_dev_dest`  
 3740 and `data_dev_src` overlap, the behavior is undefined.

3741 The `_async` version of this routine will perform the data transfers asynchronously on the async  
 3742 queue associated with `async_arg`. The routine may return before the data has been transferred;  
 3743 see Section 2.16 Asynchronous Behavior for more details. The synchronous versions will not return  
 3744 until the data has been completely transferred.

3745 **3.2.40 acc\_attach**3746 **Summary**

3747 The `acc_attach` routine updates a pointer in device memory to point to the corresponding device  
 3748 copy of the host pointer target.

3749 **Format**

C or C++:

```

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

```

3753 **Description**

3754 **ptr\_addr** must be the address of a host pointer. If the data at **\*\*ptr\_addr** is in shared memory,  
 3755 or if the pointer **\*ptr\_addr** is in shared memory or is not present in the current device memory, or  
 3756 the address to which the **\*ptr\_addr** points is not present in the current device memory, no action  
 3757 is taken. Otherwise, these routines perform the *attach* action (Section 2.7.2).

3758 These routines may issue a data transfer from local memory to device memory. The **\_async**  
 3759 version of this routine will perform the data transfers asynchronously on the async queue associated  
 3760 with **async\_arg**. The routine may return before the data has been transferred; see Section 2.16  
 3761 Asynchronous Behavior for more details. The synchronous version will not return until the data has  
 3762 been completely transferred.

3763 **3.2.41 acc\_detach**3764 **Summary**

3765 The **acc\_detach** routine updates a pointer in device memory to point to the host pointer target.

3766 **Format**

C or C++:

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

3768 **Description**

3769 The **acc\_detach** routines are passed the address of a host pointer. If the data at **\*\*ptr\_addr**  
 3770 is in shared memory, or if the pointer **\*ptr\_addr** is in shared memory or is not present in the  
 3771 current device memory, or if the *attachment counter* for the pointer **\*ptr\_addr** is zero, no action  
 3772 is taken. Otherwise, these routines perform the *detach* action (Section 2.7.2).

3773 The **acc\_detach\_finalize** routines are equivalent to an **exit data** directive with **detach**  
 3774 and **finalize** clauses, as described in Section 2.7.13 detach clause. If the data is in shared mem-  
 3775 ory, or if the pointer **\*ptr\_addr** is not present in the current device memory, or if the *attachment*  
 3776 *counter* for the pointer **\*ptr\_addr** is zero, no action is taken. Otherwise, these routines perform  
 3777 the *immediate detach* action (Section 2.7.2).

3778 These routines may issue a data transfer from local memory to device memory. The **\_async** ver-  
 3779 sions of these routines will perform the data transfers asynchronously on the async queue associated  
 3780 with **async\_arg**. These routines may return before the data has been transferred; see Section 2.16  
 3781 Asynchronous Behavior for more details. The synchronous versions will not return until the data  
 3782 has been completely transferred.

3783 **3.2.42 acc\_memcpy\_d2d**3784 **Summary**

3785 This **acc\_memcpy\_d2d** and **acc\_memcpy\_d2d\_async** routines copy the contents of an array  
 3786 on one device to an array on the same or a different device without updating the value on the host.

3787 **Format**

C or C++:

```

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

```

3788

3789

Fortran:

```

subroutine acc_memcpy_d2d(data_arg_dest, data_arg_src, &
                        bytes, dev_num_dest, dev_num_src)
subroutine acc_memcpy_d2d_async(data_arg_dest, data_arg_src, &
                                bytes, dev_num_dest, dev_num_src, &
                                async_arg_src)

```

3790

```

type(*), dimension(..) :: data_arg_dest
type(*), dimension(..) :: data_arg_src
integer :: bytes
integer :: dev_num_dest
integer :: dev_num_src
integer :: async_arg_src

```

3791

3792

3793

3794

3795

3796

3797

3798 **Description**

3799 The `acc_memcpy_d2d` and `acc_memcpy_d2d_async` routines are passed the address of des-  
3800 tination and source host pointers as well as integer device numbers for the destination and source  
3801 devices, which must both be of the current device type. If both arrays are in shared memory, then  
3802 no action is taken. If either pointer is not in shared memory, then that array must be present on its  
3803 respective device. If these conditions are met, the contents of the source array on the source device  
3804 are copied to the destination array on the destination device.

3805 For `acc_memcpy_d2d_async` the value of `async_arg_src` is the number of an async queue  
3806 on the source device. This routine will perform the data transfers asynchronously on the async queue  
3807 associated with `async_arg_src` for device `dev_num_src`; see Section 2.16 Asynchronous Behavior  
3808 for more details.



## 4. Environment Variables

3809

3810 This chapter describes the environment variables that modify the behavior of accelerator regions.  
3811 The names of the environment variables must be upper case. The values assigned environment  
3812 variables are case-insensitive and may have leading and trailing white space. If the values of the  
3813 environment variables change after the program has started, even if the program itself modifies the  
3814 values, the behavior is implementation-defined.

### 4.1 ACC\_DEVICE\_TYPE

3815

3816 The **ACC\_DEVICE\_TYPE** environment variable controls the default device type to use when ex-  
3817 ecuting parallel, kernels, and serial regions, if the program has been compiled to use more than  
3818 one different type of device. The allowed values of this environment variable are implementation-  
3819 defined. See the release notes for currently-supported values of this environment variable.

3820 Example:

```
3821     setenv ACC_DEVICE_TYPE NVIDIA  
3822     export ACC_DEVICE_TYPE=NVIDIA
```

### 4.2 ACC\_DEVICE\_NUM

3823

3824 The **ACC\_DEVICE\_NUM** environment variable controls the default device number to use when  
3825 executing accelerator regions. The value of this environment variable must be a nonnegative integer  
3826 between zero and the number of devices of the desired type attached to the host. If the value is  
3827 greater than or equal to the number of devices attached, the behavior is implementation-defined.

3828 Example:

```
3829     setenv ACC_DEVICE_NUM 1  
3830     export ACC_DEVICE_NUM=1
```

### 4.3 ACC\_PROFLIB

3831

3832 The **ACC\_PROFLIB** environment variable specifies the profiling library. More details about the  
3833 evaluation at runtime is given in section 5.3.3 Runtime Dynamic Library Loading.

3834 Example:

```
3835     setenv ACC_PROFLIB /path/to/proflib/libaccprof.so  
3836     export ACC_PROFLIB=/path/to/proflib/libaccprof.so
```



## 5. Profiling Interface

This chapter describes the OpenACC interface for tools that can be used for profile and trace data collection. Therefore it provides a set of OpenACC-specific event callbacks that are triggered during the application run. Currently, this interface does not support tools that employ asynchronous sampling. In this chapter, the term *runtime* refers to the OpenACC runtime library. The term *library* refers to the third party routines invoked at specified events by the OpenACC runtime.

There are four steps for interfacing a *library* to the *runtime*. The first is to write the data collection 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 is to use registration routines to register the data collection callbacks for the appropriate events. The data collection and registration routines are then saved in a static or dynamic library or shared object. The third is to load the *library* at runtime. The *library* may be statically linked to the application or dynamically loaded by the application or by the *runtime*. This is described in Section 5.3 Loading the Library.

The fourth step is to invoke the registration routine 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 the 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.

Subsequently, the *library* may collect information when the callback routines are invoked by the *runtime* and process or store the acquired data.

### 5.1 Events

This section describes the events that are recognized by the runtime. Most events may 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_prof.h`, which is delivered with the OpenACC implementation. 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. The behavior of a tool receiving a callback after the runtime shutdown callback is undefined.

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

```
typedef enum acc_event_t{
    acc_ev_none = 0,
    acc_ev_device_init_start = 1,
    acc_ev_device_init_end = 2,
    acc_ev_device_shutdown_start = 3,
```

```
3877     acc_ev_device_shutdown_end = 4,  
3878     acc_ev_runtime_shutdown = 5,  
3879     acc_ev_create = 6,  
3880     acc_ev_delete = 7,  
3881     acc_ev_alloc = 8,  
3882     acc_ev_free = 9,  
3883     acc_ev_enter_data_start = 10,  
3884     acc_ev_enter_data_end = 11,  
3885     acc_ev_exit_data_start = 12,  
3886     acc_ev_exit_data_end = 13,  
3887     acc_ev_update_start = 14,  
3888     acc_ev_update_end = 15,  
3889     acc_ev_compute_construct_start = 16,  
3890     acc_ev_compute_construct_end = 17,  
3891     acc_ev_enqueue_launch_start = 18,  
3892     acc_ev_enqueue_launch_end = 19,  
3893     acc_ev_enqueue_upload_start = 20,  
3894     acc_ev_enqueue_upload_end = 21,  
3895     acc_ev_enqueue_download_start = 22,  
3896     acc_ev_enqueue_download_end = 23,  
3897     acc_ev_wait_start = 24,  
3898     acc_ev_wait_end = 25,  
3899     acc_ev_last = 26  
3900 }acc_event_t;
```

3901 The value of `acc_ev_last` will change if new events are added to the enumeration, so a library  
3902 should not depend on that value.

### 3903 5.1.1 Runtime Initialization and Shutdown

3904 No callbacks can be registered for the runtime initialization. Instead the initialization of the tool is  
3905 handled as described in Section 5.3 Loading the Library.

3906 The *runtime shutdown* event name is

```
3907     acc_ev_runtime_shutdown
```

3908 The `acc_ev_runtime_shutdown` event is triggered before the OpenACC runtime shuts down,  
3909 either because all devices have been shutdown by calls to the `acc_shutdown` API routine, or at  
3910 the end of the program.

### 3911 5.1.2 Device Initialization and Shutdown

3912 The *device initialization* event names are

```
3913     acc_ev_device_init_start  
3914     acc_ev_device_init_end
```

3915 These events are triggered when a device is being initialized by the OpenACC runtime. This may be  
3916 when the program starts, or may be later during execution when the program reaches an `acc_init`



3917 call or an OpenACC construct. The **acc\_ev\_device\_init\_start** is triggered before device  
3918 initialization starts and **acc\_ev\_device\_init\_end** after initialization is complete.

