

1
2
3
4
5

The OpenACC[®]
Application Programming Interface
Version 2.6

OpenACC-Standard.org

November, 2017

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

10 © 2011-2017 OpenACC-Standard.org. All rights reserved.

11 Contents

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

51	2.6.7. Attachment Counter	32
52	2.7. Data Clauses	33
53	2.7.1. Data Specification in Data Clauses	33
54	2.7.2. Data Clause Actions	35
55	2.7.3. deviceptr clause	38
56	2.7.4. present clause	38
57	2.7.5. copy clause	39
58	2.7.6. copyin clause	39
59	2.7.7. copyout clause	40
60	2.7.8. create clause	41
61	2.7.9. no_create clause	41
62	2.7.10. delete clause	42
63	2.7.11. attach clause	42
64	2.7.12. detach clause	42
65	2.8. Host Data Construct	43
66	2.8.1. use_device clause	43
67	2.8.2. if clause	43
68	2.8.3. if_present clause	44
69	2.9. Loop Construct	44
70	2.9.1. collapse clause	45
71	2.9.2. gang clause	45
72	2.9.3. worker clause	46
73	2.9.4. vector clause	46
74	2.9.5. seq clause	47
75	2.9.6. auto clause	47
76	2.9.7. tile clause	47
77	2.9.8. device_type clause	48
78	2.9.9. independent clause	48
79	2.9.10. private clause	48
80	2.9.11. reduction clause	48
81	2.10. Cache Directive	49
82	2.11. Combined Constructs	50
83	2.12. Atomic Construct	51
84	2.13. Declare Directive	56
85	2.13.1. device_resident clause	57
86	2.13.2. create clause	58
87	2.13.3. link clause	58
88	2.14. Executable Directives	59
89	2.14.1. Init Directive	59
90	2.14.2. Shutdown Directive	60
91	2.14.3. Set Directive	60
92	2.14.4. Update Directive	62
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	65
98	2.15.2. Global Data Access	67

99	2.16. Asynchronous Behavior	68
100	2.16.1. async clause	68
101	2.16.2. wait clause	69
102	2.16.3. Wait Directive	69
103	2.17. Fortran Optional Arguments	70
104	3. Runtime Library	71
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	4. Environment Variables	93
143	4.1. ACC_DEVICE_TYPE	93
144	4.2. ACC_DEVICE_NUM	93

145	4.3. ACC_PROFLIB	93
146	5. Profiling Interface	95
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	6. Glossary	117
176	A. Recommendations for Implementors	121
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.3. Recommended Options	124
184	A.3.1. C Pointer in Present clause	124
185	A.3.2. Autoscopying	125

1. Introduction

This document describes the compiler directives, library routines, and environment variables that collectively define the OpenACC[™] 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

220 device, initiating data transfer, sending the code to the accelerator, passing arguments to the compute 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

224 Most current accelerators support two or three levels of parallelism. Most accelerators support 225 coarse-grain parallelism, which is fully parallel execution across execution units. There may be 226 limited support for synchronization across coarse-grain parallel operations. Many accelerators also 227 support fine-grain parallelism, often implemented as multiple threads of execution within a single 228 execution unit, which are typically rapidly switched on the execution unit to tolerate long latency 229 memory operations. Finally, most accelerators also support SIMD or vector operations within each 230 execution unit. The execution model exposes these multiple levels of parallelism on the device and 231 the programmer is required to understand the difference between, for example, a fully parallel loop 232 and a loop that is vectorizable but requires synchronization between statements. A fully parallel 233 loop can be programmed for coarse-grain parallel execution. Loops with dependences must either 234 be split to allow coarse-grain parallel execution, or be programmed to execute on a single execution 235 unit using fine-grain parallelism, vector parallelism, or sequentially.

236 OpenACC exposes these three *levels of parallelism* via *gang*, *worker*, and *vector* parallelism. Gang 237 parallelism is coarse-grain. A number of gangs will be launched on the accelerator. Worker paral- 238 lelism is fine-grain. Each gang will have one or more workers. Vector parallelism is for SIMD or 239 vector operations within a worker.

240 When executing a compute region on the device, one or more gangs are launched, each with one 241 or more workers, where each worker may have vector execution capability with one or more vector 242 lanes. The gangs start executing in *gang-redundant* mode (GR mode), meaning one vector lane of 243 one worker in each gang executes the same code, redundantly. When the program reaches a loop 244 or loop nest marked for gang-level work-sharing, the program starts to execute in *gang-partitioned* 245 mode (GP mode), where the iterations of the loop or loops are partitioned across gangs for truly 246 parallel execution, but still with only one worker per gang and one vector lane per worker active.

247 When only one worker is active, in either GR or GP mode, the program is in *worker-single* mode 248 (WS mode). When only one vector lane is active, the program is in *vector-single* mode (VS mode). 249 If a gang reaches a loop or loop nest marked for worker-level work-sharing, the gang transitions to 250 *worker-partitioned* mode (WP mode), which activates all the workers of the gang. The iterations 251 of the loop or loops are partitioned across the workers of this gang. If the same loop is marked for 252 both gang-partitioning and worker-partitioning, then the iterations of the loop are spread across all 253 the workers of all the gangs. If a worker reaches a loop or loop nest marked for vector-level work- 254 sharing, the worker will transition to *vector-partitioned* mode (VP mode). Similar to WP mode, the 255 transition to VP mode activates all the vector lanes of the worker. The iterations of the loop or loops 256 will be partitioned across the vector lanes using vector or SIMD operations. Again, a single loop 257 may be marked for one, two, or all three of gang, worker, and vector parallelism, and the iterations 258 of that loop will be spread across the gangs, workers, and vector lanes as appropriate.

259 The program starts executing with a single host thread, identified by a program counter and its 260 stack. The thread may spawn additional host threads, for instance using the OpenMP API. On the 261 accelerator device, a single vector lane of a single worker of a single gang is called a thread. When 262 executing on the device, a parallel execution context is created on the accelerator and may contain 263 many such threads.

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

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

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

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

289 1.3. Memory Model

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

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

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

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

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

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

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

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

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

340 1.4. Conventions used in this document

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

```
#pragma acc
```

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

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

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

#pragma acc *directive-name new-line*

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

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

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

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

352 1.5. Organization of this document

353 The rest of this document is organized as follows:

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

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

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

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

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

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

366 1.6. References

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

- 374 • *NVIDIA CUDA[™] C Programming Guide*, version 7.0, March 2015.
- 375 • *The OpenCL Specification*, version 202, Khronos OpenCL Working Group, October 2014.

376 1.7. Changes from Version 1.0 to 2.0

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

- 407 - **acc_copyin**, **acc_present_or_copyin**
- 408 - **acc_create**, **acc_present_or_create**
- 409 - **acc_copyout**, **acc_delete**
- 410 - **acc_map_data**, **acc_unmap_data**
- 411 - **acc_deviceptr**, **acc_hostptr**
- 412 - **acc_is_present**
- 413 - **acc_memcpy_to_device**, **acc_memcpy_from_device**
- 414 - **acc_update_device**, **acc_update_self**
- 415 • defined behavior with multiple host threads, such as with OpenMP
- 416 • recommendations for specific implementations
- 417 • clarified that no arguments are allowed on the **vector** clause in a parallel region

418 1.8. Corrections in the August 2013 document

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

422 1.9. Changes from Version 2.0 to 2.5

- 423 • The **_OPENACC** value was updated to **201510**; see Section 2.2 Conditional Compilation.
- 424 • The **num_gangs**, **num_workers**, and **vector_length** clauses are now allowed on the
- 425 **kernels** construct; see Section 2.5.2 Kernels Construct.
- 426 • Reduction on C++ class members, array elements, and struct elements are explicitly disal-
- 427 lowed; see Section 2.5.12 reduction clause.
- 428 • Reference counting is now used to manage the correspondence and lifetime of device data;
- 429 see Section 2.6.6 Reference Counters.
- 430 • The behavior of the **exit data** directive has changed to decrement the dynamic reference
- 431 counter. A new optional **finalize** clause was added to set the dynamic reference counter
- 432 to zero. See Section 2.6.5 Enter Data and Exit Data Directives.
- 433 • The **copy**, **copyin**, **copyout**, and **create** data clauses were changed to behave like
- 434 **present_or_copy**, etc. The **present_or_copy**, **pcopy**, **present_or_copyin**,
- 435 **pcopyin**, **present_or_copyout**, **pcopyout**, **present_or_create**, and **pcreate**
- 436 data clauses are no longer needed, though will be accepted for compatibility; see Section 2.7
- 437 Data Clauses.
- 438 • Reductions on orphaned gang loops are explicitly disallowed; see Section 2.9 Loop Construct.
- 439 • The description of the **loop auto** clause has changed; see Section 2.9.6 auto clause.

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

473 1.10. Changes from Version 2.5 to 2.6

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

- 477 the host and returns properties about any device. See Section 3.2.6.
- 478 • The text has clarified that if a variable is in a reduction which spans two or more nested loops,
479 each **loop** directive on any of those loops must have a **reduction** clause that contains the
480 variable; see Section 2.9.11 reduction clause.
 - 481 • An optional **if** or **if_present** clause is now allowed on the **host_data** construct. See
482 Section 2.8 Host Data Construct.
 - 483 • A new **no_create** data clause is now allowed on compute and **data** constructs. See Sec-
484 tion 2.7.9 no_create clause.
 - 485 • The behavior of Fortran optional arguments in data clauses and in routine calls has been
486 specified; see Section 2.17 Fortran Optional Arguments.
 - 487 • The descriptions of some of the Fortran versions of the runtime library routines were simpli-
488 fied; see Section 3.2 Runtime Library Routines.
 - 489 • To allow for manual deep copy of data structures with pointers, new *attach* and *detach* be-
490 havior was added to the data clauses, new **attach** and **detach** clauses were added, and
491 matching **acc_attach** and **acc_detach** runtime API routines were added; see Sections
492 2.6.3, 2.7.11-2.7.12 and 3.2.34-3.2.35.
 - 493 • The Intel Coprocessor Offload Interface target and API routine sections were removed from
494 the Section A Recommendations for Implementors, since Intel no longer produces this prod-
495 uct.

496 1.11. Topics Deferred For a Future Revision

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

- 502 • **Support for attaching C/C++ pointers that point to an address past the end of a memory region.**
- 503 • Full support for C and C++ structs and struct members, including pointer members.
- 504 • Full support for Fortran derived types and derived type members, including allocatable and
505 pointer members.
- 506 • Fully defined interaction with multiple host threads.
- 507 • Optionally removing the synchronization or barrier at the end of vector and worker loops.
- 508 • Allowing an **if** clause after a **device_type** clause.
- 509 • A **shared** clause (or something similar) for the loop directive.
- 510 • Better support for multiple devices from a single thread, whether of the same type or of
511 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

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

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

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

547 2.2. Conditional Compilation

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

552 2.3. Internal Control Variables

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

557 The ICVs are:

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

562 2.3.1. Modifying and Retrieving ICV Values

563 The following table shows environment variables or procedures to modify the values of the internal
564 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>
565 <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>

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

571 2.4. Device-Specific Clauses

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

582 Clauses that precede any **device_type** are *default clauses*. Clauses that follow a **device_type**
583 are *device-specific clauses*. A clause may appear both as a default clause and as a device-specific
584 clause. In that case, the value in the device-specific clause is used when compiling for that device
585 type.

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

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

596 **Syntax** The syntax of the **device_type** clause is

```

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

```

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

598 2.5. Compute Constructs

599 2.5.1. Parallel Construct

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

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

```

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

```

602 and in Fortran, the syntax is

```

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

```

603 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 )

```

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

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

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

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

627 Restrictions

- 628 • A program may not branch into or out of an OpenACC **parallel** construct.
- 629 • A program must not depend on the order of evaluation of the clauses, or on any side effects
630 of the evaluations.
- 631 • Only the **async**, **wait**, **num_gangs**, **num_workers**, and **vector_length** clauses
632 may follow a **device_type** clause.
- 633 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
634 value; in C or C++, the condition must evaluate to a scalar integer value.
- 635 • At most one **default** clause may appear, and it must have a value of either **none** or
636 **present**.

637 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
638 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**
639 clauses are described in Sections 2.5.10 and Sections 2.5.11. The **device_type** clause is de-
640 scribed in Section 2.4 Device-Specific Clauses.

641 2.5.2. Kernels Construct

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

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

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

645 and in Fortran, the syntax is

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

646 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 )
```

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

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

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

664 **Restrictions**

- 665 • A program may not branch into or out of an OpenACC **kernels** construct.
- 666 • A program must not depend on the order of evaluation of the clauses, or on any side effects
667 of the evaluations.
- 668 • Only the **async**, **wait**, **num_gangs**, **num_workers**, and **vector_length** clauses
669 may follow a **device_type** clause.
- 670 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
671 value; in C or C++, the condition must evaluate to a scalar integer value.
- 672 • At most one **default** clause may appear, and it must have a value of either **none** or
673 **present**.

674 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
675 data clauses are described in Section 2.7 Data Clauses. The **device_type** clause is described in
676 Section 2.4 Device-Specific Clauses.

677 **2.5.3. Serial Construct**

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

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

679 and in Fortran, the syntax is

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

680 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 )
```

681 **Description** When the program encounters an accelerator **serial** construct, one gang of one
682 worker with a vector length of one is created to execute the accelerator serial region sequentially.
683 The single gang begins executing the code in the structured block in gang-redundant mode, even
684 though there is a single gang. The **serial** construct executes as if it were a **parallel** construct
685 with clauses **num_gangs(1) num_workers(1) vector_length(1)**.

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

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

700 Restrictions

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

709 The **copy**, **copyin**, **copyout**, **create**, **no_create**, **present**, **deviceptr**, and **attach**
710 data clauses are described in Section 2.7 Data Clauses. The **private** and **firstprivate**
711 clauses are described in Sections 2.5.10 and Sections 2.5.11. The **device_type** clause is de-
712 scribed in Section 2.4 Device-Specific Clauses.

713 2.5.4. if clause

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

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

721 **2.5.5. async clause**

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

723 **2.5.6. wait clause**

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

725 **2.5.7. num_gangs clause**

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

732 **2.5.8. num_workers clause**

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

740 **2.5.9. vector_length clause**

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

748 **2.5.10. private clause**

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

751 **Restrictions**

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

754 2.5.11. firstprivate clause

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

759 Restrictions

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

762 2.5.12. reduction clause

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

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

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

775 Restrictions

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

781 2.5.13. default clause

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

788 2.6. Data Environment

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

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

802 2.6.1. Variables with Predetermined Data Attributes

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

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

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

819 2.6.2. Data Regions and Data Lifetimes

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

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

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

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

835 In addition to data regions, a program may create and delete data on the accelerator using **enter**
836 **data** and **exit data** directives or using runtime API routines. When the program executes
837 an **enter data** directive, or executes a call to a runtime API **acc_copyin** or **acc_create**
838 routine, each variable, array, or subarray on the directive or the variable on the runtime API argument
839 list will be made live on accelerator.

840 2.6.3. Data Structures with Pointers

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

844 When a data object is copied from host memory to device memory, the values are copied exactly.
845 If the data is a data structure that includes a pointer, or is just a pointer, the pointer value copied
846 to device memory will be the host pointer value. If the pointer target object is also allocated on
847 or copied to the device, the pointer itself needs to be updated with the device address of the target
848 object before dereferencing the pointer on the device.

849 An *attach* action updates the pointer in device memory to point to the device copy of the data that the
850 host pointer targets; see Section 2.7.1. For Fortran array pointers and allocatable arrays, this includes
851 copying any associated descriptor (dope vector) to the device copy of the pointer. When the device
852 pointer target is deallocated, the pointer in device memory should be restored to the host value, so

853 it can be safely copied back to host memory. A *detach* action updates the pointer in device memory
 854 to have the same value as the corresponding pointer in local memory; see Section 2.7.1. The *attach*
 855 and *detach* actions are performed by the **copy**, **copyin**, **copyout**, **create**, **attach**, and
 856 **detach** data clauses (Sections 2.7.3-2.7.12), and the **acc_attach** and **acc_detach** runtime
 857 API routines (Sections 3.2.34 and 3.2.35). The *attach* and *detach* actions use attachment counters
 858 to determine when the pointer on the device needs to be updated; see Section 2.6.7.

859 2.6.4. Data Construct

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

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

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

864 and in Fortran, the syntax is

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

865 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 )
```

866 **Description** Data will be allocated in the memory of the current device and copied from the host
 867 or local memory to the device, or copied back, as required. The data clauses are described in Sec-
 868 tion 2.7 Data Clauses. Structured reference counters are incremented for data when entering a data
 869 region, and decremented when leaving the region, as described in Section 2.6.6 Reference Counters.

870 **if clause**

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

878 **2.6.5. Enter Data and Exit Data Directives**

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

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

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

886 and in Fortran, the syntax is

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

887 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 )
```

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

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

889 and in Fortran, the syntax is

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

890 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
```

891 **Description** At an **enter data** directive, data may be allocated in the current device memory
892 and copied from the host or local memory to the device. This action enters a data lifetime for those
893 variables, arrays, or subarrays, and will make the data available for **present** clauses on constructs
894 within the data lifetime. Dynamic reference counters are incremented for this data, as described
895 in Section 2.6.6 Reference Counters. Pointers in device memory may be *attached* to point to the
896 corresponding device copy of the host pointer target.

897 At an **exit data** directive, data may be copied from the device memory to the host or local
898 memory and deallocated from device memory. If no **finalize** clause appears, dynamic reference
899 counters are decremented for this data. If a **finalize** clause appears, the dynamic reference
900 counters are set to zero for this data. Pointers in device memory may be *detached* so as to have the
901 same value as the corresponding host pointer.

902 The data clauses are described in Section 2.7 Data Clauses. Reference counting behavior is de-
903 scribed in Section 2.6.6 Reference Counters.

904 **if clause**

905 The **if** clause is optional; when there is no **if** clause, the compiler will generate code to allocate or
906 deallocate memory on the current accelerator device and move data from and to the local memory.
907 When an **if** clause appears, the program will conditionally allocate or deallocate device memory
908 and move data to and/or from the device. When the *condition* in the **if** clause evaluates to zero in
909 C or C++, or **.false.** in Fortran, no device memory will be allocated or deallocated, and no data
910 will be moved. When the *condition* evaluates to nonzero in C or C++, or **.true.** in Fortran, the
911 data will be allocated or deallocated and moved as specified.

912 **async clause**

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

914 **wait clause**

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

916 **finalize clause**

917 The **finalize** clause is allowed on the **exit data** directive and is optional. When no **finalize**
918 clause appears, the **exit data** directive will decrement the dynamic reference counters for vari-
919 ables and arrays appearing in **copyout** and **delete** clauses, and will decrement the attachment
920 counters for pointers appearing in **detach** clauses. If a **finalize** clause appears, the **exit**
921 **data** directive will set the dynamic reference counters to zero for variables and arrays appearing in
922 **copyout** and **delete** clauses, and will set the attachment counters to zero for pointers appearing
923 in **detach** clauses.

924 **2.6.6. Reference Counters**

925 When data is allocated on a non-shared memory device due to data clauses or OpenACC API routine
926 calls, the OpenACC implementation keeps track of that device memory and its relationship to the
927 corresponding data in host memory. Each section of device memory will be associated with two
928 *reference counters per device*, a structured reference counter and a dynamic reference counter. The
929 structured and dynamic reference counters are used to determine when to allocate or deallocate data
930 in device memory. The structured reference counter for a block of data keeps track of how many
931 nested data regions have been entered for that data. The initial value of the structured reference
932 counter for static data on the device (in a global **declare** directive) is one; for all other data, the
933 initial value is zero. The dynamic reference counter for a block of data keeps track of how many
934 dynamic data lifetimes are currently active on the device for that block. The initial value of the
935 dynamic reference counter is zero.

936 A structured reference counter is incremented when entering each data or compute region that con-
937 tain an explicit data clause or implicitly-determined data attributes for that block of memory, and
938 is decremented when exiting that region. A dynamic reference counter is incremented for each
939 **enter data copyin** or **create** clause, or each **acc_copyin** or **acc_create** API routine
940 call for that block of memory. The dynamic reference counter is decremented for each **exit data**
941 **copyout** or **delete** clause when no **finalize** clause appears, or each **acc_copyout** or
942 **acc_delete** API routine call for that block of memory. The dynamic reference counter will be
943 set to zero with an **exit data copyout** or **delete** clause when a **finalize** clause appears, or
944 each **acc_copyout_finalize** or **acc_delete_finalize** API routine call for the block of
945 memory. The reference counters are modified synchronously with the encountering thread, even if
946 the data directives include an **async** clause. When both structured and dynamic reference counters
947 reach zero, the data lifetime on the device for that data ends.

948 **2.6.7. Attachment Counter**

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

956 A pointer in device memory can be assigned a device address in two ways. The pointer can be

957 attached to a device address due to data clauses or API routines, as described in Section 2.7.2
958 Data Clause Actions, or the pointer can be assigned in a compute region on the device. Unspecified
959 behavior may result if both ways are used for the same pointer.

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

965 2.7. Data Clauses

966 These data clauses may appear on the **parallel** construct, **kernels** construct, **serial con-**
967 **struct**, **data** construct, the **enter data** and **exit data** directives, and **declare** directives.
968 In the descriptions, the *region* is a compute region with a clause appearing on a **parallel**,
969 **kernels**, or **serial** construct, a data region with a clause on a **data** construct, or an implicit
970 data region with a clause on a **declare** directive. If the **declare** directive appears in a global
971 context, the corresponding implicit data region has a duration of the program. The list argument
972 to each data clause is a comma-separated collection of variable names, array names, or subarray
973 specifications. For all clauses except **deviceptr** and **present**, the list argument may include a
974 Fortran *common block* name enclosed within slashes, if that *common block* name also appears in a
975 **declare** directive **link** clause. In all cases, the compiler will allocate and manage a copy of the
976 variable or array in the memory of the current device, creating a visible device copy of that variable
977 or array, for non-shared memory devices.

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

984 Restrictions

- 985 • Data clauses may not follow a **device_type** clause.
- 986 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
987 data clauses.

988 2.7.1. Data Specification in Data Clauses

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

AA[2 : n]

991 If the lower bound is missing, zero is used. If the length is missing and the array has known size, the
992 size of the array is used; otherwise the length is required. The subarray **AA[2 : n]** means element

993 **AA[2], AA[3], ..., AA[2+n-1].**

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

- 995 • Statically-sized array: **float AA[100][200];**
- 996 • Pointer to statically sized rows: **typedef float row[200]; row* BB;**
- 997 • Statically-sized array of pointers: **float* CC[200];**
- 998 • Pointer to pointers: **float** DD;**

999 Each dimension may be statically sized, or a pointer to dynamically allocated memory. Each of
1000 these may be included in a data clause using subarray notation to specify a rectangular array:

- 1001 • **AA[2:n][0:200]**
- 1002 • **BB[2:n][0:m]**
- 1003 • **CC[2:n][0:m]**
- 1004 • **DD[2:n][0:m]**

1005 Multidimensional rectangular subarrays in C and C++ may be specified for any array with any com-
1006 bination of statically-sized or dynamically-allocated dimensions. For statically sized dimensions,
1007 all dimensions except the first must specify the whole **extent**, to preserve the contiguous data re-
1008 striction, discussed below. For dynamically allocated dimensions, the implementation will allocate
1009 pointers on the device corresponding to the pointers on the host, and will fill in those pointers as
1010 appropriate.

