

1

# The OpenACC<sup>®</sup>

## Application Programming Interface

2

3                   **Version 2.6 Public Review Document**  
4 **Please send corrections or suggestions to [feedback@openacc.org](mailto:feedback@openacc.org)**

5                   OpenACC-Standard.org

6                   November, 2017

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

11 © 2011-2017 OpenACC-Standard.org. All rights reserved.

# 12 Contents

13	<b>1. Introduction</b>	<b>7</b>
14	1.1. Scope . . . . .	7
15	1.2. Execution Model . . . . .	7
16	1.3. Memory Model . . . . .	9
17	1.4. Conventions used in this document . . . . .	10
18	1.5. Organization of this document . . . . .	11
19	1.6. References . . . . .	11
20	1.7. Changes from Version 1.0 to 2.0 . . . . .	12
21	1.8. Corrections in the August 2013 document . . . . .	13
22	1.9. Changes from Version 2.0 to 2.5 . . . . .	13
23	1.10. Changes from Version 2.5 to 2.6 . . . . .	14
24	1.11. Topics Deferred For a Future Revision . . . . .	15
25	<b>2. Directives</b>	<b>17</b>
26	2.1. Directive Format . . . . .	17
27	2.2. Conditional Compilation . . . . .	18
28	2.3. Internal Control Variables . . . . .	18
29	2.3.1. Modifying and Retrieving ICV Values . . . . .	18
30	2.4. Device-Specific Clauses . . . . .	19
31	2.5. Compute Constructs . . . . .	20
32	2.5.1. Parallel Construct . . . . .	20
33	2.5.2. Kernels Construct . . . . .	21
34	2.5.3. <b>Serial Construct</b> . . . . .	23
35	2.5.4. if clause . . . . .	24
36	2.5.5. async clause . . . . .	25
37	2.5.6. wait clause . . . . .	25
38	2.5.7. num_gangs clause . . . . .	25
39	2.5.8. num_workers clause . . . . .	25
40	2.5.9. vector_length clause . . . . .	25
41	2.5.10. private clause . . . . .	25
42	2.5.11. firstprivate clause . . . . .	26
43	2.5.12. reduction clause . . . . .	26
44	2.5.13. default clause . . . . .	27
45	2.6. Data Environment . . . . .	27
46	2.6.1. Variables with Predetermined Data Attributes . . . . .	27
47	2.6.2. Data Regions and Data Lifetimes . . . . .	28
48	2.6.3. Data Construct . . . . .	28
49	2.6.4. Enter Data and Exit Data Directives . . . . .	29
50	2.6.5. Reference Counting . . . . .	31
51	2.6.6. <b>Attachment Counters</b> . . . . .	32

52	2.7. Data Clauses . . . . .	32
53	2.7.1. Data Specification in Data Clauses . . . . .	33
54	2.7.2. <b>Data Clause Actions</b> . . . . .	34
55	2.7.3. deviceptr clause . . . . .	37
56	2.7.4. present clause . . . . .	38
57	2.7.5. copy clause . . . . .	38
58	2.7.6. copyin clause . . . . .	39
59	2.7.7. copyout clause . . . . .	39
60	2.7.8. create clause . . . . .	40
61	2.7.9. <b>no_create clause</b> . . . . .	41
62	2.7.10. delete clause . . . . .	41
63	2.7.11. <b>attach clause</b> . . . . .	42
64	2.7.12. <b>detach clause</b> . . . . .	42
65	2.8. Host Data Construct . . . . .	42
66	2.8.1. use_device clause . . . . .	43
67	2.8.2. <b>if clause</b> . . . . .	43
68	2.8.3. <b>if_present clause</b> . . . . .	43
69	2.9. Loop Construct . . . . .	43
70	2.9.1. collapse clause . . . . .	45
71	2.9.2. gang clause . . . . .	45
72	2.9.3. worker clause . . . . .	45
73	2.9.4. vector clause . . . . .	46
74	2.9.5. seq clause . . . . .	46
75	2.9.6. auto clause . . . . .	46
76	2.9.7. tile clause . . . . .	47
77	2.9.8. device_type clause . . . . .	47
78	2.9.9. independent clause . . . . .	47
79	2.9.10. private clause . . . . .	47
80	2.9.11. reduction clause . . . . .	48
81	2.10. Cache Directive . . . . .	48
82	2.11. Combined Constructs . . . . .	49
83	2.12. Atomic Construct . . . . .	51
84	2.13. Declare Directive . . . . .	55
85	2.13.1. device_resident clause . . . . .	56
86	2.13.2. create clause . . . . .	57
87	2.13.3. link clause . . . . .	58
88	2.14. Executable Directives . . . . .	58
89	2.14.1. Init Directive . . . . .	58
90	2.14.2. Shutdown Directive . . . . .	59
91	2.14.3. Set Directive . . . . .	60
92	2.14.4. Update Directive . . . . .	61
93	2.14.5. Wait Directive . . . . .	64
94	2.14.6. Enter Data Directive . . . . .	64
95	2.14.7. Exit Data Directive . . . . .	64
96	2.15. Procedure Calls in Compute Regions . . . . .	64
97	2.15.1. Routine Directive . . . . .	64
98	2.15.2. Global Data Access . . . . .	67

99	2.16. Asynchronous Behavior . . . . .	67
100	2.16.1. async clause . . . . .	67
101	2.16.2. wait clause . . . . .	68
102	2.16.3. Wait Directive . . . . .	68
103	2.17. Fortran Optional Arguments . . . . .	69
104	<b>3. Runtime Library</b>	<b>71</b>
105	3.1. Runtime Library Definitions . . . . .	71
106	3.2. Runtime Library Routines . . . . .	72
107	3.2.1. acc_get_num_devices . . . . .	72
108	3.2.2. acc_set_device_type . . . . .	73
109	3.2.3. acc_get_device_type . . . . .	73
110	3.2.4. acc_set_device_num . . . . .	74
111	3.2.5. acc_get_device_num . . . . .	75
112	3.2.6. acc_get_property . . . . .	75
113	3.2.7. acc_init . . . . .	76
114	3.2.8. acc_shutdown . . . . .	77
115	3.2.9. acc_async_test . . . . .	77
116	3.2.10. acc_async_test_all . . . . .	78
117	3.2.11. acc_wait . . . . .	78
118	3.2.12. acc_wait_async . . . . .	79
119	3.2.13. acc_wait_all . . . . .	79
120	3.2.14. acc_wait_all_async . . . . .	80
121	3.2.15. acc_get_default_async . . . . .	80
122	3.2.16. acc_set_default_async . . . . .	81
123	3.2.17. acc_on_device . . . . .	81
124	3.2.18. acc_malloc . . . . .	82
125	3.2.19. acc_free . . . . .	82
126	3.2.20. acc_copyin . . . . .	83
127	3.2.21. acc_create . . . . .	84
128	3.2.22. acc_copyout . . . . .	85
129	3.2.23. acc_delete . . . . .	86
130	3.2.24. acc_update_device . . . . .	87
131	3.2.25. acc_update_self . . . . .	87
132	3.2.26. acc_map_data . . . . .	88
133	3.2.27. acc_unmap_data . . . . .	88
134	3.2.28. acc_deviceptr . . . . .	89
135	3.2.29. acc_hostptr . . . . .	89
136	3.2.30. acc_is_present . . . . .	89
137	3.2.31. acc_memcpy_to_device . . . . .	90
138	3.2.32. acc_memcpy_from_device . . . . .	90
139	3.2.33. acc_memcpy_device . . . . .	91
140	3.2.34. acc_attach . . . . .	91
141	3.2.35. acc_detach . . . . .	92
142	<b>4. Environment Variables</b>	<b>93</b>
143	4.1. ACC_DEVICE_TYPE . . . . .	93
144	4.2. ACC_DEVICE_NUM . . . . .	93

145	4.3. ACC_PROFLIB . . . . .	93
146	<b>5. Profiling Interface</b>	<b>95</b>
147	5.1. Events . . . . .	95
148	5.1.1. Runtime Initialization and Shutdown . . . . .	96
149	5.1.2. Device Initialization and Shutdown . . . . .	96
150	5.1.3. Enter Data and Exit Data . . . . .	97
151	5.1.4. Data Allocation . . . . .	97
152	5.1.5. Data Construct . . . . .	98
153	5.1.6. Update Directive . . . . .	98
154	5.1.7. Compute Construct . . . . .	98
155	5.1.8. Enqueue Kernel Launch . . . . .	99
156	5.1.9. Enqueue Data Update (Upload and Download) . . . . .	99
157	5.1.10. Wait . . . . .	100
158	5.2. Callbacks Signature . . . . .	100
159	5.2.1. First Argument: General Information . . . . .	101
160	5.2.2. Second Argument: Event-Specific Information . . . . .	102
161	5.2.3. Third Argument: API-Specific Information . . . . .	105
162	5.3. Loading the Library . . . . .	106
163	5.3.1. Library Registration . . . . .	107
164	5.3.2. Statically-Linked Library Initialization . . . . .	108
165	5.3.3. Runtime Dynamic Library Loading . . . . .	108
166	5.3.4. Preloading with LD_PRELOAD . . . . .	109
167	5.3.5. Application-Controlled Initialization . . . . .	110
168	5.4. Registering Event Callbacks . . . . .	110
169	5.4.1. Event Registration and Unregistration . . . . .	111
170	5.4.2. Disabling and Enabling Callbacks . . . . .	112
171	5.5. Advanced Topics . . . . .	113
172	5.5.1. Dynamic Behavior . . . . .	114
173	5.5.2. OpenACC Events During Event Processing . . . . .	115
174	5.5.3. Multiple Host Threads . . . . .	115
175	<b>6. Glossary</b>	<b>117</b>
176	<b>A. Recommendations for Implementors</b>	<b>121</b>
177	A.1. Target Devices . . . . .	121
178	A.1.1. NVIDIA GPU Targets . . . . .	121
179	A.1.2. AMD GPU Targets . . . . .	121
180	A.2. API Routines for Target Platforms . . . . .	122
181	A.2.1. NVIDIA CUDA Platform . . . . .	122
182	A.2.2. OpenCL Target Platform . . . . .	123
183	A.2.3. C Pointer in Present clause . . . . .	124
184	A.2.4. Autoscopying . . . . .	125

# 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 offloading programs written in C, C++, and Fortran programs from a *host* CPU to an attached *accelerator* device. The method described provides a model for accelerator programming that is portable across operating systems and various types of host 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 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 the 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 onto an accelerator, and similar performance-related details.

## 1.1. Scope

This OpenACC API document covers only user-directed accelerator programming, where the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The remainder of the program will be executed 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 offloaded to an accelerator.

This document does not describe automatic detection and offloading of regions of code to an accelerator by a compiler or other tool. This document does not describe splitting loops or code regions to multiple accelerators attached to a single host. While future compilers may allow for automatic offloading, or offloading to multiple accelerators of the same type, or to 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 accelerator device, such as a GPU. Much of a user application executes on the host. Compute intensive regions are offloaded to the accelerator device under control of the host. The device executes *parallel regions*, which typically contain work-sharing loops, *kernels regions*, which typically contain one or more loops which are executed as kernels on the accelerator, **or *serial regions, which are blocks of sequential code that execute on the accelerator***. Even in accelerator-targeted regions, the host may orchestrate the execution by allocating memory on the accelerator

219 device, initiating data transfer, sending the code to the accelerator, passing arguments to the compute  
220 region, queuing the device code, waiting for completion, transferring results back to the host,  
221 and deallocating memory. In most cases, the host can queue a sequence of operations to be executed  
222 on the device, one after the other.

223 Most current accelerators support two or three levels of parallelism. Most accelerators support  
224 coarse-grain parallelism, which is fully parallel execution across execution units. There may be  
225 limited support for synchronization across coarse-grain parallel operations. Many accelerators also  
226 support fine-grain parallelism, often implemented as multiple threads of execution within a single  
227 execution unit, which are typically rapidly switched on the execution unit to tolerate long latency  
228 memory operations. Finally, most accelerators also support SIMD or vector operations within each  
229 execution unit. The execution model exposes these multiple levels of parallelism on the device and  
230 the programmer is required to understand the difference between, for example, a fully parallel loop  
231 and a loop that is vectorizable but requires synchronization between statements. A fully parallel  
232 loop can be programmed for coarse-grain parallel execution. Loops with dependences must either  
233 be split to allow coarse-grain parallel execution, or be programmed to execute on a single execution  
234 unit using fine-grain parallelism, vector parallelism, or sequentially.

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

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

246 When only one worker is active, in either GR or GP mode, the program is in *worker-single* mode  
247 (WS mode). When only one vector lane is active, the program is in *vector-single* mode (VS mode).  
248 If a gang reaches a loop or loop nest marked for worker-level work-sharing, the gang transitions to  
249 *worker-partitioned* mode (WP mode), which activates all the workers of the gang. The iterations  
250 of the loop or loops are partitioned across the workers of this gang. If the same loop is marked for  
251 both gang-partitioning and worker-partitioning, then the iterations of the loop are spread across all  
252 the workers of all the gangs. If a worker reaches a loop or loop nest marked for vector-level work-  
253 sharing, the worker will transition to *vector-partitioned* mode (VP mode). Similar to WP mode, the  
254 transition to VP mode activates all the vector lanes of the worker. The iterations of the loop or loops  
255 will be partitioned across the vector lanes using vector or SIMD operations. Again, a single loop  
256 may be marked for one, two, or all three of gang, worker, and vector parallelism, and the iterations  
257 of that loop will be spread across the gangs, workers, and vector lanes as appropriate.

258 The program starts executing with a single host thread, identified by a program counter and its  
259 stack. The thread may spawn additional host threads, for instance using the OpenMP API. On the  
260 accelerator device, a single vector lane of a single worker of a single gang is called a thread. When  
261 executing on the device, a parallel execution context is created on the accelerator and may contain  
262 many such threads.

263 The user should not attempt to implement barrier synchronization, critical sections or locks across



264 any of gang, worker, or vector parallelism. The execution model allows for an implementation that  
265 executes some gangs to completion before starting to execute other gangs. This means that trying  
266 to implement synchronization between gangs is likely to fail. In particular, a barrier across gangs  
267 cannot be implemented in a portable fashion, since all gangs may not ever be active at the same time.  
268 Similarly, the execution model allows for an implementation that executes some workers within a  
269 gang or vector lanes within a worker to completion before starting other workers or vector lanes,  
270 or for some workers or vector lanes to be suspended until other workers or vector lanes complete.  
271 This means that trying to implement synchronization across workers or vector lanes is likely to fail.  
272 In particular, implementing a barrier or critical section across workers or vector lanes using atomic  
273 operations and a busy-wait loop may never succeed, since the scheduler may suspend the worker or  
274 vector lane that owns the lock, and the worker or vector lane waiting on the lock can never complete.

275 On some devices, the accelerator may also create and launch compute regions, allowing for nested  
276 parallelism. In that case, the OpenACC directives may be executed by a host thread or an acceler-  
277 ator thread. This specification uses the term *local thread* or *local memory* to mean the thread that  
278 executes the directive, or the memory associated with that thread, whether that thread executes on  
279 the host or on the accelerator.

280 Most accelerators can operate asynchronously with respect to the host thread. With such devices, the  
281 accelerator has one or more activity queues. The host thread will enqueue operations onto the device  
282 activity queues, such as data transfers and procedure execution. After enqueueing the operation, the  
283 host thread can continue execution while the device operates independently and asynchronously.  
284 The host thread may query the device activity queue(s) and wait for all the operations in a queue  
285 to complete. Operations on a single device activity queue will complete before starting the next  
286 operation on the same queue; operations on different activity queues may be active simultaneously  
287 and may complete in any order.

### 288 1.3. Memory Model

289 The most significant difference between a host-only program and a host+accelerator program is that  
290 the memory on the accelerator may be physically and/or virtually separate from host memory. This  
291 is the case with most current GPUs, for example. In this case, the host thread may not be able to  
292 read or write device memory directly because it is not mapped into the host thread's virtual memory  
293 space. All data movement between host memory and device memory must be performed by the  
294 host thread through system calls that explicitly move data between the separate memories, typically  
295 using direct memory access (DMA) transfers. Similarly, it is not valid to assume the accelerator  
296 can read or write host memory, though this is supported by some accelerator devices, often with  
297 significant performance penalty.

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

- 304 • Memory bandwidth between host memory and device memory determines the level of com-  
305 pute intensity required to effectively accelerate a given region of code.

306 • The user should be aware that a separate device memory is usually significantly smaller than  
307 the host memory, prohibiting offloading regions of code that operate on very large amounts  
308 of data.

309 • Host addresses stored to pointers on the host may only be valid on the host; addresses stored  
310 to pointers on the device may only be valid on the device. Explicitly transferring pointer  
311 values between host and device memory is not advised. Dereferencing host pointers on the  
312 device or dereferencing device pointers on the host is likely to be invalid on such targets.

313 OpenACC exposes the separate memories through the use of a device data environment. Device  
314 data has an explicit lifetime, from when it is allocated or created until it is deleted. If the device  
315 shares physical and virtual memory with the local thread, the device data environment will be shared  
316 with the local thread. In that case, the implementation need not create new copies of the data for  
317 the device and no data movement need be done. If the device has a physically or virtually separate  
318 memory from the local thread, the implementation will allocate new data in the device memory and  
319 copy data from the local memory to the device environment.

320 Some accelerators (such as current GPUs) implement a weak memory model. In particular, they do  
321 not support memory coherence between operations executed by different threads; even on the same  
322 execution unit, memory coherence is only guaranteed when the memory operations are separated  
323 by an explicit memory fence. Otherwise, if one thread updates a memory location and another reads  
324 the same location, or two threads store a value to the same location, the hardware may not guarantee  
325 the same result for each execution. While a compiler can detect some potential errors of this nature,  
326 it is nonetheless possible to write a `compute` region that produces inconsistent numerical results.

327 Similarly, some accelerators implement a weak memory model for memory shared between the  
328 host and the accelerator, or memory shared between multiple accelerators. Programmers need to  
329 be very careful that the program uses appropriate synchronization to ensure that an assignment or  
330 modification to shared data by a host thread is complete and available before that data is used by  
331 an accelerator thread. Similarly, synchronization must be used to ensure that an assignment or  
332 modification to shared data by an accelerator thread is complete and available before that data is  
333 used by a host thread or by a thread on a different accelerator.

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

## 339 1.4. Conventions used in this document

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

```
#pragma acc
```

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

```
#pragma acc directive-name
```

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

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

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

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

345 To simplify the specification and convey appropriate constraint information, a *pqr-list* is a comma-  
346 separated list of *pqr* items. For example, an *int-expr-list* is a comma-separated list of one or more  
347 integer expressions. A *var-list* is a comma-separated list of one or more variable names or array  
348 names; in some clauses, a *var-list* may include subarrays with subscript ranges or may include  
349 common block names between slashes. The one exception is *clause-list*, which is a list of one or  
350 more clauses optionally separated by commas.

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

## 351 1.5. Organization of this document

352 The rest of this document is organized as follows:

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

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

358 Chapter 4 Environment Variables, defines user-settable environment variables used to control be-  
359 havior of accelerator-enabled programs at execution.

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

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

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

## 365 1.6. References

- 366 • *American National Standard Programming Language C*, ANSI X3.159-1989 (ANSI C).
- 367 • ISO/IEC 9899:1999, *Information Technology – Programming Languages – C (C99)*.
- 368 • ISO/IEC 14882:1998, *Information Technology – Programming Languages – C++*.
- 369 • ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran – Part*  
370 *1: Base Language*, (Fortran 2003).
- 371 • *OpenMP Application Program Interface*, version 4.0, July 2013
- 372 • *PGI Accelerator Programming Model for Fortran & C*, version 1.3, November 2011

- 373 • *NVIDIA CUDA<sup>™</sup> C Programming Guide*, version 7.0, March 2015.
- 374 • *The OpenCL Specification*, version 202, Khronos OpenCL Working Group, October 2014.

## 375 1.7. Changes from Version 1.0 to 2.0

- 376 • `_OPENACC` value updated to `201306`
- 377 • `default (none)` clause on `parallel` and `kernels` directives
- 378 • the implicit data attribute for scalars in `parallel` constructs has changed
- 379 • the implicit data attribute for scalars in loops with `loop` directives with the independent
- 380 attribute has been clarified
- 381 • `acc_async_sync` and `acc_async_noval` values for the `async` clause
- 382 • Clarified the behavior of the `reduction` clause on a `gang` loop
- 383 • Clarified allowable loop nesting (`gang` may not appear inside `worker`, which may not ap-
- 384 pear within `vector`)
- 385 • `wait` clause on `parallel`, `kernels` and `update` directives
- 386 • `async` clause on the `wait` directive
- 387 • `enter data` and `exit data` directives
- 388 • Fortran *common block* names may now be specified in many data clauses
- 389 • `link` clause for the `declare` directive
- 390 • the behavior of the `declare` directive for global data
- 391 • the behavior of a data clause with a C or C++ pointer variable has been clarified
- 392 • predefined data attributes
- 393 • support for multidimensional dynamic C/C++ arrays
- 394 • `tile` and `auto` loop clauses
- 395 • `update self` introduced as a preferred synonym for `update host`
- 396 • `routine` directive and support for separate compilation
- 397 • `device_type` clause and support for multiple device types
- 398 • nested parallelism using `parallel` or `kernels` region containing another `parallel` or `kernels` re-
- 399 gion
- 400 • `atomic` constructs
- 401 • new concepts: `gang-redundant`, `gang-partitioned`; `worker-single`, `worker-partitioned`; `vector-`
- 402 `single`, `vector-partitioned`; `thread`
- 403 • new API routines:
  - 404 – `acc_wait`, `acc_wait_all` instead of `acc_async_wait` and `acc_async_wait_all`
  - 405 – `acc_wait_async`

- 406       – **acc\_copyin**, **acc\_present\_or\_copyin**
- 407       – **acc\_create**, **acc\_present\_or\_create**
- 408       – **acc\_copyout**, **acc\_delete**
- 409       – **acc\_map\_data**, **acc\_unmap\_data**
- 410       – **acc\_deviceptr**, **acc\_hostptr**
- 411       – **acc\_is\_present**
- 412       – **acc\_memcpy\_to\_device**, **acc\_memcpy\_from\_device**
- 413       – **acc\_update\_device**, **acc\_update\_self**
- 414       • defined behavior with multiple host threads, such as with OpenMP
- 415       • recommendations for specific implementations
- 416       • clarified that no arguments are allowed on the **vector** clause in a parallel region

## 417 1.8. Corrections in the August 2013 document

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

## 421 1.9. Changes from Version 2.0 to 2.5

- 422       • The **\_OPENACC** value was updated to **201510**; see Section 2.2 Conditional Compilation.
- 423       • The **num\_gangs**, **num\_workers**, and **vector\_length** clauses are now allowed on the
- 424       **kernels** construct; see Section 2.5.2 Kernels Construct.
- 425       • Reduction on C++ class members, array elements, and struct elements are explicitly disal-
- 426       lowed; see Section 2.5.12 reduction clause.
- 427       • Reference counting is now used to manage the correspondence and lifetime of device data;
- 428       see Section 2.6.5 Reference Counting.
- 429       • The behavior of the **exit data** directive has changed to decrement the dynamic reference
- 430       count. A new optional **finalize** clause was added to set the dynamic reference count to
- 431       zero. See Section 2.6.4 Enter Data and Exit Data Directives.
- 432       • The **copy**, **copyin**, **copyout**, and **create** data clauses were changed to behave like
- 433       **present\_or\_copy**, etc. The **present\_or\_copy**, **pcopy**, **present\_or\_copyin**,
- 434       **pcopyin**, **present\_or\_copyout**, **pcopyout**, **present\_or\_create**, and **pcreate**
- 435       data clauses are no longer needed, though will be accepted for compatibility; see Section 2.7
- 436       Data Clauses.
- 437       • Reductions on orphaned gang loops are explicitly disallowed; see Section 2.9 Loop Construct.
- 438       • The description of the **loop auto** clause has changed; see Section 2.9.6 auto clause.

- 439 • Text was added to the **private** clause on a **loop** construct to clarify that a copy is made  
440 for each gang or worker or vector lane, not each thread; see Section 2.9.10 private clause.
- 441 • The description of the **reduction** clause on a **loop** construct was corrected; see Sec-  
442 tion 2.9.11 reduction clause.
- 443 • A restriction was added to the **cache** clause that all references to that variable must lie within  
444 the region being cached; see Section 2.10 Cache Directive.
- 445 • Text was added to the **private** and **reduction** clauses on a combined construct to clarify  
446 that they act like **private** and **reduction** on the **loop**, not **private** and **reduction**  
447 on the **parallel** or **reduction** on the **kernels**; see Section 2.11 Combined Constructs.
- 448 • The **declare create** directive with a Fortran **allocatable** has new behavior; see Sec-  
449 tion 2.13.2 create clause.
- 450 • New **init**, **shutdown**, **set** directives were added; see Section 2.14.1 Init Directive, 2.14.2  
451 Shutdown Directive, and 2.14.3 Set Directive.
- 452 • A new **if\_present** clause was added to the **update** directive, which changes the behavior  
453 when data is not present from a runtime error to a no-op; see Section 2.14.4 Update Directive.
- 454 • The **routine bind** clause definition changed; see Section 2.15.1 Routine Directive.
- 455 • An **acc routine** without **gang/worker/vector/seq** is now defined as an error; see  
456 Section 2.15.1 Routine Directive.
- 457 • A new **default (present)** clause was added for compute constructs; see Section 2.5.13  
458 default clause.
- 459 • The Fortran header file **openacc\_lib.h** is no longer supported; the Fortran module **openacc**  
460 should be used instead; see Section 3.1 Runtime Library Definitions.
- 461 • New API routines were added to get and set the default async queue value; see Section 3.2.15  
462 **acc\_get\_default\_async** and 3.2.16 **acc\_set\_default\_async**.
- 463 • The **acc\_copyin**, **acc\_create**, **acc\_copyout**, and **acc\_delete** API routines were  
464 changed to behave like **acc\_present\_or\_copyin**, etc. The **acc\_present\_or\_** names  
465 are no longer needed, though will be supported for compatibility. See Sections 3.2.20 and fol-  
466 lowing.
- 467 • Asynchronous versions of the data API routines were added; see Sections 3.2.20 and follow-  
468 ing.
- 469 • A new API routine added, **acc\_memcpy\_device**, to copy from one device address to  
470 another device address; see Section 3.2.31 **acc\_memcpy\_to\_device**.
- 471 • A new OpenACC interface for profile and trace tools was added; see Chapter 5 Profiling Interface.

## 472 1.10. Changes from Version 2.5 to 2.6

- 473 • The **\_OPENACC** value was updated to **201711**.
- 474 • A new **serial** compute construct was added. See Section 2.5.3 **Serial Construct**.
- 475 • A new runtime API query routine was added. **acc\_get\_property** may be called from

- 476 the host and returns properties about any device. See Section 3.2.6.
- 477 • The text has clarified that if a variable is in a reduction which spans two or more nested loops,  
478 each **loop** directive on any of those loops must have a **reduction** clause that contains the  
479 variable; see Section 2.9.11 reduction clause.
  - 480 • An optional **if** or **if\_present** clause is now allowed on the **host\_data** construct. See  
481 Section 2.8 Host Data Construct.
  - 482 • A new **no\_create** data clause is now allowed on compute and **data** constructs. See Sec-  
483 tion 2.7.9 **no\_create** clause.
  - 484 • The behavior of Fortran optional arguments in data clauses and in routine calls has been  
485 specified; see Section 2.17 **Fortran Optional Arguments**.
  - 486 • The descriptions of some of the Fortran versions of the runtime library routines were simpli-  
487 fied; see Section 3.2 Runtime Library Routines.
  - 488 • New *attach* and *detach* behavior was added to the data clauses, new **attach** and **detach**  
489 clauses were added, and matching **acc\_attach** and **acc\_detach** runtime API routines  
490 were added.
  - 491 • The Intel Coprocessor Offload Interface target and API routine sections were removed from  
492 the Section A Recommendations for Implementors, since Intel no longer produces this prod-  
493 uct.

## 494 1.11. Topics Deferred For a Future Revision

495 The following topics are under discussion for a future revision. Some of these are known to  
496 be important, while others will depend on feedback from users. Readers who have feedback or  
497 want to participate may post a message at the forum at [www.openacc.org](http://www.openacc.org), or may send email to  
498 [technical@openacc.org](mailto:technical@openacc.org) or [feedback@openacc.org](mailto:feedback@openacc.org). No promises are made or implied that all these  
499 items will be available in the next revision.

- 500 • Full support for C and C++ structs and struct members, including pointer members.
- 501 • Full support for Fortran derived types and derived type members, including allocatable and  
502 pointer members.
- 503 • Fully defined interaction with multiple host threads.
- 504 • Optionally removing the synchronization or barrier at the end of vector and worker loops.
- 505 • Allowing an **if** clause after a **device\_type** clause.
- 506 • A **shared** clause (or something similar) for the loop directive.
- 507 • Better support for multiple devices from a single thread, whether of the same type or of  
508 different types.





## 2. Directives

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

### 2.1. Directive Format

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

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

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

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

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

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

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

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

The sentinel (**!\$acc**, **c\$acc**, or **\*\$acc**) must occupy columns 1-5. Fixed form line length, white space, continuation, and column rules apply to the directive line. Initial directive lines must have

534 a space or zero in column 6, and continuation directive lines must have a character other than a  
535 space or zero in column 6. Comments may appear on the same line as a directive, starting with an  
536 exclamation point on or after column 7 and continuing to the end of the line.

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

540 Only one *directive-name* can be specified per directive, except that a combined directive name is  
541 considered a single *directive-name*. The order in which clauses appear is not significant unless  
542 otherwise specified. Clauses may be repeated unless otherwise specified. Some clauses have an  
543 argument that can contain a list.