3919 The *device shutdown* event names are

```
3920     acc_ev_device_shutdown_start  
3921     acc_ev_device_shutdown_end
```

3922 These events are triggered when a device is shut down, most likely by a call to the OpenACC  
3923 **acc\_shutdown** API routine. The **acc\_ev\_device\_shutdown\_start** is triggered before  
3924 the device shutdown process starts and **acc\_ev\_device\_shutdown\_end** after the device shut-  
3925 down is complete.

### 3926 5.1.3 Enter Data and Exit Data

3927 The *enter data* and *exit data* event names are

```
3928     acc_ev_enter_data_start  
3929     acc_ev_enter_data_end  
3930     acc_ev_exit_data_start  
3931     acc_ev_exit_data_end
```

3932 The **acc\_ev\_enter\_data\_start** and **acc\_ev\_enter\_data\_end** events are triggered at  
3933 **enter data** directives, entry to data constructs, and entry to implicit data regions such as those  
3934 generated by compute constructs. The **acc\_ev\_enter\_data\_start** event is triggered before  
3935 any *data allocation*, *data update*, or *wait* events that are associated with that directive or region  
3936 entry, and the **acc\_ev\_enter\_data\_end** is triggered after those events.

3937 The **acc\_ev\_exit\_data\_start** and **acc\_ev\_exit\_data\_end** events are triggered at **exit**  
3938 **data** directives, exit from **data** constructs, and exit from implicit data regions. The  
3939 **acc\_ev\_exit\_data\_start** event is triggered before any *data deallocation*, *data update*, or  
3940 *wait* events associated with that directive or region exit, and the **acc\_ev\_exit\_data\_end** event  
3941 is triggered after those events.

3942 When the construct that triggers an *enter data* or *exit data* event was generated implicitly by the  
3943 compiler the **implicit** field in the event structure will be set to **1**. When the construct that  
3944 triggers these events was specified explicitly by the application code the **implicit** field in the  
3945 event structure will be set to **0**.

### 3946 5.1.4 Data Allocation

3947 The *data allocation* event names are

```
3948     acc_ev_create  
3949     acc_ev_delete  
3950     acc_ev_alloc  
3951     acc_ev_free
```

3952 An **acc\_ev\_alloc** event is triggered when the OpenACC runtime allocates memory from the de-  
3953 vice memory pool, and an **acc\_ev\_free** event is triggered when the runtime frees that memory.  
3954 An **acc\_ev\_create** event is triggered when the OpenACC runtime associates device memory  
3955 with local memory, such as for a data clause (**create**, **copyin**, **copy**, **copyout**) at entry to

3956 a data construct, compute construct, at an **enter data** directive, or in a call to a data API rou-  
 3957 tine (**acc\_copyin**, **acc\_create**, ...). An **acc\_ev\_create** event may be preceded by an  
 3958 **acc\_ev\_alloc** event, if newly allocated memory is used for this device data, or it may not, if  
 3959 the runtime manages its own memory pool. An **acc\_ev\_delete** event is triggered when the  
 3960 OpenACC runtime disassociates device memory from local memory, such as for a data clause at  
 3961 exit from a data construct, compute construct, at an **exit data** directive, or in a call to a data API  
 3962 routine (**acc\_copyout**, **acc\_delete**, ...). An **acc\_ev\_delete** event may be followed by  
 3963 an **acc\_ev\_free** event, if the disassociated device memory is freed, or it may not, if the runtime  
 3964 manages its own memory pool.

3965 When the action that generates a *data allocation* event was generated explicitly by the application  
 3966 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event  
 3967 is triggered because of a variable or array with implicitly-determined data attributes or otherwise  
 3968 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

### 3969 5.1.5 Data Construct

3970 The events for entering and leaving *data constructs* are mapped to *enter data* and *exit data* events  
 3971 as described in Section 5.1.3 Enter Data and Exit Data.

### 3972 5.1.6 Update Directive

3973 The *update directive* event names are

3974 **acc\_ev\_update\_start**  
 3975 **acc\_ev\_update\_end**

3976 The **acc\_ev\_update\_start** event will be triggered at an **update** directive, before any *data*  
 3977 *update* or *wait* events that are associated with the update directive are carried out, and the corre-  
 3978 sponding **acc\_ev\_update\_end** event will be triggered after any of the associated events.

### 3979 5.1.7 Compute Construct

3980 The *compute construct* event names are

3981 **acc\_ev\_compute\_construct\_start**  
 3982 **acc\_ev\_compute\_construct\_end**

3983 The **acc\_ev\_compute\_construct\_start** event is triggered at entry to a compute construct,  
 3984 before any *launch* events that are associated with entry to the compute construct. The  
 3985 **acc\_ev\_compute\_construct\_end** event is triggered at the exit of the compute construct,  
 3986 after any *launch* events associated with exit from the compute construct. If there are data clauses  
 3987 on the compute construct, those data clauses may be treated as part of the compute construct, or as  
 3988 part of a data construct containing the compute construct. The callbacks for data clauses must use  
 3989 the same line numbers as for the compute construct events.

### 3990 5.1.8 Enqueue Kernel Launch

3991 The *launch* event names are

3992 **acc\_ev\_enqueue\_launch\_start**  
 3993 **acc\_ev\_enqueue\_launch\_end**

3994 The **acc\_ev\_enqueue\_launch\_start** event is triggered just before an accelerator compu-  
3995 tation is enqueued for execution on a device, and **acc\_ev\_enqueue\_launch\_end** is trig-  
3996 gered just after the computation is enqueued. Note that these events are synchronous with the  
3997 local thread enqueueing the computation to a device, not with the device executing the compu-  
3998 tation. The **acc\_ev\_enqueue\_launch\_start** event callback routine is invoked just before  
3999 the computation is enqueued, not just before the computation starts execution. More importantly,  
4000 the **acc\_ev\_enqueue\_launch\_end** event callback routine is invoked after the computation is  
4001 enqueued, not after the computation finished executing.

4002 **Note:** Measuring the time between the start and end launch callbacks is often unlikely to be useful,  
4003 since it will only measure the time to manage the launch queue, not the time to execute the code on  
4004 the device.

### 4005 5.1.9 Enqueue Data Update (Upload and Download)

4006 The *data update* event names are

```
4007     acc_ev_enqueue_upload_start  
4008     acc_ev_enqueue_upload_end  
4009     acc_ev_enqueue_download_start  
4010     acc_ev_enqueue_download_end
```

4011 The **\_start** events are triggered just before each upload (data copy from local memory to device  
4012 memory) operation is or download (data copy from device memory to local memory) operation is  
4013 enqueued for execution on a device. The corresponding **\_end** events are triggered just after each  
4014 upload or download operation is enqueued.

4015 **Note:** Measuring the time between the start and end update callbacks is often unlikely to be useful,  
4016 since it will only measure the time to manage the enqueue operation, not the time to perform the  
4017 actual upload or download.

4018 When the action that generates a *data update* event was generated explicitly by the application  
4019 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event  
4020 is triggered because of a variable or array with implicitly-determined data attributes or otherwise  
4021 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

### 4022 5.1.10 Wait

4023 The *wait* event names are

```
4024     acc_ev_wait_start  
4025     acc_ev_wait_end
```

4026

4027 An **acc\_ev\_wait\_start** will be triggered for each relevant queue before the local thread waits  
4028 for that queue to be empty. A **acc\_ev\_wait\_end** will be triggered for each relevant queue after  
4029 the local thread has determined that the queue is empty.

4030 Wait events occur when the local thread and a device synchronize, either due to a **wait** directive  
4031 or by a *wait* clause on a synchronous data construct, compute construct, or **enter data**, **exit**  
4032 **data**, or **update** directive. For *wait* events triggered by an explicit synchronous **wait** directive

4033 or *wait* clause, the **implicit** field in the event structure will be **0**. For all other wait events, the  
 4034 **implicit** field in the event structure will be **1**.

4035 The OpenACC runtime need not trigger *wait* events for queues that have not been used in the  
 4036 program, and need not trigger *wait* events for queues that have not been used by this thread since  
 4037 the last *wait* operation. For instance, an **acc wait** directive with no arguments is defined to wait on  
 4038 all queues. If the program only uses the default (synchronous) queue and the queue associated with  
 4039 **async (1)** and **async (2)** then an **acc wait** directive may trigger *wait* events only for those  
 4040 three queues. If the implementation knows that no activities have been enqueued on the **async (2)**  
 4041 queue since the last *wait* operation, then the **acc wait** directive may trigger *wait* events only for  
 4042 the default queue and the **async (1)** queue.

## 4043 5.2 Callbacks Signature

4044 This section describes the signature of event callbacks. All event callbacks have the same signature.  
 4045 The routine prototypes are available in the header file **acc\_prof.h**, which is delivered with the  
 4046 OpenACC implementation.

4047 All callback routines have three arguments. The first argument is a pointer to a struct containing  
 4048 general information; the same struct type is used for all callback events. The second argument is  
 4049 a pointer to a struct containing information specific to that callback event; there is one struct type  
 4050 containing information for data events, another struct type containing information for kernel launch  
 4051 events, and a third struct type for other events, containing essentially no information. The third  
 4052 argument is a pointer to a struct containing information about the application programming interface  
 4053 (API) being used for the specific device. For NVIDIA CUDA devices, this contains CUDA-specific  
 4054 information; for OpenCL devices, this contains OpenCL-specific information. Other interfaces can  
 4055 be supported as they are added by implementations. The prototype for a callback routine is:

```
4056     typedef void (*acc_prof_callback)
4057         (acc_prof_info*, acc_event_info*, acc_api_info*);
```

4058 In the descriptions, the datatype **ssize\_t** means a signed 32-bit integer for a 32-bit binary and  
 4059 a 64-bit integer for a 64-bit binary, the datatype **size\_t** means an unsigned 32-bit integer for a  
 4060 32-bit binary and a 64-bit integer for a 64-bit binary, and the datatype **int** means a 32-bit integer  
 4061 for both 32-bit and 64-bit binaries. A null pointer is the pointer with value zero.