1011 In Fortran, a subarray is an array name followed by a comma-separated list of range specifications
1012 in parentheses, with lower and upper bound subscripts, such as

arr(1:high, low:100)

1013 If either the lower or upper bounds are missing, the declared or allocated bounds of the array, if
1014 known, are used. All dimensions except the last must specify the whole **extent**, to preserve the
1015 contiguous data restriction, discussed below.

1016 Restrictions

- 1017 • In Fortran, the upper bound for the last dimension of an assumed-size dummy array must be
1018 specified.
- 1019 • In C and C++, the length for dynamically allocated dimensions of an array must be explicitly
1020 specified.
- 1021 • In C and C++, modifying pointers in pointer arrays during the data lifetime, either on the host
1022 or on the device, may result in undefined behavior.
- 1023 • If a subarray is specified in a data clause, the implementation may choose to allocate memory
1024 for only that subarray on the accelerator.
- 1025 • In Fortran, array pointers may be specified, but pointer association is not preserved in the
1026 device memory.

- 1027 • Any array or subarray in a data clause, including Fortran array pointers, must be a contiguous
1028 block of memory, except for dynamic multidimensional C arrays.
- 1029 • In C and C++, if a variable or array of struct or class type is specified, all the data members
1030 of the struct or class are allocated and copied, as appropriate. If a struct or class member is a
1031 pointer type, the data addressed by that pointer are not implicitly copied.
- 1032 • In Fortran, if a variable or array with derived type is specified, all the members of that derived
1033 type are allocated and copied, as appropriate. If any member has the **allocatable** or
1034 **pointer** attribute, the data accessed through that member are not copied.
- 1035 • If an expression is used in a subscript or subarray expression in a clause on a **data** construct,
1036 the same value is used when copying data at the end of the data region, even if the values of
1037 variables in the expression change during the data region.

1038 2.7.2. Data Clause Actions

1039 Most of the data clauses perform one or more the following actions. The actions test or modify one
1040 or both of the structured and dynamic reference counters, depending on the directive on which the
1041 data clause appears.

1042 Present Increment Action

1043 A *present increment* action is one of the actions that may be performed for a **present** (Section
1044 2.7.4), **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Sec-
1045 tion 2.7.8), or **no_create** (Section 2.7.9) clause, or for a call to an **acc_copyin** (Section 3.2.20)
1046 or **acc_create** (Section 3.2.21) API routine. See those sections for details.

1047 A *present increment* action for a *var* occurs only when *var* is already present on the device.

1048 A *present increment* action for a *var* increments the structured or dynamic reference counter for *var*.

1049

1050 Present Decrement Action

1051 A *present decrement* action is one of the actions that may be performed for a **present** (Section
1052 2.7.4), **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Sec-
1053 tion 2.7.8), **no_create** (Section 2.7.9), or **delete** (Section 2.7.10) clause, or for a call to an
1054 **acc_copyout** (Section 3.2.22) or **acc_delete** (Section 3.2.23) API routine. See those sec-
1055 tions for details.

1056 A *present decrement* action for a *var* occurs only when *var* is already present on the device.

1057 A *present decrement* action for a *var* decrements the structured or dynamic reference counter for *var*,
1058 if its value is greater than zero. If the reference counter is already zero, its value is left unchanged.

1059 Create Action

1060 A *create* action is one of the actions that may be performed for a **copyout** (Section 2.7.7) or
1061 **create** (Section 2.7.8) clause, or for a call to an **acc_create** API routine (Section 3.2.21). See

1062 those sections for details.

1063 A *create* action for a *var* occurs only when *var* is not already present on the device.

1064 A *create* action for a *var*:

- 1065 • allocates device memory for *var*; and
- 1066 • sets the structured or dynamic reference counter to one.

1067 **Copyin Action**

1068 A *copyin* action is one of the actions that may be performed for a **copy** (Section 2.7.5) or **copyin**
1069 (Section 2.7.6) clause, or for a call to an **acc_copyin** API routine (Section 3.2.20). See those
1070 sections for details.

1071 A *copyin* action for a *var* occurs only when *var* is not already present on the device.

1072 A *copyin* action for a *var*:

- 1073 • allocates device memory for *var*;
- 1074 • initiates a copy of the data for *var* from the local thread memory to the corresponding device
1075 memory; and
- 1076 • sets the structured or dynamic reference counter to one.

1077 The data copy may complete asynchronously, depending on other clauses on the directive.

1078 **Copyout Action**

1079 A *copyout* action is one of the actions that may be performed for a **copy** (Section 2.7.5) or
1080 **copyout** (Section 2.7.7) clause, or for a call to an **acc_copyout** API routine (Section 3.2.22).
1081 See those sections for details.

1082 A *copyout* action for a *var* occurs only when *var* is present on the device.

1083 A *copyout* action for a *var*:

- 1084 • performs an *immediate detach* action for any pointer in *var*;
- 1085 • initiates a copy of the data for *var* from the device memory to the corresponding local thread
1086 memory; and
- 1087 • deallocates the device memory for *var*.

1088 The data copy may complete asynchronously, depending on other clauses on the directive, in which
1089 case the memory is deallocated when the data copy is complete.

1090 **Delete Action**

1091 A *present decrement* action is one of the actions that may be performed for a **present** (Section
1092 2.7.4), **copyin** (Section 2.7.6), **create** (Section 2.7.8), **no_create** (Section 2.7.9), or **delete**
1093 (Section 2.7.10) clause, or for a call to an **acc_delete** API routine (Section 3.2.23). See those
1094 sections for details.

1095 A *delete* action for a *var* occurs only when *var* is present on the device.

1096 A *delete* action for *var*:

- 1097 • performs an *immediate detach* action for any pointer in *var*; and
- 1098 • deallocates device memory for *var*.

1099 **Attach Action**

1100 An *attach* action is one of the actions that may be performed for a **present** (Section 2.7.4),
1101 **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Section 2.7.8),
1102 **no_create** (Section 2.7.9), or **attach** (Section 2.7.10) clause, or for a call to an **acc_attach**
1103 API routine (Section 3.2.34). See those sections for details.

1104 An *attach* action for a *var* occurs only when *var* is a pointer reference.

1105 If the current device is a shared memory device or if the pointer *var* is not present on the device, or
1106 if the address to which *var* points is not present on the device, no action is taken. If the *attachment*
1107 *counter* for *var* is nonzero and the pointer in device memory already points to the device copy of
1108 the data in *var*, the *attachment counter* for the pointer *var* is incremented. Otherwise, the pointer
1109 in device memory is *attached* to the device copy of the data by initiating an update for the pointer
1110 in device memory to point to the device copy of the data and setting the *attachment counter* for the
1111 pointer *var* to one. The update may complete asynchronously, depending on other clauses on the
1112 directive. The pointer update must follow any data copies due to *copyin* actions that are performed
1113 for the same directive.

1114 **Detach Action**

1115 A *detach* action is one of the actions that may be performed for a **present** (Section 2.7.4),
1116 **copy** (Section 2.7.5), **copyin** (Section 2.7.6), **copyout** (Section 2.7.7), **create** (Section 2.7.8),
1117 **no_create** (Section 2.7.9), **delete** (Section 2.7.10), or **detach** (Section 2.7.10) clause, or for
1118 a call to an **acc_detach** API routine (Section 3.2.35). See those sections for details.

1119 A *detach* action for a *var* occurs only when *var* is a pointer reference.

1120 If the current device is a shared memory device, or if *var* is not present on the device, or if the
1121 *attachment counter* for *var* for the pointer is zero, no action is taken. Otherwise, the *attachment*
1122 *counter* for the pointer *var* is decremented. If the *attachment counter* is decreased to zero, the
1123 pointer is *detached* by initiating an update for the pointer *var* in device memory to have the same
1124 value as the corresponding pointer in local memory. The update may complete asynchronously,
1125 depending on other clauses on the directive. The pointer update must precede any data copies due
1126 to *copyout* actions that are performed for the same directive.

1127 **Immediate Detach Action**

1128 An *immediate detach* action is one of the actions that may be performed for a **detach** (Section
1129 2.7.10) clause, or for a call to an **acc_detach_finalize** API routine (Section 3.2.35). See
1130 those sections for details.

1131 An *immediate detach* action for a *var* occurs only when *var* is a pointer reference and is present on
1132 the device.

1133 If the *attachment counter* for the pointer is zero, the *immediate detach* action has no effect. Other-
1134 wise, the *attachment counter* for the pointer set to zero and the pointer is *detached* by initiating an
1135 update for the pointer in device memory to have the same value as the corresponding pointer in local
1136 memory. The update may complete asynchronously, depending on other clauses on the directive.
1137 The pointer update must precede any data copies due to *copyout* actions that are performed for the
1138 same directive.

1139 2.7.3. deviceptr clause

1140 The **deviceptr** clause may appear on structured **data** and compute constructs and **declare**
1141 directives.

1142 The **deviceptr** clause is used to declare that the pointers in *var-list* are device pointers, so the
1143 data need not be allocated or moved between the host and device for this pointer.

1144 In C and C++, the variables in *var-list* must be pointer variables.

1145 In Fortran, the variables in *var-list* must be dummy arguments (arrays or scalars), and may not have
1146 the Fortran **pointer**, **allocatable**, or **value** attributes.

1147 For a shared-memory device, host pointers are the same as device pointers, so this clause has no
1148 effect.

1149 2.7.4. present clause

1150 The **present** clause may appear on structured **data** and compute constructs and **declare** di-
1151 rectives. If the current device is a shared memory device, no action is taken.

1152 For a non-shared memory **device**, the **present** clause specifies that variables or arrays in *var-list*
1153 are already present in device memory on the current device due to data regions or data lifetimes that
1154 contain the construct on which the **present** clause appears.

1155 If the current device is a non-shared memory device, the **present** clause behaves as follows, for
1156 each *var* in *var-list*.

- 1157 • At entry to the region:
 - 1158 – If *var* is not present on the current device, a runtime error is issued.
 - 1159 – Otherwise, a *present increment* action with the structured reference counter is performed.
 - 1160 – If *var* is a pointer reference, an *attach* action is performed.
- 1161 • At exit from the region:
 - 1162 – If *var* is not present on the current device, a runtime error is issued.
 - 1163 – Otherwise, a *present decrement* action with the structured reference counter is per-
1164 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
1165 and dynamic reference counters are zero, a *delete* action is performed.

1166 Restrictions

- 1167 • If only a subarray of an array is present on the current device, the **present** clause must
- 1168 specify the same subarray, or a subarray that is a proper subset of the subarray in the data
- 1169 lifetime.
- 1170 • It is a runtime error if the subarray in *var-list* clause includes array elements that are not part
- 1171 of the subarray specified in the data lifetime.

1172 2.7.5. copy clause

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

1175 If the current device is a non-shared memory device, the **copy** clause behaves as follows, for each
 1176 *var* in *var-list*.

- 1177 • At entry to the region:
 - 1178 – If *var* is present, a *present increment* action with the structured reference counter is
 - 1179 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 1180 – Otherwise, a *copyin* action with the structured reference counter is performed. If *var* is
 - 1181 a pointer reference, an *attach* action is performed.
- 1182 • At exit from the region:
 - 1183 – If *var* is not present on the current device, a runtime error is issued.
 - 1184 – Otherwise, a *present decrement* action with the structured reference counter is per-
 1185 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
 1186 and dynamic reference counters are zero, a *copyout* action is performed.

1187 The restrictions regarding subarrays in the **present** clause apply to this clause.

1188 For compatibility with OpenACC 2.0, **present_or_copy** and **pcopy** are alternate names for
 1189 **copy**.

1190 2.7.6. copyin clause

1191 The **copyin** clause may appear on structured **data** and compute constructs, on **declare** direc-
 1192 tives, and on **enter data** directives. If the current device is a shared memory device, no action is
 1193 taken.

1194 If the current device is a non-shared memory device, the **copyin** clause behaves as follows, for
 1195 each *var* in *var-list*:

- 1196 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
 1197 the dynamic reference counter is used.
 - 1198 – If *var* is present, a *present increment* action with the appropriate reference counter is
 1199 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 1200 – Otherwise, a *copyin* action with the appropriate reference counter is performed. If *var*
 1201 is a pointer reference, an *attach* action is performed.
- 1202 • At exit from the region:

- 1203 – If *var* is not present on the current device, a runtime error is issued.
- 1204 – Otherwise, a *present decrement* action with the structured reference counter is per-
1205 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
1206 and dynamic reference counters are zero, a *delete* action is performed.

1207 The restrictions regarding subarrays in the **present** clause apply to this clause.

1208 For compatibility with OpenACC 2.0, **present_or_copyin** and **pcopyin** are alternate names
1209 for **copyin**.

1210 An **enter data** directive with a **copyin** clause is functionally equivalent to a call to the **acc_copyin**
1211 API routine, as described in Section 3.2.20.

1212 2.7.7. copyout clause

1213 The **copyout** clause may appear on structured **data** and compute constructs, on **declare** di-
1214 rectives, and on **exit data** directives. If the current device is a shared memory device, no action
1215 is performed.

1216 If the current device is a non-shared memory device, the **copyout** clause behaves as follows, for
1217 each *var* in *var-list*:

- 1218 • At entry to a region:
 - 1219 – If *var* is present, a *present increment* action with the structured reference counter is
1220 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 1221 – Otherwise, a *create* action with the structured reference is performed. If *var* is a pointer
1222 reference, an *attach* action is performed.
 - 1223 • At exit from a region, the structured reference counter is used. On an **exit data** directive,
1224 the dynamic reference counter is used.
 - 1225 – If *var* is not present on the current device, a runtime error is issued.
 - 1226 – Otherwise, the reference counter is updated:
 - 1227 * On an **exit data** directive with a **finalize** clause, the dynamic reference
1228 counter is set to zero.
 - 1229 * Otherwise, a *present decrement* action with the appropriate reference counter is
1230 performed.
- 1231 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic
1232 reference counters are zero, a *copyout* action is performed.

1233 The restrictions regarding subarrays in the **present** clause apply to this clause.

1234 For compatibility with OpenACC 2.0, **present_or_copyout** and **pcopyout** are alternate
1235 names for **copyout**.

1236 An **exit data** directive with a **copyout** clause and with or without a **finalize** clause is func-
1237 tionally equivalent to a call to the **acc_copyout_finalize** or **acc_copyout** API routine,
1238 respectively, as described in Section 3.2.22.

1239 **2.7.8. create clause**

1240 The **create** clause may appear on structured **data** and compute constructs, on **declare** direc-
 1241 tives, and on **enter data** directives. If the current device is a shared memory device, no action is
 1242 taken.

1243 If the current device is a non-shared memory device, the **create** clause behaves as follows, for
 1244 each *var* in *var-list*:

- 1245 • At entry to a region, the structured reference counter is used. On an **enter data** directive,
 1246 the dynamic reference counter is used.
 - 1247 – If *var* is present, a *present increment* action with the appropriate reference counter is
 1248 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 1249 – Otherwise, a *create* action with the appropriate reference counter is performed. If *var* is
 1250 a pointer reference, an *attach* action is performed.
- 1251 • At exit from the region:
 - 1252 – If *var* is not present on the current device, a runtime error is issued.
 - 1253 – Otherwise, a *present decrement* action with the structured reference counter is per-
 1254 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
 1255 and dynamic reference counters are zero, a *delete* action is performed.

1256 The restrictions regarding subarrays in the **present** clause apply to this clause.

1257 For compatibility with OpenACC 2.0, **present_or_create** and **pcreate** are alternate names
 1258 for **create**.

1259 An **enter data** directive with a **create** clause is functionally equivalent to a call to the **acc_create**
 1260 API routine, as described in Section 3.2.21.

1261 **2.7.9. no_create clause**

1262 The **no_create** clause may appear on structured **data** and compute constructs. If the current
 1263 device is a shared memory device, no action is taken.

1264 If the current device is a non-shared memory device, the **no_create** clause behaves as follows,
 1265 for each *var* in *var-list*:

- 1266 • At entry to the region:
 - 1267 – If *var* is present, a *present increment* action with the structured reference counter is
 1268 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 1269 – Otherwise, no action is performed, and any device code in this construct will use the
 1270 local memory address for *var*.
- 1271 • At exit from the region:
 - 1272 – If *var* is not present on the current device, no action is performed.
 - 1273 – Otherwise, a *present decrement* action with the structured reference counter is per-
 1274 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
 1275 and dynamic reference counters are zero, a *delete* action is performed.

1276 The restrictions regarding subarrays in the **present** clause apply to this clause.

1277 2.7.10. delete clause

1278 The **delete** clause may appear on **exit data** directives. If the current device is a shared memory
1279 device, no action is taken.

1280 If the current device is a non-shared memory device, the **delete** clause behaves as follows, for
1281 each *var* in *var-list*:

- 1282 • If *var* is not present on the current device, a runtime error is issued.
- 1283 • Otherwise, the dynamic reference counter is updated:
 - 1284 – On an **exit data** directive with a **finalize** clause, the dynamic reference counter
1285 is set to zero.
 - 1286 – Otherwise, a *present decrement* action with the dynamic reference counter is performed.
- 1287 If *var* is a pointer reference, a *detach* action is performed. If both structured and dynamic
1288 reference counters are zero, a *delete* action is performed.

1289 An **exit data** directive with a **delete** clause and with or without a **finalize** clause is func-
1290 tionally equivalent to a call to the **acc_delete_finalize** or **acc_delete** API routine, re-
1291 spectively, as described in Section 3.2.23.

1292 2.7.11. attach clause

1293 The **attach** clause may appear on structured **data** and compute constructs and on **enter data**
1294 directives. If the current device is a shared memory device, no action is taken. Each *var* argument
1295 to an **attach** clause must be a C or C++ pointer or a Fortran variable or array with the **pointer**
1296 or **allocatable** attribute.

1297 If the current device is a non-shared memory device, the **attach** clause behaves as follows, for
1298 each *var* in *var-list*:

- 1299 • At entry to a region or at an **enter data** directive, an *attach* action is performed.
- 1300 • At exit from the region, a *detach* action is performed.

1301 2.7.12. detach clause

1302 The **detach** clause may appear on **exit data** directives. If the current device is a shared memory
1303 device, no action is taken. Each *var* argument to a **detach** clause must be a C or C++ pointer or a
1304 Fortran variable or array with the **pointer** or **allocatable** attribute.

1305 If the current device is a non-shared memory device, the **detach** clause behaves as follows, for
1306 each *var* in *var-list*: If the current device is a non-shared memory device,

- 1307 • If there is a **finalize** clause on the **exit data** directive, an *immediate detach* action is
1308 performed.
- 1309 • Otherwise, a *detach* action is performed.

1310 2.8. Host_Data Construct

1311 **Summary** The `host_data` construct makes the address of device data available on the host.

1312 **Syntax** In C and C++, the syntax of the OpenACC `host_data` construct is

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

1313 and in Fortran, the syntax is

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

1314 where *clause* is one of the following:

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

1315 **Description** This construct is used to make the device address of data available in host code.

1316 Restrictions

- 1317 • At most one `if` clause may appear. In Fortran, the condition must evaluate to a scalar logical
- 1318 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1319 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
- 1320 `use_device` clauses.

1321 2.8.1. use_device clause

1322 The `use_device` clause tells the compiler to use the current device address of any variable or
 1323 array in *var-list* in code within the construct. In particular, this may be used to pass the device
 1324 address of variables or arrays to optimized procedures written in a lower-level API. When there is
 1325 no `if_present` clause, and either there is no `if` clause or the condition in the `if` clause evaluates
 1326 to nonzero (in C or C++) or `.true.` (in Fortran), the variables or arrays in *var-list* must be present
 1327 in the accelerator memory due to data regions or data lifetimes that contain this construct. On a
 1328 shared memory accelerator, the device address may be the same as the host address.

1329 2.8.2. if clause

1330 The `if` clause is optional. When an `if` clause appears and the condition evaluates to zero in C or
 1331 C++, or `.false.` in Fortran, the compiler will not change the addresses of any variable or array

1332 in code within the construct. When there is no **if** clause, or when an **if** clause appears and the
 1333 condition evaluates to nonzero in C or C++, or **.true.** in Fortran, the compiler will replace the
 1334 addresses as described in the previous subsection.

1335 2.8.3. **if_present** clause

1336 When an **if_present** clause appears on the directive, the compiler will only change the address
 1337 of any variable or array which appears in *var-list* that is present on the current device.

1338 2.9. Loop Construct

1339 **Summary** The OpenACC **loop** construct applies to a loop which must immediately follow this
 1340 directive. The **loop** construct can describe what type of parallelism to use to execute the loop and
 1341 declare private variables and arrays and reduction operations.