## 544 2.2. Conditional Compilation

545 The `_OPENACC` macro name is defined to have a value `yyyymm` where `yyyy` is the year and `mm` is  
546 the month designation of the version of the OpenACC directives supported by the implementation.  
547 This macro must be defined by a compiler only when OpenACC directives are enabled. The version  
548 described here is **201711**.

## 549 2.3. Internal Control Variables

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

554 The ICVs are:

- 555 • *acc-device-type-var* - controls which type of accelerator device is used.
- 556 • *acc-device-num-var* - controls which accelerator device of the selected type is used.
- 557 • *acc-default-async-var* - controls which asynchronous queue is used when none is specified in  
558 an `async` clause.

### 559 2.3.1. Modifying and Retrieving ICV Values

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

ICV	Ways to modify values	Way to retrieve value
<i>acc-device-type-var</i>	<code>acc_set_device_type</code> <code>set_device_type</code> <code>ACC_DEVICE_TYPE</code>	<code>acc_get_device_type</code>
562 <i>acc-device-num-var</i>	<code>acc_set_device_num</code> <code>set_device_num</code> <code>ACC_DEVICE_NUM</code>	<code>acc_get_device_num</code>
<i>acc-default-async-var</i>	<code>acc_set_default_async</code> <code>set_default_async</code>	<code>acc_get_default_async</code>

563 The initial values are implementation-defined. After initial values are assigned, but before any  
 564 OpenACC construct or API routine is executed, the values of any environment variables that were  
 565 set by the user are read and the associated ICVs are modified accordingly. Clauses on OpenACC  
 566 constructs do not modify the ICV values. There is one copy of each ICV for each host thread. An  
 567 ICV value for a device thread may not be modified.

## 568 2.4. Device-Specific Clauses

569 OpenACC directives can specify different clauses or clause arguments for different accelerators  
 570 using the **device\_type** clause. The argument to the **device\_type** is a comma-separated list  
 571 of one or more accelerator architecture name identifiers, or an asterisk. A single directive may have  
 572 one or several **device\_type** clauses. Clauses on a directive with no **device\_type** apply to  
 573 all accelerator device types. Clauses that follow a **device\_type** up to the end of the directive  
 574 or up to the next **device\_type** are associated with this **device\_type**. Clauses associated  
 575 with a **device\_type** apply only when compiling for the accelerator device type named. Clauses  
 576 associated with a **device\_type** that has an asterisk argument apply to any accelerator device  
 577 type that was not named in any **device\_type** on that directive. The **device\_type** clauses  
 578 may appear in any order. For each directive, only certain clauses may follow a **device\_type**.

579 Clauses that precede any **device\_type** are *default clauses*. Clauses that follow a **device\_type**  
 580 are *device-specific clauses*. A clause may appear both as a default clause and as a device-specific  
 581 clause. In that case, the value in the device-specific clause is used when compiling for that device  
 582 type.

583 The supported accelerator device types are implementation-defined. Depending on the implemen-  
 584 tation and the compiling environment, an implementation may support only a single accelerator  
 585 device type, or may support multiple accelerator device types but only one at a time, or many sup-  
 586 port multiple accelerator device types in a single compilation.

587 An accelerator architecture name may be generic, such as a vendor, or more specific, such as a  
 588 particular generation of device; see Appendix A Recommendations for Implementors for recom-  
 589 mended names. When compiling for a particular device, the implementation will use the clauses  
 590 associated with the **device\_type** clause that specifies the most specific architecture name that  
 591 applies for this device; clauses associated with any other **device\_type** clause are ignored. In  
 592 this context, the asterisk is the least specific architecture name.

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

```

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

```

594 The `device_type` clause may be abbreviated to `dtype`.

## 595 2.5. Compute Constructs

### 596 2.5.1. Parallel Construct

597 **Summary** This fundamental construct starts parallel execution on the current accelerator device.

598 **Syntax** In C and C++, the syntax of the OpenACC `parallel` construct is

```

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

```

599 and in Fortran, the syntax is

```

!$acc parallel [clause-list]
    structured block
!$acc end parallel

```

600 where *clause* is one of the following:

```

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

```

601 **Description** When the program encounters an accelerator **parallel** construct, one or more  
 602 gangs of workers are created to execute the accelerator parallel region. The number of gangs, and  
 603 the number of workers in each gang and the number of vector lanes per worker remain constant for  
 604 the duration of that parallel region. Each gang begins executing the code in the structured block in  
 605 gang-redundant mode. This means that code within the parallel region, but outside of a loop with a  
 606 **loop** directive and gang-level worksharing, will be executed redundantly by all gangs.

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

610 If the **async** clause is not present, there is an implicit barrier at the end of the accelerator parallel  
 611 region, and the execution of the local thread will not proceed until all gangs have reached the end  
 612 of the parallel region.

613 If there is no **default (none)** clause on the construct, the compiler will implicitly determine data  
 614 attributes for variables that are referenced in the compute construct that do not have predetermined  
 615 data attributes and do not appear in a data clause on the compute construct, a lexically containing  
 616 **data** construct, or a visible **declare** directive. If there is no **default (present)** clause on  
 617 the construct, an array or variable of aggregate data type referenced in the **parallel** construct that  
 618 does not appear in a data clause for the construct or any enclosing **data** construct will be treated as  
 619 if it appeared in a **copy** clause for the **parallel** construct. If there is a **default (present)**  
 620 clause on the construct, the compiler will implicitly treat all arrays and variables of aggregate data  
 621 type without predetermined data attributes as if they appeared in a **present** clause. A scalar vari-  
 622 able referenced in the **parallel** construct that does not appear in a data clause for the construct  
 623 or any enclosing **data** construct will be treated as if it appeared in a **firstprivate** clause.

## 624 Restrictions

- 625 • A program may not branch into or out of an OpenACC **parallel** construct.
- 626 • A program must not depend on the order of evaluation of the clauses, or on any side effects  
627 of the evaluations.
- 628 • Only the **async**, **wait**, **num\_gangs**, **num\_workers**, and **vector\_length** clauses  
629 may follow a **device\_type** clause.
- 630 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical  
631 value; in C or C++, the condition must evaluate to a scalar integer value.
- 632 • At most one **default** clause may appear, and it must have a value of either **none** or  
633 **present**.

634 The **copy**, **copyin**, **copyout**, **create**, **no\_create**, **present**, **deviceptr**, and **attach**  
 635 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**  
 636 clauses are described in Sections 2.5.10 and Sections 2.5.11. The **device\_type** clause is de-  
 637 scribed in Section 2.4 Device-Specific Clauses.

## 638 2.5.2. Kernels Construct

639 **Summary** This construct defines a region of the program that is to be compiled into a sequence  
 640 of kernels for execution on the current accelerator device.

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

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

642 and in Fortran, the syntax is

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

643 where *clause* is one of the following:

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

644 **Description** The compiler will split the code in the kernels region into a sequence of acceler-  
645 ator kernels. Typically, each loop nest will be a distinct kernel. When the program encounters a  
646 **kernels** construct, it will launch the sequence of kernels in order on the device. The number and  
647 configuration of gangs of workers and vector length may be different for each kernel.

648 If the **async** clause is not present, there is an implicit barrier at the end of the kernels region, and  
649 the local thread execution will not proceed until all kernels have completed execution.

650 If there is no **default (none)** clause on the construct, the compiler will implicitly determine data  
651 attributes for variables that are referenced in the compute construct that do not have predetermined  
652 data attributes and do not appear in a data clause on the compute construct, a lexically containing  
653 **data** construct, or a visible **declare** directive. If there is no **default (present)** clause on  
654 the construct, an array or variable of aggregate data type referenced in the **kernels** construct that  
655 does not appear in a data clause for the construct or any enclosing **data** construct will be treated  
656 as if it appeared in a **copy** clause for the **kernels** construct. If there is a **default (present)**  
657 clause on the construct, the compiler will implicitly treat all arrays and variables of aggregate data  
658 type without predetermined data attributes as if they appeared in a **present** clause. A scalar  
659 variable referenced in the **kernels** construct that does not appear in a data clause for the construct  
660 or any enclosing **data** construct will be treated as if it appeared in a **copy** clause.

661 **Restrictions**

- 662 • A program may not branch into or out of an OpenACC **kernels** construct.
- 663 • A program must not depend on the order of evaluation of the clauses, or on any side effects  
664 of the evaluations.
- 665 • Only the **async**, **wait**, **num\_gangs**, **num\_workers**, and **vector\_length** clauses  
666 may follow a **device\_type** clause.
- 667 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical  
668 value; in C or C++, the condition must evaluate to a scalar integer value.
- 669 • At most one **default** clause may appear, and it must have a value of either **none** or  
670 **present**.

671 The **copy**, **copyin**, **copyout**, **create**, **no\_create**, **present**, **deviceptr**, and **attach**  
672 data clauses are described in Section 2.7 Data Clauses. The **device\_type** clause is described in  
673 Section 2.4 Device-Specific Clauses.

674 **2.5.3. Serial Construct**

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

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

676 and in Fortran, the syntax is

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

677 where *clause* is one of the following:

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

678 **Description** When the program encounters an accelerator **serial** construct, one gang of one  
679 worker with a vector length of one is created to execute the accelerator serial region sequentially.  
680 The single gang begins executing the code in the structured block in gang-redundant mode, even  
681 though there is a single gang. The **serial** construct executes as if it were a **parallel** construct  
682 with clauses **num\_gangs(1) num\_workers(1) vector\_length(1)**.

683 If the **async** clause is not present, there is an implicit barrier at the end of the accelerator serial  
684 region, and the execution of the local thread will not proceed until the gang has reached the end of  
685 the serial region.

686 If there is no **default (none)** clause on the construct, the compiler will implicitly determine data  
687 attributes for variables that are referenced in the compute construct that do not have predetermined  
688 data attributes and do not appear in a data clause on the compute construct, a lexically containing  
689 **data** construct, or a visible **declare** directive. If there is no **default (present)** clause on  
690 the construct, an array or variable of aggregate data type referenced in the **serial** construct that  
691 does not appear in a data clause for the construct or any enclosing **data** construct will be treated  
692 as if it appeared in a **copy** clause for the **serial** construct. If there is a **default (present)**  
693 clause on the construct, the compiler will implicitly treat all arrays and variables of aggregate data  
694 type without predetermined data attributes as if they appeared in a **present** clause. A scalar  
695 variable referenced in the **serial** construct that does not appear in a data clause for the construct  
696 or any enclosing **data** construct will be treated as if it appeared in a **firstprivate** clause.

#### 697 **Restrictions**

- 698 • A program may not branch into or out of an OpenACC **serial** construct.
- 699 • A program must not depend on the order of evaluation of the clauses, or on any side effects  
700 of the evaluations.
- 701 • Only the **async** and **wait** clauses may follow a **device\_type** clause.
- 702 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical  
703 value; in C or C++, the condition must evaluate to a scalar integer value.
- 704 • At most one **default** clause may appear, and it must have a value of either **none** or  
705 **present**.

706 The **copy**, **copyin**, **copyout**, **create**, **no\_create**, **present**, **deviceptr**, and **attach**  
707 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**  
708 clauses are described in Sections 2.5.10 and Sections 2.5.11. The **device\_type** clause is de-  
709 scribed in Section 2.4 Device-Specific Clauses.

#### 710 **2.5.4. if clause**

711 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to execute  
712 the region on the current accelerator device.

713 When an **if** clause appears, the compiler will generate two copies of the construct, one copy to  
714 execute on the accelerator and one copy to execute on the encountering local thread. When the  
715 *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the accelerator copy will be  
716 executed. When the *condition* in the **if** clause evaluates to zero in C or C++, or **.false.** in  
717 Fortran, the encountering local thread will execute the construct.



### 718 2.5.5. **async** clause

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

### 720 2.5.6. **wait** clause

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

### 722 2.5.7. **num\_gangs** clause

723 The **num\_gangs** clause is allowed **on the `parallel` and `kernels` constructs**. The value of  
724 the integer expression defines the number of parallel gangs that will execute the parallel region,  
725 or that will execute each kernel created for the kernels region. If the clause is not specified, an  
726 implementation-defined default will be used; the default may depend on the code within the con-  
727 struct. The implementation may use a lower value than specified based on limitations imposed by  
728 the target architecture.

### 729 2.5.8. **num\_workers** clause

730 The **num\_workers** clause is allowed **on the `parallel` and `kernels` constructs**. The value  
731 of the integer expression defines the number of workers within each gang that will be active af-  
732 ter a gang transitions from worker-single mode to worker-partitioned mode. If the clause is not  
733 specified, an implementation-defined default will be used; the default value may be 1, and may be  
734 different for each **`parallel`** construct or for each kernel created for a **`kernels`** construct. The  
735 implementation may use a different value than specified based on limitations imposed by the target  
736 architecture.

### 737 2.5.9. **vector\_length** clause

738 The **vector\_length** clause is allowed **on the `parallel` and `kernels` constructs**. The value  
739 of the integer expression defines the number of vector lanes that will be active after a worker transi-  
740 tions from vector-single mode to vector-partitioned mode. This clause determines the vector length  
741 to use for vector or SIMD operations. If the clause is not specified, an implementation-defined de-  
742 fault will be used. This vector length will be used for loops annotated with the **`vector`** clause on  
743 a **`loop`** directive, as well as loops automatically vectorized by the compiler. The implementation  
744 may use a different value than specified based on limitations imposed by the target architecture.

### 745 2.5.10. **private** clause

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

### 748 **Restrictions**

- 749     • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in  
750     **private** clauses.

### 751 2.5.11. firstprivate clause

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

#### 756 Restrictions

- 757     • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in  
758     **firstprivate** clauses.

### 759 2.5.12. reduction clause

760 The **reduction** clause is allowed on the **parallel** and **serial** constructs. It specifies a  
761 reduction operator and one or more scalar variables. For each variable, a private copy is created for  
762 each parallel gang and initialized for that operator. At the end of the region, the values for each gang  
763 are combined using the reduction operator, and the result combined with the value of the original  
764 variable and stored in the original variable. The reduction result is available after the region.

765 The following table lists the operators that are valid and the initialization values; in each case, the  
766 initialization value will be cast into the variable type. For **max** and **min** reductions, the initialization  
767 values are the least representable value and the largest representable value for the variable's data  
768 type, respectively. Supported data types are the numerical data types in C (**char**, **int**, **float**,  
769 **double**, **\_Complex**), C++ (**char**, **wchar\_t**, **int**, **float**, **double**), and Fortran (**integer**,  
770 **real**, **double precision**, **complex**).

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>

#### 772 Restrictions

- 773 • The reduction variable may not be an array element.
- 774 • The reduction variable may not be a C struct member, C++ class or struct member, or Fortran  
775 derived type member.
- 776 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in  
777 **reduction** clauses.

### 778 2.5.13. default clause

779 The **default** clause is optional. The **none** argument tells the compiler to require that all arrays or  
780 variables used in the compute construct that do not have predetermined data attributes to explicitly  
781 appear in a data clause on the compute construct, a **data** construct that lexically contains the  
782 compute construct, or a visible **declare** directive. The **present** argument causes all arrays or  
783 variables of aggregate data type used in the compute construct that have implicitly determined data  
784 attributes to be treated as if they appeared in a **present** clause.

## 785 2.6. Data Environment

786 This section describes the data attributes for variables. The data attributes for a variable may be  
787 *predetermined*, *implicitly determined*, or *explicitly determined*. Variables with predetermined data  
788 attributes may not appear in a data clause that conflicts with that data attribute. Variables with  
789 implicitly determined data attributes may appear in a data clause that overrides the implicit attribute.  
790 Variables with explicitly determined data attributes are those which appear in a data clause on a  
791 **data** construct, a compute construct, or a **declare** directive.

792 OpenACC supports systems with accelerators that have distinct memory from the host as well as  
793 systems with accelerators that share memory with the host. In the former case, called a non-shared  
794 memory device, the system has separate host memory and device memory. In the latter case, called  
795 a shared memory device as the accelerator shares memory with the host thread, the system has  
796 one shared memory. When a nested OpenACC construct is executed on the device, the default  
797 target device for that construct is the same device on which the encountering accelerator thread is  
798 executing. In that case, the target device shares memory with the encountering thread.

### 799 2.6.1. Variables with Predetermined Data Attributes

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

804 Variables declared in a C block that is executed in *vector-partitioned* mode are private to the thread  
805 associated with each vector lane. Variables declared in a C block that is executed in *worker-*  
806 *partitioned vector-single* mode are private to the worker and shared across the threads associated  
807 with the vector lanes of that worker. Variables declared in a C block that is executed in *worker-*  
808 *single* mode are private to the gang and shared across the threads associated with the workers and  
809 vector lanes of that gang.

810 A procedure called from a compute construct will be annotated as **seq**, **vector**, **worker**, or  
 811 **gang**, as described Section 2.15 Procedure Calls in Compute Regions. Variables declared in **seq**  
 812 routine are private to the thread that made the call. Variables declared in **vector** routine are private  
 813 to the worker that made the call and shared across the threads associated with the vector lanes of  
 814 that worker. Variables declared in **worker** or **gang** routine are private to the gang that made the  
 815 call and shared across the threads associated with the workers and vector lanes of that gang.

## 816 2.6.2. Data Regions and Data Lifetimes

817 For a shared-memory device, data is accessible to the local thread and to the accelerator. Such data  
 818 is available to the accelerator for the lifetime of the variable. For a non-shared memory device,  
 819 data in host memory is allocated in device memory and copied between host and device memory by  
 820 using data constructs, clauses, and API routines. A *data lifetime* is the duration from when the data  
 821 is first made available to the accelerator until it becomes unavailable, after having been deallocated  
 822 from device memory, for instance.

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

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

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

832 In addition to data regions, a program may create and delete data on the accelerator using **enter**  
 833 **data** and **exit data** directives or using runtime API routines. When the program executes  
 834 an **enter data** directive, or executes a call to a runtime API **acc\_copyin** or **acc\_create**  
 835 routine, each variable, array, or subarray on the directive or the variable on the runtime API argument  
 836 list will be made live on accelerator.

## 837 2.6.3. Data Construct

838 **Summary** The **data** construct defines scalars, arrays, and subarrays to be allocated in the cur-  
 839 rent device memory for the duration of the region, whether data should be copied from the host to  
 840 the device memory upon region entry, and copied from the device to host memory upon region exit.

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

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

842 and in Fortran, the syntax is

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

```

    structured block
    !$acc end data

```

843 where *clause* is one of the following:

```

    if( condition )
    copy( var-list )
    copyin( var-list )
    copyout( var-list )
    create( var-list )
    no_create( var-list )
    present( var-list )
    deviceptr( var-list )
    attach( var-list )

```

844 **Description** Data will be allocated in the memory of the current device and copied from the  
 845 host or local memory to the device, or copied back, as required. The data clauses are described in  
 846 Section 2.7 Data Clauses. Structured reference counts are incremented for data when entering a data  
 847 region, and decremented when leaving the region, as described in Section 2.6.5 Reference Counting.

848 **if clause**

849 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate  
 850 memory on the current accelerator device and move data from and to the local memory as required.  
 851 When an **if** clause appears, the program will conditionally allocate memory on, and move data  
 852 to and/or from the device. When the *condition* in the **if** clause evaluates to zero in C or C++, or  
 853 **.false.** in Fortran, no device memory will be allocated, and no data will be moved. When the  
 854 *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the data will be allocated and  
 855 moved as specified. At most one **if** clause may appear.

#### 856 2.6.4. Enter Data and Exit Data Directives

857 **Summary** An **enter data** directive may be used to define scalars, arrays and subarrays to be  
 858 allocated in the current device memory for the remaining duration of the program, or until an **exit**  
 859 **data** directive that deallocates the data. They also tell whether data should be copied from the host  
 860 to the device memory at the **enter data** directive, and copied from the device to host memory at  
 861 the **exit data** directive. The dynamic range of the program between the **enter data** directive  
 862 and the matching **exit data** directive is the data lifetime for that data.

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

```

    #pragma acc enter data clause-list new-line

```

864 and in Fortran, the syntax is

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

865 where *clause* is one of the following:

```
if( condition )
async [( int-expr )]
wait [( int-expr-list )]
copyin( var-list )
create( var-list )
attach( var-list )
```

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

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

867 and in Fortran, the syntax is

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

868 where *clause* is one of the following:

```
if( condition )
async [( int-expr )]
wait [( int-expr-list )]
copyout( var-list )
delete( var-list )
detach( var-list )
finalize
```

869 **Description** At an **enter data** directive, data may be allocated in the current device memory  
870 and copied from the host or local memory to the device. This action enters a data lifetime for those  
871 variables, arrays, or subarrays, and will make the data available for **present** clauses on constructs  
872 within the data lifetime. Dynamic reference counts are incremented for this data, as described in  
873 Section 2.6.5 Reference Counting. Pointers in device memory may be *attached* to point to the  
874 corresponding device copy of the host pointer target.

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

880 The data clauses are described in Section 2.7 Data Clauses. Reference counting behavior is de-  
881 scribed in Section 2.6.5 Reference Counting.

**882 if clause**

883 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or  
884 deallocate memory on the current accelerator device and move data from and to the local memory.  
885 When an **if** clause appears, the program will conditionally allocate or deallocate device memory  
886 and move data to and/or from the device. When the *condition* in the **if** clause evaluates to zero in  
887 C or C++, or **.false.** in Fortran, no device memory will be allocated or deallocated, and no data  
888 will be moved. When the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the  
889 data will be allocated or deallocated and moved as specified.

**890 async clause**

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

**892 wait clause**

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

**894 finalize clause**

895 The **finalize** clause is allowed on the **exit data** directive and is optional. When no **finalize**  
896 clause appears, the **exit data** directive will decrement the dynamic reference counts for variables  
897 and arrays appearing in **copyout** and **delete** clauses, and will decrement the attachment counts  
898 for pointers appearing in **detach** clauses. If a **finalize** clause appears, the **exit data** direc-  
899 tive will set the dynamic reference counts to zero for variables and arrays appearing in **copyout**  
900 and **delete** clauses, and will set the attachment counts to zero for pointers appearing in **detach**  
901 clauses.

**902 2.6.5. Reference Counting**

903 When data is allocated on a non-shared memory device due to data clauses or OpenACC API routine  
904 calls, the OpenACC implementation keeps track of that device memory and its relationship to the  
905 corresponding data in host memory. Each section of device memory will be associated with two  
906 *reference counts*. A structured reference count is incremented when entering each data or compute  
907 region that contain an explicit data clause or implicitly-determined data attributes for that block of  
908 memory, and is decremented when exiting that region. A dynamic reference count is incremented  
909 for each **enter data copyin** or **create** clause, or each **acc\_copyin** or **acc\_create** API  
910 routine call for that block of memory. The dynamic reference count is decremented for each **exit**  
911 **data copyout** or **delete** clause when no **finalize** clause appears, or each **acc\_copyout**  
912 or **acc\_delete** API routine call for that block of memory. The dynamic reference count will be  
913 set to zero with an **exit data copyout** or **delete** clause when a **finalize** clause appears,  
914 or each **acc\_copyout\_finalize** or **acc\_delete\_finalize** API routine call for the block  
915 of memory. The reference counts are modified synchronously with the encountering thread, even if  
916 the data directives include an **async** clause. When both structured and dynamic reference counts  
917 reach zero, the data lifetime for that data ends.

### 918 2.6.6. Attachment Counters

919 This section describes the behavior when attaching pointer variables in device memory to pointer  
920 targets in device memory. A pointer may be a C or C++ pointer (e.g., `float*`), a Fortran pointer  
921 or array pointer (e.g., `real, pointer, dimension(:)`), or a Fortran allocatable (e.g., `real,`  
922 `allocatable, dimension(:)`).

923 When data is copied from host memory to device memory, the values are copied exactly. If the data  
924 is a pointer or a data structure that includes a pointer, the pointer value copied to device memory  
925 will be the host pointer value. If the pointer target is also allocated on or copied to the device, an  
926 *attach* action updates the pointer in device memory to point to the device copy of the data that the  
927 host pointer targets. For Fortran array pointers and allocatable arrays, this includes copying any  
928 associated descriptor (*dope vector*) to the device copy of the pointer. When the device pointer target  
929 is deallocated, the pointer in device memory should be restored to the host value, so it can be safely  
930 copied back to host memory. A *detach* action updates the pointer in device memory to have the  
931 same value as the corresponding pointer in local memory.

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

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

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

## 948 2.7. Data Clauses

949 These data clauses may appear on the `parallel` construct, `kernels` construct, `serial con-`  
950 `struct`, `data` construct, the `enter data` and `exit data` directives, and `declare` directives.  
951 In the descriptions, the *region* is a compute region with a clause appearing on a `parallel`,  
952 `kernels`, or `serial` construct, a data region with a clause on a `data` construct, or an implicit  
953 data region with a clause on a `declare` directive. If the `declare` directive appears in a global  
954 context, the corresponding implicit data region has a duration of the program. The list argument  
955 to each data clause is a comma-separated collection of variable names, array names, or subarray  
956 specifications. For all clauses except `deviceptr` and `present`, the list argument may include a  
957 Fortran *common block* name enclosed within slashes, if that *common block* name also appears in a  
958 `declare` directive `link` clause. In all cases, the compiler will allocate and manage a copy of the  
959 variable or array in the memory of the current device, creating a visible device copy of that variable



960 or array, for non-shared memory devices.

961 OpenACC supports accelerators with physically and logically separate memories from the local  
 962 thread. However, if the accelerator can access the local memory directly, the implementation may  
 963 avoid the memory allocation and data movement and simply share the data in local memory. There-  
 964 fore, a program that uses and assigns data on the host and uses and assigns the same data on the  
 965 accelerator within a data region without update directives to manage the coherence of the two copies  
 966 may get different answers on different accelerators or implementations.

## 967 Restrictions

- 968 • Data clauses may not follow a `device_type` clause.
- 969 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in  
 970 data clauses.

### 971 2.7.1. Data Specification in Data Clauses

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

**AA[2:n]**

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

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

- 978 • Statically-sized array: `float AA[100][200];`
- 979 • Pointer to statically sized rows: `typedef float row[200]; row* BB;`
- 980 • Statically-sized array of pointers: `float* CC[200];`
- 981 • Pointer to pointers: `float** DD;`

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

- 984 • **AA[2:n][0:200]**
- 985 • **BB[2:n][0:m]**
- 986 • **CC[2:n][0:m]**
- 987 • **DD[2:n][0:m]**

988 Multidimensional rectangular subarrays in C and C++ may be specified for any array with any com-  
 989 bination of statically-sized or dynamically-allocated dimensions. For statically sized dimensions,  
 990 all dimensions except the first must specify the whole **extent**, to preserve the contiguous data re-  
 991 striction, discussed below. For dynamically allocated dimensions, the implementation will allocate  
 992 pointers on the device corresponding to the pointers on the host, and will fill in those pointers as  
 993 appropriate.

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

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

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

## 999 Restrictions

- 1000 • In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be  
 1001 specified.
- 1002 • In C and C++, the length for dynamically allocated dimensions of an array must be explicitly  
 1003 specified.
- 1004 • In C and C++, modifying pointers in pointer arrays during the data lifetime, either on the host  
 1005 or on the device, may result in undefined behavior.
- 1006 • If a subarray is specified in a data clause, the implementation may choose to allocate memory  
 1007 for only that subarray on the accelerator.
- 1008 • In Fortran, array pointers may be specified, but pointer association is not preserved in the  
 1009 device memory.
- 1010 • Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous  
 1011 block of memory, except for dynamic multidimensional C arrays.
- 1012 • In C and C++, if a variable or array of struct or class type is specified, all the data members  
 1013 of the struct or class are allocated and copied, as appropriate. If a struct or class member is a  
 1014 pointer type, the data addressed by that pointer are not implicitly copied.
- 1015 • In Fortran, if a variable or array with derived type is specified, all the members of that derived  
 1016 type are allocated and copied, as appropriate. If any member has the **allocatable** or  
 1017 **pointer** attribute, the data accessed through that member are not copied.
- 1018 • If an expression is used in a subscript or subarray expression in a clause on a **data** construct,  
 1019 the same value is used when copying data at the end of the data region, even if the values of  
 1020 variables in the expression change during the data region.

## 1021 2.7.2. Data Clause Actions

1022 Most of the data clauses perform one or more the following actions. The actions test or modify one  
 1023 or both of the structured and dynamic reference counts, depending on the directive on which the  
 1024 data clause appears.

### 1025 Present Increment Action

1026 A *present increment* action is one of the actions that may be performed for a **present** (Section  
 1027 2.7.4), **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Sec-

1028 tion 2.7.8), or **no\_create** (Section 2.7.9) clause, or for a call to an **acc\_copyin** (Section 3.2.20)  
1029 or **acc\_create** (Section 3.2.21) API routine. See those sections for details.

1030 A *present increment* action for a *var* occurs only when *var* is already present on the device.

1031 A *present increment* action for a *var* increments the structured or dynamic reference count for *var*.

### 1032 Present Decrement Action

1033 A *present decrement* action is one of the actions that may be performed for a **present** (Section  
1034 2.7.4), **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Sec-  
1035 tion 2.7.8), **no\_create** (Section 2.7.9), or **delete** (Section 2.7.10) clause, or for a call to an  
1036 **acc\_copyout** (Section 3.2.22) or **acc\_delete** (Section 3.2.23) API routine. See those sec-  
1037 tions for details.

1038 A *present decrement* action for a *var* occurs only when *var* is already present on the device.

1039 A *present decrement* action for a *var* decrements the structured or dynamic reference count for *var*,  
1040 if its value is greater than zero. If the reference count is already zero, its value is left unchanged.

### 1041 Create Action

1042 A *create* action is one of the actions that may be performed for a **copyout** (Section 2.7.7) or  
1043 **create** (Section 2.7.8) clause, or for a call to an **acc\_create** API routine (Section 3.2.21). See  
1044 those sections for details.

1045 A *create* action for a *var* occurs only when *var* is not already present on the device.

1046 A *create* action for a *var*:

- 1047 • allocates device memory for *var*; and
- 1048 • sets the structured or dynamic reference count to one.