### 4062 5.2.1 First Argument: General Information

4063 The first argument is a pointer to the **acc\_prof\_info** struct type:

```
4064     typedef struct acc_prof_info{
4065         acc_event_t event_type;
4066         int valid_bytes;
4067         int version;
4068         acc_device_t device_type;
4069         int device_number;
4070         int thread_id;
4071         ssize_t async;
4072         ssize_t async_queue;
4073         const char* src_file;
4074         const char* func_name;
```

```

4075     int line_no, end_line_no;
4076     int func_line_no, func_end_line_no;
4077 }acc_prof_info;

```

4078 The fields are described below.

- 4079 • **acc\_event\_t event\_type** - The event type that triggered this callback. The datatype  
4080 is the enumeration type **acc\_event\_t**, described in the previous section. This allows the  
4081 same callback routine to be used for different events.
- 4082 • **int valid\_bytes** - The number of valid bytes in this struct. This allows a library to inter-  
4083 face with newer runtimes that may add new fields to the struct at the end while retaining com-  
4084 patibility with older runtimes. A runtime must fill in the **event\_type** and **valid\_bytes**  
4085 fields, and must fill in values for all fields with offset less than **valid\_bytes**. The value of  
4086 **valid\_bytes** for a struct is recursively defined as:  
4087 **valid\_bytes(struct) = offset(lastfield) + valid\_bytes(lastfield)**  
4088 **valid\_bytes(type[n]) = (n-1)\*sizeof(type) + valid\_bytes(type)**  
4089 **valid\_bytes(basictype) = sizeof(basictype)**
- 4090 • **int version** - A version number; the value of **\_OPENACC**.
- 4091 • **acc\_device\_t device\_type** - The device type corresponding to this event. The datatype  
4092 is **acc\_device\_t**, an enumeration type of all the supported device types, defined in **openacc.h**.
- 4093 • **int device\_number** - The device number. Each device is numbered, typically starting at  
4094 device zero. For applications that use more than one device type, the device numbers may be  
4095 unique across all devices or may be unique only across all devices of the same device type.
- 4096 • **int thread\_id** - The host thread ID making the callback. Host threads are given unique  
4097 thread ID numbers typically starting at zero. This is not necessarily the same as the OpenMP  
4098 thread number.
- 4099 • **ssize\_t async** - The value of the **async()** clause for the directive that triggered this  
4100 callback.
- 4101 • **ssize\_t async\_queue** - If the runtime uses a limited number of asynchronous queues,  
4102 this field contains the internal asynchronous queue number used for the event.
- 4103 • **const char\* src\_file** - A pointer to null-terminated string containing the name of or  
4104 path to the source file, if known, or a null pointer if not. If the library wants to save the source  
4105 file name, it should allocate memory and copy the string.
- 4106 • **const char\* func\_name** - A pointer to a null-terminated string containing the name of  
4107 the function in which the event occurred, if known, or a null pointer if not. If the library wants  
4108 to save the function name, it should allocate memory and copy the string.
- 4109 • **int line\_no** - The line number of the directive or program construct or the starting line  
4110 number of the OpenACC construct corresponding to the event. A negative or zero value  
4111 means the line number is not known.
- 4112 • **int end\_line\_no** - For an OpenACC construct, this contains the line number of the end  
4113 of the construct. A negative or zero value means the line number is not known.

- 4114 • **int func\_line\_no** - The line number of the first line of the function named in **func\_name**.
- 4115     A negative or zero value means the line number is not known.
- 4116 • **int func\_end\_line\_no** - The last line number of the function named in **func\_name**.
- 4117     A negative or zero value means the line number is not known.

## 4118 5.2.2 Second Argument: Event-Specific Information

4119 The second argument is a pointer to the **acc\_event\_info** union type.

```
4120 typedef union acc_event_info{
4121     acc_event_t event_type;
4122     acc_data_event_info data_event;
4123     acc_launch_event_info launch_event;
4124     acc_other_event_info other_event;
4125 }acc_event_info;
```

4126 The **event\_type** field selects which union member to use. The first five members of each union  
 4127 member are identical. The second through fifth members of each union member (**valid\_bytes**,  
 4128 **parent\_construct**, **implicit**, and **tool\_info**) have the same semantics for all event  
 4129 types:

- 4130 • **int valid\_bytes** - The number of valid bytes in the respective struct. (This field is similar  
 4131 used as discussed in Section 5.2.1 First Argument: General Information.)
- 4132 • **acc\_construct\_t parent\_construct** - This field describes the type of construct  
 4133 that caused the event to be emitted. The possible values for this field are defined by the  
 4134 **acc\_construct\_t** enum, described at the end of this section.
- 4135 • **int implicit** - This field is set to 1 for any implicit event, such as an implicit wait at  
 4136 a synchronous data construct or synchronous enter data, exit data or update directive. This  
 4137 field is set to zero when the event is triggered by an explicit directive or call to a runtime API  
 4138 routine.
- 4139 • **void\* tool\_info** - This field is used to pass tool-specific information from a **\_start**  
 4140 event to the matching **\_end** event. For a **\_start** event callback, this field will be initialized  
 4141 to a null pointer. The value of this field for a **\_end** event will be the value returned by  
 4142 the library in this field from the matching **\_start** event callback, if there was one, or null  
 4143 otherwise. For events that are neither **\_start** or **\_end** events, this field will be null.

## 4144 Data Events

4145 For a data event, as noted in the event descriptions, the second argument will be a pointer to the  
 4146 **acc\_data\_event\_info** struct.

```
4147 typedef struct acc_data_event_info{
4148     acc_event_t event_type;
4149     int valid_bytes;
4150     acc_construct_t parent_construct;
4151     int implicit;
4152     void* tool_info;
4153     const char* var_name;
```

```

4154     size_t bytes;
4155     const void* host_ptr;
4156     const void* device_ptr;
4157 }acc_data_event_info;

```

4158 The fields specific for a data event are:

- 4159 • **acc\_event\_t event\_type** - The event type that triggered this callback. The events that  
4160 use the **acc\_data\_event\_info** struct are:

```

4161     acc_ev_enqueue_upload_start
4162     acc_ev_enqueue_upload_end
4163     acc_ev_enqueue_download_start
4164     acc_ev_enqueue_download_end
4165     acc_ev_create
4166     acc_ev_delete
4167     acc_ev_alloc
4168     acc_ev_free

```

- 4169 • **const char\* var\_name** - A pointer to null-terminated string containing the name of the  
4170 variable for which this event is triggered, if known, or a null pointer if not. If the library wants  
4171 to save the variable name, it should allocate memory and copy the string.
- 4172 • **size\_t bytes** - The number of bytes for the data event.
- 4173 • **const void\* host\_ptr** - If available and appropriate for this event, this is a pointer to  
4174 the host data.
- 4175 • **const void\* device\_ptr** - If available and appropriate for this event, this is a pointer  
4176 to the corresponding device data.

## 4177 Launch Events

4178 For a launch event, as noted in the event descriptions, the second argument will be a pointer to the  
4179 **acc\_launch\_event\_info** struct.

```

4180     typedef struct acc_launch_event_info{
4181         acc_event_t event_type;
4182         int valid_bytes;
4183         acc_construct_t parent_construct;
4184         int implicit;
4185         void* tool_info;
4186         const char* kernel_name;
4187         size_t num_gangs, num_workers, vector_length;
4188     }acc_launch_event_info;

```

4189 The fields specific for a launch event are:

- 4190 • **acc\_event\_t event\_type** - The event type that triggered this callback. The events that  
4191 use the **acc\_launch\_event\_info** struct are:

```

4192     acc_ev_enqueue_launch_start
4193     acc_ev_enqueue_launch_end

```

- 4194 • **const char\* kernel\_name** - A pointer to null-terminated string containing the name of  
4195 the kernel being launched, if known, or a null pointer if not. If the library wants to save the  
4196 kernel name, it should allocate memory and copy the string.
- 4197 • **size\_t num\_gangs, num\_workers, vector\_length** - The number of gangs, work-  
4198 ers and vector lanes created for this kernel launch.

## 4199 Other Events

4200 For any event that does not use the **acc\_data\_event\_info** or **acc\_launch\_event\_info**  
4201 struct, the second argument to the callback routine will be a pointer to **acc\_other\_event\_info**  
4202 struct.

```
4203     typedef struct acc_other_event_info{
4204         acc_event_t event_type;
4205         int valid_bytes;
4206         acc_construct_t parent_construct;
4207         int implicit;
4208         void* tool_info;
4209     }acc_other_event_info;
```

## 4210 Parent Construct Enumeration

4211 All event structures contain a **parent\_construct** member that describes the type of construct  
4212 that caused the event to be emitted. The purpose of this field is to provide a means to identify  
4213 the type of construct emitting the event in the cases where an event may be emitted by multi-  
4214 ple construct types, such as is the case with data and wait events. The possible values for the  
4215 **parent\_construct** field are defined in the enumeration type **acc\_construct\_t**. In the  
4216 case of combined directives, the outermost construct of the combined construct should be specified  
4217 as the **parent\_construct**. If the event was emitted as the result of the application making a  
4218 call to the runtime api, the value will be **acc\_construct\_runtime\_api**.

```
4219     typedef enum acc_construct_t{
4220         acc_construct_parallel = 0,
4221         acc_construct_kernels = 1,
4222         acc_construct_loop = 2,
4223         acc_construct_data = 3,
4224         acc_construct_enter_data = 4,
4225         acc_construct_exit_data = 5,
4226         acc_construct_host_data = 6,
4227         acc_construct_atomic = 7,
4228         acc_construct_declare = 8,
4229         acc_construct_init = 9,
4230         acc_construct_shutdown = 10,
4231         acc_construct_set = 11,
4232         acc_construct_update = 12,
4233         acc_construct_routine = 13,
4234         acc_construct_wait = 14,
4235         acc_construct_runtime_api = 15,
```



```

4236     acc_construct_serial = 16
4237 }acc_construct_t;

```

### 4238 5.2.3 Third Argument: API-Specific Information

4239 The third argument is a pointer to the `acc_api_info` struct type, shown here.

```

4240     typedef struct acc_api_info{
4241         acc_device_api device_api;
4242         int valid_bytes;
4243         acc_device_t device_type;
4244         int vendor;
4245         const void* device_handle;
4246         const void* context_handle;
4247         const void* async_handle;
4248     }acc_api_info;

```

4249 The fields are described below:

- 4250 • **acc\_device\_api device\_api** - The API in use for this device. The data type is the  
4251 enumeration `acc_device_api`, which is described later in this section.
- 4252 • **int valid\_bytes** - The number of valid bytes in this struct. See the discussion above in  
4253 Section 5.2.1 First Argument: General Information.
- 4254 • **acc\_device\_t device\_type** - The device type; the datatype is `acc_device_t`, de-  
4255 fined in `openacc.h`.
- 4256 • **int vendor** - An identifier to identify the OpenACC vendor; contact your vendor to deter-  
4257 mine the value used by that vendor's runtime.
- 4258 • **const void\* device\_handle** - If applicable, this will be a pointer to the API-specific  
4259 device information.
- 4260 • **const void\* context\_handle** - If applicable, this will be a pointer to the API-specific  
4261 context information.
- 4262 • **const void\* async\_handle** - If applicable, this will be a pointer to the API-specific  
4263 async queue information.