1342 **Syntax** In C and C++, the syntax of the **loop** construct is

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

1343 In Fortran, the syntax of the **loop** construct is

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

1344 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 )
```

1345 where *gang-arg* is one of:

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

1346 and *gang-arg-list* may have at most one **num** and one **static** argument,
 1347 and where *size-expr* is one of:

*
int-expr

1348 Some clauses are only valid in the context of a **kernels** construct; see the descriptions below.

1349 An *orphaned loop* construct is a **loop** construct that is not lexically enclosed within a **compute**
 1350 construct. The parent compute construct of a **loop** construct is the nearest compute construct that
 1351 lexically contains the **loop** construct.

1352 Restrictions

- 1353 • Only the **collapse**, **gang**, **worker**, **vector**, **seq**, **auto**, and **tile** clauses may follow
 1354 a **device_type** clause.
- 1355 • The *int-expr* argument to the **worker** and **vector** clauses must be invariant in the kernels
 1356 region.
- 1357 • A loop associated with a **loop** construct that does not have a **seq** clause must be written
 1358 such that the loop iteration count is computable when entering the **loop** construct.

1359 2.9.1. collapse clause

1360 The **collapse** clause is used to specify how many tightly nested loops are associated with the
 1361 **loop** construct. The argument to the **collapse** clause must be a constant positive integer expres-
 1362 sion. If no **collapse** clause is present, only the immediately following loop is associated with the
 1363 **loop** construct.

1364 If more than one loop is associated with the **loop** construct, the iterations of all the associated loops
 1365 are all scheduled according to the rest of the clauses. The trip count for all loops associated with the
 1366 **collapse** clause must be computable and invariant in all the loops.

1367 It is implementation-defined whether a **gang**, **worker** or **vector** clause on the construct is ap-
 1368 plied to each loop, or to the linearized iteration space.

1369 2.9.2. gang clause

1370 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
 1371 the **gang** clause specifies that the iterations of the associated loop or loops are to be executed in
 1372 parallel by distributing the iterations among the gangs created by the **parallel** construct. A
 1373 **loop** construct with the **gang** clause transitions a compute region from gang-redundant mode to
 1374 gang-partitioned mode. The number of gangs is controlled by the **parallel** construct; only the
 1375 **static** argument is allowed. The loop iterations must be data independent, except for variables
 1376 specified in a **reduction** clause. The region of a loop with the **gang** clause may not contain
 1377 another loop with the **gang** clause unless within a nested compute region.

1378 When the parent compute construct is a **kernels** construct, the **gang** clause specifies that the
1379 iterations of the associated loop or loops are to be executed in parallel across the gangs. An argument
1380 with no keyword or with the **num** keyword is allowed only when the **num_gangs** does not appear
1381 on the **kernels** construct. If an argument with no keyword or an argument after the **num** keyword
1382 is specified, it specifies how many gangs to use to execute the iterations of this loop. The region of a
1383 loop with the **gang** clause may not contain another loop with a **gang** clause unless within a nested
1384 compute region.

1385 The scheduling of loop iterations to gangs is not specified unless the **static** argument appears as
1386 an argument. If the **static** argument appears with an integer expression, that expression is used
1387 as a *chunk* size. If the **static** argument appears with an asterisk, the implementation will select a
1388 *chunk* size. The iterations are divided into chunks of the selected *chunk* size, and the chunks are
1389 assigned to gangs starting with gang zero and continuing in round-robin fashion. Two **gang** loops
1390 in the same parallel region with the same number of iterations, and with **static** clauses with the
1391 same argument, will assign the iterations to gangs in the same manner. Two **gang** loops in the
1392 same kernels region with the same number of iterations, the same number of gangs to use, and with
1393 **static** clauses with the same argument, will assign the iterations to gangs in the same manner.

1394 2.9.3. worker clause

1395 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
1396 the **worker** clause specifies that the iterations of the associated loop or loops are to be executed
1397 in parallel by distributing the iterations among the multiple workers within a single gang. A **loop**
1398 construct with a **worker** clause causes a gang to transition from worker-single mode to worker-
1399 partitioned mode. In contrast to the **gang** clause, the **worker** clause first activates additional
1400 worker-level parallelism and then distributes the loop iterations across those workers. No argu-
1401 ment is allowed. The loop iterations must be data independent, except for variables specified in a
1402 **reduction** clause. The region of a loop with the **worker** clause may not contain a loop with the
1403 **gang** or **worker** clause unless within a nested compute region.

1404 When the parent compute construct is a **kernels** construct, the **worker** clause specifies that the
1405 iterations of the associated loop or loops are to be executed in parallel across the workers within
1406 a **single** gang. An argument is allowed only when the **num_workers** does not appear on the
1407 **kernels** construct. The optional argument specifies how many workers per gang to use to execute
1408 the iterations of this loop. The region of a loop with the **worker** clause may not contain a loop
1409 with a **gang** or **worker** clause unless within a nested compute region.

1410 All workers will complete execution of their assigned iterations before any worker proceeds beyond
1411 the end of the loop.

1412 2.9.4. vector clause

1413 When the parent compute construct is a **parallel** construct, or on an orphaned **loop** construct,
1414 the **vector** clause specifies that the iterations of the associated loop or loops are to be executed in
1415 vector or SIMD mode. A **loop** construct with a **vector** clause causes a worker to transition from
1416 vector-single mode to vector-partitioned mode. Similar to the **worker** clause, the **vector** clause
1417 first activates additional vector-level parallelism and then distributes the loop iterations across those
1418 vector lanes. The operations will execute using vectors of the length specified or chosen for the

1419 parallel region. The region of a loop with the **vector** clause may not contain a loop with the
1420 **gang**, **worker**, or **vector** clause unless within a nested compute region.

1421 When the parent compute construct is a **kernels** construct, the **vector** clause specifies that the
1422 iterations of the associated loop or loops are to be executed with vector or SIMD processing. An
1423 argument is allowed only when the **vector_length** does not appear on the **kernels** construct.
1424 If an argument is specified, the iterations will be processed in vector strips of that length; if no
1425 argument is specified, the implementation will choose an appropriate vector length. The region of
1426 a loop with the **vector** clause may not contain a loop with a **gang**, **worker**, or **vector** clause
1427 unless within a nested compute region.

1428 All vector lanes will complete execution of their assigned iterations before any vector lane proceeds
1429 beyond the end of the loop.

1430 2.9.5. seq clause

1431 The **seq** clause specifies that the associated loop or loops are to be executed sequentially by the
1432 accelerator. This clause will override any automatic parallelization or vectorization.

1433 2.9.6. auto clause

1434 The **auto** clause specifies that the implementation must analyze the loop and determine whether
1435 the loop iterations are data independent and, if so, select whether to apply parallelism to this loop
1436 or whether to run the loop sequentially. The implementation may be restricted to the types of
1437 parallelism it can apply by the presence of **loop** constructs with **gang**, **worker**, or **vector**
1438 clauses for outer or inner loops. When the parent compute construct is a **kernels** construct, a
1439 **loop** construct with no **independent** or **seq** clause is treated as if it has the **auto** clause.

1440 2.9.7. tile clause

1441 The **tile** clause specifies that the implementation should split each loop in the loop nest into two
1442 loops, with an outer set of *tile* loops and an inner set of *element* loops. The argument to the **tile**
1443 clause is a list of one or more tile sizes, where each tile size is a constant positive integer expression
1444 or an asterisk. If there are n tile sizes in the list, the **loop** construct must be immediately followed
1445 by n tightly-nested loops. The first argument in the *size-expr-list* corresponds to the innermost loop
1446 of the n associated loops, and the last element corresponds to the outermost associated loop. If the
1447 tile size is specified with an asterisk, the implementation will choose an appropriate value. Each
1448 loop in the nest will be split or *strip-mined* into two loops, an outer *tile* loop and an inner *element*
1449 loop. The trip count of the element loop will be limited to the corresponding tile size from the
1450 *size-expr-list*. The *tile* loops will be reordered to be outside all the *element* loops, and the *element*
1451 loops will all be inside the *tile* loops.

1452 If the **vector** clause appears on the **loop** construct, the **vector** clause is applied to the *element*
1453 loops. If the **gang** clause appears on the **loop** construct, the **gang** clause is applied to the *tile*
1454 loops. If the **worker** clause appears on the **loop** construct, the **worker** clause is applied to the
1455 *element* loops if no **vector** clause appears, and to the *tile* loops otherwise.

1456 2.9.8. `device_type` clause

1457 The `device_type` clause is described in Section 2.4 Device-Specific Clauses.

1458 2.9.9. `independent` clause

1459 The `independent` clause tells the implementation that the iterations of this loop are data-independent
1460 with respect to each other. This allows the implementation to generate code to execute the iterations
1461 in parallel with no synchronization. When the parent compute construct is a `parallel` construct,
1462 the `independent` clause is implied on all `loop` constructs without a `seq` or `auto` clause.

1463 Note

- 1464 • It is likely a programming error to use the `independent` clause on a loop if any iteration
1465 writes to a variable or array element that any other iteration also writes or reads, except for
1466 variables in a `reduction` clause or accesses in atomic regions.

1467 2.9.10. `private` clause

1468 The `private` clause on a `loop` construct specifies that a copy of each item in *var-list* will be
1469 created. If the body of the loop is executed in *vector-partitioned* mode, a copy of the item is created
1470 for each thread associated with each vector lane. If the body of the loop is executed in *worker-*
1471 *partitioned vector-single* mode, a copy of the item is created for and shared across the set of threads
1472 associated with all the vector lanes of each worker. Otherwise, a copy of the item is created for and
1473 shared across the set of threads associated with all the vector lanes of all the workers of each gang.

1474 Restrictions

- 1475 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
1476 `private` clauses.

1477 2.9.11. `reduction` clause

1478 The `reduction` clause specifies a reduction operator and one or more scalar variables. For each
1479 reduction variable, a private copy is created in the same manner as for a `private` clause on the
1480 `loop` construct, and initialized for that operator; see the table in Section 2.5.12 reduction clause. At
1481 the end of the loop, the values for each thread are combined using the specified reduction operator,
1482 and the result combined with the value of the original variable and stored in the original variable at
1483 the end of the parallel or kernels region if the loop has gang parallelism, and at the end of the loop
1484 otherwise.

1485 In a parallel region, if the `reduction` clause is used on a loop with the `vector` or `worker`
1486 clauses (and no `gang` clause), and the scalar variable also appears in a `private` clause on the
1487 `parallel` construct, the value of the private copy of the scalar will be updated at the exit of the
1488 loop. If the scalar variable does not appear in a `private` clause on the `parallel` construct, or if
1489 the `reduction` clause is used on a loop with the `gang` clause, the value of the scalar will not be
1490 updated until the end of the parallel region.

1491 If a variable is involved in a reduction that spans multiple nested loops where two or more of those
 1492 loops have associated **loop** directives, a **reduction** clause containing that variable must appear
 1493 on each of those **loop** directives.

1494 Restrictions

- 1495 • The **reduction** clause may not be specified on an orphaned **loop** construct with the **gang**
 1496 clause, or on an orphaned **loop** construct that will generate gang parallelism in a procedure
 1497 that is compiled with the **routine gang** clause.
- 1498 • The restrictions for a **reduction** clause on a compute construct listed in in Section 2.5.12
 1499 reduction clause also apply to a **reduction** clause on a loop construct.
- 1500 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
 1501 **reduction** clauses.

1502 2.10. Cache Directive

1503 **Summary** The **cache** directive may appear at the top of (inside of) a loop. It specifies array
 1504 elements or subarrays that should be fetched into the highest level of the cache for the body of the
 1505 loop.

1506 **Syntax** In C and C++, the syntax of the cache directive is

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

1507 In Fortran, the syntax of the cache directive is

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

1508 The entries in *var-list* must be single array elements or simple subarray. In C and C++, a simple
 1509 subarray is an array name followed by an extended array range specification in brackets, with start
 1510 and length, such as

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

1511 where the lower bound is a constant, loop invariant, or the **for** loop index variable plus or minus a
 1512 constant or loop invariant, and the length is a constant.

1513 In Fortran, a simple subarray is an array name followed by a comma-separated list of range specifi-
 1514 cations in parentheses, with lower and upper bound subscripts, such as

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

1515 The lower bounds must be constant, loop invariant, or the **do** loop index variable plus or minus
 1516 a constant or loop invariant; moreover the difference between the corresponding upper and lower
 1517 bounds must be a constant.

1518 **Restrictions**

- 1519 • If an array is listed in a **cache** directive, all references to that array during execution of that
- 1520 loop iteration must not refer to elements of the array outside the index range specified in the
- 1521 **cache** directive.
- 1522 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
- 1523 **cache** directives.

1524 **2.11. Combined Constructs**

1525 **Summary** The combined OpenACC **parallel loop**, **kernels loop**, and **serial loop**

1526 constructs are shortcuts for specifying a **loop** construct nested immediately inside a **parallel**,

1527 **kernels**, or **serial** construct. The meaning is identical to explicitly specifying a **parallel**,

1528 **kernels**, or **serial** construct containing a **loop** construct. Any clause that is allowed on a

1529 **parallel** or **loop** construct is allowed on the **parallel loop** construct; any clause allowed

1530 on a **kernels** or **loop** construct is allowed on a **kernels loop** construct; and any clause

1531 allowed on a **serial** or **loop** construct is allowed on a **serial loop** construct.

1532 **Syntax** In C and C++, the syntax of the **parallel loop** construct is

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

1533 In Fortran, the syntax of the **parallel loop** construct is

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

1534 The associated structured block is the loop which must immediately follow the directive. Any of the

1535 **parallel** or **loop** clauses valid in a parallel region may appear. The **private** and **reduction**

1536 clauses, which can appear on both a **parallel** construct and a **loop** construct, are treated on a

1537 **parallel loop** construct as if they appeared on the **loop** construct.

1538 In C and C++, the syntax of the **kernels loop** construct is

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

1539 In Fortran, the syntax of the **kernels loop** construct is

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

1540 The associated structured block is the loop which must immediately follow the directive. Any of
 1541 the **kernels** or **loop** clauses valid in a kernels region may appear.

1542 In C and C++, the syntax of the **serial loop** construct is

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

   for loop
```

1543 In Fortran, the syntax of the **serial loop** construct is

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

   do loop  

[!$acc end serial loop]
```

1544 The associated structured block is the loop which must immediately follow the directive. Any of
 1545 the **serial** or **loop** clauses valid in a serial region may appear. The **private** clause, which can
 1546 appear on both a **serial** construct and a **loop** construct, is treated on a **serial loop** construct
 1547 as if it appeared on the **loop** construct.

1548 Restrictions

- 1549 • The restrictions for the **parallel**, **kernels**, **serial**, and **loop** constructs apply.

1550 2.12. Atomic Construct

1551 **Summary** An **atomic** construct ensures that a specific storage location is accessed and/or up-
 1552 dated atomically, preventing simultaneous reading and writing by gangs, workers, and vector threads
 1553 that could result in indeterminate values.

1554 **Syntax** In C and C++, the syntax of the **atomic** constructs is:

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

   expression-stmt
```

1555 OR:

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

   structured-block
```

1556 Where *atomic-clause* is one of **read**, **write**, **update**, or **capture**. The *expression-stmt* is an
 1557 expression statement with one of the following forms:

1558 If the *atomic-clause* is **read**:

```
v = x;
```

1559 If the *atomic-clause* is **write**:

```
x = expr;
```

1560 If the *atomic-clause* is **update** or not present:

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

1561 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;
```

1562 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; }
```

1563 In the preceding expressions:

- 1564 • **x** and **v** (as applicable) are both l-value expressions with scalar type.
- 1565 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
- 1566 the same storage location.

- 1567 • Neither of **v** and *expr* (as applicable) may access the storage location designated by **x**.
- 1568 • Neither of **x** and *expr* (as applicable) may access the storage location designated by **v**.
- 1569 • *expr* is an expression with scalar type.
- 1570 • *binop* is one of **+**, *****, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.
- 1571 • *binop*, *binop=*, **++**, and **--** are not overloaded operators.
- 1572 • The expression **x binop expr** must be mathematically equivalent to **x binop (expr)**. This
1573 requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by
1574 using parentheses around *expr* or subexpressions of *expr*.
- 1575 • The expression *expr binop x* must be mathematically equivalent to *(expr) binop x*. This
1576 requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*,
1577 or by using parentheses around *expr* or subexpressions of *expr*.
- 1578 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is
1579 unspecified.

1580 In Fortran the syntax of the **atomic** constructs is:

```
!$acc atomic read
    capture-statement
!$acc end atomic]
```

1581 OR

```
!$acc atomic write
    write-statement
!$acc end atomic]
```

1582 OR

```
!$acc atomic [update]
    update-statement
!$acc end atomic]
```

1583 OR

```
!$acc atomic capture
    update-statement
    capture-statement
!$acc end atomic]
```

1584 OR

```
!$acc atomic capture
    capture-statement
    update-statement
!$acc end atomic]
```

1585 OR

```

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

```

1586 where *write-statement* has the following form (if *atomic-clause* is **write** or **capture**):

```
x = expr
```

1587 where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

```
v = x
```

1588 and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **capture**,
1589 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 )

```

1590 In the preceding statements:

- 1591 • **x** and **v** (as applicable) are both scalar variables of intrinsic type.
- 1592 • **x** must not be an allocatable variable.
- 1593 • During the execution of an atomic region, multiple syntactic occurrences of **x** must designate
1594 the same storage location.
- 1595 • None of **v**, *expr*, and *expr-list* (as applicable) may access the same storage location as **x**.
- 1596 • None of **x**, *expr*, and *expr-list* (as applicable) may access the same storage location as **v**.
- 1597 • *expr* is a scalar expression.
- 1598 • *expr-list* is a comma-separated, non-empty list of scalar expressions. If *intrinsic_procedure_name*
1599 refers to **iand**, **ior**, or **ieor**, exactly one expression must appear in *expr-list*.
- 1600 • *intrinsic_procedure_name* is one of **max**, **min**, **iand**, **ior**, or **ieor**. *operator* is one of **+**,
1601 *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**
- 1602 • The expression **x operator expr** must be mathematically equivalent to **x operator (expr)**.
1603 This requirement is satisfied if the operators in *expr* have precedence greater than *operator*,
1604 or by using parentheses around *expr* or subexpressions of *expr*.
- 1605 • The expression *expr operator x* must be mathematically equivalent to **(expr) operator x**.
1606 This requirement is satisfied if the operators in *expr* have precedence equal to or greater than
1607 *operator*, or by using parentheses around *expr* or subexpressions of *expr*.

- 1608 • *intrinsic_procedure_name* must refer to the intrinsic procedure name and not to other program
1609 entities.
- 1610 • *operator* must refer to the intrinsic operator and not to a user-defined operator. All assign-
1611 ments must be intrinsic assignments.
- 1612 • For forms that allow multiple occurrences of **x**, the number of times that **x** is evaluated is
1613 unspecified.

1614 An **atomic** construct with the **read** clause forces an atomic read of the location designated by **x**.
1615 An **atomic** construct with the **write** clause forces an atomic write of the location designated by
1616 **x**.

1617 An **atomic** construct with the **update** clause forces an atomic update of the location designated
1618 by **x** using the designated operator or intrinsic. Note that when no clause is present, the semantics
1619 are equivalent to **atomic update**. Only the read and write of the location designated by **x** are
1620 performed mutually atomically. The evaluation of *expr* or *expr-list* need not be atomic with respect
1621 to the read or write of the location designated by **x**.

1622 An **atomic** construct with the **capture** clause forces an atomic update of the location designated
1623 by **x** using the designated operator or intrinsic while also capturing the original or final value of
1624 the location designated by **x** with respect to the atomic update. The original or final value of the
1625 location designated by **x** is written into the location designated by **v** depending on the form of the
1626 **atomic** construct structured block or statements following the usual language semantics. Only
1627 the read and write of the location designated by **x** are performed mutually atomically. Neither the
1628 evaluation of *expr* or *expr-list*, nor the write to the location designated by **v**, need to be atomic with
1629 respect to the read or write of the location designated by **x**.

1630 For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs
1631 enforces mutually exclusive access to the locations designated by **x**. To avoid race conditions, all
1632 accesses of the locations designated by **x** that could potentially occur in parallel must be protected
1633 with an **atomic** construct.

1634 Atomic regions do not guarantee exclusive access with respect to any accesses outside of atomic re-
1635 gions to the same storage location **x** even if those accesses occur during the execution of a reduction
1636 clause.

1637 If the storage location designated by **x** is not size-aligned (that is, if the byte alignment of **x** is not a
1638 multiple of the size of **x**), then the behavior of the atomic region is implementation-defined.

1639 Restrictions

- 1640 • All atomic accesses to the storage locations designated by **x** throughout the program are
1641 required to have the same type and type parameters.
- 1642 • Storage locations designated by **x** must be less than or equal in size to the largest available
1643 native atomic operator width.

1644 2.13. Declare Directive

1645 **Summary** A **declare** directive is used in the declaration section of a Fortran subroutine, func-
 1646 tion, or module, or following a variable declaration in C or C++. It can specify that a variable or
 1647 array is to be allocated in the device memory for the duration of the implicit data region of a func-
 1648 tion, subroutine or program, and specify whether the data values are to be transferred from the host
 1649 to the device memory upon entry to the implicit data region, and from the device to the host memory
 1650 upon exit from the implicit data region. These directives create a visible device copy of the variable
 1651 or array.

1652 **Syntax** In C and C++, the syntax of the **declare** directive is:

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

1653 In Fortran the syntax of the **declare** directive is:

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

1654 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 )
```

1655 The associated region is the implicit region associated with the function, subroutine, or program in
 1656 which the directive appears. If the directive appears in the declaration section of a Fortran *module*
 1657 subprogram or in a C or C++ global scope, the associated region is the implicit region for the whole
 1658 program. The **copy**, **copyin**, **copyout**, **present**, and **deviceptr** data clauses are described
 1659 in Section 2.7 Data Clauses.

1660 Restrictions

- 1661 • A **declare** directive must appear in the same scope as any variable or array in any of the
 1662 data clauses on the directive.
- 1663 • A variable or array may appear at most once in all the clauses of **declare** directives for a
 1664 function, subroutine, program, or module.
- 1665 • Subarrays are not allowed in **declare** directives.
- 1666 • In Fortran, assumed-size dummy arrays may not appear in a **declare** directive.

- 1667 • In Fortran, pointer arrays may be specified, but pointer association is not preserved in the
1668 device memory.
- 1669 • In a Fortran *module* declaration section, only **create**, **copyin**, **device_resident**, and
1670 **link** clauses are allowed.
- 1671 • In C or C++ global scope, only **create**, **copyin**, **deviceptr**, **device_resident** and
1672 **link** clauses are allowed.
- 1673 • C and C++ *extern* variables may only appear in **create**, **copyin**, **deviceptr**, **device_resident**
1674 and **link** clauses on a **declare** directive.
- 1675 • In C and C++, only global and *extern* variables may appear in a **link** clause. In Fortran,
1676 only *module* variables and *common* block names (enclosed in slashes) may appear in a **link**
1677 clause.
- 1678 • In C or C++, a **longjmp** call in the region must return to a **setjmp** call within the region.
- 1679 • In C++, an exception thrown in the region must be handled within the region.
- 1680 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional dummy ar-
1681 guments in data clauses, including **device_resident** clauses.

1682 2.13.1. device_resident clause

1683 **Summary** The **device_resident** clause specifies that the memory for the named variables
1684 should be allocated in the accelerator device memory and not in the host memory. The names
1685 in the argument list may be variable or array names, or Fortran *common block* names enclosed
1686 between slashes; subarrays are not allowed. The host may not be able to access variables in a
1687 **device_resident** clause. The accelerator data lifetime of global variables or common blocks
1688 specified in a **device_resident** clause is the entire execution of the program.

1689 In Fortran, if the variable has the Fortran *allocatable* attribute, the memory for the variable will
1690 be allocated in and deallocated from the current accelerator device memory when the host program
1691 executes an **allocate** or **deallocate** statement for that variable. If the variable has the Fortran
1692 *pointer* attribute, it may be allocated or deallocated by the host in the accelerator device memory, or
1693 may appear on the left hand side of a pointer assignment statement, if the right hand side variable
1694 itself appears in a **device_resident** clause.

1695 In Fortran, the argument to a **device_resident** clause may be a *common block* name enclosed
1696 in slashes; in this case, all declarations of the common block must have a matching **device_resident**
1697 clause. In this case, the *common block* will be statically allocated in device memory, and not
1698 in host memory. The *common block* will be available to accelerator routines; see Section 2.15
1699 Procedure Calls in Compute Regions.

1700 In a Fortran *module* declaration section, a variable or array in a **device_resident** clause will
1701 be available to accelerator subprograms.

1702 In C or C++ global scope, a variable or array in a **device_resident** clause will be available
1703 to accelerator routines. A C or C++ *extern* variable may appear in a **device_resident** clause
1704 only if the actual declaration and all *extern* declarations are also followed by **device_resident**
1705 clauses.

1706 2.13.2. create clause

1707 If the current device is a shared memory device, no action is taken.

1708 If the current device is a non-shared memory device, the **create** clause behaves as follows, for
1709 each *var* in *var-list*:

- 1710 • At entry to an implicit data region where the **declare** directive appears:
 - 1711 – If *var* is present, a *present increment* action with the structured reference counter is
1712 performed. If *var* is a pointer reference, an *attach* action is performed.
 - 1713 – Otherwise, a *create* action with the structured reference counter is performed. If *var* is
1714 a pointer reference, an *attach* action is performed.
- 1715 • At exit from an implicit data region where the **declare** directive appears:
 - 1716 – If *var* is not present on the current device, a runtime error is issued.
 - 1717 – Otherwise, a *present decrement* action with the structured reference counter is per-
1718 formed. If *var* is a pointer reference, a *detach* action is performed. If both structured
1719 and dynamic reference counters are zero, a *delete* action is performed.

1720 If the **declare** directive appears in a global context, then the data in *var-list* is statically allocated
1721 in device memory and the structured reference counter is set to one.

1722 In Fortran, if a variable *var* in *var-list* has the Fortran *allocatable* or *pointer* attribute, then:

- 1723 • An **allocate** statement for *var* will allocate memory from both host memory as well as the
1724 current accelerator device memory, and the dynamic reference counter will be set to one.
- 1725 • A **deallocate** statement for *var* will deallocate memory from both host memory as well as
1726 the current accelerator device memory, and the dynamic reference counter will be set to zero.
1727 If the structured reference counter is not zero, a runtime error is issued.

1728 In Fortran, if a variable *var* in *var-list* has the Fortran *pointer* attribute, then it may appear on the
1729 left hand side of a pointer assignment statement, if the right hand side variable itself appears in a
1730 **create** clause.

1731 2.13.3. link clause

1732 The **link** clause is used for large global host static data that is referenced within an accelerator
1733 routine and that should have a dynamic data lifetime on the device. The **link** clause specifies that
1734 only a global link for the named variables should be statically created in accelerator memory. The
1735 host data structure remains statically allocated and globally available. The device data memory will
1736 be allocated only when the global variable appears on a data clause for a **data** construct, compute
1737 construct, or **enter data** directive. The arguments to the **link** clause must be global data. In C
1738 or C++, the **link** clause must appear at global scope, or the arguments must be *extern* variables.
1739 In Fortran, the **link** clause must appear in a *module* declaration section, or the arguments must be
1740 *common block* names enclosed in slashes. A *common block* that is listed in a **link** clause must be
1741 declared with the same size in all program units where it appears. A **declare link** clause must
1742 be visible everywhere the global variables or common block variables are explicitly or implicitly
1743 used in a data clause, compute construct, or accelerator routine. The global variable or *common*
1744 *block* variables may be used in accelerator routines. The accelerator data lifetime of variables or

1745 common blocks specified in a **link** clause is the data region that allocates the variable or common
 1746 block with a data clause, or from the execution of the **enter data** directive that allocates the data
 1747 until an **exit data** directive deallocates it or until the end of the program.

1748 2.14. Executable Directives

1749 2.14.1. Init Directive

1750 **Summary** The **init** directive tells the runtime to initialize the runtime for that device type.
 1751 This can be used to isolate any initialization cost from the computational cost, when collecting
 1752 performance statistics. If no device type is specified all devices will be initialized. An **init**
 1753 directive may be used in place of a call to the **acc_init** runtime API routine, as described in
 1754 Section 3.2.7.

1755 **Syntax** In C and C++, the syntax of the **init** directive is:

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

1756 In Fortran the syntax of the **init** directive is:

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

1757 where *clause* is one of the following:

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

1758 device_type clause

1759 The **device_type** clause specifies the type of device that is to be initialized in the runtime. If the
 1760 **device_type** clause is present, then the *acc-device-type-var* for the current thread is set to the
 1761 argument value. If no **device_num** clause is present then all devices of this type are initialized.

1762 device_num clause

1763 The **device_num** clause specifies the device id to be initialized. If the **device_num** clause
 1764 is present, then the *acc-device-num-var* for the current thread is set to the argument value. If no
 1765 **device_type** clause is specified, then the specified device id will be initialized for all available
 1766 device types.

1767 Restrictions

- 1768 • This directive may not be called within a **compute** region.

- 1769 • If the device type specified is not available, the behavior is implementation-defined; in partic-
1770 ular, the program may abort.
- 1771 • If the directive is called more than once without an intervening **acc_shutdown** call or
1772 **shutdown** directive, with a different value for the device type argument, the behavior is
1773 implementation-defined.
- 1774 • If some accelerator regions are compiled to only use one device type, using this directive with
1775 a different device type may produce undefined behavior.

1776 2.14.2. Shutdown Directive

1777 **Summary** The **shutdown** directive tells the runtime to shut down the connection to the given
1778 accelerator device, and free any runtime resources. A **shutdown** directive may be used in place of
1779 a call to the **acc_shutdown** runtime API routine, as described in Section 3.2.8.

1780 **Syntax** In C and C++, the syntax of the **shutdown** directive is:

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

1781 In Fortran the syntax of the **shutdown** directive is:

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

1782 where *clause* is one of the following:

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

1783 **device_type** clause

1784 The **device_type** clause specifies the type of device that is to be disconnected from the runtime.
1785 If no **device_num** clause is present then all devices of this type are disconnected.

1786 **device_num** clause

1787 The **device_num** clause specifies the device id to be disconnected.
1788 If no clauses are present then all available devices will be disconnected.

1789 **Restrictions**

- 1790 • This directive may not be used during the execution of a compute region.

1791 2.14.3. Set Directive

1792 **Summary** The **set** directive provides a means to modify internal control variables using direc-
1793 tives. Each form of the **set** directive is functionally equivalent to a matching runtime API routine.

1794 **Syntax** In C and C++, the syntax of the **set** directive is:

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

1795 In Fortran the syntax of the **set** directive is:

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

1796 where *clause* is one of the following

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

1797 **default_async clause**

1798 The **default_async** clause specifies the asynchronous queue that should be used if no queue
1799 is specified and changes the value of *acc-default-async-var* for the current thread to the argument
1800 value. If the value is **acc_async_default**, the value of *acc-default-async-var* will revert to
1801 the initial value, which is implementation-defined. A **set default_async** directive is function-
1802 ally equivalent to a call to the **acc_set_default_async** runtime API routine, as described in
1803 Section 3.2.16.

1804 **device_num clause**

1805 The **device_num** clause specifies the device number to set as the default device for accelerator
1806 regions and changes the value of *acc-device-num-var* for the current thread to the argument value.
1807 If the value of **device_num** argument is negative, the runtime will revert to the default behavior,
1808 which is implementation-defined. A **set device_num** directive is functionally equivalent to the
1809 **acc_set_device_num** runtime API routine, as described in Section 3.2.4.

1810 **device_type clause**

1811 The **device_type** clause specifies the device type to set as the default device type for accelerator
1812 regions and sets the value of *acc-device-type-var* for the current thread to the argument value. If
1813 the value of the **device_type** argument is zero or the clause is not present, the selected device
1814 number will be used for all attached accelerator types. A **set device_type** directive is function-
1815 ally equivalent to a call to the **acc_set_device_type** runtime API routine, as described in
1816 Section 3.2.2.

1817 **Restrictions**

- 1818 • This directive may not be used within a **compute** region.
- 1819 • Passing **default_async** the value of **acc_async_noval** has no effect.

- 1820 • Passing **default_async** the value of **acc_async_sync** will cause all asynchronous
- 1821 directives in the default asynchronous queue to become synchronous.
- 1822 • Passing **default_async** the value of **acc_async_default** will restore the default
- 1823 asynchronous queue to the initial value, which is implementation-defined.
- 1824 • If the value of **device_num** is larger than the maximum supported value for the given type,
- 1825 the behavior is implementation-defined.
- 1826 • At least one clause must be specified.
- 1827 • Two instances of the same clause may not appear on the same directive.

1828 2.14.4. Update Directive

1829 **Summary** The **update** directive is used during the lifetime of accelerator data to update all or
 1830 part of local variables or arrays with values from the corresponding data in device memory, or to
 1831 update all or part of device variables or arrays with values from the corresponding data in local
 1832 memory.

1833 **Syntax** In C and C++, the syntax of the **update** directive is:

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

1834 In Fortran the syntax of the **update** data directive is:

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

1835 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 )
```

1836 The *var-list* argument to an **update** clause is a comma-separated collection of variable names,
 1837 array names, or subarray specifications. Multiple subarrays of the same array may appear in a *var-*
 1838 *list* of the same or different clauses on the same directive. The effect of an **update** clause is to
 1839 copy data from the accelerator device memory to the local memory for **update self**, and from
 1840 local memory to accelerator device memory for **update device**. The updates are done in the
 1841 order in which they appear on the directive. No action is taken for a variable or array in the **self**
 1842 or **device** clause if there is no device copy of that variable or array. At least one **self**, **host**, or
 1843 **device** clause must appear on the directive.

1844 **self clause**

1845 The **self** clause specifies that the variables, arrays, or subarrays in *var-list* are to be copied from
1846 the current accelerator device memory to the local memory for a non-shared memory accelerator. If
1847 the current accelerator shares memory with the encountering thread, no action is taken. An **update**
1848 directive with the **self** clause is equivalent to a call to the **acc_update_self** routine, described
1849 in Section 3.2.25.

1850 **host clause**

1851 The **host** clause is a synonym for the **self** clause.

1852 **device clause**

1853 The **device** clause specifies that the variables, arrays, or subarrays in *var-list* are to be copied from
1854 the local memory to the current accelerator device memory, for a non-shared memory accelerator.
1855 If the current accelerator shares memory with the encountering thread, no action is taken. directive
1856 with the **device** clause is equivalent to a call to the **acc_update_device** routine, described
1857 in Section 3.2.24.

1858 **if clause**

1859 The **if** clause is optional; when there is no **if** clause, the implementation will generate code to
1860 perform the updates unconditionally. When an **if** clause appears, the implementation will generate
1861 code to conditionally perform the updates only when the *condition* evaluates to nonzero in C or
1862 C++, or **.true.** in Fortran.

1863 **async clause**

1864 The **async** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1865 **wait clause**

1866 The **wait** clause is optional; see Section 2.16 Asynchronous Behavior for more information.

1867 **if_present clause**

1868 When an **if_present** clause appears on the directive, no action is taken for a variable or array
1869 which appears in *var-list* that is not present on the current device. When no **if_present** clause
1870 appears, all variables and arrays in a **device** or **self** clause must be present on the current device,
1871 and an implementation may halt the program with an error message if some data is not present.

1872 **Restrictions**

- 1873 • The **update** directive is executable. It must not appear in place of the statement following
1874 an *if*, *while*, *do*, *switch*, or *label* in C or C++, or in place of the statement following a logical
1875 *if* in Fortran.
- 1876 • If no **if_present** clause appears on the directive, each variable and array **that** appears in
1877 *var-list* must be present on the current device.
- 1878 • A variable or array **that** appears in *var-list* must be present on the current device.
- 1879 • Only the **async** and **wait** clauses may follow a **device_type** clause.
- 1880 • At most one **if** clause may appear. In Fortran, the condition must evaluate to a scalar logical
1881 value; in C or C++, the condition must evaluate to a scalar integer value.
- 1882 • Noncontiguous subarrays may be specified. It is implementation-specific whether noncon-
1883 tiguous regions are updated by using one transfer for each contiguous subregion, or whether
1884 the noncontiguous data is packed, transferred once, and unpacked, or whether one or more
1885 larger subarrays (no larger than the smallest contiguous region that contains the specified
1886 subarray) are updated.
- 1887 • In C and C++, a member of a struct or class may be specified, including a subarray of a
1888 member. Members of a subarray of struct or class type may not be specified.
- 1889 • In C and C++, if a subarray notation is used for a struct member, subarray notation may not
1890 be used for any parent of that struct member.
- 1891 • In Fortran, members of variables of derived type may be specified, including a subarray of a
1892 member. Members of subarrays of derived type may not be specified.
- 1893 • In Fortran, if array or subarray notation is used for a derived type member, array or subarray
1894 notation may not be used for a parent of that derived type member.
- 1895 • See Section 2.17 Fortran Optional Arguments for discussion of Fortran optional arguments in
1896 **self**, **host**, and **device** clauses.

1897 2.14.5. Wait Directive

1898 See Section 2.16 Asynchronous Behavior for more information.

1899 2.14.6. Enter Data Directive

1900 See Section 2.6.5 Enter Data and Exit Data Directives for more information.

1901 2.14.7. Exit Data Directive

1902 See Section 2.6.5 Enter Data and Exit Data Directives for more information.

1903 2.15. Procedure Calls in Compute Regions

1904 This section describes how routines are compiled for an accelerator and how procedure calls are
1905 compiled in compute regions. See Section 2.17 Fortran Optional Arguments for discussion of For-

1906 tran optional arguments in procedure calls inside compute regions.

1907 2.15.1. Routine Directive

1908 **Summary** The **routine** directive is used to tell the compiler to compile a given procedure for
 1909 an accelerator as well as for the host. In a file or routine with a procedure call, the **routine**
 1910 directive tells the implementation the attributes of the procedure when called on the accelerator.

1911 **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
```

1912 In C and C++, the **routine** directive without a name may appear immediately before a function
 1913 definition or just before a function prototype and applies to that immediately following function or
 1914 prototype. The **routine** directive with a name may appear anywhere that a function prototype
 1915 is allowed and applies to the function in that scope with that name, but must appear before any
 1916 definition or use of that function.

1917 In Fortran the syntax of the **routine** directive is:

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

1918 In Fortran, the **routine** directive without a name may appear within the specification part of a
 1919 subroutine or function definition, or within an interface body for a subroutine or function in an
 1920 interface block, and applies to the containing subroutine or function. The **routine** directive with
 1921 a name may appear in the specification part of a subroutine, function or module, and applies to the
 1922 named subroutine or function.

1923 A C or C++ function or Fortran subprogram compiled with the **routine** directive for an accelera-
 1924 tor is called an *accelerator routine*.

1925 The *clause* is one of the following:

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

1926 A **gang**, **worker**, **vector**, or **seq** clause specifies the *level of parallelism* in the routine.

1927 gang clause

1928 The **gang** clause specifies that the procedure contains, may contain, or may call another procedure
1929 that contains a loop with a **gang** clause. A call to this procedure must appear in code that is
1930 executed in *gang-redundant* mode, and all gangs must execute the call. For instance, a procedure
1931 with a **routine gang** directive may not be called from within a loop that has a **gang** clause.
1932 Only one of the **gang**, **worker**, **vector** and **seq** clauses may be specified for each device type.

1933 worker clause

1934 The **worker** clause specifies that the procedure contains, may contain, or may call another pro-
1935 cedure that contains a loop with a **worker** clause, but does not contain nor does it call another
1936 procedure that contains a loop with the **gang** clause. A loop in this procedure with an **auto** clause
1937 may be selected by the compiler to execute in **worker** or **vector** mode. A call to this procedure
1938 must appear in code that is executed in *worker-single* mode, though it may be in *gang-redundant*
1939 or *gang-partitioned* mode. For instance, a procedure with a **routine worker** directive may be
1940 called from within a loop that has the **gang** clause, but not from within a loop that has the **worker**
1941 clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may be specified for each
1942 device type.

1943 vector clause

1944 The **vector** clause specifies that the procedure contains, may contain, or may call another pro-
1945 cedure that contains a loop with the **vector** clause, but does not contain nor does it call another
1946 procedure that contains a loop with either a **gang** or **worker** clause. A loop in this procedure with
1947 an **auto** clause may be selected by the compiler to execute in **vector** mode, but not **worker**
1948 mode. A call to this procedure must appear in code that is executed in *vector-single* mode, though
1949 it may be in *gang-redundant* or *gang-partitioned* mode, and in *worker-single* or *worker-partitioned*
1950 mode. For instance, a procedure with a **routine vector** directive may be called from within
1951 a loop that has the **gang** clause or the **worker** clause, but not from within a loop that has the
1952 **vector** clause. Only one of the **gang**, **worker**, **vector**, and **seq** clauses may be specified for
1953 each device type.

1954 seq clause

1955 The **seq** clause specifies that the procedure does not contain nor does it call another procedure that
1956 contains a loop with a **gang**, **worker**, or **vector** clause. A loop in this procedure with an **auto**
1957 clause will be executed in **seq** mode. A call to this procedure may appear in any mode. Only one
1958 of the **gang**, **worker**, **vector** and **seq** clauses may be specified for each device type.

1959 bind clause

1960 The **bind** clause specifies the name to use when calling the procedure on the device. If the name is
1961 specified as an identifier, it is called as if that name were specified in the language being compiled.
1962 If the name is specified as a string, the string is used for the procedure name unmodified. A **bind**

1963 clause on a procedure definition behaves as if it had appeared on a declaration by changing the name
1964 used to call the function on the device; however, the procedure is not compiled for the device with
1965 either the original name or the name in the **bind** clause.

1966 If there is both a Fortran **bind** and an acc **bind** clause for a procedure definition then a call on the
1967 host will call the Fortran bound name and a call on the device will call the name in the **bind** clause.

1968 **device_type** clause

1969 The **device_type** clause is described in Section 2.4 Device-Specific Clauses.

1970 **nohost** clause

1971 The **nohost** tells the compiler not to compile a version of this procedure for the host. All calls
1972 to this procedure must appear within compute regions. If this procedure is called from other pro-
1973 cedures, those other procedures must also have a matching **routine** directive with the **nohost**
1974 clause.

1975 **Restrictions**

- 1976 • Only the **gang**, **worker**, **vector**, **seq** and **bind** clauses may follow a **device_type**
1977 clause.
- 1978 • At least one of the (**gang**, **worker**, **vector**, or **seq**) clauses must be specified. If the
1979 **device_type** clause appears on the **routine** directive, a default level of parallelism
1980 clause must appear before the **device_type** clause, or a level of parallelism clause must
1981 be specified following each **device_type** clause on the directive.
- 1982 • In C and C++, function static variables are not supported in functions to which a **routine**
1983 directive applies.
- 1984 • In Fortran, variables with the *save* attribute, either explicitly or implicitly, are not supported
1985 in subprograms to which a **routine** directive applies.
- 1986 • A **bind** clause may not bind to a routine name that has a visible **bind** clause.
- 1987 • If a function or subroutine has a **bind** clause on both the declaration and the definition then
1988 they both must bind to the same name.

1989 **2.15.2. Global Data Access**

1990 C or C++ global, file static, or *extern* variables or array, and Fortran *module* or *common block* vari-
1991 ables or arrays, that are used in accelerator routines must appear in a declare directive in a **create**,
1992 **copyin**, **device_resident** or **link** clause. If the data appears in a **device_resident**
1993 clause, the **routine** directive for the procedure must include the **nohost** clause. If the data ap-
1994 pears in a **link** clause, that data must have an active accelerator data lifetime by virtue of appearing
1995 in a data clause for a **data** construct, compute construct, or **enter data** directive.

2.16. Asynchronous Behavior

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

2.16.1. **async** clause

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

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

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

The *async-value* of any operation is the value of the *async-argument*, if present, or the value of *acc-default-async-var* if it is **acc_async_noval** or if the **async** clause had no value, or **acc_async_sync** if no **async** clause appeared. If the current device supports asynchronous operation with one or more device activity queues, the *async-value* is used to select the queue on the current device onto which to enqueue an operation. The properties of the current device and the implementation will determine how many actual activity queues are supported, and how the *async-value* is mapped onto the actual activity queues. Two asynchronous operations with the same current device and the same *async-value* will be enqueued onto the same activity queue, and therefore will be executed on the device in the order they are encountered by the local thread. Two asynchronous operations with different *async-values* may be enqueued onto different activity queues, and therefore may be executed on the device in either order relative to each other. If there are two or more host threads executing and sharing the same accelerator device, two asynchronous operations with the same *async-value* will be enqueued on the same activity queue. If the threads are not synchronized with respect to each other, the operations may be enqueued in either order and therefore may execute on the device in either order. Asynchronous operations enqueued to different devices may execute in any order, regardless of the *async-value* used for each.

2035 2.16.2. wait clause

2036 The **wait** clause may appear on a **parallel**, **kernels**, or **serial** construct, or an **enter**
 2037 **data**, **exit data**, or **update** directive. In all cases, the **wait** clause is optional. When there
 2038 is no **wait** clause, the associated compute or update operations may be enqueued or launched or
 2039 executed immediately on the device. If there is an argument to the **wait** clause, it must be a list
 2040 of one or more *async-arguments*. The compute, data or update operation may not be launched or
 2041 executed until all operations enqueued up to this point by this thread on the associated asynchronous
 2042 device activity queues have completed. One legal implementation is for the local thread to wait for
 2043 all the associated asynchronous device activity queues. Another legal implementation is for the
 2044 local thread to enqueue the compute or update operation in such a way that the operation will
 2045 not start until the operations enqueued on the associated asynchronous device activity queues have
 2046 completed.

2047 2.16.3. Wait Directive

2048 **Summary** The **wait** directive causes the local thread to wait for completion of asynchronous
 2049 operations on the current device, such as an accelerator **parallel**, **kernels**, or **serial** region or an
 2050 **update** directive, or causes one device activity queue to synchronize with one or more other ac-
 2051 tivity queues on the current device.

2052 **Syntax** In C and C++, the syntax of the **wait** directive is:

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

2053 In Fortran the syntax of the **wait** directive is:

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

2054 where *clause* is:

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

2055 The wait argument, if present, must be one or more *async-arguments*.

2056 If there is no wait argument and no **async** clause, the local thread will wait until all operations
 2057 enqueued by this thread on any activity queue on the current device have completed.

2058 If there are one or more *int-expr* expressions and no **async** clause, the local thread will wait until all
 2059 operations enqueued by this thread on each of the associated device activity queues have completed.

2060 If there are two or more threads executing and sharing the same accelerator device, a **wait** directive
 2061 with no **async** clause will cause the local thread to wait until all of the appropriate asynchronous
 2062 operations previously enqueued by that thread have completed. To guarantee that operations have
 2063 been enqueued by other threads requires additional synchronization between those threads. There
 2064 is no guarantee that all the similar asynchronous operations initiated by other threads will have
 2065 completed.

2066 If there is an **async** clause, no new operation may be launched or executed on the **async** activ-
 2067 ity queue on the current device until all operations enqueued up to this point by this thread on the
 2068 asynchronous activity queues associated with the wait argument have completed. One legal imple-
 2069 mentation is for the local thread to wait for all the associated asynchronous device activity queues.
 2070 Another legal implementation is for the thread to enqueue a synchronization operation in such a
 2071 way that no new operation will start until the operations enqueued on the associated asynchronous
 2072 device activity queues have completed.

2073 A **wait** directive is functionally equivalent to a call to one of the **acc_wait**, **acc_wait_async**,
 2074 **acc_wait_all** or **acc_wait_all_async** runtime API routines, as described in Sections 3.2.11,
 2075 3.2.12, 3.2.13 and 3.2.14.

2076 2.17. Fortran Optional Arguments

2077 This section refers to the Fortran intrinsic function **PRESENT**. A call to the Fortran intrinsic function
 2078 **PRESENT (arg)** returns **.true.**, if **arg** is an optional dummy argument and an actual argument
 2079 for **arg** was present in the argument list of the call site. This should not be confused with the
 2080 OpenACC **present** data clause.

2081 The appearance of a Fortran optional argument **arg** as a *var* in any of the following clauses has no
 2082 effect at runtime if **PRESENT (arg)** is **.false.**:

- 2083 • in data clauses on compute and **data** constructs;
- 2084 • in data clauses on **enter data** and **exit data** directives;
- 2085 • in data and **device_resident** clauses on **declare** directives;
- 2086 • in **use_device** clauses on **host_data** directives;
- 2087 • in **self**, **host**, and **device** clauses on **update** directives.

2088 The appearance of a Fortran optional argument **arg** in the following situations may result in unde-
 2089 fined behavior if **PRESENT (arg)** is **.false.** when the associated construct is executed:

- 2090 • as a *var* in **private**, **firstprivate**, and **reduction** clauses;
- 2091 • as a *var* in **cache** directives;
- 2092 • as part of an expression in any clause or directive.

2093 A call to the Fortran intrinsic function **PRESENT** behaves the same way in a compute construct or
 2094 an accelerator routine as on the host. The function call **PRESENT (arg)** must return the same value
 2095 in a compute construct as **PRESENT (arg)** would outside of the compute construct. If a Fortran
 2096 optional argument **arg** appears as an actual argument in a procedure call in a compute construct
 2097 or an accelerator routine, and the associated dummy argument **subarg** also has the **optional**
 2098 attribute, then **PRESENT (subarg)** returns the same value as **PRESENT (subarg)** would when
 2099 executed on the host.

3. Runtime Library

2101 This chapter describes the OpenACC runtime library routines that are available for use by program-
2102 mers. Use of these routines may limit portability to systems that do not support the OpenACC API.
2103 Conditional compilation using the `_OPENACC` preprocessor variable may preserve portability.

2104 This chapter has two sections:

- 2105 • Runtime library definitions
- 2106 • Runtime library routines

2107 There are four categories of runtime routines:

- 2108 • Device management routines, to get the number of devices, set the current device, and so on.
- 2109 • Asynchronous queue management, to synchronize until all activities on an async queue are
2110 complete, for instance.
- 2111 • Device test routine, to test whether this statement is executing on the device or not.
- 2112 • Data and memory management, to manage memory allocation or copy data between memo-
2113 ries.

3.1. Runtime Library Definitions

2115 In C and C++, prototypes for the runtime library routines described in this chapter are provided in
2116 a header file named `openacc.h`. All the library routines are *extern* functions with “C” linkage.
2117 This file defines:

- 2118 • The prototypes of all routines in the chapter.
- 2119 • Any datatypes used in those prototypes, including an enumeration type to describe types of
2120 accelerators.
- 2121 • The values of `acc_async_noval`, `acc_async_sync`, and `acc_async_default`.

2122 In Fortran, interface declarations are provided in a Fortran module named `openacc`. The `openacc`
2123 module defines:

- 2124 • The integer parameter `openacc_version` with a value `yyyymm` where `yyyy` and `mm` are the
2125 year and month designations of the version of the Accelerator programming model supported.
2126 This value matches the value of the preprocessor variable `_OPENACC`.
- 2127 • **Interfaces for all routines in the chapter.**
- 2128 • Integer parameters to define integer kinds for arguments to **and return values for** those rou-
2129 tines.

- 2130 • Integer parameters to describe types of accelerators.
- 2131 • **Integer parameters to define the** values of `acc_async_noval`, `acc_async_sync`, and
- 2132 **`acc_async_default`**.

2133 Many of the routines accept or return a value corresponding to the type of accelerator device. In
 2134 C and C++, the datatype used for device type values is `acc_device_t`; in Fortran, the cor-
 2135 responding datatype is `integer(kind=acc_device_kind)`. The possible values for de-
 2136 vice type are implementation specific, and are defined in the C or C++ include file `openacc.h`
 2137 and the Fortran module `openacc`. Four values are always supported: `acc_device_none`,
 2138 `acc_device_default`, `acc_device_host` and `acc_device_not_host`. For other val-
 2139 ues, look at the appropriate files included with the implementation, or read the documentation for
 2140 the implementation. The value `acc_device_default` will never be returned by any function;
 2141 its use as an argument will tell the runtime library to use the default device type for that implemen-
 2142 tation.

2143 3.2. Runtime Library Routines

2144 In this section, for the C and C++ prototypes, pointers are typed `h_void*` or `d_void*` to desig-
 2145 nate a host address or device address, when these calls are executed on the host, as if the following
 2146 definitions were included:

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

2147 Except for `acc_on_device`, these routines are only available on the host.

2148 3.2.1. `acc_get_num_devices`

2149 **Summary** The `acc_get_num_devices` routine returns the number of accelerator devices of
 2150 the given type attached to the host.

2151 **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
```