### 1049 Copyin Action

1050 A *copyin* action is one of the actions that may be performed for a **copy** (Section 2.7.5) or **copyin**  
1051 (Section 2.7.6) clause, or for a call to an **acc\_copyin** API routine (Section 3.2.20). See those  
1052 sections for details.

1053 A *copyin* action for a *var* occurs only when *var* is not already present on the device.

1054 A *copyin* action for a *var*:

- 1055 • allocates device memory for *var*;
- 1056 • initiates a copy of the data for *var* from the local thread memory to the corresponding device  
1057 memory; and
- 1058 • sets the structured or dynamic reference count to one.

1059 The data copy may complete asynchronously, depending on other clauses on the directive.

## 1060 Copyout Action

1061 A *copyout* action is one of the actions that may be performed for a **copy** (Section 2.7.5) or  
1062 **copyout** (Section 2.7.7) clause, or for a call to an **acc\_copyout** API routine (Section 3.2.22).  
1063 See those sections for details.

1064 A *copyout* action for a *var* occurs only when *var* is present on the device.

1065 A *copyout* action for a *var*:

- 1066 • performs an *immediate detach* action for any pointer in *var*;
- 1067 • initiates a copy of the data for *var* from the device memory to the corresponding local thread  
1068 memory; and
- 1069 • deallocates the device memory for *var*.

1070 The data copy may complete asynchronously, depending on other clauses on the directive, in which  
1071 case the memory is deallocated when the data copy is complete.

## 1072 Delete Action

1073 A *present decrement* action is one of the actions that may be performed for a **present** (Section  
1074 2.7.4), **copyin** (Section 2.7.6), **create** (Section 2.7.8), **no\_create** (Section 2.7.9), or **delete**  
1075 (Section 2.7.10) clause, or for a call to an **acc\_delete** API routine (Section 3.2.23). See those  
1076 sections for details.

1077 A *delete* action for a *var* occurs only when *var* is present on the device.

1078 A *delete* action for *var*:

- 1079 • performs an *immediate detach* action for any pointer in *var*; and
- 1080 • deallocates device memory for *var*.

## 1081 Attach Action

1082 An *attach* action is one of the actions that may be performed for a **present** (Section 2.7.4),  
1083 **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Section 2.7.8),  
1084 **no\_create** (Section 2.7.9), or **attach** (Section 2.7.10) clause, or for a call to an **acc\_attach**  
1085 API routine (Section 3.2.34). See those sections for details.

1086 An *attach* action for a *var* occurs only when *var* is a pointer reference.

1087 If the current device is a shared memory device or if the pointer *var* is not present on the device, or  
1088 if the address to which *var* points is not present on the device, no action is taken. If the *attachment*  
1089 *counter* for *var* is nonzero and the pointer in device memory already points to the device copy of  
1090 the data in *var*, the *attachment counter* for the pointer *var* is incremented. Otherwise, the pointer  
1091 in device memory is *attached* to the device copy of the data by initiating an update for the pointer  
1092 in device memory to point to the device copy of the data and setting the *attachment counter* for the  
1093 pointer *var* to one. The update may complete asynchronously, depending on other clauses on the  
1094 directive. The pointer update must follow any data copies due to *copyin* actions that are performed  
1095 for the same directive.

## 1096 **Detach Action**

1097 A *detach* action is one of the actions that may be performed for a **present** (Section 2.7.4),  
1098 **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Section 2.7.8),  
1099 **no\_create** (Section 2.7.9), **delete** (Section 2.7.10), or **detach** (Section 2.7.10) clause, or for  
1100 a call to an **acc\_detach** API routine (Section 3.2.35). See those sections for details.

1101 A *detach* action for a *var* occurs only when when *var* is a pointer reference.

1102 If the current device is a shared memory device, or if *var* is not present on the device, or if the  
1103 *attachment counter* for *var* for the pointer is zero, no action is taken. Otherwise, the *attachment*  
1104 *counter* for the pointer *var* is decremented. If the *attachment counter* is decreased to zero, the  
1105 pointer is *detached* by initiating an update for the pointer *var* in device memory to have the same  
1106 value as the corresponding pointer in local memory. The update may complete asynchronously,  
1107 depending on other clauses on the directive. The pointer update must precede any data copies due  
1108 to *copyout* actions that are performed for the same directive.

## 1109 **Immediate Detach Action**

1110 An *immediate detach* action is one of the actions that may be performed for a **detach** (Section  
1111 2.7.10) clause, or for a call to an **acc\_detach\_finalize** API routine (Section 3.2.35). See  
1112 those sections for details.

1113 An *immediate detach* action for a *var* occurs only when when *var* is a pointer reference and is  
1114 present on the device.

1115 If the *attachment counter* for the pointer is zero, the *immediate detach* action has no effect. Other-  
1116 wise, the *attachment counter* for the pointer set to zero and the pointer is *detached* by initiating an  
1117 update for the pointer in device memory to have the same value as the corresponding pointer in local  
1118 memory. The update may complete asynchronously, depending on other clauses on the directive.  
1119 The pointer update must precede any data copies due to *copyout* actions that are performed for the  
1120 same directive.

## 1121 **2.7.3. deviceptr clause**

1122 The **deviceptr** clause may appear on structured **data** and compute constructs and **declare**  
1123 directives.

1124 The **deviceptr** clause is used to declare that the pointers in *var-list* are device pointers, so the  
1125 data need not be allocated or moved between the host and device for this pointer.

1126 In C and C++, the variables in *var-list* must be pointer variables.

1127 In Fortran, the variables in *var-list* must be dummy arguments (arrays or scalars), and may not have  
1128 the Fortran **pointer**, **allocatable**, or **value** attributes.

1129 For a shared-memory device, host pointers are the same as device pointers, so this clause has no  
1130 effect.

#### 1131 2.7.4. present clause

1132 The **present** clause may appear on structured **data** and compute constructs and **declare** di-  
1133 rectives. If the current device is a shared memory device, no action is taken.

1134 For a non-shared memory **device**, the **present** clause specifies that variables or arrays in *var-list*  
1135 are already present in device memory on the current device due to data regions or data lifetimes that  
1136 contain the construct on which the **present** clause appears.

1137 If the current device is a non-shared memory device, the **present** clause behaves as follows, for  
1138 each *var* in *var-list*.

1139 • At entry to the region:

- 1140 – If *var* is not present on the current device, a runtime error is issued.
- 1141 – Otherwise, a *present increment* action with the structured reference count is performed.
- 1142 If *var* is a pointer reference, an *attach* action is performed.

1143 • At exit from the region:

- 1144 – If *var* is not present on the current device, a runtime error is issued.
- 1145 – Otherwise, a *present decrement* action with the structured reference count is performed.
- 1146 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
1147 reference counts are zero, a *delete* action is performed.

#### 1148 Restrictions

- 1149 • If only a subarray of an array is present on the current device, the **present** clause must  
1150 specify the same subarray, or a subarray that is a proper subset of the subarray in the data  
1151 lifetime.
- 1152 • It is a runtime error if the subarray in *var-list* clause includes array elements that are not part  
1153 of the subarray specified in the data lifetime.

#### 1154 2.7.5. copy clause

1155 The **copy** clause may appear on structured **data** and compute constructs and on **declare** direc-  
1156 tives. If the current device is a shared memory device, no action is taken.

1157 If the current device is a non-shared memory device, the **copy** clause behaves as follows, for each  
1158 *var* in *var-list*.

1159 • At entry to the region:

- 1160 – If *var* is present, a *present increment* action with the structured reference count is per-  
1161 formed. If *var* is a pointer reference, an *attach* action is performed.
- 1162 – Otherwise, a *copyin* action with the structured reference count is performed. If *var* is a  
1163 pointer reference, an *attach* action is performed.

1164 • At exit from the region:

- 1165 – If *var* is not present on the current device, a runtime error is issued.

- 1166 – Otherwise, a *present decrement* action with the structured reference count is performed.  
 1167 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
 1168 reference counts are zero, a *copyout* action is performed.

1169 The restrictions regarding subarrays in the **present** clause apply to this clause.

1170 For compatibility with OpenACC 2.0, **present\_or\_copy** and **pcopy** are alternate names for  
 1171 **copy**.

## 1172 2.7.6. copyin clause

1173 The **copyin** clause may appear on structured **data** and compute constructs, on **declare** direc-  
 1174 tives, and on **enter data** directives. If the current device is a shared memory device, no action is  
 1175 taken.

1176 If the current device is a non-shared memory device, the **copyin** clause behaves as follows, for  
 1177 each *var* in *var-list*:

- 1178 • At entry to a region, the structured reference count is used. On an **enter data** directive,  
 1179 the dynamic reference count is used.
  - 1180 – If *var* is present, a *present increment* action with the appropriate reference count is per-  
 1181 formed. If *var* is a pointer reference, an *attach* action is performed.
  - 1182 – Otherwise, a *copyin* action with the appropriate reference count is performed. If *var* is  
 1183 a pointer reference, an *attach* action is performed.
- 1184 • At exit from the region:
  - 1185 – If *var* is not present on the current device, a runtime error is issued.
  - 1186 – Otherwise, a *present decrement* action with the structured reference count is performed.  
 1187 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
 1188 reference counts are zero, a *delete* action is performed.

1189 The restrictions regarding subarrays in the **present** clause apply to this clause.

1190 For compatibility with OpenACC 2.0, **present\_or\_copyin** and **pcopyin** are alternate names  
 1191 for **copyin**.

1192 An **enter data** directive with a **copyin** clause is functionally equivalent to a call to the **acc\_copyin**  
 1193 API routine, as described in Section 3.2.20.

## 1194 2.7.7. copyout clause

1195 The **copyout** clause may appear on structured **data** and compute constructs, on **declare** di-  
 1196 rectives, and on **exit data** directives. If the current device is a shared memory device, no action  
 1197 is performed.

1198 If the current device is a non-shared memory device, the **copyout** clause behaves as follows, for  
 1199 each *var* in *var-list*:

- 1200 • At entry to a region:
  - 1201 – If *var* is present, a *present increment* action with the structured reference count is per-

1202           formed. If *var* is a pointer reference, an *attach* action is performed.

1203           – Otherwise, a *create* action with the structured reference is performed. If *var* is a pointer  
1204           reference, an *attach* action is performed.

1205           • At exit from a region, the structured reference count is used. On an **exit data** directive,  
1206           the dynamic reference count is used.

1207           – If *var* is not present on the current device, a runtime error is issued.

1208           – Otherwise, the reference count is updated:

1209               \* On an **exit data** directive with a **finalize** clause, the dynamic reference  
1210               count is set to zero.

1211               \* Otherwise, a *present decrement* action with the appropriate reference count is per-  
1212               formed.

1213           If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
1214           reference counts are zero, a *copyout* action is performed.

1215 The restrictions regarding subarrays in the **present** clause apply to this clause.

1216 For compatibility with OpenACC 2.0, **present\_or\_copyout** and **pcopyout** are alternate  
1217 names for **copyout**.

1218 An **exit data** directive with a **copyout** clause and with or without a **finalize** clause is func-  
1219 tionally equivalent to a call to the **acc\_copyout\_finalize** or **acc\_copyout** API routine,  
1220 respectively, as described in Section 3.2.22.

### 1221 2.7.8. create clause

1222 The **create** clause may appear on structured **data** and compute constructs, on **declare** direc-  
1223 tives, and on **enter data** directives. If the current device is a shared memory device, no action is  
1224 taken.

1225 If the current device is a non-shared memory device, the **create** clause behaves as follows, for  
1226 each *var* in *var-list*:

1227           • At entry to a region, the structured reference count is used. On an **enter data** directive,  
1228           the dynamic reference count is used.

1229           – If *var* is present, a *present increment* action with the appropriate reference count is per-  
1230           formed. If *var* is a pointer reference, an *attach* action is performed.

1231           – Otherwise, a *create* action with the appropriate reference count is performed. If *var* is a  
1232           pointer reference, an *attach* action is performed.

1233           • At exit from the region:

1234           – If *var* is not present on the current device, a runtime error is issued.

1235           – Otherwise, a *present decrement* action with the structured reference count is performed.  
1236           If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
1237           reference counts are zero, a *delete* action is performed.

1238 The restrictions regarding subarrays in the **present** clause apply to this clause.



1239 For compatibility with OpenACC 2.0, **present\_or\_create** and **pcreate** are alternate names  
1240 for **create**.

1241 An **enter data** directive with a **create** clause is functionally equivalent to a call to the **acc\_create**  
1242 API routine, as described in Section 3.2.21.

### 1243 2.7.9. no\_create clause

1244 The **no\_create** clause may appear on structured **data** and compute constructs. If the current  
1245 device is a shared memory device, no action is taken.

1246 If the current device is a non-shared memory device, the **no\_create** clause behaves as follows,  
1247 for each *var* in *var-list*:

- 1248 • At entry to the region:
  - 1249 – If *var* is present, a *present increment* action with the structured reference count is per-  
1250 formed. If *var* is a pointer reference, an *attach* action is performed.
  - 1251 – Otherwise, no action is performed, and any device code in this construct will use the  
1252 local memory address for *var*.
- 1253 • At exit from the region:
  - 1254 – If *var* is not present on the current device, no action is performed.
  - 1255 – Otherwise, a *present decrement* action with the structured reference count is performed.  
1256 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
1257 reference counts are zero, a *delete* action is performed.

1258 The restrictions regarding subarrays in the **present** clause apply to this clause.

### 1259 2.7.10. delete clause

1260 The **delete** clause may appear on **exit data** directives. If the current device is a shared memory  
1261 device, no action is taken.

1262 If the current device is a non-shared memory device, the **delete** clause behaves as follows, for  
1263 each *var* in *var-list*:

- 1264 • If *var* is not present on the current device, a runtime error is issued.
- 1265 • Otherwise, the dynamic reference count is updated:
  - 1266 – On an **exit data** directive with a **finalize** clause, the dynamic reference count is  
1267 set to zero.
  - 1268 – Otherwise, a *present decrement* action with the dynamic reference count is performed.  
1269 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
1270 reference counts are zero, a *delete* action is performed.

1271 An **exit data** directive with a **delete** clause and with or without a **finalize** clause is func-  
1272 tionally equivalent to a call to the **acc\_delete\_finalize** or **acc\_delete** API routine, re-  
1273 spectively, as described in Section 3.2.23.

1274 **2.7.11. attach clause**

1275 The **attach** clause may appear on structured **data** and compute constructs and on **enter data**  
 1276 directives. If the current device is a shared memory device, no action is taken. Each *var* argument  
 1277 to an **attach** clause must be a C or C++ pointer or a Fortran variable or array with the **pointer**  
 1278 or **allocatable** attribute.

1279 If the current device is a non-shared memory device, the **attach** clause behaves as follows, for  
 1280 each *var* in *var-list*:

- 1281 • At entry to a region or at an **enter data** directive, an *attach* action is performed.
- 1282 • At exit from the region, a *detach* action is performed.

1283 **2.7.12. detach clause**

1284 The **detach** clause may appear on **exit data** directives. If the current device is a shared memory  
 1285 device, no action is taken. Each *var* argument to a **detach** clause must be a C or C++ pointer or a  
 1286 Fortran variable or array with the **pointer** or **allocatable** attribute.

1287 If the current device is a non-shared memory device, the **detach** clause behaves as follows, for  
 1288 each *var* in *var-list*: If the current device is a non-shared memory device,

- 1289 • If there is a **finalize** clause on the **exit data** directive, an *immediate detach* action is  
 1290 performed.
- 1291 • Otherwise, a *detach* action is performed.

1292 **2.8. Host\_Data Construct**

1293 **Summary** The **host\_data** construct makes the address of device data available on the host.

1294 **Syntax** In C and C++, the syntax of the OpenACC **host\_data** construct is

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

1295 and in Fortran, the syntax is

```
!$acc host_data clause-list
    structured block
!$acc end host_data
```

1296 where *clause* is one of the following:

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

1297 **Description** This construct is used to make the device address of data available in host code.

### 1298 **Restrictions**

- 1299 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
- 1300 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1301 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
- 1302 **use\_device** clauses.

### 1303 **2.8.1. use\_device clause**

1304 The **use\_device clause** tells the compiler to use the current device address of any variable or  
 1305 array in *var-list* in code within the construct. In particular, this may be used to pass the device  
 1306 address of variables or arrays to optimized procedures written in a lower-level API. **When there is**  
 1307 **no *if\_present* clause**, and either there is no **if** clause or the condition in the **if** clause evaluates  
 1308 to nonzero (in C or C++) or **.true.** (in Fortran), the variables or arrays in *var-list* must be present  
 1309 in the accelerator memory due to data regions or data lifetimes that contain this construct. On a  
 1310 shared memory accelerator, the device address may be the same as the host address.

### 1311 **2.8.2. if clause**

1312 The **if** clause is optional. When an **if** clause appears and the condition evaluates to zero in C or  
 1313 C++, or **.false.** in Fortran, the compiler will not change the addresses of any variable or array  
 1314 in code within the construct. When there is no **if** clause, or when an **if** clause appears and the  
 1315 condition evaluates to nonzero in C or C++, or **.true.** in Fortran, the compiler will replace the  
 1316 addresses as described in the previous subsection.

### 1317 **2.8.3. if\_present clause**

1318 When an **if\_present** clause appears on the directive, the compiler will only change the address  
 1319 of any variable or array which appears in *var-list* that is present on the current device.

## 1320 **2.9. Loop Construct**

1321 **Summary** The OpenACC **loop** construct applies to a loop which must immediately follow this  
 1322 directive. The **loop** construct can describe what type of parallelism to use to execute the loop and  
 1323 declare private variables and arrays and reduction operations.

1324 **Syntax** In C and C++, the syntax of the **loop** construct is

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

1325 In Fortran, the syntax of the **loop** construct is

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

1326 where *clause* is one of the following:

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

1327 where *gang-arg* is one of:

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

1328 and *gang-arg-list* may have at most one **num** and one **static** argument,

1329 and where *size-expr* is one of:

```
*
int-expr
```

1330 Some clauses are only valid in the context of a **kernels** construct; see the descriptions below.

1331 An *orphaned loop* construct is a **loop** construct that is not lexically enclosed within a **compute**  
 1332 construct. The parent compute construct of a **loop** construct is the nearest compute construct that  
 1333 lexically contains the **loop** construct.

### 1334 Restrictions

- 1335 • Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **auto**, and **tile** clauses may follow  
 1336 a **device\_type** clause.
- 1337 • The *int-expr* argument to the **worker** and **vector** clauses must be invariant in the kernels  
 1338 region.
- 1339 • A loop associated with a **loop** construct that does not have a **seq** clause must be written  
 1340 such that the loop iteration count is computable when entering the **loop** construct.

### 1341 2.9.1. collapse clause

1342 The **collapse** clause is used to specify how many tightly nested loops are associated with the  
1343 **loop** construct. The argument to the **collapse** clause must be a constant positive integer expres-  
1344 sion. If no **collapse** clause is present, only the immediately following loop is associated with the  
1345 **loop** construct.

1346 If more than one loop is associated with the **loop** construct, the iterations of all the associated loops  
1347 are all scheduled according to the rest of the clauses. The trip count for all loops associated with the  
1348 **collapse** clause must be computable and invariant in all the loops.

1349 It is implementation-defined whether a **gang**, **worker** or **vector** clause on the construct is ap-  
1350 plied to each loop, or to the linearized iteration space.

### 1351 2.9.2. gang clause

1352 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,  
1353 the **gang** clause specifies that the iterations of the associated loop or loops are to be executed in  
1354 parallel by distributing the iterations among the gangs created by the **parallel** construct. A  
1355 **loop** construct with the **gang** clause transitions a compute region from gang-redundant mode to  
1356 gang-partitioned mode. The number of gangs is controlled by the **parallel** construct; only the  
1357 **static** argument is allowed. The loop iterations must be data independent, except for variables  
1358 specified in a **reduction** clause. The region of a loop with the **gang** clause may not contain  
1359 another loop with the **gang** clause unless within a nested compute region.

1360 When the parent compute construct is a **kernels** construct, the **gang** clause specifies that the  
1361 iterations of the associated loop or loops are to be executed in parallel across the gangs. An argument  
1362 with no keyword or with the **num** keyword is allowed only when the **num\_gangs** does not appear  
1363 on the **kernels** construct. If an argument with no keyword or an argument after the **num** keyword  
1364 is specified, it specifies how many gangs to use to execute the iterations of this loop. The region of a  
1365 loop with the **gang** clause may not contain another loop with a **gang** clause unless within a nested  
1366 compute region.

1367 The scheduling of loop iterations to gangs is not specified unless the **static** argument appears as  
1368 an argument. If the **static** argument appears with an integer expression, that expression is used  
1369 as a *chunk* size. If the static argument appears with an asterisk, the implementation will select a  
1370 *chunk* size. The iterations are divided into chunks of the selected *chunk* size, and the chunks are  
1371 assigned to gangs starting with gang zero and continuing in round-robin fashion. Two **gang** loops  
1372 in the same parallel region with the same number of iterations, and with **static** clauses with the  
1373 same argument, will assign the iterations to gangs in the same manner. Two **gang** loops in the  
1374 same kernels region with the same number of iterations, the same number of gangs to use, and with  
1375 **static** clauses with the same argument, will assign the iterations to gangs in the same manner.

### 1376 2.9.3. worker clause

1377 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,  
1378 the **worker** clause specifies that the iterations of the associated loop or loops are to be executed  
1379 in parallel by distributing the iterations among the multiple workers within a single gang. A **loop**  
1380 construct with a **worker** clause causes a gang to transition from worker-single mode to worker-

1381 partitioned mode. In contrast to the **gang** clause, the **worker** clause first activates additional  
1382 worker-level parallelism and then distributes the loop iterations across those workers. No argu-  
1383 ment is allowed. The loop iterations must be data independent, except for variables specified in a  
1384 **reduction** clause. The region of a loop with the **worker** clause may not contain a loop with the  
1385 **gang** or **worker** clause unless within a nested compute region.

1386 When the parent compute construct is a **kernels** construct, the **worker** clause specifies that the  
1387 iterations of the associated loop or loops are to be executed in parallel across the workers within  
1388 a **single** gang. An argument is allowed only when the **num\_workers** does not appear on the  
1389 **kernels** construct. The optional argument specifies how many workers per gang to use to execute  
1390 the iterations of this loop. The region of a loop with the **worker** clause may not contain a loop  
1391 with a **gang** or **worker** clause unless within a nested compute region.

1392 All workers will complete execution of their assigned iterations before any worker proceeds beyond  
1393 the end of the loop.

#### 1394 2.9.4. vector clause

1395 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,  
1396 the **vector** clause specifies that the iterations of the associated loop or loops are to be executed in  
1397 vector or SIMD mode. A **loop** construct with a **vector** clause causes a worker to transition from  
1398 vector-single mode to vector-partitioned mode. Similar to the **worker** clause, the **vector** clause  
1399 first activates additional vector-level parallelism and then distributes the loop iterations across those  
1400 vector lanes. The operations will execute using vectors of the length specified or chosen for the  
1401 parallel region. The region of a loop with the **vector** clause may not contain a loop with the  
1402 **gang**, **worker**, or **vector** clause unless within a nested compute region.

1403 When the parent compute construct is a **kernels** construct, the **vector** clause specifies that the  
1404 iterations of the associated loop or loops are to be executed with vector or SIMD processing. An  
1405 argument is allowed only when the **vector\_length** does not appear on the **kernels** construct.  
1406 If an argument is specified, the iterations will be processed in vector strips of that length; if no  
1407 argument is specified, the implementation will choose an appropriate vector length. The region of  
1408 a loop with the **vector** clause may not contain a loop with a **gang**, **worker**, or **vector** clause  
1409 unless within a nested compute region.

1410 All vector lanes will complete execution of their assigned iterations before any vector lane proceeds  
1411 beyond the end of the loop.

#### 1412 2.9.5. seq clause

1413 The **seq** clause specifies that the associated loop or loops are to be executed sequentially by the  
1414 accelerator. This clause will override any automatic parallelization or vectorization.

#### 1415 2.9.6. auto clause

1416 The **auto** clause specifies that the implementation must analyze the loop and determine whether  
1417 the loop iterations are data independent and, if so, select whether to apply parallelism to this loop  
1418 or whether to run the loop sequentially. The implementation may be restricted to the types of

1419 parallelism it can apply by the presence of **loop** constructs with **gang**, **worker**, or **vector**  
1420 clauses for outer or inner loops. When the parent compute construct is a **kernels** construct, a  
1421 **loop** construct with no **independent** or **seq** clause is treated as if it has the **auto** clause.

### 1422 2.9.7. tile clause

1423 The **tile** clause specifies that the implementation should split each loop in the loop nest into two  
1424 loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the **tile**  
1425 clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression  
1426 or an asterisk. If there are  $n$  tile sizes in the list, the **loop** construct must be immediately followed  
1427 by  $n$  tightly-nested loops. The first argument in the *size-expr-list* corresponds to the innermost loop  
1428 of the  $n$  associated loops, and the last element corresponds to the outermost associated loop. If the  
1429 tile size is specified with an asterisk, the implementation will choose an appropriate value. Each  
1430 loop in the nest will be split or *strip-mined* into two loops, an outer *tile* loop and an inner *element*  
1431 loop. The trip count of the element loop will be limited to the corresponding tile size from the  
1432 *size-expr-list*. The *tile* loops will be reordered to be outside all the *element* loops, and the *element*  
1433 loops will all be inside the *tile* loops.

1434 If the **vector** clause appears on the **loop** construct, the **vector** clause is applied to the *element*  
1435 loops. If the **gang** clause appears on the **loop** construct, the **gang** clause is applied to the *tile*  
1436 loops. If the **worker** clause appears on the **loop** construct, the **worker** clause is applied to the  
1437 *element* loops if no **vector** clause appears, and to the *tile* loops otherwise.

### 1438 2.9.8. device\_type clause

1439 The **device\_type** clause is described in Section 2.4 Device-Specific Clauses.

### 1440 2.9.9. independent clause

1441 The **independent** clause tells the implementation that the iterations of this loop are data-independent  
1442 with respect to each other. This allows the implementation to generate code to execute the iterations  
1443 in parallel with no synchronization. When the parent compute construct is a **parallel** construct,  
1444 the **independent** clause is implied on all **loop** constructs without a **seq** or **auto** clause.

#### 1445 Note

- 1446 • It is likely a programming error to use the **independent** clause on a loop if any iteration  
1447 writes to a variable or array element that any other iteration also writes or reads, except for  
1448 variables in a **reduction** clause or accesses in atomic regions.

### 1449 2.9.10. private clause

1450 The **private** clause on a **loop** construct specifies that a copy of each item in *var-list* will be  
1451 created. If the body of the loop is executed in *vector-partitioned* mode, a copy of the item is created  
1452 for each thread associated with each vector lane. If the body of the loop is executed in *worker-*  
1453 *partitioned vector-single* mode, a copy of the item is created for and shared across the set of threads

1454 associated with all the vector lanes of each worker. Otherwise, a copy of the item is created for and  
1455 shared across the set of threads associated with all the vector lanes of all the workers of each gang.

## 1456 Restrictions

- 1457 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in  
1458 **private** clauses.

## 1459 2.9.11. reduction clause

1460 The **reduction** clause specifies a reduction operator and one or more scalar variables. For each  
1461 reduction variable, a private copy is created in the same manner as for a **private** clause on the  
1462 **loop** construct, and initialized for that operator; see the table in Section 2.5.12 reduction clause. At  
1463 the end of the loop, the values for each thread are combined using the specified reduction operator,  
1464 and the result combined with the value of the original variable and stored in the original variable at  
1465 the end of the parallel or kernels region if the loop has gang parallelism, and at the end of the loop  
1466 otherwise.

1467 In a parallel region, if the **reduction** clause is used on a loop with the **vector** or **worker**  
1468 clauses (and no **gang** clause), and the scalar variable also appears in a **private** clause on the  
1469 **parallel** construct, the value of the private copy of the scalar will be updated at the exit of the  
1470 loop. If the scalar variable does not appear in a **private** clause on the **parallel** construct, or if  
1471 the **reduction** clause is used on a loop with the **gang** clause, the value of the scalar will not be  
1472 updated until the end of the parallel region.

1473 If a variable is involved in a reduction that spans multiple nested loops where two or more of those  
1474 loops have associated **loop** directives, a **reduction** clause containing that variable must appear  
1475 on each of those **loop** directives.

## 1476 Restrictions

- 1477 • The **reduction** clause may not be specified on an orphaned **loop** construct with the **gang**  
1478 clause, or on an orphaned **loop** construct that will generate gang parallelism in a procedure  
1479 that is compiled with the **routine gang** clause.
- 1480 • The restrictions for a **reduction** clause on a compute construct listed in in Section 2.5.12  
1481 reduction clause also apply to a **reduction** clause on a loop construct.
- 1482 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in  
1483 **reduction** clauses.

## 1484 2.10. Cache Directive

1485 **Summary** The **cache** directive may appear at the top of (inside of) a loop. It specifies array  
1486 elements or subarrays that should be fetched into the highest level of the cache for the body of the  
1487 loop.

1488 **Syntax** In C and C++, the syntax of the cache directive is



```
#pragma acc cache( var-list ) new-line
```

1489 In Fortran, the syntax of the cache directive is

```
!$acc cache( var-list )
```

1490 The entries in *var-list* must be single array elements or simple subarray. In C and C++, a simple  
1491 subarray is an array name followed by an extended array range specification in brackets, with start  
1492 and length, such as

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

1493 where the lower bound is a constant, loop invariant, or the **for** loop index variable plus or minus a  
1494 constant or loop invariant, and the length is a constant.

1495 In Fortran, a simple subarray is an array name followed by a comma-separated list of range specifi-  
1496 cations in parentheses, with lower and upper bound subscripts, such as

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

1497 The lower bounds must be constant, loop invariant, or the **do** loop index variable plus or minus  
1498 a constant or loop invariant; moreover the difference between the corresponding upper and lower  
1499 bounds must be a constant.