4264 According to the value of `device_api` a library can cast the pointers of the fields `device_handle`,  
4265 `context_handle` and `async_handle` to the respective device API type. The following device  
4266 APIs are defined in the interface below. Any implementation-defined device API type must have a  
4267 value greater than `acc_device_api_implementation_defined`.

```

4268     typedef enum acc_device_api{
4269         acc_device_api_none = 0,                /* no device API */
4270         acc_device_api_cuda = 1,              /* CUDA driver API */
4271         acc_device_api_opencl = 2,            /* OpenCL API */
4272         acc_device_api_other = 4,             /* other device API */
4273         acc_device_api_implementation_defined = 1000 /* other device API */
4274     }acc_device_api;

```

## 5.3 Loading the Library

This section describes how a tools library is loaded when the program is run. Four methods are described.

- A tools library may be linked with the program, as any other library is linked, either as a static library or a dynamic library, and the runtime will call a predefined library initialization routine that will register the event callbacks.
- The OpenACC runtime implementation may support a dynamic tools library, such as a shared object for Linux or OS/X, or a DLL for Windows, which is then dynamically loaded at runtime under control of the environment variable **ACC\_PROFLIB**.
- Some implementations where the OpenACC runtime is itself implemented as a dynamic library may support adding a tools library using the **LD\_PRELOAD** feature in Linux.
- A tools library may be linked with the program, as in the first option, and the application itself may directly register event callback routines, or may invoke a library initialization routine that will register the event callbacks.

Callbacks are registered with the runtime by calling **acc\_prof\_register** for each event as described in Section 5.4 Registering Event Callbacks. The prototype for **acc\_prof\_register** is:

```
extern void acc_prof_register
           (acc_event_t event_type, acc_prof_callback cb,
           acc_register_t info);
```

The first argument to **acc\_prof\_register** is the event for which a callback is being registered (compare Section 5.1 Events). The second argument is a pointer to the callback routine:

```
typedef void (*acc_prof_callback)
           (acc_prof_info*, acc_event_info*, acc_api_info*);
```

The third argument is usually zero (or **acc\_reg**). See Section 5.4.2 Disabling and Enabling Callbacks for cases where a nonzero value is used. The argument **acc\_register\_t** is an enum type:

```
typedef enum acc_register_t{
    acc_reg = 0,
    acc_toggle = 1,
    acc_toggle_per_thread = 2
}acc_register_t;
```

An example of registering callbacks for launch, upload, and download events is:

```
acc_prof_register(acc_ev_enqueue_launch_start, prof_launch, 0);
acc_prof_register(acc_ev_enqueue_upload_start, prof_data, 0);
acc_prof_register(acc_ev_enqueue_download_start, prof_data, 0);
```

As shown in this example, the same routine (**prof\_data**) can be registered for multiple events. The routine can use the **event\_type** field in the **acc\_prof\_info** structure to determine for what event it was invoked.

### 4308 5.3.1 Library Registration

4309 The OpenACC runtime will invoke **acc\_register\_library**, passing the addresses of the reg-  
 4310 istration routines **acc\_prof\_register** and **acc\_prof\_unregister**, in case that routine  
 4311 comes from a dynamic library. In the third argument it passes the address of the lookup routine  
 4312 **acc\_prof\_lookup** to obtain the addresses of inquiry functions. No inquiry functions are de-  
 4313 fined in this profiling interface, but we preserve this argument for future support of sampling-based  
 4314 tools.

4315 Typically, the OpenACC runtime will include a *weak* definition of **acc\_register\_library**,  
 4316 which does nothing and which will be called when there is no tools library. In this case, the library  
 4317 can save the addresses of these routines and/or make registration calls to register any appropriate  
 4318 callbacks. The prototype for **acc\_register\_library** is:

```
4319     extern void acc_register_library
4320             (acc_prof_reg reg, acc_prof_reg unreg,
4321              acc_prof_lookup_func lookup);
```

4322 The first two arguments of this routine are of type:

```
4323     typedef void (*acc_prof_reg)
4324             (acc_event_t event_type, acc_prof_callback cb,
4325              acc_register_t info);
```

4326 The third argument passes the address to the lookup function **acc\_prof\_lookup** to obtain the  
 4327 address of interface functions. It is of type:

```
4328     typedef void (*acc_query_fn) ();
4329     typedef acc_query_fn (*acc_prof_lookup_func)
4330             (const char* acc_query_fn_name);
```

4331 The argument of the lookup function is a string with the name of the inquiry function. There are no  
 4332 inquiry functions defined for this interface.

### 4333 5.3.2 Statically-Linked Library Initialization

4334 A tools library can be compiled and linked directly into the application. If the library provides an  
 4335 external routine **acc\_register\_library** as specified in Section 5.3.1 Library Registration, the  
 4336 runtime will invoke that routine to initialize the library.

4337 The sequence of events is:

- 4338 1. The runtime invokes the **acc\_register\_library** routine from the library.
- 4339 2. The **acc\_register\_library** routine calls **acc\_prof\_register** for each event to  
 4340 be monitored.
- 4341 3. **acc\_prof\_register** records the callback routines.
- 4342 4. The program runs, and your callback routines are invoked at the appropriate events.

4343 In this mode, only one tool library is supported.

### 5.3.3 Runtime Dynamic Library Loading

A common case is to build the tools library as a dynamic library (shared object for Linux or OS/X, DLL for Windows). In that case, you can have the OpenACC runtime load the library during initialization. This allows you to enable runtime profiling without rebuilding or even relinking your application. The dynamic library must implement a registration routine `acc_register_library` as specified in Section 5.3.1 Library Registration.

The user may set the environment variable `ACC_PROFLIB` to the path to the library will tell the OpenACC runtime to load your dynamic library at initialization time:

Bash:

```
export ACC_PROFLIB=/home/user/lib/myprof.so
./myapp
```

or

```
ACC_PROFLIB=/home/user/lib/myprof.so ./myapp
```

C-shell:

```
setenv ACC_PROFLIB /home/user/lib/myprof.so
./myapp
```

When the OpenACC runtime initializes, it will read the `ACC_PROFLIB` environment variable (with `getenv`). The runtime will open the dynamic library (using `dlopen` or `LoadLibraryA`); if the library cannot be opened, the runtime may abort, 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_prof_register`, the deregistration routine `acc_prof_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:

1. Initialization of the OpenACC runtime.
2. OpenACC runtime reads `ACC_PROFLIB`.
3. OpenACC runtime loads the library.
4. OpenACC runtime calls the `acc_register_library` routine in that library.
5. Your `acc_register_library` routine calls `acc_prof_register` for each event to be monitored.
6. `acc_prof_register` records the callback routines.
7. The program runs, and your callback routines are invoked at the appropriate events.

If supported, paths to multiple dynamic libraries may be specified in the `ACC_PROFLIB` environment variable, separated by semicolons (;). The OpenACC runtime will open these libraries and invoke the `acc_register_library` routine for each, in the order they appear in `ACC_PROFLIB`.

### 5.3.4 Preloading with LD\_PRELOAD

The implementation may also support dynamic loading of a tools library using the **LD\_PRELOAD** feature available in some systems. In such an implementation, you need only specify your tools library path in the **LD\_PRELOAD** environment variable before executing your program. The OpenACC runtime will invoke the **acc\_register\_library** routine in your tools library at initialization time. This requires that the OpenACC runtime include a dynamic library with a default (empty) implementation of **acc\_register\_library** that will be invoked in the normal case where there is no **LD\_PRELOAD** setting. If an implementation only supports static linking, or if the application is linked without dynamic library support, this feature will not be available.

Bash:

```
export LD_PRELOAD=/home/user/lib/myprof.so
./myapp
```

or

```
LD_PRELOAD=/home/user/lib/myprof.so ./myapp
```

C-shell:

```
setenv LD_PRELOAD /home/user/lib/myprof.so
./myapp
```

The sequence of events is:

1. The operating system loader loads the library specified in **LD\_PRELOAD**.
2. The call to **acc\_register\_library** in the OpenACC runtime is resolved to the routine in the loaded tools library.
3. OpenACC runtime calls the **acc\_register\_library** routine in that library.
4. Your **acc\_register\_library** routine calls **acc\_prof\_register** for each event to be monitored.
5. **acc\_prof\_register** records the callback routines.
6. The program runs, and your callback routines are invoked at the appropriate events.

In this mode, only a single tools library is supported, since only one **acc\_register\_library** initialization routine will get resolved by the dynamic loader.

### 5.3.5 Application-Controlled Initialization

An alternative to default initialization is to have the application itself call the library initialization routine, which then calls **acc\_prof\_register** for each appropriate event. The library may be statically linked to the application or your application may dynamically load the library.

The sequence of events is:

1. Your application calls the library initialization routine.
2. The library initialization routine calls **acc\_prof\_register** for each event to be monitored.
3. **acc\_prof\_register** records the callback routines.
4. The program runs, and your callback routines are invoked at the appropriate events.

4420 In this mode, multiple tools libraries can be supported, with each library initialization routine in-  
 4421 voked by the application.