2152 **Description** The `acc_get_num_devices` routine returns the number of accelerator devices
 2153 of the given type attached to the host. The argument tells what kind of device to count.

2154 **Restrictions**

- 2155 • This routine may not be called within a compute region.

2156 **3.2.2. acc_set_device_type**

2157 **Summary** The `acc_set_device_type` routine tells the runtime which type of device to use
2158 when executing a compute region and sets the value of `acc-device-type-var`. This is useful when the
2159 implementation allows the program to be compiled to use more than one type of accelerator.

2160 **Format**

C or C++:

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

Fortran:

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

2161 **Description** The `acc_set_device_type` routine tells the runtime which type of device to
2162 use among those available and sets the value of `acc-device-type-var` for the current thread. A call to
2163 `acc_set_device_type` is functionally equivalent to a `set device_type` directive with the
2164 matching device type argument, as described in Section 2.14.3.

2165 **Restrictions**

- 2166 • This routine may not be called within a compute region.
- 2167 • If the device type specified is not available, the behavior is implementation-defined; in partic-
2168 ular, the program may abort.
- 2169 • If some accelerator regions are compiled to only use one device type, calling this routine with
2170 a different device type may produce undefined behavior.

2171 **3.2.3. acc_get_device_type**

2172 **Summary** The `acc_get_device_type` routine returns the value of `acc-device-type-var`, which
2173 is the device type of the current device. This is useful when the implementation allows the program
2174 to be compiled to use more than one type of accelerator.

2175 **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
```

2176 **Description** The `acc_get_device_type` routine returns the value of *acc-device-type-var*
 2177 for the current thread to tell the program what type of device will be used to run the next compute
 2178 region, if one has been selected. The device type may have been selected by the program with an
 2179 `acc_set_device_type` call, with an environment variable, or by the default behavior of the
 2180 program.

2181 Restrictions

- 2182 • This routine may not be called within a compute region.
- 2183 • If the device type has not yet been selected, the value `acc_device_none` may be returned.

2184 3.2.4. `acc_set_device_num`

2185 **Summary** The `acc_set_device_num` routine tells the runtime which device to use and sets
 2186 the value of *acc-device-num-var*.

2187 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
```

2188 **Description** The `acc_set_device_num` routine tells the runtime which device to use among
 2189 those attached of the given type for compute or data regions in the current thread and sets the value
 2190 of *acc-device-num-var*. If the value of `devicenum` is negative, the runtime will revert to its default
 2191 behavior, which is implementation-defined. If the value of the second argument is zero, the selected
 2192 device number will be used for all attached accelerator types. A call to `acc_set_device_num`
 2193 is functionally equivalent to a `set device_num` directive with the matching device number argu-
 2194 ment, as described in Section 2.14.3.

2195 Restrictions

- 2196 • This routine may not be called within a compute or data region.
- 2197 • If the value of `devicenum` is greater than or equal to the value returned by `acc_get_num_devices`
 2198 for that device type, the behavior is implementation-defined.
- 2199 • Calling `acc_set_device_num` implies a call to `acc_set_device_type` with that
 2200 device type argument.

2201 **3.2.5. acc_get_device_num**

2202 **Summary** The `acc_get_device_num` routine returns the value of *acc-device-num-var* for
 2203 the current thread.

2204 **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
```

2205 **Description** The `acc_get_device_num` routine returns **the** value of *acc-device-num-var* for
 2206 the current thread.

2207 **Restrictions**

- 2208 • This routine may not be called within a compute region.

2209 **3.2.6. acc_get_property**

2210 **Summary** The `acc_get_property` and `acc_get_property_string` routines return
 2211 **the** value of a *device-property* for the specified device.

2212 **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
```

2213 **Description** The `acc_get_property` and `acc_get_property_string` routines returns
 2214 the value of the specified *property*. `devicenum` and `devicetype` specify the device being
 2215 queried. If `devicetype` has the value `acc_device_current`, then `devicenum` is ignored
 2216 and the value of the property for the current device is returned. `property` is an enumeration
 2217 constant, defined in `openacc.h`, for C or C++, or an integer parameter, defined in the `openacc`
 2218 module, for Fortran. Integer-valued properties are returned by `acc_get_property`, and string-
 2219 valued properties are returned by `acc_get_property_string`. In Fortran, `acc_get_property_string`
 2220 returns the result into the `character` variable passed as the last argument.

2221 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

2223 An implementation may support additional properties for some devices.

2224 Restrictions

- 2225 • These routines may not be called within an compute region.
- 2226 • If the value of `property` is not one of the known values for that query routine, or that
 2227 property has no value for the specified device, `acc_get_property` will return 0 and
 2228 `acc_get_property_string` will return NULL (in C or C++) or an blank string (in
 2229 Fortran).

2230 3.2.7. `acc_init`

2231 **Summary** The `acc_init` routine tells the runtime to initialize the runtime for that device type.
 2232 This can be used to isolate any initialization cost from the computational cost, when collecting
 2233 performance statistics.

2234 Format

C or C++:

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

Fortran:

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

2235 **Description** The `acc_init` routine also implicitly calls `acc_set_device_type`. A call to
 2236 `acc_init` is functionally equivalent to a `init` directive with the matching device type argument,
 2237 as described in Section 2.14.1.

2238 **Restrictions**

- 2239 • This routine may not be called within a compute region.
- 2240 • If the device type specified is not available, the behavior is implementation-defined; in partic-
2241 ular, the program may abort.
- 2242 • If the routine is called more than once without an intervening **acc_shutdown** call, with a
2243 different value for the device type argument, the behavior is implementation-defined.
- 2244 • If some accelerator regions are compiled to only use one device type, calling this routine with
2245 a different device type may produce undefined behavior.

2246 **3.2.8. acc_shutdown**

2247 **Summary** The **acc_shutdown** routine tells the runtime to shut down the connection to the
2248 given accelerator device, and free up any runtime resources. A call to **acc_shutdown** is func-
2249 tionally equivalent to a **shutdown** directive with the matching device type argument, as described
2250 in Section 2.14.2.

2251 **Format**

C or C++:

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

Fortran:

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

2252 **Description** The **acc_shutdown** routine disconnects the program from the any accelerator
2253 device of the specified device type. Any data that is present on any such device is immediately
2254 deallocated.

2255 **Restrictions**

- 2256 • This routine may not be called during execution of a compute region.
- 2257 • If the program attempts to execute a compute region or access any device data on such a
2258 device, the behavior is undefined.

2259 **3.2.9. acc_async_test**

2260 **Summary** The **acc_async_test** routine tests for completion of all associated asynchronous
2261 operations on the current device.

2262 **Format**

C or C++:

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

Fortran:

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

2263 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
2264 If that value did not appear in any **async** clauses, or if it did appear in one or more **async** clauses
2265 and all such asynchronous operations have completed on the current device, the **acc_async_test**
2266 routine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some such asyn-
2267 chronous operations have not completed, the **acc_async_test** routine will return with a zero
2268 value in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the
2269 **acc_async_test** routine will return with a nonzero value or **.true.** only if all matching asyn-
2270 chronous operations initiated by this thread have completed; there is no guarantee that all matching
2271 asynchronous operations initiated by other threads have completed.

2272 3.2.10. acc_async_test_all

2273 **Summary** The **acc_async_test_all** routine tests for completion of all asynchronous op-
2274 erations.

2275 **Format**

C or C++:

```
int acc_async_test_all( );
```

Fortran:

```
logical function acc_async_test_all( )
```

2276 **Description** If all outstanding asynchronous operations have completed, the **acc_async_test_all**
2277 routine will return with a nonzero value in C and C++, or **.true.** in Fortran. If some asynchronous
2278 operations have not completed, the **acc_async_test_all** routine will return with a zero value
2279 in C and C++, or **.false.** in Fortran. If two or more threads share the same accelerator, the
2280 **acc_async_test_all** routine will return with a nonzero value or **.true.** only if all outstand-
2281 ing asynchronous operations initiated by this thread have completed; there is no guarantee that all
2282 asynchronous operations initiated by other threads have completed.

2283 3.2.11. acc_wait

2284 **Summary** The **acc_wait** routine waits for completion of all associated asynchronous opera-
2285 tions on the current device.

2286 **Format**

C or C++:

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

Fortran:

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

2287 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
2288 If that value appeared in one or more **async** clauses, the **acc_wait** routine will not return until
2289 the latest such asynchronous operation has completed on the current device. If two or more threads
2290 share the same accelerator, the **acc_wait** routine will return only if all matching asynchronous
2291 operations initiated by this thread have completed; there is no guarantee that all matching asyn-
2292 chronous operations initiated by other threads have completed. For compatibility with version 1.0,
2293 this routine may also be spelled **acc_async_wait**. A call to **acc_wait** is functionally equiv-
2294 alent to a **wait** directive with a matching wait argument and no **async** clause, as described in
2295 Section 2.16.3.

2296 **3.2.12. acc_wait_async**

2297 **Summary** The **acc_wait_async** routine enqueues a wait operation on one *async* queue of
2298 the current device for the operations previously enqueued on another *async* queue.

2299 **Format**

C or C++:

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

Fortran:

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

2300 **Description** The arguments must be *async-arguments*, as defined in Section 2.16.1 *async* clause.
2301 The routine will enqueue a wait operation on the appropriate device queue associated with the
2302 second argument, which will wait for operations enqueued on the device queue associated with
2303 the first argument. See Section 2.16 Asynchronous Behavior for more information. A call to
2304 **acc_wait_async** is functionally equivalent to a **wait** directive with a matching wait argument
2305 and a matching **async** argument, as described in Section 2.16.3.

2306 **3.2.13. acc_wait_all**

2307 **Summary** The **acc_wait_all** routine waits for completion of all asynchronous operations.

2308 **Format**

C or C++:

```
void acc_wait_all( );
```

Fortran:

```
subroutine acc_wait_all( )
```

2309 **Description** The `acc_wait_all` routine will not return until the all asynchronous operations
2310 have completed. If two or more threads share the same accelerator, the `acc_wait_all` routine
2311 will return only if all asynchronous operations initiated by this thread have completed; there is no
2312 guarantee that all asynchronous operations initiated by other threads have completed. For com-
2313 patibility with version 1.0, this routine may also be spelled `acc_async_wait_all`. A call to
2314 `acc_wait_all` is functionally equivalent to a `wait` directive with no wait argument list and no
2315 `async` argument, as described in Section 2.16.3.

2316 **3.2.14. acc_wait_all_async**

2317 **Summary** The `acc_wait_all_async` routine enqueues wait operations on one async queue
2318 for the operations previously enqueued on all other async queues.

2319 **Format**

C or C++:

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

Fortran:

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

2320 **Description** The argument must be an *async-argument* as defined in Section 2.16.1 *async* clause.
2321 The routine will enqueue a wait operation on the appropriate device queue for each other device
2322 queue. See Section 2.16 Asynchronous Behavior for more information. A call to `acc_wait_all_async`
2323 is functionally equivalent to a `wait` directive with no wait argument list and a matching `async`
2324 argument, as described in Section 2.16.3.

2325 **3.2.15. acc_get_default_async**

2326 **Summary** The `acc_get_default_async` routine returns the value of *acc-default-async-*
2327 *var* for the current thread.

2328 **Format**

C or C++:

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

Fortran:

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

2329 **Description** The `acc_get_default_async` routine returns the value of *acc-default-async-*
2330 *var* for the current thread, which is the asynchronous queue used when an **async** clause appears
2331 without an *async-argument* or with the value `acc_async_noval`.

2332 3.2.16. `acc_set_default_async`

2333 **Summary** The `acc_set_default_async` routine tells the runtime which asynchronous queue
2334 to use when no other queue is specified.

2335 **Format**

C or C++:

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

Fortran:

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

2336 **Description** The `acc_set_default_async` routine tells the runtime to place any directives
2337 with an **async** clause that does not have an *async-argument* or with the special `acc_async_noval`
2338 value into the specified asynchronous activity queue instead of the default asynchronous activity
2339 queue for that device by setting the value of *acc-default-async-var* for the current thread. The spe-
2340 cial argument `acc_async_default` will reset the default asynchronous activity queue to the
2341 initial value, which is implementation-defined. A call to `acc_set_default_async` is func-
2342 tionally equivalent to a `set default_async` directive with a matching argument in *int-expr*, as
2343 described in Section 2.14.3.

2344 3.2.17. `acc_on_device`

2345 **Summary** The `acc_on_device` routine tells the program whether it is executing on a partic-
2346 ular device.

2347 **Format**

C or C++:

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

Fortran:

```
logical function acc_on_device( devicetype )
```

```
integer(acc_device_kind) :: devicetype
```

2348 **Description** The `acc_on_device` routine may be used to execute different paths depending
 2349 on whether the code is running on the host or on some accelerator. If the `acc_on_device` routine
 2350 has a compile-time constant argument, it evaluates at compile time to a constant. The argument must
 2351 be one of the defined accelerator types. If the argument is `acc_device_host`, then outside of an
 2352 accelerator compute region or accelerator routine, or in an accelerator compute region or accelerator
 2353 routine that is executed on the host processor, this routine will evaluate to nonzero for C or C++, and
 2354 `.true.` for Fortran; otherwise, it will evaluate to zero for C or C++, and `.false.` for Fortran. If
 2355 the argument is `acc_device_not_host`, the result is the negation of the result with argument
 2356 `acc_device_host`. If the argument is any accelerator device type, then in a compute region or
 2357 routine that is executed on an accelerator of that device type, this routine will evaluate to nonzero for
 2358 C or C++, and `.true.` for Fortran; otherwise, it will evaluate to zero for C or C++, and `.false.`
 2359 for Fortran. The result with argument `acc_device_default` is undefined.

2360 3.2.18. `acc_malloc`

2361 **Summary** The `acc_malloc` routine allocates memory on the accelerator device.

2362 **Format**

C or C++:

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

2363 **Description** The `acc_malloc` routine may be used to allocate memory on the accelerator de-
 2364 vice. Pointers assigned from this function may be used in `deviceptr` clauses to tell the compiler
 2365 that the pointer target is resident on the device.

2366 3.2.19. `acc_free`

2367 **Summary** The `acc_free` routine frees memory on the accelerator device.

2368 **Format**

C or C++:

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

2369 **Description** The `acc_free` routine will free previously allocated memory on the accelerator
 2370 device; the argument should be a pointer value that was returned by a call to `acc_malloc`.

2371 **3.2.20. acc_copyin**

2372 **Summary** The `acc_copyin` routines test to see if the data is already present on the current
 2373 device; if not, they allocate memory on the accelerator device to correspond to the specified host
 2374 memory, and copy the data to that device memory, on a non-shared memory device.

2375 **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
```

2376 **Description** The `acc_copyin` routines are equivalent to the `enter data` directive with a
 2377 `copyin` clause, as described in Section 2.7.6. In C, the arguments are a pointer to the data and
 2378 length in bytes; the synchronous function returns a pointer to the allocated device space, as with
 2379 `acc_malloc`. In Fortran, two forms are supported. In the first, the argument is a contiguous array
 2380 section of intrinsic type. In the second, the first argument is a variable or array element and the
 2381 second is the length in bytes.

2382 The behavior of the `acc_copyin` routines is:

- 2383 • If the current device is a shared memory device, no action is taken. The C `acc_copyin`
 2384 returns the incoming pointer.
- 2385 • If the data is present, a *present increment* action with the dynamic reference counter is per-
 2386 formed. The C `acc_copyin` returns a pointer to the existing device memory.
- 2387 • Otherwise, a *copyin* action with the appropriate reference counter is performed. The C
 2388 `acc_copyin` returns the device address of the newly allocated memory.

2389 This data may be accessed using the `present` data clause. Pointers assigned from the C `acc_copyin`
 2390 function may be used in `deviceptr` clauses to tell the compiler that the pointer target is resident
 2391 on the device.

2392 The `_async` versions of this function will perform any data transfers asynchronously on the `async`
 2393 queue associated with the value passed in as the `async` argument. The function may return be-
 2394 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
 2395 synchronous versions will not return until the data has been completely transferred.

2396 For compatibility with OpenACC 2.0, `acc_present_or_copyin` and `acc_pcopyin` are al-
 2397 ternate names for `acc_copyin`.

2398 **3.2.21. acc_create**

2399 **Summary** The `acc_create` routines test to see if the data is already present on the device; if
 2400 not, they allocate memory on the accelerator device to correspond to the specified host memory, on
 2401 a non-shared memory device.

2402 **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
```

2403 **Description** The `acc_create` routines are equivalent to the `enter data` directive with a
 2404 `create` clause, as described in Section 2.7.8. In C, the arguments are a pointer to the data and
 2405 length in bytes; the synchronous function returns a pointer to the allocated device space, as with
 2406 `acc_malloc`. In Fortran, two forms are supported. In the first, the argument is a contiguous array
 2407 section of intrinsic type. In the second, the first argument is a variable or array element and the
 2408 second is the length in bytes.

2409 The behavior of the `acc_create` routines is:

- 2410 • If the current device is a shared memory device, no action is taken. The C `acc_create`
 2411 returns the incoming pointer.
- 2412 • If the data is present, a *present increment* action with the dynamic reference counter is per-
 2413 formed. The C `acc_create` returns a pointer to the existing device memory.
- 2414 • Otherwise, a *create* action with the appropriate reference counter is performed. The C `acc_create`
 2415 returns the device address of the newly allocated memory.

2416 This data may be accessed using the `present` data clause. Pointers assigned from the C `acc_copyin`
 2417 function may be used in `deviceptr` clauses to tell the compiler that the pointer target is resident
 2418 on the device.

2419 The `_async` versions of these function may perform the data allocation asynchronously on the
 2420 async queue associated with the value passed in as the `async` argument. The synchronous versions
 2421 will not return until the data has been allocated.

2422 For compatibility with OpenACC 2.0, `acc_present_or_create` and `acc_pcreate` are al-
 2423 ternate names for `acc_create`.

2424 **3.2.22. acc_copyout**

2425 **Summary** The `acc_copyout` routines copy data from device memory to the corresponding
 2426 local memory, then deallocate that memory from the accelerator device, on a non-shared memory
 2427 device.

2428 **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
```

2429 **Description** The `acc_copyout` routines are equivalent to the `exit data` directive with a
 2430 `copyout` clause, and the `acc_copyout_finalize` routines are equivalent to the `exit data`
 2431 `directive with both copyout and finalize clauses, as described in Section 2.7.7. In C, the`
 2432 arguments are a pointer to the data and length in bytes. In Fortran, two forms are supported. In the
 2433 first, the argument is a contiguous array section of intrinsic type. In the second, the first argument
 2434 is a variable or array element and the second is the length in bytes.

2435 The behavior of the `acc_copyout` routines is:

- 2436 • If the current device is a shared memory device, no action is taken.
- 2437 • If the data is not present, a runtime error is issued.
- 2438 • Otherwise, a *present decrement* action with the dynamic reference counter is performed (`acc_copyout`),
 2439 or the dynamic reference counter is set to zero (`acc_copyout_finalize`). If both ref-
 2440 erence counters are then zero, a *copyout* action is performed.

2441 The `_async` versions of these functions will perform any associated data transfers asynchronously
 2442 on the `async` queue associated with the value passed in as the `async` argument. The function may
 2443 return before the data has been transferred or deallocated; see Section 2.16 Asynchronous Behavior
 2444 for more details. The synchronous versions will not return until the data has been completely trans-
 2445 ferred. Even if the data has not been transferred or deallocated before the function returns, the data
 2446 will be treated as not present on the device.

2447 **3.2.23. acc_delete**

2448 **Summary** The `acc_delete` routines deallocate the memory from the accelerator device cor-
 2449 responding to the specified local memory, on a non-shared memory device.

2450 **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
```

2451 **Description** The `acc_delete` routines are equivalent to the `exit data` directive with a
 2452 `delete` clause, and the `acc_delete_finalize` routines are equivalent to the `exit data`
 2453 directive with both `delete` clause and `finalize` clauses, as described in Section 2.7.10. The
 2454 arguments are as for `acc_copyout`.