## 1500 Restrictions

- 1501 • If an array is listed in a **cache** directive, all references to that array during execution of that  
1502 loop iteration must not refer to elements of the array outside the index range specified in the  
1503 **cache** directive.
- 1504 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in  
1505 **cache** directives.

## 1506 2.11. Combined Constructs

1507 **Summary** The combined OpenACC **parallel loop**, **kernels loop**, and **serial loop**  
1508 constructs are shortcuts for specifying a **loop** construct nested immediately inside a **parallel**,  
1509 **kernels**, or **serial** construct. The meaning is identical to explicitly specifying a **parallel**,  
1510 **kernels**, or **serial** construct containing a **loop** construct. Any clause that is allowed on a  
1511 **parallel** or **loop** construct is allowed on the **parallel loop** construct; any clause allowed  
1512 on a **kernels** or **loop** construct is allowed on a **kernels loop** construct; and any clause  
1513 allowed on a **serial** or **loop** construct is allowed on a **serial loop** construct.

1514 **Syntax** In C and C++, the syntax of the **parallel loop** construct is

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

1515 In Fortran, the syntax of the **parallel loop** construct is

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

1516 The associated structured block is the loop which must immediately follow the directive. Any of the  
1517 **parallel** or **loop** clauses valid in a parallel region may appear. The **private** and **reduction**  
1518 clauses, which can appear on both a **parallel** construct and a **loop** construct, are treated on a  
1519 **parallel loop** construct as if they appeared on the **loop** construct.

1520 In C and C++, the syntax of the **kernels loop** construct is

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

1521 In Fortran, the syntax of the **kernels loop** construct is

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

1522 The associated structured block is the loop which must immediately follow the directive. Any of  
1523 the **kernels** or **loop** clauses valid in a kernels region may appear.

1524 In C and C++, the syntax of the **serial loop** construct is

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

1525 In Fortran, the syntax of the **serial loop** construct is

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

1526 The associated structured block is the loop which must immediately follow the directive. Any of  
1527 the **serial** or **loop** clauses valid in a serial region may appear. The **private** clause, which can  
1528 appear on both a **serial** construct and a **loop** construct, is treated on a **serial loop** construct  
1529 as if it appeared on the **loop** construct.

1530 **Restrictions**

- 1531 • The restrictions for the **parallel**, **kernels**, **serial**, and **loop** constructs apply.

1532 **2.12. Atomic Construct**

1533 **Summary** An **atomic** construct ensures that a specific storage location is accessed and/or up-  
 1534 dated atomically, preventing simultaneous reading and writing by gangs, workers, and vector threads  
 1535 that could result in indeterminate values.

1536 **Syntax** In C and C++, the syntax of the **atomic** constructs is:

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

1537 OR:

```
#pragma acc atomic update capture new-line
           structured-block
```

1538 Where *atomic-clause* is one of **read**, **write**, **update**, or **capture**. The *expression-stmt* is an  
 1539 expression statement with one of the following forms:

1540 If the *atomic-clause* is **read**:

```
v = x;
```

1541 If the *atomic-clause* is **write**:

```
x = expr;
```

1542 If the *atomic-clause* is **update** or not present:

```
x++;
x--;
++x;
--x;
x binop = expr;
x = x binop expr;
x = expr binop x;
```

1543 If the *atomic-clause* is **capture**:

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

```

v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;

```

1544 The *structured-block* is a structured block with one of the following forms:

```

{v = x; x binop= expr; }
{x binop= expr; v = x; }
{v = x; x = x binop expr; }
{v = x; x = expr binop x; }
{x = x binop expr; v = x; }
{x = expr binop x; v = x; }
{v = x; x = expr; }
{v = x; x++; }
{v = x; ++x; }
{++x; v = x; }
{x++; v = x; }
{v = x; x--; }
{v = x; --x; }
{--x; v = x; }
{x--; v = x; }

```

1545 In the preceding expressions:

- 1546 • **x** and **v** (as applicable) are both l-value expressions with scalar type.
- 1547 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate  
1548 the same storage location.
- 1549 • Neither of **v** and *expr* (as applicable) may access the storage location designated by **x**.
- 1550 • Neither of **x** and *expr* (as applicable) may access the storage location designated by **v**.
- 1551 • *expr* is an expression with scalar type.
- 1552 • *binop* is one of **+**, **\***, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.
- 1553 • *binop*, *binop*=, **++**, and **--** are not overloaded operators.
- 1554 • The expression **x** *binop* *expr* must be mathematically equivalent to **x** *binop* (*expr*). This  
1555 requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by  
1556 using parentheses around *expr* or subexpressions of *expr*.
- 1557 • The expression *expr* *binop* **x** must be mathematically equivalent to (*expr*) *binop* **x**. This  
1558 requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*,  
1559 or by using parentheses around *expr* or subexpressions of *expr*.
- 1560 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is  
1561 unspecified.

1562 In Fortran the syntax of the **atomic** constructs is:

```

!$acc atomic read

```

*capture-statement*  
[!\$acc end atomic]

1563 OR

!\$acc atomic write  
*write-statement*  
[!\$acc end atomic]

1564 OR

!\$acc atomic [update]  
*update-statement*  
[!\$acc end atomic]

1565 OR

!\$acc atomic capture  
*update-statement*  
*capture-statement*  
!\$acc end atomic

1566 OR

!\$acc atomic capture  
*capture-statement*  
*update-statement*  
!\$acc end atomic

1567 OR

!\$acc atomic capture  
*capture-statement*  
*write-statement*  
!\$acc end atomic

1568 where *write-statement* has the following form (if *atomic-clause* is **write** or **capture**):

**x = expr**

1569 where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

**v = x**

1570 and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**,  
1571 or not present):

```

x = x operator expr
x = expr operator x
x = intrinsic_procedure_name ( x, expr-list )
x = intrinsic_procedure_name ( expr-list, x )

```

1572 In the preceding statements:

- 1573 • **x** and **v** (as applicable) are both scalar variables of intrinsic type.
- 1574 • **x** must not be an allocatable variable.
- 1575 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate  
1576 the same storage location.
- 1577 • None of **v**, *expr*, and *expr-list* (as applicable) may access the same storage location as **x**.
- 1578 • None of **x**, *expr*, and *expr-list* (as applicable) may access the same storage location as **v**.
- 1579 • *expr* is a scalar expression.
- 1580 • *expr-list* is a comma-separated, non-empty list of scalar expressions. If *intrinsic\_procedure\_name*  
1581 refers to **iand**, **ior**, or **ieor**, exactly one expression must appear in *expr-list*.
- 1582 • *intrinsic\_procedure\_name* is one of **max**, **min**, **iand**, **ior**, or **ieor**. *operator* is one of **+**,  
1583 **\***, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**
- 1584 • The expression **x operator expr** must be mathematically equivalent to **x operator (expr)**.  
1585 This requirement is satisfied if the operators in *expr* have precedence greater than *operator*,  
1586 or by using parentheses around *expr* or subexpressions of *expr*.
- 1587 • The expression *expr operator x* must be mathematically equivalent to **(expr) operator x**.  
1588 This requirement is satisfied if the operators in *expr* have precedence equal to or greater than  
1589 *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- 1590 • *intrinsic\_procedure\_name* must refer to the intrinsic procedure name and not to other program  
1591 entities.
- 1592 • *operator* must refer to the intrinsic operator and not to a user-defined operator. All assign-  
1593 ments must be intrinsic assignments.
- 1594 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is  
1595 unspecified.

1596 An **atomic** construct with the **read** clause forces an atomic read of the location designated by **x**.  
1597 An **atomic** construct with the **write** clause forces an atomic write of the location designated by  
1598 **x**.

1599 An **atomic** construct with the **update** clause forces an atomic update of the location designated  
1600 by **x** using the designated operator or intrinsic. Note that when no clause is present, the semantics  
1601 are equivalent to **atomic update**. Only the read and write of the location designated by **x** are  
1602 performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect  
1603 to the read or write of the location designated by **x**.

1604 An **atomic** construct with the **capture** clause forces an atomic update of the location designated  
 1605 by **x** using the designated operator or intrinsic while also capturing the original or final value of  
 1606 the location designated by **x** with respect to the atomic update. The original or final value of the  
 1607 location designated by **x** is written into the location designated by **v** depending on the form of the  
 1608 **atomic** construct structured block or statements following the usual language semantics. Only  
 1609 the read and write of the location designated by **x** are performed mutually atomically. Neither the  
 1610 evaluation of *expr* or *expr-list*, nor the write to the location designated by **v**, need to be atomic with  
 1611 respect to the read or write of the location designated by **x**.

1612 For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs  
 1613 enforces mutually exclusive access to the locations designated by **x**. To avoid race conditions, all  
 1614 accesses of the locations designated by **x** that could potentially occur in parallel must be protected  
 1615 with an **atomic** construct.

1616 Atomic regions do not guarantee exclusive access with respect to any accesses outside of atomic re-  
 1617 gions to the same storage location **x** even if those accesses occur during the execution of a reduction  
 1618 clause.

1619 If the storage location designated by **x** is not size-aligned (that is, if the byte alignment of **x** is not a  
 1620 multiple of the size of **x**), then the behavior of the atomic region is implementation-defined.

## 1621 Restrictions

- 1622 • All atomic accesses to the storage locations designated by **x** throughout the program are  
 1623 required to have the same type and type parameters.
- 1624 • Storage locations designated by **x** must be less than or equal in size to the largest available  
 1625 native atomic operator width.

## 1626 2.13. Declare Directive

1627 **Summary** A **declare** directive is used in the declaration section of a Fortran subroutine, func-  
 1628 tion, or module, or following a variable declaration in C or C++. It can specify that a variable or  
 1629 array is to be allocated in the device memory for the duration of the implicit data region of a func-  
 1630 tion, subroutine or program, and specify whether the data values are to be transferred from the host  
 1631 to the device memory upon entry to the implicit data region, and from the device to the host memory  
 1632 upon exit from the implicit data region. These directives create a visible device copy of the variable  
 1633 or array.

1634 **Syntax** In C and C++, the syntax of the **declare** directive is:

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

1635 In Fortran the syntax of the **declare** directive is:

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

1636 where *clause* is one of the following:

```

copy ( var-list )
copyin ( var-list )
copyout ( var-list )
create ( var-list )
present ( var-list )
deviceptr ( var-list )
device_resident ( var-list )
link ( var-list )

```

1637 The associated region is the implicit region associated with the function, subroutine, or program in  
 1638 which the directive appears. If the directive appears in the declaration section of a Fortran *module*  
 1639 subprogram or in a C or C++ global scope, the associated region is the implicit region for the whole  
 1640 program. The **copy**, **copyin**, **copyout**, **present**, and **deviceptr** data clauses are described  
 1641 in Section 2.7 Data Clauses.

## 1642 Restrictions

- 1643 • A **declare** directive must appear in the same scope as any variable or array in any of the  
 1644 data clauses on the directive.
- 1645 • A variable or array may appear at most once in all the clauses of **declare** directives for a  
 1646 function, subroutine, program, or module.
- 1647 • Subarrays are not allowed in **declare** directives.
- 1648 • In Fortran, assumed-size dummy arrays may not appear in a **declare** directive.
- 1649 • In Fortran, pointer arrays may be specified, but pointer association is not preserved in the  
 1650 device memory.
- 1651 • In a Fortran *module* declaration section, only **create**, **copyin**, **device\_resident**, and  
 1652 **link** clauses are allowed.
- 1653 • In C or C++ global scope, only **create**, **copyin**, **deviceptr**, **device\_resident** and  
 1654 **link** clauses are allowed.
- 1655 • C and C++ *extern* variables may only appear in **create**, **copyin**, **deviceptr**, **device\_resident**  
 1656 and **link** clauses on a **declare** directive.
- 1657 • In C and C++, only global and *extern* variables may appear in a **link** clause. In Fortran,  
 1658 only *module* variables and *common* block names (enclosed in slashes) may appear in a **link**  
 1659 clause.
- 1660 • In C or C++, a **longjmp** call in the region must return to a **setjmp** call within the region.
- 1661 • In C++, an exception thrown in the region must be handled within the region.
- 1662 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional dummy ar-  
 1663 guments in data clauses, including **device\_resident** clauses.

### 1664 2.13.1. device\_resident clause

1665 **Summary** The **device\_resident** clause specifies that the memory for the named variables  
 1666 should be allocated in the accelerator device memory and not in the host memory. The names



1667 in the argument list may be variable or array names, or Fortran *common block* names enclosed  
 1668 between slashes; subarrays are not allowed. The host may not be able to access variables in a  
 1669 **device\_resident** clause. The accelerator data lifetime of global variables or common blocks  
 1670 specified in a **device\_resident** clause is the entire execution of the program.

1671 In Fortran, if the variable has the Fortran *allocatable* attribute, the memory for the variable will  
 1672 be allocated in and deallocated from the current accelerator device memory when the host program  
 1673 executes an **allocate** or **deallocate** statement for that variable. If the variable has the Fortran  
 1674 *pointer* attribute, it may be allocated or deallocated by the host in the accelerator device memory, or  
 1675 may appear on the left hand side of a pointer assignment statement, if the right hand side variable  
 1676 itself appears in a **device\_resident** clause.

1677 In Fortran, the argument to a **device\_resident** clause may be a *common block* name enclosed  
 1678 in slashes; in this case, all declarations of the common block must have a matching **device\_resident**  
 1679 clause. In this case, the *common block* will be statically allocated in device memory, and not  
 1680 in host memory. The *common block* will be available to accelerator routines; see Section 2.15  
 1681 Procedure Calls in Compute Regions.

1682 In a Fortran *module* declaration section, a variable or array in a **device\_resident** clause will  
 1683 be available to accelerator subprograms.

1684 In C or C++ global scope, a variable or array in a **device\_resident** clause will be available  
 1685 to accelerator routines. A C or C++ *extern* variable may appear in a **device\_resident** clause  
 1686 only if the actual declaration and all *extern* declarations are also followed by **device\_resident**  
 1687 clauses.

## 1688 2.13.2. create clause

1689 If the current device is a shared memory device, no action is taken.

1690 If the current device is a non-shared memory device, the **create** clause behaves as follows, for  
 1691 each *var* in *var-list*:

- 1692 • At entry to an implicit data region where the **declare** directive appears:
  - 1693 – If *var* is present, a *present increment* action with the structured reference count is per-  
 1694 formed. If *var* is a pointer reference, an *attach* action is performed.
  - 1695 – Otherwise, a *create* action with the structured reference count is performed. If *var* is a  
 1696 pointer reference, an *attach* action is performed.
- 1697 • At exit from an implicit data region where the **declare** directive appears:
  - 1698 – If *var* is not present on the current device, a runtime error is issued.
  - 1699 – Otherwise, a *present decrement* action with the structured reference count is performed.  
 1700 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic  
 1701 reference counts are zero, a *delete* action is performed.

1702 If the **declare** directive appears in a global context, then the data in *var-list* is statically allocated  
 1703 in device memory and the structured reference count is set to one.

1704 In Fortran, if a variable *var* in *var-list* has the Fortran *allocatable* or *pointer* attribute, then:

- 1705 • An **allocate** statement for *var* will allocate memory from both host memory as well as the

- 1706 current accelerator device memory, and the dynamic reference count will be set to one.
- 1707 • A **deallocate** statement for *var* will deallocate memory from both host memory as well as
  - 1708 the current accelerator device memory, and the dynamic reference count will be set to zero.
  - 1709 If the structured reference count is not zero, a runtime error is issued.
- 1710 In Fortran, if a variable *var* in *var-list* has the Fortran *pointer* attribute, then it may appear on the
- 1711 left hand side of a pointer assignment statement, if the right hand side variable itself appears in a
- 1712 **create** clause.

### 1713 2.13.3. link clause

1714 The **link** clause is used for large global host static data that is referenced within an accelerator

1715 routine and that should have a dynamic data lifetime on the device. The **link** clause specifies that

1716 only a global link for the named variables should be statically created in accelerator memory. The

1717 host data structure remains statically allocated and globally available. The device data memory will

1718 be allocated only when the global variable appears on a data clause for a **data** construct, compute

1719 construct, or **enter data** directive. The arguments to the **link** clause must be global data. In C

1720 or C++, the **link** clause must appear **at** global scope, or the arguments must be *extern* variables.

1721 In Fortran, the **link** clause must appear in a *module* declaration section, or the arguments must be

1722 *common block* names enclosed in slashes. A *common block* that is listed in a **link** clause must be

1723 declared with the same size in all program units where it appears. A **declare link** clause must

1724 be visible everywhere the global variables or common block variables are explicitly or implicitly

1725 used in a data clause, compute construct, or accelerator routine. The global variable or *common*

1726 *block* variables may be used in accelerator routines. The accelerator data lifetime of variables or

1727 common blocks specified in a **link** clause is the data region that allocates the variable or common

1728 block with a data clause, or from the execution of the **enter data** directive that allocates the data

1729 until an **exit data** directive deallocates it or until the end of the program.

## 1730 2.14. Executable Directives

### 1731 2.14.1. Init Directive

1732 **Summary** The **init** directive tells the runtime to initialize the runtime for that device type.

1733 This can be used to isolate any initialization cost from the computational cost, when collecting

1734 performance statistics. If no device type is specified all devices will be initialized. An **init**

1735 directive may be used in place of a call to the **acc\_init** runtime API routine, as described in

1736 Section 3.2.7.

1737 **Syntax** In C and C++, the syntax of the **init** directive is:

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

1738 In Fortran the syntax of the **init** directive is:

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

1739 where *clause* is one of the following:

```

device_type ( device-type-list )
device_num ( int-expr )

```

#### 1740 **device\_type clause**

1741 The **device\_type** clause specifies the type of device that is to be initialized in the runtime. If the  
 1742 **device\_type** clause is present, then the *acc-device-type-var* for the current thread is set to the  
 1743 argument value. If no **device\_num** clause is present then all devices of this type are initialized.

#### 1744 **device\_num clause**

1745 The **device\_num** clause specifies the device id to be initialized. If the **device\_num** clause  
 1746 is present, then the *acc-device-num-var* for the current thread is set to the argument value. If no  
 1747 **device\_type** clause is specified, then the specified device id will be initialized for all available  
 1748 device types.

#### 1749 **Restrictions**

- 1750 • This directive may not be called within a **compute** region.
- 1751 • If the device type specified is not available, the behavior is implementation-defined; in partic-  
 1752 ular, the program may abort.
- 1753 • If the directive is called more than once without an intervening **acc\_shutdown** call or  
 1754 **shutdown** directive, with a different value for the device type argument, the behavior is  
 1755 implementation-defined.
- 1756 • If some accelerator regions are compiled to only use one device type, using this directive with  
 1757 a different device type may produce undefined behavior.

#### 1758 **2.14.2. Shutdown Directive**

1759 **Summary** The **shutdown** directive tells the runtime to shut down the connection to the given  
 1760 accelerator device, and free any runtime resources. A **shutdown** directive may be used in place of  
 1761 a call to the **acc\_shutdown** runtime API routine, as described in Section 3.2.8.

1762 **Syntax** In C and C++, the syntax of the **shutdown** directive is:

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

1763 In Fortran the syntax of the **shutdown** directive is:

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

1764 where *clause* is one of the following:

```

device_type ( device-type-list )
device_num ( int-expr )

```

#### 1765 **device\_type** clause

1766 The **device\_type** clause specifies the type of device that is to be disconnected from the runtime.  
 1767 If no **device\_num** clause is present then all devices of this type are disconnected.

#### 1768 **device\_num** clause

1769 The **device\_num** clause specifies the device id to be disconnected.  
 1770 If no clauses are present then all available devices will be disconnected.

#### 1771 **Restrictions**

- 1772 • This directive may not be used during the execution of a compute region.

### 1773 **2.14.3. Set Directive**

1774 **Summary** The **set** directive provides a means to modify internal control variables using direc-  
 1775 tives. Each form of the **set** directive is functionally equivalent to a matching runtime API routine.

1776 **Syntax** In C and C++, the syntax of the **set** directive is:

```

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

```

1777 In Fortran the syntax of the **set** directive is:

```

!$acc set [clause-list]

```

1778 where *clause* is one of the following

```

default_async ( int-expr )
device_num ( int-expr )
device_type ( device-type-list )

```

#### 1779 **default\_async** clause

1780 The **default\_async** clause specifies the asynchronous queue that should be used if no queue  
 1781 is specified and changes the value of *acc-default-async-var* for the current thread to the argument  
 1782 value. If the value is **acc\_async\_default**, the value of *acc-default-async-var* will revert to  
 1783 the initial value, which is **implementation-defined**. A **set default\_async** directive is function-  
 1784 ally equivalent to a call to the **acc\_set\_default\_async** runtime API routine, as described in  
 1785 Section 3.2.16.

1786 **device\_num clause**

1787 The **device\_num** clause specifies the device number to set as the default device for accelerator  
 1788 regions and changes the value of *acc-device-num-var* for the current thread to the argument value.  
 1789 If the value of **device\_num** argument is negative, the runtime will revert to the default behavior,  
 1790 which is **implementation-defined**. A **set device\_num directive** is functionally equivalent to the  
 1791 **acc\_set\_device\_num** runtime API routine, as described in Section 3.2.4.

1792 **device\_type clause**

1793 The **device\_type** clause specifies the device type to set as the default device type for accelerator  
 1794 regions and sets the value of *acc-device-type-var* for the current thread to the argument value. If  
 1795 the value of the **device\_type** argument is zero or the clause is not present, the selected device  
 1796 number will be used for all attached accelerator types. A **set device\_type** directive is func-  
 1797 tionally equivalent to a call to the **acc\_set\_device\_type** runtime API routine, as described in  
 1798 Section 3.2.2.

1799 **Restrictions**

- 1800 • This directive may not be used within a **compute** region.
- 1801 • Passing **default\_async** the value of **acc\_async\_noval** has no effect.
- 1802 • Passing **default\_async** the value of **acc\_async\_sync** will cause all asynchronous  
 1803 directives in the default asynchronous queue to become synchronous.
- 1804 • Passing **default\_async** the value of **acc\_async\_default** will restore the default  
 1805 asynchronous queue to the initial value, which is **implementation-defined**.
- 1806 • If the value of **device\_num** is larger than the maximum supported value for the given type,  
 1807 the behavior is implementation-defined.
- 1808 • At least one clause must be specified.
- 1809 • Two instances of the same clause may not appear on the same directive.

1810 **2.14.4. Update Directive**

1811 **Summary** The **update** directive is used during the lifetime of accelerator data to update all or  
 1812 part of local variables or arrays with values from the corresponding data in device memory, or to  
 1813 update all or part of device variables or arrays with values from the corresponding data in local  
 1814 memory.

1815 **Syntax** In C and C++, the syntax of the **update** directive is:

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

1816 In Fortran the syntax of the **update** data directive is:

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

1817 where *clause* is one of the following:

```

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

```

1818 The *var-list* argument to an **update** clause is a comma-separated collection of variable names,  
 1819 array names, or subarray specifications. Multiple subarrays of the same array may appear in a *var-*  
 1820 *list* of the same or different clauses on the same directive. The effect of an **update** clause is to  
 1821 copy data from the accelerator device memory to the local memory for **update self**, and from  
 1822 local memory to accelerator device memory for **update device**. The updates are done in the  
 1823 order in which they appear on the directive. No action is taken for a variable or array in the **self**  
 1824 or **device** clause if there is no device copy of that variable or array. At least one **self**, **host**, or  
 1825 **device** clause must appear on the directive.

#### 1826 **self clause**

1827 The **self** clause specifies that the variables, arrays, or subarrays in *var-list* are to be copied from  
 1828 the current accelerator device memory to the local memory for a non-shared memory accelerator. If  
 1829 the current accelerator shares memory with the encountering thread, no action is taken. An **update**  
 1830 directive with the **self** clause is equivalent to a call to the **acc\_update\_self** routine, described  
 1831 in Section 3.2.25.

#### 1832 **host clause**

1833 The **host** clause is a synonym for the **self** clause.

#### 1834 **device clause**

1835 The **device** clause specifies that the variables, arrays, or subarrays in *var-list* are to be copied from  
 1836 the local memory to the current accelerator device memory, for a non-shared memory accelerator.  
 1837 If the current accelerator shares memory with the encountering thread, no action is taken. directive  
 1838 with the **device** clause is equivalent to a call to the **acc\_update\_device** routine, described  
 1839 in Section 3.2.24.

#### 1840 **if clause**

1841 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to  
 1842 perform the updates unconditionally. When an **if** clause appears, the implementation will generate

1843 code to conditionally perform the updates only when the *condition* evaluates to nonzero in C or  
1844 C++, or `.true.` in Fortran.

#### 1845 **async clause**

1846 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

#### 1847 **wait clause**

1848 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

#### 1849 **if\_present clause**

1850 When an **if\_present** clause appears on the directive, no action is taken for a variable or array  
1851 which appears in *var-list* that is not present on the current device. When no **if\_present** clause  
1852 appears, all variables and arrays in a **device** or **self** clause must be present on the current device,  
1853 and an implementation may halt the program with an error message if some data is not present.

#### 1854 **Restrictions**

- 1855 • The **update** directive is executable. It must not appear in place of the statement following  
1856 an *if*, *while*, *do*, *switch*, or *label* in C or C++, or in place of the statement following a logical  
1857 *if* in Fortran.
- 1858 • If no **if\_present** clause appears on the directive, each variable and array **that** appears in  
1859 *var-list* must be present on the current device.
- 1860 • A variable or array **that** appears in *var-list* must be present on the current device.
- 1861 • Only the **async** and **wait** clauses may follow a **device\_type** clause.
- 1862 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical  
1863 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1864 • Noncontiguous subarrays may be specified. It is implementation-specific whether noncon-  
1865 tiguous regions are updated by using one transfer for each contiguous subregion, or whether  
1866 the noncontiguous data is packed, transferred once, and unpacked, or whether one or more  
1867 larger subarrays (no larger than the smallest contiguous region that contains the specified  
1868 subarray) are updated.
- 1869 • In C and C++, a member of a struct or class may be specified, including a subarray of a  
1870 member. Members of a subarray of struct or class type may not be specified.
- 1871 • In C and C++, if a subarray notation is used for a struct member, subarray notation may not  
1872 be used for any parent of that struct member.
- 1873 • In Fortran, members of variables of derived type may be specified, including a subarray of a  
1874 member. Members of subarrays of derived type may not be specified.
- 1875 • In Fortran, if array or subarray notation is used for a derived type member, array or subarray  
1876 notation may not be used for a parent of that derived type member.

- 1877       • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in  
1878       **self**, **host**, and **device** clauses.

### 1879   **2.14.5. Wait Directive**

1880   See Section 2.16 Asynchronous Behavior for more information.

### 1881   **2.14.6. Enter Data Directive**

1882   See Section 2.6.4 Enter Data and Exit Data Directives for more information.

### 1883   **2.14.7. Exit Data Directive**

1884   See Section 2.6.4 Enter Data and Exit Data Directives for more information.

## 1885   **2.15. Procedure Calls in Compute Regions**

1886   This section describes how routines are compiled for an accelerator and how procedure calls are  
1887   compiled in compute regions. See Section 2.17 Fortran Optional Arguments for discussion of For-  
1888   tran optional arguments in procedure calls inside compute regions.

### 1889   **2.15.1. Routine Directive**

1890   **Summary**   The **routine** directive is used to tell the compiler to compile a given procedure for  
1891   an accelerator as well as **for** the host. In a file or routine with a procedure call, the **routine**  
1892   directive tells the implementation the attributes of the procedure when called on the accelerator.

1893   **Syntax**   In C and C++, the syntax of the **routine** directive is:

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

1894   In C and C++, the **routine** directive without a name may appear immediately before a function  
1895   definition or just before a function prototype and applies to that immediately following function or  
1896   prototype. The **routine** directive with a name may appear anywhere that a function prototype  
1897   is allowed and applies to the function in that scope with that name, but must appear before any  
1898   definition or use of that function.

1899   In Fortran the syntax of the **routine** directive is:

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



1900 In Fortran, the **routine** directive without a name may appear within the specification part of a  
1901 subroutine or function definition, or within an interface body for a subroutine or function in an  
1902 interface block, and applies to the containing subroutine or function. The **routine** directive with  
1903 a name may appear in the specification part of a subroutine, function or module, and applies to the  
1904 named subroutine or function.

1905 A C or C++ function or Fortran subprogram compiled with the **routine** directive for an accelera-  
1906 tor is called an *accelerator routine*.

1907 The *clause* is one of the following:

```
gang  
worker  
vector  
seq  
bind( name )  
bind( string )  
device_type( device-type-list )  
nohost
```

1908 A **gang**, **worker**, **vector**, or **seq** clause specifies the *level of parallelism* in the routine.

#### 1909 **gang clause**

1910 The **gang** clause specifies that the procedure contains, may contain, or may call another procedure  
1911 that contains a loop with a **gang** clause. A call to this procedure must appear in code that is  
1912 executed in *gang-redundant* mode, and all gangs must execute the call. For instance, a procedure  
1913 with a **routine gang** directive may not be called from within a loop that has a **gang** clause.  
1914 Only one of the **gang**, **worker**, **vector** and **seq** clauses may be specified for each device type.

#### 1915 **worker clause**

1916 The **worker** clause specifies that the procedure contains, may contain, or may call another pro-  
1917 cedure that contains a loop with a **worker** clause, but does not contain nor does it call another  
1918 procedure that contains a loop with the **gang** clause. A loop in this procedure with an **auto** clause  
1919 may be selected by the compiler to execute in **worker** or **vector** mode. A call to this procedure  
1920 must appear in code that is executed in *worker-single* mode, though it may be in *gang-redundant*  
1921 or *gang-partitioned* mode. For instance, a procedure with a **routine worker** directive may be  
1922 called from within a loop that has the **gang** clause, but not from within a loop that has the **worker**  
1923 clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may be specified for each  
1924 device type.