## 4422 5.4 Registering Event Callbacks

4423 This section describes how to register and unregister callbacks, temporarily disabling and enabling  
 4424 callbacks, the behavior of dynamic registration and unregistration, and requirements on an Open-  
 4425 ACC implementation to correctly support the interface.

### 4426 5.4.1 Event Registration and Unregistration

4427 The library must call the registration routine `acc_prof_register` to register each callback  
 4428 with the runtime. A simple example:

```
4429     extern void prof_data(acc_prof_info* profinfo,
4430                          acc_event_info* eventinfo, acc_api_info* apiinfo);
4431     extern void prof_launch(acc_prof_info* profinfo,
4432                            acc_event_info* eventinfo, acc_api_info* apiinfo);
4433     ...
4434     void acc_register_library(acc_prof_reg reg,
4435                             acc_prof_reg unreg, acc_prof_lookup_func lookup){
4436         reg(acc_ev_enqueue_upload_start, prof_data, 0);
4437         reg(acc_ev_enqueue_download_start, prof_data, 0);
4438         reg(acc_ev_enqueue_launch_start, prof_launch, 0);
4439     }
```

4440 In this example the `prof_data` routine will be invoked for each data upload and download event,  
 4441 and the `prof_launch` routine will be invoked for each launch event. The `prof_data` routine  
 4442 might start out with:

```
4443     void prof_data(acc_prof_info* profinfo,
4444                  acc_event_info* eventinfo, acc_api_info* apiinfo){
4445         acc_data_event_info* datainfo;
4446         datainfo = (acc_data_event_info*)eventinfo;
4447         switch( datainfo->event_type ){
4448             case acc_ev_enqueue_upload_start :
4449                 ...
4450         }
4451     }
```

### 4452 Multiple Callbacks

4453 Multiple callback routines can be registered on the same event:

```
4454     acc_prof_register(acc_ev_enqueue_upload_start, prof_data, 0);
4455     acc_prof_register(acc_ev_enqueue_upload_start, prof_up, 0);
```

4456 For most events, the callbacks will be invoked in the order in which they are registered. However,  
 4457 *end* events, named `acc_ev_..._end`, invoke callbacks in the reverse order. Essentially, each  
 4458 event has an ordered list of callback routines. A new callback routine is appended to the tail of the  
 4459 list for that event. For most events, that list is traversed from the head to the tail, but for *end* events,  
 4460 the list is traversed from the tail to the head.

4461 If a callback is registered, then later unregistered, then later still registered again, the second regis-  
 4462 tration is considered to be a new callback, and the callback routine will then be appended to the tail  
 4463 of the callback list for that event.

## 4464 Unregistering

4465 A matching call to `acc_prof_unregister` will remove that routine from the list of callback  
 4466 routines for that event.

```
4467     acc_prof_register(acc_ev_enqueue_upload_start, prof_data, 0);
4468     // prof_data is on the callback list for acc_ev_enqueue_upload_start
4469     ...
4470     acc_prof_unregister(acc_ev_enqueue_upload_start, prof_data, 0);
4471     // prof_data is removed from the callback list
4472     // for acc_ev_enqueue_upload_start
```

4473 Each entry on the callback list must also have a *ref* count. This keeps track of how many times  
 4474 this routine was added to this event's callback list. If a routine is registered *n* times, it must be  
 4475 unregistered *n* times before it is removed from the list. Note that if a routine is registered multiple  
 4476 times for the same event, its *ref* count will be incremented with each registration, but it will only be  
 4477 invoked once for each event instance.

## 4478 5.4.2 Disabling and Enabling Callbacks

4479 A callback routine may be temporarily disabled on the callback list for an event, then later re-  
 4480 enabled. The behavior is slightly different than unregistering and later re-registering that event.  
 4481 When a routine is disabled and later re-enabled, the routine's position on the callback list for that  
 4482 event is preserved. When a routine is unregistered and later re-registered, the routine's position on  
 4483 the callback list for that event will move to the tail of the list. Also, unregistering a callback must be  
 4484 done *n* times if the callback routine was registered *n* times. In contrast, disabling, and enabling an  
 4485 event sets a toggle. Disabling a callback will immediately reset the toggle and disable calls to that  
 4486 routine for that event, even if it was enabled multiple times. Enabling a callback will immediately  
 4487 set the toggle and enable calls to that routine for that event, even if it was disabled multiple times.  
 4488 Registering a new callback initially sets the toggle.

4489 A call to `acc_prof_unregister` with a value of `acc_toggle` as the third argument will dis-  
 4490 able callbacks to the given routine. A call to `acc_prof_register` with a value of `acc_toggle`  
 4491 as the third argument will enable those callbacks.

```
4492     acc_prof_unregister(acc_ev_enqueue_upload_start,
4493                       prof_data, acc_toggle);
4494     // prof_data is disabled
4495     ...
4496     acc_prof_register(acc_ev_enqueue_upload_start,
4497                      prof_data, acc_toggle);
4498     // prof_data is re-enabled
```

4499 A call to either `acc_prof_unregister` or `acc_prof_register` to disable or enable a call-  
 4500 back when that callback is not currently registered for that event will be ignored with no error.

4501 All callbacks for an event may be disabled (and re-enabled) by passing `NULL` to the second argument  
 4502 and `acc_toggle` to the third argument of `acc_prof_unregister` (and `acc_prof_register`).

4503 This sets a toggle for that event, which is distinct from the toggle for each callback for that event.  
 4504 While the event is disabled, no callbacks for that event will be invoked. Callbacks for that event can  
 4505 be registered, unregistered, enabled, and disabled while that event is disabled, but no callbacks will  
 4506 be invoked for that event until the event itself is enabled. Initially, all events are enabled.

```

4507     acc_prof_unregister(acc_ev_enqueue_upload_start,
4508                       prof_data, acc_toggle);
4509     // prof_data is disabled
4510     ...
4511     acc_prof_unregister(acc_ev_enqueue_upload_start,
4512                       NULL, acc_toggle);
4513     // acc_ev_enqueue_upload_start callbacks are disabled
4514     ...
4515     acc_prof_register(acc_ev_enqueue_upload_start,
4516                      prof_data, acc_toggle);
4517     // prof_data is re-enabled, but
4518     // acc_ev_enqueue_upload_start callbacks still disabled
4519     ...
4520     acc_prof_register(acc_ev_enqueue_upload_start, prof_up, 0);
4521     // prof_up is registered and initially enabled, but
4522     // acc_ev_enqueue_upload_start callbacks still disabled
4523     ...
4524     acc_prof_register(acc_ev_enqueue_upload_start,
4525                      NULL, acc_toggle);
4526     // acc_ev_enqueue_upload_start callbacks are enabled
4527
  
```

4528 Finally, all callbacks can be disabled (and enabled) by passing the argument list (**0**, **NULL**,  
 4529 **acc\_toggle**) to **acc\_prof\_unregister** (and **acc\_prof\_register**). This sets a global  
 4530 toggle disabling all callbacks, which is distinct from the toggle enabling callbacks for each event and  
 4531 the toggle enabling each callback routine. The behavior of passing zero as the first argument and a  
 4532 non-**NULL** value as the second argument to **acc\_prof\_unregister** or **acc\_prof\_register**  
 4533 is not defined, and may be ignored by the runtime without error.

4534 All callbacks can be disabled (or enabled) for just the current thread by passing the argument list  
 4535 (**0**, **NULL**, **acc\_toggle\_per\_thread**) to **acc\_prof\_unregister** (and **acc\_prof\_register**).  
 4536 This is the only thread-specific interface to **acc\_prof\_register** and **acc\_prof\_unregister**,  
 4537 all other calls to register, unregister, enable, or disable callbacks affect all threads in the application.

## 4538 5.5 Advanced Topics

4539 This section describes advanced topics such as dynamic registration and changes of the execution  
 4540 state for callback routines as well as the runtime and tool behavior for multiple host threads.

### 4541 5.5.1 Dynamic Behavior

4542 Callback routines may be registered or unregistered, enabled or disabled at any point in the execution  
 4543 of the program. Calls may appear in the library itself, during the processing of an event. The  
 4544 OpenACC runtime must allow for this case, where the callback list for an event is modified while  
 4545 that event is being processed.



## 4546 **Dynamic Registration and Unregistration**

4547 Calls to **acc\_register** and **acc\_unregister** may occur at any point in the application. A  
4548 callback routine can be registered or unregistered from a callback routine, either the same routine  
4549 or another routine, for a different event or the same event for which the callback was invoked. If a  
4550 callback routine is registered for an event while that event is being processed, then the new callback  
4551 routine will be added to the tail of the list of callback routines for this event. Some events (the  
4552 **\_end**) events process the callback routines in reverse order, from the tail to the head. For those  
4553 events, adding a new callback routine will not cause the new routine to be invoked for this instance  
4554 of the event. The other events process the callback routines in registration order, from the head to  
4555 the tail. Adding a new callback routine for such a event will cause the runtime to invoke that newly  
4556 registered callback routine for this instance of the event. Both the runtime and the library must  
4557 implement and expect this behavior.

4558 If an existing callback routine is unregistered for an event while that event is being processed, that  
4559 callback routine is removed from the list of callbacks for this event. For any event, if that callback  
4560 routine had not yet been invoked for this instance of the event, it will not be invoked.

4561 Registering and unregistering a callback routine is a global operation and affects all threads, in a  
4562 multithreaded application. See Section 5.4.1 Multiple Callbacks.

## 4563 **Dynamic Enabling and Disabling**

4564 Calls to **acc\_register** and **acc\_unregister** to enable and disable a specific callback for  
4565 an event, enable or disable all callbacks for an event, or enable or disable all callbacks may occur  
4566 at any point in the application. A callback routine can be enabled or disabled from a callback  
4567 routine, either the same routine or another routine, for a different event or the same event for which  
4568 the callback was invoked. If a callback routine is enabled for an event while that event is being  
4569 processed, then the new callback routine will be immediately enabled. If it appears on the list of  
4570 callback routines closer to the head (for **\_end** events) or closer to the tail (for other events), that  
4571 newly-enabled callback routine will be invoked for this instance of this event, unless it is disabled  
4572 or unregistered before that callback is reached.