2455 The behavior of the `acc_delete` routines is:

- 2456 • If the current device is a shared memory device, no action is taken.
- 2457 • If the data is not present, a runtime error is issued.
- 2458 • Otherwise, a *present decrement* action with the dynamic reference counter is performed (`acc_delete`),
 2459 or the dynamic reference counter is set to zero (`acc_delete_finalize`). If both refer-
 2460 ence counters are then zero, a *delete* action is performed.

2461 The `_async` versions of these function may perform the data deallocation asynchronously on the
 2462 `async` queue associated with the value passed in as the `async` argument. The synchronous versions
 2463 will not return until the data has been deallocated. Even if the data has not been deallocated before
 2464 the function returns, the data will be treated as not present on the device.

2465 **3.2.24. acc_update_device**

2466 **Summary** The `acc_update_device` routine updates the device copy of data from the corre-
 2467 sponding local memory on a non-shared memory device.

2468 **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
```

2469 **Description** The `acc_update_device` routine is equivalent to the `update` directive with a
 2470 `device` clause, as described in Section 2.14.4. In C, the arguments are a pointer to the data and
 2471 length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array
 2472 section of intrinsic type. In the second, the first argument is a variable or array element and the
 2473 second is the length in bytes. On a non-shared memory device, the data in the local memory is
 2474 copied to the corresponding device memory. It is a runtime error to call this routine if the data is
 2475 not present on the device.

2476 The `_async` versions of this function will perform the data transfers asynchronously on the `async`
 2477 queue associated with the value passed in as the `async` argument. The function may return be-
 2478 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
 2479 synchronous versions will not return until the data has been completely transferred.

2480 **3.2.25. acc_update_self**

2481 **Summary** The `acc_update_self` routine updates the device copy of data to the correspond-
 2482 ing local memory on a non-shared memory device.

2483 **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

```

2484 **Description** The `acc_update_self` routine is equivalent to the `update` directive with a
 2485 `self` clause, as described in Section 2.14.4. In C, the arguments are a pointer to the data and
 2486 length in bytes. In Fortran, two forms are supported. In the first, the argument is a contiguous array
 2487 section of intrinsic type. In the second, the first argument is a variable or array element and the
 2488 second is the length in bytes. On a non-shared memory device, the data in the local memory is
 2489 copied to the corresponding device memory. There must be a device copy of the data on the device
 2490 when calling this routine, otherwise no action is taken by the routine.

2491 The `_async` versions of this function will perform the data transfers asynchronously on the `async`
 2492 queue associated with the value passed in as the `async` argument. The function may return be-
 2493 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
 2494 synchronous versions will not return until the data has been completely transferred.

2495 3.2.26. `acc_map_data`

2496 **Summary** The `acc_map_data` routine maps previously allocated device data to the specified
 2497 host data.

2498 **Format**

C or C++:

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

2499 **Description** The `acc_map_data` routine is similar to an `enter data` directive with a `create`
 2500 clause, except instead of allocating new device memory to start a data lifetime, the device address
 2501 to use for the data lifetime is specified as an argument. The first argument is a host address, fol-
 2502 lowed by the corresponding device address and the data length in bytes. After this call, when the
 2503 host data appears in a data clause, the specified device memory will be used. It is an error to
 2504 call `acc_map_data` for host data that is already present on the device. It is undefined to call
 2505 `acc_map_data` with a device address that is already mapped to host data. The device address
 2506 may be the result of a call to `acc_malloc`, or may come from some other device-specific API
 2507 routine.

2508 3.2.27. `acc_unmap_data`

2509 **Summary** The `acc_unmap_data` routine unmaps device data from the specified host data.

2510 **Format**

C or C++:

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

2511 **Description** The `acc_unmap_data` routine is similar to an `exit data` directive with a
2512 `delete` clause, except the device memory is not deallocated. The argument is pointer to the host
2513 data. A call to this routine ends the data lifetime for the specified host data. The device memory is
2514 not deallocated. It is undefined behavior to call `acc_unmap_data` with a host address unless that
2515 host address was mapped to device memory using `acc_map_data`.

2516 **3.2.28. acc_deviceptr**

2517 **Summary** The `acc_deviceptr` routine returns the device pointer associated with a specific
2518 host address.

2519 **Format**

C or C++:

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

2520 **Description** The `acc_deviceptr` routine returns the device pointer associated with a host
2521 address. The argument is the address of a host variable or array that has an active lifetime on the
2522 current device. If the data is not present on the device, the routine returns a NULL value.

2523 **3.2.29. acc_hostptr**

2524 **Summary** The `acc_hostptr` routine returns the host pointer associated with a specific device
2525 address.

2526 **Format**

C or C++:

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

2527 **Description** The `acc_hostptr` routine returns the host pointer associated with a device ad-
2528 dress. The argument is the address of a device variable or array, such as that returned from `acc_deviceptr`,
2529 `acc_create` or `acc_copyin`. If the device address is NULL, or does not correspond to any host
2530 address, the routine returns a NULL value.

2531 **3.2.30. acc_is_present**

2532 **Summary** The `acc_is_present` routine tests whether a host variable or array region is present
2533 on the device.

2534 **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
```

2535 **Description** The `acc_is_present` routine tests whether the specified host data is present
2536 on the device. In C, the arguments are a pointer to the data and length in bytes; the function
2537 returns nonzero if the specified data is fully present, and zero otherwise. In Fortran, two forms are
2538 supported. In the first, the argument is a contiguous array section of intrinsic type. In the second,
2539 the first argument is a variable or array element and the second is the length in bytes. The function
2540 returns `.true.` if the specified data is fully present, and `.false.` otherwise. If the byte length is
2541 zero, the function returns nonzero in C or `.true.` in Fortran if the given address is present at all
2542 on the device.

2543 **3.2.31. acc_memcpy_to_device**

2544 **Summary** The `acc_memcpy_to_device` routine copies data from local memory to device
2545 memory.

2546 **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 );
```

2547 **Description** The `acc_memcpy_to_device` routine copies `bytes` of data from the local
2548 address in `src` to the device address in `dest`. The destination address must be a device address,
2549 such as would be returned from `acc_malloc` or `acc_deviceptr`.

2550 The `_async` version of this function will perform the data transfers asynchronously on the `async`
2551 queue associated with the value passed in as the `async` argument. The function may return be-
2552 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
2553 synchronous versions will not return until the data has been completely transferred.

2554 **3.2.32. acc_memcpy_from_device**

2555 **Summary** The `acc_memcpy_from_device` routine copies data from device memory to lo-
2556 cal memory.

2557 **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 );
```

2558 **Description** The `acc_memcpy_from_device` routine copies `bytes` data from the device
2559 address in `src` to the local address in `dest`. The source address must be a device address, such as
2560 would be returned from `acc_malloc` or `acc_deviceptr`.

2561 The `_async` version of this function will perform the data transfers asynchronously on the `async`
2562 queue associated with the value passed in as the `async` argument. The function may return be-
2563 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
2564 synchronous versions will not return until the data has been completely transferred.

2565 **3.2.33. acc_memcpy_device**

2566 **Summary** The `acc_memcpy_device` routine copies data from one memory location to an-
2567 other memory location on the current device.

2568 **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 );
```

2569 **Description** The `acc_memcpy_device` routine copies `bytes` data from the device address
2570 in `src` to the device address in `dest`. Both addresses must be addresses in the current device
2571 memory, such as would be returned from `acc_malloc` or `acc_deviceptr`. If `dest` and `src`
2572 overlap, the behavior is undefined.

2573 The `_async` version of this function will perform the data transfers asynchronously on the `async`
2574 queue associated with the value passed in as the `async` argument. The function may return be-
2575 fore the data has been transferred; see Section 2.16 Asynchronous Behavior for more details. The
2576 synchronous versions will not return until the data has been completely transferred.

2577 **3.2.34. acc_attach**

2578 **Summary** The `acc_attach` routine updates a pointer in device memory to point to the corre-
2579 sponding device copy of the host pointer target.

2580 **Format**

C or C++:

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

2581 **Description** The `acc_attach` routines are passed the address of a host pointer. If the current
 2582 device is a shared memory device, or if the pointer `*ptr` is not present on the current device, or the
 2583 address to which the `*ptr` points is not present on the current device, no action is taken. Otherwise,
 2584 these routines perform the *attach* action (Section 2.7.1).

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

2590 3.2.35. `acc_detach`

2591 **Summary** The `acc_detach` routine updates a pointer in device memory to point to the host
 2592 pointer target.

2593 **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 );
```

2594 **Description** The `acc_detach` routines are passed the address of a host pointer. If the current
 2595 device is a shared memory device, or if the pointer `*ptr` is not present on the current device, or if
 2596 the *attachment counter* for the pointer `*ptr` is zero, no action is taken. Otherwise, these routines
 2597 perform the *detach* action (Section 2.7.1).

2598 The `acc_detach_finalize` routines are equivalent to an `exit data` directive with `detach`
 2599 and `finalize` clauses, as described in Section 2.7.12 detach clause. If the current device is a
 2600 shared memory device, or if the pointer `*ptr` is not present on the current device, or if the *attach-*
 2601 *ment counter* for the pointer `*ptr` is zero, no action is taken. Otherwise, these routines perform the
 2602 *immediate detach* action (Section 2.7.1).

2603 These routines may issue a data transfer from local memory to device memory. The `_async`
 2604 versions of these functions will perform the data transfers asynchronously on the async queue asso-
 2605 ciated with the value passed in as the `async` argument. These functions may return before the data
 2606 has been transferred; see Section 2.16 Asynchronous Behavior for more details. The synchronous
 2607 versions will not return until the data has been completely transferred.

4. Environment Variables

2608

2609 This chapter describes the environment variables that modify the behavior of accelerator regions.
2610 The names of the environment variables must be upper case. The values assigned environment
2611 variables are **case-insensitive** and may have leading and trailing white space. If the values of the
2612 environment variables change after the program has started, even if the program itself modifies the
2613 values, the behavior is implementation-defined.

4.1. ACC_DEVICE_TYPE

2614

2615 The **ACC_DEVICE_TYPE** environment variable controls the default device type to use when exe-
2616 cuting accelerator parallel, kernels, **and serial** regions, if the program has been compiled to use more
2617 than one different type of device. The allowed values of this environment variable are implementa-
2618 tion-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

2619

2620 The **ACC_DEVICE_NUM** environment variable controls the default device number to use when
2621 executing accelerator regions. The value of this environment variable must be a nonnegative integer
2622 between zero and the number of devices of the desired type attached to the host. If the value is
2623 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

2624

2625 The **ACC_PROFLIB** environment variable specifies the profiling library. More details about the
2626 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

2627

2628 This chapter describes the OpenACC interface for tools that can be used for profile and trace data
2629 collection. Therefore it provides a set of OpenACC-specific event callbacks that are triggered dur-
2630 ing the application run. Currently, this interface does not support tools that employ asynchronous
2631 sampling. In this chapter, the term *runtime* refers to the OpenACC runtime library. The term *library*
2632 refers to the third party routines invoked at specified events by the OpenACC runtime.

2633 There are four steps for interfacing a *library* to the *runtime*. The first is to write the data collection
2634 library callback routines. Section 5.1 Events describes the supported runtime events and the order
2635 in which callbacks to the callback routines will occur. Section 5.2 Callbacks Signature describes
2636 the signature of the callback routines for all events.

2637 The second is to use registration routines to register the data collection callbacks for the appropriate
2638 events. The data collection and registration routines are then saved in a static or dynamic library
2639 or shared object. The third is to load the *library* at runtime. The *library* may be statically linked
2640 to the application or dynamically loaded by the application or by the *runtime*. This is described in
2641 Section 5.3 Loading the Library.

2642 The fourth step is to invoke the registration routine to register the desired callbacks with the events.
2643 This may be done explicitly by the application, if the library is statically linked with the application,
2644 implicitly by including a call to the registration routine in a `.init` section, or by including an
2645 initialization routine in the library if it is dynamically loaded by the *runtime*. This is described in
2646 Section 5.4 Registering Event Callbacks.

2647 Subsequently, the *library* may collect information when the callback routines are invoked by the
2648 *runtime* and process or store the acquired data.

5.1. Events

2649

2650 This section describes the events that are recognized by the runtime. Most events may have a start
2651 and end callback routine, that is, a routine that is called just before the runtime code to handle
2652 the event starts and another routine that is called just after the event is handled. The event names
2653 and routine prototypes are available in the header file `acc_prof.h`, which is delivered with the
2654 OpenACC implementation. Event names are prefixed with `acc_ev_`.

2655 The ordering of events must reflect the order in which the OpenACC runtime actually executes them,
2656 i.e. if a runtime moves the enqueueing of data transfers or kernel launches outside the originating
2657 clauses/constructs, it needs to issue the corresponding launch callbacks when they really occur. A
2658 callback for a start event must always precede the matching end callback. The behavior of a tool
2659 receiving a callback after the runtime shutdown callback is undefined.

2660 The events that the runtime supports can be registered with a callback and are defined in the enu-
2661 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;
```

2662 5.1.1. Runtime Initialization and Shutdown

2663 No callbacks can be registered for the runtime initialization. Instead the initialization of the tool is
2664 handled as described in Section 5.3 Loading the Library.

2665 The *runtime shutdown* event name is

```
acc_ev_runtime_shutdown
```

2666 The **acc_ev_runtime_shutdown** event is triggered before the OpenACC runtime shuts down,
2667 either because all devices have been shutdown by calls to the **acc_shutdown** API routine, or at
2668 the end of the program.

2669 5.1.2. Device Initialization and Shutdown

2670 The *device initialization* event names are

```
acc_ev_device_init_start
```


acc_ev_device_init_end

2671 These events are triggered when a device is being initialized by the OpenACC runtime. This may be
2672 when the program starts, or may be later during execution when the program reaches an **acc_init**
2673 call or an OpenACC construct. The **acc_ev_device_init_start** is triggered before device
2674 initialization starts and **acc_ev_device_init_end** after initialization is complete.

2675 The *device shutdown* event names are

acc_ev_device_shutdown_start**acc_ev_device_shutdown_end**

2676 These events are triggered when a device is shut down, most likely by a call to the OpenACC
2677 **acc_shutdown** API routine. The **acc_ev_device_shutdown_start** is triggered before
2678 the device shutdown process starts and **acc_ev_device_shutdown_end** after the device shut-
2679 down is complete.

5.1.3. Enter Data and Exit Data

2681 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**

2682 The **acc_ev_enter_data_start** and **acc_ev_enter_data_end** events are triggered at
2683 **enter data** directives, entry to data constructs, and entry to implicit data regions such as those
2684 generated by compute constructs. The **acc_ev_enter_data_start** event is triggered before
2685 any *data allocation*, *data update*, or *wait* events that are associated with that directive or region
2686 entry, and the **acc_ev_enter_data_end** is triggered after those events.

2687 The **acc_ev_exit_data_start** and **acc_ev_exit_data_end** events are triggered at **exit**
2688 **data** directives, exit from **data** constructs, and exit from implicit data regions. The **acc_ev_exit_data_start**
2689 event is triggered before any *data deallocation*, *data update*, or *wait* events associated with that di-
2690 rective or region exit, and the **acc_ev_exit_data_end** event is triggered after those events.

2691 When the construct that triggers an *enter data* or *exit data* event was generated implicitly by the
2692 compiler the **implicit** field in the event structure will be set to **1**. When the construct that
2693 triggers these events was specified explicitly by the application code the **implicit** field in the
2694 event structure will be set to **0**.

5.1.4. Data Allocation

2696 The *data allocation* event names are

acc_ev_create**acc_ev_delete**

acc_ev_alloc
acc_ev_free

2697 An **acc_ev_alloc** event is triggered when the OpenACC runtime allocates memory from the de-
 2698 vice memory pool, and an **acc_ev_free** event is triggered when the runtime frees that memory.
 2699 An **acc_ev_create** event is triggered when the OpenACC runtime associates device memory
 2700 with host memory, such as for a data clause (**create**, **copyin**, **copy**, **copyout**) at entry to
 2701 a data construct, compute construct, at an **enter data** directive, or in a call to a data API rou-
 2702 tine (**acc_copyin**, **acc_create**, ...). An **acc_ev_create** event may be preceded by an
 2703 **acc_ev_alloc** event, if newly allocated memory is used for this device data, or it may not, if
 2704 the runtime manages its own memory pool. An **acc_ev_delete** event is triggered when the
 2705 OpenACC runtime disassociates device memory from host memory, such as for a data clause at exit
 2706 from a data construct, compute construct, at an **exit data** directive, or in a call to a data API
 2707 routine (**acc_copyout**, **acc_delete**, ...). An **acc_ev_delete** event may be followed by
 2708 an **acc_ev_free** event, if the disassociated device memory is freed, or it may not, if the runtime
 2709 manages its own memory pool.

2710 When the action that generates a *data allocation* event was generated explicitly by the application
 2711 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event
 2712 is triggered because of a variable or array with implicitly-determined data attributes or otherwise
 2713 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

2714 **5.1.5. Data Construct**

2715 The events for entering and leaving *data constructs* are mapped to *enter data* and *exit data* events
 2716 as described in Section 5.1.3 Enter Data and Exit Data.

2717 **5.1.6. Update Directive**

2718 The *update directive* event names are

acc_ev_update_start
acc_ev_update_end

2719 The **acc_ev_update_start** event will be triggered at an **update** directive, before any *data*
 2720 *update* or *wait* events that are associated with the update directive are carried out, and the corre-
 2721 sponding **acc_ev_update_end** event will be triggered after any of the associated events.

2722 **5.1.7. Compute Construct**

2723 The *compute construct* event names are

acc_ev_compute_construct_start
acc_ev_compute_construct_end

2724 The **acc_ev_compute_construct_start** event is triggered at entry to a compute construct,
 2725 before any *launch* events that are associated with entry to the compute construct. The **acc_ev_compute_construct**

2726 event is triggered at the exit of the compute construct, after any *launch* events associated with exit
2727 from the compute construct. If there are data clauses on the compute construct, those data clauses
2728 may be treated as part of the compute construct, or as part of a data construct containing the compute
2729 construct. The callbacks for data clauses must use the same line numbers as for the compute
2730 construct events.

2731 **5.1.8. Enqueue Kernel Launch**

2732 The *launch* event names are

```
acc_ev_enqueue_launch_start
acc_ev_enqueue_launch_end
```

2733 The **acc_ev_enqueue_launch_start** event is triggered just before an accelerator compu-
2734 tation is enqueued for execution on the device, and **acc_ev_enqueue_launch_end** is trig-
2735 gered just after the computation is enqueued. Note that these events are synchronous with the
2736 host enqueueing the computation to the device, not with the device executing the computation.
2737 The **acc_ev_enqueue_launch_start** event callback routine is invoked just before the com-
2738 putation is enqueued, not just before the computation starts execution. More importantly, the
2739 **acc_ev_enqueue_launch_end** event callback routine is invoked after the computation is en-
2740 queued, not after the computation finished executing.

2741 **Note:** Measuring the time between the start and end launch callbacks is often unlikely to be useful,
2742 since it will only measure the time to manage the launch queue, not the time to execute the code on
2743 the device.

2744 **5.1.9. Enqueue Data Update (Upload and Download)**

2745 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
```

2746 The **_start** events are triggered just before each upload (data copy from host to device) oper-
2747 ation is or download (data copy from device to host) operation is enqueued for execution on the
2748 device. The corresponding **_end** events are triggered just after each upload or download operation
2749 is enqueued.

2750 **Note:** Measuring the time between the start and end update callbacks is often unlikely to be useful,
2751 since it will only measure the time to manage the enqueue operation, not the time to perform the
2752 actual upload or download.

2753 When the action that generates a *data update* event was generated explicitly by the application
2754 code the **implicit** field in the event structure will be set to **0**. When the *data allocation* event
2755 is triggered because of a variable or array with implicitly-determined data attributes or otherwise
2756 implicitly by the compiler the **implicit** field in the event structure will be set to **1**.

2757 **5.1.10. Wait**2758 The *wait* event names are

```

acc_ev_wait_start
acc_ev_wait_end

```

2759 An **acc_ev_wait_start** will be triggered for each relevant queue before the host thread waits
 2760 for that queue to be empty. A **acc_ev_wait_end** will be triggered for each relevant queue after
 2761 the host thread has determined that the queue is empty.

2762 Wait events occur when the host and device synchronize, either due to a **wait** directive or by a
 2763 *wait* clause on a synchronous data construct, compute construct, or **enter data**, **exit data**,
 2764 or **update** directive. For *wait* events triggered by an explicit synchronous **wait** directive or
 2765 *wait* clause, the **implicit** field in the event structure will be **0**. For all other wait events, the
 2766 **implicit** field in the event structure will be **1**.

2767 The OpenACC runtime need not trigger *wait* events for queues that have not been used in the
 2768 program, and need not trigger *wait* events for queues that have not been used by this thread since
 2769 the last *wait* operation. For instance, an **acc wait** directive with no arguments is defined to wait on
 2770 all queues. If the program only uses the default (synchronous) queue and the queue associated with
 2771 **async(1)** and **async(2)** then an **acc wait** directive may trigger *wait* events only for those
 2772 three queues. If the implementation knows that no activities have been enqueued on the **async(2)**
 2773 queue since the last *wait* operation, then the **acc wait** directive may trigger *wait* events only for
 2774 the default queue and the **async(1)** queue.

2775 **5.2. Callbacks Signature**

2776 This section describes the signature of event callbacks. All event callbacks have the same signature.
 2777 The routine prototypes are available in the header file **acc_prof.h**, which is delivered with the
 2778 OpenACC implementation.

2779 All callback routines have three arguments. The first argument is a pointer to a struct containing
 2780 general information; the same struct type is used for all callback events. The second argument is
 2781 a pointer to a struct containing information specific to that callback event; there is one struct type
 2782 containing information for data events, **another struct type containing information for kernel launch**
 2783 **events, and a third struct type for other events, containing essentially no information.** The third
 2784 argument is a pointer to a struct containing information about the application programming interface
 2785 (API) being used for the specific device. For NVIDIA CUDA devices, this contains CUDA-specific
 2786 information; for OpenCL devices, this contains OpenCL-specific information. Other interfaces can
 2787 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*);

```

2788 In the descriptions, the datatype **ssize_t** means a signed 32-bit integer for a 32-bit binary and
 2789 a 64-bit integer for a 64-bit binary, the datatype **size_t** means an unsigned 32-bit integer for a

2790 32-bit binary and a 64-bit integer for a 64-bit binary, and the datatype **int** means a 32-bit integer
 2791 for both 32-bit and 64-bit binaries. A null pointer is the pointer with value zero.

2792 5.2.1. First Argument: General Information

2793 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;
```

2794 The fields are described below.

2795 • **acc_event_t event_type** - The event type that triggered this callback. The datatype
 2796 is the enumeration type **acc_event_t**, described in the previous section. This allows the
 2797 same callback routine to be used for different events.

2798 • **int valid_bytes** - The number of valid bytes in this struct. This allows a library to inter-
 2799 face with newer runtimes that may add new fields to the struct at the end while retaining com-
 2800 patibility with older runtimes. A runtime must fill in the **event_type** and **valid_bytes**
 2801 fields, and must fill in values for all fields with offset less than **valid_bytes**. The value of
 2802 **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)
```

2803 • **int version** - A version number; the value of **_OPENACC**.

2804 • **acc_device_t device_type** - The device type corresponding to this event. The datatype
 2805 is **acc_device_t**, an enumeration type of all the supported accelerator device types, de-
 2806 fined in **openacc.h**.