#### 1925 **vector clause**

1926 The **vector** clause specifies that the procedure contains, may contain, or may call another pro-  
1927 cedure that contains a loop with the **vector** clause, but does not contain nor does it call another  
1928 procedure that contains a loop with either a **gang** or **worker** clause. A loop in this procedure with

1929 an **auto** clause may be selected by the compiler to execute in **vector** mode, but not **worker**  
1930 mode. A call to this procedure must appear in code that is executed in *vector-single* mode, though  
1931 it may be in *gang-redundant* or *gang-partitioned* mode, and in *worker-single* or *worker-partitioned*  
1932 mode. For instance, a procedure with a **routine vector** directive may be called from within  
1933 a loop that has the **gang** clause or the **worker** clause, but not from within a loop that has the  
1934 **vector** clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may be specified for  
1935 each device type.

#### 1936 **seq clause**

1937 The **seq** clause specifies that the procedure does not contain nor does it call another procedure that  
1938 contains a loop with a **gang**, **worker**, or **vector** clause. A loop in this procedure with an **auto**  
1939 clause will be executed in **seq** mode. A call to this procedure may appear in any mode. Only one  
1940 of the **gang**, **worker**, **vector** and **seq** clauses may be specified for each device type.

#### 1941 **bind clause**

1942 The **bind** clause specifies the name to use when calling the procedure on the device. If the name is  
1943 specified as an identifier, it is called as if that name were specified in the language being compiled.  
1944 If the name is specified as a string, the string is used for the procedure name unmodified. A **bind**  
1945 clause on a procedure definition behaves as if it had appeared on a declaration by changing the name  
1946 used to call the function on the device; however, the procedure is not compiled for the device with  
1947 either the original name or the name in the **bind** clause.

1948 If there is both a Fortran **bind** and an acc **bind** clause for a procedure definition then a call on the  
1949 host will call the Fortran bound name and a call on the device will call the name in the **bind** clause.

#### 1950 **device\_type clause**

1951 The **device\_type** clause is described in Section 2.4 Device-Specific Clauses.

#### 1952 **nohost clause**

1953 The **nohost** tells the compiler not to compile a version of this procedure for the host. All calls  
1954 to this procedure must appear within compute regions. If this procedure is called from other pro-  
1955 cedures, those other procedures must also have a matching **routine** directive with the **nohost**  
1956 clause.

#### 1957 **Restrictions**

- 1958 • Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device\_type**  
1959 clause.
- 1960 • At least one of the (**gang**, **worker**, **vector**, or **seq**) clauses must be specified. If the  
1961 **device\_type** clause appears on the **routine** directive, a default level of parallelism  
1962 clause must appear before the **device\_type** clause, or a level of parallelism clause must  
1963 be specified following each **device\_type** clause on the directive.

- 1964 • In C and C++, function static variables are not supported in functions to which a **routine**  
1965 directive applies.
- 1966 • In Fortran, variables with the *save* attribute, either explicitly or implicitly, are not supported  
1967 in subprograms to which a **routine** directive applies.
- 1968 • A **bind** clause may not bind to a routine name that has a visible **bind** clause.
- 1969 • If a function or subroutine has a **bind** clause on both the declaration and the definition then  
1970 they both must bind to the same name.

### 1971 2.15.2. Global Data Access

1972 C or C++ global, file static, or *extern* variables or array, and Fortran *module* or *common block* vari-  
1973 ables or arrays, that are used in accelerator routines must appear in a declare directive in a **create**,  
1974 **copyin**, **device\_resident** or **link** clause. If the data appears in a **device\_resident**  
1975 clause, the **routine** directive for the procedure must include the **nohost** clause. If the data ap-  
1976 pears in a **link** clause, that data must have an active accelerator data lifetime by virtue of appearing  
1977 in a data clause for a **data** construct, compute construct, or **enter data** directive.

## 1978 2.16. Asynchronous Behavior

1979 This section describes the **async** clause and the behavior of programs that use asynchronous data  
1980 movement and compute constructs, and asynchronous API routines.

### 1981 2.16.1. **async** clause

1982 The **async** clause may appear on a **parallel**, **kernels**, or **serial** construct, or an **enter**  
1983 **data**, **exit data**, **update**, or **wait** directive. In all cases, the **async** clause is optional. When  
1984 there is no **async** clause on a compute or data construct, the local thread will wait until the compute  
1985 construct or data operations for the current device are complete before executing any of the code  
1986 that follows. When there is no **async** clause on a **wait** directive, the local thread will wait until  
1987 all operations on the appropriate asynchronous activity queues for the current device are complete.  
1988 When there is an **async** clause, the parallel, kernels, or **serial** region or data operations may be  
1989 processed asynchronously while the local thread continues with the code following the construct or  
1990 directive.

1991 The **async** clause may have a single *async-argument*, where an *async-argument* is a nonnegative  
1992 scalar integer expression (*int* for C or C++, *integer* for Fortran), or one of the special **async** values  
1993 defined below. The behavior with a negative *async-argument*, except the special **async** values de-  
1994 fined below, is implementation-defined. The value of the *async-argument* may be used in a **wait**  
1995 directive, **wait** clause, or various runtime routines to test or wait for completion of the operation.

1996 Two special **async** values are defined in the C and Fortran header files and the Fortran **openacc**  
1997 module. These are negative values, so as not to conflict with a user-specified nonnegative *async-*  
1998 *argument*. An **async** clause with the *async-argument* **acc\_async\_noval** will behave the same  
1999 as if the **async** clause had no argument. An **async** clause with the *async-argument* **acc\_async\_sync**  
2000 will behave the same as if no **async** clause appeared.

2001 The *async-value* of any operation is the value of the *async-argument*, if present, or the value  
 2002 of *acc-default-async-var* if it is **acc\_async\_noval** or if the **async** clause had no value, or  
 2003 **acc\_async\_sync** if no **async** clause appeared. If the current device supports asynchronous  
 2004 operation with one or more device activity queues, the *async-value* is used to select the queue on  
 2005 the current device onto which to enqueue an operation. The properties of the current device and the  
 2006 implementation will determine how many actual activity queues are supported, and how the *async-*  
 2007 *value* is mapped onto the actual activity queues. Two asynchronous operations with the same current  
 2008 device and the same *async-value* will be enqueued onto the same activity queue, and therefore will  
 2009 be executed on the device in the order they are encountered by the local thread. Two asynchronous  
 2010 operations with different *async-values* may be enqueued onto different activity queues, and there-  
 2011 fore may be executed on the device in either order relative to each other. If there are two or more  
 2012 host threads executing and sharing the same accelerator device, two asynchronous operations with  
 2013 the same *async-value* will be enqueued on the same activity queue. If the threads are not synchro-  
 2014 nized with respect to each other, the operations may be enqueued in either order and therefore may  
 2015 execute on the device in either order. Asynchronous operations enqueued to different devices may  
 2016 execute in any order, regardless of the *async-value* used for each.

### 2017 2.16.2. wait clause

2018 The **wait** clause may appear on a **parallel**, **kernels**, or **serial** construct, or an **enter**  
 2019 **data**, **exit data**, or **update** directive. In all cases, the **wait** clause is optional. When there  
 2020 is no **wait** clause, the associated compute or update operations may be enqueued or launched or  
 2021 executed immediately on the device. If there is an argument to the **wait** clause, it must be a list  
 2022 of one or more *async-arguments*. The compute, data or update operation may not be launched or  
 2023 executed until all operations enqueued up to this point by this thread on the associated asynchronous  
 2024 device activity queues have completed. One legal implementation is for the local thread to wait for  
 2025 all the associated asynchronous device activity queues. Another legal implementation is for the  
 2026 local thread to enqueue the compute or update operation in such a way that the operation will  
 2027 not start until the operations enqueued on the associated asynchronous device activity queues have  
 2028 completed.

### 2029 2.16.3. Wait Directive

2030 **Summary** The **wait** directive causes the local thread to wait for completion of asynchronous  
 2031 operations on the current device, such as an accelerator **parallel**, **kernels**, or **serial** region or an  
 2032 **update** directive, or causes one device activity queue to synchronize with one or more other ac-  
 2033 tivity queues on the current device.

2034 **Syntax** In C and C++, the syntax of the **wait** directive is:

```
#pragma acc wait [( int-expr-list )][clause-list] new-line
```

2035 In Fortran the syntax of the **wait** directive is:

```
!$acc wait [( int-expr-list )][clause-list]
```

2036 where *clause* is:

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

2037 The wait argument, if present, must be one or more *async-arguments*.

2038 If there is no wait argument and no **async** clause, the local thread will wait until all operations  
2039 enqueued by this thread on any activity queue on the current device have completed.

2040 If there are one or more *int-expr* expressions and no **async** clause, the local thread will wait until all  
2041 operations enqueued by this thread on each of the associated device activity queues have completed.

2042 If there are two or more threads executing and sharing the same accelerator device, a **wait** directive  
2043 with no **async** clause will cause the local thread to wait until all of the appropriate asynchronous  
2044 operations previously enqueued by that thread have completed. To guarantee that operations have  
2045 been enqueued by other threads requires additional synchronization between those threads. There  
2046 is no guarantee that all the similar asynchronous operations initiated by other threads will have  
2047 completed.

2048 If there is an **async** clause, no new operation may be launched or executed on the **async** activ-  
2049 ity queue on the current device until all operations enqueued up to this point by this thread on the  
2050 asynchronous activity queues associated with the wait argument have completed. One legal imple-  
2051 mentation is for the local thread to wait for all the associated asynchronous device activity queues.  
2052 Another legal implementation is for the thread to enqueue a synchronization operation in such a  
2053 way that no new operation will start until the operations enqueued on the associated asynchronous  
2054 device activity queues have completed.

2055 A **wait** directive is functionally equivalent to a call to one of the **acc\_wait**, **acc\_wait\_async**,  
2056 **acc\_wait\_all** or **acc\_wait\_all\_async** runtime API routines, as described in Sections 3.2.11,  
2057 3.2.12, 3.2.13 and 3.2.14.

## 2058 2.17. Fortran Optional Arguments

2059 This section refers to the Fortran intrinsic function **PRESENT**. A call to the Fortran intrinsic function  
2060 **PRESENT(arg)** returns **.true.**, if **arg** is an optional dummy argument and an actual argument  
2061 for **arg** was present in the argument list of the call site. This should not be confused with the  
2062 OpenACC **present** data clause.

2063 The appearance of a Fortran optional argument **arg** as a *var* in any of the following clauses has no  
2064 effect at runtime if **PRESENT(arg)** is **.false.**:

- 2065 • in data clauses on **compute** and **data** constructs;
- 2066 • in data clauses on **enter data** and **exit data** directives;
- 2067 • in data and **device\_resident** clauses on **declare** directives;
- 2068 • in **use\_device** clauses on **host\_data** directives;
- 2069 • in **self**, **host**, and **device** clauses on **update** directives.

2070 The appearance of a Fortran optional argument **arg** in the following situations may result in unde-  
2071 fined behavior if **PRESENT(arg)** is **.false.** when the associated construct is executed:

- 2072     • as a *var* in **private**, **firstprivate**, and **reduction** clauses;
- 2073     • as a *var* in **cache** directives;
- 2074     • as part of an expression in any clause or directive.

2075 A call to the Fortran intrinsic function **PRESENT** behaves the same way in a compute construct or  
2076 an accelerator routine as on the host. The function call **PRESENT (arg)** must return the same value  
2077 in a compute construct as **PRESENT (arg)** would outside of the compute construct. If a Fortran  
2078 optional argument **arg** appears as an actual argument in a procedure call in a compute construct  
2079 or an accelerator routine, and the associated dummy argument **subarg** also has the **optional**  
2080 attribute, then **PRESENT (subarg)** returns the same value as **PRESENT (subarg)** would when  
2081 executed on the host.

## 3. Runtime Library

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

This chapter has two sections:

- Runtime library definitions
- Runtime library routines

There are four categories of runtime routines:

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

### 3.1. Runtime Library Definitions

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

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

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

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

- 2112 • Integer parameters to describe types of accelerators.
- 2113 • **Integer parameters to define the** values of `acc_async_noval`, `acc_async_sync`, and
- 2114 `acc_async_default`.

2115 Many of the routines accept or return a value corresponding to the type of accelerator device. In  
2116 C and C++, the datatype used for device type values is `acc_device_t`; in Fortran, the cor-  
2117 responding datatype is `integer(kind=acc_device_kind)`. The possible values for de-  
2118 vice type are implementation specific, and are defined in the C or C++ include file `openacc.h`  
2119 and the Fortran module `openacc`. Four values are always supported: `acc_device_none`,  
2120 `acc_device_default`, `acc_device_host` and `acc_device_not_host`. For other val-  
2121 ues, look at the appropriate files included with the implementation, or read the documentation for  
2122 the implementation. The value `acc_device_default` will never be returned by any function;  
2123 its use as an argument will tell the runtime library to use the default device type for that implemen-  
2124 tation.

## 2125 3.2. Runtime Library Routines

2126 In this section, for the C and C++ prototypes, pointers are typed `h_void*` or `d_void*` to desig-  
2127 nate a host address or device address, when these calls are executed on the host, as if the following  
2128 definitions were included:

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

2129 Except for `acc_on_device`, these routines are only available on the host.

### 2130 3.2.1. `acc_get_num_devices`

2131 **Summary** The `acc_get_num_devices` routine returns the number of accelerator devices of  
2132 the given type attached to the host.

#### 2133 **Format**

C or C++:

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

Fortran:

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

2134 **Description** The `acc_get_num_devices` routine returns the number of accelerator devices  
2135 of the given type attached to the host. The argument tells what kind of device to count.



2136 **Restrictions**

- 2137
- This routine may not be called within a compute region.

2138 **3.2.2. acc\_set\_device\_type**

2139 **Summary** The `acc_set_device_type` routine tells the runtime which type of device to use  
2140 when executing a compute region and sets the value of `acc-device-type-var`. This is useful when the  
2141 implementation allows the program to be compiled to use more than one type of accelerator.

2142 **Format**

C or C++:

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

Fortran:

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

2143 **Description** The `acc_set_device_type` routine tells the runtime which type of device to  
2144 use among those available and sets the value of `acc-device-type-var` for the current thread. A call to  
2145 `acc_set_device_type` is functionally equivalent to a `set device_type` directive with the  
2146 matching device type argument, as described in Section 2.14.3.

2147 **Restrictions**

- 2148
- This routine may not be called within a compute region.
  - If the device type specified is not available, the behavior is implementation-defined; in particular, the program may abort.
  - If some accelerator regions are compiled to only use one device type, calling this routine with a different device type may produce undefined behavior.

2153 **3.2.3. acc\_get\_device\_type**

2154 **Summary** The `acc_get_device_type` routine returns the value of `acc-device-type-var`, which  
2155 is the device type of the current device. This is useful when the implementation allows the program  
2156 to be compiled to use more than one type of accelerator.

2157 **Format**

C or C++:

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

Fortran:

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

2158 **Description** The `acc_get_device_type` routine returns the value of *acc-device-type-var*  
 2159 for the current thread to tell the program what type of device will be used to run the next compute  
 2160 region, if one has been selected. The device type may have been selected by the program with an  
 2161 `acc_set_device_type` call, with an environment variable, or by the default behavior of the  
 2162 program.

### 2163 Restrictions

- 2164 • This routine may not be called within a compute region.
- 2165 • If the device type has not yet been selected, the value `acc_device_none` may be returned.

### 2166 3.2.4. `acc_set_device_num`

2167 **Summary** The `acc_set_device_num` routine tells the runtime which device to use and sets  
 2168 the value of *acc-device-num-var*.

### 2169 Format

C or C++:

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

Fortran:

```
subroutine acc_set_device_num( devicenum, devicetype )
  integer :: devicenum
  integer(acc_device_kind) :: devicetype
```

2170 **Description** The `acc_set_device_num` routine tells the runtime which device to use among  
 2171 those attached of the given type for compute or data regions in the current thread and sets the value  
 2172 of *acc-device-num-var*. If the value of `devicenum` is negative, the runtime will revert to its default  
 2173 behavior, which is implementation-defined. If the value of the second argument is zero, the selected  
 2174 device number will be used for all attached accelerator types. A call to `acc_set_device_num`  
 2175 is functionally equivalent to a `set device_num` directive with the matching device number argu-  
 2176 ment, as described in Section 2.14.3.

### 2177 Restrictions

- 2178 • This routine may not be called within a compute or data region.
- 2179 • If the value of `devicenum` is greater than or equal to the value returned by `acc_get_num_devices`  
 2180 for that device type, the behavior is implementation-defined.
- 2181 • Calling `acc_set_device_num` implies a call to `acc_set_device_type` with that  
 2182 device type argument.

2183 **3.2.5. acc\_get\_device\_num**

2184 **Summary** The `acc_get_device_num` routine returns the value of *acc-device-num-var* for  
 2185 the current thread.

2186 **Format**

C or C++:

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

Fortran:

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

2187 **Description** The `acc_get_device_num` routine returns **the** value of *acc-device-num-var* for  
 2188 the current thread.

2189 **Restrictions**

- 2190 • This routine may not be called within a compute region.

2191 **3.2.6. acc\_get\_property**

2192 **Summary** The `acc_get_property` and `acc_get_property_string` routines return  
 2193 the value of a *device-property* for the specified device.

2194 **Format**

C or C++:

```
size_t acc_get_property( int devicenum,
                        acc_device_t devicetype, acc_device_property_t property );
const char* acc_get_property_string( int devicenum,
                                    acc_device_t devicetype, acc_device_property_t property );
```

Fortran:

```
function acc_get_property( devicenum, devicetype, property )
subroutine acc_get_property_string( devicenum, devicetype,
                                   property, string )
integer, value :: devicenum
integer(acc_device_kind), value :: devicetype
integer(acc_device_property), value :: property
integer(acc_device_property) :: acc_get_property
character(*) :: string
```

2195 **Description** The `acc_get_property` and `acc_get_property_string` routines returns  
 2196 the value of the specified *property*. `devicenum` and `devicetype` specify the device being  
 2197 queried. If `devicetype` has the value `acc_device_current`, then `devicenum` is ignored  
 2198 and the value of the property for the current device is returned. `property` is an enumeration  
 2199 constant, defined in `openacc.h`, for C or C++, or an integer parameter, defined in the `openacc`  
 2200 module, for Fortran. Integer-valued properties are returned by `acc_get_property`, and string-  
 2201 valued properties are returned by `acc_get_property_string`. In Fortran, `acc_get_property_string`  
 2202 returns the result into the `character` variable passed as the last argument.

2203 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_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

2205 An implementation may support additional properties for some devices.

## 2206 Restrictions

- 2207 • These routines may not be called within an compute region.
- 2208 • If the value of `property` is not one of the known values for that query routine, or that  
 2209 property has no value for the specified device, `acc_get_property` will return 0 and  
 2210 `acc_get_property_string` will return NULL (in C or C++) or an blank string (in  
 2211 Fortran).

## 2212 3.2.7. acc\_init

2213 **Summary** The `acc_init` routine tells the runtime to initialize the runtime for that device type.  
 2214 This can be used to isolate any initialization cost from the computational cost, when collecting  
 2215 performance statistics.

## 2216 Format

C or C++:

```
void acc_init( acc_device_t );
```

Fortran:

```
subroutine acc_init( devicetype )
  integer(acc_device_kind) :: devicetype
```

2217 **Description** The `acc_init` routine also implicitly calls `acc_set_device_type`. A call to  
 2218 `acc_init` is functionally equivalent to a `init` directive with the matching device type argument,  
 2219 as described in Section 2.14.1.

## 2220 Restrictions

- 2221 • This routine may not be called within a compute region.
- 2222 • If the device type specified is not available, the behavior is implementation-defined; in particular, the program may abort.
- 2223
- 2224 • If the routine is called more than once without an intervening **acc\_shutdown** call, with a different value for the device type argument, the behavior is implementation-defined.
- 2225
- 2226 • If some accelerator regions are compiled to only use one device type, calling this routine with a different device type may produce undefined behavior.
- 2227

## 2228 3.2.8. **acc\_shutdown**

2229 **Summary** The **acc\_shutdown** routine tells the runtime to shut down the connection to the  
2230 given accelerator device, and free up any runtime resources. A call to **acc\_shutdown** is func-  
2231 tionally equivalent to a **shutdown** directive with the matching device type argument, as described  
2232 in Section 2.14.2.

## 2233 Format

C or C++:

```
void acc_shutdown( acc_device_t );
```

Fortran:

```
subroutine acc_shutdown( devicetype )  
integer(acc_device_kind) :: devicetype
```

2234 **Description** The **acc\_shutdown** routine disconnects the program from the any accelerator  
2235 device of the specified device type. Any data that is present on any such device is immediately  
2236 deallocated.

## 2237 Restrictions

- 2238 • This routine may not be called during execution of a compute region.
- 2239 • If the program attempts to execute a compute region or access any device data on such a  
2240 device, the behavior is undefined.

## 2241 3.2.9. **acc\_async\_test**

2242 **Summary** The **acc\_async\_test** routine tests for completion of all associated asynchronous  
2243 operations on the current device.

## 2244 Format

C or C++:

```
int acc_async_test( int );
```

Fortran:

```
logical function acc_async_test( arg )  
integer(acc_handle_kind) :: arg
```

2245 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.  
2246 If that value did not appear in any **async** clauses, or if it did appear in one or more **async** clauses  
2247 and all such asynchronous operations have completed on the current device, the **acc\_async\_test**  
2248 routine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some such asyn-  
2249 chronous operations have not completed, the **acc\_async\_test** routine will return with a zero  
2250 value in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the  
2251 **acc\_async\_test** routine will return with a nonzero value or **.true.** only if all matching asyn-  
2252 chronous operations initiated by this thread have completed; there is no guarantee that all matching  
2253 asynchronous operations initiated by other threads have completed.

### 2254 3.2.10. acc\_async\_test\_all

2255 **Summary** The **acc\_async\_test\_all** routine tests for completion of all asynchronous op-  
2256 erations.

#### 2257 Format

C or C++:

```
int acc_async_test_all( );
```

Fortran:

```
logical function acc_async_test_all( )
```

2258 **Description** If all outstanding asynchronous operations have completed, the **acc\_async\_test\_all**  
2259 routine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some asynchronous  
2260 operations have not completed, the **acc\_async\_test\_all** routine will return with a zero value  
2261 in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the  
2262 **acc\_async\_test\_all** routine will return with a nonzero value or **.true.** only if all outstand-  
2263 ing asynchronous operations initiated by this thread have completed; there is no guarantee that all  
2264 asynchronous operations initiated by other threads have completed.

### 2265 3.2.11. acc\_wait

2266 **Summary** The **acc\_wait** routine waits for completion of all associated asynchronous opera-  
2267 tions on the current device.

2268 **Format**

C or C++:

```
void acc_wait( int );
```

Fortran:

```
subroutine acc_wait( arg )  
integer(acc_handle_kind) :: arg
```

2269 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.  
2270 If that value appeared in one or more **async** clauses, the **acc\_wait** routine will not return until  
2271 the latest such asynchronous operation has completed on the current device. If two or more threads  
2272 share the same accelerator, the **acc\_wait** routine will return only if all matching asynchronous  
2273 operations initiated by this thread have completed; there is no guarantee that all matching asyn-  
2274 chronous operations initiated by other threads have completed. For compatibility with version 1.0,  
2275 this routine may also be spelled **acc\_async\_wait**. A call to **acc\_wait** is functionally equiv-  
2276 alent to a **wait** directive with a matching wait argument and no **async** clause, as described in  
2277 Section 2.16.3.

2278 **3.2.12. acc\_wait\_async**

2279 **Summary** The **acc\_wait\_async** routine enqueues a wait operation on one *async* queue of  
2280 the current device for the operations previously enqueued on another *async* queue.

2281 **Format**

C or C++:

```
void acc_wait_async( int, int );
```

Fortran:

```
subroutine acc_wait_async( arg, async )  
integer(acc_handle_kind) :: arg, async
```

2282 **Description** The arguments must be *async-arguments*, as defined in Section 2.16.1 *async* clause.  
2283 The routine will enqueue a wait operation on the appropriate device queue associated with the  
2284 second argument, which will wait for operations enqueued on the device queue associated with  
2285 the first argument. See Section 2.16 Asynchronous Behavior for more information. A call to  
2286 **acc\_wait\_async** is functionally equivalent to a **wait** directive with a matching wait argument  
2287 and a matching **async** argument, as described in Section 2.16.3.

2288 **3.2.13. acc\_wait\_all**

2289 **Summary** The **acc\_wait\_all** routine waits for completion of all asynchronous operations.

2290 **Format**

C or C++:

```
void acc_wait_all( );
```

Fortran:

```
subroutine acc_wait_all( )
```

2291 **Description** The `acc_wait_all` routine will not return until the all asynchronous operations  
2292 have completed. If two or more threads share the same accelerator, the `acc_wait_all` routine  
2293 will return only if all asynchronous operations initiated by this thread have completed; there is no  
2294 guarantee that all asynchronous operations initiated by other threads have completed. For com-  
2295 patibility with version 1.0, this routine may also be spelled `acc_async_wait_all`. A call to  
2296 `acc_wait_all` is functionally equivalent to a `wait` directive with no wait argument list and no  
2297 `async` argument, as described in Section 2.16.3.

2298 **3.2.14. acc\_wait\_all\_async**

2299 **Summary** The `acc_wait_all_async` routine enqueues wait operations on one async queue  
2300 for the operations previously enqueued on all other async queues.

2301 **Format**

C or C++:

```
void acc_wait_all_async( int );
```

Fortran:

```
subroutine acc_wait_all_async( async )  
integer(acc_handle_kind) :: async
```

2302 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.  
2303 The routine will enqueue a wait operation on the appropriate device queue for each other device  
2304 queue. See Section 2.16 Asynchronous Behavior for more information. A call to `acc_wait_all_async`  
2305 is functionally equivalent to a `wait` directive with no wait argument list and a matching `async`  
2306 argument, as described in Section 2.16.3.

2307 **3.2.15. acc\_get\_default\_async**

2308 **Summary** The `acc_get_default_async` routine returns the value of *acc-default-async-*  
2309 *var* for the current thread.

2310 **Format**



C or C++:

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

Fortran:

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

2311 **Description** The `acc_get_default_async` routine returns the value of `acc-default-async-`  
 2312 `var` for the current thread, which is the asynchronous queue used when an `async` clause appears  
 2313 without an `async-argument` or with the value `acc_async_noval`.

### 2314 3.2.16. `acc_set_default_async`

2315 **Summary** The `acc_set_default_async` routine tells the runtime which asynchronous queue  
 2316 to use when no other queue is specified.

#### 2317 **Format**

C or C++:

```
void acc_set_default_async( int async );
```

Fortran:

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

2318 **Description** The `acc_set_default_async` routine tells the runtime to place any directives  
 2319 with an `async` clause that does not have an `async-argument` or with the special `acc_async_noval`  
 2320 value into the specified asynchronous activity queue instead of the default asynchronous activity  
 2321 queue for that device by setting the value of `acc-default-async-var` for the current thread. The spe-  
 2322 cial argument `acc_async_default` will reset the default asynchronous activity queue to the  
 2323 initial value, which is **implementation-defined**. A call to `acc_set_default_async` is func-  
 2324 tionally equivalent to a `set default_async` directive with a matching argument in `int-expr`, as  
 2325 described in Section 2.14.3.

### 2326 3.2.17. `acc_on_device`

2327 **Summary** The `acc_on_device` routine tells the program whether it is executing on a partic-  
 2328 ular device.

#### 2329 **Format**

C or C++:

```
int acc_on_device( acc_device_t );
```

Fortran:

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

2330 **Description** The `acc_on_device` routine may be used to execute different paths depending  
 2331 on whether the code is running on the host or on some accelerator. If the `acc_on_device` routine  
 2332 has a compile-time constant argument, it evaluates at compile time to a constant. The argument must  
 2333 be one of the defined accelerator types. If the argument is `acc_device_host`, then outside of an  
 2334 accelerator compute region or accelerator routine, or in an accelerator compute region or accelerator  
 2335 routine that is executed on the host processor, this routine will evaluate to nonzero for C or C++, and  
 2336 `.true.` for Fortran; otherwise, it will evaluate to zero for C or C++, and `.false.` for Fortran. If  
 2337 the argument is `acc_device_not_host`, the result is the negation of the result with argument  
 2338 `acc_device_host`. If the argument is any accelerator device type, then in a compute region or  
 2339 routine that is executed on an accelerator of that device type, this routine will evaluate to nonzero for  
 2340 C or C++, and `.true.` for Fortran; otherwise, it will evaluate to zero for C or C++, and `.false.`  
 2341 for Fortran. The result with argument `acc_device_default` is undefined.

### 2342 3.2.18. `acc_malloc`

2343 **Summary** The `acc_malloc` routine allocates memory on the accelerator device.

2344 **Format**

C or C++:

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

2345 **Description** The `acc_malloc` routine may be used to allocate memory on the accelerator de-  
 2346 vice. Pointers assigned from this function may be used in `deviceptr` clauses to tell the compiler  
 2347 that the pointer target is resident on the device.

### 2348 3.2.19. `acc_free`

2349 **Summary** The `acc_free` routine frees memory on the accelerator device.

2350 **Format**

C or C++:

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

2351 **Description** The `acc_free` routine will free previously allocated memory on the accelerator  
 2352 device; the argument should be a pointer value that was returned by a call to `acc_malloc`.

2353 **3.2.20. acc\_copyin**

2354 **Summary** The `acc_copyin` routines test to see if the data is already present on the current  
 2355 device; if not, they allocate memory on the accelerator device to correspond to the specified host  
 2356 memory, and copy the data to that device memory, on a non-shared memory device.

2357 **Format**

C or C++:

```
d_void* acc_copyin( h_void*, size_t );
void acc_copyin_async( h_void*, size_t, int );
```

Fortran:

```
subroutine acc_copyin( a )
subroutine acc_copyin( a, len )
subroutine acc_copyin_async( a, async )
subroutine acc_copyin_async( a, len, async )
  type(*), dimension(..) :: a
  integer :: len
  integer(acc_handle_kind) :: async
```

2358 **Description** The `acc_copyin` routines are equivalent to the `enter data` directive with a  
 2359 `copyin` clause, as described in Section 2.7.6. In C, the arguments are a pointer to the data and  
 2360 length in bytes; the synchronous function returns a pointer to the allocated device space, as with  
 2361 `acc_malloc`. In Fortran, two forms are supported. In the first, the argument is a contiguous array  
 2362 section of intrinsic type. In the second, the first argument is a variable or array element and the  
 2363 second is the length in bytes.