4573 If a callback routine is disabled for an event while that event is being processed, that callback routine  
4574 is immediately disabled. For any event, if that callback routine had not yet been invoked for this in-  
4575 stance of the event, it will not be invoked, unless it is enabled before that callback routine is reached  
4576 in the list of callbacks for this event. If all callbacks for an event are disabled while that event is  
4577 being processed, or all callbacks are disabled for all events while an event is being processed, then  
4578 when this callback routine returns, no more callbacks will be invoked for this instance of the event.

4579 Registering and unregistering a callback routine is a global operation and affects all threads, in a  
4580 multithreaded application. See Section 5.4.1 Multiple Callbacks.

## 4581 **5.5.2 OpenACC Events During Event Processing**

4582 OpenACC events may occur during event processing. This may be because of OpenACC API rou-  
4583 tine calls or OpenACC constructs being reached during event processing, or because of multiple host  
4584 threads executing asynchronously. Both the OpenACC runtime and the tool library must implement  
4585 the proper behavior.

### 5.5.3 Multiple Host Threads

Many programs that use OpenACC also use multiple host threads, such as programs using the OpenMP API. The appearance of multiple host threads affects both the OpenACC runtime and the tools library.

#### Runtime Support for Multiple Threads

The OpenACC runtime must be thread-safe, and the OpenACC runtime implementation of this tools interface must also be thread-safe. All threads use the same set of callbacks for all events, so registering a callback from one thread will cause all threads to execute that callback. This means that managing the callback lists for each event must be protected from multiple simultaneous updates. This includes adding a callback to the tail of the callback list for an event, removing a callback from the list for an event, and incrementing or decrementing the *ref* count for a callback routine for an event.

In addition, one thread may register, unregister, enable, or disable a callback for an event while another thread is processing the callback list for that event asynchronously. The exact behavior may be dependent on the implementation, but some behaviors are expected and others are disallowed. In the following examples, there are three callbacks, A, B, and C, registered for event E in that order, where callbacks A and B are enabled and callback C is temporarily disabled. Thread T1 is dynamically modifying the callbacks for event E while thread T2 is processing an instance of event E.

- Suppose thread T1 unregisters or disables callback A for event E. Thread T2 may or may not invoke callback A for this event instance, but it must invoke callback B; if it invokes callback A, that must precede the invocation of callback B.
- Suppose thread T1 unregisters or disables callback B for event E. Thread T2 may or may not invoke callback B for this event instance, but it must invoke callback A; if it invokes callback B, that must follow the invocation of callback A.
- Suppose thread T1 unregisters or disables callback A and then unregisters or disables callback B for event E. Thread T2 may or may not invoke callback A and may or may not invoke callback B for this event instance, but if it invokes both callbacks, it must invoke callback A before it invokes callback B.
- Suppose thread T1 unregisters or disables callback B and then unregisters or disables callback A for event E. Thread T2 may or may not invoke callback A and may or may not invoke callback B for this event instance, but if it invokes callback B, it must have invoked callback A for this event instance.
- Suppose thread T1 is registering a new callback D for event E. Thread T2 may or may not invoke callback D for this event instance, but it must invoke both callbacks A and B. If it invokes callback D, that must follow the invocations of A and B.
- Suppose thread T1 is enabling callback C for event E. Thread T2 may or may not invoke callback C for this event instance, but it must invoke both callbacks A and B. If it invokes callback C, that must follow the invocations of A and B.

The `acc_prof_info` struct has a `thread_id` field, which the runtime must set to a unique value for each host thread, though it need not be the same as the OpenMP `threadnum` value.

## 4627 **Library Support for Multiple Threads**

4628 The tool library must also be thread-safe. The callback routine will be invoked in the context of the  
4629 thread that reaches the event. The library may receive a callback from a thread T2 while it's still  
4630 processing a callback, from the same event type or from a different event type, from another thread  
4631 T1. The **acc\_prof\_info** struct has a **thread\_id** field, which the runtime must set to a unique  
4632 value for each host thread.

4633 If the tool library uses dynamic callback registration and unregistration, or callback disabling and  
4634 enabling, recall that unregistering or disabling an event callback from one thread will unregister or  
4635 disable that callback for all threads, and registering or enabling an event callback from any thread  
4636 will register or enable it for all threads. If two or more threads register the same callback for the  
4637 same event, the behavior is the same as if one thread registered that callback multiple times; see  
4638 Section 5.4.1 Multiple Callbacks. The **acc\_unregister** routine must be called as many times  
4639 as **acc\_register** for that callback/event pair in order to totally unregister it. If two threads  
4640 register two different callback routines for the same event, unless the order of the registration calls  
4641 is guaranteed by some synchronization method, the order in which the runtime sees the registration  
4642 may differ for multiple runs, meaning the order in which the callbacks occur will differ as well.



## 6. Glossary

4643

4644 Clear and consistent terminology is important in describing any programming model. We define  
4645 here the terms you must understand in order to make effective use of this document and the asso-  
4646 ciated programming model. In particular, some terms used in this specification conflict with their  
4647 usage in the base language specifications. When there is potential confusion, the term will appear  
4648 here.

4649 **Accelerator** – a device attached to a CPU and to which the CPU can offload data and compute  
4650 kernels to perform compute-intensive calculations.

4651 **Accelerator routine** – a C or C++ function or Fortran subprogram compiled for the accelerator  
4652 with the **routine** directive.

4653 **Accelerator thread** – a thread of execution that executes on the accelerator; a single vector lane of  
4654 a single worker of a single gang.

4655 **Aggregate datatype** – any non-scalar datatype such as array and composite datatypes. In Fortran,  
4656 aggregate datatypes include arrays, derived types, character types. In C, aggregate datatypes include  
4657 arrays, targets of pointers, structs, and unions. In C++, aggregate datatypes include arrays, targets  
4658 of pointers, classes, structs, and unions.

4659 **Aggregate variables** – a variable of any non-scalar datatype, including array or composite variables.  
4660 In Fortran, this includes any variable with allocatable or pointer attribute and character variables.

4661 **Async-argument** – an *async-argument* is a nonnegative scalar integer expression (*int* for C or C++,  
4662 *integer* for Fortran), or one of the special values **acc\_async\_noval** or **acc\_async\_sync**.

4663 **Barrier** – a type of synchronization where all parallel execution units or threads must reach the  
4664 barrier before any execution unit or thread is allowed to proceed beyond the barrier; modeled after  
4665 the starting barrier on a horse race track.

4666 **Block construct** – a *block-construct*, as specified by the Fortran language.

4667 **Composite datatype** – a derived type in Fortran, or a **struct** or **union** type in C, or a **class**,  
4668 **struct**, or **union** type in C++. (This is different from the use of the term *composite data type* in  
4669 the C and C++ languages.)

4670 **Composite variable** – a variable of composite datatype. In Fortran, a composite variable must not  
4671 have allocatable or pointer attributes.

4672 **Compute construct** – a *parallel construct*, *kernels construct*, or *serial construct*.

4673 **Compute intensity** – for a given loop, region, or program unit, the ratio of the number of arithmetic  
4674 operations performed on computed data divided by the number of memory transfers required to  
4675 move that data between two levels of a memory hierarchy.

4676 **Compute region** – a *parallel region*, *kernels region*, or *serial region*.

4677 **Construct** – a directive and the associated statement, loop, or structured block, if any.

4678 **CUDA** – the CUDA environment from NVIDIA is a C-like programming environment used to  
4679 explicitly control and program an NVIDIA GPU.

- 4680 **Current device** – the device represented by the *acc-current-device-type-var* and *acc-current-device-*  
4681 *num-var* ICVs
- 4682 **Current device type** – the device type represented by the *acc-current-device-type-var* ICV
- 4683 **Data lifetime** – the lifetime of a data object in device memory, which may begin at the entry to  
4684 a data region, or at an **enter data** directive, or at a data API call such as **acc\_copyin** or  
4685 **acc\_create**, and which may end at the exit from a data region, or at an **exit data** directive,  
4686 or at a data API call such as **acc\_delete**, **acc\_copyout**, or **acc\_shutdown**, or at the end of  
4687 the program execution.
- 4688 **Data region** – a *region* defined by a **data** construct, or an implicit data region for a function or  
4689 subroutine containing OpenACC directives. Data constructs typically allocate device memory and  
4690 copy data from host to device memory upon entry, and copy data from device to local memory and  
4691 deallocate device memory upon exit. Data regions may contain other data regions and compute  
4692 regions.
- 4693 **Default asynchronous queue** – the asynchronous activity queue represented in the *acc-default-*  
4694 *async-var* ICV
- 4695 **Device** – a general reference to an accelerator or a multicore CPU.
- 4696 **Device memory** – memory attached to a device, logically and physically separate from the host  
4697 memory.
- 4698 **Device thread** – a thread of execution that executes on any device.
- 4699 **Directive** – in C or C++, a **#pragma**, or in Fortran, a specially formatted comment statement, that  
4700 is interpreted by a compiler to augment information about or specify the behavior of the program.
- 4701 **Discrete memory** – memory accessible from the local thread that is not accessible from the current  
4702 device, or memory accessible from the current device that is not accessible from the local thread.
- 4703 **DMA** – Direct Memory Access, a method to move data between physically separate memories;  
4704 this is typically performed by a DMA engine, separate from the host CPU, that can access the host  
4705 physical memory as well as an IO device or other physical memory.
- 4706 **GPU** – a Graphics Processing Unit; one type of accelerator.
- 4707 **GPGPU** – General Purpose computation on Graphics Processing Units.
- 4708 **Host** – the main CPU that in this context may have one or more attached accelerators. The host  
4709 CPU controls the program regions and data loaded into and executed on one or more devices.
- 4710 **Host thread** – a thread of execution that executes on the host.
- 4711 **Implicit data region** – the data region that is implicitly defined for a Fortran subprogram or C  
4712 function. A call to a subprogram or function enters the implicit data region, and a return from the  
4713 subprogram or function exits the implicit data region.
- 4714 **Kernel** – a nested loop executed in parallel by the accelerator. Typically the loops are divided into  
4715 a parallel domain, and the body of the loop becomes the body of the kernel.
- 4716 **Kernels region** – a *region* defined by a **kernels** construct. A kernels region is a structured block  
4717 which is compiled for the accelerator. The code in the kernels region will be divided by the compiler  
4718 into a sequence of kernels; typically each loop nest will become a single kernel. A kernels region

4719 may require space in device memory to be allocated and data to be copied from local memory to  
4720 device memory upon region entry, and data to be copied from device memory to local memory and  
4721 space in device memory to be deallocated upon exit.