2807 • **int device_number** - The device number. Each device is numbered, typically starting at
 2808 device zero. For applications that use more than one device type, the device numbers may be
 2809 unique across all devices or may be unique only across all devices of the same device type.

2810 • **int thread_id** - The host thread ID making the callback. Host threads are given unique
 2811 thread ID numbers typically starting at zero. This is not necessarily the same as the OpenMP
 2812 thread number.

- 2813 • **ssize_t async** - The value of the **async()** clause for the directive that triggered this
2814 callback.
- 2815 • **ssize_t async_queue** - If the runtime uses a limited number of asynchronous queues,
2816 this field contains the internal asynchronous queue number used for the event.
- 2817 • **const char* src_file** - A pointer to null-terminated string containing the name of or
2818 path to the source file, if known, or a null pointer if not. If the library wants to save the source
2819 file name, it should allocate memory and copy the string.
- 2820 • **const char* func_name** - A pointer to a null-terminated string containing the name of
2821 the function in which the event occurred, if known, or a null pointer if not. If the library wants
2822 to save the function name, it should allocate memory and copy the string.
- 2823 • **int line_no** - The line number of the directive or program construct or the starting line
2824 number of the OpenACC construct corresponding to the event. A negative or zero value
2825 means the line number is not known.
- 2826 • **int end_line_no** - For an OpenACC construct, this contains the line number of the end
2827 of the construct. A negative or zero value means the line number is not known.
- 2828 • **int func_line_no** - The line number of the first line of the function named in **func_name**.
2829 A negative or zero value means the line number is not known.
- 2830 • **int func_end_line_no** - The last line number of the function named in **func_name**.
2831 A negative or zero value means the line number is not known.

2832 5.2.2. Second Argument: Event-Specific Information

2833 The second argument is a pointer to the **acc_event_info** union type.

```

2834 typedef union acc_event_info{
2835     acc_event_t event_type;
2836     acc_data_event_info data_event;
2837     acc_launch_event_info launch_event;
2838     acc_other_event_info other_event;
2839 }acc_event_info;

```

2834 The **event_type** field selects which union member to use. The first five members of each union
2835 member are identical. The second through fifth members of each union member (**valid_bytes**,
2836 **parent_construct**, **implicit**, and **tool_info**) have the same semantics for all event
2837 types:

- 2838 • **int valid_bytes** - The number of valid bytes in the respective struct. (This field is similar
2839 used as discussed in Section 5.2.1 First Argument: General Information.)
- 2840 • **acc_construct_t parent_construct** - This field describes the type of construct
2841 that caused the event to be emitted. The possible values for this field are defined by the
2842 **acc_construct_t** enum, described at the end of this section.
- 2843 • **int implicit** - This field is set to 1 for any implicit event, such as an implicit wait at
2844 a synchronous data construct or synchronous enter data, exit data or update directive. This

2845 field is set to zero when the event is triggered by an explicit directive or call to a runtime API
2846 routine.

2847 • **void* tool_info** - This field is used to pass tool-specific information from a **_start**
2848 event to the matching **_end** event. For a **_start** event callback, this field will be initialized
2849 to a null pointer. The value of this field for a **_end** event will be the value returned by
2850 the library in this field from the matching **_start** event callback, if there was one, or null
2851 otherwise. For events that are neither **_start** or **_end** events, this field will be null.

2852 Data Events

2853 For a data event, as noted in the event descriptions, the second argument will be a pointer to the
2854 **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;
```

2855 The fields specific for a data event are:

2856 • **acc_event_t event_type** - The event type that triggered this callback. The events that
2857 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
```

2858 • **const char* var_name** - A pointer to null-terminated string containing the name of the
2859 variable for which this event is triggered, if known, or a null pointer if not. If the library wants
2860 to save the variable name, it should allocate memory and copy the string.

2861 • **size_t bytes** - The number of bytes for the data event.

2862 • **const void* host_ptr** - If available and appropriate for this event, this is a pointer to
2863 the host data.

2864 • **const void* device_ptr** - If available and appropriate for this event, this is a pointer
2865 to the corresponding device data.

2866 **Launch Events**

2867 For a launch event, as noted in the event descriptions, the second argument will be a pointer to the
2868 **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;

```

2869 The fields specific for a launch event are:

2870 • **acc_event_t event_type** - The event type that triggered this callback. The events that
2871 use the **acc_launch_event_info** struct are:

```

    acc_ev_enqueue_launch_start
    acc_ev_enqueue_launch_end

```

2872 • **const char* kernel_name** - A pointer to null-terminated string containing the name of
2873 the kernel being launched, if known, or a null pointer if not. If the library wants to save the
2874 kernel name, it should allocate memory and copy the string.

2875 • **size_t num_gangs, num_workers, vector_length** - The number of gangs, work-
2876 ers and vector lanes created for this kernel launch.

2877 **Other Events**

2878 For any event that does not use the **acc_data_event_info** or **acc_launch_event_info**
2879 struct, the second argument to the callback routine will be a pointer to **acc_other_event_info**
2880 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;

```

2881 **Parent Construct Enumeration**

2882 All event structures contain a **parent_construct** member that describes the type of construct
2883 that caused the event to be emitted. The purpose of this field is to provide a means to identify

2884 the type of construct emitting the event in the cases where an event may be emitted by multi-
 2885 ple construct types, such as is the case with data and wait events. The possible values for the
 2886 **parent_construct** field are defined in the enumeration type **acc_construct_t**. In the
 2887 case of combined directives, the outermost construct of the combined construct should be specified
 2888 as the **parent_construct**. If the event was emitted as the result of the application making a
 2889 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;
```

2890 5.2.3. Third Argument: API-Specific Information

2891 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;
```

2892 The fields are described below:

- 2893 • **acc_device_api device_api** - The API in use for this device. The data type is the
 2894 enumeration **acc_device_api**, which is described later in this section.
- 2895 • **int valid_bytes** - The number of valid bytes in this struct. See the discussion above in
 2896 Section 5.2.1 First Argument: General Information.

- 2897 • **acc_device_t device_type** - The device type; the datatype is **acc_device_t**, de-
2898 fined in **openacc.h**.
- 2899 • **int vendor** - An identifier to identify the OpenACC vendor; contact your vendor to deter-
2900 mine the value used by that vendor's runtime.
- 2901 • **const void* device_handle** - If applicable, this will be a pointer to the API-specific
2902 device information.
- 2903 • **const void* context_handle** - If applicable, this will be a pointer to the API-specific
2904 context information.
- 2905 • **const void* async_handle** - If applicable, this will be a pointer to the API-specific
2906 async queue information.

2907 According to the value of **device_api** a library can cast the pointers of the fields **device_handle**,
2908 **context_handle** and **async_handle** to the respective device API type. The following device
2909 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;

```

2910 5.3. Loading the Library

2911 This section describes how a tools library is loaded when the program is run. Four methods are
2912 described.

- 2913 • A tools library may be linked with the program, as any other library is linked, either as a
2914 static library or a dynamic library, and the runtime will call a predefined library initialization
2915 routine that will register the event callbacks.
- 2916 • The OpenACC runtime implementation may support a dynamic tools library, such as a shared
2917 object for Linux or OS/X, or a DLL for Windows, which is then dynamically loaded at runtime
2918 under control of the environment variable **ACC_PROFLIB**.
- 2919 • Some implementations where the OpenACC runtime is itself implemented as a dynamic li-
2920 brary may support adding a tools library using the **LD_PRELOAD** feature in Linux.
- 2921 • A tools library may be linked with the program, as in the first option, and the application itself
2922 can call a library initialization routine that will register the event callbacks.

2923 Callbacks are registered with the runtime by calling **acc_prof_register** for each event as
2924 described in Section 5.4 Registering Event Callbacks. The prototype for **acc_prof_register**
2925 is:

```

extern void acc_prof_register
    (acc_event_t event_type, acc_prof_callback cb,

```

```
acc_register_t info);
```

2926 The first argument to **acc_prof_register** is the event for which a callback is being registered
2927 (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*);
```

2928 The third argument is usually zero (or **acc_reg**). See Section 5.4.2 Disabling and Enabling Callbacks
2929 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;
```

2930 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);
```

2931 As shown in this example, the same routine (**prof_data**) can be registered for multiple events.
2932 The routine can use the **event_type** field in the **acc_prof_info** structure to determine for
2933 what event it was invoked.

2934 5.3.1. Library Registration

2935 The OpenACC runtime will invoke **acc_register_library**, passing the addresses of the reg-
2936 istration routines **acc_prof_register** and **acc_prof_unregister**, in case that routine
2937 comes from a dynamic library. In the third argument it passes the address of the lookup routine
2938 **acc_prof_lookup** to obtain the addresses of inquiry functions. No inquiry functions are de-
2939 fined in this profiling interface, but we preserve this argument for future support of sampling-based
2940 tools.

2941 Typically, the OpenACC runtime will include a *weak* definition of **acc_register_library**,
2942 which does nothing and which will be called when there is no tools library. In this case, the library
2943 can save the addresses of these routines and/or make registration calls to register any appropriate
2944 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);
```

2945 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);

```

2946 The third argument passes the address to the lookup function `acc_prof_lookup` to obtain the
 2947 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);

```

2948 The argument of the lookup function is a string with the name of the inquiry function. There are no
 2949 inquiry functions defined for this interface.

2950 5.3.2. Statically-Linked Library Initialization

2951 A tools library can be compiled and linked directly into the application. If the library provides an
 2952 external routine `acc_register_library` as specified in Section 5.3.1 Library Registration, the
 2953 runtime will invoke that routine to initialize the library.

2954 The sequence of events is:

- 2955 1. The runtime invokes the `acc_register_library` routine from the library.
- 2956 2. The `acc_register_library` routine calls `acc_prof_register` for each event to
 2957 be monitored.
- 2958 3. `acc_prof_register` records the callback routines.
- 2959 4. The program runs, and your callback routines are invoked at the appropriate events.

2960 In this mode, only one tool library is supported.

2961 5.3.3. Runtime Dynamic Library Loading

2962 A common case is to build the tools library as a dynamic library (shared object for Linux or OS/X,
 2963 DLL for Windows). In that case, you can have the OpenACC runtime load the library during initial-
 2964 ization. This allows you to enable runtime profiling without rebuilding or even relinking your ap-
 2965 plication. The dynamic library must implement a registration routine `acc_register_library`
 2966 as specified in Section 5.3.1 Library Registration.

2967 The user may set the environment variable `ACC_PROFLIB` to the path to the library will tell the
 2968 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
```

2969 When the OpenACC runtime initializes, it will read the **ACC_PROFLIB** environment variable (with
2970 **getenv**). The runtime will open the dynamic library (using **dlopen** or **LoadLibraryA**); if
2971 the library cannot be opened, the runtime may abort, or may continue execution with or with-
2972 out an error message. If the library is successfully opened, the runtime will get the address of
2973 the **acc_register_library** routine (using **dlsym** or **GetProcAddress**). If this routine
2974 is resolved in the library, it will be invoked passing in the addresses of the registration routine
2975 **acc_prof_register**, the deregistration routine **acc_prof_unregister**, and the lookup
2976 routine **acc_prof_lookup**. The registration routine in your library, **acc_register_library**,
2977 should register the callbacks by calling the **register** argument, and should save the addresses of
2978 the arguments (**register**, **unregister**, and **lookup**) for later use, if needed.

2979 The sequence of events is:

- 2980 1. Initialization of the OpenACC runtime.
- 2981 2. OpenACC runtime reads **ACC_PROFLIB**.
- 2982 3. OpenACC runtime loads the library.
- 2983 4. OpenACC runtime calls the **acc_register_library** routine in that library.
- 2984 5. Your **acc_register_library** routine calls **acc_prof_register** for each event to
2985 be monitored.
- 2986 6. **acc_prof_register** records the callback routines.
- 2987 7. The program runs, and your callback routines are invoked at the appropriate events.

2988 If supported, paths to multiple dynamic libraries may be specified in the **ACC_PROFLIB** environ-
2989 ment variable, separated by semicolons (;). The OpenACC runtime will open these libraries and in-
2990 voke the **acc_register_library** routine for each, in the order they appear in **ACC_PROFLIB**.

2991 5.3.4. Preloading with LD_PRELOAD

2992 The implementation may also support dynamic loading of a tools library using the **LD_PRELOAD**
2993 feature available in some systems. In such an implementation, you need only specify your tools
2994 library path in the **LD_PRELOAD** environment variable before executing your program. The Open-
2995 ACC runtime will invoke the **acc_register_library** routine in your tools library at initial-
2996 ization time. This requires that the OpenACC runtime include a dynamic library with a default
2997 (empty) implementation of **acc_register_library** that will be invoked in the normal case
2998 where there is no **LD_PRELOAD** setting. If an implementation only supports static linking, or if the
2999 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
```

3000 The sequence of events is:

- 3001 1. The operating system loader loads the library specified in **LD_PRELOAD**.
- 3002 2. The call to **acc_register_library** in the OpenACC runtime is resolved to the routine
3003 in the loaded tools library.
- 3004 3. OpenACC runtime calls the **acc_register_library** routine in that library.
- 3005 4. Your **acc_register_library** routine calls **acc_prof_register** for each event to
3006 be monitored.
- 3007 5. **acc_prof_register** records the callback routines.
- 3008 6. The program runs, and your callback routines are invoked at the appropriate events.

3009 In this mode, only a single tools library is supported, since only one **acc_register_library**
3010 initialization routine will get resolved by the dynamic loader.

3011 5.3.5. Application-Controlled Initialization

3012 An alternative to default initialization is to have the application itself call the library initialization
3013 routine, which then calls **acc_prof_register** for each appropriate event. The library may be
3014 statically linked to the application or your application may dynamically load the library.

3015 The sequence of events is:

- 3016 1. Your application calls the library initialization routine.
- 3017 2. The library initialization routine calls **acc_prof_register** for each event to be moni-
3018 tored.
- 3019 3. **acc_prof_register** records the callback routines.
- 3020 4. The program runs, and your callback routines are invoked at the appropriate events.

3021 In this mode, multiple tools libraries can be supported, with each library initialization routine in-
3022 voked by the application.

3023 5.4. Registering Event Callbacks

3024 This section describes how to register and unregister callbacks, temporarily disabling and enabling
3025 callbacks, the behavior of dynamic registration and unregistration, and requirements on an Open-
3026 ACC implementation to correctly support the interface.

3027 **5.4.1. Event Registration and Unregistration**

3028 The library must call the registration routine `acc_prof_register` to register each callback
 3029 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);
}
```

3030 In this example the `prof_data` routine will be invoked for each data upload and download event,
 3031 and the `prof_launch` routine will be invoked for each launch event. The `prof_data` routine
 3032 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 :
            ...
    }
}
```

3033 **Multiple Callbacks**

3034 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);
```

3035 For most events, the callbacks will be invoked in the order in which they are registered. However,
 3036 *end* events, named `acc_ev_..._end`, invoke callbacks in the reverse order. Essentially, each
 3037 event has an ordered list of callback routines. A new callback routine is appended to the tail of the
 3038 list for that event. For most events, that list is traversed from the head to the tail, but for *end* events,
 3039 the list is traversed from the tail to the head.

3040 If a callback is registered, then later unregistered, then later still registered again, the second regis-
 3041 tration is considered to be a new callback, and the callback routine will then be appended to the tail
 3042 of the callback list for that event.

3043 **Unregistering**

3044 A matching call to `acc_prof_unregister` will remove that routine from the list of callback
 3045 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

```

3046 Each entry on the callback list must also have a *ref* count. This keeps track of how many times
 3047 this routine was added to this event's callback list. If a routine is registered *n* times, it must be
 3048 unregistered *n* times before it is removed from the list. Note that if a routine is registered multiple
 3049 times for the same event, its *ref* count will be incremented with each registration, but it will only be
 3050 invoked once for each event instance.

3051 **5.4.2. Disabling and Enabling Callbacks**

3052 A callback routine may be temporarily disabled on the callback list for an event, then later re-
 3053 enabled. The behavior is slightly different than unregistering and later re-registering that event.
 3054 When a routine is disabled and later re-enabled, the routine's position on the callback list for that
 3055 event is preserved. When a routine is unregistered and later re-registered, the routine's position on
 3056 the callback list for that event will move to the tail of the list. Also, unregistering a callback must be
 3057 done *n* times if the callback routine was registered *n* times. In contrast, disabling, and enabling an
 3058 event sets a toggle. Disabling a callback will immediately reset the toggle and disable calls to that
 3059 routine for that event, even if it was enabled multiple times. Enabling a callback will immediately
 3060 set the toggle and enable calls to that routine for that event, even if it was disabled multiple times.
 3061 Registering a new callback initially sets the toggle.

3062 A call to `acc_prof_unregister` with a value of `acc_toggle` as the third argument will dis-
 3063 able callbacks to the given routine. A call to `acc_prof_register` with a value of `acc_toggle`
 3064 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

```

3065 A call to either `acc_prof_unregister` or `acc_prof_register` to disable or enable a call-
 3066 back when that callback is not currently registered for that event will be ignored with no error.

3067 All callbacks for an event may be disabled (and re-enabled) by passing `NULL` to the second argument
 3068 and `acc_toggle` to the third argument of `acc_prof_unregister` (and `acc_prof_register`).