2364 The behavior of the `acc_copyin` routines is:

- 2365 • If the current device is a shared memory device, no action is taken. The C `acc_copyin`  
 2366 returns the incoming pointer.
- 2367 • If the data is present, a *present increment* action with the dynamic reference count is per-  
 2368 formed. The C `acc_copyin` returns a pointer to the existing device memory.
- 2369 • Otherwise, a *copyin* action with the appropriate reference count is performed. The C `acc_copyin`  
 2370 returns the device address of the newly allocated memory.

2371 This data may be accessed using the `present` data clause. Pointers assigned from the C `acc_copyin`  
 2372 function may be used in `deviceptr` clauses to tell the compiler that the pointer target is resident  
 2373 on the device.

2374 The `_async` versions of this function will perform any data transfers asynchronously on the `async`  
 2375 queue associated with the value passed in as the `async` argument. The function may return be-  
 2376 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The  
 2377 synchronous versions will not return until the data has been completely transferred.

2378 For compatibility with OpenACC 2.0, `acc_present_or_copyin` and `acc_pcopyin` are al-  
 2379 ternate names for `acc_copyin`.

2380 **3.2.21. acc\_create**

2381 **Summary** The `acc_create` routines test to see if the data is already present on the device; if  
 2382 not, they allocate memory on the accelerator device to correspond to the specified host memory, on  
 2383 a non-shared memory device.

2384 **Format**

C or C++:

```
d_void* acc_create( h_void*, size_t );
void acc_create_async( h_void*, size_t, int async );
```

Fortran:

```
subroutine acc_create( a )
subroutine acc_create( a, len )
subroutine acc_create_async( a, async )
subroutine acc_create_async( a, len, async )
  type(*), dimension(..) :: a
  integer :: len
  integer(acc_handle_kind) :: async
```

2385 **Description** The `acc_create` routines are equivalent to the `enter data` directive with a  
 2386 `create` clause, as described in Section 2.7.8. In C, the arguments are a pointer to the data and  
 2387 length in bytes; the synchronous function returns a pointer to the allocated device space, as with  
 2388 `acc_malloc`. In Fortran, two forms are supported. In the first, the argument is a contiguous array  
 2389 section of intrinsic type. In the second, the first argument is a variable or array element and the  
 2390 second is the length in bytes.

2391 The behavior of the `acc_create` routines is:

- 2392 • If the current device is a shared memory device, no action is taken. The C `acc_create`  
 2393 returns the incoming pointer.
- 2394 • If the data is present, a *present increment* action with the dynamic reference count is per-  
 2395 formed. The C `acc_create` returns a pointer to the existing device memory.
- 2396 • Otherwise, a *create* action with the appropriate reference count is performed. The C `acc_create`  
 2397 returns the device address of the newly allocated memory.

2398 This data may be accessed using the `present` data clause. Pointers assigned from the C `acc_copyin`  
 2399 function may be used in `deviceptr` clauses to tell the compiler that the pointer target is resident  
 2400 on the device.

2401 The `_async` versions of these function may perform the data allocation asynchronously on the  
 2402 async queue associated with the value passed in as the `async` argument. The synchronous versions  
 2403 will not return until the data has been allocated.

2404 For compatibility with OpenACC 2.0, `acc_present_or_create` and `acc_pcreate` are al-  
 2405 ternate names for `acc_create`.

2406 **3.2.22. acc\_copyout**

2407 **Summary** The `acc_copyout` routines copy data from device memory to the corresponding  
 2408 local memory, then deallocate that memory from the accelerator device, on a non-shared memory  
 2409 device.

2410 **Format**

C or C++:

```
void acc_copyout( h_void*, size_t );
void acc_copyout_async( h_void*, size_t, int async );
void acc_copyout_finalize( h_void*, size_t );
void acc_copyout_finalize_async( h_void*, size_t, int async );
```

Fortran:

```
subroutine acc_copyout( a )
subroutine acc_copyout( a, len )
subroutine acc_copyout_async( a, async )
subroutine acc_copyout_async( a, len, async )
subroutine acc_copyout_finalize( a )
subroutine acc_copyout_finalize( a, len )
subroutine acc_copyout_finalize_async( a, async )
subroutine acc_copyout_finalize_async( a, len, async )
  type(*), dimension(..) :: a
  integer :: len
  integer(acc_handle_kind) :: async
```

2411 **Description** The `acc_copyout` routines are equivalent to the `exit data` directive with a  
 2412 `copyout` clause, and the `acc_copyout_finalize` routines are equivalent to the `exit data`  
 2413 `directive with both copyout and finalize clauses, as described in Section 2.7.7. In C, the`  
 2414 `arguments are a pointer to the data and length in bytes. In Fortran, two forms are supported. In the`  
 2415 `first, the argument is a contiguous array section of intrinsic type. In the second, the first argument`  
 2416 `is a variable or array element and the second is the length in bytes.`

2417 The behavior of the `acc_copyout` routines is:

- 2418 • If the current device is a shared memory device, no action is taken.
- 2419 • If the data is not present, a runtime error is issued.
- 2420 • Otherwise, a *present decrement* action with the dynamic reference count is performed (`acc_copyout`),  
 2421 or the dynamic reference count is set to zero (`acc_copyout_finalize`). If both refer-  
 2422 ence counts are then zero, a *copyout* action is performed.

2423 The `_async` versions of these functions will perform any associated data transfers asynchronously  
 2424 on the `async` queue associated with the value passed in as the `async` argument. The function may  
 2425 return before the data has been transferred or deallocated; see Section 2.16 Asynchronous Behavior  
 2426 for more details. The synchronous versions will not return until the data has been completely trans-  
 2427 ferred. Even if the data has not been transferred or deallocated before the function returns, the data  
 2428 will be treated as not present on the device.

2429 **3.2.23. acc\_delete**

2430 **Summary** The `acc_delete` routines deallocate the memory from the accelerator device cor-  
 2431 responding to the specified local memory, on a non-shared memory device.

2432 **Format**

C or C++:

```
void acc_delete( h_void*, size_t );
void acc_delete_async( h_void*, size_t, int async );
void acc_delete_finalize( h_void*, size_t );
void acc_delete_finalize_async( h_void*, size_t, int async );
```

Fortran:

```
subroutine acc_delete( a )
subroutine acc_delete( a, len )
subroutine acc_delete_async( a, async )
subroutine acc_delete_async( a, len, async )
subroutine acc_delete_finalize( a )
subroutine acc_delete_finalize( a, len )
subroutine acc_delete_finalize_async( a, async )
subroutine acc_delete_finalize_async( a, len, async )
  type(*), dimension(..) :: a
  integer :: len
  integer(acc_handle_kind) :: async
```

2433 **Description** The `acc_delete` routines are equivalent to the `exit data` directive with a  
 2434 `delete` clause, and the `acc_delete_finalize` routines are equivalent to the `exit data`  
 2435 directive with both `delete` clause and `finalize` clauses, as described in Section 2.7.10. The  
 2436 arguments are as for `acc_copyout`.

2437 The behavior of the `acc_delete` routines is:

- 2438 • If the current device is a shared memory device, no action is taken.
- 2439 • If the data is not present, a runtime error is issued.
- 2440 • Otherwise, a *present decrement* action with the dynamic reference count is performed (`acc_delete`),  
 2441 or the dynamic reference count is set to zero (`acc_delete_finalize`). If both reference  
 2442 counts are then zero, a *delete* action is performed.

2443 The `_async` versions of these function may perform the data deallocation asynchronously on the  
 2444 `async` queue associated with the value passed in as the `async` argument. The synchronous versions  
 2445 will not return until the data has been deallocated. Even if the data has not been deallocated before  
 2446 the function returns, the data will be treated as not present on the device.

2447 **3.2.24. acc\_update\_device**

2448 **Summary** The `acc_update_device` routine updates the device copy of data from the corre-  
 2449 sponding local memory on a non-shared memory device.

2450 **Format**

C or C++:

```
void acc_update_device( h_void*, size_t );
void acc_update_device_async( h_void*, size_t, int async );
```

Fortran:

```
subroutine acc_update_device( a )
subroutine acc_update_device( a, len )
subroutine acc_update_device( a, async )
subroutine acc_update_device( a, len, async )
  type(*), dimension(..) :: a
  integer :: len
  integer(acc_handle_kind) :: async
```

2451 **Description** The `acc_update_device` routine is equivalent to the `update` directive with a  
 2452 `device` clause, as described in Section 2.14.4. In C, the arguments are a pointer to the data and  
 2453 length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array  
 2454 section of intrinsic type. In the second, the first argument is a variable or array element and the  
 2455 second is the length in bytes. On a non-shared memory device, the data in the local memory is  
 2456 copied to the corresponding device memory. It is a runtime error to call this routine if the data is  
 2457 not present on the device.

2458 The `_async` versions of this function will perform the data transfers asynchronously on the `async`  
 2459 queue associated with the value passed in as the `async` argument. The function may return be-  
 2460 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The  
 2461 synchronous versions will not return until the data has been completely transferred.

2462 **3.2.25. acc\_update\_self**

2463 **Summary** The `acc_update_self` routine updates the device copy of data to the correspond-  
 2464 ing local memory on a non-shared memory device.

2465 **Format**

C or C++:

```
void acc_update_self( h_void*, size_t );
void acc_update_self_async( h_void*, size_t, int async );
```

Fortran:

```

subroutine acc_update_self( a )
subroutine acc_update_self( a, len )
subroutine acc_update_self_async( a, async )
subroutine acc_update_self_async( a, len, async )
  type(*), dimension(..) :: a
  integer :: len
  integer(acc_handle_kind) :: async

```

2466 **Description** The `acc_update_self` routine is equivalent to the `update` directive with a  
 2467 `self` clause, as described in Section 2.14.4. In C, the arguments are a pointer to the data and  
 2468 length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array  
 2469 section of intrinsic type. In the second, the first argument is a variable or array element and the  
 2470 second is the length in bytes. On a non-shared memory device, the data in the local memory is  
 2471 copied to the corresponding device memory. There must be a device copy of the data on the device  
 2472 when calling this routine, otherwise no action is taken by the routine.

2473 The `_async` versions of this function will perform the data transfers asynchronously on the `async`  
 2474 queue associated with the value passed in as the `async` argument. The function may return be-  
 2475 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The  
 2476 synchronous versions will not return until the data has been completely transferred.

### 2477 3.2.26. `acc_map_data`

2478 **Summary** The `acc_map_data` routine maps previously allocated device data to the specified  
 2479 host data.

#### 2480 **Format**

C or C++:

```
void acc_map_data( h_void*, d_void*, size_t );
```

2481 **Description** The `acc_map_data` routine is similar to an `enter data` directive with a `create`  
 2482 clause, except instead of allocating new device memory to start a data lifetime, the device address  
 2483 to use for the data lifetime is specified as an argument. The first argument is a host address, fol-  
 2484 lowed by the corresponding device address and the data length in bytes. After this call, when the  
 2485 host data appears in a data clause, the specified device memory will be used. It is an error to  
 2486 call `acc_map_data` for host data that is already present on the device. It is undefined to call  
 2487 `acc_map_data` with a device address that is already mapped to host data. The device address  
 2488 may be the result of a call to `acc_malloc`, or may come from some other device-specific API  
 2489 routine.

### 2490 3.2.27. `acc_unmap_data`

2491 **Summary** The `acc_unmap_data` routine unmaps device data from the specified host data.



2492 **Format**

C or C++:

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

2493 **Description** The `acc_unmap_data` routine is similar to an `exit data` directive with a  
2494 `delete` clause, except the device memory is not deallocated. The argument is pointer to the host  
2495 data. A call to this routine ends the data lifetime for the specified host data. The device memory is  
2496 not deallocated. It is undefined behavior to call `acc_unmap_data` with a host address unless that  
2497 host address was mapped to device memory using `acc_map_data`.

2498 **3.2.28. acc\_deviceptr**

2499 **Summary** The `acc_deviceptr` routine returns the device pointer associated with a specific  
2500 host address.

2501 **Format**

C or C++:

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

2502 **Description** The `acc_deviceptr` routine returns the device pointer associated with a host  
2503 address. The argument is the address of a host variable or array that has an active lifetime on the  
2504 current device. If the data is not present on the device, the routine returns a NULL value.

2505 **3.2.29. acc\_hostptr**

2506 **Summary** The `acc_hostptr` routine returns the host pointer associated with a specific device  
2507 address.

2508 **Format**

C or C++:

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

2509 **Description** The `acc_hostptr` routine returns the host pointer associated with a device ad-  
2510 dress. The argument is the address of a device variable or array, such as that returned from `acc_deviceptr`,  
2511 `acc_create` or `acc_copyin`. If the device address is NULL, or does not correspond to any host  
2512 address, the routine returns a NULL value.

2513 **3.2.30. acc\_is\_present**

2514 **Summary** The `acc_is_present` routine tests whether a host variable or array region is present  
2515 on the device.

2516 **Format**

C or C++:

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

Fortran:

```
logical function acc_is_present( a )
logical function acc_is_present( a, len )
  type(*), dimension(..) :: a
  integer :: len
```

2517 **Description** The `acc_is_present` routine tests whether the specified host data is present  
 2518 on the device. In C, the arguments are a pointer to the data and length in bytes; the function  
 2519 returns nonzero if the specified data is fully present, and zero otherwise. In Fortran, two forms are  
 2520 supported. In the first, the argument is a contiguous array section of intrinsic type. In the second,  
 2521 the first argument is a variable or array element and the second is the length in bytes. The function  
 2522 returns `.true.` if the specified data is fully present, and `.false.` otherwise. If the byte length is  
 2523 zero, the function returns nonzero in C or `.true.` in Fortran if the given address is present at all  
 2524 on the device.

2525 **3.2.31. acc\_memcpy\_to\_device**

2526 **Summary** The `acc_memcpy_to_device` routine copies data from local memory to device  
 2527 memory.

2528 **Format**

C or C++:

```
void acc_memcpy_to_device( d_void* dest, h_void* src, size_t bytes );
void acc_memcpy_to_device_async( d_void* dest, h_void* src,
  size_t bytes, int async );
```

2529 **Description** The `acc_memcpy_to_device` routine copies `bytes` of data from the local  
 2530 address in `src` to the device address in `dest`. The destination address must be a device address,  
 2531 such as would be returned from `acc_malloc` or `acc_deviceptr`.

2532 The `_async` version of this function will perform the data transfers asynchronously on the `async`  
 2533 queue associated with the value passed in as the `async` argument. The function may return be-  
 2534 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The  
 2535 synchronous versions will not return until the data has been completely transferred.

2536 **3.2.32. acc\_memcpy\_from\_device**

2537 **Summary** The `acc_memcpy_from_device` routine copies data from device memory to lo-  
 2538 cal memory.

2539 **Format**

C or C++:

```
void acc_memcpy_from_device( h_void* dest, d_void* src, size_t bytes );  
void acc_memcpy_from_device_async( h_void* dest, d_void* src,  
    size_t bytes, int async );
```

2540 **Description** The `acc_memcpy_from_device` routine copies `bytes` data from the device  
2541 address in `src` to the local address in `dest`. The source address must be a device address, such as  
2542 would be returned from `acc_malloc` or `acc_deviceptr`.

2543 The `_async` version of this function will perform the data transfers asynchronously on the `async`  
2544 queue associated with the value passed in as the `async` argument. The function may return be-  
2545 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The  
2546 synchronous versions will not return until the data has been completely transferred.

2547 **3.2.33. `acc_memcpy_device`**

2548 **Summary** The `acc_memcpy_device` routine copies data from one memory location to an-  
2549 other memory location on the current device.

2550 **Format**

C or C++:

```
void acc_memcpy_device( d_void* dest, d_void* src, size_t bytes );  
void acc_memcpy_device_async( d_void* dest, d_void* src,  
    size_t bytes, int async );
```

2551 **Description** The `acc_memcpy_device` routine copies `bytes` data from the device address  
2552 in `src` to the device address in `dest`. Both addresses must be addresses in the current device  
2553 memory, such as would be returned from `acc_malloc` or `acc_deviceptr`. If `dest` and `src`  
2554 overlap, the behavior is undefined.

2555 The `_async` version of this function will perform the data transfers asynchronously on the `async`  
2556 queue associated with the value passed in as the `async` argument. The function may return be-  
2557 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The  
2558 synchronous versions will not return until the data has been completely transferred.

2559 **3.2.34. `acc_attach`**

2560 **Summary** The `acc_attach` routine updates a pointer in device memory to point to the corre-  
2561 sponding device copy of the host pointer target.

2562 **Format**

C or C++:

```
void acc_attach( h_void** ptr );  
void acc_attach_async( h_void** ptr, int async );
```

2563 **Description** The `acc_attach` routines are passed the address of a host pointer. If the current  
2564 device is a shared memory device, or if the pointer `*ptr` is not present on the current device, or the  
2565 address to which the `*ptr` points is not present on the current device, no action is taken. Otherwise,  
2566 these routines perform the *attach* action (Section 2.7.2).

2567 These routines may issue a data transfer from local memory to device memory. The `_async`  
2568 version of this function will perform the data transfers asynchronously on the async queue associated  
2569 with the value passed in as the `async` argument. The function may return before the data has been  
2570 transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous version  
2571 will not return until the data has been completely transferred.

### 2572 3.2.35. `acc_detach`

2573 **Summary** The `acc_detach` routine updates a pointer in device memory to point to the host  
2574 pointer target.

#### 2575 **Format**

C or C++:

```
void acc_detach( h_void** ptr );  
void acc_detach_async( h_void** ptr, int async );  
void acc_detach_finalize( h_void** ptr );  
void acc_detach_finalize_async( h_void** ptr, int async );
```

2576 **Description** The `acc_detach` routines are passed the address of a host pointer. If the current  
2577 device is a shared memory device, or if the pointer `*ptr` is not present on the current device, or if  
2578 the *attachment counter* for the pointer `*ptr` is zero, no action is taken. Otherwise, these routines  
2579 perform the *detach* action (Section 2.7.2).

2580 The `acc_detach_finalize` routines are equivalent to an `exit data` directive with `detach`  
2581 and `finalize` clauses, as described in Section 2.7.12 *detach* clause. If the current device is a  
2582 shared memory device, or if the pointer `*ptr` is not present on the current device, or if the *attach-*  
2583 *ment counter* for the pointer `*ptr` is zero, no action is taken. Otherwise, these routines perform the  
2584 *immediate detach* action (Section 2.7.2).

2585 These routines may issue a data transfer from local memory to device memory. The `_async`  
2586 versions of these functions will perform the data transfers asynchronously on the async queue asso-  
2587 ciated with the value passed in as the `async` argument. These functions may return before the data  
2588 has been transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous  
2589 versions will not return until the data has been completely transferred.

## 4. Environment Variables

2590

2591 This chapter describes the environment variables that modify the behavior of accelerator regions.  
2592 The names of the environment variables must be upper case. The values assigned environment  
2593 variables are **case-insensitive** and may have leading and trailing white space. If the values of the  
2594 environment variables change after the program has started, even if the program itself modifies the  
2595 values, the behavior is implementation-defined.

### 4.1. ACC\_DEVICE\_TYPE

2596

2597 The **ACC\_DEVICE\_TYPE** environment variable controls the default device type to use when exe-  
2598 cuting accelerator parallel, kernels, **and serial** regions, if the program has been compiled to use more  
2599 than one different type of device. The allowed values of this environment variable are implementation-  
2600 defined. See the release notes for currently-supported values of this environment variable.

Example:

```
setenv ACC_DEVICE_TYPE NVIDIA
export ACC_DEVICE_TYPE=NVIDIA
```

### 4.2. ACC\_DEVICE\_NUM

2601

2602 The **ACC\_DEVICE\_NUM** environment variable controls the default device number to use when  
2603 executing accelerator regions. The value of this environment variable must be a nonnegative integer  
2604 between zero and the number of devices of the desired type attached to the host. If the value is  
2605 greater than or equal to the number of devices attached, the behavior is implementation-defined.

Example:

```
setenv ACC_DEVICE_NUM 1
export ACC_DEVICE_NUM=1
```

### 4.3. ACC\_PROFLIB

2606

2607 The **ACC\_PROFLIB** environment variable specifies the profiling library. More details about the  
2608 evaluation at runtime is given in section 5.3.3 Runtime Dynamic Library Loading.

Example:

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



## 5. Profiling Interface

2609

2610 This chapter describes the OpenACC interface for tools that can be used for profile and trace data  
2611 collection. Therefore it provides a set of OpenACC-specific event callbacks that are triggered dur-  
2612 ing the application run. Currently, this interface does not support tools that employ asynchronous  
2613 sampling. In this chapter, the term *runtime* refers to the OpenACC runtime library. The term *library*  
2614 refers to the third party routines invoked at specified events by the OpenACC runtime.

2615 There are four steps for interfacing a *library* to the *runtime*. The first is to write the data collection  
2616 library callback routines. Section 5.1 Events describes the supported runtime events and the order  
2617 in which callbacks to the callback routines will occur. Section 5.2 Callbacks Signature describes  
2618 the signature of the callback routines for all events.

2619 The second is to use registration routines to register the data collection callbacks for the appropriate  
2620 events. The data collection and registration routines are then saved in a static or dynamic library  
2621 or shared object. The third is to load the *library* at runtime. The *library* may be statically linked  
2622 to the application or dynamically loaded by the application or by the *runtime*. This is described in  
2623 Section 5.3 Loading the Library.

2624 The fourth step is to invoke the registration routine to register the desired callbacks with the events.  
2625 This may be done explicitly by the application, if the library is statically linked with the application,  
2626 implicitly by including a call to the registration routine in a `.init` section, or by including an  
2627 initialization routine in the library if it is dynamically loaded by the *runtime*. This is described in  
2628 Section 5.4 Registering Event Callbacks.

2629 Subsequently, the *library* may collect information when the callback routines are invoked by the  
2630 *runtime* and process or store the acquired data.

### 5.1. Events

2631

2632 This section describes the events that are recognized by the runtime. Most events may have a start  
2633 and end callback routine, that is, a routine that is called just before the runtime code to handle  
2634 the event starts and another routine that is called just after the event is handled. The event names  
2635 and routine prototypes are available in the header file `acc_prof.h`, which is delivered with the  
2636 OpenACC implementation. Event names are prefixed with `acc_ev_`.

2637 The ordering of events must reflect the order in which the OpenACC runtime actually executes them,  
2638 i.e. if a runtime moves the enqueueing of data transfers or kernel launches outside the originating  
2639 clauses/constructs, it needs to issue the corresponding launch callbacks when they really occur. A  
2640 callback for a start event must always precede the matching end callback. The behavior of a tool  
2641 receiving a callback after the runtime shutdown callback is undefined.

2642 The events that the runtime supports can be registered with a callback and are defined in the enu-  
2643 meration type `acc_event_t`.

```
typedef enum acc_event_t{
    acc_ev_none = 0,
    acc_ev_device_init_start,
    acc_ev_device_init_end,
    acc_ev_device_shutdown_start,
    acc_ev_device_shutdown_end,
    acc_ev_runtime_shutdown,
    acc_ev_create,
    acc_ev_delete,
    acc_ev_alloc,
    acc_ev_free,
    acc_ev_enter_data_start,
    acc_ev_enter_data_end,
    acc_ev_exit_data_start,
    acc_ev_exit_data_end,
    acc_ev_update_start,
    acc_ev_update_end,
    acc_ev_compute_construct_start,
    acc_ev_compute_construct_end,
    acc_ev_enqueue_launch_start,
    acc_ev_enqueue_launch_end,
    acc_ev_enqueue_upload_start,
    acc_ev_enqueue_upload_end,
    acc_ev_enqueue_download_start,
    acc_ev_enqueue_download_end,
    acc_ev_wait_start,
    acc_ev_wait_end,
    acc_ev_last
}acc_event_t;
```

### 2644 5.1.1. Runtime Initialization and Shutdown

2645 No callbacks can be registered for the runtime initialization. Instead the initialization of the tool is  
2646 handled as described in Section 5.3 Loading the Library.

2647 The *runtime shutdown* event name is

```
acc_ev_runtime_shutdown
```

2648 The **acc\_ev\_runtime\_shutdown** event is triggered before the OpenACC runtime shuts down,  
2649 either because all devices have been shutdown by calls to the **acc\_shutdown** API routine, or at  
2650 the end of the program.

### 2651 5.1.2. Device Initialization and Shutdown

2652 The *device initialization* event names are

```
acc_ev_device_init_start
```



**acc\_ev\_device\_init\_end**

2653 These events are triggered when a device is being initialized by the OpenACC runtime. This may be  
2654 when the program starts, or may be later during execution when the program reaches an **acc\_init**  
2655 call or an OpenACC construct. The **acc\_ev\_device\_init\_start** is triggered before device  
2656 initialization starts and **acc\_ev\_device\_init\_end** after initialization is complete.

2657 The *device shutdown* event names are

**acc\_ev\_device\_shutdown\_start****acc\_ev\_device\_shutdown\_end**

2658 These events are triggered when a device is shut down, most likely by a call to the OpenACC  
2659 **acc\_shutdown** API routine. The **acc\_ev\_device\_shutdown\_start** is triggered before  
2660 the device shutdown process starts and **acc\_ev\_device\_shutdown\_end** after the device shut-  
2661 down is complete.

**5.1.3. Enter Data and Exit Data**

2662 The *enter data* and *exit data* event names are

**acc\_ev\_enter\_data\_start****acc\_ev\_enter\_data\_end****acc\_ev\_exit\_data\_start****acc\_ev\_exit\_data\_end**

2664 The **acc\_ev\_enter\_data\_start** and **acc\_ev\_enter\_data\_end** events are triggered at  
2665 **enter data** directives, entry to data constructs, and entry to implicit data regions such as those  
2666 generated by compute constructs. The **acc\_ev\_enter\_data\_start** event is triggered before  
2667 any *data allocation*, *data update*, or *wait* events that are associated with that directive or region  
2668 entry, and the **acc\_ev\_enter\_data\_end** is triggered after those events.

2669 The **acc\_ev\_exit\_data\_start** and **acc\_ev\_exit\_data\_end** events are triggered at **exit**  
2670 **data** directives, exit from **data** constructs, and exit from implicit data regions. The **acc\_ev\_exit\_data\_start**  
2671 event is triggered before any *data deallocation*, *data update*, or *wait* events associated with that di-  
2672 rective or region exit, and the **acc\_ev\_exit\_data\_end** event is triggered after those events.

2673 When the construct that triggers an *enter data* or *exit data* event was generated implicitly by the  
2674 compiler the **implicit** field in the event structure will be set to **1**. When the construct that  
2675 triggers these events was specified explicitly by the application code the **implicit** field in the  
2676 event structure will be set to **0**.

**5.1.4. Data Allocation**

2678 The *data allocation* event names are

**acc\_ev\_create****acc\_ev\_delete**

**acc\_ev\_alloc**  
**acc\_ev\_free**

2679 An **acc\_ev\_alloc** event is triggered when the OpenACC runtime allocates memory from the de-  
 2680 vice memory pool, and an **acc\_ev\_free** event is triggered when the runtime frees that memory.  
 2681 An **acc\_ev\_create** event is triggered when the OpenACC runtime associates device memory  
 2682 with host memory, such as for a data clause (**create**, **copyin**, **copy**, **copyout**) at entry to  
 2683 a data construct, compute construct, at an **enter data** directive, or in a call to a data API rou-  
 2684 tine (**acc\_copyin**, **acc\_create**, ...). An **acc\_ev\_create** event may be preceded by an  
 2685 **acc\_ev\_alloc** event, if newly allocated memory is used for this device data, or it may not, if  
 2686 the runtime manages its own memory pool. An **acc\_ev\_delete** event is triggered when the  
 2687 OpenACC runtime disassociates device memory from host memory, such as for a data clause at exit  
 2688 from a data construct, compute construct, at an **exit data** directive, or in a call to a data API  
 2689 routine (**acc\_copyout**, **acc\_delete**, ...). An **acc\_ev\_delete** event may be followed by  
 2690 an **acc\_ev\_free** event, if the disassociated device memory is freed, or it may not, if the runtime  
 2691 manages its own memory pool.

2692 When the action that generates a *data allocation* event was generated explicitly by the application  
 2693 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event  
 2694 is triggered because of a variable or array with implicitly-determined data attributes or otherwise  
 2695 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

### 2696 5.1.5. Data Construct

2697 The events for entering and leaving *data constructs* are mapped to *enter data* and *exit data* events  
 2698 as described in Section 5.1.3 Enter Data and Exit Data.

### 2699 5.1.6. Update Directive

2700 The *update directive* event names are

**acc\_ev\_update\_start**  
**acc\_ev\_update\_end**

2701 The **acc\_ev\_update\_start** event will be triggered at an **update** directive, before any *data*  
 2702 *update* or *wait* events that are associated with the update directive are carried out, and the corre-  
 2703 sponding **acc\_ev\_update\_end** event will be triggered after any of the associated events.

### 2704 5.1.7. Compute Construct

2705 The *compute construct* event names are

**acc\_ev\_compute\_construct\_start**  
**acc\_ev\_compute\_construct\_end**

2706 The **acc\_ev\_compute\_construct\_start** event is triggered at entry to a compute construct,  
 2707 before any *launch* events that are associated with entry to the compute construct. The **acc\_ev\_compute\_construct**

2708 event is triggered at the exit of the compute construct, after any *launch* events associated with exit  
2709 from the compute construct. If there are data clauses on the compute construct, those data clauses  
2710 may be treated as part of the compute construct, or as part of a data construct containing the compute  
2711 construct. The callbacks for data clauses must use the same line numbers as for the compute  
2712 construct events.