4722 **Level of parallelism** – The possible levels of parallelism in OpenACC are gang, worker, vector,  
4723 and sequential. One or more of gang, worker, and vector parallelism may appear on a loop con-  
4724 struct. Sequential execution corresponds to no parallelism. The **gang**, **worker**, **vector**, and  
4725 **seq** clauses specify the level of parallelism for a loop.

4726 **Local device** – the device where the *local thread* executes.

4727 **Local memory** – the memory associated with the *local thread*.

4728 **Local thread** – the host thread or the accelerator thread that executes an OpenACC directive or  
4729 construct.

4730 **Loop trip count** – the number of times a particular loop executes.

4731 **MIMD** – a method of parallel execution (Multiple Instruction, Multiple Data) where different exe-  
4732 cution units or threads execute different instruction streams asynchronously with each other.

4733 **OpenCL** – short for Open Compute Language, a developing, portable standard C-like programming  
4734 environment that enables low-level general-purpose programming on GPUs and other accelerators.

4735 **Orphaned loop construct** - a **loop** construct that is not lexically contained in any compute con-  
4736 struct, that is, that has no parent compute construct.

4737 **Parallel region** – a *region* defined by a **parallel** construct. A parallel region is a structured block  
4738 which is compiled for the accelerator. A parallel region typically contains one or more work-sharing  
4739 loops. A parallel region may require space in device memory to be allocated and data to be copied  
4740 from local memory to device memory upon region entry, and data to be copied from device memory  
4741 to local memory and space in device memory to be deallocated upon exit.

4742 **Parent compute construct** – for a **loop** construct, the **parallel**, **kernels**, or **serial** con-  
4743 struct that lexically contains the **loop** construct and is the innermost compute construct that con-  
4744 tains that **loop** construct, if any.

4745 **Present data** – data for which the sum of the structured and dynamic reference counters is greater  
4746 than zero.

4747 **Private data** – with respect to an iterative loop, data which is used only during a particular loop  
4748 iteration. With respect to a more general region of code, data which is used within the region but is  
4749 not initialized prior to the region and is re-initialized prior to any use after the region.

4750 **Procedure** – in C or C++, a function in the program; in Fortran, a subroutine or function.

4751 **Region** – all the code encountered during an instance of execution of a construct. A region includes  
4752 any code in called routines, and may be thought of as the dynamic extent of a construct. This may  
4753 be a *parallel region*, *kernels region*, *serial region*, *data region* or *implicit data region*.

4754 **Scalar** – a variable of scalar datatype. In Fortran, scalars must not have allocatable or pointer  
4755 attributes.

4756 **Scalar datatype** – an intrinsic or built-in datatype that is not an array or aggregate datatype. In For-  
4757 tran, scalar datatypes are integer, real, double precision, complex, or logical. In C, scalar datatypes

4758 are char (signed or unsigned), int (signed or unsigned, with optional short, long or long long at-  
4759 tribute), enum, float, double, long double, \_Complex (with optional float or long attribute), or any  
4760 pointer datatype. In C++, scalar datatypes are char (signed or unsigned), wchar\_t, int (signed or  
4761 unsigned, with optional short, long or long long attribute), enum, bool, float, double, long double,  
4762 or any pointer datatype. Not all implementations or targets will support all of these datatypes.

4763 **Serial region** – a *region* defined by a **serial** construct. A serial region is a structured block which  
4764 is compiled for the accelerator. A serial region contains code that is executed by a single gang of a  
4765 single worker with a vector length of one. A serial region may require space in device memory to be  
4766 allocated and data to be copied from local memory to device memory upon region entry, and data  
4767 to be copied from device memory to local memory and space in device memory to be deallocated  
4768 upon exit.

4769 **Shared memory** – memory that is accessible from both the local thread and the current device.

4770 **SIMD** – A method of parallel execution (single-instruction, multiple-data) where the same instruc-  
4771 tion is applied to multiple data elements simultaneously.

4772 **SIMD operation** – a *vector operation* implemented with SIMD instructions.

4773 **Structured block** – in C or C++, an executable statement, possibly compound, with a single entry  
4774 at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single  
4775 entry at the top and a single exit at the bottom.

4776 **Thread** – On a host CPU, a thread is defined by a program counter and stack location; several host  
4777 threads may comprise a process and share host memory. On an accelerator, a thread is any one  
4778 vector lane of one worker of one gang.

4779 **var** – the name of a variable (scalar, array, or composite variable), or a subarray specification, or an  
4780 array element, or a composite variable member, or the name of a Fortran common block between  
4781 slashes.

4782 **Vector operation** – a single operation or sequence of operations applied uniformly to each element  
4783 of an array.

4784 **Visible device copy** – a copy of a variable, array, or subarray allocated in device memory that is  
4785 visible to the program unit being compiled.



## 4786 **A. Recommendations for Implementers**

4787 This section gives recommendations for standard names and extensions to use for implementations  
4788 for specific targets and target platforms, to promote portability across such implementations, and  
4789 recommended options that programmers find useful. While this appendix is not part of the Open-  
4790 ACC specification, implementations that provide the functionality specified herein are strongly rec-  
4791 ommended to use the names in this section. The first subsection describes devices, such as NVIDIA  
4792 GPUs. The second subsection describes additional API routines for target platforms, such as CUDA  
4793 and OpenCL. The third subsection lists several recommended options for implementations.

### 4794 **A.1 Target Devices**

#### 4795 **A.1.1 NVIDIA GPU Targets**

4796 This section gives recommendations for implementations that target NVIDIA GPU devices.

##### 4797 **Accelerator Device Type**

4798 These implementations should use the name **acc\_device\_nvidia** for the **acc\_device\_t**  
4799 type or return values from OpenACC Runtime API routines.

##### 4800 **ACC\_DEVICE\_TYPE**

4801 An implementation should use the case-insensitive name **nvidia** for the environment variable  
4802 **ACC\_DEVICE\_TYPE**.

##### 4803 **device\_type clause argument**

4804 An implementation should use the case-insensitive name **nvidia** as the argument to the **device\_type**  
4805 clause.

#### 4806 **A.1.2 AMD GPU Targets**

4807 This section gives recommendations for implementations that target AMD GPUs.

##### 4808 **Accelerator Device Type**

4809 These implementations should use the name **acc\_device\_radeon** for the **acc\_device\_t**  
4810 type or return values from OpenACC Runtime API routines.

##### 4811 **ACC\_DEVICE\_TYPE**

4812 These implementations should use the case-insensitive name **radeon** for the environment variable  
4813 **ACC\_DEVICE\_TYPE**.

##### 4814 **device\_type clause argument**

4815 An implementation should use the case-insensitive name **radeon** as the argument to the **device\_type**  
4816 clause.

### 4817 **A.1.3 Multicore Host CPU Target**

4818 This section gives recommendations for implementations that target the multicore host CPU.

#### 4819 **Accelerator Device Type**

4820 These implementations should use the name **acc\_device\_host** for the **acc\_device\_t** type  
4821 or return values from OpenACC Runtime API routines.

#### 4822 **ACC\_DEVICE\_TYPE**

4823 These implementations should use the case-insensitive name **host** for the environment variable  
4824 **ACC\_DEVICE\_TYPE**.

#### 4825 **device\_type clause argument**

4826 An implementation should use the case-insensitive name **host** as the argument to the **device\_type**  
4827 clause.

## 4828 **A.2 API Routines for Target Platforms**

4829 These runtime routines allow access to the interface between the OpenACC runtime API and the  
4830 underlying target platform. An implementation may not implement all these routines, but if it  
4831 provides this functionality, it should use these function names.

### 4832 **A.2.1 NVIDIA CUDA Platform**

4833 This section gives runtime API routines for implementations that target the NVIDIA CUDA Run-  
4834 time or Driver API.

#### 4835 **acc\_get\_current\_cuda\_device**

##### 4836 **Summary**

4837 The **acc\_get\_current\_cuda\_device** routine returns the NVIDIA CUDA device handle for  
4838 the current device.

##### 4839 **Format**

4840 C or C++:

```
4841     void* acc_get_current_cuda_device ();
```

#### 4842 **acc\_get\_current\_cuda\_context**

##### 4843 **Summary**

4844 The **acc\_get\_current\_cuda\_context** routine returns the NVIDIA CUDA context handle  
4845 in use for the current device.

##### 4846 **Format**

4847 C or C++:

```
4848     void* acc_get_current_cuda_context ();
```

**4849 acc\_get\_cuda\_stream****4850 Summary**

4851 The **acc\_get\_cuda\_stream** routine returns the NVIDIA CUDA stream handle in use for the  
4852 current device for the asynchronous activity queue associated with the **async** argument. This  
4853 argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.

**4854 Format**

4855 C or C++:

```
4856     void* acc_get_cuda_stream ( int async );
```

**4857 acc\_set\_cuda\_stream****4858 Summary**

4859 The **acc\_set\_cuda\_stream** routine sets the NVIDIA CUDA stream handle the current device  
4860 for the asynchronous activity queue associated with the **async** argument. This argument must be  
4861 an *async-argument* as defined in Section 2.16.1 *async* clause.

**4862 Format**

4863 C or C++:

```
4864     void acc_set_cuda_stream ( int async, void* stream );
```

**4865 A.2.2 OpenCL Target Platform**

4866 This section gives runtime API routines for implementations that target the OpenCL API on any  
4867 device.

**4868 acc\_get\_current\_opengl\_device****4869 Summary**

4870 The **acc\_get\_current\_opengl\_device** routine returns the OpenCL device handle for the  
4871 current device.

**4872 Format**

4873 C or C++:

```
4874     void* acc_get_current_opengl_device ();
```

**4875 acc\_get\_current\_opengl\_context****4876 Summary**

4877 The **acc\_get\_current\_opengl\_context** routine returns the OpenCL context handle in use  
4878 for the current device.