3069 This sets a toggle for that event, which is distinct from the toggle for each callback for that event.
 3070 While the event is disabled, no callbacks for that event will be invoked. Callbacks for that event can
 3071 be registered, unregistered, enabled, and disabled while that event is disabled, but no callbacks will
 3072 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

```

3073 Finally, all callbacks can be disabled (and enabled) by passing the argument list (**0**, **NULL**,
 3074 **acc_toggle**) to **acc_prof_unregister** (and **acc_prof_register**). This sets a global
 3075 toggle disabling all callbacks, which is distinct from the toggle enabling callbacks for each event and
 3076 the toggle enabling each callback routine. The behavior of passing zero as the first argument and a
 3077 non-**NULL** value as the second argument to **acc_prof_unregister** or **acc_prof_register**
 3078 is not defined, and may be ignored by the runtime without error.

3079 All callbacks can be disabled (or enabled) for just the current thread by passing the argument list
 3080 (**0**, **NULL**, **acc_toggle_per_thread**) to **acc_prof_unregister** (and **acc_prof_register**).
 3081 This is the only thread-specific interface to **acc_prof_register** and **acc_prof_unregister**,
 3082 all other calls to register, unregister, enable, or disable callbacks affect all threads in the application.

3083 5.5. Advanced Topics

3084 This section describes advanced topics such as dynamic registration and changes of the execution
 3085 state for callback routines as well as the runtime and tool behavior for multiple host threads.

3086 5.5.1. Dynamic Behavior

3087 Callback routines may be registered or unregistered, enabled or disabled at any point in the execution
3088 of the program. Calls may appear in the library itself, during the processing of an event. The
3089 OpenACC runtime must allow for this case, where the callback list for an event is modified while
3090 that event is being processed.

3091 Dynamic Registration and Unregistration

3092 Calls to **acc_register** and **acc_unregister** may occur at any point in the application. A
3093 callback routine can be registered or unregistered from a callback routine, either the same routine
3094 or another routine, for a different event or the same event for which the callback was invoked. If a
3095 callback routine is registered for an event while that event is being processed, then the new callback
3096 routine will be added to the tail of the list of callback routines for this event. Some events (the
3097 **_end**) events process the callback routines in reverse order, from the tail to the head. For those
3098 events, adding a new callback routine will not cause the new routine to be invoked for this instance
3099 of the event. The other events process the callback routines in registration order, from the head to
3100 the tail. Adding a new callback routine for such a event will cause the runtime to invoke that newly
3101 registered callback routine for this instance of the event. Both the runtime and the library must
3102 implement and expect this behavior.

3103 If an existing callback routine is unregistered for an event while that event is being processed, that
3104 callback routine is removed from the list of callbacks for this event. For any event, if that callback
3105 routine had not yet been invoked for this instance of the event, it will not be invoked.

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 Dynamic Enabling and Disabling

3109 Calls to **acc_register** and **acc_unregister** to enable and disable a specific callback for
3110 an event, enable or disable all callbacks for an event, or enable or disable all callbacks may occur
3111 at any point in the application. A callback routine can be enabled or disabled from a callback
3112 routine, either the same routine or another routine, for a different event or the same event for which
3113 the callback was invoked. If a callback routine is enabled for an event while that event is being
3114 processed, then the new callback routine will be immediately enabled. If it appears on the list of
3115 callback routines closer to the head (for **_end** events) or closer to the tail (for other events), that
3116 newly-enabled callback routine will be invoked for this instance of this event, unless it is disabled
3117 or unregistered before that callback is reached.

3118 If a callback routine is disabled for an event while that event is being processed, that callback routine
3119 is immediately disabled. For any event, if that callback routine had not yet been invoked for this in-
3120 stance of the event, it will not be invoked, unless it is enabled before that callback routine is reached
3121 in the list of callbacks for this event. If all callbacks for an event are disabled while that event is
3122 being processed, or all callbacks are disabled for all events while an event is being processed, then
3123 when this callback routine returns, no more callbacks will be invoked for this instance of the event.

3124 Registering and unregistering a callback routine is a global operation and affects all threads, in a
3125 multithreaded application. See Section 5.4.1 Multiple Callbacks.

3126 5.5.2. OpenACC Events During Event Processing

3127 OpenACC events may occur during event processing. This may be because of OpenACC API rou-
3128 tine calls or OpenACC constructs being reached during event processing, or because of multiple host
3129 threads executing asynchronously. Both the OpenACC runtime and the tool library must implement
3130 the proper behavior.

3131 5.5.3. Multiple Host Threads

3132 Many programs that use OpenACC also use multiple host threads, such as programs using the
3133 OpenMP API. The appearance of multiple host threads affects both the OpenACC runtime and the
3134 tools library.

3135 Runtime Support for Multiple Threads

3136 The OpenACC runtime must be thread-safe, and the OpenACC runtime implementation of this
3137 tools interface must also be thread-safe. All threads use the same set of callbacks for all events, so
3138 registering a callback from one thread will cause all threads to execute that callback. This means that
3139 managing the callback lists for each event must be protected from multiple simultaneous updates.
3140 This includes adding a callback to the tail of the callback list for an event, removing a callback from
3141 the list for an event, and incrementing or decrementing the *ref* count for a callback routine for an
3142 event.

3143 In addition, one thread may register, unregister, enable, or disable a callback for an event while
3144 another thread is processing the callback list for that event asynchronously. The exact behavior may
3145 be dependent on the implementation, but some behaviors are expected and others are disallowed.
3146 In the following examples, there are three callbacks, A, B, and C, registered for event E in that
3147 order, where callbacks A and B are enabled and callback C is temporarily disabled. Thread T1 is
3148 dynamically modifying the callbacks for event E while thread T2 is processing an instance of event
3149 E.

- 3150 • Suppose thread T1 unregisters or disables callback A for event E. Thread T2 may or may not
3151 invoke callback A for this event instance, but it must invoke callback B; if it invokes callback
3152 A, that must precede the invocation of callback B.
- 3153 • Suppose thread T1 unregisters or disables callback B for event E. Thread T2 may or may not
3154 invoke callback B for this event instance, but it must invoke callback A; if it invokes callback
3155 B, that must follow the invocation of callback A.
- 3156 • Suppose thread T1 unregisters or disables callback A and then unregisters or disables callback
3157 B for event E. Thread T2 may or may not invoke callback A and may or may not invoke
3158 callback B for this event instance, but if it invokes both callbacks, it must invoke callback A
3159 before it invokes callback B.
- 3160 • Suppose thread T1 unregisters or disables callback B and then unregisters or disables callback
3161 A for event E. Thread T2 may or may not invoke callback A and may or may not invoke
3162 callback B for this event instance, but if it invokes callback B, it must have invoked callback
3163 A for this event instance.
- 3164 • Suppose thread T1 is registering a new callback D for event E. Thread T2 may or may not

3165 invoke callback D for this event instance, but it must invoke both callbacks A and B. If it
3166 invokes callback D, that must follow the invocations of A and B.

- 3167 • Suppose thread T1 is enabling callback C for event E. Thread T2 may or may not invoke
3168 callback C for this event instance, but it must invoke both callbacks A and B. If it invokes
3169 callback C, that must follow the invocations of A and B.

3170 The `acc_prof_info` struct has a `thread_id` field, which the runtime must set to a unique
3171 value for each host thread, though it need not be the same as the OpenMP threadnum value.

3172 **Library Support for Multiple Threads**

3173 The tool library must also be thread-safe. The callback routine will be invoked in the context of the
3174 thread that reaches the event. The library may receive a callback from a thread T2 while it's still
3175 processing a callback, from the same event type or from a different event type, from another thread
3176 T1. The `acc_prof_info` struct has a `thread_id` field, which the runtime must set to a unique
3177 value for each host thread.

3178 If the tool library uses dynamic callback registration and unregistration, or callback disabling and
3179 enabling, recall that unregistering or disabling an event callback from one thread will unregister or
3180 disable that callback for all threads, and registering or enabling an event callback from any thread
3181 will register or enable it for all threads. If two or more threads register the same callback for the
3182 same event, the behavior is the same as if one thread registered that callback multiple times; see
3183 Section 5.4.1 Multiple Callbacks. The `acc_unregister` routine must be called as many times
3184 as `acc_register` for that callback/event pair in order to totally unregister it. If two threads
3185 register two different callback routines for the same event, unless the order of the registration calls
3186 is guaranteed by some synchronization method, the order in which the runtime sees the registration
3187 may differ for multiple runs, meaning the order in which the callbacks occur will differ as well.

3188 6. Glossary

3189 Clear and consistent terminology is important in describing any programming model. We define
3190 here the terms you must understand in order to make effective use of this document and the associ-
3191 ated programming model.

3192 **Accelerator** – a special-purpose co-processor attached to a CPU and to which the CPU can offload
3193 data and compute kernels to perform compute-intensive calculations.

3194 **Accelerator routine** – a C or C++ function or Fortran subprogram compiled for the accelerator
3195 with the **routine** directive.

3196 **Accelerator thread** – a thread of execution that executes on the accelerator; a single vector lane of
3197 a single worker of a single gang.

3198 **Aggregate datatype** – an array or structure datatype, or any non-scalar datatype. In Fortran, aggre-
3199 gate datatypes include arrays and derived types. In C, aggregate datatypes include fixed size arrays,
3200 targets of pointers, structs, and unions. In C++, aggregate datatypes include fixed size arrays, targets
3201 of pointers, classes, structs, and unions.

3202 **Aggregate variables** – an array or structure variable, or a variable of any non-scalar datatype.

3203 **Async-argument** – an *async-argument* is a nonnegative scalar integer expression (*int* for C or C++,
3204 *integer* for Fortran), or one of the special async values **acc_async_noval** or **acc_async_sync**.

3205 **Barrier** – a type of synchronization where all parallel execution units or threads must reach the
3206 barrier before any execution unit or thread is allowed to proceed beyond the barrier; modeled after
3207 the starting barrier on a horse race track.

3208 **Compute intensity** – for a given loop, region, or program unit, the ratio of the number of arithmetic
3209 operations performed on computed data divided by the number of memory transfers required to
3210 move that data between two levels of a memory hierarchy.

3211 **Construct** – a directive and the associated statement, loop, or structured block, if any.

3212 **Compute construct** – a *parallel construct*, *kernels construct*, or *serial construct*.

3213 **Compute region** – a *parallel region*, *kernels region*, or *serial region*.

3214 **CUDA** – the CUDA environment from NVIDIA is a C-like programming environment used to
3215 explicitly control and program an NVIDIA GPU.

3216 **Current device** – the device represented by the *acc-device-type-var* and *acc-device-num-var* ICVs

3217 **Current device type** – the device type represented by the *acc-device-type-var* ICV

3218 **Data lifetime** – the lifetime of a data object on the device, which may begin at the entry to a data re-
3219 gion, or at an **enter data** directive, or at a data API call such as **acc_copyin** or **acc_create**,
3220 and which may end at the exit from a data region, or at an **exit data** directive, or at a data API
3221 call such as **acc_delete**, **acc_copyout**, or **acc_shutdown**, or at the end of the program
3222 execution.

- 3223 **Data region** – a *region* defined by a **data** construct, or an implicit data region for a function or
3224 subroutine containing Accelerator directives. Data constructs typically allocate device memory and
3225 copy data from host to device memory upon entry, and copy data from device to host memory and
3226 deallocate device memory upon exit. Data regions may contain other data regions and compute
3227 regions.
- 3228 **Device** – a general reference to any type of accelerator.
- 3229 **Default asynchronous queue** – the asynchronous activity queue represented in the *acc-default-*
3230 *async-var* ICV
- 3231 **Device memory** – memory attached to an accelerator, logically and physically separate from the
3232 host memory.
- 3233 **Directive** – in C or C++, a **#pragma**, or in Fortran, a specially formatted comment statement, that
3234 is interpreted by a compiler to augment information about or specify the behavior of the program.
- 3235 **DMA** – Direct Memory Access, a method to move data between physically separate memories;
3236 this is typically performed by a DMA engine, separate from the host CPU, that can access the host
3237 physical memory as well as an IO device or other physical memory.
- 3238 **GPU** – a Graphics Processing Unit; one type of accelerator device.
- 3239 **GPGPU** – General Purpose computation on Graphics Processing Units.
- 3240 **Host** – the main CPU that in this context has an attached accelerator device. The host CPU controls
3241 the program regions and data loaded into and executed on the device.
- 3242 **Host thread** – a thread of execution that executes on the host.
- 3243 **Implicit data region** – the data region that is implicitly defined for a Fortran subprogram or C
3244 function. A call to a subprogram or function enters the implicit data region, and a return from the
3245 subprogram or function exits the implicit data region.
- 3246 **Kernel** – a nested loop executed in parallel by the accelerator. Typically the loops are divided into
3247 a parallel domain, and the body of the loop becomes the body of the kernel.
- 3248 **Kernels region** – a *region* defined by a **kernels** construct. A kernels region is a structured block
3249 which is compiled for the accelerator. The code in the kernels region will be divided by the compiler
3250 into a sequence of kernels; typically each loop nest will become a single kernel. A kernels region
3251 may require device memory to be allocated and data to be copied from host to device upon region
3252 entry, and data to be copied from device to host memory and device memory deallocated upon exit.
- 3253 **Level of parallelism** – The possible levels of parallelism in OpenACC are gang, worker, vector,
3254 and sequential. One or more of gang, worker, and vector parallelism may be specified on a loop
3255 construct. Sequential execution corresponds to no parallelism. The **gang**, **worker**, **vector**, and
3256 **seq** clauses specify the level of parallelism for a loop.
- 3257 **Local memory** – the memory associated with the local thread.
- 3258 **Local thread** – the host thread or the accelerator thread that executes an OpenACC directive or
3259 construct.
- 3260 **Loop trip count** – the number of times a particular loop executes.
- 3261 **MIMD** – a method of parallel execution (Multiple Instruction, Multiple Data) where different exe-
3262 cution units or threads execute different instruction streams asynchronously with each other.

- 3263 **OpenCL** – short for Open Compute Language, a developing, portable standard C-like programming
3264 environment that enables low-level general-purpose programming on GPUs and other accelerators.
- 3265 **Orphaned loop construct** - a **loop** construct that is not lexically contained in any compute con-
3266 struct, that is, that has no parent compute construct.
- 3267 **Parallel region** – a *region* defined by a **parallel** construct. A parallel region is a structured block
3268 which is compiled for the accelerator. A parallel region typically contains one or more work-sharing
3269 loops. A parallel region may require device memory to be allocated and data to be copied from host
3270 to device upon region entry, and data to be copied from device to host memory and device memory
3271 deallocated upon exit.
- 3272 **Parent compute construct** – for a **loop** construct, the **parallel**, **kernels**, or **serial** con-
3273 struct that lexically contains the **loop** construct and is the innermost compute construct that con-
3274 tains that **loop** construct, if any.
- 3275 **Private data** – with respect to an iterative loop, data which is used only during a particular loop
3276 iteration. With respect to a more general region of code, data which is used within the region but is
3277 not initialized prior to the region and is re-initialized prior to any use after the region.
- 3278 **Procedure** – in C or C++, a function in the program; in Fortran, a subroutine or function.
- 3279 **Region** – all the code encountered during an instance of execution of a construct. A region includes
3280 any code in called routines, and may be thought of as the dynamic extent of a construct. This may
3281 be a *parallel region*, *kernels region*, *serial region*, *data region* or *implicit data region*.
- 3282 **Scalar** – a variable of scalar datatype. In Fortran, scalars must not have allocatable or pointer
3283 attributes.
- 3284 **Scalar datatype** – an intrinsic or built-in datatype that is not an array or aggregate datatype. In For-
3285 tran, scalar datatypes are integer, real, double precision, complex, or logical. In C, scalar datatypes
3286 are char (signed or unsigned), int (signed or unsigned, with optional short, long or long long at-
3287 tribute), enum, float, double, long double, `_Complex` (with optional float or long attribute), or any
3288 pointer datatype. In C++, scalar datatypes are char (signed or unsigned), `wchar_t`, int (signed or
3289 unsigned, with optional short, long or long long attribute), enum, bool, float, double, long double,
3290 or any pointer datatype. Not all implementations or targets will support all of these datatypes.
- 3291 **Serial region** – a *region* defined by a **serial** construct. A serial region is a structured block which
3292 is compiled for the accelerator. A serial region contains code that is executed by one vector lane of
3293 one worker in one gang. A serial region may require device memory to be allocated and data to be
3294 copied from host to device upon region entry, and data to be copied from device to host memory
3295 and device memory deallocated upon exit.
- 3296 **SIMD** – A method of parallel execution (single-instruction, multiple-data) where the same instruc-
3297 tion is applied to multiple data elements simultaneously.
- 3298 **SIMD operation** – a *vector operation* implemented with SIMD instructions.
- 3299 **Structured block** – in C or C++, an executable statement, possibly compound, with a single entry
3300 at the top and a single exit at the bottom. In Fortran, a block of executable statements with a single
3301 entry at the top and a single exit at the bottom.
- 3302 **Thread** – On a host processor, a thread is defined by a program counter and stack location; several
3303 host threads may comprise a process and share host memory. On an accelerator, a thread is any one
3304 vector lane of one worker of one gang on the device.

3305 **Vector operation** – a single operation or sequence of operations applied uniformly to each element
3306 of an array.

3307 **Visible device copy** – a copy of a variable, array, or subarray allocated in device memory that is
3308 visible to the program unit being compiled.

3309 **A. Recommendations for Implementors**

3310 This section gives recommendations for standard names and extensions to use for implementations
3311 for specific targets and target platforms, to promote portability across such implementations, and
3312 recommended options that programmers find useful. While this appendix is not part of the Open-
3313 ACC specification, implementations that provide the functionality specified herein are strongly rec-
3314 ommended to use the names in this section. The first subsection describes target devices, such as
3315 NVIDIA GPUs. The second subsection describes additional API routines for target platforms, such
3316 as CUDA and OpenCL. The third subsection lists several recommended options for implementa-
3317 tions.

3318 **A.1. Target Devices**

3319 **A.1.1. NVIDIA GPU Targets**

3320 This section gives recommendations for implementations that target NVIDIA GPU devices.

3321 **Accelerator Device Type**

3322 These implementations should use the name `acc_device_nvidia` for the `acc_device_t`
3323 type or return values from OpenACC Runtime API routines.

3324 **ACC_DEVICE_TYPE**

3325 An implementation should use the case-insensitive name `nvidia` for the environment variable
3326 `ACC_DEVICE_TYPE`.

3327 **device.type clause argument**

3328 An implementation should use the case-insensitive name `nvidia` as the argument to the `device_type`
3329 clause.

3330 **A.1.2. AMD GPU Targets**

3331 This section gives recommendations for implementations that target AMD GPUs.

3332 Accelerator Device Type

3333 These implementations should use the name **acc_device_radeon** for the **acc_device_t**
3334 type or return values from OpenACC Runtime API routines.

3335 ACC_DEVICE_TYPE

3336 These implementations should use the case-insensitive name **radeon** for the environment variable
3337 **ACC_DEVICE_TYPE**.

3338 device_type clause argument

3339 An implementation should use the case-insensitive name **radeon** as the argument to the **device_type**
3340 clause.

3341 A.2. API Routines for Target Platforms

3342 These runtime routines allow access to the interface between the OpenACC runtime API and the
3343 underlying target platform. An implementation may not implement all these routines, but if it
3344 provides this functionality, it should use these function names.

3345 A.2.1. NVIDIA CUDA Platform

3346 This section gives runtime API routines for implementations that target the NVIDIA CUDA Run-
3347 time or Driver API.

3348 **acc_get_current_cuda_device**

3349 **Summary** The **acc_get_current_cuda_device** routine returns the NVIDIA CUDA de-
3350 vice handle for the current device.

3351 **Format**

C or C++:

```
void* acc_get_current_cuda_device ();
```

3352 **acc_get_current_cuda_context**

3353 **Summary** The **acc_get_current_cuda_context** routine returns the NVIDIA CUDA
3354 context handle in use for the current device.

3355 **Format**

C or C++:

```
void* acc_get_current_cuda_context ();
```

3356 **acc_get_cuda_stream**

3357 **Summary** The **acc_get_cuda_stream** routine returns the NVIDIA CUDA stream handle
3358 in use for the current device for the specified async value.

3359 **Format**

C or C++:

```
void* acc_get_cuda_stream ( int async );
```

3360 **acc_set_cuda_stream**

3361 **Summary** The **acc_set_cuda_stream** routine sets the NVIDIA CUDA stream handle the
3362 current device for the specified async value.

3363 **Format**

C or C++:

```
int acc_set_cuda_stream ( int async, void* stream );
```

3364 **A.2.2. OpenCL Target Platform**

3365 This section gives runtime API routines for implementations that target the OpenCL API on any
3366 device.

3367 **acc_get_current_opencl_device**

3368 **Summary** The **acc_get_current_opencl_device** routine returns the OpenCL device
3369 handle for the current device.

3370 **Format**

C or C++:

```
void* acc_get_current_opencl_device ();
```

3371 **acc_get_current_opencl_context**

3372 **Summary** The **acc_get_current_opencl_context** routine returns the OpenCL context
3373 handle in use for the current device.

3374 **Format**

C or C++:

```
void* acc_get_current_opengl_context ();
```

3375 **acc_get_opengl_queue**

3376 **Summary** The `acc_get_opengl_queue` routine returns the OpenCL command queue handle in use for the current device for the specified async value.
3377

3378 **Format**

C or C++:

```
cl_command_queue acc_get_opengl_queue ( int async );
```

3379 **acc_set_opengl_queue**

3380 **Summary** The `acc_set_opengl_queue` routine returns the OpenCL command queue handle in use for the current device for the specified async value.
3381

3382 **Format**

C or C++:

```
void acc_set_opengl_queue ( int async, cl_command_queue cmdqueue );
```

3383 **A.3. Recommended Options**

3384 The following options are recommended for implementations; for instance, these may be implemented as command-line options to a compiler or settings in an IDE.
3385

3386 **A.3.1. C Pointer in Present clause**

3387 This revision of OpenACC clarifies the construct:

```
void test(int n ){  
    float* p;  
    ...  
    #pragma acc data present(p)  
    {  
        // code here...  
    }  
}
```

3388 This example tests whether the pointer **p** itself is present on the device. Implementations before
3389 this revision commonly implemented this by testing whether the pointer target **p[0]** was present
3390 on the device, and this appears in many programs assuming such. Until such programs are modified
3391 to comply with this revision, an option to implement **present (p)** as **present (p[0])** for C
3392 pointers may be helpful to users.

3393 **A.3.2. Autoscopying**

3394 If an implementation implements autoscopying to automatically determine variables that are private
3395 to a compute region or to a loop, or to recognize reductions in a compute region or a loop, an option
3396 to print a message telling what variables were affected by the analysis would be helpful to users. An
3397 option to disable the autoscopying analysis would be helpful to promote program portability across
3398 implementations.

Index

- 3399 **_OPENACC**, 12–14, 18, 101
- 3400 acc-default-async-var, 18, 68
- 3401 acc-device-num-var, 18
- 3402 acc-device-type-var, 18
- 3403 **acc_async_noval**, 12, 68
- 3404 **acc_async_sync**, 12, 68
- 3405 **ACC_DEVICE_NUM**, 19, 93
- 3406 **acc_device_nvidia**, 121
- 3407 **acc_device_radeon**, 122
- 3408 **ACC_DEVICE_TYPE**, 19, 93, 121, 122
- 3409 **ACC_PROFLIB**, 93
- 3410 action
 - 3411 attach, 32, 37
 - 3412 copyin, 36
 - 3413 copyout, 36
 - 3414 create, 36
 - 3415 delete, 37
 - 3416 detach, 32, 37
 - 3417 immediate, 38
 - 3418 present decrement, 35
 - 3419 present increment, 35
- 3420 AMD GPU target, 121
- 3421 **async** clause, 31, 63, 68
- 3422 async queue, 9
- 3423 *async-argument*, 68
- 3424 asynchronous execution, 9, 68
- 3425 **atomic** construct, 12, 51
- 3426 attach action, 32, 37
- 3427 **attach** clause, 42
- 3428 attachment counter, 32
- 3429 **auto** clause, 12, 47
- 3430 autoscoping, 125
- 3431 barrier synchronization, 8, 21, 22, 24, 117
- 3432 **bind** clause, 66
- 3433 **cache** directive, 49
- 3434 **capture** clause, 55
- 3435 **collapse** clause, 45
- 3436 common block, 33, 57, 58, 67
- 3437 compute construct, 117
- 3438 compute region, 117
- 3439 construct, 117
 - 3440 **atomic**, 51
 - 3441 compute, 117
 - 3442 **data**, 29, 33
 - 3443 **host_data**, 43
 - 3444 **kernels**, 21, 22, 33
 - 3445 **kernels loop**, 50
 - 3446 **parallel**, 20, 33
 - 3447 **parallel loop**, 50
 - 3448 **serial**, 23, 33
 - 3449 **serial loop**, 50
- 3450 **copy** clause, 39
- 3451 copyin action, 36
- 3452 **copyin** clause, 39
- 3453 copyout action, 36
- 3454 **copyout** clause, 40
- 3455 create action, 36
- 3456 **create** clause, 41, 58
- 3457 CUDA, 9, 10, 117, 121, 122
- 3458 data attribute
 - 3459 explicitly determined, 27
 - 3460 implicitly determined, 27
 - 3461 predetermined, 27
- 3462 data clause, 33
- 3463 **data** construct, 29, 33
- 3464 data lifetime, 117
- 3465 data region, 28, 118
 - 3466 implicit, 28
- 3467 **declare** directive, 12, 56
- 3468 **default** clause, 27
- 3469 **default (none)** clause, 12, 21, 22
- 3470 **default (none)** clause, 24
- 3471 default(present), 21, 22, 24
- 3472 delete action, 37
- 3473 **delete** clause, 42
- 3474 detach action, 32, 37
 - 3475 immediate, 38

- 3476 **detach** clause, 42
3477 **device** clause, 63
3478 **device_resident** clause, 57
3479 **device_type** clause, 12, 19, 33, 121, 122
3480 **deviceptr** clause, 33, 38
3481 direct memory access, 9, 118
3482 DMA, 9, 118
- 3483 **enter data** directive, 30, 33
3484 environment variable
3485 **_OPENACC**, 18
3486 **ACC_DEVICE_NUM**, 19, 93
3487 **ACC_DEVICE_TYPE**, 19, 93, 121, 122
3488 **ACC_PROFLIB**, 93
3489 **exit data** directive, 30, 33
3490 explicitly determined data attribute, 27
- 3491 **firstprivate** clause, 21, 26
3492 **firstprivate** clause, 24
- 3493 gang, 21, 24
3494 **gang** clause, 45, 66
3495 gang parallelism, 8
3496 *gang-arg*, 44
3497 gang-partitioned mode, 8
3498 gang-redundant mode, 8, 21, 24
3499 GP mode, 8
3500 GR mode, 8
- 3501 **host** clause, 12, 63
3502 **host_data** construct, 43
- 3503 ICV, 18
3504 **if** clause, 30, 31, 63
3505 immediate detach action, 38
3506 implicit data region, 28
3507 implicitly determined data attribute, 27
3508 **independent** clause, 48
3509 **init** directive, 59
3510 internal control variable, 18
- 3511 **kernels** construct, 21, 22, 33
3512 **kernels loop** construct, 50
- 3513 level of parallelism, 8, 118
3514 **link** clause, 12, 33, 58
3515 local memory, 9
3516 local thread, 9
3517 **loop** construct, 44
3518 orphaned, 45
- 3519 **no_create** clause, 41
3520 **nohost** clause, 67
3521 **num_gangs** clause, 25
3522 **num_workers** clause, 25
3523 **nvidia**, 121
3524 NVIDIA GPU target, 121
- 3525 OpenCL, 9, 10, 119, 121, 123
3526 orphaned **loop** construct, 45
- 3527 **parallel** construct, 20, 33
3528 **parallel loop** construct, 50
3529 parallelism
3530 level, 8, 118
3531 parent compute construct, 45
3532 predetermined data attribute, 27
3533 **present** clause, 33, 38
3534 present decrement action, 35
3535 present increment action, 35
3536 **private** clause, 25, 48
- 3537 **radeon**, 122
3538 **read** clause, 55
3539 **reduction** clause, 26, 48
3540 reference counter, 32
3541 region
3542 compute, 117
3543 data, 28, 118
3544 implicit data, 28
3545 **routine** directive, 12, 65
- 3546 **self** clause, 12, 63
3547 sentinel, 17
3548 **seq** clause, 47, 66
3549 **serial** construct, 23, 33
3550 **serial loop** construct, 50
3551 **shutdown** directive, 60
3552 *size-expr*, 45
- 3553 thread, 119
3554 **tile** clause, 12, 47
- 3555 **update** clause, 55
3556 **update** directive, 62
3557 **use_device** clause, 43
- 3558 **vector** clause, 46, 66
3559 vector lane, 21
3560 vector parallelism, 8
3561 vector-partitioned mode, 8

- 3562 vector-single mode, 8
- 3563 **vector_length** clause, 25
- 3564 visible device copy, 120
- 3565 VP mode, 8
- 3566 VS mode, 8

- 3567 **wait** clause, 31, 63, 69
- 3568 **wait** directive, 69
- 3569 worker, 21, 24
- 3570 **worker** clause, 46, 66
- 3571 worker parallelism, 8
- 3572 worker-partitioned mode, 8
- 3573 worker-single mode, 8
- 3574 WP mode, 8
- 3575 WS mode, 8