### 2713 5.1.8. Enqueue Kernel Launch

2714 The *launch* event names are

```
acc_ev_enqueue_launch_start
acc_ev_enqueue_launch_end
```

2715 The **acc\_ev\_enqueue\_launch\_start** event is triggered just before an accelerator compu-  
2716 tation is enqueued for execution on the device, and **acc\_ev\_enqueue\_launch\_end** is trig-  
2717 gered just after the computation is enqueued. Note that these events are synchronous with the  
2718 host enqueueing the computation to the device, not with the device executing the computation.  
2719 The **acc\_ev\_enqueue\_launch\_start** event callback routine is invoked just before the com-  
2720 putation is enqueued, not just before the computation starts execution. More importantly, the  
2721 **acc\_ev\_enqueue\_launch\_end** event callback routine is invoked after the computation is en-  
2722 queued, not after the computation finished executing.

2723 **Note:** Measuring the time between the start and end launch callbacks is often unlikely to be useful,  
2724 since it will only measure the time to manage the launch queue, not the time to execute the code on  
2725 the device.

### 2726 5.1.9. Enqueue Data Update (Upload and Download)

2727 The *data update* event names are

```
acc_ev_enqueue_upload_start
acc_ev_enqueue_upload_end
acc_ev_enqueue_download_start
acc_ev_enqueue_download_end
```

2728 The **\_start** events are triggered just before each upload (data copy from host to device) oper-  
2729 ation is or download (data copy from device to host) operation is enqueued for execution on the  
2730 device. The corresponding **\_end** events are triggered just after each upload or download operation  
2731 is enqueued.

2732 **Note:** Measuring the time between the start and end update callbacks is often unlikely to be useful,  
2733 since it will only measure the time to manage the enqueue operation, not the time to perform the  
2734 actual upload or download.

2735 When the action that generates a *data update* event was generated explicitly by the application  
2736 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event  
2737 is triggered because of a variable or array with implicitly-determined data attributes or otherwise  
2738 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

2739 **5.1.10. Wait**2740 The *wait* event names are

```

acc_ev_wait_start
acc_ev_wait_end

```

2741 An **acc\_ev\_wait\_start** will be triggered for each relevant queue before the host thread waits  
 2742 for that queue to be empty. A **acc\_ev\_wait\_end** will be triggered for each relevant queue after  
 2743 the host thread has determined that the queue is empty.

2744 Wait events occur when the host and device synchronize, either due to a **wait** directive or by a  
 2745 *wait* clause on a synchronous data construct, compute construct, or **enter data**, **exit data**,  
 2746 or **update** directive. For *wait* events triggered by an explicit synchronous **wait** directive or  
 2747 *wait* clause, the **implicit** field in the event structure will be **0**. For all other wait events, the  
 2748 **implicit** field in the event structure will be **1**.

2749 The OpenACC runtime need not trigger *wait* events for queues that have not been used in the  
 2750 program, and need not trigger *wait* events for queues that have not been used by this thread since  
 2751 the last *wait* operation. For instance, an **acc wait** directive with no arguments is defined to wait on  
 2752 all queues. If the program only uses the default (synchronous) queue and the queue associated with  
 2753 **async(1)** and **async(2)** then an **acc wait** directive may trigger *wait* events only for those  
 2754 three queues. If the implementation knows that no activities have been enqueued on the **async(2)**  
 2755 queue since the last *wait* operation, then the **acc wait** directive may trigger *wait* events only for  
 2756 the default queue and the **async(1)** queue.

2757 **5.2. Callbacks Signature**

2758 This section describes the signature of event callbacks. All event callbacks have the same signature.  
 2759 The routine prototypes are available in the header file **acc\_prof.h**, which is delivered with the  
 2760 OpenACC implementation.

2761 All callback routines have three arguments. The first argument is a pointer to a struct containing  
 2762 general information; the same struct type is used for all callback events. The second argument is  
 2763 a pointer to a struct containing information specific to that callback event; there is one struct type  
 2764 containing information for data events, **another struct type containing information for kernel launch**  
 2765 **events, and a third struct type for other events, containing essentially no information.** The third  
 2766 argument is a pointer to a struct containing information about the application programming interface  
 2767 (API) being used for the specific device. For NVIDIA CUDA devices, this contains CUDA-specific  
 2768 information; for OpenCL devices, this contains OpenCL-specific information. Other interfaces can  
 2769 be supported as they are added by implementations. The prototype for a callback routine is:

```

typedef void (*acc_prof_callback)
(acc_prof_info*, acc_event_info*, acc_api_info*);

```

2770 In the descriptions, the datatype **ssize\_t** means a signed 32-bit integer for a 32-bit binary and  
 2771 a 64-bit integer for a 64-bit binary, the datatype **size\_t** means an unsigned 32-bit integer for a

2772 32-bit binary and a 64-bit integer for a 64-bit binary, and the datatype **int** means a 32-bit integer  
 2773 for both 32-bit and 64-bit binaries. A null pointer is the pointer with value zero.

### 2774 5.2.1. First Argument: General Information

2775 The first argument is a pointer to the **acc\_prof\_info** struct type:

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

2776 The fields are described below.

2777 • **acc\_event\_t event\_type** - The event type that triggered this callback. The datatype  
 2778 is the enumeration type **acc\_event\_t**, described in the previous section. This allows the  
 2779 same callback routine to be used for different events.

2780 • **int valid\_bytes** - The number of valid bytes in this struct. This allows a library to inter-  
 2781 face with newer runtimes that may add new fields to the struct at the end while retaining com-  
 2782 patibility with older runtimes. A runtime must fill in the **event\_type** and **valid\_bytes**  
 2783 fields, and must fill in values for all fields with offset less than **valid\_bytes**. The value of  
 2784 **valid\_bytes** for a struct is recursively defined as:

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

2785 • **int version** - A version number; the value of **\_OPENACC**.

2786 • **acc\_device\_t device\_type** - The device type corresponding to this event. The datatype  
 2787 is **acc\_device\_t**, an enumeration type of all the supported accelerator device types, de-  
 2788 fined in **openacc.h**.

2789 • **int device\_number** - The device number. Each device is numbered, typically starting at  
 2790 device zero. For applications that use more than one device type, the device numbers may be  
 2791 unique across all devices or may be unique only across all devices of the same device type.

2792 • **int thread\_id** - The host thread ID making the callback. Host threads are given unique  
 2793 thread ID numbers typically starting at zero. This is not necessarily the same as the OpenMP  
 2794 thread number.

- 2795 • **ssize\_t async** - The value of the **async()** clause for the directive that triggered this  
2796 callback.
- 2797 • **ssize\_t async\_queue** - If the runtime uses a limited number of asynchronous queues,  
2798 this field contains the internal asynchronous queue number used for the event.
- 2799 • **const char\* src\_file** - A pointer to null-terminated string containing the name of or  
2800 path to the source file, if known, or a null pointer if not. If the library wants to save the source  
2801 file name, it should allocate memory and copy the string.
- 2802 • **const char\* func\_name** - A pointer to a null-terminated string containing the name of  
2803 the function in which the event occurred, if known, or a null pointer if not. If the library wants  
2804 to save the function name, it should allocate memory and copy the string.
- 2805 • **int line\_no** - The line number of the directive or program construct or the starting line  
2806 number of the OpenACC construct corresponding to the event. A negative or zero value  
2807 means the line number is not known.
- 2808 • **int end\_line\_no** - For an OpenACC construct, this contains the line number of the end  
2809 of the construct. A negative or zero value means the line number is not known.
- 2810 • **int func\_line\_no** - The line number of the first line of the function named in **func\_name**.  
2811 A negative or zero value means the line number is not known.
- 2812 • **int func\_end\_line\_no** - The last line number of the function named in **func\_name**.  
2813 A negative or zero value means the line number is not known.

### 2814 5.2.2. Second Argument: Event-Specific Information

2815 The second argument is a pointer to the **acc\_event\_info** union type.

```

2816 typedef union acc_event_info{
2817     acc_event_t event_type;
2818     acc_data_event_info data_event;
2819     acc_launch_event_info launch_event;
2820     acc_other_event_info other_event;
2821 }acc_event_info;

```

2816 The **event\_type** field selects which union member to use. The first five members of each union  
2817 member are identical. The second through fifth members of each union member (**valid\_bytes**,  
2818 **parent\_construct**, **implicit**, and **tool\_info**) have the same semantics for all event  
2819 types:

- 2820 • **int valid\_bytes** - The number of valid bytes in the respective struct. (This field is similar  
2821 used as discussed in Section 5.2.1 First Argument: General Information.)
- 2822 • **acc\_construct\_t parent\_construct** - This field describes the type of construct  
2823 that caused the event to be emitted. The possible values for this field are defined by the  
2824 **acc\_construct\_t** enum, described at the end of this section.
- 2825 • **int implicit** - This field is set to 1 for any implicit event, such as an implicit wait at  
2826 a synchronous data construct or synchronous enter data, exit data or update directive. This

2827 field is set to zero when the event is triggered by an explicit directive or call to a runtime API  
2828 routine.

- 2829 • **void\* tool\_info** - This field is used to pass tool-specific information from a **\_start**  
2830 event to the matching **\_end** event. For a **\_start** event callback, this field will be initialized  
2831 to a null pointer. The value of this field for a **\_end** event will be the value returned by  
2832 the library in this field from the matching **\_start** event callback, if there was one, or null  
2833 otherwise. For events that are neither **\_start** or **\_end** events, this field will be null.

## 2834 Data Events

2835 For a data event, as noted in the event descriptions, the second argument will be a pointer to the  
2836 **acc\_data\_event\_info** struct.

```
typedef struct acc_data_event_info{
    acc_event_t event_type;
    int valid_bytes;
    acc_construct_t parent_construct;
    int implicit;
    void* tool_info;
    const char* var_name;
    size_t bytes;
    const void* host_ptr;
    const void* device_ptr;
}acc_data_event_info;
```

2837 The fields specific for a data event are:

- 2838 • **acc\_event\_t event\_type** - The event type that triggered this callback. The events that  
2839 use the **acc\_data\_event\_info** struct are:

```
acc_ev_enqueue_upload_start
acc_ev_enqueue_upload_end
acc_ev_enqueue_download_start
acc_ev_enqueue_download_end
acc_ev_create
acc_ev_delete
acc_ev_alloc
acc_ev_free
```

- 2840 • **const char\* var\_name** - A pointer to null-terminated string containing the name of the  
2841 variable for which this event is triggered, if known, or a null pointer if not. If the library wants  
2842 to save the variable name, it should allocate memory and copy the string.
- 2843 • **size\_t bytes** - The number of bytes for the data event.
- 2844 • **const void\* host\_ptr** - If available and appropriate for this event, this is a pointer to  
2845 the host data.
- 2846 • **const void\* device\_ptr** - If available and appropriate for this event, this is a pointer  
2847 to the corresponding device data.

2848 **Launch Events**

2849 For a launch event, as noted in the event descriptions, the second argument will be a pointer to the  
2850 `acc_launch_event_info` struct.

```

typedef struct acc_launch_event_info{
    acc_event_t event_type;
    int valid_bytes;
    acc_construct_t parent_construct;
    int implicit;
    void* tool_info;
    const char* kernel_name;
    size_t num_gangs, num_workers, vector_length;
}acc_launch_event_info;

```

2851 The fields specific for a launch event are:

2852 • `acc_event_t event_type` - The event type that triggered this callback. The events that  
2853 use the `acc_launch_event_info` struct are:

```

    acc_ev_enqueue_launch_start
    acc_ev_enqueue_launch_end

```

2854 • `const char* kernel_name` - A pointer to null-terminated string containing the name of  
2855 the kernel being launched, if known, or a null pointer if not. If the library wants to save the  
2856 kernel name, it should allocate memory and copy the string.

2857 • `size_t num_gangs, num_workers, vector_length` - The number of gangs, work-  
2858 ers and vector lanes created for this kernel launch.

2859 **Other Events**

2860 For any event that does not use the `acc_data_event_info` or `acc_launch_event_info`  
2861 struct, the second argument to the callback routine will be a pointer to `acc_other_event_info`  
2862 struct.

```

typedef struct acc_other_event_info{
    acc_event_t event_type;
    int valid_bytes;
    acc_construct_t parent_construct;
    int implicit;
    void* tool_info;
}acc_other_event_info;

```

2863 **Parent Construct Enumeration**

2864 All event structures contain a `parent_construct` member that describes the type of construct  
2865 that caused the event to be emitted. The purpose of this field is to provide a means to identify



2866 the type of construct emitting the event in the cases where an event may be emitted by multi-  
 2867 ple construct types, such as is the case with data and wait events. The possible values for the  
 2868 **parent\_construct** field are defined in the enumeration type **acc\_construct\_t**. In the  
 2869 case of combined directives, the outermost construct of the combined construct should be specified  
 2870 as the **parent\_construct**. If the event was emitted as the result of the application making a  
 2871 call to the runtime api, the value will be **acc\_construct\_runtime\_api**.

```
typedef enum acc_construct_t{
    acc_construct_parallel = 0,
    acc_construct_kernels,
    acc_construct_loop,
    acc_construct_data,
    acc_construct_enter_data,
    acc_construct_exit_data,
    acc_construct_host_data,
    acc_construct_atomic,
    acc_construct_declare,
    acc_construct_init,
    acc_construct_shutdown,
    acc_construct_set,
    acc_construct_update,
    acc_construct_routine,
    acc_construct_wait,
    acc_construct_runtime_api,
    acc_construct_serial
}acc_construct_t;
```

### 2872 5.2.3. Third Argument: API-Specific Information

2873 The third argument is a pointer to the **acc\_api\_info** struct type, shown here.

```
typedef union acc_api_info{
    acc_device_api device_api;
    int valid_bytes;
    acc_device_t device_type;
    int vendor;
    const void* device_handle;
    const void* context_handle;
    const void* async_handle;
}acc_api_info;
```

2874 The fields are described below:

- 2875 • **acc\_device\_api device\_api** - The API in use for this device. The data type is the  
 2876 enumeration **acc\_device\_api**, which is described later in this section.
- 2877 • **int valid\_bytes** - The number of valid bytes in this struct. See the discussion above in  
 2878 Section 5.2.1 First Argument: General Information.

- 2879 • **acc\_device\_t device\_type** - The device type; the datatype is **acc\_device\_t**, de-  
2880 fined in **openacc.h**.
- 2881 • **int vendor** - An identifier to identify the OpenACC vendor; contact your vendor to deter-  
2882 mine the value used by that vendor's runtime.
- 2883 • **const void\* device\_handle** - If applicable, this will be a pointer to the API-specific  
2884 device information.
- 2885 • **const void\* context\_handle** - If applicable, this will be a pointer to the API-specific  
2886 context information.
- 2887 • **const void\* async\_handle** - If applicable, this will be a pointer to the API-specific  
2888 async queue information.

2889 According to the value of **device\_api** a library can cast the pointers of the fields **device\_handle**,  
2890 **context\_handle** and **async\_handle** to the respective device API type. The following device  
2891 APIs are defined in this interface:

```

typedef enum acc_device_api{
    acc_device_api_none = 0,    /* no device API */
    acc_device_api_cuda,       /* CUDA driver API */
    acc_device_api_opencl,     /* OpenCL API */
    acc_device_api_coi,        /* COI API */
    acc_device_api_other       /* other device API */
}acc_device_api;

```

## 2892 5.3. Loading the Library

2893 This section describes how a tools library is loaded when the program is run. Four methods are  
2894 described.

- 2895 • A tools library may be linked with the program, as any other library is linked, either as a  
2896 static library or a dynamic library, and the runtime will call a predefined library initialization  
2897 routine that will register the event callbacks.
- 2898 • The OpenACC runtime implementation may support a dynamic tools library, such as a shared  
2899 object for Linux or OS/X, or a DLL for Windows, which is then dynamically loaded at runtime  
2900 under control of the environment variable **ACC\_PROFLIB**.
- 2901 • Some implementations where the OpenACC runtime is itself implemented as a dynamic li-  
2902 brary may support adding a tools library using the **LD\_PRELOAD** feature in Linux.
- 2903 • A tools library may be linked with the program, as in the first option, and the application itself  
2904 can call a library initialization routine that will register the event callbacks.

2905 Callbacks are registered with the runtime by calling **acc\_prof\_register** for each event as  
2906 described in Section 5.4 Registering Event Callbacks. The prototype for **acc\_prof\_register**  
2907 is:

```

extern void acc_prof_register
    (acc_event_t event_type, acc_prof_callback cb,

```

```
acc_register_t info);
```

2908 The first argument to **acc\_prof\_register** is the event for which a callback is being registered  
2909 (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*);
```

2910 The third argument is usually zero (or **acc\_reg**). See Section 5.4.2 Disabling and Enabling Callbacks  
2911 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;
```

2912 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);
```

2913 As shown in this example, the same routine (**prof\_data**) can be registered for multiple events.  
2914 The routine can use the **event\_type** field in the **acc\_prof\_info** structure to determine for  
2915 what event it was invoked.

### 2916 5.3.1. Library Registration

2917 The OpenACC runtime will invoke **acc\_register\_library**, passing the addresses of the reg-  
2918 istration routines **acc\_prof\_register** and **acc\_prof\_unregister**, in case that routine  
2919 comes from a dynamic library. In the third argument it passes the address of the lookup routine  
2920 **acc\_prof\_lookup** to obtain the addresses of inquiry functions. No inquiry functions are de-  
2921 fined in this profiling interface, but we preserve this argument for future support of sampling-based  
2922 tools.

2923 Typically, the OpenACC runtime will include a *weak* definition of **acc\_register\_library**,  
2924 which does nothing and which will be called when there is no tools library. In this case, the library  
2925 can save the addresses of these routines and/or make registration calls to register any appropriate  
2926 callbacks. The prototype for **acc\_register\_library** is:

```
extern void acc_register_library
    (acc_prof_reg register, acc_prof_reg unregister,
    acc_prof_lookup_func lookup);
```

2927 The first two arguments of this routine are of type:

```
typedef void (*acc_prof_reg)
    (acc_event_t event_type, acc_prof_callback cb,
     acc_register_t info);
```

2928 The third argument passes the address to the lookup function `acc_prof_lookup` to obtain the  
2929 address of interface functions. It is of type:

```
typedef void (*acc_query_fn) ();
typedef acc_query_fn (*acc_prof_lookup_func)
    (const char* acc_query_fn_name);
```

2930 The argument of the lookup function is a string with the name of the inquiry function. There are no  
2931 inquiry functions defined for this interface.

### 2932 5.3.2. Statically-Linked Library Initialization

2933 A tools library can be compiled and linked directly into the application. If the library provides an  
2934 external routine `acc_register_library` as specified in Section 5.3.1 Library Registration, the  
2935 runtime will invoke that routine to initialize the library.

2936 The sequence of events is:

- 2937 1. The runtime invokes the `acc_register_library` routine from the library.
- 2938 2. The `acc_register_library` routine calls `acc_prof_register` for each event to  
2939 be monitored.
- 2940 3. `acc_prof_register` records the callback routines.
- 2941 4. The program runs, and your callback routines are invoked at the appropriate events.

2942 In this mode, only one tool library is supported.

### 2943 5.3.3. Runtime Dynamic Library Loading

2944 A common case is to build the tools library as a dynamic library (shared object for Linux or OS/X,  
2945 DLL for Windows). In that case, you can have the OpenACC runtime load the library during initial-  
2946 ization. This allows you to enable runtime profiling without rebuilding or even relinking your ap-  
2947 plication. The dynamic library must implement a registration routine `acc_register_library`  
2948 as specified in Section 5.3.1 Library Registration.

2949 The user may set the environment variable `ACC_PROFLIB` to the path to the library will tell the  
2950 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
```

2951 When the OpenACC runtime initializes, it will read the **ACC\_PROFLIB** environment variable (with  
2952 **getenv**). The runtime will open the dynamic library (using **dlopen** or **LoadLibraryA**); if  
2953 the library cannot be opened, the runtime may abort, or may continue execution with or with-  
2954 out an error message. If the library is successfully opened, the runtime will get the address of  
2955 the **acc\_register\_library** routine (using **dlsym** or **GetProcAddress**). If this routine  
2956 is resolved in the library, it will be invoked passing in the addresses of the registration routine  
2957 **acc\_prof\_register**, the deregistration routine **acc\_prof\_unregister**, and the lookup  
2958 routine **acc\_prof\_lookup**. The registration routine in your library, **acc\_register\_library**,  
2959 should register the callbacks by calling the **register** argument, and should save the addresses of  
2960 the arguments (**register**, **unregister**, and **lookup**) for later use, if needed.

2961 The sequence of events is:

- 2962 1. Initialization of the OpenACC runtime.
- 2963 2. OpenACC runtime reads **ACC\_PROFLIB**.
- 2964 3. OpenACC runtime loads the library.
- 2965 4. OpenACC runtime calls the **acc\_register\_library** routine in that library.
- 2966 5. Your **acc\_register\_library** routine calls **acc\_prof\_register** for each event to  
2967 be monitored.
- 2968 6. **acc\_prof\_register** records the callback routines.
- 2969 7. The program runs, and your callback routines are invoked at the appropriate events.

2970 If supported, paths to multiple dynamic libraries may be specified in the **ACC\_PROFLIB** environ-  
2971 ment variable, separated by semicolons (;). The OpenACC runtime will open these libraries and in-  
2972 voke the **acc\_register\_library** routine for each, in the order they appear in **ACC\_PROFLIB**.

#### 2973 5.3.4. Preloading with LD\_PRELOAD

2974 The implementation may also support dynamic loading of a tools library using the **LD\_PRELOAD**  
2975 feature available in some systems. In such an implementation, you need only specify your tools  
2976 library path in the **LD\_PRELOAD** environment variable before executing your program. The Open-  
2977 ACC runtime will invoke the **acc\_register\_library** routine in your tools library at initial-  
2978 ization time. This requires that the OpenACC runtime include a dynamic library with a default  
2979 (empty) implementation of **acc\_register\_library** that will be invoked in the normal case  
2980 where there is no **LD\_PRELOAD** setting. If an implementation only supports static linking, or if the  
2981 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
```

2982 The sequence of events is:

- 2983 1. The operating system loader loads the library specified in **LD\_PRELOAD**.
- 2984 2. The call to **acc\_register\_library** in the OpenACC runtime is resolved to the routine  
2985 in the loaded tools library.
- 2986 3. OpenACC runtime calls the **acc\_register\_library** routine in that library.
- 2987 4. Your **acc\_register\_library** routine calls **acc\_prof\_register** for each event to  
2988 be monitored.
- 2989 5. **acc\_prof\_register** records the callback routines.
- 2990 6. The program runs, and your callback routines are invoked at the appropriate events.

2991 In this mode, only a single tools library is supported, since only one **acc\_register\_library**  
2992 initialization routine will get resolved by the dynamic loader.

### 2993 5.3.5. Application-Controlled Initialization

2994 An alternative to default initialization is to have the application itself call the library initialization  
2995 routine, which then calls **acc\_prof\_register** for each appropriate event. The library may be  
2996 statically linked to the application or your application may dynamically load the library.

2997 The sequence of events is:

- 2998 1. Your application calls the library initialization routine.
- 2999 2. The library initialization routine calls **acc\_prof\_register** for each event to be moni-  
3000 tored.
- 3001 3. **acc\_prof\_register** records the callback routines.
- 3002 4. The program runs, and your callback routines are invoked at the appropriate events.

3003 In this mode, multiple tools libraries can be supported, with each library initialization routine in-  
3004 voked by the application.

## 3005 5.4. Registering Event Callbacks

3006 This section describes how to register and unregister callbacks, temporarily disabling and enabling  
3007 callbacks, the behavior of dynamic registration and unregistration, and requirements on an Open-  
3008 ACC implementation to correctly support the interface.

### 3009 5.4.1. Event Registration and Unregistration

3010 The library must call the registration routine `acc_prof_register` to register each callback  
3011 with the runtime. A simple example:

```
extern void prof_data(acc_prof_info* profinfo,
                    acc_event_info* eventinfo, acc_api_info* apiinfo);
extern void prof_launch(acc_prof_info* profinfo,
                      acc_event_info* eventinfo, acc_api_info* apiinfo);
...
void acc_register_library(){
    acc_prof_register(acc_ev_enqueue_upload_start, prof_data, 0);
    acc_prof_register(acc_ev_enqueue_download_start, prof_data, 0);
    acc_prof_register(acc_ev_enqueue_launch_start, prof_launch, 0);
}
```

3012 In this example the `prof_data` routine will be invoked for each data upload and download event,  
3013 and the `prof_launch` routine will be invoked for each launch event. The `prof_data` routine  
3014 might start out with:

```
void prof_data(acc_prof_info* profinfo,
              acc_event_info* eventinfo, acc_api_info* apiinfo){
    acc_data_event_info* datainfo;
    datainfo = (acc_data_event_info*)eventinfo;
    switch( datainfo->event_type ){
        case acc_ev_enqueue_upload_start :
            ...
    }
}
```

### 3015 Multiple Callbacks

3016 Multiple callback routines can be registered on the same event:

```
acc_prof_register(acc_ev_enqueue_upload_start, prof_data, 0);
acc_prof_register(acc_ev_enqueue_upload_start, prof_up, 0);
```

3017 For most events, the callbacks will be invoked in the order in which they are registered. However,  
3018 *end* events, named `acc_ev_..._end`, invoke callbacks in the reverse order. Essentially, each  
3019 event has an ordered list of callback routines. A new callback routine is appended to the tail of the  
3020 list for that event. For most events, that list is traversed from the head to the tail, but for *end* events,  
3021 the list is traversed from the tail to the head.

3022 If a callback is registered, then later unregistered, then later still registered again, the second regis-  
3023 tration is considered to be a new callback, and the callback routine will then be appended to the tail  
3024 of the callback list for that event.

3025 **Unregistering**

3026 A matching call to `acc_prof_unregister` will remove that routine from the list of callback  
 3027 routines for that event.

```

acc_prof_register(acc_ev_enqueue_upload_start, prof_data, 0);
// prof_data is on the callback list for acc_ev_enqueue_upload_start
...
acc_prof_unregister(acc_ev_enqueue_upload_start, prof_data, 0);
// prof_data is removed from the callback list
// for acc_ev_enqueue_upload_start

```

3028 Each entry on the callback list must also have a *ref* count. This keeps track of how many times  
 3029 this routine was added to this event's callback list. If a routine is registered *n* times, it must be  
 3030 unregistered *n* times before it is removed from the list. Note that if a routine is registered multiple  
 3031 times for the same event, its *ref* count will be incremented with each registration, but it will only be  
 3032 invoked once for each event instance.

3033 **5.4.2. Disabling and Enabling Callbacks**

3034 A callback routine may be temporarily disabled on the callback list for an event, then later re-  
 3035 enabled. The behavior is slightly different than unregistering and later re-registering that event.  
 3036 When a routine is disabled and later re-enabled, the routine's position on the callback list for that  
 3037 event is preserved. When a routine is unregistered and later re-registered, the routine's position on  
 3038 the callback list for that event will move to the tail of the list. Also, unregistering a callback must be  
 3039 done *n* times if the callback routine was registered *n* times. In contrast, disabling, and enabling an  
 3040 event sets a toggle. Disabling a callback will immediately reset the toggle and disable calls to that  
 3041 routine for that event, even if it was enabled multiple times. Enabling a callback will immediately  
 3042 set the toggle and enable calls to that routine for that event, even if it was disabled multiple times.  
 3043 Registering a new callback initially sets the toggle.

3044 A call to `acc_prof_unregister` with a value of `acc_toggle` as the third argument will dis-  
 3045 able callbacks to the given routine. A call to `acc_prof_register` with a value of `acc_toggle`  
 3046 as the third argument will enable those callbacks.

```

acc_prof_unregister(acc_ev_enqueue_upload_start,
    prof_data, acc_toggle);
// prof_data is disabled
...
acc_prof_register(acc_ev_enqueue_upload_start,
    prof_data, acc_toggle);
// prof_data is re-enabled

```

3047 A call to either `acc_prof_unregister` or `acc_prof_register` to disable or enable a call-  
 3048 back when that callback is not currently registered for that event will be ignored with no error.

3049 All callbacks for an event may be disabled (and re-enabled) by passing `NULL` to the second argument  
 3050 and `acc_toggle` to the third argument of `acc_prof_unregister` (and `acc_prof_register`).



3051 This sets a toggle for that event, which is distinct from the toggle for each callback for that event.  
 3052 While the event is disabled, no callbacks for that event will be invoked. Callbacks for that event can  
 3053 be registered, unregistered, enabled, and disabled while that event is disabled, but no callbacks will  
 3054 be invoked for that event until the event itself is enabled. Initially, all events are enabled.