**4879 Format**

4880 C or C++:

```
4881     void* acc_get_current_opengl_context ();
```

**4882 acc\_get\_opengl\_queue****4883 Summary**

4884 The **acc\_get\_opengl\_queue** routine returns the OpenCL command queue handle in use for  
4885 the current device for the asynchronous activity queue associated with the **async** argument. This  
4886 argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.

4887 **Format**

4888 C or C++:

4889 `cl_command_queue acc_get_opengl_queue ( int async );`4890 **acc\_set\_opengl\_queue**4891 **Summary**

4892 The `acc_set_opengl_queue` routine returns the OpenCL command queue handle in use for  
 4893 the current device for the asynchronous activity queue associated with the `async` argument. This  
 4894 argument must be an *async-argument* as defined in Section 2.16.1 `async` clause.

4895 **Format**

4896 C or C++:

4897 `void acc_set_opengl_queue ( int async, cl_command_queue cmdqueue`  
 4898 `);`

4899 **A.3 Recommended Options**

4900 The following options are recommended for implementations; for instance, these may be imple-  
 4901 mented as command-line options to a compiler or settings in an IDE.

4902 **A.3.1 C Pointer in Present clause**

4903 This revision of OpenACC clarifies the construct:

```
4904 void test(int n ){
4905 float* p;
4906 ...
4907 #pragma acc data present(p)
4908 {
4909     // code here...
4910 }
```

4911 This example tests whether the pointer `p` itself is present in the current device memory. Implemen-  
 4912 tations before this revision commonly implemented this by testing whether the pointer target `p[0]`  
 4913 was present in the current device memory, and this appears in many programs assuming such. Until  
 4914 such programs are modified to comply with this revision, an option to implement `present(p)` as  
 4915 `present(p[0])` for C pointers may be helpful to users.

4916 **A.3.2 Automatic Data Attributes**

4917 If an implementation implements autoscoping or another analysis to automatically determine a vari-  
 4918 able's data attributes, an option to report which variables' data attributes are not as defined in Sec-  
 4919 tion 2.6 would be helpful to users. An option to disable the analysis would be helpful to promote  
 4920 program portability across implementations.

# Index

- 4921 `_OPENACC`, 15, 17–19, 21, 26, 117
- 4922 `acc-current-device-num-var`, 26
- 4923 `acc-current-device-type-var`, 26
- 4924 `acc-default-async-var`, 26, 79
- 4925 `acc_async_noval`, 15, 79
- 4926 `acc_async_sync`, 15, 79
- 4927 `acc_device_host`, 138
- 4928 `ACC_DEVICE_NUM`, 26, 109
- 4929 `acc_device_nvidia`, 137
- 4930 `acc_device_radeon`, 137
- 4931 `ACC_DEVICE_TYPE`, 26, 109, 137, 138
- 4932 `ACC_PROFLIB`, 109
- 4933 action
  - 4934 `attach`, 41, 45
  - 4935 `copyin`, 44
  - 4936 `copyout`, 44
  - 4937 `create`, 44
  - 4938 `delete`, 45
  - 4939 `detach`, 41, 45
  - 4940 `immediate`, 46
  - 4941 `present decrement`, 44
  - 4942 `present increment`, 43
- 4943 AMD GPU target, 137
- 4944 `async` clause, 40, 75, 79
- 4945 `async queue`, 11
- 4946 `async-argument`, 79
- 4947 asynchronous execution, 11, 79
- 4948 `atomic` construct, 16, 63
- 4949 `attach` action, 41, 45
- 4950 `attach` clause, 50
- 4951 attachment counter, 41
- 4952 `auto` clause, 16, 57
  - 4953 `portability`, 55
- 4954 `autoscopying`, 140
- 4955 barrier synchronization, 10, 29, 31, 133
- 4956 `bind` clause, 78
- 4957 `block` construct, 133
- 4958 `cache` directive, 61
- 4959 `capture` clause, 67
- 4960 `collapse` clause, 54
- 4961 common block, 41, 68, 79
- 4962 `compute` construct, 133
- 4963 `compute region`, 133
- 4964 `construct`, 133
- 4965 `atomic`, 63
- 4966 `compute`, 133
- 4967 `data`, 37, 41
- 4968 `host_data`, 51
- 4969 `kernels`, 30, 41
- 4970 `kernels loop`, 62
- 4971 `parallel`, 28, 41
- 4972 `parallel loop`, 62
- 4973 `serial`, 30, 41
- 4974 `serial loop`, 62
- 4975 `copy` clause, 36, 47
- 4976 `copyin` action, 44
- 4977 `copyin` clause, 47
- 4978 `copyout` action, 44
- 4979 `copyout` clause, 48
- 4980 `create` action, 44
- 4981 `create` clause, 49, 69
- 4982 CUDA, 11, 12, 133, 137, 138
- 4983 data attribute
  - 4984 `explicitly determined`, 34
  - 4985 `implicitly determined`, 34
  - 4986 `predetermined`, 34, 35
- 4987 data clause, 41
- 4988 `data` construct, 37, 41
- 4989 data lifetime, 134
- 4990 data region, 36, 134
  - 4991 `implicit`, 36
- 4992 data-independent `loop` construct, 53
- 4993 `declare` directive, 16, 67
- 4994 `default` clause, 35
- 4995 `default` clause, 34, 38
- 4996 `default (none)` clause, 35
- 4997 `default (none)` clause, 15
- 4998 `default(present)`, 35
- 4999 `delete` action, 45
- 5000 `delete` clause, 50
- 5001 `detach` action, 41, 45
  - 5002 `immediate`, 46
- 5003 `detach` clause, 51
- 5004 `device` clause, 74
- 5005 `device_resident` clause, 69
- 5006 `device_type` clause, 27
- 5007 `device_type` clause, 16, 42, 137, 138
- 5008 `deviceptr` clause, 41, 46

- 5009 direct memory access, 11, 134  
5010 DMA, 11, 134
- 5011 **enter data** directive, 38, 41  
5012 environment variable  
5013 `_OPENACC`, 26  
5014 `ACC_DEVICE_NUM`, 26, 109  
5015 `ACC_DEVICE_TYPE`, 26, 109, 137, 138  
5016 `ACC_PROFLIB`, 109  
5017 **exit data** directive, 38, 41  
5018 explicitly determined data attribute, 34
- 5019 **firstprivate** clause, 36  
5020 **firstprivate** clause, 33
- 5021 gang, 29  
5022 **gang** clause, 54, 77  
5023 implicit, 55  
5024 gang parallelism, 10  
5025 *gang-arg*, 53  
5026 gang-partitioned mode, 10  
5027 optimizations, 55  
5028 gang-redundant mode, 10, 29  
5029 GP mode, 10  
5030 GR mode, 10
- 5031 **host**, 138  
5032 **host** clause, 16, 74  
5033 **host\_data** construct, 51
- 5034 ICV, 26  
5035 **if** clause, 38, 40, 71–73, 75, 81  
5036 immediate detach action, 46  
5037 implicit data region, 36  
5038 implicit **gang** clause, 55  
5039 implicitly determined data attribute, 34  
5040 **independent** clause, 56  
5041 **init** directive, 70  
5042 internal control variable, 26
- 5043 **kernels** construct, 30, 41  
5044 **kernels loop** construct, 62
- 5045 level of parallelism, 10, 135  
5046 **link** clause, 16, 41, 70  
5047 local device, 11  
5048 local memory, 11  
5049 local thread, 11  
5050 **loop** construct, 52  
5051 data-independent, 53  
5052 orphaned, 53  
5053 sequential, 53
- 5054 **no\_create** clause, 49  
5055 **nohost** clause, 78  
5056 **num\_gangs** clause, 32  
5057 **num\_workers** clause, 32  
5058 **nvidia**, 137  
5059 NVIDIA GPU target, 137
- 5060 OpenCL, 11, 12, 135, 137, 139  
5061 optimizations  
5062 gang-partitioned mode, 55  
5063 orphaned **loop** construct, 53
- 5064 **parallel** construct, 28, 41  
5065 **parallel loop** construct, 62  
5066 parallelism  
5067 level, 10, 135  
5068 parent compute construct, 53  
5069 portability  
5070 **auto** clause, 55  
5071 predetermined data attribute, 34, 35  
5072 **present** clause, 35, 41, 46, 47  
5073 present decrement action, 44  
5074 present increment action, 43  
5075 **private** clause, 33, 57
- 5076 **radeon**, 137  
5077 **read** clause, 66  
5078 **reduction** clause, 33, 57  
5079 reference counter, 40  
5080 region  
5081 compute, 133  
5082 data, 36, 134  
5083 implicit data, 36  
5084 **routine** directive, 16, 76
- 5085 **self** clause, 16, 74  
5086 sentinel, 25  
5087 **seq** clause, 56, 78  
5088 sequential **loop** construct, 53  
5089 **serial** construct, 30, 41  
5090 **serial loop** construct, 62  
5091 **shutdown** directive, 71  
5092 *size-expr*, 53  
5093 structured-block, 136
- 5094 thread, 136

- 5095 **tile** clause, 16, 57
  
- 5096 **update** clause, 66
- 5097 **update** directive, 74
- 5098 **use\_device** clause, 52
  
- 5099 **vector** clause, 56, 77
- 5100 vector lane, 29
- 5101 vector parallelism, 10
- 5102 vector-partitioned mode, 10
- 5103 vector-single mode, 10
- 5104 **vector\_length** clause, 32
- 5105 visible device copy, 136
- 5106 VP mode, 10
- 5107 VS mode, 10
  
- 5108 **wait** clause, 40, 75, 80
- 5109 **wait** directive, 80
- 5110 worker, 29
- 5111 **worker** clause, 55, 77
- 5112 worker parallelism, 10
- 5113 worker-partitioned mode, 10
- 5114 worker-single mode, 10
- 5115 WP mode, 10
- 5116 WS mode, 10