```

acc_prof_unregister(acc_ev_enqueue_upload_start,
    prof_data, acc_toggle);
// prof_data is disabled
...
acc_prof_unregister(acc_ev_enqueue_upload_start,
    NULL, acc_toggle);
// acc_ev_enqueue_upload_start callbacks are disabled
...
acc_prof_register(acc_ev_enqueue_upload_start,
    prof_data, acc_toggle);
// prof_data is re-enabled, but
// acc_ev_enqueue_upload_start callbacks still disabled
...
acc_prof_register(acc_ev_enqueue_upload_start, prof_up, 0);
// prof_up is registered and initially enabled, but
// acc_ev_enqueue_upload_start callbacks still disabled
...
acc_prof_register(acc_ev_enqueue_upload_start,
    NULL, acc_toggle);
// acc_ev_enqueue_upload_start callbacks are enabled

```

3055 Finally, all callbacks can be disabled (and enabled) by passing the argument list (**0**, **NULL**,  
 3056 **acc\_toggle**) to **acc\_prof\_unregister** (and **acc\_prof\_register**). This sets a global  
 3057 toggle disabling all callbacks, which is distinct from the toggle enabling callbacks for each event and  
 3058 the toggle enabling each callback routine. The behavior of passing zero as the first argument and a  
 3059 non-**NULL** value as the second argument to **acc\_prof\_unregister** or **acc\_prof\_register**  
 3060 is not defined, and may be ignored by the runtime without error.

3061 All callbacks can be disabled (or enabled) for just the current thread by passing the argument list  
 3062 (**0**, **NULL**, **acc\_toggle\_per\_thread**) to **acc\_prof\_unregister** (and **acc\_prof\_register**).  
 3063 This is the only thread-specific interface to **acc\_prof\_register** and **acc\_prof\_unregister**,  
 3064 all other calls to register, unregister, enable, or disable callbacks affect all threads in the application.

## 3065 5.5. Advanced Topics

3066 This section describes advanced topics such as dynamic registration and changes of the execution  
 3067 state for callback routines as well as the runtime and tool behavior for multiple host threads.

### 3068 5.5.1. Dynamic Behavior

3069 Callback routines may be registered or unregistered, enabled or disabled at any point in the execution  
3070 of the program. Calls may appear in the library itself, during the processing of an event. The  
3071 OpenACC runtime must allow for this case, where the callback list for an event is modified while  
3072 that event is being processed.

#### 3073 Dynamic Registration and Unregistration

3074 Calls to **acc\_register** and **acc\_unregister** may occur at any point in the application. A  
3075 callback routine can be registered or unregistered from a callback routine, either the same routine  
3076 or another routine, for a different event or the same event for which the callback was invoked. If a  
3077 callback routine is registered for an event while that event is being processed, then the new callback  
3078 routine will be added to the tail of the list of callback routines for this event. Some events (the  
3079 **\_end**) events process the callback routines in reverse order, from the tail to the head. For those  
3080 events, adding a new callback routine will not cause the new routine to be invoked for this instance  
3081 of the event. The other events process the callback routines in registration order, from the head to  
3082 the tail. Adding a new callback routine for such a event will cause the runtime to invoke that newly  
3083 registered callback routine for this instance of the event. Both the runtime and the library must  
3084 implement and expect this behavior.

3085 If an existing callback routine is unregistered for an event while that event is being processed, that  
3086 callback routine is removed from the list of callbacks for this event. For any event, if that callback  
3087 routine had not yet been invoked for this instance of the event, it will not be invoked.

3088 Registering and unregistering a callback routine is a global operation and affects all threads, in a  
3089 multithreaded application. See Section 5.4.1 Multiple Callbacks.

#### 3090 Dynamic Enabling and Disabling

3091 Calls to **acc\_register** and **acc\_unregister** to enable and disable a specific callback for  
3092 an event, enable or disable all callbacks for an event, or enable or disable all callbacks may occur  
3093 at any point in the application. A callback routine can be enabled or disabled from a callback  
3094 routine, either the same routine or another routine, for a different event or the same event for which  
3095 the callback was invoked. If a callback routine is enabled for an event while that event is being  
3096 processed, then the new callback routine will be immediately enabled. If it appears on the list of  
3097 callback routines closer to the head (for **\_end** events) or closer to the tail (for other events), that  
3098 newly-enabled callback routine will be invoked for this instance of this event, unless it is disabled  
3099 or unregistered before that callback is reached.

3100 If a callback routine is disabled for an event while that event is being processed, that callback routine  
3101 is immediately disabled. For any event, if that callback routine had not yet been invoked for this in-  
3102 stance of the event, it will not be invoked, unless it is enabled before that callback routine is reached  
3103 in the list of callbacks for this event. If all callbacks for an event are disabled while that event is  
3104 being processed, or all callbacks are disabled for all events while an event is being processed, then  
3105 when this callback routine returns, no more callbacks will be invoked for this instance of the event.

3106 Registering and unregistering a callback routine is a global operation and affects all threads, in a  
3107 multithreaded application. See Section 5.4.1 Multiple Callbacks.

### 3108 5.5.2. OpenACC Events During Event Processing

3109 OpenACC events may occur during event processing. This may be because of OpenACC API rou-  
3110 tine calls or OpenACC constructs being reached during event processing, or because of multiple host  
3111 threads executing asynchronously. Both the OpenACC runtime and the tool library must implement  
3112 the proper behavior.

### 3113 5.5.3. Multiple Host Threads

3114 Many programs that use OpenACC also use multiple host threads, such as programs using the  
3115 OpenMP API. The appearance of multiple host threads affects both the OpenACC runtime and the  
3116 tools library.

### 3117 Runtime Support for Multiple Threads

3118 The OpenACC runtime must be thread-safe, and the OpenACC runtime implementation of this  
3119 tools interface must also be thread-safe. All threads use the same set of callbacks for all events, so  
3120 registering a callback from one thread will cause all threads to execute that callback. This means that  
3121 managing the callback lists for each event must be protected from multiple simultaneous updates.  
3122 This includes adding a callback to the tail of the callback list for an event, removing a callback from  
3123 the list for an event, and incrementing or decrementing the *ref* count for a callback routine for an  
3124 event.

3125 In addition, one thread may register, unregister, enable, or disable a callback for an event while  
3126 another thread is processing the callback list for that event asynchronously. The exact behavior may  
3127 be dependent on the implementation, but some behaviors are expected and others are disallowed.  
3128 In the following examples, there are three callbacks, A, B, and C, registered for event E in that  
3129 order, where callbacks A and B are enabled and callback C is temporarily disabled. Thread T1 is  
3130 dynamically modifying the callbacks for event E while thread T2 is processing an instance of event  
3131 E.

- 3132 • Suppose thread T1 unregisters or disables callback A for event E. Thread T2 may or may not  
3133 invoke callback A for this event instance, but it must invoke callback B; if it invokes callback  
3134 A, that must precede the invocation of callback B.
- 3135 • Suppose thread T1 unregisters or disables callback B for event E. Thread T2 may or may not  
3136 invoke callback B for this event instance, but it must invoke callback A; if it invokes callback  
3137 B, that must follow the invocation of callback A.
- 3138 • Suppose thread T1 unregisters or disables callback A and then unregisters or disables callback  
3139 B for event E. Thread T2 may or may not invoke callback A and may or may not invoke  
3140 callback B for this event instance, but if it invokes both callbacks, it must invoke callback A  
3141 before it invokes callback B.
- 3142 • Suppose thread T1 unregisters or disables callback B and then unregisters or disables callback  
3143 A for event E. Thread T2 may or may not invoke callback A and may or may not invoke  
3144 callback B for this event instance, but if it invokes callback B, it must have invoked callback  
3145 A for this event instance.
- 3146 • Suppose thread T1 is registering a new callback D for event E. Thread T2 may or may not

3147        invoke callback D for this event instance, but it must invoke both callbacks A and B. If it  
3148        invokes callback D, that must follow the invocations of A and B.

- 3149        • Suppose thread T1 is enabling callback C for event E. Thread T2 may or may not invoke  
3150        callback C for this event instance, but it must invoke both callbacks A and B. If it invokes  
3151        callback C, that must follow the invocations of A and B.

3152        The `acc_prof_info` struct has a `thread_id` field, which the runtime must set to a unique  
3153        value for each host thread, though it need not be the same as the OpenMP threadnum value.

### 3154        **Library Support for Multiple Threads**

3155        The tool library must also be thread-safe. The callback routine will be invoked in the context of the  
3156        thread that reaches the event. The library may receive a callback from a thread T2 while it's still  
3157        processing a callback, from the same event type or from a different event type, from another thread  
3158        T1. The `acc_prof_info` struct has a `thread_id` field, which the runtime must set to a unique  
3159        value for each host thread.

3160        If the tool library uses dynamic callback registration and unregistration, or callback disabling and  
3161        enabling, recall that unregistering or disabling an event callback from one thread will unregister or  
3162        disable that callback for all threads, and registering or enabling an event callback from any thread  
3163        will register or enable it for all threads. If two or more threads register the same callback for the  
3164        same event, the behavior is the same as if one thread registered that callback multiple times; see  
3165        Section 5.4.1 Multiple Callbacks. The `acc_unregister` routine must be called as many times  
3166        as `acc_register` for that callback/event pair in order to totally unregister it. If two threads  
3167        register two different callback routines for the same event, unless the order of the registration calls  
3168        is guaranteed by some synchronization method, the order in which the runtime sees the registration  
3169        may differ for multiple runs, meaning the order in which the callbacks occur will differ as well.

## 3170 6. Glossary

3171 Clear and consistent terminology is important in describing any programming model. We define  
3172 here the terms you must understand in order to make effective use of this document and the associ-  
3173 ated programming model.

3174 **Accelerator** – a special-purpose co-processor attached to a CPU and to which the CPU can offload  
3175 data and compute kernels to perform compute-intensive calculations.

3176 **Accelerator routine** – a C or C++ function or Fortran subprogram compiled for the accelerator  
3177 with the **routine** directive.

3178 **Accelerator thread** – a thread of execution that executes on the accelerator; a single vector lane of  
3179 a single worker of a single gang.

3180 **Aggregate datatype** – an array or structure datatype, or any non-scalar datatype. In Fortran, aggre-  
3181 gate datatypes include arrays and derived types. In C, aggregate datatypes include fixed size arrays,  
3182 targets of pointers, structs, and unions. In C++, aggregate datatypes include fixed size arrays, targets  
3183 of pointers, classes, structs, and unions.

3184 **Aggregate variables** – an array or structure variable, or a variable of any non-scalar datatype.

3185 **Async-argument** – an *async-argument* is a nonnegative scalar integer expression (*int* for C or C++,  
3186 *integer* for Fortran), or one of the special async values **acc\_async\_noval** or **acc\_async\_sync**.

3187 **Barrier** – a type of synchronization where all parallel execution units or threads must reach the  
3188 barrier before any execution unit or thread is allowed to proceed beyond the barrier; modeled after  
3189 the starting barrier on a horse race track.

3190 **Compute intensity** – for a given loop, region, or program unit, the ratio of the number of arithmetic  
3191 operations performed on computed data divided by the number of memory transfers required to  
3192 move that data between two levels of a memory hierarchy.

3193 **Construct** – a directive and the associated statement, loop, or structured block, if any.

3194 **Compute construct** – a *parallel construct*, *kernels construct*, or *serial construct*.

3195 **Compute region** – a *parallel region*, *kernels region*, or *serial region*.

3196 **CUDA** – the CUDA environment from NVIDIA is a C-like programming environment used to  
3197 explicitly control and program an NVIDIA GPU.

3198 **Current device** – the device represented by the *acc-device-type-var* and *acc-device-num-var* ICVs

3199 **Current device type** – the device type represented by the *acc-device-type-var* ICV

3200 **Data lifetime** – the lifetime of a data object on the device, which may begin at the entry to a data re-  
3201 gion, or at an **enter data** directive, or at a data API call such as **acc\_copyin** or **acc\_create**,  
3202 and which may end at the exit from a data region, or at an **exit data** directive, or at a data API  
3203 call such as **acc\_delete**, **acc\_copyout**, or **acc\_shutdown**, or at the end of the program  
3204 execution.

- 3205 **Data region** – a *region* defined by a **data** construct, or an implicit data region for a function or  
3206 subroutine containing Accelerator directives. Data constructs typically allocate device memory and  
3207 copy data from host to device memory upon entry, and copy data from device to host memory and  
3208 deallocate device memory upon exit. Data regions may contain other data regions and compute  
3209 regions.
- 3210 **Device** – a general reference to any type of accelerator.
- 3211 **Default asynchronous queue** – the asynchronous activity queue represented in the *acc-default-*  
3212 *async-var* ICV
- 3213 **Device memory** – memory attached to an accelerator, logically and physically separate from the  
3214 host memory.
- 3215 **Directive** – in C or C++, a **#pragma**, or in Fortran, a specially formatted comment statement, that  
3216 is interpreted by a compiler to augment information about or specify the behavior of the program.
- 3217 **DMA** – Direct Memory Access, a method to move data between physically separate memories;  
3218 this is typically performed by a DMA engine, separate from the host CPU, that can access the host  
3219 physical memory as well as an IO device or other physical memory.
- 3220 **GPU** – a Graphics Processing Unit; one type of accelerator device.
- 3221 **GPGPU** – General Purpose computation on Graphics Processing Units.
- 3222 **Host** – the main CPU that in this context has an attached accelerator device. The host CPU controls  
3223 the program regions and data loaded into and executed on the device.
- 3224 **Host thread** – a thread of execution that executes on the host.
- 3225 **Implicit data region** – the data region that is implicitly defined for a Fortran subprogram or C  
3226 function. A call to a subprogram or function enters the implicit data region, and a return from the  
3227 subprogram or function exits the implicit data region.
- 3228 **Kernel** – a nested loop executed in parallel by the accelerator. Typically the loops are divided into  
3229 a parallel domain, and the body of the loop becomes the body of the kernel.
- 3230 **Kernels region** – a *region* defined by a **kernels** construct. A kernels region is a structured block  
3231 which is compiled for the accelerator. The code in the kernels region will be divided by the compiler  
3232 into a sequence of kernels; typically each loop nest will become a single kernel. A kernels region  
3233 may require device memory to be allocated and data to be copied from host to device upon region  
3234 entry, and data to be copied from device to host memory and device memory deallocated upon exit.
- 3235 **Level of parallelism** – The possible levels of parallelism in OpenACC are gang, worker, vector,  
3236 and sequential. One or more of gang, worker, and vector parallelism may be specified on a loop  
3237 construct. Sequential execution corresponds to no parallelism. The **gang**, **worker**, **vector**, and  
3238 **seq** clauses specify the level of parallelism for a loop.
- 3239 **Local memory** – the memory associated with the local thread.
- 3240 **Local thread** – the host thread or the accelerator thread that executes an OpenACC directive or  
3241 construct.
- 3242 **Loop trip count** – the number of times a particular loop executes.
- 3243 **MIMD** – a method of parallel execution (Multiple Instruction, Multiple Data) where different exe-  
3244 cution units or threads execute different instruction streams asynchronously with each other.

- 3245 **OpenCL** – short for Open Compute Language, a developing, portable standard C-like programming  
3246 environment that enables low-level general-purpose programming on GPUs and other accelerators.
- 3247 **Orphaned loop construct** - a **loop** construct that is not lexically contained in any compute con-  
3248 struct, that is, that has no parent compute construct.
- 3249 **Parallel region** – a *region* defined by a **parallel** construct. A parallel region is a structured block  
3250 which is compiled for the accelerator. A parallel region typically contains one or more work-sharing  
3251 loops. A parallel region may require device memory to be allocated and data to be copied from host  
3252 to device upon region entry, and data to be copied from device to host memory and device memory  
3253 deallocated upon exit.
- 3254 **Parent compute construct** – for a **loop** construct, the **parallel**, **kernels**, or **serial** con-  
3255 struct that lexically contains the **loop** construct and is the innermost compute construct that con-  
3256 tains that **loop** construct, if any.
- 3257 **Private data** – with respect to an iterative loop, data which is used only during a particular loop  
3258 iteration. With respect to a more general region of code, data which is used within the region but is  
3259 not initialized prior to the region and is re-initialized prior to any use after the region.
- 3260 **Procedure** – in C or C++, a function in the program; in Fortran, a subroutine or function.
- 3261 **Region** – all the code encountered during an instance of execution of a construct. A region includes  
3262 any code in called routines, and may be thought of as the dynamic extent of a construct. This may  
3263 be a *parallel region*, *kernels region*, *serial region*, *data region* or *implicit data region*.
- 3264 **Scalar** – a variable of scalar datatype. In Fortran, scalars must not have allocatable or pointer  
3265 attributes.
- 3266 **Scalar datatype** – an intrinsic or built-in datatype that is not an array or aggregate datatype. In For-  
3267 tran, scalar datatypes are integer, real, double precision, complex, or logical. In C, scalar datatypes  
3268 are char (signed or unsigned), int (signed or unsigned, with optional short, long or long long at-  
3269 tribute), enum, float, double, long double, `_Complex` (with optional float or long attribute), or any  
3270 pointer datatype. In C++, scalar datatypes are char (signed or unsigned), `wchar_t`, int (signed or  
3271 unsigned, with optional short, long or long long attribute), enum, bool, float, double, long double,  
3272 or any pointer datatype. Not all implementations or targets will support all of these datatypes.
- 3273 **Serial region** – a *region* defined by a **serial** construct. A serial region is a structured block which  
3274 is compiled for the accelerator. A serial region contains code that is executed by one vector lane of  
3275 one worker in one gang. A serial region may require device memory to be allocated and data to be  
3276 copied from host to device upon region entry, and data to be copied from device to host memory  
3277 and device memory deallocated upon exit.
- 3278 **SIMD** – A method of parallel execution (single-instruction, multiple-data) where the same instruc-  
3279 tion is applied to multiple data elements simultaneously.
- 3280 **SIMD operation** – a *vector operation* implemented with SIMD instructions.
- 3281 **Structured block** – in C or C++, an executable statement, possibly compound, with a single entry  
3282 at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single  
3283 entry at the top and a single exit at the bottom.
- 3284 **Thread** – On a host processor, a thread is defined by a program counter and stack location; several  
3285 host threads may comprise a process and share host memory. On an accelerator, a thread is any one  
3286 vector lane of one worker of one gang on the device.

3287 **Vector operation** – a single operation or sequence of operations applied uniformly to each element  
3288 of an array.

3289 **Visible device copy** – a copy of a variable, array, or subarray allocated in device memory that is  
3290 visible to the program unit being compiled.



## 3291 **A. Recommendations for Implementors**

3292 This section gives recommendations for standard names and extensions to use for implementations  
3293 for specific targets and target platforms, to promote portability across such implementations, and  
3294 recommended options that programmers find useful. While this appendix is not part of the Open-  
3295 ACC specification, implementations that provide the functionality specified herein are strongly rec-  
3296 ommended to use the names in this section. The first subsection describes target devices, such as  
3297 NVIDIA GPUs. The second subsection describes additional API routines for target platforms, such  
3298 as CUDA and OpenCL. The third subsection lists several recommended options for implementa-  
3299 tions.

### 3300 **A.1. Target Devices**

#### 3301 **A.1.1. NVIDIA GPU Targets**

3302 This section gives recommendations for implementations that target NVIDIA GPU devices.

##### 3303 **Accelerator Device Type**

3304 These implementations should use the name `acc_device_nvidia` for the `acc_device_t`  
3305 type or return values from OpenACC Runtime API routines.

##### 3306 **ACC\_DEVICE\_TYPE**

3307 An implementation should use the case-insensitive name `nvidia` for the environment variable  
3308 `ACC_DEVICE_TYPE`.

##### 3309 **device.type clause argument**

3310 An implementation should use the case-insensitive name `nvidia` as the argument to the `device_type`  
3311 clause.

#### 3312 **A.1.2. AMD GPU Targets**

3313 This section gives recommendations for implementations that target AMD GPUs.

### 3314 Accelerator Device Type

3315 These implementations should use the name `acc_device_radeon` for the `acc_device_t`  
3316 type or return values from OpenACC Runtime API routines.

### 3317 ACC\_DEVICE\_TYPE

3318 These implementations should use the case-insensitive name `radeon` for the environment variable  
3319 `ACC_DEVICE_TYPE`.

### 3320 device.type clause argument

3321 An implementation should use the case-insensitive name `radeon` as the argument to the `device_type`  
3322 clause.

## 3323 A.2. API Routines for Target Platforms

3324 These runtime routines allow access to the interface between the OpenACC runtime API and the  
3325 underlying target platform. An implementation may not implement all these routines, but if it  
3326 provides this functionality, it should use these function names.

### 3327 A.2.1. NVIDIA CUDA Platform

3328 This section gives runtime API routines for implementations that target the NVIDIA CUDA Run-  
3329 time or Driver API.

#### 3330 `acc_get_current_cuda_device`

3331 **Summary** The `acc_get_current_cuda_device` routine returns the NVIDIA CUDA de-  
3332 vice handle for the current device.

#### 3333 **Format**

C or C++:

```
void* acc_get_current_cuda_device ();
```

#### 3334 `acc_get_current_cuda_context`

3335 **Summary** The `acc_get_current_cuda_context` routine returns the NVIDIA CUDA  
3336 context handle in use for the current device.

3337 **Format**

C or C++:

```
void* acc_get_current_cuda_context ();
```

3338 **acc\_get\_cuda\_stream**

3339 **Summary** The **acc\_get\_cuda\_stream** routine returns the NVIDIA CUDA stream handle  
3340 in use for the current device for the specified async value.

3341 **Format**

C or C++:

```
void* acc_get_cuda_stream ( int async );
```

3342 **acc\_set\_cuda\_stream**

3343 **Summary** The **acc\_set\_cuda\_stream** routine sets the NVIDIA CUDA stream handle the  
3344 current device for the specified async value.

3345 **Format**

C or C++:

```
int acc_set_cuda_stream ( int async, void* stream );
```

3346 **A.2.2. OpenCL Target Platform**

3347 This section gives runtime API routines for implementations that target the OpenCL API on any  
3348 device.

3349 **acc\_get\_current\_opencl\_device**

3350 **Summary** The **acc\_get\_current\_opencl\_device** routine returns the OpenCL device  
3351 handle for the current device.

3352 **Format**

C or C++:

```
void* acc_get_current_opencl_device ();
```

3353 **acc\_get\_current\_opencl\_context**

3354 **Summary** The **acc\_get\_current\_opencl\_context** routine returns the OpenCL context  
3355 handle in use for the current device.

3356 **Format**

C or C++:

```
void* acc_get_current_opengl_context ();
```

3357 **acc\_get\_opengl\_queue**

3358 **Summary** The **acc\_get\_opengl\_queue** routine returns the OpenCL command queue handle in use for the current device for the specified async value.

3360 **Format**

C or C++:

```
cl_command_queue acc_get_opengl_queue ( int async );
```

3361 **acc\_set\_opengl\_queue**

3362 **Summary** The **acc\_set\_opengl\_queue** routine returns the OpenCL command queue handle in use for the current device for the specified async value.

3364 **Format**

C or C++:

```
void acc_set_opengl_queue ( int async, cl_command_queue cmdqueue );
```

3365 **A.2.3. C Pointer in Present clause**

3366 This revision of OpenACC clarifies the construct:

```
void test(int n ){
float* p;
...
#pragma acc data present (p)
{
    // code here...
}
```

3367 This example tests whether the pointer **p** itself is present on the device. Implementations before  
3368 this revision commonly implemented this by testing whether the pointer target **p[0]** was present  
3369 on the device, and this appears in many programs assuming such. Until such programs are modified  
3370 to comply with this revision, an option to implement **present (p)** as **present (p[0])** for C  
3371 pointers may be helpful to users.

**3372 A.2.4. Autoscopying**

3373 If an implementation implements autoscopying to automatically determine variables that are private  
3374 to a compute region or to a loop, or to recognize reductions in a compute region or a loop, an option  
3375 to print a message telling what variables were affected by the analysis would be helpful to users. An  
3376 option to disable the autoscopying analysis would be helpful to promote program portability across  
3377 implementations.



# Index

- 3378 `_OPENACC`, 12–14, 18, 101
- 3379 `acc-default-async-var`, 18, 68
- 3380 `acc-device-num-var`, 18
- 3381 `acc-device-type-var`, 18
- 3382 `acc_async_noval`, 12, 67
- 3383 `acc_async_sync`, 12, 67
- 3384 `ACC_DEVICE_NUM`, 19, 93
- 3385 `acc_device_nvidia`, 121
- 3386 `acc_device_radeon`, 122
- 3387 `ACC_DEVICE_TYPE`, 19, 93, 121, 122
- 3388 `ACC_PROFLIB`, 93
- 3389 `action`
  - 3390 `attach`, 32, 36
  - 3391 `copyin`, 35
  - 3392 `copyout`, 36
  - 3393 `create`, 35
  - 3394 `delete`, 36
  - 3395 `detach`, 32, 37
  - 3396 `immediate`, 37
  - 3397 `present decrement`, 35
  - 3398 `present increment`, 35
- 3399 AMD GPU target, 121
- 3400 `async` clause, 31, 63, 67
- 3401 `async queue`, 9
- 3402 `async-argument`, 67
- 3403 asynchronous execution, 9, 67
- 3404 `atomic` construct, 12, 51
- 3405 `attach` action, 32, 36
- 3406 `attach` clause, 42
- 3407 attachment counter, 32
- 3408 `auto` clause, 12, 46
- 3409 `autoscopying`, 125
- 3410 barrier synchronization, 8, 21, 22, 24, 117
- 3411 `bind` clause, 66
- 3412 `cache` directive, 48
- 3413 `capture` clause, 55
- 3414 `collapse` clause, 45
- 3415 common block, 32, 57, 58, 67
- 3416 `compute` construct, 117
- 3417 `compute` region, 117
- 3418 `construct`, 117
  - 3419 `atomic`, 51
  - 3420 `compute`, 117
  - 3421 `data`, 28, 32
  - 3422 `host_data`, 42
  - 3423 `kernels`, 21, 22, 32
  - 3424 `kernels loop`, 49
  - 3425 `parallel`, 20, 32
  - 3426 `parallel loop`, 49
  - 3427 `serial`, 23, 32
  - 3428 `serial loop`, 49
- 3429 `copy` clause, 38
- 3430 `copyin` action, 35
- 3431 `copyin` clause, 39
- 3432 `copyout` action, 36
- 3433 `copyout` clause, 39
- 3434 `create` action, 35
- 3435 `create` clause, 40, 57
- 3436 CUDA, 9, 10, 117, 121, 122
- 3437 `data` attribute
  - 3438 `explicitly determined`, 27
  - 3439 `implicitly determined`, 27
  - 3440 `predetermined`, 27
- 3441 `data` clause, 32
- 3442 `data` construct, 28, 32
- 3443 `data` lifetime, 117
- 3444 `data` region, 28, 118
  - 3445 `implicit`, 28
- 3446 `declare` directive, 12, 55
- 3447 `default` clause, 27
- 3448 `default (none)` clause, 12, 21, 22
- 3449 `default (none)` clause, 24
- 3450 `default(present)`, 21, 22, 24
- 3451 `delete` action, 36
- 3452 `delete` clause, 41
- 3453 `detach` action, 32, 37
  - 3454 `immediate`, 37

- 3455 **detach** clause, 42
- 3456 **device** clause, 62
- 3457 **device\_resident** clause, 56
- 3458 **device\_type** clause, 12, 19, 33, 121, 122
- 3459 **deviceptr** clause, 32, 37
- 3460 direct memory access, 9, 118
- 3461 DMA, 9, 118
- 3462 **enter data** directive, 29, 32
- 3463 environment variable
- 3464     **\_OPENACC**, 18
- 3465     **ACC\_DEVICE\_NUM**, 19, 93
- 3466     **ACC\_DEVICE\_TYPE**, 19, 93, 121, 122
- 3467     **ACC\_PROFLIB**, 93
- 3468 **exit data** directive, 29, 32
- 3469 explicitly determined data attribute, 27
- 3470 **firstprivate** clause, 21, 26
- 3471 **firstprivate** clause, 24
- 3472 gang, 21, 24
- 3473 **gang** clause, 45, 65
- 3474 gang parallelism, 8
- 3475 *gang-arg*, 44
- 3476 gang-partitioned mode, 8
- 3477 gang-redundant mode, 8, 21, 24
- 3478 GP mode, 8
- 3479 GR mode, 8
- 3480 **host** clause, 12, 62
- 3481 **host\_data** construct, 42
- 3482 ICV, 18
- 3483 **if** clause, 29, 31, 62
- 3484 immediate detach action, 37
- 3485 implicit data region, 28
- 3486 implicitly determined data attribute, 27
- 3487 **independent** clause, 47
- 3488 **init** directive, 58
- 3489 internal control variable, 18
- 3490 **kernels** construct, 21, 22, 32
- 3491 **kernels loop** construct, 49
- 3492 level of parallelism, 8, 118
- 3493 **link** clause, 12, 32, 58
- 3494 local memory, 9
- 3495 local thread, 9
- 3496 **loop** construct, 43
- 3497     orphaned, 44
- 3498 **no\_create** clause, 41
- 3499 **nohost** clause, 66
- 3500 **num\_gangs** clause, 25
- 3501 **num\_workers** clause, 25
- 3502 **nvidia**, 121
- 3503 NVIDIA GPU target, 121
- 3504 OpenCL, 9, 10, 119, 121, 123
- 3505 orphaned **loop** construct, 44
- 3506 **parallel** construct, 20, 32
- 3507 **parallel loop** construct, 49
- 3508 parallelism
- 3509     level, 8, 118
- 3510 parent compute construct, 44
- 3511 predetermined data attribute, 27
- 3512 **present** clause, 32, 38
- 3513 present decrement action, 35
- 3514 present increment action, 35
- 3515 **private** clause, 25, 47
- 3516 **radeon**, 122
- 3517 **read** clause, 54
- 3518 **reduction** clause, 26, 48
- 3519 reference count, 31
- 3520 region
- 3521     compute, 117
- 3522     data, 28, 118
- 3523     implicit data, 28
- 3524 **routine** directive, 12, 64
- 3525 **self** clause, 12, 62
- 3526 sentinel, 17
- 3527 **seq** clause, 46, 66
- 3528 **serial** construct, 23, 32
- 3529 **serial loop** construct, 49
- 3530 **shutdown** directive, 59
- 3531 *size-expr*, 44
- 3532 thread, 119
- 3533 **tile** clause, 12, 47
- 3534 **update** clause, 54
- 3535 **update** directive, 61
- 3536 **use\_device** clause, 43
- 3537 **vector** clause, 46, 65
- 3538 vector lane, 21
- 3539 vector parallelism, 8
- 3540 vector-partitioned mode, 8



- 3541 vector-single mode, 8
- 3542 **vector\_length** clause, 25
- 3543 visible device copy, 120
- 3544 VP mode, 8
- 3545 VS mode, 8
  
- 3546 **wait** clause, 31, 63, 68
- 3547 **wait** directive, 68
- 3548 worker, 21, 24
- 3549 **worker** clause, 45, 65
- 3550 worker parallelism, 8
- 3551 worker-partitioned mode, 8
- 3552 worker-single mode, 8
- 3553 WP mode, 8
- 3554 WS mode, 8